TREE-BASED CODE OPTIMIZATION

BY

CARL D. MCCONNELL

(mcconnel@cs.uiuc.edu)
DRAFT OF May 12, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Control Flow Graphs Considered Harmful

Optimizing compilers typically consist of a front-end that does syntactic and semantic analysis, and a back-end that does code optimization and machine code generation [ASU86]. The main data structure in the front-end is an abstract syntax tree (AST), while in the back-end it is a control flow graph (CFG). The CFG consists of nodes representing computations and edges representing control flow. To convert the AST into a CFG, the compiler must essentially "flatten" it into a tangle of GOTOs (edges), thus losing a great deal of information: unlike an AST, a CFG cannot directly express the hierarchical organization of a program, nor the high-level control structures like WHILE-DO and IF-THEN that impose this organization.

But it is exactly these high-level control structures that are a central concern in code optimization! For example, loops are important to optimizations such as code motion and strength reduction. However, since they are not explicit in CFGs, they must be recovered one way or another. Even register allocation can benefit from knowledge of a program's hierarchical control structure [CK91]. CFGs oblige us to rediscover this structure, which was apparent in the AST. If we could somehow avoid this salvage effort, we would end up with simpler optimization algorithms, and hence simpler compilers.

In this dissertation I contend that ASTs are superior to CFGs for purposes of code optimization because they yield simpler algorithms. In particular, I propose using the ASTs of a certain simple intermediate language. This intermediate language is suitable for low-level optimization, yet is high-level enough to retain the structure present in the original program. As we will see, the code optimization algorithms for these ASTs are easier to understand and implement than their CFG-based counterparts. This is because the structure inherent in an AST makes unnecessary many concepts used to impose structure on CFGs, such as depth-first spanning trees, dominators, and intervals.

Powell [Pow84] comments that "A survey of the literature on optimization uncovers many techniques that are hard to understand, harder to implement, more general than necessary, and of marginal value." I agree with this assessment, and believe that the CFG is the main cause of the first two afflictions.

1.1 Kicking the Control Flow Graph Habit

Why are CFGs treated as a *sine qua non* of code optimization? One reason is probably historical in nature. As early as 1946, researchers chose to describe programs using flow charts [Knu73], a decision that subsequent workers apparently never really reconsidered, even when high-level programming languages became commonplace. Another more important reason for the prevalence of CFGs is that they describe control flow in a way that is easy to express with the GOTO-like branching instructions computers possess. Yet another reason is the infamous GOTO statement itself, which is a feature of many popular programming languages. Although ASTs can and do express GOTOs, their tree structure does not reflect the potential flow of control: by definition, each sub-tree in an AST has a single entry point, but GOTOs can cause control to jump into the middle of a sub-tree, although this almost never happens in practice. On the other hand, CFGs nicely describe the arbitrary control flow that GOTOs can induce, and so all other control structures are expressed in terms of GOTO as well. Thus, the GOTO tail wags the control structure dog [Ros77]: high-level control structures are sacrificed for the sake of GOTOs.

(The preceding remarks concerning the hegemony of CFGs are only true in the procedural language community. The Lisp/Scheme community uses ASTs in their compilers [Ste78] [Kra88]. For the most part, however, code optimization to them means applying closure analysis techniques to reduce a program to a procedural form, at which point their compilers generate code. They do not discuss how to perform traditional optimizations using ASTs. Thus, their point of arrival is our point of departure.)

As Powell says [Pow84], "The value of constructing flow graphs for languages or programs without GOTO statements (as well as the payoff from considering all the perverse possibilities in languages that do have them) is minimal." One way to avoid constructing CFGs, and thus to avoid sacrificing control structures, is to use the AST of the front-end in the back-end as well. However, there are two problems with this approach. One is that this AST is language-dependent, while the back-end should be language-independent. The other problem is that the control structures described in this AST are generally too high-level to expose opportunities for low-level optimization: a Pascal FOR-DO statement, for example, does not correspond to an instruction on a typical computer.

To solve these problems, I introduce an intermediate language that for convenience I will call CORTL, for "Control-Oriented Register Transfer Language." Figure 1.1 shows the abstract syntax of CORTL. The conditional, assignment, and sequencing statements play the usual roles. The `region` and `depart` constructs are similar to the label and `leave` constructs of Bliss [WJW+75]: `region` I S declares a block of code named I consisting of S, and `depart` I jumps to the statement after the region. On the other hand, `repeat` I jumps to S, the statement *inside* the region, thus providing a means of looping; there is no loop construct as such. Both `depart` and `repeat` statements may only refer to enclosing `region` statements.

CORTL can easily express the control structures used in languages such as C or Fortran without forfeiting the hierarchical organization they impose on a program, and a CORTL AST is at least as easy to generate from the source language AST as a CFG is. At the same time, the CORTL control constructs are much like the branching instructions that computers have. This allows CORTL ASTs to be used for low-level optimization in the back-end.

By way of proof that CORTL can express the control structures of any procedural language, Figure 1.2 shows the CORTL equivalents of the standard ones. The templates in the left column

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{region}\ I\ S \\
 & | & \texttt{depart}\ I \\
 & | & \texttt{repeat}\ I \\
 & | & \texttt{if}\ E\ \texttt{then}\ S \\
 & | & S\ S \\
 & | & I\ \texttt{:=}\ E \\
 & & \\
I & \rightarrow & \text{identifier} \\
E & \rightarrow & \text{expression}
\end{array}
$$

Figure 1.1: Abstract syntax of CORTL

span the range of non-GOTO constructs used in procedural languages such as Fortran and Pascal. (Although not shown, GOTO-like constructs such as the **break** and **continue** statements of the C programming language are obviously expressible as `depart` and `repeat` statements.) These examples show that translating a control structure into CORTL only increases its size by a small additive factor. Thus, translating a program into CORTL will not result in an unacceptable (e.g., polynomial) expansion in size.

The one control construct that is not always expressible in CORTL is the GOTO. GOTOs come in two varieties: structured (those that do not jump into the middle of control structures) and unstructured (those that do). Translating structured GOTOs into `depart` and `repeat` statements poses no problem. On the other hand, unstructured GOTOs clash with the premise underlying CORTL: that control structures are entered at the top. Therefore, they must be turned into structured GOTOs. This can be done by duplicating code ("node splitting"), which can cause an exponential increase in program size, but is unlikely to do so in practice. An algorithm appearing later in this chapter shows how to translate programs having both kinds of GOTOs into CORTL programs.

Programs can be classified according to their control flow structure as follows, from most to least general:

- *Unstructured* programs have unstructured GOTOs.
- *Reducible* programs[1] have no GOTOs into the middle of loops [Hec77].
- *Tree-structured* programs [Weg73] are composed only of looping, conditional, sequencing, and multi-level loop exiting constructs; alternatively, they are reducible programs having no GOTOs into the middle of conditionals. All CORTL programs are tree-structured. (In passing, notice that tree-structured programs are exactly those that can be written using only downward-passed, statement-level continuations. This observation, made with regard to the Smalltalk programming language, was the inspiration for this dissertation, although we will not make use of it.)
- *Semi-structured* programs [FKZ76] are composed only of looping, conditional, sequencing, and single-level loop exiting constructs.

[1] This is just one of several possible characterizations of reducible programs.

Control structure	CORTL equivalent

`if` E `then`
  S_1
`else`
  S_2
	`region R` $\{$
	  `if` E `then` $\{$
	    S_1`;`
	    `depart R;`
	  $\}$
	  S_2`;`
	$\}$

`repeat`
  S
`until` E
	`region R` $\{$
	  S`;`
	  `if` E `then`
	    `depart R;`
	  `repeat R;`
	$\}$

`while` E `do`
  S
	`region R`
	  `if` E `then` $\{$
	    S`;`
	    `repeat R;`
	  $\}$

`case` E `in`
  E_1`:` S_1
  $\ldots$
  E_n`:` S_n
  `default:` D
`end case`
	`region R` $\{$
	  `if` E `=` E_1 `then` $\{$
	    S_1`;`
	    `depart R;`
	  $\}$
	  $\ldots$
	  `if` E `=` E_n `then` $\{$
	    S_n`;`
	    `depart R;`
	  $\}$
	  D`;`
	$\}$

Figure 1.2: CORTL equivalents of standard control structures

- *Structured* programs are composed only of looping, conditional, and sequencing constructs.

As far as writing code optimization algorithms is concerned, one would like to consider only structured programs, since these have the simplest structure. However, this restriction is generally recognized as too severe for practical purposes. On the other hand, studies show that 95% of programs are reducible [MJ81, p. 22]. Thus, as a compromise between simplicity and generality, many optimization algorithms apply only to reducible programs.

As this dissertation will demonstrate, tree-structured programs are the happy medium between the generality of reducible programs and the simplicity of structured programs. The single-entry property of the control structures in tree-structured programs allows code optimization algorithms to be simpler than they are for reducible programs. It also makes an AST representation more appropriate, since control enters a sub-tree only at the root. But what percentage of programs are tree-structured? A study of 500 Fortran routines [KZ77] found that 82% were semi-structured, even though the routines were all written before structured programming became common practice; one would expect the percentage of tree-structured programs to be even larger. A study of over 119,000 C routines [BM] found only 2.9% contained any GOTOs at all, and that 68% of these were GOTOs were structured, meaning that over 99% of the routines were tree-structured. These studies indicate that the assumption of tree-structuredness is only slightly more restrictive than the common assumption of reducibility. Thus, to belabor an earlier metaphor, the GOTO tail is a very small part of the control structure dog, and we can treat it accordingly.

