Section Notes on Dataflow Analysis

November 2, 2003

For many compiler optimizations, the compiler must determine the truth of some property at a certain point in the program to perform the optimization safely. For example, to eliminate an assignment statement `x := e` as dead code, the compiler must know that `x` is not *live* immediately after the statement, *ie.* no other statement reads the value written into `x` before it is re-assigned (determining this information is known as *liveness analysis*). While finding this information for a single basic block (local optimization) is fairly straightforward, things get more interesting when you try to handle conditionals and loops (global optimization). Luckily, dataflow analysis is a general, well-understood method by which we can solve this and many other program analysis problems.

A key property of dataflow analysis is that the information we need to compute often depends on information transferred from preceding or subsequent program points. A *transfer function* succinctly captures this flow of information through statements. For liveness analysis, let $L_{in}(x, s)$ be true if variable x is live immediately before statement s, and let $L_{out}(x, s)$ be true if variable x is live immediately after statement s. For a single basic block, we can define our transfer function as follows:

$$\begin{aligned} L_{in}(x, s) &= L_{out}(x, s) \text{ if s does not mention x} \\ L_{in}(x, s) &= \textbf{true} \text{ if s uses x} \\ L_{in}(x, x := e) &= \textbf{false} \text{ if e does not mention x} \end{aligned}$$

First, note that since we define $L_{in}(x, s)$ in terms of what s and its successors do, this is a *backwards analysis*; information gets transferred backwards through the program. The first rule states that if s does not mention x, then x is live before s if and only if it is live after s (in which case some statement following s makes it live). The second rule says that if s uses x, then clearly x is live immediately before s. If we have a statement `x := e` in which e does not mention x, then x is not live immediately before the statement, since the statement kills the old value of x. If e mentions x, then the second rule applies.

Let's apply these rules to variable a in the following basic block:

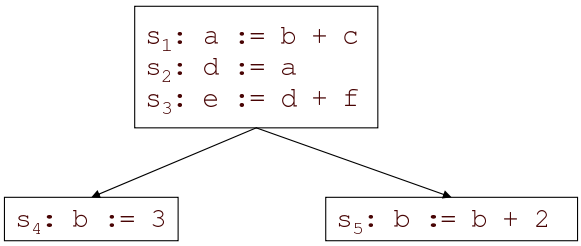
```
s1: a := b + c
s2: d := a
s3: e := d + f
```

Assume that initially, $L_{out}(a, s_3) = \textbf{false}$. Since s_3 does not mention a, we have $L_{in}(a, s_3) = \textbf{false}$ by the first rule. By the second rule, $L_{in}(a, s_2) = \textbf{true}$, since it mentions a on its right-hand side. Finally, by the third rule, $L_{in}(a, s_1) = \textbf{false}$, since it assigns to a but does not mention it on its right-hand side.

What happens when a statement has more than one successor, for example because of a conditional branch? In general, a compiler cannot determine which branch of a conditional will be taken at runtime, so it must assume that both branches can execute. Therefore, to be safe, we must conclude that $L_{out}(x, s) = \texttt{true}$ if x is live before *any* of the successors of s. This intuition corresponds to ORing together liveness information from successors, leading to the following additional rule:

$$L_{out}(x, s) \quad = \quad \bigvee \{L_{in}(x, s') \mid s' \in succ(s)\}$$

Our new rule can be applied to the following simple example, when trying to figure out the liveness of b:



Again, assume that L_{out} is initialized to **false** for all statements and variables. By our previous rules, we have $L_{in}(x, s_4) =$ **false** and $L_{in}(x, s_5) = \texttt{true}$ (you should be able to see why). Now, by our new rule, we find that $L_{out}(x, s_3) = \texttt{true}$, since s_5 uses x and we assume that either s_4 or s_5 can execute. Finding the rest of the liveness values for b should be straightforward.

With these four rules, we can compute liveness information for an entire control-flow graph, including conditionals and loops. The algorithm essentially applies the rules to update liveness values until all values stop changing. How do we know this process will terminate? Notice that if we initialize all liveness values to **false**, our rules will only change a **false** value to **true**, and not vice-versa. So, in the worst-case, eventually the analysis will make all liveness values **true** and then will terminate. More formally, we can order our facts by making **true** higher than **false**, and then our ORing of values at control-flow merge points corresponds to the least-upper-bound operation discussed in lecture.