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 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 Global Value Numbering

Answering the challenges of Cocke, AWZ described what would be the de facto value numbering algorithm for several years, and rightly so. It was a properly *global* value numbering algorithm, working across an entire control flow graph (CFG) instead of on single basic blocks. Their paper was important in another very relevant way: it is the first published reference to SSA form, including an algorithm for its construction.

cite-like

cite-like

cite Van-Drunen

Though we could try to extend the scope of Factor's local value numbering, it is still inherently pessimistic. The algorithm of AWZ, which is commonly referred to simply as AWZ, uses a modification of Hopcroft's minimization algorithm for finite state automata. It works on an *optimistic* assumption by first assuming every value has the same value number, then trying to prove that values are actually different. It does this by treating value numbers as *congruence classes* that partition the set of virtual registers. If two values are in the same class, then they are congruent, where congruence is defined as in **??**.

cite-like

gls?

cite-like

Such a partition is not unique, in general. For instance, a trivial one places each value in its own congruence class. So, we look for the *maximal fixed point*—the solution that has the most congruent values and therefore the fewest congruence classes. We must start with a congruence class for each operation so that, say, all values computed by `##add`s are grouped together, those computed by `##mul`s are in the same class, etc. We must then iteratively look at our collection of classes, separating them when we discover incongruent values. For an static single assignment (SSA) variable in class P, we look at its defining expression. If an operand at position i belongs to

class Q, then the i^{th} operand of every other value in P should also be in Q. Otherwise, P must be *split* by removing those variables whose i^{th} operands are not in class Q and placing them in a new congruence class. We keep splitting classes until the partitioning stabilizes.

The optimistic assumption may seem dangerous. Is it possible that we're "overoptimistic"? That two values assumed to be congruent and not proven incongruent might actually be inequivalent when the program is run? The AWZ paper dedicates a section to proving that two congruent variables are equivalent at a point p in the program if their definitions dominate p. The proof is a bit quirky, but reasonable. They develop a dynamic notion of dominance in a running program which implies static dominance in the code, then show that congruence implies runtime equality (though equivalence does not imply congruence).

gls?

AWZ made the need for global value numbering (GVN) algorithms apparent. However, finite state automata minimization makes for a more complicated algorithm than hash-based value numbering. A naïve implementation can be quadratic, although careful data structure and procedure design can make it $O(n \log n)$. Furthermore, it's resistant to the same improvements we easily added to the local value numbering. To even consider the commutativity of operations requires changes in operand position tracking and splitting—the heart of the algorithm. It is generally limited by what the programmer writes down: deeper congruences due to, say, algebraic identities can't be discovered.

gls?

In fact, by performing an optimization that uses the GVN information, more GVN congruences may arise. If we can somehow perform the two analyses simultaneously, they'll produce better results. This generalizes to interdependent compiler optimizations at large, as elucidated in Click's dissertation, which describes a method for formalizing and combining separate optimizations that make optimistic assumptions (whatever they happen to be for each particular analysis). He uses this to merge GVN with *conditional constant propagation*, which itself is a combination of constant propagation and unreachable code elimination (pretty much like the `propagate` pass from **??**). Furthermore, GVN is extended to handle algebraic identities, propagate constants, and fold redundant ϕs. Unfortunately, the straightforward algorithm for this is $O(n^2)$, while the $O(n \log n)$ version presented is not just complicated, but can also miss some congruences between ϕ-functions.

cite-like

cite Click, Simpson

Hot on the heels of this work, Simpson's dissertation provides probably the most exhaustive treatment of GVN algorithms. He presents several extensions, such as incorporating hash-based local value numbering into SSA construction, handling commutativity in AWZ GVN, and performing redundant store elimination. He builds off of the two classical algorithms independently, which underlines their inherent differences and limitations. In general, hash-based value numbering is easy to extend without greatly impacting the runtime complexity, as is the case in Factor's implementation.

cite-like

gls?

