Properties of data flow frameworks

A unified model

T.J. Marlowe and B.G. Ryder
Department of Computer Science Rutgers University, New Brunswick, NJ 08903, USA

Received June 6, 1988/August 10, 1990

Summary. A comprehensive overview of data flow frameworks and their characterizing properties is presented, to clarify property definitions and demonstrate their interrelation. Properties ensuring the existence of a solution are differentiated from those guaranteeing particular convergence behavior for specific solution procedures. Examples illustrate the orthogonality of these precision and convergence properties. In addition, several data flow problems are categorized with respect to these properties.

Contents

1. Introduction ........................................................................................................ 122
2. Why do data flow analysis? ............................................................................. 123
3. Data flow frameworks ...................................................................................... 124
   3.1. An example: reaching definitions ......................................................... 124
   3.2. Data flow frameworks: extensions and refinements ............................. 126
      3.2.1. Abstract interpretation and data flow analysis ............................. 128
4. Solutions and approximations ........................................................................ 129
   4.1. The equation set and solution methods ............................................... 129
   4.2. Solutions ................................................................................................. 130
   4.3. Safety, acceptability, and approximating lattices ................................. 132
5. Solution procedures ......................................................................................... 133
   5.1. The flow graph ....................................................................................... 133
   5.2. Cycles ...................................................................................................... 134
   5.3. Iteration solution procedures .............................................................. 134
   5.4. Elimination solution procedures ........................................................... 135
   5.5. Path algebra solution procedures .......................................................... 136
6. Other approaches ............................................................................................. 137
   5.6.1. Other solution methods .................................................................... 137
6.6.2. Data flow problems on derived representations .................................. 138
6. Properties of data flow frameworks ................................................................ 139
   6.1. The semilattice ....................................................................................... 139
6.2. The function space ...................................................................................... 141
6.3. Other framework properties ..................................................................... 147
6.3.1. Intraprocedural and interprocedural problems ................................... 147
6.4. Framework properties of known problems .............................................. 148
1. Introduction

Data flow analysis algorithms gather facts about the use and definition of data, and information about control and data dependencies in programs [2, 41, 52]. For example, given the program statement

\[ x = y + z \]

- What definitions of \( y \) might have set the value of \( y \) used here?
- Where will the value of \( x \) computed here be used in the program?
- Does \( z \) always have the same value at this point in the program?

Each of these questions generalizes to a data flow problem asking for similar information about all instances of all program variables.

Data flow frameworks are algebraic structures used to encode and solve data flow problems. A data flow framework for a problem involves a flow graph, a semilattice of values, and a set of functions from the semilattice to itself. Properties of these components (e.g., reducibility of the flow graph, descending chain condition on the semilattice, or monotonicity of the function space) affect existence of an exact or approximate solution, applicability of methods of arriving at that solution, and complexity of the method used.

Our work provides an overview of dataflow frameworks and the interrelationships of their properties. Although several papers have discussed such frameworks [11, 27, 39, 50, 72, 88] and defined or redefined properties, none have had an overview as a principal aim. The contribution of our overview is to explain the relationships between properties which have not previously been compared in the data flow literature. We do so by specifying the model, reviewing theoretical results and clarifying property definitions. We differentiate between properties ensuring the existence of a solution and properties guaranteeing particular convergence criteria for various solution procedures, and give examples to show that precision and convergence properties are essentially independent.

First, we review data flow frameworks and their properties. In Sect. 2, we motivate the study of data flow and the use of frameworks. In Sect. 3, we give a formal definition of a framework and illustrate it for the Reaching Definitions problem. Sections 4 and 5 discuss solutions to data flow problems, and the principal solution procedures: elimination, iteration, and parallel algebra. Section 6 then discusses properties affecting the existence and precision of a solution or the speed of convergence. Section 7 reviews complexity results depending on these properties. In Sect. 8, we present a new classification scheme for framework properties and give examples.

2. Why do data flow analysis?

Data flow analysis algorithms [41, 52] take a program as input, capture static information (of a nature dependent on the particular problem), and return derived information as a solution. Such algorithms are used to solve problems in optimization [2, 9], verification [5, 27], debugging [96], testing [68], and parallelization [13, 32, 37, 60], vectorization [4], and parallel programming and environments [19, 36, 61, 80, 84]. They provide information about a program or environment without executing the code.

Naturally, it is impossible to determine the exact output of a program, or the operation of an environment, without actual execution; this clearly subserves the halting problem. Rather, data flow analysis uses static information to give approximations, and to obtain only approximate solutions to some of the problems. Given a problem, a solution can err by overdetermination — allowing stronger conclusions that are actually valid, or underdetermination — failing to report as strong or precise a conclusion as possible; we naturally prefer valid if imprecise information, and most algorithms in use are guaranteed to generate such information. Further, there is a sense in which it is possible to speak about a best possible static solution (see Sect. 4).

Data flow analysis divides naturally into two domains: intra-procedural and inter-procedural. Other applications are at a still lower (e.g., statement [27]) or higher (e.g., module, or parallel thread [34]) level. Techniques do not differ greatly, but the objects manipulated are different, and the types of problems and the incidence of structural properties may differ. Intra-procedural analysis deals with the statements or basic blocks [2] of a single procedure and with transfers of control between them; inter-procedural analysis [7, 24] treats the procedures of a program and their calling relationships.

Flow graphs provide a convenient model in both cases. The flow graph is a labelled, directed multi-graph, consisting of a set of nodes (the basic blocks, or the procedures), a set of arcs (transfers of control, or calls of procedure bodies), and usually a distinguished start vertex, corresponding to the initial block of the program, or the procedure main, whose indegree is usually taken to be zero. Nodes are invariably single-entry, and usually single-exit. There may also be a distinguished terminal vertex. Information is attached as labels to the nodes and/or arcs of the flow graph. Combining and manipulating information requires interpreting the labels as elements of an algebraic structure; the principal model uses functions on a semilattice.

A data flow solution assigns a value to each node, which may be a set of facts, a relation, or a set of assertions. A set of equations relates the assignment at a given node to the values at other nodes, the labels of the nodes/edges, additional information specified for the entry or exit node(s), and possibly some global constants [78]. Labels represent local information, such as preserved and generated definitions. The nodes appearing in the equation for the solution

1. Or procedure, module, software system, etc.
2. Often called complex-time information
3. But see Rosen [43, 72, 73]
4. Cousot and Cousot [27, 28] also use a flow graph for program verification, where nodes are statements and edges transfers of control (one target for assignments and two for tests) for proof of program correctness, and still other applications are possible.
5. Or edge.
at a node \( v \) are almost invariably a subset of the neighbors of \( v \); usually, they are either the successors or predecessors of \( v \) in the flow graph.\(^6\)

These systems of equations can be represented by a \textit{dependency graph}, in which there is an edge from node \( u \) to node \( v \) if node \( u \) appears non-trivially in the equation for \( v \). Data flow problems can be classified as \textit{forward} or \textit{backward}. In a forward problem, information and control (flow graph edges) flow in the same direction, and the dependency graph and the flow graph are the same; in a backward problem, information flows in a direction opposite control. The dependency graph for a backward problem thus corresponds to the \textit{converse graph} of \( G \), which is \( G \) with all the edges given reversed directions.

Solution methods seek a computable solution to this set of equations, usually (in some sense) an \textit{extreme solution} [27, 67]. When an exact solution cannot be computed, these methods will return an approximation. The interpretation which can be given to the assignment returned by the solution method, its accuracy, the complexity with which it can be found, and even the existence of a solution or applicability of a method, all depend on properties of the framework. For this reason, this paper presents an overview and codification of frameworks, solution methods and properties.

3. Data flow frameworks

To facilitate discussion of extreme solutions and speed of convergence, data flow problems are often formulated in an environment combining flow graph structure with lattice properties.\(^8\) This environment, first defined by Kildall [53], and refined by Kan and Ullman [50, 51] and Graham and Wegman [39], has the form of Fig. 3-1.

In the literature, \( D \) is usually called an \textit{instance} of a \textit{data flow framework} or a \textit{data flow problem}, and \textit{data flow framework} is reserved for the pair \((L, F)\).

We will use these terms interchangeably, for two reasons: first, because \( L \) and \( F \) usually depend parametrically on program structure (the set of definitions in a procedure, or procedures in a program, etc.), and second, because discussion of the properties of frameworks requires consideration of \( G \) and \( M \) as well as \( L \) and \( F \).\(^9\)

We will call \( M \) an \textit{edge transition function} or \textit{label function}.

3.1. An example: Reaching definitions

The \textit{Reaching Definitions Problem} is an intraprocedural data flow problem; a given variable \( X \) may be defined in several basic blocks of the procedure, and the interesting question is: Which of these definitions reach the entry to some other basic block? This information could be used to answer the questions: \( X \) is used anywhere before it is defined? A definition \( d \) of \( X \) in block \( B \) is \textit{downward-exposed} if no other definition of \( X \) occurs after \( d \) in \( B \). A downward-exposed definition of \( X \) in block \( B' \) reaches the entry of block \( B \) if there is a path from the exit of \( B' \) to \( B \) on which no other definition of \( X \) occurs.

This leads to the following set of equations:

\[
\text{Reach}(B) = \bigcup_{p \in \text{Pred}(B)} \left[ \text{Reach}(B') \cap \text{Pres}(B') \cup \text{Gen}(B') \right]
\]

\[
\text{Reach}(p) = \emptyset,
\]

(3.1)

where \( \text{Reach}(B) \) is the set of definitions reaching the top of \( B \), \( \text{Pres}(B) \) is the set of definitions preserved through \( B \) (that is, not superseded by more recent

\( \text{Gen}(B') \) is the set of definitions generating definitions in \( B' \)
3.2. Data flow frameworks: extensions and refinements

Many problems in the literature, like Reaching Definitions, were not originally phrased as meet problems; we review the translation from join to meet problems. Also, we consider "initial values" of frameworks, specified for entry or exit nodes, and the way in which a framework defines a set of equations. Finally, we consider approximating lattices.

