

An optimizing C0 compiler in Haskell targeting x86-64

Correctness submission

gradescope.com/courses/1203615/assignments/7347240/submissions/408330129

Benchmark submission

gradescope.com/courses/1203615/assignments/7347239/submissions/408329223

Review commit hash

[216b5578e729b3527b11e1cb4dad0a832d4f445e](https://github.com/15-411-2026-Compiler-Design/commit/216b5578e729b3527b11e1cb4dad0a832d4f445e)

Throughout the report, whenever we talk about specific components of the compiler, we will link the relevant Haskell modules in our repository in margin notes, whether it be **Datatypes**, **Passes**, or other types of **Code**.

1. Structure of the compiler

In this section we provide a brief overview of the structure of our compiler, as well as some specific design challenges we encountered that we think are worth writing about.

1.1. Parsing

We initially used Alex [6] and Happy [12], the *de facto* standard Haskell lexer/parser generators, for our compiler. Alex allowed specification of token classes as regular expressions, while Happy allowed writing productions rules in a BNF-like form, not unlike that used by ANTLR [1]. Happy is also a LALR(1) parser generator and so generates very efficient code.

However, these tools proved to be finicky when interacting with the context-sensitive grammar of Lab 4. Any type alias defined by a typedef needs to be tracked, so that `alias *var;` is parsed as a pointer-typed declaration instead of a multiplication expression. Within the framework of Alex and Happy, this meant that we either need to perform direct inspection of the lexed token stream before parsing in order to find typedef `<ident>` patterns, or maintain a second rudimentary parser that does the same before the main parser.

Therefore, we eventually decided to switch to the Megaparsec [13] parser combinator library, which itself is a refinement of the Parsec [11] library. By exploiting the flexibility of Haskell's syntax and the expressivity of its type system, Megaparsec allowed for recursive descent parsers to be written in a naturalistic way. Due to the fact that the parser is in fact just regular program code, we were able to easily wire in state-tracking code that allowed us to finish parsing in one pass. We completed the migration with surprisingly few net line count delta.

1.2. Typechecking and static analysis (AST)

Haskell's facilities for algebraic data types and pattern matching simplified the typechecking pass significantly. However, the need for tracking state (for global declarations, and tracking the definedness of local declarations) and error reporting (when the typechecker encounters an error) required us to pull in a library for computational effects. As opposed to utilizing a monad transformer stack, we used the Effectful effect handlers library [18], which borrows ideas from algebraic effects.

One specific quirk we noticed was that a return statement marked all variables in scope as defined, although they might not have a value assigned to them yet. This means that code succeeding a return could refer to variables that do not have a def site, which broke analysis. This was resolved by removing all code succeeding returns in a scope.

1.3. TreeIR and TreeCFG

TreeIR is an IR that sits in the middle of a purely structural AST and an abstract assembly; it makes evaluation order explicit by isolating all effectful operations into standalone statements and flattens structural control flow into jumps, but still retains pure expressions.

We perform some preliminary optimizations on TreeIR, most notably inlining (2.1). We also transform the program into a CFG at this stage.

Datatypes

`Compile.Types.Cst`

Passes

`Compile.Parse.*`

`Compile.Desugar`

Datatypes

`Compile.Types.Ast`

Passes

`Compile.Elaborate.*`

`Compile.LowerAst`

Datatypes

`Compile.Types.Program`

`Compile.Types.TreeIr`

`Compile.Types.TreeCfg`

Passes

`Compile.LowerTreeIr`

`Compile.LowerTreeCfg`

1.4. Abstract Assembly (AASM) IR

The AASM IR is a three-operand abstract assembly language that operates over an abstract RISC machine with an unlimited number of virtual registers, or *temps*. This IR is where we perform the majority of optimizations, which we elaborate on in Section 2.

1.4.1. SSA construction and deconstruction

The two flavors of AASM are non-SSA and SSA. TreeCFG is initially lowered into non-SSA AASM, and then converted into SSA via an *SSA construction* pass. We make use of the standard dominance-based algorithm due to Cooper et al. [4] to insert ϕ -nodes and rewrite variable occurrences.

After the optimization pipeline, but prior to instruction selection, *SSA deconstruction* is carried out on AASM so that x86-64 assembly does not contain ϕ -nodes. In this stage, we resolve ϕ -nodes as *parallel moves* in their predecessor blocks, per the algorithm in Laurence et al. [17].