Drawing from this experience, Simpson's hallmark algorithm combines the best of both worlds by taking the hash-based algorithm which is easy to understand, implement, and extend, and making it global, so it identifies more congruences. Dubbed the "reverse postorder (RPO) algorithm", it simply applies hash-based value numbering iteratively over the CFG until we reach the same fixed point computed by AWZ. (The fact that it computes the *same* fixed point is proven fairly straightforwardly in the dissertation.) It could technically traverse the CFG in any topological order, but Simpson defaults to reverse postorder.

gls?

Because it is based off the hashing algorithm, we get the benefits essentially for free. The same simplifications can be performed, but with the added knowledge of global congruences. Since the

majority of Factor's value numbering code is dedicated to the `rewrite` generic, it makes sense to reuse as much of that code as possible. Therefore, to convert Factor's local algorithm to a global one, I modified the existing code to use the RPO algorithm.

The most fundamental change is to the expression graph. Referring to Listing 1 on the following page, we see most of the same code as in **??** on page ??, with changes indicated by arrows (⟶). Two more global variables have been added, namely `changed?` and `final-iteration?`. The former is what we use to guide the fixed-point iteration. As long as value numbers are changing, we keep iterating. An important side effect of this is that we can no longer perform `rewrite` online, since the transformations we make aren't guaranteed to be sound on any iteration except the final one. This makes the RPO algorithm work *offline*, first discovering redundancies, then eliminating them in a separate pass. When this elimination pass starts, we'll set `final-iteration?` to **`t`**.

A key change is in the `vreg>vn` word, which now makes an optimistic assumption about previously unseen values. Given a new virtual register that wasn't in the `vregs>vns` table, the old version would map the register to itself, making the value its own canonical representative. However, if this version tries to look up a key that does not exist in the hash table, it will simply return **`f`** (which Factor will do by default with the **`at`** word). Therefore, every value in the CFG starts off with the same value "number", **`f`**. By the end of the GVN pass, there should be no value left that hasn't been put in the `vregs>vns` table, as we'll have processed every definition.

To keep track of whether `vregs>vns` changes, we simply need to alter `set-vn`. Here, we use **`maybe-set-at`**, a utility from the `assocs` vocabulary. This works like **`set-at`**, establishing a mapping in the hash table. In addition, it returns a boolean indicating change: if a new key has been added to the table, we return **`t`**. Otherwise, we return **`t`** only in the case where an old key is mapped to a new value. If an old key is mapped to the same value that's already in the table, **`maybe-set-at`** returns **`f`**. Therefore, when `vregs>vns` does change, we set `changed?` to **`t`** (which is what the **`on`** word does).

Finally, we define a new utility word, `clear-exprs`, which resets the `exprs>vns` and `vns>insns` tables. Unlike the local value numbering phase, we don't reset the entire expression graph. Instead, we make a pass over the whole CFG at a time. The only reason optimism works is that we keep trying to disprove our foolhardy assumptions. Really, `vregs>vns` establishes congruence classes of value numbers. At first, every value belongs in one class, **`f`**. We make a pass over the CFG to disprove whatever we can about this. If we've introduced new congruence classes (new values in the `vregs>vns` hash), we do another iteration. But each time, we use the congruence classes discovered from the previous iteration. At the start of each new pass, the expressions and instructions in `exprs>vns` and `vns>insns` are invalidated—their results are based on old information. So, these are erased on each iteration. Much like AWZ, we keep splitting classes until they can't be split anymore.

gls?

