1 Loops, dominators, natural loops, control trees, dominator analysis, SSA conversion

Loop optimizations are a fruitful class of optimizations because most execution time in programs is spent in loops: a 90/10 split between loops and non-loops is typical. Consequently, many loop optimizations have been developed: for example, loop-invariant code motion, loop unrolling, loop peeling, strength reduction using induction variables, removing bounds checks, and loop tiling.

When should we do loop optimizations? In source or high-level IR form, loops are easy to recognize, but there may be many kinds of loops, all of which can benefit from the same optimizations. Furthermore, loop optimizations often benefit from other optimizations that we want to do on a lower-level representation. We want to be able to interleave loop optimizations with these other optimizations.

In order to do loop optimizations, the first problem we must tackle is to define what we mean by "a loop" at the IR level, and to efficiently find these loops.

2 Definition of a loop

At the level of a control-flow graph, a *loop* is a set of nodes that form a strongly connected subgraph: every loop node is reachable from every other node following edges within the subgraph. In addition, there is one distinguished node called the *loop header*. There are no entering edges to the loop from the rest of the CFG except to the loop header. There may be any number (including 0) nodes with outgoing edges, however; these are *loop exit nodes*.

For example, the CFG in Figure 1 contains three loops as indicated, with header nodes are marked in the corresponding color.

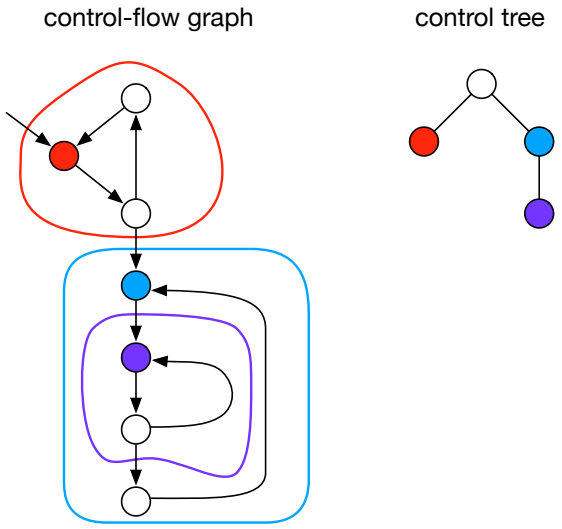

Figure 1: Loops in a CFG

A given CFG may contain multiple loops, and loops, as sets of nodes, may contain each other. If the nodes in loop 1 are a strict superset of the nodes in loop 2, we say that these are nested loops, with loop 2 nested inside loop 1.

Assuming that any two loops only intersect when one is nested inside the other, the loops in the CFG form a *control tree* in which the nodes are loops and the edges define the nesting relationship.

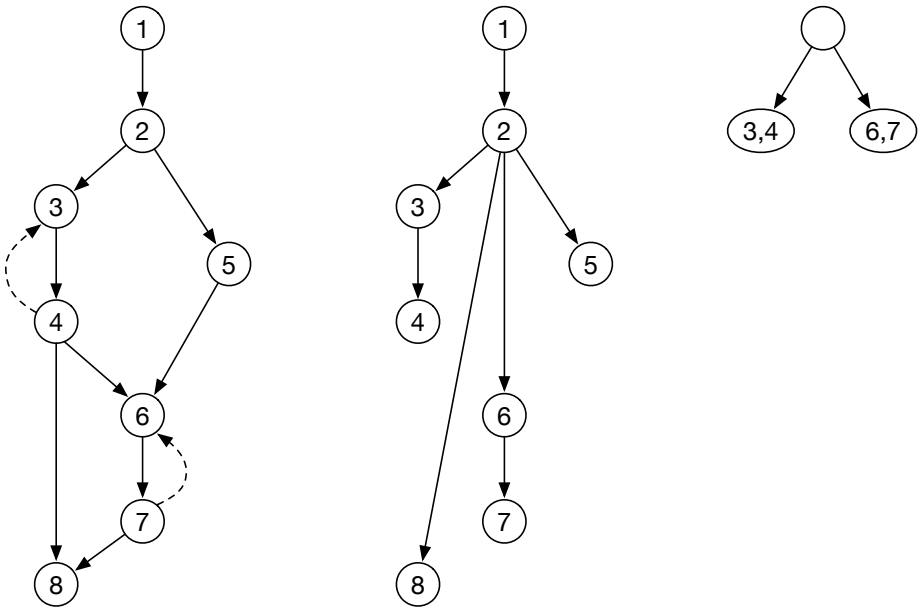

Figure 2: A control-flow graph (left), its dominator tree (middle), and its control tree (right). Back edges in the control-flow graph, indicating loops, are dashed.

