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

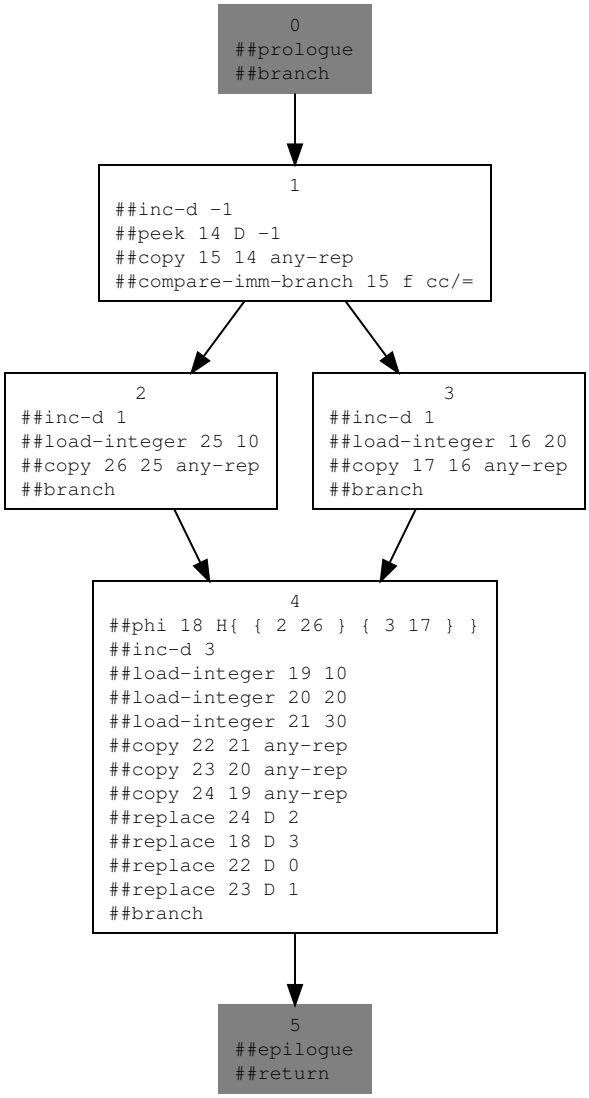

Figure 1: 10 is not available in block 4

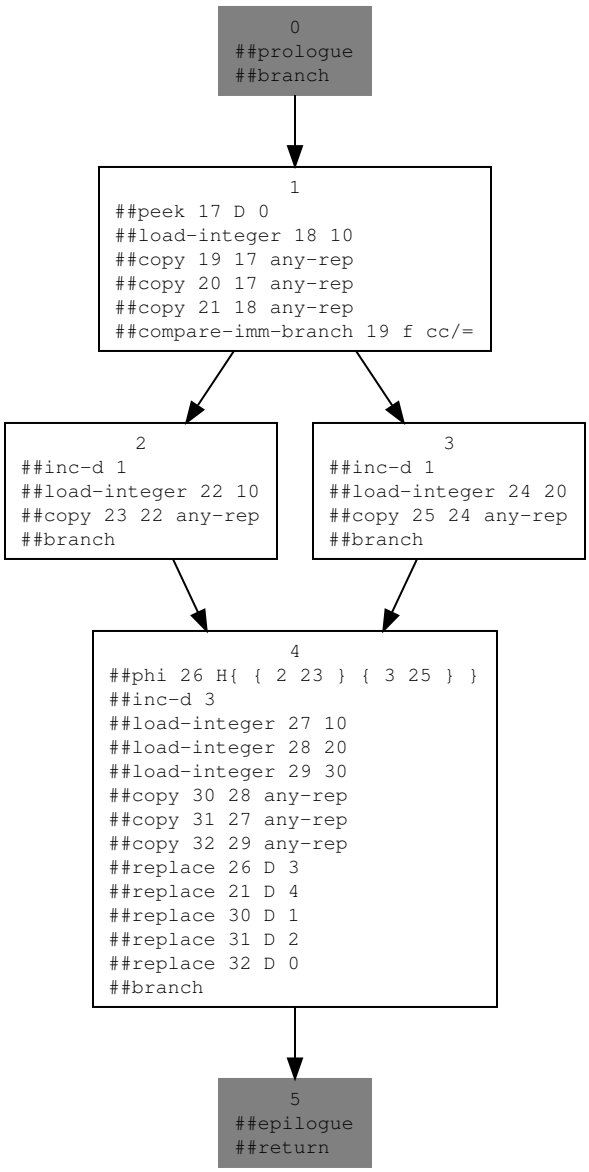

Figure 2: 10 is available in block 4

```
! 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

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, the subscripts indicate the basic block number. $\texttt{defined}_i$ denotes the result of the `defined` word from Listing 1 on the previous page. This returns the set of virtual registers defined in a basic block. Since we use virtual registers as value numbers, this is the same as giving us all the value numbers produced by a basic block. "Killed" definitions are impossible by the static single assignment (SSA) property, so we needn't track redefinitions of any virtual register. Using set intersection as the confluence operator means that the `avail-in` set will contain those values which are available on all paths from the start of the CFG to that block.

Using Factor's `compiler.cfg.dataflow-analysis` vocabulary, the implementation takes all of two lines of code. The `FORWARD-ANALYSIS: avail` line automatically defines several objects, variables, words, and methods that don't warrant full detail here. One we're immediately concerned with is the `transfer-set` generic, which dispatches upon the particular type of analysis being performed and is invoked on the proper in-set and basic block. There is no default implementation, as it is the chief difference between analyses. So, the next line uses `defined` and **`assoc-union`** to calculate the result of the data flow equation. Other pieces we'll see used are the top-level `compute-avail-sets` word that actually performs the analysis, the `avail-ins` hash table that maps basic blocks to their in-sets, and the `avail-in` word that is shorthand for looking up a basic block's in-set.

We want to use the results of this analysis in the `rewrite` methods so that they won't overstep their boundaries, and only make meaningful rewrites. However, we also want to use `rewrite` in the `determine-value-numbers` pass, where we don't care about availability. In fact, we want to ignore availability altogether, so that we can discover as many congruences as possible. In order to separate these concerns, we need to have two modes for checking availability. Listing 1 on the preceding page defines the `available?