This logic is captured in Listing 2 on page 5. Rather than reset the tables when we start processing each basic block in `value-numbering-step` like before, we call `clear-exprs` on each iteration over the CFG in `value-numbering-iteration`. Note that `value-numbering-step` no longer returns the changed instructions, as we aren't replacing them online. `value-numbering-iteration` uses `simple-analysis` instead of `simple-optimization`, which only expects global state to change—no instructions are updated in the block. Much to our advantage, `simple-analysis` already traverses the CFG in reverse postorder, so we needn't worry about traversal order. The top-level word

```
! Copyright (C) 2008, 2010 Slava Pestov, 2011 Alex Vondrak.
! See http://factorcode.org/license.txt for BSD license.
USING: accessors kernel math namespaces assocs ;
IN: compiler.cfg.gvn.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

! boolean to track whether vregs>vns changes
SYMBOL: changed?

! boolean to track when it's safe to alter the CFG in a rewrite
! method (i.e., after vregs>vns stops changing)
SYMBOL: final-iteration?

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

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

: set-vn ( vn vreg -- )
    vregs>vns get maybe-set-at [ changed? on ] when ;

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

: clear-exprs ( -- )
    exprs>vns get clear-assoc
    vns>insns get clear-assoc ;

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

Should I keep the same vocab names?

Listing 1: The `compiler.cfg.gvn.graph` vocabulary

```
: value-numbering-step ( insns -- )
    [ simplify value-number ] each ;

: value-numbering-iteration ( cfg -- )
    clear-exprs [ value-numbering-step ] simple-analysis ;

: determine-value-numbers ( cfg -- )
    final-iteration? off
    init-value-graph
    '[
        changed? off
        _ value-numbering-iteration
        changed? get
    ] loop ;
```

Listing 2: Main logic in `compiler.cfg.gvn`

`determine-value-numbers` ties this all together by calling `value-numbering-iteration` until we can get through it with `changed?` remaining false.

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

M: insn simplify dup rewrite [ simplify ] [ ] ?if ;
M: array simplify [ simplify ] map ;
M: ##copy simplify ;
```

Listing 3: Iterated rewriting `compiler.cfg.gvn`

Note that the work of `value-numbering-step` is further divided into two words, `simplify` and `value-number`. These combine to do much the same work as `process-instruction` in **??** on page ??. `simplify` makes the repeated calls to `rewrite` until the instruction cannot be simplified further. Its definition is in Listing 3. We then pass the simplified instruction to `value-number`, which is defined in Listing 4 on the next page. This also has a similar structure to `process-instruction`. The main difference is that instructions are no longer returned (again, they aren't altered in place). So, the **`array`** method uses **`each`** instead of **`map`** to recurse into the results of `rewrite`.

A subtle change is necessary with the `alien-call-insn` and `##callback-inputs` methods. Whereas `process-instruction` merely skipped over certain instructions that could not be rewritten, here we don't have that luxury. We need to be careful to `set-vn` every virtual register that gets defined by any instruction. While making a pessimistic assumption, it didn't matter if we did this: any unseen value would be presumed important by `vreg>vn`. However, with the optimistic assumption, `vreg>vn` will give the impression that unseen values are all the same by returning **`f`**. Therefore, we simply record the virtual registers defined in instructions that may define one or

```
GENERIC: value-number ( insn -- )

M: array value-number [ value-number ] each ;

: record-defs ( insn -- ) defs-vregs [ dup set-vn ] each ;

M: alien-call-insn value-number record-defs ;
M: ##callback-inputs value-number record-defs ;

M: ##copy value-number [ src>> vreg>vn ] [ dst>> ] bi set-vn ;

: redundant-instruction ( insn vn -- )
    swap dst>> set-vn ;

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

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

M: ##phi value-number
    dup inputs>> values [ vreg>vn ] map sift
    dup all-equal? [
        [ drop ] [ first redundant-instruction ] if-empty
    ] [ drop check-redundancy ] if ;

M: insn value-number
    dup defs-vregs length 1 = [ check-redundancy ] [ drop ] if ;
```

Listing 4: Assigning value numbers in `compiler.cfg.gvn`

more of them. Specifically, `alien-call-insn` and `##callback-inputs` are classes that correspond to foreign function interface (FFI) instructions.