1.5. x86-64 Assembly (x86ASM)

The x86ASM IR is a representation of the subset of x86-64 instructions that are emitted by our compiler. It closely mirrors the structure of those instructions, with some caveats: 1) the program is still in CFG form, 2) it permits the use of temps instead of actual registers, and 3) it introduces a dedicated pseudoinstruction for zero-extension and truncation.

AASM is converted into x86ASM via the *instruction selection* pass; we then perform register allocation (2.14) and some peephole optimizations (2.6.3) specific to the x86-64 ISA. Of note, we decided against performing register allocation directly on the SSA-form AASM, as the technique, as described by Pereira and Palsberg [15], seemed too complex.

The final step in the compilation pipeline is *linearization*, which flattens the x86ASM CFG after register allocation so it may be written to a .s text file and assembled by as. This pass concerns code layout, which is discussed in Section 2.15.

1.6. Unsafe mode (`—unsafe`)

C0 is a *safe language*, in the sense that no undefined behavior occurs; any erroneous operation is instead checked in runtime and results in an exception. This is achieved by inserting checks before after any potentially erroneous operation: shifting by over 31 bits, accessing a null pointer, or indexing an array out of bounds. This results in performance penalties despite our best effort to statically remove redundant safety checks (2.7, 2.10).

The `—unsafe` compiler flag enables *unsafe mode*, where no safety checks listed above are emitted. It also introduces undefined behavior, making C0 more similar to C: indeed, every erroneous operation becomes undefined since the compiler is free to optimize with the assumption that the program is always well-behaved.

1.7. No-optimize mode (`-O0`)

`c8c` is an *optimizing compiler*, which means it spends considerable time on performing optimization that do not change the semantics of the program but improves its efficiency. However, there are scenarios where compile speed is more important than program efficiency. In that case, the `-O0` option enables the *no-optimize mode*, which skips performing all optimizations listed in this report, except ones integrated into the register allocator (2.14).

2. Optimizations implemented

We provide an exhaustive account of all optimizations we implemented in our compiler. Indeed, we implemented the good part of suggested optimizations listed in the Lab 5 handout [7], with the exception of some advanced ones such as live range splitting, fully general partial redundancy elimination (PRE), and loop optimizations such as loop unrolling and induction variable elimination (IVE).

Datatypes
Compile.Types.Cfg
Compile.Types.Aasm

Passes
Compile.Ssa.*

Datatypes
Compile.Types.x86asm

Passes
Compile.InstrSel

Datatypes
Compile.Types.LinearAsm

Code
Exec.Args

Passes
Compile.LowerAst

Code
Exec.Compile

Of note, any optimizations performed on SSA-form AASM (anything past 2.3 tail recursion elimination till 2.13 aggressive dead code elimination) are in a *quiescence loop*: this part of the optimization pipeline is repeatedly applied until the CFG no longer changes (reaches *quiescence*), or until a maximum of 5 rounds of pipeline has been run. This ceiling is determined so that all cases in the benchmark suite reach quiescence.

2.1. Inlining

It was suggested that inlining be performed on CFG's during lecture [8]. However, we performed inlining on the linear TreeIR, which we found to be easier as we do not need to consider interactions with ϕ -nodes. This also unfortunately introduces some limitations, such as not being able to interact with other SSA-based optimizations.

We made use of the following heuristic when deciding whether to inline a function:

$$\begin{aligned} \text{base cost} &= \sum_{i \in \text{instructions}} \text{cost}(i) \\ \text{penalty} &= 2 \times \min(20, \text{inner loop depth}) - \min(4, \text{outer loop depth}) \\ &\quad - \begin{cases} 8 & \text{if function accepts a constant/pointer} \\ 0 & \text{otherwise} \end{cases} \\ \text{inline if cost} &= \text{base cost} \times 2^{\text{penalty}} \leq 80 \end{aligned}$$

combined with a size ceiling of 5,000 base cost before inlining is disabled for a function entirely, in order to avoid size blowups and very high register pressure in pathological cases.

Inlining by itself brings relatively little benefit, perhaps pulling hot code closer and reducing the performance penalty of function calls in hot loops (hence the $-\min(4, \text{outer loop depth})$ penalty term). The main value instead lies in the optimizations it enables, by replacing a “black-box” function call with its actual code and thus allowing all intraprocedural optimizations to act on it and its surrounding context.

2.2. Accumulator transformation

The accumulator transformation pass, along with the tightly related tail recursion elimination (2.3), are the only passes performed on AASM *before* SSA construction. This is so we need not to care about the handling of ϕ -nodes, and it is made possible since these optimizations do not heavily benefit from the IR being in SSA form.

