6

Single Static Assignment Form as Intermediate Representation

In this chapter we present an intermediate form of programs called single static assignment (SSA) form that is useful for many optimizations. Because of the sparseness of def-use chains in the representation, optimizations based on SSA form can be performed efficiently.

6.1 Introduction

The result of many data flow analyses can be represented by superimposing structures called def-use or use-def chains on the CFG of a program. As mentioned in Section 2.3.3, a def-use chain associates with each definition a list of statements that are reached by the definition and contain uses of the variable being defined. Similarly, a use-def chain associates with each use of a variable, a list of statements containing definitions of the variable that reach the use. Def-use chains can be computed by extending liveness analysis. In this extension, the data flow information is a set of tuples (x,n) where $x \in \mathbb{V}\mathsf{ar}$ and n is a basic block, where it is assumed that each statement forms a basic block by itself. The CFG is traversed backwards as in liveness analysis. The use of a variable x in a statement at n generates the tuple (x,n). If a statement at n' contains a definition of the variable x, then, for each (x,n) in $\mathsf{Out}_{n'}$, n' is chained to n. (x,n) is subsequently killed by the statement at n'. Similarly, use-def chains can be found by a minor modification of reaching definitions analysis. Optimizations like dead code elimination make use of def-use chains whereas constant propagation and loop-invariant detection make use of use-def chains. Figure 6.1 shows an example program and its CFG on which the def-use chains have been superimposed. A def-use chain is concretely represented by a set of def-use edges connecting the definition with its uses.

Def-use chains are used to propagate data flow information. A def-use edge may bypass a path through a number of control flow edges and directly connect a definition with its use. Clearly, the time taken for performing an optimization based on def-use or use-def chains will depend on the number of def-use edges in the graph. Any optimization over the example program shown in Figure 6.1 will have to repeatedly iterate over the 12 chains, propagating a data flow value from a definition to its

```
switch(machineId)
{  case1:
      st = initState₁;
      break;
   case2:
      st = initState₂;
      break;
   case3:
      st = initState₃;
}
while (1)
{  sym = getsym();
   if(isAlpha(sym))
      st = next[st,sym];
   elseif(sym == '\n')
   {  printf("%d\n", st);
      nextline();
   }
   else
   {  printf("%d\n", st);
      break;
   }
}
```

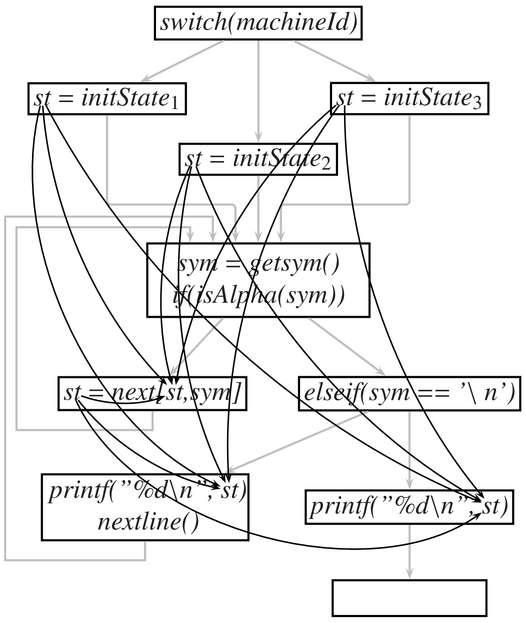


FIGURE 6.1
Example of def-use chains.

corresponding uses in each iteration. The number of def-use edges tend to proliferate when each of several definitions of a variable reach several uses of the same variable through a join node in the CFG. As an example, *m* definitions reaching each of *n* uses result in $m \times n$ def-use chains.

6.1.1 An Overview of SSA

A program in SSA form reduces the number of def-use (or use-def) chains by introducing a separate variable version for each definition of the same variable reaching a join node. Thus st_1, st_2, st_3 and st_4 are four different versions of the same variable *st*. Each version corresponds to a definition of state. The values carried by the four versions are transferred to a new version, st_5, at a join node. This is done using a notational mechanism called a ϕ-instruction. A ϕ-instruction is a special kind of assignment whose right hand side consists of a ϕ-function applied to the incoming variable versions (st_1, st_2, st_3 and st_4 for the example), and the left hand side consists of the new version (st_5). The variable st_5 reaches each of several uses in the original program. These uses are also modified to receive their values from st_5. Thus there are *m* def-use chains, one for each st_i reaching the ϕ-instruction, and *n* def-use chains corresponding to the definition involving the ϕ-instruction reaching each of *n* uses,

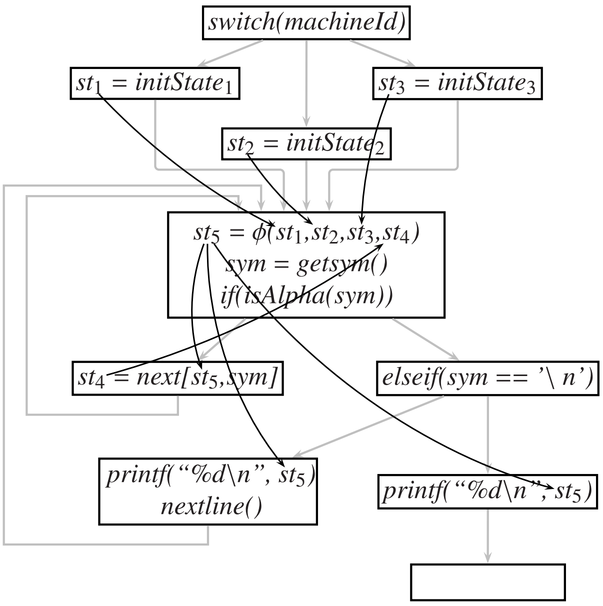


FIGURE 6.2
The earlier example in SSA form.

making up a total of $m+n$ chains. Figure 6.2 shows the earlier program in SSA form. The reduction in the number of def-use chains can be clearly observed.

The variables involved in ϕ-instructions are called ϕ-variables. The variables on the right hand side of a ϕ-instruction are called the *arguments* of the ϕ-instruction and the variable on the left hand side is called the *result*. Since the transformation to SSA form includes insertion of ϕ-instructions, it is important to describe the semantics of the ϕ-instructions. Consider a basic block with k predecessors. Then the block could have several ϕ-instructions, all placed at the beginning of the block. These are denoted as:

$$\begin{aligned}
y_1 &= \phi(x_{11}, x_{12}, \ldots, x_{1k}) \\
y_2 &= \phi(x_{21}, x_{12}, \ldots, x_{2k}) \\
&\vdots \\
y_n &= \phi(x_{n1}, x_{n2}, \ldots, x_{nk})
\end{aligned}$$

During execution, if the block containing these instructions is reached along predecessor edge j, then the effect of these instructions is that of simultaneously executing the assignment statements $y_1 = x_{1j}$, $y_2 = x_{2j}$, $\ldots$, $y_n = x_{nj}$ along the edge from the jth

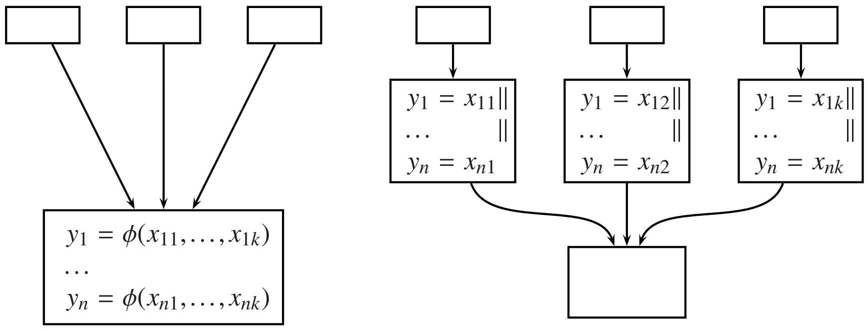


FIGURE 6.3
Semantics of ϕ-instruction.

predecessor block to the block containing the ϕ-instructions. This is shown in Figure 6.3. A simultaneous execution of $y_1 = e_1$ and $y_2 = e_2$, denoted $y_1 = e_1 \parallel y_1 = e_2$, first evaluates the expressions e_1 and e_2 and then assigns the resulting values to y_1 and y_2 respectively. As we shall see, the semantics becomes important when we transform the program into and out of SSA form.

6.1.2 Benefits of SSA Representation

Transformation of a program to SSA form results in a sparser representation of def-use chains. The benefit that results due to this sparsity is an improvement in time to perform the optimization. To see this, consider a generic work list based algorithm that uses def-use edges. Such an algorithm will propagate data flow values from the definition of a variable to its uses. Therefore, we can associate data flow values with the definition end and the use end of each def-use edge. At any point of time, the work list will hold def-use edges for which the data flow value has yet to be propagated from the definition to the use. After this is done, the propagated value is used to compute the value of the definition that depends on this use and, provided this is a new value, all def-use edges which have this definition as the argument are put on the work list.

The algorithm takes time proportional to the product of the total number of def-use edges and the number of times each edge can be inserted in the work list. The number of times each def-use edge can be put on the work list is the same as the maximum number of changes in the data flow value, and this is the same as the height of the data flow lattice. Thus, if we fix the data flow lattice, the time required for the analysis depends on the number of def-use edges in the program representation. As we have argued earlier, the number of def-use edges in a program in SSA form is smaller than the program from which it was constructed, thus reducing the time required for the analysis.

The second benefit is that certain analyses or optimizations become easier due to the nature of the SSA form itself. In a SSA form program, there is exactly one definition reaching each use of a variable. To see a use of this property, consider a method for detection of induction variables in a program. Figure 6.4 shows a program along with its SSA form. The def-use edges from the definitions to the uses of variables are shown explicitly. To discover that i is an induction variable of the original program, we note that statements $i_3 = \phi(i_1, i_2)$ and $i_2 = i_3 + 2$ form a strongly connected region (SCR) involving (versions of) the variable i in the SSA form program. The initial value of the variable is supplied by the statement $i_1 = 0$ and the statements constituting the SCR increase i by a constant in each iteration. In addition, since the SCR passes through the ϕ-instruction, i is identified as an induction variable of the outer loop. This information is not readily available in the original program with def-use chains. By a similar reasoning, j is detected to be an induction variable of the inner loop of the original program. Its increment, i, is detected to be a loop invariant of the inner loop because the definition of i_2 reaching the statement $j_2 = j_3 + i_2$ in the SSA form program is outside the SCR formed by the statements $j_3 = \phi(j_1, j_2)$ and $j_2 = j_3 + i_2$.

A larger example of use of SSA form will be presented later in the chapter when we discuss a method for register allocation that exploits the special properties of SSA form programs.

6.2 Construction of SSA Form Programs

As in reaching definitions analysis (Section 2.3.3), we assume that the node *Start* contains an assignment of the special value *undef* to every variable. Thus along any path in the CFG from *Start* to the use of a variable, there is at least one definition of the variable reaching the use. Programs which satisfy this property are called *strict programs*.

As mentioned earlier, the ϕ-instructions should be inserted where more than one definition coming along different paths converge. We first formalize the notion of converging paths.

DEFINITION 6.1 *Let $\rho_1 = (n_1, n_2, \ldots, o)$ and $\rho_2 = (m_1, m_2, \ldots, o)$ be non-null paths. ρ_1 and ρ_2 are said to converge, if:*

1. *The start nodes of ρ_1 and ρ_2 are different, i.e., $n_1 \neq m_1$.*
2. *The two paths are disjoint except for the node o.*

Note that the common node o could occur in more than one position in the two paths. An interesting example of converging paths for the CFG in Figure 6.5 is (n_1, n_5, n_7) and (n_7, n_9, n_{10}, n_7). If a variable is defined in nodes 1 and 7 of the CFG,

```
i = 0;
while (...)
{  i = i + 2;
   j = i;
   while (...)
      j = j + i;
}
```

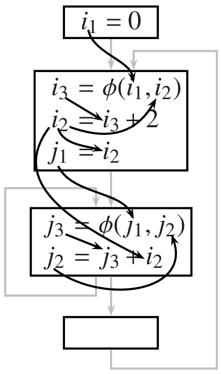


FIGURE 6.4
Detecting induction variables using SSA form.

then there must be a ϕ-instruction for this variable at the entry of 7. The example shows why the end node is allowed to occur in more than one position in the paths—the converging paths may include loops*. The pair of paths (n_5, n_7, n_8, n_{10}) and (n_6, n_7, n_9, n_{10}) is an example of paths that are non-converging.

We now specify the properties of a valid transformation of a program to SSA form. The algorithm that we describe later will be proved to be correct with respect to this specification.

DEFINITION 6.2 *The transformation of a program to another is a valid SSA-transformation, if the following two conditions are satisfied:*

1. *Correctness of form: Each variable mentioned in the transformed program must have exactly one definition.*

2. *Semantic invariance: Consider an execution path leading to a use of a variable x in the original program and a corresponding execution path leading to the variable version x_i in the program in SSA form. Then, under the execution semantics of ϕ-instructions described earlier, the two variables x and x_i must have the same value.*

Unless stated otherwise, by the phrase 'a program in SSA form' we shall mean a program that has been obtained by a valid SSA-transformation of a strict program.

A program in SSA form is *minimal*, if it results from a transformation satisfying the properties listed above and has a minimum number of ϕ-instructions. A program in SSA form is *pruned*, if it has the added restriction that a ϕ-instruction is inserted only if the result variable of the instruction is used later along some path.

*Observe that (n_1, n_5, n_7) and $(n_6, n_7, n_9, n_{10}, n_7)$ are also converging paths by the definition. This is clearly not necessary since their role is subsumed by the pair of paths (n_1, n_5, n_7) and (n_6, n_7).

