10.2 ELIMINATING USELESS AND UNREACHABLE CODE

Sometimes, programs contain computations that have no externally visible effect. If the compiler can determine that a given operation does not affect the program's results, it can eliminate the operation. Most programmers do not write such code intentionally. However, it arises in most programs as the direct result of optimization in the compiler and often from macro expansion or naive translation in the compiler's front end.

Useless
An operation is *useless* if no operation uses its result, or if all uses of the result are, themselves dead.

Unreachable
An operation is *unreachable* if no valid control-flow path contains the operation.

Two distinct effects can make an operation eligible for removal. The operation can be *useless*, meaning that its result has no externally visible effect. Alternatively, the operation can be *unreachable*, meaning that it cannot execute. If an operation falls into either category, it can be eliminated. The term *dead code* is often used to mean either useless or unreachable code; we use the term to mean useless.

Removing useless or unreachable code shrinks the IR form of the code, which leads to a smaller executable program, faster compilation, and, often, to faster execution. It may also increase the compiler's ability to improve the code. For example, unreachable code may have effects that show up in the results of static analysis and prevent the application of some transformations. In this case, removing the unreachable block may change the analysis results and allow further transformations (see, for example, sparse conditional constant propagation, or SCCP, in Section 10.7.1).

Some forms of redundancy elimination also remove useless code. For instance, local value numbering applies algebraic identities to simplify the code. Examples include `x + 0` ⇒ `x`, `y x 1` ⇒ `y`, and `max(z,z)` ⇒ `z`. Each of these simplifications eliminates a useless operation—by definition, an operation that, when removed, makes no difference in the program's externally visible behavior.

Because the algorithms in this section modify the program's control-flow graph (CFG), we carefully distinguish between the terms *branch*, as in an ILOC `cbr`, and *jump*, as in an ILOC `jump`. Close attention to this distinction will help the reader understand the algorithms.

10.2.1 Eliminating Useless Code

The classic algorithms for eliminating useless code operate in a manner similar to mark-sweep garbage collectors with the IR code as data (see Section 6.6.2). Like mark-sweep collectors, they perform two passes over the code. The first pass starts by clearing all the mark fields and marking "critical" operations as "useful." An operation is *critical* if it sets return values for the procedure, it is an input/output statement, or it affects the value in

An operation can set a return value in several ways, including assignment to a call-by-reference parameter or a global variable, assignment through an ambiguous pointer, or passing a return value via a return statement.

```
Mark( )
 WorkList ← ∅
 for each operation i
     clear i's mark
     if i is critical then
        mark i
        WorkList ← WorkList ∪ {i}
 while (WorkList ≠ ∅)
     remove i from WorkList
        (assume i is x ← y op z)
     if def(y) is not marked then
        mark def(y)
        WorkList ← WorkList ∪ {def(y)}
     if def(z) is not marked then
        mark def(z)
        WorkList ← WorkList ∪ {def(z)}
     for each block b ∈ RDF(block(i))
        let j be the branch that ends b
        if j is unmarked then
           mark j
           WorkList ← WorkList ∪ {j}
```

(a) The *Mark* Routine

```
Sweep( )
 for each operation i
     if i is unmarked then
        if i is a branch then
           rewrite i with a jump
             to i's nearest marked
             postdominator
        if i is not a jump then
           delete i
```

(b) The *Sweep* Routine

■ **FIGURE 10.1** Useless Code Elimination.

a storage location that may be accessible from outside the current procedure. Examples of critical operations include a procedure's prologue and epilogue code and the precall and postreturn sequences at calls. Next, the algorithm traces the operands of useful operations back to their definitions and marks those operations as useful. This process continues, in a simple worklist iterative scheme, until no more operations can be marked as useful. The second pass walks the code and removes any operation not marked as useful.

Figure 10.1 makes these ideas concrete. The algorithm, which we call *Dead*, assumes that the code is in SSA form. SSA simplifies the process because each use refers to a single definition. *Dead* consists of two passes. The first, called *Mark*, discovers the set of useful operations. The second, called *Sweep*, removes useless operations. *Mark* relies on reverse dominance frontiers, which derive from the dominance frontiers used in the SSA construction (see Section 9.3.2).

The treatment of operations other than branches or jumps is straightforward. The marking phase determines whether an operation is useful. The sweep phase removes operations that have not been marked as useful.

The treatment of control-flow operations is more complex. Every jump is considered useful. Branches are considered useful only if the execution of a useful operation depends on their presence. As the marking phase discovers useful operations, it also marks the appropriate branches as useful. To map from a marked operation to the branches that it makes useful, the algorithm relies on the notion of control dependence.

Postdominance
In a CFG, *j postdominates i* if and only if every path from *i* to the exit node passes through *j*.

See also the definition of dominance on page 478.

The definition of control dependence relies on *postdominance*. In a CFG, node *j* postdominates node *i* if every path from *i* to the CFG's exit node passes through *j*. Using postdominance, we can define control dependence as follows: in a CFG, node *j* is control-dependent on node *i* if and only if

1. There exists a nonnull path from *i* to *j* such that *j* postdominates every node on the path after *i*. Once execution begins on this path, it must flow through *j* to reach the CFG's exit (from the definition of postdominance).
2. *j* does not strictly postdominate *i*. Another edge leaves *i* and control may flow along a path to a node not on the path to *j*. There must be a path beginning with this edge that leads to the CFG's exit without passing through *j*.

In other words, two or more edges leave block *i*. One or more edges leads to *j* and one or more edges do not. Thus, the decision made at the branch-ending block *i* can determine whether or not *j* executes. If an operation in *j* is useful, then the branch that ends *i* is also useful.

This notion of control dependence is captured precisely by the *reverse dominance frontier* of *j*, denoted RDF(*j*). Reverse dominance frontiers are simply dominance frontiers computed on the reverse CFG. When `Mark` marks an operation in block *b* as useful, it visits every block in *b*'s reverse dominance frontier and marks their block-ending branches as useful. As it marks these branches, it adds them to the worklist. It halts when that worklist is empty.

`Sweep` replaces any unmarked branch with a jump to its first postdominator that contains a marked operation. If the branch is unmarked, then its successors, down to its immediate postdominator, contain no useful operations. (Otherwise, when those operations were marked, the branch would have been marked.) A similar argument applies if the immediate postdominator contains no marked operations. To find the nearest useful postdominator, the algorithm can walk up the postdominator tree until it finds a block that contains a useful operation. Since, by definition, the exit block is useful, this search must terminate.

After `Dead` runs, the code contains no useless computations. It may contain empty blocks, which can be removed by the next algorithm.