19

Static Single-Assignment Form

dom-i-nate: to exert the supreme determining or guiding influence on

Webster's Dictionary

Many dataflow analyses need to find the use-sites of each defined variable or the definition-sites of each variable used in an expression. The *def-use chain* is a data structure that makes this efficient: for each statement in the flow graph, the compiler can keep a list of pointers to all the *use* sites of variables defined there, and a list of pointers to all *definition* sites of the variables used there. In this way the compiler can hop quickly from use to definition to use to definition.

An improvement on the idea of def-use chains is *static single-assignment form*, or *SSA form*, an intermediate representation in which each variable has only one definition in the program text. The one (static) definition-site may be in a loop that is executed many (dynamic) times, thus the name *static* single-assignment form instead of single-assignment form (in which variables are never redefined at all).

The SSA form is useful for several reasons:

1. Dataflow analysis and optimization algorithms can be made simpler when each variable has only one definition.
2. If a variable has N uses and M definitions (which occupy about $N + M$ instructions in a program), it takes space (and time) proportional to $N \cdot M$ to represent def-use chains – a quadratic blowup (see Exercise 19.8). For almost all realistic programs, the size of the SSA form is linear in the size of the original program (but see Exercise 19.9).
3. Uses and defs of variables in SSA form relate in a useful way to the dominator structure of the control-flow graph, which simplifies algorithms such as interference-graph construction.

$a \leftarrow x + y$	$a_1 \leftarrow x + y$
$b \leftarrow a - 1$	$b_1 \leftarrow a_1 - 1$
$a \leftarrow y + b$	$a_2 \leftarrow y + b_1$
$b \leftarrow x \cdot 4$	$b_2 \leftarrow x \cdot 4$
$a \leftarrow a + b$	$a_3 \leftarrow a_2 + b_2$
(a)	(b)

FIGURE 19.1. (a) A straight-line program. (b) The program in single-assignment form.

4. Unrelated uses of the same variable in the source program become different variables in SSA form, eliminating needless relationships. An example is the program,

   for $i \leftarrow 1$ **to** N **do** $A[i] \leftarrow 0$
   for $i \leftarrow 1$ **to** M **do** $s \leftarrow s + B[i]$

   where there is no reason that both loops need to use the same machine register or intermediate-code temporary variable to hold their respective loop counters, even though both are named i.

In straight-line code, such as within a basic block, it is easy to see that each instruction can define a fresh new variable instead of redefining an old one, as shown in Figure 19.1. Each new definition of a variable (such as a) is modified to define a fresh new variable ($a_1, a_2, \ldots$), and each use of the variable is modified to use the most recently defined version. This is a form of *value numbering* (see page 392).

But when two control-flow paths merge together, it is not obvious how to have only one assignment for each variable. In Figure 19.2a, if we were to define a new version of a in block 1 and in block 2, which version should be used in block 4? Where a statement has more than one predecessor, there is no notion of "most recent."

To solve this problem we introduce a notational fiction, called a ϕ-function. Figure 19.2b shows that we can combine a_1 (defined in block 1) and a_2 (defined in block 3) using the function $a_3 \leftarrow \phi(a_1, a_2)$. But unlike ordinary mathematical functions, $\phi(a_1, a_2)$ yields a_1 if control reaches block 4 along the edge $2 \rightarrow 4$, and yields a_2 if control comes in on edge $3 \rightarrow 4$.

How does the ϕ-function know which edge was taken? That question has two answers:

- If we must execute the program, or translate it to executable form, we can

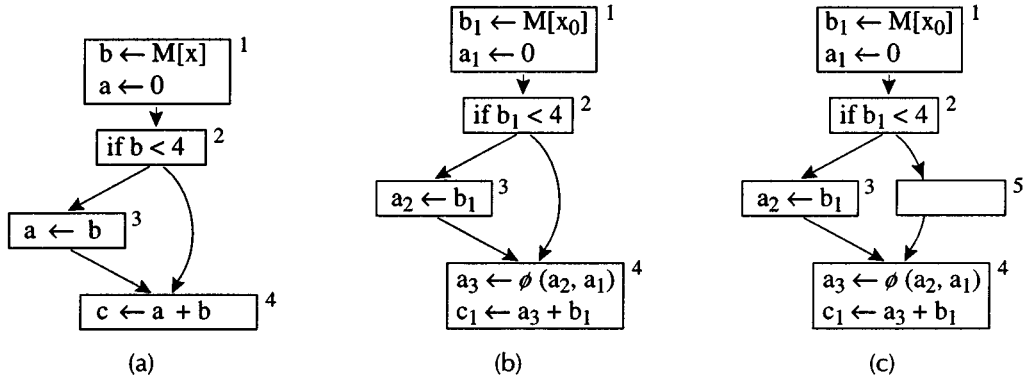


FIGURE 19.2. (a) A program with a control-flow join; (b) the program transformed to single-assignment form; (c) edge-split SSA form.

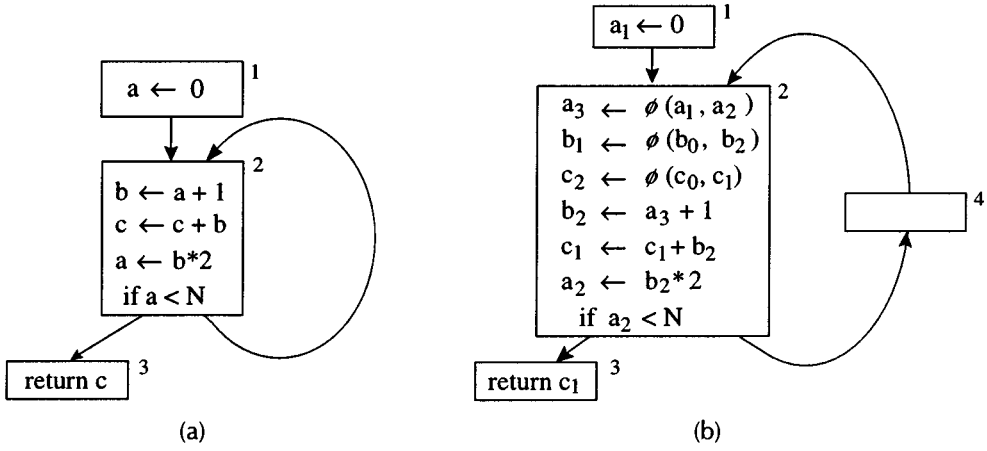


FIGURE 19.3. (a) A program with a loop; (b) the program transformed to edge-split single-assignment form. a_0, b_0, c_0 are initial values of the variables before block 1.

"implement" the ϕ-function using a MOVE instruction on each incoming edge, as shown in Section 19.6.

- In many cases, we simply need the connection of uses to definitions, and don't need to "execute" the ϕ-functions during optimization. In these cases, we can ignore the question of which value to produce.

Consider Figure 19.3a, which contains a loop. We can convert this to static single-assignment form as shown in Figure 19.3b. Note that variables a and c each need a ϕ-function to merge their values that arrive on edges $1 \rightarrow 2$ and

$2 \rightarrow 2$. The ϕ-function for b_1 can later be deleted by dead-code elimination, since b_1 is a dead variable. The variable c is live on entry (after conversion to SSA, the implicit definition c_0 is live); this might be an uninitialized variable, or perhaps c is a formal parameter of the function whose body this is.

The assignment $c_1 \leftarrow c_2 + b_2$ will be executed many times; thus the variable c_1 is updated many times. This illustrates that we do not have a program with *dynamic* single-assignment (like a pure functional program), but a program in which each variable has only one *static* site of definition.

19.1 CONVERTING TO SSA FORM

The algorithm for converting a program to SSA form first adds ϕ functions for the variables, then renames all the definitions and uses of variables using subscripts. The sequence of steps is illustrated in Figure 19.4.

CRITERIA FOR INSERTING ϕ-FUNCTIONS

We could add a ϕ-function for every variable at each *join* point (that is, each node in the control-flow graph with more than one predecessor). But this is wasteful and unnecessary. For example, block 4 in Figure 19.2b is reached by the same definition of b along each incoming edge, so it does not need a ϕ-function for b. The following criterion characterizes the nodes where a variable's data-flow paths merge:

Path-convergence criterion. There should be a ϕ-function for variable a at node z of the flow graph exactly when *all* of the following are true:

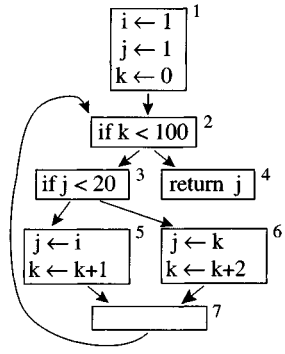
1. There is a block x containing a definition of a,
2. There is a block y (with $y \neq x$) containing a definition of a,
3. There is a nonempty path P_{xz} of edges from x to z,
4. There is a nonempty path P_{yz} of edges from y to z,
5. Paths P_{xz} and P_{yz} do not have any node in common other than z, *and*
6. The node z does not appear within *both* P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.

We consider the start node to contain an implicit definition of *every* variable, either because the variable may be a formal parameter or to represent the notion of $a \leftarrow$ *uninitialized* without special cases.

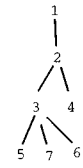
Note, however, that a ϕ-function itself counts as a definition of a, so the *path-convergence criterion* must be considered as a set of equations to be

```
i ← 1
j ← 1
k ← 0
while k < 100
    if j < 20
        j ← i
        k ← k + 1
    else
        j ← k
        k ← k + 2
return j
```

(a) Program



(b) CFG



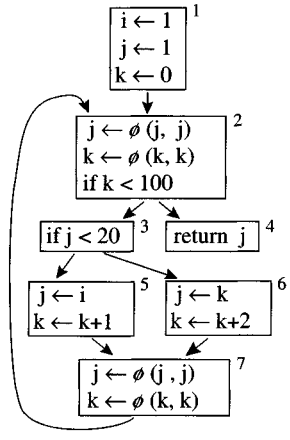
(c) Dominator tree

n	$DF(n)$
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

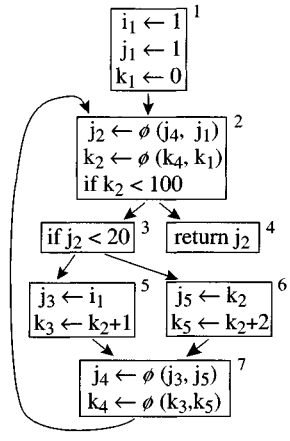
(d) Dominance frontiers

Variable j defined in node 1, but $DF(1)$ is empty. Variable j defined in node 5, $DF(5)$ contains 7, so node 7 needs $\phi(j, j)$. Now j is defined in 7 (by a ϕ-function), $DF(7)$ contains 2, so node 2 needs $\phi(j, j)$. $DF(6)$ contains 7, so node 7 needs $\phi(j, j)$ (but already has it). $DF(2)$ contains 2, so node 2 needs $\phi(j, j)$ (but already has it). Similar calculation for k. Variable i defined in node 1, $DF(1)$ is empty, so no ϕ-functions necessary for i.

