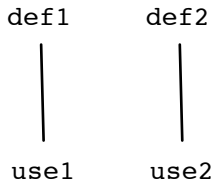
1 Reaching definitions

Register allocation allocates registers to variables. But sometimes allocating just one register to a variable is not important. For example, consider the following code:

```
int i = 1
  ⋮
i = i + 1
  ⋮
a[i] = 0
```

There are two definitions of `i` in this code, and two uses. It is defined at the first and second lines shown, and used as the second and third lines. If we refer to these defs and uses as `def1` and `def2`, and `use1` and `use2` respectively, and then draw a graph in which each definition (def) is connected to each use that it can affect, we get a disjoint graph:



This suggests we can use two different registers to hold `i`, since the two uses of the variable don't communicate. Each of these uses has a smaller live range than the whole variable, so it may help us do a better job of register allocation.

Register allocation is one motivation for the analysis known as *reaching definitions*, though other optimizations also need this analysis. Reaching definitions attempts to determine which definitions may *reach* each node in the CFG. A definition reaches a node if there is a path from the definition to the node, along which the defined variable is never redefined.

1.1 Dataflow analysis

We can set up reaching definitions as a dataflow analysis. Since there is only one definition per node, we can represent the set of definitions reaching a node as a set of nodes. A definition reaches a node if it may reach any incoming edge to the node:

$$in(n) = \bigcup_{n' \prec n} out(n')$$

A definition reaches the exiting edges of a node if it reaches the incoming edges and is not overwritten by the node, or if it is defined by the node:

$$out(n) = gen(n) \cup (in(n) - kill(n))$$

With $defs(x)$ denoting the set of nodes that define variable x, $gen(n)$ and $kill(n)$ are defined very straighforwardly:

n	$gen(n)$	$kill(n)$
$x = e$	n	$defs(x)$
everything else	$\emptyset$	$\emptyset$

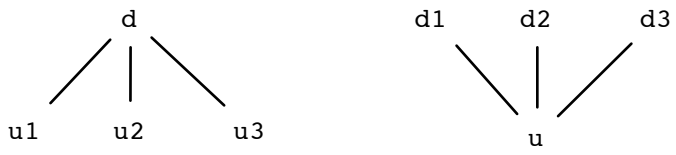


Figure 1: DU-chain and UD-chain

Viewing this analysis through the lens of dataflow analysis frameworks, we can see that it works correctly and finds the meet-over-paths solution. It is a forward analysis where the meet operation is $\cup$, so the ordering on lattice values is $\supseteq$ and the top value is $\emptyset$. The height of the lattice is the number of definitions in the CFG, which is bounded by the number of nodes. So we have a finite-height lower semilattice with a top element. The transfer functions have the standard form we have already analyzed, which is monotonic and distributive, so the analysis is guaranteed to converge on the meet-over-paths solution.

2 Webs

Using the reaching definitions for a CFG, we can analyze how defs and uses relate for a given variable. Suppose that for a given variable, we construct an undirected graph we will call the DU-graph, in which there is a node for each def of that variable and a node for each distinct use (if a CFG node both uses and defines a variable, the def and the use will be represented as distinct nodes in the DU-graph). In this there is an edge from a def to a use if the def reaches the node containing the use. The graph above is an instance of the DU-graph for the variable `i`.

Any connected component of the DU-graph graph represents a set of defs and uses that ought to agree on where the variable will be stored. For example, we showed that the DU-graph for variable `i` had two connected components. We refer to these connected components as *webs*. Webs are the natural unit of register allocation; in general, their liveness is less than that of variables, so using them avoids creating spurious edges in the inference graph. Therefore the graph coloring problem is less constrained and the compiler can do a better job of allocating registers.

A standard way to think about construction of webs is in terms of *DU-chains* and *UD-chains*, depicted in Figure 1. A DU-chain is a subgraph of the DU-graph connecting just one def to all of the uses it reaches. A UD-chain is a subgraph connecting a use to each of the defs that reach it. If a UD-chain and a DU-chain intersect, they must be part of the same web. So a web is a maximal union of intersecting DU- and UD-chains.