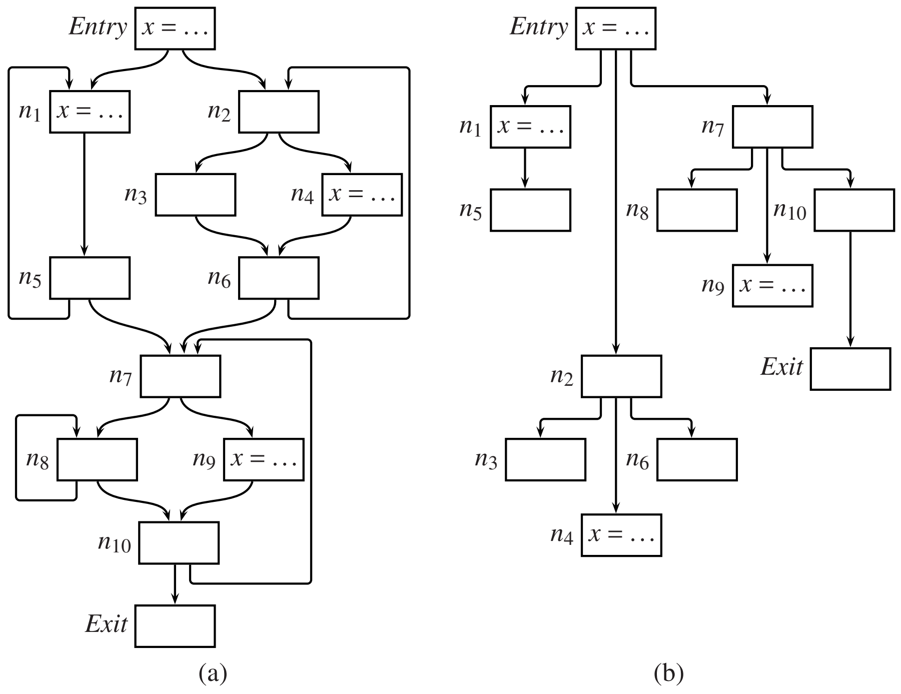


FIGURE 6.5
(a)A CFG and (b) its dominator tree.

To distinguish between the predecessor and successor relation in the CFG and the same relation in the dominator tree, we use the terms *ancestor* and *descendant* in the latter case. An immediate descendant will be called a child.

6.2.1 Dominance Frontier

The key idea behind insertion of ϕ-instructions is that of *dominance frontier*. To develop this idea, we first define the concept of dominance in a graph. Recall the definition of dominance from Section 3.1.

DEFINITION 6.3 *Let n and m be nodes in the CFG. The node n is said to dominate m, denoted $n \geq m$, if every path from* Start *to m passes through n.*

We also need a notion of dominance that is not reflexive.

DEFINITION 6.4 *If $n \geq m$ and $n \neq m$, then we say that n strictly dominates m and denote this as $n > m$. Further, the closest strict dominator of a node n*

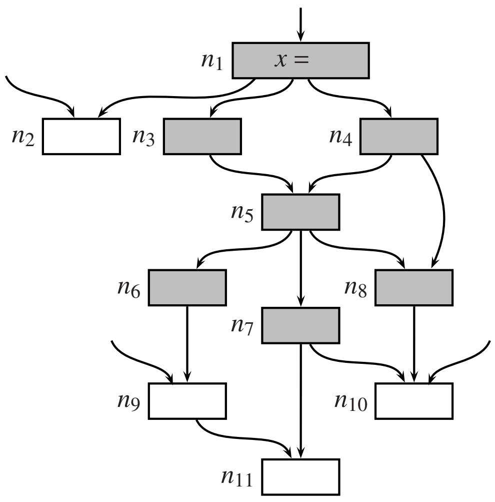


FIGURE 6.6
Idea behind ϕ insertion through dominance frontier.

is called the immediate dominator of n and is denoted as $\mathsf{idom}(n)$.

We use the notation $n \not\succ m$ to mean n does not strictly dominate m. Figure 6.5 shows a CFG and its *dominator tree* in which the edges represent immediate dominance. We shall sometimes consider dominance to be a relation between program points instead of nodes.

OBSERVATION 6.1 *If the nodes n and m both dominate a node o then either* $n \succeq m$ *or* $m \succeq n$.

Consider Figure 6.6 in which the node n_1 contains a definition of the variable x. The node n_6 is dominated by n_1 and so are all the shaded nodes in the figure. Each shaded node will have the property of a single definition of x reaching it and will thus not require a ϕ-instruction for x. Now consider the node n_9 which is an immediate successor of n_6 and is not dominated by n_1. This node needs a ϕ-instruction because, apart from the definition in n_1, some other definition, possibly the one that is assumed to initialize the value of x to *undef* at *Start*, will reach n_9. Nodes such as n_9, and n_{10} are said to be in the *dominance frontier* of n_1 and need a ϕ-instruction for the variable x. We shall now formalize this idea.

A straightforward translation of the idea represented by Figure 6.6 gives a first definition of dominance frontier. The dominance frontier of a node n, denoted $\mathsf{df}(n)$, is given as

$$\mathsf{df}(n) = \{m \mid \exists p \in \mathsf{pred}(m), (n \succeq p \text{ and } n \not\succ m)\}$$

By this definition, a loop header will be included in its own dominance frontier. This is reasonable since a variable in the loop header may have two reaching definitions—one from inside the loop, the other from outside. As an example, if n_1 is a loop header in Figure 6.6, a program point in n_1 before the definition of x will have more than one reaching definition—one reaching from outside the loop and the other from the definition in n_1 itself. In such a situation there will be a ϕ-instruction at the beginning of n_1.

A direct implementation of the above definition will find $\mathit{df}(n)$ by considering each node m dominated by n and checking whether it has an immediate successor in the CFG that is not strictly dominated by n. The problem with this approach is that it finds the dominance frontier of each node independently of the dominance frontier of other nodes. A more efficient algorithm that exploits the relation between dominance frontiers of different nodes is based on the following observations:

1. Consider Figure 6.6 as an example. Nodes that are immediate successors of n_1 and not strictly dominated by n_1 are in $\mathit{df}(n_1)$. An example of such a node is n_2. We call such nodes as $\mathit{df}_{base}(n_1)$ as these nodes are included in what can be considered as the base step of an inductive definition for df.

$$\mathit{df}_{base}(n) = \{m \in \mathit{succ}(n) \mid n \not\succ m\}$$

2. We shall now relate the dominance frontier of n_1 in Figure 6.6 to the dominance frontier of its children. Consider n_5 as an example of a child of n_1. The node n_9, which is in $\mathit{df}(n_5)$, is also in $\mathit{df}(n_1)$. However, n_8, which is also in $\mathit{df}(n_5)$ is not in $\mathit{df}(n_1)$. The reason is that while n_5 does not dominate n_8, its immediate dominator n_1 dominates n_8. We call this component of df as df_{ind}, the inductive step of the definition of df.

$$\mathit{df}_{ind}(n) = \bigcup_{m \in \mathit{children}(n)} \{p \in \mathit{df}(m) \mid \mathit{idom}(m) \not\succ p\}$$

Combining the two:

$$\mathit{df}(n) = \mathit{df}_{base}(n) \cup \mathit{df}_{ind}(n)$$

We can reformulate df_{base} and df_{ind} so that they use the easily checkable $\neq$ relation instead of $\not\succ$. If m is a successor of n then the condition $n \succ m$ is exactly the same as $n = \mathit{idom}(m)$. Thus

$$\mathit{df}_{base}(n) = \{m \in \mathit{succ}(n) \mid n \neq \mathit{idom}(m)\}$$

Similarly, if m is a child of n and p is in $\mathit{df}(m)$, then the condition $n \succ p$ is exactly the same as $n = \mathit{idom}(p)$. To see this, we first observe that any strict dominator of p is also a strict dominator of m. Assume to the contrary that o is a strict dominator of p and either o is the same as m or o is unrelated to m in the dominance relationship. In the first case o cannot dominate p, because p is in $\mathit{df}(o)$. In the second, if o is

Input: A CFG with the dominance frontier for each node.
Output: The dominance frontier of each node n in the CFG computed in a variable DF_n.
Algorithm:

```
0   for each $n$ in a bottom up traversal of the dominator tree do
1       {   $DF_n = \emptyset$
2           for each $m \in succ(n)$ do                      /* Calculate $df_{base}$ */
3               if $idom(m) \neq n$ then $DF_n = DF_n \cup \{m\}$;
4           for each $m \in children(n)$ do                  /* Calculate $df_{ind}$ */
5               for each $p \in DF_m$ do
6                   if $idom(p) \neq n$ then $DF_n = DF_n \cup \{p\}$;
7       }
```

FIGURE 6.7
The algorithm for dominance frontier.

unrelated to m, o cannot dominate p since there is an alternate path from *Start* to p through m which does not pass through o. Thus we have a contradiction.

Now since n is the closest ancestor of m that strictly dominates p, we must have $n = idom(p)$. Thus we can rewrite df_{ind} as

$$df_{ind}(n) = \bigcup_{m \in children(n)} \{p \in df(m) \mid n \neq idom(p)\}$$

The algorithm in Figure 6.7 computes the dominance frontier using the formulation presented above. The table in Figure 6.8 gives df_{base} and df_{ind} for the nodes in the CFG in Figure 6.5.

Let E and N be the number of edges and nodes in the CFG. To calculate df_{base}, the algorithm clearly visits each edge once, so its complexity is $O(E)$. Let $|df(n)|$ denote the size of dominance frontier of the node n. Then the complexity of the part that calculates df_{ind} is bounded by $O(\Sigma_n|df(n)|)$. This is $O(N^2)$ for arbitrary CFGs, which gives an overall complexity of $O(E+N^2)$. However, it can be shown that for CFGs programs composed of assignments, if-then-else and while-dos, $|df(n)|$ is a constant. For such CFGs, both $O(\Sigma_n|df(n)|)$ and E are $O(N)$. Thus the complexity of the algorithm is also $O(N)$.

6.2.2 Placement of ϕ-instructions

The algorithm for placing ϕ-instructions is shown in Figure 6.9. It considers each variable in turn and maintains a work list for nodes that are yet to be examined. For every variable it starts by inserting the nodes that contain an assignment to the variable in the work list. The dominance frontier of each node in the work list is examined. ϕ-instructions are inserted in the nodes forming the dominance frontier, and these nodes are in turn inserted in the work list.

The algorithm, maintains the following variables.

Node	*Exit*	10	9	8	7	6	5	4	3	2	1	*Entry*
df_{base}	∅	7	8,10	10	∅	7,2	7,1	6	6	∅	∅	∅
df_{ind}	∅	∅	∅	∅	7	∅	∅	∅	∅	7,2	7,1	∅

FIGURE 6.8
df_{base} and df_{ind} for the CFG in Figure 6.5 on page 191.

- *inWorklist*: If $inWorklist_n$ is x, it means that the node n has been inserted in the work list in connection with the variable x.
- *inserted*: If $inserted_n$ is x, it means that a ϕ-instruction has been inserted in node n for the variable x.
- *assign*: $assign_x$ is the set of nodes containing an assignment to the variable x in the original program.

It is possible for the following situation to arise: A node n has been put in the work list in connection with a variable but a ϕ-instruction for the variable has not yet been inserted in n. This could happen, for instance, when the node being examined is a loop header containing an assignment to a variable. Thus $inserted_m$ and $inWorklist_m$ could have different values when entering the body of the **for** loop in line 13 and therefore checking the condition $inWorklist_m \neq x$ in line 16 is not redundant.

For the CFG in Figure 6.5 on page 191, a ϕ-instruction for the variable x is inserted at node 1 since 1 contains a definition of x and is in its own dominance frontier. Similarly a ϕ-instruction is inserted in 6 which is in $df(4)$ and 2 which is in $df(6)$. ϕ-instructions are also inserted in 7 and 10.

From a single node, the notion of dominance frontier can be generalized to a set of nodes in the following way:

$$df(S) = \bigcup_{x \in S} df(x)$$

It is easy to see that df is monotonic, i.e., $S_1 \subseteq S_2$ implies $df(S_1) \subseteq df(S_2)$.

If S_x is the set of nodes containing assignments to the variable x, then the ϕ-instructions placed by the ϕ-placement algorithm is given by the *iterated dominance frontier* of S_x denoted as $idf^+(S_x)$. This is defined as the limit of the increasing sequence $idf^i(S)$:

$$idf^1(S) = df(S) \tag{6.1}$$
$$idf^{i+1}(S) = df(S \cup idf^i(S)) \tag{6.2}$$

Let $A_{orig}(n)$ and $A_{trans}(n)$ represent the number of assignments in node n in the original and the transformed program. Observe that nodes are put in the work list $O(A_{trans})$ number of times, and for each node n that has been put in the work list $O(|df(n)|)$ nodes are examined. Let *avgcost* represent this work averaged over all the assignments in the transformed program. Thus

Input: A CFG with the dominance frontier for each node.
Output: The CFG with the ϕ-instructions inserted but without variable renaming.
Algorithm:

```
$worklist = \emptyset$
for each node $n$ do
{   $inserted_n = x_0$                    /* $x_0$ should not occur in the program */
    $inWorklist_n = x_0$
}
for each variable $x$ do
    for each $n \in assign(x)$ do
    {   $inWorklist_n = x$
        $worklist = worklist \cup \{n\}$
    }
    while $worklist \neq \emptyset$ do
    {   remove a node $n$ from $worklist$
        for each $m \in df_n$ do
            if $inserted_m \neq x$ then
            {   place a $\phi$-instruction for $x$ at $m$
                $inserted_m = x$
                if $inWorklist_m \neq x$ then
                {   $inWorklist_m = x$
                    $worklist = worklist \cup \{m\}$
                }
            }
    }
```

FIGURE 6.9
The algorithm for ϕ placement.

$$avgcost = \sum_n (A_{trans}(n) \times |df(n)|)/\Sigma_n(A_{trans}(n))$$

Then the cost for computation of the iterated dominance frontier is $avgcost \times \Sigma_n(A_{trans}(n))$. However, for CFGs consisting of assignments, if-then-else and while-dos, $avgcost$ is a constant and the complexity reduces to $O(\Sigma_n(A_{trans}(n)))$.

