

SPECIFICATION AND SOLUTION OF MULTISOURCE DATA FLOW
PROBLEMS

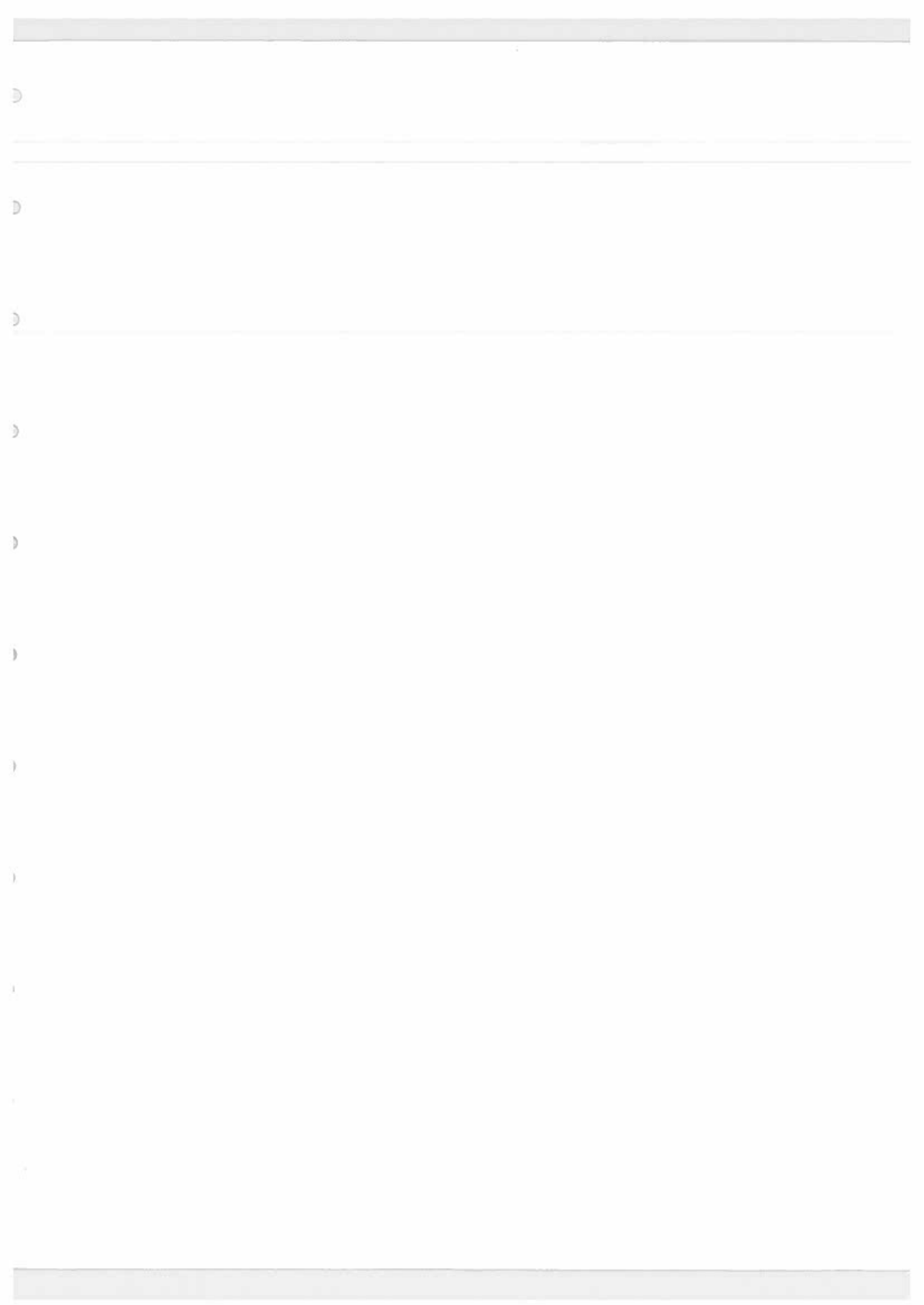
by

JOHN HOWARD ELI FISKIO-LASSETER

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2006



SPECIFICATION AND SOLUTION OF MULTISOURCE DATA FLOW
PROBLEMS

by

JOHN HOWARD ELI FISKIO-LASSETER

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2006

"Specification and Solution of Multisource Data Flow Problems," a dissertation prepared by John Howard Eli Fiskio-Lasseter in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:



Dr. Michal Young, Chair of the Examining Committee



Date

Committee in charge:

Dr. Michal Young, Chair
Dr. Dejing Dou
Dr. Michael Pangburn
Dr. Andrzej Proskurowski

Accepted by:



Dean of the Graduate School

© 2006 John Howard Eli Fiskio-Lasseter

An Abstract of the Dissertation of

John Howard Eli Fiskio-Lasseter for the degree of Doctor of Philosophy
in the Department of Computer and Information Science

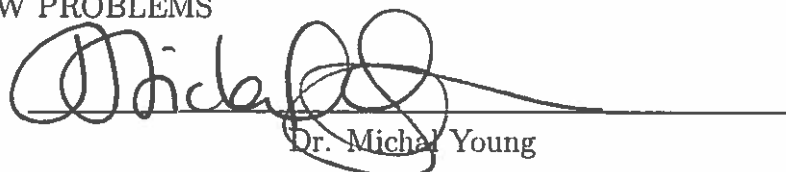
to be taken

December 2006

Title: SPECIFICATION AND SOLUTION OF MULTISOURCE DATA

FLOW PROBLEMS

Approved:

A handwritten signature in black ink, appearing to read "Michal Young", is written over a horizontal line. The signature is stylized and cursive.

Dr. Michal Young

The unified approach embodied in data flow frameworks has inspired several toolkits that partially automate the construction of solvers for various data flow problem classes. Unfortunately, existing toolkits have fairly limited application. Although most can handle both classical and more advanced analyses, the full breadth of flow analysis forms—particularly those arising in new applications of the technique—lies outside the range of any single system.

This dissertation extends and reformulates the traditional approach, in order to support the automatic generation of solvers for multisource data flow analysis problems. In this very general class, a data flow problem is modeled by a directed graph in which more than one type of edge may be defined and information about the type of an edge considered along with the flow value it carries. While this describes, roughly, all forms of flow analysis, attempts to unify the members of the multisource family

have so far held only theoretical interest. The approach we present here thus offers a significant increase in flexibility to existing flow analysis tools.

Our method is based on a new approach to user-level specification of data flow problems. Existing approaches require instantiation of the four parameters of a data flow framework: value lattice, function space, flow graph, and local semantic functional. All such approaches presume a fixed global semantics, an assumption that fails in the general multisource case. Instead, we propose a domain-specific language, which harkens back to the earliest “flow equation” forms, allowing us to view the global abstract semantics itself as a fifth parametric component.

We then leverage this language-theoretic view to develop a new technique for the automatic generation of efficient solvers. There are several improvements to the basic iterative solution strategy that exploit properties of the flow graph model in order to discover advantageous orderings and avoid redundant computation in the global solution. All rely on a known relationship between the structure of the flow graph and the flow of information between nodes. To adapt these to the multisource case, we present a new method for discovering the information flow relation induced by arbitrary global semantic specifications.

CURRICULUM VITAE

NAME OF AUTHOR: John Howard Eli Fiskio-Lasseter

PLACE OF BIRTH: Raleigh, NC, U.S.A.

DATE OF BIRTH: December 29, 1968

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Earlham College

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science, 2006,
University of Oregon

Master of Science in Computer and Information Science, 1998,
University of Oregon

Bachelor of Arts in Philosophy, 1992, Earlham College

AREAS OF SPECIAL INTEREST:

Static Analysis
Type Theory
Programming Language Semantics

PROFESSIONAL EXPERIENCE:

Lausanne Postdoctoral Fellow, Computer Science Department,
Willamette University

2006–present

Visiting Instructor, Mathematics and Computer Science Department,
Lewis and Clark College

2005–2006

Adjunct Instructor, Linfield College

2004

Graduate Research Fellow, Computer and Information Science Department,
University of Oregon

2001–2003

Graduate Teaching Fellow, Computer and Information Science Department,
University of Oregon

1996–2001, 2004

AWARDS AND HONORS:

University of Oregon / Mortarboard Society

Graduate Teaching Fellows Award, 2001

University of Oregon, Computer & Information Science Dept.

Best GTF Award, 2000

PUBLICATIONS:

X. Zhang, M. Young, and J. H. E. F. Lasseter. Refining Code-Design Mapping with Flow Analysis. In *Proceedings of the 2004 ACM Symposium on Foundations of Software Engineering (SIGSOFT 04)*, pages 231–240, 2004.

J. Fiskio-Lasseter and M. Young. Flow Equations as a Generic Programming Tool for Manipulation of Attributed Graphs. In *Proceedings of the Fourth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 02)*, pages 69–76, 2002.

J. Fiskio-Lasseter and A. Sabry. Putting Operational Techniques to the Test: A Syntactic Theory for Behavioral Verilog. *Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS 99)*, September 30–October 1, 1999. Paris, France. Elsevier Electronic Notes in Theoretical Computer Science, Volume 26.

ACKNOWLEDGMENTS

My thanks, first and foremost, goes to my dissertation advisor, Michal Young. Throughout the many years I spent bringing this project to fruition, Michal has offered a seemingly limitless supply of patience, faith, support, unnervingly encyclopedic knowledge, priceless teaching advice, excellent home-roasted coffee (strong, the way God intended), and the always well-placed tough question. If I achieve any of his own measure of integrity, scholarship, and educational vision, I will be proud.

I am grateful, too, for the support and sound advice of my other committee members—Andrzej Proskurowski, Dejing Dou, and Michael Pangburn. For your help in bringing this thing to completion, and for years of exciting and collegial interaction, both in the classroom and out: thank you.

Although they did not serve official roles in this research project, I want to acknowledge a particular debt of gratitude to Amr Sabry (my M.S. thesis advisor, and still an inspiration to me as a scholar and teacher) and Zena Ariola (for two excellent years spent as her teaching assistant for our undergraduate programming languages course, and the extraordinary opportunities that provided).

My final year as a grad student was spent very happily in a visiting position at Lewis and Clark College. Both my students and fellow teachers—especially Yung-Pin Chen, Peter Drake, Jeff Ely, Liz Stanhope, and Iva Stavrov—provided a first professional experience beyond my most optimistic dreams.

Through ten years of grad school, I owe my sanity and more than a few moments of inspiration to a great many other people, some colleagues, some office mates, some in town, some far away. All friends: Star Holmberg, Jan Saunders, Cheri Smith, (the Keepers of the Department); Paul Bloch, Lauradel Collins, and David Sullivan (our crack systems staff); the Graduate Teaching Fellows Federation (AFT / AFL-CIO, Local 3544); the Oregon Natural Desert Association; my beloved sisters Kate and Helen, brothers William, Austin, and Benjamin, sisters through marriage Beth, Leticia, and Carol, and my second parents, Mike and Ginger Fiskio; Peter Boothe,

Doug Botka, Kevin Glass, Kevin Huck, Tim Jackson, Ted Kirkpatrick, Lauren and Uri Lessing, Jennifer Mori, Amitabha Roy, Lorena Reynolds, Deborah Seuss, Max Skorodinsky, Shasta Willson, Justin Wirth, and Xiaofang Zhang. Thank you, one and all.

Finally, and most of all, I am grateful to my parents, Rollin and Ruth Lasseter, for their unwavering love and support. Even after 38 years, I can hardly believe my luck.

And to Janet, my wife, confidant, colleague, and best pal, to whom this work is dedicated. For friendship, love, hiking, excellent musical taste, our cats, shop talk on teaching, and so much more.

For my family: Janet, Pavarotti, and Pippin.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
II. BACKGROUND	4
2.1 Mathematical Preliminaries	4
2.1.1 Lattices	5
2.1.2 Examples	7
2.1.3 Lattice Properties	10
2.1.4 Function Properties	13
2.1.5 Fixed Points	14
2.2 Program Models for Data Flow Analysis	16
2.3 Some Illustrative Analyses	18
2.4 Data Flow Frameworks	21
2.5 Applications of the Framework View	28
2.5.1 Theoretical Properties	28
2.5.2 Solution Algorithms	35
2.5.3 Aside: Correctness and Abstract Interpretation	45
2.6 Data Flow Analysis Construction Kits	46
2.6.1 Flow Values	47
2.6.2 Function Space	48
2.6.3 Flow Graph	50
2.6.4 Flow Map	51
2.7 Advanced Forms	52
2.7.1 Interprocedural Analysis	52
2.7.2 Analysis of Concurrent Programs	54
2.7.3 Bidirectional Data Flow Analysis	55
2.8 Limitations of the Toolkit Approach	57
2.8.1 Data Flow Equations and the MFP Specification	58
2.8.2 Toward Toolkit Support for Multisource Flow Analysis	61

Chapter	Page
III. FRAMEWORK SPECIFICATION	62
3.1 <i>K</i> -tuple Data Flow Frameworks	62
3.1.1 Properties	65
3.2 Specification of Data Flow Framework Instances	68
3.2.1 Semantic Domain	69
3.2.2 Syntax	70
3.2.3 Behavior	74
3.2.4 Examples	79
3.3 Limitations of the <i>K</i> -tuple Approach	86
IV. SPECIFICATION OF MULTISOURCE PROBLEMS	92
4.1 A Flow Equation Language	93
4.2 Examples	95
4.3 Properties	99
4.4 Related Work	106
V. SOLUTION TECHNIQUES	109
5.1 The Influence of Influence	110
5.2 Challenges to the Determination of Influence	114
5.3 A Hybrid Technique for Static Determination	118
5.3.1 Factoring Complex Flow Expressions	118
5.3.2 Static Construction of Influence Functions	119
5.3.3 Application	121
5.4 Examples From the Literature	125
5.5 Related Work	132
VI. CONCLUSION AND FUTURE WORK	133

Chapter	Page
APPENDIX: EXAMPLE SPECIFICATIONS	136
A.1 Live Variables Analysis (Roke)	137
A.2 Callahan's <i>Kill</i> Analysis (Roke)	137
A.3 PRE Flow Equations (Roke)	138
A.4 PRE Analysis, <i>K</i> -Tuple Form (Roke)	140
A.5 Duesterwald/Soffa Ordering Analysis (Roke)	142
A.6 Ordering Analysis (GenSet)	145
 BIBLIOGRAPHY	 150

LIST OF FIGURES

Figure		Page
2.1	The lattices $(2^{\{1,2,3\}}, \subseteq, \cup, \emptyset, \{1, 2, 3\})$ and $(2^{\{1,2,3\}}, \supseteq, \cap, \{1, 2, 3\}, \emptyset)$. . .	9
2.2	A flat lattice.	9
2.3	A lattice of integer intervals.	11
2.4	(a) C code to quicksort elements of global array <code>a</code> , and (b) 3-address code representation of <code>qsort</code> fragment ([ASU86], p. 588–590)	19
2.5	A control flow graph, in basic block form, built from the 3-address code of Fig. 2.4(b). ([ASU86], p. 591)	20
2.6	<i>MOP</i> and <i>MFP</i> forms, forward analysis	25
2.7	Morel-Renvoise Algorithm (modified version in [KD94])	56
3.1	Syntax of global semantic specifications as \mathcal{K} programs	73
3.2	Type rules for \mathcal{K} —definition, equivalence, and declaration rules	77
3.3	Type rules for \mathcal{K} —values	78
3.4	Modified Morel-Renvoise Algorithm [Dha91]	85
3.5	Example MIG, taken from [DS91], p. 39.	89
4.1	A banned nested iteration and an example portion of a flow graph on which this analysis might be defined. Note the way that flow information at the intermediate node w is itself used in the transformation of the value from y to x	103
5.1	Example IFG arising from LV analysis	117
5.2	The influence expression function.	120

LIST OF TABLES

Table	Page
2.1 Four classical analyses as instances of a distributive framework.	27
2.2 Relationship of common user parameters to framework components . . .	46
3.1 Types and type judgments	76

CHAPTER I

INTRODUCTION

It is possible to automate the construction of efficient solvers for data flow analysis problems over flow graph models with an arbitrary number of edge types.

This is significant because the manual construction of a data flow analyzer is both time-consuming and error-prone. Further, many new and experimental applications of data flow analysis involve the development of a new flow graph model and usually have a more complex relationship between flow graph structure and the abstract semantics of the analysis, increasing the development burden further.

In traditional data flow analysis this burden is mitigated by a well-developed unified theory. This theory presents the technique of data flow analysis in terms of a family of algebraic structures known as *data flow frameworks* [MR90a], providing a rigorous classification of problem families and a clear methodology for reasoning about the convergence, precision, and complexity of a given analysis. Coupled with generic solution algorithms, the theory also yields a straightforward way to solve a given flow analysis problem with guaranteed results.

Indeed, the parametric nature of these generic algorithms suggests immediately that such solvers can be constructed automatically, by implementing the generic components of a data flow framework as the building blocks of an analyzer generation toolkit. The approach is straightforward: The lattice structures of the framework guide construction of data structures in the toolkit; the solution algorithm, properly

instantiated, becomes the heart of the generated analyzer; and framework abstractions become user contracts, constrained by simple interfaces.

While a variety of these systems have been developed [Las04], there remain several important families of data flow problems for which automatic construction of solvers is unrealized. For many such families, this is primarily a matter of inadequate demand, in the sense that all of the technical components—framework and generic solution algorithm—have known solutions.

In other cases, the technical problems themselves have remained open. One important example is this family of data flow problems over edge-typed flow graphs, known collectively as *multisource* data flow analysis. In this very general problem class, a data flow problem may be modeled with a directed multi-graph in which more than one type of edge is defined and information about the type of an edge considered along with the flow value it carries. Such problems occur in many advanced forms of data flow analysis, including interprocedural analysis, analysis of concurrent programs, bidirectional analysis, and a number of non-traditional forms.

Although automatic solver generation has been realized for many of these problems, the solutions are specialized to their respective domains. A general framework for multi-source problems was presented by Masticola *et al.* in [MMR95], but no solution algorithm was given.

This dissertation presents an extension of the approaches used in existing data flow analysis toolkits to encompass the automatic generation of solvers for arbitrary multisource data flow analysis problems. We begin in Chapter II with an extended tutorial on data flow analysis frameworks, along with an account of the ways they can be implemented in analyzer construction toolkits. We finish the chapter with a discussion of the shortcomings of the traditional approach, even those supporting advanced analysis forms, concluding that this approach must be fundamentally extended if we are to support the general multisource case. In particular, there must be a mechanism for the user to specify, in addition to the standard data flow framework/instance components, the global abstract semantics.

In Chapter III, we develop an extension of framework-based specification, in the form of a domain specific language for direct encoding of multisource problems as

instances of a cross-product lattice framework. The framework is essentially that of Masticola *et al.* [MMR95], although we extend and improve their formulation to encompass heterogeneous cross-product formulation. Unfortunately, the resulting specifications are often highly obscure and artificially complex. We conclude that, while useful for reasoning about essential properties of an analysis, the framework view itself imposes a needlessly heavy burden for analysis specification.

Chapter IV therefore abandons the traditional framework-based approach to specification of flow analyses. Instead, we develop the language Roke,¹ a declarative-style, domain-specific language for specifying analyses in a declarative style that is reminiscent of early “flow equation” forms. While this results in specifications that lack a direct encoding as instances of a data flow framework, we are nonetheless able to achieve many of the same guarantees of convergence and low-order polynomial-time complexity.

More importantly, we still have efficient solution algorithms available, which means that, from the viewpoint of efficiency, we pay nothing for giving up the framework-based specification approach. This matter is the subject of Chapter V. While specifications in our language can be solved with a naive iterative fixpoint algorithm, the result is a loss in the performance, owing to wasted effort on the part of the solver. To improve on this, we develop a new hybrid static/dynamic technique that is used to generate automatically a workset solver for any given specification.

¹In homage to the great storyteller, Ursula K. LeGuin.

CHAPTER II

BACKGROUND

The material in this chapter is divided into two overarching themes. In Sections 2.1 through 2.6, we review mathematical foundations and the basic elements of the data flow analysis technique. These sections motivate and explain the fundamental ideas underlying data flow frameworks, which provide a unified view of large classes of flow analyses. We also review some practical considerations that arise in the adaptation of a framework as the basis for an analyzer generation toolkit.

Having established the general foundations, we then present some of the leading examples of multisource data flow analysis, reviewing the state of the science in extending the classical framework and solution techniques to these advanced forms.

From this background review, we establish a motivation for the remainder of this dissertation: that the traditional approach to the construction of data flow analysis toolkits, although effective in specialized problem domains, is fundamentally unsuited as a basis for the specification and automatic generation of arbitrary multisource problems.

2.1 Mathematical Preliminaries

Lattices, in particular complete ones, form a crucial part of the underlying principles of data flow analysis. In this section, we review the ideas behind their

construction, along with some properties that are significant in guaranteeing the finite convergence of a data flow analysis. Much of the following is taken from [DP02].

2.1.1 Lattices

Definition 2.1.1 (Partially-Ordered Sets). A *partially-ordered set* (or *poset*) is a pair (\mathcal{D}, \leq) of set \mathcal{D} and binary relation \leq satisfying, for all $a, b, c \in \mathcal{D}$:

$$\begin{aligned} \text{(reflexivity)} \quad & a \leq a \\ \text{(antisymmetry)} \quad & a \leq b \text{ and } b \leq a \implies a = b \\ \text{(transitivity)} \quad & a \leq b \text{ and } b \leq c \implies a \leq c \end{aligned}$$

The related notation $<$ and the dual orderings \geq and $>$ have the expected meanings.

Definition 2.1.2 (Bounds). Let (\mathcal{D}, \leq) be a poset, and let $B \subseteq \mathcal{D}$.

1. An element $a \in \mathcal{D}$ is an *upper bound* of B if $a \geq b$, for all $b \in B$. Similarly, a is a *lower bound* of B if $a \leq b$, for all $b \in B$.
2. Assuming it exists, a is the *least upper bound* of B , written $\sqcup B$, if it is an upper bound of B and for all $b \in \mathcal{D}$, if b is an upper bound of B then $a \leq b$. The *greatest lower bound* of B , written $\sqcap B$ is defined analogously.

Alternately, $\sqcup B$ is called the *join* (or *supremum*) of B , while $\sqcap B$ is known as the *meet* (or *infimum*). Both can be considered as binary operators. For $\{a, b\}$, we write $a \sqcup b$ for $\sqcup\{a, b\}$ and $a \sqcap b$ for $\sqcap\{a, b\}$. If they exist, both $\sqcup B$ and $\sqcap B$ are unique.

Definition 2.1.3 (Lattices). Let $(\mathcal{D}, \leq_{\mathcal{D}})$ be a non-empty poset.

1. \mathcal{D} is a *lattice* if $a \sqcup b$ and $a \sqcap b$ exist, for all $a, b \in \mathcal{D}$.
2. \mathcal{D} is a *complete lattice* if $\sqcup B$ and $\sqcap B$ exist, for all $B \subseteq \mathcal{D}$.
3. B is a (complete) *sublattice* of \mathcal{D} if $(B, \leq_{\mathcal{D}})$ is a (complete) lattice and $B \subseteq \mathcal{D}$.

If they exist, the *least element* (or *bottom*) of \mathcal{D} is $\perp = \sqcup \emptyset = \sqcap \mathcal{D}$, and the *greatest element* (or *top*) is $\top = \sqcap \emptyset = \sqcup \mathcal{D}$. A lattice with both \top and \perp elements is *bounded*. All complete lattices are bounded, and all finite lattices are complete.

Much of the data flow analysis literature makes use of lattice structures that omit either the \sqcup or \sqcap operator. These are known as *semilattices* or *complete semilattices*, and are merely a convenience. The missing operator can always be recovered from the defined one:

Proposition 2.1.1 (Connecting Lemma). *Let (\mathcal{D}, \leq) be a lattice and let $a, b \in \mathcal{D}$. The following are equivalent:*

1. $a \leq b$
2. $a \sqcup b = b$
3. $a \sqcap b = a$

Proof. [DP02], Lemma. 2.8 □

Consequently, a lattice can be characterized by either its order relation or its \sqcap and \sqcup operators. We can therefore describe a lattice in a number of equivalent ways, including:

$$(\mathcal{D}, \leq) \equiv (\mathcal{D}, \sqcup, \sqcap) \equiv (\mathcal{D}, \leq, \sqcup, \perp, \top) \equiv (\mathcal{D}, \leq, \sqcup, \sqcap, \perp, \top)$$

In keeping with established practice, we will generally use the explicit semilattice form, or where the context is clear, simply \mathcal{D} .

Finally, the *dual* of any lattice-theoretical proposition is obtained by interchanging \sqcap and \sqcup , \top and \perp , and \leq and \geq . One of the more elementary but useful facts regarding lattices is the “duality principle”:

Proposition 2.1.2 (Duality). *A proposition is true of a partial order (\mathcal{D}, \leq) if and only if its dual holds for the partial order (\mathcal{D}, \leq_s) , where $a \leq_s b \iff a \geq b$.*

Proof. Trivial. □

This tells us that anything we can say about a lattice can also be said (dually) about the lattice when we turn it “upside down”.

Among the many applications of duality is to reconcile the lattice structures used in most of the data flow analysis literature with those used in the literature on abstract interpretation [CC77a]. The data flow analysis literature commonly uses meet semilattice structures (*i.e.* omitting the \sqcup operator), while descriptions of the same analyses in most abstract interpretation papers—as well as in this thesis—employ join semilattices. Prop. 2.1.2 tells us that this is irrelevant.

Finally, a few elementary algebraic properties of \sqcap and \sqcup :

Proposition 2.1.3. *Let \mathcal{D} be a lattice. The following identities hold for all $a, b, c \in \mathcal{D}$:*

$$\begin{aligned}
 (\text{associativity}) \quad & (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c), \text{ and} \\
 & (a \sqcap b) \sqcap c = a \sqcap (b \sqcap c) \\
 (\text{commutativity}) \quad & a \sqcup b = b \sqcup a, \text{ and} \\
 & a \sqcap b = b \sqcap a \\
 (\text{idempotence}) \quad & a \sqcup a = a \\
 & = a \sqcap a \\
 (\text{absorbtion}) \quad & a \sqcup (a \sqcap b) = a \\
 & = a \sqcap (a \sqcup b)
 \end{aligned}$$

Proof. [DP02], Theorem. 2.9

□

2.1.2 Examples

Some complete lattices that prove useful in data flow analysis are the following:

Example 2.1.1. The lattice of boolean values $(\{1, 0\}, 0 < 1, \wedge, 0, 1)$:

$$\begin{array}{c}
 1 \\
 | \\
 0
 \end{array}$$

Example 2.1.2 (Powerset Lattices). Let A be a set, and 2^A denote the powerset of A . On this set, we can consider the order $(2^A, \subseteq, \cup, \emptyset, A)$ and its dual $(2^A, \supseteq, \cap, A, \emptyset)$, both complete lattices. Figure 2.1 gives two examples of powerset lattices, over the set $\{1, 2, 3\}$.

Example 2.1.3 (Flat Lattices). Let A be any set, with $\top, \perp \notin A$. A *flat lattice* can be constructed from $A \cup \{\top, \perp\}$ with the ordering $\forall a, b \in A : a \leq b \iff a = b$ and $\forall a \in A : \perp < a < \top$.

A flat lattice of the integers is given in Figure 2.2.

Example 2.1.4 (Cartesian Product). Let \mathcal{D}_1 and \mathcal{D}_2 be complete lattices. The *product lattice* $\mathcal{D} = \mathcal{D}_1 \times \mathcal{D}_2$ is defined by

$$\mathcal{D} = \{\langle d_1, d_2 \rangle \mid d_i \in \mathcal{D}_i\}$$

and for all $\langle a, b \rangle$ and $\langle c, d \rangle \in \mathcal{D}$

$$\langle a, b \rangle \leq \langle c, d \rangle \iff a \leq_1 c \wedge b \leq_2 d$$

Further, $\perp = \langle \perp_1, \perp_2 \rangle$, and likewise for \top . Similarly, \sqcup is “pointwise” defined.

This has a straightforward extension to a product of k lattices, $\mathcal{D}_1 \times \dots \times \mathcal{D}_k$.

Example 2.1.5 (Lattices of Functions). Let $(\mathcal{D}, \leq, \sqcup, \perp, \top)$ be a complete lattice. Define \mathcal{F} , the *total function space on \mathcal{D}* , by

$$\mathcal{F} = \{f : \mathcal{D} \rightarrow \mathcal{D} \mid f \text{ a total function}\}$$

and

$$\forall f, g \in \mathcal{F} : f \leq g \iff (\forall d \in \mathcal{D} : f(d) \leq g(d))$$

Further, for all $F \subseteq \mathcal{F}$

$$\sqcup F = \lambda v. \sqcup \{f(v) \mid f \in F\}$$

and $\perp = \lambda v. \perp$ (similarly for \sqcap and \top).¹

¹The notation $\lambda v. E$ is taken from the functional programming literature; it denotes a function of a single argument with body E .

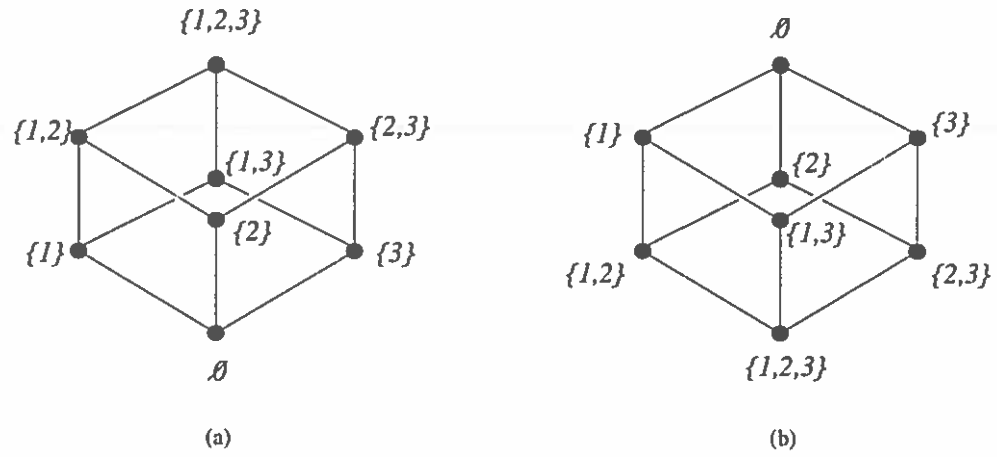


FIGURE 2.1: The lattices $(2^{\{1,2,3\}}, \subseteq, \cup, \emptyset, \{1, 2, 3\})$ and $(2^{\{1,2,3\}}, \supseteq, \cap, \{1, 2, 3\}, \emptyset)$.

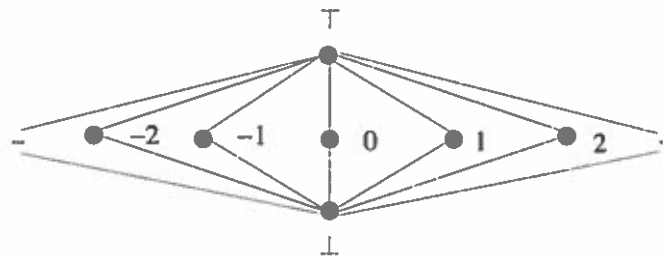


FIGURE 2.2: A flat lattice.

2.1.3 Lattice Properties

The height of a lattice is an important property for reasoning about the convergence of data flow analyses. More generally, we will use the concept of a *chain*. Conceptually related, both ideas give a measure of the largest set of distinct, comparable elements within a lattice:

Definition 2.1.4. An *ascending chain* is a sequence d_0, d_1, \dots of distinct elements in \mathcal{D} , such that $\forall i : d_i \leq d_{i+1}$. A descending chain is defined analogously. The *length* of a chain, if finite, is equal to 1 less than the number of elements. The *height* of \mathcal{D} is the length of the longest chain (ascending or descending).

The boolean value lattice and flat lattices have heights 1 and 2, respectively. The powerset lattice 2^A has height $|A|$. The product lattice $\mathcal{D}_1 \times \dots \times \mathcal{D}_k$ (whose components have heights h_1, \dots, h_k) has height $\sum_{i=1}^k h_i$.

Definition 2.1.5. A lattice \mathcal{D} has the *ascending chain condition (ACC)* property if, for every ascending chain d_0, d_1, \dots in \mathcal{D} , there is an index j such that $d_j = d_k$, for every $k > j$. The *descending chain condition (DCC)* is defined analogously.

It is clear that a lattice has finite height if and only if it has both the *ACC* and *DCC* properties, although neither *ACC* nor *DCC* alone guarantees finite height. All finite lattices have finite height. The lattice of natural numbers $(\mathbb{N} \cup \{\infty\}, \leq, \min(), \max(), 0, \infty)$, with \leq as ordinary numerical comparison, has the *DCC* property, but not *ACC*, while the lattice $(\mathbb{Z} \cup \{-\infty, \infty\}, \leq, \min, \max, 0, \infty)$ has neither.

Although less common, lattices of infinite height do occur in program analysis:

Example 2.1.6 (Interval Analysis). Figure 2.3 depicts the lattice of integer intervals $(\mathcal{D}, \leq, \sqcup, \perp, [-\infty, \infty])$. Elements of this lattice are pairs $[d_1, d_2]$, where $d_i \in \mathbb{Z} \cup \{-\infty, \infty\}$ and $d_1 \leq d_2$. They are ordered by inclusion:

$$[a, b] \leq [c, d] \iff (c \leq a) \wedge (b \leq d)$$

with $[a, b] \sqcup [c, d] = [\min(a, c), \max(b, d)]$. This lattice has infinite ascending and descending chains.

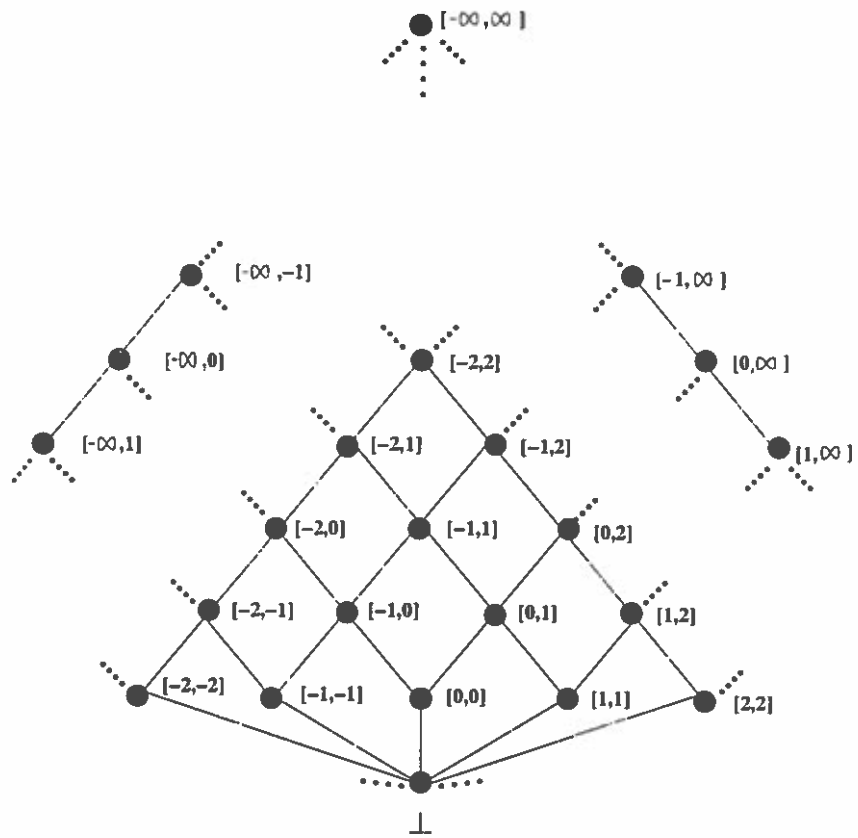


FIGURE 2.3: A lattice of integer intervals.

Definition 2.1.6. A lattice \mathcal{D} is *distributive* if it satisfies the law

$$\forall a, b, c \in \mathcal{D} : a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$$

\mathcal{D} is *modular* if it satisfies

$$\forall a, b, c \in \mathcal{D} : a \geq c \implies a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup c$$

The powerset lattices of Figure 2.1 are both distributive and modular. In fact, all distributive lattices are modular ([DP02], Lemma 4.2), though not *vice-versa*. On the other hand, consider the lattices \mathbf{M}_3 (“diamond”) and \mathbf{N}_5 (“pentagon”):



\mathbf{M}_3 , although modular, is not distributive, while \mathbf{N}_5 is neither modular nor distributive.

Proposition 2.1.4. Let \mathcal{D} be a lattice. \mathcal{D} is non-modular if and only if it has a sublattice isomorphic to \mathbf{N}_5 . \mathcal{D} is non-distributive if and only if it has a sublattice isomorphic to either \mathbf{N}_5 or \mathbf{M}_3 .

Proof. [DP02], Theorem 4.10. □

The most common example of a non-distributive lattice in data flow analysis is the flat lattice over a set with more than two elements, for which the embedding of \mathbf{M}_3 as a sublattice is easy to see.

Definition 2.1.7. Let \mathcal{D} be a lattice. For $a \in \mathcal{D}$, the element $b \in \mathcal{D}$ is a *complement* of a if both $a \sqcap b = \perp$ and $a \sqcup b = \top$. If it exists, the *unique complement* of a is written \bar{a} . We say that \mathcal{D} is *uniquely complemented* if $a \in \mathcal{D} \implies \bar{a} \in \mathcal{D}$ (in which case $\bar{\bar{a}} = a$).

A relatively straightforward use of Prop. 2.1.1 shows that in a distributive lattice, an element can have at most one complement. Examples of elements with more than one complement are also easy to find: take any flat lattice over a set with more than two elements.

Definition 2.1.8. A complete lattice \mathcal{D} is a *boolean lattice* if it is both distributive and uniquely complemented.

This notion is useful in data flow analysis, since the most common lattices we will use—powerset lattices—are boolean. Indeed, every finite boolean lattice is isomorphic to a finite powerset lattice ([DP02], Theorem 5.5 and Corollary 5.6), although this is not true for boolean lattices over infinite sets.

