Data Flow Analysis Frameworks for Concurrent Programs
1

Matthew B. Dwyer

Department of Computer Science
University of Massachusetts, Amherst

Abstract

Data flow analysis is a well studied family of static program analyses. A rich theoretical basis for data flow analysis has been developed. Central to this theory is the concept of *data flow framework*. These frameworks are a semantically well-founded formalism for specifying a data flow analysis. A variety of solution algorithms for problems specified as data flow frameworks have been developed.

The bulk of the research on data flow analysis has been done in the context of analysis of sequential programs. Unfortunately, a number of important assumptions of this work are violated when we apply data flow analysis results to concurrent programs. In this paper, we extend data flow frameworks for sequential program analysis to accommodate concurrent programs. We present solution algorithms for these extended frameworks and reason about their convergence and complexity.

[1]*This work was supported by the Advanced Research Projects Agency under Grant F30602-94-C-0137.

1 Introduction

Data flow analysis is a process for uncovering facts about executable program behavior without actually running the program. With applications to compiler optimization, program testing, validation and verification, data flow analysis is an important technique for a variety of software development activities. The constant desire for faster programs has spurred tremendous theoretical and practical advances in program analysis and optimization techniques. The bulk of this work has been for sequential programs.

The development of data flow frameworks by Kildall [Kil73] marked the beginning of a new era in compiler optimization. Setting data flow analysis on a sound formal foundation spurred a wealth of research into both generalizing, e.g., [KU77, CC77, GW76, Tar81] and specializing Kildall's result, e.g., [KU76, WZ91, Tar81]. This resulted in formalisms for specifying and classifying data flow analysis problems, and the development of algorithms for solving any problem in a given class. This meant that compiler developers no longer had to hand-craft a system of equations and a solver for an analysis problem. Instead, the formal machinery of data flow frameworks could be used to describe a data flow problem and to guide the selection of an appropriate ready-made solution algorithm for that problem, thereby, reducing the cost of building analyzers.

Compiler optimization research has adapted to the emergence of concurrent and parallel computing into the programming mainstream A large body of work has been developed for using data flow analyses to aid in parallelizing of sequential programs, e.g., [ZC91]. More recently, data flow analyses and optimizations for explicitly concurrent and parallel programs have been investigated, e.g., [SHW93, GS93]. Unfortunately, the theory of data flow frameworks has not been updated to accommodate the new program representations and analyses that concurrency and parallelism have brought.

In this the next section, we present a review of data flow frameworks. Section 3 follows with a discussion of the limitations of those frameworks. In Section 4 we present a generalization of Kam and Ullman's monotone data flow analysis framework [KU77]; this generalization, called a *complete-lattice monotone data flow analysis framework*, supports the formulation of data flow analyses over flow graphs for concurrent programs. We present solution algorithms for data flow analysis problems formulated as complete-lattice frameworks and prove results related to their correctness and complexity. To illustrate these ideas, in Section 5 we discuss the formulation of a well-studied data flow analysis problem as a complete-lattice framework. Section 6 mentions future directions and concludes.

2 Background

We begin with some definitions and terminology. In this section, we present results about data flow analyses formulated as *data flow frameworks*, using the terminology and definitions in Hecht [Hec77].

Every data flow analysis problem computes a different kind of *problem information*. This information captures facts about executable program behavior that the analysis is designed to gather; this information is inherently approximate. We can gather more(less) precise approximations with a commensurate increase(decrease) in the cost of analysis.

Definition 1 *A* ***meet semi-lattice*** *is a triple,* $L = (V, \sqsubseteq, \sqcap)$*, where* V *is a set of values,* $\sqsubseteq$ *is a partial-order defined over* V*, and* $\sqcap$ *is binary operation defined over* V*, such that the following* semi-lattice properties *hold:*

$$\begin{aligned}
x \sqcap y &\sqsubseteq y && (1)\\
x \sqcap y = x &\Leftrightarrow x \sqsubseteq y && (2)\\
x \sqcap x &= x && (3)\\
x \sqcap y &= y \sqcap x && (4)\\
(x \sqcap y) \sqcap z &= x \sqcap (y \sqcap z) && (5)\\
\exists \bot \in V &: \forall x \in V : \bot \sqsubseteq x && (6)
\end{aligned}$$

The lattice values encode information about the program that we are interested in collecting. The values in V are partially-ordered by the $\sqsubseteq$ operator; this describes which values contain the information of other values. The *meet* operator $\sqcap | V \times V \to V$ is used for combining values. Intuitively, the bottom value, $\bot$, is less than all other lattice values if we interpret $\sqsubseteq$ as $<$. We can optionally have a top value, $\top$, that is greater than all other values. A *chain* in the lattice is a collection of values that are all ordered with respect to one-another; it is a linear ordering of lattice elements. While V need not be finite, for our purposes we are only interested in lattices that have chains of finite length; note that we allow infinitely many such chains. The *height* of a lattice is the length of the longest chain. The standard example of a meet-semilattice is a powerset, $\mathcal{P}(S)$, where the values are subsets of a given set, $S_i \subseteq S$, the values are ordered by $\subseteq$ and meet is $\cap$. For this lattice $\bot = \emptyset$, $\top = S$, and its height is the number of elements in S.

Definition 2 *A* ***function space*** F *over a meet semi-lattice, is set of functions,* $f | V \to V$*, defined over the lattice values.*

The functions in F, called *transfer function*, are used to capture the local effects of parts of the programs computation. There are a number of *function space properties* that can be used to classify a function space with respect to a lattice:

$$\begin{aligned}
\exists f_{ident} \in F &: \forall v \in V : f_{ident}(v) = v && (7)\\
\forall v \in V &: \exists f_v \in F : \forall x \in V : f_v(x) = v && (8)\\
\forall f, g \in F &: f \circ g \in F && (9)\\
\forall f \in F &: \forall x, y \in V : f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y) && (10)\\
\forall f \in F &: \forall x, y \in V : f(x \sqcap y) = f(x) \sqcap f(y) && (11)
\end{aligned}$$

Definition 3 *A* ***monotone function space*** *satisfies properties 7-10.*

Definition 4 *A* ***distributive function space*** *satisfies properties 7-11.*

Combining a lattice of values and a collection of transfer functions yields a, somewhat abstract, description of a class of data flow analysis problems.