6.2.3 Renaming of Variables

The algorithm for renaming is given in Figure 6.11. In order to generate new versions of the variable, the algorithm maintains a counter for each variable. To rename the use of a variable in an assignment, the algorithm needs to keep track of the definition that reaches the use. The algorithm maintains a array of stacks (called $stacks$), one for each variable for this purpose. As the algorithm traverses the program, the version of x that reaches a program point is given by the value of $top(stacks_x)$. In addition, to rename the arguments of a ϕ-function, we need to know the predecessor number

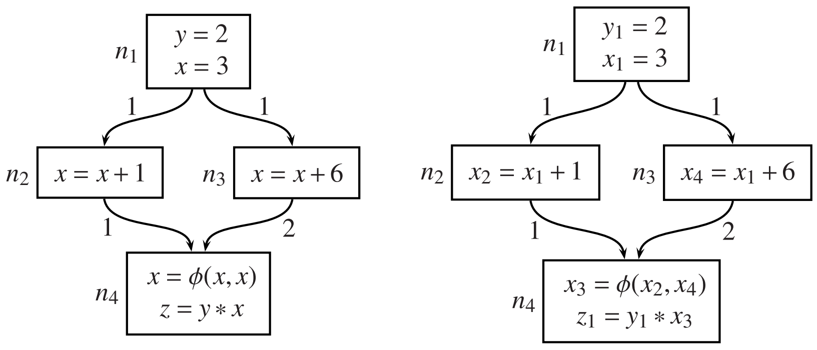


FIGURE 6.10
CFG before and after renaming variables.

of a node with respect to its successor[†]. This is given by *predNumber*(m,n), where n is a predecessor of m.

Consider a call to *rename*(n). If a variable x is used by an ordinary assignment, it is renamed to the version x_i given by *top*(*stacks*$_x$). The definition of a variable y, whether defined by a ordinary assignment or a ϕ-instruction, is renamed to a new version y_j. The new version number j is inserted in the stack for y. The call to *rename*(n) also renames the arguments of the ϕ-function in each successor m of n. The reason why this is done during a call to *rename*(n) is the following. To rewrite the ith argument of a ϕ-function, we need to know the variable version whose definition reaches the end of the ith predecessor. If x is the current variable being renamed, it is at this point of time that we know that *top*(*stacks*$_x$) is the version of x reaching the end of *predNumber*(m,n). This information is used for renaming. Thus the uses on the right hand side of a ϕ-instruction and an ordinary assignment are renamed during different calls to *rename*.

Example 6.1

We illustrate the algorithm for renaming variables through the example in Figure 6.10. The labels on the edges number the predecessors of a node. Thus n_2 and n_3 are the first and the second predecessors of n_4.

- The algorithm does a reverse postorder traversal of the dominator tree starting with node n_1. The variables on the left hand side of the assignments are renamed to y_1 and x_1. Since none of n_1's successors contain a ϕ-instruction, the children of n_1 are processed next. The values of *top*(*stacks*$_x$) and *top*(*stacks*$_y$) are both 1 at this time.

[†]The predecessors of a node are assumed to be ordered.

- Assume that n_2 is the node that is selected next. The variable x on the right hand side of n_2 is renamed to the variable version whose definition reaches this use. This is indicated by stacks_x as being x_1. The variable on the left hand side is changed to a new variable version x_2 and $\mathsf{top}(\mathsf{stacks}_x)$ is changed to 2. Since n_2 has a successor which has a ϕ-instruction, the first variable on the right hand side of this assignment is renamed to x_2. After n_2 is processed, stacks_x is popped.

- Since n_4 is also a child of n_1, assume it is picked next. The left hand side of the ϕ-instruction is renamed to a new variable x_3. Since the values of $\mathsf{top}(\mathsf{stacks}_x)$ and $\mathsf{top}(\mathsf{stacks}_y)$ are 3 and 1, the assignment following the ϕ-instruction is renamed to $z_1 = y_1 * x_3$. The stacks for x and z are popped.

- Finally, the block n_3 is rewritten as shown in the figure. Since n_4 is a successor of n_3 and this has a ϕ-instruction, the second argument of the ϕ-instruction is renamed to x_4, the version of x reaching this program point.

  □

Let $M_{trans}(n)$ denote the number of mentions (uses and definitions) of variables in the block n of the transformed program. Then the algorithm is linear in total number of mentions of variables in the entire transformed program, i.e., the complexity is $O(\Sigma_n(M_{trans}(n)))$.

6.2.4 Correctness of the Algorithm

We now show that the algorithm to calculate the dominance frontier, the ϕ-placement algorithm and the renaming algorithm together satisfy the specification of a valid transformation to SSA form. To do this we first need to prove the following important property regarding placement of ϕ-instructions in the transformed program: If two non-null paths which begin with the definitions of different versions of the same variable converge at a node n, then there is a ϕ-instruction for the variable at n. Note that the definitions at the beginning of the converging paths could themselves involve ϕ-instructions.

DEFINITION 6.5 *Given a set of nodes S, the join of S is defined as:*

$$\mathsf{join}(S) = \{n \mid \exists \text{ converging paths } m_1 \xrightarrow{+} n \text{ and } m_2 \xrightarrow{+} n,\ m_1, m_2 \in S\} \tag{6.3}$$

The ***iterated join*** *of a set of nodes S, denoted $\mathsf{ij}^+(S)$, is defined as the limit of the increasing sequence $\mathsf{ij}^i(S)$:*

$$\mathsf{ij}^1(S) = \mathsf{join}(S) \tag{6.4}$$
$$\mathsf{ij}^{i+1}(S) = \mathsf{join}(S \cup \mathsf{ij}^i(S)) \tag{6.5}$$

Input: A CFG with ϕ-instruction inserted.
Output: The same CFG with variables renamed.
Algorithm:

```
for each variable $x$ do
{   $counter_x = 0$; $stacks_x = emptyStack$
}
$rename(Start)$

function $rename(n)$
{   for each assignment $a$ in $n$ do
    {   if $a$ is an ordinary assignment then
            for each variable $x$ in $RHS(a)$ do
                replace $x$ by $x_i$ where $i = top(stacks_x)$
        let $y$ be $LHS(a)$ in
        {   $i = counter_y$
            replace $y$ by new variable $y_i$ in $y = e$
            push $i$ onto $stacks_y$
            $counter_y = i + 1$
        }
    }
    for each $m \in succ(n)$ do
    {   $j = predNumber(m, n)$
        for each $\phi$-instruction $a$ in $m$ do
            replace $j$-th operand $x$ in $RHS(a)$ by $x_i$, where $i = top(stacks_x)$
    }
    for each $m \in children(n)$ do $rename(n)$
    for each assignment $a$ in $n$ do
        $pop(stacks_z)$, where $z$ is the original variable of $LHS(a)$
}
```

FIGURE 6.11
Algorithm for renaming.

The property regarding placement of ϕ-instructions can now be recast as follows: If S_x is the set of definition involving a variable x in the original program, then there must be a ϕ-instruction for x in every node in $ij^+(S_x)$. To prove this, we need a result relating the end node of a non-null path with the iterated dominance frontier of the start node of the path.

LEMMA 6.1
Consider a path $\rho : n \overset{+}{\rightarrow} m$. We can find a node n' on ρ such that $n' \in \{n\} \cup idf^+(\{n\})$ and n' dominates m. Further, if n does not dominate each node in ρ, $n' \in idf^+(\{n\})$.

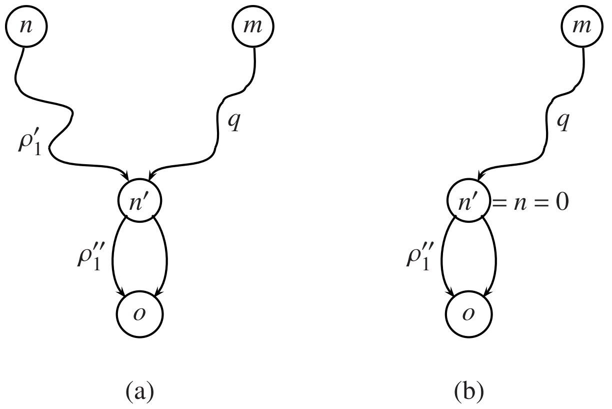


FIGURE 6.12
Figure illustrating Lemma 6.2.

PROOF Clearly, if n dominates each node in ρ, then n' is the same as n. Now assume that there are nodes in ρ that are not dominated by n. Let the path ρ be $(n = n_0, n_1, \ldots, n_k = m)$. Since n does not dominate all nodes in ρ, there will be some nodes in ρ that are in $\mathit{idf}^+(\{n\})$. Let n_j be the node with the largest index j that is in $\mathit{idf}^+(\{n\})$. Claim that n_j is the required n'.

Suppose n_j does not dominate m. Then consider the closest node n_i on ρ that is not dominated by n_j. All nodes in between n_j and n_{i+1} are dominated by n_j. Thus $n_i \in \mathit{df}(\{n_j\})$ and since $n_j \in \mathit{idf}^+(\{n\})$, $n_i \in \mathit{idf}^+(\{n\})$. Thus n_j is not the node with the largest index that is in $\mathit{idf}^+(\{n\})$. ▮

The second lemma shows that a one step join of two nodes is contained in the union of their iterated dominance frontier.

LEMMA 6.2
Let n and m be two distinct nodes in the CFG. Then $\mathit{join}(\{n,m\}) \subseteq \mathit{idf}^+(\{n\}) \cup \mathit{idf}^+(\{m\})$.

PROOF Let $o \in \mathit{join}(\{n,m\})$. Then there are non-null paths $\rho_1 : n \overset{+}{\rightarrow} o$ and $\rho_2 : m \overset{+}{\rightarrow} o$ converging at o. From Lemma 6.1, there is a node n' on ρ_1 and a node m' on ρ_2 such that both n' and m' dominate o. We prove the lemma by case analysis:

1. *n' is also on ρ_2:* The general situation is illustrated by Figure 6.12(a) where ρ_1 is a concatenation of the paths ρ_1' and ρ_1''. Of course, one of the path segments ρ_1' and ρ_1'' could be null. From the definition of convergence, n' is the same as o. If n does not dominate all nodes in ρ_1

then Lemma 6.1 gives us $n' \in \mathit{idf}^+(\{n\})$ and we are through. If n dominates all the nodes in ρ_1 then the situation is illustrated by Figure 6.12(b) obtained by considering ρ'_1 to be a null path. Clearly n' and n are the same and $n' \in \mathit{df}(\{n\})$. Therefore $n' \in \mathit{idf}^+(\{n\})$.

2. *m' is also on ρ_1:* The reasoning for this case is similar to the previous case.

3. *n' is not on ρ_2 and m' is not on ρ_1:* We shall show that this is not possible. Since n' and m' both dominate o, from Observation 6.1, either $n' \geq m'$ or $m' \geq n'$. The condition $m' \geq n'$ along with $n' \geq o$ implies that every path from m' to o has n' on it. In particular, n' is on ρ_2. Since this is not the case, the only possibility is $n' \geq m'$. By a symmetrical reasoning, we also have $m' \geq n'$. This gives $n' = m'$ contradicting the initial assumption that n' is not on ρ_2.

∎

It is easy to generalize Lemma 6.2 to any finite set of nodes.

COROLLARY 6.1
For a set of nodes S, $\mathit{join}(S) \subseteq \mathit{idf}^+(S)$.

PROOF Induction on the number of nodes in S and use of Lemma 6.2.
∎

We now show that dominance frontier is contained in joins.

LEMMA 6.3
Let S be a set of CFG nodes that contains the Start *node. Then $\mathit{df}(S) \subseteq \mathit{join}(S)$.*

PROOF Let $n \in S$ and $m \in \mathit{df}(\{n\})$. Then there is a path ρ_1 from n to m in which all the nodes till the predecessor of m are dominated by n. Of course, m could be the same as n. Further, since $m \in \mathit{df}(\{n\})$, there is a path ρ_2 from *Start* to m which does not pass through any node in ρ_1 except m. Since the two paths converge at m, m is in $\mathit{join}(S)$. ∎

We finally show that iterated dominance frontier computes the same set that is specified by iterated joins.

LEMMA 6.4
Let S be a set of nodes in a CFG that contains the Start *node. Then*

$$\mathit{ij}^+(S) = \mathit{idf}^+(S)$$

PROOF We first prove

$$\mathit{ij}^+(S) \subseteq \mathit{idf}^+(S)$$

by an induction on the iteration index in the definition of ij^+. Specifically, we show that for all k,

$$\mathit{ij}^k(S) \subseteq \mathit{idf}^+(S)$$

Then, since $\mathit{ij}^+(S) = \mathit{ij}^k(S)$ for some finite k, we shall have shown the containment in the limit.

