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numbering (at least, the earliest point where discussions in the literature seem to start) appears in an oft-cited but unpublished work of Cocke.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 $\mathtt{x} \cong \mathtt{y}$) 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 may still pick a topological order that ignores back-edges, but we may encounter operands whose values flow along those back-edges. We'll later address the issue of encountering instructions whose operands haven't been processed yet.

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.