Definition 5 *A* ***data flow framework*** *is a pair,* $D = (L, F)$*, where* L *is a meet semi-lattice and* F *is a function space defined over that lattice.*

The set of possible program executions are modeled as a rooted directed graph.

Definition 6 *A* ***flow graph*** *is a directed graph,* $G = (N, E, n_0)$*, where* N *is set of nodes,* E *is an edge relation over nodes, and* $n_0 \in N$ *is a distinguished start node.*

For our purposes we restrict the set of edges such that $\forall m \in N : (m, n_0) \notin E$. For convenience, we define the set of predecessor, $Preds(n) = \{m : (m, n) \in E\}$, and successor nodes, $Succs(n) = \{m : (n, m) \in E\}$.

Associating the components of a data flow framework with components of a flow graph yields a more concrete description of a class of data flow analysis problem.

Definition 7 *An* ***instance****,* $I = (G, M)$ ***of a data flow framework****,* D*, is the binding of transfer functions to flow graph nodes. This binding is accomplished through the definition of a* ***function map****,* $M|N \rightarrow F$.

This mapping associates a function from F with each node; the transfer function captures the effects of that node with respect to the information being gathered and represented as lattice values. For convenience, we will write the function $M(n)$ as f_n.

The solution of a data flow analysis problem is an approximation to the problem information at each node in the flow graph. Data flow analysis problems can be formulated directly as equations that give the information for a node in terms of the information at its predecessors and functions that capture the local effects of a node:

$$X[n] \quad = \quad f_n(X[n_{pred_1}], \ldots, X[n_{pred_k}])$$

Where $X[n]$ is the problem information at node n. A data flow framework is a formalism for describing a class of data flow analysis problems. It provides the types for the $X[.]$ variables and the functions f. An instance of a framework provides information that tells us which variables and functions to equate; thus, providing a system of related equations:

$$\begin{aligned} X[n_0] &= \bot \\ \forall n \in N - \{n_0\} : X[n] &= \sqcap_{i \in Preds(n)} f_i(X[i]) \end{aligned}$$

A data flow analysis problem is *solved* by computing a fixed point of the equation system. This solution associates a lattice value, representing problem information, to each flow graph node. For problems formulated as an instance of a monotone data flow framework, the *maximum fixed point* (MFP) can be computed efficiently and is known to be unique. If the framework is distributive, then the MFP solution contains, for each node, the maximum amount of problem information that can be computed by considering every path from the start node to the given node; this is called the *meet over paths* (MOP) solution [2]. This solution is made possible due to the fact that $\sqcap$ distributes over $f \in F$. We can move the $\sqcap$ operations to the outside and apply it a single time to $f_p(\perp)$. Where for a path, $p = n_0, n_1, \ldots, n_k$, the composition of transfer function of the nodes on the path is $f_p(.) = f_{n_k}(\ldots(f_{n_1}(f_{n_0}(.)))\ldots)$. We define the set of paths leading from n_0 to node n as $Paths(n) = \{P | P = n_0, \ldots, n_k = n\}$ where, for integer i such that $1 < i < k$ we have $(n_{i-1}, n_i) \in E$. Given this we can write the MOP solution as:

$$A[n] \quad = \quad \sqcap_{p \in Paths(n)} f_p(\perp)$$

3 Limitations of Semi-lattice Frameworks

Placing data flow analysis on a sound formal foundation has brought many advantages to developers of new analyses. The ability to characterize an analysis framework [MR90], as rapid, continuous, distributive, monotone, or k-bounded for example, allows one to apply existing theoretical results to reason about convergence, bounds on running time, and the precision of analyses. In addition, a wide variety of general and specialized algorithms have been developed for classes of frameworks. Making it easy for developers of new analyses to choose the algorithm that is most appropriate for their data flow framework formulation of a problem.

The application of data flow analysis has been primarily in compiler optimization. Historically, the main thrust of that work has been on producing small/fast executables for programs written in sequential programming languages. The algorithmic and theoretical developments related to data flow analysis have mirrored this work by focusing on sequential programs. There has been some recent work on applying data flow analysis to concurrent programs. Like the early work on sequential program analysis this newer work has, to a great extent, involved the development of ad-hoc solutions to important practical problems.

As with all static analyses, data flow analyses reason about a model of program execution, i.e., the flow graph. A flow graph model can represent varying amounts of program information; it need only be sure to represent all the information that is necessary to support safe or conservative analyses. Given the wide variety of possible flow graph models it is fruitful to examine some common assumptions about these models. We discuss two such assumptions that underly much of the data flow analysis work to date.

[2]The MOP solution is often referred to as the *best* possible static solution for a given analysis problem.

A Path Models a Program Execution

Flow graph representations are often constructed so that for each prefix of a program execution that ends at the statement corresponding to node n, there is a path in the graph from n_0 to n. Typically, flow graphs will contain some paths that do not correspond to feasible executions of the program; this is done in order to represent the large, potentially infinite, number of different program executions in a tractable form.

For sequential programs it is easy to construct a flow graph from the source text. For programs written in high-level programming languages such a flow graph is typically sparse; each node only has a few neighbors. For interacting concurrent programs flow graph is more complicated to construct and often has a higher degree of node-connectedness.

Many models of concurrency allow for communication between processes and support multiple senders and receivers over a given communication channel. A flow graph representation for a program using this model must pessimistically assume that all pairs of senders and receivers for a given channel can communicate; although analysis may be able to determine the impossibility of certain communications. Thus, the number of nodes in the flow graph will, in the worst-case, be quadratic in the number of program statements rather than linear.

In order for all feasible program executions of a concurrent program to be represented as a flow graph path, we must represent all possible interleavings of the independently executing statements in program processes. Even with exact information about the statements that can execute concurrently, the flow graph would still be very dense, i.e., $|E|$ will approach $|N|^2$. Unfortunately, the problem of computing that information is NP-complete [Tay83].