**Lattices and semilattices, meets and joins**: Meet semilattices are by induction closed under arbitrary (non-empty) finite meets, but need not be closed under infinite meets or an empty meet of lattice elements. ∧ is by definition 1, so a semilattice admits the empty meet if and only if it includes 1. Although the definition of data flow framework in Sect. 3 included an element 1 ∈ L, this is not required in general. The presence or absence of the 1 will principally affect complexity. It is frequently possible to add an artificial 1 and retain the properties of the framework. Also, as Rosen [72] points out, a semilattice with 1, closed under infinite meets, is in fact a lattice (take x ∨ y = x ∧ y = x and x ≤ x).

The use of meet is in some sense arbitrary; by duality [10], everything could be phrased in terms of meet semilattices instead. In a join semilattice, references are over all paths; "maximum fixed point", "greater solution", should be replaced respectively by "join over all paths", "maximum fixed point", and "lesser solution", and so on.

Alternatively, we can constrain the natural join to be the meet in a related lattice. In a lattice with both 0 and 1, we can consider join as the meet in the opposite order (X ≤_Y if Y ≤ X). Alternatively, in any lattice with a complement-like involution c (c is bijective, c = c, and x ≤ y implies c(y) ≤ c(x)), such as a powerset lattice, one can always replace a join semilattice framework solving for X by a meet semilattice for c(X), and subsequently recover X. For these reasons, all semilattices henceforward will be meet semilattices unless specified.

---

**Properties of data flow frameworks**

In [28], Cousot and Cousot classify data flow frameworks by triples formed from (meet/join) × (forward/backward) × (max solution/min solution). Some algorithms [90, 91] use different combinations of these triples in different phases.

**Entry values**: For a forward problem, we assign to p an initial value p, the facts true on entry to the program (facts true on exit from the exit vertices if D is a backward problem). Often p will be 0 (no facts are true on entry). If there is no unique start vertex, or the start vertex is on a cycle, the usual linear programming trick of creating a supersource creates an acceptable p. The same trick will if necessary create a unique exit vertex p', or sink. Rosen [43, 72, 73] allows an entry value function En: H → L assigning arbitrary values on entry to vertices in the entry node set, which he allows to be greater than just the singleton {p}. This simplifies unified treatment of forward and backward problems (since a unique sink is not required in the standard framework), and may permit more efficient interprocedural or incremental algorithms. However, since the resulting analysis is more complex, and since previous work usually considered only single-entry flow graphs, we focus our attention on that case.

It should also be noted that when H = {p}, there is a reasonable assumption that all nodes in V are reachable from p. In the more general situation, the assumption is rather that every node of V is reachable from some node of the entry set H. Structured programming does not assume, however, that all execution paths will reach an exit; errors and infinite loops are possible. Correspondingly, not every node need reach an exit.

The equation set of D: We think of the framework as implying a system of equations Q, parametrized by the nodes of the flow graph, whose terms involve the values of an unknown function X: V → L and the values of M [78]. The equations depend on whether the problem is forward or backward, on the initial value n, and on interpretation, in particular, on whether information is being evaluated at node entry or node exit. If iteration is used to solve the equations, we also require an initial guess X0: V → L. In many cases, the initial guess will be 1 at non-entry nodes; it must be n usually 1 or 0 at p. For Reaching Definitions, the initial guess X0 is X0(p) = 1 = # for all w ∈ V. That is, no definition reaches the entry node, and a definition reaches B only if we find a path on which it reaches B.

The pair (Q, X0) is implicit in the framework; the set of possible values for X0 is restricted by the problem, and given X0, the system of equations is determined. We consider the equation set and its relation to data flow problem solutions in Sect. 4.1.

**Subobjects and approximations**: The semilattice L may be contained in another semilattice K as a subobject: if L = (A, ∪, L, ≤A, ∧A) and K = (B, ∪, L, ≤B, ∧A), then A ⊆ B and ≤B is the restriction of ≤A to A. Then

\[
\forall x, y, z \in L, x \leq y \wedge z, \exists x, y, z \in L
\]

so that meets in L approximate those in K. If in addition all K-meets of elements in L are also in L, then the two meet are equal, and L is a subsemilattice.

---

10 Note however that if domain M is considered to be V × V (rather than V), then the target is used to decide if the value of M(n, w) is as specified (if there is an edge), or 1 (otherwise).

11 Howitz et al. [45] allow 0 or 1.

12 Rosen's notation differs significantly from the notation used here.

13 More generally, X_v(p) = En(v) for all entry nodes v.
of $K$. A restriction to a (well-behaved) subposet or subsemilattice may be necessary, as for type determination in Tenenbaum [91], or for aliasing in the presence of pointers in Weihl [95]; or may be desirable solely to provide lower complexity at the cost of a measure of accuracy [28]. In such approximating frameworks, not only the underlying lattice, but the function space $F$ and the assignment $M$ may also change [27].

There may be several approximating lattice models for a given problem. Usually these are all sublattices of a single natural (but intractable) semilattice; occasionally, however, there may be no obvious natural lattice formulation. In those cases there may seem to be several possible incompatibly lattice formulations, or no natural formulation at all, when the lattice itself will seem somewhat of an artifice, as in Constant Propagation (a meet semilattice problem) [18, 50, 94]. Cousot and Cousot [30] recover an optimal approximating framework by formal techniques, which may not however be practicable. They also show [28] how several data flow problems (including the checking of array bounds, determining if variable values are positive or negative, and Constant Propagation) can be seen most naturally expressed as different sublattices of the same original lattice of program assertions about the possible values of variables of integer type.

In the same way that less precise information can be achieved more cheaply, more precise information can sometimes be obtained with a concomitant increase in complexity. Holley and Rosen [45] refine results for data flow by combining two lattices on the flow graph; basically, one is the original data flow problem lattice, and the other a lattice of sets of conditions which affect when information will be passed along a flow graph edge. They term such refinements qualified path problems; these obtain finer approximations to execution-time information flow even for the classical problems, but with a much larger framework $D'$. In one of their two algorithms, the flow graph $G'$ is much larger than $G$, in the other the framework $\langle L, F' \rangle$ is larger.

3.2.1. Abstract interpretation and data flow analysis

Data flow frameworks as lattices of assertions: Since a data flow problem seeks program information, the semilattice $L$ will in general be embeddable as a subposet of the lattice of possible program assertions about a program [28, 30]. The operations on frameworks can then be interpreted as operations on assertions in an abstract semantics for the programming language. In the most general context, the elements of $L$ are partial (control, memory) states, the functions of $F$ are partial state transition functions, and a solution will report provable partial state information at each vertex of $V$. The meet in $L$ is logical and [14], the flow functions $(M)$ will be the standard semantic flow functions.

This is a two-way transformation. We can consider symbolic execution in a formal semantics as data flow in the lattice of program assertions (or of (control, memory) states). However, this cannot discover all correct assertions about run-time semantics, first, because of issues of computability, and second, because an unbounded amount of information is needed. However, we can take an approximating semantics, by "collapsing" or "abstraction" the set of (control, memory) states into a manageable set of equivalence classes. In [48], for example, collapsing includes ignoring the addresses of variables in memory, ignoring the particular integer values stored in a variable, and collapsing multiple levels of pointer references.

However, the approximating semantics must reflect the underlying formal semantics; this requires the states in the approximating semantics to form a semilattice, and approximation to be a safe map (that is, approximating a state and applying a transition function should give an approximate state which is a safe state for the actual resulting state).

Determination of program properties by application of approximating semantics is often called abstract interpretation [1, 56, 64, 75]; this approach is formally (although not conceptually) equivalent to the algebraic framework approach presented here.

4. Solutions and approximations

4.1. The equation set and solution methods

Solution procedures appear to divide into three principal classes. Two of these, elimination and iteration, are based on the solution of the system of equations $Q$ for $f$. We briefly explain them here and consider them in greater detail in Sect. 5.

The set of equations $Y = Q(Y)$ for the values of a typical forward data flow information function $f$ defined on the nodes $v \in V$ has a form similar to:

Forward Problem – Information on Node Entry

$$X(v) = \land (M(e)(X(u)); e = (u, v) \in E), \quad (4.1)$$

where each equation may also include a constant or node-dependent term $W(v)$. The set of equations for a forward problem with node exit information is formally identical, save that entry information $\eta$ and the initial guess $X_0$ may differ.

The corresponding systems for backward problems are the same as those for forward problems on the converse graph:

Backward-Problem – Information on Node Exit

$$Y(u) = \land (M(e)(Y(v)); e = (u, v) \in E) \quad (4.2)$$

Iteration is used by Kildall [53], Kam and Ullman [50, 51], Horwitz et al. [45], Tarjan [86] and others. Iterative procedures begin with a "safe"
of Q cannot result in information contradicting that in X. In particular, we cannot receive new information about X at v from a neighbor, that is,

$$V(u,v) \in E: X(v) \subseteq M([u,v,X(u)])$$

(4.4)

The optimal solution in a data flow framework will be found if we can capture precisely the information which would have flowed from ρ along each possible path; the meet over all paths solution MOP [50] is given by:

$$MOP(v) = \bigwedge \{ M(ρ) | \rho \in Paths(ρ,v) \}$$

where Paths(ρ,v) is the set of paths from ρ to v. Thus a solution is valid, or safe [39], if it is everywhere less than the MOP solution 25, or equivalently,

$$X(v) \subseteq M(ρ)$$

(4.5)

for every v ∈ V and every path ρ ∈ Paths(ρ,v).

