Acta Informatica 28, 121–163 (1990)

# 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

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 for arriving at that solution, and complexity of the method used.

Our work provides an overview of data flow 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 relationship 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 path 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[1] 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 [6, 27], debugging [96], testing [68], and parallelization [13, 32, 37, 66], vectorization [4], and parallel programming and environments [19, 36, 61, 80, 84]. They provide information about a program or environment without executing the code.[2]

Naturally, it is impossible to determine the exact output of a program, or the operation of an environment, without actual execution; this clearly subsumes 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: *intraprocedural* and *interprocedural*. Other applications are at a still lower (e.g., statement [27]) or higher (e.g., module, or parallel thread [84]) 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. Intraprocedural analysis deals with the statements or basic blocks [2] of a single procedure and with transfers of control between them; interprocedural 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 $\rho$, corresponding to the initial block of the program, or the procedure *main*, whose indegree is usually taken to be zero.[3,4] 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[5], 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 $u$ appearing in the equation for the solution

[1] Or procedure, module, software system, etc.
[2] Often called *compile-time information*
[3] But seen 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 *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 *forward* or *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 *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 *extreme solution* [27, 67].[7] 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 Kam and Ullman [50, 51] and Graham and Wegman [39], has the form of Fig. 3-1.

In the literature, **D** is usually called an *instance of a data flow framework* or a *data flow problem*, and *data flow framework* is reserved for the pair $\langle L, F\rangle$. 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 *edge transition function* or *label function.*

### 3.1. An example: Reaching definitions

The *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:

[6] In some cases, a subset of one or the other [43]

[7] Manna and Shamir [57, 58], however, give frameworks for which they claim the *optimal fixed point*, which can often be computed but in general is not extreme, is the best solution

[8] For a more thorough review of lattices see [10, 27]

[9] Also note that $F$ may be given by a generating set S, whose closure under meets and compositions (and closures) specifies $F$

$\mathbf{D} = \langle G, L, F, M \rangle$, where

$G = \langle V, E, H \rangle$ is the flow graph,
where $H$ = the entry node set
(usually $H = \rho$ is the unique entry node
and the indegree of $\rho$ is zero),

$L = \langle A, \underline{0}, \underline{1}, \leqq, \wedge \rangle$ is a meet semilattice, that is,
$A$ is a set (often a power set),
$\underline{0}$ and $\underline{1}$ are distinguished elements of $A$,
$\leqq$ is a reflexive partial order,
and $\wedge$ is an operation *meet*
with the following properties:
$\wedge$ is commutative and associative
$a \leqq b$ iff $a \wedge b = a$
$a \wedge a = a$
$a \wedge b \leqq a$
$a \wedge \underline{0} = \underline{0}$
$a \wedge \underline{1} = a$

$F$ is a class of functions
$F \subseteq \{f\colon L \to L\} = L^L$; $F$ contains the identity function $\iota$,
and usually the constant functions $\bar{0}$ and $\bar{1}$, and
is closed under composition and pointwise meet (that is,
$\forall f, g \in F\colon f \circ g \in F$
(where the notation $f^k$ represents iterated composition of
$f$ and $f^0 = \iota$),
and, if $h(x) = f(x) \wedge g(x)$, then $h \in F$);
and $M\colon E \to F$
(or $M\colon V \to F$ by ignoring the source (or target) of the edge).

If $p = (p_0, p_1, \ldots, p_n)$ is a path in $G$
with $e_i = (p_{i-1}, p_i)$,
(if $G$ is a multigraph, then there are *source* and *target* maps
$s, t\colon E \to V \times V$, and the criterion on $e$ is
$(s \times t)(e_i) = (p_{i-1}, p_i)$)
then $M(\Lambda) = \iota$ where $\Lambda$ is the empty path
and $M(p) = M(e_n) \circ M(e_{n-1}) \circ \ldots \circ M(e_1)$
for forward problems.
For backward problems the proper definition is
and $M(p) = M(e_1) \circ M(e_2) \circ \ldots \circ M(e_n)$.

**Fig. 3-1.** Data flow framework definition

Which definitions of $X$ reach a given use of $X$ in an expression? Is $X$ used anywhere before it is defined?

A definition $d$ of $X$ in block B is *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:

$$\text{Reach(B)} = \bigcup_{B' \in Preds(B)} [\text{Reach(B')} \cap \text{Pres(B')} \cup \text{Gen(B')}]$$
$$\text{Reach}(\rho) = \varnothing, \tag{3.1}$$

where Reach(B) is the set of definitions reaching the top of B, Pres(B) is the set of definitions *preserved* through B (that is, not superseded by more recent definitions), and Gen(B) is the set of downward-exposed definitions *generated* in B.

Reaching definitions is a forward problem; the flow graph $G$ and the dependency graph are the same. $L$ is the power set lattice on the set of downward-exposed definitions in the procedure (with the reversed order, meet is union, $\underline{1} = \varnothing$ and $\underline{0}$ = the universal set of all downward-exposed definitions). The reverse ordering is required to make the meet semilattice model fit, since the lattice paradigm with meet corresponds naturally to "intersection" problems, in which facts are falsified and discarded, rather than affirmed and inserted [14]. $F$ is the set of functions $\{f(X) = X \cap A \cup B \mid A, B \in L\}$; and $M$ is the mapping assigning to an edge (B, B′) the function $f(X) = X \cap Pres(B) \cup Gen(B)$.

We will see that it is significant that $L$ is finite, and the functions of $F$ have certain nice properties: each $f \in F$ is idempotent ($f \circ f = f$), and "$f$ distributes through union", $f(A \cup B) = f(A) \cup f(B)$. Also, $M(e)$ does not depend on the target, but only on the source of $e$, so $M$ could be viewed as defined on $V$ alone.[10]

### 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. $\wedge \varnothing$ is by definition $\underline{1}$, so a semilattice admits the empty meet if and only if it includes $\underline{1}$. Although the definition of data flow framework in Sect. 3 included an element $\underline{1} \in L$, this is not required in general. The presence or absence of the $\underline{1}$ will principally affect complexity. It is frequently possible to add an artificial $\underline{1}$ and retain the properties of the framework. Also, as Rosen [72] points out, a semilattice with $\underline{1}$, closed under infinite meets, is in fact a lattice (take $x \vee y = \wedge \{z : z \geqq x \text{ and } z \geqq y\}$).

The use of meet is in some sense arbitrary; by duality [10], everything could be phrased in terms of *join semilattices* instead. In a join semilattice, references to "meet over all paths", "maximum fixed point", "greater solution", should be replaced respectively by "join over all paths", "minimum 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 $\underline{0}$ and $\underline{1}$, we can consider join as the meet in the *opposite order* ($X \leqq_{opp} Y$ iff $Y \leqq X$). Alternatively, in any lattice with a complement-like involution $c$ ($c$ is bijective, $c \circ c = \iota$ and $x \leqq y$ implies $c(y) \leqq c(x)$), 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 otherwise specified.

[10] Note however that if domain $M$ is considered to be $V \times V$ (rather than $E$), then the target is used to decide if the value of $M(v, w)$ is as specified (if there is an edge), or $\underline{1}$ (otherwise)

In [28], Cousot and Cousot classify data flow frameworks by tuples 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 $\rho$ an initial value $\eta$, the *facts true on entry* to the program (*facts true on exit* from the exit vertices if D is a backward problem). Often $\eta$ will be $\underline{0}$ (no facts are true on entry).[11] 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 $\rho$. The same trick will if necessary create a unique exit vertex $\rho'$, or sink. Rosen [43, 72, 73] allows an entry-value function $Ent$: $H \rightarrow L$ assigning arbitrary values on entry to vertices in the entry node set, which he allows to be greater than just the singleton $\{\rho\}$.[12] This simplifies unified treatment of forward and backward problems (since a unique exit 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 $H = \{\rho\}$, there is a reasonable assumption that all nodes in $V$ are reachable from $\rho$. 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 $X$: $V \rightarrow L$ and the values of $M$ [78]. The equations depend on whether the problem is forward or backward, on the initial value $\eta$, 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 $X_0$: $V \rightarrow L$. In many cases, the initial guess will be $\underline{1}$ at non-entry nodes; it must be $\eta$, usually $\underline{1}$ or $\underline{0}$, at $\rho$.[13] For Reaching Definitions, the initial guess $X_0$ is $X_0(v) = \underline{1} = \emptyset$ for all $v \in V$. 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 $\langle Q, X_0 \rangle$ is implicit in the framework; the set of possible values for $X_0$ is restricted by the problem, and given $X_0$, 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 *subposet:* if $L = \langle A, \underline{0}_L, \underline{1}_L, \leqq_L, \wedge_L \rangle$ and $K = \langle B, \underline{0}_K, \underline{1}_K, \leqq_K, \wedge_K \rangle$, then $A \subseteq B$ and $\leqq_L$ is the restriction of $\leqq_K$ to A. Then

$$x \wedge_L y \leqq x \wedge_K y,$$

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 meets are equal, and $L$ is a *subsemilattice*