Once reaching definitions have been determined, webs can be computed efficiently using a disjoint-set-union data structure (union-find with path compression). If a def reaches a use, then both must be in the same web. The algorithm works by finding a representative node in each web. Each node keeps track of its representative node during the algorithm; initially every node is its own representative. For each def–use edge, one node's representative node is changed to point to the other's, with path compression used to flatten the chain of representative-node pointers. The running time of this algorithm is nearly linear in the number of def–use edges.

3 Single static assignment form

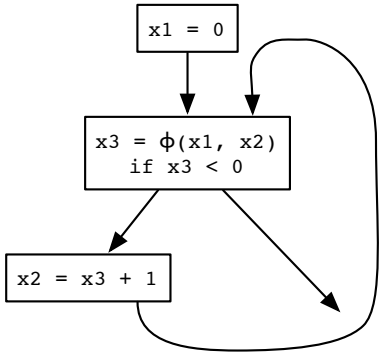
Different compiler optimizations can enable each other, and in general we want the compiler to run multiple optimizations repeatedly until no further improvement is possible. However, optimizations can also invalidate analyses needed by other optimizations. Rerunning these analyses repeatedly makes the compiler more complex and slower. Reaching definitions is a good example of an analysis that ends up being run repeatedly.

Modern compilers typically use a slightly different CFG representation than the one we have been studying. It is called *single static assignment form*, or SSA for short. The idea is to avoid nontrivial UD-chains: each variable in SSA has exactly one def, and therefore each use does too.

For example, consider the following code:

```
x = 0
while (x < 10) {
  x = x + 1
}
```

This code has two definitions of x, so in SSA form it must have at least two distinct variables representing the original x. The resulting SSA CFG would be something like this:



Boxes have been drawn around the basic blocks in this CFG, rather than around nodes. In the SSA literature it is standard to work with basic blocks as the CFG nodes rather than with individual statements as we have been doing. However, there is not much difference, and everything we have to say can be translated to the statement-per-node approach.

Note that the variable x has become three variables x1, x2, and x3 in this CFG. Variables x1 and x2 correspond to the two definitions in the original program. Variable x3 arises because the use x<0 has a non-trivial UD-chain: both of the other definitions reach it. To preserve the property that each variable has just one definition, SSA form introduces a fictitious function φ (phi) that picks among its arguments according to the edge along which control reached the current basic block. All uses of φ must occur at the beginning of their basic block. Operationally, we can think of φ as implemented by simple assignment statements that occur along the incoming edges to the node: in the example, an assignment x2=x1 on the left incoming edge from the top, and an assignment x2=x3 on the right incoming edge.

With the use of the φ-function, every use has exactly one def, and the webs are all disjoint DU-chains that can be (and are) given distinct names.

3.1 Using SSA

The advantage of SSA is that it simplifies analysis and optimization. For example, consider the following four optimizations, which all become simpler in SSA form.

- *Dead code removal*. A definition x=e is dead iff there are no uses of x. We assume that for each def in the program, we keep track of the set of corresponding uses. If that set is empty, the definition is dead and can be removed. All use sets for variables in expression e can then be updated correspondingly to remove this use.
- *Constant propagation*. An assignment x=c, where c is a constant, can be propagated by replacing each use of x with c. This works because there is only one definition of x. The assignment is then dead code and can be removed as described above.
- *Copy propagation*. An assignment x=y can similarly be propagated, just like the copy propagation case.

- *Unreachable code*. Unreachable code can be removed, updating all use sets for variables used in the code.

In fact, given code in SSA form and use sets for each variable, all four of these optimizations can be performed in an interleaved fashion without further analysis. By contrast, performing interleaved optimizations in the original non-SSA form would require redoing dataflow analyses between optimization passes.

3.2 Converting to SSA

This improvement does not come for free, however. We have to convert our CFG to SSA form, which is tricky to do efficiently. The challenge is where to insert φ-functions so that every use has exactly one def. Once this property has been achieved, the resulting webs can be renamed (e.g., by adding subscripts to their variable name) accordingly.