2.1.4 Function Properties

Definition 2.1.9. Let \mathcal{D} be a complete lattice. A function $f : \mathcal{D} \rightarrow \mathcal{D}$ is

1. *monotone*, if $\forall a, b \in \mathcal{D} : f(a \sqcup b) \geq f(a) \sqcup f(b)$;
2. *additive* (or, *distributive*), if $\forall a, b \in \mathcal{D} : f(a \sqcup b) = f(a) \sqcup f(b)$;
3. *completely additive* (\sqcup -*continuous*), if $\forall B \subseteq \mathcal{D}$ s.t. $\sqcup B \in B : f(\sqcup B) = \sqcup f(B)$

The dual notion of additivity is *multiplicativity*, which preserves \sqcap . An additive/multiplicative function is also called a *lattice homomorphism*. For continuity, it is important that \mathcal{D} be a complete lattice. If not, then the property of \sqcup -continuity (resp. \sqcap) applies only to *directed* subsets—i.e. $B \neq \emptyset$, in which for every $A \subseteq B$, $\sqcup A \in B$ (resp. $\sqcap A \in B$). Note that

$$\text{continuity} \implies \text{distributivity} \implies \text{monotonicity}$$

For finite \mathcal{D} , continuity and distributivity are equivalent properties, but this does not hold for the infinite case. Monotonicity is a strictly weaker property than distributivity in either case.

Monotone functions are also called *order-preserving*, as the following makes clear:

Proposition 2.1.5. *Let \mathcal{D} be a lattice, and $f : \mathcal{D} \rightarrow \mathcal{D}$ a function. The following are equivalent:*

1. $\forall a, b \in \mathcal{D} : f(a \sqcup b) \geq f(a) \sqcup f(b);$
2. $\forall a, b \in \mathcal{D} : f(a \sqcap b) \leq f(a) \sqcap f(b).$
3. $\forall a, b \in \mathcal{D} : a \leq b \implies f(a) \leq f(b);$

Proof. [DP02], Prop. 2.19. □

2.1.5 Fixed Points

Much of the early development of the theory of data flow analysis used only semilattices, without an explicit requirement that such lattices be complete or even bounded (e.g. [Kil73, KU77, MR90a]). More recent developments have required the use of complete lattice structures (e.g., [CC79b, Kno98]). One reason for this is that it makes available the classical fixed point existence theorems. These are sometimes known collectively as the “Tarski-Knaster/Kleene Fixpoint Theorems”, although the origins of the basic ideas go back to the 19th century. Lassez *et al.* [LNS82] present some interesting archeological detective work on the basic ideas.

For the following, let \mathcal{D} be the complete lattice $(\mathcal{D}, \leq, \sqcup, \sqcap, \perp, \top)$:

Definition 2.1.10. Let $f : \mathcal{D} \rightarrow \mathcal{D}$ be a monotone function. Define:

1. The set of *extensive* values for f : $Ext(f) = \{v \mid f(v) \geq v\}$
2. The set of *reductive* values for f : $Red(f) = \{v \mid f(v) \leq v\}$
3. The set of *fixed points* of f : $Fix(f) = \{v \mid f(v) = v\}$

We say that f is *extensive* if $Ext(f) = \mathcal{D}$ and *reductive* if $Red(f) = \mathcal{D}$.² (Note that if $Fix(f) = \mathcal{D}$, then f is the identity function.)

²Such functions are also commonly known as *inflationary* (resp. *deflationary*).

Theorem 2.1.6 (Tarski–Knaster [Tar55]). *For a monotone function $f : \mathcal{D} \rightarrow \mathcal{D}$, the set $\text{Fix}(f)$ of fixed points is a complete lattice with ordering \leq inherited from \mathcal{D} . In particular, the least and greatest fixed points are*

- $\text{lfp}(f) = \sqcap \text{Red}(f)$
- $\text{gfp}(f) = \sqcup \text{Ext}(f)$

□

A more constructive formulation is expressed in terms of the iterative application sequence f^i , defined as

$$f^i(v) = \begin{cases} v & i = 0 \\ f \circ f^{i-1}(v) & i > 0 \end{cases}$$

The following result is usually attributed to Kleene (specifically, [Kle52], p.348):

Theorem 2.1.7. *Let f be a function $\mathcal{D} \rightarrow \mathcal{D}$. If f is \sqcup -continuous, then $\text{lfp}(f)$ exists, and is equal to*

$$\text{lfp}(f) = \bigsqcup_{i \geq 0} f^i(\perp)$$

Dually, if f is \sqcap -continuous, then

$$\text{gfp}(f) = \bigsqcap_{i \geq 0} f^i(\top)$$

Proof. [DP02], Theorem. 8.15, which implies the result for complete lattices. □

In fact, f need only be either monotone or extensive (*resp.* reductive) for the results of Theorem 2.1.7 to hold ([DP02], pp.187–188). As a corollary, we have:

Theorem 2.1.8. *If \mathcal{D} satisfies ACC and $f : \mathcal{D} \rightarrow \mathcal{D}$ is \sqcup -continuous (monotone, extensive), then there is some k such that*

$$\text{lfp}(f) = \bigsqcup_{i \geq 0} f^i(\perp) = \bigsqcup_{i \geq 0}^k f^i(\perp)$$

Dually, if \mathcal{D} satisfies DCC and f is \sqcap -continuous (monotone, reductive), then there is some j such that

$$\text{gfp}(f) = \bigsqcap_{i \geq 0} f^i(\top) = \bigsqcap_{i \geq 0}^j f^i(\top)$$

□

2.2 Program Models for Data Flow Analysis

The term *static analysis* refers to a family of techniques, each of which provides a way to extract information about a program's runtime behavior without actually running the program. Naturally, precise determination of almost all such information is impossible: the infinite number of potential program execution states, together with fundamental undecidability results, form an insurmountable barrier. All forms of static analysis must therefore rely on a finite model of program execution, which is at best an approximation of actual behavior. In general, we want this approximation to be *conservative*: the model represents all possible behaviors, but may also include behaviors that do not correspond to any execution.³ As a consequence, the information obtained from analysis will always be valid, but may be uninformative.

In data flow analysis, the approximation of program behavior is carried out in two parts, one representing the program counter and the other the store. In practical terms, this means that we first construct a model of program control flow as a directed graph $G = (N, E)$, with a finite set of *nodes* N and a set of *directed edges*, $E \subseteq N \times N$. For each node $x \in N$ we denote by $E(x)$ set of immediate *successors* of x in G ; in addition, we will denote the *predecessors* of x by $E^{-1}(x) = \{w \mid (w, x) \in E\}$. In most formulations, it is also common to identify two distinguished nodes in N : $s \in N$ is the unique *start* node, with $E^{-1}(s) = \emptyset$, and a path from s to every node in G , while

³We can also err in the other direction, using an approximation that does not consider non-executable behavior at the cost of omitting some actual execution behavior. This is the idea behind runtime testing.

$e \in N$ is the *exit* node, with $E(e) = \emptyset$, and a path from every node in G to e .⁴ The construction of this graph model, *control flow analysis*, serves as a preprocessing step to the data flow analysis itself.

The traditional flow graph model is built from program source code, starting at any level of abstraction, from abstract syntax tree to intermediate (3-address) code to machine code. Conceptually, each node in G is associated with a program point, although the precise meaning of “program point” (and hence the choice of nodes) can vary. N can be chosen to represent the individual instructions, basic blocks, and so on.

Edges record the flow of information from point to point. This is usually determined by the execution order of the program statements, which is in turn determined by the program control flow. Elements of E may therefore correspond to program counter advances, procedure calls, syntactic or statement ordering dependencies, static single assignments, and synchronization or interleaving of tasks, among other choices. The choice of program representation can affect the accuracy and performance of a flow analysis, but it is irrelevant to the analysis technique itself.

In general, it is not possible to determine such flows precisely, and thus the edges in G represent a *superset* of what can actually occur at runtime. For example, the most common application for data flow analysis is in the compile-time optimization of procedure bodies in traditional imperative programs [ASU86]. In this setting, known as *intraprocedural* analysis, the basic flow graph construction is a *control flow graph* (*CFG*), in which the edges model control transfers between statements or basic blocks. The resulting flow graph structure models all branch statements as nondeterministic, as if both branches could be taken.

A reasonably precise approximation of intraprocedural, imperative control flow can be derived from the syntactic structure of the source code. The *CFG* contains an edge (m, n) if the source code has a control transfer from m to n , ignoring any other program semantics. As an example, consider the C program given in Fig. 2.4(a), specifically, the fragment delimited by the source code comments. A typical data

⁴The presence of s and e are a matter of convenience; since it is always possible to add them as needed, their inclusion imposes no real restrictions.

flow analysis would work with the intermediate representation given in 2.4(b). A corresponding *CFG* is given in Fig. 2.5

2.3 Some Illustrative Analyses

We perform a data flow analysis in order to annotate each node in the flow graph with one of a fixed set of assertions. Following are two of the best-known examples:

Example 2.3.1. The *available expressions (AE)* analysis determines for each node (program point) x , the expressions whose values are guaranteed to have been computed before execution reaches x . Specifically, it annotates each x with a subset of the set of expressions occurring in the program. For the example in Fig. 2.4(b), this is equal to the set *Exp* of temporary variables: $\{t_i \mid 1 \leq i \leq 15\}$.

The set of expressions *guaranteed* available at x is constrained by those that are guaranteed available at the predecessors of x . To wit: let w be a predecessor of x , and let $AE(w)$ denote the expressions available at w . Further, let $f(AE(w))$ denote the expressions that can be inferred from the execution of the program fragment at w , given $AE(w)$. Then the strongest assertion we can infer for x is that

$$AE(x) \subseteq f(AE(w))$$

Hence, the best (*i.e.* largest) value we can infer for $AE(x)$ is the largest set that satisfies the inequalities

$$AE(x) \subseteq \begin{cases} \emptyset & , \text{ if } x = s \\ \bigcap_{w \in E^{-1}(x)} f(AE(w)) & , \text{ otherwise} \end{cases}$$

Since f is intended to simulate execution at each x , we will use the following:

$$f(v) = (v \setminus Kill_{AE}(x)) \cup Gen_{AE}(x)$$

$Kill_{AE}(x)$ is the set of expressions which contain a variable assigned to in x , while $Gen_{AE}(x)$ is the set of expressions whose values are computed (without having any of the constituent variables subsequently reassigned) in x .

```

void qsort(int m,int n) {
    int i,j;
    int v,x;
    if (n <= m) return;

    /* fragment begins here */

    i = m-1; j=n; v=a[n];
    while(1) {
(a)      do i++; while(a[i] < v);
          do j--; while(a[j] > v);
          if(i >= j) break;
          x=a[i]; a[i]=a[j]; a[j]=x;
    }
    x=a[i]; a[i]=n; a[n]=x;

    /* end fragment */

    qsort(m,j); qsort(i+1,n);
}

```

1. $i \leftarrow m-1$	11. $t_5 \leftarrow a[t_4]$	21. $a[t_{10}] \leftarrow x$
2. $j \leftarrow n$	12. if ($t_5 > v$) goto 9	22. goto 5
3. $t_1 \leftarrow 4*n$	13. if ($i \geq j$) goto 23	23. $t_{11} \leftarrow 4*i$
4. $v \leftarrow a[t_1]$	14. $t_6 \leftarrow 4*i$	24. $x \leftarrow a[t_{11}]$
5. $i \leftarrow i+1$	15. $x \leftarrow a[t_6]$	25. $t_{12} \leftarrow 4*i$
6. $t_2 \leftarrow 4*i$	16. $t_7 \leftarrow 4*i$	26. $t_{13} \leftarrow 4*n$
7. $t_3 \leftarrow a[t_2]$	17. $t_8 \leftarrow 4*j$	27. $t_{14} \leftarrow a[t_{13}]$
8. if ($t_3 < v$) goto 5	18. $t_9 \leftarrow a[t_8]$	28. $a[t_{12}] \leftarrow t_{14}$
9. $j \leftarrow j-1$	19. $a[t_7] \leftarrow t_9$	29. $t_{15} \leftarrow 4*n$
10. $t_4 \leftarrow 4*j$	20. $t_{10} \leftarrow 4*j$	30. $a[t_{15}] \leftarrow x$

FIGURE 2.4: (a) C code to quicksort elements of global array *a*, and (b) 3-address code representation of *qsort* fragment ([ASU86], p. 588–590)

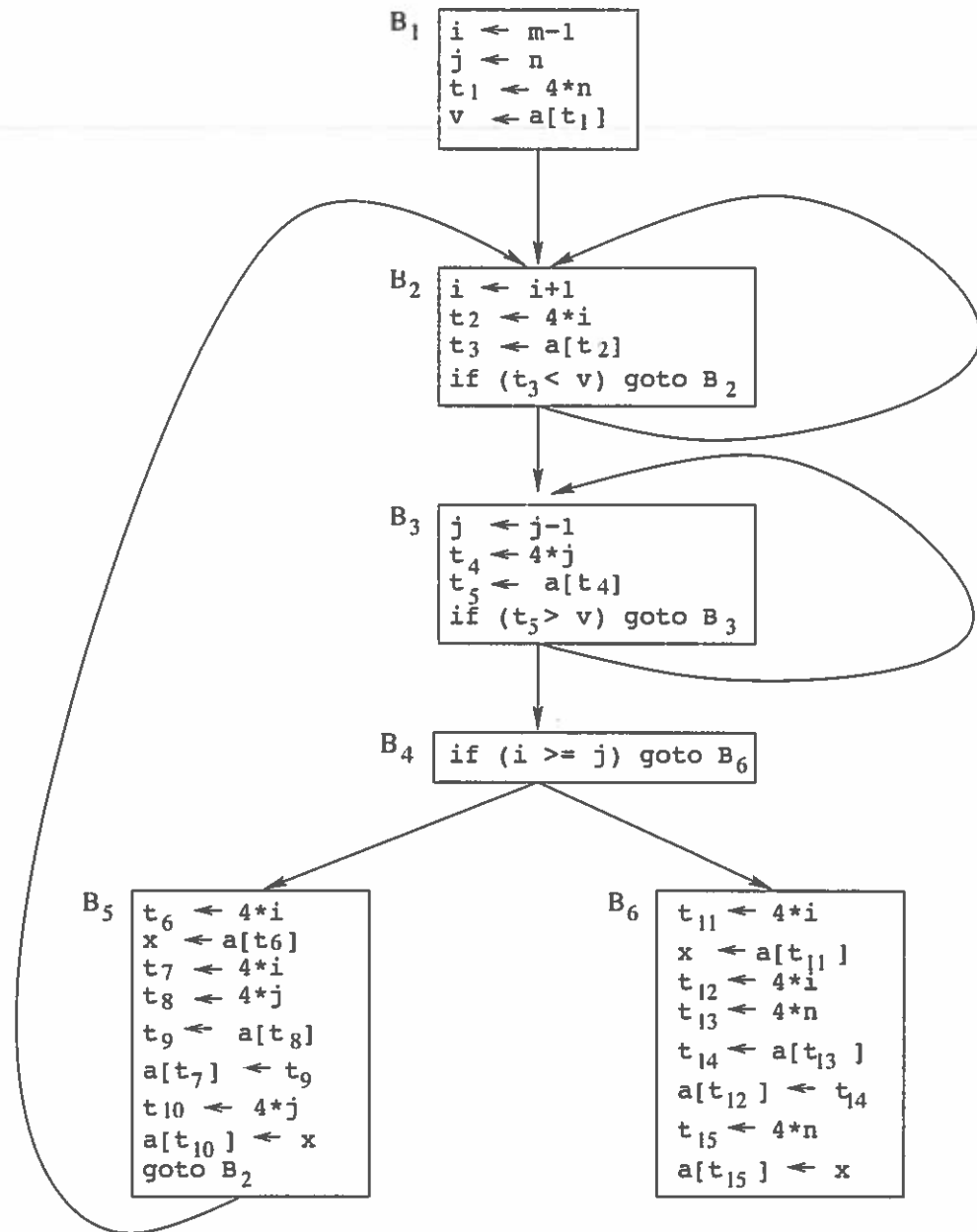


FIGURE 2.5: A control flow graph, in basic block form, built from the 3-address code of Fig. 2.4(b). ([ASU86], p. 591)

Note that AE actually denotes a *system* of constraints, which, if there are cycles in the flow graph, may be individually or mutually recursive. It is not obvious, therefore, that a solution exists for every system of AE constraints. We shall revisit this concern in Section 2.5.

Example 2.3.2. A *live variables (LV)* analysis determines for each program point x the set of variables whose current values could be used at a later point, before being overwritten. This analysis annotates each x with a subset of the set Var of variables declared in the program: in Fig. 2.4(b) the set $\{i, j, m, n, v, x, a\}$. Unlike AE , information in LV propagates from control flow *successors*, *i.e.* backwards, with respect to the direction of arcs in G . Further, $LV(x)$ must be no *smaller* than any of the LV values of its successors. Specifically, the best annotation we can deduce for $LV(x)$ is the smallest set which satisfies the inequalities

$$LV(x) \supseteq \begin{cases} \emptyset & , \text{ if } x = e \\ \bigcup_{y \in E(x)} f(LV(y)) & , \text{ otherwise} \end{cases}$$

where

$$f(v) = (v \setminus Kill_{LV}(x)) \cup Gen_{LV}(x)$$

Although the definition of f is similar to one given for AE , here $Kill_{LV}(x)$ is the set of variables assigned to in x , while $Gen_{LV}(x)$ is the set of variables used (without being earlier assigned to) in x .

2.4 Data Flow Frameworks

Data flow analyses can be broadly classified by to three independent characteristics:

- *Quantification over execution paths.* A *may-analysis* (or “any-paths”) determines properties that hold for some possible executions of a program, while a

must-analysis (“all-paths”) determines properties that hold on every possible execution.

- *Scope of analysis.* A *local* data flow analysis is performed on a basic block—*i.e.* a maximal sequence of branch-free instructions, with only the first instruction in the block having incoming edges. *Global (intraprocedural)* analysis extends across basic blocks, but within a single procedure body. An *interprocedural* analysis extends across procedure calls. All three of these are traditionally confined to sequential programs, but each can be extended to a *concurrent* setting.
- *Direction (with respect to execution order).* Program statements in a *forward* analysis are related to each other in the order they would execute at run time. They are related in the reverse direction for *backward* analysis. There are also *bidirectional* analyses that use information from both directions, although this requires significant extensions of the classical formulation.

Both *AE* and *LV* are intraprocedural analyses. *AE* is a forward, must-analysis, while *LV* is a backward, may-analysis.

Most important, however, is the similarity between the *AE*, *LV*, and other analysis specifications. Indeed, the key insight behind every data flow analysis toolkit is that all flow analyses solve essentially the same problem. Each form of analysis can be understood as an instance of a family of algebraic structures known as *data flow frameworks*. This unified view was originally presented by Kildall [Kil73], with several additional developments over the next decade ([GW76, CC77a, CC79b, KU76, KU77, Ros80], among others). See Marlowe and Ryder [MR90a] for a comprehensive survey.

In general, the gross structure of a data flow analysis can be decomposed into two parts: a *local semantics* modeling the execution of each program statement and a *global semantics* which specifies the constraints that the final analysis must satisfy [CC77a]. The local component can in turn be decomposed into *abstract values*, and *abstract statements*:

Definition 2.4.1 (Data Flow Framework). A *data flow framework* is a pair $(\mathcal{D}, \mathcal{F})$, where

- $\mathcal{D} \equiv (\mathcal{D}, \leq, \sqcup, \perp, \top)$ is a complete lattice;
- $\mathcal{F} \subseteq \{f : \mathcal{D} \rightarrow \mathcal{D}\}$ is a set of total functions on \mathcal{D} satisfying
 - $f_{id} \equiv (\lambda v.v) \in \mathcal{F}$
 - $\forall f, g \in \mathcal{F}, f \circ g \equiv (\lambda v.f(g(v))) \in \mathcal{F}$
 - $\forall f, g \in \mathcal{F}, f \sqcup g \equiv (\lambda v.f(v) \sqcup g(v)) \in \mathcal{F}$

Each element of \mathcal{D} , called a *flow value*, is an assertion about the possible program states that can hold at runtime. The intuition behind the lattice order is the information content of the assertions, with higher values containing more “noise” (and less information). The elements of \mathcal{F} are the *transfer functions*. They abstract the behavior of each program point by capturing the effect on the flow values as control is transferred to the next point. If every $f \in \mathcal{F}$ is monotone (*resp.* distributive/continuous), we have a monotone (distributive/continuous) data flow framework.

A given data flow framework defines a single analysis problem, parameterized on the program of interest. We instantiate this problem for a program by constructing a flow graph model of the program and assigning to it an abstract semantics in the framework.

Definition 2.4.2 (Data Flow Problem). A *data flow problem* (or *instance*) is a pair $I = (G, \llbracket \cdot \rrbracket)$, where

- $G = (N, E, s, e)$ is a flow graph;
- $\llbracket \cdot \rrbracket : E \rightarrow \mathcal{F}$ is the flow map, assigning a transfer function to each edge in G .

Alternatively, we can define $\llbracket \cdot \rrbracket$ on *nodes* rather than, or in addition to, edges (the association with nodes may be seen as a special case in which the target of an edge is ignored [MR90a]).

A local abstract semantics thus defines a simulation of the program represented by the dependence flow graph G on elements of \mathcal{D} . *What* this simulation should

produce—*i.e.* an overall specification of the analysis requirements—is given by the global abstract semantics. In particular, a *solution* to a data flow problem $(G, \llbracket \cdot \rrbracket)$ will be an assignment $\sigma : N \rightarrow \mathcal{D}$, associating a flow value with each node. Note that we can view σ as an element of the lattice $(N \rightarrow \mathcal{D})$ of finite maps from N to \mathcal{D} , with elements ordered pointwise:

$$\sigma_1 \leq \sigma_2 \iff \forall x \in N, \sigma_1(x) \leq \sigma_2(x)$$

As with the *AE* and *LV* examples, we want the choice of σ to reflect the constraint that no assertion be made for a node x beyond what can be deduced from the nodes that immediately influence it.

This motivates the classical “maximal fixed point” (*MFP*) and “meet over all paths” (*MOP*) constraint forms, as given in Figure 2.6.⁵ The use of the ι_E function is an alternative to the conditional presentation used in Examples 2.3.1 and 2.3.2, and allows us to jettison the assumption that the extremal node s has an isolated entry. The constant value ι , which varies according to the particular framework, is a special “entry value” used to annotate the entry node s . Strictly speaking, it is not necessary, but it is both useful and very common. For the *MOP* equations, $PATHS_G(s, x)$ is the set of all finite paths from s to x in G . We write f_{wx} for the transfer function given by $\llbracket (w, x) \rrbracket$, while f_π is the composition of the transfer functions for each edge on path π , defined as

$$\begin{aligned} f_\epsilon &= \lambda x. x \\ f_{\pi', (w, x)} &= f_{wx} \circ f_{\pi'} \end{aligned}$$

The difference between the two forms is in the candidate sources of information to propagate. In the *MFP* form, we consider only the information at the immediate “upstream” neighbors of x , while the *MOP* solution requires that we consider every possible way that information could have been propagated to x . In both cases, we

⁵The terms “meet” and “maximal” suggest a reverse lattice order from the one we use here. As mentioned above in 2.1, this is an historical artifact of early literature, in which data flow frameworks were presented as meet semi-lattices.

$$(a) \text{ MOP ("meet-over-all-paths")} \quad \sigma(x) \geq \left(\bigsqcup_{\pi \in \text{PATHS}_G(s,x)} f_{\pi}(\iota) \right) \sqcup \iota_E(x)$$

$$(b) \text{ MFP ("maximal fixed point")} \quad \sigma(x) \geq \left(\bigsqcup_{w \in E^{-1}(x)} f_{wx}(\sigma(w)) \right) \sqcup \iota_E(x)$$

$$\text{where } \iota_E(x) = \begin{cases} \iota, & \text{if } x = s \\ \perp, & \text{otherwise} \end{cases}$$

FIGURE 2.6: *MOP* and *MFP* forms, forward analysis

want the best information consistent with the constraints, which is to say we require the smallest assignment

$$\sigma = (\sigma(1), \dots, \sigma(n))$$

that satisfies the inequalities. As with *AE* and *LV*, we can replace all inequalities with equations, if \mathcal{F} is a monotone space.

The *MFP* and *MOP* forms of Figure 2.6 assume a forward data flow analysis. For backward analyses, the *MFP* form is the same, but we replace the $E^{-1}(x)$ term with $E(x)$ (the successors of x), and s with e in the definition of ι_E . The backward *MOP* form is likewise an “edge reversed” version of the forward one, merging all paths from x to the exit node e .

Example 2.4.1. Available Expressions. The *AE* analysis is a data flow framework with lattice $(2^{\text{Exp}}, \supseteq, \cap, \text{Exp}, \emptyset)$, where Exp is the set of expressions occurring in the program, ordered by subset inclusion. In our `qsort` example, this is equal to the set of temporary variables: $\{t_i \mid 1 \leq i \leq 15\}$. Values are merged using set intersection,

and the overall analysis is a forward one. The ι value is $AE(s) = \emptyset$. The transfer function at every edge (x, y) is of the form

$$f_{xy}(v) = (v \setminus Kill_{AE}(x)) \cup Gen_{AE}(x)$$

where $Kill_{AE}(x)$ is the set of expressions which contain a variable assigned to in x and $Gen_{AE}(x)$ the set of expressions whose values are computed (without having any of the constituent variables subsequently reassigned) in x .

Example 2.4.2. Live Variables. *LV* analysis uses the lattice $(2^{Var}, \subseteq, \cup, \emptyset, Var)$, where Var is the set of variables declared in the program: in Fig. 2.4(b) the set $\{i, j, m, n, v, x, a\}$. It is a backward analysis. The ι value is $LV(e) = \emptyset$. The transfer functions are of the same form as *AE*: $Kill_{LV}(x)$ is the set of variables assigned to in x , while $Gen_{LV}(x)$ is the set of variables used (without being earlier assigned to) in x .

Example 2.4.3. Reaching Definitions. The *reaching definitions (RD)* analysis finds, for each node (program point) x , those assignments that might have determined the values of program variables at x . It uses $(2^{Asgn}, \subseteq, \cup, \emptyset, Asgn)$ for a property lattice (*c.f.* Fig. 2.1 (b)), where *Asgn* is the set of assignments occurring in the program, and elements of 2^{Asgn} are ordered by superset inclusion (*i.e.* $x \leq y \iff x \supseteq y$). In Fig. 2.4(b), these are the assignments at lines 1,2,4,5,9,15,19,21,24,28, and 30 (we exclude the temporary variables). As it is a “may” analysis, the flow values are merged using set union. *RD* is a forward analysis. We use entry value $RD(s) = \iota = \emptyset$. Transfer functions again follow the same form as those for *AE* and *LV*: $Kill_{RD}(x)$ is the set of assignments whose left-hand sides are re-assigned in x , and $Gen_{RD}(x)$ the assignments that occur (without the variable on the left-hand side being subsequently re-assigned) in x .

Example 2.4.4. Very Busy Expressions. The *very busy (VB)* expressions at node x are those expressions that are guaranteed to be used at a later point, before any variable occurring in them is re-assigned. It is also a backward, “must” analysis, and uses the same flow value lattice as in *AE*, with ι value $VB(e) = \emptyset$. Transfer functions

are of the same form as the previous three, and $Kill_{VB}(x)$ is the same as that used for AE . However, $Gen_{VB}(x)$ is the set of expressions whose values are computed in x , without having any constituent variables assigned to *beforehand*.

Each of these examples is an instance of a distributive data flow framework. Table 2.1 gives a summary:

TABLE 2.1: Four classical analyses as instances of a distributive framework.

	AE	LV	RD	VB
\mathcal{D}	2^{Exp}	2^{Var}	2^{Asgn}	2^{Exp}
\leq	\supseteq	\subseteq	\subseteq	\supseteq
\sqcup	\cap	\cup	\cup	\cap
\perp, \top	Exp, \emptyset	\emptyset, Var	$\emptyset, Asgn$	Exp, \emptyset
ι	\emptyset	\emptyset	\emptyset	\emptyset
<i>direction</i> (w.r.t. CFG)	<i>forward</i>	<i>backward</i>	<i>forward</i>	<i>backward</i>
$\llbracket(x, y)\rrbracket$	$\{f : \mathcal{D} \rightarrow \mathcal{D} \mid \exists v_K, v_G \in \mathcal{D}. f(v) = (v \setminus v_K) \cup v_G\}$			

Example 2.4.5. Constant Propagation. In *constant propagation (CP)*, we determine for each variable u at each node x , whether u has a guaranteed constant value at x , and if so, what that value is. The lattice is $(Store, \leq, \sqcap, \perp, \top)$, which is

a Cartesian product of pair values: each element of *Store* is a set of pairs $\{(u, v)\}$, one for each program variable u . The value u is drawn from the flat lattice (ref. Ex. 2.1.3) of values corresponding to the type of u (integers, booleans, etc.), with the \perp element carrying the intuition of no available information, and \top denoting a “not constant” value. To merge together two pair values, we apply the flat lattice \sqcup operation pairwise:

$$(u, v_1) \sqcup (u, v_2) = \begin{cases} (u, v_1) & , \text{if } v_2 = \perp \\ (u, v_2) & , \text{if } v_1 = \perp \\ (u, v_2) & , \text{if } v_1 = v_2 \\ (u, \top) & , \text{otherwise} \end{cases}$$

This induces an ordering on *Store* defined as $(u_1, v_1) \leq (u_2, v_2) \iff u_1 = u_2 \wedge v_1 \leq v_2$. Note that this also makes *CP* a “must” analysis. The \perp value is (u, \perp) , for all program variables u , and similarly for \top . *CP* is a forward analysis.

The transfer functions are somewhat involved, but in essence all specify versions of the concrete operations, extended so that they are defined on the \perp and \top values of the flat lattices. For example, $2+2=4$, while $2+\top=\top$. This results in a function space that is monotone, but *not* distributive [KU77].

2.5 Applications of the Framework View

2.5.1 Theoretical Properties

If an analysis is formulated in terms of a data flow framework, several key theoretical properties come for free. Of these, perhaps the most important is a guarantee of convergence. If we are working with a monotone framework in which the flow value lattice \mathcal{D} satisfies the *ACC* condition, the *MFP* solution, σ_{MFP} , is in fact the least fixed point, and it is computable by an easy iterative algorithm: guess initial useful values for every $\sigma(j)$, then iteratively evaluate the right-hand and update the left-hand sides of the equations until the solution stabilizes. We can formalize this by regarding

the *MFP* constraints together with a given data flow problem $I = ((N, E, s, e), [\])$ as defining an evaluation function $F : (N \rightarrow \mathcal{D}) \rightarrow (N \rightarrow \mathcal{D})$ such that

$$F(\sigma) = \lambda n. eval_{\sigma}(n, \sigma)$$

where $eval_{\sigma}(n, \rho)$ is the result of evaluating the right hand side of the *MFP* form of Fig. 2.6, under the substitution $x \mapsto n$ and solution ρ . For forward analysis, this is

$$eval_{\sigma}(x, \rho) = \left(\bigsqcup_{w \in E^{-1}(x)} f_{w_x}(\rho(w)) \right) \sqcup \iota_E(x)$$

Similarly, backward analysis uses

$$eval_{\sigma}(x, \rho) = \left(\bigsqcup_{y \in E(x)} f_{x_y}(\rho(y)) \right) \sqcup \iota_E(y)$$

Algorithm 2.5.1 (Chaotic Iteration).

Input: Data flow problem $I = ((N, E, s, e), [\])$, with $|N| = n$.

Output: $\sigma_{MFP} = (\sigma(1), \dots, \sigma(n))$, satisfying *MFP*

Method:

for each $j \in N$:

$\sigma(j) \leftarrow \iota_E(j)$

while $(\exists j \in N \text{ s.t. } \sigma(j) \not\approx eval_{\sigma}(j, \sigma))$:

$\sigma \leftarrow F(\sigma)$

□

Note that there is an implicit requirement here (and in all variants of the iterative approach) that j be chosen according to a fair strategy [CC77b, CC79a].

Early proofs of termination [KU76, KU77, Kil73] proceeded by induction on the number of iterations to show that iterated application of each transfer function eventually reaches the \top element, if it does not stabilize earlier. If we restrict our framework to complete semilattices, we can take a more general approach by applying Theorems 2.1.6 and 2.1.7 [CC77b, CC79a].

Lemma 2.5.1. *The following function sets are closed under composition and join:*

- *Monotone functions*
- *Distributive functions*
- *Extensive functions*

Proof. [Mas93], Lemmas 6–10. □

Note that duality then gives us, as well

Corollary 2.5.2. *The set of reductive functions is closed under composition and meet.* □

Theorem 2.5.3. *For a monotone framework $(\mathcal{D}, \mathcal{F})$, where \mathcal{D} satisfies ACC, Algorithm 2.5.1 computes $lfp(F)$.*

Proof. Every transfer function is monotone, and so, therefore, is $eval_\sigma$ (Lemma 2.5.1). In turn, this implies that F is monotone, and so we can apply Theorem. 2.1.6 to guarantee the existence of a (unique) least fixed point, $lfp(F)$. Since \mathcal{D} satisfies ACC and N is finite, $(N \rightarrow \mathcal{D})$ satisfies ACC, as well. By Theorem. 2.1.7, we therefore have

$$\sigma_{MFP} = lfp(F) = \bigsqcup_{i \geq 0} F^i(\perp) = \bigsqcup_{i \geq 0}^k F^i(\perp)$$

for some (fixed) k . □

In fact, we need not begin iteration at the \perp element [CP89]. Provided $\iota_E \leq \sigma_{MFP}$,

$$\sigma_{MFP} = \bigsqcup_{i \geq 0}^j F^i(\iota_E)$$

where $j \leq k$. This justifies our use of ι_E as an “initialization” function. As an immediate consequence, we can prove the convergence of Algorithm 2.5.1 on all five

examples from 2.3 above, thus establishing the existence and finite computability of solutions for instances of each analysis.

There are also alternatives to monotonicity that are sufficient to guarantee convergence. For example, Chen [Che03] and Geser *et al.* [GKL⁺96] observe that even when a transfer function is not monotone, a finite number of compositions of the function may be, and such “delay monotonicity” is a sufficient condition for convergence to $lfp(F)$. Another choice is the use of an *inflationary least fixed point* ($ilfp$). In concrete terms, we replace the update

$$\sigma \leftarrow F(\sigma)$$

with

$$\sigma \leftarrow F(\sigma) \sqcup \sigma$$

If F is monotone then $ilfp(F) = lfp(F)$. Otherwise, the inflationary form will still converge to a *minimal* solution for the *MFP* constraint (assuming a lattice with *ACC*), but it may be the case that

$$ilfp(F)(x) > \left(\bigsqcup_{w \in E^{-1}(x)} f_{wx}(\sigma(w)) \right) \sqcup \iota_E(x)$$

and this solution is not necessarily unique [AHV95].

Ideally, we would like to do better than *MFP*. Of the two forms given in Fig. 2.6, *MOP* represents a greater sensitivity to the possible runtime behavior, and so the solution we would really like for I is the assignment σ_{MOP} , which is the smallest one that satisfies the *MOP* inequation.

Unfortunately, we cannot always find it. In particular, Kam and Ullman showed [KU77] that there are a number of ways to formulate *MFP*—all decidable—but the formulations do not always produce identical results, and in every case, the solution may be strictly worse than σ_{MOP} . Specifically, every non-distributive framework has a problem instance for which $\sigma_{MOP} < \sigma_{MFP}$.

In fact, a general solution of the *MOP* specification is impossible:

Theorem 2.5.4 (Kam/Ullman [KU77]). *A general algorithm to compute σ_{MOP} does not exist.*

Proof. [KU77] showed a reduction of *MOP* to the Modified Post Correspondence Problem, which is undecidable. \square

As an informal version of their proof, there are some data flow problems for which a solution of *MOP* would require explicit enumeration of all finite paths in the flow graph, and there are an infinite number of these paths (consider loops).

The *MFP* specification turns out to serve as a reasonable alternative to *MOP* in the sense that it is often a precise computation of *MOP*, and that in no realistic case does it introduce assertions (bogusly) beyond what could be deduced under *MOP*.

Theorem 2.5.5 (Coincidence Theorem [Kil73, KU77]). *Let D be a distributive data flow framework. For every data flow problem on D , $\sigma_{MOP} = \sigma_{MFP}$.* \square

Definition 2.5.1 (Graham/Wegman [GW76]). Let I be a data flow problem. A *safe* solution for I is a solution $\sigma : N \rightarrow \mathcal{D}$ such that $\sigma_{MOP} \leq \sigma$.

Theorem 2.5.6 (Kam/Ullman [KU77]). *Let D be a monotone data flow framework. For every data flow problem on D , the *MFP* solution is safe. That is, $\sigma_{MOP} \leq \sigma_{MFP}$.* \square

As a consequence of its safety and decidability, the *MFP* specification is used for almost all practical analysis settings. Sometimes, however, the result will be strictly worse than the *MOP* ideal. The *MFP* specification thus represents a pessimistic upper bound on what analysis can achieve, and so we usually insist that the global abstract semantics do no worse than this:

Definition 2.5.2 (Graham/Wegman [GW76]). Let I be a data flow problem. An *acceptable* solution for I is a solution $\sigma : N \rightarrow \mathcal{D}$ such that $\sigma \leq \sigma_{MFP}$.

The *MOP* solution σ_{MOP} is therefore the minimum safe solution for I , while σ_{MFP} is the maximum acceptable one.

Depending on the particular framework, the data flow problem, and the chosen implementation strategy, there are a number of other characteristics of the framework

that will be of interest. The standard reference for this is Marlowe and Ryder's survey [MR90a]. What follows is a brief outline of the principal ideas.

Definition 2.5.3 (Local Finiteness Properties).

1. A monotone function f is *k-bounded* if there is some k for which

$$f^k(x) = \bigsqcup_{i=0}^{k-1} f^i(x)$$

We call f *fast* if it is 2-bounded, i.e. $f \circ f = f \sqcup id$.