[11] Horwitz et al. [45] allow $\underline{0}$ or $\underline{1}$

[12] Rosen's notation differs significantly from the notation used here

[13] More generally, $X_0(v) = Ent(v)$ 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 subposets of a single natural (but intractable) semilattice; occasionally, however, there may be no obvious natural lattice formulation. In these cases there may seem to be several possible incompatible lattice formulations, or no natural formulation at all, when the lattice itself will seem somewhat of an artifice, as in *Constant Propagation*[14] (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 [43] 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*[15]; 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

[14] This problem consists of ascertaining for each program point, that set of variables which have constant value at that point; it can be formulated as an intraprocedural or interprocedural problem

[15] Or *or*

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 "abstracting" 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 reference.[16]

However, the approximating semantics must reflect the underlying formal semantics[17]; 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 estimate[18] 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

$$X(v) = \wedge \{M(e)(X(u)) : e = (u, v) \in E\}, \tag{4.1}$$

where each equation may also include a constant or node-dependent term $W(v)$.[19] 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.[20]

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

Backward-Problem – Information on Node Exit

$$Y(u) = \wedge \{M(e)(Y(v)) : e = (u, v) \in E\} \tag{4.2}$$

*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"

[16] Somewhat as in [95]

[17] Larus [55] notes an example where this does not occur

[18] See Sect. 4.3

[19] When $M$ is considered a function on the nodes, then $M((u, v))$ will be equal to $M(u)$ for forward/entry problems

[20] For $M$ a node function, $M((u, v))$ will be $M(v)$

approximation[21] $X_0$, and repeatedly substitute current values into the right side of each equation, assigning the resulting value ot the variable on the left. If at some step the output is the same as the input, the algorithm has reached a fixed point. A solution $Y$ is a *fixed point* of the set of equations Q if and only if for all vertices $v$,

$$Y(v) = [Q(Y)](v)^{22} \tag{4.3}$$

A *maximum fixed point* **MFP** is a fixed point which is greater than or equal to any other fixed point. Minimum fixed points can be defined similarly. Iteration procedures differ in the order in which they evaluate equations, and whether changed values are propagated immediately or after a group of evaluations.[23]

*Elimination* procedures [3, 39, 42, 89] are analogous to Gaussian elimination, and use flow graph decomposition and reduction to derive ever smaller systems of equations which can then be solved in a simple way. A survey of elimination methods, stated for the classical problems, appears in [78]. Elimination methods differ in the transformations they apply to move from one dependency graph/equation set to the next, and in the conditions under which the transformations will be applied. Such methods generally work effectively on a restricted class of flow graphs.

The third solution method uses *path algebra* [72, 73, 88, 89]. The solution procedure is strongly analogous to the procedure for solving for regular expressions between all pairs of nodes in the labelled flow graph [44]. Its approach is closer to elimination methods than iteration in flavor, but it does not necessarily assume the existence of a lattice framework [88], and does not necessarily give a solution to a set of equations.

## 4.2. Solutions

Compile-time solutions to run-time problems can only be *precise up to symbolic execution;* that is, they can yield precise results under the assumption that all paths in the flow graph are executable.[24] In general, we will not even be able to generate the most desirable compile-time information; issues of computability and efficiency are involved (see Sect. 7). We can however discuss the solution given by a solution procedure for a set of equations Q.

Solutions assign a value in $L$ to each vertex $v \in V$, and are thus $|V|$-vectors of semilattice points. For an assignment $X: V \to L$ to be valid, another evaluation

[21] Namely, one guaranteed to be higher in the semilattice than the desired solution (*lower than* for join semilattices)

[22] In this definition, fixed points of Q are precisely solutions of the equation $X = Q(X)$

[23] An alternative formulation replaces equality in Eqs. 4.1 and 4.2 with $\leqq$ (with $\geqq$ for join problems). Because iteration begins above the solution, and iterates to the *maximum* fixed point, these two formulations, and a third, given by 5.1, Sect. 5.5, are equivalent for problems solved in the lattice framework. We use the term *fixed point* of Q interchangeably for these concepts

[24] This is not quite correct. We may be able at compile time to determine that *some* paths will not be executed [43, 94], and adjust the solution accordingly. We cannot, however, expect to find *all* non-executable paths; this is equivalent to the halting problem

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,

$$\forall (u, v) \in E: \; X(v) \leqq M((u, v))\, X(u). \tag{4.4}$$

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

$$MOP(v) = \wedge \{M(\rho)\, \eta \mid p \in Paths(\rho, v)\}$$

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

$$X(v) \leqq M(p)\, \eta \tag{4.5}$$

for every $v \in V$ and every path $p \in Paths(\rho, \mathrm{v})$.

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 $v \in V$ and all fixed points $Y$ of Q, $X(v) \geqq Y(v)$. 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).

[25] A "safe" initial approximation for iteration will not ordinarily be a safe solution, nor conversely. For a meet semilattice, a safe approximation wil 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

[26] 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 points 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 even 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 an optimal solution to 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 path algebra framework, or because the lattice or the function space for the original problem may be difficult or expensive. In *Aliasing*[27] [7, 8, 23, 95] and *Type Determination*[28] [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 iterations[29] [28], or by apparently non-iterative methods [30, 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 in [51], 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 $\eta$ 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

[27] When two distinct symbols refer to the same storage location simultaneously during program execution they are termed *aliases*

[28] This data flow problem determines types of variables in implicitly typed programming languages

[29] If the approximation is above the MFP for the set of equations used, iteration with its solution will not improve the value

result in semantics-changing "optimizations", while an unacceptable assignment will ordinarily result in not performing valid and usually easily discovered optimizations. Thus safety and acceptability seem to be relevant criteria. However, even though an assignment is safe and acceptable, it has these properties 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 (DFST) of the flow graph (rooted at $\rho$), or *dominators*.[30]

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 consits of a nested set of single entry regions [78]. 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 $\rho$ are distinct (except that $a$ and $\rho$ may be the same) [92]. A graph is *converse-reducible* if its converse graph is reducible. In practice [41], most program flow graphs are reducible.

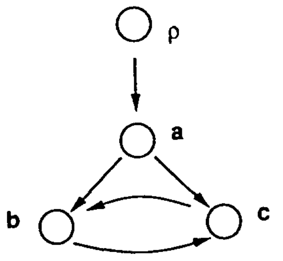

**Fig. 5-1.** The canonical irreducible graph

Intuitively, the *nesting depth*[31] 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

[30] Relative to $\rho$, $u$ dominates $v$ if every path from $\rho$ to $v$ passes through $u$

[31] Or loop-connectivity parameter

[32] The nesting depth appears to be bounded by a small constant for intraprocedural flow graphs [54, 78]

on a decomposition of the flow graph into *intervals* [3, 78, 85, 89]. Tarjan intervals are essentially equivalent to program natural loops [2],[33] 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 $|E| = O(|V|)$.[34]

## 5.2. Cycles

If all flow graphs were acyclic, then paths from $\rho$ to $v$ would form a finite set of paths of bounded length, and the MOP solution $\wedge \{M(p)(\eta) | p \in Paths(\rho, v)\}$ could be computed accurately and efficiently, simply by following all such paths and comparing the information obtained.[35]

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 evaluation using the most-recently derived values for each variable.[36] The latter approach has scheduling variants, including non-deterministic, nodelist, round-robin and worklist meth-

[33] The outer interval containing $\rho$ is acyclic

[34] In intraprocedural contexts, some models, for example, in [86], explicitly require that no node have outdegree greater than 2

[35] Even if none of the standard non-exhaustive procedures would find that value (see Sect. 7)

[36] Cousot and Cousot [27, 29] show that such *chaotic* or asynchronous iterative algorithms behave properly (see also [67]). In [16], iterates are allowed to lie *between* the old value (RHS of the equation) and the new (LHS)

ods [41, 86]. Particularly useful is a round-robin method in which evaluation order is driven by reverse postorder on the DFST of the flow graph.[37]

Horwitz et al. [45] 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 $k^{\text{th}}$ 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 $t$.[38]

## 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.[39] 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.[40]

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.

[37] Since reverse postorder is closely related to the interval structure of the flow graph
[38] Chaotic iteration will also need such a mechanism
[39] Note that these implicitly also encode the decomposition
[40] 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 [88] 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 reuse 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. Fong's *Incrementally Available Expressions Problem* 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 framework, 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

$\langle \Lambda$, edge $e$, concatenation, union, Kleene-star$\rangle$
by
$\langle \iota$, $M(e)$, composition (in reverse order), lattice meet, closure$\rangle$,

where *closure* handles cycles/intervals, 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 $\Lambda$ by the identity function $\iota$, preserving all definitions, $e$ by the function defined by the *Pres* and *Gen* labelling edge $e$, meet by union, and closure also by the identity, since Reaching Definitions is *rapid* (see Sect. 6.2). For Bound Set, $\Lambda$, concatenation, and meet have similar definitions; $e$ is replaced by the "lozenge" operation with local bindings $A$, $M(e)(X) = X \circ A \cup A$, 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

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

Covers, one can then instantiate the values and operations for the particular data flow problem, as compared to instantiating at each opportunity. One then obtains a safe and acceptable approximation for the solution of the given problem. The Flow Cover Problem is thus a *universal object* for distributive data flow.[42]

Rosen's data flow analysis algorithm uses these contexts and a graph decomposition related to intervals. Tarjan's general path algebra algorithm [89] is an elimination-like algorithm which computes path expressions for the *irreducible components* of a flow graph – which are trivial if $G$ or $G^r$ is reducible – and combines these by applying reduction operators similar to those of Hecht and Ullman [42] or Graham and Wegman [39].

