Chapter 9

Data-Flow Analysis

CHAPTER OVERVIEW

Compilers analyze the IR form of the program being compiled to identify opportunities where the code can be improved and to prove the safety and profitability of transformations that might improve the code. Data-flow analysis is the classic technique for compile-time program analysis. It allows the compiler to reason about the runtime flow of values in the program.

This chapter explores iterative data-flow analysis, which uses a simple fixed-point algorithm. From the basics of data-flow analysis, it builds up the construction of static single-assignment (SSA) form, illustrates the use of SSA form, and introduces interprocedural analysis.

Keywords: Data-flow Analysis, SSA Form, Dominance, Constant Propagation

9.1 INTRODUCTION

As we saw in Chapter 8, optimization is the process of analyzing a program and transforming it in ways that improve its runtime behavior. Before the compiler can improve the code, it must locate points in the program where changing the code is likely to improve it, *and* the compiler must prove that changing the code at those points is safe. Both of these tasks require a deeper understanding of the code than the compiler's front end typically derives. To gather the information needed to locate opportunities for optimization and to justify those optimizations, compilers use some form of static analysis.

In general, static analysis involves compile-time reasoning about the runtime flow of values. This chapter explores techniques that compilers use to analyze programs in support of optimization. It presents data-flow analysis at a deeper level than provided in Chapter 8. Next, Section 9.3 presents

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00009-8

algorithms for the construction and destruction of static single-assignment form. Section 9.4 discusses issues in whole-program analysis. The advanced topics section presents further material on computing dominance and a discussion of graph reducibility.

Conceptual Roadmap

Compilers use static analysis to determine where optimizing transformations can be safely and profitably applied. In Chapter 8, we saw that optimizations operate on different scopes, from local to interprocedural. In general, a transformation needs analytical information that covers at least as large a scope as the transformation; that is, a local optimization needs at least local information, while a whole-procedure, or global, optimization needs global information.

Static analysis generally begins with control-flow analysis—analyzing the IR form of the code to understand the flow of control between operations. The result of control-flow analysis is a control-flow graph. Next, compilers analyze the details of how values flow through the code. They use the resulting information to find opportunities for improvement and to prove the safety of transformations. The optimization community developed global data-flow analysis to answer these questions.

Static single assignment form is an intermediate representation that unifies the results of control-flow analysis and data-flow analysis in a single sparse data structure. It has proven useful in both analysis and transformation and has become a standard representation used in both research and production compilers.

Overview

Chapter 8 introduced the subject of analysis and transformation of programs by examining local methods, regional methods, global methods, and interprocedural methods. Value numbering is algorithmically simple, even though it achieves complex effects; it finds redundant expressions, simplifies code based on algebraic identities and zero, and propagates known constant values. In contrast, finding an uninitialized variable is conceptually simple but requires that the compiler analyze the entire procedure to track definitions and uses.

Join point
In a CFG, a *join point* is a node that has multiple predecessors.

The difference between these two problems lies in the kinds of control flows that each method must understand. Local and superlocal value numbering only deal with subsets of the CFG that form trees. To identify an uninitialized variable, the compiler must reason about the entire CFG, including cycles and *join points*, both of which complicate the analysis. In general, methods that restrict themselves to control-flow graphs that can be expressed as trees are amenable to online solutions, while those that must deal with cycles in the CFG require offline solutions—the entire analysis must complete before rewriting can begin.

Static, or compile-time, analysis is a collection of techniques that compilers use to prove the safety and profitability of a potential transformation. Static analysis over single blocks or trees of blocks is typically straightforward. This chapter focuses on global analysis, where the CFG can contain both cycles and join points. It will mention several problems in interprocedural analysis; these problems operate over the program's call graph or some related graph. To perform interprocedural analysis, the compiler must have access to information about other procedures in the program.

In simple cases, static analysis can produce precise results—the compiler can know exactly what will happen when the code executes. If the compiler can derive precise information, it might replace the runtime evaluation of an expression or function with an immediate load of the result. On the other hand, if the code reads values from any external source, involves even modest amounts of control flow, or encounters any ambiguous memory references (from pointers, array references, or call-by-reference parameters), then static analysis becomes much harder and the results of the analysis are less precise.

This chapter begins by examining classic problems in data-flow analysis. We focus on an iterative algorithm for solving these problems because it is simple, robust, and easy to understand. Section 9.3 presents an algorithm for constructing a static single-assignment form for a procedure. The construction relies heavily on results from data-flow analysis. The "Advanced Topics" section explores the notion of flow-graph reducibility, presents a faster approach to calculating dominators, and provides an introduction to interprocedural data-flow analysis.

9.2 ITERATIVE DATA-FLOW ANALYSIS

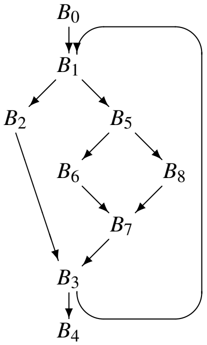
Compilers use data-flow analysis, a collection of techniques for compile-time reasoning about the runtime flow of values, to locate opportunities for optimization and to prove the safety of specific transformations. As we saw with live analysis in Section 8.6.1, problems in data-flow analysis take the form of a set of simultaneous equations defined over sets associated with the nodes and edges of a graph that represents the code being analyzed. Live analysis is formulated as a global data-flow problem that operates on the control-flow graph (CFG) of a procedure.

In this section, we will explore the properties of global data-flow problems and their solutions in more depth than was possible in Chapter 8. We will focus on one specific solution technique: an iterative fixed-point algorithm. It has the twin advantages of simplicity and robustness. As an initial example, we will examine the computation of dominance information. When we need a more complex example, we will return to consideration of LIVEOUT sets.

9.2.1 **Dominance**

Dominance
In a flow graph with entry node b_0, node b_i *dominates* node b_j, written $b_i \gg b_j$, if and only if b_i lies on every path from b_0 to b_j. By definition, $b_i \geqq b_i$.

Many optimization techniques must reason about the structural properties of the underlying code and its control-flow graph. A key tool that compilers use to reason about the shape and structure of the CFG is the notion of *dominators*. As we will see, dominators play a key role in the construction of static single-assignment form. While many algorithms have been proposed to compute dominance information, an extremely simple data-flow problem will suffice to annotate each node b_i in the CFG, which represents a basic block, with a set DOM(b_i) that contains the names of all nodes that dominate b_i.



To make this notion of dominance concrete, consider the node B_6 in the CFG shown in the margin. (Note that this CFG differs slightly from the example in Chapter 8.) Nodes B_0, B_1, B_5, and B_6 all lie on every path from B_0 to B_6, so DOM(B_6) is {B_0, B_1, B_5, B_6}. The full set of DOM sets for the CFG are as follows:

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DOM(*n*)	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}

To compute these sets, the compiler can solve the following data-flow problem:

$$\text{DOM}(n) = \{n\} \cup \left(\bigcap_{m \in preds(n)} \text{DOM}(m) \right)$$

with the initial conditions that DOM(n_0) = {n_0}, and $\forall n \neq n_0$, DOM(n) = N, where N is the set of all nodes in the CFG. These equations concisely capture the notion of dominance. Given an arbitrary flow graph—that is, a directed graph with a single entry and a single exit—the equations will correctly compute the DOM set for each node. Because they compute DOM(n) as a function of n's predecessors, denoted $preds(n_i)$, these equations form a forward data-flow problem.

```
n ← |N| - 1
Dom(0) ← {0}
for i ← 1 to n
    Dom(i) ← N

changed ← true
while (changed)
    changed ← false

    for i ← 1 to n
        temp ← {i} ∪ ( $\bigcap_{j \in preds(i)}$ Dom( j ) )

        if temp ≠ Dom(i) then
            Dom(i) ← temp
            changed ← true
```

■ **FIGURE 9.1** Iterative Solver for Dominance.

To use the equations, the compiler can use the same three-step procedure used for live analysis in Section 8.6.1. It must (1) build a CFG, (2) gather initial information for each block, and (3) solve the equations to produce the Dom sets for each block. For Dom, step 2 is trivial. Recall that the equations for LiveOut used two sets per block: UEVar(b) and VarKill(b). Since dominance deals only with the structure of the graph and not with the behavior of the code in each block, the only local information needed for a block b_i is its name, i.

Figure 9.1 shows a round-robin iterative solver for the dominance equations. It considers the nodes in order by their CFG name, B_0, B_1, B_2, and so on. It initializes the Dom set for each node, then repeatedly recomputes those Dom sets until they stop changing. It produces the following values in the Dom sets for our example:

	DOM(*n*)								
	B_0	**B_1**	**B_2**	**B_3**	**B_4**	**B_5**	**B_6**	**B_7**	**B_8**
—	{0}	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*N*
1	{0}	{0,1}	{0,1,2}	{0,1,2,3}	{0,1,2,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,6,7}	{0,1,5,8}
2	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
3	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}

The first column shows the iteration number; the row marked with a dash shows the initial values for the Dom sets. The first iteration computes correct Dom sets for any node with a single path from B_0, but computes overly large Dom sets for B_3, B_4, and B_7. In the second iteration, the smaller Dom set for B_7 corrects the set for B_3, which, in turn shrinks Dom(B_4). Similarly, the set for B_8 corrects the set for B_7. The third iteration is required to recognize that the algorithm has reached a fixed point. Note that the final Dom sets agree with our earlier table.

Three critical questions arise regarding this solution procedure. First, does the algorithm halt? It iterates until the Dom sets stop changing, so the argument for termination is not obvious. Second, does it produce correct Dom sets? The answer is critical if we are to use Dom sets in optimization. Finally, how fast is the solver? Compiler writers should avoid algorithms that are unnecessarily slow.

Termination

Iterative calculation of the Dom sets halts because the sets that approximate Dom shrink monotonically throughout the computation. The algorithm initializes the Dom set for n_0 to $\{0\}$, for the entry node n_0, and it initializes all the other Dom sets to N, the set of all nodes. A Dom set can be no smaller than one node name and can be no larger than $|N|$. Careful reasoning about the while loop shows that a Dom set, say Dom(n_i), cannot grow from iteration to iteration. Either it shrinks, as the Dom set of one of its predecessors shrinks, or it remains unchanged.

The while loop halts as soon as it makes a pass over the nodes in which no Dom set changes. Since the Dom sets can only change by shrinking and the Dom sets are bounded in size, the while loop must eventually halt. When it halts, it has found a fixed point for this particular instance of the Dom computation.

Correctness

Recall the definition of a dominator. Node n_i dominates n_j if every path from the entry node n_0 to n_j contains n_i. Dominance is a property of paths in the CFG.

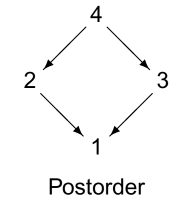
Dom(n_j) contains i if and only if $i \in$ Dom(n_k) for all $k \in preds(j)$, or if $i = j$. The algorithm computes Dom(n_j) as the intersection of the Dom sets of all n_j's predecessors, plus n_j itself. How does this local computation over individual edges relate to the dominance property defined over all paths through the CFG?

Meet operator
In the theory of data-flow analysis, the *meet operator* is used to combine facts at the confluence of two paths.

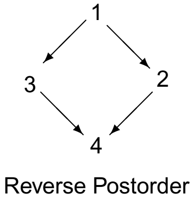
The Dom sets computed by the iterative algorithm form a fixed-point solution to the equations for dominance. The theory of iterative data-flow analysis, which is beyond the scope of this text, assures us that a fixed point exists for these particular equations and that the fixed point is unique [210]. The all-paths solution of the definition is also a fixed-point for the equations, called the *meet-over-all-paths* solution. The uniqueness of the fixed point guarantees that the solution found by the iterative algorithm is the meet-over-all-paths solution.

Efficiency

The uniqueness of the fixed-point solution to the DOM equations for a specific CFG ensures that the solution is independent of the order in which the solver computes those sets. Thus, the compiler writer is free to choose an order of evaluation that improves the analyzer's running time.



Postorder



Reverse Postorder

A *reverse postorder* (RPO) traversal of the graph is particularly effective for the iterative algorithm. A postorder traversal visits as many of a node's children as possible, in a consistent order, before visiting the node. (In a cyclic graph, a node's child may also be its ancestor.) An RPO traversal is the opposite—it visits as many of a node's predecessors as possible before visiting the node itself. A node's RPO number is simply $|N|+1$ minus its postorder number, where N is the set of nodes in the graph. Most interesting graphs will have multiple reverse postorder numberings; from the perspective of the iterative algorithm, they are equivalent.

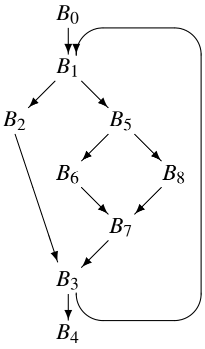
Postorder number
Label the nodes in the graph with their visitation order in a postorder traversal.

Reverse CFG
The CFG with its edges reversed; the compiler may need to add a unique exit node so that the reverse CFG has a unique entry node.

For a forward data-flow problem, such as DOM, the iterative algorithm should use an RPO computed on the CFG. For a backward data-flow problem, such as LIVEOUT, the algorithm should use an RPO computed on the *reverse* CFG.

To see the impact of ordering, consider the impact of an RPO traversal on our example DOM computation. One RPO numbering for the example CFG is:

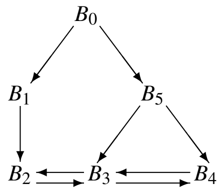
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
RPO(*n*)	0	1	6	7	8	2	4	5	3



Visiting the nodes in this order produces the following iterations and values:

	DOM(*n*)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
—	{0}	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*N*
1	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
2	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}

Working in RPO, the algorithm computes accurate DOM sets for this graph on the first iteration and halts after the second iteration. Using RPO, the algorithm halts in two passes over the graph rather than three. As we shall see, it does not compute accurate DOM sets in the first pass for all graphs.



As a second example, consider the CFG shown in the margin. Its structure is more complex than the earlier CFG. It has two loops, (B_2,B_3) and (B_3,B_4), with multiple entries. In particular, (B_2,B_3) has entries from both (B_0,B_1,B_2) and (B_0,B_5,B_3), while (B_3,B_4) has entries from (B_0,B_5,B_3) and (B_0,B_5,B_4). This property makes the graph more difficult to analyze (see Section 9.5.1).

To apply the iterative algorithm, we need a reverse postorder numbering. One RPO numbering for this CFG is:

	B_0	B_1	B_2	B_3	B_4	B_5
RPO(*n*)	0	2	3	4	5	1

With this RPO numbering, the algorithm executes the following iterations:

	DOM(*n*)					
	B_0	B_1	B_2	B_3	B_4	B_5
—	{0}	*N*	*N*	*N*	*N*	*N*
1	{0}	{0,1}	{0,1,2}	{0,3}	{0,4}	{0,5}
2	{0}	{0,1}	{0,2}	{0,3}	{0,4}	{0,5}
3	{0}	{0,1}	{0,2}	{0,3}	{0,4}	{0,5}

The algorithm requires two iterations to compute the correct DOM sets. The final iteration recognizes that the computation has reached a fixed point.

The dominance calculation relies only on the structure of the graph. It ignores the behavior of the code in any of the CFG's blocks. As such, it might be considered a form of control-flow analysis. Most data-flow problems involve reasoning about the behavior of the code and the flow of data between operations. As an example of this kind of calculation, we will revisit the analysis of live variables.

9.2.2 Live-Variable Analysis

In Section 8.6.1, we used the results of live analysis to identify uninitialized variables. Compilers use live information for many other purposes, such as register allocation and construction of some variants of SSA form. We formulated live analysis as a global data-flow problem with the equation:

$$\text{LIVEOUT}(n) = \bigcup_{m \in succ(n)} (\text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)}))$$

and the initial condition that $\text{LIVEOUT}(n) = \emptyset, \forall n$.

NAMING SETS IN DATA-FLOW EQUATIONS

In writing the data-flow equations for classic problems, we have renamed many of the sets that contain local information. The original papers used more intuitive set names. Unfortunately, those names clash with each other across problems. For example, available expressions, live variables, reaching definitions, and anticipable expressions all use some notion of a *kill* set. These four problems, however, are defined over three distinct domains: expressions (AVAILOUT and ANTOUT), definition points (REACHES), and variables (LIVEOUT). Thus, using a single set name, such as KILL or KILLED, leads to confusion across problems.

The names that we have adopted encode both the domain and a hint as to the set's meaning. Thus, VARKILL(*n*) contains the set of variables killed in block *n*, while EXPRKILL(*n*) contains the set of expressions killed in the same block. Similarly, UEVAR(*n*) contains the set of upward-exposed variables in block *n*, while UEEXPR(*n*) contains the set of upward-exposed expressions. While these names are somewhat awkward, they make explicit the distinction between the notion of kill used in available expressions (EXPRKILL) and the one used in reaching definitions (DEFKILL).

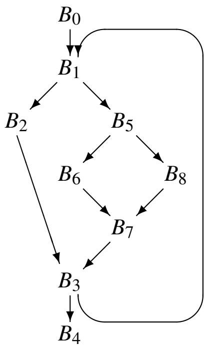
Comparing the equations for LIVEOUT and DOM reveals differences between the problems. LIVEOUT is a backward data-flow problem, in that LIVEOUT(*n*) is computed as a function of the information known on entry to each of *n*'s successors in the CFG. DOM is a forward data-flow problem, in that DOM(*n*) is computed as a function of the information known at the end of each of *n*'s predecessors in the CFG. LIVEOUT looks for a future use on *any path* in the CFG; thus, it joins information from multiple paths with the union operator. DOM looks for predecessors that lie on *all paths* from the entry node; thus it joins information from multiple paths with the intersection operator. Finally, LIVEOUT reasons about the effects of operations. For this reason, it uses the block-specific constant sets UEVAR and VARKILL that are derived from the code for each block. By contrast, DOM only deals with the CFG's structure. Accordingly, its block-specific constant set contains only the name of the block.