2. f is *k-semibounded* if for all $x, y \in \mathcal{D}$

$$f^k(x) = \left(\bigsqcup_{i=0}^{k-1} f^i(x) \right) \sqcup f^k(y)$$

We call f *rapid* if it is 1-semibounded.

Tarjan notes [Tar81b] that k -boundedness $\implies k$ -semiboundedness $\implies (k+1)$ -boundedness.

Local finiteness properties relate the convergence of iterated application of the transfer functions to flow graph structure, in particular the summarization of flow graph cycles (which correspond, loosely speaking, to program loops). In algebraic formulations of data flow analysis, they play a significant role in determining the computability, runtime complexity, and precision of a number of solution techniques.

If a transfer function is non-monotone, it may not have fixed points. When the flow value lattice has infinite descending chains, monotonicity of the transfer functions guarantees the existence of fixed points, but is insufficient to guarantee that these values are computable. Even if they are computable, the flow value lattice may have such large height as to make the convergence time unacceptable. In such cases, we may need to make do either with a coarser flow value lattice that suitably approximates the original, or else with an approximation to iterated application of the transfer functions. The possibilities for approximate application constitute the *closure properties* of the function space.

Definition 2.5.4 (Closure Properties of \mathcal{F}). Let $f \in \mathcal{F}$. Define:

1. The *reflexive, transitive closure* of f

$$f^* = \bigsqcup_{i=0} f^i$$

2. The iterated sequence $f^{(i)} = (f \sqcup id)^{i-1}$

$$f^{(i)}(x) = (f(x) \sqcup x)^{i-1}$$

3. If $f^{(i)}$ is of finite length k , then $f^{(k)}$ is the *fastness closure* of f .

4. The *extended fastness closure* of f

$$\bar{f}_E = \bigsqcup_{i=0} (f \sqcup id)^{i-1}$$

Note that f^* is finite if and only if it is k -bounded, and that if f is not monotone, f^* need not exist. On the other hand, $f^{(i)}$ is always monotone and increasing, and so it has fixed points. In particular, $f^{(k)}(\perp)$ is the least *inflationary fixed point* of f , i.e. the smallest x that satisfies

$$x = f(x) \sqcup x$$

Note also that $(f \sqcup id)^i \geq f^i$, and hence is a safe approximation of f . Indeed, as mentioned above (p. 31), $f \sqcup id$ has the same fixed points as f , assuming f is monotone. However, this inequality may be strict, if monotonicity fails. Further, it doesn't really give a stronger termination guarantee, unless f is non-monotone: \bar{f}_E is not computable if the flow value lattice lacks the *ACC* property.

In such cases, f can be approximated further still, by replacing the \sqcup operator for merging iterates with an operator that also yields increasing values, but ones that increase “fast enough” to guarantee convergence. In general we will need some form of upper bound operator to do the trick:

Definition 2.5.5 ([NNH99]). An operator $\phi : (\mathcal{D} \times \mathcal{D}) \rightarrow \mathcal{D}$ is an *upper bound operator* if $v_1 \leq (v_1 \phi v_2) \geq v_2$, for all $v_1, v_2 \in \mathcal{D}$.

Note that other than the comparison relation, no particular properties of ϕ are required. In particular, $v_1 \phi v_2$ may be strictly greater than $v_1 \sqcup v_2$, and ϕ need not be monotone, commutative, associative, or idempotent. Examples in the literature include Rosen's @ operator [Ros80, Tar81b, Tar81a] and the family of *widenings* [CC77a].

Function space properties thus provide a number of powerful tools for reasoning about any given analysis. Of particular interest are the boundedness guarantees, which determine not only the closure properties, but also convergence speed. This view is a refinement to Theorem 2.1.7 rather than an alternative. Although they are related, k -boundedness of the function space is not determined by the height h of the flow value lattice: for example, we would have $k < h$ if an upper bound operator other than \sqcup was used (*ref.* Definition. 2.5.5) [CC77a, Ros80]. It is also possible for $k > h$ (constant propagation, for example).

Nonetheless, we can sometimes obscure the difference:

Lemma 2.5.7. *Let \mathcal{D} be a complete lattice of height k , and let $\mathcal{F} \subseteq \{f : \mathcal{D} \rightarrow \mathcal{D}\}$ be a function space. If \mathcal{F} is extensive or reductive, it is at most $(k + 1)$ -bounded.*

Proof. If $f \in \mathcal{F}$ is extensive, then $f^{j+1} \geq f^j$, $j \geq 0$. Hence, $f^{j+1} \geq \bigsqcup_{i=0}^j f^i$. Since the chain $id \sqcup f \sqcup f^2 \sqcup \dots$ is of length at most k , we have $f^{k+1} = \bigsqcup_{i=0}^k f^i$. Duality yields the result for the reductive case. \square

2.5.2 Solution Algorithms

The practical significance of frameworks comes from the the family of generic solution algorithms that have been developed: with the right parametric definition, a solver for one data flow problem is a solver for all problems in that class.

The work to date can be broadly classified according to whether the algorithm is based on techniques of iteration or elimination. The most widely-known iterative approaches appear in several textbooks, notably Hecht [Hec77] and Nielson *et al.* [NNH99]. A recent, high-level survey of elimination approaches is given in [Las04], while detailed presentations of various algorithms can be found in Hecht [Hec77], Ryder and Paull [RP86], and in most compilers textbooks [ASU86, Muc97].

Elimination algorithms exploit flow graph structure in a divide-and-conquer approach. The key insight behind this class of solvers is that in many flow graphs—in particular, those that arise from the control flow of structured (*i.e.* jump-free) programs—there are nodes whose flow values have a disproportionate influence on substructures within the graph. We can consider these nodes as representatives of the substructures they influence, and thereby compute a solution on a reduced representatives-only flow graph.

For a large number of problems, these solvers are the fastest known, offering order-of-magnitude (or better) improvement over the performance of any iterative approach. Of the various solvers presented in the literature [AC76, GW76, Ros80, Sha80, Tar81a, UII73], the *path algebra* approach developed by Tarjan [Tar81b, Tar81a] (and, independently, by Rosen [Ros80, Ros82]) is the most general-purpose. This approach is the only one that makes direct, practical use of the \circ and \sqcup operators on a framework's function space.

A notable disadvantage of elimination algorithms is that all of them come at the cost of great programming effort. The greatest disadvantage, however, is that all elimination approaches fail on programs with ill-structured control flow, specifically those that give rise to *irreducible* flow graphs [HU72, HU74]. While extensions exist to handle these graphs [Sha80, Tar81a, SGL98], all degrade performance to that of an iterative solver, and make the overall programming burden even heavier. While it is possible to transform irreducible flow graphs to equivalent reducible ones, this cannot be done without combinatorial explosion [CFT03].

By contrast, all *iteration* methods will work, regardless of flow graph structure, with no need to transform the flow graph. Such algorithms offer various improvements over the basic fixpoint computation given in Algorithm 2.5.1. As a rule, they have the advantages of being easy to implement and of working on all problem instances, regardless of flow graph structure. A common implementation of the basic form is the following:

(Alg. 2.5.1—Chaotic Iteration)

Input: *Data flow problem* $((N, E, s, e), [\])$, with $|N| = n$.

Output: $\sigma_{MFP} = (\sigma(1), \dots, \sigma(n))$, *satisfying MFP*

Method:

```

for each  $j \in N$ :
     $\sigma(j) \leftarrow \iota_E(j)$ 
change  $\leftarrow true$ 
while(change):
    change  $\leftarrow false$ 
    for each  $j \in N$ :
        newval  $\leftarrow eval_\sigma(j, \sigma)$ 
        if(newval  $\not\leq \sigma(j)$ ):
             $\sigma(j) \leftarrow newval \sqcup \sigma(j)$ 
            change  $\leftarrow true$ 

```

□

Per the discussion above (p. 31), we have adopted the inflationary update here.

For a framework $(\mathcal{D}, \mathcal{F})$, where \mathcal{F} is k -bounded, and problem instance $((N, E, s, e), [\])$, the iterative approach of Alg. 2.5.1 guarantees a solution to the *MFP* specification (namely, the inflationary *lfp*). If \mathcal{F} is monotone, we have a solution to the equational form (*i.e.* the least fixed point). Further, we have a fairly precise measure of the convergence speed:

Theorem 2.5.8. *Let $(G \equiv (N, E, s, e), [\])$ be an instance of a monotone function data flow framework $(\mathcal{D}, \mathcal{F})$, where \mathcal{D} has height h and where M_o is the largest number of edges in E that share a common sink. Algorithm 2.5.1 converges in $\mathcal{O}(M_o |N|^2 \cdot h)$ join operations.*

Proof. For each node j in the inner loop, we perform $\mathcal{O}(M_o)$ join operations. The inner loop iterates $|N|$ times. If there is any node j whose value $\sigma(j)$ changes, then

its value must increase, and this can happen at most h times. Hence the outer loop can run $\mathcal{O}(|N|h)$ times, giving an overall cost of $\mathcal{O}(M_o |N|^2 h)$.

□

In the worst case (where G is a complete graph), $M_o = |N|$. However, there are few if any practical situations in which this is the case. For classical analysis, in fact, it is customary to assume that M_o is bounded by some constant for all programs. Similarly, the largest number of edges sharing a common source, M_s , is generally assumed to be bounded. For example, in flow graphs modeling the intraprocedural flow of imperative programs, we usually take $M_o = M_s = 2$. This yields the more common bound for Algorithm 2.5.1 of $\mathcal{O}(|N|^2 h)$. Lastly, if we consider the k -boundedness of \mathcal{F} instead of h , we may be able to refine the bound further (if $k < h$), substituting k for h . Tarjan shows that this refined bound is tight not only for Algorithm 2.5.1 but for all iteration-based algorithms ([Tar76], Theorem 5).

Although serviceable, Algorithm 2.5.1 represents a fair amount of wasted effort. In the first place, each iteration involves a recomputation of the flow value at *every* node, regardless of whether that value has changed. Second, in applying the evaluation step F , there is no constraint on the order in which the nodes of N are considered. While it makes little difference for convergence of the solution, for performance purposes, some orders are better than others.

This motivates two improvements to Algorithm 2.5.1, the *round robin* and *workset* solvers. These are given on the following two pages as Algorithms 2.5.2 and 2.5.3. We present the forward forms only. For the backward applications of round robin, we use a postorder instead of reverse postorder traversal. For workset solvers, the backward-analysis definition of *infl* uses elements in $E^{-1}(j)$ instead of $E(j)$. Backward analysis under each algorithm requires also the appropriate definition of *eval*.

The idea behind the round-robin solver is to choose an evaluation order—namely, a reverse post-order traversal of the flow graph—that follows program control flow as much as possible. Since Algorithm 2.5.2 differs from Algorithm 2.5.1 only in the order in which nodes are considered (which is clearly a fair strategy), we can apply the same arguments for convergence and correctness.

Algorithm 2.5.2 (Round Robin [HU75]).

Input: *Data flow problem* $((N, E, s, e), [\])$, with $|N| = n$.

Output: $\sigma_{MFP} = (\sigma(1), \dots, \sigma(n))$

Method:

for each $j \in N$:

$\sigma(j) \leftarrow \iota_E(j)$

$change \leftarrow true$

while ($change$):

$change = false$

for $j = 2$ to n : // ordered by a reverse postorder traversal of G

$newval \leftarrow eval_\sigma(j, \sigma)$

if ($newval \not\leq \sigma(j)$):

$\sigma(j) \leftarrow newval \sqcup \sigma(j)$

$change \leftarrow true$

□

Algorithm 2.5.3 (Basic Workset Solver).

Input: *Data flow problem* $((N, E, s, e), [])$, with $|N| = n$.

Output: $\sigma_{MFP} = (\sigma(1), \dots, \sigma(n))$

Method:

for each $j \in N$:
 $\sigma(j) \leftarrow \iota_E(j)$
 $W.add(j)$ // Initialize W to contain every node

while ($!W.empty()$):
 $j \leftarrow W.extract()$
 $newval \leftarrow eval_\sigma(j, \sigma)$
 if ($newval \not\leq \sigma(j)$):
 $\sigma(j) \leftarrow \sigma(j) \sqcup newval$
 $W \leftarrow infl(W, \sigma, j)$

Where:

$infl(W, \sigma, j) \stackrel{def}{=}$
 for each $k \in E(j)$:
 $W.add(k)$
 return W

□

By contrast, the workset solver ignores consideration of advantageous ordering but uses an elegant mechanism to avoid redundant effort by considering at each iteration only the flow values at nodes which might have changed. When a change is detected at some node j , we consider exactly the nodes that are directly influenced by j , and record that these nodes may require reevaluation by placing them in a *workset*, W .

For the worst case of a complete or nearly-complete graph, this does not offer any savings, since we would still need to reevaluate σ on $\mathcal{O}(|N|)$ nodes with each update. In most practical settings, however, the savings are substantial:

Theorem 2.5.9. *Let $(G \equiv (N, E, s, e), [\])$ be an instance of a monotone function data flow framework $(\mathcal{D}, \mathcal{F})$, where \mathcal{D} has height h . Algorithm 2.5.3 converges in $\mathcal{O}(M_\bullet M_\bullet |N| h)$ join operations.*

Proof. On each iteration of the while loop, we remove one element j from W . We then perform $\mathcal{O}(M_\bullet)$ join operations, after which $\sigma(j)$ is either left unchanged or else assigned the new value *newval*. If we assign to $\sigma(j)$, it is with a strictly larger value. This can happen at most h times. With each of these assignments, the call to *infl* adds $\mathcal{O}(M_\bullet)$ new elements to W , so each $j \in N$ adds $\mathcal{O}(h \cdot M_\bullet)$ elements to W . Hence, we perform $\mathcal{O}(M_\bullet |N| h)$ iterations of the while loop, for an overall cost of $\mathcal{O}(M_\bullet M_\bullet |N| h)$ join operations. □

W is an instance of an abstract data type \mathcal{W} with operations *add()* (which adds to W the node given as argument), *extract()* (which removes and returns one node currently in W), and an *empty()* test. We make no commitment to a particular implementation; common choices include a simple queue, a hash table, and a height-balanced tree. We can combine the round-robin and workset approaches, by implementing W as a priority queue, ordered according to a reverse postorder numbering of N [NNH99]. A number of further extensions can be found in the literature. One notable example is to compute the strong components of the graph and compute a local depth-first spanning tree for each strong component. The strong components form a directed acyclic graph, which is processed in topological order. The develop-

ment of this approach is described in [HDT87]. In every case, the cost to maintain W is secondary to the overall cost of the algorithm.

More important are the internals of the *infl* function. We have already discussed its contribution to the complexity of the solver. In addition, the correctness of Algorithm 2.5.3 depends in part on *infl* preserving a key invariant of W . For the technical definition, we adopt the following conventions:

- We define the predicate

$$I_W(x) \stackrel{\text{def}}{=} \sigma(x) \not\leq \text{eval}_\sigma(x, \sigma) \implies x \in W$$

- We write the sequential execution of s and t as $(s; t)$

Definition 2.5.6. Given a function $f : \mathcal{W} \rightarrow (N \rightarrow \mathcal{D}) \rightarrow N \rightarrow \mathcal{W}$, workset $W : \mathcal{W}$, a finite map $\sigma : N \rightarrow \mathcal{D}$, and an assignment a to $\sigma(j)$ we say that f is a *workset function* for $\langle a, \sigma, W \rangle$ if it satisfies the following:

If $I_W(x)$ holds for all $x \in N \setminus \{j\}$ before execution of a , then after $(a; f(W, \sigma, j))$, $I_W(x)$ holds for all $x \in N$.

For the next lemma, we let a_σ denote the assignment $\sigma(j) \leftarrow \sigma(j) \sqcup \text{newval}$ in Algorithm 2.5.3.

Lemma 2.5.10. *Let $(G \equiv (N, E, s, e), [\])$ be an instance of a monotone data flow framework $(\mathcal{D}, \mathcal{F})$, where \mathcal{D} satisfies ACC. If *infl* is a workset function for $\langle a_\sigma, \sigma, W \rangle$, then Algorithm 2.5.3 terminates with $\sigma = \sigma_{MFP}$.*

Proof. (Adapted from the proof of Lemma 6.4 in Nielson *et al.* [NNH99])

We show first that the while loop maintains the following two invariants:

1. $\forall x \in N : \sigma(x) \leq \sigma_{MFP}(x)$

On initial entry to the loop, the invariant holds (by assumption of ι_E). Now suppose it holds at the beginning of iteration $m \geq 1$. If $\sigma(j)$ is left unchanged at the end of the loop, then the invariant is preserved. Otherwise, we assign to $\sigma(j)$ so that the new value, denoted $\sigma(j)_m$, satisfies:

$$\begin{aligned}
\sigma(j)_m &= \sigma(j)_{m-1} \sqcup \text{newval} \\
&= \sigma(j)_{m-1} \sqcup \bigsqcup_{i \in E^{-1}(j)} f_{ij}(\sigma(i)_{m-1}) \\
&\leq \sigma_{MFP}(j) \sqcup \bigsqcup_{i \in E^{-1}(j)} f_{ij}(\sigma_{MFP}(i)) \\
&= \sigma_{MFP}(j) \sqcup F(\sigma_{MFP})(j) \\
&= \sigma_{MFP}(j)
\end{aligned}$$

The inequality in the third step holds by the inductive hypothesis and the monotonicity of f_{ij} , the fourth step is by definition of $F(\sigma)(j)$, and the fifth because σ_{MFP} is a fixed point of F .

2. $\forall x \in N : I_W(x)$

On initial entry to the loop, the invariant holds (since all nodes are in the workset). Within the loop body, the invariant assumption implies that when we extract j from W , it is the only node not on the workset for which $I_W(j)$ could fail. Hence we have $I_W(x)$, for all $x \in N \setminus \{j\}$. By assumption, the execution of $(a_\sigma; \text{infl}(W, \sigma, j))$ therefore yields $\forall x \in N : I_W(x)$, so the invariant is preserved.

On termination of the loop (by Theorem 2.5.9), W is empty. By Invariant 2, we therefore have $\sigma(j) \geq F(\sigma)(j)$, for all $j \in N$. F is therefore reductive on σ , and so $\sigma \geq \sigma_{MFP}$ (by Theorem 2.1.6). Combining this with the Invariant 1 yields the final result.

□

Lemma 2.5.11. *Let σ , infl , and W be defined as in Algorithm 2.5.3, and let a_j denote the assignment $\sigma(j) \leftarrow \sigma(j) \sqcup \text{newval}$. The function infl is a workset function for $\langle a_j, \sigma, W \rangle$.*

Proof. Suppose that immediately before execution of a_j we have $\forall x \in N \setminus \{j\} : I_W(x)$. If there is no arc $(j, j) \in E$, then a_j increases the value of $\sigma(j)$ such that $\sigma(j) \geq \text{eval}_\sigma(j, \sigma)$. Otherwise, $j \in E(j)$, and all such nodes are added to W . So $I_W(j)$ holds.

Now suppose that immediately after a_j there is some k such that $\sigma(k) \not\leq \text{eval}_\sigma(k, \sigma)$ but $k \notin W$ (by assumption, there is no such k beforehand). This must be because $\text{eval}_\sigma(k, \sigma)$ has changed with the new value of $\sigma(j)$, and this is only possible if $j \in E^{-1}(k)$. But then $k \in E(k)$, and so it is added to W by *infl*, completing the proof. □

As an immediate corollary, we therefore have:

Theorem 2.5.12. *Given $(\mathcal{D}, \mathcal{F})$ and $(G, \llbracket \rrbracket)$ as above, Algorithm 2.5.3 terminates with $\sigma = \sigma_{MFP}$.*

Remark. The proofs of Lemmas 2.5.10 and 2.5.11 assume a forward analysis. Adaptation of these lemmas to the backward case is straightforward. □

A common refinement to Algorithm 2.5.3 is to store *edges* in W , instead of nodes. This allows us to avoid propagation flow values from *every* predecessor of a node. Rather, we propagate only from those whose values have changed, and only then if it makes a difference. Asymptotically, there is no improvement over the node-workset approach, but it does reduce the overall number of \sqcup operations (at the cost of a corresponding increase in comparisons and while-loop iterations), and thus may offer some savings if \sqcup is disproportionately expensive.

Other refinements of the workset approach apply in more limited circumstances. If we are interested in the solution for only a small subset of the flow graph nodes, the analysis can be performed using only the nodes of interest and those nodes whose flow values can reach (and hence influence) them. The basic idea is very simple: we modify Alg. 2.5.3 so that the initialization of W adds only the entry node s . Solution algorithms of this kind are known as *local solvers*. Such solvers are uncommon in classical analyses, particularly in the application to program optimization; in this setting, whole-program solutions are almost always required. On the other hand, their development has proven useful in the analysis of logic programming languages [CH94, FS98, FS99, Jør94, VWL94] and some forms of program verification [OO90].

2.5.3 Aside: Correctness and Abstract Interpretation

None of the framework properties discussed here gives an assurance that the analysis is *correct*: *i.e.* that the analysis results are assertions about the possible states of a program at each control point, guaranteed to hold for every possible program execution. From another viewpoint, this is a requirement that an analysis be sound with respect to a program's formal semantics. While important, this consideration is completely decoupled from properties of the data flow framework, which concerns only the relationship of the value and function lattices to the flow graph model.

The most widely-used approach to correctness is that of *abstract interpretation*, originally due to Cousot and Cousot [CC77a, CC79b]. In the literature, this term denotes two related ideas. First is the theoretical technique for establishing that a static analysis is correct: *i.e.* that the abstract semantics of the analysis are a true abstraction of a program's concrete semantics. The other sense of the term refers to an analysis technique, whereby the abstract syntax tree of a program is interpreted with (1) an abstract value domain and (2) a nondeterministic execution of control constructs.

It is tempting to confuse the analysis technique with data flow analysis, particularly since the original formulation given in [CC77a] uses a flow graph model, as well. The difference is that an abstract interpretation simulates program execution *directly* on a program's AST. Unlike data flow analysis, there is no initial construction of a flow graph.

This has some advantages in space savings, and may be somewhat more intuitive than the use of an explicit flow graph construction. The cost is a smaller range of efficient solution techniques. Whether or not an *a priori* control representation is necessary, most of the efficient solvers in the flow analysis literature have yet to be transferred to the various abstract interpretation applications.⁶

⁶One exception is the local solver approach to workset algorithms. Also see Chen *et al.* [CHY95], which presents a variant of the round robin algorithm.

TABLE 2.2: Relationship of common user parameters to framework components

\mathcal{D}	flow value definitions/constructors/copy constructors equality test merge operator second merge operator
\mathcal{F}	the functions defined with the flow map the entry value
G	flow graph
(\mathcal{I})	flow map, initialization value

2.6 Data Flow Analysis Construction Kits

From the user's point of view, a toolkit provides an interface that corresponds to the key elements of any data flow analysis, which Knoop et al [KRS96] refer to as a "cookbook" approach. In practice, existing data flow toolkits organize the user specification of an analysis in terms of a generic "analyzer structure", which includes some or all of the following components: construction and access mechanisms for the *flow values*, an *equality test* (for detecting stabilization of analysis), a *merge operation* for combining flow values, *flow graph*, *flow map*, the *entry value* (associated with the flow graph's start node), an *initialization value*, the *direction* of the analysis, and (optionally) a second, *generalized merge operation*.

This structure is better understood as a practical realization of the generic view of data flow analysis presented above: the nine components listed are precisely the components of a data flow framework, instance, and global abstract semantics. The tabular description in Table 2.2 makes this clear.

Existing toolkits for this family are distinguished by the level of abstraction offered in the user interface for instantiation of the framework and instance components [Las04]. Those designed for integration with industrial-strength systems [CDG96, DC96, HMCCR93, KKKS96, TH92, ZME06] provide APIs which the user instantiates in a general-purpose programming language, while those that exist as free-standing systems [Mar98, Ven92, YH93] usually provide the user with high-level,

domain-specific languages for analyzer construction. Systems designed for specialized applications of flow analysis such as the optimizer generator Gospel [WS94, WS97] or the Cesar [OO90, OO92], FLAVERS [DCCN04], or ASTRÉE [CCF⁺05] verification tools, also provide high-level specification facilities.

2.6.1 Flow Values

If an analyzer toolkit has a sufficiently narrow application domain, the designer may forgo user specification of flow values altogether by restricting the available lattices to a small set of built-in definitions [FLY02, KKKS96, KRS96, OO90, DCCN04]. More general uses of the toolkit require a flexible means for the user to specify complex value lattices.

In general, there are two paths for the specification mechanism. On the one hand, a toolkit can require that the user implement all desired flow value lattices in some general-purpose programming language [TH92, HMCCR93, CDG96, DC96], which are then integrated with the generated analyzer. In this case the toolkit will provide a collection of interfaces that supports definition of the lattice elements and their elementary operations: comparison (\leq) and combination (\sqcup). The user must of course verify that any given implementation satisfies the necessary algebraic properties.

Alternately, a toolkit can provide a domain-specific language tailored to the design of nonstandard semantic domains [CCF⁺05, Mar98, NLM95, Ven92, YH93]. Such a language will typically come with a few built-in sets to account for the most obvious value types—integer, boolean, and so on. For most of these, there are a few common orderings (*true* > *false*, flat ordering on integers, etc.) that will also be built in. Also built-in will be several constructors, which produce new lattices from sets or from other lattices.

A further possibility is support for explicit construction of orderings on small sets [Mar98, NLM95]. For all but flat orderings, this takes the form of an enumeration of pairs, representing the “immediately less than” relation, from which the rest of the order can be derived. Here, as with general-purpose language implementations of a

lattice API, it is usually the user's responsibility to ensure that the specified partial order is indeed a lattice.

Finally, there are several flow value lattices with infinite ascending chains that are useful in some forms of analysis. Examples include some lattices for type estimation [Ten74], integer intervals [CC77a], and various real number approximations [Min01, Min04, Fer04, Fer05]. If a toolkit includes support for such structures, there must be additional facilities to guarantee convergence of analyses. This is usually in the form of support for upper/lower bound operators that will be used in widenings/narrowings [CC77a], and these are almost always user-defined [CDG96, Mar98].

An alternative—either in addition to or in lieu of widenings—is the inclusion of a *projection* operator for constructing a new lattice as the image of another under some projection function [YH93, Mar98]. Given the lattice $(S, \leq_S, \sqcup_S, \perp_S, \top_S)$ and projection $p : S \rightarrow S$ we construct the lattice $(p(S), \leq_S, \sqcup_S, p(\perp_S), p(\top_S))$. Even without infinite-height lattices, this feature offers a convenient means of redefining an analysis over a smaller flow value lattice, which allows the user to quickly trade off performance and accuracy. It has been described by Martin as a limited form of widening [Mar98], but more accurately, the projection function is (or should be) an *upper closure operator* on the original lattice—monotone, idempotent, and extensive ([CC79b], p.272).

2.6.2 Function Space

As with flow values, the design of user support for the specification of transfer functions begins with a decision on the specification mechanism. Here, the design task is a decision on the language of expressions for the transfer function bodies.

Again, the balance to be struck here is between expressiveness and various correctness and performance guarantees. In particular, we would like to ensure that evaluation of E is finite and that the transfer functions possess desirable order preservation properties of monotonicity or even distributivity. Precisely where to set this balance depends on the class of data flow problems we intend the toolkit to handle. As with flow value definition, existing approaches range from supplying only built-

in forms [KKKS96] to domain-specific languages of limited expressiveness [FLY02, Mar98, Ven92, YH93] to support for general-purpose programming languages [DC96, HMCCR93, LGC02, TH92].

The use of a general-purpose language offers the greatest flexibility, but many opportunities for correctness/performance guarantees are also lost. As complete languages, termination of the individual transfer functions is impossible to enforce. But so too is monotonicity, which is known to be undecidable even for a language as limited in expressive power as the relational calculus [AHV95]. Thus, any data flow toolkit that supports sufficiently expressive transfer functions must itself resort to static analysis techniques for approximate guarantees of such correctness properties. An interesting approach to this is given by Murawski and Yi [MY02], who define a type-and-effect system for approximate determination of monotonicity in a language similar to that used in System Z.

Regardless of the approach taken to support user specification, the program constructs that may be represented in the flow graph guide the functions that are defined. At a minimum, the user will need to define a transfer function corresponding to each possible program construct. Indeed, specification of the transfer functions in practice is almost always combined with the specification of the *flow map* itself.

One flow map component that is often user-driven is the specification of the flow value to be assigned to all non-entry nodes at initialization. This is usually the \perp element of the flow lattice, a choice that can safely be fixed by the toolkit designer. In general, however, any flow value known to be \leq the least fixed point solution will do [CP89].

An important side note here concerns the use of constant lattice values such as the *Gen* and *Kill* sets which are presumed to annotate flow graph nodes in the bitvector problems (Examples 2.4.1–2.4.4: although they are generally quite easy to compute, these values must come from somewhere. Depending on the specifics of the data flow problem and the flow graph model used, we can either encode them directly in the transfer functions or rely on a “pre-processing” analysis. Encoding them in the transfer functions is easier if the flow graph nodes represent individual statements

rather than basic blocks, or if the flow graph is automatically extracted from a user-supplied abstract syntax tree; these variants on the flow graph are discussed below.

A *complete* definition of the function space would have to include not only transfer functions corresponding to every interesting language construct, but also the identity function and the closures of each transfer function under composition and pointwise join (and perhaps meet). In practice, however, this further specification is not needed, unless the toolkit will support the generation of elimination-style solution algorithms, as discussed in 2.5 above. Only here is it important to guarantee that every element of the function space is defined, since an elimination algorithm may during flow graph reduction construct a new flow function by combining the functions associated with each edge that is collapsed.

In such case, it suffices to define the remainder of the function space implicitly [TH92], by supplying implementations of an identity function and of the operators \circ , \sqcup , and \star , corresponding to function composition, join, and closure:

$$\begin{aligned}(f \circ g)(x) &= f(g(x)) \\ (f \sqcup g)(x) &= f(x) \sqcup g(x) \\ (f^\star)(x) &= \bigsqcup_{k=0} f^k(x)\end{aligned}$$

As a very simple automatic approach, a toolkit could keep symbolic representations of the actual uses of the operators in defining new, complex functions. Functions built by various combinations of the operators can be represented internally by a graph structure that corresponds to an abstract syntax tree representation and can be executed by a simple interpreter. This is essentially how an elimination solver is built in an optimizing compiler—see Muchnick ([Muc97], 8.7.3), for example.

2.6.3 Flow Graph

User specification of the flow graph model will generally involve code to access the nodes and edges, along with some form of switch indicating the *direction* of the analysis. The user can also be required to supply access routines (successor,

predecessor, etc.) [TH92], although it is possible to build these into the toolkit directly [FLY02].

On the other hand, the toolkit can forgo flow graph construction entirely by interpreting the program's AST itself (*e.g.* [Ven92], [Mar98], [CCF⁺05], and also Rosen [Ros77]). Other than the use of abstract values and transfer functions, the principal difference from a “concrete” interpreter is in the determination of control flow. With either form of interpretation, the flow of control is determined dynamically by the syntactic structure and the current machine state—the main difference in the static analysis case is that interpretation of a branch statement requires that *both* branches be executed.

This approach likely offers better ease-of-use for the toolkit user, but it exacts a price in flexibility. Aside from the obvious differences in graph structure, a flow graph differs conceptually from an abstract syntax tree in that the control structure of the program is explicit. In the absence of knowledge beforehand of a program's control flow structure, many of the efficient generic solvers—round robin, Kennedy's iterative node listing solver [Ken75], and all elimination algorithms, for example—are much more difficult to implement, and such a toolkit is therefore limited in the choice of available algorithms (*ref.* Section 2.5.3). Further, there is no inherent relation between program analysis and data flow analysis [Ete04, KBC⁺99, NACO97, ZYL04], yet a toolkit is committed to such a view if the user can only supply source code for a problem instance.

2.6.4 Flow Map

Conceptually, this portion of the specification, along with the flow graph, is part of a “data flow instance,” and like the flow graph, the specification of the flow map consists of information sufficient to *generate and use* the flow map at run time. Practically, this involves the definition of an interpreter for the target language, but one that is defined over the abstract flow values, by specifying an association of one of the defined transfer functions with each possible program construct.

A finer point here is the question of whether transfer functions will be mapped to nodes or to edges. In the case of the unidirectional analysis problems assumed for this chapter, it does not make much difference: as mentioned in 2.4 above, one form can always be recovered from the other. In more advanced forms of data flow analysis, such as bidirectional analysis [KD94], it is useful to distinguish the two. An edge-map does sometimes offer a performance advantage in calculating flow value combination at a node entrance, since the new combined value is simply the old one merged with the new edge value: see [Mar98], for example.

Finally, although most of the existing toolkits separate its specification from that of the flow map, the chosen start value ι , which holds on entry to (*resp.* exit from) the program, is in fact a constant-valued member of the function space. Conceptually, the specification of this value, is thus a part of the flow map specification.

2.7 Advanced Forms

Classical data flow analysis is performed on sequential, imperative procedure bodies, either forward or backward (but not both), with neither procedure calls nor concurrent execution of other tasks. It therefore suffices to consider only one type of flow in the model of program execution. However, more advanced analysis forms relax some or all of these assumptions, calling for analyses in which we distinguish between these different flows.

2.7.1 Interprocedural Analysis

Many interprocedural analyses are based on a flow graph that distinguishes between ordinary intraprocedural flow, the flow from and back to the call site of a procedure (to model the binding of arguments and effects on the call stack), and the flow between a call site and corresponding return point (to restore local state) [AM95, SP81, Kno98].

Most of these approaches are derived from the paper by Sharir and Pnueli [SP81]. In that work, the authors extend the classical data flow framework to encompass

interprocedural analyses that are both *flow sensitive* (in that intraprocedural flow is considered as well) and *context sensitive* (in that the value at the entry node for a procedure is determined by the abstract state at the call site). They offer two approaches, one based on iterative techniques (“call string”), the other on elimination (“functional”).

An important insight underlying both the call string and functional approaches (and the extensions in [AM95, Kno98]) is that a sensitivity to the kind of edge on which information flows offers a provable increase in precision. For example, analysis can safely avoid consideration of any call/return flow that violates the call stack discipline. In [SP81], the authors rely on this sensitivity to formulate an ideal measure of precision, known as the *meet over valid paths* (MVP) solution. They show that this solution can be strictly better than *MOP*, present an MFP form that is satisfied by both the call string and functional approaches, and exhibit coincidence and safety theorems, analogous to those of Theorems 2.5.5 and 2.5.6.

Although call and flow-sensitive analysis provides better accuracy for interprocedural problems, it can be quite expensive. For the sake of performance on very large programs, several interprocedural approaches adopt coarser flow graph models for analyses that are relatively insensitive to context, intraprocedural flow, or both. A successful example of this approach is Callahan’s *program summary graph* [Cal88], which facilitates analyses with context sensitivity, but not flow sensitivity.

Unfortunately, Callahan’s approach does not fit neatly into the Sharir/Pnueli framework. In the first place, the program summary graph is a substantial departure from their more detailed flow graph model, employing only flow from call to return points, and a very limited form of intraprocedural flow that ignores everything except procedure entry, exit, call, and return points. Second, the analysis is in a *nonsingular* form [KD94]: it makes use of both the lattice \sqcup operator and also \sqcap (where it can safely be used to sharpen information). Rather than an algebraic framework, the various analyses in the paper are therefore presented as systems of flow equations, in the manner of the *AE* and *LV* analyses in Section 2.3 above. Callahan also provides a manual adaptation of the workset algorithm for these analyses.

2.7.2 Analysis of Concurrent Programs

There are several extensions of the classical flow analyses for compile-time optimization, adapted to various forms of concurrency [CK94, GS93, RS90, KSV96, LPM99]. However, most of the application effort in the concurrent setting has been directed toward verification, in particular the canonical problems of deadlock and race freedom [CKS90, DS91, DCCN04, LC91, MR91, PP91, TO80]. Other verification approaches also benefit from incorporation of flow analysis, largely for discovering opportunities for state space reduction. Here it is used for the approximate determination of instructions that cannot execute in parallel [MR93, Mer92, NA98, NAC99] or cannot interfere with each others' effects [Cor00, FBG03].

Because of the wide variation in the type of concurrency that may be adopted, there is little agreement on a standard flow graph model. Examples include the *synchronized control flow graph* [CS88], *event spanning graph* [RS90], *parallel flow graph* [GS93], *module interaction graph* [DS91], Masticola and Ryder's *sync graph* [MR90b] and *sync hypergraph* [MR91], and Dwyer and Clarke's *trace flow graph* [DC94]. In general, any such model must represent not only ordinary intraprocedural control flow, but also information regarding the possible communication between and interleavings of concurrent tasks: synchronization actions, cross-task control precedence (sometimes called "may immediately precede" edges), and so on.

As with flow graph models, there is a variety of data flow frameworks for the concurrent setting. Examples in the literature include Dwyer's "complete lattice" framework [Dwy95], Reif and Smolka [RS90], Knoop *et al.* [KSV96], and Masticola's *join-of-meets* framework [Mas93]. Of these, only Dwyer's framework has been realized in a general-purpose toolkit [DCCN04].

Almost all data flow frameworks for concurrency share in common a recognition that certain edge types can be used to refine the assertions derived from intraprocedural flow. For example, on any successor edges from a synchronization point, we can safely assume that *all* incoming arcs have been traversed. Consequently, the merge of information on incoming "wait" edges at a sync point can be done using the \sqcap operator instead of \sqcup (*e.g.* [Dwy95], pp.39–42), thereby sharpening information. Hence,

the overall solution is specified in a nonsingular form, in which both operators are used.

Although existing frameworks for concurrency offer substantial flexibility over the classical approach, they are nonetheless specialized in their presentation to one or another model of concurrency. Many new applications of data flow analysis to concurrent programs are therefore unable to leverage any particular framework. Instead, many of these analyses are given in terms of complex flow equations alone.