Path algebra can handle some problems whose solution cannot be expressed as the solution to a set of equations, because when not all paths are used, paths of length $(n+1)$ are not necessarily generated from paths of length $n$ by adding edges in a uniform manner. In these cases we cannot think of path generation as iteration with a fixed function, and we cannot assume a unique maximal fixed point which is also the solution of a set of equations. Thus, fixed points are often defined[43] by inequalities rather than equalities [43, 72, 88].[44] $Y$ will be a fixed point if it satisfies the inequalities

$$\begin{aligned} &Y(v) \leq M(e)\, Y(u), \\ &Y(h) \leq Ent(h) \\ &\quad \text{for all } h \in H \quad \text{and} \quad e=(u,v) \in E. \end{aligned} \tag{5.1}$$

## 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 defs and uses of the form $x=x$, 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.

[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 PVT 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.[45] 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 intraprocedural reaching information. The PDG and variants use *ud* 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. The PDG breaks successor edges between statements which are neither control- nor data-dependent. The PSG expands the call graph, but replaces each (interprocedural) node with a highly condensed copy of the corresponding intraprocedural flow graph.
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].[46] 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*, *anti*, *output*, and as *loop-carried* or *loop-independent*, and as *forward* or *backward*), 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

[45] Zadeck's PVT method can be viewed in a similar light
[46] Compare also the representative edges of [21]

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 [43, 94] 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 often 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. Finite 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

$$x_1 > x_2 > \dots$$

is finite.[49] If $L$ satisfies the d.c.c., then every meet $\bigwedge_{\alpha \in A} v_i$ of elements of $L$ is equivalent to the meet of a finite subset. $\left\{\bigwedge_{i=1}^{k} v_i\right\}_{k=1}^{\infty}$ 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 $\bigwedge_{i=1}^{j} v_i$ is not equal to the infinite meet, then there is a $v_r$ so that $\bigwedge_{i=1}^{r} v_i$ is strictly smaller. But by d.c.c., this process must eventually halt.

If, in addition, the length of chains with $x_1 = b$ is bounded by some function of $b$, we say that $L$ has *finite height* [41];[50] if it is bounded by some constant $k$ independent of $x_1$, then we say it has *strictly finite height*, and has *height* $k$ for the given value of $k$. For a semilattice with $\underline{1}$, $L$ has finite height if and only if it has strictly finite height; however, the natural numbers $\mathbf{N}$ (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 $\omega + 1 = \mathbf{N} \cup \{\omega\}$, is a semilattice with d.c.c. but not finite height [10].

| | |
|---|---|
| Closed | Closed under arbitrary meets. |
| d.c.c. | All descending chains are finite. |
| Height k | Length of a descending chain is at most k. |
| Finite Height | Length bounded by function of first element. |
| Strictly finite height | Height k for some k. |
| Complete | (With respect to a function space $F$): Evaluation takes $F \times \underline{0}$ onto $L$. |

**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 $f(\underline{0})$ for some $f \in F$. Graham and Wegman [39] remark that, given $G, F$ and $M, L$ can always be taken to be only the values actually reachable from $\underline{0}$; we can restrict to $L'$, the set of points expressible as $\bigwedge_{i \in A} [M(p_i)](\underline{0})$, 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, some infinite sequence of function iterates may not have a lower bound. Let $C$ be a cycle and consider, as usual, the set of paths corresponding to zero or more passes around $C$,

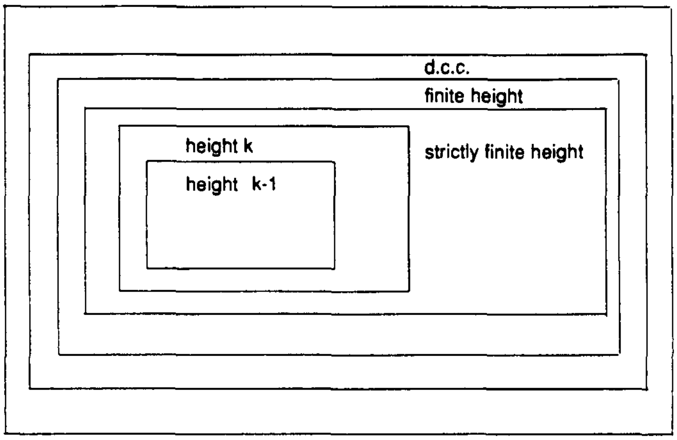

**Fig. 6-2.** Lattice local finiteness properties

[50] Or is *well-founded* [11]

$\{\iota, M(C), M(C|C), M(C|C|C), \ldots\}$. 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 $F$:

$$\begin{array}{ll} f \leqq_F g & \text{if and only if for all } x \in L, \quad f(x) \leqq_L g(x), \quad \text{and} \\ h = f \wedge g & \text{if and only if } h = \max\{j \mid j(x) \leqq f(x) \wedge g(x)\, \forall x \in L\}. \end{array}$$

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 $\iota$, and the constant functions $\bar{0}$ and, if there is a $\underline{1} \in L$, $\bar{1}$.

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.[51] 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.[52] $F$ 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 $C$, $\bigwedge_{i=0}^{\infty} f^i$ represents "$M(C^*)$". 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

$$\forall f \in F\, \forall x,\ y \in L\colon\ x \leqq y \rightarrow f(x) \leqq f(y),$$

[51] 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

[52] That is, maps in $F$ preserve algebraic structure in $L$

and *anti-monotone* if

$$\forall f \in F \forall x,\ y \in L\colon\ x \leqq y \rightarrow f(x) \geqq f(y).$$

Monotonicity is equivalent [41, 50] to

$$\forall f \in F \forall x,\ y \in L\colon\ f(x \wedge y) \leqq f(x) \wedge f(y).$$

$f$ is *distributive* if

$$\forall f \in F \forall x,\ y \in L\colon\ f(x \wedge y) = f(x) \wedge f(y),$$

and *continuous* if $f$ is a morphism of closed semilattices, that is, $L$ is closed under arbitrary (non-empty) meets and

$$\forall f \in F \forall \text{ non-empty sets } \{x_i\}_{i \in I} \subseteq L\colon\ f(\bigwedge_{i \in I} x_i) = \bigwedge_{i \in I} f(x_i).$$

That is, monotone functions take chains in the domain to chains of images, distributive functions take finite meets to meets, and continuous functions take arbitrary meets to meets.[53] Every continuous function is distributive, and every distributive function is monotone, but not conversely.[54] 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.[55]

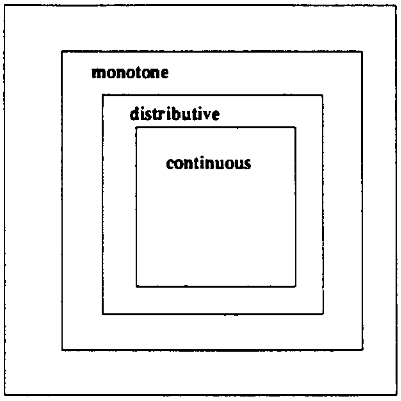

**Fig. 6-3.** Algebraic properties of function spaces

[53] If $L$ is a lattice, we can also define *anti-distributive* and *anti-continuous* functions, which take meets to joins

[54] However, if $F$ or $L$ is finite, then distributive implies continuous, and if $L$ is a chain, monotone implies distributive

[55] But see [15, 16], where anti-montone 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]. If $L$ has d.c.c.,[56] then for each $f \in F$ and $x \in L$, the sequence $\bigwedge_{i=0}^{k} f^i(x)$ stabilizes at some iterate $n(x)$, but the set of $n(x)$ need not be bounded, in which case the sequence $\bigwedge_{i=0}^{k} f^i$ will not stabilize. Thus $F$ need not have d.c.c.

Conversely, even if $L$ does not have d.c.c., $F$ may be sufficiently restricted that chains of iterates behave nicely. For $f \in F$, define

$$f^{[k]} = \bigwedge_{i=0}^{k-1} f^i.$$

We say that $F$ is *bounded* if for any $f \in F$ the chain $\{f^{[i]}\}$ 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 $r \geqq k$

$$f^r(x) \geqq f^{[k]}(x) \wedge f^k(y).$$

$k$-boundedness is defined for functions in general; for monotone functions it is equivalent to

$$f^k \geqq f^{[k]};$$

$k$-boundedness implies $k$-semiboundedness which in turn implies $(k+1)$-boundedness [86, 88]. Call $F$ *uniformly bounded* if it is $k$-bounded for some $k$. Semiboundedness (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 chains $\{f^{[i]}\}$ is bounded by the *effective height* of $L = \max \{\text{height } L_i\}$ [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 height of $L_1 = r =$ 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

$$f \circ f \geqq f \wedge \iota;$$

fastness is clearly related to *idempotence*, $f \circ f = f$, or *weak idempotence*, $f \circ f \geqq f$. 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].