Basis: Follows from Corollary 6.1.

$$\mathit{ij}(S) = \mathit{join}(S) \subseteq \mathit{idf}^+(S)$$

Inductive step:

$$\begin{aligned}
\mathit{ij}^k(S) &= \mathit{ij}(S \cup \mathit{ij}^k(S)) & \\
&\subseteq \mathit{ij}(S \cup \mathit{idf}^+(S)) & \text{(induction hypothesis, monotonicity of } \mathit{ij}\text{)} \\
&= \mathit{join}(S \cup \mathit{idf}^+(S)) & \text{(definition of } \mathit{ij}\text{)} \\
&\subset \mathit{idf}^+(S \cup \mathit{idf}^+(S)) & \text{(Corollary 6.1)} \\
&= \mathit{idf}^+(S) & \text{(definition of } \mathit{idf}\text{)}
\end{aligned}$$

The proof of $\mathit{idf}^+(S) \subseteq \mathit{ij}^+(S)$ is very similar.

▮

Let S_x represent the set of nodes which contain a definition of x. By our assumption, $\mathit{Start} \in S_x$. Therefore $\mathit{ij}^+(S_x) = \mathit{idf}^+(S_x)$ for any variable x in the program.

We next prove the first condition in the specification of valid SSA-transformation is satisfied by the algorithm.

LEMMA 6.5
Each variable in the SSA form program is assigned exactly once.

PROOF After renaming the definition of a variable x in the original program, $\mathit{counter}_x$ is incremented before renaming the next definition. So there is at most one assignment to a variable x_i. Thus we have to show that for each variable x_i which has a use occurrence in the SSA form program, there is at least one assignment to x_i.

When the renaming algorithm was renaming the use occurrence of x to x_i, the value of $\mathit{top}(\mathit{stacks}_x)$ must have been i. Since $\mathit{top}(\mathit{stacks}_x)$ is set to a value i only after renaming a definition of x to x_i, there is at least one definition of x_i. ▮

For an assignment statement a, let the notations $\mathit{before}(a)$ and $\mathit{after}(a)$ denote program points just before and after a. Similarly, if n is a block then $\mathit{after}(n)$ will denote a program point just after the last statement in n.

Finally we show that the SSA-transformation algorithm maintains semantic invariance. We show that the value of a variable x at a statement in the original program is the same as the renamed variable at the same statement in the SSA form program. This requires us to know what the renamed variable at different program points are. The version of x at the program point p in the transformed program is denoted as $\mathit{version}(x, p)$. This is the version that corresponds to the value of $\mathit{top}(\mathit{stacks}_x)$ when the renaming algorithm is at the program point p during its traversal of the CFG. Clearly, the following relations hold:

1. If the statement a_1 is followed by the statement a_2 in a block, then

$$\mathit{version}(x, \mathit{after}(a_1)) = \mathit{version}(x, \mathit{before}(a_2))$$

2. If a is the last statement in a block n, then

$$\mathit{version}(x, \mathit{after}(a)) = \mathit{version}(x, \mathit{after}(n))$$

LEMMA 6.6
Let x be a variable and $n \to m$ be an edge in the CFG such that m does not have a ϕ-instruction for x. Then

$$\mathit{version}(x, \mathit{after}(n)) = \mathit{version}(x, \mathit{after}(\mathit{idom}(m)))$$

PROOF If $n = \mathit{idom}(m)$, there is nothing to be proven. So assume that $n \neq \mathit{idom}(m)$. Since n dominates a predecessor of m (namely itself) and does not strictly dominate m, $m \in \mathit{df}(n)$. Further, from Lemma 6.4, since m does not have a ϕ-instruction for x, n does not have a definition for x.

Observe that $\mathit{idom}(m)$ strictly dominates n; otherwise there would be a path from *Start* to m through n which bypasses $\mathit{idom}(m)$. Consider the node o that is closest to m, strictly dominates m and defines x. Then

$$\mathit{version}(x, \mathit{after}(n)) = \mathit{version}(x, \mathit{after}(\mathit{idom}(m))) = \mathit{version}(x, \mathit{after}(o))$$

▮

Given a variable x and a control flow path ρ from *Start* to a program point p, let $\mathit{val}(x, \rho)$ denote the value of x at program point p when execution takes place along ρ.

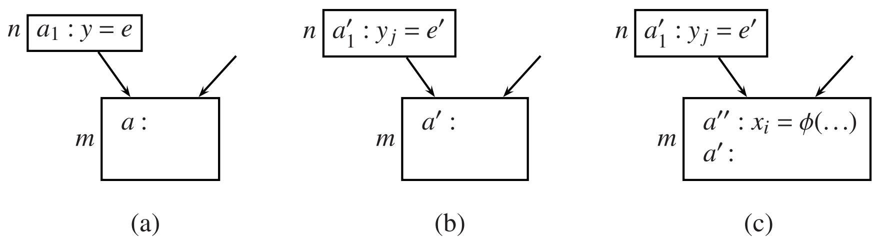


FIGURE 6.13
Figure illustrating Lemma 6.7. (a) represents the original program and (b) and (c) represent the transformed program.

LEMMA 6.7
Consider a path ρ in the original program from Start *to an assignment statement a. Let ρ' and a' be the corresponding path and statement in the SSA-transformed program. Then, for any variable x,*

$$\mathit{val}(x,\rho) = \mathit{val}(\mathit{version}(x, \mathit{before}(a')),\rho')$$

PROOF The proof is by induction on number of statements in the path ρ. In the proof, a will denote the last statement of ρ and a' and ρ' will denote the corresponding statement and path in the transformed program. Further $\rho - \{a\}$ will denote the path obtained by deleting the last statement a from ρ.

Basis: Consider the path ρ from *Start* to the first statement a of the successor node of *Start*. Clearly for any variable x,

$$\begin{aligned} \mathit{val}(x,\ \rho) &= \mathit{undef} \\ &= \mathit{val}(x_0,\ \rho') \\ &= \mathit{val}(\mathit{version}(x,\ \mathit{before}(a')),\ \rho') \end{aligned}$$

Inductive step: Now assume that the lemma holds for all paths of length $k-1$ or less. Consider a path ρ of length k. We consider the following cases.

1. *a is not the first statement of the containing block m.* Consider the statements before a and a' as shown below. e' has been obtained from e by renaming each variable y in e to $\mathit{version}(y,\ \mathit{before}(x_i = e'))$

$$\begin{array}{ll} x = e & \qquad x_i = e' \\ a : & \qquad a' : \end{array}$$

   By the induction hypothesis, for any variable y,

$$\mathit{val}(y,\ \rho - \{x = e\}) = \mathit{val}(\mathit{version}(y,\ \mathit{before}(x_i = e')),\rho' - \{x_i = e'\})$$

Thus from the induction hypothesis and the semantics of the assignment statement,

$$\mathit{val}(x, \rho) = \mathit{val}(\mathit{version}(x, \mathit{before}(a')),\rho')$$

Variables other than x remain unchanged and the statement of the lemma holds for them because of the induction hypothesis.

2. *a is the first statement of the containing block m.* Assume that the control flows along the edge $n \to m$. This situation is shown in Figure 6.13. The original program is shown in part (a) and the two cases of the transformed program are shown in parts (b) and (c). Let the paths to the end of node n be denoted as ρ_1 and ρ_1'. We first show that for any variable x, $\mathit{val}(x, \rho_1) = \mathit{val}(\mathit{version}(x, \mathit{after}(n)), \rho_1')$. Consider the last statement of n denoted as a_1 and the corresponding statement in the transformed program a_1'. Because of the induction hypothesis, we have for any variable y,

$$\mathit{val}(y, \rho_1 - \{a_1\}) = \mathit{val}(\mathit{version}(y, \mathit{before}(a_1')), \rho_1' - \{a_1'\})$$

Once again, using the induction hypothesis and the semantics of assignment, we have:

$$\mathit{val}(x, \rho_1) = \mathit{val}(\mathit{version}(x, \mathit{after}(a_1')), \rho_1')$$

and therefore

$$\mathit{val}(x, \rho_1) = \mathit{val}(\mathit{version}(x, \mathit{after}(n)), \rho_1')$$

We now have to show that the values of any variable x and its renamed version $\mathit{version}(x, \mathit{before}(a'))$ match. For this consider two subcases:

(a) m does not have a ϕ-instruction for x. Let the path to the immediate dominator of m be ρ_1''. Then:

$$\begin{aligned}\mathit{val}(x,\rho) &= \mathit{val}(x, \rho_1)\\ &= \mathit{val}(\mathit{version}(x, \mathit{after}(n)), \rho_1')\\ &= \mathit{val}(\mathit{version}(x, \mathit{after}(\mathit{idom}(m))), \rho_1'') && \text{(Lemma 6.6)}\\ &= \mathit{val}(\mathit{version}(x, \mathit{before}(a')), \rho') \\ & && \text{(Renaming algorithm, line 22)}\end{aligned}$$

(b) If m has a ϕ-instruction for x, then:

$$\begin{aligned}\mathit{val}(x, \rho) &= \mathit{val}(x, \rho_1)\\ &= \mathit{val}(\mathit{version}(x, \mathit{after}(n)), \rho_1')\\ &= \mathit{val}(\mathit{version}(x, \mathit{before}(a'')), \rho_1')\\ & && \text{(Renaming algorithm, lines 18-20)}\\ &= \mathit{val}(x_i, \rho') && \text{(Semantics of } \phi\text{-instruction)}\\ &= \mathit{val}(\mathit{version}(x, \mathit{before}(a')), \rho')\\ & && \text{(Renaming algorithm, line 8-9)}\end{aligned}$$

∎

The following theorem ties the previous results into an statement of correctness of the entire algorithm.

THEOREM 6.1
The algorithms for ϕ-placement and renaming together constitute a valid SSA-transformation.

PROOF Follows from Lemmas 6.5 and 6.7. ∎

We finally prove a property about programs in SSA form that will be used in later sections. Let the program point associated with the definition of a variable x be represented as $\mathsf{def}(x)$. This is the point just before the statement that has a definition of x, where the defining statement may also be a ϕ-instruction. Program points associated with the uses of a variable x are denoted as $\mathsf{use}(x)$, and are defined as follows:

DEFINITION 6.6 *A program point p is in $\mathsf{use}(x)$ iff*

1. *The statement just after p is an ordinary assignment (not a ϕ-instruction) and x occurs on the right hand side of the assignment.*
2. *p is $\mathsf{after}(n)$, n is the ith predecessor of a block m and m contains a ϕ-function with x as the ith argument.*

We shall use the term $\mathsf{use}(x)$ to refer to any of the points denoted by it.

LEMMA 6.8
(SSA dominance property) *Consider the SSA transformation of a strict program. For any variable x in the transformed program, $\mathsf{def}(x) \geq \mathsf{use}(x)$.*

PROOF Because of the semantic invariance property of an SSA transformation, the SSA of a strict program is also strict. Now assume that there is a use of a variable that is not dominated by its definition. By Lemma 6.5, the variable has a single definition. If this definition does not dominate the use, then the SSA form program is not strict, a contradiction. ∎

The program in SSA form must be finally converted into executable code. However no real processor has instructions that can directly capture the semantics of ϕ-instructions. Therefore the ϕ-instructions have to be replaced by code fragments inserted at appropriate places. The elimination of ϕ-instructions from a program in SSA form is called *SSA destruction*. The intermediate form of the program that

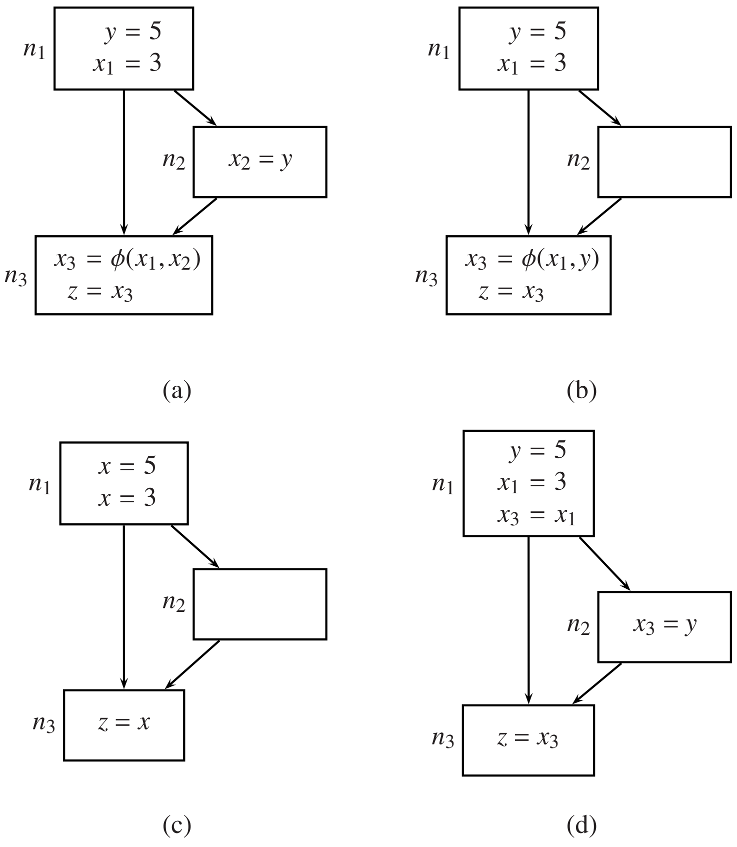


FIGURE 6.14
(a) A program in CSSA form. (b) The same program in TSSA form after copy propagation. (c) Eliminating ϕ assignments by merging variable versions results in an incorrect program. (d) A correct program obtained by inserting copy statements.

emerges as the result of applying the SSA construction algorithm discussed earlier is called *canonical SSA (CSSA)*. This is to distinguish it from the SSA form after optimizations called *transformed SSA (TSSA)*.

