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. Section 1.1 describes the way Factor's current algorithm works, and highlights its shortcomings to motivate the main work of this thesis, which is covered in **??**. 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 Local Value Numbering

Tracing the exact origins of value numbering is difficult. It's thought to have originally been invented in the 1960s by Balke. The earliest tangible reference to a value numbering (at least, the earliest point where discussions in the literature seem to start) appears in an oft-cited but unpublished work of Cocke. The technique is relatively simple, but not as powerful as other methods for reasons described hereafter.

cite Simpson

cite-like

The algorithm considers a single basic block. For each instruction (from top to bottom) in the block, we essentially let the value number of the assignment target be a hash of the operator and the value numbers of the operands. That is, we hash the *expression* being computed by an instruction. Thus, assuming a proper hash function, two expressions are *congruent* (denoted $\texttt{x} \cong \texttt{y}$) if

- they have the same operators and
- their operands are congruent.

This is our approximation of runtime equivalence. The first property is fulfilled by basing the hash, in part, on the operator. The second property holds because the hash is based on the value numbers of the statement's operands—not just the operands as they appear in code (i.e., *lexical* equivalence). Any information about congruence is propagated through the value numbers. We'll have discovered any such equivalences among the operands before computing the value number of the assignment target because every value in a basic block is either defined before it's used, or

else defined at some point in a predecessor of the block, which we don't care about when only considering one basic block.

This is the first shortcoming of the algorithm. It is *local*, focusing on only one basic block at a time. Any definitions outside the boundaries of the basic block won't be reused, even if they reach the block. This severely limits the scope of the redundancies we can discover. We could improve upon this by considering the algorithm across an entire loop-free control flow graph (CFG) in any *topological order*. In such an ordering, a basic block B comes before any other block B' to which it has an edge. Thus, any "outside" variables that instructions in B' rely on must have come from B or earlier, which will have already been computed in a traversal of such an ordering. However, CFGs usually contain cycles or loops (at least interesting ones do), which make such an ordering impossible. We could still pick a topological order that ignores back-edges, but we may encounter operands whose values flow along those back-edges, so haven't been processed yet. We'll address this issue later.

In Factor, expressions are basically instructions (the `insn` objects discussed in **??**) that have had their destination registers stripped. Instructions can be converted to expressions with the `>expr` word defined in the `compiler.cfg.value-numbering.expressions` vocabulary. For instance, an `##add` instruction with the destination register `1` and source registers `2` and `3` will be converted into an array of three elements:

- The `##add` class word, indicating the expression is derived from an `##add` instruction.
- The value number of the virtual register `2`.
- The value number of the virtual register `3`.

Some instructions are not *referentially transparent*, meaning they can't be replaced with the value they compute without changing the program's behavior. For example, `##call` and `##branch` cannot reasonably be converted into expressions. In these cases, `>expr` merely returns a unique value.

The hashing of expressions takes place in the so-called *expression graph* implemented in the vocabulary shown in Listing 1 on the following page. This consists of three global hash tables that relate virtual registers, value numbers, instructions, and expressions. Since virtual registers are just integers, we actually use them as value numbers, too. `vregs>vns` maps virtual registers to their value numbers. If a virtual register is mapped to itself in this table, its definition is the canonical instruction that we use to compute the value. This instruction is stored in the `vns>insns` table. Finally, the most important mapping is `exprs>vns`. True to its name, it uses expressions as keys, which of course are implicitly hashed. Thus, we can use this table to determine equivalence of expressions.