[56] Or even if $L$ has finite height but does not have a $\perp$

$f$ is (Kam-Ullman) *rapid* [50] if it is 1-semibounded; for monotone functions, this is equivalent to

$$\forall x,\ y \in L\colon f(y) \geqq y \wedge x \wedge f(x).$$

Kam and Ullman's original definition required

$$\forall f,\ g \in F \forall x \in L\colon fg(\underline{0}) \geqq g(\underline{0}) \wedge f^{[1]}(x);$$

the two definitions are equivalent for complete lattices. If $L$ has a $\underline{1}$ and $F$ is monotone, rapidity is equivalent to

$$\forall y \in L\colon f(y) \geqq y \wedge f(\underline{1}).$$

$k$-boundedness implies that all contributions to the MFP occur prior to the $k$-th iterate; $k$-semiboundedness that the contribution of that $k$-th iterate is constant. Thus fastness 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 $\{f^{[k]}\}$ is infinite, there may exist a computable meet, or a good 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 closures:* For $f \in F$, $f \wedge \imath$ is monotone, decreasing, and less than $f$; we can use meets of its iterates to bound meets of iterates of $f$. Define $f^{\{k\}} = (f \wedge \imath)^{k-1}$. If $F$ is monotone, $f^{\{i\}} \leqq f^{[i]}$, and if $F$ is distributive they are equal. Graham and Wegman [39] define the *fastness closure* $\bar{f}_{GW}$ of $f$ to be the last term $f^{\{r\}}$ in the decreasing chain $\{f^{\{i\}}\}$. $\bar{f}_{GW}$ is idempotent and therefore fast. If $F$ is uniformly bounded by $k$, then $\bar{f}_{GW}$ exists (with no greater $k$), but $\bar{f}_{GW}$ may exist even if $\{f^{[k]}\}$ is infinite.

A different definition, essentially the following, is given by Rosen [72]. $\bar{f}_R$ is the first term $f^{\{r\}}$ less than all terms in the sequence $\{f^{[k]}\}$; that is, $f^k \geqq f^{\{r\}}$ for all $k \geqq r$. Again, for monotone $F$, it is sufficient that $f^r \geqq f^{\{r\}}$. $\bar{f}_R$ is not guaran-

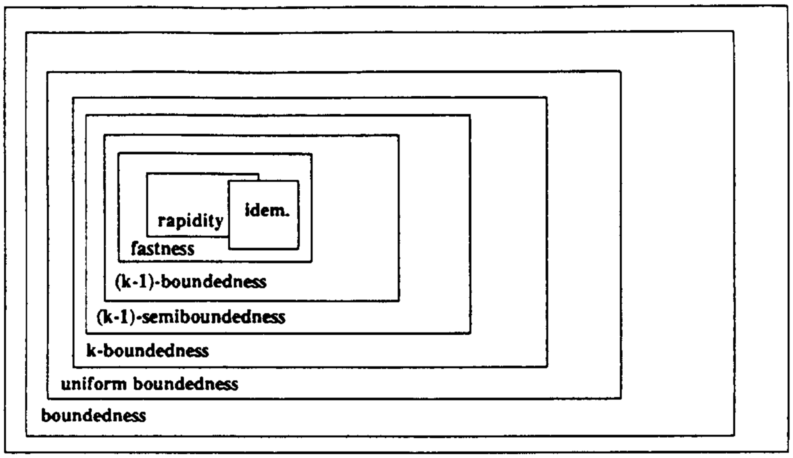

**Fig. 6-4.** Finiteness properties for function spaces

teed to be fast, but exists whenever $\bar{f}_{GW}$ does, and the last "exponent" $r$ for $\bar{f}_R$ is no greater than that for $\bar{f}_{GW}$. When $f$ is distributive, they are equal. Henceforward, we will use $\bar{f}$ for $\bar{f}_{GW}$, and identify Rosen's fastness closure by $\bar{f}_R$. Many of the results nonetheless hold for both fastness closures.

If $f$ is monotone, then $f \wedge \iota$, 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 $(f \circ g) \wedge \iota = (f \wedge \iota) \circ (g \wedge \iota)$, it will not in general be true that $(f \wedge g) \wedge \iota = (f \wedge \iota) \wedge (g \wedge \iota)$, nor that $(f \circ g)^k = f^k \circ g^k$, the fastness closure of $f \circ g$ will not necessarily be $\bar{f} \circ \bar{g}$, nor will the closure of $f \wedge g$ be $\bar{f} \wedge \bar{g}$. 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 Aliasing for which in general there are no fastness closures.

*Infinite limits:* Let $f^* = \bigwedge_{i=0}^{\infty} \{f^i\} = \bigwedge_{i=0}^{\infty} \{f^{[i]}\}$ (by analogy with regular expressions), if it 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, $\bar{f}_E = \bigwedge_{i=0}^{\infty} \{f^{\{i\}}\}$. Since the $f^{[i]}$ and the $f^{\{i\}}$ form descending chains, $f^*$ and $\bar{f}_E$ are really *limits* of sequences of iterates when they exist. If $f^*$ exists for all $f \in F$, then so will $\bar{f}_E = (f \wedge \iota)^*$, but the converse need not hold. If $f^* = f^{[r]}$, then $f^*$ can be evaluated with $O(r)$ meets and compositions; if $\bar{f}_E = \bar{f} = f^{\{r\}}$, then $\bar{f}$ can be computed with $O(\log_2 r)$ operations.

| | |
|---|---|
| $f^k$ | The $k$-th iterate of $f$ |
| $f^{[k]}$ | The meet of the first $k-1$ iterates of $f$, $\bigwedge_0^{k-1} f^i$ |
| $f^{\{k\}}$ | The $(k-1)$-st iterate of $f \wedge \iota$, $(f \wedge \iota)^{k-1}$ |
| $\bar{f} = \bar{f}_{GW}$ | (The fastness closure) the finite limit of the descending chain $f^{\{k\}}$ |
| $\bar{f}_E$ | (The extended fastness closure) The limit, not necessarily finite, of $f^{\{k\}}$ |
| $\bar{f}_R$ | (The Rosen fastness closure) The first term in $\{f^{\{k\}}\}$ which is a lower bound for $\{f^{[k]}\}$. |
| $f^*$ | (The reflexive transitive closure) The limit, not necessarily finite, of $f^{[k]}$ |
| $f^{(@)}$ | (A pseudotransitive closure) An approximation to $f^*$ with certain nice properties |
| $t_{(@)}$ | The Rosen rapidity parameter |
| $k$-bounded | Iterates of $f$ from $k$ on don't change $f^{[r]}$ |
| $k$-semibounded | Intuitively, $f^{[k+1]} \geq f^{[k]} \wedge$ constant |
| Bounded | Each $f \in F$ is bounded for some $k$ |
| Uniformly bounded | Each $f \in F$ is bounded for the same $k$ |

**Fig. 6-5.** Summary of notation for Sect. 6.2

*Approximations:* A lower approximation $f^{@}$ to $f^*$ is called a *pseudotransitive closure* [72, 73, 88, 89] of $f$ if

(*i*) $f^{@}(x) \leqq f^i(x)\ \forall x \in L,\ i \geqq 0$, and

(*ii*) if $x \in L$ is such that $x \leqq f(x)$ then $x \leqq f^{@}(x)$.

$\bar{f}_E$ 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 $M$ (and $G$) 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 $t_{@}$ if there is a pseudotransitive closure of $f$ computable with at most $t_{@}$ elementary operations (usually meets and compositions) [72].[57] Again, $F$ and D have transitive ($f^*$) or pseudotransitive ($f^{@}$) closure, if the property holds for all $f \in F$. $F$ and D are Rosen rapid if each $f$ in $F$ is Rosen rapid for some single parameter $t_{@}$. Data flow problems with fastness closures give an example of a Rosen rapid framework. If the framework is $k$-bounded, then $\bar{f} = f^{\{k\}}$ and $t_{@} = \mathrm{O}(\log_2 k)$; if $f$ is distributive, then $\bar{f}$ agrees with $f^*$. If $f$ is fast, then $\bar{f} = f \wedge \iota$ and $t_{@} = 1$.