For instance, a safe solution for the Available Expressions Problem 26 (a meet problem) would report a subset of those expressions actually available at a given flow graph node. A safe solution to the Reaching Definitions Problem (a join problem) would include every definition in the true solution and possibly some others. Since reporting that fewer expressions are available, or that more definitions reach, may make some optimizations impossible but cannot: suggest invalid optimizations, this overestimate will result in fewer optimizations, but cannot introduce an error. Similarly, for Constant Propagation, reporting fewer variables constant than actually are may again cause some optimizations not to be performed, but will not result in using inaccurate information.

However, the MOP solution is not obtainable for every problem [50]; and we can have arbitrarily bad safe solutions. Moreover, as we have indicated, even if the MOP solution is available, it may be computationally prohibitive to find. However, reasonable assumptions on a data flow framework will guarantee the existence of fixed points of Q; frequently, in fact, an MFP solution exists. The MFP solution will always be a safe approximation of the MOP; moreover, it is the safe approximation normally discovered by iteration or elimination. Thus any solution will be acceptable if it at least as good as the MFP.

A solution X of Q, a system of equations of form 4.3, is acceptable [39] if and only if for all v ∈ V and all fixed points Y of Q, X(v) ⊆ Y(v). Clearly any such solution must be at least as large as the MFP solution, if it exists. Given a data flow framework, an assignment is safe and acceptable precisely if it lies between the MFP and MOP solutions. If they exist, the MFP solution is the smallest acceptable solution and the MOP solution is the largest safe solution. In general, the "larger" or "higher" a safe, acceptable solution is in the semilattice, the better we will consider the solution (i.e., the more data flow information it captures).

1 A "safe" initial approximation for iteration will not ordinarily be a safe solution, nor conversely. For a meet semilattice, a safe approximation will lie between Top and the MOP solution; a safe solution must (be or) lie below the MOP. For join semilattices, the situation is reversed. For Reaching Definitions, for example, a safe initial guess is that no definition reaches any node, an underestimate of the solution; a safe solution must be an overestimate.

13 This intraprocedural data flow problem ascertains which binary expressions have the same value at a program point as when they were last calculated on a path to that point [41].
4.3. Safety, acceptability, and approximating lattices

Acceptability and safety refer to the current framework, and in particular to the current F and M. If F or M is changed, and the fixed point of the new equation set Q' become smaller than those of Q, then smaller, less accurate assignments become acceptable. Conversely, if M or F is changed and fixed points become larger, then the new MFP may be a more accurate assignment than the original MFP. A change in F or M can affect not only the MFP solution, but also the MOP solution, and the MOP solution is guaranteed to be the most accurate not for the underlying problem but only for the given framework D.

The goal in solving a data flow problem is to find a safe, maximal acceptable approximation to the MOP solution. Ideally, of course, one would like to iterate to the true solution, which we would like to be the MOP solution for the semilattice. There are three problems: first, the true solution may not be expressible in the semilattice L; second, the true solution may not be a fixed point of the set of equations; and third, iteration to the true solution may require too much computation (or an infinite amount). In Constant Propagation, for example, the MOP solution is not reachable by iteration (the second case).

In general, even if the MOP solution is available, it need not be optimal to reach the original problem; some approximation may already have occurred in translating into a semilattice framework, either because the original problem may be impossible to formulate in a semilattice or that algebra frame- work, or because the lattice or the function space for the original problem may be difficult or expensive. In Altasing,7 8, 23, 95 and Type Determination8 91, the semilattice and equation set are only approximate. In fact, no finite lattice can handle type or pointer information precisely; we can get more accurate information in these cases by choosing a larger L with finer granularity (more levels of indirection for pointers, in the first case above).

If the framework in which the solution was found was only approximate, then it may be possible to refine the approximation further by plugging it into the set of equations, or a derived set, and driving the value upward, toward the MOP for the exact framework, through a fixed number of iterations8 28, or by apparently non-iterative methods8 91.

There are also problem transformations which preserve the MOP solution but give different MFP results. In the two sets of equations for Constant Propagation given in51, the second mapping M gives a more precise approximation to the MOP solution; essentially, it gives values on node exit rather than node entry. In this example, G, L, and F remain the same, but the assignment M and the entry information n are changed.

Data flow solution procedures ordinarily construct assignments which lie between the MOP and MFP solutions (possibly for a transformed problem). Further, taking code optimization as an example, an unsafe assignment will result in semantics-changing “optimizations”, while an unacceptable assignment will necessarily result in not performing valid and generally easily discovered optimizations. Thus safety and acceptability seem to be relevant criteria. However, even though an assignment is safe and acceptable, it may have this property with respect to the current, not necessarily the original, set of equations; further, it need not be a fixed point of the current or original system.

5. Solution procedures

Discussion of solution procedures requires consideration of the properties of flow graphs. Most of the difficulty in solution is caused by the presence of cycles in the flow graph. We look at how each of these methods achieves its solution and handles cycles.

5.1. The flow graph

The key properties of flow graphs will be acyclicity, reducibility and nesting depth. Most models and applications also require a unique start vertex of indegree 0. Solution algorithms, and their analyses and complexities, may use the depth-first spanning tree (DFS) of the flow graph (rooted at ρ), or dominators.20

A reducible flow graph is one which can be reduced by a sequence of transformations in a particular class to the trivial graph; intuitively, it consists of a nested set of single entry regions78. It can be shown that a rooted, directed graph G is reducible if and only if the graph of Fig. 5-1 is not contained in G in the following sense: the edges represent node disjoint paths in G and nodes a, b, c and ρ are distinct (except that a and ρ may be the same)92.

A graph is converse-reducible if its converse graph is reducible. In practice41, most program flow graphs are reducible.

Intuitively, the nesting depth31 of a graph is the program loop-nesting depth and is a measure of complexity of both the program and its associated data flow problems and their solutions.32 Elimination algorithms in particular rely

![Fig. 5-1. The canonical irreducible graph](image-url)
on a decomposition of the flow graph into intervals [3, 78, 85, 89]. Tarjan intervals are essentially equivalent to program natural loops [2], and define an interval dependency tree given by interval containment; the nesting depth of a flow graph is equal to the depth of that tree. Once embedded intervals are condensed, an interval is a union, not necessarily disjoint, of cycles passing through the interval head node, which dominates its other nodes. Dominator information is used more fully by some algorithms [20, 21, 70, 89] rather than or in addition to the interval tree; the immediate dominator tree can be used as an alternative to the interval dependency tree.

One other standard assumption for complexity analysis, justified both from observations of real programs [54, 77, 78] and from structured programming principles, is that the average out-degree of a flow graph node is constant, that is, that the edge set has cardinality $|E| = O(|V|)$.

5.2. Cycles

If all flow graphs were acyclic, then paths from $p$ to $v$ would form a finite set of paths of bounded length, and the MOP solution $\sum (M(p, q) \cdot Path(p, q))$ could be computed accurately and efficiently, simply by following all such paths and comparing the information obtained.

Once we have cycles, however, the set of paths is in general infinite, and an exhaustive solution procedure, checking each path separately, has no hope of success. Algebraic, finiteness, or closure properties must be used to compute, or at least to estimate, the contribution of a cycle $C$ to the solution; the three families of solution procedures use these properties together with intelligent evaluation order. This order is easier to find, and the resulting computations simpler, when the flow graph is reducible.

The summarizing or approximation of the effect of zero or more passes through an interval, is strictly analogous to the way in which information is captured in program verification by loop invariants; this connection is explicit in [27, 28, 29, 30].

5.3. Iteration solution procedures

For a set of equations $Q$, iteration toward a fixed point begins with a safe initial guess, and proceeds by (1) simultaneously evaluating each equation with the current values, determining new values for each unknown (simple iteration), or (2) evaluating equations in a given order, each equation using the most recently derived values for each variable. The latter approach has scheduling variants, including non-deterministic, nodelist, round-robin and worklist meth-

ods [41, 86]. Particularly useful is a round-robin method in which evaluation order is driven by reverse postorder on the DFS of the flow graphs.

Horwitz et al. [42] propose a priority queue algorithm, lowering complexity in situations in which the acyclic outer portion of the flow graph is large. A second version combines priority queue (topological) order on the strong component condensation of the flow graph with round-robin iteration within components. Finally, a hybrid algorithm performs at least as well as priority queue, worklist or round-robin methods, to within a factor of 2, effectively by running the two known efficient algorithms (round-robin and priority queue) in parallel.

For simultaneous evaluation, the results of the $k$th evaluation at $v$ express the effects of all paths of length $k$ or smaller ending at $v$ (beginning at $v$ for backwards problems). However graphs with cycles contain paths of arbitrary length. Thus for simple iteration to succeed in $t$ steps there must be some way to discount the effects of paths longer than $t$.

5.4. Elimination solution procedures

Elimination algorithms are based on representing the solutions $X(v)$, for some subset of nodes, on values at a small number of other nodes, whose equations are then transformed so as not to depend, even indirectly, on values for the original set of nodes. Intuitively, this reduces the set of equations into two smaller sets, one for the remaining nodes, in a form similar to the original equations, the other, for the subset represented, involving only back substitution once the solution for the smaller subset is known. The process can be driven (and motivated) by Gaussian elimination using interval structure, dominators, or graph transformations.

An elimination method is specified by the preconditions under which its reductions hold, and the expansions which specify how the solutions for the reduced graph and the intervals – the induced subgraphs, possibly with some representative edges used to save needed information [21, 81] – should be combined. Some algorithms have variants specified in the preconditions by the priority assigned to applicable transformations. A particular elimination algorithm will succeed on the class of graphs it reduces to the trivial one-node graph. For many elimination algorithms, this class is precisely the set of reducible (and converse reducible) flow graphs.

