1 Value Numbering

At a very basic level, most optimization techniques revolve around avoiding redundant or unnecessary computation. Thus, it's vital that we discover which values in a program are equal. That way, we can simplify the code that wastes machine cycles repeatedly calculating the same values. Classic optimization phases like constant/copy propagation, common subexpression elimination, loop-invariant code motion, induction variable elimination, and others discussed in the de facto treatise, "The Dragon Book", perform this sort of redundancy elimination based on information about the equality of expressions.

cite

In general, the problem of determining whether two expressions in a program are equivalent is undecidable. Therefore, we seek a *conservative* solution that doesn't necessarily identify all equivalences, but is nevertheless correct about any equivalences it does identify. Solving this equivalence problem is the work of *value numbering* algorithms. These assign every value in the program a number such that two values have the same value number if and only if the compiler can prove they will be equal at runtime.

Value numbering has a long history in literature and practice, spanning many techniques. In **??** we saw the `value-numbering` word, which is actually based on some of the earliest—and least effective—methods of value numbering. **??** describes the way Factor's current algorithm works, and highlights its shortcomings to motivate the main work of this thesis, which is covered in **??** and Section 1.1. We finish the section by analyzing the results of these changes and reviewing the literature for further enhancements that can be made to this optimization pass.

1.1 Redundancy Elimination

Now that we've identified congruences across the entire control flow graph (CFG), we must eliminate any redundancies found. Since value numbering is now offline, this entails another pass. However, replacing instructions is more subtle with global value numbers than it is with local ones. Because values come from all over the CFG, we must consider if a definition is *available* at the point where we want to use it.

Figures 1 and 2 on pages 2–3 show the difference. In the former, we can see the CFG before value numbering for the code `[ 10 ] [ 20 ] if 10 20 30`. The two extra integers being pushed at the end are there to avoid branch splitting (see **??** on page ??). In block 4, there's a `##load-integer 27 10`, which loads the value `10`. In globally numbering values, we associate the `##load-integer 22 10` in block 2 with the value `10` first, making it the canonical representative. However, we can't replace the instruction in block 4 with `##copy 27 22`, because control flow doesn't necessarily go through block 2, so the virtual register `22` might not even be defined. However, in Figure 2 on page 3, we see the CFG for the code `10 swap [ 10 ] [ 20 ] if 10 20 30`. In this case, the first definition of the value `10` comes from block 1, which dominates block 4. So, the definition is available, and we can replace the `##load-integer` in block 4 with a `##copy`.

There are several ways to decide if we can use a definition at a certain point. For instance, we could use dominator information, so that if a definition in a basic block B can be used by any basic block dominated by B. However, here we'll use a data flow analysis called *available expression analysis*, since it was readily implemented. Mercifully, Factor has a vocabulary that automatically defines data flow analyses with little more than a single line of code.