CORTL embodies ideas that arose during the structured programming debate [Knu74]. To summarize, during the late 1960s and early 1970s the programming community gradually realized that using control structures such as IF-THEN and WHILE-DO instead of the ubiquitous GOTO improved program clarity [Dij68]. At the same time, it became apparent that not all uses of GOTO were harmful. For example, using a GOTO to perform an exit from a WHILE-DO loop was often much clearer than the GOTO-less alternative: it was generally GOTOs *into* control structures, not *out* of them, that were harmful. Peterson *et al.* [PKT73] defined a *well-formed* program to be one in which all control structures have a single entry point; thus, being well-formed is equivalent to being tree-structured. They showed that such a program need only use IF-THEN, DO-FOREVER, and multi-level loop EXIT statements as control structures. CORTL uses what amounts to the same control structures, with the more instruction-like `repeat` statement taking the place of DO-FOREVER.

1.2 Related Work

Others have noted the merits of using ASTs or other tree-like representations for code optimization, but have only discussed their advantages with regard to traditional data flow analysis problems, such as computing live variables. Rosen [Ros77] proposes representing each statement in a program by a graph, and an entire program by an AST-like hierarchy of graphs. This makes data flow analysis efficient because the graphs for known control structures such as IF-THEN can be "analyzed" when a data flow analysis algorithm is written (at "define-time"); other kinds of graphs must undergo a dynamic analysis (at compile-time). Sharir [Sha80] proposes parsing a program's CFG into sub-graphs representing conventional control structures like IF-THEN-ELSE, and then applying techniques similar to Rosen's. Farrow *et al.* [FKZ76] propose an approach similar to Sharir's, although only for the CFGs of semi-structured programs. Because it uses the AST directly, the approach most similar in spirit to the one taken in this dissertation is Babich's "method of attributes" [BJ78], which associates data flow values ("attributes") with each node in an AST, and then iterates over the AST re-computing these values in a manner depending on the node type until they stabilize. Except for that of Farrow *et al.*, these approaches all handle arbitrary GOTOs.

The approach we take here differs from those just described in several respects. First, rather than accommodating unstructured GOTOs in each and every algorithm, we will assume they have been eliminated "up front" as described later. This allows us to concentrate the intellectual overhead associated with such GOTOs in one place, rather than scattering it throughout every algorithm we develop. It also lays the foundation for our use of ASTs instead of directed graphs as the program representation, another difference from most of the above approaches. Finally, we will go beyond the traditional data flow analysis problems with which the works mentioned above concern themselves. In recent years, notions such as *static single assignment form* [CFR+91] and *control dependence* [FOW87] have been proposed and used in optimization algorithms. We will consider these in the context of ASTs, as well as the "optimization" of register allocation.

Despite the advantages of ASTs, optimizing compilers have rarely used them. Zelkowitz and Bail describe a compiler for a structured programming language that performs common subexpression elimination and code motion using an AST [ZB74]. The Bliss compiler uses an AST for a few optimizations and to locate loops, but otherwise uses a CFG [WJW+75]. Powell describes a Modula-2 compiler that performs common subexpression elimination and code motion using an AST [Pow84]. These compilers all use language-specific ASTs for optimization, whereas we propose using CORTL ASTs in order to achieve language independence. Furthermore, we propose using these ASTs for the entire optimization and code generation process, including register allocation.

1.3 Conventions

We now establish some terminology and conventions:

- We will refer to the AST for a CORTL program as a CAST to distinguish it from ASTs for other languages. More formally, a CAST $T = (\rho, N)$ is an abstract syntax tree for a program in the language of Figure 1.1, where ρ is the root of the tree and N is the (non-empty and finite) set of nodes it contains.

- We will often deliberately confuse CORTL statements with the CAST nodes that represent them, as well as a CORTL program and the CAST for that program.

- We will call a region that is the target of one or more `repeat` statements a *loop region* or *loop*. We will call a region that is not the target of any `repeat` statement a *non-loop region*.

- Without loss of generality, we will assume that region names are unique.

- In addition to the CAST node types already described, we will assume that there is also an *empty statement* node type for representing empty statements. Since all instances of such a node are essentially identical, we can (and usually will) just use a prototypical instance

CORTL statement	Corresponding node type	Node attributes
`region` I S	`region`	*child*: node corresponding to S
`depart` I	`depart`	*region*: node corresponding to `region` I
`repeat` I	`repeat`	*region*: node corresponding to `region` I
`if` E `then` S	`if-then`	*condition*: expression corresponding to E *child*: node corresponding to S
S_1 S_2	`list`	*first*: node corresponding to S_1 *rest*: node corresponding to S_2
I `:=` E	`assignment`	*destination*: identifier corresponding to I *source*: expression corresponding to E

Figure 1.3: Description of CAST nodes

called `Nothing`. However, in CAST algorithms we will generally ignore the existence of empty statements in order to avoid clutter.

- We will represent statement lists using Lisp-style pairs. The end-of-list marker is `Nothing`, just like `nil` is in Lisp-style lists. However, unlike many Lisp implementations, we will *not* consider an empty statement node to be a special kind of statement list node. (We raise this technicality because sometimes in our definitions we will want to exclude statement list nodes, but never empty statement nodes.)

- We will access the attributes of CAST nodes using ".", as in *X.child*. Figure 1.3 shows the items in the abstract syntax, the corresponding CAST node types, and their core attributes; other attributes will be introduced as necessary.

- We will use the expression `new XNode(c1, c2, ...)` to create a new CAST node of type `X` with core attribute values `c1`, `c2`, ... For example, `new ListNode(s, Nothing)` creates a new statement list containing only one statement, `s`, and is analogous to `(cons s nil)` in Lisp.

-

A straightforward CAST implementation will use records to represent nodes and their components to represent attributes. However, it can be advantageous to instead represent an attribute A using a table that maps an arbitrary CAST node X (or a unique label thereof) to $X.A$. One example of this is the *region* attribute: if represented as a record component, it would turn the tree into a graph, making it more difficult to manipulate. In practice, it would likely be represented by a record component giving only the name of the region, which could then be mapped to the corresponding CAST node using a table.

In algorithms, we will often use a `Table` data structure. A `Table` represents a mapping from indices to values: if `t` is a `Table`, then the statement `t[i]` $\leftarrow$ `x` for an index `i` causes subsequent references to `t[i]` to yield `x`. We create a table using the expression `new Table`. If we need a table that yields a default value α for undefined index values, we use the expression `new Table with` α.

1.4 Eliminating GOTOs

CFG-based code optimization algorithms usually apply to any reducible program. In contrast, CAST-based optimization algorithms obviously apply only to any tree-structured program. This slight limitation creates a potential objection to using the algorithms described in this dissertation. To overcome it as well as the problem of GOTOs in general, we present Algorithm **Structure** in Figure "To Appear", which translates a CORTL program with both structured and unstructured GOTOs into a CORTL program without them. We will assume this algorithm is applied "up front", if necessary, to ensure that all programs are tree-structured.

[*To appear, maybe*]

Chapter 2

Data Flow Analysis

To optimize a program, it is usually necessary to answer certain questions regarding the flow of data. Such *data flow problems* include the classical ones of reaching definitions, live variables, available expressions, and busy expressions [ASU86]. These problems are usually formulated in an algebraic setting called a *monotone data flow analysis framework* [MR90], and usually solved using either *iterative* [KU76] [HDT87] or *elimination* [RP86] algorithms. The main result in this chapter is a simple, efficient elimination algorithm for CORTL programs. We will deviate slightly from normal practice by framing the exposition in terms of *path expressions*, which are regular expressions representing sets of program paths. This will allow us to use Tarjan's elegant results relating path expressions to data flow problems [Tar81]. The resulting CAST algorithm is as fast as any CFG algorithm for the vast majority of programs. Furthermore, in keeping with the contention of this dissertation, it is simpler: simpler to understand, to reason about, and to implement.

Most of the background material in this chapter is taken from [Tar81], with occasional modifications because we are working with ASTs rather than CFGs. The presentation here is rather terse, since the concepts involved in data flow analysis will be familiar to many readers. A more verbose presentation appears in [Hec77], and [MR90] provides a good survey of the whole topic of data flow analysis.

2.1 Paths and Path Expressions

In order to apply the ideas in [Tar81], we need to define the notion of a *path* through a CAST. The next several definitions supply the concepts necessary to do this.

Definition 2.1 *Consider a CAST $T = (\rho, N)$. We define $\tilde{N}$ to be N minus any statement list nodes, and $\tilde{\rho}$ to be the first non-list node appearing in a pre-order traversal of T. Let $s \in \tilde{N}$. We define s_a to be the first node in $\tilde{N}$ appearing after s—or s.region if s is a* `depart` *statement—in a post-order traversal of T, or* `Nothing` *if there is no such node. We define s_u to be the first node in $\tilde{N}$ appearing after s—or s.region if s is a* `repeat` *statement—in a pre-order traversal of the tree rooted at s.*

Definition 2.2 *The* successors *of s, $succ(s)$, are defined as follows:*

$$succ(s) = \begin{cases} \{s_u\} & \textit{if } s \textit{ is a region node} \\ \{s_a\} & \textit{if } s \textit{ is a depart node} \\ \{s_u\} & \textit{if } s \textit{ is a repeat node} \\ \{s_a\} & \textit{if } s \textit{ is an assignment node} \\ \{s_u, s_a\} & \textit{if } s \textit{ is an if-then node} \end{cases}$$

The predecessors *of s, $pred(s)$, is the set of nodes having s as a successor.*

This definition essentially ignores statement list nodes. This is because they serve only to order statements, and have no computational significance themselves.

Definition 2.3 *Let $T = (\rho, N)$ be a CAST. An* edge *in T is a pair (s', s'') of nodes in $\tilde{N}$ such that $s'' \in succ(s')$. If $e = (s', s'')$ is an edge, then $h(e) = s'$ is the* head *of e, and $t(e) = s''$ is the* tail*; we regard e as leading from $h(e)$ to $t(e)$. We use $\vec{s_u}$ and $\vec{s_a}$ to respectively denote the edges (s, s_u) and (s, s_a). We denote the set of edges in T by T_E.*

(Later on, the edges between a statement and its successors will crop up frequently, hence the abbreviations.)