As a consequence of this second implication, researchers have developed flow graphs that do not explicitly represent statement interleavings; these include the SCG [CKS90], PFG [GS93], MIG [Due91], and sync hypergraph [MR93]. There are also flow graphs that can be viewed with or without explicit interleavings, such as the TFG [DC94]. Figure 1 depicts flow graphs for a sequential and a concurrent program. The sequential flow graph is standard. The concurrent flow graph can be viewed as a collection of sequential control flow graphs, one for each process in the program (enclosed in a dashed box), consisting of (circular) nodes. Additional (square) nodes represent synchronization or communication points. Note that we are not attempting to define an all inclusive model of flow graphs for concurrent programs, rather we are identifying a number of common structural and semantic modeling features that differentiate them from flow graphs for sequential programs. The specific flow graphs mentioned above are all quite similar to the concurrent flow graph as described here.

These concurrent flow graphs model a program execution as a collection of paths, one for each process sub-graph, where those paths intersect at process interaction points. For such graphs, the MOP is no longer the *best* possible solution; in fact, it can be quite imprecise. Consider a classic any-path forward-flow data flow analysis problem such as reaching definitions [ASU85]. A definition of variable v at node d is said to *reach* a use at node u if there exists a path from d to u on which d is not killed, i.e., v is not re-defined. For the concurrent flow graph in Figure 1, if def_1, def_2 and use all refer to the same variable, the MOP solution for this problem would say that def_1 reaches use. If the successor square node of def_1 is a synchronization node, however, then def_1 is killed by def_2 on all program executions leading

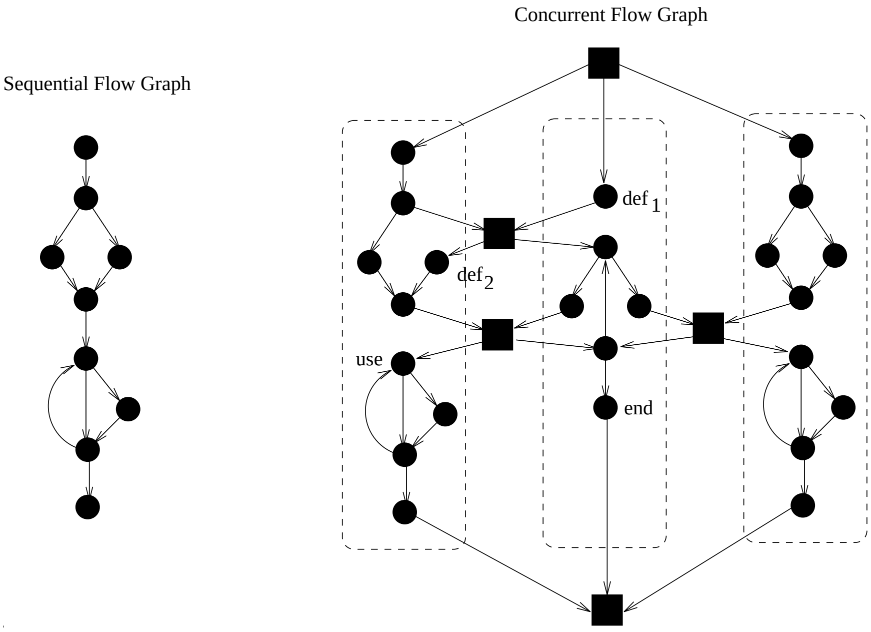

Figure 1: Typical Sequential and Concurrent Flow Graphs

to *use*.

Semantics of flow graph merge points

The semantics of merge points in such a sequential flow graph are uniform; a merge represents the confluence of two distinct program executions. In a concurrent flow graph nodes and edges model control flow information, synchronization information and communication information. Consequently, merge points can represent the confluence of parts of two distinct program executions or different parts of the same program execution.

For data flow analysis, flow graph merge points serve an important role; they are the points at which data gathered from separate regions of sequential program execution are combined. In meet semi-lattice data flow frameworks this combining operation is the meet operation. For the concurrent flow graphs described above, it appears that being limited to a single combining operation can lead to significant imprecision in analysis results. Srinivasan [SHW93] has also identified the necessity of multiple combining operations, represented as ϕ and ψ nodes, in defining an SSA form for explicitly parallel programs.

4 Complete-lattice Frameworks

In this section, we generalize semi-lattice frameworks to allow formulation of data flow problems over a complete-lattice. We extend a number of important results for monotone semi-lattice frameworks to complete-lattice frameworks. These new results address the limitations of semi-lattice frameworks and make the advantages of frameworks available to developers of analyses for concurrent programs.

Definition 8 *A* ***complete-lattice*** *is a triple,* $L = (V, \sqsubseteq, \sqcap, \sqcup)$*, where* V *is a set of values,* $\sqsubseteq$ *is a partial-order defined over* V*, and* $\sqcap$ *and* $\sqcup$ *are binary operation defined over* V*. These operations are such that for all subsets* $S \subseteq V$*,* $\sqcap_{s \in S}$ *and* $\sqcup_{s \in S}$ *are defined*[3]*. In addition to the semi-lattice properties (1 - 6), the components of* L *must satisfy the following* lattice properties*:*

$$\begin{aligned}
x \sqcup y &\sqsupseteq y && (12)\\
x \sqcup y = y &\Leftrightarrow x \sqsubseteq y && (13)\\
x \sqcup x &= x && (14)\\
x \sqcup y &= y \sqcup x && (15)\\
(x \sqcup y) \sqcup z &= x \sqcup (y \sqcup z) && (16)\\
x \sqcup (x \sqcap y) &= x && (17)\\
x \sqcap (x \sqcup y) &= x && (18)
\end{aligned}$$

A complete-lattice is essentially a meet semi-lattice with an additional idempotent, associative, commutative operation $\sqcup$ called *join*. Height and $\top$ are defined in the same way as for a semi-lattice. The standard example of a complete-lattice is a powerset, $\mathcal{P}(S)$, where the values are subsets of a given set, $S_i \subseteq S$, the values are ordered by $\subseteq$, meet is $\cap$, and join is $\cup$. For this lattice $\perp = \emptyset$, $\top = S$, and its height is the number of elements in S.

Observation 1 *If we have a finite collection of lattice values* $x_1, x_2, \ldots, x_k, x_1', x_2', \ldots, x_k' \in V$*, where for* i *such that* $1 \leq i \leq k$ *we have* $x_i \sqsubseteq x_i'$*, then* $\sqcap_{1 \leq i \leq k} x_i \sqsubseteq \sqcap_{1 \leq i \leq k} x_i'$ *and* $\sqcup_{1 \leq i \leq k} x_i \sqsubseteq \sqcup_{1 \leq i \leq k} x_i'$*.*

This follows from Lattice properties 1, 18 and the associativity, and commutativity of $\sqcap$ and $\sqcup$.

