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 **????**. 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 Future Work

The global value numbering (GVN) code presented in this thesis can be improved in various specific ways. Furthermore, the literature on GVN is extensive, and there are more overarching algorithmic strategies that have yet to be explored in the Factor code base. In this section, we explore some of these options for possible directions that Factor's compiler can take from here.

As it stands, the new pass could be smarter. For instance, it does not consider the commutativity of certain operations. This would be straightforward to solve by making `>expr` sort the operands of commutative instructions' expressions, thereby placing arguments in a canonical order. This would increase the number of congruences discovered between `##add`s, `##mul`s, and even `##phi`s. Also, the `copy-propagation` pass is remarkably similar to the new `value-numbering`—in fact, it could be removed altogether. All it does is collect global information about congruences as established by `##copy` instructions (by a similar fixed-point iteration), then replace the virtual registers of instructions with the original value (i.e., the one not established by a `##copy`). This allows `copy-propagation` to remove all `##copy` instructions. But the information calculated by `value-numbering` is a superset of this copy-equivalence data, so it should be easy to do global copy propagation in the GVN phase and save time on the redundant fixed-point iteration.

There remains an open question about the GVN implementation's use of availability, too. As it stands, it's rather strict: if the canonical value number for an expression is not directly available, `rewrite` gives up on reusing that value. However, the virtual registers which map to a single value number form a congruence class. We need not look just at the canonical leader (the first virtual

register in the whole program to compute the particular expression). `rewrite` could change an instruction to reuse any member of the congruence class that was available. It remains to be seen when and if such a rewrite would be useful or desirable.

Existing literature also gives plenty of material for a better implementation. We can make the existing reverse postorder (RPO) algorithm more efficient in practice by observing that the only times we need to iterate are where there are cyclic dependencies between values in the control flow graph (CFG). For instance, the example from **??** only has cyclic dependencies in the induction variables: the `##phi`s are defined by uses of virtual registers that are themselves defined by uses of the `##phi` targets in `##add`s. The RPO algorithm degenerates into the hash-based local algorithm of **??** on straight-line code. Thus, a more efficient algorithm will only iterate over the cycles between definitions instead of over the whole CFG.

Conceptually, we build a *value graph* (also known as *static single assignment (SSA) graph*) whose nodes represent definitions and directed edges represent uses. Since it just codifies def-use information, we needn't build an actual graph data structure. Using an algorithm due to Tarjan, SCCs of the value graph are iterated upon, while single nodes are processed only once. The strongly connected component (SCC) algorithm is more efficient than the RPO algorithm in practice, but the principles are the same. This gives us a comparatively simple, easily-extended GVN algorithm with complexity $O(nd)$, where n is the number of vertices in the value graph (i.e., the number of values we're numbering) and d is the *loop connectedness* (the maximum number of back edges on any acyclic path) of the value graph. Though d can theoretically be $O(n)$, in practice it seems to be bounded by a small constant. In Simpson'sexperiments, the maximum value of d was 4.

cite

cite-like

cite-like

A more thorough overhaul could incorporate further rewriting of the instructions. Gargi proposes a *predicated* value numbering algorithm that combines

- optimistic value numbering, thus emulating Simpson'sRPO and SCC algorithms;
- constant folding, algebraic simplification, and unreachable code elimination, thus emulating Click'sstrongest combination;
- global reassociation, thus performing the work already done in Factor;
- predicate inference, which can infer the values of comparisons dominated by some related predicate (i.e., comparisons in a block that is only reached via a particular conditional);
- value inference, which can infer the values of variables dominated by some related predicate (similar to range propagation, as seen in **??**); and
- ϕ-predication, which (if possible) associates each input of a ϕ with the predicate that controls the path that leads to the selection of that argument, thus letting us find flow-sensitive congruences.

cite

cite

This combination is given in a *sparse* formulation, which makes it efficient enough to apply all of these optimizations. Essentially, when optimistic assumptions are invalidated (which, of course, happens as we iterate until reaching the fixed point), instead of recalculating every result (as in the RPO algorithm), we only recalculate the values that may yet change from this new information (as in the SCC algorithm).

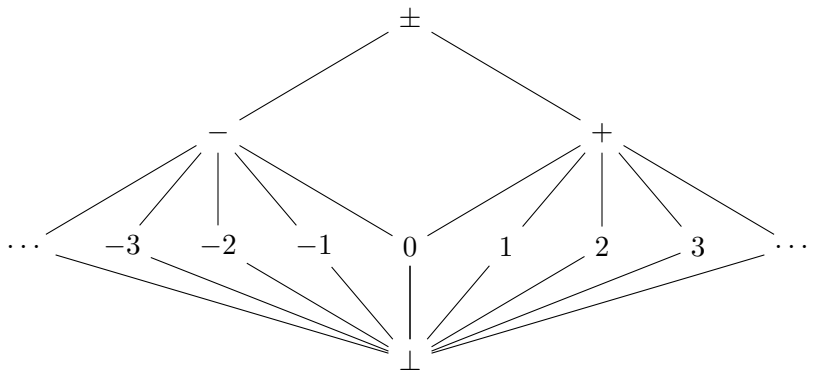

Figure 1: Abstract interpretation over signs

Any portion of Gargi's algorithm may be selectively disabled, thus letting us tweak it for specific compile time vs. code quality trade-offs we might have. It promotes a fairly good separation of concerns in the algorithm, too, letting the pseudocode be presented piecemeal for each optimization. Gargi presents compelling examples of predicated value numbering's strength, and its addition to Factor could prove very worthwhile.

cite-like

cite-like

Historically, the development of GVN algorithms has forked along two concurrent bloodlines. Those we've studied thus far reflect the more "practical" line, which was largely implementation-driven and less formal than the other algorithms. But those focused on formal reasoning have recently become much more viable, and the wealth of ideas from them are worthwhile.