The `##copy` method uses `set-vn` the same way as before. `redundant-instruction`, `useful-instruction`, and `check-redundancy` are also largely the same. These have just been tweaked to not return instructions.

```
M: ##phi >expr
    inputs>> values [ vreg>vn ] map
    basic-block get number>> prefix
    \ ##phi prefix ;
```

Listing 5: ϕ expressions in `compiler.cfg.gvn.expressions`

The `##phi` method in Listing 4 on the preceding page represents a major change. Before, `##phi`s were left uninterpreted. Congruences between induction variables that flowed along back-edges would not be identifiable. But now, by checking for redundant `##phi`s, we may reduce them to copies. Each `##phi` object has an `inputs` slot, which is a hash table from basic block to the virtual register that flows from that block. Thus, there is one input for each predecessor. The `values` of the hash will be the virtual registers that might be selected for the `dst` value. We look up the value numbers of these, removing all instances of `f` with the `sift` word. If all of the inputs are congruent, we can call `redundant-instruction`, setting the value number of the `##phi`'s `dst` to the value number of its first input (without loss of generality). The `all-equal?` word will return `t` if the sequence is empty (as it's vacuously true), so we must make sure not to call `first` on the sequence, since this will be a runtime error. If the sequence is empty, we needn't note the redundancy, as the `##phi`'s `dst` will already have the optimistic value number `f` anyway. Otherwise, we call `check-redundancy`. The purpose of this is to identify `##phi`s that are equal to each other. Even if its inputs are incongruent, a `##phi` might still represent a copy of another induction variable. So that `check-redundancy` works, we also define a `>expr` method in `compiler.cfg.gvn.expressions`, as seen in Listing 5. Here, the expression is an array consisting of the `##phi` class word, the current basic block's number, and the inputs' value numbers. We include the basic block number because only `##phi`s within the same block can be considered equivalent to each other.

The final method in Listing 4 on the preceding page defines the default behavior for `value-number`, which calls `check-redundancy` on the simplified instruction if it defines a single virtual register. Note that we separate the `alien-call-insn` and `##callback-inputs` logic from this, since they happen to define a variable number of registers. If particular instances define only one register, we still don't want to call `check-redundancy` on them, since they don't have a `dst` slot. To avoid calling `dst>>` and triggering an error in `useful-instruction`, we needed separate methods for the FFI classes.

With these changes, we can globally identify value numbers, including equivalences that arise from simplifying instructions (even though no replacements are actually done yet). To illustrate this, consider again the example `0 100 [ 1 fixnum+fast ] times`, reproduced in Figure 1 on the next page. As the expression graph changes frequently in this new algorithm, instead of showing the literal hash tables we'll use a shorthand notation. Virtual registers will be integers,