Other definitions in Listing 1 on the next page manipulate expressions and the graph. The global variable `input-expr-counter` is used in the generation of unique expressions discussed earlier. `init-value-graph` initializes this and all the tables. `set-vn` establishes a mapping from a virtual register to a value number in `vregs>vns`. `vn>insn` gives terse access to the `vns>insns` table. `vreg>insn` uses `vregs>vns` and `vns>insns` to get the canonical instruction that defines a given virtual register. Finally, `vreg>vn` looks up the value of a key in the `vregs>vns` table. Importantly, if the key is not yet present in the table, it is automatically mapped to itself—it's assumed that the virtual register does not correspond to a redundant instruction.

```
! Copyright (C) 2008, 2010 Slava Pestov.
! See http://factorcode.org/license.txt for BSD license.
USING: accessors kernel math namespaces assocs ;
IN: compiler.cfg.value-numbering.graph

SYMBOL: input-expr-counter

! assoc mapping vregs to value numbers
! this is the identity on canonical representatives
SYMBOL: vregs>vns

! assoc mapping expressions to value numbers
SYMBOL: exprs>vns

! assoc mapping value numbers to instructions
SYMBOL: vns>insns

: vn>insn ( vn -- insn ) vns>insns get at ;

: vreg>vn ( vreg -- vn ) vregs>vns get [ ] cache ;

: set-vn ( vn vreg -- ) vregs>vns get set-at ;

: vreg>insn ( vreg -- insn ) vreg>vn vn>insn ;

: init-value-graph ( -- )
    0 input-expr-counter set
    H{ } clone vregs>vns set
    H{ } clone exprs>vns set
    H{ } clone vns>insns set ;
```

Listing 1: The `compiler.cfg.value-numbering.graph` vocabulary

This is the second shortcoming of the algorithm. It must make a *pessimistic* assumption about congruences. That is, it starts by assuming that every expression has a unique value number, then tries to prove that there are some values which are actually congruent. This fails to discover congruences for values that flow along back-edges, whether we consider a single basic block or an entire topological ordering.

One the other hand, an advantage of this local value numbering algorithm is its simplicity. It makes a single pass over all the instructions, identifying and replacing redundancies *online* (i.e., rewriting as it goes). It's straightforward to write, and even to extend. In particular, there's nothing stopping the online replacements from being more complex than `##copy` instructions. At every step, the currently known value numbers will be sound, and we can use this information for copy/constant propagation, constant folding, and common subexpression elimination.

```
: value-numbering-step ( insns -- insns' )
    init-value-graph
    [ process-instruction ] map flatten ;

: value-numbering ( cfg -- cfg )
    dup [ value-numbering-step ] simple-optimization

    cfg-changed predecessors-changed ;
```

Listing 2: Main words from `compiler.cfg.value-numbering`

To see how Factor accomplishes these extensions, we'll take a look at the `compiler.cfg.value-numbering` vocabulary. Listing 2 shows the main words that start the optimization pass. The `value-numbering-step` word is called on the sequence of instructions that comprise each basic block. It starts the expression graph from a blank slate with `init-value-graph`, then **`map`**s the word `process-instruction` on each of them. This is a generic word that we'll study momentarily; it returns either a single `insn` object or a sequence of them (in the case that an instruction is replaced by several others). Then, the work of `value-numbering` is to just call `value-numbering-step` on each basic block, which is done with a combinator called `simple-optimization`. The words `cfg-changed` and `predecessors-changed` alter some internal state of the CFG that has been potentially invalidated by some transformations performed by `process-instruction`.

The methods of `process-instruction` are shown in Listing 3 on the next page. The default behavior for dispatching on an `insn`is to invoke yet another generic word, `rewrite`. This word will return either a replacement `insn` (or sequence thereof) or **`f`**, indicating that no change has taken place. Thus, by recursively calling `process-instruction`, we can do more specialized processing on this rewritten replacement (e.g., dispatching on `insn` again, which applies `rewrite` once more). If the instruction can't be simplified further, we simply return it. (Note that `[ X ] [ Y ] ?if` is the same as **`dup [ nip X ] [ drop Y ] if`**.)

line numbers?

