Optimizing Compilers Exercises 4

```
void h(int a, int b)
{
        int     x;
        int     y;

        if (a < b)
                x = a * b;
        else
                x = a - b;
        y = a * b;

        return x * y;
}
```

Figure 1: Partial redundancy elimination example 1.

```
void h(int a, int b, int n)
{
        int     s = 0;
        int     i = 0;

        do {
                s += a * b;
                i += 1;
        } while (i < n);

        return s;
}
```

Figure 2: Partial redundancy elimination example 2.

1. Show how SSAPRE optimizes the program in Figure 1.
2. Show how SSAPRE optimizes the program in Figure 2.

Solutions

1. The parts of the program relevant for SSAPRE are shown in Figure 3.
   The major steps taken by SSAPRE to optimise the program are as follows.

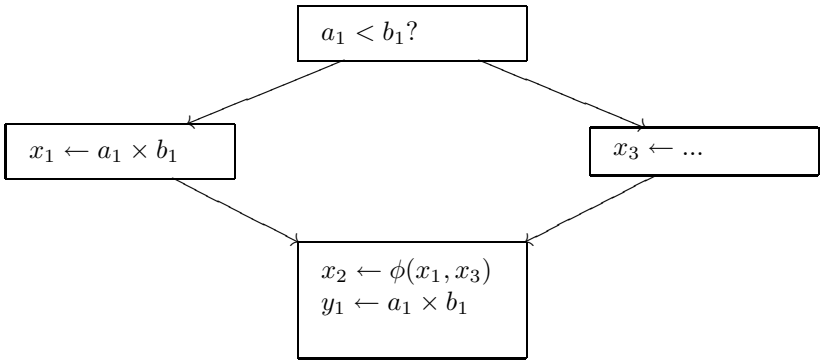

Figure 3: Relevant parts translated to SSA-form.

(a) Insert Φ-functions. They are inserted in the iterated dominance frontiers of all evaluations of an expression plus all places where there is a ϕ-function for one of the operands of the expression. Note that the Φ-functions are should be put just after the last ϕ-function in a particular basic block. In our example, a Φ-function is inserted in the last basic block. See Figure 4.

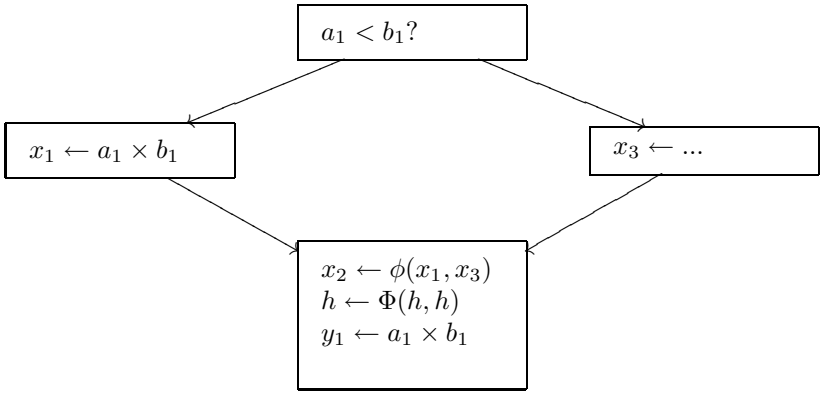

Figure 4: After insertion of a Φ-function.

(b) Rename. Conceptually, all variables are now renamed again just as in translation to SSA-form (this is not done explicitly in a practical implementation). In addition, the hyphotetical variable h is introduced. There is an assignment to h at each evaluation of the expression. The are three occurence types of an expression:

i. Φ-occurence,

ii. Real occurence, and

iii. Φ-operand occurence.

The dominator tree is traversed and each occurence of the expression is inspected. The first occurence found is $x_1 \leftarrow a_1 \times b_1$. Since the expression stack of $a \times b$ is empty (if it was not empty, a new version of h would be assigned if the operands don't match, as explained in the book), a new version of h is assigned and the real occurence is pushed on the expression stack. The next occurence is the Φ-operand in the same basic block. The version of each operand matches those of the expression on the top of expression stack (ie, $a_1 \times b_1$) so a pointer from the operand to the real occurence is set. Leaving this basic block, the expression stack is popped and now becomes empty.

We assume next that we visit the basic block with the Φ-function (we might just as well visit the block with $x_3 \leftarrow ...$ first). At the Φ-function, a new version of h is assigned and the Φ is pushed on the stack. The next occurence to inspect is real occurence $y_1 \leftarrow a_1 \times b_1$. Since the versions of a and b match those at the Φ-function, a pointer from the real occurencee to the Φ-function is set. Next, the real occurence is pushed on the expression stack. Since we now reach the end of the program (or more properly the exit basic block) we check to see whether there is a Φ-function on the top of the expression stack. There is not, but if there would have been one, it would have been marked as *not downsafe*. Leaving this basic block, the expression stack is popped and again (happens to) become empty.