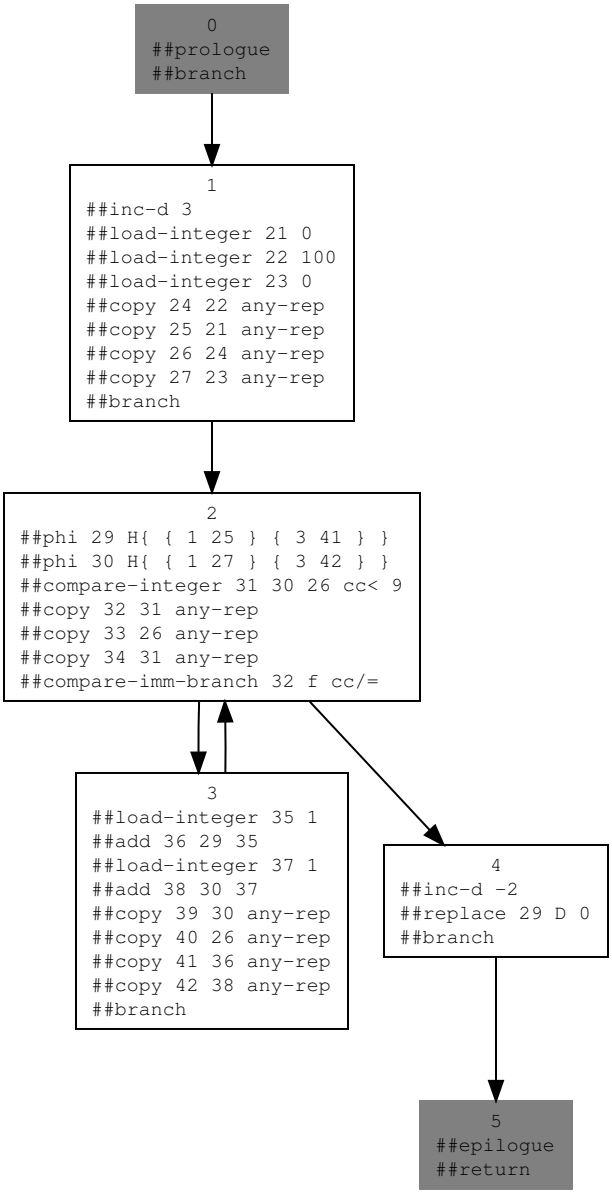

Figure 1: `0 100 [ 1 fixnum+fast ] times` before the new value numbering pass

and to avoid confusion value numbers will be written in brackets, like $\langle n \rangle$. Then, we'll show `vreg>vn` mappings with the notation $n \to \langle n \rangle$, where n is the register and $\langle n \rangle$ is the value number. If there is a corresponding expression in `exprs>vns`, it will be denoted after the mapping, like $n \to \langle n \rangle$ (*expression*). With the expressions, the instructions in `vns>insns` are a bit redundant for understanding the value numbering process, so they will be elided. Any mappings to `f` will be elided, as they're understood to be implicit when a key is absent.

Might make separate figures of each block, for easier reference

`determine-value-numbers` starts the first iteration, which of course starts at basic block 1. `##inc-d` is a no-op, but the first two `##load-integer`s are established as useful instructions. `##load-integer 23 0` is recognized as redundant, since at this point we know that `21` has the value `0`. The `##copy` instructions all pile on value number mappings, leaving us with the following:

$$\begin{aligned}
21 &\to \langle 21 \rangle \quad (0) \\
22 &\to \langle 22 \rangle \quad (100) \\
23 &\to \langle 21 \rangle \\
24 &\to \langle 22 \rangle \\
25 &\to \langle 21 \rangle \\
26 &\to \langle 22 \rangle \\
27 &\to \langle 21 \rangle
\end{aligned}$$

At iteration 1, basic block 2, the first `##phi` has inputs `25` (from block 1) and `41` (from block 3). The former has the value number $\langle 21 \rangle$, while the latter is still at `f`. We treat this value number much like a $\top$ element, unifying it with the other input to give us the assumption that `29` will be a copy of `25`. Thus, it gets the same value number. A similar choice happens for the second `##phi`. The instruction `##compare-integer 31 30 26 cc< 9` is an interesting case. Due to our optimistic assumptions thus far, we believe `30` is carrying the value `0`, and that `26` is set to `100`. Thus, this instruction gets constant-folded by `simplify` into `##load-reference 31 t`. The CFG isn't changed, but the expression graph reflects this belief. Later, this assumption will be invalidated. The following copies are processed as usual, with the distinct difference here that `##copy 33 26 any-rep` has the global knowledge of the value number of `26`. Because the `##compare-integer` was constant-folded, so is the `##compare-imm-branch`—and to the same value, no less. This leaves us with:

$$\begin{aligned}
21 &\to \langle 21 \rangle \quad (0) \\
22 &\to \langle 22 \rangle \quad (100) \\
23 &\to \langle 21 \rangle \\
24 &\to \langle 22 \rangle \\
25 &\to \langle 21 \rangle \\
26 &\to \langle 22 \rangle \\
27 &\to \langle 21 \rangle \\
29 &\to \langle 21 \rangle \\
30 &\to \langle 21 \rangle
\end{aligned}$$

$$
\begin{aligned}
31 &\to \langle 31 \rangle \quad (\texttt{t}) \\
32 &\to \langle 31 \rangle \\
33 &\to \langle 22 \rangle \\
34 &\to \langle 31 \rangle
\end{aligned}
$$