(e) Insertion criteria for ϕ-functions



(f) ϕ-functions inserted



(g) Variables renamed

FIGURE 19.4. Conversion of a program to static single-assignment form. Node 7 is a *postbody* node, inserted to make sure there is only one loop edge (see Exercise 18.6); such nodes are not strictly necessary but are sometimes helpful.

satisfied. As usual, we can solve them by iteration.

Iterated path-convergence criterion:

while there are nodes x, y, z satisfying conditions 1–5
and z does not contain a ϕ-function for a
do insert $a \leftarrow \phi(a, a, \ldots, a)$ at node Z

where the ϕ-function has as many a arguments as there are predecessors of node z.

Dominance property of SSA form. An essential property of static single-assignment form is that definitions dominate uses; more specifically,

1. If x is the ith argument of a ϕ-function in block n, then the definition of x dominates the ith predecessor of n.
2. If x is used in a non-ϕ statement in block n, then the definition of x dominates n.

Section 18.1 defines the dominance relation: d dominates n if every path from the start node to n goes through d.

THE DOMINANCE FRONTIER

The iterated path-convergence algorithm for placing ϕ-functions is not practical, since it would be very costly to examine every triple of nodes x, y, z and every path leading from x and y. A much more efficient algorithm uses the dominator tree of the flow graph.

Definitions. x *strictly dominates* w if x dominates w and $x \neq w$. In this chapter I use *successor* and *predecessor* to refer to *graph* edges, and *parent* and *child* to refer to *tree* edges. Node x is an *ancestor* of y if there is a path $x \rightarrow y$ of tree edges, and is a *proper ancestor* if that path is nonempty.

The *dominance frontier* of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.

Figure 19.5a illustrates the dominance frontier of a node; in essence, it is the "border" between dominated and undominated nodes.

Dominance frontier criterion. Whenever node x contains a definition of some variable a, then any node z in the dominance frontier of x needs a ϕ-function for a.

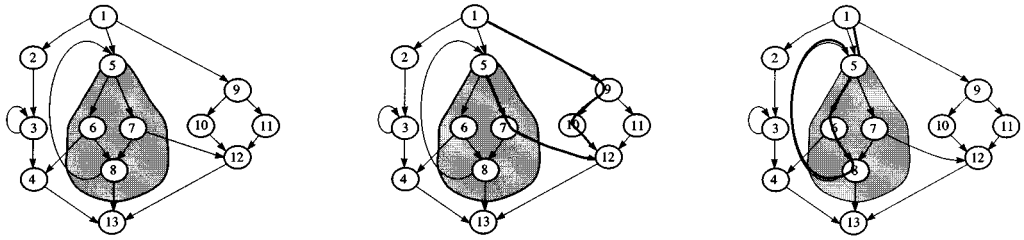


FIGURE 19.5. Node 5 dominates all the nodes in the grey area. (a) Dominance frontier of node 5 includes the nodes (4, 5, 12, 13) that are targets of edges crossing from the region dominated by 5 (grey area including node 5) to the region not strictly dominated by 5 (white area including node 5). (b) Any node in the dominance frontier of n is also a point of convergence of nonintersecting paths, one from n and one from the root node. (c) Another example of converging paths $P_{1,5}$ and $P_{5,5}$.

Iterated dominance frontier. Since a ϕ-function itself is a kind of definition, we must iterate the dominance-frontier criterion until there are no nodes that need ϕ-functions.

Theorem. The *iterated dominance frontier criterion* and the *iterated path-convergence criterion* specify exactly the same set of nodes at which to put ϕ-functions.

The end-of-chapter bibliographic notes refer to a proof of this theorem. I will sketch one half of the proof, showing that if w is in the dominance frontier of a definition, then it must be a point of convergence. Suppose there is a definition of variable a at some node n (such as node 5 in Figure 19.5b), and node w (such as node 12 in Figure 19.5b) is in the dominance frontier of n. The root node implicitly contains a definition of every variable, including a. There is a path P_{rw} from the root node (node 1 in Figure 19.5) to w that does not go through n or through any node that n dominates; and there is a path P_{nw} from n to w that goes only through dominated nodes. These paths have w as their first point of convergence.

Computing the dominance frontier. To insert all the necessary ϕ-functions, for every node n in the flow graph we need $DF[n]$, its dominance frontier. Given the dominator tree, we can efficiently compute the dominance frontiers of all the nodes of the flow graph in one pass. We define two auxiliary sets

$DF_{\text{local}}[n]$: The successors of n that are not strictly dominated by n;

$DF_{\text{up}}[n]$: Nodes in the dominance frontier of n that are not dominated by n's immediate dominator.

The dominance frontier of n can be computed from DF_{local} and DF_{up}:

$$DF[n] = DF_{\text{local}}[n] \cup \bigcup_{c \in children[n]} DF_{\text{up}}[c]$$

where $children[n]$ are the nodes whose immediate dominator ($idom$) is n.

To compute $DF_{\text{local}}[n]$ more easily (using immediate dominators instead of dominators), we use the following theorem: $DF_{\text{local}}[n]$ = the set of those successors of n whose immediate dominator is not n.

The following computeDF function should be called on the root of the dominator tree (the start node of the flow graph). It walks the tree computing $DF[n]$ for every node n: it computes $DF_{\text{local}}[n]$ by examining the successors of n, then combines $DF_{\text{local}}[n]$ and (for each child c) $DF_{\text{up}}[c]$.

```
computeDF[n] =
    S ← {}
    for each node y in succ[n]                    This loop computes DF_local[n]
        if idom(y) ≠ n
            S ← S ∪ {y}
    for each child c of n in the dominator tree
        computeDF[c]
        for each element w of DF[c]               This loop computes DF_up[c]
            if n does not dominate w
                S ← S ∪ {w}
    DF[n] ← S
```

This algorithm is quite efficient. It does work proportional to the size (number of edges) of the original graph, plus the size of the dominance frontiers it computes. Although there are pathological graphs in which most of the nodes have very large dominance frontiers, in most cases the total size of all the DFs is approximately linear in the size of the graph, so this algorithm runs in "practically" linear time.

INSERTING ϕ-FUNCTIONS

Starting with a program not in SSA form, we need to insert just enough ϕ-functions to satisfy the iterated dominance frontier criterion. To avoid re-

```
Place-φ-Functions =
    for each node n
        for each variable a in A_orig[n]
            defsites[a] ← defsites[a] ∪ {n}
    for each variable a
        W ← defsites[a]
        while W not empty
                remove some node n from W
                for each y in DF[n]
                    if y ∉ A_φ[a]
                      insert the statement a ← φ(a, a, ..., a) at the top
                        of block y, where the φ-function has as many
                        arguments as y has predecessors
                      A_φ[a] ← A_φ[a] ∪ {y}
                      if a ∉ A_orig[y]
                        W ← W ∪ {y}
```

ALGORITHM 19.6. Inserting ϕ-functions.

examining nodes where no ϕ-function has been inserted, we use a work-list algorithm.

Algorithm 19.6 starts with a set V of variables, a graph G of control-flow nodes – each node is a basic block of statements – and for each node n a set $A_{\text{orig}}[n]$ of variables defined in node n. The algorithm computes $A_\phi[a]$, the set of nodes that must have ϕ-functions for variable a. Sometimes a node may contain both an ordinary definition and a ϕ-function for the same variable; for example, in Figure 19.3b, $a \in A_{\text{orig}}[2]$ and $2 \in A_\phi[a]$.

The outer loop is performed once for each variable a. There is a work-list W of nodes that might violate the dominance-frontier criterion.

The representation for W must allow quick testing of membership and quick extraction of an element. Work-list algorithms (in general) do not care *which* element of the list they remove, so an array or linked list of nodes suffices. To quickly test membership in W, we can use a mark bit in the representation of every node n which is set to **true** when n is put into the list, and **false** when n is removed. If it is undesirable to modify the node representation, a list plus a hash table will also work efficiently.

This algorithm does a constant amount of work (a) for each node and edge in the control-flow graph, (b) for each statement in the program, (c) for each element of every dominance frontier, and (d) for each inserted ϕ-function. For a program of size N, the amounts a and b are proportional to N, c is usually approximately linear in N. The number of inserted ϕ-functions (d) could be N^2 in the worst case, but empirical measurement has shown that it is usually proportional to N. So in practice, Algorithm 19.6 runs in approximately linear time.

RENAMING THE VARIABLES

After the ϕ-functions are placed, we can walk the dominator tree, renaming the different definitions (including ϕ-functions) of variable a to a_1, a_2, a_3 and so on.

In a straight-line program, we would rename all the definitions of a, and then each use of a is renamed to use the most recent definition of a. For a program with control-flow branches and joins whose graph satisfies the dominance-frontier criterion, we rename each use of a to use the closest definition d of a that is above a in the dominator tree.

Algorithm 19.7 renames all uses and definitions of variables, after the ϕ-functions have been inserted by Algorithm 19.6. In traversing the dominator tree, the algorithm "remembers" for each variable the most recently defined version of each variable, on a separate stack for each variable.

Although the algorithm follows the structure of the dominator tree – not the flow graph – at each node in the tree it examines all outgoing flow edges, to see if there are any ϕ-functions whose operands need to be properly numbered.

This algorithm takes time proportional to the size of the program (after ϕ-functions are inserted), so in practice it should be approximately linear in the size of the original program.