For instances of `foldable-insn` (i.e., `insn`s that can be converted to useful expressions with `>expr`), we similarly invoke `rewrite` recursively until no more rewriting occurs. When that happens, rather than just return the instruction, we invoke `check-redundancy`—though only if the instruction defines exactly 1 virtual register, which will be stored in a slot named `dst`. `check-redundancy`

```
GENERIC: process-instruction ( insn -- insn' )

: redundant-instruction ( insn vn -- insn' )
    [ dst>> ] dip [ swap set-vn ] [ <copy> ] 2bi ;

:: useful-instruction ( insn expr -- insn' )
    insn dst>> :> vn
    vn vn vregs>vns get set-at
    vn expr exprs>vns get set-at
    insn vn vns>insns get set-at
    insn ;

: check-redundancy ( insn -- insn' )
    dup >expr dup exprs>vns get at
    [ redundant-instruction ] [ useful-instruction ] ?if ;

M: insn process-instruction
    dup rewrite [ process-instruction ] [ ] ?if ;

M: foldable-insn process-instruction
    dup rewrite
    [ process-instruction ]
    [ dup defs-vregs length 1 = [ check-redundancy ] when ] ?if ;

M: ##copy process-instruction
    dup [ src>> vreg>vn ] [ dst>> ] bi set-vn ;

M: array process-instruction
    [ process-instruction ] map ;
```

Listing 3: The workhorse of `compiler.cfg.value-numbering`

checks if the expression being computed by the instruction is already a key of the `exprs>vns` table. If it is, the instruction is redundant, and we call `redundant-instruction`; otherwise, we call `useful-instruction`. The former uses `set-vn` to map the instruction's `dst` virtual register to the same value number as the expression that was a key of `exprs>vns`. Since value numbers are actually virtual registers, we may also use these two integers the source and destination registers in a new `##copy` instruction, which is then returned. On the other hand, `useful-instruction` saves the instruction's information in the expression graph by setting the appropriate values in `vregs>vns`, `exprs>vns`, and `vns>insns`. Note that in definitions using the syntax `:: word-name ( stack -- effect ) ... ;`, the input values in the stack effect are actually named lexical variables, like in most programming languages. Furthermore, the first line `insn dst>> :> vn` assigns the input instruction's destination register to the variable `vn`, which is used later in the definition of `useful-instruction`.

move this to primer?

The `##copy` method of `process-instruction` cannot do anything to simplify the instruction, but will set the value number of the destination register to that of the source. By calling `vreg>vn` on the source register, we make sure to call `set-vn` between the destination and the canonical value number of the source.

Finally, the **`array`** method is used for the purposes of recursion, in the case that `rewrite` returns a sequence of replacement instructions. The correct action is, of course, to descend into this new sequence of instructions with `process-instruction`.

Underlying all of the redundancy elimination is the `rewrite` generic word. It has too many methods to look at the source code in-depth here, but it's instructive to give an overview of the transformations. These methods actually make up the bulk of the `compiler.cfg.value-numbering` code. They're spread across various sub-vocabularies. `compiler.cfg.value-numbering.rewrite` defines the generic itself, along with a handful of utilities. The method for the most general instruction class, `insn`, is defined to unconditionally return **`f`**, meaning no rewriting is performed by default. That way, we need only define `rewrite` methods for more specific instruction classes to get specialized behavior.

`compiler.cfg.value-numbering.alien` contains methods that simplify nodes related to Factor's foreign function interface (FFI). Most involve fusing together the results of intermediate arithmetic. The instructions that access raw memory (namely `##load-memory`, `##load-memory-imm`, `##store-memory`, and `##store-memory-imm`) tend to have inputs to perform address arithmetic. Each has slots for a `base` register containing an address and a literal `offset` from it. But if `base` is defined by an `##add-imm` instruction, we can just update the `offset`, incrementing it by the literal operand of the `##add-imm`. Then, `base` will just be changed to the register operand of the `##add-imm`. This removes the memory instruction's need for the `##add-imm`, increasing the chances that the latter will become dead code to be removed later. Unlike the `-imm` variants, `##load-memory` and `##store-memory` also take a `displacement` register, which works like a non-immediate `offset`. Therefore, `##add`s can be similarly fused into `##load-memory-imm` and `##store-memory-imm` by transforming them into `##load-memory` and `##store-memory` instructions with the `##add`'s operand as the `displacement`. A few other similar transformations are also done, including rewrites for `##box-displaced-alien`s and `##unbox-any-c-ptr`s.