A simple-minded approach is just to insert φ-functions for every variable at the head of every basic block with more than one predecessor. But this creates a more complex CFG than necessary, with extra variables that slow down optimization.

When does basic block z truly need to have the φ-function for a variable a; that is, inserting $\mathtt{a} = \varphi(\mathtt{a}, \mathtt{a})$ at its beginning? This question can be answered using the *path convergence* criterion: the $\varphi(\mathtt{a}, \mathtt{a})$ is needed when:

- There exist two nodes x and y that define variable a.
- There are nonempty paths from x and y to z that are disjoint except at the final z node. Along these two paths, a is defined *only* at x and y.

This rule implies that z must be a node with multiple predecessors, because otherwise two paths would have to share the single predecessor and therefore would not be disjoint. It similarly implies that z can also appear in the *middle* of *one* of the paths from x and y to z, but cannot be in the middle of both.

Note that for evaluating the path convergence criterion, we consider the start node of the CFG to implicitly define every variable, representing its initial value (initialized or uninitialized) on entry.

Although path convergence gives us a clear criterion for when to insert a φ-function, it is expensive to evaluate directly. SSA conversion is therefore usually done using a dominator analysis.

4 Dominators

SSA conversion uses 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 without there being some domination relationship between the two dominating 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.

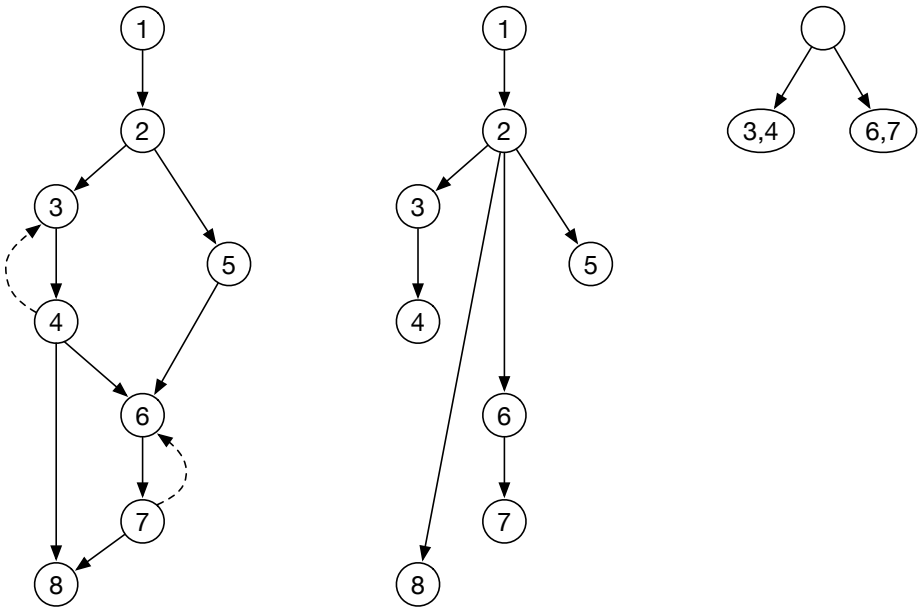


Figure 2: A control-flow graph (left), its dominator tree (middle), and its control tree (right). Back edges in the control-flow graph, indicating loops, are dashed.

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 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 $x = \varphi(x, x)$.

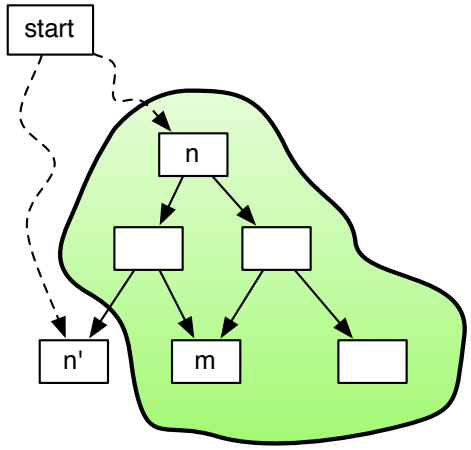


Figure 3: Dominance frontier

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 3, 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 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 4 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