EDGE SPLITTING

Some analyses and transformations are simpler if there is never a control-flow edge that leads from a node with multiple successors to a node with multiple predecessors. To give the graph this *unique successor or predecessor* property, we perform the following transformation: For each control-flow edge $a \to b$ such that a has more than one successor and b has more than one predecessor, we create a new, empty control-flow node z, and replace the $a \to b$ edge with an $a \to z$ edge and a $z \to b$ edge.

```
Initialization:
    for each variable a
        Count[a] ← 0
        Stack[a] ← empty
        push 0 onto Stack[a]

Rename(n) =
    for each statement S in block n
        if S is not a φ-function
          for each use of some variable x in S
              i ← top(Stack[x])
              replace the use of x with x_i in S
        for each definition of some variable a in S
            Count[a] ← Count[a] + 1
            i ← Count[a]
            push i onto Stack[a]
            replace definition of a with definition of a_i in S
    for each successor Y of block n,
        Suppose n is the jth predecessor of Y
        for each φ-function in Y
            suppose the jth operand of the φ-function is a
            i ← top(Stack[a])
            replace the jth operand with a_i
    for each child X of n
        Rename(X)
    for each definition of some variable a in the original S
        pop Stack[a]
```

ALGORITHM 19.7. Renaming variables.

An SSA graph with this property is in *edge-split SSA form*. Figure 19.2 illustrates edge splitting. Edge splitting may be done before or after insertion of ϕ-functions.

19.2 EFFICIENT COMPUTATION OF THE DOMINATOR TREE

A major reason for using SSA form is that it makes the optimizing compiler faster. Instead of using costly iterative bit-vector algorithms to link uses to definitions (to compute reaching definitions, for example), the compiler can just look up the (unique) definition, or the list of uses, of each variable.

For SSA to help make a compiler faster, we must be able to compute the SSA form quickly. The algorithms for computing SSA from the dominator tree are quite efficient. But the iterative set-based algorithm for computing dominators, given in Section 18.1, may be slow in the worst case. An industrial-strength compiler that uses dominators should use a more efficient algorithm for computing the dominator tree.

The near-linear-time algorithm of Lengauer and Tarjan relies on properties of the *depth-first spanning tree* of the control-flow graph. This is just the recursion tree implicitly traversed by the *depth-first search* (DFS) algorithm, which numbers each node of the graph with a *depth-first number (dfnum)* as it is first encountered.

The algorithm is rather technical; those readers who feel content just knowing that the dominator tree can be calculated quickly can skip to Section 19.3.

DEPTH-FIRST SPANNING TREES

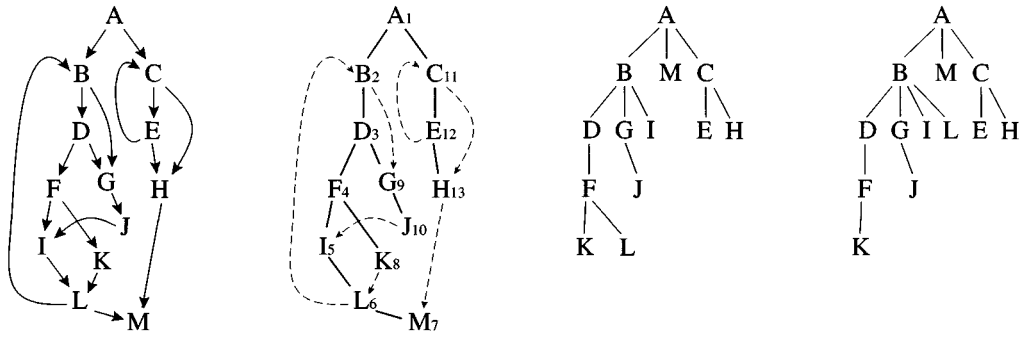
We can use depth-first search to calculate a depth-first spanning tree of the control-flow graph. Figure 19.8 shows a CFG and a depth-first spanning tree, along with the *dfnum* of each node.

A given CFG may have many different depth-first spanning trees. From now on I will assume that we have arbitrarily picked one of them – by depth-first search. When I say "a is an *ancestor* of b" I mean that there is some path from a to b following only spanning-tree edges, or that $a = b$; "a is a *proper ancestor* of b" means that a is an ancestor of b and $a \neq b$.

Properties of depth-first spanning trees. The start node r of the CFG is the root of the depth-first spanning tree.

If a is a proper ancestor of b, then $dfnum(a) < dfnum(b)$.

Suppose there is a CFG path from a to b but a is not an ancestor of b. This means that some edge on the path is not a spanning-tree edge, so b must have been reached in the depth-first search before a was (otherwise, after visiting a the search would continue along tree-edges to b). Thus, $dfnum(a) > dfnum(b)$.



(a) Graph (b) Depth-first spanning tree (c) Semidominator tree (d) Dominator tree

FIGURE 19.8. A control-flow graph and trees derived from it. The numeric labels in part (b) are the *dfnum*s of the nodes.

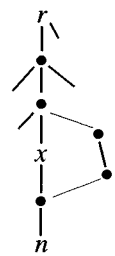
Therefore, if we know that there is a path from a to b, we can test whether a is an ancestor of b just by comparing the *dfnum*'s of a and b.

When drawing depth-first spanning trees, we order the children of a node in the order that they are visited by the depth-first search, so that nodes to the right have a higher *dfnum*. This means that if a is an ancestor of b, and there is a CFG path from a to b that departs from the spanning tree, it must branch off to the right of the tree path, never to the left.

Dominators and spanning-tree paths. Consider a non-root node n in the CFG, and its immediate dominator d. The node d must be an ancestor of n in the spanning tree – because any path (including the spanning-tree path) from r to n must include d. Therefore $dfnum(d) < dfnum(n)$.

Now we know that n's immediate dominator must be on the spanning-tree path between r and n; all that's left is to see how high up it is.

If some ancestor x does not dominate n, then there must be a path that departs from the spanning-tree path above x and rejoins it below x. The nodes on the by-passing path are not ancestors of n, so their *dfnum*'s are higher than n's. The path might rejoin the spanning-tree path to n either at n or above n.

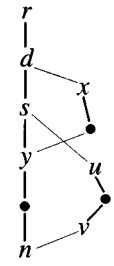


SEMIDOMINATORS

Paths that bypass ancestors of n are useful for proving that those ancestors do not dominate n. Let us consider, for now, only those bypassing paths that rejoin the spanning tree at n (not above n). Let's find the path that departs from the tree at the highest possible ancestor s, and rejoins the tree at n. We will call s the *semidominator* of n.

Another way of saying this is that s is the node of smallest *dfnum* having a path to n whose nodes (not counting s and n) are not ancestors of n. This description of semidominators does not explicitly say that s must be an ancestor of n, but of course any non-ancestor with a path to n would have a higher *dfnum* than n's own parent in the spanning tree, which itself has a path to n with no non-ancestor internal nodes (actually, no internal nodes at all).

Very often, a node's semidominator is also its immediate dominator. But as the figure at right shows, to find the dominator of n it's not enough just to consider bypassing paths that rejoin the tree at n. Here, a path from r to n bypasses n's semidominator s, but rejoins the tree at node y, above n. However, finding the semidominator s is still a useful step toward finding the dominator d.



Semidominator Theorem. To find the semidominator of a node n, consider all predecessors v of n in the CFG.

- If v is a proper ancestor of n in the spanning tree (so $dfnum(v) < dfnum(n)$), then v is a candidate for $semi(n)$.
- If v is a non-ancestor of n (so $dfnum(v) > dfnum(n)$) then for each u that is an ancestor of v (or $u = v$), let $semi(u)$ be a candidate for $semi(n)$.

Of all these candidates, the one with lowest *dfnum* is the semidominator of n.

Proof. See the Further Reading section.

Calculating dominators from semidominators. Let s be the semidominator of n. If there is a path that departs from the spanning tree above s, bypasses s, and rejoins the spanning tree at some node between s and n, then s does not dominate n.

However, if we find the node y between s and n with the smallest-numbered semidominator, and $semi(y)$ is a proper ancestor of s, then y's immediate dominator also immediately dominates n.

Dominator Theorem. On the spanning-tree path below $semi(n)$ and above or including n, let y be the node with the smallest-numbered semidominator (minimum $dfnum(semi(y))$). Then,

$$idom(n) = \begin{cases} semi(n) & \text{if } semi(y) = semi(n) \\ idom(y) & \text{if } semi(y) \neq semi(n) \end{cases}$$

Proof. See the Further Reading section.

THE LENGAUER-TARJAN ALGORITHM

Using these two theorems, Algorithm 19.9 uses depth-first search (DFS) to compute *dfnum*'s for every node.

Then it visits the nodes in order, from highest *dfnum* to lowest, computing semidominators and dominators. As it visits each node, it puts the node into a spanning forest for the graph. It's called a *forest* because there may be multiple disconnected fragments; only at the very end will it be a single spanning *tree* of all the CFG nodes.

Calculating semidominators requires that, given some edge $v \rightarrow n$, we look at all ancestors of v in the spanning tree that have higher *dfnum* than n. When Algorithm 19.9 processes node n, only nodes with higher *dfnum* than n will be in the forest. Thus, the algorithm can simply examine all ancestors of v that are already in the forest.

We use the Dominator Theorem to compute the immediate dominator of n, by finding node y with lowest semidominator on the path from $semi[n]$ to n. When $s = semi[n]$ is being computed, it's not yet possible to determine y; but we will be able to do so later, when s is being added to the spanning forest. Therefore with each semidominator s we keep a *bucket* of all the nodes that s semidominates; when s is linked into the spanning forest, we can then calculate the *idom* of each node in $[s]$.

The forest is represented by an *ancestor* array: for each node v in the forest, $ancestor[v]$ points to v's parent. This makes searching upward from v easy.

Algorithm 19.10a shows a too-slow version of the AncestorWithLowestSemi and Link functions that manage the spanning forest. Link sets the *ancestor* relation, and AncestorWithLowestSemi searches upward for the ancestor whose semidominator has the smallest *dfnum*.