In general, these flow equations are much more intricate than those that comprise the usual MFP specification.⁷ However, this form of declarative specification is an easier development burden than the whole-cloth formulation of an algebraic framework, and some key properties (*e.g.* monotonic growth of the equations' right-hand sides) impose only a modest effort. Other properties (*e.g.* fastness or rapidity) remain unresolved, however. Worse, there is frequently a lack of efficient solution algorithms. Instead, many of these "flow equation" approaches are solved with a general fixpoint algorithm, à la Algorithm 2.5.1. Duesterwald/Soffa [DS91], Callahan *et al.* [CKS90], and [GS93] are illustrative of this approach.

2.7.3 Bidirectional Data Flow Analysis

In bidirectional problems, the data flow property holding at a given node depends on both the successors and predecessors of that node in the flow graph. The earliest such analysis in the literature appears in Tenenbaum's thesis as a part of his type estimation analysis for SETL [Ten74]. In practice, however, most of the interest in this technique arises from its application to problems of code motion, especially the partial redundancy elimination analysis of Morel and Renvoise [MR79]. This analysis unifies several traditional optimizations by composing a number of classical forms together. It is defined over an ordinary intraprocedural flow graph $G \equiv (N, E, s, e)$ by a set of intricate data flow equations consisting of

- local properties of expressions associated with each node x : $Antloc(x)$ (Gen_{v_B} , from the very-busy expressions analysis), and $Transp(x)$ ($Exp \cap \overline{Kill_{v_B}(x)}$)

⁷See Grunwald/Srinivasan [GS93] for a particularly dramatic example.

- the properties $AE_{in}(x)/AE_{out}(x)$ and $VB_{out}(x)/VB_{in}(x)$, along with the “partially available” expressions, $PAE_{in}(x)/PAE_{out}(x)$ (a may-analysis variant on available expressions)
- analyses to determine those expressions that can safely be placed at the beginning/end of x (PP_{in} / PP_{out}), those that can usefully be placed at the end of x ($INSERT$), and those that are redundant in x ($REDUND$)

The main equations are given in Fig. 2.7.

Khedker and Dhamdhere [KD94] extend the classical framework view to encompass both unidirectional and bidirectional analyses. Their framework accommodates *singular* data flow problems (*i.e.* those with a single confluence operator, as in Morel/Renvoise), but not nonsingular ones (*e.g.* the variant on MRA given in [Dha91]).

$$\begin{aligned}
 PP_{in}x &= \begin{cases} \emptyset & , \text{if } x = s \\ PAE_{in}(x) & , \text{otherwise} \\ \cap [[PP_{out}(x) \cap Transp(x)] \cup Antloc(x)] \\ \cap \bigcap_{w \in E^{-1}(x)} [AE_{out}(w) \cup PP_{out}(w)] \end{cases} \\
 PP_{out}(x) &= \begin{cases} \emptyset & , \text{if } x = e \\ \bigcap_{y \in E(x)} PP_{in}(y) & , \text{otherwise} \end{cases} \\
 INSERT(x) &= PPOUT(x) \cap \overline{AE_{out}(x)} \cap [\overline{PP_{in}(x)} \cup \overline{Transp(x)}] \\
 REDUND(x) &= PP_{in}(x) \cap Antloc(x)
 \end{aligned}$$

FIGURE 2.7: Morel-Renvoise Algorithm (modified version in [KD94])

In [KD94], both workset and round-robin algorithms are given to solve this extended MFP form, along with precise measures of the complexity, and a characterization of analyses for which decomposition into a unidirectional equivalent is feasible.

An extension of elimination solution techniques to the bidirectional case is given in [DP93], but it is only applicable to a limited class of these analyses.

2.8 Limitations of the Toolkit Approach

Flow graph models with these or other heterogeneous edge classes all belong to the family of *multisource* data flow problems. In each form, the abstract semantics arising from the framework involves not only consideration of the information represented by a flow value, but a sensitivity to the *kind* of source from which a value is received (*e.g.* control successor/predecessor, call site, synchronization point, etc.).

Historically, this classification of data flow problems has not received much notice. Where specialized algebraic frameworks are known for certain problem families, the theoretical foundations have been extended from classical analysis. For all three of the advanced problem families discussed above, generic solvers have been developed for one or more frameworks. For interprocedural and concurrent flow analysis, some frameworks have been realized as toolkits to support automatic generation.⁸

Indeed, the “cookbook” approach described for classical flow analysis-based tools remains an appealing feature of toolkits in all three of these families. With the global semantics and solution algorithms fixed in the generation tool, the user burden is essentially unchanged from the classical case [KRS96].

Unfortunately, none of these extensions are sufficiently expressive for the specification of general multisource data flow problems. The problem lies not just in a particular framework and its corresponding toolkit implementation, but in the framework-based approach itself. If we want to extend the benefits of efficient, executable specification to the general multisource case, we must reconsider the

⁸D. Dhamdhere reports the existence of a toolkit based on the theoretical work in [KD94]. However, this appears to be unavailable for research. See http://www.cse.iitb.ac.in/dmd/tech_transfer.html.

requirements of the cookbook for user specification, in order to identify a reasonably benign—yet sufficiently expressive—set of parametric ingredients.

2.8.1 Data Flow Equations and the MFP Specification

We find justification for this claim—along with the key insight behind the solution we develop in this dissertation—in the historical development of both classical and advanced analysis forms, prior to the discovery of a unifying data flow framework. Invariably, new forms of flow analysis have been specified according to the “flow equation” approach, from the earliest bit-vector analyses onward. From this vantage, the conceptual starting point for the development of data flow frameworks is the observation that the flow equations defining various analyses follow a common pattern. By the time of Kildall’s seminal paper [Kil73], there was already wide recognition of this fact (for example, see [AU73, Sch73]). Lacking a precise characterization, however, such a common form could serve only as a potential source of insight, not as a basis for the systematic transfer of theoretical properties and practical techniques.⁹

The framework view of data flow analysis provides a formalization of this pattern, specifying all problems within an analysis family in terms of a single flow equation. For classical analysis, this is the MFP equation, and in the pattern it represents lies an important philosophical point. Specifically, it represents the view that a flow analysis abstracts concrete program semantics in two parts: the individual instructions on the one hand, and program control flow on the other.

To illustrate, consider the two approaches to specification for the available expressions analysis. Using the “flow equation” approach, we would write

$$\begin{aligned}
 AE_{in}(x) &= \bigcap_{w \in Pred(x)} AE_{out}(w) \\
 AE_{out}(x) &= (AE_{in}(x) \setminus Kill_{AE}(x)) \cup Kill_{AE}(x)
 \end{aligned}$$

This form makes the twofold nature of the specification apparent. The equation

⁹As an illustrative example, two of the exercises in the last chapter of [AU73] require a manual adaptation of Ullman’s elimination algorithm for available expressions [Ull73] to other analysis problems.

for $AE_{out}(x)$ is defined only in terms of the AE_{in} property holding at x ($Kill_{AE}(x)$ and $Kill_{AE}(x)$ are constants); in particular, it makes no use of any element in the flow graph besides x itself. On the other hand, the definition for $AE_{in}(x)$ relies almost exclusively on flow graph structure, consisting of an expression to define some set of nodes in the flow graph ($Pred(x)$), an expression parameterized on elements of this set ($AE_{out}(w)$), and a description of how to combine the resulting set of values (\bigcap). The “behavior” is quite limited, consisting only of AE_{out} values.

We can therefore understand AE_{out} as defining the “instruction”, or “local” component of the specification, while AE_{in} defines the “flow”, or “global” element. This is even more apparent if we treat AE_{out} as a function to which AE_{in} is passed as an argument:

$$AE_{in}(x) = \bigcap_{w \in Pred(x)} AE_{out}(w, AE_{in}(x))$$

$$AE_{out}(x, v) = (v \setminus Kill_{AE}(x)) \cup Gen_{AE}(x)$$

From this version, the last abstractions to reach the framework specification are straightforward. AE_{out} is the transfer function assigned to each program point, or, using the taxonomy of Cousot and Cousot [CC77a], the *local* abstract semantics component of the analysis. The *global* component is AE_{in} , and in keeping with the framework approach, we do not need to focus on the specifics of its definition beyond knowing that the \sqcup operator is \bigcap and that the analysis is forward with respect to the flow graph: the global semantics specifies how the property holding at a given point can be guaranteed consistent with every point from which the program could transfer control, and in the classical setting, this is always the MFP equation [CC79b]. Rosen’s algebraic approach [Ros80] expresses the same idea in the requirement that, for every arc (x, y) with associated transfer function f_{xy} , the flow annotation $\sigma : N \rightarrow \mathcal{D}$ computed by some analysis must satisfy $f_{xy}(\sigma(x)) \geq_{\mathcal{D}} \sigma(y)$.

Implicit in these and other works that rely on the MFP equational form is the assumption that every arc in a flow graph constrains in the same way what can safely be deduced as a flow property: that a path in the flow graph models actual program execution ([Dwy95]). Because of this, the classical MFP formulation is insufficient to describe more advanced forms of flow analysis.

In fact, the arcs in a flow graph constitute only a set of *constraints* on a program's control flow, with the global abstract semantics defining the relationship of these constraints to the abstraction of individual instructions represented by the transfer functions. In classical data flow analysis, these constraints yield a structure that corresponds very closely to actual control flow semantics, and the accompanying MFP equation is therefore quite simple. In the general multisource setting, however, the correspondence is frequently more subtle.

To our knowledge, Dwyer's explicit observation of the flow path/execution assumption is the earliest in the literature, but the idea had already occurred in other forms: in the "meet-over-valid-paths" formulation of Sharir and Pnueli [SP81], for example, and in several works that apply data flow analysis to concurrent programs [DS91, GS93]. In each case, successful extensions of the classical framework are accompanied often by an adaptation of the MOP form as a precision benchmark, and invariably by an MFP form specialized to the edge labels assumed for the flow graph model (*e.g.* [KD94, Kno98, KSV96]).

Unfortunately, none of these extensions will do for the general setting. The problem is that both the classical and extended framework views of flow analysis rest on the assumption of a fixed universe of discourse from which the flow graph model is constructed. In other words, the possible types of information flow—*i.e.* the set of edge labels in a flow graph—are always fixed a priori. Naturally, this assumption is violated in the general multisource case, and as a consequence, existing data flow frameworks fail to capture more than a limited subset of these problems.

Consequently, it is still common to present new forms of flow analysis in terms of complex flow equations, particularly when the analysis employs a new understanding of the relationship between flow graph structure and program control flow.¹⁰ This is unobjectionable—when the assumptions about information flows are changed, we must also change the global semantic constraints to accommodate this shift—but it denies these advanced forms the power offered by a unified framework view.

¹⁰Examples include nearly all of the work described in the preceding three sections [CS88, DS91, GS93, Cal88], the first approaches to type estimation [Ten74, KU80], and the various forms of partial redundancy elimination [Dha91, MR79] and dead code elimination [KRS94].

In the absence of a better choice, these equations are solved with a general fixpoint algorithm, along the lines of Algorithm 2.5.1 (*e.g.* [Mas93]). In other cases (*e.g.* [Cal88]), there is an ad hoc solver that improves on the basic strategy (usually a variant on the workset heuristic). In most cases, the development includes proofs of key properties such as the existence of solutions and convergence of the solver, but these must be done manually, as there is no framework from which to derive them.

Because of the variability in the relation of flow graph structure to information flow, it is not obvious how to formulate problems as instances of an underlying algebraic framework, or even whether a given framework is suitable. Indeed, we are in something of a prisoner's dilemma, if we insist on retaining the standard four-part parametric approach to data flow analysis toolkits. Support for the development of new forms requires either that the new analysis be a member of an existing framework or that a new framework be developed. The first of these requirements places an artificial limit on discovery, while the second imposes a substantial additional burden on new development.

2.8.2 Toward Toolkit Support for Multisource Flow Analysis

Specification of general multisource data flow problems therefore requires the analysis designer to provide another component. In addition to the standard value lattice, transfer function, flow graph, and flow map components, we also require a definition of the global abstract semantics.

This leaves us with three tasks. First, we need a specification mechanism, in the form of a suitable, domain-specific language. Second, and concomitantly, we want the establishment of key properties of the specified analysis to be as easy as possible. Ideally, this comes from an underlying data flow framework that is sufficiently general to formulate such problems, and which is inferable from the global semantics. Finally, we need good solution algorithms.

We develop a suitable data flow framework in the next chapter, along with the beginnings of a language useful for specification. Development of this language is completed in Chapter IV. The problem of efficient solution is taken up in Chapter V.

CHAPTER III

FRAMEWORK SPECIFICATION

The primary contribution of this chapter and the next one is an extension of the traditional approach to user specification in a data flow analysis toolkit, in order to support arbitrary multisource data flow problems. Our approach is simple: since multisource problems vary according to the global semantic specification, we add to the traditional framework components a mechanism for expression of this semantics.

In this chapter, we develop a high-level language for direct encoding of problems as instances of a data flow framework. The framework we use, described in Section 3.1, is essentially Masticola's k -tuple formalism [Mas93, MMR95], although our version offers some extensions from the approach originally proposed. We present a language based on this framework in Section 3.2.

3.1 K -tuple Data Flow Frameworks

One approach to a broader unified view of multisource data flow problems is the *k-tuple frameworks* formalism of Masticola, Marlowe, and Ryder [Mas93, MMR95]. The basic idea is that, in order to solve a data flow problem over a lattice \mathcal{D} , with a flow graph model consisting of k edge types, we instead form a homogeneous cross product lattice \mathcal{D}^k . Elements of this lattice are k -tuples, which keep separate the values that propagate along a certain type of edge. Correspondingly, we extend the transfer functions so that they “select” the appropriate index along which to propagate.

In addition to adapting several key theoretical classifications to these frameworks, Masticola *et al.* exhibit k -tuple formulations of some well-known multisource analyses: the partial redundancy analysis of Morel and Renvoise [MR79] and several analyses over Callahan's program summary graph [Cal88]. We are further encouraged by the fact that very similar uses of cross-product lattice frameworks appear in earlier works to formulate other multisource problems (*c.f.* [DRZ92, Ten74]).

Here, we recall and extend the basic ideas from their work.

A (typed) flow graph is a directed graph $G \equiv (N, E)$. N is a set of nodes, and E is a set of triples in $N \times N \times i$, where $1 \leq i \leq k$ is one of k distinct edge labels. Similar to the case of a single-source flow graph, we write $E_i(x)$ to denote the successors of x along edges with label i and $E_i^{-1}(x)$ to denote the i -predecessors of x . Note that G may be a multigraph, although there is at most one triple (x, y, i) , for each i . Note, too, that we have omitted the distinguished entry/exit nodes. While it does not make much difference from a technical standpoint, there is less point to distinguishing these elements here, since the property of isolated entry/exit is relative to a particular edge type.

Definition 3.1.1 (*k*-Tuple Data Flow Frameworks). Let $(\mathcal{D}_i, \leq_i, \sqcup_i, \sqcap_i, \perp_i, \top_i)_{1 \leq i \leq k}$ be a collection of complete lattices, not necessarily distinct. A *k*-tuple data flow framework is a pair $(\vec{\mathcal{D}}, \mathcal{F})$, where

- $\vec{\mathcal{D}}$ is the cartesian product lattice $\mathcal{D}_1 \times \dots \times \mathcal{D}_k$
- $\mathcal{F} : \vec{\mathcal{D}} \rightarrow \vec{\mathcal{D}}$ is a set of total functions of the form

$$f(d_1, \dots, d_k) = (\perp_{\mathcal{D}_1}, \dots, \perp_{\mathcal{D}_{i-1}}, g(d_1, \dots, d_k), \perp_{\mathcal{D}_{i+1}}, \dots, \perp_{\mathcal{D}_k})$$

and satisfying

- $f_{id} \equiv (\lambda v. v) \in \mathcal{F}$
- $\forall f, g \in \mathcal{F}, f \circ g \equiv (\lambda v. f(g(v))) \in \mathcal{F}$
- $\forall f, g \in \mathcal{F}, f \sqcup g \equiv (\lambda v. (f(v)_1 \sqcup_1 g(v)_1, \dots, f(v)_k \sqcup_k g(v)_k)) \in \mathcal{F}$

As before, a data flow problem is a pair $(G, \llbracket \cdot \rrbracket)$, with G an edge-typed flow graph and $\llbracket \cdot \rrbracket : E \rightarrow \mathcal{F}$ a flow map.

The intuition here is that each index in a flow value of $\vec{\mathcal{D}}$ represents one of k possible kinds of information flow. In order to preserve information about the way in which an assertion flows through the graph model, these k properties are kept separate at each node. Each transfer function f_i acts as a selector, propagating the flow values transformed by $g : \vec{\mathcal{D}} \rightarrow \mathcal{D}_i$ along index i , with all other indices propagating the \perp -neutral values for their respective domains.

In data flow analysis, the earliest use of a cross-product lattice form appears in A. Tenenbaum's Ph.D. thesis [Ten74], where each flow value tuple index corresponds to one of the flow equations constituting his type estimation analysis. Dhamdhere *et al.* also use a cross-product framework with similar conventions for the indices in their formulation of the PRE analysis [DRZ92]. What distinguishes the k -tuple approach of Masticola *et al.* is their association of indices with *edge types* rather than direct representations of flow equations, which "captures the information at [node] y needed to compute the edge functions for edges entering or leaving y " ([MMR95], p. 792).

Our formulation of k -tuple frameworks differs from that of Masticola *et al.* in the allowable forms for $\vec{\mathcal{D}}$. Whereas they insist on cross-product frameworks in which only a single lattice is used for all k indices, we allow for heterogeneous cross-products, as well. The primary motivation for this arises from the observation made in [MMR95] (and elsewhere) that values may be combined at a node in different ways, \sqcup or \sqcap , according to the types of the incoming edges. Indeed, this is one of the principal motivations for separating the values that flow along different edge types. If we are to insist on the unified MFP form above, we are therefore constrained to loosen the definition of a k -tuple lattice value, so that each index is a value either from \mathcal{D} or its dual, \mathcal{D}^δ . Since we assume that each \mathcal{D} is a complete lattice, this does not introduce any new difficulties. We add the remaining flexibility on the grounds that the prohibition on heterogeneity is technically unnecessary, so long as we ensure that all operations combining values from different indices are semantically well-formed.

While an explicit *MFP* form for the global semantics is not given in [MMR95], it is not difficult to construct one, for both their version and our extended one:

Definition 3.1.2. The *minimal fixed point (MFP)* solution for a k -tuple data flow problem, $\vec{\sigma}_{MFP}$, is the (pointwise) smallest assignment $\vec{\sigma} : N \rightarrow \vec{\mathcal{D}}$, satisfying the inequality

$$\vec{\sigma}(y) \geq \left(\bigsqcup_{\substack{x \in \bigcup_{i \in 1..k} E_i^{-1}(y)}} [(x, y, i)](\vec{\sigma}(x)) \right) \sqcup \vec{\iota}_E(y) \quad (3.1)$$

As with the classical forms given in Figure 2.6, $\vec{\iota}_E : N \rightarrow \vec{\mathcal{D}}$ is a constant-valued function that can be used to initialize extremal nodes. In this case, however, it provides a k -tuple of such initializations:

$$\vec{\iota}_E(y) = (\iota_{E_1}(y), \dots, \iota_{E_k}(y))$$

A corresponding “MOP” form is less clear. In the classical case, the paths we consider are simply composed from the control flow graph’s successor relation (or its reverse, for backwards analyses). However, this is an overestimate in the case of multiple edge types, since it may be that flows along certain compositions are impossible. This insight is well known for certain special cases of multisource analysis. It is the insight behind Sharir and Pnueli’s “meet over valid paths” formulation [SP81], for example. Similarly, the impossibility of certain compositions of intraprocedural and synchronization flow forms part of Dwyer’s argument for his “complete lattice framework” [Dwy95].

For the general multisource case, we require first a way to reason about the possible flows (or “valid paths”, in Sharir and Pnueli’s terminology). The determination of such flows is the subject of Chapter 5.

3.1.1 Properties

Although the added complexity of a k -tuple formulation makes some details more involved, most of the classical properties carry over in a straightforward manner.

Theorem 3.1.1. Let $\vec{\mathcal{D}} = \mathcal{D}_1 \times \dots \times \mathcal{D}_k$ such that each \mathcal{D}_i has finite height h_i , and let \mathcal{F} be a monotone space. Let $(G \equiv (N, E), \square)$ be an instance of $(\vec{\mathcal{D}}, \mathcal{F})$, where for each

edge type $1 \leq i \leq k$, M_{o_i} is the largest number of i -edges in E that share a common sink. Algorithm 2.5.1 converges in $\mathcal{O}\left(\left(\sum_{i=1}^k M_{o_i}\right) |N|^2 \left(\sum_{i=1}^k h_i\right)\right)$ operations.

Proof. The $\sum_{i=1}^k h_i$ term is the height of $\vec{\mathcal{D}}$, while the largest number of predecessors of any node is bounded above by $\sum_{i=1}^k M_{o_i}$. Otherwise, the details are similar to the proof of Theorem 2.5.8. \square

Lemma 3.1.2. *A function*

$$f_i(\vec{d}) = (\perp_{\mathcal{D}_1}, \dots, \perp_{\mathcal{D}_{i-1}}, g(\vec{d}), \perp_{\mathcal{D}_{i+1}}, \dots, \perp_{\mathcal{D}_k})$$

- is monotone if and only if

$$\forall \vec{d}_1, \vec{d}_2 \in \vec{\mathcal{D}} : \vec{d}_1 \leq_{\vec{\mathcal{D}}} \vec{d}_2 \implies g(\vec{d}_1) \leq_{\mathcal{D}_i} g(\vec{d}_2)$$

- distributes over \sqcup if and only if

$$\forall \vec{d}_1, \vec{d}_2 \in \vec{\mathcal{D}} : g(\vec{d}_1 \sqcup_{\vec{\mathcal{D}}} \vec{d}_2) = g(\vec{d}_1) \sqcup_{\mathcal{D}_i} g(\vec{d}_2)$$

Proof. [MMR95], Lemmas 3.1.1 and 3.2.1. \square

Establishing either property for g proceeds along the same lines as for more traditional formulations, with one exception. Part of the original motivation for Masticola's k -tuple formulation was the need to account for flow graph nodes at which a property holds if it holds on *any* of the node's predecessors. In [MMR95], analyses of this sort are expressed as “join of meets” frameworks.¹ This is a special case of the nonsingular phenomenon discussed above (ref. 2.7), in which analysis makes use of both the \sqcap and \sqcup lattice operators to combine information at a node. It, too, reflects a situation in which we can deduce a *stronger* assertion than that which is propagated along any single edge.

The formulation involves transfer functions that begin by combining all k pieces of information in the “wrong” way:

¹“Meet of joins”, in our dual formulation.

$$f_i(d_1, \dots, d_k) = (\perp_{\mathcal{D}_1}, \dots, \perp_{\mathcal{D}_{i-1}}, g(d_1, \dots, d_k), \perp_{\mathcal{D}_{i+1}}, \dots, \perp_{\mathcal{D}_k})$$

where

$$g(d_1, \dots, d_k) = h(d_1 \sqcap \dots \sqcap d_k)$$

One of the more striking results of their work ([MMR95], Corollary 3.2.2) is a proof that such frameworks are almost never distributive. Specifically, they show that distributivity of f over \sqcup holds if and only if h is a constant-valued function.

Unfortunately, their theorem applies only to homogeneous k -tuple frameworks, and the proof they give applies only to cross-products of the lattice of boolean values (ref. Example 2.1.1). We prove a version of the negative half of their result here, which applies to both homogeneous and heterogeneous product lattices, of any type.

Definition 3.1.3. A function $g(v_1, \dots, v_k) = e$, with $k > 1$, is a *sharpening function* if e contains a subterm $v_j \sqcap v_{j'}$, $1 \leq j, j' \leq k$, and $j \neq j'$. If g is a sharpening function, then a *sharpening view of g* is a pair (h, e_s) , where $e_s = v_j \sqcap v_{j'}$ is a subterm in e and

$$g(v_1, \dots, v_k) = h(e_s)$$

Definition 3.1.4. A k -tuple framework $(\vec{\mathcal{D}}, \mathcal{F})$ is a *sharpening framework* if there is some $f_i \in \mathcal{F}$

$$f_i(d_1, \dots, d_k) = (\perp_{\mathcal{D}_1}, \dots, \perp_{\mathcal{D}_{i-1}}, g_i(d_1, \dots, d_k), \perp_{\mathcal{D}_{i+1}}, \dots, \perp_{\mathcal{D}_k})$$

such that $g_i : \vec{\mathcal{D}} \rightarrow \mathcal{D}_i$ is a sharpening function.

Observation. A k -tuple framework $(\mathcal{D}_1 \times \dots \times \mathcal{D}_k, \mathcal{F})$ is a sharpening framework only if there exist $1 \leq j, j' \leq k$, with $j \neq j'$, such that $\mathcal{D}_j = \mathcal{D}_{j'}$ (otherwise, the \sqcap operator cannot be applied to distinct elements of a k -tuple).

The following theorem (adapted from [MMR95], Cor. 3.2.2) tells us that if sharpened information is actually used in a function, it matters when we use it:

Theorem 3.1.3. *In a sharpening framework, a function f of the form given in Definition 3.1.4 distributes over \sqcup only if, for all sharpening views $(h, v_j \sqcap v_{j'})$ of g_i and for all $a, b \in \mathcal{D}_i$, $h(a \sqcap b) = h(a \sqcup b)$.*

Proof. We assume without loss of generality that $k = 2$, that $f((a, b)) = (h(a \sqcap b), \perp)$, and that $a \neq b$. Now suppose that $h(a \sqcap b) \neq h(a \sqcup b)$. Then

$$\begin{aligned}
 f((a, b)) \sqcup f((b, a)) &= (h(a \sqcap b), \perp) \sqcup (h(b \sqcap a), \perp) \\
 &= (h(a \sqcap b), \perp) \sqcup (h(a \sqcap b), \perp) \\
 &= (h(a \sqcap b) \sqcup h(a \sqcap b), \perp) \\
 &= (h(a \sqcap b), \perp) \\
 &\neq (h(a \sqcup b), \perp) \\
 &= (h((a \sqcup b) \sqcap (a \sqcup b)), \perp) \\
 &= f((a \sqcup b, a \sqcup b)) \\
 &= f((a \sqcup b, b \sqcup a)) \\
 &= f((a, b) \sqcup (b, a))
 \end{aligned}$$

□

3.2 Specification of Data Flow Framework Instances

Although the results in [MMR95] are promising, it is unclear whether specification of new data flow problems as k -tuple instances is a reasonable burden. The formulations given in [MMR95] are extremely complex and much less straightforward than the original flow equational forms. Comparable formulations likely require considerable expertise on the part of the designer.

Further, the presentation in [MMR95] does not offer an efficient generic solution algorithm for solving k -tuple problems. Although the authors observe that their unified view might suggest new domains for existing approaches, their focus throughout is explicitly of theoretical interest only. Indeed, they suggest that a general solver is unlikely to compete in efficiency with problem-specific algorithms ([MMR95], pp.779–80).

Nonetheless, this approach seems to offer the best promise for a successful toolkit. To investigate its potential, we therefore begin our language development with a language to support direct expression of data flow problems as instances of a k -tuple data flow framework.

3.2.1 Semantic Domain

Let $(\mathcal{D}_i, \leq_i, \sqcup_i, \sqcap_i, \perp_i, \top_i)_{i \geq 1}$ be a collection of complete lattices. We require that \leq_i , \sqcup_i , and \sqcap_i be computable. Unless stated explicitly, we require no further properties of any \mathcal{D}_i ; in particular, we do not assume that \mathcal{D}_i is either a complemented or distributive lattice, nor that it satisfies any finite chain properties. Where the context is clear, we drop the subscript, and write \mathcal{D} instead. To account for commonly-used structures, we write 2^S to abbreviate the power set lattice $(2^S, \subseteq, \cup, \cap, \emptyset, S)$, and for a lattice \mathcal{D} , we denote the dual lattice by \mathcal{D}^δ .

We let \mathcal{A} denote a finite set of atoms, and let $G \equiv (N, E)$ be a directed graph with node set $N \subseteq \mathcal{A}$ and edge set $E \subseteq N \times N \times i$ (with $1 \leq i \leq k$, for some k). In addition, we assume a set of finite maps of the form $\sigma : S \rightarrow \mathcal{D}$ (with $S \subseteq \mathcal{A}$), defined prior to the present analysis. We will work with the extension of each map to \mathcal{A} , defining $\sigma_{ext}(x)$, the “application” of σ to an atom x , as

$$\sigma_{ext}(x) = \begin{cases} d & , \text{if } (x, d) \in \sigma \\ \perp & , \text{otherwise} \end{cases} \quad (3.2)$$

For convenience, however, we will generally blur the distinction in names, dropping the “ext” subscript.

In the case where \mathcal{D} is the powerset lattice 2^S , we can also view σ as a binary relation on S . Then $\sigma(x) = \{y \mid (x, y) \in \sigma\}$ is simply the projection of x through σ , and we can in addition define the inverse projection $\sigma^{-1}(x) = \{w \mid (w, x) \in \sigma\}$. Examples of this include the k edge types in a flow graph,² and solutions to data flow problems defined over $\mathcal{D} = 2^S$.

²which yields the familiar interpretation of $E_i(x)$ and $E_i^{-1}(x)$ as the successor/predecessor sets of node x along edges with label i (c.f. [SS93])

3.2.2 Syntax

Let $\vec{\sigma}(y)$ be a k -tuple flow map, as defined by Inequality 3.1. Writing $\vec{\sigma}(y)_i$ for the i^{th} index of $\vec{\sigma}(y)$, we now observe the following:

Proposition 3.2.1. *Let $(x, y, i) \in E$, $1 \leq i \leq k$. Then*

$$\begin{aligned} \vec{\sigma}(y)_1 &\geq_1 \llbracket (x, y, i) \rrbracket (\vec{\sigma}(x))_1 \\ &\quad \vdots \\ \wedge \vec{\sigma}(y)_{i-1} &\geq_{i-1} \llbracket (x, y, i) \rrbracket (\vec{\sigma}(x))_{i-1} \\ \wedge \vec{\sigma}(y)_{i+1} &\geq_{i+1} \llbracket (x, y, i) \rrbracket (\vec{\sigma}(x))_{i+1} \\ &\quad \vdots \\ \wedge \vec{\sigma}(y)_k &\geq_k \llbracket (x, y, i) \rrbracket (\vec{\sigma}(x))_k \end{aligned}$$

Proof. $\llbracket (x, y, i) \rrbracket (\vec{\sigma}(x))_j = \perp_j$, for $j \neq i$. □

Corollary 3.2.2. *Let $y \in N$, and suppose that $\vec{\sigma}(y) \geq \vec{\iota}_E(y)$. Then for all $(x, y, i) \in E$,*

$$\vec{\sigma}(y) \not\geq \left(\llbracket (x, y, i) \rrbracket (\vec{\sigma}(x)) \right) \sqcup \vec{\iota}_E(y)$$

if and only if

$$\vec{\sigma}(y)_i \not\geq_i \llbracket (x, y, i) \rrbracket (\vec{\sigma}(x))_i$$

□

As a consequence, we can rewrite Inequality 3.1 as the system of parametric inequalities

$$\begin{aligned}
\sigma_1(y) &\geq \left(\bigsqcup_{x \in E_1^{-1}(y)} g_{xy_1}(\vec{\sigma}(x)) \right) \sqcup \iota_{E_1}(y) \\
&\quad \vdots \\
\sigma_i(y) &\geq \left(\bigsqcup_{x \in E_i^{-1}(y)} g_{xy_i}(\vec{\sigma}(x)) \right) \sqcup \iota_{E_i}(y) \\
&\quad \vdots \\
\sigma_k(y) &\geq \left(\bigsqcup_{x \in E_k^{-1}(y)} g_{xy_k}(\vec{\sigma}(x)) \right) \sqcup \iota_{E_k}(y)
\end{aligned} \tag{3.3}$$

where each g_{xy_i} is the function selected for index i by

$$[[(x, y, i)]](\vec{\sigma}(x)) = (\perp_{\mathcal{D}_1}, \dots, \perp_{\mathcal{D}_{i-1}}, g_{xy_i}(\vec{\sigma}(x)), \perp_{\mathcal{D}_{i+1}}, \dots, \perp_{\mathcal{D}_k})$$

This yields the overall form of our specification language, \mathcal{K} , whose syntax is given in Fig. 3.1. Here, we present only the syntactic core, omitting the various unary and binary operators defined on non-lattice data elements (denoted in the grammar by γ_1 and γ_2), along with such immaterial forms as the language constructs to define lattice domains, flow graphs, and various external maps. While these are interesting in their own right (*e.g.* [Mar98, NLM95, Ven92, YH93]), they distract from the present discussion.

The scoping rules are fairly straightforward. A function definition

$$\text{def } g(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau = e;$$

introduces bound variables x_1, \dots, x_n , whose scope is e . The scope of g extends from the point following its definition; in particular, it is impossible to define functions recursively.

By contrast, flow constraints may be individually or mutually recursive. Specifically, in a block of the form

```

for  $x$  in  $e$  def
   $\sigma_1(x) : \tau_1 \geq \dots$  ;
  :
   $\sigma_k(x) : \tau_k \geq \dots$  ;
end

```

the scope of each σ_i ($1 \leq i \leq k$) includes the right hand sides of all k constraints within the same block (and also forward from the end of the block).

The rule concerning the parametric variable x (e.g. $\sigma(x)$) does have one unusual feature. While x has the expected property of being bound in any given inequality, we also require that it be declared in a “domain initialization” expression (for x in e). Note that x cannot occur in e itself.

We place three additional non-syntactic limits on \mathcal{K} specifications. First, we forbid recursion within the “iterator” component of a flow constraint. For example, in a constraint

$$\sigma_i(x) : \tau \geq / \text{lub } w \text{ in } \hat{\sigma}'_i(x) : g(\dots) \text{ lub } \iota_E(x);$$

we require σ'_i to be defined prior to the block in which this constraint occurs. Likewise, ι_E , if this optional term is included, must be defined before any block in which it is used. This ensures that both the initialization and the flow graph model for a given constraint block remain fixed as the constraints are solved. Third, the “transfer function application” $g(\dots)$ is required to match a specific form:

$$\sigma_i(x) : \tau \geq / \text{lub } w \text{ in } \hat{\sigma}_i(x) : \underline{g(w, x, \sigma_1(w), \dots, \sigma_k(w))} \text{ lub } \iota_E(x);$$

for a block consisting of the definitions of $\sigma_1, \dots, \sigma_k$.

This last convention can (and should) be relaxed in a practical setting. It suffices to require that both w and x each appear as arguments to g at most once, that only expressions of the form $\sigma_i(w)$ ($1 \leq i \leq k$) be allowed as arguments otherwise, and that each $\sigma_i(w)$ appear as an argument to g at most once.

$$\begin{aligned}
\langle spec \rangle &::= [(\langle block \rangle \mid \langle fnddecl \rangle)]^+ \\
\langle block \rangle &::= \text{for } x \text{ in } \langle exp \rangle \text{ def } \langle ineqn \rangle^+ \text{ end} \\
\langle fnddecl \rangle &::= \text{def } g(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau = \langle exp \rangle; \\
\langle ineqn \rangle &::= \sigma(x) : \tau \geq / \text{lub } w \text{ in } \sim \sigma'(x) : g(\langle exp \rangle, \dots, \langle exp \rangle) [\text{lub } \iota_E(x)]^?; \\
&::= \sigma(x) : \tau \leq / \text{glb } w \text{ in } \sim \sigma'(x) : g(\langle exp \rangle, \dots, \langle exp \rangle) [\text{glb } \iota_E(x)]^?; \\
\langle exp \rangle &::= w, x, y, z, \dots && \text{(variables)} \\
& \mid \text{top} \mid \text{bot} \mid \langle const \rangle && \text{(constants)} \\
& \mid \langle id \rangle(\langle exp \rangle, \dots, \langle exp \rangle) && \text{(application)} \\
& \mid \sim \sigma(x) && (\sigma^{-1}(x)) \\
& \mid \langle exp \rangle \text{ lub } \langle exp \rangle \mid \langle exp \rangle \text{ glb } \langle exp \rangle && (\sqcup / \sqcap) \\
& \mid \sim \langle exp \rangle && \text{(complement)} \\
& \mid \langle exp \rangle - \langle exp \rangle && \text{(relative complement)} \\
& \mid \text{if } \langle exp \rangle \text{ then } \langle exp \rangle \text{ else } \langle exp \rangle && \text{(conditionals)} \\
& \mid \gamma_1(\langle exp \rangle) \mid \gamma_2(\langle exp \rangle, \langle exp \rangle) && \text{(other operators)}
\end{aligned}$$

FIGURE 3.1: Syntax of global semantic specifications as \mathcal{K} programs

Finally, we note that consideration of the dual “/glb” form is technically unnecessary. Since we work only with complete lattices, we can always regard a constraint

$$\sigma(x):\tau \leq /glb \ w \ \text{in} \ \sim\sigma(x): \ g(\langle exp \rangle, \dots, \langle exp \rangle) \ glb \ \iota_E(x);$$

as syntactic sugar for

$$\sigma(x):\tau^\delta \geq /lub \ w \ \text{in} \ \sim\sigma(x): \ g(\langle exp \rangle, \dots, \langle exp \rangle) \ lub \ \iota_E(x);$$

Consequently, we will not consider this form in the remainder of our discussion.

3.2.3 Behavior

The basic program unit is a block of k maps, of the form

```

for  $x$  in  $e$  def
   $\sigma_1(x):\tau_1 \geq /lub \ w \ \text{in} \ \sim\sigma_1'(x): \ g(w, x, \sigma_1(w), \dots, \sigma_k(w)) \ lub \ \iota_{E_1}(x);$ 
   $\vdots$ 
   $\sigma_k(x):\tau_k \geq /lub \ w \ \text{in} \ \sim\sigma_k'(x): \ g(w, x, \sigma_1(w), \dots, \sigma_k(w)) \ lub \ \iota_{E_k}(x);$ 
end

```

which corresponds to the form given by Inequality 3.3 above. Incorporation of the initialization ι_{E_i} term is optional: if this is omitted, we assume the default map $\lambda x.\perp$. This block defines the maps $\sigma_i : S \rightarrow \mathcal{D}_i$ ($1 \leq i \leq k$), where $S \subseteq \mathcal{A}$ is a set of atoms defined by evaluation of e . As with the GenSet language [FLY02] from which we borrow this form, the reader should avoid the intuition familiar to imperative languages; we do not simply iterate through the elements in S , and “execution” of the block does not correspond to an iterated sequence of assignments. Rather, once we have computed S , execution consists of solving each constraint σ_i , for each $x \in S$, to compute the least $\sigma_i(x)$ that is \geq_i the corresponding right-hand side. Equation blocks are solved in source code order.

