6.110 Compiler Design Document

Luca, Hannah, Maya, Lili

Spring 2024

I. Design

Welcome to our compiler!

In this section, we detail at a high-level how our compiler works, discussing our strategies for scanning, parsing, semantic checking, low-level IR generation, codegen, and dataflow optimizations, and conclude with a deep dive into our register allocation and further optimizations implemented for phase 5.

I. Scanning and Parsing

Our scanner operates through Regular Expressions. For each keyword, we provide a RegEx description and then run through the text file, consuming the next available token from our current position. We output a list of token nodes, which contain the location and text of the token as well as its type.

For parsing, we use a functional visitor pattern to implement a recursive descent parser. Specifically, each function corresponds to a particular pattern and returns that pattern's object for the parse tree. When doing pattern matching, we do so greedily, checking if there is any possible match from the incoming token, occasionally looking one more token ahead, such as when matching "+" versus "++".

Our final result after parsing and scanning is a parse tree of the entire program. Each node contains its type (field declaration, if loop, block, etc.), location information, and its relevant children, such as the method block for a method declaration.

II. Semantic checks

Our high-level IR is largely the same as the tree outputted by our parser, with the addition of scope information and symbol tables attached to the program structure, as well as to every method and block. Our semantic checking has two stages; the building stage, where we construct the symbol tables and scope information, and the checking stage, where we traverse our scope-augmented parse tree and perform additional semantic checks.

In the building stage, which occurs in our `IRBuilder` structure, we ensure that the program has a main function defined, that no identifier is used before it has been declared, and that no identifier is declared more than once in the same scope. We chose to perform these checks while building the scope symbol tables as they largely overlapped with the checks we were already doing to build the tables. We store any errors that come up in a vector associated with the `IRBuilder`.

In the checking stage, which occurs in the `IRChecker`, we conduct the remaining semantic checks. As in the `IRBuilder`, we traverse our scope-augmented parse tree recursively and save any errors we find to a vector associated with the `IRChecker`. We pass scope information down to the smaller tree blocks (e.g. `Expr`, `Location`, `Identifier`), and use this to perform the symbol table lookups needed to perform type checks.

III. Low-level IR

After we have our high level IR, we then need to turn this structure into something amicable for code generation. This low level IR has two primary tasks: efficiently representing control flow and linearizing all statements. This remainder of this section will describe the construction of our `CFG` object which aims to achieve each of these tasks.

III.I Control Flow

At the top level, our `CFG` aims to represent control flow in a way similar to assembly. We do this by breaking blocks and their statements into multiple `BasicBlocks`. We maintain a HashMap of block IDs to `BasicBlock` objects, and have each block store its predecessor and successor IDs. We do this instead of a direct reference tree structure due to Rust's borrow checker.

Each `BasicBlock` has three forms: `Determinate`, `Conditional`, or `Return`. `Determinate` blocks have consistent control flow, and always point to a single next block. `Determinate` blocks are useful for setup for functions, typically containing variable declarations, but also do have effects on function control flow. For example, we represent break and continue statements using `Determinate` blocks since they have a singular, consistent exit point and do not require extra behavior like return statements.

`Conditional` blocks relate to if, while, and for statements, and lead to conditional control flow within our program. These blocks have two possible next blocks, decided by if the conditional evaluates to true or false. Otherwise, they are nearly identical to `Determinate` blocks. These blocks are also useful for short circuiting, where we can break a complex boolean expression into two or more linked `Conditional` blocks.

`Return` blocks contain only statements and their predecessors, since they are the ceding of control flow for the method. Since blocks know about their containing method, we do not need to store a broader method end id in the block itself.

III.II Linearizing

Every variable in our program is assigned an `TempId` corresponding to it, which is managed by the `TempManager`. In this process, we also build some other necessary tables, specifically mapping each `TempId` to a method offset which the temp manager increments by the size of each variable as we traverse through a method. We also use this structure to track register allocations, which is explained in section VI.

For expressions, we then run a `Linearize` function on them which returns a list of `LinearizedStatements`. At its base case, we allocate a new temporary ID, increment the function offset, and set the temporary to be either the parsed literal value or the value of the identifier's temp it is referencing. Otherwise, we recursively run on the left and right children of the expression, combine the results into a new temp using a single binary operation, and then add this new operation statement to the concatenated vector of the left and right children statements. Since the ID assigned to a temp is determined by a global variable that is incremented for every temp, we will never have clashing temporary variables.