A complementary view appears in [43]. A reduction summarizes the effect of a set of paths from here to there by a label on a single (representative) edge from here to there. The sets of paths to be summarized and the order in which the reductions are performed are guided by the motivating structure imposed on the problem. One such reduction or part of some reduction must be able to summarize the contribution of one or more passes around a cycle or through an interval. Once a solution is obtained for the final trivial graph, the expansions then indicate how to derive the solutions for all nodes from the solution for that final node.

33 Since reverse postorder is closely related to the interval structure of the flow graph
34 Chaotic iteration will also need such a mechanism
35 Note that these implicitly also encode the decomposition
36 Graham-Wegman elimination can accommodate irreducible graphs [39]
5.5. Path algebra solution procedures

Path algebra solution procedures are closely related to elimination and to regular expression solution procedures. They also use a framework of graph, semilattice, function space, assignment, and entry values, but in not quite the same way as iteration or elimination methods. The difference is that not every path (sequence of function applications) need contribute to the path algebra solution. Because of this distinction, we will use the term context for the framework of a path algebra problem.

One can thus pose problems in a given context which would not necessarily be solvable by other methods in the equivalent framework; Tarjan [38] gives an example due to Fong [38]. In the classical problems, a definition, expression or use will be generated, preserved or killed in a node. Since killed can usually be expressed in terms of generated and preserved, the classical problems can be expressed neatly in terms of an equation set, as in Sect. 3.1. With set-valued variables, however, it may be cheaper to re-use an expression even after an incremental redefinition of one of its variables. Following Tarjan's terminology, we call an expression injured if a subsequent use requires only a small modification.

For a parameter n, allows an expression to be available if it requires n or fewer incremental changes. The path algebra solution is straightforward, but the problem apparently cannot be posed naturally as a (semilattice) data flow problem, although it can be posed in the formalism of Holley and Rosen [43] or Cousot and Cousot [30] as a qualified path problem, or as a problem with a larger, more costly and less natural framework.

Intuitively, path algebra procedures solve for the regular expression representing an appropriate set of paths, replacing at each stage the regular expression values and operations

\[ \langle A, \text{edge e}, \text{concatenation}, \text{union}, \text{Kleene-star} \rangle \]

by

\[ \langle \text{m(e)}, \text{composition in reverse order}, \text{lattice meet, closure} \rangle, \]

where closure handles cycles/interval, essentially by taking the reflexive transitive closure of the function on the cycle/interval (see Sect. 6.2).

For example, in solving Reaching Definitions by path algebra, we solve for summary flow functions as for regular expressions, but we replace A by the identity function i, preserving all definitions, e by the function defined by the Pres and Gen labeling edge e, meet by union, and closure also by the identity, since Reaching Definitions is rapid (see Sect. 6.2). For Bound Set A, concatenation, and meet have similar definitions; e is replaced by the "lozenge" operation with local bindings A, M(e)(X) = X \cdot A, and Kleene star (of a sum of closed paths), by reflexive transitive closure of the corresponding binding relation.

Rosen [73] shows that any data flow problem can be reduced to a path algebra solution of a canonical context on the given graph. This context leads to the Flow Cover Problem: find regular expressions representing the set of paths between each pair of vertices in the graph.41 In the solution for Flow

\[ Y(e) \leq M(e) Y(a), \]

\[ Y(h) \leq \text{Ent}(h) \]

for all \( h \in H \) and \( e = (u, v) \in E. \) (5.1)

5.6. Other approaches

5.6.1. Other solution methods

Reif and Tarjan [70] give an algorithm more like path algebra than iteration or elimination, using dominators. It solves the classical problems with birthpoints, which capture summary information using auxiliary data values (such as pseudo-assignments [82]), additional def's and uses of the form \( x = \text{aux} \), which allow information along multiple paths to be captured by a single (artificial) piece of data at their junction, so that every programmer-specified use is reached by exactly one definition of the given variable. Effectively, we obtain greater information by extending the lattice to include auxiliary information and propagating this information. The algorithm is efficient but more imprecise than path algebra algorithms. A number of subsequent articles (for example, [5, 33, 34, 69, 74, 94]) use birthpoints or related constructions in data flow algorithms, for improved accuracy or efficiency. The solution procedure, once such data structures have been introduced, may vary, but is often a variant of iteration.

41 For Flow Covers, iteration to the MOP solution may require infinite work, the third case mentioned in Sect. 4.3

42 Flow Covers is distributive, and for non-distributive problems will give a best distributive approximation. For MFP solutions, however, the universal problem for non-distributive problems is Program Assertions [59]

43 As noted in Sect. 4.1, this definition of fixed point is compatible with the alternative definitions in that section when a maximum fixed point is sought

44 This definition also appears in the earlier literature, for example, in [39]
In [79], a method is proposed for data flow using general attribute grammars; it is related to both iteration and elimination, and may also need a loop-breaking or closure transformation [67, 78]. Other general attribute grammar algorithms (see for example [93]), using techniques similar to interval analysis or iteration, can also be used for solution of data flow problems.

Finally, the PIV technique of Zadeck [97, 98], which uses graph and problem information to bypass some evaluations (see Sect. 6.3) is applicable to a restricted class of problems.

5.6.2. Data flow problems on derived representations

Not all data flow analysis is performed on the flow graph; various related graph structures have been proposed to facilitate representation, computation and use of data flow information. These include the binding graph [25, 26], the program summary graph (PSG) [17, 40], and the program dependence graph (PDG) [37, 46, 47, 65] and the interprocedural system dependence graph [49]. Such structures are particularly useful in interprocedural analysis, parallelization, and incremental data flow analysis. These structures are usually related to the standard flow graph in well-understood ways:

1. Creation uses data flow information determined on the flow graph. The PSG uses the call graph and interprocedural reaching information. The PDG and variants use *ad* and *du* chains [2]. Both rely on prior solution of the Alias problem.

2. The structure breaks some edges or loops of the flow graph, where the dependence is only apparent or can be summarized, or splits nodes. The binding graph is constructed analogously to the flow graph, but a vertex representing a procedure is split into vertices for each parameter, so that (immediate or transitive) dependence between distinct formal parameters of a single procedure will not report spurious loops. Zadeck's construction also splits vertices.

3. Conversely, other edges are added to summarize data flow on sequences of edges. The PSG summarizes def-clear paths with edges. The PDG includes data dependence edges, and the system dependence graph uses summary edges like the characteristic edges of [71]. A similar construction is used on the PSG by [40]. The edges in the structure are classified. In the PDG, edges represent either control- or data-dependence, and the data-dependence edges are typically classified (for example, as *true*, *ant*, *output*, and as loop-carried or loop-independent, and annotated (with direction, or loop-name, etc.). The interprocedural edges and the summary edges in the system dependence graph form two other classes. In the PSG, there are *call* and *return* edges, as well as edges representing def-clear paths, and in [40], summary edges.

Once a structure has been constructed, properties and solution procedures are analogous to those for the flow graph. Additional data flow information can

...be extracted by standard methods on the new graph. The advantage in using the derived structures arises because the split of nodes, and the deletion, addition, and classification of edges, allow some data flow problems to be formulated, initialized, or computed more efficiently, or allow inference of certain properties (as for the binding graph or for Zadeck).

Other frameworks exist in which the lattice, rather than the flow graph, is modified. These include the qualified flow problems of [47, 92] and the hybrid algorithms of [60].

6. Properties of data flow frameworks

We have already indicated that not all data flow problems always can be solved precisely and efficiently. Various properties of the framework, not just the flow graph properties we mentioned in Sect. 5.1, will affect the existence and accuracy of a solution, and the methods by which it can be obtained and their complexity. Many such properties have been proposed, but definitions vary, with both different terms used for the same property, and the same term used for several, often just slightly different, but sometimes completely independent, properties. Further, although many of these properties have been discussed, nowhere are the distinctions and implications clearly set down, with their effects on data flow solution procedure complexity reviewed. Also, it is not always immediately clear in the literature whether the properties being discussed are properties of the semilattice or the function space (or the assignment, initial values or flow graph). This section addresses these issues.

6.1. The semilattice

Just as the most easily handled flow graphs are those in which every path is finite, the easiest semilattices are finite. Infinite semilattices are closed under arbitrary (non-empty) meets, but infinite semilattices need only be closed under finite meets; we will say a semilattice L is closed 47 if it is closed under arbitrary non-empty meets, including infinite meets.

"Local finiteness" conditions will imply not only that L is closed, but that every infinite meet is equal to a meet of only finitely many of its arguments. We say a semilattice has the *descending chain condition* d.c.c. [10] if any descending chain 48 of semilattice elements

\[ x_1 > x_2 > ... \]

is finite. 49 If L satisfies the d.c.c., then every meet \( \bigwedge v_i \) of elements of L is equivalent to the meet of a finite subset. \( \bigwedge_{i=1}^{k} v_i \) forms a non-increasing

47 Other terms in the literature are well-founded [35] and complete [88]; however, both of these terms are also used for other properties.

48 i.e., a linear order

49 There is an analogous ascending chain condition.
chain; if \( \bigwedge_{i=1}^{n} v_i \) is not equal to the infinite meet, then there is a \( v_k \) so that \( \bigwedge_{i=1}^{n} v_i \) is strictly smaller. But by d.c.c., this process must eventually halt.

If, in addition, the length of chains with \( x_i = b \) is bounded by some function of \( b \), we say that \( L \) has finite height \([41]\). If it is bounded by some constant \( k \) independent of \( x_i \), then we say it has strictly finite height, and has height \( k \) for the given value of \( k \). For a semilattice with \( 1 \), \( L \) has finite height if and only if it has strictly finite height; however, the natural numbers \( \mathbb{N} \) (with less than or equal to, \( \leq \)) form a semilattice with finite height which does not have strictly finite height, and any larger ordinal, such as \( \omega + 1 = \mathbb{N} \cup \{\omega\} \), is a semilattice with d.c.c. but not finite height \([10]\).