But each call to AncestorWithLowestSemi could take linear time (in N, the number of nodes in the graph) if the spanning tree is very deep; and AncestorWithLowestSemi is called once for each node and edge. Thus Algorithm 19.9+19.10a has quadratic worst-case time complexity.

```
DFS(node p, node n) =
    if dfnum[n] = 0
        dfnum[n] ← N;  vertex[N] ← n;  parent[n] ← p
        N ← N + 1
        for each successor w of n
            DFS(n, w)

Link(node p, node n) = add edge p → n to spanning forest implied by ancestor array
AncestorWithLowestSemi(node n) = in the forest, find the nonroot ancestor of n that
                                 has the lowest-numbered semidominator

Dominators() =
    N ← 0;  ∀n. bucket[n] ← {}
    ∀n. dfnum[n] ← 0,  semi[n] ← ancestor[n] ← idom[n] ← samedom[n] ← none
    DFS(none, r)
    for i ← N − 1 downto 1                          Skip over node 0, the root node.
        n ← vertex[i];   p ← parent[n];   s ← p
        for each predecessor v of n
            if dfnum[v] ≤ dfnum[n]                  These lines calculate the semidominator of n, based
                s' ← v                              on the Semidominator Theorem.
            else s' ← semi[AncestorWithLowestSemi(v)]
            if dfnum[s'] < dfnum[s]
                s ← s'
        semi[n] ← s
        bucket[s] ← bucket[s] ∪ {n}                 Calculation of n's dominator is deferred
        Link(p, n)                                  until the path from s to n has been linked
        for each v in bucket[p]                     into the forest.
            y ← AncestorWithLowestSemi(v)           Now that the path from p to v has been linked into
            if semi[y] = semi[v]                    the spanning forest, these lines calculate the dom-
                idom[v] ← p                         inator of v, based on the first clause the Domina-
            else samedom[v] ← y                     tor Theorem, or else defer the calculation until y's
        bucket[p] ← {}                              dominator is known.
    for i ← 1 to N − 1
        n ← vertex[i]                               Now all the deferred dominator calcula-
        if samedom[n] ≠ none                        tions, based on the second clause of the
            idom[n] ← idom[samedom[n]]              Dominator Theorem, are performed.
```

ALGORITHM 19.9. Lengauer-Tarjan algorithm for computing dominators.

```
AncestorWithLowestSemi(node v) =
    u ← v
    while ancestor[v] ≠ none
        if dfnum[semi[v]] < dfnum[semi[u]]
            u ← v
        v ← ancestor[v]
    return u

Link(node p, node n) =
    ancestor[n] ← p
```

(a) Naive version, $O(N)$ per operation.

```
AncestorWithLowestSemi(node v) =
    a ← ancestor[v]
    if ancestor[a] ≠ none
        b ← AncestorWithLowestSemi(a)
        ancestor[v] ← ancestor[a]
        if dfnum[semi[b]] <
                dfnum[semi[best[v]]]
            best[v] ← b
    return best[v]

Link(node p, node n) =
    ancestor[n] ← p;    best[n] ← n
```

(b) With path-compression, $O(\log N)$ per operation.

ALGORITHM 19.10. Two versions of AncestorWithLowestSemi and Link functions for operations on spanning forest. The naive version (a) takes $O(N)$ per operation (so the algorithm runs in time $O(N^2)$) and the efficient version (b) takes $O(\log N)$ amortized time per operation, for an $O(N \log N)$ algorithm.

Path compression. The algorithm may call AncestorWithLowestSemi(v) several times for the same node v. The first time, AncestorWithLowestSemi traverses the nodes from v to a_1, some ancestor of v, as shown in Figure 19.11a. Then perhaps some new links $a_3 \rightarrow a_2 \rightarrow a_1$ are added to the forest above a_1, so the second AncestorWithLowestSemi(v) searches up to a_3. But we would like to avoid the duplicate traversal of the path from v to a_1. Furthermore, suppose we later call AncestorWithLowestSemi(w) on some child of v. During that search we would like to be able to skip from v to a_1.

The technique of *path compression* makes AncestorWithLowestSemi faster. For each node v in the spanning forest, we allow *ancestor*$[v]$ to point to some ancestor of v that may be far above v's parent. But then we must remember – in *best*$[v]$ – the best node in the skipped-over path between *ancestor*$[v]$ and v.

ancestor$[v]$ = Any node above v in the spanning forest.
best$[v]$ = The node whose semidominator has lowest *dfnum*, in the skipped-over path from *ancestor*$[v]$ down to v (including v but not *ancestor*$[v]$).

Now, when AncestorWithLowestSemi searches upwards, it can compress

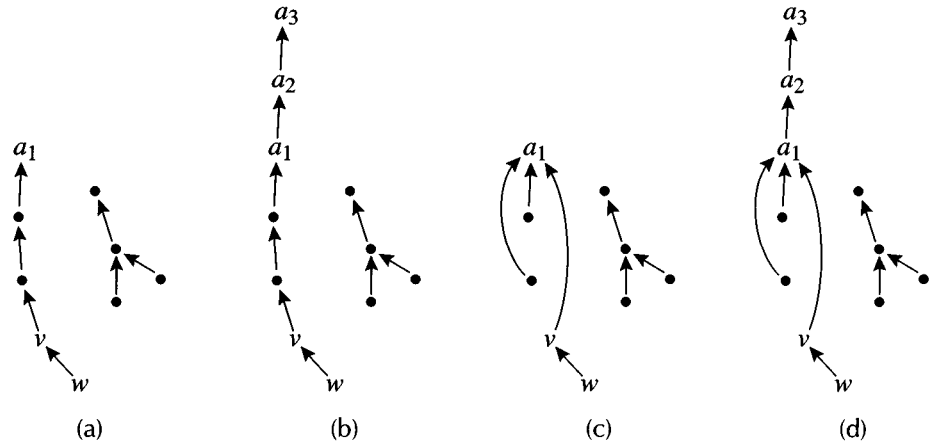


FIGURE 19.11. Path compression. (a) *Ancestor* links in a spanning tree; AncestorWithLowestSemi(v) traverses three links. (b) New nodes a_2, a_3 are linked into the tree. Now AncestorWithLowestSemi(w) would traverse 6 links. (c) AncestorWithLowestSemi(v) with path compression redirects ancestor links, but *best*[v] remembers the best intervening node on the compressed path between v and a_1. (d) Now, after a_2 and a_3 are linked, AncestorWithLowestSemi(w) traverses only 4 links.

paths by setting *ancestor*[v] $\leftarrow$ *ancestor*[*ancestor*[v]], as long as it updates *best*[v] at the same time. This is shown in Algorithm 19.10b.

In a graph of K nodes and E edges, there will be $K - 1$ calls to Link and $E + K - 1$ calls to AncestorWithLowestSemi. With path compression it can be shown that this takes $O(E \log K)$ time. In terms of the "size" $N = E + K$ of the control-flow graph, Algorithm 19.9+19.10b takes $O(N \log N)$ time.

Balanced path compression. The most sophisticated version of the Lengauer-Tarjan algorithm is Algorithm 19.9 with Link and AncestorWithLowestSemi functions that rebalance the spanning trees, so that the work of path compression is undertaken only when it will do the most good. This algorithm has time complexity $O(N \cdot \alpha(N))$, where $\alpha(N)$ is the slowly growing inverse-Ackermann function that is for all practical purposes constant. In practice it appears that this sophisticated algorithm is about 35% faster than the $N \log N$ algorithm (when measured on graphs of up to 1000 nodes). See also the Further Reading section of this chapter.

19.3 OPTIMIZATION ALGORITHMS USING SSA

Since we are primarily interested in SSA form because it provides quick access to important dataflow information, we should pay some attention to data-structure representations of the SSA graph.

The objects of interest are *statements, basic blocks, and variables*:

Statement Fields of interest are *containing block, previous statement in block, next statement in block, variables defined, variables used.* Each statement may be an *ordinary assignment, ϕ-function, fetch, store, or branch.*

Variable Has a *definition site (statement)* and a list of *use sites.*

Block Has a *list of statements, an ordered list of predecessors, a successor (for blocks ending with a conditional branch, more than one successor).* The order of predecessors is important for determining the meaning $\phi(v_1, v_2, v_3)$ inside the block.

DEAD-CODE ELIMINATION

The SSA data structure makes dead-code analysis particularly quick and easy. A variable is live at its site of definition if and only if its list of uses is not empty. This is true because there can be no other definition of the same variable (it's single-assignment form!) and the definition of a variable dominates every use – so there must be a path from definition to use.[1]

This leads to the following iterative algorithm for deleting dead code:

while there is some variable v with no uses
  and the statement that defines v has no other side-effects
    do delete the statement that defines v

In deleting a statement $v \leftarrow x \oplus y$ or the statement $v \leftarrow \phi(x, y)$, we take care to remove the statement from the list of uses of x and of y. This may cause x or y to become dead, if it was the last use. To keep track of this efficiently, Algorithm 19.12 uses a work-list W of variables that need to be reconsidered. This takes time proportional to the size of the program plus the number of variables deleted (which itself cannot be larger than the size of the program) – or linear time overall. The only question is how long it takes to delete S from a (potentially long) list of uses of x_i. By keeping x_i's list of uses as a doubly linked list, and having each use of x_i point back *to its own entry in this list*, the deletion can be done in constant time.

[1] As usual, we are considering only connected graphs.

```
W ← a list of all variables in the SSA program
while W is not empty
    remove some variable v from W
    if v's list of uses is empty
      let S be v's statement of definition
      if S has no side effects other than the assignment to v
        delete S from the program
        for each variable x_i used by S
            delete S from the list of uses of x_i
            W ← W ∪ {x_i}
```

ALGORITHM 19.12. Dead-code elimination in SSA form.

If run on the program of Figure 19.3b, this algorithm would delete the statement $b_1 \leftarrow \phi(b_0, b_2)$.

A more aggressive dead-code-elimation algorithm, which uses a different definition of *dead*, is shown on page 455.

SIMPLE CONSTANT PROPAGATION

Whenever there is a statement of the form $v \leftarrow c$ for some constant c, then any use of v can be replaced by a use of c.

Any ϕ-function of the form $v \leftarrow \phi(c_1, c_2, \ldots, c_n)$, where all the c_i are equal, can be replaced by $v \leftarrow c$.

Each of these conditions is easy to detect and implement using the SSA data structure, and we can use a simple work-list algorithm to propagate constants:

```
W ← a list of all statements in the SSA program
while W is not empty
    remove some statement S from W
    if S is v ← φ(c, c, ..., c) for some constant c
      replace S by v ← c
    if S is v ← c for some constant c
      delete S from the program
      for each statement T that uses v
          substitute c for v in T
          W ← W ∪ {T}
```

If we run this algorithm on the SSA program of Figure 19.4g, then the assignment $j_3 \leftarrow i$ will be replaced with $j_3 \leftarrow 1$, and the assignment $i_1 \leftarrow 1$ will be deleted. Uses of variables j_1 and k_1 will also be replaced by constants.

The following transformations can all be incorporated into this work-list algorithm, so that in linear time all these optimizations can be done at once:

Copy propagation A single-argument ϕ-function $x \leftarrow \phi(y)$ or a copy assignment $x \leftarrow y$ can be deleted, and y substituted for every use of x.

Constant folding If we have a statement $x \leftarrow a \oplus b$, where a and b are constant, we can evaluate $c \leftarrow a \oplus b$ at compile time and replace the statement with $x \leftarrow c$.