Despite these differences, the framework for solving an instance of LIVEOUT is the same as for an instance of DOM. The compiler must:

1. Perform control-flow analysis to build a CFG, as in Figure 5.6 on page 241.
2. Compute the values of the initial sets, as in Figure 8.14a on page 447.
3. Apply the iterative algorithm, as in Figure 8.14b on page 447.

```
B0:  i ← 1
     → B1
B1:  a ← ···
     c ← ···
     (a < c) → B2,B5
B2:  b ← ···
     c ← ···
     d ← ···
     → B3
B3:  y ← a + b
     z ← c + d
     i ← i + 1
     (i ≤ 100) → B1,B4

B4:  return
B5:  a ← ···
     d ← ···
     (a ≤ d) → B6,B8
B6:  d ← ···
     → B7
B7:  b ← ···
     → B3
B8:  c ← ···
     → B7
```

(a) Code for the Basic Blocks



(b) Control-Flow Graph

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
UEVAR	∅	∅	∅	{a,b,c,d,i}	∅	∅	∅	∅	∅
VARKILL	{i}	{a,c}	{b,c,d}	{y,z,i}	∅	{a,d}	{d}	{b}	{c}

(c) Initial Information

■ **FIGURE 9.2** Live Analysis Example.

To see the issues that arise in solving instances of LIVEOUT, consider the example in Figure 9.2. It fleshes out the example CFG that we have used throughout this chapter. Figure 9.2a shows code for each basic block. Figure 9.2b shows the CFG and Figure 9.2c shows the UEVAR and VARKILL sets for each block.

Figure 9.3 shows the progress of the iterative solver on the example from Figure 9.2, using the same RPO that we used in the DOM computation, namely B_0, B_1, B_5, B_8, B_6, B_7, B_2, B_3, B_4. Although the equations for LIVEOUT are more complex than those for DOM, the arguments for termination, correctness, and efficiency are similar to those for the dominance equations.

Termination

Recall that in DOM the sets shrink monotonically.

Iterative live-variable analysis halts because the sets grow monotonically. Each time that the algorithm evaluates the LIVEOUT equation at a node in the CFG, that LIVEOUT set either grows or it remains the same. The equation cannot shrink the LIVEOUT set. On each iteration, one or more LIVEOUT sets grows in size, unless they all remain unchanged. Once the complete set of LIVEOUT sets remain unchanged in one iteration, they will not change in subsequent iterations. It will have reached a fixed point.

	LIVEOUT(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
—	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	∅	∅	{a,b,c,d,i}	∅	∅	∅	∅	{a,b,c,d,i}	∅
2	∅	{a,i}	{a,b,c,d,i}	{i}	∅	∅	{a,c,d,i}	{a,b,d,c,i}	{a,c,d,i}
3	{i}	{a,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
4	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
5	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}

■ **FIGURE 9.3** Progress of the Iterative Live Solver on the Example From Figure 9.2.

We know that the algorithm will reach a fixed point because the LIVEOUT sets are finite. The size of any LIVEOUT set is bounded by the number of variables, $|V|$; any LIVEOUT set is either V or a proper subset of V. In the worst case, one LIVEOUT set would grow by one element in each iteration; that behavior would halt after $n \cdot |V|$ iterations, where n is the number of nodes in the CFG.

V is {a, b, c, d, i, y, z} in the code from Figure 9.2. |*V*| is seven.

This property, the termination of the iterative algorithm because of the combination of monotonicity and the finite number of possible values for the underlying sets, is often called the *finite descending chain property*. In the dominance problem, the DOM sets shrink monotonically and the DOM sets are bounded by the number of nodes in the CFG. That combination, monotonicity and bounded size, again guarantees termination.

Correctness

Iterative live analysis is correct if and only if it finds all the variables that satisfy the definition of liveness at the end of each block. Recall the definition: A variable v is *live* at point p if and only if there is a path from p to a use of v along which v is not redefined. Thus, liveness is defined in terms of paths in the CFG. A path that contains no definitions of v must exist from p to a use of v. We call such a path a v-clear path.

LIVEOUT(n) should contain v if and only if v is live at the end of block n. To form LIVEOUT(n), the iterative solver computes the contribution to LIVEOUT(n) of each successor of n in the CFG. It combines these contributions using union because $v \in$ LIVEOUT(n) if v is live on *any* path leaving n. How does this local computation over individual edges relate to liveness defined over all paths?

The LIVEOUT sets computed by the iterative solver are a fixed-point solution to the live equations. Again, the theory of iterative data-flow analysis

STATIC ANALYSIS VERSUS DYNAMIC ANALYSIS

The notion of static analysis leads directly to the question, What about dynamic analysis? By definition, static analysis tries to estimate, at compile time, what will happen at runtime. In many situations, the compiler cannot tell what will happen, even though the answer might be obvious with knowledge of one or more runtime values.

Consider, for example, the C fragment

```
x = y * z + 12;
*p = 0;
q = y * z + 13;
```

It contains a redundant expression, y * z, if and only if p does not contain the address of either y or z. At compile time, the value of p and the address of y and z may be unknown. At runtime, they are known and can be tested. Testing these values at runtime would allow the code to avoid recomputing y * z, where compile-time analysis might be unable to answer the question.

However, the cost of testing whether p == &y or p == &z or neither and acting on the result is likely to exceed the cost of recomputing y * z. For dynamic analysis to make sense, it must be a priori profitable—that is, the savings must exceed the cost of the analysis. This happens in some cases; in most cases, it does not. In contrast, the cost of static analysis can be amortized over multiple runs of the executable code, so it is more attractive, in general.

assures us that these particular equations have a unique fixed point [210]. The uniqueness of the fixed point guarantees that the fixed-point solution computed by the iterative algorithms is identical to the meet-over-all-paths solution called for by the definition.

Efficiency

It is tempting to think that RPO on the reverse CFG is equivalent to reverse preorder on the CFG. See Exercise 4 at the end of the chapter for a counter-example.

For a backward problem, such as LiveOut, the solver should use an RPO traversal on the reverse CFG, as shown in Figure 9.4. The iterative evaluation shown earlier used RPO on the CFG. For the example CFG, one RPO on the reverse CFG is

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
RPO(*n*)	8	7	6	1	0	5	4	2	3

```
for i ← 0 to |N| - 1
    LiveOut(i) ← ∅
changed ← true
while (changed)
  changed ← false
  for i ← 1 to |N| - 1
     j ← RPO[i]      // Computed on reverse CFG
     $\text{LiveOut}(j) \leftarrow \bigcup_{k \in succ(j)} \text{UEVar}(k) \cup (\text{LiveOut}(k) \cap \overline{\text{VarKill}(k)})$
     if LiveOut(j) has changed then
        changed ← true
```

■ **FIGURE 9.4** Round-Robin, Reverse Postorder Solver for LiveOut.

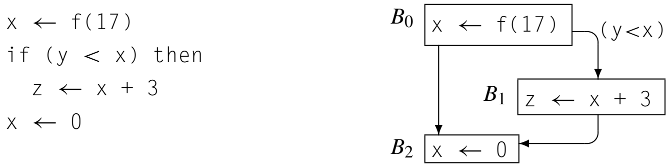
	LiveOut(*n*)								
	***B*₀**	***B*₁**	***B*₂**	***B*₃**	***B*₄**	***B*₅**	***B*₆**	***B*₇**	***B*₈**
—	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	{i}	{a,c,i}	{a,b,c,d,i}	∅	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
2	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
3	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}

■ **FIGURE 9.5** Iterations of Live Analysis Using RPO on the Reverse CFG.

Visiting the nodes in RPO on the reverse CFG produces the iterations shown in Figure 9.5. Now, the algorithm halts in three iterations, rather than the five iterations required with a traversal ordered by RPO on the CFG. Comparing this table against the earlier computation, we can see why. On the first iteration, the algorithm computed correct LiveOut sets for all nodes except B_3. It took a second iteration for B_3 because of the back edge—the edge from B_3 to B_1. The third iteration is needed to recognize that the algorithm has reached its fixed point.

9.2.3 Limitations on Data-Flow Analysis

There are limits to what a compiler can learn from data-flow analysis. In some cases, the limits arise from the assumptions underlying the analysis. In other cases, the limits arise from features of the language being analyzed. To make informed decisions, the compiler writer must understand what data-flow analysis can do and what it cannot do.



```
x ← f(17)
if (y < x) then
  z ← x + 3
x ← 0
```

(a) Simple If-Then Construct

(b) Corresponding Control-Flow Graph

■ **FIGURE 9.6** Control Flow Limits the Precision of Data-Flow Analysis.

When it computes the LiveOut set for a node n in the CFG, the iterative algorithm uses the sets LiveOut, UEVar, and VarKill for all of n's successors in the CFG. This implicitly assumes that execution can reach all of those successors; in practice, one or more of them may not be reachable. Consider the code fragment shown in Figure 9.6 along with its CFG.

The assignment to x in B_0 is live because of the use of x in B_1. The assignment to x in B_2 kills the value set in B_0. If B_1 cannot execute, then x's value from B_0 is not live past the comparison with y, and $x \notin$ LiveOut(B_0). If the compiler can prove that the test (y < x) is always false, then control will never transfer to block B_1 and the assignment to z will never execute. If the call to f has no side effects, the entire statement in B_0 is useless and need not be executed. Since the test's result is known, the compiler can completely eliminate both blocks B_0 and B_1.

The equations for LiveOut, however, take the union over all successors of the block, not just the block's executable successors. Thus, the analyzer computes LiveOut(B_0) as

$$\begin{gathered}\textsc{UEVar}(B_1) \cup (\textsc{LiveOut}(B_1) \cap \overline{\textsc{VarKill}(B_1)}) \bigcup \\ \textsc{UEVar}(B_2) \cup (\textsc{LiveOut}(B_2) \cap \overline{\textsc{VarKill}(B_2)})\end{gathered}$$

Data-flow analysis assumes that all paths through the CFG are feasible. Thus, the information that they compute summarizes the possible data-flow events, assuming that each path can be taken. This limits the precision of the resulting information; we say that the information is precise "up to symbolic execution." With this assumption, $x \in$ LiveOut(B_0) and both B_0 and B_1 must be preserved.

Another way that imprecision creeps into the results of data-flow analysis comes from the treatment of arrays, pointers, and procedure calls. An array reference, such as A[i,j,k], refers to a single element of A. However, without analysis that reveals the values of i, j, and k, the compiler cannot tell which element of A is being accessed. For this reason, compilers have traditionally lumped together all references to an array A. Thus, a use of `A[x,y,z]` counts as a use of `A`, and a definition of `A[c,d,e]` counts as a definition of `A`.

Some care must be taken, however, to avoid making too strong an inference. The compiler, knowing that its information on arrays is imprecise, must interpret that information conservatively. Thus, if the goal of the analysis is to determine where a value is no longer live (that is, the value must have been killed), a definition of `A[i,j,k]` does not kill the value of `A`. If the goal is to recognize where a value *might* not survive, then a definition of `A[i,j,k]` *might* define any element of `A`.

Pointers add another level of imprecision to the results of static analysis. Explicit arithmetic on pointers makes matters worse. Without an analysis that specifically tracks the values of pointers, the compiler must interpret an assignment to a pointer-based variable as a potential definition for every variable that the pointer might reach. Type safety can limit the set of objects potentially defined by an assignment through a pointer; a pointer declared as pointing to an object of type *t* can only be used to modify objects of type *t*. Without analysis of pointer values or a guarantee of type safety, assignment to a pointer-based variable can force the analyzer to assume that every variable has been modified. In practice, this effect often prevents the compiler from keeping the value of a pointer-based variable in a register across any pointer-based assignment. Unless the compiler can specifically prove that the pointer used in the assignment cannot refer to the memory location corresponding to the enregistered value, it cannot safely keep the value in a register.

The complexity of analyzing pointer use leads many compilers to avoid keeping values in registers if they can be the target of a pointer. Usually, some variables can be exempted from this treatment—such as a local variable whose address has never been explicitly taken. The alternative is to perform data-flow analysis aimed at disambiguating pointer-based references—reducing the set of possible variables that a pointer might reference at each point in the code. If the program can pass pointers as parameters or use them as global variables, pointer disambiguation becomes inherently interprocedural.

Procedure calls provide a final source of imprecision. To understand the data flow in the current procedure, the compiler must know what the callee can do to each variable that is accessible to both caller and callee. The callee may, in turn, call other procedures that have their own potential side effects.

Unless the compiler computes accurate summary information for each procedure call, it must estimate their worst-case behavior. While the specific assumptions vary from problem to problem, the general rule is to assume that the callee both uses and modifies every variable that it can address and that call-by-reference parameters create ambiguous references. Since few procedures exhibit this behavior, this assumption typically overestimates the effects of a call and introduces further imprecision into the results of data-flow analysis.

9.2.4 Other Data-Flow Problems

Compilers use data-flow analyses to prove the safety of applying transformations in particular situations. Thus, many distinct data-flow problems have been proposed, each to drive a particular optimization.

Available Expressions

To identify redundant expressions, the compiler can compute information about the *availability* of expressions. An expression e is *available* at point p in a procedure if and only if on every path from the procedure's entry to p, e is evaluated and none of its constituent subexpressions is redefined between that evaluation and p. This analysis annotates each node n in the CFG with a set AVAILIN(n), which contains the names of all expressions in the procedure that are available on entry to the block corresponding to n. To compute AVAILIN, the compiler initially sets

$$\text{AVAILIN}(n_0) = \emptyset$$

$$\text{AVAILIN}(n) = \{\, \textit{all expressions}\, \}, \forall\, n \neq n_0$$

Next, it solves the following equations:

$$\text{AVAILIN}(n) = \bigcap_{m \in preds(n)} (\text{DEEXPR}(m) \;\cup\; (\text{AVAILIN}(m) \;\cap\; \overline{\text{EXPRKILL}(m)}))$$

Here, DEEXPR(n) is the set of downward exposed expressions in n. An expression $e \in$ DEEXPR(n) if and only if block n evaluates e and none of e's operands is defined between the last evaluation of e in n and the end of n. EXPRKILL(n) contains all those expressions that are "killed" by a definition in n. An expression is killed if one or more of its operands are redefined in the block. Note that the equation defines a forward data-flow problem.

An expression e is available on entry to n if and only if it is available on exit from each of n's predecessors in the CFG. As the equation states, an expression e is available on exit from some block m if one of two conditions holds: either e is downward exposed in m, or it is available on entry to m and is not killed in m.

AVAILIN sets can be used to perform global redundancy elimination, sometimes called *global common subexpression elimination*. Perhaps the simplest way to achieve this effect is to compute AVAILIN sets for each block and use them in local value numbering (see Section 8.4.1). The compiler can simply initialize the hash table for a block b to AVAILIN(b) before value numbering b. Lazy code motion is a stronger form of common subexpression elimination that also uses availability (see Section 10.3.1).

Reaching Definitions

In some cases, the compiler needs to know where an operand was defined. If multiple paths in the CFG lead to the operation, then multiple definitions may provide the value of the operand. To find the set of definitions that reach a block, the compiler can compute *reaching definitions*. The domain of REACHES is the set of definitions in the procedure. A definition d of some variable v *reaches* operation i if and only if i reads the value of v and there exists a path from d to i that does not define v.

The compiler annotates each node n in the CFG with a set REACHES(n), computed as a forward data-flow problem:

$$\text{REACHES}(n) = \emptyset, \forall n$$

$$\text{REACHES}(n) = \bigcup_{m \in preds(n)} (\text{DEDEF}(m) \cup (\text{REACHES}(m) \cap \overline{\text{DEFKILL}(m)}))$$

DEDEF(m) is the set of downward-exposed definitions in m: those definitions in m for which the defined name is not subsequently redefined in m. DEFKILL(m) contains *all* the definition points that are obscured by a definition of the same name in m; $d \in$ DEFKILL(m) if d defines some name v and m contains a definition that also defines v. Thus $\overline{\text{DEFKILL}(m)}$ consists of the definition points that are not obscured in m.

DEDEF and DEFKILL are both defined over the set of definition points, but computing each of them requires a mapping from names (variables and compiler-generated temporaries) to definition points. Thus, gathering the initial information for reaching definitions is more complex than it is for live variables.

Anticipable Expressions

An expression e is considered *anticipable*, or *very busy*, on exit from block b if and only if (1) every path that leaves b evaluates and subsequently uses e, and (2) evaluating e at the end of b would produce the same result as

IMPLEMENTING DATA-FLOW FRAMEWORKS

The equations for many global data-flow problems show a striking similarity. For example, available expressions, live variables, reaching definitions, and anticipable expressions all have propagation functions of the form

$$f(x) = c_1 \;\; op_1 \; (x \;\; op_2 \;\; c_2)$$

where c_1 and c_2 are constants determined by the actual code and op_1 and op_2 are standard set operations such as ∪ and ∩. This similarity shows up in the problem descriptions. It should also show up in their implementations.

The compiler writer can easily abstract away the details in which these problems differ and implement a single, parameterized analyzer. The analyzer needs functions to compute c_1 and c_2, implementations of the operators, and an indication of the problem's direction. In return, it produces the desired data-flow information.