Accumulator transformation essentially transforms a *near*-tail-recursive function into a tail-recursive one by introducing an accumulator variable, a common trick in functional programming:

```
int fac(int a) {
    if (a == 0) return 1;
    return a * fac(a - 1);
}
    ⇨
int fac_worker(int a, int accum) {
    if (a == 0) return accum * 1;
    return fac_worker(a - 1, accum * a);
}
int fac(int a) { return fac_worker(a, 1); }
```

This transformation also fires on functions that call itself multiple times in the return statement, such as the naïve Fibonacci function. However, this does not make them fully tail recursive, as only one of the two calls undergo transformation. Thus, the performance gain in this case is more limited.

2.3. Tail recursion elimination

Tail recursive functions undergo tail recursion elimination, which eliminates recursive calls and turn them into loops. This is a highly fruitful optimization for functions that are candidates, since it both eliminates the overhead of function calls, and removes stack growth entirely:

Passes
Compile.Inline

Code
Compile.CallGraph

Passes
Compile.Accum

Passes
Compile.Tailrec

```

int fac_worker(int a, int accum) {
    if (a == 0) return accum * 1;
    return fac_worker(a - 1, accum * a);
}
    ⇒
int fac_worker(int a, int accum) {
    loop { // unconditional loop
        if (a == 0) return accum * 1;
        (a, accum) = (a - 1, accum * a); // parallel move
    }
}

```

A generalization of tail recursion elimination is *tail call optimization (TCO)*, which translates non-self-recursive calls in the tail position to jumps as well. We did not implement this as there were not many such code patterns in the benchmark suite. This would also require us to perform stack frame teardown prior to tail calls, so it is not a trivial extension of tail recursion elimination.

2.4. Purity analysis

Purity analysis does not perform any optimization by itself, but traverses the program call graph, analyzes and attaches purity information to each function definition (*i.e.* what types of side effects they may perform), which in turn are useful to optimizations downstream. We delineate side effects into three types: exceptions, memory read, and memory write. Each function may have zero or more of these flags set.

Functions with fully encapsulated memory operations, *i.e.* those that do not read memory locations passed in and do not leak allocated memory outside, may be considered pure (*i.e.* neither reading nor writing memory). For the sake of time, we implemented a conservative approximation of this analysis, namely that *functions are memory-pure if they do not receive or return pointer parameters*, which is sound since C0 does not support casting from or into a pointer.

2.5. Type-based alias analysis (TBAA)

A full points-to analysis such as Steensgaard's algorithm [19] is outside of our scope. However, exploiting C0's lack of type casts, we implemented a rudimentary version of type-based alias analysis (TBAA) [5]:

- Pointers to distinct allocations never alias.
- Pointers to the same allocation with provably disjoint offsets never alias.
- Pointers to distinct types never alias. Array lengths have the `ArrayLength` pseudo-type that doesn't alias regular ints.

These rules were able to render multiple no-alias verdicts that prove to be useful in redundant load/store elimination (RLE/RSE: 2.9, 2.11).

2.6. Peephole optimizations

Peephole optimizations in our compiler are divided into two passes: one pass performs ISA-agnostic peepholes, which are included in the SSA-based optimization pipeline (and thus in the quiescence loop). The other pass performs a smaller number of peepholes specific to the x86-64 ISA, and fires once after register allocation.

2.6.1. Algebraic laws and strength reduction

We apply multiple algebraic laws to reduce the overall instruction count, not dissimilar from LLVM's `InstCombine` pass:

- Associativity, *e.g.* $(x + c_1) + c_2 = x + \llbracket c_1 + c_2 \rrbracket$
- Distributivity, *e.g.* $(x * c_1) + (x * c_2) = x * \llbracket c_1 + c_2 \rrbracket$
- Cancellation, *e.g.* $b + (a - b) = a$
- Duality, *e.g.* $a + (-b) = a - b$

We also apply strength reduction to arithmetic operations:

- Multiplication: $a * 2 = a + a$, $a * 2^c = a \ll c$;
- Unit and zero elements: $a \parallel 0 = a$, $a \parallel \bar{0} = a$, $a \parallel \bar{a} = -1$, ...

