CHAPTER 4

Preliminary Transformations

4.1 Introduction

Most of the dependence tests presented Chapter 3 require that subscript expressions be affine functions of loop induction variables, with known constant coefficients and at most a symbolic additive constant. Obviously for these tests to be effective in constructing an accurate dependence graph for a program, all (or at least most) of the subscripts in that program must meet these constraints. Unfortunately, the quirks of many FORTRAN compilers, FORTRAN language versions, and idiosyncratic individuals have encouraged some programming practices that defeat these constraints. The following example is not atypical.

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      KI = KI + INC
```

```
         U(KI) = U(KI) + W(J)
      ENDDO
      S(I)= U(KI)
   ENDDO
```

Few subscripts in this loop nest are affine functions of the loop induction variables. In particular, the expression `U(KI)` (which is involved in every dependence test) is not a direct affine function of the loop induction variables; `KI` varies within the loop in a manner that makes the subscript unamenable to dependence testing. If the tests from Chapter 3 are to be applied to this example, the code must be transformed

The first transformation to apply is induction *variable substitution*. Because the variable `INC` is invariant within the inner loop, the assignment to `KI` within the `J` loop increments its value by a constant amount on each loop iteration. This increment makes `KI` an *auxiliary induction variable*; it is essentially another loop variable that tracks along the regular loop index (`J`), but with a different increment or starting point. Induction variable substitution transforms this auxiliary increment into a direct function of the loop induction variable. Applying this transformation to the inner loop yields.

```
   INC = 2
   KI = 0
   DO I = 1, 100
      DO J = 1, 100
         CONTINUE
         U(KI + J*INC) = U(KI + J*INC) + W(J)
      ENDDO
S1    KI = KI + 100 * INC
      S(I)= U(KI)
   ENDDO
```

The uses of `KI` within the loop have been changed to a function of the loop induction variable; the increment has been deleted (changed to a `CONTINUE` statement); and S_1 has been inserted to set `KI` to the correct value outside of the loop. Looking at S_1 in the context of the outer loop, it too makes `KI` an auxiliary induction variable, which may be removed by a repetition of induction variable substitution:

```
   INC = 2
   KI = 0
   DO I = 1, 100
      DO J = 1, 100
```

```
      CONTINUE
      U(KI + J*INC) = U(KI + J*INC) + W(J)
   ENDDO
   CONTINUE
   S(I)= U(KI + I * (100*INC))
ENDDO
KI = KI + 100*100*INC
```

Now all subscripts are affine functions of loop induction variables, although the coefficients are all symbolic.

The next step in simplifying this program is to recognize that the values of `INC` and `KI` used within the loops are actually just constant values. *Constant propagation* of the values from outside the loop will eliminate these symbolic quantities, yielding

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(0 + J*2) = U(0 + J*2) + W(J)
   ENDDO
   CONTINUE
   S(I)= U(0 + I * (100*2))
ENDDO
KI =0 + 100*100*2
```

After simplification, this becomes

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(J*2) = U(J*2) + W(J)
   ENDDO
   CONTINUE
   S(I)= U(I*200)
ENDDO
KI =20000
```

The subscripts are now all in a form that can be handled by dependence tests.

Although this example is now in a form amenable to dependence testing, it has some useless redundancy in it. In particular, many of the constant valued assignments are never used (the first assignment to KI is obviously not used; the remaining assignment and that to INC could possibly be used later in the program, but we will assume that they are not, which would be the case in most programs). Rather than wasting code space and execution time for these assignments, they can be eliminated via *dead code elimination*. This yields

