CS 4120/5120 Lecture 23 — Reaching definitions, webs, SSA — 21 March 2016

Lecturer: Andrew Myers

## 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:

$$\begin{tikzcd}
\texttt{def1} \arrow[d, dash] & \texttt{def2} \arrow[d, dash] \\
\texttt{use1} & \texttt{use2}
\end{tikzcd}$$

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 | $\emptyset$ | $\emptyset$ |

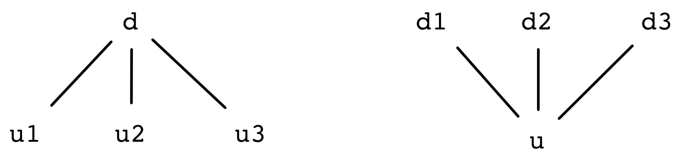


Figure 1: DU-chain and UD-chain

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).

## 3 Single static assignment form

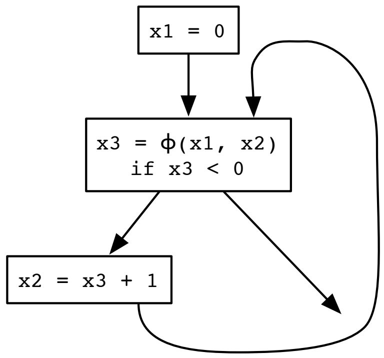
Different compiler optimizations can enable each other, and in general we want the compiler to run multiple optimizations repeatedly until no further improvement is possible. However, optimizations can also invalidate analyses needed by other optimizations. Rerunning these analyses repeatedly makes the 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 $\varphi$ (phi) that picks among its arguments according to the edge along which control reached the current basic block. All uses of $\varphi$ must occur at the beginning of their basic block. Operationally, we can think of $\varphi$ 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 $\varphi$-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 $\varphi$-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 $\varphi$-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 $\varphi$-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.

Although path convergence gives us a clear criterion for when to insert a $\varphi$-function, it is expensive to evaluate directly. SSA conversion is therefore usually done using control flow analysis, which we will talk about shortly.

## 4 Conditional constant propagation

Constant propagation and constant folding together form an important optimization that moves computations to compile time rather than run time, that allows deletion of code that is never executed, and that simplifies the programs enabling further optimizations. *Conditional* constant propagation is a version of constant propagation that takes advantage of different information along different exit edges from conditional nodes. *Sparse* conditional constant propagation, introduced by Wegman and Zadeck, takes advantage of SSA to keep track of less information about each node. Although constant propagation, copy propagation, constant folding, and unreachable code elimination can achieve similar effects when used together, conditional constant propagation enables strictly more optimization.

The goal of conditional constant propagation is to simplify code like the following:

```
x = 1
if (x<2) {
  y = 5+x
} else {
  y = f(x)
}
z = y*y
```

into code that uses constants:

```
x=1
y=6
z=36
```

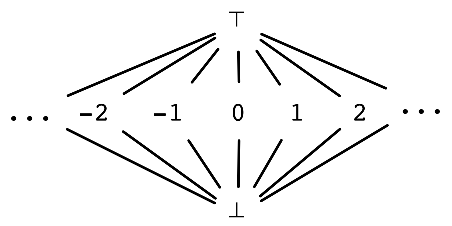


Figure 2: Conditional constant propagation lattice

The assignments to x and y may also be dead code at this point. Notice that we have removed the conditional entirely because its guard expression is constant, and this facilitates the optimization of the assignment to z.

The analysis is a forward analysis. For each variable in the program, we keep track of its possible values using the lattice in Figure 2, due to Killdall. The value $\top$ represents an undefined variable, or one that we have not yet seen a definition for. The value $\bot$ represents an "overdefined" variable, one that has more than one possible value and that therefore cannot be predicted at compile time. Therefore, for a given constant $c$, merging a path where no definition has been seen yet with one on which a constant value $c$ has been seen yields $c \sqcap \top = c$ since it's safe to replace an undefined value with $c$. Two different constant values combine to yield $\bot$ (that is, $c_1 \sqcap c_2 = \bot$), and once a variable is non-constant, it remains so: $\bot \sqcap c = \bot$.

We can keep track of the possible values of a whole set of $k$ variables $x_1, \ldots, x_k$ in a tuple of elements of this lattice, which we will write $(v_1, \ldots, v_k)$. The tuple of elements is of course itself a lattice with top element $(\top, \ldots, \top)$.

In the non-SSA representation, we need to keep track of this tuple of elements at every point in the code. In the SSA representation, there is only one definition of each variable, so a variable is either constant or not; we can keep track of a single tuple $(v_1, \ldots, v_k)$ for the entire analysis.

