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 Results

Move the "list of shit I did" to the intro?

In total, the following code was produced for this thesis:

- A `graphviz` vocabulary, which provides bindings to create and manipulate graphs in Factor and output them using Graphviz.

  cite?

- The `compiler.cfg.graphviz` vocabulary, which uses the Graphviz bindings to output illustrations of Factor's low-level CFGs. This was responsible for the illustrations seen throughout the thesis.
- The `compiler.cfg.gvn` vocabulary, which was created by copying then modifying `compiler.cfg.value-numb` to make the pass global.

The goal of improving the optimization in Factor is, of course, to reduce the average running time of programs, and to do so without changing their semantics. Short of formal verification, the latter requirement makes it necessary to thoroughly test any code that gets compiled with the new pass enabled. To this end, we'll employ Factor's extensive unit test coverage. Because Factor is self-hosting, its standard vocabularies are written in Factor code, typically coupled with tests. While some vocabularies will have more test coverage than others, the total amount of tests is quite large. By compiling each vocabulary and running their tests, we're indirectly testing the compiler:

if tests that used to pass no longer do, then the new pass is changing the semantics of the code somehow. Though all tests passing does not guarantee the algorithm is correct, it does let us know that no known regressions have been introduced. Happily, with the new `value-numbering` phase enabled, all the old unit tests still pass.

The efficacy of the changes, on the other hand, must be measured relative to old benchmarks. Again, Factor has its bases covered, with a suite of 80 benchmarks run by the `benchmark` vocabulary. Each benchmark is run 5 times, where the garbage collector is run before each iteration. The minimum time from these runs is then used as the benchmark result. The data below comes from two separate runs of the `benchmarks` word, which invokes all the benchmark sub-vocabularies. The "before" time used the local value numbering, while "after" times had `value-numbering` replaced with the global value numbering (GVN) pass. The "change" is measured by the formula

$$\frac{\text{before} - \text{after}}{\text{before}} \times 100$$

to indicate the relative running times. Negative values in this column are good, as that means the running time has decreased.

Guess I should provide my PC's specs

Benchmark	Before (seconds)	After (seconds)	Change (%)
`benchmark.3d-matrix-scalar`	3.705816738	3.046126696	−17.80
`benchmark.3d-matrix-vector`	0.161298778	0.089539887	−44.49
`benchmark.backtrack`	4.280001561	2.358672591	−44.89
`benchmark.base64`	5.127831493	2.853612485	−44.35
`benchmark.beust1`	7.531546384	4.604929188	−38.86
`benchmark.beust2`	20.308680548	12.843534349	−36.76
`benchmark.binary-search`	3.729776895	2.349520427	−37.01
`benchmark.binary-trees`	9.403166818	6.518867479	−30.67
`benchmark.bootstrap1`	32.472196349	30.887877896	−4.88
`benchmark.chameneos-redux`	2.923900422	2.041007328	−30.20
`benchmark.continuations`	0.273525202	0.200695972	−26.63
`benchmark.crc32`	0.010623653	0.005282642	−50.27
`benchmark.dawes`	1.588111926	1.027176578	−35.32
`benchmark.dispatch1`	7.640720326	5.106558985	−33.17
`benchmark.dispatch2`	5.221652668	3.984754032	−23.69
`benchmark.dispatch3`	9.710520454	6.203527737	−36.12
`benchmark.dispatch4`	8.224931156	4.098265543	−50.17
`benchmark.dispatch5`	4.74357434	3.478219608	−26.68
`benchmark.e-decimals`	3.903754723	2.646958072	−32.19
`benchmark.e-ratios`	4.774454589	3.658075473	−23.38
`benchmark.empty-loop-0`	0.251816164	0.199189271	−20.90
`benchmark.empty-loop-1`	1.039242509	0.857588545	−17.48
`benchmark.empty-loop-2`	0.472215346	0.387974286	−17.84
`benchmark.euler150`	37.785852299	27.05450689	−28.40
`benchmark.fannkuch`	9.627490235	6.8970571	−28.36