```
DO I = 1, 100
   DO J = 1, 100
      CONTINUE
      U(J*2) = U(J*2) + W(J)
   ENDDO
   CONTINUE
   S(I)= U(I*200)
ENDDO
```

This loop fragment is now an efficient program that is amenable to dependence testing.

Since in practice many programs are similar to our first fragment, the ability to transform code is important for dependence testing to be successful. This chapter presents methods for performing preliminary transformations on programs to make them fit the requirements of dependence tests. These transformations become much more difficult in the presence of control flow, but this topic will be delayed until the next chapter.

4.2 Necessary Information

When discussing methods for dependence testing, Chapter 3 assumed that a number of items regarding loops be true. For instance, most of the dependence tests are stated assuming that loops have a step of 1. In order to implement these tests, it is necessary to gather information regarding which loops meet these requirements, and to do something about those that do not. Furthermore, implementing the transformations overviewed in the previous section requires knowledge about the structure and use of data within a program. The information which is necessary includes:

1. Knowledge that loops have strides of 1. Dependence tests as described so far assume that all loop steps are 1. This assumption is not a hard restriction as the tests can easily be recast to accommodate any loop

stride, but dependence (and other transformations to be described later) is much simpler in the presence of unitary strides.

2. Knowledge of loop invariant quantities. Auxiliary induction variables are characterized by repetitively incrementing a variable by a loop invariant quantity. Any compiler that cannot determine loop invariant expressions cannot fully perform induction variable substitution.
3. Knowledge of constant-valued assignments. Constant propagation involves recognizing variables that have constant values over a region; since those variables will be generated by constant-valued assignments, recognition of constant-valued assignments is essential.
4. Knowledge of which variables are used in which statements. Propagating a constant-valued assignment requires knowing which statements uses the variable defined by that assignment. Similarly, dead code recognition involves identifying statements whose output is never used.

Adequately preparing a program for dependence testing requires answering all the questions listed above. These questions all fall into the realm of scalar optimization, and as such, have been studied extensively. In particular, the last three are usually answered by use of *dataflow analysis*. The remainder of this chapter introduces the theory necessary to answer all of these questions. The next section deals with the first question; following sections develop the foundations for dataflow analysis. Finally, applications of dataflow analysis are presented to the specific transformations of induction variable substitution, constant propagation, and dead code elimination.

4.3 Loop Normalization

As stated previously, while the dependence tests presented in Chapter 3 can be adapted to work on a loop with an arbitrary stride, they are simpler (and easier to understand) when presented to work on loops with unit stride. Because of this simplicity, many advanced compilers have normalized all loops to run from a lower bound of 1 to some upper bound with a stride of 1—basically making all loops "count off" the iterations they execute and using auxiliary induction variables for the original induction variables. This transformation is called *loop normalization.* As originally defined, the term "loop normalization" applied specifically to this one transformation. However, dependence tests requires lots of information about loops, e.g. which loops surround which statements; which induction variables control which loops; what bounds control each loop.

Since this information must be gathered sometime prior to testing proper, and since loop normalization must examine all loops anyway, many compilers have used loop normalization as a general normalization and information-gathering phase. As a result, the term is often applied not only to the specific transformation, but also to a general phase for gathering information about loops. We will use "loop normalization" interchangeably for either the transformation or the phase; the context should clarify the intended meaning.

Loop normalization (the transformation) provides a number of advantages to a writer of advanced compilers, including

1. Easier dependence testing. Dependence tests are simpler to understand and easier to program when applied to normalized loops.
2. Easier recognition of loop induction variables. Dependence testing requires identification of the coefficients of loop induction variables, which, of course, requires identification of the variables themselves. Since loop normalization replaces all loop indices with variables it generates, it can make those variables easily identifiable.
3. Easier program transformation. Many loop transformations require knowledge of the stride or the iteration count; these transformations are much simpler to program when every loop is known to have a unit stride and an iteration count equal to its upper bound. Induction variable substitution is one example. Replacing the induction inside the loop requires knowledge of which iteration number (not necessarily the loop index proper) is executing; moving the induction outside the loop requires knowledge of the number of times the loop is iterated.

However, loop normalization also possesses some severe disadvantages as well. The most prominent one is the possibility of distorting the properties of some dependences. The following fragment[1] presents one example of the type of distortion possible:

```
    DO I = 1, M
       DO J = I, N
S1        A(J,I) = A(J,I-1) + 5
       ENDDO
    ENDDO
```

1. This example is taken from Michael Wolfe's dissertation.

This is not exactly Wolfe's example (my copy of his thesis is still packed away), but it shows the point. If you have a copy easily available, you should double check this.

The self true dependence of S_1 has a direction vector of (<,=). When the inner loop is normalized, the following results:

```
      DO I = 1, M
         DO J = 1, N-I+1
S1          A(J+I-1,I) = A(J+I-1,I-1) + 5
         ENDDO
      ENDDO
```

Since this is a safe reordering transformation, the dependence must still exist. However, the change in iteration space has transformed the direction vector from (<,=) into (<,>). This transformation is not completely harmless. Consider, for instance, attempting to interchange the `I` and `J` loops (loop interchange is discussed fully in Section 5.1). The effect of loop interchange on direction vectors is to swap the appropriate entries. As a result, loop interchange applied to the original direction vector of (<,=) yields a new direction vector of (=,<), which preserves the dependence and is obviously safe. Loop interchange applied to the normalized direction vector of (<,>) yields a new direction vector of (>,<), which reverses the dependence, and is obviously *not* safe. Loop normalization has skewed the dependence to invalidate loop interchange.[1]

Because loop normalization can be a valuable transformation for certain architectures, the following is a simple algorithm for loop normalization which can be applied to any `FORTRAN DO`-loop that has integer control parameters. The algorithm requires three steps:

1. Replace the loop header
   ```
   DO I = L, U, S
   ```
   with the adjusted loop header
   ```
   DO i = 1, (U-L+S)/S
   ```
   where `i` is the new loop induction variable. It is useful for debugging and later transformations if `i` is named so as to permit easy identification with the loop (e.g. an artificial name that contains the nesting level of the loop).
2. Replace each reference to `I` within the loop by
   ```
   i*S - S + L
   ```

1. In all fairness, loop interchange in the original example is a very complicated process due to the dependence of the bounds of the inner loop on the outer loop.

This expression, which is equivalent to `L + (i-1)*S` with the subtraction distributed over the multiplication, is a simple form that is easy for the dependence tests to digest later.

3. Following the loop insert a finalization assignment to the old loop induction variable