Constant conditions In block L, a conditional branch **if** $a < b$ **goto** L_1 **else** L_2, where a and b are constant, can be replaced by either **goto** L_1 or **goto** L_2, depending on the (compile-time) value of $a < b$. The control-flow edge from L to L_2 (or L_1, respectively) must be deleted; this reduces the number of predecessors of L_2 (or L_1), and the ϕ-functions in that block must be adjusted accordingly (by removing an argument).

Unreachable code Deleting a predecessor may cause block L_2 to become unreachable. In this case, all the statements in L_2 can be deleted; use-lists of all the variables that are used in these statements must be adjusted accordingly. Then the block itself should be deleted, reducing the number of predecessors of *its* successor blocks.

CONDITIONAL CONSTANT PROPAGATION

In the program of Figure 19.4b, is j always equal to 1?

- If $j = 1$ always, then block 6 will never execute, so the only assigment to j is $j \leftarrow i$, so $j = 1$ always.
- If sometimes $j > 20$, then block 6 will eventually execute, which assigns $j \leftarrow k$, so that eventually $j > 20$.

Each of these statements is self-consistent; but which is true in practice? In fact, when this program executes, j is never set to any value other than 1. This is a kind of *least fixed point* (analogous to what is described in Section 10.1 on page 217).

The "simple" constant-propagation algorithm has the problem of assuming the block 6 might be executed, and therefore that j might not be constant, and therefore that perhaps $j \geq 20$, and therefore that block 6 might be executed. Simple constant propagation finds a fixed point that is not the least fixed point.

Why would programmers put never-executed statements in their programs? Many programs have statements of the form `if debug then ...` where `debug` is a constant *false* value; we would not like to let the statements in the debug-clauses get in the way of useful optimizations.

The *SSA conditional constant propagation* finds the least fixed point: it *does not assume a block can be executed until there is evidence that it can be*, and *does not assume a variable is non-constant until there is evidence*, and so on.

The algorithm tracks the run-time value of each variable as follows:

$\mathcal{V}[v] = \bot$ We have seen no evidence that any assignment to v is ever executed.
$\mathcal{V}[v] = 4$ We have seen evidence that an assignment $v \leftarrow 4$ is executed, but no evidence that v is ever assigned any other value.
$\mathcal{V}[v] = \top$ We have seen evidence that v will have, at various times, at least two different values, or some value (perhaps read from an input file or from memory) that is not predictable at compile time.

Thus we have a lattice of values, with $\bot$ meaning *never defined,* 4 meaning *defined as 4,* and $\top$ meaning *overdefined:*

... 3 4 5 6 7 ...

New information can only move a variable up in the lattice.[2]

We also track the executability of each block, as follows:

$\mathcal{E}[B] = false$ We have seen no evidence that block B can ever be executed.
$\mathcal{E}[B] = true$ We have seen evidence that block B can be executed.

Initially we start with $\mathcal{V}[\,] = \bot$ for all variables, and $\mathcal{E}[\,] = false$ for all blocks. Then we observe the following:

1. Any variable v with no definition, which is therefore an input to the program, a formal parameter to the procedure, or (horrors!) an uninitialized variable, must have $\mathcal{V}[v] \leftarrow \top$.
2. The start block B_1 is executable: $\mathcal{E}[B_1] \leftarrow true$.
3. For any executable block B with only one successor C, set $\mathcal{E}[C] \leftarrow true$.

[2] Authors in the subfield of dataflow analysis use $\bot$ to mean overdefined and $\top$ to mean never defined; authors in semantics and abstract interpretation use $\bot$ for undefined and $\top$ for overdefined; I am following the latter practice.

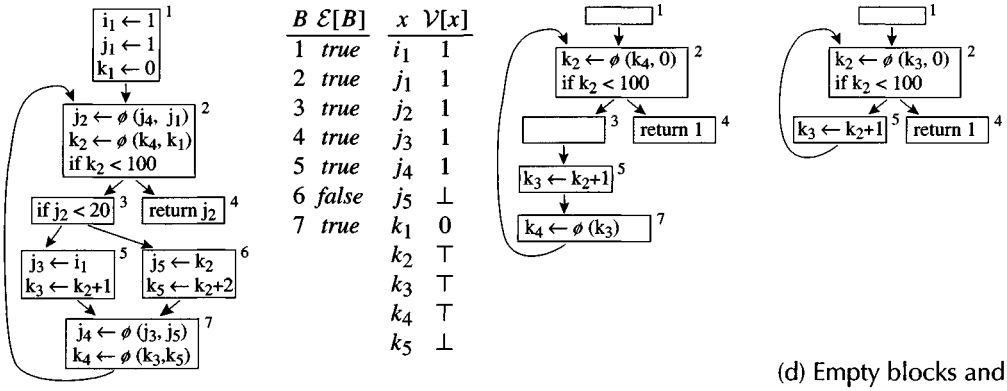
4. For any executable assignment $v \leftarrow x \oplus y$ where $\mathcal{V}[x] = c_1$ and $\mathcal{V}[y] = c_2$, set $\mathcal{V}[v] \leftarrow c_1 \oplus c_2$.
5. For any executable assignment $v \leftarrow x \oplus y$ where $\mathcal{V}[x] = \top$ or $\mathcal{V}[y] = \top$, set $\mathcal{V}[v] \leftarrow \top$.
6. For any executable assignment $v \leftarrow \phi(x_1, \ldots, x_n)$, where $\mathcal{V}[x_i] = c_1$, $\mathcal{V}[x_j] = c_2$, $c_1 \neq c_2$, the ith predecessor is executable, and the jth predecessor is executable, set $\mathcal{V}[v] \leftarrow \top$.
7. For any executable assignment $v \leftarrow$ MEM() or $v \leftarrow$ CALL(), set $\mathcal{V}[v] \leftarrow \top$.
8. For any executable assignment $v \leftarrow \phi(x_1, \ldots, x_n)$ where $\mathcal{V}[x_i] = \top$ and the ith predecessor is executable, set $\mathcal{V}[v] \leftarrow \top$.
9. For any assignment $v \leftarrow \phi(x_1, \ldots, x_n)$ whose ith predecessor is executable and $\mathcal{V}[x_i] = c_1$; and for every j either the jth predecessor is not executable, or $\mathcal{V}[x_j] = \bot$, or $\mathcal{V}[x_j] = c_1$, set $\mathcal{V}[v] \leftarrow c_1$.
10. For any executable branch **if** $x < y$ **goto** L_1 **else** L_2, where $\mathcal{V}[x] = \top$ or $\mathcal{V}[y] = \top$, set $\mathcal{E}[L_1] \leftarrow$ *true* and $\mathcal{E}[L_2] \leftarrow$ *true*.
11. For any executable branch **if** $x < y$ **goto** L_1 **else** L_2, where $\mathcal{V}[x] = c_1$ and $\mathcal{V}[y] = c_2$, set $\mathcal{E}[L_1] \leftarrow$ *true* or $\mathcal{E}[L_2] \leftarrow$ *true* depending on $c_1 < c_2$.

An *executable assignment* is an assignment statement in a block B with $\mathcal{E}[B] =$ *true*. These conditions "ignore" any expression or statement in an unexecutable block, and the ϕ-functions "ignore" any operand that comes from an unexecutable predecessor.

The algorithm can be made quite efficient using work-lists: there will be one work-list W_v for variables and and another work-list W_b for blocks. The algorithm proceeds by picking x from W_v and considering conditions 4–9 for any statement in x's list of uses; or by picking an block B from W_b and considering condition 3, and conditions 4–9 for any statement within B. Whenever a block is newly marked executable, it *and its executable successors* are added to W_e. Whenever $\mathcal{V}[x]$ is "raised" from $\bot$ to c or from c to $\top$, then x is added to W_v. When both W_v and W_b are empty, the algorithm is finished. The algorithm runs quickly, because for any x it raises $\mathcal{V}[x]$ at most twice, and for any B it changes $\mathcal{E}[B]$ at most once.

We use this information to optimize the program as follows. After the analysis terminates, wherever $\mathcal{E}[B] =$ *false*, delete block B. Wherever $\mathcal{V}[x] = c$, substitute c for x and delete the assignment to x.

Figure 19.13 shows the conditional constant propagation algorithm executed on the program of Figure 19.4. The algorithm finds that all the j variables are constant (with value 1), k_1 is constant (with value 0), and block 6 is not executed. Deleting unreachable blocks, and replacing uses of constant variables with the constant value – deleting their definitions – leads to some



(a) SSA program (same as 19.4g) (b) $\mathcal{E}$ and $\mathcal{V}$ (c) After constant propagation (d) Empty blocks and single-argument ϕ-functions deleted

FIGURE 19.13. Conditional constant propagation.

empty blocks and a ϕ-function that has only one argument; these can be simplified, leaving the program of Figure 19.13d.

The *unique successor or predecessor* property is important for the proper operation of this algorithm. Suppose we were to do conditional constant propagation on the graph of Figure 19.2b, in a case where $M[x]$ is known to be 10. Then blocks 1, 2, 3, and 4 will be marked executable, but it will not be clear that edge $2 \rightarrow 4$ cannot be taken. In Figure 19.2c, block 5 would not be executable, making the situation clear. By using the *edge-split SSA form*, we avoid the need to mark *edges* (not just blocks) executable.

PRESERVING THE DOMINANCE PROPERTY

Almost every reasonable optimizing transformation – including the ones described above – preserves the *dominance property* of the SSA program: the definition of a variable dominates each use (or, when the use is in a ϕ-function, the predecessor of the use).

It is important to preserve this property, since some optimization algorithms (such as Algorithm 19.17) depend on it. Also, the very definition of SSA form – that there is a ϕ-function at the convergence point of any two dataflow paths – implicitly requires it.