6.3 Destruction of SSA

Before embarking on the issues related to SSA destruction, we define live ranges for SSA form programs. Recall that the last use of the *i*th argument of a ϕ-function in a block n is considered to be at the end of the ith predecessor block of n. This means that this argument, say x_i, is not live at the entry of the block containing the ϕ-instruction. In contrast, the result of a ϕ-instruction is live at the entry of the block containing the ϕ-instruction. This follows from the semantics of ϕ-instruction which places a copy statement $\ldots = x_i$ on the edge from the ith predecessor to n.

DEFINITION 6.7 *The live range of a variable is its def-use chain. It includes all the program points between the definition and each of its uses. Two live ranges* **interfere** *if there is a program point that is common to both the live ranges.*

Because of the single definition property of SSA form programs, the definition associates a live range with a variable. In contrast, for non-SSA form programs, the live range is defined as the maximal union of intersecting def-use chains[‡]. Since def-use chains for SSA form programs do not intersect, the simple definition given above suffices.

A naive method to convert a SSA form program into an executable form may simply merge different versions of the same variable into one. While merging variable versions may be a trivial matter for a CSSA—it simply means going back to the original program—it can affect correctness in the case of TSSAs. Figure 6.14(a) shows an example of a program in CSSA form. Part (b) shows copy-propagation applied to the program. Notice that the definitions of both y and x_1 interfere at the end of the predecessor block n_1. This has important consequences for SSA destruction. Part (c) shows the result of replacing y, x_1 and x_3 by a single variable x and eliminating the ϕ-instruction. The resulting program is incorrect. However, in keeping with the semantics of the ϕ-instruction, we can insert copy statements at the end of blocks n_1 and n_2. This is shown in part (d). While this is correct for this example, we shall show later that removing ϕ-instructions by inserting copy statements may still result in incorrect programs. Besides, the copy at the end of n_1 is obviously redundant. The subtleties involved in SSA destruction through insertion of copy statements are illustrated through two well-known problems called the *lost-copy problem* and the *swap problem*.

The lost copy problem is illustrated in Figure 6.15 on the next page. The original program and its SSA form are shown in Figures 6.15, parts (a) and (b). The program after copy propagation and dead-code elimination is shown in part (c). Finally, part (d) shows the program after insertion of copy statements. The resulting program is incorrect because it prints the value of x in the last iteration instead of the penultimate iteration.

The reason for the incorrectness is a departure from the semantics of ϕ-instructions. This requires us to insert the copy $x_3 = x_2$ on the back edge from node n_2 to itself. This edge is a *critical edge*. An edge $n \rightarrow m$ is a critical edge if n has more than one successor and m has more than one predecessor. What we have done is to hoist the

[‡]Also called a web.

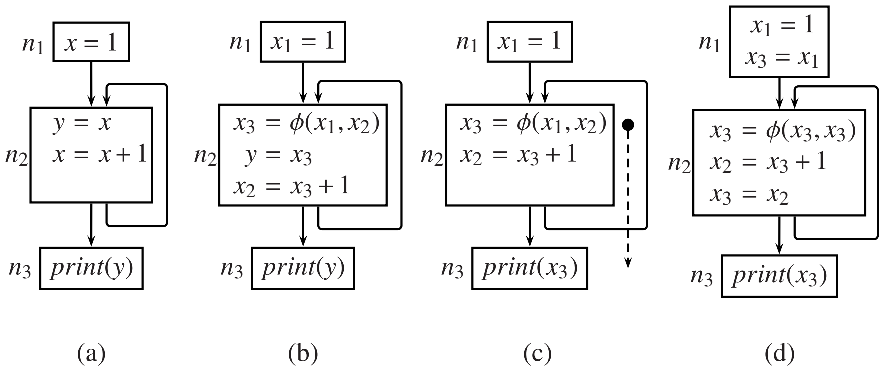


FIGURE 6.15
The lost copy problem.

copy statement across the critical edge. As a result this copy interferes with the live range of x_3 (shown using the dotted arrow).

The swap problem: The swap problem is illustrated in Figure 6.16 on the following page. In this case also the problem arises because the process of SSA destruction does not follow the semantics of ϕ-instructions. The program in Figure 6.16(c) is correct because of the implied translation of the ϕ-instruction to the simultaneous assignment $x_3 = y_3 \parallel y_3 = x_3$. The actual translation, however, replaces the simultaneous assignment by a sequence of assignments resulting in a dependence between them.

6.3.1 An Algorithm for SSA Destruction

Since the algorithm will require us to talk about variables which are related through ϕ-instructions, we introduce the following definitions.

DEFINITION 6.8 *A pair of variables are ϕ-related if they occur in the same ϕ-instruction.*

The idea of variables related through ϕ-instructions, which we have been informally calling variable versions, is captured through *ϕ-congruence*.

DEFINITION 6.9 *For a SSA variable x, ϕ-congruence(x) is the least set defined by the following two rules:*

1. *if y and x are ϕ-related, then y is in ϕ-congruence(x).*
2. *if y and z are ϕ-related and z is in ϕ-congruence(x) then y is in ϕ-congruence(x).*

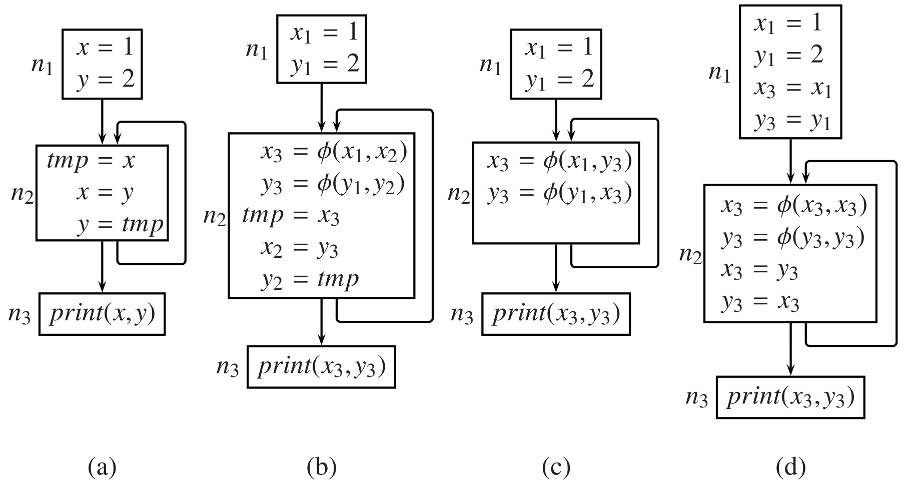


FIGURE 6.16
(a) The original program (b) After conversion to SSA form (c) After copy propagation (d) Destruction of the SSA form program through copying results in an incorrect program.

In other words, variables in ϕ-*congruence*(x) are directly or transitively connected to x through ϕ-instructions. Further, if y is in ϕ-*congruence*(x) then x is also in ϕ-*congruence*(y), and we say that x and y are in the same ϕ-*congruence* class. The notion of ϕ-*congruence* class is very similar to the notion of live range (or web) for programs which are not in SSA form.

For a program in CSSA form, all variables that are in the same ϕ-*congruence* class can be replaced by a common variable and the ϕ-instruction can be eliminated. Our objective now is to modify TSSA programs so that ϕ-instructions can be eliminated in the same way as CSSA programs, i.e., by renaming ϕ-*congruence* variables to the same name.

The reason why SSA-destruction through merging of ϕ-*congruent* variables poses problems in the case of TSSA programs can be better explained through live ranges. Observe in part (a) of Figure 6.17 on the next page that the ϕ-*congruent* variables x_1 and y interfere with each other and thus cannot be replaced by the same variable. Replacing both the variables by a single variable effectively kills the earlier definition. So a key idea might be to make the ϕ-*congruent* variables non-interfering by inserting copy statements. As shown in part (b) of the figure, this has been achieved for the example by inserting the copy statement $x'_1 = x_1$ in block n_1, $x'_2 = y$ in block n_2 and $x_3 = x'_3$ in n_3. Further, the ϕ-instruction has been rewritten to refer to the new variables x'_1, x'_2 and x'_3. Since the live ranges of these ϕ-variables are non-interfering, they can be renamed to a single variable x and the ϕ-instruction can be eliminated. The result is shown in Figure 6.17(c).

A further refinement of the idea is to minimize the number of copy instructions by

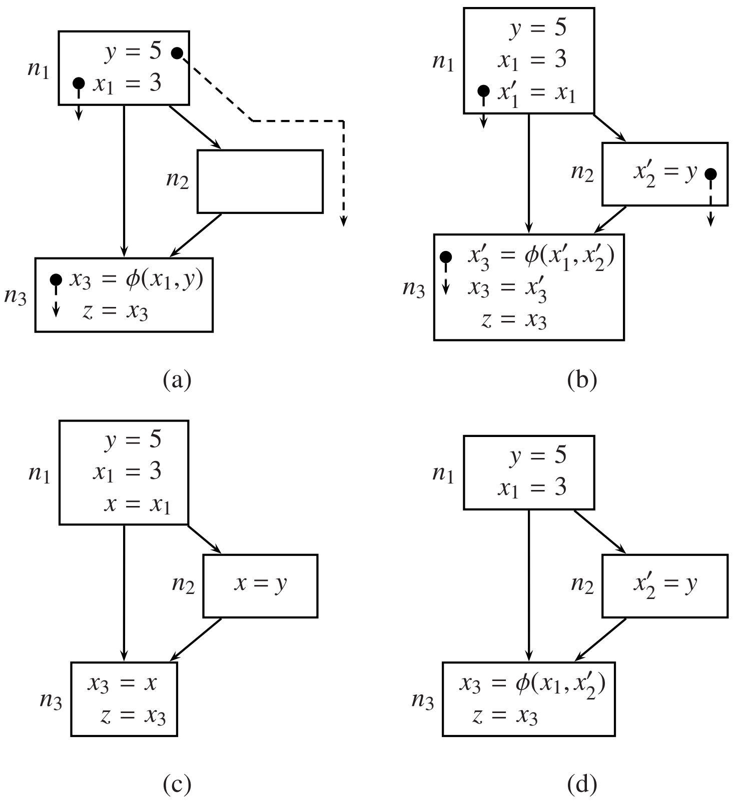


FIGURE 6.17
(a) Interference of Live ranges of the ϕ variables x_1 and y. The dashed lines represent live ranges of variables. (b) Breaking the interference through copy statements. The new ϕ variables x'_1 and x'_2 do not interfere. (c) Eliminating ϕ assignments now results in a correct program. (d) The copy statement for x_1 is redundant.

introducing the copy statement $x'_2 = y$ only. This also makes the live ranges of the ϕ-variables x_1, x'_2 and x_3 non-interfering. The result of this minimization is shown in Figure 6.17(d). Observe that inserting a copy statement $x'_1 = x_1$ at the end of block n_1 instead of $x'_2 = y$ does not break the interference between the live ranges of the variables x'_1 and y which are now *ϕ-congruent*.

The basis for the decision that the insertion of a single copy statement $x'_2 = y$ is enough is as follows. First notice that the only interference that has to be broken is between x_1 and y; x_3 does not interfere with these variables. Now y is live at the exit of n_1. Therefore insertion of a copy statement $x'_1 = x_1$ at the end of n_1 is useless since the new *ϕ-congruent* variables x'_1 and y will still continue to interfere. However insertion of the statement $x'_2 = y$ creates the *ϕ-congruent* variables x_1 and x'_2 which

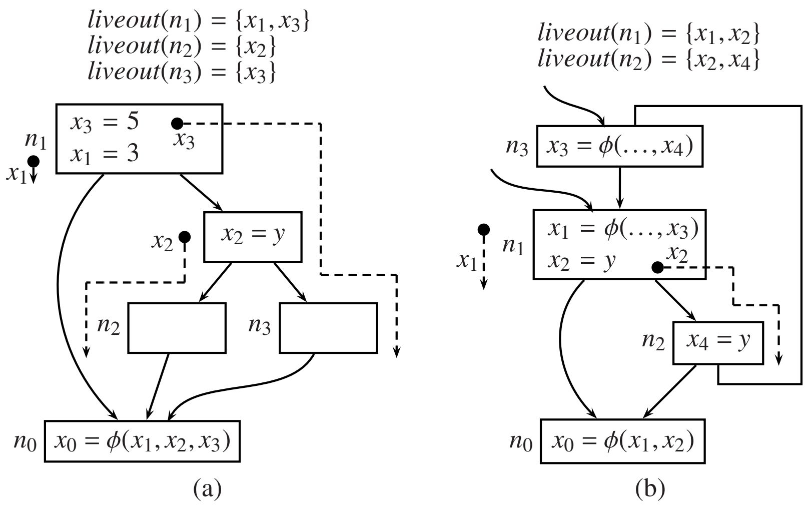


FIGURE 6.18
(a) Example to illustrate the condition when both ϕ-*congruent*$(x_i) \cap liveout(n_j) = \phi$ and ϕ-*congruent*$(x_j) \cap liveout(n_i) = \phi$. Live ranges and *liveout* sets are shown. (b) Example illustrating the condition ϕ-*congruent*$(x_i) \cap liveout(n_j) \neq \emptyset$ and ϕ-*congruent*$(x_j) \cap liveout(n_i) \neq \emptyset$.

do not interfere with each other.