```
I = i*S - S + L
```

The purpose of this assignment is to ensure that uses of the original loop index that occur outside the loop receive the correct value.

4.4 Data Flow Analysis

4.4.1 Definition-Use Chains

One ubiquitous need throughout all the described preliminary transformations was that of being able to easily get from a definition of a variable to all locations that could consume that defined value. Definition-use chains are a data structure designed precisely to ease that operation:

> **Definition 4.1.** The *definition-use graph* is a graph that contains an edge from each definition point in the program to every possible use of the variable at run time.

We use the term "definition-use graph" instead of the more traditional "definition-use chains" because "graph" more correctly characterizes the nature of the information it contains. The definition-use graph is essentially the scalar version of the true dependences within a program.

Constructing definition-use edges within a single straight-line block[1] of code is very simple. One walks through each statement in order in the basic block, noting the variables defined by each statement (also called its *definitions*) as well as the variables used by each statement (its *uses*). For each use, an edge is added to the use-definition graph for that use back to the last *exposed* definition in the block of that variable—in other words, to every definition that *reaches* the use. Whenever a new definition is encountered for a variable, the new definition *kills* the existing definition, so that later uses are linked only to the new definition, not to the old. At the end of the block is reached, the local graph is complete.

1. Such blocks are also called *basic blocks.* A basic block is a maximal group of statements such that one statement in the group is executed if and only if every statement is executed. In other words, there is no control flow into or out of a basic block except at its very beginning or its very end. For more information, see \cite Kenn80.

In addition to the local graph, the basic block computation also produces a number of useful sets that characterize the behavior of the block. These include:

uses(b): the set of all variables used within the block *b* that have no prior definitions within the block. In other words, these are uses that are not "satisfied" within the block, so they are exposed to any definitions that reach this block from previous blocks.

defines(b): the set of all definitions within block *b* that are not killed within the block. In other words, these are all the definitions that can reach other blocks outside of *b*.

kills(b): the set of all variables killed by definitions within block *b*. Any definitions from other blocks that try to "reach through" *b* will be stopped by the kills set.

These sets provide the basic tools for constructing the definition-use graph for the whole program, rather than individual blocks. Given these local sets, the one missing set necessary for computation of global edges is the set *reaches(b)*—the set of all definitions from all blocks (including *b*) that can possibly reach *b*. For any given block *b*, all global definition-use edges can be obtained by adding an edge for each element in *reaches(b)* to all appropriate elements of *uses(b)*. As a result, if we can find a way to compute *reaches* for all blocks, we can build the definition-use graph.

In order to understand how to compute *reaches* globally, it is first useful to look at the problem on a very limited graph: one basic block *b* and some number of predecessors, each of which can reach *b* via some form of control flow. In this simple graph, *reaches(b)* along any one predecessor *p* is the set of all definitions that reach *p* (*reaches(p)*) and that also are not killed inside *p* (the negation of *kills(p)*), plus those definitions in *p* that reach out directly into *b* (*defines(p)*). Expressed more formally, *reaches(b)* can be defined by the following equation:

$$\text{reaches}(b) = \bigcup_{p \in P(b)} (\text{defines}(p) \cup (\text{reaches}(p) \cap \neg \text{killed}(p))) \quad \textbf{(EQ 4.1)}$$

This equation is obviously simple to solve over the limited graph described above, but is not so simple over general control flow graphs. The complication is that computing *reaches* for one block *b* may immediately change all other *reaches* (including that for *b* itself), since *reaches(b)* is an input into other *reaches* equations. Achieving the correct solution requires simultaneously solving all individual equations.

Fortunately, the underlying mathematics of this problem guarantee that repeatedly iterating the individual solutions will eventually terminate in a stable solution that is exactly that obtained by simultaneously solving all solutions. That proof is beyond the scope of this book[1]. However, the proof does permit some very simple methods for solving this problem. The simplest, which is also asymptotically the slowest, is the iterative method, presented in Figure 4.1. Simply stated, a very simple initial approximation is computed by assuming that all *reaches* in the inputs are the null set, and using local sets for the remainder of Equation 4.1. The method then repeatedly iterates until it reaches a fixed point (a point where each pass through the blocks produce no further changes). That solution is the global solution.

FIGURE 4.1 Iterative Method

```
procedure iterate(g);