But there is one kind of optimization that does not preserve the dominance property. In the program of Figure 19.14a, we can prove that – because the condition $z > 0$ evaluates the same way in blocks 1 and 4 – the use of x_2 in

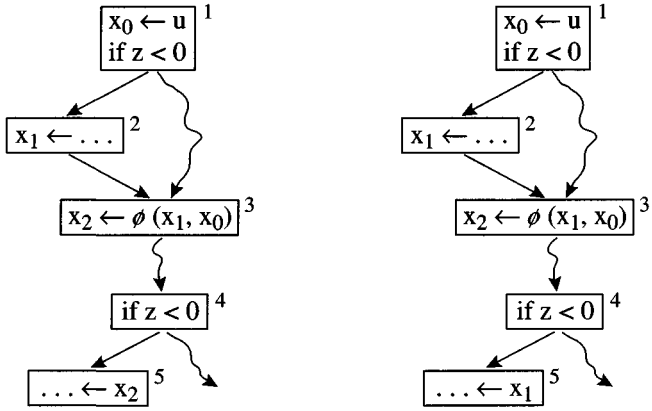


FIGURE 19.14. This transformation does not preserve the dominance property of SSA form, and should be avoided.

block 5 always gets the value x_1, never x_0. Thus it is tempting to substitute x_1 for x_2 in block 5. But the resulting graph does not have the dominance property: block 5 is not dominated by the definition of x_1 in block 2.

Therefore this kind of transformation – based on the knowledge that two conditional branches test the same condition – is not valid for SSA form.

19.4 ARRAYS, POINTERS, AND MEMORY

For many purposes in optimization, parallelization, and scheduling, the compiler needs to know, "how does statement B depend on statement A?" The transformations of constant propagation and dead-code removal have relied on this dependence information.

There are several kinds of dependence relations:

Read-after-write A defines variable v, then B uses v.
Write-after-write A defines v, then B defines v.
Write-after-read A uses v, then B defines v.
Control A controls whether B executes.

Read-after-write dependences are evident in the SSA graph: A defines v, v's list of uses points to B; or B's *use* list contains v, and v's def-site is A.

Control dependences will be discussed in Section 19.5.

In SSA form, there are no write-after-write or write-after-read dependences. Statements A and B can never write to the same variable, and any use must be "after" (that is, dominated by) the variable's definition.

MEMORY DEPENDENCE

The discussion thus far of assigments and ϕ-function has been only for scalar non-escaping variables. Real programs must also load and store memory words.

One way to get a single-assignment property for memory is to ensure that each memory word is written only once. Although this seems severe, it is just what a pure functional programming language does (see Chapter 15) – with a garbage collector behind the scenes to allow actual reuse of physical memory locations.

However, in an imperative language we must do something else. Consider a sequence of stores and fetches such as this one:

$$\begin{array}{ll} 1 & M[i] \leftarrow 4 \\ 2 & x \leftarrow M[j] \\ 3 & M[k] \leftarrow j \end{array}$$

We cannot treat each individual memory location as a separate variable for static-single-assigment purposes, because we don't know whether i, j, and k are the same address.

We could perhaps treat memory as a "variable," where the *store* instruction creates a new value (of the entire memory):

$$\begin{array}{ll} 1 & M_1 \leftarrow store(M_0, i, 4) \\ 2 & x \leftarrow load(M_1, j) \\ 3 & M_2 \leftarrow store(M_1, k, j) \end{array}$$

This creates the def-use edges $1 \xrightarrow{M_1} 2$ and $1 \xrightarrow{M_1} 3$. These def-use edges are like any SSA def-use relationship, and we make ϕ-functions for them at join points in the same way.

But there is no edge from $2 \rightarrow 3$, so what prevents the compiler from reordering the statements as follows?

$$\begin{array}{ll} 1 & M_1 \leftarrow store(M_0, i, 4) \\ 3 & M_2 \leftarrow store(M_1, k, j) \\ 4 & x \leftarrow load(M_1, j) \end{array}$$

The functional dependences are still correct – if M_1 is viewed as a snapshot of memory after statement 1, then statement 4 is still correct in loading from address j in that snapshot. But it is inefficient – to say the least! – for the computer to keep more than one copy of the machine's memory.

We would like to say that there is a *write-after-read* dependence $2 \rightarrow 3$ to prevent the compiler from creating M_2 before all uses of M_1 have been computed. But calculation of accurate dependence information for memory locations is beyond the scope of this chapter.

A naive but practical solution. In the absence of write-after-read and write-after-write dependence information, we will just say that a store instruction is always presumed live – we will not do dead-code elimination on stores – and we will not transform the program in such a way as to interchange a load and a store, or two stores. Store instructions can be unreachable, however, and unreachable stores can be deleted.

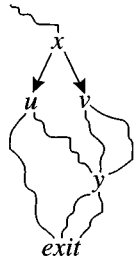
The optimization algorithms presented in this chapter do not reorder instructions, and do not attempt to propagate dataflow information through memory, so they implicitly use this naive model of loads and stores.

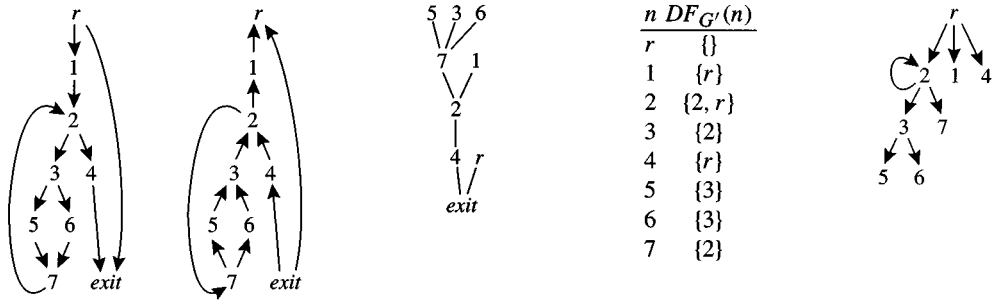
19.5 THE CONTROL-DEPENDENCE GRAPH

Can node x directly control whether node y is executed? The answer to this question can help us with program transformations and optimizations.

Any flow graph must have an *exit* node. If a control-flow graph represents a single function, then this is the **return** statement of the function; if there are several **return** statements, we assume that each one of them is really a control-flow edge to some unique canonical *exit* node of the CFG.

We say that a node y is *control-dependent* on x if from x we can branch to u or v; from u there is a path to *exit* that avoids y, and from v every path to *exit* hits y:





(a) CFG (from Fig. 19.4b) (b) Reverse CFG (c) Post-dominators (d) Postdominance frontiers (e) CDG

FIGURE 19.15. Construction of the control-dependence graph.

The *control-dependence graph* (CDG) has an edge from x to y whenever y is control-dependent on x.

We say that y *postdominates* v when y is on every path from v to *exit* – that is, y dominates v in the *reverse* control-flow graph.

Construction of the control-dependence graph. To construct the CDG of a control-flow graph G,

1. Add a new *entry*-node r to G, with an edge $r \rightarrow s$ to the start node s of G (indicating that the surrounding program might enter G) and an edge $r \rightarrow$ *exit* to the exit node of G (indicating that the surrounding program might not execute G at all).
2. Let G' be the *reverse control-flow graph* that has an edge $y \rightarrow x$ whenever G has an edge $x \rightarrow y$; the *start* node of G' corresponds to the *exit* node of G.
3. Construct the dominator tree of G' (its root corresponds to the *exit* node of G).
4. Calculate the dominance frontiers $DF_{G'}$ of the nodes of G'.
5. The CDG has edge $x \rightarrow y$ whenever $x \in DF_{G'}[y]$.

That is, x directly controls whether y executes, if and only if x is in the dominance frontier of y in the reverse control-flow graph.

Figure 19.15 shows the CDG for the program of Figure 19.4.

With the SSA graph and the control-dependence graph, we can now answer questions of the form, "must A be executed before B?" If there is any path $A \rightarrow B$ composed of SSA use-def edges and CDG edges, then there is a trail of data- and control-dependence requiring A to be performed before B.

AGGRESSIVE DEAD-CODE ELIMINATION

One interesting use of the control-dependence graph is in dead-code elimination. Suppose we have a situation such as the one in Figure 19.13d, where conventional dead-code analysis (as described in Section 17.3 or Algorithm 19.12) determines:

- k_2 is live because it's used in the definition of k_3,
- k_3 is live because it's used in the definition of k_2,

but neither variable contributes anything toward the eventual result of the calculation.

Just as conditional constant propagation assumes a block is unreachable unless there is evidence that execution can be reach it, *aggressive dead-code elimination* assumes a statement is dead unless it has evidence that it contributes to the eventual result of the program.

Algorithm. Mark *live* any statement that:

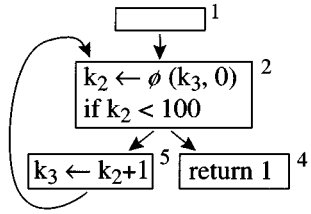
1. Performs input/output, stores into memory, returns from the function, or calls another function that might have side effects;
2. Defines some variable v that is used by another live statement; or
3. Is a conditional branch, upon which some other live statement is control-dependent.

Then delete all unmarked statements.

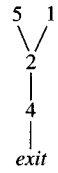
This can be solved by iteration (or by a work-list algorithm). Figure 19.16 shows the amusing result of running the this algorithm on the program of Figure 19.13d: the entire loop is deleted, leaving a very efficient program!

Caveat. The aggressive dead-code elimination algorithm will remove output-free infinite loops, which does change the meaning of the program. Instead of producing nothing, the program will execute the statements after the loop, which may produce output. In many environments this is regarded as unacceptable.

But on the other hand, the control-dependence graph is often used in parallelizing compilers: any two statements that are not control-dependent or data-dependent can be executed in parallel. Even if such a compiler did not delete a useless infinite loop, it might choose to execute the loop in parallel with successor statements (that are not control-dependent on it); this would have approximately the same effect as deleting the infinite loop.



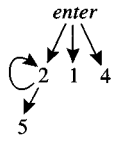
(a) SSA program



(b) Postdominators

n	$DF_{G'}(n)$
1	{}
2	{2}
4	{}
5	{2}

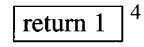
(c) Postdominance frontiers



(d) Control-dependence graph

Block 4 returns, so it is live; no live block is control-dependent on 2, and no live assignment is data-dependent on k_2 or k_3, so nothing else is live.

(e) Finding live statements



(f) After deletion of dead statements

FIGURE 19.16. Aggressive dead-code elimination

19.6 CONVERTING BACK FROM SSA FORM

After program transformations and optimization, a program in static single-assignment form must be translated into some executable representation without ϕ-functions. The definition $y \leftarrow \phi(x_1, x_2, x_3)$ can be translated as "move $y \leftarrow x_1$ if arriving along predecessor edge 1, move $y \leftarrow x_2$ if arriving along predecessor edge 2, and move $y \leftarrow x_3$ if arriving along predecessor edge 3." To "implement" this definition in an edge-split SSA form, for each i we insert the move $y \leftarrow x_i$ at the end of the ith predecessor of the block containing the ϕ-function.