![Diagram](image)

**Closed**
- Closed under arbitrary meets.
- All descending chains are finite.
- Height \( k \)
- Length bounded by function of first element.

**Finite Height**
- Height \( k \) for some \( k \).

**Strictly finite height**
- (With respect to a function space \( F \)); Evaluation takes \( F \times \emptyset \) onto \( L \).

**Complete**
- Closed under arbitrary meets.
- All descending chains are finite.

Fig. 6-1. Summary of notation for Sect. 6.1

Finally, we say that a lattice \( L \) is complete \([50]\) with respect to a function space \( F \) if and only if every element of \( L \) is \( f(x) \) for some \( f \in F \). Graham and Wegman \([20]\) remark that, given \( G \), \( F \) and \( M \), \( L \) can always be taken to be only the values actually reachable from \( 0 \); we can restrict to \( L_0 \), the set of points expressible as \( \bigwedge_{i=1}^{n} \{M[p_i] \} \), for finite sets of indices \( A \), where each \( p_i \) is a path in \( G \) starting at \( \rho \). This subsemilattice is always closed under finite meets, but need not be closed for infinite meets. In particular, infinite sequence of function iterates may not have a lower bound. Le: \( C \) is a cycle and consider, as usual, the set of paths corresponding to zero or more passes around \( C \).

![Diagram](image)

Fig. 6-2. Lattice local finiteness properties

\(^50\) Or is well-founded \([11]\)

Properties of data flow frameworks

\( \{1, M(C), M(C), M(C)(C), \ldots \} \). The meet of this infinite set of iterates need not be in \( L \) even if it was in \( L \). Similarly, \( F \) need not be closed, that is, the meet of an infinite set of functions need not be in \( F \). There is a tension between the closure and completeness of \( L \) which corresponds to the tension between the descriptive power and the ease of representation of functions in \( F \). If \( L \) has d.c.c., then all meets are finite, and these problems do not occur.

6.2. The function space

Most interesting properties are properties of \( F \). The semilattice structure on \( L \) induces a semilattice structure on \( F \):

\[
 f \leq g \quad \text{if and only if} \quad \forall x \in L, \quad f(x) \leq g(x), \quad \text{and} \quad h = f \land g \quad \text{if and only if} \quad h = \max \{f(x) \land g(x) \mid x \in L\}.
\]

Thus the meet of \( f \) and \( g \) in \( F \) is always no larger than the pointwise meet of \( f \) and \( g \) in \( L \); our earlier definition and most applications require the meet in \( F \) to be the pointwise meet. Further, we insist that \( F \) contain the identity function \( 1 \), and the constant functions \( 0 \) and, if there is a \( 1 \in L \), \( 1 \).

However, approximating semilattices and function spaces may not have meets in \( F \) equal to pointwise meets. If the function space \( F \) is approximated by a smaller space \( F', F \) may not be closed under pointwise meet. But then it is at least possible that some sequence of meets of iterates of a function \( f \) are no longer be expressible in \( F \).

There are two main classes of function space properties. Most such properties \( P \) are defined for individual functions \( f \in F \); we will say \( F \) has \( P \) if each \( f \in F \) does. Algebraic properties are morphism properties of \( F \) as a space of functions on \( L \) considered as a poset, semilattice or closed semilattice.\(^{12} \) \( F \) is also closed under composition, and meets of sequences of iterates of functions are particularly important, intuitively since if \( f = M(C) \) for a cycle \( C \), \( \bigwedge_{i=0}^{n} f \) represents "\( M(C^n) \)." Properties of meets of iterates divide into local finiteness and closure properties. Local finiteness properties allow replacement of the infinite meet of function iterates by a finite initial segment; closure properties allow capture of summary information by an exact or approximate function even when the meet of iterates does not necessarily truncate.

**Algebraic properties**

A function \( f \) is monotone if it is a poset morphism of \( L \), that is, if

\[
 \forall f \in F \forall x, y \in L, \quad x \leq y \implies f(x) \leq f(y).
\]

\(^{12} \) There are two separable issues here. \( F \) may be closed under pointwise meet but be represented by a generating set which is not. Rosen \([72]\) claims that the sets of functions usually specified for classical monotone data flow frameworks are not necessarily closed under pointwise meet. Horwitz et al. \([45]\) show that there may be efficiently expressible functions whose composition is not efficiently expressible, and thus that it may be more efficient not to compute function compositions, but use only iterated application. Thus the functions explicitly specified in a problem – which together with the lattice form the algebraic context – need not be all the functions implicitly associated with the data flow framework.

\(^{11} \) That is, maps in \( F \) preserve algebraic structure in \( L \).
and anti-monotone if
\[ \forall f \in F \forall x, y \in L: x \leq y \Rightarrow f(x) \geq f(y). \]

Monotonicity is equivalent \([41, 50]\) to
\[ \forall f \in F \forall x, y \in L: f(x \land y) \leq f(x) \land f(y). \]

\( f \) is distributive if
\[ \forall f \in F \forall x, y \in L: f(x \land y) = f(x) \land f(y), \]

and continuous if \( f \) is a morphism of closed semilattices, that is, \( L \) is closed under arbitrary (non-empty) meets and
\[ \forall f \in F \forall \text{ non-empty sets } \{x_i\}_{i \in L}: f(\bigwedge_{i \in L} x_i) = \bigwedge_{i \in L} f(x_i). \]

That is, monotone functions take chains in the cochain to chains of images, distributive functions take finite meets to meets, and continuous functions take arbitrary meets to meets.\(^{13}\) Every continuous function is distributive, and every distributive function is monotone, but not conversely.\(^{14}\) If each \( f \in F \) is monotone (distributive, continuous), then so are the function space \( F \) and the data flow problem \( D \). Data flow problems with non-monotone function classes are not usually considered.\(^{15}\)

![Diagram of algebraic properties of function spaces](image)

**Fig. 6.3.** Algebraic properties of function spaces

\(^{13}\) If \( L \) is a lattice, we can also define anti-distributive and anti-continuous functions, which take meets to joins.

\(^{14}\) However, if \( F \) or \( L \) is finite, then distributive implies continuous, and if \( L \) is a chain, monotone implies distributive.

\(^{15}\) But see \([1, 16]\), where anti-monotone functions are considered at length.

---

**Local finiteness properties**

Local finiteness properties of \( L \) imply pointwise finiteness for a function \( f \in F \), but not in general any uniform property \([83, 86, 88]\). If \( L \) has d.c.c.,\(^{16}\) then for each \( f \in F \) and \( x \in L \), the sequence \( \lim_{i \to \infty} f(i) \) stabilizes at some iterate \( f^n(x) \), but the set of \( n(x) \) need not be bounded, in which case the sequence \( \lim_{i \to \infty} f^i \) will not stabilize. Thus \( L \) need not have d.c.c.

Conversely, even if \( L \) does not have d.c.c., \( F \) may be sufficiently restricted so that chains of iterates behave nicely. For \( f \in F \), define
\[ f_0 = \bigwedge_{i \geq 0} f^i. \]

We say that \( F \) is **bounded** if for any \( f \in F \) the chain \( \{f^i(x)\} \) is finite, **k-bounded** if the length of that chain is at most \( k \) (independent of \( f \)), and **k-semibounded** for individual functions if for all \( x, y \in L \) and \( r \leq k \)
\[ f^k(x) \leq f^r(y), \]

\( k \)-boundedness is defined for functions in general; for monotone functions it is equivalent to
\[ f^k \leq f^r; \]

\( k \)-boundedness implies \( k \)-semiboundedness which in turn implies \( (k+1) \)-boundedness \([86, 88]\). Call \( F \) uniformly bounded if it is \( k \)-bounded for some \( k \). Semi-boundedness (except for 1-semiboundedness) does not appear to be important in solving "unrestricted" problems, but Tarjan \([88]\) shows it occurs naturally in qualified path problems.

If \( L \) has height \( k \), then \( F \) will be \( k \)-bounded, but the converse need not hold. In particular, if \( L \) is a product lattice,
\[ L = L_1 \times L_2 \times \ldots \times L_p, \]

and \( F \) is a product of functions on the components,
\[ F = F_1 \times F_2 \times \ldots \times F_p, \]

then the length of \( (f^0) \) is bounded by the effective height \( L_e = \max \{ \text{height } L_i \} \) \([72]\). If the \( L_i \) and the \( F_i \) are repeated identical factors, and each of height \( r \), then \( L \) will have height \( rp \), but \( F \) will have height the effective height of \( L, r = \text{the effective height of } L \).

**Fastness and rapidity:** 2-boundedness and 1-semiboundedness are historically important in data flow analysis. \( f \) is fast \([39]\) if it is 2-bounded, that is, for monotone functions, if
\[ f \leq f \land 1; \]

fastness is clearly related to idempotence, \( f \circ f = f \), or weak idempotence, \( f \circ f \leq f \). Idempotence is a property of the functions in each of the four classical intraprocedural data flow problems: Reaching Definitions, Live Uses, Available and Very Busy Expressions \([11, 81]\). Fastness justifies a "drop the term" loop-breaking rule for elimination algorithms \([67, 78]\).
f is (Kam-ullman rapid [50] if it is 1-semibounded; for monotone functions, this is equivalent to
\[ \forall x, y \in L : f(y) \geq y \land x \land f(x). \]

Kam and Ullman's original definition required
\[ \forall f, g \in F \land x \in L : f(g(0)) \geq g(0) \land f(0)(x); \]
the two definitions are equivalent for complete lattices. If L has a 1 and F is monotone, rapidity is equivalent to
\[ \forall y \in L : f(y) \geq y \land f(1). \]

k-boundedness implies that all contributions to the MFP occur prior to the k-th iterate; k-semiboundedness that the contribution of the k-th iterate is constant. Thus rapidity implies that one pass around a cycle will summarize its contribution; rapidity that the contribution of the cycle is independent of the value at the cycle entry node.