Passes

Compile.Optimize.Purity

Passes

Compile.Optimize.Alias

Passes

Compile.Optimize.Peep

Code

Compile.Optimize.SsaDb

2.6.2. Constant division rewrite

Integer division by a constant can be rewritten into a series of less expensive arithmetic operations such as multiplication and shift, as discussed in *Hacker's Delight* [22:Ch.10]. Specifically, division by powers of 2 has a less involved form than arbitrary divisors. We implemented both in our compiler; the addition of an `Imul32Hi` AASM instruction (which compiles down to the single-operand form of `imul` during instruction selection) reduced the size of emitted division code further by 2-3 instructions each.

2.6.3. x86-64-specific optimizations

There are roughly two groups of x86-64-specific optimizations:

- Equivalent instructions with shorter encoding. This includes:

```

mov reg, 0  $\mapsto$  xor reg, reg
cmp reg, 0  $\mapsto$  test reg, reg
xor reg64, reg64  $\mapsto$  xor reg32, reg32
add/sub r/m, 1  $\mapsto$  inc/dec r, m
mov rsp, rbp; pop rbp  $\mapsto$  leave

```

- `lea`-based optimizations. `lea` can act as a three-operand instruction for some addition and multiplication:

```

mov dst, src; add dst, rhs  $\mapsto$  lea dst, [src + rhs]
imul reg1, reg2, 3/5/9  $\mapsto$  lea reg1, [reg2, reg2 * 2/4/8]

```

These optimizations are scheduled *after* register allocation, since most of them require some or all of their operands to be in registers, which we cannot be sure of before register allocation (some temps may get spilled to stack).

2.7. Sparse conditional constant propagation (SCCP)

We implemented SCCP according to Wegman and Zadeck's description in their paper [23] without much alteration.

2.8. CFG simplification

Cooper's textbook describes various forms of useless control flow that could be eliminated [3:Ch.10.2.2]. We approached them differently in our compilers.

- *Redundant branches*, where two destinations of a branch coincide, cannot happen in our compiler.
- *Branch hoisting* is also known as jump threading. We implemented this optimization but ultimately find it of no help to performance, thus removed it.
- *Removing empty blocks* is a subset of *dead block elimination*, where blocks with no predecessors are removed. This runs after every pass that may change control flow, namely SCCP and ADCE (2.7, 2.13).
- *Block combination* combines any two blocks $A \rightarrow B$ if there are no other successors of A or predecessors of B . This too runs after every pass that may change control flow.

Block combination turns out to be especially useful, since it allows some local (*i.e.* per-block) optimizations to fire when they otherwise would not.

2.9. Global value numbering (GVN)

We implemented GVN as introduced in the guest lecture [20]. We noticed that GVN subsumes copy propagation, so we did not need to introduce a separate pass for it.

As an extension, our GVN implements a limited form of redundant load elimination (RLE). Loads are numbered similar to pure instructions. They are allowed to propagate down the dominance tree, unless:

- A block has more than one predecessors, so not all control paths guarantee the loads are up to date;
- A store to an address that may alias that load, or call to a function that may write memory, happened.

Passes
Compile.RegAlloc.
PostPeep

Passes
Compile.Optimize.Sccp

Passes
Compile.Optimize.
MergeCfg

Passes
Compile.Optimize.Gvn

2.10. Redundant safety check elimination

Provably redundant arithmetic safety checks are eliminated as a direct consequence of SCCP (2.7). However, those involving memory need special mechanism:

- A *null check* may be eliminated if we know it points to an allocation, or if a null check for the same base pointer has been performed somewhere higher in the dominance tree.
- The case for *array bounds checks* is more interesting. If the size of an array allocation is statically known, we may propagate that to all loads tagged as `ArrayLength` for the same base address. If the comparison is with a statically known subscript, then SCCP will resolve it to an unconditional jump, naturally achieving array bounds check elimination.

2.11. Simple redundant store elimination (RSE)

We perform *local* redundant store elimination, that is, within every basic block. This is the most small-scale optimization pass in our compiler, and it provides modest benefits. Just like how RLE integrates with GVN, RSE theoretically could integrate with ADCE. However, a practical implementation would likely require techniques such as Memory SSA [14], which we did not have time to implement.

Passes
Compile.Optimize.Rse

2.12. Loop invariant code motion (LICM)

We ultimately decided against implementing partial redundancy elimination (PRE), since it is complex to implement, partially overlaps with GVN, and is likely expensive to run. We also briefly considered implementing a generalization of *both* PRE and GVN, the *Value-Based PRE* algorithm by VanDrunen and Hosking [21], but decided that its theoretical elegance did not outweigh its overwhelming complexity.