This implementation strategy encourages code reuse. It hides the low-level details of the solver. At the same time, it creates a situation in which the compiler writer can profitably invest effort in optimizing the implementation. For example, a scheme that allows the framework to implement $f(x) = c_1 \; op_1 \; (x \; op_2 \; c_2)$ as a single function may outperform an implementation that uses $f_1(x) = c_1 \; op_1 \; x$ and $f_2(x) = x \; op_1 \; c_2$ and computes $f(x)$ as $f_1(f_2(x))$. This scheme lets all the client transformations benefit from optimizing set representations and operator implementations.

the first evaluation of *e* along each of those paths. The term "anticipable" derives from the second condition, which implies that an evaluation of *e* at *b* anticipates the subsequent evaluations along all paths. The set of expressions anticipable on output from a block can be computed as a backward data-flow problem on the CFG. The domain of the problem is the set of expressions.

$$\textsc{AntOut}(n_f) = \emptyset$$

$$\textsc{AntOut}(n) = \{\; \textit{all expressions} \;\}, \forall n \neq n_f$$

$$\textsc{AntOut}(n) \;=\; \bigcap_{m \in succ(n)} (\textsc{UEExpr}(m) \cup (\textsc{AntOut}(m) \cap \overline{\textsc{ExprKill}(m)}))$$

Here $\textsc{UEExpr}(m)$ is the set of upward-exposed expressions—those used in *m* before they are killed. $\textsc{ExprKill}(m)$ is the set of expressions defined in *m*; it is the same set that appears in the equations for available expressions.

The results of anticipability analysis are used in code motion both to decrease execution time, as in lazy code motion, and to shrink the size of the compiled code, as in code hoisting. Both transformations are discussed in Section 10.3.

Interprocedural Summary Problems

When analyzing a single procedure, the compiler must account for the impact of each procedure call. In the absence of specific information about the call, the compiler must make worst-case assumptions that account for all the possible actions of the callee, or any procedures that it, in turn, calls. These worst-case assumptions can seriously degrade the quality of the global data-flow information. For example, the compiler must assume that the callee modifies every variable that it can access; this assumption essentially stops the propagation of facts across a call site for all global variables, module-level variables, and call-by-reference parameters.

To limit such impact, the compiler can compute summary information on each call site. The classic summary problems compute the set of variables that might be modified as a result of the call and that might be used as a result of the call. The compiler can then use these computed summary sets in place of its worst case assumptions.

The *interprocedural may modify problem* annotates each call site with a set of names that the callee, and procedures it calls, might modify. May modify is one of the simplest problems in interprocedural analysis, but it can have a significant impact on the quality of information produced by other analyses, such as global constant propagation. May modify is posed as a set of data-flow equations over the program's call graph that annotate each procedure with a MAYMOD set.

$$\textsc{MayMod}(p) = \textsc{LocalMod}(p) \cup \left(\bigcup_{e=(p,q)} \mathit{unbind}_e(\textsc{MayMod}(q)) \right)$$

where $e = (p,q)$ is an edge from p to q in the call graph. The function unbind_e maps one set of names into another. For a call-graph edge $e = (p,q)$, $\mathit{unbind}_e(x)$ maps each name in x from the name space of q to the name space of p, using the bindings at the specific call site that corresponds to e. Finally, LOCALMOD(p) contains all the names modified locally in p that are visible outside p. It is computed as the set of names defined in p minus any names that are strictly local to p.

Flow insensitive
This formulation of MAYMOD ignores control flow inside procedures. Such a formulation is said to be *flow insensitive*.

To solve for MAYMOD, the compiler can set MAYMOD(p) to LOCALMOD(p), for all procedures p, and then iteratively evaluate the equation for MAYMOD until it reaches a fixed point. Given the MAYMOD sets for each procedure, the compiler can compute the set of names that might be modified at a specific call, $e = (p,q)$, by computing a set S as $\mathit{unbind}_e(\textsc{MayMod}(q))$ and then adding to S any names that are aliased inside procedure p to names in S.

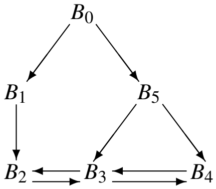
The compiler can also compute information on what variables might be referenced as a result of executing a procedure call, the *interprocedural may reference problem*. The equations to annotate each procedure *p* with a set MAYREF(*p*) are similar to the equations for MAYMOD.

SECTION REVIEW

Iterative data-flow analysis works by repeatedly re-evaluating the data-flow equation at each node in the underlying graph until the sets defined by the equations reach a fixed point. Many data-flow problems have a unique fixed point, which ensures a correct solution independent of the evaluation order evaluation, and the finite descending chain property, which guarantees termination independent of the evaluation order. Since the analyzer can choose any order, it should choose one that produces rapid termination. For most forward data-flow problems, that order is reverse postorder; for most backward problems, that order is reverse postorder on the reverse CFG. These orders force the iterative algorithm to evaluate as many predecessors (for forward problems) or successors (for backward problems) as possible before it evaluates a node *n*.

Many data-flow problems appear in the literature and in modern compilers. Examples include live analysis, used in register allocation; availability and anticipability, used in redundancy elimination and code motion; and interprocedural summary information, used to sharpen the results of single-procedure data-flow analysis. SSA form, described in the next section, provides a unifying structure that encodes both data-flow information, such as reaching definitions, and control-flow information, such as dominance. Many modern compilers use SSA form as an alternative to solving multiple distinct data-flow problems.

Review Questions



1. Compute DOM sets for the CFG shown in the margin, evaluating the nodes in the order {B_4, B_2, B_1, B_5, B_3, B_0}. Explain why this calculation takes a different number of iterations than the version shown on page 482.
2. Before a compiler can compute interprocedural data-flow information, it must build a call graph for the program. Just as ambiguous jumps complicate CFG construction, so too can ambiguous calls complicate call-graph construction. What language features might lead to an ambiguous call site—one where the compiler was uncertain as to the identify of the callee?

9.3 STATIC SINGLE-ASSIGNMENT FORM

Over time, many different data-flow problems have been formulated. If each transformation uses its own idiosyncratic analysis, the amount of time and effort spent implementing, debugging, and maintaining the analysis passes can grow unreasonably large. To limit the number of analyses that the compiler writer must implement and that the compiler must run, it is desirable to use a single analysis to perform multiple transformations.

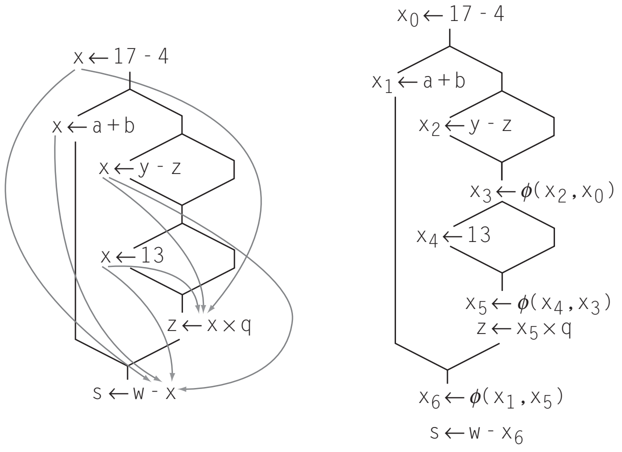
One strategy for implementing such a "universal" analysis involves building a variant form of the program that encodes both data flow and control flow directly in the IR. SSA form, introduced in Sections 5.4.2 and 8.5.1, has this property. It can serve as the basis for a large set of transformations. From a single implementation that translates the code into SSA form, a compiler can perform many of the classic scalar optimizations.

Consider the various uses of the variable x in the code fragment shown in Figure 9.7a. The gray lines show which definitions can reach each use of x. Figure 9.7b shows the same fragment, rewritten to convert x to SSA form. Definitions of x have been renamed, with subscripts, to ensure that each definition has a unique SSA name. For simplicity, we have left the references to other variables unchanged.

The SSA form of the code includes new assignments (to x_3, x_5, and x_6) that reconcile the distinct SSA names for x with the uses of x (in the assignments to s and z). These assignments ensure that, along each edge in the CFG, the current value of x has been assigned a unique name, independent of which path brought control to the edge. The right sides of these assignments contain a special function, a ϕ-function, that combines the values from distinct edges.

A ϕ-function takes as arguments the SSA names for the values associated with each edge that enters the block. When control enters a block, all the ϕ-functions in the block execute, concurrently. They evaluate to the argument that corresponds to the edge along which control entered the block. Notationally, we write the arguments left-to-right to correspond to the edges left-to-right. On the printed page, this is easy. In an implementation, it requires some bookkeeping.

The SSA construction inserts ϕ-functions after each point in the CFG where multiple paths converge—each join point. At join points, distinct SSA names must be reconciled to a single name. After the entire procedure has been converted to SSA form, two rules hold: (1) each definition in the procedure creates a unique name, and (2) each use refers to a single definition.



(a) Original Code Fragment (b) With x in SSA Form

■ **FIGURE 9.7** SSA: Encoding Control Flow into Data Flow.

To transform a procedure into SSA form, the compiler must insert the appropriate ϕ-functions for each variable into the code, and it must rename variables with subscripts to make the two rules hold. This simple, two-step plan produces the basic SSA construction algorithm.

9.3.1 A Simple Method for Building SSA Form

To construct the SSA form of a program, the compiler must insert ϕ-functions at join points in the CFG, and it must rename variables and temporary values to conform with the rules that govern the SSA name space. The algorithm follows this outline:

1. *Inserting ϕ-functions* At the start of each block that has multiple predecessors, insert a ϕ-function, such as y ← ϕ(y, y), for every name y that the code either defines or uses in the current procedure. The ϕ-function should have one argument for each predecessor block in the CFG. This rule inserts a ϕ-function in every case where one is needed. It also inserts many extraneous ϕ-functions.
   The algorithm can insert the ϕ-functions in arbitrary order. The definition of ϕ-functions requires that all the ϕ-functions at the top of a block execute concurrently—that is, they all read their input parameters simultaneously, then write their output values simultaneously. This lets the algorithm avoid many minor details that an ordering might introduce.

2. *Renaming* After ϕ-functions have been inserted, the compiler can compute reaching definitions (see Section 9.2.4). Because the inserted ϕ-functions are also definitions, they ensure that only one definition reaches any use. Next, the compiler can rename each use, both the variables and the temporaries, to reflect the definition that reaches it.

   The compiler must sort out the definitions that reach each ϕ-function and make the names correspond to the paths along which they reach the block that contains the ϕ-function. While conceptually simple, this task requires some bookkeeping.

This algorithm constructs a correct SSA form for the program. Each variable is defined exactly once, and each reference uses the name of a distinct definition. However, it produces SSA form that has, potentially, many more ϕ-functions than necessary. The extra ϕ-functions are problematic. They decrease the precision of some kinds of analysis when performed over SSA form. They occupy space, so the compiler wastes memory representing ϕ-functions that are either redundant (that is, $x_j \leftarrow \phi(x_i, x_i)$) or are not live. They increase the cost of any algorithm that uses the resulting SSA form, since it must traverse all the extraneous ϕ-functions.

We call this version of SSA *maximal* SSA *form*. To build SSA form with fewer ϕ-functions requires more work; in particular, the compiler must analyze the code to determine where potentially distinct values converge in the CFG. This computation relies on the dominance information described in Section 9.2.1.

The next three subsections present, in detail, an algorithm to build *semipruned* SSA *form*—a version with fewer ϕ-functions. Section 9.3.2 shows how dominance information introduced in Section 9.2.1 can be used to compute *dominance frontiers* to guide insertion of ϕ-functions. Section 9.3.3 gives an algorithm to insert ϕ-functions, and Section 9.3.4 shows how to rewrite variable names to complete the construction of SSA form. Section 9.3.5 discusses the difficulties that can arise in converting the code back into an executable form.

9.3.2 Dominance Frontiers

The primary problem with maximal SSA form is that it contains too many ϕ-functions. To reduce their number, the compiler must determine more carefully where they are required. The key to placing ϕ-functions lies in understanding which variables need a ϕ-function at each join point. To solve this problem efficiently and effectively, the compiler can turn the question around. It can determine, for each block *i*, the set of blocks that will need a

ϕ-function for any definition in block i. Dominance plays a critical role in this computation.

Consider a definition in node n of the CFG. That value could potentially reach every node m where $n \in \text{Dom}(m)$ without need for a ϕ-function, since every path that reaches m passes through n. The only way that the value does not reach m is if another definition of the same name intervenes—that is, it occurs in some node p between n and m. In this case, the definition in n does not force the presence of a ϕ-function; instead, the redefinition in p does.

Strict dominance
a strictly dominates *b* if and only if $a \in \text{Dom}(b) - \{b\}$.

A definition in node n forces a ϕ-function at join points that lie just outside the region of the CFG that n dominates. More formally, a definition in node n forces a corresponding ϕ-function at any join point m where (1) n dominates a predecessor of m ($q \in preds(m)$ and $n \in \text{Dom}(q)$), and (2) n does not *strictly dominate* m. (Using strict dominance rather than dominance allows a ϕ-function at the start of a single-block loop. In that case, $n = m$, and $m \notin \text{Dom}(n) - \{n\}$.) We call the collection of nodes m that have this property with respect to n the *dominance frontier* of n, denoted DF(n).

Informally, DF(n) contains the first nodes reachable from n that n does not dominate, on each CFG path leaving n. In the CFG of our continuing example, B_5 dominates B_6, B_7, and B_8, but does not dominate B_3. On every path leaving B_5, B_3 is the first node that B_5 does not dominate. Thus, DF(B_5) = $\{B_3\}$.

Dominator Trees

Dominator tree
a tree that encodes the dominance information for a flow graph

Before giving an algorithm to compute dominance frontiers, we must introduce one further notion, the *dominator tree*. Given a node n in a flow graph, the set of nodes that strictly dominate n is given by $(\text{Dom}(n) - n)$. The node in that set that is closest to n is called n's immediate dominator, denoted IDom(n). The entry node of the flow graph has no immediate dominator.

The dominator tree of a flow graph contains every node of the flow graph. Its edges encode the IDom sets in a simple way. If m is IDom(n), then the dominator tree has an edge from m to n. The dominator tree for our example CFG appears in the margin. Notice that B_6, B_7, and B_8 are all children of B_5, even though B_7 is not an immediate successor of B_5 in the CFG.

The dominator tree compactly encodes both the IDom information and the complete Dom sets for each node. Given a node n in the dominator tree, IDom(n) is just its parent in the tree. The nodes in Dom(n) are exactly the nodes that lie on the path from the root of the dominator tree to n,

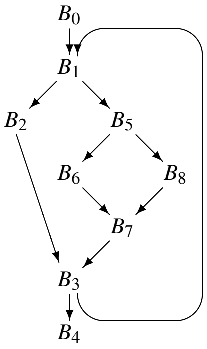
```
for all nodes, n, in the CFG
    DF(n) ← ∅

for all nodes, n, in the CFG
    if n has multiple predecessors then
        for each predecessor p of n
            runner ← p
            while runner ≠ IDom(n)
                DF(runner) ← DF(runner) ∪ {n}
                runner ← IDom(runner)
```

■ **FIGURE 9.8** Algorithm for Computing Dominance Frontiers.

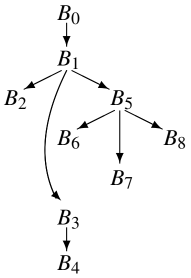
inclusive of both the root and *n*. From the tree, we can read the following sets:

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DOM	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
IDOM	—	0	1	1	3	1	5	5	5

These Dom sets match those computed earlier;—indicates an undefined value.



The Example CFG



Its Dominator Tree

Computing Dominance Frontiers

To make ϕ-insertion efficient, we need to calculate the dominance frontier for each node in the flow graph. We could formulate a data-flow problem to compute DF(n) for each n in the graph. Using both the dominator tree and the CFG, we can formulate a simple and direct algorithm, shown in Figure 9.8. Since only nodes that are join points in the CFG can be members of a dominance frontier, we first identify all of the join points in the graph. For a join point j, we examine each of its CFG predecessors.

The algorithm is based on three observations. First, nodes in a DF set must be join points in the graph. Second, for a join point j, each predecessor k of j must have $j \in$ DF(k), since k cannot dominate j if j has more than one predecessor. Finally, if $j \in$ DF(k) for some predecessor k, then j must also be in DF(l) for each $l \in$ Dom(k), unless $l \in$ Dom(j).

The algorithm follows these observations. It locates nodes j that are join points in the CFG. Then, for each predecessor p of j, it walks up the dominator tree from p until it finds a node that dominates j. From the second and third observations in the preceding paragraph, j belongs in DF(l) for each node l that the algorithm traverses in this dominator-tree walk, except for the final node in the walk, since that node dominates j. A small amount of bookkeeping is needed to ensure that any n is added to a node's dominance frontier only once.

To see how this works, consider again the example CFG and its dominance tree. The analyzer examines the nodes in some order, looking for nodes with multiple predecessors. Assuming that it takes the nodes in name order, it finds the join points as B_1, then B_3, then B_7.

1. B_1 For CFG-predecessor B_0, the algorithm finds that B_0 is IDOM(B_1), so it never enters the while loop. For CFG-predecessor B_3, it adds B_1 to DF(B_3) and advances to B_1. It adds B_1 to DF(B_1) and advances to B_0, where it halts.
2. B_3 For CFG-predecessor B_2, it adds B_3 to DF(B_2), advances to B_1 which is IDOM(B_3), and halts. For CFG-predecessor B_7, it adds B_3 to DF(B_7) and advances to B_5. It adds B_3 to DF(B_5) and advances to B_1, where it halts.
3. B_7 For CFG-predecessor B_6, it adds B_7 to DF(B_6), advances to B_5 which is IDOM(B_7), and halts. For CFG-predecessor B_8, it adds B_7 to DF(B_8) and advances to B_5, where it halts.

