1 Cascaded dataflow analyses

Some analyses lead to optimizations that enable more optimization. For example, CSE plus copy propagation can lead to more CSE. To avoid rerunning analyses after optimizations, we can design analyses to take into account optimizations that will be performed. For example, we can change live variable analysis to take into account the removal of dead code. Compared to CSE, local variable numbering is also a cascaded analysis, at least within the scope of an extended basic block.

2 Partial redundancy elimination

CSE eliminates computation of *fully redundant* expressions: those computed on all paths leading to a node. Partially redundant expressions are those computed at least twice along some path, but not necessarily all paths. Partial redundancy elimination (PRE) eliminates these partially redundant expressions. PRE subsumes CSE and loop-invariant code motion.

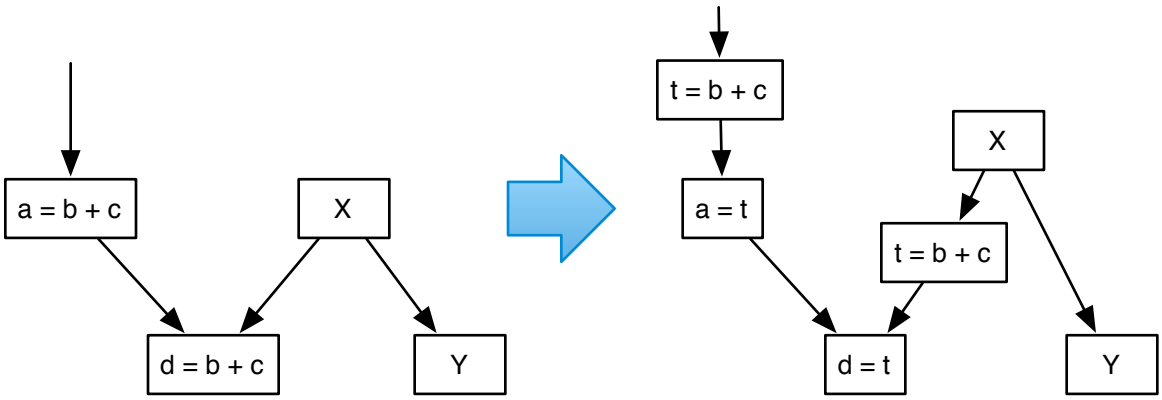

Figure 1: Partial redundancy elimination

Figure 1 shows an example of PRE. The computation `b+c` is redundant along some paths but not others. To make it fully redundant, we place computation of `b+c` onto earlier edges so that it has always been computed at each point where it is needed.

2.1 Lazy code motion

The idea of lazy code motion is to eliminate all redundant computations while avoid creating any unnecessary computation: computations are moved earlier in the CFG. Further, we want to make sure that although the computations are moved earlier in the CFG, they are postponed as long as possible, to avoid creating register pressure.

The approach is to first identify candidate locations where the partially redundant expression could have been moved in order to make it fully redundant, without creating extra computations. Then among these candidates we choose the one that comes latest along each path that needs it.

2.2 Anticipated expressions

The *anticipated expressions* analysis (also known as *very busy expressions*) find expressions that are needed along every path leaving a given node. If an expression is needed along every path leaving the node, then there can be no wasted computation if the expression is moved to that node.

This is a backward analysis, in which the dataflow values are sets of expressions and the meet operator is $\cap$.

Once we know the anticipated expressions at each node, we tentatively place computations of these expressions and use an available expressions analysis to find expressions that are fully redundant under the assumption that the anticipated expressions are computed everywhere anticipated. These fully redundant expressions are the expressions to which we can apply the PRE optimization.

2.3 Postponable expressions

At this point we know some set of nodes where the expression can be moved, and we know where it is used. We need to pick a set of edges that separate these two parts of the CFG, and put the computation of the expression on those edges. We want to postpone the computation as long as possible. The *postponable expressions* analysis finds expressions e that are are anticipated at program point p but not yet used: every path from the start to p contains an anticipation of e and no use before p. This is a forward analysis with meet operator $\cap$.

Once postponable expressions have been computed, certain edges form a *frontier* where the expression transitions from postponable to not postponable. It is on these edges that the new node computing the expressions is placed.