` word to do just this. It will only check the actual availability if `final-iteration?` is true, otherwise defaulting to **`t`**. Therefore, during the value numbering phase, everything is considered available. We further define the utilities `available-uses?` and `with-available-uses?`. The former checks if all an instruction's uses are available, and the latter does this only if another quotation first returns true. That way, we can guard instruction predicates with a test for available uses, like `[ ##add-imm? ] with-available-uses?`.

Finding all the instances where `rewrite` needed to be altered was subtle. Since the old `value-numbering` was an online optimization, it didn't need to worry about modifying an instruction in memory. But by doing the fixed-point iteration, we cannot permit `rewrite` to destructively modify any object instance until the final iteration. An obvious instance was in `compiler.cfg.value-numbering.comparisons` with the word `fold-branch`, responsible for modifying the CFG to remove an untaken branch. We definitely would not want the branch removed while doing the fixed-point iteration, because the transformation is not necessarily sound. So, we can protect it with a check for `final-iteration?`.

More typical were words like `self-inverse` from `compiler.cfg.value-numbering.math` (refer to Listing 3 on the next page). The idea is to change

```
! Before
: fold-branch ( ? -- insn )
    0 1 ?
    basic-block get [ nth 1vector ] change-successors drop
    \ ##branch new-insn ;

! After
: fold-branch ( ? -- insn )
    final-iteration? get [
        0 1 ?
        basic-block get [ nth 1vector ] change-successors drop
    ] [ drop ] if
    \ ##branch new-insn ;
```

Listing 2: Branch folding before and after

```
: self-inverse ( insn -- insn' )
    [ dst>> ] [ src>> vreg>insn src>> ] bi <copy> ;

! Before
M: ##neg rewrite
    {
        { [ dup src>> vreg>insn ##neg? ] [ self-inverse ] }
        { [ dup unary-constant-fold? ] [ unary-constant-fold ] }
        [ drop f ]
    } cond ;

! After
: self-inverse? ( insn quot -- ? )
    [ src>> vreg>insn ] dip with-available-uses? ; inline

M: ##neg rewrite
    {
        { [ dup [ ##neg? ] self-inverse? ] [ self-inverse ] }
        { [ dup unary-constant-fold? ] [ unary-constant-fold ] }
        [ drop f ]
    } cond ;
```

Listing 3: Rewriting words that are their own inverses