Accumulating these results, we obtain the following dominance frontiers:

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DF	$\emptyset$	$\{B_1\}$	$\{B_3\}$	$\{B_1\}$	$\emptyset$	$\{B_3\}$	$\{B_7\}$	$\{B_3\}$	$\{B_7\}$

9.3.3 Placing ϕ-Functions

The naive algorithm placed a ϕ-function for every variable at the start of every join node. With dominance frontiers, the compiler can determine more precisely where ϕ-functions might be needed. The basic idea is simple. A definition of x in block b forces a ϕ-function at every node in DF(b). Since that ϕ-function is a new definition of x, it may, in turn, force the insertion of additional ϕ-functions.

The compiler can further narrow the set of ϕ-functions that it inserts. A variable that is only live within a single block can never have a live ϕ-function. To apply this observation, the compiler can compute the set of names that

```
Globals ← ∅
Initialize all the Blocks sets to ∅
for each block b
    VarKill ← ∅
    for each operation i in b, in order
        assume that $op_i$ is “x ← y op z”
        if y ∉ VarKill then
            Globals ← Globals ∪ {y}
        if z ∉ VarKill then
            Globals ← Globals ∪ {z}
        VarKill ← VarKill ∪ {x}
        Blocks(x) ← Blocks(x) ∪ {b}
```

(a) Finding Global Names

```
for each name x ∈ Globals
    WorkList ← Blocks(x)
    for each block b ∈ WorkList
        for each block d in DF(b)
            if d has no φ-function for x then
                insert a φ-function for x in d
                WorkList ← WorkList ∪ {d}
```

(b) Rewriting the Code

■ **FIGURE 9.9** ϕ-Function Insertion.

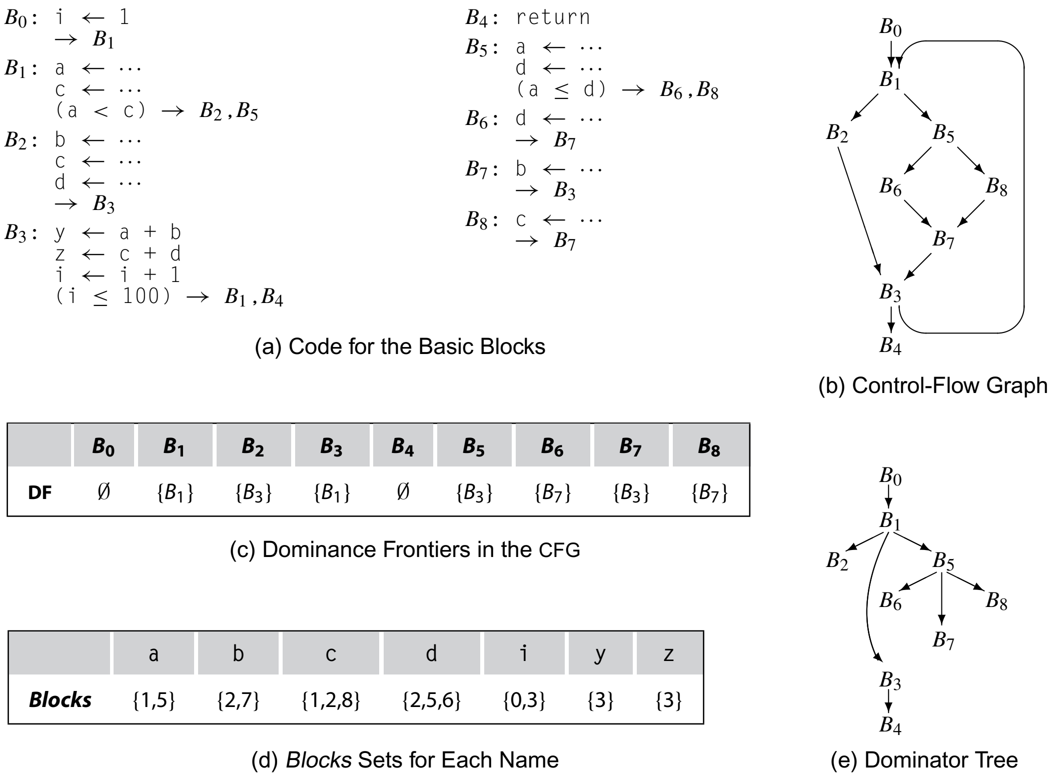
are live across multiple blocks—a set that we will call the *global names*. It can insert ϕ-functions for those names and ignore any name that is not in that set. (This restriction distinguishes semipruned SSA form from other varieties of SSA form.)

The word *global* is used here to mean of interest across the entire procedure.

The compiler can find the global names cheaply. In each block, it looks for names with upward-exposed uses—the UEVar set from the live-variables calculation. Any name that appears in one or more LiveOut sets must be in the UEVar set of some block. Taking the union of all the UEVar sets gives the compiler the set of names that are live on entry to one or more blocks and, hence, live in multiple blocks.

The algorithm, shown in Figure 9.9a, is derived from the obvious algorithm for computing UEVar. It constructs a single set, `Globals`, where the LiveOut computation must compute a distinct set for each block. As it builds the `Globals` set, it also constructs, for each name, a list of all blocks that contain a definition of that name. These block lists serve as an initial worklist for the ϕ-insertion algorithm.

The algorithm for inserting ϕ-functions is shown in Figure 9.9b. For each global name x, it initializes `WorkList` with `Blocks(x)`. For each block b on the `WorkList`, it inserts ϕ-functions at the head of every block d in b's dominance frontier. Since all the ϕ-functions in a block execute concurrently, by definition, the algorithm can insert them at the head of d in any order. After



```
B0: i ← 1                         B4: return
    → B1                          B5: a ← ···
B1: a ← ···                           d ← ···
    c ← ···                           (a ≤ d) → B6,B8
    (a < c) → B2,B5               B6: d ← ···
B2: b ← ···                           → B7
    c ← ···                       B7: b ← ···
    d ← ···                           → B3
    → B3                          B8: c ← ···
B3: y ← a + b                         → B7
    z ← c + d
    i ← i + 1
    (i ≤ 100) → B1,B4
```

(a) Code for the Basic Blocks

(b) Control-Flow Graph

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DF	Ø	$\{B_1\}$	$\{B_3\}$	$\{B_1\}$	Ø	$\{B_3\}$	$\{B_7\}$	$\{B_3\}$	$\{B_7\}$

(c) Dominance Frontiers in the CFG

	a	b	c	d	i	y	z
Blocks	{1,5}	{2,7}	{1,2,8}	{2,5,6}	{0,3}	{3}	{3}

(d) *Blocks* Sets for Each Name

(e) Dominator Tree

■ **FIGURE 9.10** Example SSA for ϕ-function Insertion.

adding a ϕ-function for x to d, the algorithm adds d to the *WorkList* to reflect the new assignment to x in d.

Example

Figure 9.10 recaps our running example. Panel a shows the code; panel b shows the CFG; panel c shows the dominance frontiers for each block; and panel e shows the dominator tree built from the CFG.

The first step in the ϕ-function insertion algorithm finds global names and computes the *Blocks* set for each name. For the code in Figures 9.10a, the global names are {a,b,c,d,i}. Figure 9.10d shows the *Blocks* sets. Notice that the algorithm creates *Blocks* sets for y and z, even though they are not in *Globals*. Separating the computation of *Globals* from that of

```
B0: i ← 1
    → B1
B1: a ← φ(a,a)
    b ← φ(b,b)
    c ← φ(c,c)
    d ← φ(d,d)
    i ← φ(i,i)
    a ← ···
    c ← ···
    (a < c) → B2,B5
B2: b ← ···
    c ← ···
    d ← ···
    → B3

B3: a ← φ(a,a)
    b ← φ(b,b)
    c ← φ(c,c)
    d ← φ(d,d)
    y ← a + b
    z ← c + d
    i ← i + 1
    (i ≤ 100) → B1,B4
B4: return
B5: a ← ···
    d ← ···
    (a ≤ d) → B6,B8

B6: d ← ···
    → B7
B7: c ← φ(c,c)
    d ← φ(d,d)
    b ← ···
    → B3
B8: c ← ···
    → B7
```

■ **FIGURE 9.11** Example Code with ϕ-Functions, Before Renaming.

Blocks would avoid instantiating these extra sets, at the cost of another pass over the code.

The ϕ-function rewrite algorithm works on a name-by-name basis. Consider its actions for the variable a in the example. It initializes the worklist to Blocks(a), which contains B_1 and B_5. The definition in B_1 causes it to insert a ϕ-function at the start of each block in $\text{DF}(B_1) = \{B_1\}$. This action also enters B_1 back into the worklist. Next, it removes B_5 from the worklist and inserts a ϕ-function in each block of $\text{DF}(B_5) = \{B_3\}$. The insertion at B_3 also places B_3 on the worklist. When B_3 comes off the worklist, it tries to add a ϕ-function in B_1, because $B_1 \in \text{DF}(B_3)$. The algorithm notices that B_1 already has that ϕ-function, so it does not perform an insertion. Thus, processing of a halts with an empty worklist. The algorithm follows the same logic for each name in Globals, to produce the following insertions:

	a	b	c	d	i
ϕ-functions	$\{B_1,B_3\}$	$\{B_1,B_3\}$	$\{B_1,B_3,B_7\}$	$\{B_1,B_3,B_7\}$	$\{B_1\}$

The resulting code appears in Figure 9.11.

Limiting the algorithm to global names lets it avoid inserting dead ϕ-functions for x and y in block B_1. ($B_1 \in \text{DF}(B_3)$ and B_3 contains definitions of both x and y.) However, the distinction between local names and global names is not sufficient to avoid all dead ϕ-functions. For example, the ϕ-function for b in B_1 is not live because b is redefined before its value is used. To avoid inserting these ϕ-functions, the compiler can construct

THE DIFFERENT FLAVORS OF SSA FORM

Several distinct flavors of SSA form have been proposed in the literature. The flavors differ in their criteria for inserting ϕ-functions. For a given program, they can produce different sets of ϕ-functions.

Minimal SSA inserts a ϕ-function at any join point where two distinct definitions for the same original name meet. This is the minimal number consistent with the definition of SSA. Some of those ϕ-functions, however, may be dead; the definition says nothing about the values being live when they meet.

Pruned SSA adds a liveness test to the ϕ-insertion algorithm to avoid adding dead ϕ-functions. The construction must compute LIVEOUT sets, so the cost of building pruned SSAs is higher than that of building minimal SSA.

Semipruned SSA is a compromise between minimal SSAs and pruned SSAs. Before inserting ϕ-functions, the algorithm eliminates any names that are not live across a block boundary. This can shrink the name space and reduce the number of ϕ-functions without the overhead of computing LIVEOUT sets. This is the algorithm given in Figure 9.9.

Of course, the number of ϕ-functions depends on the specific program being converted into SSA form. For some programs, the reductions obtained by semipruned SSAs and pruned SSAs are significant. Shrinking the SSA form can lead to faster compilation, since passes that use SSA form then operate on programs that contain fewer operations—and fewer ϕ-functions.

LIVEOUT sets and add a test based on liveness to the inner loop of the ϕ-insertion algorithm. That modification causes the algorithm to produce *pruned* SSA *form*.

Efficiency Improvements

To improve efficiency, the compiler should avoid two kinds of duplication. First, the algorithm should avoid placing any block on the worklist more than once per global name. It can keep a checklist of blocks that have already been processed. Since the algorithm must reset the checklist for each global name, the implementation should use a sparse set or a similar structure (see Appendix B.2.3).

Second, a given block can be in the dominance frontier of multiple nodes that appear on the `WorkList`. As shown in the figure, the algorithm must search the block to look for a pre-existing ϕ-function. To avoid this search, the compiler can maintain a checklist of blocks that already contain ϕ-functions for x. This takes a single sparse set, reinitialized along with `WorkList`.

9.3.4 Renaming

In the description of maximal SSA form, we stated that renaming variables was conceptually straightforward. The details, however, require some explanation.

In the final SSA form, each global name becomes a base name, and individual definitions of that base name are distinguished by the addition of a numerical subscript. For a name that corresponds to a source-language variable, say x, the algorithm uses x as the base name. Thus, the first definition of x that the renaming algorithm encounters will be named x_0 and the second will be x_1. For a compiler-generated temporary, the algorithm must generate a distinct base name.

The algorithm, shown in Figure 9.12, renames both definitions and uses in a preorder walk over the procedure's dominator tree. In each block, it first renames the values defined by ϕ-functions at the head of the block, then it visits each operation in the block, in order. It rewrites the operands with current SSA names, then it creates a new SSA name for the result of the operation. This latter act makes the new name current. After all the operations in the block have been rewritten, the algorithm rewrites the appropriate ϕ-function parameters in each CFG successor of the block, using the current SSA names. Finally, it recurs on any children of the block in the dominator tree. When it returns from those recursive calls, it restores the set of current SSA names to the state that existed before the current block was visited.

To manage this process, the algorithm uses a counter and a stack for each global name. A global name's stack holds the subscript of the name's current SSA name. At each definition, the algorithm generates a new subscript for the targeted name by pushing the value of its current counter onto the stack and incrementing the counter. Thus, the value on top of the stack for n is always the subscript of n's current SSA name. As the final step in processing a block, the algorithm pops all the names generated in that block off their respective stacks to restore the names that held at the end of that block's immediate dominator. Those names may be needed to process the block's remaining siblings in the dominator tree.

The stack and the counter serve distinct and separate purposes. As control in the algorithm moves up and down the dominator tree, the stack is managed to simulate the lifetime of the most recent definition in the current block.

```
for each global name i
  counter[i] ← 0
  stack[i] ← ∅
Rename($n_0$)

NewName(n)
 i ← counter[n]
 counter[n] ← counter[n] + 1
 push i onto stack[n]
 return "$n_i$"
```

```
Rename(b)
  for each ϕ-function in b, "x ← ϕ(···)"
      rewrite x as NewName(x)

  for each operation "x ← y op z" in b
      rewrite y with subscript top(stack[y])
      rewrite z with subscript top(stack[z])
      rewrite x as NewName(x)

  for each successor of b in the CFG
      fill in ϕ-function parameters

  for each successor s of b in the dominator tree
      Rename(s)

  for each operation "x ← y op z" in b
      and each ϕ-function "x ← ϕ(···)"
      pop(stack[x])
```

■ **FIGURE 9.12** Renaming After ϕ-Insertion.

The counter, on the other hand, grows monotonically to ensure that each successive definition receives a unique SSA name.

Figure 9.12 summarizes the algorithm. It initializes the stacks and counters, then calls Rename on the root of the dominator tree—the entry node of the CFG. Rename rewrites the block and recurs on successors in the dominator tree. To finish with the block, Rename pops any names that were pushed onto stacks while processing the block. The function NewName manipulates the counters and stacks to create new SSA names as needed.

One final detail remains. At the end of block b, Rename must rewrite ϕ-function parameters in each of b's CFG successors. The compiler must assign an ordinal parameter slot in those ϕ-functions for b. When we draw the SSA form, we always assume a left-to-right order that matches the left-to-right order in which the edges are drawn. Internally, the compiler can number the edges and parameter slots in any consistent fashion that produces the desired result. This requires cooperation between the code that builds the SSA form and the code that builds the CFG. (For example, if the CFG implementation uses a list of edges leaving each block, the order of that list can determine the mapping.)

Example

To finish the continuing example, let's apply the renaming algorithm to the code in Figure 9.11. Assume that a_0, b_0, c_0, and d_0 are defined on entry to B_0. Figure 9.13 shows the states of the counters and stacks for global names at various points during the process.

	a	b	c	d	i
Counters	1	1	1	1	0
Stacks	a_0	b_0	c_0	d_0	

(a) Initial Condition, Before B_0

	a	b	c	d	i
Counters	1	1	1	1	1
Stacks	a_0	b_0	c_0	d_0	i_0

(b) On Entry to B_1

	a	b	c	d	i
Counters	3	2	3	2	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

(c) On Entry to B_2

	a	b	c	d	i
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_2	c_2	d_2	
			c_3		

(d) End of B_2

	a	b	c	d	i
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

(e) On Entry to B_3

	a	b	c	d	i
Counters	4	4	5	4	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_3	c_2	d_3	i_2
	a_3		c_4		

(f) At End of B_3

	a	b	c	d	i
Counters	4	4	5	4	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

(g) On Entry to B_5

	a	b	c	d	i
Counters	5	4	5	5	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

(h) Entry to B_6

	a	b	c	d	i
Counters	5	4	5	6	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

(i) Entry to B_7

	a	b	c	d	i
Counters	5	5	6	7	32
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

(j) On Entry to B_8

■ **FIGURE 9.13** States in the Renaming Example.