3 Dominators

Control-flow analysis builds on the key idea of *dominators*. A node A *dominates* another node B (written A dom B) if every path from the start node of the CFG to node B includes A. An edge from A to B is called a *forward edge* if A dom B and a *back edge* if B dom A. Every loop must contain at least one back edge.

The domination relation has some interesting properties. It is reflexive, because a node dominates itself. It is also obviously transitive. Finally, it is antisymmetric. If A dom B and B dom A, they must be the same node. If they were different, then A dom B means you can get to A without going through B, so B can't dominate A. These three properties mean that domination is a partial order.

In addition, two nodes cannot both dominate a third node with there being some domination relationship between the two nodes. Suppose for purpose of contradiction that A dom C and B dom C but neither A dom B nor B dom A. But getting to C requires going through both A and B in some order. Suppose it's A and then B. But in that case if A doesn't dominate B, there must be a path from start to B to C that doesn't go through A. So A couldn't dominate C.

These properties of the domination relationship imply that domination is essentially tree-like. In particular, the Hasse diagram for a CFG is always a tree rooted at the start node. For example, Figure 2 shows a control-flow graph on the left and its corresponding dominator tree on the right.

4 Natural loops

Each back edge from node n to node h in the CFG defines a *natural loop* with h as its header node. The natural loop is a strongly connected subgraph that contains both n and h. It consists of the nodes that are dominated by h and that can reach n without going through h. Thus, the CFG in Figure 2 contains two natural loops.

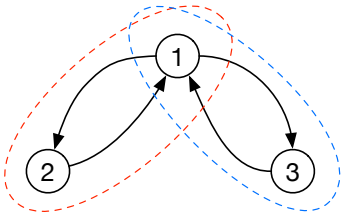

Figure 3: Overlapping natural loops

5 Dominator dataflow analysis

The domination relation can be computed efficiently using a dataflow analysis. Define $out(n)$ to be the nodes that dominate n. Since domination is reflexive, $out(n)$ includes n itself. Other nodes that dominate n must also dominate all predecessors of n, since otherwise there would be a path to n that misses them. From this reasoning, we obtain the following dataflow equations:

$$out(n) = \{n\} \cup \bigcap_{n' \prec n} out(n') \qquad \text{(for nodes other than the start node)}$$
$$out(\mathbf{START}) = \{\mathbf{START}\}$$

We can solve this as a forward analysis starting with all variables initialized to the set of all nodes. This initial value is the top element of the lattice in which $\cap$ is the meet operator. Note that the transfer function is monotonic and distributive.

6 Postdominators

We say that node A *postdominates* node B if all paths from B to an exit node go through node A. This happens exactly when A dominates B in the transposed (dual) CFG, in which all edge directions are reversed and start and exit nodes are interchanged.

7 Finding natural loops and control trees

The first step in construction of the program control tree is to compute the domination relation, which can be done as described above.

Then, for each back edge $n \to h$ (that is, h dom n), we run a depth-first search starting from n in the transposed CFG. However, the search is stopped when node h is encountered. The set of reached nodes that are dominated by h are the natural loop.

Note that it's possible in a general CFG to reach nodes that are not dominated by h, but in this case the nodes are unreachable code that doesn't need to be included in the loop (or in the program).

Having found the natural loops, we can assemble them into a control tree. Observe that two natural loops can be disjoint or can be nested. Or they can intersect without being nested, but only if they share the same header node. In this case, we can treat them as a single loop for the purpose of constructing the control tree. Figure 3 shows an example of two natural loops that share a header and can be merged together in this fashion.

Once overlapping natural loops are merged, all loops are either disjoint or nested. The control tree then falls out directly from subset inclusion on the various loops.