Block 3 of iteration 1 gives the `##load-integer`s' destinations the same value number, corresponding to the integer 1. Because optimism makes the algorithm think that `29` and `30` correspond to the integer 0, the `##add`s are constant-folded. This leaves us with:

$$
\begin{aligned}
21 &\to \langle 21 \rangle \quad (0) \\
22 &\to \langle 22 \rangle \quad (100) \\
23 &\to \langle 21 \rangle \\
24 &\to \langle 22 \rangle \\
25 &\to \langle 21 \rangle \\
26 &\to \langle 22 \rangle \\
27 &\to \langle 21 \rangle \\
29 &\to \langle 21 \rangle \\
30 &\to \langle 21 \rangle \\
31 &\to \langle 31 \rangle \quad (\texttt{t}) \\
32 &\to \langle 31 \rangle \\
33 &\to \langle 22 \rangle \\
34 &\to \langle 31 \rangle \\
35 &\to \langle 35 \rangle \quad (1) \\
36 &\to \langle 35 \rangle \\
37 &\to \langle 35 \rangle \\
38 &\to \langle 35 \rangle \\
39 &\to \langle 21 \rangle \\
40 &\to \langle 22 \rangle \\
41 &\to \langle 35 \rangle \\
42 &\to \langle 35 \rangle
\end{aligned}
$$

While block 4 is visited in each iteration, it doesn't define any registers, so doesn't affect the state of value numbering. Therefore, the above is the state left at the end of iteration 1.

Since `vregs>vns` clearly changed, iteration 2 commences by clearing the expressions, though the value numbers remain. Block 1 doesn't change from iteration 1, giving us:

$$
\begin{aligned}
21 &\to \langle 21 \rangle \quad (0) \\
22 &\to \langle 22 \rangle \quad (100) \\
23 &\to \langle 21 \rangle \\
24 &\to \langle 22 \rangle
\end{aligned}
$$

$25 \to \langle 21 \rangle$
$26 \to \langle 22 \rangle$
$27 \to \langle 21 \rangle$
$29 \to \langle 21 \rangle$
$30 \to \langle 21 \rangle$
$31 \to \langle 31 \rangle$
$32 \to \langle 31 \rangle$
$33 \to \langle 22 \rangle$
$34 \to \langle 31 \rangle$
$35 \to \langle 35 \rangle$
$36 \to \langle 35 \rangle$
$37 \to \langle 35 \rangle$
$38 \to \langle 35 \rangle$
$39 \to \langle 21 \rangle$
$40 \to \langle 22 \rangle$
$41 \to \langle 35 \rangle$
$42 \to \langle 35 \rangle$

Now that we're in iteration 2, the inputs to the `##phi`s of block 2 have been processed once before. For instance, we still believe that `25` corresponds to the integer 0 (which is incidentally correct), but now that `41` has the value number $\langle 35 \rangle$, we think it corresponds to the integer 1. While this is incorrect, it does break the congruence between the inputs, making the first `##phi` a useful instruction. The second `##phi`, however, still looks like a copy of the first. Even so, this is sufficiently different that the following `##compare-integer` cannot be constant-folded like before. However, it can still be converted to a `##compare-integer-imm`, as one of its operands corresponds to an integer. The redundant `##compare-imm-branch` gets rewritten to the same expression as the `##compare-integer`, so winds up getting the same value number. This gives us:

$21 \to \langle 21 \rangle$ (0)
$22 \to \langle 22 \rangle$ (100)
$23 \to \langle 21 \rangle$
$24 \to \langle 22 \rangle$
$25 \to \langle 21 \rangle$
$26 \to \langle 22 \rangle$
$27 \to \langle 21 \rangle$
$29 \to \langle 29 \rangle$ (`##phi 2 21 35`)
$30 \to \langle 29 \rangle$
$31 \to \langle 31 \rangle$ (`##compare-integer-imm 29 100 cc<`)
$32 \to \langle 31 \rangle$
$33 \to \langle 22 \rangle$