The algorithm makes a preorder walk over the dominator tree, which corresponds to visiting the nodes in ascending order by name, B_0 through B_8. The initial configuration of the stacks and counters appears in Figure 9.13a. As the algorithm proceeds through the blocks, it takes the following actions:

- *Block* B_0 This block contains only one operation. *Rename* rewrites `i` with `i`$_0$, increments the counter, and pushes `i`$_0$ onto the stack for `i`. Next, it visits B_0's CFG-successor, B_1, and rewrites the ϕ-function parameters that correspond to B_0 with their current names: `a`$_0$, `b`$_0$, `c`$_0$, `d`$_0$, and `i`$_0$. It then recurs on B_0's child in the dominator tree, B_1. After that, it pops the stack for `i` and returns.
- *Block* B_1 *Rename* enters B_1 with the state shown in Figure 9.13b. It rewrites the ϕ-function targets with new names, `a`$_1$, `b`$_1$, `c`$_1$, `d`$_1$, and `i`$_1$. Next, it creates new names for the definitions of `a` and `c` and rewrites them. It rewrites the uses of `a` and `c` in the comparison. Neither of B_1's CFG successors have ϕ-functions, so it recurs on B_1's dominator-tree children, B_2, B_3, and B_5. Finally, it pops the stacks and returns.
- *Block* B_2 *Rename* enters B_2 with the state shown in Figure 9.13c. This block has no ϕ-functions to rewrite. *Rename* rewrites the definitions of `b`, `c`, and `d`, creating a new SSA name for each. It then rewrites ϕ-function parameters in B_2's CFG successor, B_3. Figure 9.13d shows the stacks and counters just before they are popped. Finally, it pops the stacks and returns.
- *Block* B_3 *Rename* enters B_3 with the state shown in Figure 9.13e. Notice that the stacks have been popped to their state when *Rename* entered B_2, but the counters reflect the names created inside B_2. In B_3, *Rename* rewrites the ϕ-function targets, creating new SSA names for each. Next, it rewrites each assignment in the block, using current SSA names for the uses and then creating new SSA names for the definition. (Since `y` and `z` are not global names, it leaves them intact.)

  B_3 has two CFG successors, B_1 and B_4. In B_1, it rewrites the ϕ-function parameters that correspond to the edge from B_3, using the stacks and counters shown in Figure 9.13f. B_4 has no ϕ-functions. Next, *Rename* recurs on B_3's dominator-tree child, B_4. When that call returns, *Rename* pops the stacks and returns.
- *Block* B_4 This block just contains a return statement. It has no ϕ-functions, definitions, uses, or successors in either the CFG or the dominator tree. Thus, *Rename* performs no actions and leaves the stacks and counters unchanged.

```
$B_0$: $i_0 \leftarrow 1$
       $\rightarrow B_1$
$B_1$: $a_1 \leftarrow \phi(a_0, a_3)$
       $b_1 \leftarrow \phi(b_0, b_3)$
       $c_1 \leftarrow \phi(c_0, c_4)$
       $d_1 \leftarrow \phi(d_0, d_3)$
       $i_1 \leftarrow \phi(i_0, i_2)$
       $a_2 \leftarrow \cdots$
       $c_2 \leftarrow \cdots$
       $(a_2 < c_2) \rightarrow B_2, B_5$
$B_2$: $b_2 \leftarrow \cdots$
       $c_3 \leftarrow \cdots$
       $d_2 \leftarrow \cdots$
       $\rightarrow B_3$

$B_3$: $a_3 \leftarrow \phi(a_2, a_4)$
       $b_3 \leftarrow \phi(b_2, b_4)$
       $c_4 \leftarrow \phi(c_3, c_5)$
       $d_3 \leftarrow \phi(d_2, d_6)$
       $y \leftarrow a_3 + b_3$
       $z \leftarrow c_4 + d_3$
       $i_2 \leftarrow i_1 + 1$
       $(i_2 \le 100) \rightarrow B_1, B_4$
$B_4$: return
$B_5$: $a_4 \leftarrow \cdots$
       $d_4 \leftarrow \cdots$
       $(a_4 \le d_4) \rightarrow B_6, B_8$

$B_6$: $d_5 \leftarrow \cdots$
       $\rightarrow B_7$
$B_7$: $c_5 \leftarrow \phi(c_2, c_6)$
       $d_6 \leftarrow \phi(d_5, d_4)$
       $b_4 \leftarrow \cdots$
       $\rightarrow B_3$
$B_8$: $c_6 \leftarrow \cdots$
       $\rightarrow B_7$
```

■ **FIGURE 9.14** Example after Renaming.

- *Block* B_5 After B_4, *Rename* pops through B_3 back to B_1. With the stacks as shown in Figure 9.13g, it recurs down into B_1's final dominator-tree child, B_5. B_5 has no ϕ-functions. *Rename* rewrites the two assignment statements and the expression in the conditional, creating new SSA names as needed. Neither of B_5's CFG successors has ϕ-functions. *Rename* next recurs on B_5's dominator-tree children, B_6, B_7, and B_8. Finally, it pops the stacks and returns.
- *Block* B_6 *Rename* enters B_6 with the state shown in Figure 9.13h. B_6 has no ϕ-functions. *Rename* rewrites the assignment to d, generating the new SSA name d_5. Next, it visits the ϕ-functionsin B_6's CFG successor B_7. It rewrites the ϕ-function arguments that correspond to the path from B_6 with their current names, c_2 and d_5. Since B_6 has no dominator-tree children, it pops the stack for d and returns.
- *Block* B_7 *Rename* enters B_7 with the state shown in Figure 9.13i. It first renames the ϕ-function targets with new SSA names, c_5 and d_6. Next, it rewrites the assignment to b with new SSA name b_4. It then rewrites the ϕ-function arguments in B_7's CFG successor, B_3, with their current names. Since B_7 has no dominator-tree children, it pops the stacks and returns.
- *Block* B_8 *Rename* enters B_8 with the state shown in Figure 9.13j. B_8 has no ϕ-functions. *Rename* rewrites the assignment to c with new SSA name c_6. It examines B_8's CFG successor, B_7 and rewrites the corresponding ϕ-function arguments with their current names, c_6 and d_4. Since B_8 has no dominator-tree children, it pops the stacks and returns.

Figure 9.14 shows the code after *Rename* halts.

A Final Improvement

A clever implementation of *NewName* can reduce the time and the space expended on stack manipulation. The primary use of the stacks is to reset the name space on exit from a block. If a block redefines the same base name several times, *NewName* only needs to keep the most recent name. This happened with a and c in block B_1 of the example. *NewName* may overwrite the same stack slot multiple times within a single block.

This makes the maximum stack sizes predictable; no stack can be larger than the depth of the dominator tree. It lowers the overall space requirements, avoids the need for overflow tests on each push, and decreases the number of push and pop operations. It requires another mechanism for determining which stacks to pop on exit from a block. *NewName* can thread together the stack entries for a block. *Rename* can use the thread to pop the appropriate stacks.

9.3.5 Translation Out of SSA Form

Because modern processors do not implement ϕ-functions, the compiler needs to translate SSA form back into executable code. From the examples, it is tempting to believe that the compiler can just drop the subscripts from the SSA names, revert to base names, and delete the ϕ-functions. If the compiler simply builds SSA form and converts it back into executable code, this approach will work. If, however, the code has been rearranged or values have been renamed, this approach can produce incorrect code.

As an example, we saw in Section 8.4.1 that using SSA names could allow local value numbering (LVN) to discover and eliminate more redundancies.

Before LVN	After LVN
$a \leftarrow x + y$ $b \leftarrow x + y$ $a \leftarrow 17$ $c \leftarrow x + y$	$a \leftarrow x + y$ $b \leftarrow a$ $a \leftarrow 17$ $c \leftarrow x + y$

Original Name Space

Before LVN	After LVN
$a_0 \leftarrow x_0 + y_0$ $b_0 \leftarrow x_0 + y_0$ $a_1 \leftarrow 17$ $c_0 \leftarrow x_0 + y_0$	$a_0 \leftarrow x_0 + y_0$ $b_0 \leftarrow a_0$ $a_1 \leftarrow 17$ $c_0 \leftarrow a_0$

SSA Name Space

The table on the left shows a four-operation block and the results that LVN produces when it uses the code's own name space. The table on the right shows the same example using the SSA name space. Because the SSA name space gives a_0 a distinct name from a_1, LVN can replace the evaluation of $x_0 + y_0$ in the final operation with a reference to a_0.

```
B0: i0 ← 1
    a1 ← a0
    b1 ← b0
    c1 ← c0
    d1 ← d0
    i1 ← i0
    → B1
B1: a2 ← ···
    c2 ← ···
    (a2 < c2) → B2,B5
B2: b2 ← ···
    c3 ← ···
    d2 ← ···
    a3 ← a2
    b3 ← b2
    c4 ← c3
    d3 ← d2
    → B3
B3: y ← a3 + b3
    z ← c4 + d3
    i2 ← i1 + 1
    (i2 ≤ 100) → B9,B4
B4: return
B5: a4 ← ···
    d4 ← ···
    (a4 ≤ d4) → B6,B8
B6: d5 ← ···
    c5 ← c2
    d6 ← d5
    → B7
B7: b4 ← ···
    a3 ← a4
    b3 ← b4
    c4 ← c5
    d3 ← d6
    → B3
B8: c6 ← ···
    c5 ← c6
    d6 ← d4
    → B7
B9: a1 ← a3
    b1 ← b3
    c1 ← c4
    d1 ← d3
    i1 ← i2
    → B1
```

■ **FIGURE 9.15** Example after Copy Insertion to Eliminate ϕ-functions.

Notice, however, that simply dropping the subscripts on variable names produces incorrect code, since c receives the value 17. More aggressive transformations, such as code motion and copy folding, can rewrite the SSA form in ways that introduce more subtle problems.

To avoid such problems, the compiler can keep the SSA name space intact and replace each ϕ-function with a set of copy operations—one along each incoming edge. For a ϕ-function $x_i \leftarrow \phi(x_j, x_k)$, the compiler should insert $x_i \leftarrow x_j$ along the edge carrying the value x_j and $x_i \leftarrow x_k$ along the edge carrying x_k.

Figure 9.15 shows the running example after ϕ-functions have been replaced with copy operations. The four ϕ-functions that were in B_3 have been replaced with a set of four copies in each of B_2 and B_7. Similarly, the two ϕ-functions in B_7 induce a pair of copies in each of B_6 and B_8. In both these cases, the compiler can insert the copies into the predecessor blocks.

The ϕ-functions in B_1 reveal a more complicated situation. The compiler can insert copies directly into its predecessor B_0, but not into its predecessor B_3. Since B_3 has multiple successors, inserting copies for the ϕ-functions from B_1 at the end of B_3 would also cause them to execute along the path from B_3 to B_4, where they are not necessary and might produce incorrect results. To remedy this problem, the compiler can split the edge (B_3, B_1), insert a new block between B_3 and B_1, and place the copies in that new

If the names defined by the copies are not LiveIn in B_4, then the copies would be harmless. The compiler's strategy, however, must work if the names are LiveIn.

block. The new block is labelled B_9 in Figure 9.15. After copy insertion, the example appears to have many superfluous copies. Fortunately, the compiler can remove most, if not all, of these copies with subsequent optimizations, such as copy folding (see Section 13.4.6).

Critical edge

In a CFG, an edge whose source has multiple successors and whose sink has multiple predecessors is called a *critical edge*.

We call an edge such as (B_3, B_1) a *critical edge*. When the compiler inserts a block in the middle of a critical edge, it *splits* the critical edge. Some transformations on SSA form assume that the compiler splits all critical edges before it applies the transformation.

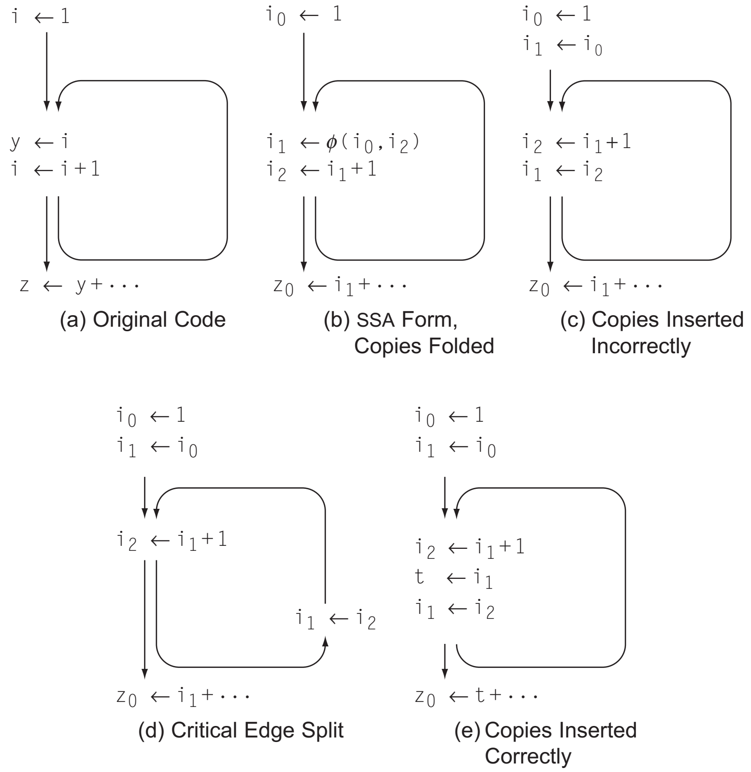
In out-of-SSA translation, the compiler can split critical edges to create locations for the necessary copy operations. This transformation cures most of the problems that arise during out-of-SSA translation. However, two more subtle problems can arise. The first, which we call the lost-copy problem, arises from a combination of aggressive program transformations and unsplit critical edges. The second, which we call the swap problem, arises from an interaction of some aggressive program transformations and the detailed definition of SSA form.

The Lost-Copy Problem

Many SSA-based algorithms require that critical edges be split. Sometimes, however, the compiler cannot, or should not, split critical edges. For example, if the critical edge is the closing branch of a heavily executed loop, adding a block with one or more copy operations and a jump may have an adverse impact on execution speed. Similarly, adding blocks and edges in the late stages of compilation can interfere with regional scheduling, with register allocation, and with optimizations such as code placement.

The lost-copy problem arises from the combination of copy folding and critical edges that cannot be split. Figure 9.16 shows an example. Panel a shows the original code—a simple loop. In panel b, the compiler has converted the loop into SSA form and folded the copy from i to y, replacing the sole use of y with a reference to i_1. Panel c shows the code produced by straightforward copy insertion into the ϕ-function's predecessor blocks. This code assigns the wrong value to z_0. The original code assigns z_0 the second to last value of i; the code in panel c assigns z_0 the last value of i. With the critical edge split, as in panel d, copy insertion produces the correct behavior. However, it adds a jump to every iteration of the loop.

The combination of an unsplit critical edge and copy folding creates the lost copy. Copy folding eliminated the assignment y ← i by folding i_1 into the reference to y in the block that follows the loop. Thus, copy folding extended the lifetime of i_1. Then, the copy-insertion algorithm replaced the



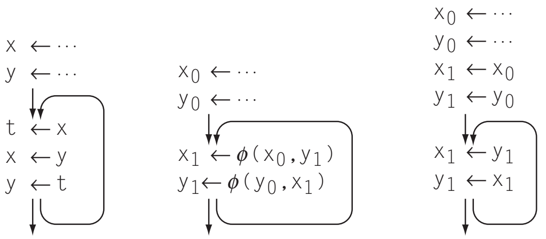
■ **FIGURE 9.16** An Example of the Lost-Copy Problem.

ϕ-function at the top of the loop body with a copy operation in each of that block's predecessors. This inserts the copy $i_1 \leftarrow i_2$ at the bottom of the block—at a point where i_1 is still live.

The compiler can avoid the lost-copy problem by checking the liveness of the target name for each copy that it tries to insert during out-of-SSA translation. When it discovers a copy target that is live, it must preserve the live value in a temporary name and rewrite subsequent uses to refer to the temporary name. This rewriting step can be done with an algorithm modelled on the renaming step of the SSA construction algorithm. Figure 9.16e shows the code that this approach produces.

The Swap Problem

The swap problem arises from the definition of ϕ-function execution. When a block executes, all of its ϕ-functions execute concurrently before any other statement in the block. That is, all the ϕ-functions simultaneously read their



(a) Original Code (b) SSA Form, Copies Folded (c) After Naive Copy Insertion

■ **FIGURE 9.17** An Example of the Swap Problem.

appropriate input parameters and then simultaneously redefine their target values.

Figure 9.17 shows a simple example of the swap problem. Panel a shows the original code, a simple loop that swaps the values of x and y. Panel b shows the code after conversion to SSA form and aggressive copy folding. In this form, with the rules for evaluating ϕ-functions, the code retains its original meaning. When the loop body executes, the ϕ-function parameters are read before any of the ϕ-function targets are defined. On the first iteration, it reads x_0 and y_0 before defining x_1 and y_1. On subsequent iterations, the loop body reads x_1 and y_1 before redefining them. Panel c shows the same code, after the naive copy-insertion algorithm has run. Because copies execute sequentially, rather than concurrently, both x_1 and y_1 receive the same value, an incorrect outcome.

At first glance, it might appear that splitting the back edge—a critical edge—helps. However, splitting the edge simply places the same two copies, in the same order, in another block. The straightforward fix for this problem is to adopt a two-stage copy protocol. The first stage copies each of the ϕ-function arguments to its own temporary name, simulating the behavior of the original ϕ-functions. The second state then copies those values to the ϕ-function targets.

Unfortunately, this solution doubles the number of copy operations required to translate out of SSA form. In the code from Figure 9.17a, it would require four assignments: $s \leftarrow y_1$, $t \leftarrow x_1$, $x_1 \leftarrow s$, and $y_1 \leftarrow t$. All of these assignments execute on each iteration of the loop. To avoid this loss of efficiency, the compiler should attempt to minimize the number of copies that it inserts.

In fact, the swap problem can arise without a cycle of copies; all it takes is a set of ϕ-functions that have, as inputs, variables defined as outputs of other ϕ-functions in the same block. In the acyclic case, in which ϕ-functions reference the results of other ϕ-functions in the same block, the compiler can avoid the problem by carefully ordering the inserted copies.

To solve this problem, in general, the compiler can detect cases in which ϕ-functions reference the targets of other ϕ-functions in the same block. For each cycle of references, it must insert a copy to a temporary that breaks the cycle. Then, it can schedule the copies to respect the dependences implied by the ϕ-functions.