cite Simpson

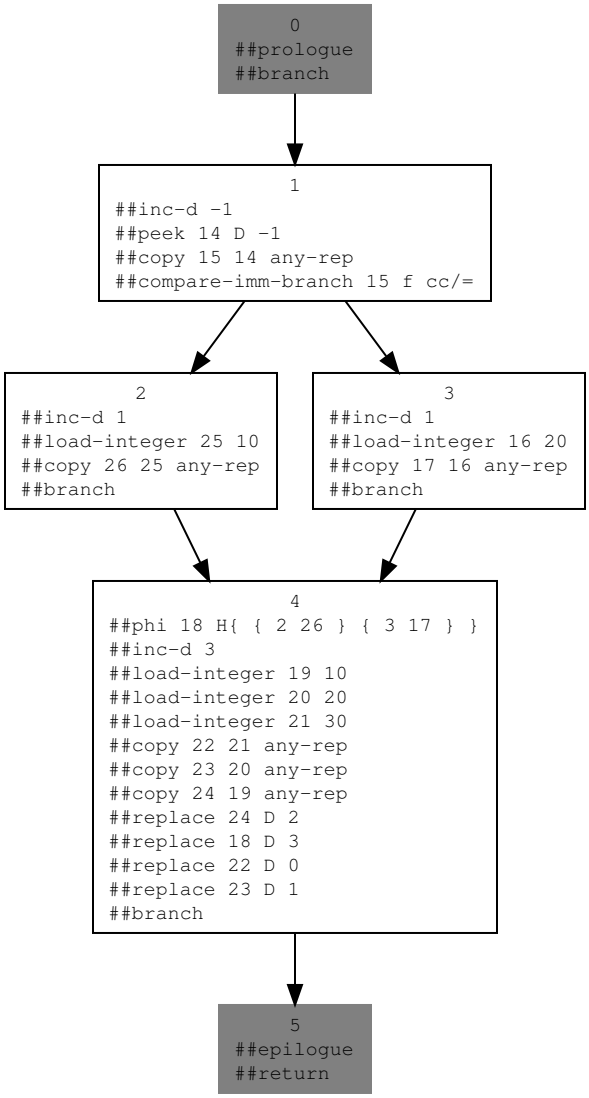

Figure 1: 10 is not available in block 4

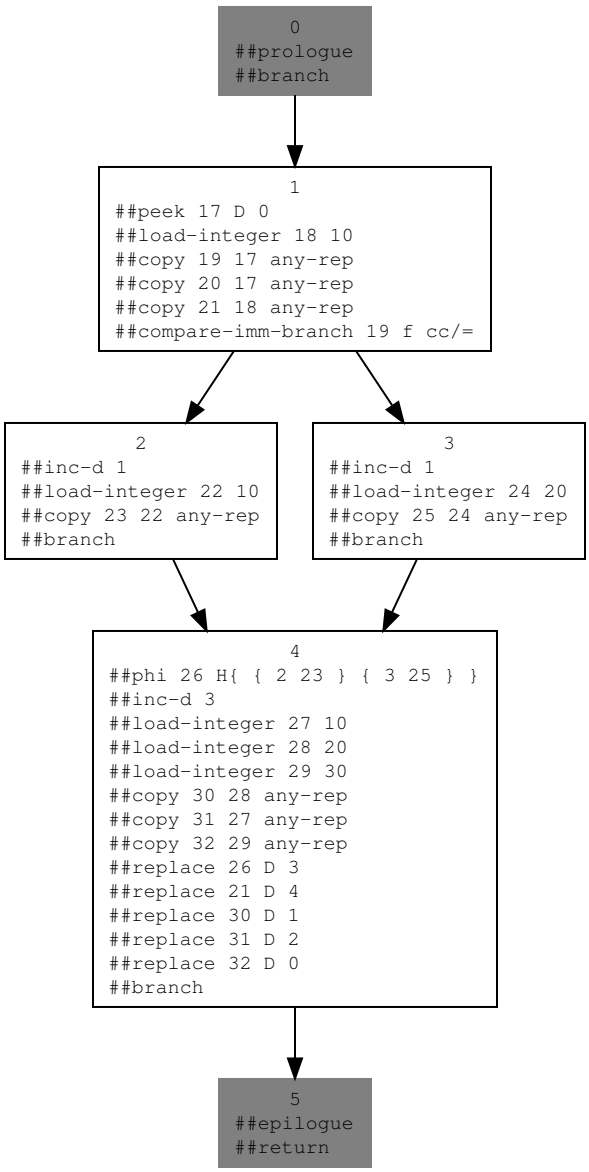

Figure 2: 10 is available in block 4

```
! Copyright (C) 2011 Alex Vondrak.
! See http://factorcode.org/license.txt for BSD license.
USING: accessors assocs hashtables kernel namespaces sequences
sets
compiler.cfg
compiler.cfg.dataflow-analysis
compiler.cfg.def-use
compiler.cfg.gvn.graph
compiler.cfg.predecessors
compiler.cfg.rpo ;
FROM: namespaces => set ;
IN: compiler.cfg.gvn.avail

: defined ( bb -- vregs )
    instructions>> [ defs-vregs ] map concat unique ;

FORWARD-ANALYSIS: avail

M: avail-analysis transfer-set drop defined assoc-union ;

: available? ( vn -- ? )
    final-iteration? get [
        basic-block get avail-in key?
    ] [ drop t ] if ;

: available-uses? ( insn -- ? )
    uses-vregs [ available? ] all? ;

: with-available-uses? ( quot -- ? )
    keep swap [ available-uses? ] [ drop f ] if ; inline

: make-available ( vreg -- )
    basic-block get avail-ins get [ dupd clone ?set-at ] change-at ;
```

Listing 1: The `compiler.cfg.gvn.avail` vocabulary—available expressions analysis

Listing 1 on the preceding page shows the vocabulary that defines the available expression analysis. It is a forward analysis based on the flow equations below:

cite?

$$\texttt{avail-in}_i = \begin{cases} \varnothing & \text{if } i = 0 \\ \bigcap_{j \in \text{pred}(i)} \texttt{avail-out}_j & \text{if } i > 0 \end{cases}$$

$$\texttt{avail-out}_i = \texttt{avail-in}_i \cup \texttt{defined}_i$$

Here, i indicates the basic block number.