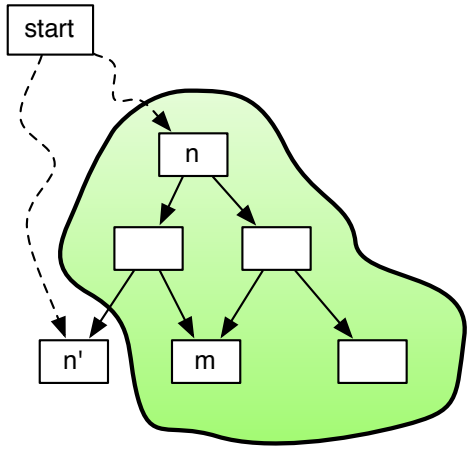

Figure 4: Dominance frontier

8 Using dominators for SSA conversion

We have seen that SSA form is a convenient form for optimization and analysis of code. However, converting code to SSA form is itself not trivial. Conversion can be broken down into two steps:

1. Insert uses of φ-functions for various variables at the beginnings of basic blocks; that is, for a variable x that needs to use a φ-function, we insert $x = \varphi(x, x)$.
2. When enough uses of φ have been inserted, each use of a variable is reached by just one definition. Therefore, we can give each definition its own unique variable name, and rename all the corresponding uses reached by that definition. Note that the definitions of a variable include the new definitions using φ that were inserted in step 1.

In Step 1 we don't want to use φ more often than necessary, because this will create unnecessary variable names and impede optimization.

Path Convergence Criterion
A φ function is needed for variable x at node m if:

- There are node n_1 and n_2 that both define x (where $n_1 \neq n_2$)
- There are two paths of length 1 or greater, $n_1 \rightarrow^* m$ and $n_2 \rightarrow^* m$, such that these paths only intersect at m itself.

Intuitively, φ is needed at a node when it is the earliest place that two paths from different definitions converge. The path convergence criterion identifies such earliest convergence points, but it does not naturally lead to an efficient algorithm for finding these locations.

Instead, dominators can be used to efficiently insert φ exactly where the path convergence criterion says it should. The intuition is that if a node n defines a variable x, the path convergence criterion will not demand that φ be used for x at any node dominated by n where the definition reaches. As illustrated in Figure 4, the nodes inside the colored boundary are all dominated by node n, and φ is not needed. Unless there is another definition of x inside the dominated region, the definition at n must be the only one that reaches the node. Thus, node m does not need a φ-function for x, even though it has two predecessors. On the other hand, node n' does need a φ-function, because it has a predecessor dominated by n, yet it is not itself dominated by n.

We say that an edge crossing from a node dominated by n to a node not dominated by n lies on the *dominance frontier* for n. And we consider the destination node of that edge (such as n') to also lie on the