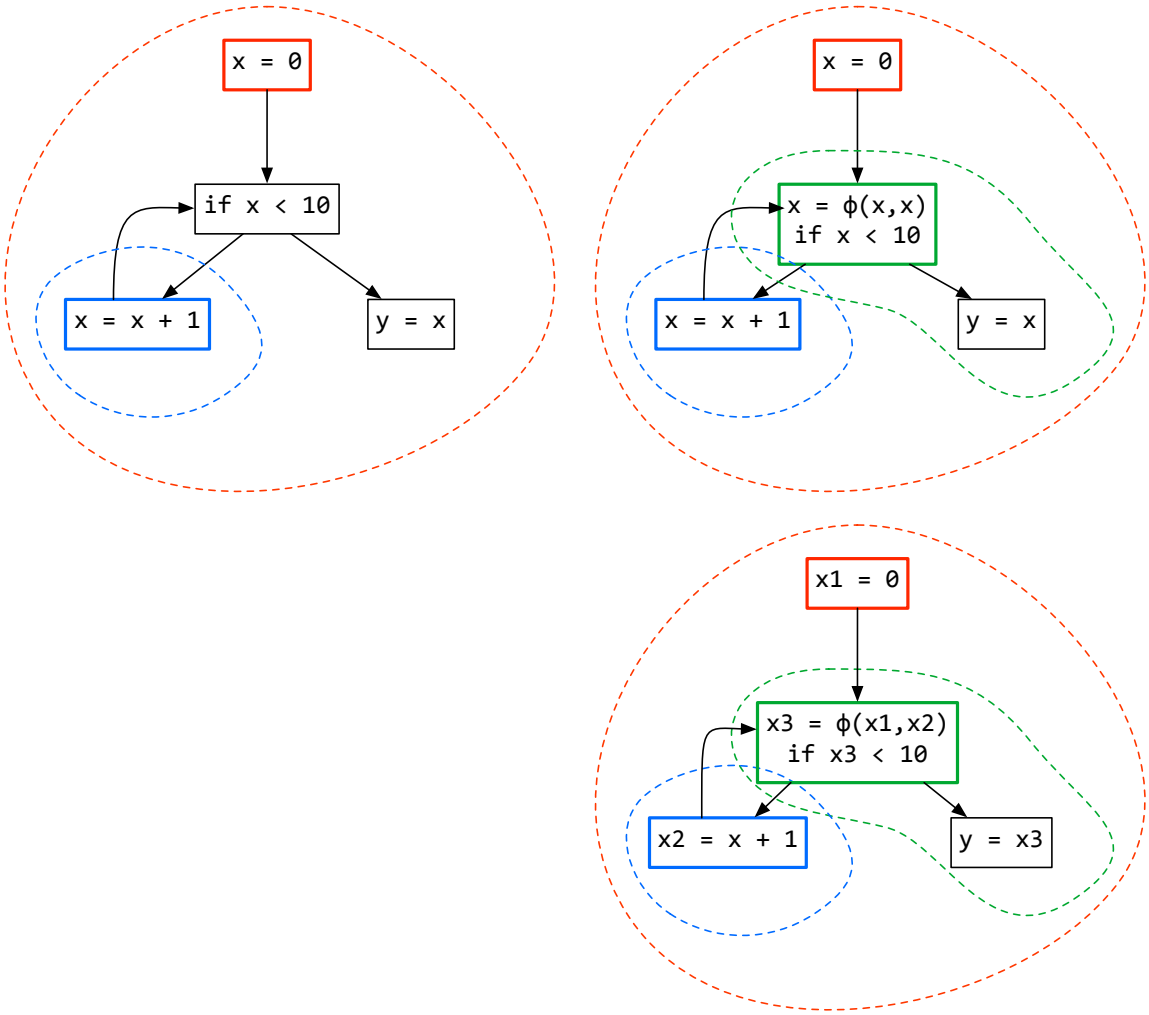


Figure 4: SSA conversion using iterated dominance frontiers

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.

8 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.