Definition 2.4 *A* path *in T is a sequence of edges $(e_1, e_2, \ldots, e_k)$ in T_E such that $t(e_i) = h(e_{i+1})$ for $1 \leq i \leq k-1$. The path is* from *$h(e_1)$ to $t(e_k)$. There is a path with no edges from any node to itself. A* cycle *is a non-empty path from a node to itself.*

We now review the concept of a *regular expression*, and relate it to that of a path.

Definition 2.5 *An* alphabet *is a finite set of symbols. A* string *over an alphabet Σ is a finite sequence of symbols from Σ. The* empty string, *ϵ, is the string with no symbols. If x and y are strings, then the* concatenation *of x and y is the string xy. If P and Q are sets of strings, then $PQ = \{xy \mid x \in P \text{ and } y \in Q\}$, and $P^* = \bigcup_{k=0}^{\infty} P^k$, where $P^0 = \{\epsilon\}$ and $P^i = PP^{i-1}$.*

Definition 2.6 *Let Σ be an alphabet. The* regular expressions over Σ *and the set of strings they denote are defined recursively as follows:*

1. *$\emptyset$ is a regular expression and denotes the empty set.*
2. *ϵ is a regular expression and denotes the set $\{\epsilon\}$.*
3. *For each A in Σ, A is a regular expression and denotes the set $\{A\}$.*
4. *If P and Q are regular expressions denoting the sets p and q, respectively, then $(P \mid Q)$, $(P \cdot Q)$, and (P^*) are regular expressions that denote the sets $p \cup q$, pq, and p^*, respectively.*

The set of strings denoted by a regular expression R is $\sigma(R)$.

If P and Q are regular expressions, then $P = Q$ if and only if $\sigma(P) = \sigma(Q)$. Also, the following rules hold for any regular expression R:

$$\begin{aligned} &R \mid \epsilon = \epsilon \mid R = R \\ &R \cdot \epsilon = \epsilon \cdot R = R \\ &(\epsilon)^* = \epsilon \\ &R \mid \emptyset = \emptyset \mid R = R \\ &R \cdot \emptyset = \emptyset \cdot R = \emptyset \\ &(\emptyset)^* = \epsilon \end{aligned}$$

Definition 2.7 *Let $T = (\rho, N)$ be a CAST, and let s' and s'' be nodes in $\tilde{N}$. A* path expression *representing the paths from s' to s'' is a regular expression P over T_E such that every string in $\sigma(P)$ is a path from s' to s''.*

One path expression worth mentioning is $\emptyset$, which represents the *absence* of a path; intutively speaking, it means "no path at all". Such a path expression is quite convenient, as the algorithms appearing later will show.

2.2 Data Flow Frameworks and Problems

Solving a data flow problem means determining what facts must be true at a given point in a program, regardless of the path taken to get there. These facts can then be used to perform code optimization. To represent the universe of data flow facts that could be true at any point in the program, we use an algebraic structure called a *semilattice* that has an operator $\wedge$ such that $x \wedge y$ represents the facts that must be true if the facts x and y represent are true.

Definition 2.8 *A* semilattice *is a set L with a commutative, associative, and idempotent binary* meet *operation $\wedge$. We say that $x \leq y$ if and only if $x \wedge y = x$ for $x, y \in L$. We assume L has a* bottom *element $\perp$ and a* top *element $\top$ such that $\perp \leq x$ and $x \leq \top$ for $x \in L$. We also assume that L is* closed*: for every subset $S \subseteq L$, $\bigwedge S \in L$, where $\bigwedge S = \top$ if S is empty, and $\bigwedge S = \bigwedge_{x \in S} x$ otherwise.*

Obviously, $\perp \wedge x = \perp$, $\top \wedge x = \top$, and $x \wedge y \leq y$ for $x, y \in L$. The semilattice element $\perp$ represents what it is safe to assume at the beginning of any program, and $\top$ represents what it is safe to assume at any point in any program. Our assumption that any semilattice has a top element is not strictly necessary, and in fact Tarjan does not use it in [Tar81]. However, it simplifies the exposition, and is fairly innocuous: most data flow problems (including the classical ones) have a natural top element, and an artificial one can frequently be defined for those that do not.

To represent the effect of program statements on the universe of facts, we will associate with each edge e a function f_e such that if fact x is true on entry to $h(e)$, then $f_e(x)$ is true on entry to $t(e)$. The following definition extends this idea to paths.

Definition 2.9 *Let $T = (\rho, N)$ be a CAST, and assume that for every edge $e \in T_E$ there is a corresponding function f_e. For a given path p, we define $f_p(x) = x$ if p is the empty path, and $f_p(x) = (f_{e_k} \circ f_{e_{k-1}} \circ \ldots \circ f_{e_1})(x)$ if $p = (e_1, e_2, \ldots, e_k)$.*

For a given program, what we would like to compute is $\wedge\{f_p(\perp) \mid p \in P(s)\}$ for each statement s, where $P(s)$ is the set of paths from the beginning of the program to s. However, as discussed below, this is not always possible: we must sometimes settle for an approximate solution. For example, it is not always possible to find an exact summary of the effect of traversing a cyclic path infinitely many times. We need to place restrictions on our data flow functions in order to guarantee at least an approximate solution. This motivates the following definitions.

Definition 2.10 *A* (data flow) framework *(L, F) consists of a semilattice L with meet operation $\wedge$ and a function space F of functions from L to L having the following properties for $f, g \in F$ and $x, y \in L$:*

1. There exists $\iota \in F$ such that $\iota(x) = x$

2. There exists $\tau \in F$ such that $\tau(x) = \top$

3. $f \circ g \in F$

4. $f \wedge g \in F$

where $(f \wedge g)(x) = f(x) \wedge g(x)$. A framework is monotone *if $x \leq y$ implies that $f(x) \leq f(y)$. A framework is* continuous *if $f(\bigwedge S) = \bigwedge\{f(z) \mid z \in S\}$ for any non-empty set $S \subseteq L$. A framework is* closed *if $f^* \in F$, where $f^*(x) = \bigwedge\{f^i(x) \mid i \geq 0\}$. A framework is* pseudo-closed *if there exists $f^@ \in F$ such that for $i \geq 0$*

1. $f^@ \leq f^i$

2. $f \geq \iota$ implies that $f^@ \geq \iota$

where $f \leq g$ if $f(x) \leq g(x)$. f^* is called the closure *of f, and $f^@$ the* pseudo-closure.

Like our earlier assumption that every semilattice has a top element, our assumption that every function space has an element τ is not strictly necessary, and is not used in [Tar81]. Notice that whether a framework is monotone or continuous is independent of whether it is closed or pseudo-closed. However, a continuous framework is also monotone, and a closed framework is also pseudo-closed, since one choice for $f^@$ is f^*.

Definition 2.11 *A* data flow problem *consists of a CAST $T = (\rho, N)$, a data flow framework (L, F), and a mapping M from T_E to F. When the problem in question is clear from the context, we write f_e for $M(e)$. A* solution *to the problem is a mapping from $\tilde{N}$ to L; in what follows, $s \in \tilde{N}$. A* fixed point *solution Z satisfies*

$$\begin{aligned} Z(\tilde{\rho}) &= \perp \\ f_e(Z(h(e))) &\geq Z(t(e)) \end{aligned}$$

for $e \in T_E$. The meet over all paths (MOP) *solution Z_{mop} satisfies $Z_{mop}(s) = \bigwedge\{f_p(\perp) \mid p \in P(s)\}$, where $P(s)$ is the set of paths from $\tilde{\rho}$ to s. A* safe *[GW76] solution Z_{safe} satisfies $Z_{safe}(s) \leq Z_{mop}(s)$. An* acceptable *[GW76] solution Z_{acc} satisfies $Z_{acc}(s) \geq Z(s)$ for all fixed point solutions Z.*

In most data flow problems, f_e depends only on $h(e)$, and so M is essentially a map from program statements to data flow functions. We let a data flow problem inherit the properties of its framework: for example, we say a problem is closed and monotone if its framework is.

The best solution to a given data flow problem is the MOP one. However, in general this is only computable in continuous problems [KU77]. Most problems satisfy the *descending chain condition*: every descending chain $x_1 > x_2 > \ldots$ in the semilattice has finite length. Such problems are continuous if they are *distributive*: $f(x \wedge y) = f(x) \wedge f(y)$ for all $f \in F$ and $x, y \in L$. The classical data flow problems are continuous, since they are distributive and satisfy the descending chain condition. Thus, an MOP solution can be obtained for them.

If the MOP solution is unobtainable, we must settle for an approximation to it. One such approximation is the maximum fixed point (MFP) solution. A solution is safe and acceptable precisely if it lies between the MFP and MOP solutions [MR90]. If they exist, the MFP solution is the smallest acceptable solution and the MOP solution is the largest safe solution. In general, the "higher" a safe and acceptable solution is in the semilattice, the more data flow information it captures, and hence the better it is.

We now consider the relationship between data flow problems and path expressions.

Definition 2.12 *For a given pseudo-closed monotone data flow problem with CAST $T = (\rho, N)$, we define the mapping f from path expression to data flow functions as follows:*

$$\begin{aligned} f(\epsilon) &= \iota \\ f(\emptyset) &= \tau \\ f(e) &= f_e \\ f(P_1 \cup P_2) &= f(P_1) \wedge f(P_2) \\ f(P_1 \cdot P_2) &= f(P_2) \circ f(P_1) \\ f(P_1^*) &= f(P_1)^{@} \end{aligned}$$

where P_1 and P_2 are path expressions, and $e \in T_E$.