Benchmark	Before (seconds)	After (seconds)	Change (%)
benchmark.fasta	7.25292282	5.640517069	−22.23
benchmark.fib1	0.179389215	0.164933805	−8.06
benchmark.fib2	0.205853157	0.138174211	−32.88
benchmark.fib3	0.785036151	0.539739186	−31.25
benchmark.fib4	0.391805799	0.260370111	−33.55
benchmark.fib5	1.508625224	1.002724851	−33.53
benchmark.fib6	19.202504502	13.146010511	−31.54
benchmark.gc0	7.360087104	5.508594031	−25.16
benchmark.gc1	0.418173431	0.281497214	−32.68
benchmark.gc2	25.611210221	19.716168704	−23.02
benchmark.gc3	2.757943071	2.210785891	−19.84
benchmark.hashtables	8.068216942	7.997106348	−0.88
benchmark.heaps	4.360368411	4.32169158	−0.89
benchmark.iteration	7.875561986	6.277891729	−20.29
benchmark.javascript	17.881224721	12.74204052	−28.74
benchmark.knucleotide	5.490420772	3.5704101	−34.97
benchmark.mandel	0.251711276	0.198695557	−21.06
benchmark.matrix-exponential-scalar	16.451432774	12.017000042	−26.95
benchmark.matrix-exponential-simd	0.681684747	0.536850343	−21.25
benchmark.md5	10.40516678	9.198666403	−11.60
benchmark.mt	33.91981743	29.961085146	−11.67
benchmark.nbody	9.203478441	6.795154145	−26.17
benchmark.nbody-simd	0.845814208	0.854773096	+1.06
benchmark.nested-empty-loop-1	0.097090973	0.068475608	−29.47
benchmark.nested-empty-loop-2	0.893126911	0.861327078	−3.56
benchmark.nsieve	1.086110659	1.137648699	+4.75
benchmark.nsieve-bits	2.707271763	2.815509077	+4.00
benchmark.nsieve-bytes	0.785041878	1.211421146	+54.31
benchmark.partial-sums	3.762171661	4.130144177	+9.78
benchmark.pidigits	2.182877913	2.195385034	+0.57
benchmark.random	5.66540782	5.71913683	+0.95
benchmark.raytracer	5.047070171	4.39514879	−12.92
benchmark.raytracer-simd	1.072588515	0.980927338	−8.55
benchmark.recursive	2.703509403	2.529087637	−6.45
benchmark.regex-dna	2.208584014	1.808859571	−18.10
benchmark.reverse-complement	2.801163847	2.353254665	−15.99
benchmark.ring	1.822206473	1.62482491	−10.83
benchmark.sfmt	2.675838657	2.463367198	−7.94
benchmark.sha1	11.964973943	11.142380303	−6.88
benchmark.simd-1	1.857778672	1.703206011	−8.32
benchmark.sockets	10.636346636	10.516448454	−1.13
benchmark.sort	0.695635429	0.581855635	−16.36
benchmark.spectral-norm	3.433630383	2.960833789	−13.77
benchmark.spectral-norm-simd	2.743240011	3.237017655	+18.00

Benchmark	Before (seconds)	After (seconds)	Change (%)
`benchmark.stack`	1.580016742	2.004478602	+26.86
`benchmark.struct-arrays`	2.180774222	2.421915609	+11.06
`benchmark.sum-file`	0.883097981	0.957151577	+8.39
`benchmark.terrain-generation`	1.611800222	1.887047663	+17.08
`benchmark.tuple-arrays`	0.262747557	0.329399609	+25.37
`benchmark.typecheck1`	1.750223408	1.674592158	−4.32
`benchmark.typecheck2`	1.674738245	1.553203741	−7.26
`benchmark.typecheck3`	1.891206648	1.735390184	−8.24
`benchmark.ui-panes`	0.305595039	0.29985214	−1.88
`benchmark.xml`	3.013709363	2.722223892	−9.67
`benchmark.yuv-to-rgb`	0.398174487	0.318891664	−19.91

The results are promising: of 80 benchmarks, only 13 showed any increase in running time. And of those, even fewer showed significant increases. Duplicated below for convenience are the benchmarks that ran slower, sorted in decreasing order of the percent difference between running times. We can see the last five or six benchmarks exhibited negligible differences—not only is the relative change tiny, but the absolute difference in running times is less than 0.1 seconds. (The `benchmark.tuple-arrays` results also show a similar absolute change, but it is relatively much larger.)

Benchmark	Before (seconds)	After (seconds)	Change (%)
`benchmark.nsieve-bytes`	0.785041878	1.211421146	+54.31
`benchmark.stack`	1.580016742	2.004478602	+26.86
`benchmark.tuple-arrays`	0.262747557	0.329399609	+25.37
`benchmark.spectral-norm-simd`	2.743240011	3.237017655	+18.00
`benchmark.terrain-generation`	1.611800222	1.887047663	+17.08
`benchmark.struct-arrays`	2.180774222	2.421915609	+11.06
`benchmark.partial-sums`	3.762171661	4.130144177	+9.78
`benchmark.sum-file`	0.883097981	0.957151577	+8.39
`benchmark.nsieve`	1.086110659	1.137648699	+4.75
`benchmark.nsieve-bits`	2.707271763	2.815509077	+4.00
`benchmark.nbody-simd`	0.845814208	0.854773096	+1.06
`benchmark.random`	5.66540782	5.71913683	+0.95
`benchmark.pidigits`	2.182877913	2.195385034	+0.57

Overall, even transitioning to a relatively simple GVN algorithm amounts to a positive change in Factor's compiler. More redundancies are eliminated, resulting in speedier programs. The implementation is at least as sound as the previous local value numbering pass, according to the unit tests, as all the same ones have passed.