The *unique successor or predecessor* property prevents redundant moves from being inserted; in Figure 19.2b (without the property), block 2 would need a move $a_3 \leftarrow a_1$ that is redundant if the *then* branch is taken; but in Figure 19.2c, the move $a_3 \leftarrow a_1$ would be in block 5, and never executed redundantly.

Now we can do register allocation on this program, as described in Chapter 11. Although it is tempting simply to assign x_1 and x_2 the same register if they were derived from the same variable x in the original program, it could be that program transformations on the SSA form have made their live ranges interfere (see Exercise 19.11). Thus, we ignore the original derivation of the

```
LivenessAnalysis() =
  for each variable v
      M ← {}
      for each site-of-use s of v
          if s is a φ-function with
                v as its ith argument
            let p be the ith predecessor of
                  the block containing s
            LiveOutAtBlock(p, v)
          else LiveInAtStatement(s, v)

LiveOutAtBlock(n, v) =
  v is live-out at n
  if n ∉ M
    M ← M ∪ {n}
    let s be the last statement in n
    LiveOutAtStatement(s, v)

LiveInAtStatement(s, v) =
  v is live-in at s
  if s is the first statement of some block n
    v is live-in at n
    for each predecessor p of n
        LiveOutAtBlock(p, v)
  else
    let s' be the statement preceding s
    LiveOutAtStatement(s', v)

LiveOutAtStatement(s, v) =
  v is live-out at s
  let W be the set of variables that s defines
  for each variable w ∈ (W − {v})
      add (v, w) to interference graph
  if v ∉ W
      LiveInAtStatement(s, v)
```

ALGORITHM 19.17. Calculation of live ranges in SSA form, and building the interference graph. The graph-walking algorithm is expressed as a mutual recursion between *LiveOutAtBlock*, *LiveInAtStatement*, and *LiveOutAtStatement*. The recursion is bounded whenever *LiveOutAtBlock* finds an already walked block, or whenever *LiveOutAtStatement* reaches the definition of v.

different SSA variables, and we rely on coalescing (copy propagation) in the register allocator to eliminate almost all of the move instructions.

LIVENESS ANALYSIS FOR SSA

We can efficiently construct the interference graph of an SSA program, just prior to converting the ϕ-functions to move instructions. For each variable v, Algorithm 19.17 walks backward from each use, stopping when it reaches v's definition. The *dominance property of SSA form* ensures that the algorithm will always stay in the region dominated by the definition of v. For many variables this region is small; contrast this with the situation in Figure 19.14 (a non-SSA program), where the algorithm applied to variable x_1 would walk upwards through the $1 \rightarrow 3$ edge and traverse the entire program. Because this algorithm processes only the blocks where v is live, its running time

is proportional to the size of the interference graph that it constructs (see Exercise 19.12).

Algorithm 19.17 as shown uses recursion (when *LiveInAtStatement* calls *LiveOutAtBlock*), and also *tail recursion* (when *LiveInAtStatement* calls *LiveOutAtStatement*, when *LiveOutAtStatement* calls *LiveInAtStatement*, and when *LiveOutAtBlock* calls *LiveOutAtStatement*). Some programming languages or compilers can compile tail recursion very efficiently as a *goto* – see Section 15.6. But when implementing this algorithm in compilers that do not support efficient tail calls, then instead of tail recursion it might be best to use explicit goto's, or use work-lists for *LiveOutAtStatement* and *LiveInAtStatement*.

19.7 A FUNCTIONAL INTERMEDIATE FORM

A *functional* programming language is one in which (as discussed in Chapter 15) execution proceeds by binding variables to values, and never modifying a variable once it is initialized. This permits equational reasoning, which is useful to the programmer.

But equational reasoning is even *more* useful to the compiler – many compiler optimizations involve the rewriting of a slow program into an equivalent faster program. When the compiler doesn't have to worry about x's value *now* versus x's value *later*, then these transformations are easier to express.

This single-assignment property is at the heart of both functional programming and SSA form. There is a close relationship between the functional intermediate representations used by functional-language compilers and the SSA form used by imperative-language compilers.

Figure 19.18 shows the abstract syntax of the kind of intermediate representation used in modern functional-language compilers. It aspires to the best qualities of quadruples, SSA form, and lambda-calculus. As in quadruple notation, expressions are broken down into primitive operations whose order of evaluation is specified, every intermediate result is an explicitly named temporary, and every argument of an operator or function is an *atom* (variable or constant). As in SSA form and lambda-calculus, every variable has a single assignment (or *binding*), and every use of the variable is within the *scope* of the binding. As in lambda-calculus, scope is a simple syntactic notion, not requiring calculation of dominators.

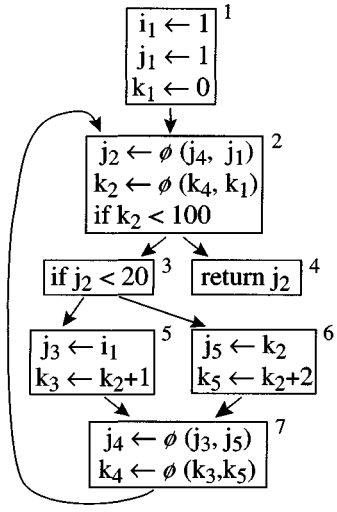
atom	$\rightarrow c$	Constant integer
atom	$\rightarrow s$	Constant string pointer
atom	$\rightarrow v$	Variable
exp	$\rightarrow$ **let** *fundefs* **in** *exp*	Function declaration
exp	$\rightarrow$ **let** $\underline{v} = atom$ **in** *exp*	Copy
exp	$\rightarrow$ **let** $\underline{v} = binop(atom, atom)$ **in** *exp*	Arithmetic operator
exp	$\rightarrow$ **let** $\underline{v} = M[atom]$ **in** *exp*	Fetch from memory
exp	$\rightarrow M[atom]{:=}atom;\ exp$	Store to memory
exp	$\rightarrow$ **if** *atom relop atom* **then** *exp* **else** *exp*	Conditional branch
exp	$\rightarrow atom(args)$	Tail call
exp	$\rightarrow$ **let** $\underline{v} = atom(args)$ **in** *exp*	Non-tail call
exp	$\rightarrow$ **return** *atom*	Return
args	$\rightarrow$	
args	$\rightarrow$ *atom args*	
fundefs	$\rightarrow$	
fundefs	$\rightarrow$ *fundefs* **function** $\underline{v}(formals) = exp$	
formals	$\rightarrow$	
formals	$\rightarrow \underline{v}$ *formals*	
binop	$\rightarrow$ **plus** | **minus** | **mul** | ...	
relop	$\rightarrow$ **eq** | **ne** | **lt** | ...	

FIGURE 19.18. Functional intermediate representation. Binding occurrences of variables are underlined.

Scope. No variable name can be used in more than one binding. Every binding of a variable has a scope within which all the uses of that variable must occur. For a variable bound by **let** $v = \ldots$ **in** *exp*, the scope of v is just the *exp*. The scope of a function-variable f_i bound in

let function $f_1(\ldots) = exp_1$
$\vdots$
function $f_k(\ldots) = exp_k$
in *exp*

includes *all* the exp_j (to allow for mutually recursive functions) as well as the *exp*. For a variable bound as the formal parameter of a function, the scope is the body of that function.




```
let i1 = 1 in
 let j1 = 1 in
  let k1 = 0 in
   let function f2(j2, k2)=
          if k2 < 100 then
           let function f7(j4, k4) =
                 f2(j4, k4)
            in if j2 < 20 then
                 let j3 = i1 in
                  let k3 = k2 + 1 in
                   f7(j3, k3)
               else
                 let j5 = k2 in
                  let k5 = k2 + 2 in
                   f7(j5, k5)
          else return j2
    in f2(j1, k1)
```

PROGRAM 19.19. SSA program of Figure 19.4g converted to functional intermediate form.

These scope rules make many optimizations easy to reason about; we will take *inline expansion of functions* as an example. As discussed in Section 15.4, when we have a definition $f(x) = E$ and a use $f(z)$ we can replace the use by a copy of E but with all the x's replaced by z's. In the `Tree` language of Chapter 7 this is difficult to express because there are no functions; in the functional notation of Chapter 15 the substitution can get complicated if z is a non-atomic expression (as shown in Algorithm 15.8b). But in the functional intermediate form of Figure 19.18, where all actual parameters are atomic, inline-expansion becomes very simple, as shown in Algorithm 15.8a.

Translating SSA into functional form. Any SSA program can be translated into this functional form, as shown in Algorithm 19.20. Each control-flow node with more than one predecessor becomes a function. The arguments of that function are precisely the variables for which there are ϕ-functions at the node. If node f dominates node g, then the function for g will be nested inside the body of the function for f. Instead of jumping to a node, a control-flow edge into a ϕ-containing node is represented by a function call. Program 19.19 shows how a translated program looks.

Translate(*node*) =
  let C be the children of *node* in the dominator tree
  let $p_1, \ldots, p_n$ be the nodes of C that have more than one predecessor
  for $i \leftarrow 1$ to n
    let $a_1, \ldots, a_k$ be the targets of ϕ functions in p_i (possibly $k = 0$)
    let $S_i =$ Translate(p_i)
    let $F_i =$ "`function` $f_{p_i}(a_1, \ldots, a_k) = S_i$"
  let $F = F_1 F_2 \cdots F_n$
  return Statements(*node*, 1, F)

Statements(*node*, j, F) =
if there are $< j$ statements in *node*
  then let s be the successor of *node*
    if s has only one predecessor
    then return Statements(s, 1, F)
    else s has m predecessors
      suppose *node* is the ith predecessor of s
      suppose the ϕ-functions in s are $a_1 \leftarrow \phi(a_{11}, \ldots, a_{1m}), \ \ldots$
                                $a_k \leftarrow \phi(a_{k1}, \ldots, a_{km})$
      return "`let` F `in` $f_s(a_{1i}, \ldots, a_{ki})$"
else if the jth statement of *node* is a ϕ-function
  then return Statements(*node*, $j + 1$, F)
else if the jth statement of *node* is "return a"
  then return "`let` F `in return` a"
else if the jth statement of *node* is $a \leftarrow b \oplus c$
  then let $S =$ Statements(*node*, $j + 1$, F)
    return "`let` $a = b \oplus c$ `in` S"
else if the jth statement of *node* is "if $a < b$ goto s_1 else s_2"
  then (in edge-split SSA form) s_1 has only one predecessor, as does s_2
    let $S_1 =$ Translate(s_1)
    let $S_2 =$ Translate(s_2)
    return "`let` F `in if` $a < b$ `then` S_1 `else` S_2"

The cases for $a \leftarrow b$, $a \leftarrow M[b]$, and $M[a] \leftarrow b$ are similar.