In the last basic block to check, there is a Φ-function operand and it is marked as $\perp$ since there is no valid expression on the stack.

(c) Propagation of *not downsafe*. No propagation is done here since the only Φ-function is downsafe.

(d) Propagation of *can_ be_ available*. No propagation is done here since there is only one Φ-function, for which *can_ be_ available* is set to true.

(e) Propagation of *later*. Since there is an operand of the Φ-function which is "has real use" (ie, the operand gets its value from a real occurence), *later* is set to false for our Φ-function.

(f) Compute *will_ be_ available*. It becomes true based on the attributes of the Φ-function: *can_ be_ available* and not *later*.

(g) Finalise. Here the dominator tree is traversed again and the $\perp$ operand is replaced by a newly inserted $a_1 \times b_1$ and the Φ-function will save the value of $a_1 \times b_1$ in a temporary which will be used by the real occurence following it, as described by Algorithm 13.3 in the book.

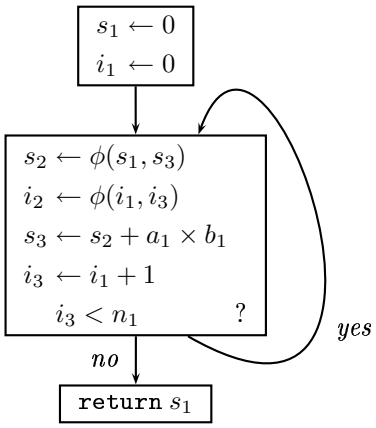

Figure 5: Program in Figure 2 translated to SSA form.

2. The major steps taken by SSAPRE to optimise the program in Figure 2 with respect to the expression $a \times b$ are as follows.

   (a) Insert Φ-functions. There is a real occurence in the loop and the dominance frontier of this vertex is that vertex itself. So, a Φ-function is inserted after the ϕ-functions. See Figure 6.

   (b) Rename. When traversing the dominator tree, the first occurence reached is the Φ-function operand in the first basic block. The Expression stack for $a \times b$ is empty so the operand is $\perp$.

   In the loop vertex, the first occurence is the Φ-function, for which a new version of h is assigned, and the Φ-function is pushed on the expression stack. Next the only real occurence is checked. The operands have not been modified since the Φ-function, so a pointer from the real occurence to the Φ-function is set and no new version of h is assigned. The real occurence is pushed on the stack. The last occurence checked in this vertex is the Φ-function operand corresponding to the flow graph arc denoted *yes*. This operand has a real use (on the top of the stack) and so a pointer is set to it. Nothing happens in the last vertex.

   (c) Propagation of *not downsafe*. No propagation is done here since the only Φ-function is downsafe (it was not marked downsafe when we reached the last vertex).

   (d) Propagation of *can_be_available*. No propagation is done here since there is only one Φ-function, for which *can_be_available* is set to true, since it is downsafe.

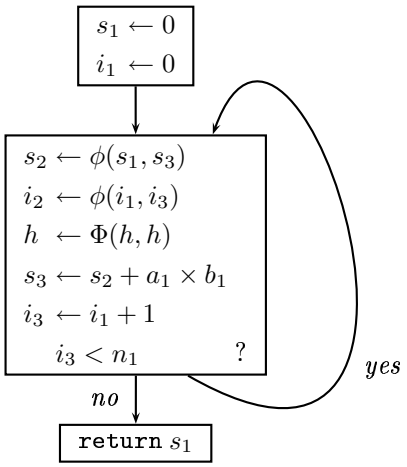

Figure 6: Program in Figure 5 after Φ insertion.

(e) Propagation of *later*. Since there is an operand of the Φ-function which is "has real use" (ie, the operand gets its value from a real occurence), *later* is set to false for our Φ-function.

(f) Compute *will_be_available*. It becomes true based on the attributes of the Φ-function: *can_be_available* and not *later*.

(g) Finalise. Here the dominator tree is traversed again and the $\perp$ operand is replaced by a newly inserted $a_1 \times b_1$ and the Φ-function will save the value of $a_1 \times b_1$ in a temporary which will be used by the real occurence following it, again as described by Algorithm 13.3 in the book. The result is seen in Figure 7. Note that the ϕ-function for h actually can be removed, but that is done by another optimisation.

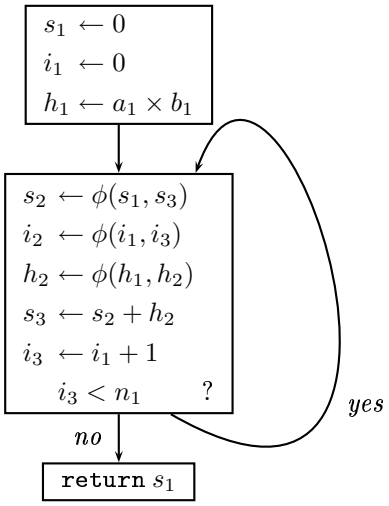

Figure 7: Program in Figure 6 after finalise.