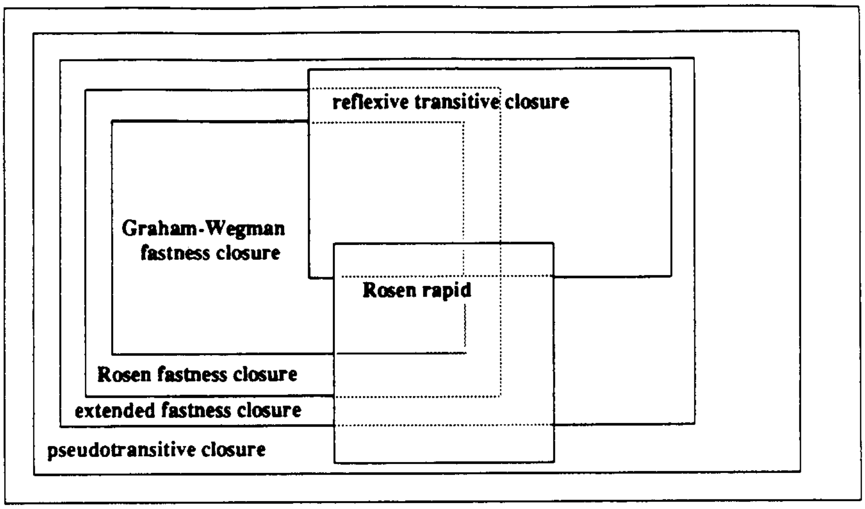
reflexive transitive closure
Graham-Wegman fastness closure
Rosen rapid
Rosen fastness closure
extended fastness closure
pseudotransitive closure

**Fig. 6-6.** Closure properties for function spaces

Cousot and Cousot [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 *widened* flow functions. A *widening operator* $\nabla$, a binary operator like Rosen's spiral, @, is guaranteed to replace arbitrary

[57] Rosen's definitions of @ and rapidity differ slightly from, and are slightly more general than, those given here. Rosen's *spiral* operator @ is binary, where $f@g$ approximates the effect of zero or more cycles in the interval with function $g$, followed by $f$

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 intraprocedural 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-exit (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 $\rho$), 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 intraprocedural 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 $v$ the set of all transitive closure edges ending at $v$. 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 $v$ the set of edges resulting from paths through $v$.

The *Faint Variable Problem* [45] is a set-valued problem; a definition "$d$=(variable $x$, program point $p$)" 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 $k$-bounded, where $k$ 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 (Kleene-*). The *Flow Cover* or *Path Expressions Problem* for a given graph $G$ 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.[58] In a given instance it is $k$-bounded, where $k$ 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 Banning [7]) 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.

[58] 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!

*Conditional Constant Propagation* [94] and Incrementally Available Expressions [38, 88] can be posed as Qualified Path Expression Problems, but are not easily expressed as problems on the obvious framework [43, 88]. Incrementally Available Expressions is $n$-bounded, where $n$ is the parameter in the definition.

Data flow problems not bounded a priori by a fixed $k$ can have instances with arbitrarily bad complexity. Myers [63] shows that May-Be-Modified is NP-complete, and *Interprocedural Available Expressions* is co-NP-complete, in the presence of aliasing, even without pointers of recursion. Weihl [95] shows that solution of *Procedure Parameters*, that is, the actual procedures bound to procedure variables by a sequence of calls, is P-space hard, even without aliasing, procedure nesting, or "significant control flow".

## 7. Complexity results

Algorithms for data flow problems may find an acceptable assignment $\mathbf{X}$ 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 $G$ and/or $\eta$ to find the assignment, or by computing intermediate values in $L$ for the vertices of $G$ and combining them to find $\mathbf{X}$. 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 problems, 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 $L$ has the d.c.c., or $F$ is bounded, value-based algorithms often find a suitable $\mathbf{X}$ in finite time, while function-based algorithms may need stronger local finiteness properties on $L$ or $F$. Further, complexity arguments will require such properties.

Kildall [53] showed that if $F$ is distributive and $L$ has the d.c.c. (or if each $f \in F$ is bounded), and the set of equations has form of 4.1, then iteration initialized at a suitable point ($\underline{0}$ at points without predecessors and $\underline{1}$ elsewhere) converges in finite time to the MFP solution, which agrees with the MOP solution. In a semilattice without a $\underline{1}$, 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 $F$, iteration using DFST evaluation requires at most $d+2$ passes if $L$ contains a $\underline{1}$, and at most $d+3$ passes otherwise (the extra pass is used to construct an initial overestimate to the solution), where $d$ is the maximum number of DFST back edges on an acyclic path in $G$ (essentially the nesting depth of $G$).

If $L$ has infinite chains, then distributivity is insufficient; Tarjan [88] shows that continuity of $F$, or an essentially equivalent condition on chains of $\{f^{[i]}\}$, is required; if in addition $F$ is $k$-bounded, the MOP solution can be computed in time $O(\log_2 k)$.

Tarjan [78, 89] gives what essentially is an elimination algorithm for data flow on a reducible flow graph with $n$ nodes and $m$ edges, using a stratified path-compressed tree data structure [87] whose complexity is $O(m\alpha(m, n))$

(where $\alpha$ is the functional inverse of Ackermann's function), given that a closure operator $f^*$ (possibly the $f^@$ of [88]) is available at cost O(1).

Wegman and Zadeck [94] give a set of algorithms for Constant Propagation, with solutions of increasing precision, using ever more precise estimates of compile-time path reachability, and constant values found by previous algorithms. Most data flow problems, especially the more complex problems, have similar families of algorithms of increasing precision and cost.

## *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 reducible graph, which is then solved by the previous algorithm. The complexity is $O(m\alpha(m,n)+b)$, 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 single-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 will handle arbitrary flow graphs, although efficient schemes for solution may no longer be applicable. One could also try to 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 $f^{@}$ may improve the complexity of finding the MFP.

Frequently, $f^{@}$ 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 $f^{@}$ will be less than or equal to the original $f^*$ [91]. Iteration (possibly using closures) will reach $f^{@}$, which will still give *an acceptable solution for the original problem.*[60] Use of $f^{@}$ 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. $f^{@}$ can often be considered the MFP in an approximation

[61] Note that such an algorithm would determine sharp loop invariants!

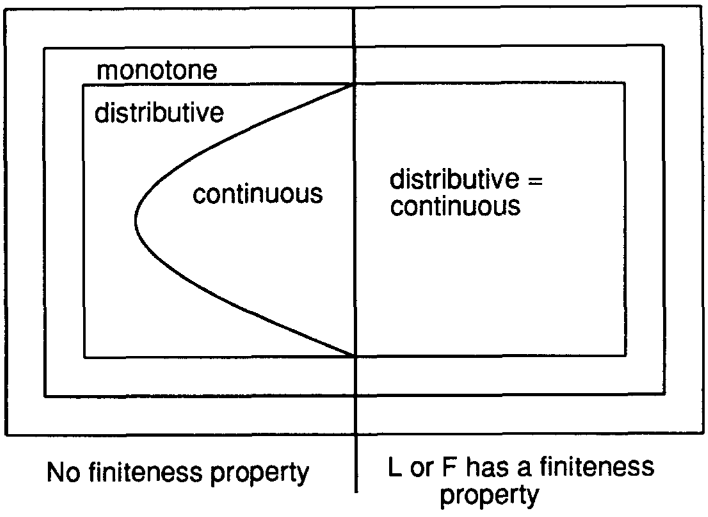

**Fig. 8-1.** Algebraic properties of function spaces 2

*Finiteness and closure properties:* Since $f^*$ and $\bar{f}_E$ are valid pseudotransitive closures, $f^@$ will exist for a given $f \in F$ if either of these does. When closures $f^*$ exist for all $f \in F$, then so will the extended fastness closures $\bar{f}_E$, since these are closures of $f \wedge \iota$. By definition, Rosen rapidity implies that $f^@$ exists.

Boundedness implies $f^*$ exists, and the existence of fastness closures. Existence of fastness closure $\bar{f}$ implies $\bar{f}_E$ exists. Graham-Wegman fastness closure implies Rosen fastness closure, which in turn implies Rosen rapidity for the given $f$.

Strictly finite height in $L$ will imply uniform boundedness, since chains of function values cannot be longer than arbitrary chains in $L$; in fact, height $k$ in $L$ implies $k$-boundedness in $F$. As discussed above, $(k-1)$-boundedness implies $(k-1)$-semiboundedness implies $k$-boundedness implies boundedness; fastness is 2-boundedness, rapidity is 1-semiboundedness; and idempotence implies fastness.

Thus the only finiteness and closure properties which can be independent of the existence of a pseudotransitive closure operator are d.c.c. or finite height in $L$. However, if value-based iteration is used to solve a framework in which the lattice has d.c.c. (and the flow graph is finite), it reaches the MFP in finite time, even though a pseudotransitive closure operator may not exist for all of $F$. Let simultaneous iteration be applied to a safe initial guess $X_0$. By induction, given a vertex $v$, the sequence of values $X_n(v)$ seen at a given vertex $v$ will be non-increasing. Further, until the MFP has been reached, some component of $X$ will change in each iteration. But since the value at $v$ can decrease only a finite number of times.