`compiler.cfg.value-numbering.comparisons` defines methods for the various branching and comparison instructions (which simply store booleans in registers, rather than branching upon them). The major optimizations performed are as follows:

- If possible, instructions are converted to more specific forms. For example, non-immediate instructions (e.g., `##compare`) may be turned into their `-imm` counterparts (e.g., `##compare-imm`) if one of their source registers corresponds to a literal value. `##compare-integer-imm` is also converted to `##test` if the architecture supports it. This corresponds to a special instruction in x86 that performs a bitwise AND for its side effects on particular flags, discarding the actual result. This can be more efficient when using the AND result as a boolean.
- If both inputs to a comparison or branch are literals, we may constant-fold the instruction. In the case of comparisons, this means converting it into a `##load-reference` of the proper boolean. In branches, this modifies the CFG so that the path which isn't taken is removed completely.
- Like a novice programmer writing `if (some_boolean != false) { ... }` in Java, the compiler may generate redundant boolean comparisons that need cleaning up. That is, the intermediate boolean values are eliminated when the result of a comparison is used by another comparison, collapsing the whole thing into a single instruction.

`compiler.cfg.value-numbering.folding` defines some auxiliary words for constant-folding arithmetic words. Mainly, `unary-constant-fold` and `binary-constant-fold` perform the actual operation on the one or two constant inputs provided. These words are used in `compiler.cfg.value-numbering.ma` which predictably simplifies math via standard rules. Arithmetic identities are rewritten—conceptually, $x + 0$ becomes just x, for instance. If self-inverting instructions (namely `##neg` for numerical negation and `##not` for boolean negation) are called on registers that themselves correspond to the same instruction, we can safely rewrite them into `##copy` instructions. Non-immediate instructions are converted to their `-imm` forms, if possible, and if both operands are constant, the expression is folded. The most interesting math optimizations use the associative and distributive laws. *Reassociation* conceptually converts $(x \otimes y) \otimes z$ into $x \otimes (y \otimes z)$ when both y and z are constants and $\otimes$ is associative. So, for example,

```
##add-imm 1 X Y
##add-imm 2 1 Z
```

is converted into just

```
##add-imm 2 X (Y+Z)
```

where `X` is a virtual register, and `Y` and `Z` are constants. *Distribution* converts $(x \oplus y) \otimes z$ into $(x \otimes z) \oplus (y \otimes z)$, where y and z are constants, $\oplus$ corresponds to addition or subtraction, and $\otimes$ to multiplication or left bitwise shifts. Therefore,

```
##add-imm 1 X Y
##mul-imm 2 1 Z
```