Theorem 2.1 ([Tar81]) *Consider a pseudo-closed monotone data flow problem with CAST $T = (\rho, N)$. Let $P(s)$ be a path expression representing all paths from $\tilde{\rho}$ to s for $s \in \tilde{N}$. Then the mapping $Z(s) = f(P(s))(\perp)$ is a safe and acceptable solution to the data flow problem. Furthermore, if the problem is closed and continuous, and $f^{@}$ is taken to be f^*, then Z is the MOP solution.*

The previous theorem reduces the solution of data flow problems to the computation of path expressions and pseudo-closures. We will discuss the former shortly. As for the latter, we now consider a class of monotone frameworks for which pseudo-closures can easily be computed.

Definition 2.13 *A framework (L, F) is* k-bounded *if $f^k \geq \bigwedge\{f^i \mid 0 \leq i \leq k-1\}$ for all $f \in F$.*

Theorem 2.2 ([Tar81]) *A k-bounded framework (L, F) is closed, with $f^* = \bigwedge\{f^i \mid 0 \leq i \leq k-1\}$ for all $f \in F$.*

Theorem 2.3 ([Tar81]) *In a monotone k-bounded framework (L, F), we can define $f^{@}$ by $f^{@} = (\iota \wedge f)^{k-1}$ for $f \in F$.*

The two preceding theorems tell us that a monotone k-bounded framework (L, F) is closed. For such a framework, they also give us a means of computing f^* for $f \in F$ using $O(k)$ function meets and compositions, and $f^{@}$ using $O(\lg k)$ such operations (through recursive doubling). This allows us to trade accuracy for speed by computing pseudo-closures instead of closures. Furthermore, if the framework is continuous, it is easy to show that $f^* = f^{@}$, and so we can compute the MOP solution using only $O(\lg k)$ function operations per closure.

There are two distinctions among data flow problems that we have thus far ignored. First, we have only discussed *forward* data flow problems, in which the data flow information at a given statement depends on that at preceding statements. Just as important are *backward* data flow problems, in which the data flow information at a given statement depends on that at succeeding statements. Most declarations about forward problems have obvious analogs for backward problems. Thus, in this chapter we direct most of our remarks at forward problems. When we do discuss backward problems, we will use the following modification to the earlier, "forward" definitions.

Definition 2.14 (backward contexts) *In the context of backward problems, $\tilde{\rho}$ will denote the* last *non-list node appearing in a pre-order traversal of a program with root ρ. An edge will be a pair s', s'' of nodes such that $s'' \in pred(s')$. Likewise, $\vec{s_u}$ and $\vec{s_a}$ will denote the edges (s_u, s) and (s_a, s).*

Second, we have only discussed *meet* semilattices, which are the natural models for "for all" data flow problems, in which a fact is true at a given statement s only if it is true after *each* predecessor of s. However, *join* semilattices are the natural models for "there exists" data flow problems, in which a fact is true at a given statement s only if it is true after *any* predecessor of s. Rather than a meet operator $\wedge$, a join semilattice L has an join operator $\vee$ such that $x \leq y$ if and only if $x \vee y = y$ for $x, y \in L$, implying that $\bot \vee x = x$, $\top \vee x = \top$, and $x \vee y \geq y$ for $x, y \in L$. Join and meet semilattices are dual concepts: a true statement about meet semilattices can be turned into a true statement about join semilattices by respectively replacing references to "meet (over all paths)", "maximum (fixed point)", "greater (solution)", and the like with "join (over all paths)", "minimum (fixed point)", "lesser (solution)", and so on [MR90]. Thus, we generally confine our attention to meet semilattices in this chapter.

Once we have an algorithm for computing path expressions, we will be able to easily solve all monotone, k-bounded data flow problems, a category that includes all of the common ones. In particular, the classical data flow problems are all monotone, 2-bounded, and continuous, and so we will be able to arrive at an efficient algorithm for finding the MOP solution to them.

2.3 Computing Path Expressions

Like many problems in program analysis, computing path expressions would be simple if it were not for loops. Hence, in deriving a CAST algorithm for computing path expressions, we focus on how to handle regions. In what follows, $T = (\rho, N)$ will be the program under consideration, r is a region, and $s \in \tilde{N}$. Also, although we will generally phrase the discussion in terms of sets of paths, we will sometimes deliberately confuse these sets with the path expressions that represent them. We will use the following definitions:

- The expression *home*(s) denotes the region immediately surrounding s. To ensure that there always is such a region, we assume that every program is embedded in an imaginary `region` statement.
- $I(s)$ is the set of regions surrounding s, including s itself if s is a region.
- A statement s is *immediately inside* a region r if *home*$(s) = r$, and *inside* r if $r \in I(s)$.
- $P(s', s'')$ is the set of paths from s' to s'', and we let $P(s) = P(\tilde{\rho}, s)$.
- We let $P_r(s', s'') = \{(s', s_1), \ldots, (s_k, s'') \mid r \in I(s_i) \text{ and } s_i \neq r_u\}$ for $1 \leq i \leq k$. Thus, $P_r(s', s'')$ consists of the paths from s' to s'' that do not cycle around r, and whose nodes are inside r, with the possible exceptions of s' and s''.
- We let $P_r(s) = P_r(r_u, s)$, and $C_r = P_r(r_u)$.

The paths from the beginning of the program to a statement s immediately inside or after a region r must pass through r_u, so

$$P(s) = P(r_u) \cdot P_r(s)$$

where either $r = home(s)$ or $s = r_a$. If we let $s = r_u$, we obtain the recurrence relation

$$P(r_u) = P(r_u) \cdot C_r$$

since $P_r(r_u) = C_r$. Since $P(r_u)$ includes the path $P(r) \cdot \vec{r_u}$, the solution to this recurrence is obviously

$$P(r_u) = P(r) \cdot \vec{r_u} \cdot C_r^* \tag{2.1}$$

Substituting back into the equation for $P(s)$ yields

$$P(s) = P(r) \cdot \vec{r_u} \cdot C_r^* \cdot P_r(s) \tag{2.2}$$

Let us assume for the moment that we know C_r for every region r in the program, and $P_r(s)$ for every statement s, where $r = home(s)$. Then Equation 2.2 gives us a recursive prescription for computing $P(s)$ from $P(r)$. The base case occurs when r is the imaginary `region` statement containing the entire program: in this case, $P(r) = \emptyset$.

We now consider how to compute C_r and $P_r(s)$, where $r = home(s)$. If there is no `region` statement between r_u and s, then $P_r(s)$ is easy to compute. Hence, suppose r' is the last region statement before s. Since all paths to s must go through r'_u and then r'_a, we have

$$P_r(s) = P_r(r'_u) \cdot P_{r'}(r'_a) \cdot P_r(r'_a, s) \tag{2.3}$$

where

$$P_r(r'_u) = P_r(r') \cdot \vec{r'_u} \cdot C_{r'}^* \tag{2.4}$$

by a slight variation on Equation 2.1. Equation 2.3 and Equation 2.4 reduce the problem of computing $P_r(s)$ to that of computing $P_r(r_a)$ and C_r^* (which we already need for Equation 2.2) for each region r. Let $D(r, r') = P_r(r'_a)$ and $R(r, r') = P_r(r'_u)$ for $r' \in I(r)$. Also, let $Regions(r)$ be the set of `region` statements immediately inside r, and $Departs(r, r')$ and $Repeats(r, r')$ respectively be the set of `depart` and `repeat` statements immediately inside r that refer to r'. Then we have

$$\begin{aligned}
C_r &= R(r, r) && (2.5)\\
P_r(r_a) &= D(r, r) && (2.6)\\
R(r, r') &= \left(\bigcup\{P_r(x) \mid x \in Repeats(r, r')\}\right) \\
&\quad \cup \left(\bigcup\{P_r(x_u) \cdot R(x, r') \mid x \in Regions(r)\}\right) && (2.7)\\
D(r, r') &= \left(\bigcup\{P_r(x) \mid x \in Departs(r, r')\}\right) \\
&\quad \cup \left(\bigcup\{P_r(x_u) \cdot D(x, r') \mid x \in Regions(r)\}\right) && (2.8)
\end{aligned}$$

where $P_r(x_u)$ is given by Equation 2.4. Equation 2.5 and Equation 2.6 are true by definition. Equation 2.7 expresses the fact that the paths $R(r, r')$ consist of the paths $P_r(x)$ for every `repeat` statement x immediately inside r, or the paths $P_r(x_u)$ for every `region` statement x immediately inside r, followed by the paths $R(x, r')$. Similar reasoning holds for Equation 2.8. These equations provide a recursive prescription for computing the paths on the left-hand sides. The base case occurs when r contains no other regions, making it simple to compute $R(r, r')$, $D(r, r')$, and $P_r(s)$.

This reasoning leads to Algorithm **F-Path** for computing path expressions in Figure 2.1 and Figure 2.2. The input to the algorithm is a CAST $T = (\rho, N)$, and its output is a table *Path* such that *Path[s]* is a path expression representing $P(s)$ for $s \in \tilde{N}$. Algorithm **F-Path** consists of an *elimination* phase and a *propagation* phase. The bulk of the elimination phase is devoted to computing path expressions representing $R(r, r')$ and $D(r, r')$: the `region` clause of `Eliminate()` basically computes the second terms of Equation 2.7 and Equation 2.8, and the `depart` and `repeat` clauses compute the first terms. Using Equation 2.5, the elimination phase leaves a path expression representing C_r^* in *Closure[r]* for each region r; if r is not a loop, *Closure[r]* contains ϵ. Using Equation 2.3, it also leaves a path expression representing $P_r(s)$ in *Path[s]*. The propagation phase finishes the job of computing path expressions by using *Closure*, *Path*, and Equation 2.2 to update *Path* so that *Path[s]* contains a path expression representing $P(s)$.