We also support the definition of auxiliary transfer and utility functions

$$\text{def } g(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau = e;$$

For the most part, the behavior of expressions is self-explanatory, with the possible exception of the *flow expressions*—*i.e.* right hand sides of flow constraints

$$/\text{lub } w \text{ in } \sigma'(x): e$$

Here, we evaluate first the “index set” expression $\sigma'(x)$, the value of which is a set of atoms. We then iteratively bind w to each element of this set, and evaluate e with each binding. Results are collected using the \sqcup operator. The scope of w is the subexpression e .

\mathcal{K} is a statically-typed language, with explicit annotations required for the formal parameters (*e.g.* $x : \tau$) and return types of functions and for the left-hand sides of flow constraints.

Types consist of boolean values, the set `atom` of atoms, user-defined subsets of `atom`, and user-defined lattice datatypes, along with value tuples, maps, and powerset and dual lattices. These are summarized in the first part of Table 3.1. The type judgments produced by the proof rules are of the form $\Gamma \vdash \rho$, meaning that ρ holds given the judgments in Γ . The available type judgments are listed in the lower half of Table 3.1. Γ is often referred to as a “type environment,” although the usual sense of this term denotes a finite map from variables to types. Here Γ includes also any declarations and equality or subset constraints.

The type rules are presented in Figures 3.2 and 3.3, according to whether they concern the effect of declarations (*i.e.* judgments of the form $\Gamma \vdash D \vdash \rho$), equality of types ($\Gamma \vdash \tau_1 = \tau_2$), or value judgments ($\Gamma \vdash E : \tau$). The rules *Decl Fun* and *Decl Eqn-Blk* are standard, the latter resembling the usual typing rule for a mutually-recursive block of values. *Decl Seq* transforms a declaration into a piece of the environment. *Equiv Dual* expresses the fact that duality is its own inverse.

Most of the *Val* rules are straightforward. *Val \sim* and *Val \setminus* restrict the binary and unary \sim operators to powerset lattices as a safe approximation of the boolean

TABLE 3.1: Types and type judgments

<u>Types</u>	bool	$\{true, false\}$
	atom	Set \mathcal{A} of atoms
	$S_1, \dots, S_m \subseteq \text{atom}$	Other declared sets
	$\mathcal{D}_1, \dots, \mathcal{D}_n$	Declared lattice type identifiers
	$\mathcal{D}_i^\delta, 1 \leq i \leq n$	Dual of lattice \mathcal{D}_i
	$2^{S_i}, 1 \leq i \leq m$	Powerset lattice $(2^{S_i}, \subseteq, \cup, \cap, \emptyset, S_i)$
	$\tau_1 \times \dots \times \tau_p$	Tuples
	$\tau_1 \rightarrow \tau_2$	Maps
<u>Judgments</u>	$\Gamma \vdash \tau_1 = \tau_2$	Equality of types
	$\Gamma \vdash e : \tau$	e is a well-formed expression of type τ
	$\Gamma \vdash d : \rho$	d is a well-formed declaration of judgment ρ

lattices on which these operators must be defined (*ref.* Proposition 2.1.4). The rule *Val Subs* offers a limited form of the subsumption rule provided by subtyping.³ The rule for *Val Dual* expresses the useful fact that, since the identity map between the two structures forms a lattice dual isomorphism [Grä98], any element of a lattice is also an element of the dual structure, without loss of information regarding order.

On the other hand the *omission* of a rule for constant values is surprising. Usually, such a rule is given as

$$\frac{\text{TypeOf}(d) = \tau}{\Gamma \vdash d : \tau} \text{ (Val Const)}$$

where the function *TypeOf* assigns types to constants. In our setting, however, this will not work, since an element may belong to any number of partial orderings. Instead, we treat constants, along with the symbols \perp and \top , as constant-valued functions, overloaded according to the defined lattice types. Like the unary and binary operators, their types are inferred according to context.

³We do not need the rules stating the partial-order properties, since these are already associated with \subseteq .

$$\frac{x_1:\tau_1, \dots, x_p:\tau_p, \Gamma \vdash e:\tau}{\Gamma \vdash (\text{def } f(x_1:\tau_1, \dots, x_p:\tau_p):\tau = e) \therefore (f:\tau_1 \times \dots \times \tau_p \rightarrow \tau)} \text{ (Decl Fun)}$$

$$\frac{\begin{array}{l} \Gamma \vdash e_{init} : \text{atom} \\ x : \text{atom}, \sigma_1 : \text{atom} \rightarrow \tau_1, \dots, \sigma_k : \text{atom} \rightarrow \tau_k, \Gamma \vdash e_1:\tau_1 \\ \quad \vdots \\ x : \text{atom}, \sigma_1 : \text{atom} \rightarrow \tau_1, \dots, \sigma_k : \text{atom} \rightarrow \tau_k, \Gamma \vdash e_k:\tau_k \end{array}}{\Gamma \vdash \left(\begin{array}{l} \text{for } x \text{ in } e_{init} \\ \text{def} \\ \quad \sigma_1(x):\tau_1 \geq e_1; \\ \quad \vdots \\ \quad \sigma_k(x):\tau_k \geq e_k; \\ \text{end} \end{array} \right) \therefore (\sigma_1 : \text{atom} \rightarrow \tau_1, \dots, \sigma_k : \text{atom} \rightarrow \tau_k)} \text{ (Decl EqnBlk)}$$

$$\frac{\Gamma \vdash d_1 \therefore \rho_1 \quad \rho_1, \Gamma \vdash d_2 \therefore \rho_2}{\Gamma \vdash (d_1 \ d_2) \therefore (\rho_1, \rho_2)} \text{ (Decl Seq)} \quad \frac{}{\Gamma \vdash (\tau^\delta)^\delta = \tau} \text{ (Equiv Dual)}$$

FIGURE 3.2: Type rules for \mathcal{K} -definition, equivalence, and declaration rules

$$\begin{array}{c}
\frac{\Gamma \vdash e : 2^S \quad S \subseteq T}{\Gamma \vdash e : 2^T} \text{ (Val Subs)} \qquad \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 = \tau_2}{\Gamma \vdash e : \tau_2} \text{ (Val Equiv)} \\
\\
\frac{x \notin \Gamma_1}{\Gamma_1, x : \tau, \Gamma_2 \vdash x : \tau} \text{ (Val Var)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau^\delta} \text{ (Val Dual)} \\
\\
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 \text{ glb } e_2) : \tau} \text{ (Val } \sqcap) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 \text{ lub } e_2) : \tau} \text{ (Val } \sqcup) \\
\\
\frac{\Gamma \vdash e_1 : 2^{\text{atom}} \quad w : \text{atom}, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (/ \text{ lub } w \text{ in } e_1 : e_2) : \tau} \text{ (Val } \sqcup\text{-iter)} \\
\\
\frac{\Gamma \vdash e_1 : 2^{\text{atom}} \quad w : \text{atom}, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (/ \text{ glb } w \text{ in } e_1 : e_2) : \tau} \text{ (Val } \sqcap\text{-iter)} \\
\\
\frac{\Gamma \vdash e : 2^S}{\Gamma \vdash (- e) : 2^S} \text{ (Val -)} \qquad \frac{\Gamma \vdash e_1 : 2^S \quad \Gamma \vdash e_2 : 2^S}{\Gamma \vdash (e_1 \sim e_2) : 2^S} \text{ (Val Rel -)} \\
\\
\frac{\Gamma \vdash f : \tau_1 \times \dots \times \tau_n \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash f(e_1, \dots, e_n) : \tau'} \text{ (Val Appl)} \\
\\
\frac{\Gamma \vdash \sigma : \text{atom} \rightarrow 2^{\text{atom}} \quad \Gamma \vdash x : \text{atom}}{\Gamma \vdash \sim \sigma(x) : 2^{\text{atom}}} \text{ (Val Inv)} \\
\\
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau} \text{ (Val if)}
\end{array}$$

FIGURE 3.3: Type rules for \mathcal{K} -values

The *Val Inv* rule is also fairly subtle. Despite its appearance, this rule is not redundant with *Val Appl*, and we cannot make do with the more straightforward

$$\frac{x : S, \Gamma \vdash \sigma(x) : 2^A}{x : S, \Gamma \vdash \sigma^{-1}(x) : 2^A}$$

The problem is that this would allow us to type the expression $\ulcorner \sigma(x)$, whenever we could show that $\sigma : \mathcal{A} \rightarrow (2^A)^\delta$. We need to prevent this because of the requirement that $\ulcorner \sigma(x)$ evaluate to a set of atoms, the desire to view this set as inverse projection in a binary relation, and the interplay of this goal with Equation 3.2. Consider, for example, the power set lattice $\mathcal{D} = (2^{\{a,b,c\}}, \subseteq, \cup, \cap, \emptyset, \{a, b, c\})$, and suppose we have a map $\sigma : \{a, b, c\} \rightarrow \mathcal{D}^\delta$ defined as

$$\sigma = \{(a, \{b, c\})\}$$

The map $\sigma^\delta : \{a, b, c\} \rightarrow \mathcal{D}$, consisting of the identical pair, corresponds precisely to the directed graph



so that an expression such as $\ulcorner \sigma^\delta(b) = \{a\}$ gives the expected “predecessor” value. By Eqn. 3.2, however,

$$\sigma(b) = \sigma(c) = \perp_{\mathcal{D}^\delta} = \{a, b, c\}$$

which would give $\ulcorner \sigma(b) = \{a, b, c\}$.

3.2.4 Examples

Example 3.2.1. We can specify a *live variables* analysis as follows. We begin by considering the set $\text{progvars} \subseteq \text{atom}$ of variables in the program source code, and our value lattice will then be $\text{Vars} = 2^{\text{progvars}}$. The flow graph consists of a node set

N and has only one kind of edge, $\text{flow} : N \rightarrow 2^N$. We can recover N for flow by using the γ_1 operator $\text{base} : \text{atom} \rightarrow 2^{\text{atom}}$, defined as

$$\text{base}(E) = \{x \mid \exists y : (x, y) \in E \vee (y, x) \in E\}$$

We will use the reversed edge set, revFlow . Lastly, we will need to have available the maps $\text{isExit} : \text{atom} \rightarrow \text{bool}$, $\text{genV} : \text{atom} \rightarrow \text{Vars}$, and $\text{killV} : \text{atom} \rightarrow \text{Vars}$, and we assume these have been computed beforehand. We will use the default initialization of $\lambda x. \perp$.

The analysis itself is then specified by:

```
def revFlow(y:atom):(atom set) = ~flow(y);

def f(y:atom,x:atom,v:Vars):Vars = (v ~ killV(y)) lub genV(y);

for x in (base flow) def
  LV(x) >= /lub y in ~revFlow(x): f(y,x,LV(y));
end
```

A complete version is given in Appendix A.1.

Example 3.2.2. Callahan's *program summary graph* [Cal88] (*ref.* 2.7.1) is a control flow model for large programs that replaces the intraprocedural flow graphs of procedure bodies with compact summarizations. The nodes in a PSG represent four kinds of program points: procedure call/return and entry/exit. For every procedure call in a program, we add to the PSG a quadruple of each of these four node types, one for each variable involved in an actual/formal parameter binding. A PSG has two kinds of edges: those between call/entry and exit/return nodes (essentially the interprocedural parameter-passing edges used in the call string approach of Sharir and Pnueli [SP81]) and those between entry/call, entry/exit, return/call, and return/exit points (summarizing intraprocedural flow). While the interprocedural edges are determined purely by the call structure of a program, an intraprocedural summary edge (x, y) is added for a variable v only if the definition holding for v at x reaches y (the necessary reaching definitions analysis is performed for each procedure in order to build a PSG).

Among the analyses that Callahan defines on this graph structure is an interprocedural *Kill* analysis that determines the variables that *must* be modified by a

procedure call. This is a backwards, must-analysis. Using the lattice $false < true$, with $\sqcup = \vee$ and $\sqcap = \wedge$, we define $Kill(x)$ as the largest solution of the following equations ([Cal88], p. 50):

$$Kill(x) = \begin{cases} false & , \text{if } x \text{ is an exit node} \\ \prod_{(x,y) \in E} Kill(y) & , \text{if } x \text{ is an entry or return node} \\ Kill(y) \sqcup Kill(z) & , \text{if } x \text{ is a call node, } y \text{ the corresponding return} \\ & \text{node, and } z \text{ the corresponding entry node} \end{cases}$$

This analysis makes use (implicitly) of an additional source of information flow [MMR95]: that between a call node and its corresponding return (analogous to the interprocedural summary edges used in Sharir and Pnueli's "functional" approach). We will denote this additional edge type by E_{sum} . Moreover, the $Kill$ analysis does not distinguish between the interprocedural parameter-passing and intraprocedural summary edge types in a PSG, and thus we will simply denote these two by a single type, E .

With this expanded edge set, we can rewrite Callahan's equations in a more homogeneous form

$$Kill(x) = \left(\left(\prod_{(x,y) \in E} Kill(y) \right) \sqcap \iota_E(x) \right) \sqcup \left(\left(\prod_{(x,y) \in E_{sum}} Kill(y) \right) \sqcap \iota_{sum}(x) \right)$$

where

$$\iota_E(x) = \begin{cases} \perp & , \text{if } x \text{ is an exit node} \\ \top & , \text{otherwise} \end{cases}$$

$$\iota_{sum}(x) = \begin{cases} \top & , \text{if } x \text{ is a call node} \\ \perp & , \text{otherwise} \end{cases}$$

This version leads to a natural k -tuple, and hence \mathcal{K} , formulation. The flow graph consists of two edge types, $revPSG$ and $revSum$, and a node set $N = base(revPSG) \cup base(revSum)$. The $revPSG$ edges are exactly the PSG edges, with all directions reversed; $revSum$ corresponds to the E_{sum} edges, also with reversed direction. We

further assume the maps $\text{isExit} : \text{atom} \rightarrow \text{bool}$ and $\text{isCall} : \text{atom} \rightarrow \text{bool}$. Boolean is the lattice $\text{false} < \text{true}$. Because Callahan’s original analysis computes the *greatest* fixed point, we will use the matching, dual formulation here:

```

def i_psg(x:atom):Boolean = if (isExit(y)) then bot else top;
def i_sum(x:atom):Boolean = if (isCall(y)) then top else bot;
def g(u:ToKill,v:ToKill):Boolean = u lub v;

for x in ((base revPSG) | (base revSum)) def
  Kill_psg(x):Boolean <=
    ( /glb y in ^revPSG(x):
      g(Kill_psg(y),Kill_sum(y)) )
      glb i_psg(x);
  Kill_sum(x):Boolean <=
    ( /glb y in ^revSum(x):
      g(Kill_psg(y),Kill_sum(y)) )
      glb i_sum(x);
end

def Kill(x:atom):Boolean = Kill_psg(x) lub Kill_sum(x);

```

- ⌘ The final definition of `Kill` combines the values from each edge type that we have kept separate. This is what Masticola *et al.* call an “interpretation function” ([MMR95], p.784). See Appendix A.2 for a complete version.

Callahan’s analysis is one of the example k -tuple problems given in [MMR95], and our version is essentially a transliteration of their formulation. Since g is monotone, the function space induced by the closure of g under \sqcap and composition is monotone (Lemma 2.5.1). However, it is not distributive. To see this, observe that $g(u, v)$ is just the identity function applied to a (dual) sharpening of u and v , and apply Theorem 3.1.3.

Example 3.2.3. In Section 2.7.3, we presented the flow equations that Morel and Renvoise used to define their partial redundancy elimination analysis (Fig. 2.7). To formulate this as an \mathcal{K} specification, we need, in addition to the flow graph relation `flow`, the following:

- The relation `revFlow`, as in Example 3.2.1
- The domain `Exps`, which is the power set lattice 2^T , with `T` the set of temporaries used to store each expression value in the program
- The map `avin` : `atom` \rightarrow `Exps`, computed by an available expressions analysis
- The “gen” maps `antloc` : `atom` \rightarrow `Exps` (the available expressions *Gen* map) and `comp` : `atom` \rightarrow `Exps` (the very-busy expressions *Gen* map)
- The map `transp` : `atom` \rightarrow `Exps` (the *kill* map from *AE* and *VB*)

We will also define a may-analysis form of available expressions, in order to compute the expressions that are *partially available*:

```

def initF(x:atom):Exps = if (isEntry(x)) then bot else top;
def initB(x:atom):Exps = if (isExit(x)) then bot else top;

def avout(x:atom):Exps = (avin(x) glb transp(x)) lub comp(x);
def pavout(w:atom,x:atom,v:Exps):Exps =
    (v glb transp(w)) lub comp(w);
for x in (base flow) def
    pav_in(x):Exps >=
        (/lub w in ^flow(x): pavout(w,x,pav_in(w)) )
        lub initF(x);
end

```

The “profitable to place” component of the analysis is then:

```

def pp_out(w:atom,x:atom,vb:Exps,vf:Exps):Exps = vb;
def pp_in(y:atom,x:atom,vb:Exps,vf:Exps):Exps =
    pav_in(y) glb ( (pp_out(y,x,vb,vf) glb transp(y))
                    lub antloc(y) );
def f(w:atom,x:atom,vb:Exps,vf:Exps):Exps =
    pp_out(w,x,vb,vf) lub avout(w);

```

```

for x in (base flow) def
  pp_forw(x):Exps <=
    ( /glb w in ~flow(x):
      f(w,x,pp_back(w),pp_forw(w)) )
      glb initB(x);
  pp_back(x):Exps <=
    ( /glb y in ~revFlow(x):
      pp_in(y,x,pp_back(y),pp_forw(y)) )
      glb initF(y);
end

```

As with the Kill map at the end of our formulation of Callahan's analysis, the `insert` and `redund` maps serve as interpretation functions:

```

def insert(x:atom):Exps = pp_forw(x) glb ~avout(x)
                          glb ~(pp_back(x) glb transp(x)) ;
def redund(x:atom):Exps = pp_in(x) glb antloc(x) ;

```

Appendix A.4 gives the complete version.

Example 3.2.4. Among the advantages of allowing heterogeneous k -tuple frameworks is the possibility of expressing *non-singular* data flow problems, *i.e.* problems whose global abstract semantics involves both the \sqcap and \sqcup operators.

One example of this is Dhamdhere's modification of the PRE analysis to incorporate what he calls "edge placement" [Dha91]. The equational form is given in Figure 3.4.

The "profitable to place" component of the analysis is expressed as the following k -tuple problem:

```

def pp_out(w:atom,x:atom,vb:Exps,vf:Exps):Exps = vb;
def pp_in(y:atom,x:atom,vb:Exps,vf:Exps):Exps =
  pav_in(y) glb ( (pp_out(y,x,vb,vf) glb transp(y))
                  lub antloc(y) );
def g(w:atom,x:atom,vb:Exps,vf:Exps):Exps =
  (pp_in(w,x,vb,vf) ~ antloc(w)) lub avout(w);

```

$$\begin{aligned}
 PP_{in}x &= \begin{cases} \emptyset & , \text{if } x = s \\ PAE_{in}(x) & , \text{otherwise} \\ \cap [[PP_{out}(x) \cap Transp(x)] \cup Antloc(x)] \\ \cap \bigcup_{w \in E^{-1}(x)} [(PP_{in}(w) \setminus Antloc(w)) \cap AE_{out}(w)] \end{cases} \\
 PP_{out}(x) &= \begin{cases} \emptyset & , \text{if } x = e \\ \bigcap_{y \in E(x)} PP_{in}(y) & , \text{otherwise} \end{cases}
 \end{aligned}$$

FIGURE 3.4: Modified Morel-Renvoise Algorithm [Dha91]

```

for x in (base flow) def
  pp_forw(x):Exps >=
    ( /lub w in ~flow(x):
      g(w,x,pp_back(w),pp_forw(w)) )
    glb initB(x);
  pp_back(x):Exps <=
    ( /glb y in ~revFlow(x):
      pp_in(y,x,pp_back(y),pp_forw(y)) )
    glb initF(y);
end

```

In this formulation, the map (PP_{forw}, PP_{back}) is monotone, and the accompanying transfer functions are, in fact, distributive. However, we can apply Theorem 3.1.3 to see that (PP_{forw}, PP_{back}) is itself not distributive.

The syntactic differences between this and the formulation of PRE are small, but they are nonetheless important. Note in particular that for PP_{forw} , we require the *least* solution to the constraints, while PP_{back} uses the greatest. This gives us a value lattice of $\text{Exps} \times \text{Exps}^\delta$, which is not expressible in Masticola's original formulation.

3.3 Limitations of the K -tuple Approach

The family of k -tuple frameworks extends the classical unified view to many if not all multisource data flow problems. This has a clear theoretical utility. For example, Masticola *et al.* were able to show boundedness and precision properties for a number of significant analyses from the literature. With our extension of the approach to heterogeneous frameworks, we can capture an even larger class of analyses, including some previously unaccounted-for, nonsingular forms.

As a practical specification mechanism, however, k -tuple frameworks have some notable disadvantages. Perhaps the most obvious one is the obscurity of k -tuple formulations, as represented by our \mathcal{K} language. The language's constrained syntax requires that many analyses with a natural declarative specification be given in a form that is artificially homogenized, for the sole purpose of facilitating a direct algebraic formulation. While useful for reasoning about the specification, the cost is an added layer of obfuscation to confound the analysis designer. This is illustrated by Examples 3.2.2 and 3.2.3, in which the corresponding k -tuple formulations bear little resemblance to the original, rather straightforward, flow equations.

A more substantial shortcoming lies in the manner in which a k -tuple framework can model information flow for a family of multisource flow graphs. Masticola's original formulation requires that "information must always propagate across some explicit edge" ([MMR95], p.782), and a lattice is constructed so that "[e]ach position in a k -tuple in $\{\mathcal{D}^k\}$ represents propagation from one source, across one edge class" (p. 783).

The problem with this formulation is that the correspondence between flow graph edges and information flow is not always direct. Indeed, part of the role of the global abstract semantics is to define this correspondence. Thus, a correct accounting of information flow in a flow graph requires consideration of both edge type and the context in which the edge appears in the global semantics. In the PRE analysis of Figure 2.7, for example, information flows both forwards and backwards with respect to the direction of the flow graph arcs.

We can instead use the positions in a k -tuple to represent propagation across the different *kinds of information flow*, rather than across explicit edge classes. In fact, this is a complementary approach to Masticola's own solution; in [MMR95], the correspondence between edge types and information flows is handled by constructing a new flow graph from the original, with new edge classes added to correspond to each form of information flow. In the PRE analysis, this means we add a second class of edges equal to those of the original, with directions reversed, thus doubling the size of the edge set. This is reflected above in the use of the `revFlow` relation, Examples 3.2.3 and 3.2.4.

Ultimately, this is merely a matter of taste. In the examples of the present chapter, we have adopted Masticola's approach, but in general, we would like to avoid the increase in model size that arises from explicit representation of the information flows. However, we are then left with the burden of specifying in the global abstract semantics exactly what those flows are. In addition, consideration of practical solution algorithms will need to include a mechanism for extracting the structure of this information flow from the original flow graph model and global semantic specification, a topic we address in Chapter V.

Regardless, both approaches are limited by the assumption of a fixed relationship between flow graph structure and information flow. To see how this might fail, let us consider an extended example.

Example 3.3.1. In [DS91], Duesterwald and Soffa present a technique for determining a partial execution order of program regions. The technique consists of a data flow analysis over a flow graph that models (and distinguishes between) intraprocedural and interprocedural control flow, along with inter-task synchronization. It is essentially a polynomial-time approximation of the set of all pairs of statements that may happen in parallel [Tay83]. The language model is that of explicit concurrency, in the style of Ada tasks, with a rendezvous synchronization mechanism and remote procedure call. Hence the interaction statements consist of procedure calls, task entry calls and *accept* statements, and procedure or task entry / exit.

Using a set of nodes N , we model a program in this language with a special form of flow graph, called a *module interaction graph* (MIG). A MIG is a triple (U, E_{syn}, E_{call}) , where $E_{syn}, E_{call} \subseteq N \times N$, and $E_{syn} \cap E_{call} = \emptyset$. U is a set of directed graphs of the form (M, E_{cfg}, s, e) , with $M \subseteq N$ and $E_{cfg} \subseteq (N \times N) \setminus (E_{syn} \cup E_{call})$. For each digraph, $s \in M$ is the unique start node for which there is no edge $(n, s) \in E_{cfg}$ and such that every node in M is reachable from s ; likewise $e \in M$ is the exit node for which there is no edge $(e, n) \in E_{cfg}$ and such that e is reachable from every node in M . We require that for any two $(M_1, E_{1cfg}, s_1, e_1)$ and $(M_2, E_{2cfg}, s_2, e_2)$,

$$M_1 \cap M_2 = E_{1cfg} \cap E_{2cfg} = \emptyset$$

and that

$$\bigcup \{M \mid (M, E_{cfg}, s, e) \in U\} = N$$

The three edge types represent ordinary control flow (E_{cfg}), synchronization (E_{syn}), and procedure/task call and return (E_{call}). The latter edge class is an elaboration on the model given in [DS91], where only E_{cfg} and E_{syn} are used. However, their analysis makes use of the information flow arising from call/return structure, and our formulation of this imposes no conceptual changes.

The following code, originally from [DS91], illustrates the construction:

```

task Main;                task T;                proc P(v);
  while (y <> 0) do        read(y);                if v = 0 then
    read(y);              accept Q;                v := 1;
    x := x+1;             P(y);                    else
  enddo;                  v := 2;
  create T;
  T.Q;
  P(y);

```

The corresponding MIG is reproduced in Fig. 3.5.

For this analysis, we represent the three edge types with the relations `cfg`, `sync`, and `callflow`, which have the expected meanings. We will also need the maps `isEntry : atom → bool`, `isExit : atom → bool`, and `unit : atom → (atom)` (the latter maps each node to the task or procedure that contains it). We use the lattice `Nodes`, which is the powerset lattice over the nodes in `cfg` (*i.e.* `base cfg`).

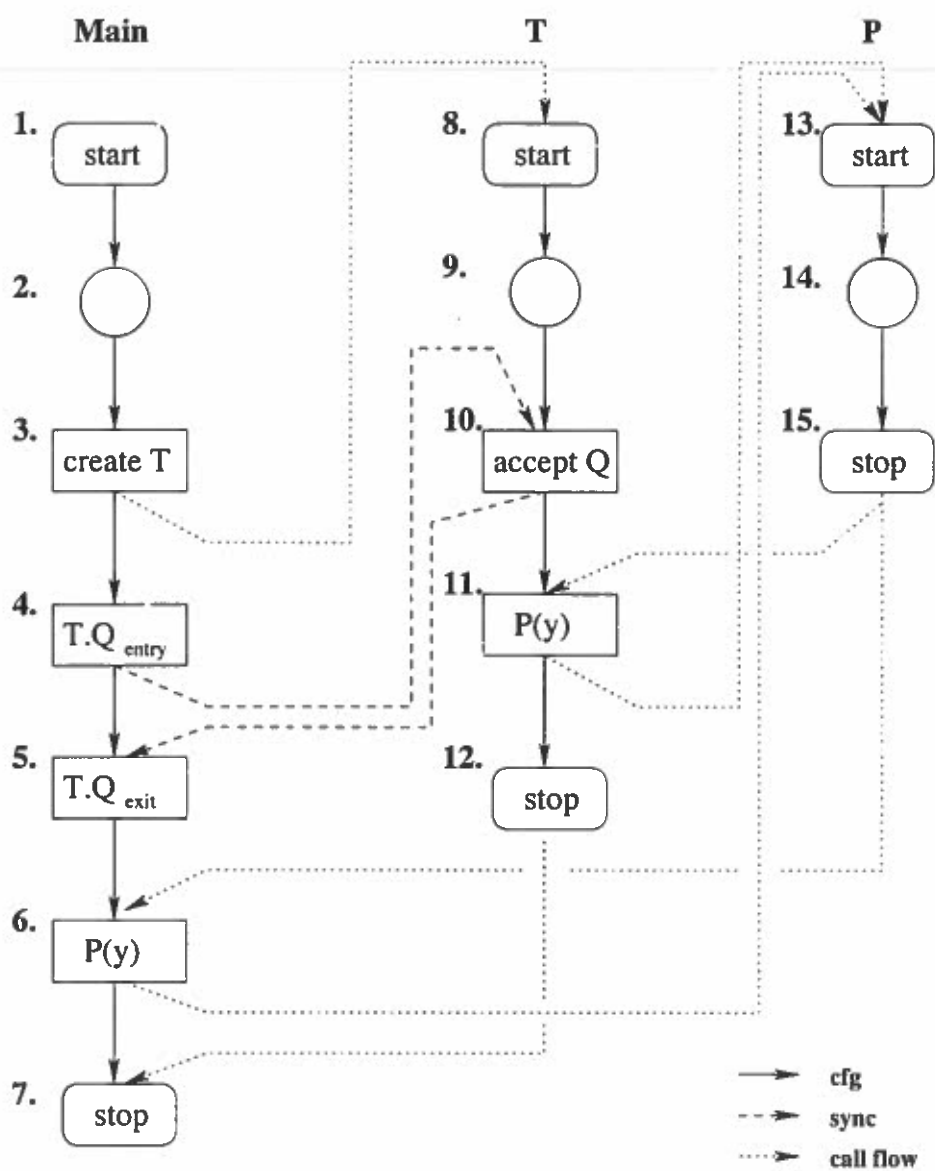


FIGURE 3.5: Example MIG, taken from [DS91], p. 39.

The analysis to determine whether two regions are ordered—*i.e.* whether it is impossible that they can execute concurrently—occurs in two phases. The first phase proceeds on the assumption that no task or procedure can execute concurrently with itself. The goal of this phase is to compute two binary relations, *before* and *after*, such that $x \in \text{before}(y)$ if and only if every instance of x executes before every instance of y , and inversely for $y \in \text{after}(x)$.⁴ In the second phase, the information represented by the *before* / *after* relations is combined with information about those tasks and procedures that can execute in parallel with themselves, in order to complete the overall partial ordering of execution.

We will forgo the task of formulating the various components of this analysis as instances of a k -tuple data flow framework. For the most part, the translation is straightforward (see [Las05] for a sketch).

Instead, it is the ‘mutual update’ phase as realized in the analysis’ *before* / *after* component that interests us here. Unlike other parts of the analysis, it is not at all clear how to formulate this component as an instance of any framework. In [DS91], it is given in a procedural form rather than as a flow analysis, but the corresponding \mathcal{K} form is quite simple:

```
def fBef(v:Nodes):Nodes = v;
def fAft(x:atom,v:Nodes):Nodes = y lub set{x};

for x in (base cfg) def
  before(x):Nodes >=
    (/lub w in ^after(x): fBef(before(w))) lub before_glob(x);
  after(x):Nodes >=
    (/lub y in ^before(x): fAft(y,after(y))) lub after_glob(x);
end
```

Unfortunately, this cannot be a real \mathcal{K} specification, since the flow constraints violate the rule against recursion in the index set expressions:

```
before(x):Nodes >= ... /lub w in ^after(x): ... ;
```

⁴In fact, *before* and *after* are relational inverses of each other, so we could make do with only one of them. Nonetheless, we will follow Duesterwald and Soffa’s formulation here.

More generally, this analysis represents the removal of an assumption that is often considered basic to data flow analysis: that the flow graph, once determined, remains fixed throughout the analysis. In essence, this analysis works by rewriting on the fly the model that we are annotating, and using those changes to direct the remainder of the analysis.

That idea is not wholly new, of course, although we are unaware of any approach in which the rewriting is encoded directly as a part of the analysis' global abstract semantics. There have been several works in the literature that incorporate information about the flow graph in the flow values themselves, collectively known as "qualified" analyses [HR81, SP81] (and also *c.f.* Cousot [CC79b]). Other works have investigated the formulation of flow analyses as restricted graph rewriting problems [Aßm00, LGC02, Rep98]. In fact, we suspect that these approaches are closely related to each other, particularly as presented in [HR81] and [LGC02].

The significance to future research is that these approaches can offer performance improvements to flow analysis solvers, by pruning away irrelevant parts of the flow graph model. However, the possibility of a dynamically-changing program model complicates static reasoning about the correctness of an analysis. It also complicates our efforts at efficient solution, a matter that we will revisit in Chapter V. Finally, we must address the practical task of toolkit-based specification, as we will need to relax the requirement of modeling only flow graph edge types in the value lattice. This is the subject of the next chapter.

CHAPTER IV

SPECIFICATION OF MULTISOURCE PROBLEMS

Although it has proven useful as a mechanism for reasoning about analysis, most of the existing data flow frameworks are too limited to serve as the basis for specification of general multisource analyses. The restrictions placed on the global abstract semantics make them insufficiently expressive to capture the full range of interesting problems. As we discussed in Section 3.3, this is true even for Masticola's k -tuple frameworks and the minor extensions developed in the previous chapter. Notwithstanding this objection, we are still faced with the fact that specifications constrained to a strict k -tuple form are often quite abstruse.

For the specification of multisource analyses, it is therefore best to abandon entirely the traditional toolkit approach that follows closely the components of an underlying data flow framework. History suggests instead that a more natural choice for specification lies in a more general form of query language; this is, after all, the form most commonly favored in the literature in the presentation of new flow analysis forms.

In this chapter, we develop an approach to the specification of multisource flow analyses based on a language that facilitates a direct translation from flow equations common in the literature to working, executable specifications. Unlike other toolkit approaches, we do not consider separate definitions of the function space \mathcal{F} or the

local semantic functional $\llbracket \cdot \rrbracket$. While the definition of auxiliary transfer functions is useful (and supported in our language), both \mathcal{F} , $\llbracket \cdot \rrbracket$, and global abstract semantics remain only implicit in a specification, just as they do in those analyses presented in terms of complex flow equations.

This approach to toolkit specification does have its drawbacks. First of all, it complicates the task of reasoning about the analysis. However, the tools for that are now well-established, and it should be possible to apply them as needed even to analyses of the form given in this chapter. A more substantial objection is the family of generic solution algorithms made available by an underlying data flow framework. This latter point will be addressed in Chapter V.

4.1 A Flow Equation Language

Our language, Roke, is a strict superset of \mathcal{K} . As before, we view the specification of a data-flow problem as a system of simultaneous, parametric equations, drawn from a language of first-order, recursive queries, but we now allow more flexibility of form.

The difference in syntax consists of two changes. First, we replace the definition of $\langle \text{ineqn} \rangle$ in the grammar of Figure 3.1 with the following productions

$$\begin{aligned} \langle \text{ineqn} \rangle &::= \sigma(x) : \tau \langle \text{con} \rangle e; \\ \langle \text{con} \rangle &::= \leq = \mid \geq = \end{aligned}$$

In addition, we add to the definition of expressions the productions

$$\begin{aligned} e &::= \dots \\ &\quad \mid / \text{lub } w \text{ in } e : e \\ &\quad \mid / \text{glb } w \text{ in } e : e \end{aligned}$$

As before, we specify a data flow analysis problem as a block of recursive constraints, each of the form

$$\sigma(x) : \tau \diamond e;$$

in which τ indicates the type of e , where \diamond is either \geq or \leq , depending on whether we want the least or greatest solution to the constraint (respectively).

This gives us a specification language that is essentially \mathcal{K} but for three primary differences:

- The right-hand side of a constraint can be any expression.
- We drop all of the restrictions against recursion within a block of flow constraints: a variable that is the left-hand side of a flow constraint may be used anywhere in the right hand side of any constraint within the same block. Recursion is still disallowed outside of `for`-blocks.
- *Flow expressions*—i.e. the iterated `/lub` and `/glb` terms—have more flexible syntax. Both forms can occur in arbitrary expressions, and both can be iterated over arbitrary expressions, not just those of the form $\sim\sigma(x)$.

Happily, the type rules given previously (Figs. 3.2 and 3.3) apply as well to our more flexible specification language; no modification is necessary. However, with the greater flexibility of flow expressions, we will need two additional non-syntactic limits. First, in either of the flow expressions

$$\text{/lub } w \text{ in } e_1 : e_2$$

$$\text{/glb } w \text{ in } e_1 : e_2$$

we allow e_2 to be any expression *except* another flow expression. In a similar spirit, a function definition

$$\text{def } f(x_1, \dots, x_n) : \tau = e;$$

cannot contain a flow expression.¹ The reason for these restrictions is discussed in Section 4.3, below.

¹It is also possible to embed these restrictions in the syntax, but that requires a duplication of the definitional clauses for expressions, in order to distinguish legal and illegal contexts for the iterated forms.

4.2 Examples

Example 4.2.1. The k -tuple formulation of Morel and Renvoise's PRE analysis from Example 3.2.3 is essentially the one given in Masticola's paper. With this version, they are able to prove easily both the monotonicity and 3-boundedness (but not 2-boundedness) of the function space. Unfortunately, the translation used to derive this version from Morel and Renvoise's original flow equations is somewhat opaque. Indeed, it is not even obvious that the two formulations are equivalent.²

Our full language makes possible a more straightforward version (see also Appendix A.3):

```

for x in (base flow) def
  pp_in(x):Exps <=
    if (isEntry(x) then bot
    else ( pav_in(x)
          glb (antloc(x) lub
              (pp_out(x) glb transp(y) ) )
          glb
          /glb w in ~flow(x): (pp_out(w) lub avout(w))
        );
  pp_out(x):Exps <= if (isExit(x) then bot
                    else /glb y in flow(x): pp_in(y) ;
end

for x in (base flow) def
  insert(x):Exps >= pp_forw(x) glb ~avout(x)
                  glb ~(pp_out(x) glb transp(x));
  redund(x):Exps >= pp_in(x) glb antloc(x);
end

```

Example 4.2.2. We conclude this section by reconsidering the Duesterwald/Soffa analysis, discussed previously in Example 3.3.1. This serves to illustrate how a complete, complex flow analysis can be turned into an executable specification, using the approach developed in this chapter.