```
##neg 1 2
##neg 3 1
```

into

```
##neg 1 2
##copy 3 2 any-rep
```

since `##neg` undoes itself. But `rewrite` only has knowledge of one instruction at a time, so it looks up the redundant `##neg`'s source register in the `vregs>insns` table to see if it's computed by another `##neg` instruction. For straight-line code this is alright, but the source of the originating `##neg` (in the example, the virtual register `2`) isn't necessarily available. So, we have to use `with-available-uses?` to make sure the virtual registers used by the result of a `vreg>insn` can themselves be used before we rewrite anything.

An even subtler issue that led to infinite loops occured in simplifcations like the arithmetic distribution in `compiler.cfg.value-numbering.math`. The problem is that the `rewrite` method would generate instructions that assigned to entirely brand new registers. These, of course, would invariably get value numbered, triggering a change in the `vregs>vns` table. A new iteration would begin, and (since it gets called on the same instructions as the previous iteration) `rewrite` would generate new virtual registers all over again. Therefore, the `vregs>vns` table would never stop changing. As a stop-gap, distribution had to be disabled altogether until the final iteration.

make it so

Armed with the correct rewrite rules, availability information, and global value numbers, we can perform global common subexpression elimination (GCSE). The logic in the `gcse` generic in Listing 4 on the following page is similar to `process-instruction` from **??** on page ?? and `value-number` from **??** on page ??. Unlike `value-number`, we do return an instruction (or sequence thereof) representing the replacement. Thus, the **`array`** method uses **`map`** instead of **`each`**, to hold onto the resulting sequence when recursing into several instructions.

`defs-available` is similar to `record-defs`, except that value numbers have already stabilized, so we don't call `set-vn`. Instead, we use the `make-available` word, which was the last one defined in Listing 1 on page 4. As we process the instructions of a block in order, we want to make sure that local rewrites can still be performed. We have to ensure that after processing an instruction, any register it defines is available to future instructions in the same block. Thus, we add the virtual register to that block's `avail-in` (which acts like a set, even though it's implemented by a hash table by Factor's data flow analysis framework). `alien-call-insns`, `##callback-inputs` instructions, and instances of `##copy` don't get rewritten any further, so we simply note that their definitions are available and move on.

The `?eliminate` word transforms an instruction into a `##copy` of the canonical value number that computes it. If the value number isn't available, we don't do anything but post-process with `defs-available`. If it is, a `##copy` is produced and its destination is made available. Thus, `eliminate-redundancy` works like `check-redundancy` from **??** on page ??. We look up the expression computed by the instruction in the `exprs>vns` table. If it's there, we call `?eliminate`, but otherwise we leave the instruction alone and make its definitions available.

If the inputs to a `##phi` are all congruent, we'll call `?eliminate` to transform it into a `##copy` of its first input (without loss of generality). Otherwise, we check for equivalent `##phi`s with `eliminate-redundancy`.

```
GENERIC: gcse ( insn -- insn' )

M: array gcse [ gcse ] map ;

: defs-available ( insn -- insn )
    dup defs-vregs [ make-available ] each ;

M: alien-call-insn gcse defs-available ;
M: ##callback-inputs gcse defs-available ;
M: ##copy gcse defs-available ;

: ?eliminate ( insn vn -- insn' )
    dup available? [
        [ dst>> dup make-available ] dip <copy>
    ] [ drop defs-available ] if ;

: eliminate-redundancy ( insn -- insn' )
    dup >expr exprs>vns get at
    [ ?eliminate ] [ defs-available ] if* ;

M: ##phi gcse
    dup inputs>> values [ vreg>vn ] map sift
    dup all-equal? [
        [ first ?eliminate ] unless-empty
    ] [ drop eliminate-redundancy ] if ;

M: insn gcse
    dup defs-vregs length 1 = [ eliminate-redundancy ] when ;

: gcse-step ( insns -- insns' )
    [ simplify gcse ] map flatten ;

: eliminate-common-subexpressions ( cfg -- )
    final-iteration? on
    dup compute-avail-sets
    [ gcse-step ] simple-optimization ;
```

Listing 4: Global common subexpression elimination in `compiler.cfg.gvn`

Finally, the `insn` method will default to calling `eliminate-redundancy` on instructions that define only one value, much like how `value-number` worked.