Closure properties

Even if the chain \( \{f^{(i)}\} \) is infinite, there may exist a computable meet, or a lower bound estimating the meet. Even the chain is finite, there may exist a quicker method for computing the meet, or at least an approximation.

Fastness closure: For \( f \in F \), \( f \land i \) is monotone, decreasing, and less than \( f \); we can use meets of its iterates to bound meets of iterates of \( f \). Define \( f^{(i)} = (f \land i)^{\omega-1} \).

If \( f \) is monotone, \( f^{(i)} \leq f^{(0)} \), and if \( F \) is distributive they are equal. Graham and Wegman [39] define the fastness closure \( F_{\omega} \) of \( f \) to be the last term \( f^{(i)} \) in the decreasing chain \( \{f^{(i)}\} \). \( F_{\omega} \) is idempotent and therefore fast. If \( F \) is uniformly bounded by \( k \), then \( F_{\omega} \) exists (with no greater \( k \)), but \( F_{\omega} \) may exist even if \( \{f^{(i)}\} \) is infinite.

A different definition, essentially the following, is given by Rosen [72]. \( F_{\omega} \) is the first term \( f^{(i)} \) less than all terms in the sequence \( \{f^{(i)}\} \); that is, \( f^{(i)} \leq f^{(j)} \) for all \( k \geq r \). Again, for monotone \( F \), it is sufficient that \( f^{(i)} \leq f^{(j)} \), \( F_{\omega} \) is not guaran-

teed to be fast, but exists whenever \( F_{\omega} \) does, and the last "exponent" \( r \) for \( f^{(r)} \) is no greater than that for \( F_{\omega} \). When \( f \) is distributive, they are equal. Henceforward, we will use \( f \) for \( F_{\omega} \), and identify Rosen's fastness closure by \( F_{\omega} \).

Many of the results nonetheless hold for both fastness closures.

If \( f \) is monotone, then \( f \land i \), and therefore the fastness closure, has the same fixed points as \( f \). However, fastness closure is not a map of function spaces; it does not distribute through meet or composition. Even though \( (f \land g) \land (f \land i) = (f \land i) \land (g \land i) \), it will not in general be true that \( (f \land g) \land (f \land i) = (f \land i) \land (g \land i) \), or that \( f \land g = (f \land i) \land (g \land i) \). The fastness closure of \( f \land g \) will not necessarily be \( f \land g \), nor will the closure of \( f \land g \) be \( f \land g \). Thus the equivalence of fixed points buys less than first appears, since successive or multiple cycles may not be handled precisely. Also, there are data flow problems for which \( F \) does not have fastness closures; Cooper [23] gives a distributive (in fact, continuous) framework for Alias-ing for which in general there are no fastness closures.

Infinite limits: Let \( f^{*} = \bigwedge_{k=0}^{\omega} \{f^{(i)}\} \). By analogy with regular expressions, if \( f \) exists. If the semilattice is not closed, or \( F \) is not continuous, \( f^{*} \) need not exist in \( F \). \( f^{*} \) is essentially the (reflexive) transitive closure of \( f \). We could also define an extension of the fastness closure, \( F_{\omega} = \bigwedge_{i=0}^{\omega} \{f^{(i)}\} \). Since the \( f^{(i)} \) and the \( f^{(i)} \) form descending chains, \( f^{*} \) and \( F_{\omega} \) are limits of sequences of iterates when they exist. If \( f^{*} \) exists for all \( f \in F \), then so will \( F_{\omega} = (f \land i)^{\omega} \), but the converse need not hold. If \( f^{*} = f^{(i)} \), then \( f^{*} \) can be evaluated with \( O(i) \) meets and compositions; if \( F_{\omega} = f^{(i)} \), then \( f^{*} \) can be computed with \( O(\log i) \) operations.

\[
\begin{align*}
\text{\( f^{*} \)} & \quad \text{The k-th iterate of } f \\
\text{\( f^{(i)} \)} & \quad \text{The meet of the first } k-1 \text{ iterates of } f \land i \\
\text{\( f^{(i)} \)} & \quad \text{The (k-1)-st iterate of } f \land i, (f \land i)^{\omega-1} \\
\text{\( f = f_{\omega} \)} & \quad \text{(The fastness closure) the finite limit of the descending chain } f^{(i)} \\
\text{\( f_{\omega} \)} & \quad \text{(The extended fastness closure) The limit, not necessarily finite, of } f^{(i)} \\
\text{\( f_{R} \)} & \quad \text{(The Rosen fastness closure) The first term in } f^{(i)} \text{ which is a lower bound for } f^{(i)} \\
\text{\( f^{*} \)} & \quad \text{(The reflexive transitive closure) The limit, not necessarily finite, of } f^{(i)} \\
\text{\( f^{\omega} \)} & \quad \text{(A pseudotransitive closure) An approximation to } f^{*} \text{ with certain nice properties} \\
\text{\( i_{0} \)} & \quad \text{The Rosen rapidity parameter} \\
\text{\( x \)-bounded \)} & \quad \text{Iterates of } f \text{ from } k \text{ on don't change } f^{(i)} \\
\text{\( x \)-semibounded} & \quad \text{Intuitively, } f^{(i+1)} \geq f^{(i)} \text{ constant} \\
\text{Bound} & \quad \text{Each } f \text{ is bounded for some } k \\
\text{Uniformly bounded} & \quad \text{Each } f \text{ is bounded for the same } k
\end{align*}
\]
Approximations: A lower approximation \( f^\delta \) to \( f^* \) is called a pseudotransitive closure \([72, 73, 88, 89]\) if \( f^\delta \) satisfies the definition, as does \( f^* \), whenever they exist in \( F \). In fact, \( f^* \) is the maximum such function.

Closures and approximating lattices: Fastness closures and pseudotransitive closures give ways of replacing the data flow problem \( D \) by another, \( D' \), frequently with lower accuracy but better time complexity. In general, \( L \) and \( F \) are unchanged, but \( M \) may be modified – this view treats closures as elimination operators. The solution for \( M \) may not have been obtainable in finite time; \( M' \) is guaranteed to be tractable.

A function \( f \) is Rosen rapid with a parameter \( \ell_0 \) if there is a pseudotransitive closure of \( f \) computable with at most \( \ell_0 \) elementary operations (usually meets and compositions) \([72]\). Again, \( F \) and \( D \) have pseudotransitive \( (f^\delta)^\delta \) closure, if the property holds for all \( f \in F \). For pseudotransitive \( (f^\delta)^\delta \) closure, if each \( f \) in \( F \) is Rosen rapid for some single parameter \( \ell_0 \). Data flow problems with fastness closures give an example of a Rosen rapid framework. If the framework is \( \ell \)-bounded, then \( f = f^\delta \) and \( \ell_0 = O(\log_2 k) \); if \( f \) is distributive, then \( f \) agrees with \( f^\delta \). If \( f \) is fast, then \( f = f^\delta \) and \( \ell_0 = 1 \).

Fig. 6-6. Closure properties for function spaces

Coussot and Coussot \([27, 28]\) have an alternative approach. The effect of cyclic flow in an interval is summarized and approximated by replacing flow functions on some of its edges by widening flow functions. A widening operator \( F \), a binary operator like Rosen's spiral, \( @ \), is guaranteed to replace arbitrary

\[ \text{descending chains of meets with finite chains. At least one edge of every cycle must be widened. For reducible intervals, the flow functions on edges out of interval head nodes are widened; otherwise, a cut set of the interval is chosen at random. Since any sufficiently long path in an interval must use some simple cycle more than any given number of times, the sequence of meets of values seen at that point must stabilize, and so eventually must the flow within the interval. We will not consider this approach further in this paper.} \]

6.3. Other framework properties

Other framework properties relate to the representability or locality of the solution. A data flow problem is said to be bitvector \([41]\) if \( L \) can be naturally imbedded in a power set lattice (for example, the power set of definitions for Reaching Definitions), so that each \( f \) in \( F \) operates separately on each element of the underlying set. Bitvector problem lattices can be represented as lattices on bit strings, and functions as standard boolean operations. The four classical problems are usually presented as bitvector problems.

A problem is partitionable \([97, 98]\) if \( L \) is a product lattice and \( F \) acts on \( L \) by acting meet-linearly on each factor separately (that is, on a given factor, every function is constant, or a meet with a constant). In a partitionable problem, graphs can be constructed for each factor problem in which the fixed point can be computed in a single topological order graph traversal (however, graph construction can be costly, and the number of factor problems can be large).

The terms flow sensitive and flow insensitive are often used to distinguish among data flow problems; flow sensitive problems, intuitively, return results which depend more heavily on the paths along which the information could have come. One possible distinction for interprocedural problems is between those affected by intraprocedural changes, and those not affected \([12]\). However, there is no consensus as to the meaning of the terms, and most of the definitions encountered in the literature appear to apply only to subclasses of problems.

6.3.1. Intraprocedural and interprocedural problems

Most data flow problems can be viewed either as set-theoretic problems, involving solution of a set-linear system of equations, or else as closure problems, where the solution is a qualified closure of a relation. The classical problems, and many other intraprocedural problems, are set-theoretic, while the standard interprocedural problems, including Bound Set and Alias, are essentially closure problems, but there are interprocedural closure problems and interprocedural set-theoretic problems. In general, interprocedural problems are harder than intraprocedural problems, and closure problems harder than set-theoretic problems. Interprocedural problems not only have to summarize intraprocedural information, but their flow graphs can have parallel edges, corresponding to multiple calls to the same procedure, and multiple exits (but single-entry) nodes, corresponding to different call sites within the calling procedure. Non-trivial self-loops are also more common. Closure problems have more difficulty with cycles, and usually need to maintain more information.
Problems which cannot easily be expressed as either set-theoretic or closure problems, such as Constant Propagation, tend to be even harder. Moreover, the interprocedural form of any data flow problem is typically harder to solve than the intraprocedural version of the same problem.