The algorithm systematically creates ϕ-*congruent* classes that are non-interfering. Initially ϕ-*congruent*(x) is set to $\{x\}$ for every variable x. It considers in sequence all the ϕ-instructions. For each pair of operands x_i and x_j mentioned in a ϕ-instruction, it checks whether ϕ-*congruent*(x_i) interferes with ϕ-*congruent*(x_j). If it does not, ϕ-*congruent*(x_i) and ϕ-*congruent*(x_j) are merged into the same ϕ-*congruent* class. Otherwise copy statements are inserted to change the variables in the ϕ-instruction itself so that the ϕ-*congruence* classes of the new variables are non-interfering. The method for choosing the copy statement to be inserted is described below. In the description below, n is the block that contains the ϕ-instruction, n_i refers to the ith predecessor of n, and x_i and x_j are the interfering variables. We first consider the case when both x_i and x_j are arguments of the ϕ-instruction:

1. ϕ-*congruent*$(x_i) \cap liveout(n_j) \neq \emptyset$ and ϕ-*congruent*$(x_j) \cap liveout(n_i) = \emptyset$: This situation is similar to Figure 6.17 with y playing the role of x_i and x_1 playing the role of x_j. In this case, a copy statement $x_i' = x_i$ is needed at the end of n_i. x_i is marked to record this fact.

2. ϕ-*congruent*$(x_j) \cap liveout(n_i) \neq \emptyset$ and ϕ-*congruent*$(x_i) \cap liveout(n_j) = \emptyset$: This is similar to the previous situation and the variable x_j is marked.

3. ϕ-*congruent*$(x_i) \cap liveout(n_j) = \emptyset$ and ϕ-*congruent*$(x_j) \cap liveout(n_i) = \emptyset$: This situation is as shown in Figure 6.18(a), where x_2 and x_3 play the roles of x_i

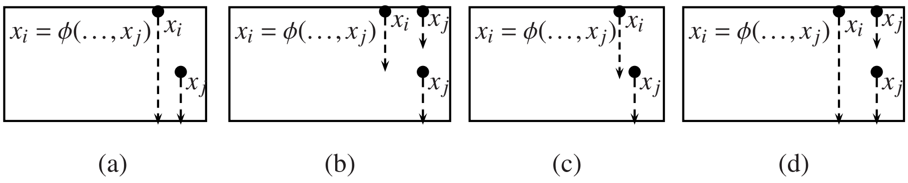


FIGURE 6.19
Interference between argument and result variables of a ϕ-instruction.

and x_j. In this case, either a copy statement $x'_2 = x_2$ at the end of n_2 or a copy statement $x'_3 = x'_3$ at the end of n_3 will break the interference. The better choice is to insert $x'_2 = x_2$ as it will also break the interference between x_3 and x_1. Since we cannot know this till we examine the pairs x_1 and x_2, we defer the insertion of the copy statement till we have examined all the pairs.

4. $\phi\text{-}congruent(x_i) \cap \mathsf{liveout}(n_j) \neq \emptyset$ and $\phi\text{-}congruent(x_j) \cap \mathsf{liveout}(n_i) \neq \emptyset$: This situation is represented by Figure 6.18(b) with x_1 and x_2 playing the roles of x_i and x_j. Note that x_4 is in $\phi\text{-}congruence(x_1)$. In this situation copy statements are needed for both x_i and x_j, so both the variables are marked.

When one of the interfering variables, say x_i, is the result and the other variable x_j is an argument of the ϕ-instruction, the situation is slightly more complex. The program point for inserting the copy statement involving the result variable is just after the ϕ-instruction. Consider the block which has the ϕ-instruction. As shown in Figure 6.19(a)–(d), there are four cases:

1. $\phi\text{-}congruent(x_i) \cap \mathsf{liveout}(n) \neq \emptyset$ and $\phi\text{-}congruent(x_j) \cap \mathsf{livein}(n) = \emptyset$: We have to insert the copy statement $x_i = x'_i$ just after the ϕ-instruction. The result variable of the ϕ-instruction is changed to x'_i.

2. $\phi\text{-}congruent(x_i) \cap \mathsf{liveout}(n) = \emptyset$ and $\phi\text{-}congruent(x_j) \cap \mathsf{livein}(n) \neq \emptyset$: As we shall see later when we re-examine the swap problem, this situation occurs when x_j is also the result of a subsequent ϕ-instruction. This requires the copy statement $x'_j = x_j$. The argument variable x_j is changed to x'_j.

3. $\phi\text{-}congruent(x_i) \cap \mathsf{liveout}(n_j) = \emptyset$ and $\phi\text{-}congruent(x_j) \cap \mathsf{liveout}(n_i) = \emptyset$: Here we can insert either a copy statement for x_i or for x_j. As explained earlier, the choice is deferred.

4. $\phi\text{-}congruent(x_i) \cap \mathsf{liveout}(n_j) \neq \emptyset$ and $\phi\text{-}congruent(x_j) \cap \mathsf{liveout}(n_i) \neq \emptyset$: In this situation copy statements are needed for both x_i and x_j.

The variables for which copy statements are to be inserted are added to a marked or deferred list as before. After all variables have been considered, we obtain two lists—a list of variables which have been marked and for which we need copy statements

Input: A CFG of a TSSA program.
Output: The corresponding program with the ϕ-instructions eliminated.
Algorithm:

```
 0  Initialize the φ-congruent class of each φ variable x to {x}.
 1  for each φ-instruction I do
 2  {   Initialize the marked and deferred lists to the empty list.
 3      for each pair $x_i$ and $x_j$ of argument variables in I do
 4          if $x_i$ and $x_j$ interfere, then proceed according to the four cases
            described in Section 6.3.1.
 5      for the result variable $x_i$ and each argument variable $x_j$ do
 6          if $x_i$ and $x_j$ interfere, then proceed according to the four cases
            described in Section 6.3.1.
 7      if a variable x is in the marked list, then remove all pairs which have x as
        one of the components from the deferred list.
 8      while there are elements in the deferred list do
 9      {   Select the variable x that appears maximum number of times in the
            deferred list.
10          Insert x in the marked list.
11          Remove all pairs which have x as one of the components from the
            deferred list
12      }
13      for each element x in the marked list do
14      {   Insert the copy statement $x' = x$ at the appropriate program
            point.
15          Update I to contain $x'$ instead of x.
16          Modify the interference graph to reflect this change.
17      }
18      Put all the variables of I in the same congruence class.
19  }
20  Eliminate all φ-instructions.
```

FIGURE 6.20
An algorithm for SSA destruction.

and the other a list of pair of variables, the choice from which has been deferred. We now choose a variable which appears the largest number of times in the deferred list and enter it in the marked list. All pairs in which it appears are removed from the deferred list. This is repeated till the deferred list is empty.

The last step consists of taking each variable from the marked list, inserting a copy statement for the variable, updating the ϕ-instruction, and updating the live ranges of the old and the new variables. The ϕ-instructions which now contain variables in the same *ϕ-congruent* class are now eliminated. The algorithm for breaking interference through insertion of copy statements is shown in Figure 6.20.

We now explain how the algorithm works on the lost-copy problem and the swap

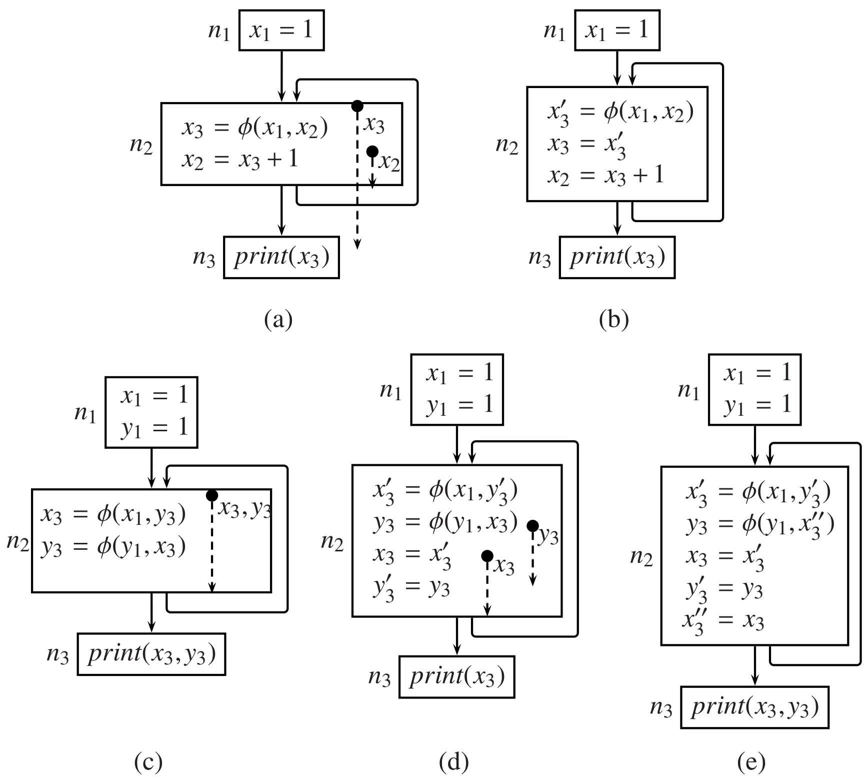


FIGURE 6.21
Illustrating the effect of the algorithm on the lost-copy (Figures (a) and (b)) and the swap problem (Figures (c), (d) and (e)).

problem. Consider the SSA corresponding to the lost-copy problem shown in part (a) of Figure 6.21. The live ranges of x_2 and x_3 interfere. While x_3 is in *liveout*(n_2), x_2 is not in *livein*(n_2). Therefore, as shown in Figure 6.21(b), a copy $x_3 = x'_3$ inserted after the ϕ-instruction breaks the interference.

The SSA form of the program illustrating the swap problem is shown in Figure 6.21(c). The live ranges of both x_3 and y_3 span the entire block n_2. Now consider the first assignment. Since x_3 is in *liveout*(n_2) a copy is needed for x_3. Similarly, since y_3 is in *livein*(n_2), a copy is needed for y_3. Now there is no interference between the variables of the first assignment. Considering the second assignment in Figure 6.21(d), we see that the live ranges of y_3 and x_3 interfere. However, neither y_3 is in *liveout*(n_2) nor x_3 is in *livein*(n_2). Therefore a copy statement for either x_3 or y_3 can be inserted. We choose x_3 and the result is shown in Figure 6.21(e).

6.3.2 SSA Destruction and Register Allocation

In the traditional sequence of events during compilation, the program in SSA form is destructed before register allocation takes place. However, as we shall explain later, the SSA form program has properties that are useful for register allocation through graph coloring. After register allocation is done, SSA destruction can be viewed as a form of coalescing registers.

Overview

The idea behind register allocation through graph coloring is as follows. The main data structure used is a graph called *interference graph*. An interference graph has a node for every live range in the program. Since, for programs in SSA form, each live range corresponds to a variable, we can also associate the nodes of the interference graph with variables. An edge is drawn between live ranges if they interfere, i.e., they range over common program points. In such a situation, the variables corresponding to the live ranges cannot be allocated the same register. Thus the problem of register allocation reduces to one of coloring the interference graph with a number of colors equal to the number of available registers so that no two adjacent nodes have the same color.

The *chromatic number* of a graph is the minimum number of colors required to color a graph as described above. If the chromatic number of a graph is larger than the number of available registers, then an attempt is made to reduce the interference by spilling, i.e., inserting stores after definitions and loads before uses. This effectively replaces a long live range by a number of shorter live ranges. The reduction in interference has the possible consequence of bringing down the chromatic number.

As shown in Figure 6.22, a typical register allocator that uses graph coloring repeats the following steps till the graph is colored.

1. Constructing or updating the interference graph.

2. Coalescing live ranges. If the live range of x ends with a copy statement to y, then the live ranges for x and y can be combined by replacing subsequent references to the uses of y in the live range by x and eliminating the copy. As this will change the interference graph, it has to be updated.

3. Attempt to color nodes. If this requires a node to be spilled, the interference graph has to be updated. So we go back to step 1.

There are two problems with this approach. While coalescing eliminates copy statements, it might also result in a graph with a larger chromatic number. Moreover, since every spill may not reduce the chromatic number of the graph, the interference graph may have to be constructed several times. This can be costly.

Now consider register allocation for a program in SSA form. We shall show that the interference graph of a SSA form program is a special kind of graph called chordal graph. The chromatic number of such a graph is the same as the size of the largest clique. Moreover, the largest clique in a SSA form program is equal to

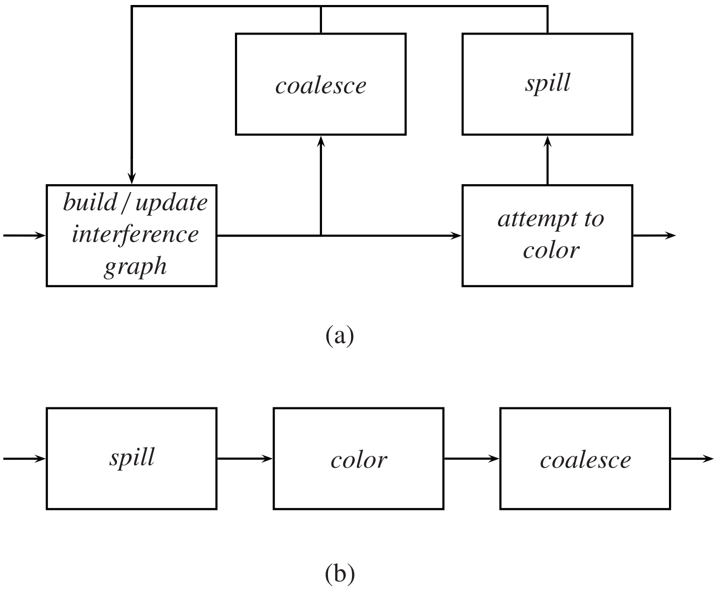


FIGURE 6.22
(a) Traditional register allocation. (b) Register allocation for SSA form programs.

the maximum number of variables live at a program point. So we can spill variables till the largest number of variables live at any program point equals the given number of registers. The interference graph of the resulting program is now guaranteed to be colorable with colors equal to the available number of registers. In fact, for the SSA form program, there is also an efficient algorithm to find the coloring.