We now prove that Algorithm **F-Path** is correct.

Lemma 2.1 *The elimination phase is correct.*

Proof [*To appear*]
□

Theorem 2.4 *The propagation phase is correct. Hence, Algorithm* **F-Path** *is correct.*

Proof [*To appear*]
□

We now analyze the time complexity of Algorithm **F-Path**.

Definition 2.15 *Consider a program $T = (\rho, N)$, and let X be the set of all* `depart` *and* `repeat` *statements in N. The* structural complexity *sc of T is given by $sc(T) = \sum_r out(r)$ for all* `region` *statements $r \in N$, where $out(r) = |\{x \mid r \in I(x) \text{ and } r \neq home(x)\}|$ for $x \in X$. Equivalently, $sc(T) = \sum_x d(x.region) - d(home(x))$ for $x \in X$, where the nesting depth $d(s) = |I(s)|$.*

The higher the structural complexity of a program, the less semi-structured it is. In particular, a program's structural complexity is zero if and only if the program is semi-structured. Hence, most programs will have a structural complexity of zero. Furthermore, although one can construct a program for which the structural complexity is $O(|N|^2)$, it seems unlikely that such programs would occur naturally. In practice, the structural complexity should never exceed $|N|$.

Theorem 2.5 *For a program $T = (\rho, N)$, the elimination phase of Algorithm* **F-Path** *performs $O(|N| + sc(T))$ "$|$" and "$\cdot$" operations, and $O(|N|)$ "$*$" operations. The propagation phase performs $O(|N|)$ "$\cdot$" operations, and no "$|$" or "$*$" operations. Thus, Algorithm* **F-Path** *performs a total of $O(|N| + sc(T))$ operations.*

```
  Path ← new Table;
  Closure ← new Table;
  Eliminate(ρ, ε, new Table, new Table);

Eliminate(s, fromRegion, departs, repeats)
  if s is not a statement list node then
    Path[s] ← fromRegion;
  case s is a
    region node:
      localDeparts ← new Table with ∅;
      localRepeats ← new Table with ∅;
      defaultDepart ← Eliminate(s.child, ε, localDeparts, localRepeats);
      localDeparts[s] ← localDeparts[s] | defaultDepart;
      Closure[s] ← (localRepeats[s])*;
      toChild ← fromRegion · s⃗_u · Closure[s];
      for each index r of localDeparts such that r ≠ s do
        departs[r] ← departs[r] | (toChild · localDeparts[r]);
      for each index r of localRepeats such that r ≠ s do
        repeats[r] ← repeats[r] | (toChild · localRepeats[r]);
      return toChild · localDeparts[s];

    depart node:
      departs[s.region] ← departs[s.region] | (fromRegion · s⃗_a);
      return ∅;

    repeat node:
      repeats[s.region] ← repeats[s.region] | (fromRegion · s⃗_u);
      return ∅;

    if-then node:
      return (fromRegion · s⃗_a) | Eliminate(s.child, fromRegion · s⃗_u, departs, repeats);

    assignment node:
      return fromRegion · s⃗_a;

    statement list node:
      return Eliminate(s.rest, Eliminate(s.first, fromRegion, departs, repeats), departs, repeats);
  end case
end procedure
```

Figure 2.1: Algorithm **F-Path**, elimination phase

```
  Propagate(ρ, ε);

Propagate(s, toRegion)
  if s is not a statement list node then
    Path[s] ← toRegion · Path[s];
  case s is a
    region node:
      Propagate(s.child, Path[s] · s⃗_u · Closure[s]);

    if-then node:
      Propagate(s.child, toRegion);

    statement list node:
      Propagate(s.rest, Propagate(s.first, toRegion), toRegion);
  end case
end procedure
```

Figure 2.2: Algorithm **F-Path**, propagation phase

Proof The elimination phase performs $O(out(r))$ "$|$" and "$\cdot$" operations for every region node r, and a constant number of operations for every other kind of node. The propagation phase performs a constant number of operations for each node, and in particular zero "$|$" and "$*$" operations. The theorem follows from the fact that both phases consider each node just once. □

Since most programs are semi-structured, the last result means that the time complexity of Algorithm **F-Path** is $O(|N|)$ for typical programs, and "almost" $O(|N|)$ even for atypical ones.

To solve backward data flow problems, we need to come up with an algorithm analogous to Algorithm **F-Path** that computes path expressions representing the paths that go from the *end* of a program *backwards* to a given statement. There is one significant difference between computing forward and backward path expressions: from the forward perspective, regions have a *single* entrance and *multiple* exits, while from the backward perspective, they have *multiple* entrances and a *single* exit. In spite of this difference, the backward algorithm will turn out to be quite similar to Algorithm **F-Path**. We will use the same definitions that we used to discuss the forward algorithm, but in the "backward" sense: $\tilde{\rho}$ is the *last* non-list statement in the program, and all paths are *backward* (in other words, contrary to the flow of control). In particular, notice that $P_r(s)$ now denotes the paths from r_u to a `repeat` statement referring to r and then *backwards* to s. We will let $\bar{P}_r(s) = P_r(s, r_u)$ denote the complementary set of paths from s backwards to r_u.

The paths from the end of the program to a statement s in a region $r = home(s)$ fall into three categories:

1. The paths $A(s)$ that pass through r'_a for $r' \in I(s)$:

$$A(s) = \bigcup \{P(r'_a) \cdot P_r(r'_a, s) \mid r' \in I(s)\}$$

2. The paths $B(s)$ that pass through r'_u for $r' \in I(s) - r$:

$$B(s) = \bigcup \{P(r'_u) \cdot P_r(r'_u, s) \mid r' \in I(s) - r\}$$

3. The paths that pass through r_u:

$$P(r_u) \cdot P_r(r_u, s) = P(r_u) \cdot P_r(s)$$

Thus,

$$P(s) = A(s) \cup B(s) \cup (P(r_u) \cdot P_r(s)) \tag{2.9}$$

If we let $s = r_u$, we obtain the recurrence relation

$$P(r_u) = A(r_u) \cup B(r_u) \cup (P(r_u) \cdot C_r)$$

since $P_r(r_u) = C_r$. As is easily checked, the solution to this recurrence is

$$P(r_u) = (A(r_u) \cup B(r_u)) \cdot C_r^* \tag{2.10}$$

Let us assume for the moment that we know C_r, $\bar{P}_r(r'_a)$ (for computing $A(r_u)$), and $\bar{P}_r(r'_u)$ (for computing $B(r_u)$) for every region r in the program. Then Equation 2.9 together with Equation 2.10 gives us a recursive prescription for computing $P(s)$ from $P(r'_a)$ for $r' \in I(s)$ and $P(r'_u)$ for $r' \in I(s) - r$, where $r = \mathit{home}(s)$. The base case occurs when r is the imaginary `region` statement containing the entire program: in this case, $P(r_a) = \epsilon$, so $A(s) = P_r(r'_a, s)$; and $I(s) - r = \emptyset$, so $B(s) = \emptyset$.

Using reasoning (and notation) similar to that used in the forward case, we now consider how to compute C_r, $D(r, r') = \bar{P}_r(r'_a)$, and $R(r, r') = \bar{P}_r(r'_u)$ for $r' \in I(r)$. In fact, the necessary equations are simply the backward versions of those for the forward case:

$$\begin{aligned}
C_r &= R(r, r) && (2.11)\\
R(r', r) &= \left(\bigcup\{\bar{P}_r(x) \mid x \in \mathit{Repeats}(r, r')\}\right) &&\\
&\quad \cup \left(\bigcup\{R(r', x) \cdot \bar{P}_r(x_u) \mid x \in \mathit{Regions}(r)\}\right) && (2.12)\\
D(r', r) &= \left(\bigcup\{\bar{P}_r(x) \mid x \in \mathit{Departs}(r, r')\}\right) &&\\
&\quad \cup \left(\bigcup\{D(r', x) \cdot \bar{P}_r(x_u) \mid x \in \mathit{Regions}(r)\}\right) && (2.13)\\
& && (2.14)
\end{aligned}$$

where

$$\bar{P}_r(x_u) = C_x^* \cdot \vec{x_u} \cdot P_r(x)$$

which we obtain merely by reversing Equation 2.4. Equation 2.11 is true by definition. Equation 2.12 expresses the fact that the paths $R(r', r)$ consist of the paths $\bar{P}_r(x)$ for every `repeat` statement x immediately inside r, or the paths $R(r', x)$ followed by the paths $\bar{P}_r(x_u)$ for every `region` statement x immediately inside r. Similar reasoning holds for Equation 2.13. These equations provide a recursive prescription for computing the paths on the left-hand sides. The base case occurs when r contains no other regions, making it simple to compute $R(r', r)$ and $D(r', r)$, and $\bar{P}_r(x_u)$.

This reasoning leads to Algorithm **B-Path** for computing path expressions for backward paths in Figure 2.3 and Figure 2.4. The input to the algorithm is a CAST $T = (\rho, N)$, and its output is a table *Path* such that *Path[s]* is a path expression representing $P(s)$ for $s \in \mathit{tildeN}$. Like Algorithm **F-Path**, Algorithm **B-Path** consists of an elimination phase and a propagation phase, both of which bear a strong resemblance to their forward counterparts. In fact, the elimination phase of Algorithm **B-Path** is identical to that of Algorithm **F-Path** except that `Eliminate()`

- concatenates paths in the reverse order;
- retains *localDeparts* and *localRepeats* in *Departure[r]* and *Repeature[r]* after it processes a region r for later use in the propagation phase; and
- uses *Repeature[r][r]* rather than a separate *Closure* table to hold the closure of r.