We define a function space, F, over the values, V, just as we do for a semi-lattice. Monotonicity and distributivity of the function space requires that function space properties (7-10) and (7-11), respectively, be met.

Definition 9 *A* ***complete-lattice data flow analysis framework****, is a pair,* $D = (L, F)$*, where* L *is a complete-lattice and* F *is a function space defined over that lattice.*

Definition 10 *An* ***instance****,* $I = (G, N_\sqcap, N_\sqcup, M)$*,* ***of a complete-lattice data flow analysis framework****,* D*, is a quadruple consisting of a flow graph* G*, subsets of the flow graph nodes* $N_\sqcap$ *and* $N_\sqcup$*, and a function map* $M|N \to F$*, where:*

[3] This is the standard lattice-theoretic definition of a complete lattice [DP90].

$$
\begin{aligned}
n_0 &\notin N_\sqcap \\
n_0 &\notin N_\sqcup \\
N_\sqcap \cap N_\sqcup &= \emptyset \\
N &= N_\sqcap \cup N_\sqcup \cup \{n_0\}
\end{aligned}
$$

The set $N_\sqcap$ contains the flow graph nodes for which $\sqcap$ is used to merge predecessor values. The set $N_\sqcup$ contains the flow graph nodes for which $\sqcup$ is used to merge predecessor values. Intuitively, these sets serve to map merge operations onto flow graph nodes just as M maps transfer functions onto flow graph nodes. A complete-lattice framework describes a class of data flow analysis problems; it induces the following system of simultaneous equations:

$$
\begin{aligned}
X[n_0] &= \bot \\
\forall n \in N_\sqcap : X[n] &= \sqcap_{p \in Preds(n)} f_p(X[p]) \\
\forall n \in N_\sqcup : X[n] &= \sqcup_{p \in Preds(n)} f_p(X[p])
\end{aligned}
$$

We note that if $N_\sqcup = \emptyset$ then this equation system is equivalent to one derived from the semi-lattice framework embedded in the complete-lattice framework. A data flow analysis problem is solved by computing a fixed point of the equation system, as described above.

Solving Complete-Lattice Data Flow Problems

We generalize existing results for monotone semi-lattice frameworks to monotone complete-lattice frameworks by mirroring Kam and Ullman's generalization of Kildall's distributive semi-lattice frameworks [KU77].

In the following, we assume the existence of $\top$. If it does not exist we can introduce a new element and adjust the lattice and functions accordingly, such that:

$$
\begin{aligned}
\forall f \in F : \forall x \in V &: \quad \top \sqcap x = x \sqcap \top = x \\
\top \sqcup x &= x \sqcup \top = \top \\
f(\top) &= \top
\end{aligned}
$$

Algorithm 1 (General Iterative Solver)
Input:

An instance $I = (G, N_\sqcap, N_\sqcup, M)$ of $D = (L, F)$, a monotone complete-lattice framework.

Output:

A lattice value for each node, $\forall n \in N : A[n] \in V$.

Initialization Step:

$$\forall n \in N \qquad A[n] = \begin{cases} \bot & \text{if } n = n_0 \\ \top & \text{otherwise} \end{cases}$$

Iteration Step:

Visit the nodes, other than n_0, in order $n_1, n_2, \ldots$. Nodes can be visited multiple times and the order is not fixed prior to running the algorithm. A *visit* of a node, n, is the evaluation of one of the assignments:

$$\begin{array}{ll} A[n] = \sqcap_{p \in Preds(n)} f_p(A[p]) & \text{if } n \in N_\sqcap \\ A[n] = \sqcup_{p \in Preds(n)} f_p(A[p]) & \text{if } n \in N_\sqcup \end{array}$$

The sequence of nodes visited must satisfy two conditions:

No premature termination If there exists a node $n \in N_\sqcap$ ($n \in N_\sqcup$) such that $A[n] \neq \sqcap_{p \in Preds(n)} f_p(A[p])$ ($A[n] \neq \sqcup_{p \in Preds(n)} f_p(A[p])$) after visiting node n_i in the sequence, then there is an integer $j > i$ such that $n_j = n$. Intuitively, if we reach a point in the sequence where some node's current value has not been updated by *newer* information at its predecessors then the sequence must continue until that update is performed.

Termination after stabilization If after visiting node n_i, $A[n] = \sqcap_{p \in Preds(n)} f_p(A[p])$ for all $n \in N_\sqcap$ and $A[n] = \sqcup_{p \in Preds(n)} f_p(A[p])$ for all $n \in N_\sqcup$, then the sequence will eventually halt. Intuitively, once we have reached a point where all the current node values are up to date it is safe to terminate the algorithm.

In the presentation that follows we will refer to the j^{th} step of Algorithm 1 to mean the state of the algorithm after visiting the first j nodes in the node sequence. The value at a node, n, on the j^{th} step will be denoted $A^{(j)}[n]$.

The constraints on the node visitation sequence express certain requirements of the algorithm with respect to termination; they do not, however, guarantee termination.

Lemma 1 (General Iterative Solver Terminates)
Given an instance instance, $I = (G, N_\sqcap, N_\sqcup, M)$, of a monotone complete-lattice data flow analysis, $D = (L, F)$, if we apply Algorithm 1 it will eventually terminate.

Proof: The proof is by induction on m, the number of steps performed by Algorithm 1. We show that for all nodes, n, in G, $A^{(m+1)}[n] \sqsubseteq A^{(m)}[n]$.

Base Step: $m = 0$

At this point the algorithm has completed its initialization phase and visited a single node, n_1. For the start node, n_0, we have $A^{(1)}[n_0] = \perp \sqsubseteq \perp = A^{(0)}[n_0]$. We have yet to visit any of the other nodes, $n \in N - \{n_0, n_1\}$, and those nodes remain at their initial values, so $A^{(1)}[n] = \top \sqsubseteq \top = A^{(0)}[n]$. We visit n_1 on this step, but since the value at step 0 at n_1 was $\top$ we do not need to bother evaluating $A^{(1)}[n_1]$; by definition all values are $\sqsubseteq \top$.