    /* Iterate takes as input a control flow graph g comprised of basic */
    /* blocks b. Each block has computed for it the local sets defines */
    /* and kills. */

    foreach b ε g do
        reaches(b) = ϕ;
    od

    changed = true;
    while (changed) do
        changed = false;
        foreach b ε g do
            newreaches = reaches(b);
            foreach p ε predecessors(b) do
                newreaches = newreaches union (defines(p) union
                    (reaches(p) intersect notkilled(p)))
            od
            if newreaches != reaches(b) then
                reaches(b) = newreaches;
                changed = true;
            fi
    od  od
end
```

1. See \cite Kenn80 for more details.

Procedure *iterate* is not the only (or even the fastest, in the worst case) method for building global *reaches* sets, and from there, definition-use chains. However, it is the simplest method, and is extremely efficient in practice. The reader interested in other methods should see \cite Kenn 80.

Once the definition-use graph has been constructed, the problems of induction variable substitution, constant propagation, and dead code elimination can be attacked. In a compiler, those problems are typically attacked in that order. However, in terms of difficulty of understanding, induction variable substitution is the hardest to understand and dead code elimination is the simplest. As a result, the remaining sections discuss these problems in inverse chronological order.

4.4.2 Dead Code Elimination

Dead code is code whose results are never used in any useful statement. "Useful statements", at a first approximation, are simply output statements since those are the only ones that perform any action directly seen by the outside world. Of course, any statements that compute values used by output statements are also useful, as are statements that compute values used by useful statements. This crude definition forms the basis of an algorithm for dead code elimination, based on finding all useful statements and eliminating all others. This algorithm, presented in Figure 4.2, starts with a set *worklist* of all absolutely useful statements (i.e. output statements, control flow statements, and input statements[1]). It then repeatedly adds statements that are necessary to compute the current members of *worklist* until no more statements are necessary. At that point, *worklist* contains all useful statements from the program, and all others may be removed.

FIGURE 4.2 Dead code elimination.

```
let worklist:= {absolutely useful statements};

while worklist ≠ ∅ do
        x:= an arbitrary element of worklist;
        worklist:= worklist – {x};
        mark x useful;
        for all (y,x) ∈ defuse do
```

1. If any input statement is not executed, even if its results are not useful, other input† statements can easily receive the wrong values.

```
            if y is not marked useful then worklist := worklist – {x} fi
        od
od
```

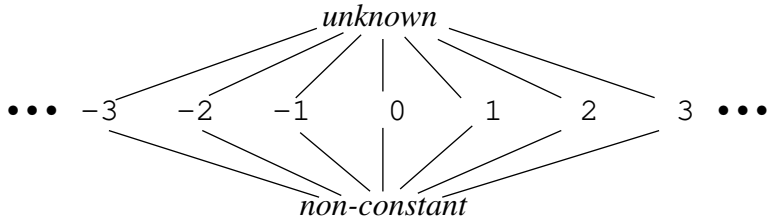
The power of this algorithm gets often displayed on carelessly written user benchmarks. In a benchmark, the programmer usually cares about the time required by the computation, rather than the results of the computation. As a result, users always remember to print out the execution time, but often forget to print out any results. When that happens, the entire computation becomes dead, resulting in extremely fast execution times—the generated code will typically read the clock once, read it again, take the difference, and print it out.

The next section presents a more complicated definition-use based algorithm, constant propagation.

4.4.3 Constant Propagation

Constant propagation attempts to replace all variables that it can prove have constant values at runtime with those constant values. One way of analyzing this problem is in terms of a lattice of constant values depicted in Figure 4.3. The lattice represents the information that can be gathered about variables. At the top level ("unknown"), no information is available regarding a particular variable. The middle level represents a variable having one known constant value—the situation desired by constant propagation. Note that this level is infinitely wide, since any integral constant (assuming we are propagating integers; we can equally well place floating numbers in this level) can appear. The bottom level represents a variable that is known to take on more than one value, or whose constant value cannot be known at compile time. While this lattice has infinite width, it has finite depth, since the longest downward chain has length two.

FIGURE 4.3 Constant propagation lattice.



The basic idea of the constant propagation algorithm is to begin with each variable approximated by the top element of the lattice. Whenever a statement with constant output is found, its output variable value is lowered to the constant value. Then, definition-use edges are employed to locate all instructions that use the output value. At each use, the input variable's approximate value is adjusted by taking the meet of the old and new value in the lattice. The full algorithm is given in Figure 4.4.

FIGURE 4.4 Constant Propagation Algorithm

```
/* valin(w, s) is the current approximate value of input w to s */
/* valout(v, s) is the current value of output v from s         */
/* μ(s)(inputs to s) is the result of symbolic interpretation of */
/*     statement s over the lattice values of its inputs. The   */
/*     output is the lattice value of the output of the statement */

for all statements s in the program do
    for each output v of s do valout(v, s) := unknown od;
    for each input w of s do
        if w is a variable then valin(w, s) := unknown
        else valin(w, s) := the constant value of w fi
    od
od;

worklist := {all statements of constant form, e.g., X = 5};

while worklist ≠ ∅ do
    pick an arbitrary statement x from worklist;
    worklist := worklist – {x};
    let v denote the output variable for x;