6.4. Framework properties of known problems

Reaching (the set of flow graph vertices reachable on paths from ), and its converse, are graph connectivity problems definable in terms of the flow graph alone (independent of the associated code); they are distributive and rapid. The four classical interprocedural problems do depend on the code, but are also distributive and rapid. The semilattice in each case is a cross-product of power sets, and the functions are set functions defined using union and intersection. Reflexive Transitive Closure is closely related to Reaching, but is not a union-intersection problem [76]. The (Forward) Transitive Closure Problem assigns to vertex the set of all transitive closure edges ending at . It is distributive and fast but not rapid. A combination of the forward and backward problems, followed by a single iteration "in place", would then capture at the set of edges resulting from paths through .

The Pint Variable Problem [45] is a set-valued problem; a definition " = (variable , program point )" is faint if it is never used or is used only by faint definitions. This problem is distributive but not fast; in any given instance it is -bounded, where depends on the loop structure and the length of definition chains. Partially Available Expressions [62] is another set-valued problem which is apparently not fast and possibly not distributive. Constant Propagation is monotone but not distributive, and is fast. Regular Expression Computation is definable in a framework. It is distributive but not bounded, since it requires a closure (Kleeen*-). The Flow Cover or Path Expressions Problem for a given graph is essentially equivalent to finding Regular Expressions for the set of paths between each pair of vertices which define the paths in the graph. The interprocedural Bound Set Problem [11] is likewise distributive. In a given instance it is -bounded, where depends on the length of the longest simple calling chain and on orders of permutations of formal parameters induced by a cyclic sequence of calls. Aliasing subsumes the Bound Set Problem. May-Modify and Must-Preserve are interprocedural variants of Reaching; their complexity is at least that of Bound Sets. Must-Be-Defined [7] on the other hand, is in the original sense of flapping [73]; flow-sensitive, and is monotone but not distributive. Must-Be-Defined uses Bound Set implicitly, and must be at least as complex.

Type Inference [91] and Aliasing With Pointers [22, 31, 95] are defined on semilattices of infinite height, but solved on approximating lattices of finite height; Type Inference is also not distributive. The Program Assertion (Loop Invariant) Problem [27] cannot be either bounded or distributive, since it subsumes all data flow problems (on monotone frameworks), but is monotone, since the stronger the precondition of a statement, the stronger the postcondition.

In the absence of pointers, dynamic storage, and procedure variables, Larus [55] shows that precise intraprocedural aliasing on dynamic structures with pointers, in which instances of the same structure are distinguished, need not even be monotone!

7. Complexity results

Algorithms for data flow problems may find an acceptable assignment in either of two distinct ways: either by working with flow functions, finding a summary flow function which expresses the solution, and then applying it to the given and/or to find the assignment, or by computing intermediate values in for the vertices of in combing them to find . Call the former group function-based and the latter value-based [83]. Most iterative methods are value-based, whereas Rosen's Flow Cover method [73] is function-based, and path algebra methods in general are almost entirely function-based. Elimination methods can appear value-based for simple programs, although with some function-based aspects; for more complicated problems, or when used incrementally [21], their essentially function-based nature becomes clear.

Given a data flow problem in which has the d.c.c., or is bounded, value-based algorithms often find a suitable in finite time, while function-based algorithms may need stronger local finiteness properties on or . Further, complexity arguments will require such properties.

Kildall [53] showed that if is distributive and has the d.c.c. (or if each is bounded), and the set of equations has form of 4.1, then iteration initialized at a suitable point () at points without predecessors and elsewhere) converges in finite time to the solution, which agrees with the MOP solution. In a semilattice without a , Kildall's and subsequent results are still valid if computation can be initialized at an upper bound [76].

Kam and Ullman [50] showed that in a distributive data flow framework with Kam-Ullman rapid flow, iteration using DFSST evaluation recurses at most +2 passes if contains a , and at most +3 passes otherwise (the extra pass is used to construct an initial overestimate to the solution), where is the maximum number of DFSST back edges on an acyclic path in (essentially the nesting depth of ).

If has infinite chains, then distributivity is insufficient; Tarjan [88] shows that continuity of , or an essentially equivalent condition on chains of (59), is required, if in addition is -bounded, the MOP solution can be computed in time O(log).

Tarjan [78, 89] gives what essentially is an elimination algorithm for data flow on a reducible flow graph with nodes and edges, using a stratified path-compressed tree data structure [87] whose complexity is .
7.1. Irreducible flow graphs

Tarjan's algorithm can be extended to cover irreducible graphs as follows. We construct the immediate dominator tree, construct a derived graph using the original flow graph and the tree, and find strong components in the derived graph. These strong components capture irreducibility. The non-trivial strong components are solved as regular expressions, and are represented by vertices in a reduced and irreducible graph, which is then solved by the previous algorithm. The complexity is \(O(m \log m + n + b)\), where \(b\) is the sum of the lengths of the path expressions for the irreducible components.

There are other methods for handling irreducible flow graphs. If iteration works in a given function space, it will work regardless of whether the flow graph is irreducible; only the applicability of particular efficient schemes and the complexity of obtaining the answer may differ. Some elimination algorithms [39] work for irreducible graphs, but with modifications which affect complexity. Others [11, 81] use iteration on improper regions, minimal singular-entry but not necessarily strongly connected regions which contain the irreducible region. This introduces a factor of \(r\) - the number of regions (intervals and improper regions) - into the complexity of computing the intervals, with additional cost for iteration.

When the set of equations Q admits a Gaussian-elimination-type solution, this method can handle arbitrary flow graphs, although efficient schemes for solution may no longer be applicable. One could also replace the framework by an equivalent framework in which the flow graph was reducible. Node splitting [41] does this, with a possibly substantial increase in the graph size, and therefore complexity.

7.2. MOP and MFP

Kam and Ullman [50, 51] showed that iteration with an appropriate \(X_0\), applied to a monotone framework, will converge to the MFP solution. They further showed

1. that different choices of \(M\) and \(\eta\) may have different MFP solutions, even with the same MOP solution and, in fact, that different algorithms may find the MOP solution for different classes of frameworks (compare Sect. 4.3); however,
2. that no algorithm will compute the MOP for all monotone data flow frameworks (that is, for any algorithm, there is a flow graph for which the MOP solution is strictly greater than the MFP solution for the equation set);\(^{59}\) and, further,
3. that finding the MOP is an undecidable problem.

Also, some choices of \(M\) and \(\eta\) may lead to MFP solutions less accurate than that for \(M\), but perhaps easier to compute. Pseudotransitive closures deal with this problem. If \(L\) is not closed, a sequence of iterates need not have a greatest lower bound, and \(f^*\) may not be in \(F\). However, an approximation for the equation set may exist for which the sequence of iterates does have a MFP. Further, even when \(F\) contains \(f^*\), use of an approximating \(f^k\) may improve the complexity of finding the MFP.

Frequently, \(f^k\) can be taken to be the closure \(f^*\) in an approximating data flow problem (with appropriate approximating semilattice \(L\) or function space \(F\), or assignment \(M\)); if \(M\) and \(\eta\) have not been changed, then \(f^k\) will be less than or equal to the original \(f^*\) [91]. Iteration (possibly using closures) will reach \(f^k\), which will still give an acceptable solution for the original problem.\(^{60}\) Use of \(f^k\) will not always yield a fixed point, but the resulting assignment will still be less than the original MOP solution, and usually less than the original MFP solution as well.

Although algorithms such as Graham-Wegman elimination can easily find the MFP for fast, monotone frameworks, neither iterative nor elimination algorithms on data flow frameworks, nor path algebra without an oracle for the closure relation, can solve exactly even every data flow problem, since by the results of Kam and Ullman above, there are non-distributive frameworks which for any algorithm have instances which either cannot be solved by that algorithm, or yield an MFP strictly less than the MOP solution.\(^{61}\) Thus no alternative to data flow frameworks and path algebra will always give precise solutions.

8. Implications among flow framework properties

We extend the Venn diagram of Fig. 6-3 for algebraic properties in Fig. 8-1. In Fig. 8-2, we give an implication diagram for finiteness and local closure properties. We have merged the lattice and function space finiteness diagrams; properties in italics are lattice properties, those in bold are function space properties. We also show part of the interaction between algebraic and finiteness/closure properties in Fig. 8-5, and summarize our problem classification results in Table 8-1.

\(d,c,c\) and algebraic properties: \(d,c,c\) in \(L\) is sufficient to guarantee that distributivity implies continuity in \(F\), since every meet of values in \(L\) is equivalent to a finite meet. Boundedness of \(F\) together with distributivity will also give exact results for the MOP solution, at least for reducible flow graphs. (This can be seen by considering elimination. The only infinite meets involve sequences of iterates of functions representing flow around intervals, but these are equivalent to finite meets by boundedness.)

\(^{59}\) If \(G\) has only a finite set of paths we can do better (see Sect. 5.2)

\(^{60}\) Note that \(f^*\) is like a reflexive transitive closure, so is an MFP of sorts. \(f^k\) can often be considered the MFP in an approximation.

\(^{61}\) Note that such an algorithm would determine sharp loop invariants.
Properties of data flow frameworks

L and F contains functions 0, 1, and the identity 1, and is closed under composition and pointwise meet. Since all of these lattices are finite, we have also shown the same result for continuous function spaces.

Finally, we show that k−1 and k boundedness are distinct in each of the algebraic classifications. Undoubtedly, similar examples could show (k−1)-boundedness, (k−1)-semiboundedness and k-boundedness are distinct in each case. Figure 8-3 summarizes our conclusions.