²In fact, Khedker and Dhamdhere claim that they are not equivalent [KD99], although this is given only as a passing comment, without evidence.

Part 1.1: Local ordering ([DS91], Eqns. 1 & 2). This part computes the ordering information available from control flow alone. It consists of transitive closures over the *cfg* edges and their reverse, which are used as filters in determining the local *before/after* information. We shall also find useful the relation $gen(x)$, which is $\{x\}$ if x does not occur on a loop, and \emptyset otherwise. Our implementation of this part is the following pair of equation blocks:

```

for x in (base cfg) def
  may_before(x):Nodes >=
    /lub w in ^cfg(x): (may_before(w) lub set{w});
  may_after(x):Nodes >=
    /lub y in cfg(x): (may_after(y) lub set{y});
end

for x in (base cfg) def
  before_loc(x):Nodes >= may_before(x) ~ may_after(x);
  after_loc(x):Nodes >= may_after(x) ~ may_before(x);

  gen(x):Nodes >= set{x} ~ may_after(x);
end

```

Part 1.2: Synchronization ordering ([DS91], Eqn. 3). In this next step, we propagate the local ordering information across synchronization edges in the model, obtaining the relations $before_{sync}$ and $after_{sync}$:

```

for x in (base cfg) def
  C_before_sync(x):Nodes <=
    if empty (^cfg(x)) then bot
    else /glb w in ^cfg(x):
      (C_before_sync(w) lub S_before_sync(w));
  S_before_sync(x):Nodes <=
    if empty (^sync(x)) then bot
    else /glb w in ^sync(x):
      ((C_before_sync(w) lub S_before_sync(w))
       lub (before_loc(w) lub gen(w))
      ) ;
end

```

```

for x in (base cfg) def
  C_after_sync(x):Nodes <=
    if empty (cfg(x)) then bot
    else /glb w in ^cfg(x):
      (C_after_sync(w) lub S_after_sync(w));
  S_after_sync(x):Nodes <=
    if empty (sync(x)) then bot
    else /glb w in ^sync(x):
      ((C_after_sync(w) lub S_after_sync(w))
       lub (after_loc(w) lub gen(w))
      ) ;
end

```

```

for x in (base cfg) def
  after_sync(x):Nodes >=
    (C_after_sync(x) lub S_after_sync(x))
    ~ after_loc(x);
  before_sync(x):Nodes >=
    (C_before_sync(x) lub S_before_sync(x))
    ~ before_loc(x);
end

```

Part 1.2–3: Activation contexts / global propagation ([DS91], Eqn. 4).

Once we have the *before* / *after* information for both *cfg* and *sync* edges, we can combine them to obtain global versions. Concomitantly, we must also consider the ordering information that flow to (resp. from) the entry (resp. exit) points of each unit. In [DS91], these are presented as (respectively) the final part of step 1.2 and the initialization phase of step 1.3. In our implementation, we combine them into a single initialization step for the activation context analysis:

```

for x in (base cfg) def
  before_glob(x):Nodes <=
    if (isEntry(x) and not (empty callflow(x)) )
    then
      /glb w in ^callflow(x):
        (before_glob(w) lub gen(w))
    else
      before_loc(x) lub before_sync(x) ;
  after_glob(x):Nodes <=
    if (isExit(x) and not (empty callflow(x)) )
    then
      /glb w in callflow(x): after_glob(w)
    else
      after_loc(x) lub after_sync(x) ;
end

```

The rather complicated-looking test in both conditionals simply checks whether x is a *start* (resp. *stop*) node of some unit that is actually called during execution.

Following this initialization, the “mutual update” phase propagates the *before* and *after* information that holds on unit entry/exit into the unit bodies and across *callflow* edges (ref. p.90):

```

for x in (base cfg) def
  before(x):Nodes >=
    before_glob(x) lub (/lub w in ^after(x): before(w));

  after(x):Nodes >=
    after_glob(x)
    lub (/lub y in ^before(x): after(y) lub set{y});
end

```

Note that the final line in the definition of *after* fixes a bug in [DS91], in which call targets are never included in *after* sets.

Part 2: Analysis of Parallel Units([DS91], Eqn. 5). Once we have computed them, the *before* and *after* sets can be combined to yield the overall partial execution order. Specifically, our goal here is to compute for each node x the set $Ord(x)$ of all nodes in the program model that cannot execute simultaneously with x .

For our implementation, we begin with three auxiliary properties. The set $local(x)$ consist of those nodes that are in the same unit as x (the *restrict* type operator is

a limited, safe form of casting). The head of the unit to which x belongs is denoted by $head(x)$, while the set of all call sites of x 's unit is $head_callers(x)$. Once we have these, we compute Ord_{init} , which is the initial estimate of Ord (p. 43, par. 2).

```

def local(x:atom):(atom set) = ^unit(unit(x));
def localNodes = restrict(local,Nodes);

for x in (base cfg) def
  head(x):Nodes >=
    /lub h in local(x): (if isEntry(h) then set{h} else bot);
  head_callers(x):Nodes >= /lub h in head(x): ^callflow(h);

end
def Ord_init(x:atom):Nodes >=
  before(x) lub after(x) lub localNodes(x);

```

If there are no parallel execution units, then $Ord = Ord_{init}$. Duesterwald and Soffa observe that in order to take into account parallel execution of a unit, it suffices to consider the Ord sets holding at the call sites of that unit, using the result as a filter. This is Equation #5 in [DS91]. While not directly representable in our syntax, the following is an equivalent form:

```

for x in (base cfg) def
  unit_filter(x):Nodes <=
    if (empty head_callers(x)) then top
    else
      /glb c in head_callers(x): Ord(c) ;

  Ord(x):Nodes <= Ord_init(x) glb unit_filter(x) );
end

```

4.3 Properties

At this point, it is worth reflecting on some of the trade-offs we have made in balancing the expressiveness, decidability, and complexity of our language. Roke is essentially a lightly sugared query language, over first-order fixpoint inequalities. Strictly speaking, we do not really need to support any lattice datatypes other than

powersets, and if all our queries were monotone, we could safely express all inequalities as equalities, since least/greatest solutions would be guaranteed to exist.

Not all Roke-expressible queries are monotone, however, and monotonicity is not a decidable property for any but the simplest query language forms. Nonetheless, it is possible to restrict the syntax (*e.g.* with a positivity or stratification requirement [AHV95]) or to apply static analysis techniques that facilitate approximate enforcement (*e.g.* the type and effect systems given in [MY02, YE02]). However, monotonicity is a sufficient but not necessary condition for termination, and non-monotonic equations have some existing uses in program analysis [Che03, FS99, GKL⁺96]. We therefore take the view that this responsibility is best left to the user of our language.

Similarly, we can adopt an *inflationary semantics* for the language, in which we consider the meaning of a query to be its least inflationary (or greatest deflationary) fixed point (*ref.* Section 2.5.1). This is a more palatable requirement than those than monotonicity, since (1) we are always guaranteed the existence of a minimal solution, if a unique least fixed point does not exist, and (2) in the case of monotone queries, the least fixed point and inflationary fixed point solutions coincide. As discussed in the following chapter, this is, in fact the approach we choose to adopt.

Roke is therefore closest to a form of Datalog with unrestricted negation, but with inflationary semantics, over unordered databases [AHV95, Imm99].³ While this means that all queries take time polynomial in the size of the flow graph, this may still result in a prohibitively high complexity: the problem is complete for PTIME [AV91].

In fact, we have accomplished a much tighter bound for constraints expressible in Roke (and therefore also \mathcal{K}). To understand this, we define the following:

Definition 4.3.1. The *elementary operators* in Roke are the binary operators glb , lub , $-$, and γ_2 , the unary operators $-$ and γ_1 , and application/inverse applications $\sigma(x)/\hat{\sigma}(x)$, where $\sigma : \text{atom} \rightarrow \mathcal{D}$. An *elementary expression* is either a term or an elementary operator applied to operands that are elementary expressions.

³While lattices *are* partial orders, the base elements may not be. For example, in the lattice $(2^S, \subseteq)$, where $S = \{a, b, c\}$, there is no order among the atoms a , b , and c .

Note that we have limited the syntax of function definitions so that all function bodies are elementary expressions. Consequently, every definable function is total, and assuming a constant bound on program length, along with

Proviso 4.3.1. *The elementary operators can each be computed in time bounded by some constant.*

it is clear that

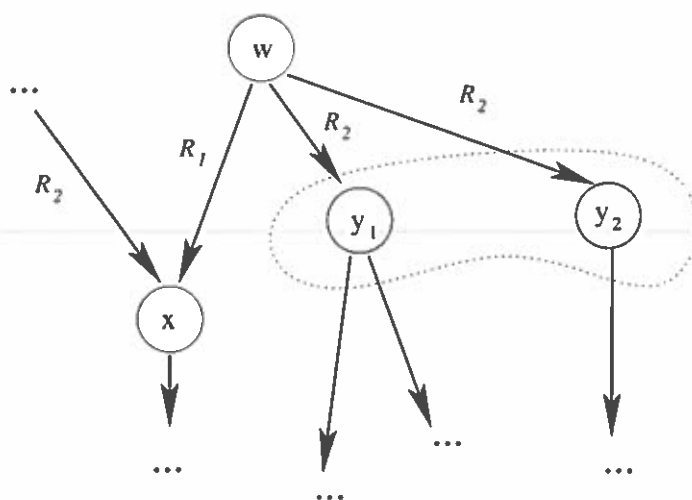
Lemma 4.3.2. *The execution time of every function application in Roke is bounded by some constant.*

□

In general, we require that these properties be preserved by any other constructs we might add to the language. Consequently, we explicitly exclude from the syntax of allowable function body expressions any construct that permits recursion or iteration.

Under more precise measures, of course, Proviso 4.3.1 is unrealistic, and we should perhaps use the more noncommittal requirement that function application have a cost no worse than that of an elementary expression. In fact, the cost of such high-level operators will vary, some times significantly, depending on the underlying data structures we choose. For a more general-purpose language, this problem cannot be ignored. The possibility of determining the running time of an operation from a program's source code is termed *computational transparency* by Cai and Paige [CP93], and there have been several efforts directed toward achieving this property with very high level constructs such as our elementary operators (*c.f.* [CP87, CP89, CFH⁺91, CP93, Goy00]).

Unlike high-level iterative and recursive forms, however, the complexity of elementary expressions depends only on the value domain that we use, not on the flow graph model that arises from a particular problem. Since the number of value domains we use is likely to be small compared to the range of flow graph models, it is therefore reasonable to consider the cost of elementary expressions as in some sense indivisible, much as arithmetic operations are viewed in more conventional, imperative languages.



$$A(x):T \Leftarrow /glb\ w\ in\ R_1(x): (\ /lub\ y\ in\ R_2(w): (A(y)\ glb\ B(w)) - B(x));$$

FIGURE 4.1: A banned nested iteration and an example portion of a flow graph on which this analysis might be defined. Note the way that flow information at the intermediate node w is itself used in the transformation of the value from y to x .

It is this desire to separate the complexity of various constructs that further motivates the restricted usage of flow expressions; *i.e.* the prohibition against “nested iteration.” The reason for this is the following

Lemma 4.3.3. *Let ϕ be an expression of length l . On a flow graph (N, E) evaluation of ϕ requires at most $\mathcal{O}(l \cdot |N|)$ elementary operations.*

Proof. By structural induction on ϕ . We observe that the only interesting cases are when ϕ is of the form “ $/lub\ w\ in\ \psi: e$ ”, “ $(/lub\ w\ in\ \psi: e)\ op\ e_2$ ”, “ $/glb\ w\ in\ \psi: e$ ”, or “ $(/glb\ w\ in\ \psi: e)\ op\ e_2$ ” (with $op \in \{lub, glb, -\}$). We consider only the first case, “ $/lub\ w\ in\ \psi: e$ ”, since the others are similar. By assumption, e consists of $\mathcal{O}(l)$ elementary operations. Now it is clear that e must be evaluated $\mathcal{O}(|\psi|)$ times, where $|\psi|$ is the cardinality of the set resulting from evaluation of ψ . This is always a value of the lattice 2^N whose largest element is N , and evaluation of ψ itself requires only $\mathcal{O}(l \cdot |N|)$ elementary operations (by I.H.), which completes the argument. More complex cases for ϕ follow from the inductive hypothesis. \square

Assuming that l is bounded by a constant, we have

Corollary 4.3.4. *Evaluation of the right hand side of any flow constraint requires $\mathcal{O}(|N|)$ elementary operations.* \square

Note that this proposition is false if nested flow expressions are allowed. In this case, evaluation of ϕ would grow as $\mathcal{O}(|N|^d)$, where d is the largest nesting depth of any flow subexpression of ϕ .

The example in Figure 4.1 makes clear that a flow equation containing a nested flow expression is of a quite different character from one that does not. In general, a constraint of the form

$$\sigma(x) : \tau \succ= \text{/lub } y \text{ in } \psi : \phi;$$

defines $\sigma(x)$ in terms of an index set ψ which is used (along with x itself) to establish a set of values, as defined by the expression ϕ . Note that ϕ can only be defined in terms of y and x , since these are the only free variables that can occur. In this way, ϕ serves to transform directly the flow of information between y and x . By contrast, the equation in Fig. 4.1 also makes use of information about the intermediate nodes on a *path* from y to x . Although such analyses might be useful in practice, the kind of path-sensitivity represented by allowing these “super arcs” of information flow is clearly different from the more direct notion of flow embodied in our language. Lemma 4.3.3 also shows that there is a cost exacted in performance.

On the other hand, we give up little in the way of practical expressiveness by adopting this restriction. We know of only one use of a nested iterative form in the flow analysis literature (Tenenbaum [Ten74], Eqn. 14), and even here, later formulations of this analysis such as [KU80, KDM03] make no use the nested form.

Note that we do permit flow expressions within the definition of the index set itself, as in the following example (adapted from [Ten74], Eqn. 12, which computes the type of an assignment x , based on def-use chains in the basic block where x occurs.):

```

// T = ... (lattice of types)
// N = ... (powerset lattice of nodes)

for x in nodes def
  idx(x):N >= du(x) glb
    (/lub w in (/lub b in block(x): ~block(x) ~ set{x}):
      rhs(w));
  back(x):T <= /glb w in idx(x): backtype(w);
end

```

From a conceptual view, we allow the use at x of flow values holding at any other node w in the flow graph model, regardless of whether x is a c -successor of w , for any edge type c . However, we discard information about the path from w to x . Instead, the atoms in the index set can be thought of as “predecessors” of x along some form of information flow.⁴ This flow will often correspond directly to edges in the flow graph, but in our language, the existing edges are a strict subset of the possible forms of flow. From an operational view, this incurs no additional penalty, since the index set is evaluated exactly once, rather than $\mathcal{O}(|N|)$ times.

The result of all this is that the time to solve specifications by chaotic iteration in our extended syntax is comparable to that of the more constrained k -tuple form. Practically speaking, we need only refine the evaluation function $F : (N \rightarrow \vec{\mathcal{D}}) \rightarrow (N \rightarrow \vec{\mathcal{D}})$ and accompanying $eval_\sigma$ (ref. p. 29) to handle the more general Roke forms:

$$F(\sigma) = \lambda n. (eval(1, n, \vec{\sigma}), \dots, eval(k, n, \vec{\sigma}))$$

where $eval(i, m, \rho)$ is the result of evaluating the right hand side of the inequality defining $\sigma_i(x)$, under the substitution $x \mapsto m$ and solution ρ .

(Alg. 2.5.1—Chaotic Iteration)

Input: *Multisource analysis specification* $\sigma_1(x): \tau_1 \geq \phi_1; \dots \sigma_k(x): \tau_k \geq \phi_k$; and
problem instance defined by the flow graph $(N, E \subseteq \bigcup_{i=1}^l N \times N \times i)$, with $|N| = n$.

⁴In fact, this is the purpose of the equation for idx , which allows us to consider in the definition of $back$ only those expressions that can use x and are also in the same block as x .

Output: The least $\vec{\sigma}$ satisfying the constraints $\sigma_1(x) : \tau_1 \geq \phi_1; \dots \sigma_k(x) : \tau_k \geq \phi_k;$

Method:

```

for each  $(i \in \{1, \dots, k\}, j \in N)$ :
     $\sigma_i(j) \leftarrow \iota_i(j)$ 
change  $\leftarrow$  true
while(change):
    change  $\leftarrow$  false
    for each  $(i \in \{1, \dots, k\}, j \in N)$ :
        newval  $\leftarrow$  eval( $i, j, \vec{\sigma}$ )
        if (newval  $\not\leq$   $\sigma_i(j)$ ):
             $\sigma_i(j) \leftarrow$  newval  $\sqcup$   $\sigma_i(j)$ 
            change  $\leftarrow$  true

```

□

We then have:

Theorem 4.3.5. Let $\mathcal{D}_1, \dots, \mathcal{D}_k$ be complete lattices such that each \mathcal{D}_i has finite height h_i , and let $\vec{\sigma} = \sigma_1, \dots, \sigma_k$ be defined by a set of simultaneous, mutually recursive queries over $\mathcal{D}_1, \dots, \mathcal{D}_k$, such that the right hand side of each σ_i is monotone. Let (N, E) be a flow graph. Algorithm 2.5.1 converges in $\mathcal{O}(k^2 |N|^3 \left(\sum_{i=1}^k h_i\right))$ elementary operations.

Proof. The $\sum_{i=1}^k h_i$ term is the height of $\vec{\mathcal{D}} = \mathcal{D}_1 \times \dots \times \mathcal{D}_k$. For each (σ_i, j) in the inner loop of, we perform $\mathcal{O}(|N|)$ elementary operations (Corollary 4.3.4). The inner loop iterates $\mathcal{O}(k|N|)$ times. If there is any node j whose value $\sigma(j)$ changes, then its value must increase, and for each of the $k|N|$ pairs (σ_i, j) this can happen at most h_i times, giving an overall cost of $\mathcal{O}(k|N|^2 \left(\sum_{i=1}^k k|N|h_i\right)) = \mathcal{O}(k^2 |N|^3 \left(\sum_{i=1}^k h_i\right))$. □

Further, as in Theorem 2.5.3,

Theorem 4.3.6. For a monotone specification S over the lattice $\vec{\mathcal{D}}$ satisfying ACC, Algorithm 2.5.1 computes $\text{lfp}(S)$. □

In practice, k (the number of “flow equations”) is likely to be quite small. Indeed, it is even reasonable to assume a constant bound on this value. Even so, the overall complexity of Algorithm 2.5.1 is unappealing. We take up the matter of improved solvers in the next chapter.

4.4 Related Work

In this chapter, we have developed a domain-specific language, Roke, that is suitable for specifying not only the transfer function and flow map components of a multisource analysis, but also the global abstract semantics. The nearest ancestor of Roke is the GenSet project of Young *et al.* [FLY02, ZYL04]. The two languages are very similar in both syntax and behavior, and many ideas in the design of Roke were taken directly from this earlier work. The main differences are (1) the definition of lattice operators over varying types (GenSet is restricted only to sets of atoms), (2) the restrictions on nested iteration in order to guarantee complexity bounds, and (3) the possibility in Roke of defining auxiliary transfer functions and multiple lattice types. The syntax and semantics of the data type and function definition constructs owe a debt to the PAG system of Martin *et al.* [Mar98, Mar99].

While the language has been developed with an eye toward natural specification, performance guarantees, and simplicity of reasoning, the particulars of the language are less important than its place in our overall approach. Other specification mechanisms may also be suitable.

An obvious choice is the use of a complete, declarative language such as SETL [SDDS86], an implementation of the abstract language SQ+ [CP89, Goy00], or some form of logic or partial-order programming [OJP99]. The disadvantage of all of these approaches is that, being general-purpose programming languages, there is a greater burden on the analysis designer to prove convergence. Moreover, these very high-level languages lack computational transparency [CP93], making it more difficult to establish fine-grained complexity bounds. On the other hand, Dawson *et al.* [DRW96] report experimental results that suggest general logic programming formulations of

program analyses can offer competitive performance, and this approach is worth further study.

Use of a deductive database language such as Datalog [AHV95] offers many of the benefits of a general-purpose language, but with convergence and complexity guarantees similar to ours. Further, there is a fairly rich body of work on implementing such languages, and existing implementation strategies can be leveraged to yield improved solution algorithms for program analysis.

McAllester [McA02] implements several forms of program analysis—including data flow analysis—as Datalog programs, and develops techniques for the establishment of precise complexity bounds, competitive with those of more standard implementations of the analyses. In addition, Reps *et al.* [HRS95, Rep94b, Rep94a, Rep98, SRH95] have applied this approach to formulate context and flow-sensitive interprocedural analyses as Datalog programs. Using this formulation, they can obtain a demand-driven solver for free.⁵

However, Datalog is strictly less expressive than Roke (since it cannot express negation, or at best is limited to stratified negation). In addition, for both Datalog and more powerful Prolog-like languages, the iterated \cup operation $\bigcup_{i=1}^k e_i$ is embedded implicitly in the multiple rules and solutions to rules that may exist. This is acceptable so long as our iterated \sqcup operator is set union and we have no need of flow expressions over \sqcap , as well. However, a k -ary intersection $\bigcap_{i=1}^k e_i$ cannot be expressed statically for arbitrary k , and while workarounds exist in many cases (*c.f.* Reps [Rep94a], p.13), in general an analysis requiring k -ary intersection is at best awkward to construct.

Perhaps the most compelling alternative is the formulation of multisource analyses as set constraint problems, which can then be supplied to a general constraint system, such as Banshee [KA05]. Set constraints are highly expressive, and have been successfully applied to a large array of program analyses [NNH99]. The disadvantage is that set constraint languages are strictly more expressive than we need, and consequently have a worse worst-case complexity [Aik99]. On the other hand, there are restricted

⁵... either by using the magic sets transformation of Bancilhon *et al.* [BMSU86] with bottom-up evaluation on the transformed program that results, or else by evaluating the (original) Datalog program with a top-down tabulating evaluator [War92].

forms of set constraints that have polynomial ($\mathcal{O}(N^3)$) complexity, which could well be expressive enough for our needs. In many ways, our language, much like its predecessor GenSet, can be understood as an explicit restriction to this “sweet spot”, with added syntactic sugar appropriate to the domain of flow analysis problems.

Other than the one developed in this chapter, we are unaware of any approach to toolkit development that encompasses arbitrary multisource data flow analyses. Lerner *et al.* [LGC02] indicate that a toolkit was built by Masticola’s group to support the work in [MMR95], but this is incorrect.⁶ As discussed in Chapter II, there are a number of toolkits to support the automatic generation of data flow analyzers, but even those supporting advanced analysis forms are unsuitable for the general multisource case.

It is possible that, with a sufficiently rich language for defining the value lattice and transfer functions, we could simply implement a k -tuple data flow problem as a specification provided to a traditional toolkit. We believe it likely, however, that the resulting specification would be nearly as complex as a hand-built implementation, since we would have to encode all information about edge types (along with the associated global semantics) within the transfer functions alone. Even then, we still have not accounted for the cases where there is no clear expression of the problem as a framework instance, such as a global abstract semantics defined on a dynamically-changing flow graph model.

More directly, we could abandon the multisource data flow view, and instead implement such analyses as abstract interpretations (for example, [CCF⁺05, Mar98, YH93]). As discussed above, this restricts application to program analysis, which is but one application of the data flow analysis technique.

⁶Barbara Ryder, personal communication.

CHAPTER V

SOLUTION TECHNIQUES

This dissertation supports the thesis that the implementation of program analyzers for multisource data flow problems can be largely automated with an analysis generation toolkit. In the previous two chapters, we have shown how to extend the traditional approach to toolkit-level problem specification, in order to support the specifications in this more general problem family. Here, we show that efficient implementations for these specifications can be constructed automatically.

The primary contribution of this chapter is the development of a new solution technique for the fixpoint constraint systems that arise from Roke specifications. Because Roke is expressive enough for nearly all multisource data flow problems, we have therefore overcome the main impediment to the automated implementation of efficient solvers.

In contrast to existing data flow analysis toolkits, our approach does not assume any particular relationship between flow graph structure and global abstract semantics. Rather, this relationship is itself part of a Roke specification, whose structure guides the generated solver. The technique, while being fairly easy to implement, is quite general: the version presented here handles not only static constraint systems—*i.e.* those that remain fixed throughout the solution process—but also systems in which the constraints themselves can change dynamically.

5.1 The Influence of Influence

Theorem 4.3.5 shows that ordinary chaotic iteration (Algorithm 2.5.1) can be used to solve monotone Roke specifications. Unfortunately, the complexity of this approach—cubic in the number of flow graph nodes—may be prohibitively expensive. As with the traditional version of the solver, it is also very often unnecessary. Again, the primary culprit is the wasted effort that results from re-computing the value at *every* node, for every flow constraint, whenever a change occurs.

To avoid this waste, we want a change in value to induce re-computation only for those constraint/node pairs whose solutions have destabilized as a result. The standard approach (for example, [NNH99]) is to view such influences as arcs in a directed graph, which can itself be used to control the re-computation of solutions.

Definition 5.1.1. Let $\vec{\sigma} \stackrel{\text{def}}{=} \sigma_1(x) : \mathcal{D}_1 \succ e_1; \dots \sigma_k(x) : \mathcal{D}_k \succ e_k$; be a specification of a data flow analysis over the lattice $\vec{\mathcal{D}} \equiv \mathcal{D}_1 \times \dots \times \mathcal{D}_k$ and let the flow graph $G = (N, E)$ be an instance of this analysis.

- For $m, n \in N$, solution $\rho : N \rightarrow \vec{\mathcal{D}}$, and maps $\sigma_i : N \rightarrow \mathcal{D}_i, \sigma_j : N \rightarrow \mathcal{D}_j$ ($1 \leq i, j \leq k$), we say that $\sigma_i(m)$ *influences* $\sigma_j(n)$ in ρ if there exist distinct $a, b \in \mathcal{D}_i$ and $c \in \mathcal{D}_j$ such that

$$\text{eval}(e_j, n, \rho[\sigma_i(m) \mapsto a, \sigma_j(n) \mapsto c]) \neq \text{eval}(e_j, n, \rho[\sigma_i(m) \mapsto b, \sigma_j(n) \mapsto c])$$

- The *information flow graph (IFG)* for $(\vec{\sigma}, G, \rho)$ is a directed graph $(N_{IFG}^\rho, E_{IFG}^\rho)$, where $N_{IFG}^\rho = \{(i, n) \mid 1 \leq i \leq k \wedge n \in N\}$ and $E_{IFG}^\rho = \{(x, y) \mid x, y \in N_{IFG}^\rho \wedge x \text{ influences } y \text{ in } \rho\}$

Given a specification, a problem instance, and the corresponding IFG, we can apply the workset improvement to obtain a solver that can in practice offer a significant performance improvement:

Algorithm 5.1.1 (IFG Workset Solver).

Input: Flow analysis specification $\sigma_1(x):\mathcal{D}_1 \succ= e_1; \dots \sigma_k(x):\mathcal{D}_k \succ= e_k$; and information flow graph $(N_{IFG}^\rho, E_{IFG}^\rho)$

Output: The least $\vec{\sigma} : N \rightarrow \vec{\mathcal{D}}$, satisfying $\sigma_1(x):\mathcal{D}_1 \succ= e_1; \dots \sigma_k(x):\mathcal{D}_k \succ= e_k$;

Method:

```

for each  $(i, m) \in N_{IFG}^\rho$ :
     $\sigma_i(m) \leftarrow \perp$ 
     $W.add((i, m))$ 

while ( !W.empty() ):
     $(i, m) \leftarrow W.extract()$ 
     $newval \leftarrow eval(e_i, m, \vec{\sigma})$ 
    if  $(newval \not\leq \sigma_i(m))$ :
         $\sigma_i(m) \leftarrow \sigma_i(m) \sqcup newval$ 
         $W \leftarrow infl_{IFG}(W, \vec{\sigma}, m, i)$ 

```

Where:

```

 $infl_{IFG}(W, \rho, m, i) \stackrel{def}{=}$ 
    for each  $(j, n) \in E_{IFG}^\rho((i, m))$ :
         $W.add((j, n))$ 
    return  $W$ 

```

□

With only minor changes, this is essentially the solver given by Algorithm 2.5.3, the main difference being the need to include the updated map σ_i as a parameter. Likewise, much of the technical development parallels the earlier work in Section 2.5.2.

Theorem 5.1.1. Let $\vec{\mathcal{D}} = \mathcal{D}_1 \times \dots \times \mathcal{D}_k$ be complete lattices such that each \mathcal{D}_i has finite length h_i , and let $\sigma_1(x):\mathcal{D}_1 \succ= e_1; \dots \sigma_k(x):\mathcal{D}_k \succ= e_k$; be a monotone specification of a flow analysis over $\vec{\mathcal{D}}$. Let the flow graph $G = (N, E)$ be an instance of this analysis with IFG $(N_{IFG}^\rho, E_{IFG}^\rho)$. Denote by M'_\bullet the largest number of edges in E_{IFG}^ρ that share a common source. Algorithm 5.1.1 converges in $\mathcal{O}(M'_\bullet(k|N|^2) \left(\sum_{i=1}^k h_i \right))$ elementary operations.

Proof. On each iteration of the while loop, we remove one element (i, m) from W . We then perform $\mathcal{O}(|N|)$ join operations, after which $\sigma_i(m)$ is either left unchanged or else assigned the new value *newval*. If we assign to $\sigma_i(m)$, it is with a strictly larger value. This can happen at most h_i times. With each of these assignments, the call to infl_{IFG} adds $\mathcal{O}(M'_\bullet)$ new elements to W , so each $(i, m) \in N_{IFG}^\rho$ adds $\mathcal{O}(h_i \cdot M'_\bullet)$ elements to W . Hence, we perform $\mathcal{O}(M'_\bullet |N_{IFG}^\rho| \left(\sum_{i=1}^k h_i \right)) = \mathcal{O}(M'_\bullet (k|N|) \left(\sum_{i=1}^k h_i \right))$ iterations of the while loop, for an overall cost of $\mathcal{O}(M'_\bullet (k|N|^2) \left(\sum_{i=1}^k h_i \right))$ elementary operations. \square

Actually, the bounds are a little better than this, since one of the $|N|$ factors comes from the cost of evaluating each right hand side (Corollary 4.3.4). Although an actual proof requires the development of the next section, it is not hard to show that this cost is in fact bounded by M'_\circ , the largest number of edges in E_{IFG}^ρ that share a common sink. Hence we can bound our workset solver by $\mathcal{O}(M'_\circ M'_\bullet k |N| \left(\sum_{i=1}^k h_i \right))$, which more closely matches the result of Theorem 2.5.8. Similarly, we can use M'_\circ to tighten the bound in Theorem 4.3.5. Either way, both M'_\circ and M'_\bullet are always bounded by $|N|$, and in many cases, the bounds are constant.

Correctness also follows along lines very similar to those followed earlier (*ref.* Theorem 2.5.12). As above, we write the sequential execution of steps s and t in the solver as $(s; t)$. However, in this case we make use of the predicate

$$I_W((\sigma_i, x)) \stackrel{\text{def}}{=} \sigma_i(x) \not\geq \text{eval}(e_i, x, \vec{\sigma}) \implies (\sigma_i, x) \in W$$

the difference lying mainly in the use of the IFG. The notion of a *workset function* for $\langle s, \vec{\sigma}, W \rangle$ is defined with respect to this predicate. Denoting by a_σ , the assignment $\sigma_i(m) \leftarrow \sigma_i(m) \sqcup \text{newval}$ (line 8), we have

Lemma 5.1.2. *If infl_{IFG} is a workset function for $\langle a_\sigma, \vec{\sigma}, W \rangle$, then, on a monotone specification S over a lattice \vec{D} that satisfies ACC, Algorithm 5.1.1 terminates with $\vec{\sigma} = \text{lfp}(S)$.*

Proof. Similar to Lemma 2.5.10. \square

Lemma 5.1.3. *The function infl_{IFG} is a workset function for $\langle a_{\sigma_i}, \vec{\sigma}, W \rangle$.*

Proof. As in the proof of Lemma 2.5.11, only in this case, we are considering the IFG. In particular, we argue from the structure of E_{IFG}^p , whose arcs represent all influence between $((i, m), (j, n))$ pairs. □

Corollary 5.1.4. *On a monotone specification S over a lattice \vec{D} that satisfies ACC, Algorithm 5.1.1 terminates with $\vec{\sigma} = \text{lfp}(S)$.* □

That the development here should parallel that of the earlier classical case is hardly a coincidence. For both the forward and backward versions of Algorithm 2.5.3, correctness is due to the fact that, in each case, the accompanying *infl* functions guarantee the propagation of flow value changes to every directly-affected node in the flow graph. The nodes to which a change must be propagated are precisely the successor nodes in the problem's IFG. For forward intraprocedural analyses, the IFG edges are tightly approximated by the edges in the control-flow graph (*i.e.* $E \supseteq E_{IFG}^p$). For backwards analyses, we use the reversed CFG.

In both cases, the approximation may be an overestimate,¹ but this is all we need:

Lemma 5.1.5. *In Algorithm 5.1.1, replace infl_{IFG} with the function f , defined as*

```

f( $W, \rho, m, i$ )  $\stackrel{\text{def}}{=}
  \text{for each } (j, n) \in \text{ReQ}^p(i, m):
    W.\text{add}((j, n))
  \text{return } W$ 

```

If

$$\forall (i, m) \in N_{IFG}^p : \text{ReQ}^p(i, m) \supseteq E_{IFG}^p((i, m))$$

then f is a workset function for $\langle a_{\sigma_i}, \vec{\sigma}, W \rangle$.

Proof. Easy. □

¹For example, if the flow value holding at node x is a constant, then we do not need to propagate from control flow successors / predecessors of x .

In order to discover an efficient solver for a multisource data flow problem (given as a Roke specification), we should therefore determine a good approximation of the problem's IFG. To construct such solvers automatically, we need to automate this discovery, as well. This problem is taken up in the following section.

5.2 Challenges to the Determination of Influence

In classical intraprocedural analysis, static determination of influence is a simple problem: the IFG is essentially the control flow graph for forward analyses, and the reverse of the control flow graph for backward ones. Although it is unclear how one would construct a reliable metric, this approximation of the influence relation for any classical unidirectional analysis problem is probably quite good. Program points whose properties cannot be affected by their control predecessors/successors are not common. Indeed, we believe it is reasonable to assume that any overestimate is within a constant factor.

In many advanced forms, the relationship between the global abstract semantics and a problem's IFG is also well-understood. For context/flow sensitive interprocedural analyses, workset solvers are presented in Sharir and Pnueli [SP81] and Knoop *et al.* [KRS96], and several are implemented in the PAG toolkit [Mar98]. For analysis of parallel programs, the situation is somewhat more complicated, as determination of the IFG depends on the chosen model of concurrency. Nonetheless, there are several implementations of flow analyses for concurrent programs that successfully adapt the workset heuristic (*e.g.* [Dwy95, KSV96]). Khedker and Dhamdhere's unified approach to bidirectional and unidirectional bit-vector analyses [KD94] considers four different flows—control flow information and its reverse, through both nodes and arcs—and includes a workset-based solver for instances of this framework.

Unfortunately, the limitation of all of these solution variants is the same one that makes their underlying frameworks unsuitable for specification of arbitrary multisource flow analyses: all presume a fixed form for the global abstract semantics. Consequently, the relationship between a flow graph model and its IFG is also fixed,

and the accompanying solvers are thus limited to analyses only within their respective families.

What we want is a means of determining the influence relation from arbitrary Roke specifications. For this, the most straightforward method is the “generate and solve” approach embodied in various forms of set constraint-based analysis [Aik99] (see also [NNH99], pp. 366-368).

The insight here is that a system of flow inequations

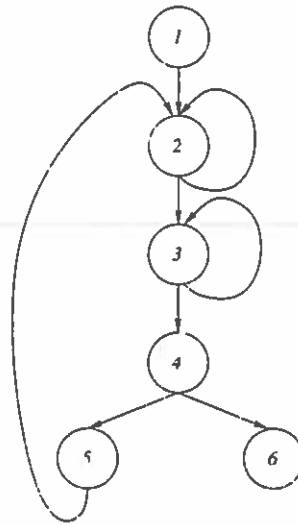
$$\begin{aligned}\sigma_1(x) : \tau_1 &\geq e_1; \\ &\vdots \\ \sigma_k(x) : \tau_k &\geq e_k;\end{aligned}$$

is parametric on the set of flow graph nodes, and thus serves as a set of constraint *generators*. To determine the IFG for a given problem, we apply σ_j to each $n \in N$ ($1 \leq j \leq k$), expand the corresponding right hand sides (by unrolling all flow subexpressions in $e_j\{n/x\}$), and consider the resulting $k|N|$ (fully instantiated) constraints. The left hand sides constitute the IFG node set N_{IFG}^ρ . E_{IFG}^ρ is constructed by adding, for every $\sigma_j(n)$, the edge $((i, m), (j, n))$, for each $\sigma_i(m) \in e_j\{n/x\}$.

Consider, for example, a specification of liveness analysis in the traditional IN/OUT form:

```
LV_in(x) >= (LV_out(x) ~ Kill(x)) lub Gen(x);
LV_out(x) >= if (isExit(x)) then bot
             else
               /lub y in flow(x): LV_in(y);
```

On the flow graph



we have the constraint system

$$\begin{aligned}
 LV_in(1) &\geq (LV_out(1) - Kill(1)) \text{ lub } Gen(1) \\
 LV_in(2) &\geq (LV_out(2) - Kill(2)) \text{ lub } Gen(2) \\
 LV_in(3) &\geq (LV_out(3) - Kill(3)) \text{ lub } Gen(3) \\
 LV_in(4) &\geq (LV_out(4) - Kill(4)) \text{ lub } Gen(4) \\
 LV_in(5) &\geq (LV_out(5) - Kill(5)) \text{ lub } Gen(5) \\
 LV_in(6) &\geq (LV_out(6) - Kill(6)) \text{ lub } Gen(6)
 \end{aligned}$$

$$\begin{aligned}
 LV_out(1) &\geq LV_in(2) \\
 LV_out(2) &\geq LV_in(2) \text{ lub } LV_in(3) \\
 LV_out(3) &\geq LV_in(3) \text{ lub } LV_in(4) \\
 LV_out(4) &\geq LV_in(5) \text{ lub } LV_in(6) \\
 LV_out(5) &\geq LV_in(2) \\
 LV_out(6) &\geq \text{bot}
 \end{aligned}$$

This yields the IFG of Figure 5.1.