Inductive Step: At step $m+1$ we have $\forall n \in N : A^{(m)}[n] \sqsubseteq A^{(m-1)}[n]$

At this point in the algorithm we are at step $m+1$ and have just visited node n_{m+1}. We note that at each step in the algorithm the value of at most a single node can change; for step i it is the value at node n_i. Thus, for the values at node $n \in N - \{n_{m+1}\}$ will remain unchanged, so:

$$\forall n \in N - \{n_{m+1}\} : A^{(m+1)}[n] = A^{(m)}[m] \;\;\Rightarrow\;\; A^{(m+1)}[n] \sqsubseteq A^{(m)}[m]$$

Let l be the most recently preceding step at which node n_{m+1} was visited; if n_{m+1} has never been visited then $l = 0$. The sequence of node visits is $\ldots n_l, \ldots, n_m, n_{m+1} \ldots$, where $n_l = n_{m+1}$.

If for all i such that $l < i < m+1$, it is the case that $n_i \notin \mathit{Preds}(n_{m+1})$, then $A^{(m+1)}[n_{m+1}] = A^{(m)}[n_{m+1}] = A^{(l)}[n_{m+1}]$, since none of its predecessors have changed value. In this case the induction trivially holds.

If there exists an i such that $l < i < m+1$ and $n_i \in \mathit{Preds}(n_{m+1})$, then some number of predecessors may have changed value. We refer to the set of predecessors of n_{m+1} who have changed value since step l as $\mathit{Changed}$; we also define $\mathit{NotChanged} = \mathit{Preds}(n_{m+1}) - \mathit{Changed}$. There are two cases to consider in evaluating the new value at n_{m+1}:

case 1: $n_{m+1} \in N_\sqcap$

We write out the values at steps m and $m+1$ more explicitly to show their relationship.

$$\begin{aligned} A^{(m)}[n_{m+1}] &= A^{(l)}[n_{m+1}] \\ &= (\sqcap_{p \in \mathit{Changed}} f_p(A^{(l)}[p])) \sqcap (\sqcap_{p \in \mathit{NotChanged}} f_p(A^{(l)}[p])) \\ A^{(m+1)}[n_{m+1}] &= (\sqcap_{p \in \mathit{Changed}} f_p(A^{(m)}[p])) \sqcap (\sqcap_{p \in \mathit{NotChanged}} f_p(A^{(l)}[p])) \end{aligned}$$

Focusing on the right hand side of these two equations, we have:

$$\begin{aligned} \forall p \in \mathit{NotChanged} &: f_p(A^{(l)}[p])) \sqsubseteq f_p(A^{(l)}[p])) \\ \forall p \in \mathit{Changed} &: f_p(A^{(m)}[p])) \sqsubseteq f_p(A^{(l)}[p])) \end{aligned}$$

from the definition of $\sqsubseteq$ and the inductive hypothesis. By Observation 1 the iterated meets of the elements on the right hand sides satisfy the $\sqsubseteq$ relation. So, $A^{(m+1)}[n_{m+1}] \sqsubseteq A^{(m)}[n_{m+1}]$ and the induction holds.

case 2: $n_{m+1} \in N_{\sqcup}$

We write out the values at steps m and $m+1$ more explicitly to show their relationship.

$$\begin{aligned}
A^{(m)}[n_{m+1}] &= A^{(l)}[n_{m+1}] \\
&= (\sqcup_{p \in Changed} f_p(A^{(l)}[p])) \sqcup (\sqcup_{p \in NotChanged} f_p(A^{(l)}[p])) \\
A^{(m+1)}[n_{m+1}] &= (\sqcup_{p \in Changed} f_p(A^{(m)}[p])) \sqcup (\sqcup_{p \in NotChanged} f_p(A^{(l)}[p]))
\end{aligned}$$

Focusing on the right hand side of these two equations, we have:

$$\begin{aligned}
\forall p \in NotChanged &: f_p(A^{(l)}[p])) \sqsubseteq f_p(A^{(l)}[p])) \\
\forall p \in Changed &: f_p(A^{(m)}[p])) \sqsubseteq f_p(A^{(l)}[p]))
\end{aligned}$$

from the definition of $\sqsubseteq$ and the inductive hypothesis. By Observation 1 the iterated joins of the elements on the right hand sides satisfy the $\sqsubseteq$ relation. So, $A^{(m+1)}[n_{m+1}] \sqsubseteq A^{(m)}[n_{m+1}]$ and the induction holds.

We now turn our attention to the conditions for node visitation sequences. After the i^{th} step we have one of two cases, either the values have stabilized, in which case the sequence will terminate, or there is some value that has yet to be updated and the sequence continues. The premature termination condition guarantees that a node that needs updating will get updated. There are a finite number of nodes in G; for each node there are a bounded number of values that can be taken on in descending through the lattice, since the lattice has finite length chains and a $\perp$ element. Consequently, node updates can occur only a finite number of times. Therefore, the sequence is finite and the Algorithm 1 terminates.

□

We now show that the solution computed by this algorithm is, in fact, the maximum fix-point of the system of equations described earlier.

Theorem 1 (General Iterative Solver Computes MFP)
Given an instance, $I = (G, N_{\sqcap}, N_{\sqcup}, M)$, of a monotone complete-lattice data flow analysis framework, $D = (L, F)$, if we apply Algorithm 1 then the solution, the $A[n]$, is the maximum fix point solution of the following system of equations:

$$\begin{aligned}
X[n_0] &= \perp \\
\forall n \in N_{\sqcap} : X[n] &= \sqcap_{p \in Preds(n)} f_p(X[p]) \\
\forall n \in N_{\sqcup} : X[n] &= \sqcup_{p \in Preds(n)} f_p(X[p])
\end{aligned}$$

Proof: By the definition of the algorithm it is clear that once Algorithm 1 halts, the $A[n]$ are solutions to the above equations. We need to show that these $A[n]$ are greater or equal to any other solution to the equations, $B[n]$. We prove, by induction on m, the number of steps taken in Algorithm 1, that after the m^{th} step $\forall n \in N : B[n] \sqsubseteq A^{(m)}[n]$.

Base Step: $m = 0$

The only possible solution at node n_0 is $\perp$, so $B[n_0] = \perp$. At step 0 we have completed initialization so $A^{(0)}[n_0] = \perp$, and the induction holds. For the other nodes $n \in N - \{n_0\}$ the initial value is $\top$, so regardless of its value $B[n] \sqsubset \top = A^{(0)}[n]$.