For those acquainted with Chapter 9 of The Dragon Book, this work will seem familiar, as it's rooted in the results of Kildall and Cousot, upon which the chapter is based. The former was a precursor to GVN, in that it described an algorithm for common subexpression elimination that partitioned expressions into congruence classes. However, its method was phrased in terms of *lattices*, which are algebraic structures that we can reason about formally. This is as in The Dragon Book: a lattice is a partially-ordered set for which any two elements have a unique *least upper bound* (or *join*) and *greatest lower bound* (or *meet*). By defining meet and join operators on a partially-ordered set of abstract values, we can represent many sorts of analyses on our programs.

cite

cite

cite

cite-like

Cousot formalize the salient properties of such interpretation over lattices in a framework dubbed *abstract interpretation*. To understand it intuitively, consider some arithmetic expression like (-5×14). Our first inclination is probably to interpret it with respect to numeric values, but we can understand it in several different contexts. Let's use signs ($+$, $-$, and $\pm$) as our abstract domain and consider the operators to be defined by the rules of signs. Figure 1 shows a lattice we can use for this. Using a version of $\times$ cast in the context of this lattice, we can interpret (-5×14) as

cite

$$-5 \times 14 \quad \to \quad (-) \times (+) \quad \to \quad (-)$$

proving that (-5×14) is negative. Using this framework, the results are correct, but only useful within the confines of what we define. For instance, we can interpret $(-5 + 14)$ as

$$-5 + 14 \quad \to \quad (-) + (+) \quad \to \quad (\pm)$$

proving very little—the result is either positive or negative.

Despite the inherent limitations, we find the results useful as approximations of more complex properties. For example, we used congruence to approximate runtime equivalence. Only a year before AWZ was published, Steffen showed that Kildall's approach could be framed as abstract interpretation over *Herbrand equivalences*—that is, equivalences where operators are uninterpreted. This is actually the same notion of congruence we had from before: expressions are equivalent if their operators and operands are equivalent, irrespective of the result of applying the operator.

gls?
cite
cite-like

The primary strength of the abstract interpretation approaches are that they are *complete*; intuitively, there is no loss of information at each step of abstract interpretation. However, this "loss of information" is relative to the information encompassed by the abstract domains. While we can find all Herbrand equivalences, we aren't guaranteed to find equivalences induced by interpreting operators, which was effectively the work done by combining optimizations (e.g., constant folding is the interpretation of certain operators upon constant operands). So, while complete, these approaches vary in *preciseness*. Most of the work in the abstract interpretation of GVN did little to study the results of interpreting the same operators we saw before, but note it's a promising direction for future research.

cite

The cost of this completeness has traditionally been exponential time complexities. There have been several attempts to remedy this. RKS note AWZ is incomplete, since it treats ϕ functions as uninterpreted, so fails to discover congruences between ϕs and ordinary expressions. Their attempt to improve upon it alternately applies AWZ and the normalization rules

cite
gls?
gls?

$$\begin{aligned} \phi(a \otimes b, c \otimes d) &\rightarrow \phi(a, c) \otimes \phi(b, d) \\ \phi(x, x) &\rightarrow x \end{aligned}$$

until the partitioning reaches a fixed point. However, this is $O(n^2 \log n)$ in the expected case—$O(n^4 \log n)$ in the worst case—and it turned out to be incomplete not just in the presence of cyclesbut also in certain acyclic code.

cite RKS

Later, Gulwani04 furthered the quest for an efficient, complete GVN algorithm in a novel way by using *randomized interpretation* (which is what it sounds like). The paper even explored various interpretations—specifically of linear combinations, bitwise operators, memory loads/stores, and integer division—that could make results more precise. But it was still $O(n^4)$ and ran a small chance of making incorrect inferences due to its randomized nature. For compilers, this isn't really acceptable, though such a scheme could be used in things like program verification tools.

cite Gulwani07
cite
cite Nie

From their trip back to the drawing board, Gulwani07 returned with a polynomial time algorithm for GVN that is complete for all Herbrand equivalences among terms of a limited size. Choosing a size bound equal to the size of the entire program is clearly sufficient. Note, however, that this is specifically for Herbrand equivalences; they do not show their results for any interpreted operators, but note it's an important area for exploration. Adding to this, Nie present a similar algorithm, except based on SSA form. Both wind up using the same size restrictions to guarantee complexity. Both also use an additional special-purpose data structure to represent the set of Herbrand equivalences and to perform abstract evaluations over them, which adds a conceptual load to the algorithms and might make them more difficult to implement. However, unlike most other abstract interpretation-based algorithms, Nie's is demonstrably practical, as the authors implemented it for GCC. In their experiments, the size restriction turned out to be unnecessary for avoiding the exponential case, showing that the main bottleneck in complete GVN algorithms is typically their poor data structure choices.

cite
cite
cite
gls?

Clearly, there is much room for future exploration. Even without crossing the boundaries of GVN into the scope of other compiler optimizations, we can eliminate all sorts of redundancies. The literature has a wealth of algorithms that all warrant experimentation. With varying degrees of aggressiveness, there are several opportunities to make Factor a more efficient high-level language.