ALGORITHM 19.20. Translating SSA to functional intermediate form.

Translating functional programs into functional intermediate form. A functional program in a language such as PureFun-Tiger starts in a form that obeys all the scope rules, but arguments are not atomic and variables are not unique. It is a simple matter to introduce well-scoped intermediate temporaries by a recursive walk of expression trees; dominator and SSA calculations are unnecessary.

All of the SSA-based optimization algorithms work equally well on a functional intermediate form; so will the optimizations and transformations on functional program described in Chapter 15. Functional intermediate forms can also be made explicitly typed, type-checkable, and polymorphic as described in Chapter 16. All in all, this kind of intermediate representation has much to recommend it.

FURTHER READING

The IBM *Fortran H* compiler used dominators to identify loops in control-flow graphs of basic blocks of machine instructions [Lowry and Medlock 1969]. Lengauer and Tarjan [1979] developed the near-linear-time algorithm for finding dominators in a directed graph, and prove the related theorems mentioned in this chapter. It is common to use this algorithm while mentioning the existence [Harel 1985] of a more complicated linear-time algorithm. Finding the "best" node above a given spanning-forest node is an example of a *union-find* problem; analyses of balanced path-compression algorithms for union-find (such as the "sophisticated" version of the Lengauer-Tarjan algorithm) can be found in many algorithms textbooks (e.g. Sections 22.3–22.4 of Cormen et al. [1990]).

Static single-assignment form was developed by Wegman, Zadeck, Alpern, and Rosen [Alpern et al. 1988; Rosen et al. 1988] for efficient computation of dataflow problems such as global value numbering, congruence of variables, aggressive dead-code removal, and constant propagation with conditional branches [Wegman and Zadeck 1991]. Control-dependence was formalized by Ferrante et al. [1987] for use in an optimizing compiler for vector parallel machines. Cytron et al. [1991] describe the efficient computation of SSA and control-dependence graphs using dominance frontiers and prove several of the theorems mentioned in this chapter.

Wolfe [1996] describes several optimization algorithms on SSA (which he

calls *factored use-def chains*), including induction-variable analysis.

It is useful to perform several transformations on the flow graph *before* conversion to SSA form. These include the conversion of while-loops to repeat-loops (Section 18.2); and the insertion of loop *preheader* nodes (see page 410), *postbody* nodes [Wolfe 1996] (Exercise 18.6), and *landing pads* for loop-exit edges [Rosen et al. 1988] (edge splitting effectively accomplishes the insertion of landing pads). Such transformations provide locations into which statements (such as loop-invariant computations or common subexpressions) may be placed.

Varieties of functional intermediate representations. Functional intermediate forms are all based on lambda-calculus, more or less, but they differ in three important respects:

1. Some are *strict* and some are *lazy* (see Chapter 15).
2. Some have arbitrary nesting of subexpressions; some have *atomic arguments*; and some have *atomic arguments* +λ meaning that all arguments except anonymous functions are atomic.
3. Some permit non-tail calls *(direct style)* and some support only tail calls *(continuation-passing style)*.

Distinction (1) ceases to matter in continuation-passing style.

The design-space of these options has been well explored, as this table shows:

	Direct style		Continuation-passing
	Strict	Lazy	
Arbitrarily nested sub-expressions	Cardelli [1984], Cousineau et al. [1985]	Augustsson [1984]	
Atomic arguments + λ	Flanagan et al. [1993]		Steele [1978], Kranz et al. [1986]
Atomic arguments	Tarditi [1997]	Peyton Jones [1992]	Appel [1992]

The *functional intermediate form* shown in Figure 19.18 fits in the lower left-hand corner, along with Tarditi [1997]. Kelsey [1995] shows how to convert between SSA and continuation-passing style.

EXERCISES

19.1 Write an algorithm, using depth-first search, to number the nodes of a tree in depth-first order *and* to annotate each node with the number of its highest-numbered descendent. Show how these annotations can be used – once your preprocessing algorithm has been run on a dominator tree – to answer a query of the form "does node i dominate node j?" in constant time.

19.2 Use Algorithm 19.9 to calculate the dominators of the flow graph of Exercise 18.1, showing the semidominators and spanning forest at various stages.

19.3 For each of the graphs of Figure 18.1 and Figure 18.2, calculate the immediate dominator tree (using either Algorithm 19.9 or the algorithm in Section 18.1), and for each node n calculate $DF_{\text{local}}[n]$, $DF_{\text{up}}[n]$, and DF.

***19.4** Prove that, for any node v, Algorithm 19.9+19.10b always initializes $best[v] \leftarrow v$ (in the Link function) before calling AncestorWithLowestSemi(v).

19.5 Calculate the dominance frontier of each node in each of these graphs:

a. The graph of Figure 2.8.

b. The graph of Exercise 2.3a.

c. The graph of Exercise 2.5a.

d. The graph of Figure 3.27.

****19.6** Prove that

$$DF[n] = DF_{\text{local}}[n] \cup \bigcup_{Z \in children[n]} DF_{\text{up}}[Z]$$

as follows:

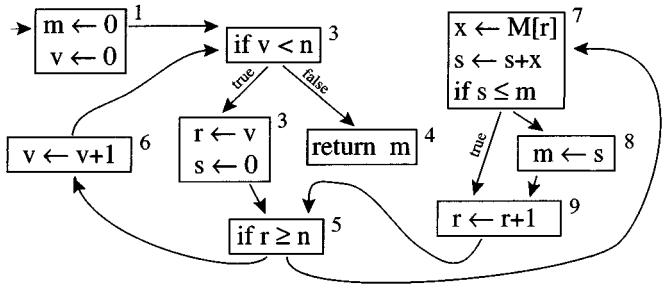
a. Show that $DF_{\text{local}}[n] \subseteq DF[n]$;

b. Show that for each child Z of n, $DF_{\text{up}}[Z] \subseteq DF[n]$;

c. If there is a node Y in $DF[n]$, then therefore there is an edge $U \rightarrow Y$ such that n dominates U but does not strictly dominate Y. Show that if $Y = n$ then $Y \in DF_{\text{local}}[n]$, and if $Y \neq n$ then $Y \in DF_{\text{up}}[Z]$ for some child Z of N.

d. Combine these lemmas into a proof of the theorem.

19.7 Convert this program to SSA form:



Show your work after each stage:

a. Add a start node containing initializations of all variables.

b. Draw the dominator tree.

c. Calculate dominance frontiers.

d. Insert ϕ-functions.

e. Add subscripts to variables.

f. Use Algorithm 19.17 to build the interference graph.

g. Convert back from SSA form by inserting *move* instructions in place of ϕ-functions.

19.8 This C (or Java) program illustrates an important difference between def-use chains and SSA form:

```
int f(int i, int j) {
  int x,y;
  switch(i) {
    case 0:  x=3;
    case 1:  x=1;
    case 2:  x=4;
    case 3:  x=1;
    case 4:  x=5;
    default: x=9;
  }
  switch(j) {
    case 0:  y=x+2;
    case 1:  y=x+7;
    case 2:  y=x+1;
    case 3:  y=x+8;
    case 4:  y=x+2;
    default: y=x+8;
  return y;
}
```

a. Draw the control-flow graph of this program.

b. Draw the use-def and def-use data structures of the program: for each definition site, draw a linked-list data structure pointing to each use-site, and vice versa.

c. Starting from the CFG of part (a), convert the program to SSA form. Draw data structures representing the uses, defs, and ϕ-functions, as described at the beginning of Section 19.3.

d. Count the total number of data-structure nodes in the use-def data, and the total number in the SSA data structure. Compare.

e. Approximate the total sizes of the use-def data structures, and the SSA data structures, if there were N cases in each switch instead of 6.

***19.9** Suppose the graph of Exercise 2.3a is the control-flow graph of a program, and in block 1 there is an assigment to a variable v.

a. Convert the graph to SSA form (insert ϕ-functions for v).

b. Show that for any N, there is a "ladder" CFG with $O(N)$ blocks, $O(N)$ edges, and $O(N)$ assignment statements (all in the first block!), such that the number of ϕ-functions in the SSA form is N^2.

c. Write a program whose CFG looks like this.

d. Show that a program containing deeply nested **repeat-until** loops can have the same N^2 blowup of ϕ-functions.

***19.10** Algorithm 19.7 uses a stack for each variable, to remember the current active definition of the variable. This is equivalent to using environments to process nested scopes, as Chapter 5 explained for type-checking.

a. Rewrite Algorithm 19.7, calling upon the functional-style environments of the `Symbol` module (whose interface is given in Program 5.5) instead of using explicit stacks.

b. Rewrite Algorithm 19.7, using the imperative-style symbol tables whose `SYMBOL` signature is described on page 111.

19.11 Show that optimization on an SSA program can cause two SSA variables a_1 and a_2, derived from the same variable a in the original program, to have overlapping live ranges as described on page 456. **Hint:** Convert this program to SSA, and then do exactly one constant-propagation optimization.

```
while  c<0   do (b := a;  a := M[x]; c := a+b);
return a;
```

***19.12** Let V_c and E_c be the nodes and edges of the CFG, and V_i and E_i be the nodes and edges of the interference graph produced by Algorithm 19.17. Let $N = |V_c| + |E_c| + |V_i| + |E_i|$.

a. Show that the run time of Algorithm 19.17 on the following (weird) program is asymptotically proportional to $N^{1.5}$:

$$
\begin{array}{ll}
 & v_1 \leftarrow 0 \\
 & v_2 \leftarrow 0 \\
 & \vdots \\
 & v_m \leftarrow 0 \\
 & \text{goto } L_1 \\
L_1: & \text{goto } L_2 \\
L_2: & \text{goto } L_3 \\
 & \vdots \\
L_{m^2}: & \\
 & w_1 \leftarrow v_1 \\
 & w_2 \leftarrow v_2 \\
 & \vdots \\
 & w_m \leftarrow v_m
\end{array}
$$

*b. Show that if every block defines at least one variable, and has no more than c statements and no more than c out-edges (for some constant c), then the time complexity of Algorithm 19.17 is $O(N)$. **Hint:** Whenever LiveOutAtBlock is called, there will be at most c calls to LiveOutAtStatement, and at least one will add an edge to the interference graph.