Passes
Compile.Optimize.Licm

Instead, we relied on the combination of GVN and LICM, which suffice for most tasks. For LICM, we also attempt to hoist loads in the loop body if we can prove that it is safe to do so, that is, either:

- The load is located in the loop header, therefore will always run;
- The load is a provably in-bounds access to a provably non-null pointer.

2.13. Aggressive dead code elimination (ADCE)

We implemented ADCE as introduced in lecture [9]. During that, we encountered one pitfall that was not covered in lecture: a live ϕ -node would need to mark branches in its predecessors as live, too. Otherwise, such branches may be rewritten to jump *past* the block containing the ϕ -node, resulting in miscompilation.

Passes
Compile.Optimize.Adce

2.14. Improvements to register allocation

Our compiler makes use of a Briggs-style [2] graph-coloring register allocator. Since we perform register allocation after SSA deconstruction, we did not use MCS-based coloring. Our coalescing strategy is the recommended *greedy coalescing* [16], which yields acceptable results.

Passes
Compile.RegAlloc.*

Early in the project, we briefly considered introducing a linear scan register allocator to improve compilation speed, but ultimately decided against it since such an implementation would also need live range splitting to generate good code.

2.14.1. Spill cost

The spill cost heuristic that we use to decide spill candidates is the same one as presented in lecture [10]:

$$\text{spill cost} = \sum_{x \in \text{def/use sites}} \exp(\text{loop depth of } x),$$

Code
Compile.RegAlloc.Tally

However, there were pathological cases where the register allocator kept spilling spill reload temps, since they have very few use sites and were therefore seen as cheap to spill. This eventually results in the allocator making no progress. We guarded against this by assigning all spill reload temps a spill cost of $+\infty$.

2.15. Code layout

For any given CFG, laying the blocks in reverse postorder (RPO) is a good baseline for good layout. In addition, we made some efforts to further improve layout:

- **Branch weighting:** When presented with two branch targets to lay after a block, we attempt to guess which branch will be “hot” and put that branch immediately after:
 - We consider branches to be hotter when they have a deeper loop depth (indication of loop body).
 - We consider branches to be colder when they end in a raise (indication of safety check failure).
- **Redundant jumps:** When a block is directly adjacent to its jump target, that jump may be omitted entirely. Additionally, conditional branches are translated into two jumps; when either of those targets are immediately succeeding, we eliminate one of the jumps (`jcc'` is the negation of `jcc`):

```

        jmp .L;   .L: ↦      .L:
jcc .L1; jmp .L2; .L2: ↦ jcc .L1; .L2:
jcc .L1; jmp .L2; .L1: ↦ jcc' .L2; .L1:

```

3. Benchmark results

We benchmarked our compiler on the provided benchmark suite under a variety of configurations listed below; these benchmarks are run sequentially on the Intel Core i9-13900H mobile processor with 32GB of memory.

- With either `-O0` or `-O1`, and with either `—unsafe` or `without`;
- Under `-O1 —unsafe`, but with individual passes disabled. Tested passes include accumulation transformation, ADCE, register coalescing, GVN, inlining, LICM, peephole, purity analysis, RSE, SCCP, CFG simplification, and tail recursion elimination.

Benchmark results are listed in Table 1. We make a few observations on notable passes:

- Inlining (2.1) was by far the most influential pass; by replacing black-box function invocations with their bodies, it unlocks numerous optimizations that require considering the interaction between the caller and the callee.

However, there was one case (`julia`) where inline severely *negatively* affected performance. Upon inspection of the generated code, it was found that the pure `collatz()` function was inlined into `render_collatz()`. Originally, the call to `collatz()` was loop-invariant, and so could be hoisted by LICM. However, the inlining introduced control flow which LICM could not hoist. One way to resolve this is to more aggressively penalize pure function calls inside loops.