The minimal code for the example would use one extra copy; it is similar to the code in Figure 9.17a.

9.3.6 **Using SSA Form**

A compiler uses SSA form because it improves the quality of analysis, the quality of optimization, or both. To see how analysis over SSA form differs from the classical data-flow analysis techniques presented in Section 9.2, consider performing global constant propagation on SSA form, using an algorithm called sparse simple constant propagation (SSCP).

Semilattice
a set L and a *meet* operator $\wedge$ such that, $\forall$ a, b, and $c \in L$,

1. $a \wedge a = a$,
2. $a \wedge b = b \wedge a$, and
3. $a \wedge (b \wedge c) = (a \wedge b) \wedge c$

Compilers use semilattices to model the data domains of analysis problems.

In the SSCP algorithm, the compiler annotates each SSA name with a value. The set of possible values forms a *semilattice*. A semilattice consists of a set L of values and a meet operator, $\wedge$. The meet operator must be idempotent, commutative, and associative; it imposes an order on the elements of L as follows:

$$\begin{array}{lll} a \geq b & \text{if and only if} & a \wedge b = b, \text{ and} \\ a > b & \text{if and only if} & a \geq b \text{ and } a \neq b \end{array}$$

A semilattice has a bottom element, $\perp$, with the properties that

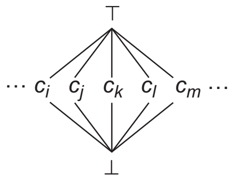
$$\forall\, a \in L, a \wedge \perp \;=\; \perp, \quad \text{and} \quad \forall\, a \in L, a \geq \perp.$$

Some semilattices also have a top element, $\top$, with the properties that

$$\forall\, a \in L, a \wedge \top \;=\; a \quad \text{and} \quad \forall\, a \in L, \top \geq a.$$

In constant propagation, the structure of the semilattice used to model program values plays a critical role in the algorithm's runtime complexity. The semilattice for a single SSA name appears in the margin. It consists of $\top$, $\perp$, and an infinite set of distinct constant values. For any two constants, c_i and c_j, $c_i \wedge c_j = \perp$.

In SSCP, the algorithm initializes the value associated with each SSA name to $\top$, which indicates that the algorithm has no knowledge of the SSA name's value. If the algorithm subsequently discovers that SSA name x has the known



Semilattice for Constant Propagation

```
// Initialization Phase
WorkList $\leftarrow$ $\emptyset$

for each SSA name n
    initialize Value(n) by rules specified in the text

    if Value(n) $\neq$ $\top$ then
        WorkList $\leftarrow$ WorkList $\cup$ {n}

// Propagation Phase - Iterate to a fixed point
while (WorkList $\neq$ $\emptyset$)
   remove some n from WorkList            // Pick an arbitrary name

   for each operation op that uses n
       let m be the SSA name that op defines

       if Value(m) $\neq$ $\bot$ then              // Recompute and test for change
           t $\leftarrow$ Value(m)
           Value(m) $\leftarrow$ result of interpreting op over lattice values

           if Value(m) $\neq$ t
               then WorkList $\leftarrow$ WorkList $\cup$ {m}
```

FIGURE 9.18 Sparse Simple Constant Propagation Algorithm.

constant value c_i, it models that knowledge by assigning `Value(x)` the semilattice element c_i. If it discovers that x has a changing value, it models that fact with the value $\bot$.

The algorithm for SSCP, shown in Figure 9.18, consists of an initialization phase and a propagation phase. The initialization phase iterates over the SSA names. For each SSA name `n`, it examines the operation that defines `n` and sets `Value(n)` according to a simple set of rules. If `n` is defined by a ϕ-function, SSCP sets `Value(n)` to $\top$. If `n`'s value is a known constant c_i, SSCP sets `Value(n)` to c_i. If `n`'s value cannot be known—for example, it is defined by reading a value from external media—SSCP sets `Value(n)` to $\bot$. Finally, if `n`'s value is not known, SSCP sets `Value(n)` to $\top$. If `Value(n)` is not $\top$, the algorithm adds `n` to the worklist.

The propagation phase is straightforward. It removes an SSA name `n` from the worklist. The algorithm examines each operation `op` that uses `n`, where `op` defines some SSA name `m`. If `Value(m)` has already reached $\bot$, then no further evaluation is needed. Otherwise, it models the evaluation of `op` by interpreting the operation over the lattice values of its operands. If the result is lower in the lattice than `Value(m)`, it lowers `Value(m)` accordingly and adds `m` to the worklist. The algorithm halts when the worklist is empty.

Interpreting an operation over lattice values requires some care. For a ϕ-function, the result is simply the meet of the lattice values of all the ϕ-function's arguments; the rules for meet are shown in the margin, in order of precedence. For other kinds of operations, the compiler must apply operator-specific knowledge. If any operand has the lattice value $\top$, the evaluation returns $\top$. If none of the operands has the value $\top$, the model should produce an appropriate value.

$$\begin{array}{ll} \top \wedge x = x & \forall\, x \\ \bot \wedge x = \bot & \forall\, x \\ c_i \wedge c_j = c_i & \text{if } c_i = c_j \\ c_i \wedge c_j = \bot & \text{if } c_i \neq c_j \end{array}$$

Rules for Meet

For each value-producing operation in the IR, SSCP needs a set of rules that model the operands' behavior. Consider the operation $a \times b$. If $a = 4$ and $b = 17$, the model should produce the value 68 for $a \times b$. However, if $a = \bot$, the model should produce $\bot$ for any value of b except 0. Because $a \times 0 = 0$, independent of a's value, $a \times 0$ should produce the value 0.

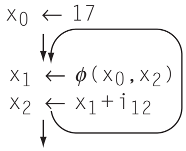
Complexity

The propagation phase of SSCP is a classic fixed-point scheme. The arguments for termination and complexity follow from the length of descending chains through the lattice that it uses to represent values, shown in Figure 9.18. The `Value` associated with any SSA name can have one of three initial values—$\top$, some constant c_i other than $\top$ or $\bot$, or $\bot$. The propagation phase can only lower its value. For a given SSA name, this can happen at most twice—from $\top$ to c_i to $\bot$. SSCP adds an SSA name to the worklist only when its value changes, so each SSA name appears on the worklist at most twice. SSCP evaluates an operation when one of its operands is removed from the worklist. Thus, the total number of evaluations is at most twice the number of uses in the program.

Optimism: The Role of Top

The SSCP algorithm differs from the data-flow problems in Section 9.2 in that it initializes unknown values to the lattice element $\top$. In the lattice for constant values, $\top$ is a special value that represents a lack of knowledge about the SSA name's value. This initialization plays a critical role in constant propagation; it allows values to propagate into cycles in the graph, which are caused by loops in the CFG.

Because it initializes unknown values to $\top$, rather than $\bot$, it can propagate some values into cycles in the graph—loops in the CFG. Algorithms that begin with the value $\top$, rather than $\bot$, are often called *optimistic* algorithms. The intuition behind this term is that initialization to $\top$ allows the algorithm to propagate information into a cyclic region, optimistically assuming that the value along the back edge will confirm this initial propagation. An initialization to $\bot$, called *pessimistic*, disallows that possibility.



```
x0 ← 17
x1 ← φ(x0,x2)
x2 ← x1+i12
```

(a) The Code Fragment

Time	Lattice Values					
Step	Pessimistic			Optimistic		
	x_0	x_1	x_2	x_0	x_1	x_2
0	17	$\perp$	$\perp$	17	$\top$	$\top$
1	17	$\perp$	$\perp$	17	17	$17 + i_{12}$

(b) Results of Pessimistic and Optimistic Analyses

■ **FIGURE 9.19** Optimistic Constant Example.

To see this, consider the SSA fragment in Figure 9.19. If the algorithm pessimistically initializes x_1 and x_2 to $\perp$, it will not propagate the value 17 into the loop. When it evaluates the ϕ-function for x_1, it computes $17 \wedge \perp$ to yield $\perp$. With x_1 set to $\perp$, x_2 also gets set to $\perp$, even if i_{12} has a known value, such as 0.

If, on the other hand, the algorithm optimistically initializes unknown values to $\top$, it can propagate the value of x_0 into the loop. When it computes a value for x_1, it evaluates $17 \wedge \top$ and assigns the result, 17, to x_1. Since x_1's value has changed, the algorithm places x_1 on the worklist. The algorithm then reevaluates the definition of x_2. If, for example, i_{12} has the value 0, then this assigns x_2 the value 17 and adds x_2 to the worklist. When it reevaluates the ϕ-function, it computes $17 \wedge 17$ and proves that x_1 is 17.

Consider what would happen if i_{12} has the value 2, instead. Then, when SSCP evaluates $x_1 + i_{12}$, it assigns x_2 the value 19. Now, x_1 gets the value $17 \wedge 19$, or $\perp$. This, in turn, propagates back to x_2, producing the same final result as the pessimistic algorithm.

The Value of SSA Form

In the SSCP algorithm, SSA form leads to a simple and efficient algorithm. To see this point, consider a classic data-flow approach to constant propagation. It would associate a set ConstantsIn with each block in the code, define an equation to compute ConstantsIn(b_i) as a function of the ConstantsOut sets of b_i's predecessors, and define a procedure for interpreting the code in a block to derive ConstantsOut(b_i) from ConstantsIn(b_i). In contrast, the algorithm in Figure 9.18 is relatively simple. It still has an idiosyncratic mechanism for interpreting operations, but otherwise it is a simple iterative fixed-point algorithm over a particularly shallow lattice.

In SSA form, the propagation step is sparse; it only evaluates expressions of lattice values at operations (and ϕ-functions) that use those values. Equally important, assigning values to individual SSA names makes the optimistic initialization natural rather than contrived and complicated. In short, SSA leads to an efficient, understandable sparse algorithm for global constant propagation.

SECTION REVIEW

SSA form encodes information about both data flow and control flow in a conceptually simple intermediate form. To make use of SSA, the compiler must first transform the code into SSA form. This section focused on the algorithms needed to build *semipruned* SSA *form*. The construction is a two step process. The first step inserts ϕ-functions into the code at join points where distinct definitions can converge. The algorithm relies heavily on dominance frontiers for efficiency. The second step creates the SSA name space by adding subscripts to the original base names during a systematic traversal of the entire procedure.

Because modern machines do not directly implement ϕ-functions, the compiler must translate code out of SSA form before it can execute. Transformation of the code while in SSA form can complicate out-of-SSA translation. Section 9.3.5 examined both the "lost copy problem" and the "swap problem" and described approaches for handling them. Finally, Section 9.3.6 showed an algorithm that performs global constant propagation over the SSA form.

Review Questions

1. Maximal SSA form includes useless ϕ-functions that define nonlive values and redundant ϕ-functions that merge identical values (e.g. $x_8 \leftarrow \phi(x_7, x_7)$). How does the semipruned SSA construction deal with these unneeded ϕ-functions?
2. Assume that your compiler's target machine implements swap r_1, r_2, an operation that simultaneously performs $r_1 \leftarrow r_2$ and $r_2 \leftarrow r_1$. What impact would the swap operation have on out-of-SSA translation?

   swap can be implemented with the three operation sequence:

   $r_1 \leftarrow r_1 + r_2$
   $r_2 \leftarrow r_1 - r_2$
   $r_1 \leftarrow r_1 - r_2$

   What would be the advantages and disadvantages of using this implementation of swap in out-of-SSA translation?

9.4 INTERPROCEDURAL ANALYSIS

The inefficiencies introduced by procedure calls appear in two distinct forms: loss of knowledge in single-procedure analysis and optimization that arises from the presence of a call site in the region being analyzed and transformed and specific overhead introduced to maintain the abstractions inherent in the procedure call. Interprocedural analysis was introduced to address the former problem. We saw, in Section 9.2.4, how the compiler can compute sets that summarize the side effects of a call site. This section explores more complex issues in interprocedural analysis.

9.4.1 Call-Graph Construction

The first problem that the compiler must address in interprocedural analysis is the construction of a call graph. In the simplest case, in which every procedure call invokes a procedure named by a literal constant, as in "`call foo(x, y, z)`", the problem is straightforward. The compiler creates a call-graph node for each procedure in the program and adds an edge to the call graph for each call site. This process takes time proportional to the number of procedures and the number of call sites in the program; in practice, the limiting factor will be the cost of scanning procedures to find the call sites.

Source language features can make call-graph construction much harder. Even FORTRAN and C programs have complications. For example, consider the small C program shown in Figure 9.20a. Its precise call graph is shown in Figure 9.20b. The following subsections outline the language features that complicate call-graph construction.

Procedure-Valued Variables

If the program uses procedure-valued variables, the compiler must analyze the code to estimate the set of potential callees at each call site that invokes a procedure-valued variable. To begin, the compiler can construct the graph specified by the calls that use explicit literal constants. Next, it can track the propagation of functions as values around this subset of the call graph, adding edges as indicated.

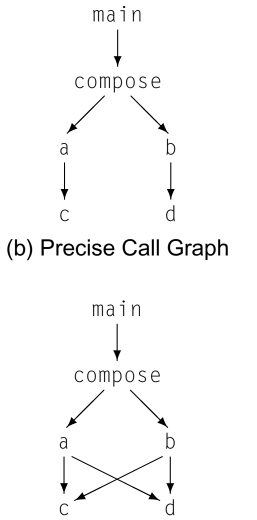
In SSCP, initialize function-valued formals with known constant values. Actuals with the known values reveal where functions are passed through.

The compiler can use a simple analog of global constant propagation to transfer function values from a procedure's entry to the call sites that use them, using set union as its meet operation. To improve its efficiency, it can construct expressions for each parameter-valued variable used in a procedure (see the discussion of jump functions in Section 9.4.2).

As the code in Figure 9.20a shows, a straightforward analysis may overestimate the set of call-graph edges. The code calls `compose` to compute `a(c)` and `b(d)`. A simple analysis, however, will conclude that the formal parameter `g` in `compose` can receive either `c` or `d`, and that, as a result, the program

```
int compose( int f(), int g()) {
  return f(g);
}
int a( int z() ) {
  return z();
}
int b( int z() ) {
  return z();
}
int c( ) {
  return ...;
}
int d( ) {
  return ...;
}
int main(int argc, char *argv[]) {
  return compose(a,c)
       + compose(b,d);
}
```

(a) Example C Program



(b) Precise Call Graph

(c) Approximate Call Graph

■ **FIGURE 9.20** Building a Call Graph with Function-Valued Parameters.

might compose any of a(c), a(d), b(c), or b(d), as shown in Figure 9.20c. To build the precise call graph, it must track sets of parameters that are passed together, along the same path. The algorithm could then consider each set independently to derive the precise graph. Alternatively, it might tag each value with the path that the values travel and use the path information to avoid adding spurious edges such as (a,d) or (b,c).

Contextually-Resolved Names

Some languages allow programmers to use names that are resolved by context. In object-oriented languages with an inheritance hierarchy, the binding of a method name to a specific implementation depends on the class of the receiver and the state of the inheritance hierarchy.

If the inheritance hierarchy and all the procedures are fixed at the time of analysis, then the compiler can use interprocedural analysis of the class structure to narrow the set of methods that can be invoked at any given call site. The call-graph constructor must include an edge from that call site to each procedure or method that might be invoked.

Dynamic linking, used in some operating systems to reduce virtual memory requirements, introduces similar complications. If the compiler cannot determine what code will execute, it cannot construct a complete call graph.

For a language that allows the program to import either executable code or new class definitions at runtime, the compiler must construct a conservative call graph that reflects the complete set of potential callees at each call site. One way to accomplish that goal is to construct a node in the call graph that represents unknown procedures and endow it with worst-case behavior; its MayMod and MayRef sets should be the complete set of visible names.

Analysis that reduces the number of call sites that can name multiple procedures can improve the precision of the call graph by reducing the number of spurious edges—edges for calls that cannot occur at runtime. Of equal or greater importance, any call sites that can be narrowed to a single callee can be implemented with a simple call; those with multiple callees may require runtime lookups for the dispatch of the call (see Section 6.3.3). Runtime lookups to support dynamic dispatch are much more expensive than a direct call.

Other Language Issues

In intraprocedural analysis, we assume that the control-flow graph has a single entry and a single exit; we add an artificial exit node if the procedure has multiple returns. In interprocedural analysis, language features can create the same kinds of problems.

For example, Java has both initializers and finalizers. The Java virtual machine invokes a class initializer after it loads and verifies the class; it invokes an object initializer after it allocates space for the object but before it returns the object's hashcode. Thread start methods, finalizers, and destructors also have the property that they execute without an explicit call in the source program.

The call-graph builder must pay attention to these procedures. Initializers may be connected to sites that create objects; finalizers might be connected to the call-graph's entry node. The specific connections will depend on the language definition and the analysis being performed. MayMod analysis, for example, might ignore them as irrelevant, while interprocedural constant propagation needs information from initialization and start methods.

9.4.2 Interprocedural Constant Propagation

Interprocedural constant propagation tracks known constant values of global variables and parameters as they propagate around the call graph, both through procedure bodies and across call-graph edges. The goal of interprocedural constant propagation is to discover situations where a procedure always receives a known constant value or where a procedure always returns a known constant value. When the analysis discovers such a constant, it can specialize the code for that value.

Conceptually, interprocedural constant propagation consists of three subproblems: discovering an initial set of constants, propagating known constant values around the call graph, and modelling transmission of values through procedures.

Discovering an Initial Set of Constants

The analyzer must identify, at each call site, which actual parameters have known constant values. A wide range of techniques are possible. The simplest method is to recognize literal constant values used as parameters. A more effective and expensive technique might use a full-fledged global constant propagation step (see Section 9.3.6) to identify constant-valued parameters.