$34 \to \langle 31 \rangle$
$35 \to \langle 35 \rangle$
$36 \to \langle 35 \rangle$
$37 \to \langle 35 \rangle$
$38 \to \langle 35 \rangle$
$39 \to \langle 21 \rangle$
$40 \to \langle 22 \rangle$
$41 \to \langle 35 \rangle$
$42 \to \langle 35 \rangle$

Block 3 of iteration 2 also changes, since the `##add`s can't be constant-folded like before due to our new discovery about the `##phi`s. However, the first one can still be converted to an `##add-imm`, and the second is marked the same as the first. This leaves the following value numbers:

$21 \to \langle 21 \rangle$ (0)
$22 \to \langle 22 \rangle$ (100)
$23 \to \langle 21 \rangle$
$24 \to \langle 22 \rangle$
$25 \to \langle 21 \rangle$
$26 \to \langle 22 \rangle$
$27 \to \langle 21 \rangle$
$29 \to \langle 29 \rangle$ (`##phi 2 21 35`)
$30 \to \langle 29 \rangle$
$31 \to \langle 31 \rangle$ (`##compare-integer-imm 29 100 cc<`)
$32 \to \langle 31 \rangle$
$33 \to \langle 22 \rangle$
$34 \to \langle 31 \rangle$
$35 \to \langle 35 \rangle$ (1)
$36 \to \langle 36 \rangle$ (`##add-imm 29 1`)
$37 \to \langle 35 \rangle$
$38 \to \langle 36 \rangle$
$39 \to \langle 29 \rangle$
$40 \to \langle 22 \rangle$
$41 \to \langle 36 \rangle$
$42 \to \langle 36 \rangle$

Since the value numbers changed, we start iteration 3. The expressions are cleared, and block 1 once again does not change anything. The first `##phi` in block 2 still gets classified as useful, so no value numbers change. The major difference, though, is that the previous iteration's value

numbers for registers in block 3 update the expression we have for the `##phi`. Whereas before we thought it was choosing between $\langle 21 \rangle$ (the integer 0) and $\langle 35 \rangle$ (the integer 1), the `##add` wasn't constant-folded in the previous iteration. Therefore, the virtual register `41` now corresponds to the result of the `##add` with the value number $\langle 36 \rangle$. We still can't disprove that the second `##phi` is different (because it, in fact, isn't). So, we're left with the following after iteration 3 finishes with block 2:

$21 \to \langle 21 \rangle$ (0)
$22 \to \langle 22 \rangle$ (100)
$23 \to \langle 21 \rangle$
$24 \to \langle 22 \rangle$
$25 \to \langle 21 \rangle$
$26 \to \langle 22 \rangle$
$27 \to \langle 21 \rangle$
$29 \to \langle 29 \rangle$ (`##phi 2 21 36`)
$30 \to \langle 29 \rangle$
$31 \to \langle 31 \rangle$ (`##compare-integer-imm 29 100 cc<`)
$32 \to \langle 31 \rangle$
$33 \to \langle 22 \rangle$
$34 \to \langle 31 \rangle$
$35 \to \langle 35 \rangle$
$36 \to \langle 36 \rangle$
$37 \to \langle 35 \rangle$
$38 \to \langle 36 \rangle$
$39 \to \langle 29 \rangle$
$40 \to \langle 22 \rangle$
$41 \to \langle 36 \rangle$
$42 \to \langle 36 \rangle$

Blocks 3 and 4 do not produce any more changes, so GVN has stabilized after 3 iterations, with our final congruence classes being:

$$\langle 21 \rangle = \{21, 23, 25, 27\}$$
$$\langle 22 \rangle = \{22, 24, 26, 33, 40\}$$
$$\langle 29 \rangle = \{29, 30, 39\}$$
$$\langle 31 \rangle = \{31, 32, 34\}$$
$$\langle 35 \rangle = \{35, 37\}$$
$$\langle 36 \rangle = \{36, 38, 41, 42\}$$

teletype the numbers in the align*s, I guess