- Accumulator transformation (2.2) and tail recursion elimination (2.3) were both high-impact passes with narrow scope: they are very effective in optimizing programs that frequently call tail-recursive or near-tail-recursive functions, but do not have productive interactions with other passes.
- GVN (2.9) has a modest speedup effect on many benchmark cases, but none very significant (>20%). The main role of GVN is copy propagation and common subexpression elimination, which by itself reduces the work of the final program, but rarely provides useful information for downstream passes.
- Surprisingly, SCCP (2.7) has little effect on the performance. We suspect that in unsafe mode, the benchmark suite does not have a meaningful amount of statically determinable branches; thus, SCCP is mostly reduced to the role of simple constant propagation.
- ADCE (2.13) proves to be highly impactful on select passes that deliberately introduce dead loops. Of note is the `jen` case, which also depends on the peephole passes. This is the most visible case in the suite that depends on multiple optimizations working in concert:
 - Algebraic identities rely on the insight provided by the peephole pass to render them dead;
 - ADCE must be in the pipeline to remove such dead code;
 - The x86-specific PostPeep pass is needed to remove self-moves (an artifact of register allocation).

- The cycle count of the *pierre* case fluctuates wildly in the `no-simplify` configuration. After investigation, we are confident that this is an extreme case of fluctuation, as the produced assembly was identical to that of other configurations.

3.1. Executable size

We additionally measured the executable sizes generated by different compilation modes (`-O0/-O1`, `-unsafe`). The results are presented in Table 2. For `unsafe` mode, optimization generally increases executable size somewhat, although not significantly; this is likely due to inlining. In cases where considerable dead code (e.g. `jen`), executable size is considerably reduced by ADCE.

For `safe` mode, the difference in executable size between optimization and not is more evenly split. While inlining introduces bloat, there are also a considerable amount of statically eliminable safety checks that are removed in `-O1`.

3.2. Compile time

We additionally measured the compile times of different compilation modes (`-O0/-O1`, `-unsafe`). The results are presented in Table 3. In general, both `safe` mode and optimization increase compile time, with `safe` mode’s effect more even over the board and optimization’s effect more concentrated on “large” cases ripe of optimization opportunities.

In a few cases, optimization resulted in shorter compile time. This is usually again due to the removal of dead code, which reduced the work of the register allocator. It is likely possible to reduce the compile time penalty introduced by optimizations by implementing some form of caching of analysis results, including dominator trees, def-use chains, and dataflow analyses, or by combining passes in order to reduce the number of traversals over the CFG.

4. Discussion

We conclude this report by discussing some other relevant topics not already covered in the preceding sections.

4.1. Pass ordering

How pass ordering affects pass interaction has been a topic emphasized in class. However, most of the optimizations in our compiler are placed in a quiescence loop, each pass would continuously feed information to other passes *both* downstream and upstream (in the next iteration of the loop).

Thus, it seemed that pass ordering would not affect the final output of the optimization pipeline, but merely affects how many rounds it might take in order to reach quiescence. One theoretical risk is that two passes will constantly produce code patterns that are targets of each other, thus never allowing the code to reach quiescence. In practice, we do not see this happening with any of our passes for any case in the benchmark suite.

4.2. Future work

Most of the future work that seem fruitful center around memory. A reasonable next step would to implement some form of points-to analysis, such as Steensgaard’s algorithm [19]. Memory SSA might also make certain classes of memory optimizations easier [14]. In addition, some form of interprocedural analysis seems useful.

The largest performance bottleneck of our compiler is currently register allocation. Due to Haskell’s immutable-by-default nature, we used persistent data structures such as radix trees to store the interference graph. This leads to highly suboptimal performance when the interference graph is dense; indeed, this alone has caused us anywhere between 10–20 compiler timeouts. A refactor of the register allocator to use a non-persistent, mutable bit array as an adjacency matrix would likely improve performance significantly.