```
: value-numbering ( cfg -- cfg )
    needs-predecessors
    dup determine-value-numbers
    dup eliminate-common-subexpressions

    cfg-changed predecessors-changed ;
```

Listing 5: New global `value-numbering` word in `compiler.cfg.gvn`

The main loop works similarly to `determine-value-numbers`. `final-iteration?` is turned on (set to `t`), and we make sure to compute the `avail-in` sets needed to make `available?` work. Then, using `simple-optimization`, we iterate over each basic block. For each instruction, we first use `simplify` (refer to **??** on page ??), then call `gcse` on the rewritten instruction. Thus, `rewrite` does the work of propagating available subexpressions and simplifying instructions, then `gcse` cleans up redundant instructions by converting them into `##copy` instructions if possible. The new `value-numbering` word can be seen in Listing 5.

oh no! relative time stamps

Consider for the last timethe example `0 100 [ 1 fixnum+fast ] times`. Before, we had the CFG in Figure 3 on the next page. Making a final pass with `eliminate-common-subexpressions` gives us the CFG in Figure 4 on page 11. Compared to the CFG after the old `value-numbering` word was called (see **??** on page ??), we have identified several more redundancies:

- The second `##phi` in block 2 has been turned into a `##copy` of the first.
- The `##compare-integer` of block 2 has been simplified to a `##compare-integer-imm`, since its operand `26` is both available and known to correspond to the integer value `100`.
- Instead of a `##compare-integer-branch`, we similarly have a `##compare-integer-imm-branch` at the end of block 2.
- Because the `##phi`s have been recognized as copies (i.e., the induction variables are congruent), the second `##add-imm` resulting from the old version of `value-numbering` is turned into a `##copy` of the first.