While variables are replaced by registers after coloring, the ϕ-instructions are still present. As mentioned earlier, SSA destruction is a form of coalescing. For instance, assume that the ϕ-instructions in a basic block are:

$$R_1 = \phi(R_1, R_3)$$
$$R_2 = \phi(R_2, R_4)$$

For both the ϕ-instructions, the first operand is the same as the result and therefore no transfer of values need take place. An attempt is made to recolor R_3 to R_1 and R_4 to R_2. If this succeeds, then the ϕ-instructions can simply be eliminated. Otherwise copy instructions must be inserted.

The overall scheme for register allocation for SSA form programs is shown in Figure 6.22(b). Note that the process is not iterative. More importantly, all the above steps can be carried without constructing the interference graph.

Spilling

We now show certain properties of interference graphs of SSA form programs which makes it easy to determine whether enough variables have been spilled so as to make the interference graph colorable.

LEMMA 6.9

Let a variable x be live at a program point p. Then $\mathsf{def}(x) \geq p$.

PROOF Assume to the contrary that $\mathsf{def}(x) \not\geq p$. Since x is live at p, there is a path from p to some use of x, $\mathsf{use}(x)$. Then $\mathsf{def}(x) \not\geq \mathsf{use}(x)$ contradicting the SSA dominance property (Lemma 6.8). ∎

LEMMA 6.10

If x and y interfere either $\mathsf{def}(x) \geq \mathsf{def}(y)$ *or* $\mathsf{def}(y) \geq \mathsf{def}(x)$.

PROOF Since x and y interfere, there is a program point p where they are both live. From Lemma 6.9, both $\mathsf{def}(x)$ and $\mathsf{def}(y)$ dominate p. The result then follows from Observation 6.1. ∎

LEMMA 6.11

Assume $\mathsf{def}(x) \geq \mathsf{def}(y)$. *Then x and y interfere if and only if then x is live at* $\mathsf{def}(y)$.

PROOF The if part is obvious. For the only if part observe that under the condition $\mathsf{def}(x) \geq \mathsf{def}(y)$ if x is not live at $\mathsf{def}(y)$ then there is no point p such that x is live at p and there is a path from $\mathsf{def}(y)$ to p. It follows that x and y cannot interfere, leading to a contradiction. ∎

LEMMA 6.12

Let x — y and y — z be edges in the interference graph G of a SSA form program. Further, assume that x — z is not an edge in G. If $\mathsf{def}(x) \geq \mathsf{def}(y)$, *then* $\mathsf{def}(y) \geq \mathsf{def}(z)$.

PROOF Since x and y interfere and $\mathsf{def}(x) \geq \mathsf{def}(y)$, x must be live at $\mathsf{def}(y)$. Further, since y and z interfere, either $\mathsf{def}(y) \geq \mathsf{def}(z)$ or $\mathsf{def}(z) \geq \mathsf{def}(y)$. If $\mathsf{def}(z) \geq \mathsf{def}(y)$, then z must also be live at $\mathsf{def}(y)$ and x and z interfere. This is a contradiction and we must have $\mathsf{def}(y) \geq \mathsf{def}(z)$. ∎

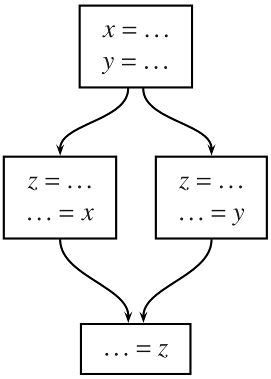


FIGURE 6.23
An example to show that Lemma 6.14 does not hold for programs not in SSA form.

LEMMA 6.13
Let G be the interference graph of a program in SSA form and let $C \subseteq G$ be a clique whose vertex set is $\{x_1, \ldots, x_n\}$. Then there is a permutation π of $\{1, \ldots, n\}$ such that $\mathit{def}(x_{\pi(1)}) \geq, \ldots, \geq \mathit{def}(x_{\pi(n)})$.

PROOF For $i, j \in \{1, \ldots n\}$, x_i and x_j interfere. Therefore from Lemma 6.10, either $\mathit{def}(x_i) \geq \mathit{def}(x_j)$ or $\mathit{def}(x_j) \geq \mathit{def}(x_i)$. π can be obtained by sorting $\{x_1, \ldots, x_n\}$ on $\geq$. ▮

LEMMA 6.14
Let G be the interference graph of a program in SSA form and let $C \subseteq G$ be a induced subgraph with vertex set $\{x_1, \ldots, x_n\}$. C is a clique if and only if there is a program point where $x_1, \ldots, x_n$ are all live.

PROOF The if part is trivial. Let C be a clique. By Lemma 6.13, there is a permutation π of $\{1, \ldots, n\}$ such that $\mathit{def}(x_{\pi(1)}) \geq \ldots \geq \mathit{def}(x_{\pi(n)})$. Therefore, from Lemma 6.11, $x_1, \ldots, x_n$ are all live just after $\mathit{def}(x_{\pi(n)})$. ▮

The above lemma does not hold for programs which are not in SSA form. Consider, for example, Figure 6.23. The live ranges of x, y and z form a clique in the interference graph. However, there is no program point where all three variables are live.

DEFINITION 6.10 *A* chordal graph *is a graph which does not have any induced cycle of length more than three.*

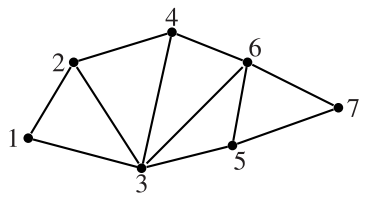


FIGURE 6.24
Example of a chordal graph.

Figure 6.24 gives an example of a chordal graph.

An interesting property of a chordal graph is its chromatic number is the same as the size of its largest clique. This can be seen in the example graph where the maximum clique size is three which is also the minimum number of colors required to color the graph. Therefore, if we can show that the interference graph of a program in SSA form is chordal, then, by Lemma 6.14, its chromatic number will be determined by the largest liveness set at any point in the program.

LEMMA 6.15
Let G be an interference graph of a program in SSA form. Then G is chordal.

PROOF Assume to the contrary that it is not. Then there will be at least one induced cycle $C = x_1, x_2, \ldots, x_n, x_1$ with $n \geq 4$. Now consider the sequence $x_1, x_2, \ldots, x_n$. Clearly, we do not have an edge x_i — x_j, such that $j > i+1$, or C would not be a cycle.

Since x_1 and x_2 interfere, either $def(x_1) \geq def(x_2)$ or $def(x_2) \geq def(x_1)$ by Lemma 6.10. Assume without loss of generality, $def(x_1) \geq def(x_2)$. Since x_2 — x_3 is an edge and x_1 — x_3 is not an edge, by Lemma 6.12, $def(x_2) \geq def(x_3)$. Using this idea, we can show by induction that there is a chain of dominance $def(x_1) \geq def(x_2) \geq \ldots \geq def(x_n)$.

Now since x_1 and x_n interfere, there is a program point p where both x_1 and x_n are live. Further, by Lemma 6.9, $def(x_n)$ dominates p. Because of the chain of dominances each $def(x_i)$ dominates p.

Consider a x_i, where i is not 1 or n. Since $def(x_i)$ dominates p and does not dominate $def(x_1)$, there is at least one path from $def(x_i)$ to p that does not have a definition of x_1. Thus x_1 is live at $def(x_i)$. This means there is an interference edge between 1 and i, leading to a contradiction. ∎

The Spilling Algorithm

The chromatic number of an SSA form program is the same as the maximum number of variables that are live at any point in the program. Hence we should ensure through spilling that the size of the set of live variables at any program point is no larger than the available number of registers. This is done without constructing the interference graph.

For every variable x we assume that there is a memory location x. These memory locations are not in contention for registers. A spill is an assignment $\mathsf{x} = x$. A reload of a variable is an assignment $x = \mathsf{x}$. For every basic block, the spilling algorithm decides:

1. The variables that get into registers at the entry of the basic block.

2. For every assignment statement, the variables that have to be spilled so that the operands on the right hand side of the assignment can be accommodated into registers, by reloading if necessary.

The spilling decision at a program point is based on the nearest distance at which a variable is subsequently used. We call this the next use of the variable and is captured through a function called *nextuse*. The next variable to be spilled at a program point is the one whose next use is the farthest. This is with the expectation that a free register can be found for the variable by the time the next use is reached.

For a basic block n and a variable x, the function *nextuse* is defined as follows:

$$\mathit{nextuse}(n,x) = \begin{cases} \infty & x \text{ is not live} \\ 0 & x \text{ is used in } n \\ 1 + \min\limits_{n' \in \mathit{succ}(n)} \mathit{nextuse}(n',x) & \end{cases}$$

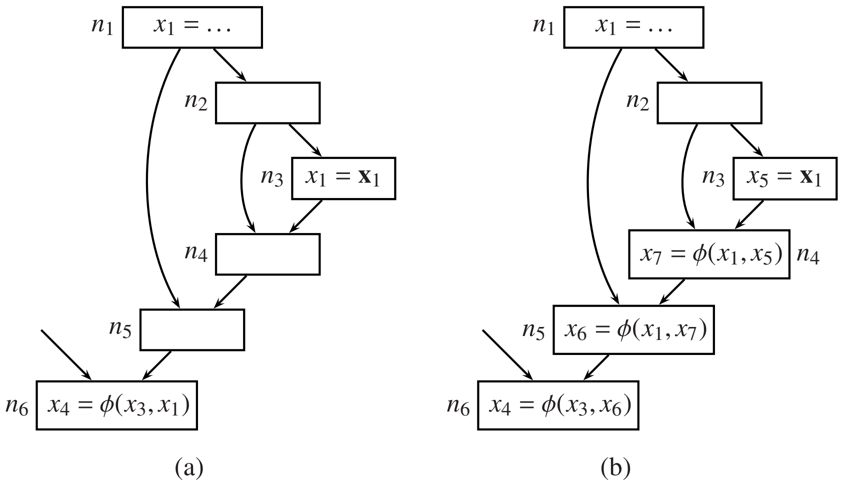
Assume that the number of available registers is k. At the entry of each block, we consider only the variables that are live and select k variables with the lowest *nextuse*. These are the variables to be held in registers at the entry of the block.

Similarly, for an assignment $p:\ x = \mathit{op}(y_1, \ldots y_i)$, if any of the variables $y_1, \ldots, y_i$ have to be brought into a register, the variable z with the highest value of $\mathit{nextuse}(p,z)$ value is spilled. Since the assignment to x takes place after the computation of $\mathit{op}(y_1, \ldots y_i)$, to find a register for x, we spill the variable z with the highest value of $\min_{p' \in \mathit{succ}(p)} \mathit{nextuse}(p',z)$.

Let us assume that based on the above consideration, we have decided to assign registers to the set of variables I at the beginning of a basic block n. Consider any predecessor n' of n. If O is the set of variables that have been decided to be kept in registers at the exit of the predecessor, we have to reload the variables in $I - O$ at the edge connecting n' and n.

Reloads introduce definitions that were not present in the original program. As a result of reloads, the program may not be in SSA form. However, for the PEO based coloring that we discuss later to be applicable, the program must be brought back to SSA form. We explain with an example how this is done.

Assume that in Figure 6.25(a), the variable x_1 had to be reloaded in block n_3 so that the program is no longer in SSA form. We have to bring the program back to SSA form and rewrite the uses of x_1. We start by calculating the iterated dominance frontier of the node n_3 which contains the reload. Next, the variable x_1 in n_3 is



n_1 $x_1 = \ldots$
n_2
n_3 $x_1 = \mathbf{x}_1$
n_4
n_5
n_6 $x_4 = \phi(x_3, x_1)$
(a)
n_1 $x_1 = \ldots$
n_2
n_3 $x_5 = \mathbf{x}_1$
$x_7 = \phi(x_1, x_5)$ n_4
n_5 $x_6 = \phi(x_1, x_7)$
n_6 $x_4 = \phi(x_3, x_6)$
(b)

FIGURE 6.25
Example to illustrate SSA reconstruction.

renamed to a new variable x_5 to bring back the single definition property of SSA form. We now rewrite the use occurrence of each variable to the definition reaching it. In the process ϕ-instructions are inserted wherever necessary.

To start with, we have to decide how to rewrite the use of x_1 in n_6. We observe that the predecessor n_5 of n_6 is in the iterated dominance frontier of n_3. Since this does not have a ϕ-instruction for x_1, we have to insert one. The result of this ϕ-instruction, x_6, is the definition reaching x_1 and thus will replace x_1. Now we recursively try to find the definitions reaching the first and second argument of the inserted ϕ-instruction at n_5. The definition reaching the first argument comes from x_1 at n_1. The search for the definition reaching the second argument results in the insertion of another ϕ-instruction at n_4. The result of this ϕ instruction, x_7, reaches the second argument of the ϕ function at n_5. The arguments of the ϕ function at n_4 are similarly found to be x_1 and x_5.

Coloring

We now describe properties of a chordal graph due to which it can be colored efficiently.

DEFINITION 6.11 *A node in a graph G is called simplicial if its neighboring nodes induce a clique in G.*

DEFINITION 6.12 *A* perfect elimination order (*PEO*) *of a graph G is an order based on elimination of the nodes of G as follows: At each step eliminate a simplicial node in the remaining graph.*

It can be verified that 1,2,4,3,6,5,7 is a *PEO* for the graph in Figure 6.24.

Let the maximum size of a clique in a graph be k. Consider the following procedure to color the graph with k colors using a *PEO* ordering: Starting with the empty graph, at each step we color and add a node, say n, in reverse order of *PEO*. While adding n we are assured that its neighbors form a clique of size at most $k-1$. Thus a color can always be found for n.