Benchmark	Configuration	o0	o0-safe	o1	o1-safe	no-accum	no-ADCE	no-coalesce	no-GVN	no-inline	no-LICM	no-peep	no-postpeep	no-purity	no-RSE	no-SCCP	no-simplify	no-tailrec
Geometric mean		2.31	2.77	1.00	1.13	1.07	1.19	0.98	1.03	1.18	1.01	1.12	1.10	1.01	1.00	1.00	1.06	1.19
albert		3.28	4.43	1.00	1.05	0.96	1.20	0.95	0.99	2.93	0.96	0.96	1.05	0.96	0.95	1.09	0.95	0.96
arrays_and_loops		2.77	2.97	1.00	0.96	0.97	0.96	0.95	0.99	0.98	1.41	1.09	0.98	0.98	0.96	1.08	0.97	0.99
daisy		1.25	1.41	1.00	1.00	0.97	0.96	0.96	0.99	0.94	0.97	1.24	1.00	0.98	0.96	0.95	0.96	0.97
danny		3.26	3.30	1.00	1.24	0.99	1.01	0.93	1.10	2.61	0.90	0.74	1.16	0.99	0.96	1.10	0.96	0.96
fannkuch		1.14	1.97	1.00	1.15	0.99	0.98	0.91	0.95	1.19	0.89	0.98	0.92	0.99	0.99	0.87	0.98	1.00
frank		2.69	3.48	1.00	1.58	0.98	0.98	0.95	1.06	2.63	1.01	0.99	1.03	0.98	0.98	1.00	0.98	1.31
georgy		1.55	1.52	1.00	0.89	1.00	0.99	0.97	1.00	1.48	1.00	0.97	0.90	0.99	1.00	0.97	0.99	1.00
jack		0.98	2.21	1.00	1.84	1.01	0.99	0.93	1.07	0.96	0.98	1.04	0.98	1.04	1.08	0.96	1.08	1.03
janos		1.05	1.13	1.00	1.06	1.01	1.00	1.04	1.01	1.05	1.02	0.99	1.00	1.01	1.04	0.98	1.00	1.11
jen		3.72	3.83	1.00	1.01	0.99	2.79	1.07	1.10	0.99	1.12	2.28	1.90	0.99	1.01	1.10	0.99	1.00
julia		2.81	2.83	1.00	1.02	1.01	1.00	1.04	1.19	0.21	0.99	2.66	1.14	0.99	1.00	1.04	0.99	1.00
leonardo		1.43	1.61	1.00	1.06	1.50	0.99	0.98	1.12	1.00	0.99	1.13	1.01	0.99	1.00	0.98	0.99	1.43
loooops		1.72	1.80	1.00	0.85	1.20	1.20	0.99	1.00	1.03	1.01	1.00	1.11	1.00	0.99	0.97	1.00	1.24
mat		0.99	1.77	1.00	1.69	0.99	0.99	0.99	1.12	1.00	1.00	0.99	1.02	1.00	0.98	0.98	1.01	1.00
mist		95.69	96.53	1.00	0.99	1.03	9.14	1.00	0.95	1.02	0.99	1.02	1.01	1.20	1.02	1.00	1.00	1.04
monica		2.08	2.10	1.00	1.01	2.13	1.00	1.00	1.00	1.00	1.00	1.00	1.68	1.00	1.00	1.00	1.00	3.09
ncik		1.33	1.46	1.00	1.02	1.00	1.00	1.00	1.01	1.01	1.00	1.32	1.07	1.01	1.00	1.00	1.01	1.00
pierre		4.50	4.26	1.00	1.08	1.12	1.04	1.00	0.90	1.00	1.09	1.10	1.23	1.00	1.00	1.00	3.33	4.43
ronald		1.25	1.77	1.00	1.47	1.04	1.04	1.03	1.11	1.28	1.03	1.03	1.23	1.05	1.05	1.00	1.05	1.04
yyb		2.76	3.70	1.00	1.12	1.00	1.00	1.00	1.03	2.77	1.02	1.00	1.00	1.00	1.00	1.02	1.03	1.00

Table 1: Relative cycle count (-o1 —unsafe baseline). Red cells indicate regression and blue cells indicate speedup.

Configuration	Geometric mean	albert	arrays_and_loops	daisy	danny	fannkuch	frank	georgy	jack	janos	jen	julia	leonardo	looops	mat	mist	monica	ncik	pierre	ronald	yyb
o0	1.02	0.91	1.02	1.01	0.83	1.02	1.16	0.83	0.71	0.94	2.53	1.05	0.98	1.03	0.93	1.01	1.00	0.98	1.00	0.87	1.29
o1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
o0-safe	1.27	1.36	1.09	2.28	1.10	1.17	1.39	1.11	1.18	1.18	2.54	1.22	1.06	1.10	1.15	1.09	1.12	1.36	1.01	1.24	1.53
o1-safe	1.12	0.91	1.03	1.79	1.22	1.13	1.20	1.21	1.07	1.30	1.01	1.08	1.11	1.01	1.20	1.08	1.09	1.19	1.01	0.96	1.08

Table 2: Relative executable sizes of different compilation modes (-o1 —unsafe baseline). Red cells indicate larger than baseline and blue cells indicate smaller.

o0	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
o1	1.55	2.11	1.46	1.80	3.12	0.89	1.43	2.25	3.64	1.44	0.69	1.20	1.68	1.00	1.92	1.48	0.74	1.08	1.02	1.74	4.21
o0-safe	1.43	2.72	1.80	2.43	1.34	1.02	1.29	1.07	2.18	1.14	1.14	1.38	1.29	1.17	1.58	1.33	1.34	1.45	0.80	2.18	1.29
o1-safe	2.34	4.22	2.13	5.59	5.86	1.60	2.15	3.23	6.21	4.14	0.69	1.76	3.17	1.37	2.64	1.15	1.45	3.86	0.84	2.39	1.52

Table 3: Relative compile times of different compilation modes (-o0 —unsafe baseline). Red cells indicate longer than baseline and blue cells indicate shorter.