The “generate and solve” approach works for many, but not all, specifications. The problem is that the generation of constraints from the original parametric forms

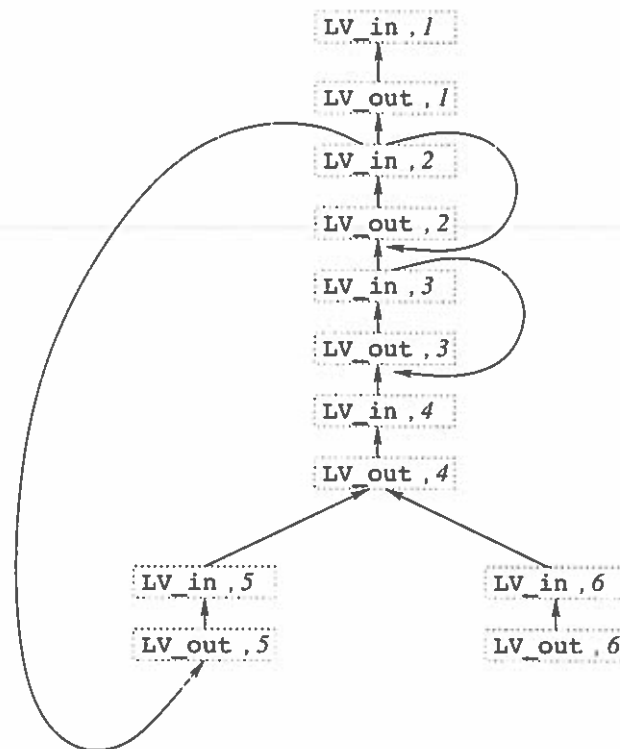


FIGURE 5.1: Example IFG arising from LV analysis

happens all at once, before we begin solving them. While this “separation of specification and implementation” is an appealing feature of set constraint-based analyses [Aik99], it cannot handle the case in which the constraints themselves change dynamically during intermediate results of the solution process.

In data flow analysis, this situation arises whenever intermediate results are used to prune or add edges from or to the flow graph. Examples of this include qualified data flow analysis [HR81], composed analysis and optimization [BGS97, CC95, LGC02], and various uses of reachability analysis [DS91]. In Roke specifications, it corresponds to a flow expression “/lub w in ϕ : e ” or “/glb w in ϕ : e ”, in which the index set expression ϕ is recursive; *i.e.* it contains an occurrence of some flow map σ , which is itself defined in the same block. The *before/after* analysis in Example 3.3.1 is representative of this.

5.3 A Hybrid Technique for Static Determination

For the general multisource case, the key to our approach lies in the fact that the Roke specification of a data flow analysis is merely a program. The significance of this is the possibility that static analysis techniques can be applied fruitfully to the specification itself, in order to improve the underlying solver. Specifically, we will employ a static analysis in order to discover information about the (implicitly-defined) IFG of a specification. While we cannot always determine the influence relation statically, there remains still the possibility of determining, statically, *how* to find this information, dynamically.

5.3.1 Factoring Complex Flow Expressions

In order to simplify the construction, we begin with a normalization step, which factors out complex index sets from every flow expression. These are any index set expressions other than applications ($\sigma(x)$) or inverse applications ($\overset{\sim}{\sigma}(x)$), and we replace each one with a new abstraction of the expression and an application of that abstraction in the original context. For example,

$$\begin{aligned} \sigma_1(x) : \mathcal{D}_1 &\geq e_1; \\ &\vdots \\ \sigma_i(x) : \mathcal{D}_i &\geq / \text{lub } w \text{ in } \phi : e_i; \\ &\vdots \\ \sigma_k(x) : \mathcal{D}_k &\geq e_k; \end{aligned}$$

becomes

$$\begin{aligned} \sigma_1(x) : \mathcal{D}_1 &\geq e_1; \\ &\vdots \\ \sigma_i(x) : \mathcal{D}_i &\geq / \text{lub } w \text{ in } \sigma_{k+1}(x) : e_i; \\ &\vdots \\ \sigma_k(x) : \mathcal{D}_k &\geq e_k; \\ \sigma_{k+1}(x) : 2^{\text{atom}} &\geq \phi; \end{aligned}$$

This transformation is applied iteratively until there are no complex ϕ remaining. Since at each iteration, we only remove a complex index set expression (and never add one), termination of the transformation is assured. The result is that every flow expression will be of the form “ $\square w \text{ in } \sigma(x):e$ ” or “ $\square w \text{ in } \sigma(x):e$ ” (where \square is either of \sqcap, \sqcup).

It should be noted that this step provides another reason for the prohibition in Roke on nested flow expressions. Were we to allow such expressions, the transformation would break on any constraint of the form

$$\sigma(x):\mathcal{D} \triangleright= /op w \text{ in } e_1: (/op y \text{ in } e_2: e_3);$$

in which x appears in e_3 , since the result would move x outside of its scope.

5.3.2 Static Construction of Influence Functions

Our approach, which extends the earlier results of [FLY02], is based on a static analysis of the specification’s source code, in which we perform a partial evaluation [JGS93], specializing each equational definition to a set of “influence functions”. Specifically, we associate with each equation definition σ_i a set of finite maps, $d_i^j : (N \rightarrow \vec{\mathcal{D}}) \rightarrow N \rightarrow 2^N$, one for each parametric equation σ_j (including σ_i itself). We assume that the space required to represent a single d_i^j is bounded by a constant. Since the number of these functions is quadratic in the number of parametric equations defined, we have a quadratic space bound overall. In practice the number of these definitions is likely small, so this overhead should be manageable.

The construction of each function d_i^j is by induction on the right hand side of σ_j . It represents the computation necessary to determine the nodes on which σ_j might change, when the value of $\sigma_i(m)$ changes for some node m , *i.e.* those nodes that are the immediate successors of (i, m) in the IFG. Formally, for a constraint $\sigma_i(x) \triangleright= e_i$, the influence function for a definition $\sigma_j(x) \triangleright= e_j$ is

$$d_i^j = \lambda\rho.\lambda m.dep(\rho, i, e_j, m)$$

where $dep(\rho, i, e_j, m)$ is defined inductively on the structure of e_j as follows:

$$\begin{aligned}
dep(\rho, i, x, m) &= \emptyset \\
dep(\rho, i, c, m) &= \emptyset \\
dep(\rho, i, \sigma_j(x), m) &= \begin{cases} \{m\} \\ \emptyset \end{cases} \\
dep(\rho, i, f(e_1 \dots e_q), m) &= dep(\rho, i, e_1, m) \cup \dots \cup dep(\rho, i, e_q, m) \\
dep(\rho, i, \tilde{\sigma}_j(x), m) &= \begin{cases} eval(\sigma_i(x), m, \rho) \\ \emptyset \end{cases} \\
dep(\rho, i, e_1 \bullet e_2, m) &= dep(\rho, i, e_1, m) \cup dep(\rho, i, e_2, m) \\
dep(\rho, i, o e, m) &= dep(\rho, i, e, m) \\
dep(\rho, i, \text{if } e_1 \text{ then } e_2 \text{ else } e_3, m) &= dep(\rho, i, e_1, m) \cup dep(\rho, i, e_2, m) \cup dep(\rho, i, e_3, m) \\
dep(\rho, i, /op w \text{ in } r(x) : e, m) &= \begin{cases} dep(\rho, i, r(x), m) \\ eval(\tilde{r}(x), m, \rho) \cup dep(\rho, i, r(x), m) \end{cases} \\
dep(\rho, i, /op w \text{ in } \tilde{r}(x) : e, m) &= \begin{cases} dep(\rho, i, \tilde{r}(x), m) \\ eval(r(x), m, \rho) \cup dep(\rho, i, \tilde{r}(x), m) \end{cases}
\end{aligned}$$

$$\begin{aligned}
(x \in \langle id \rangle) \\
(c \in \{\text{top, bot}\} \cup \langle const \rangle) \\
(\text{if } i = j) \\
(\text{otherwise}) \\
(\text{if } i = j) \\
(\text{otherwise}) \\
(\bullet \in \{\text{lub, glb, } \sim\} \cup \gamma_2) \\
(o \in \{\sim\} \cup \gamma_1) \\
(\text{if } dep(\rho, i, e, m) = \emptyset) \\
(\text{otherwise}) \\
(\text{if } dep(\rho, i, e, m) = \emptyset) \\
(\text{otherwise})
\end{aligned}$$

FIGURE 5.2: The influence expression function.

Example 5.3.1. For the live variables specification given above as Example 3.2.1

```

for x in (base flow) def
  LV(x) = /lub y in ^revFlow(x): f(y,x, LV(y));
end

```

we have

$$dep(\rho, LV, f(y, x, LV(y)), x) = \emptyset \cup \emptyset \cup dep(\rho, LV, LV(y), x) = \{x\}$$

and so the influence function is the familiar form

$$d_{LV}^{LV}(\rho) = \lambda n. eval(\text{revFlow}(x), n, \rho)$$

5.3.3 Application

We can use the statically-constructed influence functions to obtain a workset algorithm in which all necessary influence information is determined dynamically, thereby providing an efficient solver for any multisource flow analysis expressible in Roke:

Algorithm 5.3.1 (General Workset Solver).

Input: Flow analysis specification $\sigma_1(x): \mathcal{D}_1 \triangleright e_1; \dots \sigma_k(x): \mathcal{D}_k \triangleright e_k$; and flow graph model (N, E)

Output: The least $\vec{\sigma} : N \rightarrow \vec{\mathcal{D}}$, satisfying $\sigma_1(x): \mathcal{D}_1 \triangleright e_1; \dots \sigma_k(x): \mathcal{D}_k \triangleright e_k$;

Method:

```

for each (i, m):
   $\sigma_i(m) \leftarrow \perp$ 
  W.add((i, m))
while (!W.empty()):
  (i, m)  $\leftarrow$  W.extract()
  newval  $\leftarrow$  eval( $e_i, m, \vec{\sigma}$ )
  if (newval  $\not\leq$   $\sigma_i(m)$ ):
     $\sigma_i(m) \leftarrow \sigma_i(m) \sqcup$  newval
    W  $\leftarrow$  inflR(W,  $\vec{\sigma}, i, m$ )

```

Where:

```

 $infl_{\mathcal{R}}(W, \rho, i, m) \stackrel{def}{=}$ 
  for each  $(j, n) \in ReQ^\rho(i, m)$ :
     $W.add((j, n))$ 
  return  $W$ 

```

And:

```

 $ReQ^\rho(i, m) \stackrel{def}{=}$ 
   $rq \leftarrow \emptyset$ 
  for each  $j \in \{1, \dots, k\}$  :
    for each  $n \in d_i^j(\rho)(m)$ :
       $rq \leftarrow rq \cup (j, n)$ 
  return  $rq$ 

```

□

The difference between this version and Algorithm 5.1.1 lies in the input requirements and in the substance of the $infl_{\mathcal{R}}$ function. Here, we require only the specification and flow graph model itself rather than assuming construction *a priori* of the IFG. Of course N_{IFG}^ρ is easily recovered from $\vec{\sigma}$ and N . To see that we also have E_{IFG}^ρ , it suffices to show that

Lemma 5.3.1. $ReQ^\rho \supseteq E_{IFG}^\rho$.

Proof. Writing $N_{(i,m)}^j \stackrel{def}{=} \{n \mid (j, n) \in E_{IFG}^\rho((i, m))\}$, observe that $E_{IFG}^\rho((i, m))$ is equal to

$$\bigcup_{1 \leq j \leq k} \{(j, n) \mid n \in N_{(i,m)}^j\}$$

and that these sets partition $E_{IFG}^\rho((i, m))$. We claim that, for all $m \in N$ and $\sigma_j(x) : \mathcal{D}_j \rightarrow e_j$, $d_i^j(\rho)(m) = dep(\rho, i, e_j, m) \supseteq N_{(i,m)}^j$. The proof is by induction on e_j :

$e_j \equiv [\text{bot} \mid \text{top} \mid \langle \text{const} \rangle \mid \langle \text{id} \rangle]$. For all $a, b \in \mathcal{D}_i$ and $c \in \mathcal{D}_j$ we have

$$eval(e_j, n, \rho[\sigma_i(m) \mapsto a, \sigma_j(n) \mapsto c]) = eval(e_j, n, \rho[\sigma_i(m) \mapsto b, \sigma_j(n) \mapsto c])$$

Hence, $N_{(i,m)}^j = \emptyset = dep(\rho, i, [\text{bot} \mid \text{top} \mid \langle \text{const} \rangle \mid \langle \text{id} \rangle], m)$.

$e_j \equiv \sigma(x)$. If $\sigma \neq \sigma_i$, then $N_{(i,m)}^j = \emptyset$. Otherwise,

$$\begin{aligned} N_{(i,m)}^j &= \{ n \mid \exists a, b \in \mathcal{D}_i, c \in \mathcal{D}_j : \\ &\quad \text{eval}(\sigma_i(x), n, \rho[\sigma_i(m) \mapsto a, \sigma_j(n) \mapsto c]) \\ &\quad \neq \text{eval}(\sigma_i(x), n, \rho[\sigma_i(m) \mapsto b, \sigma_j(n) \mapsto c]) \} \\ &= \{m\} \\ &= \text{dep}(\rho, i, \sigma_i(x), m) \end{aligned}$$

$e_j \equiv f(e_1 \dots e_q)$. For all $a, b \in \mathcal{D}_i, c \in \mathcal{D}_j$

$$\begin{aligned} &\text{eval}(f(e_1 \dots e_q), n, \rho[\sigma_i(m) \mapsto a, \sigma_j(n) \mapsto c]) \\ &\quad \neq \text{eval}(f(e_1 \dots e_q), n, \rho[\sigma_i(m) \mapsto b, \sigma_j(n) \mapsto c]) \end{aligned}$$

only if there is some e_p ($1 \leq p \leq q$) that evaluates to different results under $\rho[\sigma_i(m) \mapsto a, \sigma_j(n) \mapsto c]$ and $\rho[\sigma_i(m) \mapsto b, \sigma_j(n) \mapsto c]$. Thus,

$$\begin{aligned} N_{(i,m)}^j &\subseteq \bigcup_{1 \leq p \leq q} \{ n \mid \exists a, b \in \mathcal{D}_i, c \in \mathcal{D}_j : \\ &\quad \text{eval}(e_p, n, \rho[\sigma_i(m) \mapsto a, \sigma_j(n) \mapsto c]) \\ &\quad \neq \text{eval}(e_p, n, \rho[\sigma_i(m) \mapsto b, \sigma_j(n) \mapsto c]) \} \\ &\subseteq \bigcup_{1 \leq p \leq q} \text{dep}(\rho, i, e_p, m) \text{ [I.H.]} \\ &= \text{dep}(\rho, i, f(e_1 \dots e_q), m) \end{aligned}$$

$e_j \equiv \sim \sigma(x)$. If $\sigma \neq \sigma_i$, then $N_{(i,m)}^j = \emptyset$. Otherwise, consider the assignment

$$\sigma_i(m) \leftarrow \sigma_i(m) \sqcup \text{newval}$$

and let a and b denote the values of $\sigma_i(m)$ before (*resp.* after) this assignment. Note that $b \supset a$, and so, for all n

$$\text{eval}(\sim \sigma_i(x), n, \rho[\sigma_i(m) \mapsto a]) \subseteq \text{eval}(\sim \sigma_i(x), n, \rho[\sigma_i(m) \mapsto b])$$

(the inequality comes from the possibilities $n \in b$ and $n \notin b$). We therefore have

$$\begin{aligned} N_{(i,m)}^j &\subseteq \text{eval}(\sigma_i(x), m, \rho[\sigma_i(m) \mapsto b]) \\ &= \text{dep}(\rho, i, \sigma_i(x), m) \end{aligned}$$

$e_j \equiv e_1 \bullet e_2$ ($\bullet \in \{\text{lub}, \text{glb}, -\} \cup \gamma_2$). Apply the inductive hypothesis to e_1 and e_2 , and proceed on the same lines as the case $e_j \equiv \sigma(x)$.

$e_j \equiv \circ e$ ($\circ \in \{-\} \cup \gamma_1$). Apply the inductive hypothesis to e , and proceed on the same lines as the case $e_j \equiv \sigma(x)$.

$e_j \equiv \text{if } e_1 \text{ then } e_2 \text{ else } e_3$. Apply the inductive hypothesis to e_1 , e_2 , and e_3 . Proceed on the same lines as the case $e_j \equiv \sigma(x)$.

$e_j \equiv /op w \text{ in } r(x):e$. Denote by $N_{(i,m)}^r$ (resp. $N_{(i,m)}^e$) the set of nodes for which the value of $r(x)$ (resp. e) can change with $\sigma_i(m)$. Formally,

$$\begin{aligned} N_{(i,m)}^r &= \{ n \mid \exists a, b \in \mathcal{D}_i, c \in \mathcal{D}_j : \\ &\quad \text{eval}(r(x), n, \rho[\sigma_i(m) \mapsto a, \sigma_j(n) \mapsto c]) \\ &\quad \neq \text{eval}(r(x), n, \rho[\sigma_i(m) \mapsto b, \sigma_j(n) \mapsto c]) \} \\ N_{(i,m)}^e &= \{ n \mid \exists a, b \in \mathcal{D}_i, c \in \mathcal{D}_j : \\ &\quad \text{eval}(e, n, \rho[\sigma_i(m) \mapsto a, \sigma_j(n) \mapsto c]) \\ &\quad \neq \text{eval}(e, n, \rho[\sigma_i(m) \mapsto b, \sigma_j(n) \mapsto c]) \} \end{aligned}$$

If $N_{(i,m)}^e = \emptyset$ then

$$N_{(i,m)}^j \subseteq N_{(i,m)}^r = \text{dep}(\rho, i, r(x), m) \quad [I.H.]$$

Otherwise

$$\begin{aligned} N_{(i,m)}^j &\subseteq N_{(i,m)}^r \cup (N_{(i,m)}^e \cap \text{eval}(\sim r(x), m, \rho)) \\ &\subseteq N_{(i,m)}^r \cup \text{eval}(\sim r(x), m, \rho) \\ &= \text{dep}(\rho, i, r(x), m) \cup \text{eval}(\sim r(x), m, \rho) \\ &= \text{dep}(\rho, i, e_j, m) \end{aligned}$$

$e_j \equiv /op w \text{ in } \sim r(x):e$. Similar to the case $e_j \equiv /op w \text{ in } r(x):e$.

We thus have, for all $(i, m) \in N_{IFG}^\rho$

$$\begin{aligned} \text{ReQ}^\rho(i, m) &= \bigcup_{1 \leq j \leq k} \{(j, n) \mid n \in d_i^j(\rho)(m)\} \quad [\text{defn.}] \\ &\supseteq \bigcup_{1 \leq j \leq k} \{(j, n) \mid n \in N_{(i,m)}^j\} \\ &= E_{IFG}^\rho((i, m)) \end{aligned}$$

Remark. The proof of case $e_j \equiv \sim\sigma(x)$ relies on the inflationary update in Algorithm 5.3.1. If we use the non-inflationary form instead (as in Roke’s predecessor, GenSet), then it is also possible for an edge (m, n) to be *removed* from σ_i . In this case, $N_{(i,m)}^j$ may also include elements under the old value of $\sigma_i(m)$ that are missing in the new one. Hence, $dep(\rho, i, \sim\sigma(x), m)$ would need to be modified to include also the nodes in $eval(\sigma_i(x), m, \rho')$, where ρ' is the environment just *before* the update to $\sigma_i(m)$.

□

Combining this with Lemmas 5.1.2 and 5.1.5, we have

Theorem 5.3.2. *On a monotone specification S over a lattice $\vec{\mathcal{D}}$ that satisfies ACC, Algorithm 5.3.1 terminates with $\vec{\sigma} = lfp(S)$.* □

5.4 Examples From the Literature

Example 5.4.1. We consider the construction of a solver for the “mutual update” phase of Duesterwald and Soffa’s analysis, as given above in Example 3.3.1.

The static evaluation of this analysis will construct the influence functions

$$\begin{aligned}
 d_{before}^{after}(\rho)(m) &= dep(\rho, before, \\
 &\quad after_glob(x) \text{ lub} \\
 &\quad (/lub y \text{ in } \sim before(x): after(w)) \text{ lub set}\{y\}), m) \\
 &= dep(\rho, before, after_glob(x), m) \\
 &\quad \cup dep(\rho, before, \\
 &\quad \quad /lub y \text{ in } \sim before(x): after(w)) \text{ lub set}\{y\}, m) \\
 &= \emptyset \cup dep(\rho, before, \sim before(x), m) \cup eval(before(x), m, \rho) \\
 &= eval(before(x), m, \rho) \cup eval(before(x), m, \rho) \\
 &= eval(before(x), m, \rho)
 \end{aligned}$$

$$\begin{aligned}
d_{before}^{before}(\rho)(m) &= dep(\rho, before, \\
&\quad before_glob(x) \text{ lub} \\
&\quad \quad (/lub w \text{ in } \sim after(x): before(w)) , m) \\
&= dep(\rho, before, before_glob(x), m) \\
&\quad \cup dep(\rho, before, \\
&\quad \quad /lub y \text{ in } \sim after(x): before(w)), m) \\
&= \emptyset \cup dep(\rho, before, \sim after(x), m) \cup eval(after(x), m, \rho) \\
&= \emptyset \cup \emptyset \cup eval(after(x), m, \rho) \\
&= eval(after(x), m, \rho)
\end{aligned}$$

$$\begin{aligned}
d_{after}^{before}(\rho)(m) &= dep(\rho, after, \\
&\quad before_glob(x) \text{ lub} \\
&\quad \quad (/lub w \text{ in } \sim after(x): before(w)) , m) \\
&= dep(\rho, after, before_glob(x), m) \\
&\quad \cup dep(\rho, after, \\
&\quad \quad /lub y \text{ in } \sim after(x): before(w)), m) \\
&= \emptyset \cup dep(\rho, after, \sim after(x), m) \\
&= eval(after(x), m, \rho)
\end{aligned}$$

$$\begin{aligned}
d_{after}^{after}(\rho)(m) &= dep(\rho, after, \\
&\quad after_glob(x) \text{ lub} \\
&\quad \quad (/lub y \text{ in } \sim before(x): after(w)) \text{ lub set}\{y\}, m) \\
&= dep(\rho, after, after_glob(x), m) \\
&\quad \cup dep(\rho, after, \\
&\quad \quad /lub y \text{ in } \sim before(x): after(w)) \text{ lub set}\{y\}, m) \\
&= \emptyset \cup dep(\rho, after, \sim before(x), m) \cup eval(before(x), m, \rho) \\
&= \emptyset \cup \emptyset \cup eval(before(x), m, \rho) \\
&= eval(before(x), m, \rho)
\end{aligned}$$

This gives us a ReQ^ρ function for Algorithm 5.3.1 that corresponds operationally to the following:

```

 $ReQ^\rho(i, m) =$ 
   $rq \leftarrow \emptyset$ 
  if  $i = before$ :
    for each  $n \in eval(before(x), m, \rho)$ : //  $d_{before}^{after}(\rho)(m)$ 
       $rq \leftarrow rq \cup (after, n)$ 
    for each  $n \in eval(after(x), m, \rho)$ : //  $d_{before}^{before}(\rho)(m)$ 
       $rq \leftarrow rq \cup (before, n)$ 
  else:
    for each  $n \in eval(after(x), m, \rho)$ : //  $d_{after}^{before}(\rho)(m)$ 
       $rq \leftarrow rq \cup (before, n)$ 
    for each  $n \in eval(before(x), m, \rho)$ : //  $d_{after}^{after}(\rho)(m)$ 
       $rq \leftarrow rq \cup (after, n)$ 
  return  $rq$ 

```

Example 5.4.2. The implementation of Callahan's interprocedural *must-kill* analysis that was given in Example 3.2.2 has the following influence functions:

$$\begin{aligned}
 d_{Kill_{sum}}^{Kill_{psg}}(\rho)(m) &= dep(\rho, Kill_{sum}, \\
 &\quad (/glb y \text{ in } \tilde{revPSG}(x): \\
 &\quad \quad g(Kill_{psg}(y), Kill_{sum}(y)) \text{) glb } i_{psg}(x), m) \\
 &= dep(\rho, Kill_{sum}, (/glb y \text{ in } \tilde{revPSG}(x): \\
 &\quad \quad g(Kill_{psg}(y), Kill_{sum}(y)) \text{) }, m) \\
 &\quad \cup dep(\rho, Kill_{sum}, i_{psg}(x), m) \\
 &= dep(\rho, Kill_{sum}, \tilde{revPSG}(x), m) \cup eval(revPSG(x), m, \rho) \cup \emptyset \\
 &= \emptyset \cup eval(revPSG(x), m, \rho) \cup \emptyset \\
 &= eval(revPSG(x), m, \rho)
 \end{aligned}$$

$$\begin{aligned}
d_{Kill_{sum}}^{Kill_{sum}}(\rho)(m) &= dep(\rho, Kill_{sum}, \\
&\quad (/glb y \text{ in } \hat{revSum}(x): \\
&\quad \quad g(Kill_{psg}(y), Kill_{sum}(y)) \text{) glb } i_{sum}(x), m) \\
&= dep(\rho, Kill_{sum}, (/glb y \text{ in } \hat{revSum}(x): \\
&\quad \quad g(Kill_{psg}(y), Kill_{sum}(y)) \text{) }, m) \\
&\quad \cup dep(\rho, Kill_{sum}, i_{sum}(x), m) \\
&= dep(\rho, Kill_{sum}, \hat{revSum}(x), m) \cup eval(revSum(x), m, \rho) \cup \emptyset \\
&= \emptyset \cup eval(revSum(x), m, \rho) \cup \emptyset \\
&= eval(revSum(x), m, \rho)
\end{aligned}$$

$$\begin{aligned}
d_{Kill_{psg}}^{Kill_{sum}}(\rho)(m) &= dep(\rho, Kill_{psg}, \\
&\quad (/glb y \text{ in } \hat{revSum}(x): \\
&\quad \quad g(Kill_{psg}(y), Kill_{sum}(y)) \text{) glb } i_{sum}(x), m) \\
&= dep(\rho, Kill_{psg}, (/glb y \text{ in } \hat{revSum}(x): \\
&\quad \quad g(Kill_{psg}(y), Kill_{sum}(y)) \text{) }, m) \\
&\quad \cup dep(\rho, Kill_{psg}, i_{sum}(x), m) \\
&= dep(\rho, Kill_{psg}, \hat{revSum}(x), m) \cup eval(revSum(x), m, \rho) \cup \emptyset \\
&= \emptyset \cup eval(revSum(x), m, \rho) \cup \emptyset \\
&= eval(revSum(x), m, \rho)
\end{aligned}$$

$$\begin{aligned}
d_{Kill_{psg}}^{Kill_{psg}}(\rho)(m) &= dep(\rho, Kill_{psg}, \\
&\quad (/glb y \text{ in } \hat{revPSG}(x): \\
&\quad \quad g(Kill_{psg}(y), Kill_{sum}(y)) \text{) glb } i_{psg}(x), m) \\
&= dep(\rho, Kill_{psg}, (/glb y \text{ in } \hat{revPSG}(x): \\
&\quad \quad g(Kill_{psg}(y), Kill_{sum}(y)) \text{) }, m) \\
&\quad \cup dep(\rho, Kill_{psg}, i_{psg}(x), m) \\
&= dep(\rho, Kill_{psg}, \hat{revPSG}(x), m) \cup eval(revPSG(x), m, \rho) \cup \emptyset \\
&= \emptyset \cup eval(revPSG(x), m, \rho) \cup \emptyset \\
&= eval(revPSG(x), m, \rho)
\end{aligned}$$

This gives us a ReQ^ρ function for Algorithm 5.3.1 that corresponds operationally to the following:

```

 $ReQ^\rho(i, m) =$ 
   $rq \leftarrow \emptyset$ 
  if  $i = Kill_{sum}$ :
    for each  $n \in eval(\text{revPSG}(x), m, \rho)$ : //  $d_{Kill_{sum}}^{Kill_{psg}}(\rho)(m)$ 
       $rq \leftarrow rq \cup (Kill_{psg}, n)$ 
    for each  $n \in eval(\text{revSum}(x), m, \rho)$ : //  $d_{Kill_{sum}}^{Kill_{sum}}(\rho)(m)$ 
       $rq \leftarrow rq \cup (Kill_{sum}, n)$ 
  else:
    for each  $n \in eval(\text{revSum}(x), m, \rho)$ : //  $d_{Kill_{psg}}^{Kill_{sum}}(\rho)(m)$ 
       $rq \leftarrow rq \cup (Kill_{sum}, n)$ 
    for each  $n \in eval(\text{revPSG}(x), m, \rho)$ : //  $d_{Kill_{psg}}^{Kill_{psg}}(\rho)(m)$ 
       $rq \leftarrow rq \cup (Kill_{psg}, n)$ 
  return  $rq$ 

```

Example 5.4.3. Khedker and Dhamdhere [KD94] extended the classical framework view to encompass both unidirectional and bidirectional analyses. Their framework accommodates *singular* data flow problems (*i.e.* those with a single confluence operator, as in Morel/Rennoise), but not nonsingular ones (*e.g.* the variant on MRA given as Example 3.2.4 above).

The core insight in their work is that the flow of information in a bidirectional analysis can be classified in four ways—through nodes and through edges, both forward and backward. Consequently, there are four types of transfer functions to consider. Writing g_{mn}^f (resp. g_{mn}^b) for forward (backward) edge flow functions and f_m^f (resp. f_m^b) for forward (backward) node functions, the solution to a given data flow problem (unidirectional or bidirectional) is the largest pair of assignments $(\sigma_{IN}, \sigma_{OUT})$ satisfying the equations ([KD94], p.1491)

$$\begin{aligned}\sigma_{IN}(x) &= \prod_{w \in Pred(x)} g_{wx}^f(\sigma_{OUT}(w)) \sqcap f_x^b(\sigma_{OUT}(x)) \sqcap C_{in}(x) \\ \sigma_{OUT}(x) &= \prod_{y \in Succ(x)} g_{xy}^b(\sigma_{IN}(y)) \sqcap f_x^f(\sigma_{IN}(x)) \sqcap C_{out}(x)\end{aligned}$$

A specification of this form gives rise to the following influence functions:

$$\begin{aligned}d_{\sigma_{IN}}^{\sigma_{IN}}(\rho)(m) &= dep(\rho, \sigma_{IN}, \left(\prod_{w \in Pred(x)} g_{wx}^f(\sigma_{OUT}(w)) \sqcap f_x^b(\sigma_{OUT}(x)) \sqcap C_{in}(x) \right), m) \\ &= dep(\rho, \sigma_{IN}, \left(\prod_{w \in Pred(x)} g_{wx}^f(\sigma_{OUT}(w)) \right), m) \\ &\quad \cup dep(\rho, \sigma_{IN}, f_x^b(\sigma_{OUT}(x)), m) \cup dep(\rho, \sigma_{IN}, C_{in}(x), m) \\ &= dep(\rho, \sigma_{IN}, Pred(x), m) \cup dep(\rho, \sigma_{IN}, \sigma_{OUT}(x), m) \cup \emptyset \\ &= \emptyset \cup \emptyset \cup \emptyset \\ &= \emptyset\end{aligned}$$

$$\begin{aligned}d_{\sigma_{IN}}^{\sigma_{OUT}}(\rho)(m) &= dep(\rho, \sigma_{IN}, \left(\prod_{y \in Succ(x)} g_{xy}^b(\sigma_{IN}(y)) \sqcap f_x^f(\sigma_{IN}(x)) \sqcap C_{out}(x) \right), m) \\ &= dep(\rho, \sigma_{IN}, \left(\prod_{y \in Succ(x)} g_{xy}^b(\sigma_{IN}(y)) \right), m) \\ &\quad \cup dep(\rho, \sigma_{IN}, f_x^f(\sigma_{IN}(x)), m) \cup dep(\rho, \sigma_{IN}, C_{out}(x), m) \\ &= dep(\rho, \sigma_{IN}, Succ(x), m) \cup eval(Pred(x), m, \rho) \\ &\quad \cup dep(\rho, \sigma_{IN}, \sigma_{IN}(x), m) \cup \emptyset \\ &= \emptyset \cup eval(Pred(x), m, \rho) \cup \{m\} \cup \emptyset \\ &= eval(Pred(x), m, \rho) \cup \{x\}\end{aligned}$$

$$d_{\sigma_{OUT}}^{\sigma_{IN}}(\rho)(m) = dep(\rho, \sigma_{OUT}, \left(\prod_{w \in Pred(x)} g_{wx}^f(\sigma_{OUT}(w)) \sqcap f_x^b(\sigma_{OUT}(x)) \sqcap C_{in}(x) \right), m)$$

$$\begin{aligned}
&= \text{dep}(\rho, \sigma_{OUT}, \left(\prod_{w \in \text{Pred}(x)} g_{wx}^f(\sigma_{OUT}(w)) \right), m) \\
&\quad \cup \text{dep}(\rho, \sigma_{OUT}, f_x^b(\sigma_{OUT}(x)), m) \cup \text{dep}(\rho, \sigma_{OUT}, C_{in}(x), m) \\
&= \text{dep}(\rho, \sigma_{OUT}, \text{Pred}(x), m) \cup \text{eval}(\text{Succ}(x)), m, \rho) \\
&\quad \cup \text{dep}(\rho, \sigma_{OUT}, \sigma_{OUT}(x), m) \cup \emptyset \\
&= \emptyset \cup \text{eval}(\text{Succ}(x)), m, \rho) \cup \{m\} \cup \emptyset \\
&= \text{eval}(\text{Succ}(x)), m, \rho) \cup \{x\} \\
d_{\sigma_{OUT}}^{\sigma_{OUT}}(\rho)(m) &= \text{dep}(\rho, \sigma_{OUT}, \left(\prod_{y \in \text{Succ}(x)} g_{xy}^b(\sigma_{IN}(y)) \sqcap f_x^f(\sigma_{IN}(x)) \sqcap C_{out}(x) \right), m) \\
&= \text{dep}(\rho, \sigma_{IN}, \left(\prod_{y \in \text{Succ}(x)} g_{xy}^b(\sigma_{IN}(y)) \right), m) \\
&\quad \cup \text{dep}(\rho, \sigma_{IN}, f_x^f(\sigma_{IN}(x)), m) \cup \text{dep}(\rho, \sigma_{IN}, C_{out}(x), m) \\
&= \text{dep}(\rho, \sigma_{IN}, \text{SUCC}(x), m) \cup \text{dep}(\rho, \sigma_{OUT}, \sigma_{IN}(x), m) \cup \emptyset \\
&= \emptyset \cup \emptyset \cup \emptyset \\
&= \emptyset
\end{aligned}$$

This gives us a solution algorithm for bidirectional bitvector data flow analysis problems using a ReQ^p function for Algorithm 5.3.1 that corresponds operationally to the following:

```

ReQp(i, m) =
  rq ← ∅
  if i = σIN:
    for each n ∈ (eval(Pred(x)), m, ρ) ∪ {x}: // dσINσOUT(ρ)(m)
      rq ← rq ∪ (σOUT, n)
  else:
    for each n ∈ (eval(Succ(x)), m, ρ) ∪ {x}: // dσOUTσIN(ρ)(m)
      rq ← rq ∪ (σIN, n)
  return rq

```

5.5 Related Work

The development of this chapter is an adaption and extension of earlier work carried out by Lasseter and Young [FLY02]. In that paper, we investigated the problem of influence determination for the GenSet language. GenSet is specialized to a single lattice of sets of atoms, ordered by \subseteq . Within this domain, it is somewhat more expressive than Roke, in that nested flow expressions are possible. However, relaxing this restriction breaks the normalization step of Section 5.3.1. Consequently, influence determination is coarser than in the present work, as no method was ever found to handle the case of complex index set expressions.

For our purposes, the work most pertinent to the problem of dependence graph determination is the body of research on *local solvers*, which have been applied mainly to the abstract interpretation of logic programs ([CH94, FS98, FS99, Jør94, VWL94], although see also [CHY95]). In this family, we are given a flow graph, but only want the solution at a particular node (or small subset of nodes). Like our approach, static determination of the dependence graph is bypassed, with the algorithm instead determining on the fly only those parts of the graph that are needed at each stage.

Without careful implementation, this “neededness” approach can result in a worst-case choice of node order for evaluation. Since evaluation at every node other than the chosen goal is delayed until it is needed, this can result in the slowest possible propagation of information from a flow graph’s entry point to the goal. If the workset is implemented as a priority queue, however, the use of timestamps to control the priority of a node within the queue can alleviate this shortcoming [FS98, FS99], and offers a good approximation of the topological ordering found in more conventional round robin solvers. The possibility of integrating the timestamp method with our hybrid approach offers an intriguing opportunity for future work.

CHAPTER VI

CONCLUSION AND FUTURE WORK

In this dissertation, we have shown how to restructure the traditional approach to data flow analyzer toolkits, in order to support the specification and efficient solution of multisource data flow problems. The results presented herein consist of three main contributions. First, we have extended the k -tuple lattice framework approach of Masticola *et al.* [MMR95], in order to facilitate expression of nonsingular analyses and other problems naturally expressed with a heterogeneous value lattice. Second, and more substantial, we have developed a domain-specific language-based approach to the specification of data flow problems that is expressive enough to encompass arbitrary multisource data flow problems. Finally, we have presented a hybrid static/dynamic method for the determination of influence in a constraint system, and we have shown how to leverage this to generate automatically an efficient solver for any specified analysis.

This approach opens several opportunities for future research. An obvious project is the construction of an industrial-strength implementation of Roke. Martin's PAG system [Mar98, Mar99] showed by example that one could generate data flow analyzers from concise specifications that are competitive with the best manually-constructed versions. The data type and auxiliary function definition mechanisms in Roke were heavily influenced by those in PAG, and bear more than a passing re-

semblance. Many of the data structures and implementation techniques used there should transfer easily to our system. For the Roke language itself, we would likely want to implement the type rules as a static system, in order to aid the selection of good data structures. Work on an implementation is ongoing at the time of this writing.