Inductive Step: At step m we have $\forall n \in N : B[n] \sqsubseteq A^{(m-1)}[n]$

At step m we visit node n_m to compute a new value; all other nodes keep their same value, $\forall n \in N - \{n_m\} : A^{(m)}[n] = A^{(m-1)}[n]$. Using the inductive hypothesis we see that the inductive step holds for these nodes.

To compute the new value at node n_m there are two cases:

$n_m \in N_\sqcap$

By definition the solution we have $B[n_m]$ and from the proof of Lemma 1 we have:

$$\begin{aligned} B[n_m] &= \sqcap_{p \in Preds(n)} f_p(X[p]) \\ A^{(m)}[n_m] &= \sqcap_{p \in Preds(n)} f_p(A^{(m-1)}[p]) \end{aligned}$$

From the inductive hypothesis we know that

$$\forall p \in Preds(n_m) \quad : \quad B[p] \sqsubseteq A^{(m-1)}[p]$$

The monotonicity of the function space gives us that

$$\forall p \in Preds(n_m) \quad : \quad f_p B[p] \sqsubseteq f_p(A^{(m-1)}[p])$$

From Observation 1, since the elements of the iterated meets in the definitions of $B[n_m]$ and $A^{(m)}[n_m]$ are pair-wise ordered so are their iterated meets. Thus, $B[n_m] \sqsubseteq A^{(m)}[n_m]$ and the inductive step holds for n_m.

$n_m \in N_\sqcap$

By definition the solution we have $B[n_m]$ and from the proof of Lemma 1 we have:

$$\begin{aligned} B[n_m] &= \sqcup_{p \in Preds(n)} f_p(X[p]) \\ A^{(m)}[n_m] &= \sqcup_{p \in Preds(n)} f_p(A^{(m-1)}[p]) \end{aligned}$$

From the inductive hypothesis we know that

$$\forall p \in Preds(n_m) \quad : \quad B[p] \sqsubseteq A^{(m-1)}[p]$$

The monotonicity of function space gives us that

$$\forall p \in Preds(n_m) \quad : \quad f_p B[p] \sqsubseteq f_p(A^{(m-1)}[p])$$

From Observation 1, since the elements of the iterated joins in the definitions of $B[n_m]$ and $A^{(m)}[n_m]$ are pair-wise ordered so are their iterated joins. Thus, $B[n_m] \sqsubseteq A^{(m)}[n_m]$ and the inductive step holds for n_m.

The theorem follows from these results and the fact that Algorithm 1 is guaranteed to terminate by Lemma 1.

□

Algorithm 1 is specified as weakly as possible; more refined implementations are free to choose from a wide variety of data structures and node visitation orders. We describe one such implementation here. It is an adaptation of Hecht's iterative worklist algorithm [Hec77].

Algorithm 2 (Iterative Worklist Solver)

Input:

An instance $I = (G, N_\sqcap, N_\sqcup, M)$ of $D = (L, F)$, a monotone complete-lattice framework.

Output:

A lattice value for each node, $\forall n \in N : A[n] \in V$.

We use two auxiliary data structures in this algorithm: W which is an ordered sequence of nodes with at most $|N|$ elements and v a single lattice value.

Initialization:

$$\begin{array}{lcl} \forall n \in N & & A[n] = \begin{cases} \bot & \textit{if } n = n_0 \\ \top & \textit{otherwise} \end{cases} \\ W & = & \{m : (n_0, m) \in E\} \end{array}$$

Main Loop:

We evaluate the following statements repeatedly until $W = \emptyset$: [4]

at this point $W = n, n_1, n_2 \ldots n_k$

(1) $W = n_1, n_2, \ldots n_k$

(2) $v = A[n]$

(3) if $n \in N_\sqcap$ then

[4]We describe the worklist as initially containing $k+1$ elements; this is a notational convenience and if $k = 0$ then $W = n$.

(4) $\qquad A[n] = \sqcap_{p \in Preds(n)} f_p(A[p])$

else

(5) $\qquad A[n] = \sqcup_{p \in Preds(n)} f_p(A[p])$

end if

(6) if $v \neq A[n]$ then

(7) $\qquad W = n_1, n_2, \ldots, n_k, n_{s_1}, n_{s_2}, \ldots$

where $n_{s_i} \in Succs(n)$ and for $1 \leq l \leq k \quad n_{s_i} \neq n_l$

end if

We now show that this iterative worklist solution algorithm computes the same results as the general solver algorithm, which was shown in Theorem 1 to be the MFP of the induced system of equations.

Theorem 2 (Iterative Worklist Solver Correctness)
Given an instance, $I = (G, N_\sqcap, N_\sqcup, M)$, of a monotone complete-lattice data flow analysis framework, $D = (L, F)$, if we apply Algorithm 2 then the solution, the $A[n]$, is the maximum fix point solution of the following system of equations:

$$\begin{aligned} X[n_0] &= \perp \\ \forall n \in N_\sqcap : X[n] &= \sqcap_{p \in Preds(n)} f_p(X[p]) \\ \forall n \in N_\sqcup : X[n] &= \sqcup_{p \in Preds(n)} f_p(X[p]) \end{aligned}$$

Proof: Recall that Algorithm 1 operated by performing a sequence of node visits, where the sequence had two constraints. We prove this theorem by showing that the computation performed by Algorithm 2 is equivalent to such a sequence of node visits.

The initialization phases of both algorithms are identical. A visit in Algorithm 1 is equivalent to executing statements 3-5 in Algorithm 2. The rest of the statements in Algorithm 2 are used to maintain the worklist, W, which enforces the node visitation sequence. We show that these statements work together to satisfy the conditions on visitation sequences expressed in Algorithm 1. We consider just the case of nodes in $N_\sqcap$; for nodes in $N_\sqcup$ we make an identical argument.

No premature termination The condition states:

> if $\exists n \in N_\sqcap : A[n] \neq \sqcap_{p \in Preds(n)} f_p(A[p])$ after visiting node n_i in the sequence, then there is an integer $j > i$ such that $n_j = n$.