Contrary to what one might expect, `Eliminate()` traverses the program forwards, not backwards: concatenating paths in reverse suffices to build the necessary backward path expressions.

The propagation phase the job of computing path expressions by using Equation 2.9 and the information in *Departure* and *Repeature* to initialize the table *Path* so that *Path[s]* contains a path expression representing $P(s)$. It is slightly more complicated than its counterpart in Algorithm **F-Path**, since it must deal with the fact that from a backward perspective, regions have multiple entrances. Upon entering a region r, `Propagate()` stores $P(r_a)$ in *departs[r]* and $P(r_u)$—computed using Equation 2.10—in *repeats[r]*. Thus, when it comes to a `depart` or `repeat` statement referring to r as it sweeps backward through the program, *departs[r]* and *repeats[r]* contain the relevant path expressions.

We now prove that Algorithm **B-Path** is correct.

Lemma 2.2 *The elimination phase of Algorithm* **B-Path** *is correct.*

Proof [*To appear*]
□

Theorem 2.6 *The propagation phase of Algorithm* **B-Path** *is correct. Hence, Algorithm* **B-Path** *is correct.*

Proof [*To appear*]
□

The following theorem shows that time complexity of Algorithm **B-Path** is the same as that of Algorithm **F-Path**, although in practice its running time would be roughly twice as great.

Theorem 2.7 *For a program $T = (\rho, N)$, the elimination phase of Algorithm* **B-Path** *performs $O(|N| + sc(T))$ "$|$" and "$\cdot$" operations, and $O(|N|)$ "$*$" operations. The propagation phase performs $O(|N| + sc(T))$ "$|$" and "$\cdot$" operations, and no "$*$" operations. Thus, Algorithm* **B-Path** *performs a total of $O(|N| + sc(T))$ operations.*

Proof The for-do loops in the `region` clauses of `Eliminate` and `Propagate` obviously execute approximately the same number of times. Hence, the theorem follows from the same reasoning used in Theorem 2.5 to analyze the time complexity of Algorithm **F-Path**. □

```
  Departure ← new Table;
  Repeature ← new Table;
  Eliminate(ρ, ε, new Table, new Table);

Eliminate(s, fromRegion, departs, repeats)
  case s is a
    region node:
      localDeparts ← new Table with ∅;
      localRepeats ← new Table with ∅;
      defaultDepart ← Eliminate(s.child, ε, localDeparts, localRepeats);
      localDeparts[s] ← localDeparts[s] | defaultDepart;
      localRepeats[s] ← (localRepeats[s])*;
      Departure[s] ← localDeparts;
      Repeature[s] ← localRepeats;
      toChild ← localRepeats[s] · s⃗_u · fromRegion;
      for each index r of localDeparts such that r ≠ s do
        departs[r] ← departs[r] | (localDeparts[r] · toChild);
      for each index r of localRepeats such that r ≠ s do
        repeats[r] ← repeats[r] | (localRepeats[r] · toChild);
      return localDeparts[s] · toChild;

    depart node:
      departs[s.region] ← departs[s.region] | (s⃗_a · fromRegion);
      return ∅;

    repeat node:
      repeats[s.region] ← repeats[s.region] | (s⃗_u · fromRegion);
      return ∅;

    if-then node:
      return (s⃗_a · fromRegion) | Eliminate(s.child, s⃗_u · fromRegion, departs, repeats);

    assignment node:
      return s⃗_a · fromRegion;

    statement list node:
      return Eliminate(s.rest, Eliminate(s.first, fromRegion, departs, repeats), departs, repeats);
  end case
end procedure
```

Figure 2.3: Algorithm **B-Path**, elimination phase

```
  Path ← new Table;
  Propagate(ρ, ε, new Table);

Propagate(s, toHere, departs, repeats)
  if s is not a statement list node then
    Path[s] ← toHere;
  case s is a
    region node:
      departs[s] ← toHere;
      tmp ← ∅;
      for each index r of Departure[s] do
        tmp ← tmp | (departs[r] · Departure[r]);
      for each index r ≠ s of Repeature[s] do
        tmp ← tmp | (repeats[r] · Repeature[r]);
      repeats[s] ← tmp · Repeature[s][s];
      return Propagate(s.child, toHere, departs, repeats);

    depart node:
      return departs[s.region] · s⃗_a;

    repeat node:
      return repeats[s.region] · s⃗_u;

    if-then node:
      return (s⃗_a · toHere) | Propagate(s.child, s⃗_u · toHere, departs, repeats);

    assignment node:
      return s⃗_a · toHere;

    statement list node:
      return Propagate(s.first, Propagate(s.rest, toHere, departs, repeats), departs, repeats);
  end case
end procedure
```

Figure 2.4: Algorithm **B-Path**, propagation phase

2.4 Solving Data Flow Problems

As already mentioned, Algorithm **F-Path** together with Theorem 2.1 gives us a technique for solving data flow problems involving CASTs by first computing $P(s)$ for each statement s and then computing $f(P(s))(\bot)$. In practice, however, one would rarely compute $P(s)$. Instead, one would compute $f(P(s))(\bot)$ directly using Algorithm **F-Flow** in Figure 2.5 and Figure 2.6. Algorithm **F-Flow** is simply Algorithm **F-Path** with the path expression symbols "$\cdot$", "$|$", "$*$", and "ϵ" respectively replaced by the corresponding function operations "$\circ$", "$\wedge$", "@", and "ι" or "$\bot$"; with each edge replaced by the corresponding function; and with function compositions reversed, as Definition 2.12 requires. The input to Algorithm **F-Flow** is a data flow problem with CAST $T = (\rho, N)$, framework (L, F), and mapping M, and the output is a table *Before* such that for $s \in \tilde{N}$, *Before[s]* $= f(P(s))(\bot)$.

The time complexity of Algorithm **F-Flow** is essentially proportional to the boundedness of the data flow problem, as the following theorem shows.

Theorem 2.8 *Consider a monotone k-bounded data flow problem with CAST $T = (\rho, N)$ and framework (L, F). Algorithm* **F-Flow** *performs $O(k|N| + sc(T))$ function operations (meets, compositions, and applications) to find a safe and acceptable solution to the problem if we let $f^{@} = f^{*} = \bigwedge\{f^{i} \mid 0 \leq i \leq k-1\}$ for $f \in F$, and $O(|N| \lg k + sc(T))$ function operations if we let $f^{@} = (\iota \wedge f)^{k-1}$. Furthermore, the solution Algorithm* **F-Flow** *finds in the former case may be more accurate than that found in the latter, unless the problem is continuous, in which case Algorithm* **F-Flow** *finds the MOP solution using either definition of $f^{@}$.*

Proof By Theorem 2.5, Algorithm **F-Flow** performs $O(|N| + sc(T))$ function compositions, $O(|N|)$ function applications, and $O(c|N| + sc(T))$ function meets, where $c = k$ or $c = \lg k$, depending on whether f^{*} is approximated according to Theorem 2.2 or Theorem 2.3, respectively. The theorem follows directly from this and Theorem 2.1. $\Box$

In keeping with standard practice in the literature on function-based data flow algorithms, we expressed Theorem 2.8 in terms of numbers of function operations. However, as pointed out in [HDT87], such results can be misleading, since they may not accurately reflect the actual running time of the algorithm: even if it is possible to compute the functions $f \circ g$ and $f \wedge g$ in constant time for $f, g \in F$, it may not be possible to evaluate their *applications* $(f \circ g)(x)$ and $(f \wedge g)(x)$ in constant time for $x \in L$. To see why, consider how function composition (for example) would be represented in a language having first-class functions, say Scheme. In the absence of special knowledge about the function space F, the function $f \circ g$ would have to be represented by the lambda function `(lambda (x) (f (g x)))`. Composing functions in this manner could conceivably result in lambda functions whose evaluation would entail the evaluation of f_e for every edge $e \in T_E$. This would result in $O(|N|^2)$ applications of "primitive" functions (ones not built up from the functions f_e). Thus, counts of function operations—the typical figure of merit for elimination algorithms—should be taken with a grain of salt.

Algorithm **B-Flow** in Figure 2.7 and Figure 2.8 solves backward data flow problems. As with Algorithm **F-Flow**, the input to Algorithm **B-Flow** is a data flow problem with CAST $T = (\rho, N)$, framework (L, F), and mapping M, and the output is a table *After* such that for $s \in \tilde{N}$, *After[s]* $= f(P(s))(\bot)$. Most of the observations made about Algorithm **F-Flow** also apply to Algorithm **B-Flow**. However, the time complexity of Algorithm **B-Flow** is slightly different, as the following theorem shows.

```
  Path ← new Table;
  Closure ← new Table;
  Eliminate(ρ, ι, new Table, new Table);

Eliminate(s, fromRegion, departs, repeats)
  if s is not a statement list node then
    Path[s] ← fromRegion;
  case s is a
    region node:
      localDeparts ← new Table with τ;
      localRepeats ← new Table with τ;
      defaultDepart ← Eliminate(s.child, ι, localDeparts, localRepeats);
      localDeparts[s] ← localDeparts[s] ∧ defaultDepart;
      Closure[s] ← (localRepeats[s])^@;
      toChild ← Closure[s] ∘ f(s⃗_u) ∘ fromRegion;
      for each index r of localDeparts such that r ≠ s do
        departs[r] ← departs[r] ∧ (localDeparts[r] ∘ toChild);
      for each index r of localRepeats such that r ≠ s do
        repeats[r] ← repeats[r] ∧ (localRepeats[r] ∘ toChild);
      return localDeparts[s] ∘ toChild;

    depart node:
      departs[s.region] ← departs[s.region] ∧ (f(s⃗_a) ∘ fromRegion);
      return τ;

    repeat node:
      repeats[s.region] ← repeats[s.region] ∧ (f(s⃗_u) ∘ fromRegion);
      return τ;

    if-then node:
      return (f(s⃗_a) ∘ fromRegion) ∧ Eliminate(s.child, f(s⃗_u) ∘ fromRegion, departs, repeats);

    assignment node:
      return f(s⃗_a) ∘ fromRegion;

    statement list node:
      return Eliminate(s.rest, Eliminate(s.first, fromRegion, departs, repeats), departs, repeats);
  end case
end procedure
```