As part of this implementation, it will be worthwhile to investigate what other solution strategies are made available by our influence discovery technique. For example, the round robin algorithm, generalized to the Roke setting, would need to use a topological ordering (or weak ordering [Bou93]) on the influence graph. One approach to this would be the “generate and solve” method from constraint-based analysis. As discussed in Chapter V, however, a data flow problem whose IFG can change during solution as a consequence of intermediate results cannot be solved in this manner. It would be useful, therefore, to investigate the possibility of adapting some of the local solver algorithms to our approach, particular those that achieve weak topological orderings without requiring *a priori* construction of the IFG [CHY95, FS98, FS99].

On the theoretical end, there remain several opportunities for generalization of standard flow analysis properties to the multisource case. For example, the notion of a control flow graph’s *depth*—the maximum length of a sequence of back edge traversals with respect to a topological ordering of the flow graph nodes—has long played a role in the refinement of other solution algorithms. It is used to characterize the complexity of the round robin solver, for example, where Hecht and Ullman [HU75] showed a striking, essentially linear bound for the algorithm on the class of *bitvector* problems (which includes RD, LV, AE, and VB). Khedker and Dhamdhere [KD94] generalized this to the notion of *width*, which can also be used to reason about bidirectional analyses. Further generalization to the multisource case is an interesting challenge.

We also see an opportunity to make precise the relationship between the limited form of graph rewriting supported in Roke and the various uses in the literature of on-the-fly flow graph rewriting. There have been several works that incorporate information about the flow graph in the flow values themselves, collectively known as “qualified” analyses [CC79b, HR81, SP81]. Such “super-analyses” can be impor-

tant, because there are combinations of analyses and transformations which when run together produce more accurate analysis results and find more available transformations than can be obtained by running the component analyses in any sequential order [CC95, WZ91]. The result is the construction of more aggressive, but nonetheless safe, code-improving transformations. Further, although it has not received much attention, these approaches appear to be closely related to each other, particularly as presented in [HR81] and [LGC02].

The practical significance to the present discussion is twofold. On the one hand, such “super analyses” are even more cumbersome to construct than ordinary forms of flow analysis. Efforts in the formulation of flow analyses as restricted graph rewriting problems [Aßm00, LGC02, Rep98] offer some relief here, and have proven useful in both prototyping and final implementation. We conjecture that the concise specifications offered by an approach such as that of Roke would simplify the task even further. Conversely, these integrated approaches offer performance improvements to flow analysis solvers, by pruning away irrelevant parts of the flow graph model. Consequently, their extension to the multisource family offers another opportunity for the development of efficient solvers.

APPENDIX

EXAMPLE SPECIFICATIONS

Florian Martin described the basic goal of the PAG analyzer generator as allowing “the automatic generation of program analyzers from clear and concise specifications” ([Mar99], p. v). That spirit has driven the design of the Roke language, as well. Indeed, in those aspects that did not need further extension to the general multisource case, we have borrowed freely from Martin’s DATLA and FULA languages, as well as from GenSet [FLY02], the predecessor of Roke. This is particularly true of the language elements for definition of value lattices and transfer functions external to global abstract semantic specifications.

As of the writing of this dissertation, Roke is an abstract language, but it is nonetheless worth presenting some examples of complete specifications, to give the reader a feel for the conciseness of the language. In many of these cases, the same analyses have also seen implementations in the earlier GenSet language, which are also included here for the sake of completeness.

A.1 Live Variables Analysis (Roke)

```

/***** LiveVars.rk */
flowgraph is
  [flow]
with
  [gen:(atom set), kill:(atom set), isExit:bool]
end
domain is
  Vars = psetOf(progvars);
with
  progvars:atom = (rng gen) + (rng kill);
end

analysis is
  def genV:Vars = restrict(gen,Vars);
  def killV:Vars = restrict(kill,Vars);
  def revFlow(y:atom):(atom set) = ^flow(y);

  def f(y:atom,x:atom,v:Vars):Vars = (v ~ killV(y)) lub genV(y);
  for x in (base flow) def
    LV(x) = /lub y in ^revFlow(x): f(y,x,LV(y)) ;
  end
end

present is [LV] end

```

A.2 Callahan's *Kill* Analysis (Roke)

```

/***** Callahan_Ktuple_Kill.rk */
/* K-tuple form of Callahan's interprocedural Kill analysis */

flowgraph is
  [intraproc, // entry to call or exit, return to exit or call
  ip_param, // call->entry, exit->return
  ip_summary // call->return
  ]
with
  [isExit:bool, isCall:bool]
end

```

```

domain is
  Boolean;
end

analysis is
  def E_psg(x:atom):(atom set) = intraproc(x) | ip_param(x);

  def revPSG(y:atom):(atom set) = ^E_psg(y);
  def revSum(y:atom):(atom set) = ^ip_summary(y);

  def i_psg(x:atom):Boolean = if (isExit(y)) then bot else top;
  def i_sum(x:atom):Boolean = if (isCall(y)) then top else bot;
  def g(u:ToKill,v:ToKill):Boolean = u lub v;

  for x in ((base revPSG) | (base revSum)) def
    Kill_psg(x):Boolean <=
      ( /glb y in ^revPSG(x):
        g(Kill_psg(y),Kill_sum(y)) )
      glb i_psg(x);
    Kill_sum(x):Boolean <=
      ( /glb y in ^revSum(x):
        g(Kill_psg(y),Kill_sum(y)) )
      glb i_sum(x);
  end

  def Kill(x:atom):Boolean = Kill_psg(x) lub Kill_sum(x);

present is [ mustKill ] end

```

A.3 PRE Flow Equations (Roke)

```

/***** PRE.rk */

flowgraph is
  [flow]
with
  [genVB:(atom set), genAE:(atom set), kill:(atom set),
  AE:(atom set), isEntry:bool, isExit:bool]
end

domain is

```

```

    Exps = psetOf(progexps);
with
    progexps:atom = (rng gen) + (rng kill);
end

analysis is
////////////////////////////////////
// To begin, we'll need some elementary properties concerning the
// availability of various expressions

def antloc = restrict(genVB,Exps);
                // "locally anticipate" (upwards exposed)
def comp    = restrict(genAE,Exps); // downwards exposed
def transp  = complement(kill,Exps); // not killed
def avin    = restrict(AE,Exps);    // available at exit of node

def initF(x:atom):Exps = if (isEntry(x)) then bot else top;
def initB(x:atom):Exps = if (isExit(x)) then bot else top;

def avout(x:atom):Exps = (avin(x) glb transp(x)) lub comp(x);

def pavout(w:atom,x:atom,v:Exps):Exps =
                (v glb transp(w)) lub comp(w);
                // partially available at exit of node

for x in (base flow) def
    pav_in(x):Exps >=
                (/lub w in ^flow(x): pavout(w,x,pav_in(w)) )
                lub initF(x);
end

////////////////////////////////////
// Now the analysis itself

for x in (base flow) def
    pp_in(x):Exps <=
        if (isEntry(x)) then bot
        else ( pav_in(x)
                glb (antloc(x) lub
                    (pp_out(x) glb transp(y) )
                )
                glb

```

```

        /glb w in ^flow(x): (pp_out(w) lub avout(w))
    );
    pp_out(x):Exps <= if (isExit(x) then bot
        else (
            /glb y in flow(x): pp_in(y)
        );
end

for x in (base flow) def
    insert(x):Exps >= pp_forw(x) glb ^avout(x)
        glb ~(pp_out(x) glb transp(x));
    redund(x):Exps >= pp_in(x) glb antloc(x);
end

end // analysis

present is
    [insert, redund]
end

```

A.4 PRE Analysis, *K*-Tuple Form (Roke)

```

/***** PRE_Ktup.rk */

flowgraph is
    [flow]
with
    [genVB:(atom set), genAE:(atom set), kill:(atom set),
    AE:(atom set), isEntry:bool, isExit:bool]
end

domain is
    Exps = psetOf(progexps);
with
    progexps:atom = (rng gen) + (rng kill);
end

analysis is
    def antloc = restrict(genVB,Exps);
        // "locally anticipate" (upwards exposed)
    def comp = restrict(genAE,Exps); // downwards exposed

```

```

def transp = complement(kill,Exps); // not killed
def avin = restrict(AE,Exps);
// available at exit of node

def initF(x:atom):Exps = if (isEntry(x)) then bot else top;
def initB(x:atom):Exps = if (isExit(x)) then bot else top;

def avout(x:atom):Exps = (avin(x) glb transp(x)) lub comp(x);

def pavout(w:atom,x:atom,v:Exps):Exps =
    (v glb transp(w)) lub comp(w);
// partially available at exit of node

for x in (base flow) def
    pav_in(x):Exps >=
        (/lub w in ^flow(x): pavout(w,x,pav_in(w)) )
        lub initF(x);
end

def initF(x:atom):Exps = if (isEntry(x)) then bot else top;
def initB(x:atom):Exps = if (isExit(x)) then bot else top;

def pp_out(w:atom,x:atom,vb:Exps,vf:Exps):Exps = vb;
def pp_in(y:atom,x:atom,vb:Exps,vf:Exps):Exps =
    pav_in(y) glb ( (pp_out(y,x,vb,vf) glb transp(y))
        lub antloc(y) );
def f(w:atom,x:atom,vb:Exps,vf:Exps):Exps =
    pp_out(w,x,vb,vf) lub avout(w);

for x in (base flow) def
    pp_back(x):Exps <=
        ( /glb y in ^revFlow(x):
            pp_in(y,x,pp_back(y),pp_forw(y)) )
            glb initF(y);
    pp_forw(x):Exps <=
        ( /glb w in ^flow(x):
            f(w,x,pp_back(w),pp_forw(w)) )
            glb initB(x);
end

```

```

def insert(x:atom):Exps = pp_forw(x) glb ~avout(x)
                        glb ~(pp_back(x) glb transp(x)) ;
def redund(x:atom):Exps = pp_in(x) glb antloc(x) ;

present is
  [insert, redund]
end

```

A.5 Duesterwald/Soffa Ordering Analysis (Roke)

Roke specification of the “Can’t Happen Together” analysis, used in the race detection work of Duesterwald and Soffa [DS91].

```

/***** dues-soff_CHT-eqns.rk */

flowgraph is
  [cfg, sync, callflow]
with
  [isEntry:bool, isExit:bool, unit:atom]
end

domain is
  Nodes = psetOf(nodeset);
with
  nodeset:atom = base cfg;
end

analysis is
  // PHASE 1: Assuming source code text models control behavior
  //           (i.e. no parallel execution of any units)

  // STEP 1.1: compute "local" before and after sets

  for x in (base cfg) def
    may_before(x):Nodes >=
      /lub w in ~cfg(x): (may_before(w) lub set{w});
    may_after(x):Nodes >=
      /lub y in cfg(x): (may_after(y) lub set{y});
  end
end

```



```

for x in (base cfg) def
  before_loc(x):Nodes >= may_before(x) ~ may_after(x);
  after_loc(x):Nodes >= may_after(x) ~ may_before(x);

  gen(x):Nodes >= set{x} ~ may_after(x);
end

```

```
// STEP 1.2: Synchronization analysis
```

```

for x in (base cfg) def
  C_before_sync(x):Nodes <=
    if empty (~cfg(x)) then bot
    else /glb w in ^cfg(x):
      (C_before_sync(w) lub S_before_sync(w));
  S_before_sync(x):Nodes <=
    if empty (~sync(x)) then bot
    else /glb w in ^sync(x):
      ((C_before_sync(w) lub S_before_sync(w))
       lub (before_loc(w) lub gen(w))
      ) ;
end

```

```

for x in (base cfg) def
  C_after_sync(x):Nodes <=
    if empty (cfg(x)) then bot
    else /glb w in ^cfg(x):
      (C_after_sync(w) lub S_after_sync(w));
  S_after_sync(x):Nodes <=
    if empty (sync(x)) then bot
    else /glb w in ^sync(x):
      ((C_after_sync(w) lub S_after_sync(w))
       lub (after_loc(w) lub gen(w))
      ) ;
end

```

```

for x in (base cfg) def
  after_sync(x):Nodes >=
    (C_after_sync(x) lub S_after_sync(x))
    ~ after_loc(x);

```

```

    before_sync(x):Nodes >=
        (C_before_sync(x) lub S_before_sync(x))
        ~ before_loc(x);
end

// Completion of STEP 1.2:
// Combine control and sync flow ordering.

for x in (base cfg) def
    before_glob(x):Nodes <=
        if (isEntry(x) and not (empty callflow(x)) )
            then
                /glb w in ^callflow(x):
                    (before_glob(w) lub gen(w))
            else
                before_loc(x) lub before_sync(x) ;
        after_glob(x):Nodes <=
            if (isExit(x) and not (empty callflow(x)) )
                then
                    /glb w in callflow(x): after_glob(w)
                else
                    after_loc(x) lub after_sync(x) ;
            end
end

// STEP 3: The "mutual update", expressed equationally

for x in (base cfg) def
    before(x):Nodes >=
        before_glob(x) lub (/lub w in ^after(x): before(w));

    after(x):Nodes >=
        after_glob(x)
        lub (/lub y in ^before(x): after(y) lub set{y});
end

// PHASE 2: Computation of the Ord() relation,
//          with or without parallel execution

// Initial annotations:

def local(x:atom):(atom set) = ^unit(unit(x));
def localNodes = restrict(local,Nodes);

```

```

for x in (base cfg) def
  head(x):Nodes >=
    /lub h in local(x): (if isEntry(h) then set{h} else bot);
  head_callers(x):Nodes >= /lub h in head(x): ^callflow(h);

end
def Ord_init(x:atom):Nodes >=
  before(x) lub after(x) lub localNodes(x);

// Direct translation of Eqn #5 (p. 43), from their paper:

for x in (base cfg) def
  unit_filter(x):Nodes <=
    if (empty head_callers(x)) then top
    else
      /glb c in head_callers(x): Ord(c) ;

  Ord(x):Nodes <= Ord_init(x) glb unit_filter(x) );
end

end // analysis

present is [Ord] end

```

A.6 Ordering Analysis (GenSet)

GenSet specification of the “Can’t Happen Together” analysis, used in the race detection work of Duesterwald and Soffa [DS91].

```

// Equations from Duesterwald and Soffa [TAV '91].
// Accompanying example file is duesterwald91tav_MIG_Fig2.rsf

// PHASE 1: Assuming source code text models control behavior
//           (i.e. no parallel execution of any units)

// STEP 1.1: compute "local" before and after sets

```

```

for x in (base cfg) do
  may_before(x) :=
    /union w in _cfg(x): (may_before(w) union single(w));
  may_after(x) :=
    /union y in cfg(x): (may_after(y) union single(y));
od;

for x in (base cfg) do
  before_loc(x) := may_before(x) - may_after(x);
  after_loc(x) := may_after(x) - may_before(x);

  gen(x) := single(x) - may_after(x);
od;

// STEP 1.2: Synchronization analysis

for x in (base cfg) do
  C_before_sync(x) :=
    most (
      if empty (_cfg(x))
      then null
      // else /intersect w in _cfg(x): before_sync(w)
      else /intersect w in _cfg(x):
        (C_before_sync(w) union S_before_sync(w))
      fi );
  S_before_sync(x) :=
    most (
      if empty (_sync(x))
      then null
      else
        /intersect w in _sync(x):
          //( before_sync(w)
          ( (C_before_sync(w) union S_before_sync(w))
            union (before_loc(w) union gen(w))
          )
        fi );
od;

for x in (base cfg) do
  C_after_sync(x) :=
    most (
      if empty (cfg(x))

```

```

        then null
        // else /intersect w in cfg(x): after_sync(w)
        else /intersect w in cfg(x):
            (C_after_sync(w) union S_after_sync(w))
        fi );
S_after_sync(x) :=
    most (
        if empty (sync(x))
        then null
        else
            /intersect w in sync(x):
                // ( after_sync(w)
                ( (C_after_sync(w) union S_after_sync(w))
                  union (after_loc(w) union gen(w))
                )
            fi );
od;

for x in (base cfg) do
    after_sync(x) := (C_after_sync(x) union S_after_sync(x))
                    - after_loc(x);
    before_sync(x) := (C_before_sync(x) union S_before_sync(x))
                     - before_loc(x);
od;

// Completion of STEP 1.2:
// Combine control and sync flow ordering.

for x in (base cfg) do
    before_glob(x) :=
        most (
            if not empty (extremal(x) intersect single("start"))
            and not empty (_callflow(x))
            // if x is a start node
            then
                (/intersect w in _callflow(x):
                    (before_glob(w) union gen(w))
                )
            else
                before_loc(x) union before_sync(x)
            fi
        );

```

```

after_glob(x) :=
  most (
    if not empty (extremal(x) intersect single("stop"))
      and not empty (callflow(x))
      // if x is an exit node
    then
      (/intersect w in callflow(x): after_glob(w) )
    else
      after_loc(x) union after_sync(x)
    fi
  );
od;

// STEP 3: The "mutual update", expressed equationally

for x in (base cfg) do
  before(x) := before_glob(x)
              union (/union w in _after(x): before(w));

  after(x) :=
    after_glob(x)
    union (/union y in _before(x): after(y) union single(y));
od;

// PHASE 2: Computation of the Ord() relation,
//           with or without parallel execution

// Initial annotations:

for x in (base cfg) do
  local(x) := /union U in unit(x): _unit(U);
  head(x) := local(x) intersect _extremal("start");
  head_callers(x) := /union h in head(x): _callflow(h);
  Ord_init(x) := before(x) union after(x) union local(x);

od;

// Direct translation of Eqn #5 (p. 43), from their paper:
for x in (base cfg) do
  unit_filter(x) :=
    most(

```

```
        if empty (head_callers(x))
          then (base cfg)
          else /intersect c in head_callers(x): Ord(c)
        fi
      Ord(x) := most( Ord_init(x) intersect unit_filter(x) );
    od;
```

BIBLIOGRAPHY

- [AC76] F.E. Allen and J. Cocke. A program data flow analysis procedure. *Comm. ACM*, 19(3):137–147, March 1976.
- [AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [Aik99] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2–3):79–111, 1999.
- [AM95] Martin Alt and Florian Martin. Efficient generation of interprocedural analyzers with PAG. In *Static Analysis (SAS '95), 2nd International Symposium*, pages 33–50. Springer-Verlag, 1995. LNCS 983.
- [Aßm00] Uwe Abßmann. Graph rewrite systems for program optimization. *ACM TOPLAS*, 22(4):583–637, July 2000.
- [ASU86] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [AU73] Alfred Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. 2: Compiling*. Prentice Hall, Englewood Cliffs, NJ USA, 1973.
- [AV91] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, August 1991.
- [BGS97] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Foundations of Software Engineering (SIGSOFT '97), 5th International Symposium*, pages 361–377. Springer-Verlag, 1997. LNCS 1301.
- [BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Principles of Database Systems (PODS '86), 5th ACM Symposium*, pages 1–15. ACM Press, 1986.

- [Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Formal Methods in Programming and their Applications, 1993 Intl. Conference.*, pages 128–141. Springer-Verlag, 1993. LNCS 735.
- [Cal88] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Programming Language Design and Implementation (PLDI '88), 1988 ACM SIGPLAN Conference*, pages 47–56. ACM Press, 1988.
- [CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL '77), Proc. of the 4th ACM Symposium*, pages 238–252. ACM Press, 1977.
- [CC77b] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Artificial Intelligence and Programming Languages, Proc. of the 1977 ACM SIGART Symposium*, pages 1–12. ACM Press, 1977.
- [CC79a] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.
- [CC79b] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL '79), Proc. of the 6th ACM Symposium*, pages 269–282. ACM Press, 1979.
- [CC95] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM TOPLAS*, 17(2):181–196, March 1995.
- [CCF⁺05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In M. Sagiv, editor, *2005 European Symposium on Programming (ESOP '05)*, pages 21–30. Springer-Verlag, 2005. LNCS 3444.
- [CDG96] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for intra- and interprocedural analysis. Technical Report 96-11-02, University of Washington, November 1996.
- [CFH⁺91] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In B. Möller, editor, *Constructing Programs from Specifications*, pages 125–164. Elsevier Science Publishers B.V., Amsterdam, 1991.

- [CFT03] Larry Carter, Jeanne Ferrante, and Clark Thomborson. Folklore confirmed: Reducible flow graphs are exponentially larger. In *Principles of Programming Languages (POPL '03)*, 30th ACM Symposium, pages 106–114. ACM Press, 2003.
- [CH94] Baudouin Le Charlier and Pascal Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for PROLOG. *ACM TOPLAS*, 16(1):35–101, 1994.
- [Che03] Yifeng Chen. A fixpoint theory for non-monotonic parallelism. *Theoretical Computer Science*, 308(1–3):367–392, November 2003.
- [CHY95] Li-Ling Chen, Williams L. Harrison, and Kwangkeun Yi. Efficient computation of fixpoints that arise in complex program analysis. *Journal of Programming Languages*, 3(1):31–68, 1995.
- [CK94] Shing Chi Cheung and Jeff Kramer. Tractable data flow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, August 1994.
- [CKS90] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Principles and Practice of Parallel Programming (PPoPP '90)*, 2nd ACM Symposium. ACM Press, 1990.
- [Cor00] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. *ACM TOSEM*, 9(1):51–93, 2000.
- [CP87] Jiazhen Cai and Robert Paige. Binding performance at language design time. In *Principles of Programming Languages, 14th ACM Symposium (POPL '87)*, pages 85–97. ACM Press, 1987.
- [CP93] Jiazhen Cai and Robert Paige. Towards increased productivity of algorithm implementation. In *Foundations of Software Engineering, 1993 ACM Symposium (SIGSOFT '93)*, pages 71–78. ACM Press, 1993.
- [CP89] Jiazhen Cai and Robert Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(4):197–261, 1988/89.
- [CS88] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Parallel and Distributed Debugging, 1988 ACM/SIGOPS Workshop*, pages 100–111. ACM Press, 1988.

- [DC94] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Foundations of Software Engineering (SIGSOFT '94)*, 2nd ACM Symposium, pages 62–75. ACM Press, 1994.
- [DC96] Matthew B. Dwyer and Lori A. Clarke. A flexible architecture for building data flow analyzers. In *Software Engineering (ICSE '96)*, 18th International Conference, pages 554–564. IEEE Computer Society Pr., Los Alamitos CA, USA, 1996.
- [DCCN04] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM TOSEM*, 13(4):359–430, October 2004.
- [Dha91] D. M. Dhamdhere. Practical adaption of the global optimization algorithm of Morel and Renvoise. *ACM TOPLAS*, 13(2):291–294, April 1991.
- [DP93] D. M. Dhamdhere and H. Patil. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM TOPLAS*, 15(2):312–336, April 1993.
- [DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order (2nd Ed.)*. Cambridge U. P., 2002.
- [DRW96] Steven Dawson, C.R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Programming Language Design and Implementation (PLDI '96)*, 9th ACM SIGPLAN Conference, pages 117–126, 1996.
- [DRZ92] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *Programming Language Design and Implementation (PLDI '92)*, 5th ACM SIGPLAN Conference, pages 212–223. ACM Press, 1992.
- [DS91] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Symposium on Testing, Analysis, and Verification (TAV '91)*, pages 36–48. ACM Press, 1991.
- [Dwy95] Matthew B. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs*. PhD thesis, University of Massachusetts Amherst, Amherst, MA, USA, 1995.

- [Ete04] K. Etessami. Analysis of recursive game graphs using data flow equations. In B. Steffen and G. Levi, editors, *Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, Fifth International Conference, pages 282–296. Springer-Verlag, 2004. LNCS 2937.
- [FBG03] J.-C. Fernandez, M. Bozga, and L. Ghirvu. State space reduction based on live variables analysis. *Science of Computer Programming*, 47(2–3):203–220, May–June 2003.
- [Fer04] Jérôme Feret. Static analysis of digital filters. In D. Schmidt, editor, *Programming Languages and Systems (ESOP '04)*, pages 33–48. Springer-Verlag, 2004. LNCS 2986.
- [Fer05] Jérôme Feret. The arithmetic-geometric progression abstract domain. In R. Cousot, editor, *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, Sixth International Conference, pages 42–58, 2005. LNCS 3385.
- [FLY02] John Fiskio-Lasseter and Michal Young. Flow equations as a generic programming tool for manipulation of attributed graphs. In *Program Analysis for Software Tools and Engineering (PASTE '02)*, 4th ACM Workshop, pages 69–76. ACM Press, 2002.
- [FS98] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *Nordic Journal of Computing*, 5(4):304–329, 1998.
- [FS99] Christian Fecht and Helmut Seidl. A faster solver for general systems of equations. *Science of Computer Programming*, 35(2–3):137–161, 1999.
- [GKL+96] Alfons Geser, Jens Knoop, Gerald Lüttgen, Oliver Rüthing, and Bernhard Steffen. Non-monotone fixpoint iterations to resolve second order effects. In *Compiler Construction (CC '96)*, 6th International Conference, LNCS 1060, pages 106–120. Springer-Verlag, 1996.
- [Goy00] Deepak Goyal. *A Language-Theoretic Approach to Algorithms*. PhD thesis, New York University, New York, NY USA, 2000.
- [Grä98] George Grätzer. *General Lattice Theory*. Birkhäuser Verlag, Boston, MA USA, 1998. 2nd Ed.
- [GS93] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Principles and Practice of Parallel Programming (PPoPP '93)*, Proc. of the 4th ACM Symposium, pages 159–168. ACM Press, 1993.

- [GW76] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.
- [HDT87] Susan Horwitz, Alan Demers, and Tim Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.
- [Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Publishers B.V., Amsterdam, 1977.
- [HMCCR93] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and René G. Rodríguez. FIAT: A framework for interprocedural analysis and transformation. In *Proc. 6th Workshop on Parallel Languages and Compilers*, pages 522–545. Springer-Verlag, 1993. LNCS 768.
- [HR81] L. H. Holley and B. K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, SE-7(1):60–78, January 1981.
- [HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Foundations of Software Engineering (SIGSOFT '95), 3rd International Symposium*, pages 104–115. ACM Press, 1995.
- [HU72] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. *SIAM Journal on Computing*, 1(2):188–202, June 1972.
- [HU74] Matthew S. Hecht and Jeffrey D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, July 1974.
- [HU75] Matthew S. Hecht and Jeffrey D. Ullman. A simple algorithm for global data flow analysis. *SIAM Journal on Computing*, 4(4):519–532, December 1975.
- [Imm99] Neil Immerman. *Descriptive Complexity*. Springer-Verlag, 1999.
- [JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. Available online at <http://www.dina.kvl.dk/~sestoft/pebook/>.
- [Jør94] Niels Jørgensen. Finding fixpoints in finite function spaces using needness analysis and chaotic iteration. In *Static Analysis (SAS '94), 1st International Symposium*, pages 329–345. Springer-Verlag, 1994. LNCS 864.

- [KA05] John Kodumal and Alex Aiken. Banshee: A scalable constraint-based analysis toolkit. In C. Hankin and I. Silveroni, editors, *Static Analysis (SAS '05), 12th International Symposium*, pages 218–234. Springer-Verlag, 2005. LNCS 3672.
- [KBC⁺99] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujan, and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM TOPLAS*, 21(6):1251–1297, November 1999.
- [KD94] U. P. Khedker and D. M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM TOPLAS*, 16(5):1472–1511, September 1994.
- [KD99] U. P. Khedker, , and D. M. Dhamdhere. Bidirectional data flow analysis: Myths and reality. *ACM SIGPLAN Notices*, 34(6):47–57, June 1999.
- [KDM03] Uday P. Khedker, Dhananjay M. Dhamdhere, and Alan Mycroft. Bidirectional data flow analysis for type inferencing. *Computer Languages, Systems, and Structures*, 29(1–2):15–44, April–July 2003.
- [Ken75] Kenneth W. Kennedy. Node listings applied to data flow analysis. In *Principles of Programming Languages (POPL '75), 2nd ACM Symposium*, pages 10–21. ACM Press, 1975.
- [Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Principles of Programming Languages (POPL '73), 1st ACM Symposium*, pages 194–206. ACM Press, 1973.
- [KKKS96] Marion Klein, Jens Knoop, Dirk Koschützski, and Bernhard Steffen. DFA & OPT-METAFRAME: A toolkit for program analysis and optimization. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 2nd International Workshop (TACAS '96)*, pages 422–426. Springer-Verlag, 1996. LNCS 1055.
- [Kle52] Stephen C. Kleene. *Introduction to metamathematics*. North-Holland, Amsterdam, NL, 1952.
- [Kno98] Jens Knoop. *Optimal Interprocedural Program Optimization: A New Framework and its Applications*. Springer-Verlag, 1998. LNCS 1428.
- [KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *Programming Language Design and Implementation (PLDI '94), 8th ACM SIGPLAN Conference*, pages 147–158. ACM Press, 1994.

- [KRS96] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *Journal of Programming Languages*, 4(4):211–246, 1996.
- [KSV96] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM TOPLAS*, 18(3):268–299, May 1996.
- [KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.
- [KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
- [KU80] Marc A. Kaplan and Jeffrey D. Ullman. A scheme for the automatic inference of variable types. *Journal of the ACM*, 27(1):128–145, January 1980.
- [Las04] John H. E. F. Lasseter. Toolkits for the automatic construction of data flow analyzers. Technical Report CIS-TR-04-03, University of Oregon, 2004.
- [Las05] John H. E. F. Lasseter. Notes on the algebraic formulation and automatic implementation of Duesterwald and Soffa’s data flow-based concurrency analysis. Unpublished manuscript, January 2005.
- [LC91] D. Long and L.A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Symposium on Testing, Analysis, and Verification (TAV '91)*, pages 21–35. ACM Press, 1991.
- [LGC02] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Principles of Programming Languages (POPL '02), 29th ACM Symposium*, pages 270–282. ACM Press, 2002.
- [LNS82] J.-L. Lassez, V.L. Nguyen, and E.A. Sonenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 14(3):112–116, 16 May 1982.
- [LPM99] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *Principles and Practice of Parallel Programming (PPoPP '99), 7th ACM Symposium*, pages 1–12. ACM Press, 1999.
- [Mar98] Florian Martin. PAG—An efficient program analyzer generator. *Software Tools for Technology Transfer*, 2(1):46–67, 1998.

- [Mar99] Florian Martin. *Generating Program Analyzers*. PhD thesis, Universität des Saarlandes, 1999.
- [Mas93] Stephen P. Masticola. *Static Detection of Deadlocks in Polynomial Time*. PhD thesis, Rutgers University, 1993.
- [McA02] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, July 2002.
- [Mer92] N. Mercouroff. An algorithm for analyzing communicating processes. In *Mathematical Foundations of Programming Semantics, 7th International Conf.*, pages 312–325. Springer-Verlag, 1992. LNCS 598.
- [Min01] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In O. Danvy and A. Filinski, editors, *Proc. 2nd Symp. on Program as Data Objects (PADO '01)*, pages 155–172. Springer-Verlag, 2001. LNCS 2053.
- [Min04] Antoine Miné. Relational abstract domains for the detection of floating-point runtime errors. In D. Schmidt, editor, *Programming Languages and Systems (ESOP '04)*, pages 3–17. Springer-Verlag, 2004. LNCS 2986.
- [MJ81] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [MMR95] Stephen P. Masticola, Thomas J. Marlowe, and Barbara G. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM TOPLAS*, 17(5):777–803, September 1995.
- [MR79] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Comm. ACM*, 22(2):96–103, February 1979.
- [MR90a] Thomas J. Marlowe and Barbara G. Ryder. Properties of dataflow frameworks: A unified model. *Acta Informatica*, 28(2):121–163, 1990.
- [MR90b] Stephen P. Masticola and Barbara G. Ryder. Static infinite wait anomaly detection in polynomial time. In *2nd International Conf. on Parallel Processing (ICPP '90)*, pages 78–187. U. Pennsylvania Pr., 1990.
- [MR91] Stephen P. Masticola and Barbara G. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 97–107. ACM Press, 1991.

- [MR93] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '93)*, pages 129–138. ACM Press, 1993.
- [Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, San Francisco, CA USA, 1997.
- [MY02] Andrzej S. Murawski and Kwangkeun Yi. Static monotonicity analysis for λ -definable functions over lattices. In Agostino Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI '02)*, 3rd International Workshop, pages 139–153. Springer-Verlag, 2002. LNCS 2294.
- [NA98] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Foundations of Software Engineering (SIGSOFT '98)*, 6th International Symposium, pages 24–34. ACM Press, 1998.
- [NAC99] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Foundations of Software Engineering (SIGSOFT '99)*, 7th International Symposium, pages 338–354. ACM Press, 1999.
- [NACO97] Gleb Naumovich, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Applying static analysis to software architectures. In *Foundations of Software Engineering (SIGSOFT '97)*, 5th International Symposium, pages 77–93. Springer-Verlag, 1997. LNCS 1301.
- [NLM95] Magnus Nordin, Thomas Lindgren, and Håkan Millroth. IGOR: A tool for developing Prolog dataflow analyzers. Technical Report 111, Uppsala University, 1995.
- [NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, New York, 1999.
- [OJP99] Mauricio Osorio, Bharat Jayaraman, and David A. Plaisted. Theory of partial-order programming. *Science of Computer Programming*, 34(3):207–238, 1999.
- [OO90] Kurt M. Olender and Leon J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.
- [OO92] Kurt M. Olender and Leon J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM TOSEM*, 1(1):21–52, January 1992.

- [PP91] Wuxu Peng and S. Puroshothaman. Data flow analysis of communicating finite state machines. *ACM TOPLAS*, 13(3):399–442, July 1991.
- [Rep94a] Thomas Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Pr., 1994.
- [Rep94b] Thomas Reps. Solving demand versions of interprocedural analysis problems. In *Compiler Construction (CC '94), 5th International Conference*, pages 389–403. Springer-Verlag, 1994. LNCS 786.
- [Rep98] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, November/December 1998.
- [Ros77] Barry K. Rosen. High level data flow analysis. *Comm. ACM*, 20(10):712–724, October 1977.
- [Ros80] Barry K. Rosen. Monoids for rapid data flow analysis. *SIAM Journal on Computing*, 9(1):159–196, February 1980.
- [Ros82] Barry K. Rosen. A lubricant for data flow analysis. *SIAM Journal on Computing*, 11(3):493–511, August 1982.
- [RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, 1986.
- [RS90] John H. Reif and Scott A. Smolka. Data flow analysis of distributed communicating processes. *Int. Journal of Parallel Programming*, 19(1):1–30, 1990.
- [Sch73] Marvin Schaeffer. *A Mathematical Theory of Global Program Optimization*. Prentice Hall, Englewood Cliffs, NJ USA, 1973.
- [SDDS86] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming With Sets: An Introduction to SETL*. Springer-Verlag, 1986.
- [SGL98] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. A new framework for elimination-based data flow analysis using DJ graphs. *ACM TOPLAS*, 20(2):388–435, March 1998.
- [Sha80] Micha Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.
- [SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In [MJ81], pages 189–233, 1981.

- [SRH95] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Formal Approaches in Software Engineering (FASE '95), 1995 Colloquium*, pages 651–665. Springer-Verlag, 1995. LNCS 915.
- [SS93] Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs*. Springer-Verlag, 1993.
- [Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
- [Tar76] Robert Endre Tarjan. Iterative algorithms for global flow analysis. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Research Results*, pages 71–101. Academic Pr., 1976.
- [Tar81a] Robert Endre Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.
- [Tar81b] Robert Endre Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.
- [Tay83] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19(1):57–83, 1983.
- [Ten74] Aaron M. Tenenbaum. *Type Determination for Very High-Level Languages*. PhD thesis, New York University, New York, NY USA, 1974.
- [TH92] Steven W.K. Tjiang and John L. Hennessy. Sharlit: A tool for building optimizers. In *Programming Language Design and Implementation (PLDI '92), 5th ACM SIGPLAN Conference*, pages 82–93. ACM Press, 1992.
- [TO80] Richard N. Taylor and Leon J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, SE-6(3):265–277, 1980.
- [Ull73] Jeffrey D. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2:191–213, 1973.
- [Ven92] G. A. Venkatesh. SPARE: A development environment for program analysis algorithms. *IEEE Transactions on Software Engineering*, 18(4):304–318, April 1992.
- [VWL94] B. Vergauwen, J. Wauman, and J. Lewi. Efficient fixpoint computation. In *Static Analysis (SAS '94), 1st International Symposium*, pages 314–328. Springer-Verlag, 1994. LNCS 864.

- [War92] D. S. Warren. Memoing for logic programs. *Comm. ACM*, 35(3):93–111, March 1992.
- [WS94] Deborah L. Whitfield and Mary Lou Soffa. The design and implementation of Genesis. *Software—Practice and Experience*, 24(3):307–325, March 1994.
- [WS97] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code-improving transformations. *ACM TOPLAS*, 19(6):1053–1084, November 1997.
- [WZ91] Mark N. Wegman and Frank K. Zadeck. Constant propagation with conditional branches. *ACM TOPLAS*, 13(2):181–210, April 1991.
- [YE02] Kwangkeun Yi and Hyunjun Eo. Static extensionality analysis for λ -definable functions over lattices. Technical Report ROPAS-2002-17, Korea Advanced Institute of Science and Technology, 2002. Available online at <http://ropas.snu.ac.kr/zoo/>.
- [YH93] Kwangkeun Yi and Williams L. Harrison. Automatic generation and management of interprocedural program analyses. In *Principles of Programming Languages (POPL '93), 20th ACM Symposium*, pages 93–103. ACM Press, 1993.
- [ZME06] Jia Zeng, Chuck Mitchell, and Stephen A. Edwards. A domain-specific language for generating dataflow analyzers. *Electronic Notes in Theoretical Computer Science*, 164:103–119, 2006.
- [ZYL04] Xiaofang Zhang, Michal Young, and John H. E. F. Lasseter. Refining code-design mapping with flow analysis. In *Foundations of Software Engineering, 2004 ACM Symposium (SIGSOFT '04)*, pages 231–240. ACM Press, 2004.