is converted into

```
##mul-imm 3 X Y
##add-imm 2 3 (Y*Z)
```

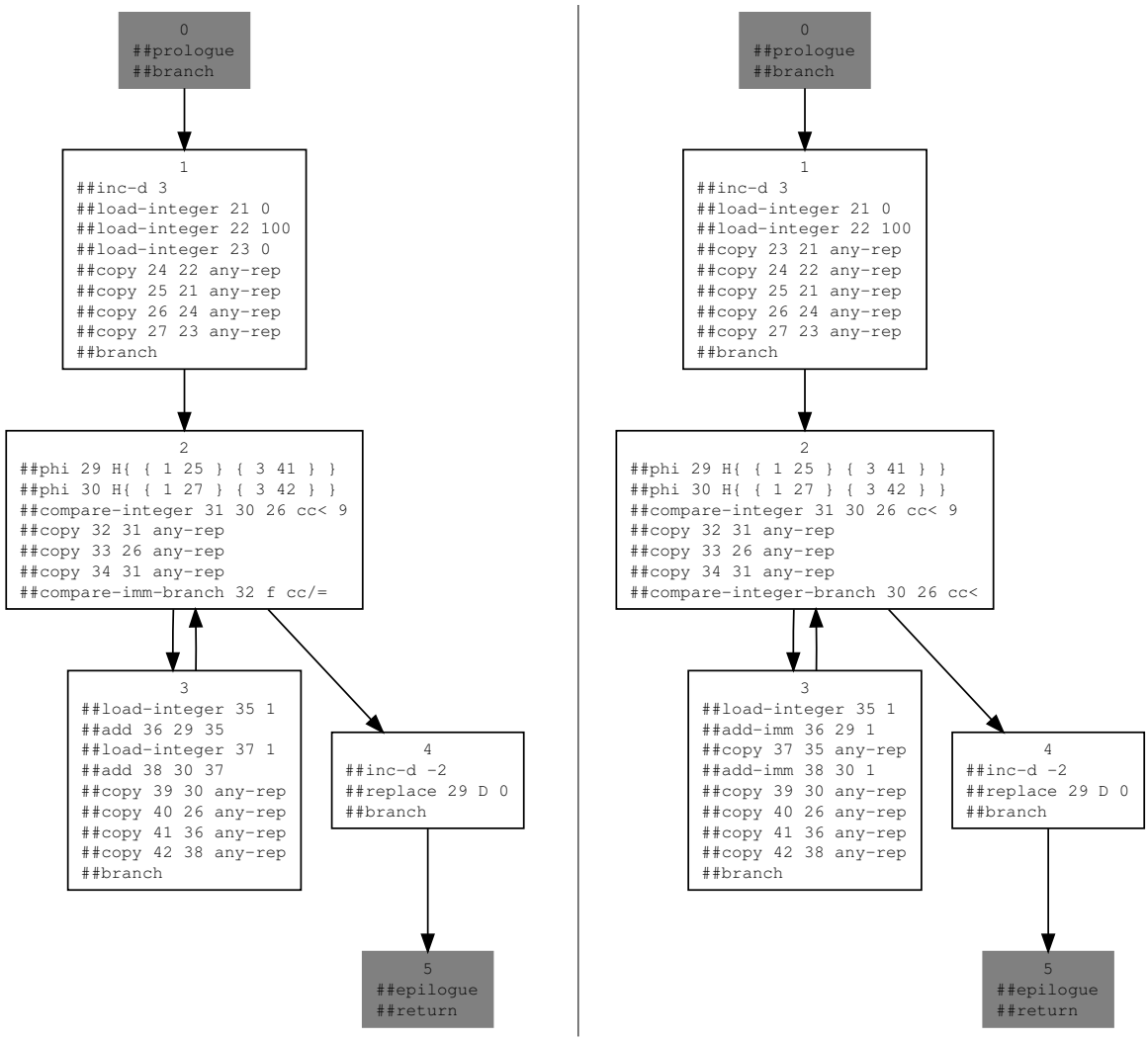

Figure 1: `0 100 [ 1 fixnum+fast ] times` before and after `value-numbering`

Notice that a new intermediate virtual register, `3`, had to be created. However, if the product of `Y` and `Z` can be computed at compile-time and fits in an immediate operand, then we save cycles by using `##mul-imm` on a smaller number.

The last few methods of `rewrite` provide some obvious simplifications. `compiler.cfg.value-numbering.simd` performs some limited constant-folding for vector operations. `compiler.cfg.value-numbering.slots` propagates `##add-imm` address calculation to `##slot`, `##set-slot`, and `##write-barrier` instructions in a manner similar to `compiler.cfg.value-numbering.alien`. Finally, `compiler.cfg.value-numbering.m` provides a single method to rewrite `##replace` into `##replace-imm` if possible.

make sure margins don't get overflown by long vocab names

make sure side-by-side figures aren't too crammed in final draft

To finish the discussion of local value numbering and Factor's particular implementation, we'll

examine the example from **??** on page ?? in depth. For convenience, the before/after snapshot of the CFG is reproduced in Figure 1 on the previous page.