Bibliography

- [1] The ANTLR Project. 2012. ANTLR (ANother Tool for Language Recognition) v4. Retrieved from <https://github.com/antlr/antlr4>
- [2] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1994. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 428–455. <https://doi.org/10.1145/177492.177575>
- [3] Keith D Cooper and Linda Torczon. 2022. *Engineering a compiler*. Morgan Kaufmann.
- [4] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1–10 (2001), 1–8.
- [5] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*, 1998. Association for Computing Machinery, Montreal, Quebec, Canada, 106–117. <https://doi.org/10.1145/277650.277670>
- [6] Chris Dornan and Simon Marlow. 1995. Alex: a lexical analyser generator for Haskell. Retrieved from <https://github.com/haskell/alex>
- [7] Seth Copen Goldstein. 2026. 15-411 Compiler Design, Lab 5 (Spring 2026). Retrieved from <https://www.cs.cmu.edu/~411/labs/lab5.pdf>
- [8] Seth Copen Goldstein. 2026. 15-411 Lecture Slides: Function Inlining (Spring 2026). Retrieved from <https://www.cs.cmu.edu/~411/slides/21-inlining.pdf>
- [9] Seth Copen Goldstein. 2026. 15-411 Lecture Slides: Optimization 1 (Spring 2026). Retrieved from <https://www.cs.cmu.edu/~411/slides/17-opt1.pdf>
- [10] Seth Copen Goldstein. 2026. 15-411 Lecture Slides: SSA-based Register Allocation (Spring 2026). Retrieved from <https://www.cs.cmu.edu/~411/slides/03-registerallocation.pdf>
- [11] Daan Leijen and Paolo Martini. 1999. Parsec: a monadic parser combinator library. Retrieved from <https://github.com/haskell/parsec>
- [12] Simon Marlow and Andy Gill. 2001. The Happy parser generator for Haskell. Retrieved from <https://github.com/haskell/happy>
- [13] Megaparsec contributors. 2015. Megaparsec: industrial-strength monadic parser combinator library. Retrieved from <https://github.com/mrkrp/megaparsec>
- [14] Diego Novillo and others. 2007. Memory SSA—a unified approach for sparsely representing memory operations. 2007.
- [15] Fernando Magno Quintao Pereira and Jens Palsberg. 2009. SSA elimination after register allocation. In *International Conference on Compiler Construction*, 2009. 158–173.
- [16] Frank Pfenning, Rob Simmons, and Jan Hoffman. 2026. 15-411 Lecture Nodes: Optimizations of Register Allocation (Fall 2018). Retrieved from <https://www.cs.cmu.edu/~janh/courses/411/18/lec/18-regopt.pdf>
- [17] Laurence Rideau, Bernard Paul Serpette, and Xavier Leroy. 2008. Tilting at Windmills with Coq: Formal Verification of a Compilation Algorithm for Parallel Moves. *J. Autom. Reason.* 40, 4 (May 2008), 307–326. <https://doi.org/10.1007/s10817-007-9096-8>
- [18] Andrzej Rybczak. 2021. Effectful: An easy to use, fast extensible effects library. Retrieved from <https://github.com/haskell-effectful/effectful>

- [19] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, 1996. Association for Computing Machinery, St. Petersburg Beach, Florida, USA, 32–41. <https://doi.org/10.1145/237721.237727>
- [20] Ben Titzer. 2026. 15-411 Lecture Slides: Global Value Numbering and Control Flow Optimizations (Spring 2026). Retrieved from <https://www.cs.cmu.edu/~411/slides/19a-gvn.pdf>
- [21] Thomas VanDrunen and Antony L. Hosking. 2004. Value-Based Partial Redundancy Elimination. In *Compiler Construction*, 2004. Springer Berlin Heidelberg, Berlin, Heidelberg, 167–184.
- [22] Henry S Warren. 2013. *Hacker's delight*. Pearson Education.
- [23] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210. <https://doi.org/10.1145/103135.103136>