Let the last time we visited node n be at step *last* where $last < i$, $n_{last} = n$. We have $A^{(last)}[n] = \sqcap_{p \in Preds(n)} f_p(A^{(last-1)}[p])$. The only way that $\sqcap_{p \in Preds(n)} f_p(A[p])$ can change value is if one of the predecessors, say p_1, changes value and that requires a visit to node p_1. Let such a visit occur at step *new* where $last < new \leq i$ and there are no other visits to p_1 at steps *last* through $new - 1$. This visit will occur on the new^{th} iteration of the body of

Algorithm 2; we remove p_1 from the head of W. Line 2 of the algorithm saves the old value of $A[p_1]$ which is $A^{(last)}[p_1]$ in our local variable v. Our visit of p_1, lines 3-5, compute a new value $A^{(new)}[p_1] \sqsubset A^{(last)}[p_1]$; the values are not equal because of the assumption that the predecessor changes its value. The test at line 6 succeeds and we proceed to add all successors of p_1 to W that are not already in W. Thus, we are assured that n is on the worklist after our visit to p_1. Since, the worklist contains at most $|N|$ elements we are guaranteed to visit node n before step $i+|N|+1$. Thus, the condition is satisfied since there exists a j such that $i < j < i + |N| + 1$.

Termination after stabilization The condition states:

> If after visiting node n_i, $A[n] = \sqcap_{p \in Preds(n)} f_p(A[p])$ for all $n \in N_\sqcap$, then the sequence will eventually halt.

After we visit node n_i we may visit additional nodes. Let n be the next node we visit at step $i+1$; it is taken off of W at line 1. We store the current value $A^{(i)}[n]$ in local variable v at line 2. Our visit to node n, in lines 3-5, computes $A^{(i+1)}[n] = \sqcap_{p \in Preds(n)} f_p(A^{(i)}[p])$. The assumption in the condition is that $A^{(i)}[n] = \sqcap_{p \in Preds(n)} f_p(A^{(i)}[p])$ for all $n \in N_\sqcap$. So, $A^{(i+1)}[n] = A^{(i)}[n] = v$ and the test at line 6 fails; we add no nodes to W. Subsequent iterations all behave in the same way because the visits $i, i+1, \ldots$ cause no change to $A[n]$ for all $n \in N_\sqcap$. Since we never execute line 7 in any of these visits, each visit reduces the number of elements on W by 1. Thus, there can be at most $|N|-1$ more iterations of the body of Algorithm 2 after the i^{th}, and the sequence of node visits halts at j where $j < i + |N|$.

Since all phases of Algorithm 2 meet the specification of Algorithm 1 our theorem holds by Theorem 1.

□

We can express a bound on the running time of this algorithm in terms of the height, k, of L.

Theorem 3 (Iterative Worklist Solver Complexity)
Given an instance, $I = (G, N_\sqcap, N_\sqcup, M)$, of a monotone complete-lattice data flow analysis, $D = (L, F)$, where L has height k, if we apply Algorithm 2 it will terminate in $O(k|N|^2)$ time.

Proof:

The initialization phase requires $O(|N|)$ operations to assign values to each node and $O(|N|)$ operations to initialize W.

We use W as a queue that contains at most a single instance of each element. Lines 1 and 7 require standard enqueue/ dequeue operations. Querying whether

W has a given successor node in line 7 requires a containment test. Both of these requirements can be handled efficiently by implementing W as a collection of nodes that is threaded as a queue and also threaded as bucket elements of a hash table; hashing on node identity is straightforward. In this case, lines 1-3 and 6 all require $O(1)$ operations.

The cost of lines 4 and 5 are equal, so we consider only one of them. Line 4 is the iterated meet over all predecessors, of the predecessors transfer function applied to the predecessors current value. A standard, and reasonable, assumption is that transfer functions, f_n, require $O(1)$ operations to evaluate. Thus, line 4 requires $O(|N|)$ [5] operations since in the worst-case a node may have all other nodes as predecessors.

Line 7 requires that for each successor, we check for containment in W; if not contained we append the successor to W. The containment check requires $O(1)$ operations, since it is a hash table lookup, and the append requires $O(1)$ operations. Thus, line 7 requires $O(|N|)$ operations since in the worst-case a node may have all other nodes as successors.

In total the body of the main loop requires $O(|N|)$ operations.

The number of iterations of the body, or visits to a node, can be bounded by making use of the conditions on the node visitation sequence from Algorithm 1. It was shown in the proof of Theorem 2 that the sequence of nodes taken from the W satisfies those conditions. Recall, from the proof of Theorem 1, that the values at all nodes in the graph are descending monotonically through the lattice from $\top$; the lowest they can go is $\bot$. In the worst case, each iteration of the main loop causes a single node to descend a single value in the lattice; multiple nodes descending multiple values in a single iteration is quite possible but only makes the algorithm converge faster. At least one node will descend otherwise the **termination after stabilization** condition guarantees that Algorithm 2 we will terminate in at most $|N|$ steps. At most a node can descend k, the height of the lattice, times before it hits $\bot$. Since there are $|N|$ nodes in the graph there are at most $k|N|$ descending iterations. Thus, there are $O(k|N|)$ iterations of the main loop.

The theorem holds since $O(k|N|)$ iterations of $O(|N|)$ operations is $O(k|N|^2)$.

□

Algorithm 2 is a practical improvement over Algorithm 1 because of the node visitation order it enforces. Nodes are only visited when they have the potential to change value. If the sequence of nodes removed from the worklist is $\ldots n_i, n_{i+1}, \ldots, n_j \ldots$ and $n_i = n_j = n$ then there is an integer k such that $i < k < j$ and $n_k \in Preds(n)$; intuitively, we only put a node on the worklist if one of its predecessors changes values.

Careful consideration of Algorithm 2 reveals a number of additional opportunities for eliminating unnecessary computation. As described, for each node the algorithm stores a value that does not reflect the effects of the transfer function at that node; this is called

[5] Evaluating a binary-tree of meets would require only $O(log|N|)$ operations. In this analysis we don't bother doing that because the cost of this step is dominated by the cost of line 7.

the in value for a node. We can easily modify Algorithm 2 to keep track of the value that reflects the effects of a node's transfer function; this is called the *out* value for a node. We can perform the comparison $out(n) \not\sqsupseteq in(s)$ where $s \in Succs(n)$ to determine whether the value computed at n can possibly affect the value at s. If it cannot, then there is no need to put s on the worklist. We can also keep track of the set of predecessors of a node n that have changed value, call it $Changed$, and only compute $in(n) = in(n) \sqcap (\sqcap_{p \in Changed} out(p))$ rather than considering all predecessors. These optimizations do not improve the worst-case time bound for Algorithm 2, but for dense graphs and graphs whose nodes have high fan-in or fan-out they can yield significant practical speedup.