In either representation, we want to keep track of one more piece of information for each node: whether the node is unreachable. We will let $u$ represent unreachability of a node, with $u = \top$ meaning the node is unreachable, and $u = \bot$ meaning it may be reachable. Treating $\top$ as "true" and $\bot$ as "false", the meet operation is conjunction: $u_1 \sqcap u_2 = u_1 \wedge u_2$. For the non-SSA representation, the dataflow values will be tuples $(u, (v_1, \ldots, v_k))$, with the usual componentwise lattice ordering lifted from the orderings on $u$ and $v_i$.

The transfer functions $F_n(u, (\vec{v}))$ are defined as follows:

- $F(\top, (\vec{v})) = (\top, (\vec{\top}))$ because we need not consider definitions made by unreachable code.
- $F(\bot, (v_1, \ldots, v_k))) = (v'_1, \ldots, v'_k)$ where for all $i$, $v'_i = v_i$ unless $n$ defines $x_i$. In this case the assignment $x_i = e$ is interpreted abstractly using the current values for the variables occurring in $e$. For example, $2 + 2 = 4$, but $2 + \bot = \bot$, and $2 + \top = \top$, and $f(x) = \bot$.
- One exception to the previous case is conditionals. For a conditional node containing `if` $e$, the analysis interprets $e$ abstractly as in the previous case. If the result is $\bot$, the same information is propagated on both exiting edges. But if the result is `true` or `false`, the information propagated on the edge not taken is $(\top, (\vec{\top}))$, because that code is unreachable from this `if`.

For example, applying this (non-sparse) analysis to the example from earlier, we obtain the result shown in Figure 3.

Notice that the assignment $y = f(x)$ is marked as unreachable, so it can be removed along with the conditional. Once the analysis completes, all definitions $x_i = e$ for which $v_i = c$ on the outgoing edge can be replaced with $x = c$. Dead code removal can then be used to remove unnecessary assignments.

When the analysis is done in SSA form, we keep track of just one tuple of values $(v_1, \ldots, v_k)$, and only unreachability is propagated through the CFG. Dead code removal can then be performed easily in the manner outlined earlier, assuming that use sets are updated as we rewrite definitions to use constants.

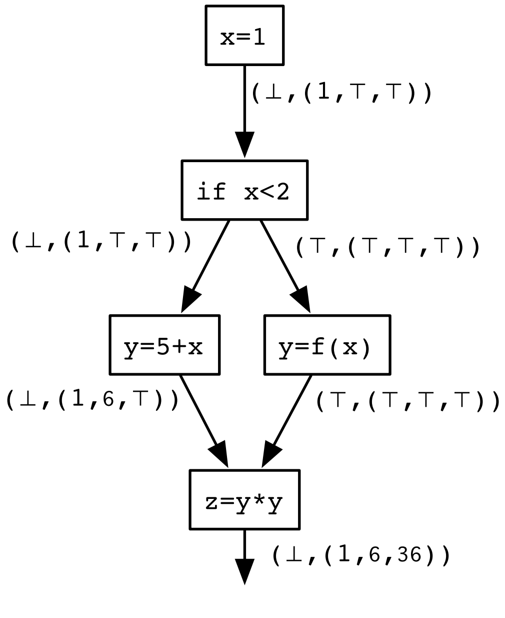


Figure 3: Example of conditional constant propagation

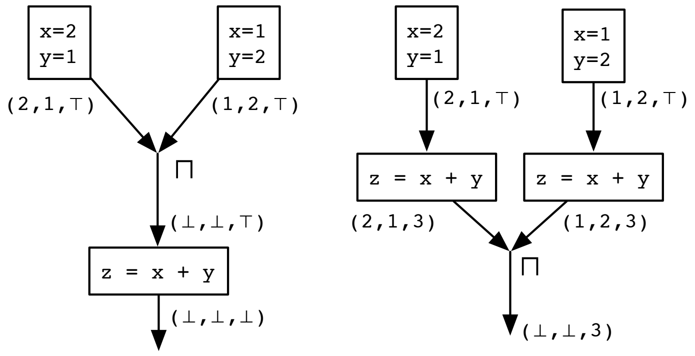


Figure 4: Conditional constant propagation does not give the meet-over-paths solution

### 4.1 Solution quality

In contrast to the analyses we've seen earlier, conditional constant propagation does not achieve a MOP solution. The reason is that the transfer functions are not distributive. To see this, consider Figure 4, in which the meet is taken both before a join point in control flow (corresponding to the way that dataflow analysis works) and after (corresponding to the meet-over-paths criterion). Nevertheless, it's an important and effective optimizaton.