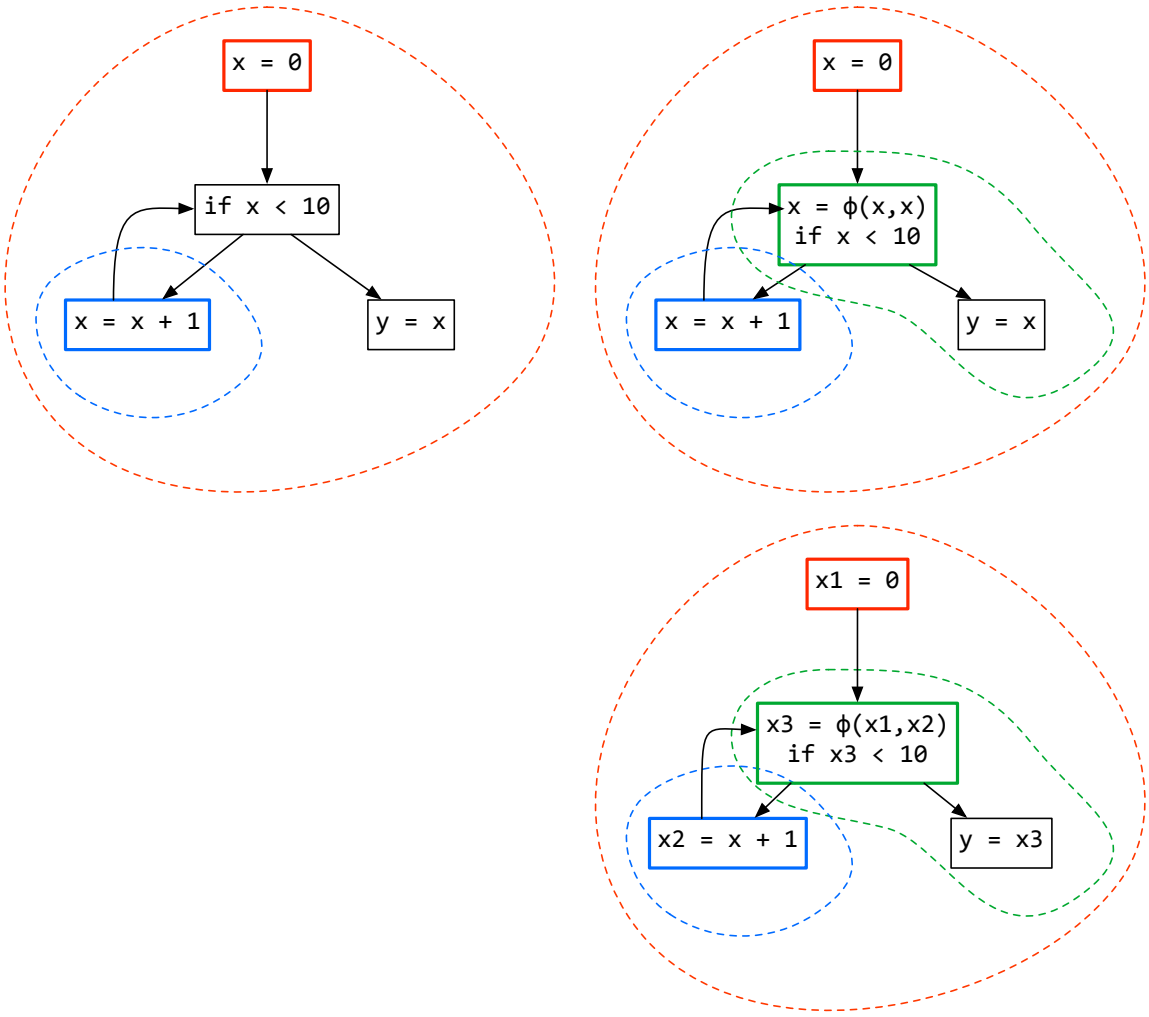

Figure 5: SSA conversion using iterated dominance frontiers

dominance frontier. The nodes lying on the dominance frontier of n are exactly the nodes that to have the new definition $x = \varphi(x, x)$ added to them.

Notice that adding this definition indeed adds a new definition to the control flow graph. And this new definition has its own dominance frontier that may induce additional definitions using φ. However, this *iterated dominance frontier* process will eventually converge on a set of φ definitions such that every node on the dominance frontier of every definition of a variable x starts with a corresponding definition $x = \varphi(x, x)$.

Figure 5 shows a small example of this process. We start with the code on the upper left, which has two defs of `x` and is therefore not in SSA form. Each of these defs has a dominated region indicated by the dashed bubble of the corresponding color. The edge from node `x=x+1` to node `if x<10` crosses the boundary of the region dominated by `x=x+1`, so node `if x<10` is on the dominance frontier of this assignment. Therefore it acquires a new (green) def using φ, as shown in the upper right. This new def has its own dominated region, and we again look for nodes on its dominance frontier. There are none, so we can number the different defs of `x` and rename all the uses accordingly to arrive at the SSA code on the lower right.

9 Computing the dominance frontier

Let $DF(n)$ denote the dominance frontier of node n: the set of nodes not dominated by n, but with a predecessor dominated by n. Assuming we have computed the dominance relation, we can easily check whether any given node lies on the dominance frontier of node n. This observation leads to an obvious quadratic algorithm.

However, we can make the computation of dominance frontiers more efficient by observing that every node on the dominance frontier of n is either:

- a direct successor of n
- on the dominance frontier of some child c of n in the dominator tree.

Thus, to compute the dominance frontier of n, we recursively compute the dominance frontier of each of n's children in the dominator tree, then iterate over all the nodes in the childrens' dominance frontiers and over the direct successors, checking whether each of these nodes is on the dominance frontier of n itself.