In Sect. 8.1, we give examples to show that there is no universal relationship between algebraic and finiteness/closure properties. In fact, although non-monotone functions are rarely of interest, the strongest of the finiteness properties, rapidity, is no guarantee of the weakest of algebraic properties, monotonicity. We show that given any pair of properties (in $\{$rapid, fast, neither$\}$ $\times$ $\{$distributive, monotone, neither$\}$, there is a small data flow framework with exactly that pair of properties. Our examples are frameworks where $L$ is a lattice with a

$\underline{1}$, and $F$ contains functions $\bar{0}$, $\bar{1}$, and the identity $\iota$, 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 8-5 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 $\underline{0}$ and $\underline{1}$ 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 $F_1=\{\iota, \bar{0}, \bar{1}, f(0, a, 1)=(1, a, 1)\}$.
Then $F_1$ is rapid but $f$ is not monotone.

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

Let $L_2=\{0, a, b, 1\}$ with $a \leqq b$, and

$$F_2=\{\iota, \bar{0}, \bar{1}, f(0, a, b, 1)=(0, b, a, 1),$$
$$g(0, a, b, 1)=(0, a, a, 1), h(0, a, b, 1)=(0, b, b, 1)\}.$$

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 $L_3=\{0, a, b, 1\}$, with $a$ and $b$ in comparable, and

$$F_3=\{\iota, \bar{0}, \bar{1}, f(0, a, b, 1)=(0, b, b, b),$$
$$g(0, a, b, 1)=(0, a, a, a)\}.$$

Then $F_3$ is rapid and monotone, but $f(a \wedge b) \neq f(a) \wedge f(b)$, so $F_3$ is not distributive.

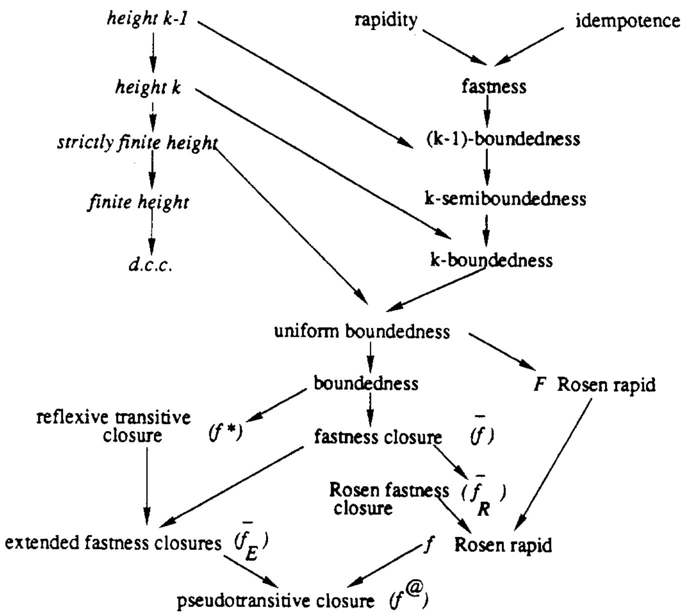

**Fig. 8-2.** Finiteness and closure properties lattices and function spaces implication digraph

Except for Rosen rapidity and uniform boundedness, properties of $F$ are stated in terms of properties of $f \in F$. In general, $f$ has a given property if $F$ does, and $F$ has the property if every $f \in F$ does. However, for Rosen rapidity, the constant $t_@$ may not be global and uniform boundedness is a property only of $F$.

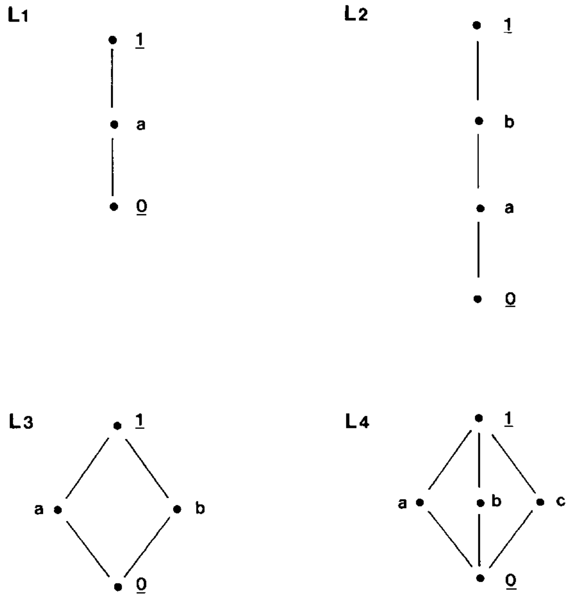

**Fig. 8-3.** Lattices for Sect. 8.1

**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

$$F_4 = \{\iota, \bar{0}, \bar{1}, f(0, a, b, c, 1) = (0, b, c, a, 1),$$
$$f^2 = f \circ f, \text{ and } g(0, a, b, c, 1) = (0, 0, 0, 0, 1)\}.$$

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 $L = \prod_{i=1}^{\infty} L_i$, $L$ will not have d.c.c. even if the factors are as simple as $L_i = \{0, 1\}$ for all $i$; but $F$ = 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 $F'$ = (the functions $\bar{1}$ and $\iota$ together with) the set of functions which take any point with an infinite number of components 0 to $\underline{0}$, then $F'$ will be distributive but not continuous.
c. Similarly, if $L' \subseteq L$ consists of elements of $L$ which are almost everywhere constant (only a finite number of 0 components, or only a finite number of 1's), then $L'$ will not be closed, so that the set of all distributive functions on $L'$ will not be continuous.
d. Letting $L_i$ 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

$$F_i = \{\iota, \bar{0}, \bar{1}, f(0, a, b, 1) = (0, b, a, 1),$$
$$g(0, a, b, 1) = (0, 0, 0, 1)\},$$

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.[62]

## 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 $k \geqq 1$, let $L_1 = \{0, x_1, x_2, \ldots, x_k, 1\}$, with the linear ordering given by $x_i \leqq x_{i+1}$, and let $F_1$ be the space of monotone functions on $L_1$.

[62] Lattices in which meet does not distributive over join (and vice-versa). $K$ can have as few as five elements [10]

Note $f(0, x_1, x_i, 1) = (0, 0, x_{i-1}, x_k) \in F_1$. $F_1$ is distributive (since $L_1$ is a linear order) and $(k+2)$-bounded since any (decreasing) chain $\{f^{[i]}\}$ of length $k+2$ reaches 0 or a fixed point, but $f^{[k+1]}(1) = x_1 \neq 0 = f^{k+1}(1)$.

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

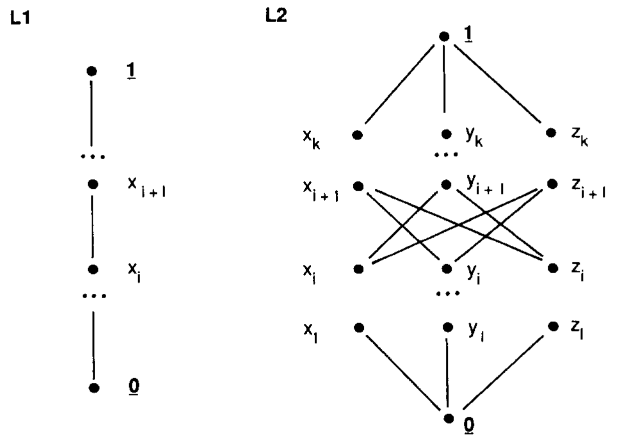

**Fig. 8-4.** Lattices for Sect. 8.2

Given $k \geqq 1$, let $L_2 = \{0, x_i, y_i, z_i, 1\}_{i=1}^{k}$, with $x_{i-1} = y_i \wedge z_i$, 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, $g(0, x_i, y_i, z_i, x_k, y_k, z_k, 1) = (0, x_{i+1}, y_{i+1}, z_{i+1}, 1, 1, 1, 1)$ is in $F_2$, but is not distributive, since $g(x_k \wedge y_k) = g(z_{k-1}) = z_k \neq 1 = g(x_k) \wedge g(y_k)$. (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 $L_3 = \{0, x_1, \ldots, x_k, 1\}$, again with the linear order, and let $F_3$ be generated by $\{\iota, \bar{0}, \bar{1}, g(0, x_1, x_i, 1) = (1, 0, x_{i-1}, x_k)\}$. $F_3$ is not monotone. For each $f \in F_3$ and $x \in L_3$, the set $\{f^i(x)\}_{i=0}^{r}$ is invariant for $r \geqq k+1$, and thus $F_3$ is $(k+2)$-bounded. However, $g^{k+1}(1) = 0 \neq x_1 = g^{[k+1]}(1)$, so $g$ 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 $2k$-bounded, since every iterate of $f$ will be larger than $(f \wedge \iota)(f^2)^{[k]}$. For instance, the set-valued function (for $X^c$ = the complement of $X$) $f(X) = X^c \cap B \cup C$ 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 $RT_2$ follows:

$$\begin{aligned} RT_2(v) &= \Lambda & , v = \rho \\ &= \Lambda \cup \Big( \bigcup_{e=(u,v)} M(e)\, RT_2(u) \Big) & , \text{otherwise} \end{aligned}$$

where

$$M(e)\,X = e \cup \{(w,u,v) : (w,u) \in X\} \cup \{(x,y,v),(x,u,v),(y,u,v) : (x,y,u) \in X\}.$$

Note that $RT_2$ is a set, and not a multiset. We show $RT_2$ is distributive, and is 3-bounded, but not 2-bounded.