`value-numbering-step` begins at block 1, where `process-instruction` is **map**ped across the instructions. `##inc-d 3` does not have a `rewrite` method, so remains untouched; it is also not a `foldable-insn`, so it is simply returned. While `##load-integer 21 0` doesn't have a `rewrite` method, it is a `foldable-insn`, so `process-instruction` calls `check-redundancy`. At this point, the expression graph is empty. Calling `>expr` converts this instruction into an `integer-expr` object representing `0`. `useful-instruction` leaves the tables as follows:

```
! vregs>vns
H{ { 21 21 } }

! exprs>vns

H{ { T{ integer-expr { value 0 } } 21 } }

! vns>insns
H{
    { 21 T{ ##load-integer { dst 21 } { val 0 } { insn# 1 } } }
}
```

The next instruction in block 1, `##load-integer 22 100`, behaves similarly, leaving:

```
! vregs>vns
H{ { 21 21 } { 22 22 } }

! exprs>vns
H{
    { T{ integer-expr { value 0 } } 21 }
    { T{ integer-expr { value 100 } } 22 }
}

! vns>insns
H{
    { 21 T{ ##load-integer { dst 21 } { val 0 } { insn# 1 } } }
    {
        22
        T{ ##load-integer { dst 22 } { val 100 } { insn# 2 } }
    }
}
```

The following instruction is `##load-integer 23 0`. In calling `check-redundancy`, we discover that the integer expression for `0` is already in `exprs>vns`, so this is turned into a `##copy`, and the value

number is noted. The remaining instructions in block 1 (aside from `##branch`) are all instances of `##copy`. `process-instruction` thus only sets their value numbers in the `vregs>vns` table, leaving them with the following at the end of block 1:

```
! vregs>vns
H{
    { 21 21 }
    { 22 22 }
    { 23 21 }
    { 24 22 }
    { 25 21 }
    { 26 22 }
    { 27 21 }
}

! exprs>vns
H{
    { T{ integer-expr { value 0 } } 21 }
    { T{ integer-expr { value 100 } } 22 }
}

! vns>insns
H{
    { 21 T{ ##load-integer { dst 21 } { val 0 } { insn# 1 } } }
    {
        22
        T{ ##load-integer { dst 22 } { val 100 } { insn# 2 } }
    }
}
```

yay, another term

Next, block 2 in Figure 1 on page 8 is processed. The tables are all reset, so even though block 1 happens to dominate block 2, none of its definitions are known to `value-numbering`. The `##phi`s are ignored, as no important methods dispatch upon them. In trying to rewrite the `##compare-integer`, we call `vreg>vn` on the operands. Since they aren't in the `vregs>vns` table yet, they are assumed to be unique values. This assumption is pessimistic—we'd rather the values be the same, so we can remove redundancy. It happens to be correct here, though, as `26` corresponds to the integer `100`, while `30` is an induction variable of the loop. However, `##compare-integer` cannot be rewritten into an immediate form, since our focus is local to the basic block, so we don't know that `26` has the value `100`. The `##copy` instructions are processed as usual, and `##compare-imm-branch 32 f cc/=` is rewritten into a `##compare-integer-branch`, as the virtual register `32` has the same value (through the copies) as the `##compare-integer` result. This is a case of simplifying the `if (some_boolean != false) { ... }` pattern, and the definition of the register `31` becomes dead code after `rewrite` finishes with this last instruction. The expression graph is populated thus by the end:

```
! vregs>vns
H{
    { 32 31 }
    { 33 26 }
    { 34 31 }
    { 26 26 }
    { 30 30 }
    { 31 31 }
}

! exprs>vns
H{ { { ##compare-integer 30 26 cc< } 31 } }

! vns>insns
H{
    {
        31
        T{ ##compare-integer
            { dst 31 }
            { src1 30 }
            { src2 22 }
            { cc cc< }
            { temp 9 }
            { insn# 2 }
        }
    }
}
```