    /* Symbolic interpretation of the statement x */
    newval := μ(x)(valin(v, x), for all inputs v to x);
    if newval ≠ valout(v, x) then
    valout(x, v) := newval;
        for all (x, y) ∈ defuse do
            oldval := valin(v, y);
            valin(v, y) := oldval ∧ valout(v, x) ;
            if valin(v, y) ≠ oldval then worklist := worklist ∪ {x} fi
        od
    fi
od
```

Intuitively, constant propagation starts with a set of all assignments that set a variable to be a constant value. One element is selected from that set; definition-use edges are used to find all inputs that that definition can reach. For each of these inputs, the definition-use edges are traced backwards to find all definitions that can reach a specific input. If all definitions have the same constant value, the input is replaced with that value; otherwise, it is not known to be constant. If the input is replaced, that may create a new constant assignment; if it does, the assignment is added to the *worklist* set.

The time required by this constant propagation algorithm is given by the following theorem:

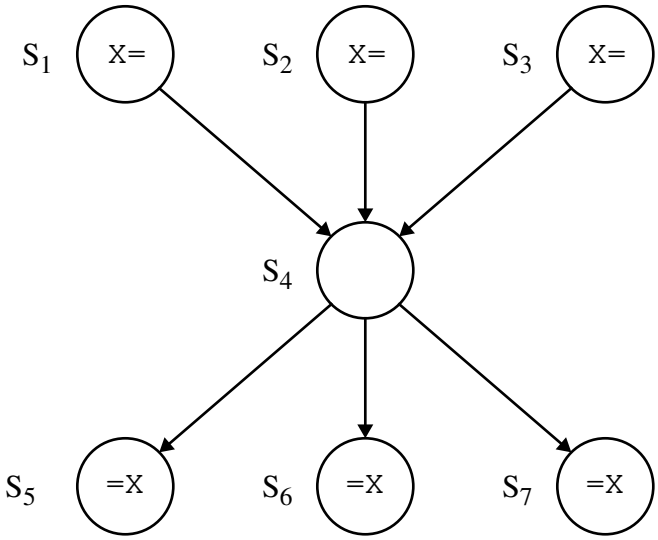
> **Theorem 4.1.** The constant propagation algorithm in Figure 4.4 takes O(N+E) time, where N is the number of statements in the program and E is the number of edges in the definition-use graph.

Proof. Note that a statement can be put on the worklist at most two times because it is only added to the worklist if its output value is lowered in the lattice. Since the longest downward chain has no more than two steps, the output value can only be lowered two times. Hence, the body of the main loop is executed O(N) times. The innermost forall loop iterates over definition-use edges that emanate from a single statement. In the aggregate, the body of this loop is executed at most twice for each edge, since the source of the edge can be taken from the worklist no more than two times. Thus the innermost forall body takes O(E) time and the entire procedure takes O(N+E) time. Q.E.D.

4.4.4 Static Single-Assignment Form

One problem with the constant propagation algorithm in Figure 4.4 is the fact that the number of definition-use edges can grow very large in the presence of control flow. Figure 4.5 presents an example that illustrates this problem. Statements S_1, S_2, and S_3 all define the variable X. These definitions all reach the uses in statements S_5, S_6, and S_7 by passing through statement S_4. Because each definition can reach every use, the number of definition-use edges is proportional to the square of the number of statements. In this particular case, there are nine edges: (S_1,S_5), (S_1,S_6), (S_1,S_7), (S_2,S_5), (S_2,S_6), (S_2,S_7), (S_3,S_5), (S_3,S_6) and (S_3,S_7). Since constant propagation takes time O(N+E) and since E can be proportional to N^2, the time required for the overall algorithm can be quadratic in the number of statements.

FIGURE 4.5 Definition-use example



One way to reduce this number of operations is to put a special pseudo-operation in the node for statement S_4:

```
X = X
```

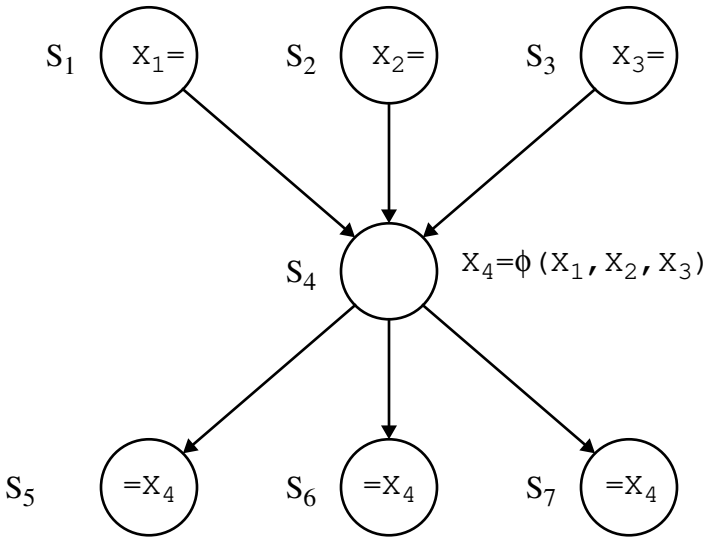
Because this definition kills the values of X that are created in statements S_1, S_2 and S_3, the total number of definition-use edges in the modified program is six: (S_1,S_4), (S_2,S_4), (S_3,S_4), (S_4,S_5), (S_4,S_6) and (S_4,S_6).

This idea can be combined with a method for providing unique names for each scalar variable range to produce *static single-assignment form* (commonly abbreviated SSA), a variation on the definition-use graph with the following properties:

1. each assignment creates a different variable name
2. at points where control flow joins, a special operation is inserted to merge different incarnations of the same variable.

The static single assignment graph for the example in Figure 4.5 is given in Figure 4.6

FIGURE 4.6 Static single-assignment form for Figure 4.5



It is no coincidence that this graph resembles the original control flow graph, since merge nodes are inserted at points where control flow paths merge.

This is a more visible reminder that we should develop the SSA construction algorithm here.

The static single-assignment form representation of the definition-use graph has a number of advantages for analysis, the most important of which is the improved performance of algorithms like constant propagation and the reduced size of the graph.

Here we should develop the algorithm for constructing the SSA graph.

4.5 Induction Variable Substitution

Dead code elimination and constant propagation are both transformations that rely on fairly simple patterns in the definition-use graph. Induction variable substitution is more complicated, and requires recognition of fairly complex patterns. As a result, induction variable substitution typically involves a sophisticated framework for analyzing a program. Given the existence of this framework, as well as the fact that the original need

for induction variable substitution arises from the necessity of transforming common programming practices into a form more amenable for dependence testing, an induction variable substitution phase generally does more than just replace auxiliary induction variables. One common addition is forward substitution of region-invariant expressions into subscripts. The following is an example where this transformation is useful:

```
     DO I = 1, 100
        K = I-2
S1      A(K) = A(K) + 5
     ENDDO
```

This programmer has done some of the work of the compiler by performing common subexpression elimination on his code, compressing the two subscript calculations into a single variable. Unfortunately, a dependence tester will have problems calculating the dependences of `A`, because its variance is hidden inside `K`. Substituting `K` forward, yielding

```
     DO I = 1, 100
        CONTINUE
S1      A(I-2) = A(I-2) + 5
     ENDDO
```

produces a form which dependence testing can easily handle. Furthermore, if this example is being compiled for a vector machine, the generated code will be much more efficient. Assuming the first form can be determined to be vectorizable (it is), it is vectorized by expanding `K` into a temporary vector and using that temporary vector as a scatter-gather index for `A`. The second form is easily vectorized into a direct vector operation.

Performing induction variable substitution and forward expression substitution requires not only definition-use edges but also some control flow analysis. For a statement to define an induction variable, it must be executed on every iteration of the loop. Similarly, before a definition can be forward substituted, it is necessary to guarantee that the definition is always executed on a loop iteration before the statement into which it is substituted. This information can be determined by use of *dominators* \cite something here.

> **Definition 4.2.** Statement S_1 pre-dominates (also referred to as dominates) statement S_2 with respect to statement S_0 if every possible execution path from S_0 to S_2 contains S_1.

Simply stated, Definition 4.2 says that S_1 dominates S_2 if S_1 is always executed before S_2. With respect to induction variable substitution and forward substitution, dominators provide the control flow information necessary to determine the safety of the transformation. An expression can be forward substituted from statement S_1 to statement S_2 if S_1 dominates S_2 with respect to the first statement in the loop (the first statement is executed on every iteration of the loop). Similarly, assuming that loops with exit branches are bypassed and that loops are set up so that the last statement is executed on every iteration (this can be accomplished by inserting a null statement), a statement S can define an induction variable if S dominates the last statement in the loop with respect to the first statement.

With these preliminaries out of the way, it is time to present algorithms for expression folding and induction variable substitution. Figure 4.7 presents an algorithm for forward expression substitution.

FIGURE 4.7

Forward Expression Substitution

```
procedure ForwardSub(S);

    /* Forward Sub determines whether an assignment S is an */
    /* assignment of a loop invariant expression. If it is, it */
    /* forward substitutes S into all statements into which it can*/
    /* safely and profitably be substituted.*/

    /* Determine loop invariance. */
    loop_invariant = true;
    foreach edge e  into S do
        S_s = SourceStmt(e);
        if common_nesting(S,S_s) == nesting_level(S)
            then loop_invariant = false;
    od  fi

    if not loop_invariant or has_side_effects(S)
        then return;
    fi

    /* Perform substitution into uses within the loop. */
    all_uses_gone = true;
    all_loop_uses_gone = true;
    foreach edge e emanating from S do
        S_t = target_stmt(S);
        if common_nesting(S, S_t) == nesting_level(S) then
```