For simplicity, consider a flow graph $G$ with a unique cycle $C$ through a vertex $v$. $(v,v,v) \in RT_2$, and can only be a subsequence of a path containing $C|C$. 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 $(w,u,v) \in RT_2(v)$, there must be some simple path (or simple cycle) from $w$ to $u$, and likewise from $u$ and $v$. Thus every element of $RT_2(v)$ is a subsequence of some path in $G$ with no more than two cycles. Thus $RT_2$ is 3-bounded. Likewise, $RT_n$ is $n+1$-bounded, but not $n$-bounded.

Table 8-1 gives an overview of problems in our property classification.

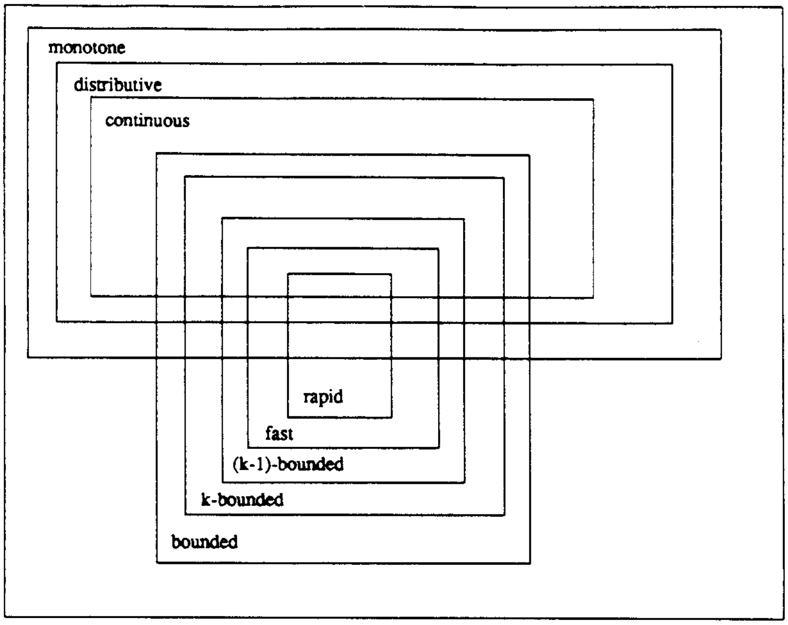

**Fig. 8-5.** Interaction of algebraic and local/finiteness/closure properties

| | rapid | fast | k-bounded (k fixed) | unif. bounded (k varies) | unbounded |
|---|---|---|---|---|---|
| non-monotone | **8.1** | **8.2** | **8.8**<br>Complements Example | *All Set Maps (if L has d.c.c.)* | *All Set Maps (L without d.c.c.)* |
| monotone | **8.3** | *Constant Propagation* | **8.7**<br>**8.5e**<br>*Type Inference (k)* | *Conditional Constant Propagation*<br>*PAvail*<br>*Must_Define* | *Loop Invariants*<br>*Type Inference* |
| distributive | **8.5b** | **8.5d** | *IAvail (k)*<br>*Aliasing w/ Pointers (k)* | *Faint Vars* | **8.5c**<br>*Aliasing w/ Pointers* |
| continuous | *Classical Problems*<br>*Reach* | **8.4**<br>*RT (1)*<br>**8.5a** | **8.6**<br>*RT (k)* | *Interprocedural Bound Set Aliasing May_Pres. Must_Mod.* | *Path Expressions* |

References in **BOLD type** are to examples in Sects. 8.1 and 8.2;
those in *italics* are to problems mentioned in Sect. 6.4;
Where a problem can be parametrized, (k) signifes the parameter; where none is given, the parameter is infinite.

**Table 8-1.** Overview of problem classification

## 9. Summary

Semilattice frameworks are a general setting for the formulation and solution of data flow problems. The framework is a 4-tuple: a flow graph, a semilattice, a class of functions, and a choice of particular defining equations (i.e., the assignment map $M$). We have seen how properties of each of the tuple elements may affect the solution to the data flow problem defined in terms of its existence, accuracy and cost. These solution factors may also be affected by the algorithm chosen and/or by the approximation method/framework/closure selected.

To understand data flow analysis solution procedures, it is necessary to understand clearly the formulation of these frameworks and the definition of their properties. We have surveyed frameworks, solution methods and properties, to provide a clear general overview of their interrelation. By comparing these using a single underlying model of a data flow problem, we can understand the necessary interrelations of certain semilattice and function space properties and distinguish between those assuring convergence and those guaranteeing some complexity criteria. We have demonstrated through examples that there 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

1. Abramsky, S., Hankin, C. (eds.): Abstract interpretation of declarative languages. Chichester: Ellis Horwood 1987
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Reading, MA: Addison-Wesley 1986
3. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Commun. ACM **19**, 137–147 (1977)
4. Allen, R., Kennedy, K.: Automatic translation of FORTRAN programs to vector form. ACM Trans. Program. Lang. Syst. **9**, 491–542 (1987)
5. Alpern, B., Wegman, M., Zadeck, F.K.: Detecting inequality of values in programs. In: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languates, pp. 1–11, 1988
6. Apt, K.R.: A static analysis of CSP programs. Proceedings of 1983 Conference on Logics of Programs. (Lect. Notes Comput. Sci., vol. 164, pp. 1–17) Berlin Heidelberg New York: Springer 1983
7. Banning, J.P.: An efficient way to find the side effects of procedure calls and the aliases of variables. In: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, pp. 29–41, 1979
8. Barth, J.M.: A practical interprocedural data flow analysis algorithm. Commun. ACM **21**, 724–737 (1978)
9. Beatty, J.: Register assignment algorithm for generation of highly optimized object code. IBM J. Res. Dev. **18**, 20–39 (1974)
10. Birkhoff, G.: Lattice theory. American Mathematical Society Colloquium Publications, Washington, DC, 1967
11. Burke, M.: An interval analysis approach toward exhaustive and incremental interprocedural data flow analysis. ACM Trans. Program. Lang. Syst. **12**, 341–395 (1990)
12. Burke, M.: Private communication (1989)
13. Burke, M., Cytron, R.: Interprocedural dependence analysis and parallelization. In: Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, pp. 162–175. SIGPLAN Notices **21**, No. 6 (1986)
14. Burke, M., Ryder, B.G.: A critical analysis of incremental iterative data flow analysis algorithms. IEEE Trans. Software Eng **16**, 723–728 (1990)
15. Cai, J.: Fixed point computation and transformational programming. PhD thesis, Department of Computer Science, Rutgers University, 1987. Rutgers Technical Report DCS-TR-217
16. Cai, J., Paige, R.: Program derivation by fixed point computation. Sci. Comput. Program. **11**, 197-261 (1989)
17. Callahan, D.: The program summary graph and flow sensitive interprocedural data flow analysis. In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 47–56, 1988
18. Callahan, D., Cooper, K., Kennedy, K., Torczon, L.: Interprocedural constant propagation. In: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, pp. 152–161, 1986
19. Callahan, D., Subhlok, J.: Static analysis of low-level synchronization. In: Conference Record of the 1988 Workshop on Parallel and Distributed Debugging, pp. 100–111, 1988
20. Carroll, M.: Data flow update via attribute and dominator updates. PhD thesis, Department of Computer Science, Rutgers University, 1988

21. Carroll, M., Ryder, B.: Incremental data flow update via attribute and dominator updates. In: Conference Record of the Fiftheenth Annual ACM Symposium on Principles of Programming Languages, pp. 274–284, 1988
22. Chow, A., Rudmik, A.: The design of a data flow analyzer. In: Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, pp. 106–113, 1982
23. Cooper, K.: Analyzing aliases of reference formal parameters. In: Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pp. 281–290, 1984
24. Cooper, K., Kennedy, K.: Efficient computation of flow insensitive interprocedural summary information. In: Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pp. 247–258. SIGPLAN Notices **19**, No. 6 (1984)
25. Cooper, K., Kennedy, K.: Interprocedural side-effect analysis in linear time. In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 57–66, 1988
26. Cooper, K., Kennedy, K.: Fast interprocedural alias analysis. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, pp. 49–59, 1989
27. Cousot, P.: Semantic foundations of program analysis. In: Program Flow Analysis: Theory and Applications, Chap. 10. Englewood Cliffs, NJ: Prentice Hall 1981
28. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages, pp. 238–252. SIGPLAN Notices **12**, No. 1 (1977)
29. Cousot, P., Cousot, R.: Automatic synthesis of optimal invariant assertions: Mathematical foundations. In: Proceedings ACM Symposium on Artificial Intelligence and Programming Languages, pp. 1–12. SIGPLAN Notices **12**, No. 8 (1977)
30. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, pp. 269–282. SIGPLAN Notices **14**, No. 1 (1979)
31. Coutant, D.S.: Retargetable high-level alias analysis. In: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, pp. 110–118. SIGPLAN Notices **21**, No. 1 (1986)
32. Cytron, R.: Do-across: Beyond vectorization for multiprocessors. In: Conference Record of the 1986 International Conference on Parallel Processing, pp. 836–844, 1986
33. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, pp. 25–35, 1989
34. Cytron, R., Lowry, A., Zadeck, F.K.: Code motion of control structures in high-level languages. In: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, pp. 70–85, 1986
35. Dijkstra, E.W., Gasteren, A. van: A simple fixpoint argument without the restriction to continuity. Acta Inf. **23**, 1–8 (1986)
36. Emrath, P.A., Padua, D.A.: Automatic detection of nondeterminacy in parallel programs. In: Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, pp. 89–99, 1988
37. Ferrante, J., Ottenstein, K., Warren, J.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**, 319–349 (1987)
38. Fong, A.C.: Generalized common subexpressions in very high level languages. In: Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages, pp. 48–57, 1977
39. Graham, S., Wegman, M.: A fast and usually linear algorithm for global data flow analysis. J. ACM **23**, 172–202 (1976)
40. Harrold, M.J., Soffa, M.L.: Computation of interprocedural definition and use dependencies. In: Proceedings of the IEEE Comp. Soc. 1990 Int. Conference on Computer Languages, pp. 297–306, 1990
41. Hecht, M.S.: Flow analysis of computer programs. Amsterdam: Elsevier North-Holland 1977