Once again, the tables are reset and we proceed to block 3. The first instruction, `##load-integer 35 1`, is entered into the expression graph. Since `35` is an operand of `##add 36 29 35`, `rewrite` changes this instruction into an `##add-imm`, as we know the constant value of the operand. The next `##load-integer` gets turned into a `##copy`, like in block 1, and the next `##add` is similarly changed to `##add-imm`. The copies do little but set more value numbers. As `process-instruction` calls `vreg>vn` on their sources, we'll insert entries into `vregs>vns` for those defined outside of the block, like `26`. This leaves us with the following tables:

```
! vregs>vns
H{
    { 35 35 }
    { 36 36 }
    { 37 35 }
    { 38 38 }
    { 39 30 }
    { 40 26 }
```

```
    { 41 36 }
    { 26 26 }
    { 42 38 }
    { 29 29 }
    { 30 30 }
}

! exprs>vns
H{
    { { ##add-imm 30 1 } 38 }
    { { ##add-imm 29 1 } 36 }
    { T{ integer-expr { value 1 } } 35 }
}

! vns>insns
H{
    { 36 T{ ##add-imm { dst 36 } { src1 29 } { src2 1 } } }
    { 38 T{ ##add-imm { dst 38 } { src1 30 } { src2 1 } } }
    { 35 T{ ##load-integer { dst 35 } { val 1 } { insn# 0 } } }
}
```

The fourth invocation of `value-numbering-step` does not do anything interesting, as the `##replace` cannot be changed into a `##replace-imm`.

In summary, we managed to replace redundancies within basic blocks online by maintaining some simple hash tables. After copy propagation and dead code elimination, the CFG gets finalized to the one shown in Figure 2 on the next page. Because the value numbering algorithm was local, the `##compare-integer-branch` in block 2 could not be simplified to a `##compare-integer-imm-branch`, and we instead have to waste a register on the integer `100`. But it's important to note that even considering a topological ordering of the CFG wouldn't have worked, as we'd have to ignore back-edges. The `##phi`s that used to be in block 2 had inputs that flowed along the back-edge, and our pessimistic assumption would have to classify these values as distinct. One is for the counter introduced by **`times`**, and the other is from the top value of the stack being incremented by `fixnum+fast`. In this case, however, these induction variables are actually equal: both start at `0` and are incremented by `1` on each loop. In terms of the CFG in Figure 2 on the following page, the `EAX` and `EDX` registers are equivalent. Yet the combination of the pessimism and locality of the algorithm keep us from discovering this.

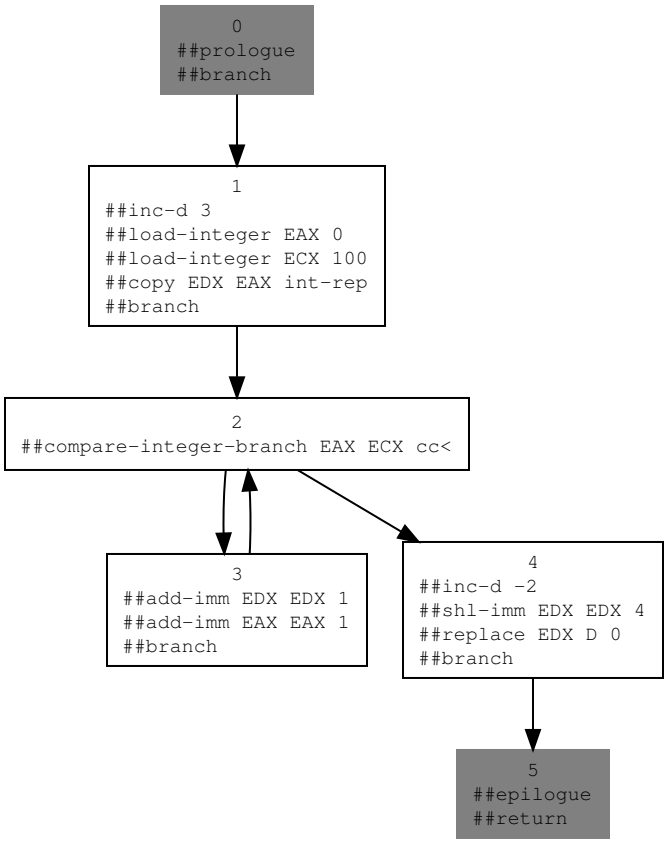

Figure 2: The final representation for `0 100 [ 1 fixnum+fast ] times`