```
            if dominates(S, S_t,, loop_entry) then
                replace target(e) with rhs(S);
                foreach edge in into rhs(S) do
                    add new edge from source_stmt(in) to S_t;
                od
                remove edge e;
            else all_loop_uses_gone = false; fi
        else all_uses_gone = false; fi
    od

    if all_uses_gone and all_loop_uses_gone
        then delete(S);
        else if all_loop_uses_gone
            then move S outside loop;
    fi;  fi
end ForwardSub;
```

When presented with a statement *S* for forward substitution, *ForwardSub* first analyzes all input edges into the statement, to ensure that any definitions come from outside the innermost loop (the one in which substitution is occurring). The check is performed by comparing the common nesting level of *S* and incoming definitions to the nesting level of *S*; if these quantities are equal, then the variable associated with the definition is defined in the innermost loop, and the right hand side of *S* is not loop invariant. Otherwise, assuming the statement has no side effects, it can be substituted within the loop. The substitution is effected by finding all candidates that use the output of *S* (tracing down output edges) that are within the innermost loop (determined by checking nesting level), then replacing. To maintain the correctness of the definition-use graph, new edges have to be added into the substituted expression; these edges are just copies of the edges that come into *S*. Finally, if all uses of the definition of *S* have been replaced, *S* is dead can be deleted. If all uses within the loop have been replaced, then *S* can be moved outside the loop to save execution time.

ForwardSub also requires one property of the definition-use graph which is worth noting. The first loop in the graph traces all edges coming into a statement from outside. The second loop in the graph traces all edges going out of a statement into other statements. Any data structure chosen to implement the definition-use graph must possess this bidirectional property for this algorithm to work.

Having dealt with forward substitution, it is now time to focus on induction variable substitution. Figure 4.8 presents an algorithm for determining whether a statement represents an assignment to an auxiliary induction variable. If it is, the algorithm replaces all uses of the induction variable.

FIGURE 4.8 Induction Variable Substitution

```
procedure IVSub(S);

    /* IVSub examines assignment statement S to determine whether */
    /* it is an induction variable, and replaces it if so. */

    /* First determine whether the statement defines an induction var */

    is_iv = false;
    if dominates(S, loop_exit, loop_entry) and has_one_self_edge(S)
    then
        let iv_edge be the one edge from S to S;
        let iv_expr be the target of iv_edge;
        let cexpr be rhs(S) with iv_expr zeroed out;
        if controlling_operation(iv_expr)== addition and
            controlling_operation(cexpr) == addition or subtraction
            then
                loop_invariant = true;
                foreach edge e coming into cexpr do
                    if common_nesting(source(e),S) == nesting_level(S)
                        then loop_invariant = false; fi
                od
                if controlling_operation(cexpr) == subtraction
                    then cexpr = -cexpr; fi
                if loop_invariant
                    then is_iv = true; fi
    fi  fi

    if not is_iv then return; fi

    /* The statement is known to be an induction variable; */
    /* next is substitution. */

    foreach edge e other than iv_edge out of S do
        if common_nesting(S, target(e)) == nesting_level(S) then
            if target(e) < S then
                replace target_expr(e) with
```

```
                target_expr(e) + (((I - L)/S) * cexpr)
            else
                replace target_expr(e) with
                    target_expr(e) + (((I - L + S)/S) * cexpr)
            fi
            foreach edge ie into cexpr do
                add a new edge from source(ie) to target(e);
            od
            foreach edge ie into iv_expr other than iv_edge do
                add a newedge from source(ie) to target_expr(e);
            od
            if not blocking_definition(iv_expr) then
                delete edge e;
    od  fi  fi

    move S outside loop, changing cexpr to iteration_count * cexpr;
    if blocking_definition(iv_expr) then
        delete edge iv_edge;
    fi
end IVSub;
```

IVSub begins by ensuring that the candidate statement *S* is executed on every iteration of the loop ("dominates (*S*, loop_exit, loop_entry)") and that the variable it defines is updated on every loop iteration (the self-edge requirement). Next, it ensures that the potential induction candidate is in fact dominated by addition (and not multiplication or some operator) and that a constant amount is either added or subtracted on each iteration. If the loop invariant part is subtracted, it is negated, so that the substitution phase can assume that addition is the controlling operation. The remainder of the algorithm performs the substitution; for statements that occur prior to *S* in the loop, the multiplier to be used is one less than the current iteration number. For statements after *S*, the multiplier is the current iteration number (*I* is the loop index; *L* is its lower bound; *U* is its upper bound; and *S* is its stride).

In the process of updating the definition-use graph to reflect the changes to the program, an important question that the algorithm must answer is whether to maintain the edges involving the induction variable from the statement *S* to all replaced uses of that variable. More specifically, if induction variable substitution is performed on `K` in the following:

```
     DO I = 1, N
        DO J = 1, M
S          K = K+1
```

```
S1          ... = A(K)
         ENDDO
      ENDDO
```

the following results

```
      DO I = 1, N
         DO J = 1, M
            CONTINUE
S1          ... = A(K+J)
         ENDDO
S        K = K + M
      ENDDO
```

Even after the induction variable `K` is replaced, the expression still uses `K`. Similarly, the induction statement still defines and uses `K` after it is moved out of the loop. In this case, it is obvious that the value of `K` defined in *S* can still reach both *S* and S_1, and as a result, it is necessary to retain the induction edge in the graph. If the example is changed slightly,

