1 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. 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
```

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 1, 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 $(\top, \ldots, \top)$.

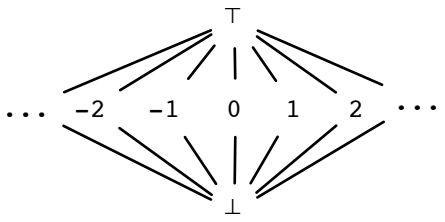

Figure 1: Conditional constant propagation lattice

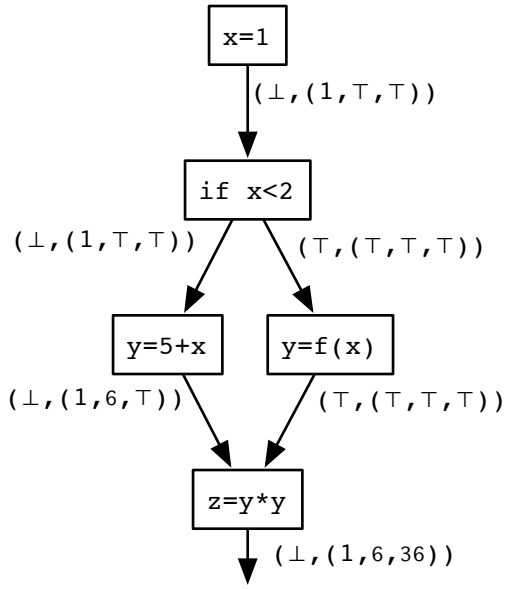

Figure 2: Example of conditional constant propagation

Since a variable can be assigned multiple times in the code, the analysis is more precise if we keep track of this tuple of elements at every point in the code.

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 2.

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.

1.1 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 3, in which the meet is taken both before a join point in control flow (corresponding to the way that dataflow

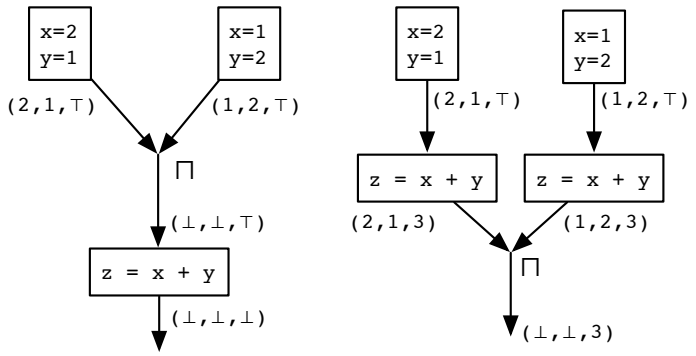

Figure 3: Conditional constant propagation does not give the meet-over-paths solution

analysis works) and after (corresponding to the meet-over-paths criterion). Nevertheless, it's an important and effective optimizaton.

Sparse conditional constant propagation, introduced by Wegman and Zadeck, takes advantage of SSA to keep track of less information about each node. In the SSA representation, which we will discuss soon, 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. 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.