A graph is chordal if and only if it admits of a *PEO* ordering. If a graph has a *PEO*, not only is its colorability equal to the maximum size over all cliques of the graph, there is a polynomial time algorithm to obtain the coloring.

In the context of programs in SSA form, the following result gives a *PEO* of interference graphs and thus forms the basis of the coloring algorithm.

LEMMA 6.16
Let G be the interference graph of a program in SSA form. Consider an ordering of the nodes of the graph in which a node v is included only if all the nodes whose definitions are dominated by the definition of v have been already added to the ordering. Then the ordering is a PEO.

PROOF We have to show that v is simplicial at the point when it is included in the ordering. Consider two nodes u and x both of which interfere with v. Then we have to show that u and x interfere with each other.

Since the nodes v and u interfere, following Lemma 6.10 we have either *def*$(v) \geq$ *def*(u) or *def*$(u) \geq$ *def*(v). Since all definitions that are dominated by u have already been added to the ordering and eliminated from the interference graph, it must be the case that *def*$(u) \geq$ *def*(v). Therefore u is live at *def*(v). For similar reasons x is also live at *def*(u). Thus u and x interfere. ∎

The function *colorNode* uses the dominator based *PEO* to color the interference graph and is shown in Figure 6.26. The function is initially called with *Start*. It processes a node in the dominator tree before processing its children, and within a node it processes the statements in sequence. This ensures that the corresponding interference graph is colored in reverse *PEO* order. Observe that this happens without actually constructing the interference graph.

Coalescing by Recoloring

At the end of the coloring phase, the residency of variables in registers is as follows:

1. Some variables which do not participate in any ϕ-instruction could be assigned registers. If such a variable is live across a join node, it is held in a register

Input: A node n of the CFG of the SSA form program, the dominator tree of the CFG and the set of live variables at the entry of n. The function *color* returns the color of a already colored node.
Output: A coloring of the variables in n.
Algorithm:

```
function colorNode(n)
{   allocated = ∅
    for each variable x in livein(n) do
    {   allocated = allocated ∪ color(x)
        for each statement s in n in sequence do
            for each variable y used in s do
            {   if the last use of y is in s then
                    allocated = allocated − {color(y)}
                let x be the variable defined in s and c be an unallocated color
                in
                {   color(x) = c
                    allocated = allocated − {c}
                }
            }
    }
    for each child m of node n do colorNode(m)
}
```

FIGURE 6.26
Algorithm for coloring the interference graph.

along all paths reaching the join node. This situation is different for a ϕ variable which could be held in a register along one of the paths reaching the join point.

2. If the result of a ϕ-instruction is in a register, then it is ensured that the arguments of the ϕ-instruction are also in registers. Since the number of registers available for allocation to ϕ-variables is the same along all paths to a join node, the result of the coloring algorithm can always be altered to satisfy this condition. Similarly, if the result of a ϕ-instruction is a memory location, then the arguments of the ϕ-instruction are also made to reside memory locations.

The destruction of ϕ-instructions is viewed as a form of coalescing. Let *alloc* be an assignment of variables to registers and consider the ϕ-instruction

$$\mathit{alloc}(y) = \phi(\mathit{alloc}(x_1), \ldots, \mathit{alloc}(x_i), \ldots, \mathit{alloc}(x_n))$$

Destruction of this ϕ-instruction is the transfer of values from the registers $\mathit{alloc}(x_i)$ to $\mathit{alloc}(y)$ through register copies. If $\mathit{alloc}(y)$ is the same as $\mathit{alloc}(x_i)$, then no transfer of value needs take place. Otherwise, a copy statement has to be issued to transfer the value from $\mathit{alloc}(x_i)$ to $\mathit{alloc}(y)$. The problem is to color the interference graph of a program so as to minimize the transfer cost. This problem is called the *SSA-coalescing* problem. We shall now define it formally.

SSA coalescing

Call a pair of variables *ϕ-assigned*, if one of them is an argument and the other the result of a ϕ-instruction. Assume that we have a function c which associates a cost with every pair of edges that are *ϕ-assigned*. This cost takes into account (i) the cost of transferring a value x to y, assuming the variables are in separate registers, and (ii) the frequency of execution of the basic block which has the ϕ-instruction containing x and y. Given a coloring alloc, we define the cost of the coloring for the *ϕ-assigned* pair (x, y) as

$$\mathsf{cost}_{\mathsf{alloc}}(x,y) = \begin{cases} 0 & \mathsf{alloc}(x) = \mathsf{alloc}(y) \\ c(x,y) & \text{otherwise} \end{cases}$$

And the cost of the coloring for the entire program P as:

$$\mathsf{cost}_{\mathsf{alloc}}(P) = \sum_{(x,y)\in P,\ \phi-assigned(x,y)} \mathsf{cost}_{\mathsf{alloc}}(x,y)$$

DEFINITION 6.13 *Given a program in SSA form and its interference graph, the SSA-coalescing problem is to find a coloring* alloc *for which* $\mathsf{cost}_{\mathsf{alloc}}(P)$ *is minimum.*

Since this problem is NP-hard, we now present a heuristic for solving the problem. The idea is that we take the output of the coloring algorithm described before and modify the coloring so as to minimize the cost of transfer of values.

To start with, the algorithm forms groups of variables. Each group consists of maximal number of *ϕ-congruence* variables that are non-interfering. Interfering variables cannot be given the same color. Define the cost of each such group g as the cost of all the ϕ-related edges between variables in the group g, i.e.,

$$\mathsf{cost}_{\mathsf{group}}(g) = \sum_{x,y\in g,\ \phi-assigned(x,y)} \mathsf{cost}_{\mathsf{alloc}}(x,y)$$

Clearly, a successful coloring of a costlier group will yield more benefits in transfer costs. Therefore, the groups are sorted by decreasing cost and entered into a priority queue for recoloring in this order.

Now the groups in the priority queue are attempted to be recolored. We take each color c in turn and attempt a recoloring with c. Not all nodes in the group G can be colored with c. As a result, the recoloring attempt results in several subgroups $g_1, g_2, \ldots, g_n$ such that:

1. Each variable in each subgroup can be recolored to c.

2. Each g_i forms a *ϕ-congruence* class.

The subgroup g_i with maximum value of $\mathsf{cost}_{group}(g_i)$ is the candidate subgroup sg_c for the color c. This is done for all colors and the final decision is to chose c' with the maximum $sg_{c'}$. The corresponding subgroup's color is fixed at c' and never changed thereafter. This ensures the termination of the algorithm. A new group $G - sg_{c'}$ is formed and entered in the priority queue at an appropriate place depending on its cost. This process is repeated till the priority queue is empty.

To recolor a node with the color c, the algorithm checks that none of its neighbors have the color c. If this is true then the recoloring attempt is successful. Otherwise the algorithm recursively attempts to recolor the offending neighbor with a color different from c. The recoloring attempt fails if the color of the node is already fixed to a color that is different from the color for which the recoloring attempt is being made, or the node cannot be colored because of a lack of color.

It might appear that the recoloring step requires construction of the interference graph to, for example, determine non-interfering *ϕ-congruence* groups. However, this is not the case. Assume that we want to decide whether x interferes with y. We first determine whether $\mathsf{def}(x)$ and $\mathsf{def}(y)$ are related by a dominance relationship. If they are not, then by Lemma 6.10 they do not interfere. On the other hand, if there is a dominance relationship and $\mathsf{def}(x) \geq \mathsf{def}(y)$, for example, then by Lemma 6.11, x and y interfere if x is live at $\mathsf{def}(y)$.

Register Copies

The last step in the method is to arrange for transfer of value for *ϕ-congruent* variables that could not be colored with the same color. To take into account that ϕ-instructions within the same basic block are to be simultaneously executed, we consider all the ϕ-instructions in the basic block together. As an example consider the ϕ-instructions

$$\begin{aligned} R_1 &= \phi(\ldots, R_2, \ldots) \\ R_2 &= \phi(\ldots, R_2, \ldots) \\ R_3 &= \phi(\ldots, R_5, \ldots) \\ R_4 &= \phi(\ldots, R_3, \ldots) \\ R_5 &= \phi(\ldots, R_4, \ldots) \end{aligned}$$

In the example, we limit ourselves to the registers at one of the argument positions. We can represent this transfer of value through a graph shown in Figure 6.27. While in the example, we have restricted ourselves to the registers at one of the argument positions, the graph has to be extended to other argument positions. The resulting graph is called the *register transfer graph*. Now we generate instructions to effect the value transfers suggested by the register transfer graph. Each step is repeated as many times as possible.

1. If there is a edge $R_i \rightarrow R_j$ in the graph such that R_j does not have any out edges, then a copy statement $R_j = R_i$ is issued. This is illustrated by the edge $R_2 \rightarrow R_1$ in the example, for which a copy statement $R_1 = R_2$ has to be issued.

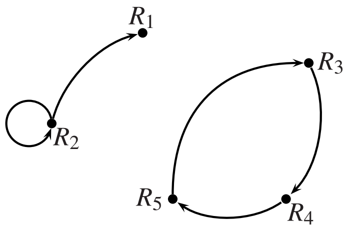


FIGURE 6.27
Graph indicating transfer of values between registers.

2. Now the register transfer graph will consist of one or more cycles. The cycles of length 1 like R_2 are eliminated.
3. The cyclic transfers values indicated by loops of length more than one like R_3, R_4, R_5 can be effected in more than one way:
   (a) Transfer using a free register as a temporary. For the example, assuming R_0 is a free register, the instructions generated are:

$$\begin{aligned} R_0 &= R_3 \\ R_3 &= R_4 \\ R_4 &= R_5 \\ R_5 &= R_0 \end{aligned}$$

   (b) If there is no free register, then pairwise swap operations can be used. For the example, the transfer can be effected through the following swaps:

$$\begin{aligned} &\mathit{swap}\ R_3\ \ R_4 \\ &\mathit{swap}\ R_4\ \ R_5 \\ &\mathit{swap}\ R_5\ \ R_3 \end{aligned}$$

   If the underlying machine does not directly support a swap operation, it may be simulated through *xor* operations.

6.4 Summary and Concluding Remarks

In this chapter we described a useful intermediate representation of programs called the SSA form. In this representation every variable has exactly one definition, and this definition dominates each use of the variable. The number of def-use chains in SSA form programs is much smaller than corresponding programs not in SSA form. As a consequence, optimizations performed on SSA form programs are faster. Apart from the sparseness of def-use chains, a program in SSA form also has other interesting properties that could be used in various applications. An example that was presented is register allocation.

The transformation of a program to SSA form involves finding program points where ϕ-functions are to be inserted. These points are identified by iterated dominance frontiers. After ϕ-functions are inserted, variables are renamed to satisfy the single definition property. Both these steps can be done efficiently. The transformation of programs to their SSA form can be thought of as being the result of some form of data flow analysis. Destruction of SSA form programs is based on creating *ϕ-congruence* variables that are also non-interfering. This is through insertion of copy statements. The *ϕ-congruence* variables are then renamed to the same variable and the ϕ-instruction is removed.

We also presented register allocation as a way of destructing SSA form programs. Register allocation of SSA form programs through graph coloring is convenient because the interference graphs of such programs have properties that enable us to (a) determine how much spilling is required so that the interference graph becomes colorable, and (b) obtain a coloring. Removal of ϕ-instructions is through register coalescing. Interestingly, all these steps can be done without actually constructing the interference graph.

SSA-based optimizations are more difficult when the entity involved in the optimization is not a variable, as in the redundancy elimination optimizations. The problem is that the expressions representing redundant computations may not be lexically the same; they may have different versions of a variable. Detecting these occurrences and eliminating the redundant ones by exploiting the sparseness of def-use chains is not straightforward.

6.5 Bibliographic Notes

The earliest papers on SSA form are by Rosen, Wegman and Zadeck [85] and Alpern, Wegman and Zadeck [8]. The first comprehensive method for construction of SSA form programs is by Cytron, Ferrante, Rosen, Wegman, and Zadeck [28]. The method described in this chapter is based on this paper. A later paper by Sreedhar and Gao [95] gives a linear time algorithm for placing ϕ-instructions using a data structure called DJ-graphs. Both methods involve finding the dominator tree of a program. Lengauer and Tarjan [68] give a fast algorithm for finding dominators in a graph. The methods above construct minimal SSA. Choi, Cytron and Ferrante [22] present a method to create programs in pruned SSA form and Briggs, Cooper, Harvey, and Simpson [18] describe construction of semi-pruned SSA.

While Cytron, Ferrante, Rosen, Wegman, and Zadeck [28] discuss destruction of

SSA, the method that they suggest has shortcomings. The method discussed here is based on the work by Sreedhar, Dz-Ching Ju, Gillies and Santhanam [96]. Briggs, Cooper, Harvey, and Simpson [18] discuss SSA-destruction by placing copy statements along edges. The method for SSA destruction by register allocation is by Hack [41] and by Hack, Grund, and Goos [42].

Many applications of SSA form can be found in literature. Rosen, Wegman and Zadeck [85] describe a method to eliminate redundant computations among expressions that may not be lexically identical. Alpern, Wegman and Zadeck [8] describe how to conservatively detect equality of variables in a program. Kennedy, Chan, Liu, Lo, Tu, and Chow [58] use SSA for partial redundancy elimination and Wegman and Zadeck [103] for conditional constant propagation. As mentioned earlier, Hack, Grund, and Goos [42] perform register allocation over SSA form programs while Knobe and Sarkar [64] use a variation of SSA form for parallelization.

Appel [11] describes the similarity between SSA programs and functional programs written in continuation passing style and Dhamdhere, Rosen and Zadeck [31] point out the difficulties in using SSA form for partial redundancy elimination.