Figure 2.5: Algorithm **F-Flow**, elimination phase

```
   Before ← new Table;
   Propagate(ρ, ⊥);

Propagate(s, toRegion)
   if s is not a statement list node then
      Before[s] ← Path[s](toRegion);
   case s is a
      region node:
         Propagate(s.child, Closure[s](f(s⃗_u)(Path[s])));

      if-then node:
         Propagate(s.child, toRegion);

      statement list node:
         Propagate(s.rest, Propagate(s.first, toRegion), toRegion);
   end case
end procedure
```

Figure 2.6: Algorithm **F-Flow**, propagation phase

Theorem 2.9 *Consider a monotone k-bounded data flow problem with CAST $T = (\rho, N)$ and framework (L, F). Algorithm* **B-Flow** *performs $O(k|N| + sc(T))$ function operations (meets, compositions, and applications) to find a safe and acceptable solution to the problem if we let $f^{@} = f^* = \bigwedge\{f^i \mid 0 \le i \le k-1\}$ for $f \in F$, and $O(|N| \lg k + sc(T))$ function operations if we let $f^{@} = (\iota \wedge f)^{k-1}$; in either case, Algorithm* **B-Flow** *also performs $O(|N| + sc(T))$ semilattice meets. Furthermore, the solution Algorithm* **B-Flow** *finds in the former case may be more accurate than that found in the latter, unless the problem is continuous, in which case Algorithm* **B-Flow** *finds the MOP solution using either definition of $f^{@}$.*

Proof By Theorem 2.7, Algorithm **B-Flow** performs $O(|N|+sc(T))$ semilattice meets, $O(|N|+sc(T))$ function compositions and applications, and $O(c|N|+sc(T))$ function meets, where $c = k$ or $c = \lg k$, depending on whether f^* is approximated according to Theorem 2.2 or Theorem 2.3, respectively. The theorem follows directly from this and Theorem 2.1. $\square$

2.5 Examples: Reaching Definitions and Live Variables

By way of example, as well as for future reference, we now respectively apply Algorithm **F-Flow** and Algorithm **B-Flow** to the two most common data flow problems: reaching definitions and live variables. (The resulting algorithms can easily be modified to solve the other two classical data flow problems: available expressions and busy expressions.) The reaching definitions and live variables problems have the following properties in common:

- They are "there exists" problems, and so a join semilattice is the most natural model for them. The elements of this semilattice are sets of appropriate program components, and the semilattice itself is the power set of the set of all such program components. These components are assignments in the case of reaching definitions, and variables in the case of live variables. In both cases, the join operator $\vee$ is set union, the bottom element $\perp$ is

```
   Departure ← new Table;
   Repeature ← new Table;
   Eliminate(ρ, ι, new Table, new Table);

Eliminate(s, fromRegion, departs, repeats)
   case s is a
      region node:
         localDeparts ← new Table with τ;
         localRepeats ← new Table with τ;
         defaultDepart ← Eliminate(s.child, ι, localDeparts, localRepeats);
         localDeparts[s] ← localDeparts[s] ∧ defaultDepart;
         localRepeats[s] ← (localRepeats[s])^@;
         Departure[s] ← localDeparts;
      ¿ Repeature[s] ← localRepeats;
         toChild ← fromRegion ∘ f(s⃗_u) ∘ localRepeats[s];
         for each index r of localDeparts such that r ≠ s do
            departs[r] ← departs[r] ∧ (toChild ∘ localDeparts[r]);
         for each index r of localRepeats such that r ≠ s do
            repeats[r] ← repeats[r] ∧ (toChild ∘ localRepeats[r]);
         return toChild ∘ localDeparts[s];

      depart node:
         departs[s.region] ← departs[s.region] ∧ (fromRegion ∘ f(s⃗_a));
         return τ;

      repeat node:
         repeats[s.region] ← repeats[s.region] ∧ (fromRegion ∘ f(s⃗_u));
         return τ;

      if-then node:
         return (fromRegion ∘ f(s⃗_a)) ∧ Eliminate(s.child, fromRegion ∘ f(s⃗_u), departs, repeats);

      assignment node:
         return fromRegion ∘ f(s⃗_a);

      statement list node:
         return Eliminate(s.rest, Eliminate(s.first, fromRegion, departs, repeats), departs, repeats);
   end case
end procedure
```

Figure 2.7: Algorithm **B-Flow**, elimination phase

```
  After ← new Table;
  Propagate(ρ, ⊥, new Table);

Propagate(s, toHere, departs, repeats)
  if s is a statement list node then
    After[s] ← toHere;
  case s is a
    region node:
      departs[s] ← toHere;
      tmp ← ⊤;
      for each index r of Departure[s] do
        tmp ← tmp ∧ Departure[r](departs[r]);
      for each index r ≠ s of Repeature[s] do
        tmp ← tmp ∧ Repeature[r](repeats[r]);
      repeats[s] ← Repeature[s][s](tmp);
      return Propagate(s.child, toHere, departs, repeats);

    depart node:
      return f(s⃗_a)(departs[s.region]);

    repeat node:
      return f(s⃗_u)(repeats[s.region]);

    if-then node:
      return f(s⃗_a)(toHere) ∧ Propagate(s.child, f(s⃗_u)(toHere), departs, repeats);

    assignment node:
      return f(s⃗_a)(toHere);

    statement list node:
      return Propagate(s.first, Propagate(s.rest, toHere, departs, repeats), departs, repeats);
  end case
end procedure
```

Figure 2.8: Algorithm **B-Flow**, propagation phase

the empty set, and the top element $\top$ is the "universal" set $\mathcal{U}$ of all the components in the program.

- The data flow function associated with an edge (s', s'') depends only on s', and so we let $f_{s'}$ denote this function. Furthermore, $f_s = \iota$ for any statement s that is not an assignment or `if-then` statement.

- We can express the data flow function f_s in the form $f_s(x) = (x - K) \cup G = \langle G, K\rangle(x)$, where G and K are sets such that $G \cap K = \emptyset$: the *Gen* set G is the semilattice element representing the facts s "generates" (makes true), and the *Kill* set K is the semilattice element representing the facts s "kills" (makes false). In this representation, $\iota = \langle\emptyset, \emptyset\rangle$, and $\tau = \langle\mathcal{U}, \mathcal{U}\rangle$. Let $f_1 = \langle G_1, K_1\rangle$ and $f_2 = \langle G_2, K_2\rangle$. Then

$$\begin{aligned} f_2 \circ f_1 &= \langle (G_1 - K_2) \cup G_2, (K_1 - G_2) \cup K_2\rangle \\ f_2 \vee f_1 &= \langle G_1 \cup G_2, K_1 \cap K_2\rangle \end{aligned}$$

  by a derivation in [GW76]. (In the algorithms to follow, we will leave function symbols "uninterpreted", but replace semilattice symbols with their set equivalents.)

Analogous remarks hold for the other two classical data flow problems, which are "for all" problems that use meet semilattices.

We say that an assignment a defining a variable v *reaches* a statement s if there is a path from a to s in which v is not redefined. The reaching definitions problem is to determine the set of assignments that reach each statement in a program. Algorithm **Reach** in Figure 2.9 and Figure 2.10 solves this problem. The input to the algorithm is a CAST $T = (\rho, N)$, and the output is a table *Before* such that *Before[s]* is the set of all the assignments that reach s for $s \in \tilde{N}$. In the algorithm, the expression *definitions*(v) yields the set of all assignments in N that define the variable v.

We say that a variable v used in statement s' is *live* after a statement s if there is a path from s to s' in which v is not redefined. The live variables problem is to determine the set of variables that are live after each statement in a program. Algorithm **Live** in Figure 2.11 and Figure 2.12 solves this problem. The input to the algorithm is a CAST $T = (\rho, N)$, and the output is a table *After* such that *After[s]* is the set of all the variables that are live after s for $s \in \tilde{N}$. In the algorithm, the expression *variables*(x) yields the set of all variables used in the expression x.

As is easily verified, the classical data flow problems are 2-bounded and continuous, so Algorithm **Reach** and Algorithm **Live** will compute MOP solutions using $O(|N| + sc(T))$ set operations by Theorem 2.8 and Theorem 2.9. Thus, these algorithms are essentially linear in the size of the program.