5 An Example

To illustrate the utility of complete-lattice data flow frameworks, we consider an example data flow analysis problem, which computes flow graph dominators. Intuitively, for a sequential program represented by a statement flow graph domination captures information about statements that *must* precede the execution of other statements on any program executions in which they both occur; we call this *statement precedence* information. Formally, $Dom(m) = n$ if all paths from n_0 to m in G pass through n. This is a safe approximation to statement precedence information in the sense that $Dom(m) = n$ implies that $statement_n$ precedes $statement_m$ on all program executions in which both occur.

Hecht presents a formulation of it as a distributive semi-lattice data flow analysis framework [Hec77]:

$$\begin{aligned} L &= (\mathcal{P}(N), \subseteq, \cap) \\ F &= \{f_{ident}(S) = S : S \subseteq \mathcal{P}(N)\} \cup \\ &\quad \{f_n(S) = S \cup n : n \in N \wedge S \subseteq \mathcal{P}(N)\} \cup \\ &\quad \{f_{misc}(X) = Y : X, Y \subseteq \mathcal{P}(N)\} \end{aligned}$$

Where the f_{misc} are included to make F closed under composition. Note that, for this problem, $\top = N$ and $\perp = \emptyset$. An instance of this framework is defined as (G, M) where $M(n) = f_n$.

For concurrent programs we are also interested in gathering statement precedence information [CKS90, CK93, DS91, GS93, Mas93]. Unfortunately, using Dom information often yields an overly pessimistic safe approximation. Recall the example concurrent flow graph in Figure 1. The statement *end* is clearly preceded by the statement def_1, since all program executions that lead to the last statement of the process must pass through the first statement of the process. The existence of paths through the other processes crossing to the middle process via one of the square nodes means that this precedence relationship will not be captured by the Dom information.

Intuitively, we want to treat control flow merge points differently from communication/synchronization merge points. At a control flow merge point we know that one, but not all, of the predecessor statements has executed immediately prior to the merge. In this case, only statements that precede all of those predecessors are guaranteed to precede the merge point. At a synchronization merge point we know that all of the predecessor statements have executed immediately prior to the merge. In this case, the statements that precede any of those predecessors are guaranteed to precede the merge point. This problem can be formulated as a complete-lattice framework problem as follows:

$$\begin{aligned}
L &= (\mathcal{P}(N), \subseteq, \cap, \cup) \\
F &= \{f_{ident}(S) = S : S \subseteq \mathcal{P}(N)\} \cup \\
&\quad \{f_n(S) = S \cup n : n \in N \wedge S \subseteq \mathcal{P}(N)\} \cup \\
&\quad \{f_{misc}(X) = Y : X, Y \subseteq \mathcal{P}(N)\}
\end{aligned}$$

Where the f_{misc} are included to make F closed under composition. Note that, for this problem, $\top = N$ and $\perp = \emptyset$. An instance of this framework is defined as $(G, N_\sqcup, N_\sqcap, M)$ G is a concurrent flow graphs whose control flow merge points and synchronization merge points are $N_\sqcup$ and $N_\sqcap$ respectively. The function map is defined as in the semi-lattice case $M(n) = f_n$. Since the lattice is the powerset of the set of flow graph nodes its height is $|N|$. Thus, applying Algorithm 2 gives an $O(|N|^3)$ solution procedure for the problem. The bound of this algorithm is equal to the bound of algorithms developed by Callahan [CKS90] and Grunwald [GS93] for the statement precedence problem.

6 Conclusions

We have extended the theoretical foundations of data flow analysis to accommodate a number of natural analysis problems for concurrent programs. The formulation of complete-lattice frameworks is compatible with previous semi-lattice frameworks. Thus, a general implementation for the complete-lattice frameworks, such as Algorithm 2, is applicable to semi-lattice frameworks as well. We have illustrated, by way of example, the ease with which analysis problems for concurrent programs can be formulated as a complete-lattice framework. On the theoretical front we intend to look at the extent to which distributivity, continuity and rapidity [MR90] can be exploited in complete-lattice frameworks. We intend to investigate a wide variety of standard and special purpose data flow analysis problems to understand the extent to which they are accommodated in the new theory. Given that complete-lattice frameworks promise improved precision for data flow analysis of concurrent programs, we intend to empirically investigate the extent to which this occurs in practice.

References

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[CC77] P. Cousot and R. Cousot. 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*, pages 238–252, 1977.

[CK93] S.C. Cheung and J. Kramer. Tractable flow analysis for anomaly detection in distributed programs. In *Proceedings of the European Software Engineering Conference*, 1993.

[CKS90] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the Second Symposion on Principles and Practice of Parallel Programming.* ACM, 1990.

[DC94] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. *Software Engineering Notes*, 19(5):62–75, December 1994. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[DS91] E. Duesterwald and M.L. Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification (TAV4)*, October 1991.

[Due91] Evelyn Duesterwald. Static concurrency analysis in the presence of procedures. Department of Computer Science 91-6, University of Pittsburgh, Pittsburgh, Penssylvania 15260, March 1991.

[GS93] Dirk Grunwald and Harini Srinivasan. Efficient computation of precedence information in parallel programs. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[GW76] S.L. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of the ACM*, 23(1):172–202, 1976.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* The Computer Science Library Programming Language Series. Elsevier North-Holland, 1977.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, October 1973.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23:158–171, 1976.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Mas93] S.P. Masticola. *Static Detection of Deadlocks in Polynomial Time.* PhD thesis, Rutgers University, May 1993.

[MR90] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[MR93] S.P. Masticola and B.G. Ryder. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[SHW93] Harini Srinivasan, James Hook, and Michael Wolfe. Static single assignment for explicitly parallel programs. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 260–272, Charleston, South Carolina, January 1993.

[Tar81] Robert Endre Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, jul 1981.

[Tay83] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *Transactions on Programming Languages and Systems*, 13(2):181–210, apr 1991.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1991.