8.1. Independence of algebraic and finiteness properties—examples

Examples of fast monotone and rapid distributive functions occur throughout the literature, as do examples of monotone frameworks which are not fast. The Flow Cover framework is continuous and therefore distributive but not bounded; in fact, the underlying lattice does not have the d.c.c. If F = the space of all set maps from L to L for a sufficiently large lattice L, the resulting framework will be neither monotone nor bounded.

We give examples of the other four possibilities below, with the following conventions. We distinguish lattice elements 0 and 1 from constant functions by omitting underscores on the former and using overscores on the latter, and we write a function f which takes a to b and c to d as f(a,b) = (c,d). The lattices and function space lattices are illustrated in Fig. 8-3.

Ex. 8.1: A rapid but non-monotone framework:

Let \( L_1 = \{0, a, 1\} \) and \( F_1 = \{i, \theta, T, f(0, a, 1) = (1, a, 1)\} \)

Then \( F_1 \) is rapid but \( f \) is not monotone.

Ex. 8.2: A fast but non-monotone framework:

Let \( L_2 = \{a, b, 1\} \) with \( a \leqslant b \), and

\[
\begin{align*}
F_2 &= \{i, \theta, T, f(0, a, b, 1) = (0, b, a, 1), g(0, a, b, 1) = (0, a, b, 1), h(0, a, b, 1) = (0, b, a, 1)\}
\end{align*}
\]

Then \( F_2 \) is fast but not rapid, and \( f \) is not monotone.

Ex. 8.3: A rapid and monotone, but not distributive, framework:

Let \( L_3 = \{a, b, 1\} \), with \( a \) and \( b \) in comparable, and

\[
\begin{align*}
F_3 &= \{i, \theta, T, f(0, a, b, 1) = (0, b, b, b), g(0, a, b, 1) = (0, a, a, a)\}
\end{align*}
\]

Then \( F_3 \) is rapid and monotone, but \( f(a \land b) \neq f(a) \land f(b) \), so \( F_3 \) is not distributive.
Properties of data flow frameworks

Ex. 8.4: A fast and distributive, but not rapid, framework:
Let \( L_4 = \{0, a, b, c, 1\} \), with \( a, b \) and \( c \) incomparable, and
\[
F_4 = \{(1,0,1, f(0,a,b,c,1) = (0,b,c,a,1)),
\]
\[
 f^2 = f \circ f \text{ and } g(0,a,b,c,1) = (0,0,0,0,1)).
\]
Then \( F_4 \) is distributive and fast, but \( f(a) \) is not greater than or equal to \( a \wedge 1 \), so \( F_4 \) is not rapid.

The distinctions involving continuity cannot of course be illustrated with finite lattices; we offer some easy infinite examples.

Ex. 8.5: Some examples from product lattices:

a. In an infinite product lattice framework, with \( L = \prod_{i=1}^{m} L_i \), \( L \) will not have d.c.c. even if the factors are as simple as \( L_i = \{0, 1\} \) for all \( i \); but \( F \) is the set of all componentwise monotone functions on \( L \) (with factor lattice \( \{0, 1\} \)) will be rapid and continuous.

b. If \( F \) is restricted to \( F = \{\text{the functions } f \text{ and } g \text{ together with the set of functions which take any point with an infinite number of components } 0 \text{ to } 1 \text{ then } F' \text{ will be distributive but not continuous.}

Similarly, if \( L' \subseteq L \) consists of elements of \( L \) which are almost everywhere constant (only a finite number of \( 0 \)'s, then \( L' \) will not be closed, so that the set of all distributive functions on \( L' \) will not be continuous.

d. Letting \( L_4 \) be the lattice of Example 8.3, and \( L' \) the elements of \( L \) which are almost everywhere \( 0 \) or almost everywhere \( 1 \), the set of all distributive functions is \( 2 \)-bounded and not continuous. Further, there is a proper subset of this class, spanned by the component functions

\[
F_4 = \{(1,0,1, f(0,a,b,1) = (0,b,c,1)),
\]
\[
g(0,a,b,1) = (0,0,0,0,1)).
\]

which is fast but not rapid.

e. Further, there are small lattices \( K \) so that the infinite product lattice all of whose factors are \( K \) will have the space of monotone and the space of distributive functions distinct.\(^a\)

8.2. Algebraic properties and boundedness

We now give examples of small frameworks which are \((k+1)\)-bounded but not \( k \)-bounded; the lattices for these examples are illustrated in Fig. 8-4.

Ex. 8.6: Distributive, \((k+2)\)-bounded but not \((k+1)\) bounded:
Given \( k \geq 1 \), let \( L_4 = \{0, x_1, x_2, \ldots, x_k, 1\} \), with the linear ordering given by \( x_i \leq x_{i+1} \), and let \( F_i \) be the space of monotone functions on \( L_i \).

\(^a\) Lattices in which meet does not distributive over join (and vice-versa). \( K \) can have as few as five elements \([10]\)
Note that if \( f(0, x_1, x_1) = (0, 0, x_{i-1}, x_i, x_{i+1}) \in F_1 \), \( F_1 \) is distributive (since it is a linear order) and \((k+2)\)-bounded since any (decreasing) chain \( \{ f^{(s)} \} \) of length \( k+2 \) reaches \( 0 \) or a fixed point, but \( f^{(s+1)}(1) = x_i = 0 = f^{(s+1)}(1) \).

**Ex. 8.7: Monotone, \((k+2)\)-bounded but not \((k+1)\) bounded:**

![Lattices for Sect. 8.2](image)

Given \( k \geq 1 \), let \( L_2 = \{ (0, x_1, y_1, 1) \}_{x_1=1} \), with \( x_{i-1} = y_i \land z_i \), and likewise cyclically. Again take \( F_2 \) to be the space of all monotone functions on \( L_2 \). Again, \( F_2 \) is \((k+2)\)-bounded and not \((k+1)\) bounded. Also, \( g(0, x_1, y_1, z_1, x_2, y_2, z_2, 1) = (0, x_{i+1}, y_{i+1}, z_{i+1}, 0, 1, 1, 1, 1) \) is in \( F_2 \), but is not distributive, since \( g(x_1 \land y_1) = g(x_1) = z_1 + 1 = g(y_1) \). (For \( k = 1 \), non-distributivity still holds, but the proof is different.)

**Ex. 8.8: Non-monotone, \((k+2)\)-bounded but not \((k+1)\)-bounded:**

Let \( L_2 = \{ (0, x_1, ..., x_2, 1) \} \), again with the linear order, and let \( F_2 \) be generated by \( \{ 0, 1, f(0, x_1, x_1) = (1, 0, x_{i-1}, x_i) \} \). \( F_2 \) is not monotone. For each \( f \in F_2 \) and \( x \in L_2 \), the set \( \{ f^{(i)}(x) \}_{i=0} \) is invariant for \( r \geq k+1 \), and thus \( F_2 \) is \((k+2)\) bounded. However, \( f^{(s+1)}(1) = 0 + x_i = f^{(s+1)}(1) \), so \( F_2 \) is not \((k+1)\)-bounded.

Although the properties of non-monotone functions are in general uninteresting, the following may be useful. If \( f \) is anti-monotone and \( f^2 \) is \( k \)-bounded, then \( f \) will be 2-\( k \)-bounded, since every iterate of \( f \) will be greater than \( (f \land i)(f^2)^{\infty} \). For instance, the set-valued function (for \( X \) -the complement of \( X \)) \( f(X) = X \cap B \cup C \) has \( f^2(X) = X \cap B \cup C \), and is 4\-bounded without being monotone.

**Reaches through problem:** Most uniformly bounded problems in the data flow literature are either fast, or are \( k \)-bounded for some \( k \) determined by the code. We now give a problem, naturally expressible in a lattice framework, \( n \)-bounded independent of code or flow graph. The (Forward) **Reaches Through Problem** with parameter \( n \) returns for a vertex \( v \) the set of all paths of length at most \( n \) ending at \( v \), together with the set of all length \( n + 1 \) subsequences of paths ending at \( v \); that is, each path of length greater than \( n \) will be represented by its endpoints and each set of \( n + 1 \) intermediate vertices. A definition of \( RT_2(n) \) follows:

\[
RT_2(n) = A \cup \bigcup_{\rho \in M(n)} M(e) \cdot RT_2(\rho)
\]

where

\[
M(e) = \rho \cup \{ (w, u, v) : (w, u, v) \in X \} \cup \{ (x, y, v, (x, y, v) : (x, y, v) \in X \}
\]

Note that \( RT_2(n) \) is a set, and not a multiset. We show \( RT_2 \) is distributive, and \( 3 \)-bounded, but not \( 2 \)-bounded.

For simplicity, consider a flow graph \( G \) with a unique cycle \( C \) through a vertex \( v \). \( (v, u, v) \in RT_2 \), and can only be a subsequence of a path containing \( C \). Thus two passes around \( C \) are needed to capture some information, so \( RT_2 \) cannot be \( 2 \)-bounded. On the other hand, given any element \( (w, u, v) \in RT_2 \), there must be some simple path (or simple cycle) from \( w \) to \( u \), and likewise from \( u \) and \( v \). Thus every element of \( RT_2(n) \) is a subsequence of some path in \( G \) with no more than two cycles. Thus \( RT_2 \) is \( 3 \)-bounded. Likewise, \( RT_2 \) is \( n + 1 \)-bounded, but not \( n \)-bounded.

Table 8.1 gives an overview of problems in our property classification.

![Interaction of algebraic and local/finiteness/closure properties](image)
are several essentially independent dimensions of framework properties, a conclusion unachievable by mere inspection of previous results.

This study provides a solid foundation for further theoretical and practical investigations of iteration and elimination as solution procedures; it is specifically directed at design of incremental update algorithms for fixed point iteration [14, 76].

References

Properties of data flow frameworks


44. Hopcroft, J., Ullman, J.D.: Introduction to automata theory, languages, and computation. Reading, MA: Addison-Wesley 1979


Properties of data flow frameworks