2.6 Discussion

As already mentioned, there are two kinds of CFG-based data flow algorithms in general use: iterative and elimination. An iterative algorithm initializes the data flow value associated with each flow node to $\perp$, then iterates over the flow nodes in some order and updates the data flow value at each node according to the values at the predecessors [Kil73] [KU77]. Different choices of traversal order yield several variations on this extremely simple algorithm [HDT87].

```
  Path ← new Table;
  Closure ← new Table;
  Eliminate(ρ, ι, new Table, new Table);

Eliminate(s, fromRegion, departs, repeats)
  if s is not a statement list node then
    Path[s] ← fromRegion;
  case s is a
    region node:
      localDeparts ← new Table with τ;
      localRepeats ← new Table with τ;
      defaultDepart ← Eliminate(s.child, ι, localDeparts, localRepeats);
      localDeparts[s] ← localDeparts[s] ∨ defaultDepart;
      Closure[s] ← localRepeats[s] ∨ ι;
      toChild ← Closure[s] ∘ fromRegion;
      for each index r of localDeparts such that r ≠ s do
        departs[r] ← departs[r] ∨ (localDeparts[r] ∘ toChild);
      for each index r of localRepeats such that r ≠ s do
        repeats[r] ← repeats[r] ∨ (localRepeats[r] ∘ toChild);
      return localDeparts[s] ∘ toChild;

    depart node:
      departs[s.region] ← departs[s.region] ∨ fromRegion;
      return τ;

    repeat node:
      repeats[s.region] ← repeats[s.region] ∨ fromRegion;
      return τ;

    if-then node:
      return fromRegion ∨ Eliminate(s.child, fromRegion, departs, repeats);

    assignment node:
      return ⟨{s}, definitions(s.destination)⟩ ∘ fromRegion;

    statement list node:
      return Eliminate(s.rest, Eliminate(s.first, fromRegion, departs, repeats), departs, repeats);
  end case
end procedure
```

Figure 2.9: Algorithm **Reach**, elimination phase

```
  Before ← new Table;
  Propagate(ρ, ∅);

Propagate(s, toRegion)
  if s is not a statement list node then
    Before[s] ← Path[s](toRegion);
  case s is a
    region node:
      Propagate(s.child, Closure[s](Path[s]));

    if-then node:
      Propagate(s.child, toRegion);

    statement list node:
      Propagate(s.rest, Propagate(s.first, toRegion), toRegion);
  end case
end procedure
```

Figure 2.10: Algorithm **Reach**, propagation phase

In particular, if a "depth-first" order is chosen, then the algorithm requires only $O(dN)$ primitive operations (primitive function applications and semilattice meets) to solve a classical data flow problem[1], where N is the size of the program and d is its maximum loop nesting depth [KU76]; thus, in the worst case it can require $O(N^2)$ primitive operations. Babich and Jazayeri considered the corresponding AST-based algorithm for a canonical programming language having only WHILE-DO, IF-THEN, and GOTO as control structures [BJ78], and showed that it required no more than two passes over programs containing no GOTOs (in other words, structured programs). They also showed that it required no more than $W + L + 1$ passes in general, where W is the number of WHILE-DO statements in the program, and L the number of labels that are the targets of backward GOTOs. That result is much coarser than Theorem 2.8, which (roughly speaking) relates how the program actually *uses* GOTOs to the time complexity of Algorithm **F-Flow**.[2]

Elimination algorithms decompose the flow graph into ever smaller structures that are easy to analyze, then merge the analyses of these structures to arrive at a solution to the data flow problem [RP86]. Algorithm **F-Flow** is basically the CAST version of the simplest elimination algorithm, Allen-Cocke interval analysis [AC77]; it is also very similar to the algorithm in [Bur90]. Although elimination algorithms are generally considered more efficient than iterative algorithms, they are also harder to understand and implement [BR90]. This is due to the analysis required to reveal the structure in the flow graph, as well as the bookkeeping required to manipulate the flow graph according to this structure. In contrast, Algorithm **F-Flow** suffers from none of these drawbacks: it does not need to compute intervals, since these essentially correspond to regions, and it does not need to perform the graph manipulations necessary to record what happens as intervals are eliminated. All this makes Algorithm **F-Flow** much simpler than any CFG-based elimination algorithm. Furthermore, its $O(N + structuralcomplexity)$

[1] To be precise, this result applies to all *1-semibounded* problems, which includes all 1-bounded problems, but not all 2-bounded problems.

[2] We will take Algorithm **F-Flow** as representative of the other algorithms in this chapter.

```
   Departure ← new Table;
   Repeature ← new Table;
Eliminate(ρ, ι, new Table, new Table);

Eliminate(s, fromRegion, departs, repeats)
   case s is a
      region node:
         localDeparts ← new Table with τ;
         localRepeats ← new Table with τ;
         defaultDepart ← Eliminate(s.child, ι, localDeparts, localRepeats);
         localDeparts[s] ← localDeparts[s] ∨ defaultDepart;
         localRepeats[s] ← localRepeats[s] ∨ ι;
         Departure[s] ← localDeparts[s];
         Repeature[s] ← localRepeats[s];
         toChild ← fromRegion ∘ localRepeats[s];
         for each index r of localDeparts such that r ≠ s do
            departs[r] ← departs[r] ∨ (toChild ∘ localDeparts[r]);
         for each index r of localRepeats such that r ≠ s do
            repeats[r] ← repeats[r] ∨ (toChild ∘ localRepeats[r]);
         return toChild ∘ localDeparts[s];

      depart node:
         departs[s.region] ← departs[s.region] ∨ fromRegion;
         return τ;

      repeat node:
         repeats[s.region] ← repeats[s.region] ∨ fromRegion;
         return τ;

      if-then node:
         return (fromRegion ∘ ⟨variables(s.condition), ∅⟩) ∨ Eliminate(s.child, fromRegion, departs, repeats);

      assignment node:
         return fromRegion ∘ ⟨variables(s.source), {s.destination}⟩;

      statement list node:
         return Eliminate(s.rest, Eliminate(s.first, fromRegion, departs, repeats), departs, repeats);
   end case
end procedure
```

Figure 2.11: Algorithm **Live**, elimination phase

```
  After ← new Table;
  Propagate(ρ, ∅, new Table);

Propagate(s, toHere, departs, repeats)
  if s is a statement list node then
    After[s] ← toHere;
  case s is a
    region node:
      departs[s] ← toHere;
      tmp ← ⊤;
      for each index r of Departure[s] do
        tmp ← tmp ∧ Departure[r](departs[r]);
      for each index r ≠ s of Repeature[s] do
        tmp ← tmp ∧ Repeature[r](repeats[r]);
      repeats[s] ← Repeature[s][s](tmp);
      return Propagate(s.child, toHere, departs, repeats);

    depart node:
      return departs[s.region];

    repeat node:
      return repeats[s.region];

    if-then node:
      return ⟨variables(s.condition), ∅⟩(toHere) ∪ Propagate(s.child, toHere, departs, repeats);

    assignment node:
      return ⟨variables(s.source), {s.destination}⟩(toHere);

    statement list node:
      return Propagate(s.first, Propagate(s.rest, toHere, departs, repeats), departs, repeats);
  end case
end procedure
```

Figure 2.12: Algorithm **Live**, propagation phase

time complexity is better than the best known time complexity $O(dN)$ of the roughly comparable Allen-Cocke algorithm. In fact, the time complexity of Algorithm **F-Flow** is the same as that of the much more complex elimination algorithm in [GW76]. However, the worst-case time complexity of that algorithm—like most other elimination algorithms—is $O(N \lg N)$, while that of Algorithm **F-Flow** is $O(N^2)$. But as mentioned earlier, the possibility of a "naturally occurring" program provoking this worst-case behavior in Algorithm **F-Flow** seems extremely remote.

To summarize,

1. Algorithm **F-Flow** is simpler than any CFG-based elimination algorithm.

2. For the vast majority of programs, Algorithm **F-Flow** is as fast as any other data flow analysis algorithm.

We conclude that CAST algorithms are superior to CFG algorithms, at least as far as data flow analysis is concerned.

BIBLIOGRAPHY

[AC77] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19:137–147, 1977.

[ASU86] Alfred C. Aho, Ravi Seth, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BJ78] W. A. Babich and M. Jazayeri. The method of attributes for data flow analysis, parts I and II. *Acta Informatica*, 10(3):245–272, 1978.

[BM] Robert A. Ballance and Arthur B. Maccabe. Program dependence graphs for the rest of us. Technical Report 92-10, Department of Computer Science, University of New Mexico.

[BR90] M. Burke and B. G. Ryder. A critical analysis of incremental iterative data-flow algorithms. *IEEE Transactions on Software Engineering*, 16(7), July 1990.

[Bur90] Michael Burke. An interval-based approach to interprocedural analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Symposium on Programming Language Design and Implementation, SIGPLAN Notices*, pages 192–203, 1991.

[Dij68] E. W. Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11(3):147–148,538,541, March 1968.

[FKZ76] R. Farrow, K. Kennedy, and L. Zucconi. Graph grammars and global program data flow analysis. In *Proceedings of the Seventeenth Annual IEEE Symposium on Foundations of Computer Science*, pages 42–56, 1976.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GW76] S. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of the ACM*, 23(1):172–202, 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] M. S. Hecht. *Flow Analysis of Computer Programs.* North-Holland, 1977.

[Kil73] G. A. Kildall. Global expression optimization during compilation. In *Conference Record of the 1st Annual ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[Knu73] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, 1973.

[Knu74] Donald Knuth. Structured programming with goto statements. *ACM Computing Surveys*, 6(4):261–301, December 1974.

[Kra88] David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Yale University, 1988.

[KU76] J. B. Kam and J. D Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[KU77] J. B. Kam and J. D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–318, 1977.

[KZ77] Ken Kennedy and Linda Zucconi. Applications of a graph grammar for program control flow analysis. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 72–85, 1977.

[MJ81] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

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

[PKT73] W. W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat, and exit statements. *Communications of the ACM*, 16(8):503–512, 1973.

[Pow84] Michael L. Powell. A portable optimizing compiler for Modula-2. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, pages 310–318, 1984.

[Ros77] Barry K. Rosen. High-level data flow analysis. *Communications of the ACM*, 20(10):712–724, October 1977.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–315, September 1986.

[Sha80] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.

[Ste78] Guy L. Steele. Rabbit: A compiler for scheme. Technical report, Massachusetts Institute of Technology, May 1978.

[Tar81] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.

[Weg73] Eberhard Wegner. Tree-structured programs. *Communications of the ACM*, 16(11):704–705, November 1973.

[WJW+75] W. A. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler*. American Elsevier, 1975.

[ZB74] M. V. Zelkowitz and W. G. Bail. Optimization of structured programs. *Software—Practice and Experience*, 4(1):51–57, January 1974.