Afterwards, the `copy-propagation` pass cleans up all of these newly identified copies, as seen in Figure 5 on page 12. `eliminate-dead-code` now gets rid of more instructions than before, such as the second `##load-integer` in block 1, since it has been propagated to the `-imm` instructions in block 2. See Figure 6 on page 13. At last, after `finalize-cfg` in Figure 7 on page 14, we see a loop that uses a single register—down from the three in **??** on page ??.

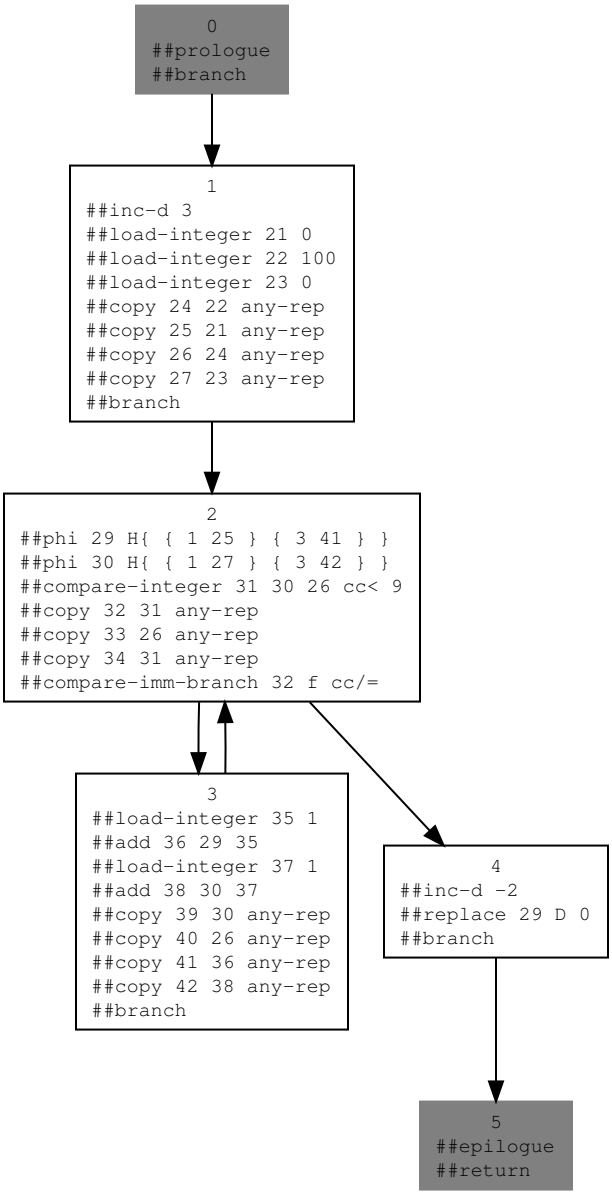

Figure 3: `0 100 [ 1 fixnum+fast ] times` before the new `value-numbering`

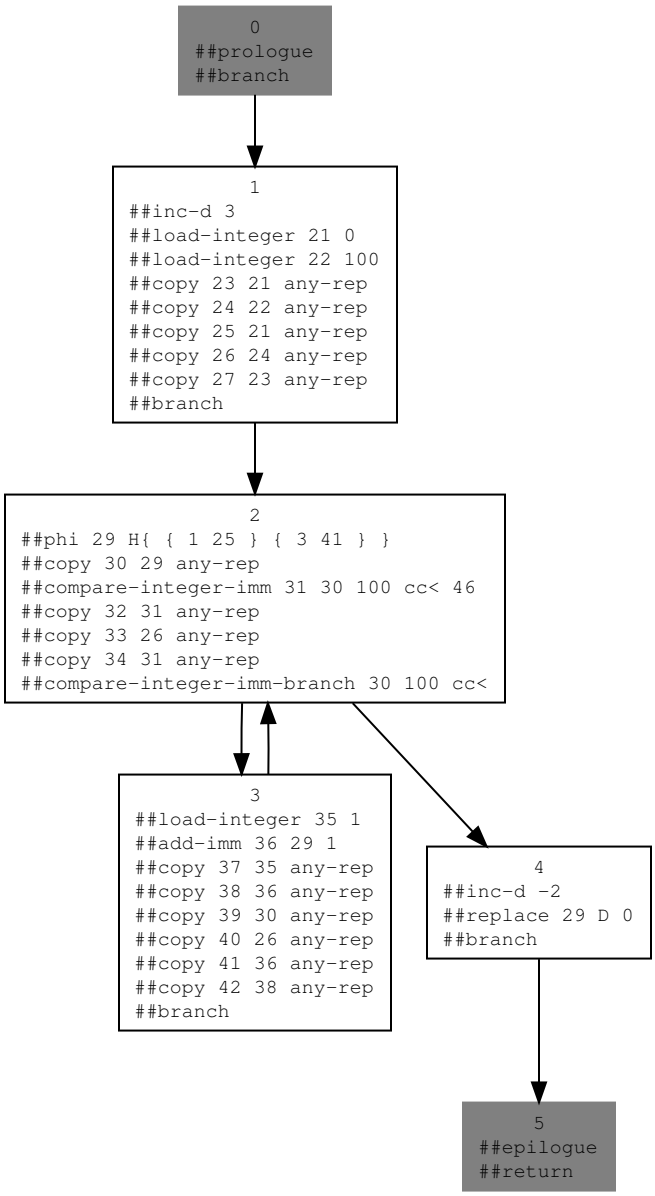

Figure 4: `0 100 [ 1 fixnum+fast ] times` after the new `value-numbering`

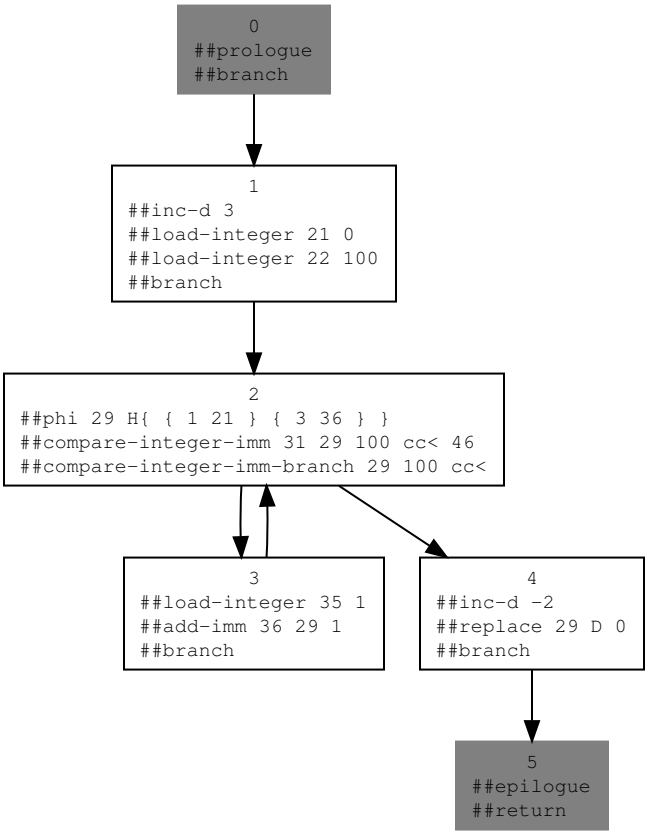

Figure 5: `0 100 [ 1 fixnum+fast ] times` after `copy-propagation`

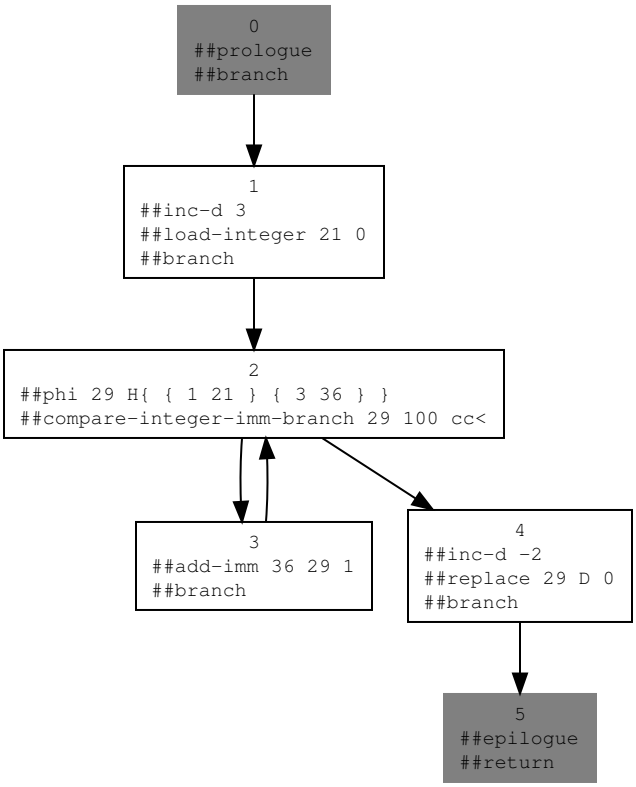

Figure 6: `0 100 [ 1 fixnum+fast ]` **`times`** after `eliminate-dead-code`

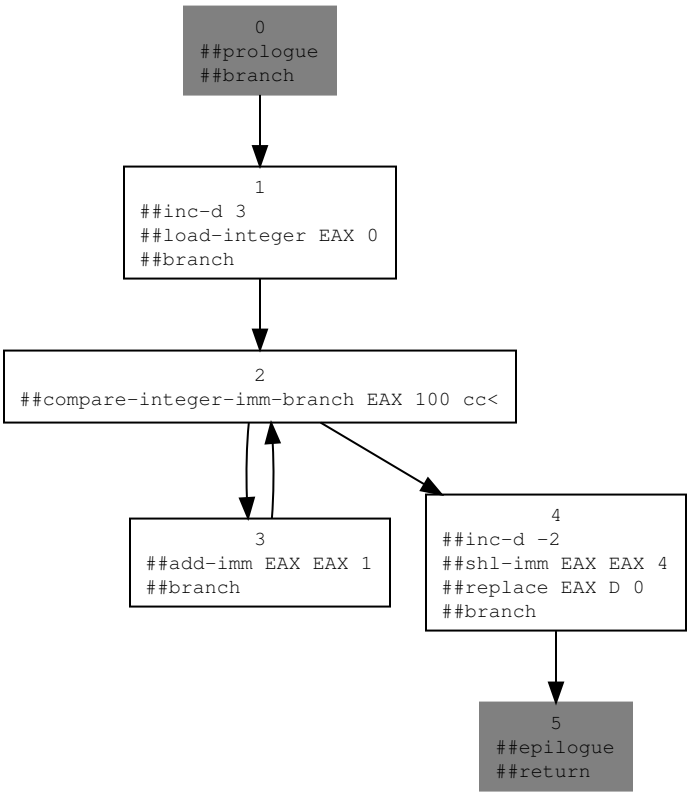

Figure 7: `0 100 [ 1 fixnum+fast ]` **`times`** after `finalize-cfg`