Propagating Known Constant Values around the Call Graph

Given an initial set of constants, the analyzer propagates the constant values across call-graph edges and through the procedures from entry to each call site in the procedure. This portion of the analysis resembles the iterative data-flow algorithms from Section 9.2. This problem can be solved with the iterative algorithm, but the algorithm can require significantly more iterations than it would for simpler problems such as live variables or available expressions.

Modeling Transmission of Values through Procedures

Each time it processes a call-graph node, the analyzer must determine how the constant values known at the procedure's entry affect the set of constant values known at each call site. To do so, it builds a small model for each actual parameter, called a *jump function*. A call site s with n parameters has a vector of jump functions, $\mathcal{J}_s = \langle \mathcal{J}_s^a, \mathcal{J}_s^b, \mathcal{J}_s^c, \ldots, \mathcal{J}_s^n \rangle$, where a is the first formal parameter in the callee, b is the second, and so on. Each jump function, $\mathcal{J}_s^x$, relies on the values of some subset of the formal parameters to the procedure p that contains s; we denote that set as *Support*($\mathcal{J}_s^x$).

For the moment, assume that $\mathcal{J}_s^x$ consists of an expression tree whose leaves are all formal parameters of the caller or literal constants. We require that $\mathcal{J}_s^x$ return $\top$ if *Value*(y) is $\top$ for any $y \in$ *Support*($\mathcal{J}_s^x$).

The Algorithm

Figure 9.21 shows a simple algorithm for interprocedural constant propagation across the call graph. It is similar to the SSCP algorithm presented in Section 9.3.6.

The algorithm associates a field *Value*(x) with each formal parameter x of each procedure p. (It assumes unique, or fully qualified, names for each

```
// Phase 1: Initializations
Build all jump functions and Support mappings
Worklist ← Ø

for each procedure p in the program
    for each formal parameter f to p
        Value(f) ← $\top$                      // Optimistic initial value
        Worklist ← Worklist ∪ {f}

for each call site s in the program
    for each formal parameter f that receives a value at s
        Value(f) ← Value(f) ∧ $\mathcal{J}_s^f$          // Initial constants factor in to $\mathcal{J}_s^f$

// Phase 2: Iterate to a fixed point
while (Worklist ≠ Ø)
    pick parameter f from Worklist          // Pick an arbitrary parameter
    let p be the procedure declaring f

    // Update the Value of each parameter that depends on f
    for each call site s in p and parameter x such that f ∈ Support($\mathcal{J}_s^x$)
        t ← Value(x)
        Value(x) ← Value(x) ∧ $\mathcal{J}_s^x$          // Compute new value
        if (Value(x) < t)
            then Worklist ← Worklist ∪ {x}

// Post-process Val sets to produce CONSTANTS
for each procedure p
    CONSTANTS(p) ← Ø
    for each formal parameter f to p
        if (Value(f) = $\top$)
            then Value(f) ← $\bot$

        if (Value(f) ≠ $\bot$)
            then CONSTANTS(p) ← CONSTANTS(p) ∪ {⟨f, Value(f)⟩}
```

■ **FIGURE 9.21** Iterative Interprocedural Constant Propagation Algorithm.

formal parameter.) The initialization phase optimistically sets all the *Value* fields to $\top$. Next, it iterates over each actual parameter *a* at each call site *s* in the program, updates the *Value* field of *a*'s corresponding formal parameter *f* to *Value*(*f*) $\wedge$ $\mathcal{J}_s^f$, and adds *f* to the worklist. This step factors the initial set of constants represented by the jump functions into the *Value* fields and sets the worklist to contain all of the formal parameters.

The second phase repeatedly selects a formal parameter from the worklist and propagates it. To propagate formal parameter *f* of procedure *p*, the analyzer finds each call site *s* in *p* and each formal parameter *x* (which corresponds to an actual parameter of call site s) such that $f \in Support(\mathcal{J}_s^x)$. It evaluates $\mathcal{J}_s^x$ and combines it with $Value(x)$. If that changes $Value(x)$, it adds x to the worklist. The worklist should be implemented with a data structure, such as a sparse set, that only allows one copy of x in the worklist (see Section B.2.3).

The second phase terminates because each *Value* set can take on at most three lattice values: $\top$, some c_i, and $\bot$. A variable x can only enter the worklist when its initial *Value* is computed or when its *Value* changes. Each variable x can appear on the worklist at most three times. Thus, the total number of changes is bounded and the iteration halts. After the second phase halts, a post-processing step constructs the sets of constants known on entry to each procedure.

Jump Function Implementation

Implementations of jump functions range from simple static approximations that do not change during analysis, through small parameterized models, to more complex schemes that perform extensive analysis at each jump-function evaluation. In any of these schemes, several principles hold. If the analyzer determines that parameter x at call site s is a known constant c, then $\mathcal{J}_s^x = c$ and $Support(\mathcal{J}_s^x) = \emptyset$. If $y \in Support(\mathcal{J}_s^x)$ and $Value(y) = \top$, then $\mathcal{J}_s^x = \top$. If the analyzer determines that the value of $\mathcal{J}_s^x$ cannot be determined, then $\mathcal{J}_s^x = \bot$.

For example, *Support*($\mathcal{J}_s^x$) might contain a value read from a file, so $\mathcal{J}_s^x = \bot$.

The analyzer can implement $\mathcal{J}_s^x$ in many ways. A simple implementation might only propagate a constant if x is the SSA name of a formal parameter in the procedure containing s. (Similar functionality can be obtained using REACHES information from Section 9.2.4.) A more complex scheme might build expressions composed of SSA names of formal parameters and literal constants. An effective and expensive technique would be to run the SSCP algorithm on demand to update the values of jump functions.

Extending the Algorithm

The algorithm shown in Figure 9.21 only propagates constant-valued actual parameters forward along call-graph edges. We can extend it, in a straightforward way, to handle both returned values and variables that are global to a procedure.

Just as the algorithm builds jump functions to model the flow of values from caller to callee, it can construct *return jump functions* to model the values returned from callee to caller. Return jump functions are particularly important for routines that initialize values, whether filling in a common block in FORTRAN or setting initial values for an object or class in Java. The algorithm can treat return jump functions in the same way that it handled ordinary jump functions; the one significant complication is that the implementation must avoid creating cycles of return jump functions that diverge (e.g. for a tail-recursive procedure).

To extend the algorithm to cover a larger class of variables, the compiler can simply extend the vector of jump functions in an appropriate way. Expanding the set of variables will increase the cost of analysis, but two factors mitigate the cost. First, in jump-function construction, the analyzer can notice that many of those variables do not have a value that can be modelled easily; it can map those variables onto a universal jump function that returns ⊥ and avoid placing them on the worklist. Second, for the variables that might have constant values, the structure of the lattice ensures that they will be on the worklist at most twice. Thus, the algorithm should still run quickly.

SECTION REVIEW

Compilers perform interprocedural analysis to capture the behavior of all the procedures in the program and to bring that knowledge to bear on optimization within individual procedures. To perform interprocedural analysis, the compiler needs access to all of the code in the program. A typical interprocedural problem requires the compiler to build a call graph (or some analog), to annotate it with information derived directly from the individual procedures, and to propagate that information around the graph.

The results of interprocedural information are applied directly in intraprocedural analysis and optimization. For example, MAYMOD and MAYREF sets can be used to mitigate the impact of a call site on global data-flow analyses, or to avoid the necessity for ϕ-functions after a call site. Information from interprocedural constant propagation can be used to initialize a global algorithm, such as SSCP or SCCP.

Review Questions

1. What features of modern software might complicate interprocedural analysis?
2. How might the analyzer incorporate MAYMOD information into interprocedural constant propagation? What effect would you expect it to have?

9.5 ADVANCED TOPICS

Section 9.2 focused on iterative data-flow analysis. The text emphasizes the iterative approach because it is simple, robust, and efficient. Other approaches to data-flow analysis tend to rely heavily on structural properties of the underlying graph. Section 9.5.1 discusses flow-graph reducibility—a critical property for most of the structural algorithms.

Section 9.5.2 revisits the iterative dominance framework from Section 9.2.1. The simplicity of that framework makes it attractive; however, more specialized and complex algorithms have significantly lower asymptotic complexities. In this section, we introduce a set of data structures that make the simple iterative technique competitive with the fast dominator algorithms for flow graphs of up to several thousand nodes.

9.5.1 Structural Data-Flow Algorithms and Reducibility

In Chapters 8 and 9, we present the iterative algorithm because it works, in general, on any set of well-formed equations on any graph. Other data-flow analysis algorithms exist; many of these work by deriving a simple model of the control-flow structure of the code being analyzed and using that model to solve the equations. Often, that model is built by finding a sequence of transformations to the graph that reduce its complexity—by combining nodes or edges in carefully defined ways. This graph-reduction process lies at the heart of almost every data-flow algorithm *except* the iterative algorithm.

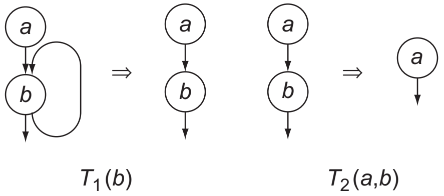
Noniterative data-flow algorithms typically work by applying a series of transformations to a flow graph; each transformation selects a subgraph and replaces it by a single node to represent the subgraph. This creates a series of derived graphs in which each graph differs from its predecessor in the series by the effect of a single transformation step. As the analyzer transforms the graph, it computes data-flow sets for the new representer nodes in each successive derived graph. These sets summarize the replaced subgraph's effects. The transformations reduce well-behaved graphs to a single node. The algorithm then reverses the process, going from the final derived graph, with its single node, back to the original flow graph. As it expands the graph back to its original form, the analyzer computes the final data-flow sets for each node.

In essence, the reduction phase gathers information from the entire graph and consolidates it, while the expansion phase propagates the effects in the consolidated set back out to the nodes of the original graph. Any graph for which such a reduction phase succeeds is deemed *reducible*. If the graph cannot be reduced to a single node, it is *irreducible*.

Reducible graph
A flow graph is *reducible* if the two transformations, T_1 and T_2, will reduce it to a single node. If that process fails, the graph is *irreducible*.

Other tests for reducibility exist. For example, if the iterative DOM framework, using an RPO traversal order, needs more than two iterations over a graph, that graph is irreducible.

Figure 9.22 shows a pair of transformations that can be used to test reducibility and to build a structural data-flow algorithm. T_1 removes a self loop, an edge that runs from a node back to itself. The figure shows T_1 applied



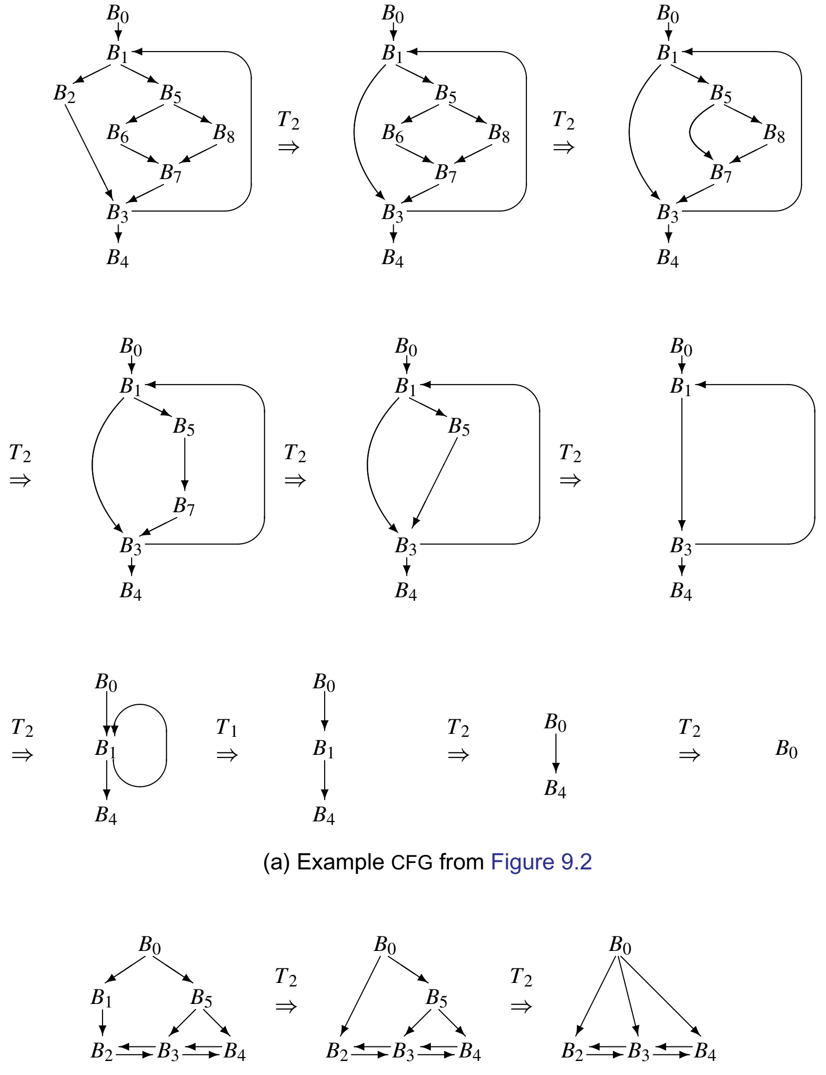
■ **FIGURE 9.22** Transformations T_1 and T_2.

to b, denoted $T_1(b)$. T_2 folds a node b that has exactly one predecessor a back into a; it removes the edge $\langle a, b\rangle$, and makes a the source of any edges that originally left b. If this leaves multiple edges from a to some node n, it consolidates those edges. Figure 9.22 shows T_2 applied to a and b, denoted $T_2(a, b)$. Any graph that can be reduced to a single node by repeated application of T_1 and T_2 is deemed reducible. To understand how this works, consider the CFG from our continuing example. Figure 9.23a shows one sequence of applications of T_1 and T_2 that reduces it to a single-node graph. It applies T_2 until no more opportunities exist: $T_2(B_1, B_2)$, $T_2(B_5, B_6)$, $T_2(B_5, B_8)$, $T_2(B_5, B_7)$, $T_2(B_1, B_5)$, and $T_2(B_1, B_3)$. Next, it uses $T_1(B_1)$ to remove the loop, followed by $T_2(B_0, B_1)$ and $T_2(B_0, B_4)$ to complete the reduction. Since the final graph is a single node, the original graph is reducible.

Other application orders also reduce the graph. For example, if we start with $T_2(B_1, B_5)$, it leads to a different series of transformations. T_1 and T_2 have the finite Church-Rosser property, which ensures that the final result is independent of the order of application and that the sequence terminates. Thus, the analyzer can apply T_1 and T_2 opportunistically—finding places in the graph where one of them applies and using it.

Figure 9.23b shows what can happen when we apply T_1 and T_2 to a graph with multiple-entry loops. The analyzer uses $T_2(B_0, B_1)$ followed by $T_2(B_0, B_5)$. At that point, however, no remaining node or pair of nodes is a candidate for either T_1 or T_2. Thus, the analyzer cannot reduce the graph any further. (No other order will work either.) The graph is not reducible to a single node; it is irreducible.

The failure of T_1 and T_2 to reduce this graph arises from a fundamental property of the graph. The graph is irreducible because it contains a loop, or cycle, that has edges that enter it at different nodes. In terms of the source language, the program that generated the graph has a loop with multiple entries. We can see this in the graph; consider the cycle formed by B_2 and B_3. It has edges entering it from B_1, B_4, and B_5. Similarly, the cycle formed by B_3 and B_4 has edges that enter it from B_2 and B_5.

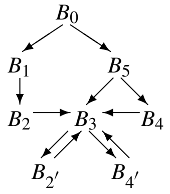


(a) Example CFG from Figure 9.2

(b) An Irreducible Graph

■ **FIGURE 9.23** Reduction Sequences for Example Graphs.

Irreducibility poses a serious problem for algorithms built on transformations like T_1 and T_2. If the reduction sequence cannot complete, producing a single-node graph, then the method must either report failure, modify the graph by splitting one or more nodes, or use an iterative approach to solve the system on the reduced graph. In general, the methods based on structurally reducing the flow graph are limited to reducible graphs. The iterative algorithm, in contrast, works correctly on an irreducible graph.



Irreducible Graph
After Node Splitting

To transform an irreducible graph to a reducible graph, the analyzer can split one or more nodes. The simplest split for the example graph, shown in the margin, clones B_2 and B_4 to create $B_{2'}$ and $B_{4'}$, respectively. The analyzer then retargets the edges (B_3, B_2) and (B_3, B_4) to form a complex loop, $\{B_3, B_{2'}, B_{4'}\}$. The new loop has a single entry, through B_3.

This transformation creates a reducible graph that executes the same sequence of operations as the original graph. Paths that, in the original graph, entered B_3 from either B_2 or B_4 now execute as prologues to the loop $\{B_3, B_{2'}, B_{4'}\}$. Both B_2 and B_4 have unique predecessors in the new graph. B_3 has multiple predecessors, but it is the sole entry to the loop and the loop is reducible. Thus, node splitting produced a reducible graph, at the cost of cloning two nodes.

Both folklore and published studies suggest that irreducible graphs rarely arise in global data-flow analysis. The rise of structured programming in the 1970s made programmers much less likely to use arbitrary transfers of control, like a `goto` statement. Structured loop constructs, such as `do`, `for`, `while`, and `until` loops, cannot produce irreducible graphs. However, transferring control out of a loop (for example, C's `break` statement) creates a CFG that is irreducible to a backward analysis. (Since the loop has multiple exits, the reverse CFG has multiple entries.) Similarly, irreducible graphs may arise more often in interprocedural analysis due to mutually recursive subroutines. For example, the call graph of a hand-coded, recursive-descent parser is likely to have irreducible subgraphs. Fortunately, an iterative analyzer can handle irreducible graphs correctly and efficiently.