42. Hecht, M.S., Ullman, J.D.: Analysis of a simple algorithm for global flow problems. In: Conference Record of the ACM Symposium on Principles of Programming Languages, pp. 207–217, 1973
43. Holley, I., Rosen, B.: Qualified data flow problems. IEEE Trans. Software Eng. SE-**7**, 60–78 (1981)
44. Hopcroft, J., Ullman, J.D.: Introduction to automata theory, languages, and computation. Reading, MA: Addison-Wesley 1979
45. Horwitz, S., Demers, A., Teitelbaum, T.: An efficient general iterative algorithm for data-flow analysis. Acta Inf. **24**, 679–694 (1987)
46. Horwitz, S., Prins, J., Reps, T.: Integrating non-interfering versions of programs. In: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, pp. 133–145, 1988
47. Horwitz, S., Prins, J., Reps, T.: On the adequacy of program dependence graphs representing programs. In: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, pp. 146–157, 1988
48. Horwitz, S., Pfeiffer, P., Reps, T.: Dependence analysis for pointer variables. In: Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, pp. 28–40, 1989
49. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 35–46, 1988
50. Kam, J.B., Ullman, J.D.: Global data flow analysis and iterative algorithms. J. ACM **23**, 158–171 (1976)
51. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Inf. **7**, 305–317 (1977)
52. Kennedy, K.: A survey of data flow analysis techniques. In: Program flow analysis: theory and applications, Chap. 1. Englewood Cliffs, NJ: Prentice Hall 1981
53. Kildall, G.: A unified approach to global program optimization. In: Conference Record of the ACM Symposium on Principles of Programming Languages, pp. 194–206, 1973
54. Knuth, D.E.: An empirical study of FORTRAN programs. Software Pract. Exper. **1**, 105–133 (1971)
55. Larus, J., Hilflinger, P.: Detecting conflicts in structure accesses. In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 21–34, 1988
56. Lindstrom, G.: Static evaluation of functional programs. In: Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, pp. 196–206. SIGPLAN Notices **21**, No. 7 (1986)
57. Manna, Z., Shamir, A.: The theoretical aspects of the optimal fixedpoint. SIAM J. Comput. **5**, 414–426 (1976)
58. Manna, Z., Shamir, A.: The optimal approach to recursive programs. Commun. ACM **20**, 824–831 (1977)
59. Marlowe, T.J.: Incremental iteration and data flow analysis. PhD thesis, Department of Computer Science, Rutgers University, 1989
60. Marlowe, T.J., Ryder, B.G.: An efficient hybrid algorithm for incremental data flow analysis. In: Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, pp. 184–196, 1990
61. McDowell, C.E.: A practical algorithm for static analysis of parallel programs. J. Parallel Distrib. Comput. **22**, 96–103 (1979)
62. Morel, E., Renvoise, C.: Global optimization by suppression of redundancies. Commun. ACM **22**, 96–103 (1979)
63. Myers, E.W.: A precise interprocedural data flow algorithm. In: Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, pp. 219–230, 1981
64. Neirynck, A., Panangaden, P., Demers, A.: Computation of aliases and support sets. In: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, pp. 274–283, 1987

65. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. In: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 177–184, 1984
66. Padua, D., Wolfe, M.J.: Advanced compiler optimizations for supercomputers. Commun. ACM **22**, 1184–1201 (1986)
67. Paull, M.C.: Algorithm design – A recursion transformation framework. New York, NY: Wiley Interscience 1988
68. Rapps, S., Weyuker, E.: Selecting software test data using data flow information. IEEE Trans. Software Eng. Se-**11**, 367–375 (1985)
69. Reif, J., Lewis, H.: Efficient symbolic analysis of programs. J. Comput. Syst. Sci. **11**, 280–313 (1986)
70. Reif, J., Tarjan, R.E.: Symbolic program analysis in almost-linear time. SIAM J. Comput. **9**, 81–93 (1981)
71. Reps, T., Teitelbaum, T., Demers, A.: Incremental context-dependent analysis for language-based editors. ACM Trans. Program. Lang. Syst. **5**, 449–477 (1983)
72. Rosen, B.: Monoids for rapid data flow analysis. SIAM J. Comput. **9**, 159–196 (1980)
73. Rosen, B.: A lubricant for data flow analysis. SIAM J. Comput. **11**, 493–511 (1982)
74. Rosen, B., Wegman, M., Zadeck, F.K.: Global value numbers and redundant computations. In: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, pp. 12–27, 1988
75. Ruggieri, C., Murtagh, T.: Lifetime analysis of dynamically allocated objects. In: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, pp. 285–293, 1988
76. Ryder, B.G., Marlowe, T.J., Paull, M.C.: Conditions for incremental iteration: examples and counterexamples. Sci. Comput. Program. **11**, 1–15 (1988)
77. Ryder, B.G., Pande, H.: The interprocedural structure of C programs: An empirical study. Laboratory for Computer Science Research Technical Report LCSR-TR-99, Rutgers University, 1988
78. Ryder, B.G., Paull, M.C.: Elimination algorithms for data flow analysis. ACM Comput. Surv. **18**, 277–316 (1986)
79. Sagiv, S., Edelstein, O., Francez, N., Rodeh, M.: Resolving circularity in attribute grammars with applications to data flow analysis. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, pp. 36–46, 1989
80. Schonberg, E.: On-the-fly detection of access anomalies. In: Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, pp. 285–297, 1989
81. Schwartz, J.T., Sharir, M.: A design for optimizations of the bitvectoring class. (Courant Comput. Sci. Rep. 17) New York, NY: New York University 1979
82. Shapiro, R.M., Saint, H.: The representation of algorithms. Technical Report CA-7002-1432. Massachusetts Computer Associates, 1970
83. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, Chap. 7. Englewood Cliffs, NJ: Prentice Hall 1981
84. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. **10**, 282–312 (1988)
85. Tarjan, R.E.: Testing flow graph reducibility. J. Comput. Syst. Sci. **9**, 355–365 (1974)
86. Tarjan, R.E.: Iterative Algorithms for global Flow Analysis. In: Algorithms and Complexity – New Directions and Recent Results, pp. 71–101. New York: Academic Press 1976
87. Tarjan, R.E.: Applications of path compression on balanced trees. J. ACM **26**, 690–715 (1979)
88. Tarjan, R.E.: A unified approach to path problems. J. ACM **28**, 576–593 (1981)
89. Tarjan, R.E.: Fast algorithms for solving path problems. J. ACM **28**, 594–614 (1981)
90. Tenenbaum, A.: Determination of types in very high level languages. PhD thesis, Courant Institute of Mathematical Sciences, New York University, 1974. Technical Report Number CCSR #3
91. Tenenbaum, A.: Compile time type determination in SETL. In: Proc. ACM 1974 Annual Conference, pp. 95–100, 1974
92. Ullman, J.D., Hecht, M.S.: Flow graph reducibility. SIAM J. Comput. **1**, 188–202 (1972)

93. Walz, J., Johnson, G.: Incremental evaluation for a general class of circular attribute grammars. In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 209–221. SIGPLAN Notices **23**, No. 7 (1988)
94. Wegman, M., Zadeck, F.K.: Constant propagation with conditional branches. In: Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, pp. 291–299, 1985
95. Weihl, W.: Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In: Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, pp. 83–94, 1980
96. Weiser, M.: Program slicing. IEEE Trans. Software Eng. SE-**10**, 352–357 (1984)
97. Zadeck, F.K.: Incremental data flow analysis in a structured program editor. PhD thesis, Rice University, 1983
98. Zadeck, F.K.: Incremental data flow analysis in a structured program editor. In: Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, pp. 132–143. SIGPLAN Notices **19**, No. 6 (1984)