```
      DO I = 1, N
S2       K = 0
         DO J = 1, M
            CONTINUE
S1          ... = A(K+J)
         ENDDO
S        K = K + M
      ENDDO
```

it is equally obvious that the value of `K` defined in *S* no longer reaches itself or S_1. The definition in S_2 blocks the former induction from reaching around the loop, so that the induction edge is no longer valid. The routine “blocking_definition” is intended to detect situations exactly like this—definitions that are guaranteed to block the induction. Since it is conservative to leave the edge in, blocking_definition must only return true when it can positively guarantee the block.

Having developed individual algorithms for applying forward substitution and induction variable substitution, it is now time to develop a driving algorithm to tie the individual transformations together. There are a couple of key considerations. First, since induction variable substitution performed on an inner loop may produce a new induction on an outer loop (see, for instance, the examples on page 139), induction variable substitution should be performed from the inside out. Second, substitut-

ing loop invariant expressions forward may create some new inductions, meaning it should be performed first. These observations are both encapsulated in the algorithm *IVDrive*, presented in Figure 4.9.

FIGURE 4.9 Induction Variable Substitution Driving Algorithm

```
procedure IVDrive(loop);
    /* IVDrive performs forward substitution and induction variable */
    /* substitution on the loop loop, recursively calling itself where */
    /* necessary. */

    record loop parameters I, L, U, and S.
    foreach statement S in loop do
        case(kind(S))
            assignment:
                ForwardSub(S);
                IVSub(S);
            DO-loop:
                IVDrive(S);
            default:
        esac
    od
end IVDrive;
```

There are two final comments worth noting on induction variable substitution. The first regards the interaction between loop normalization (Section 4.3) and induction variable substitution. When induction variable substitution is performed on an unnormalized loop, extremely inefficient code can result. Consider, for instance, the very general example presented below

```
DO I = L, U, S
   K = K + N
   ... = A(K)
ENDDO
```

Applying induction variable substitution as described in Figure 4.8 yields

```
DO I = L, U, S
   CONTINUE
   ... = A(K + (I - L + S)/ S * N)
ENDDO
K = K + (U - L + S) / S * N
```

Integer divides and multiplies are implemented in hardware on many machines; even when they are, they are very inefficient instructions to use. By introducing both an integer divide and an integer multiply in the loop, induction variable substitution has produced much less efficient code. Normally, an optimization known as strength reduction is applied to attempt to reduce most common uses of integer multiply, but the form which appears inside the loop cannot be eliminated by this transformation. Furthermore, the nonlinear nature of the new subscript means that dependence testing will fail for it. The summary is that induction variable substitution is a very fruitless transformation to apply to this loop.

If the loop is normalized prior to induction variable substitution, however

```
I = 1
DO i= 1, (U-L+S)/S, 1
   K = K + N
   ... = A(K)
   I = I + 1
ENDDO
```

the code resulting from induction variable substitution is much more palatable

```
I = 1
DO i= 1, (U-L+S)/S, 1
   CONTINUE
   ... = A(K + i*N)
   CONTINUE
ENDDO
K = K + (U-L+S)/S * N
I = I + (U-L+S)/S
```

This code has only one integer multiply within the loop, which strength reduction can easily eliminate. Furthermore, the subscript can easily be handled by dependence tests described in Chapter 3. The summary is that loop normalization and induction variable substitution must be carefully tied together.

The second comment involves the version of forward substitution described in this section. The algorithm presented in Figure 4.7 is a simple version designed to demonstrate the general principles involved. A production compiler requires a more advanced version that is able to analyze for invariance over short regions, not just loops. For instance, a common transformation applied by programmers is loop unrolling. If a loop

containing an auxiliary induction variable is unrolled, something like the following can result

```
DO I = 1, N, 2
   K = K + 1
   A(K) = A(K) + 1
   K = K + 1
   A(K) = A(K) + 1
ENDDO
```

As described, neither induction variable substitution or forward substitution will do anything with any statement in this loop, leaving it totally unamenable for any form of optimization. However, when forward substitution is enhanced to recognize region invariance, the first assignment to `K` can be propagated forward, yielding:

```
DO I = 1, N, 2
   CONTINUE
   A(K+1) = A(K+1) + 1
   K = K+1 + 1
   A(K) = A(K) + 1
ENDDO
```

At this point, normal induction variable substitution will complete elimination of `K`. The extensions to forward substitution to recognize this case are straightforward, and are described in \cite Allen Supercomputer.

4.6 Summary

We have presented a number of algorithms for scalar analysis that are designed to improve dependence testing. In particular, constant propagation, replaces unknown variables with constants known at compile time and induction variable substitution, eliminates auxiliary induction variables, replacing them with linear functions of the standard loop induction variable. A simple extension of the induction variable substitution algorithm performs expression folding in loop nests.