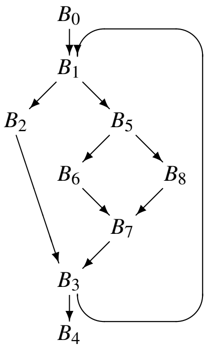
9.5.2 Speeding up the Iterative Dominance Framework

The iterative framework for computing dominance is particularly simple. Where most data-flow problems have equations involving several sets, the equations for Dom involve computing a pairwise intersection over Dom sets and adding a single element to those sets. The simple nature of these equations presents an opportunity to use a particularly simple data-structure to improve the speed of the Dom calculation.

The iterative Dom framework uses a discrete Dom set at each node. We can reduce the amount of space required by the Dom sets by observing that the same information can be represented with a single fact at each node, its immediate dominator, or IDom. From the IDoms for the nodes, the compiler can compute all the other dominance information that it needs.

Recall our example CFG from Section 9.2.1, repeated in the margin with its dominator tree. Its IDom sets are as follows:

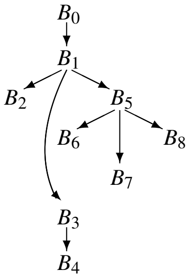
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
IDOM(*n*)	**?**	0	1	1	3	1	5	5	5



The Example CFG

Notice that the dominator tree and the IDoms are isomorphic. IDom(b) is just b's predecessor in the dominator tree. The root of the dominator tree has no predecessor; accordingly, its IDom set is undefined.

The compiler can read a graph's Dom sets from its dominator tree. For a node n, its Dom set can be read as the set of nodes that lie on the path from n to the root of the dominator tree, inclusive of the end points. In the example, the dominator-tree path from B_7 to B_1 consists of (B_7, B_5, B_1, B_0), which matches the set computed for Dom(B_7) in Section 9.2.1.



Its Dominator Tree

Thus, we can use the IDom sets as a proxy for the Dom sets, provided we can provide efficient methods to initialize the sets and to intersect them. To handle the initializations, we will reformulate the iterative algorithm slightly. To intersect two Dom sets from their IDom sets, we will use the algorithm shown in procedure `Intersect` at the bottom of Figure 9.24. It relies on two critical facts.

1. When the algorithm walks the path from a node to the root to recreate a Dom set, it encounters the nodes in a consistent order. The intersection of two Dom sets is simply the common suffix of the labels on the paths from the nodes to the root.
2. The algorithm must be able to recognize the common suffix. It starts at the two nodes whose sets are being intersected, `i` and `j`, and walks upward from each toward the root. If we name the nodes by their RPO numbers, then a simple comparison will let the algorithm discover the nearest common ancestor—the IDom of `i` and `j`.

The `Intersect` algorithm in Figure 9.24 is a variant of the classic "two finger" algorithm. It uses two pointers to trace paths upward through the tree. When they agree, they both point to the node representing the result of the intersection.

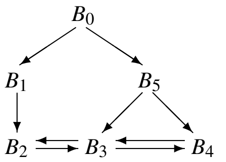
```
for all nodes, b    // initialize the dominators array
    IDoms[b] ← Undefined
IDoms[$b_0$] ← $b_0$
Changed ← true
while (Changed)
   Changed ← false
   for all nodes, b, in reverse postorder (except root)
       NewIDom ← first (processed) predecessor of b   // pick one
       for all other predecessors, p, of b
           if IDoms[p] ≠ Undefined   // i.e., Doms[p] already calculated
               then NewIdom ← Intersect(p, NewIdom)
       if IDoms[b] ≠ NewIdom then
           IDoms[b] ← NewIdom
           Changed ← true

Intersect( i, j )
    finger1 ← i
    finger2 ← j
    while (finger1 ≠ finger2)
       while (RPO(finger1) > RPO(finger2))
          finger1 = IDoms[finger1]
       while (RPO(finger2) > RPO(finger1))
          finger2 = IDoms[finger2]
   return finger1
```

■ **FIGURE 9.24** The Modified Iterative Dominator Algorithm.

The algorithm assigns IDom(b_0) the value b_0 to simplify the rest of the algorithm.

The top of Figure 9.24 shows a reformulated iterative algorithm that avoids the issue of initializing the IDom sets and uses the `Intersect` algorithm. It keeps the IDom information in an array, *IDoms*. It initializes the IDom entry for the root, b_0, to itself. It then processes the nodes in reverse postorder. In computing intersections, it ignores predecessors whose IDoms have not yet been computed.

To see how the algorithm operates, consider the graph in Figure 9.25a. Figure 9.25b shows an RPO for this graph that illustrates the problems caused by irreducibility. Using this order, the algorithm miscomputes the IDoms of B_3, and B_4 in the first iteration. It takes two iterations for the algorithm to correct those IDoms, and a final iteration to recognize that the IDoms have stopped changing.



(a) An Irreducible Graph

	B_0	B_1	B_2	B_3	B_4	B_5
RPO(n)	0	1	5	4	3	2

(b) A Worst-Case RPO

	IDOM(n)					
	B_0	B_1	B_2	B_3	B_4	B_5
—	0	?	?	?	?	?
1	0	0	0	5	5	0
2	0	0	0	0	5	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0

(c) Progress of the IDOM Computation

■ **FIGURE 9.25** A Graph with a More Complex Shape.

This improved algorithm runs quickly. It has a small memory footprint. On any reducible graph, it halts in two passes: the first pass computes the correct IDOM sets and the second pass confirms that no changes occur. An irreducible graph will take more than two passes. In fact, the algorithm provides a rapid test for reducibility—if any IDOM entry changes in the second pass, the graph is irreducible.

9.6 SUMMARY AND PERSPECTIVE

Most optimization tailors general-case code to the specific context that occurs in the compiled code. The compiler's ability to tailor code is often limited by its lack of knowledge about the program's range of runtime behaviors.

Data-flow analysis allows the compiler to model the runtime behavior of a program at compile time and to draw important, specific knowledge out of the models. Many data-flow problems have been proposed; this chapter presented several of them. Many of those problems have properties that lead to efficient analyses. In particular, problems that can be expressed in iterative frameworks have efficient solutions using simple iterative solvers.

SSA form is an intermediate form that encodes both data-flow information and control-dependence information into the name space of the program. Working with SSA form often simplifies both analysis and transformation. Many modern transformations rely on the SSA form of the code.

CHAPTER NOTES

Credit for the first data-flow analysis is usually given to Vyssotsky at Bell Labs in the early 1960s [338]. Earlier work, in the original FORTRAN compiler, included the construction of a control-flow graph and a Markov-style analysis over the CFG to estimate execution frequencies [26]. This analyzer, built by Lois Haibt, might be considered a data-flow analyzer.

Iterative data-flow analysis has a long history in the literature. Among the seminal papers on this topic are Kildall's 1973 paper [223], work by Hecht and Ullman [186], and two papers by Kam and Ullman [210, 211]. The treatment in this chapter follows Kam's work.

This chapter focuses on iterative data-flow analysis. Many other algorithms for solving data-flow problems have been proposed [218]. The interested reader should explore the structural techniques, including interval analysis [17, 18, 62]; T_1-T_2 analysis [336, 185]; the Graham-Wegman algorithm [168, 169]; balanced-tree, path-compression algorithm [330, 331]; graph grammars [219]; and the partitioned-variable technique [359].

Dominance has a long history in the literature. Prosser introduced dominance in 1959 but gave no algorithm to compute dominators [290]. Lowry and Medlock describe the algorithm used in their compiler [252]; it takes at least $O(N^2)$ time, where N is the number of statements in the procedure. Several authors developed faster algorithms based on removing nodes from the CFG [8, 3, 291]. Tarjan proposed an $O(N \log N + E)$ algorithm based on depth-first search and union find [329]. Lengauer and Tarjan improved this time bound [244], as did others [180, 23, 61]. The data-flow formulation for dominators is taken from Allen [12, 17]. The fast data structures for iterative dominance are due to Harvey [100]. The algorithm in Figure 9.8 is from Ferrante, Ottenstein, and Warren [145].

The SSA construction is based on the seminal work by Cytron et al. [110]. It, in turn, builds on work by Shapiro and Saint [313]; by Reif [295, 332]; and by Ferrante, Ottenstein, and Warren [145]. The algorithm in Section 9.3.3 builds semipruned SSA form [49]. The details of the renaming algorithm and the algorithm for reconstructing executable code are described by Briggs et al. [50]. The complications introduced by critical edges have long been recognized in the literature of optimization [304, 133, 128, 130, 225]; it should not be surprising that they also arise in the translation from SSA back into executable code. The sparse simple constant algorithm, SSCP, is due to Reif and Lewis [296]. Wegman and Zadeck reformulate SSCP to use SSA form [346, 347].

The IBM PL/I optimizing compiler was one of the earliest systems to perform interprocedural data-flow analysis [322]. A large body of literature has emerged on side-effect analysis [34, 32, 102, 103]. The interprocedural constant propagation algorithm is from Torczon's thesis and subsequent papers [68, 172, 263]; both Cytron and Wegman suggested other approaches to the problem [111, 347]. Burke and Torczon [64] formulated an analysis that determines which modules in a large program must be recompiled in response to a change in a program's interprocedural information. Pointer analysis is inherently interprocedural; a growing body of literature describes that problem [348, 197, 77, 238, 80, 123, 138, 351, 312, 190, 113, 191]. Ayers, Gottlieb, and Schooler described a practical system that analyzed and optimized a subset of the entire program [25].

■ EXERCISES

Section 9.2

1. The algorithm for live analysis in Figure 9.2 initializes the LiveOut set of each block to ϕ. Are other initializations possible? Do they change the result of the analysis? Justify your answer.

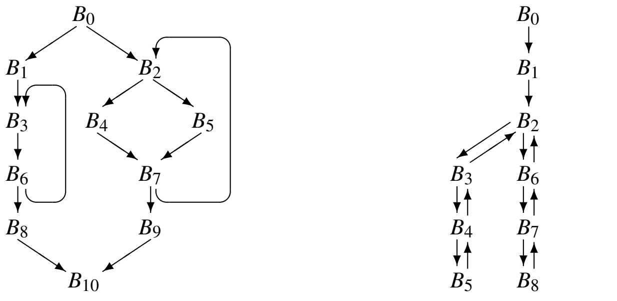
2. In live-variable analysis, how should the compiler treat a block containing a procedure call? What should the block's UEVar set contain? What should its VarKill set contain?

3. In the computation of available expressions, the initialization sets

$$\text{AvailIn}(n_0) = \emptyset, \quad \text{and}$$
$$\text{AvailIn}(n) = \{\textit{all expressions}\}, \forall n \neq n_0$$

Construct a small example program that shows why the latter initialization is necessary. What happens on your example if the AvailIn sets are uniformly initialized to Ø?

4. For each of the following control-flow graphs:



(a) Multiple Loops (b) Doubled Loop Body

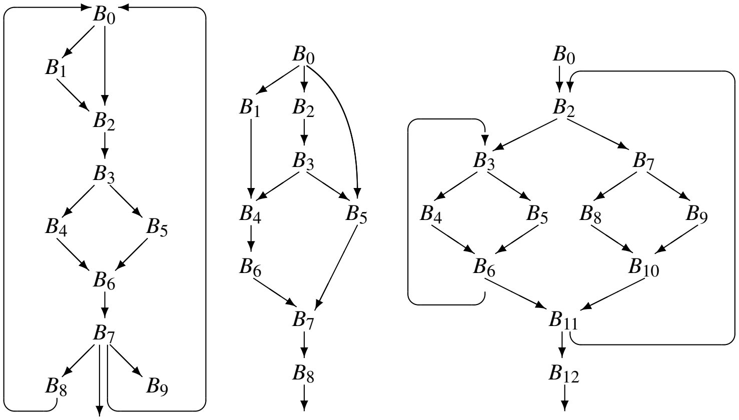
a. Compute reverse postorder numberings for the CFG and the reverse CFG.

b. Compute reverse preorder on the CFG.

c. Is reverse preorder on the CFG equivalent to postorder on the reverse CFG?

Section 9.3

5. Consider the three control-flow graphs shown below.



(a) (b) (c)

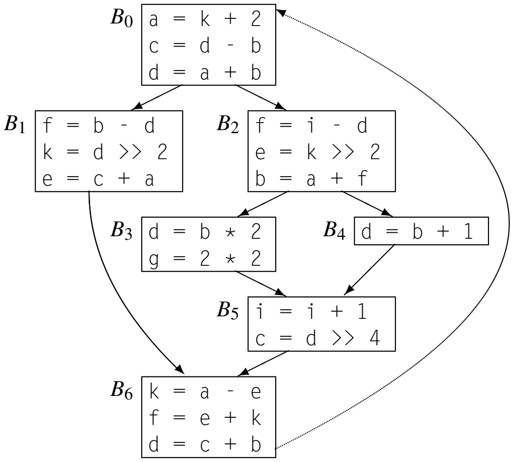
a. Compute the dominator trees for CFGS a, b, and c.

b. Compute the dominance frontiers for nodes 3 and 5 of CFG a, nodes 4 and 5 of CFG b, and nodes 3 and 11 of CFG c.

6. Translate the code shown in Figure 9.26 to SSA form. Show only the final code, after both ϕ-insertion and renaming.

7. Consider the set of all blocks that receive a ϕ-function because of an assignment x ← . . . in some block b. The algorithm in Figure 9.9 inserts a ϕ-function in each block in $\text{DF}(b)$. Each of those blocks is added to the worklist; they, in turn, can add nodes in their DF sets to the worklist. The algorithm uses a checklist to avoid adding a block to the worklist more than once. Call the set of all these blocks $\text{DF}^{+}(b)$. We can define $\text{DF}^{+}(b)$ as the limit of the sequence

$$
\begin{aligned}
&\text{DF}_1(b) = \text{DF}(b)\\
&\text{DF}_2(b) = \text{DF}_1(b) \cup_{x \in \text{DF}_1(b)} \text{DF}_1(x)\\
&\text{DF}_3(b) = \text{DF}_2(b) \cup_{x \in \text{DF}_2(b)} \text{DF}_2(x)\\
&\quad \cdots\\
&\text{DF}_i(b) = \text{DF}_{i-1}(b) \cup_{x \in \text{DF}_{i-1}(b)} \text{DF}_{i-1}(x)
\end{aligned}
$$



```
B0: a = k + 2
    c = d - b
    d = a + b

B1: f = b - d
    k = d >> 2
    e = c + a

B2: f = i - d
    e = k >> 2
    b = a + f

B3: d = b * 2
    g = 2 * 2

B4: d = b + 1

B5: i = i + 1
    c = d >> 4

B6: k = a - e
    f = e + k
    d = c + b
```

■ **FIGURE 9.26** CFG for Problem 6.

Using these extended sets, $DF^+(b)$, leads to a simpler algorithm for inserting ϕ-functions.

a. Develop an algorithm for computing $DF^+(b)$.

b. Develop an algorithm for inserting ϕ-functions using these DF^+ sets.

c. Compare the overall cost of your algorithm, including the computation of DF^+ sets, to the cost of the ϕ-insertion algorithm given in Section 9.3.3.

8. The maximal SSA construction is both simple and intuitive. However, it can insert many more ϕ-functions than the semipruned algorithm. In particular, it can insert both redundant ϕ-functions ($\mathtt{x_i} \leftarrow \phi(\mathtt{x_j},\mathtt{x_j})$) and dead ϕ-functions—where the result is never used.

a. Propose a method for detecting and removing the extra ϕ-functions that the maximal construction inserts.

b. Can your method reduce the set of ϕ-functions to just those that the semipruned construction inserts?

c. Contrast the asymptotic complexity of your method against that of the semipruned construction.

9. Dominance information and SSA form allow us to improve the superlocal value numbering algorithm (SVN) from Section 8.5.1. Assume the code is in SSA form.

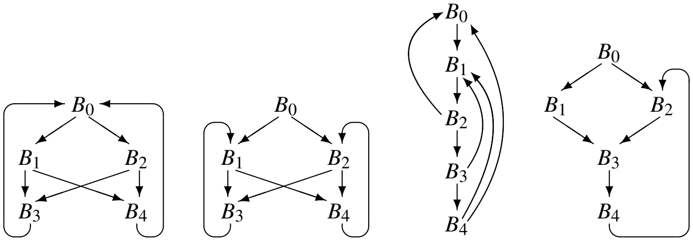
a. For each node in the CFG with multiple predecessors, SVN begins with an empty hash table. For such a block, b_i, can you use dominance information to select a block whose facts must hold on entry to b_i?

b. On what properties of SSA form does this algorithm rely?

c. Assuming that the code is already in SSA form, with dominance information available, what is the extra cost of this dominator-based value numbering?

Section 9.4

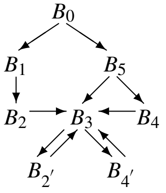
10. For each of the following control-flow graphs, show whether or not it is reducible:



(a) (b) (c) (d)

11. Prove that the following definition of a reducible graph is equivalent to the definition that uses the transformations T_1 and T_2: “A graph G is reducible if and only if for each cycle in G, there exists a node n in the cycle with the property that n dominates every node in that cycle.”

12. Show a sequence of reductions, using T_1 and T_2, that reduce the following graph:


B_0
B_1
B_5
B_2
B_3
B_4
$B_{2'}$
$B_{4'}$