IV. Code Generation

To convert the CFG to assembly code, we start by initializing our global variables and strings in the `.data` section of our program. All global variables are initialized using `.quad` or `.zero` (at the suggestion of the TAs; thank you!), and all non-initialized variables have a default value of 0. Initializing global variables in the data section, instead of in the beginning of the main method, allows us to recursively call `main` without re-initializing our global variables.

We choose to store arrays of length n as n + 1 elements, where the first element of an array points to the size of the array. We use this later on to make bounds checking easier.

After initializing globals and strings, we build our program method by method. For each method, we generate a method label, a procedure prologue and epilogue, and convert the procedure body block-by-block using the CFG. For each block in the CFG, we generate a block label and convert each linearized statement to assembly. For conditional statements, we also add jump instructions to the different branches using the condition at the end of the block. At the last determinate or at a return block, we include a jump to the procedure epilogue.

To convert a block's linearized statements to assembly, we use a function called `linearize_to_assembly`. This function converts each type of linearized statement to a vector of strings representing assembly code. For every operation, we pull the temp ID from the stack, do the operation, and push the result of the computation to a result temp ID slot which has already been allocated on the stack. For array stores and loads, we also insert a bounds check, which checks that `0 <= index < arr.length` for any access of the form `arr[index]`. If either condition is violated, we jump to an error block labeled `array_bounds_error`, which exits the program with error code -1. To convert method calls, in cases where there were 6 or fewer parameters, we simply pushed them to the expected

registers in order. For more than 6 parameters, we added instructions to push the overflow arguments to the stack.

Finally, to handle the runtime check for method returns, we use `%r11` as a “flag” register. We set `%r11` to 1 when a procedure has returned a value. If the method type is not void, the procedure epilogue includes a check to ensure that `%r11` is set to 1 after a procedure call ends, otherwise we jump to an error block labeled `control_fall_off_error`, which exits the program with error code -2.

V. Dataflow Optimizations

Our dataflow optimizations all operate on our basic CFG struct, taking in a CFG as input and mutating it to get a more optimized version. This structure makes both implementing new optimizations and stacking optimizations on top of one another very easy. To add a new optimization into the flow, you just need to create a new struct that takes a CFG as input and modifies it. Since every optimization struct inputs and outputs a CFG, constructing a series of sequential optimizations is simple.

Both CSE and DCE operate on blocks, and each of their associated structs maintains a mapping of block indices to the relevant sets (in + out sets for both, and then use + def sets for DCE).

V.I Dead Code Elimination

Our dead code elimination works for local, non-array variables. We make a few passes over the CFG in order to generate relevant dataflow sets for each block and then eliminate dead code statements. We start by generating dataflow information with `build_data_flow()`. This function constructs use/def sets with `initialize_use_def(block_id)`, then building in/out sets for all blocks using `define_in_out()`. To generate in/out sets, we find an exit block for the CFG and initialize its out set to contain the global variables. We then apply the above dataflow equations to the graph to construct the rest of the sets. After constructing these sets we run `eliminate_dead_code()` to mutate the CFG, eliminating statements from each block that are deemed dead code based on the generated use, def, in, and out sets. We repeat the process of running `build_data_flow()` and then `eliminate_dead_code()` until we don’t find any more dead code statements. The final CFG can then be pulled out from the DCE struct and passed into future optimizations or the final codegen struct.

V.II Common Subexpression Elimination

We implemented CSE for simple BinStatement and CmpStatement which have a op b where a and b are local non array variables and op is a BinOp or CmpOp. Notably more complex statements like !a && b are not eliminated because they are broken down into two statements: a UnaryStatement and a BinStatement.

To do this, we split our CFG such that it had one statement per block and made in and out hashmaps that mapped linearized expressions to their result variable for each block, `bb_to_in_hash` and `bb_to_out_hash`. For each method we used BFS to go through the blocks of the cfg to populate the in and out hashmaps following the algorithm described in lecture. If we found a Cmp or an Bin LinearizedStatement we would add the expression to our hashmaps and if we found a redefinition, a Store of a LoadImm, we deleted all expressions that relied on the old value of the variable.

Then, we iterated through the blocks one more to actually do the elimination, following the logic of if we found a BinStatement or CmpStatement whose expression was available at that point, we replaced it with a Load Statement of the result variable.

V.III Copy Propagation

We implement copy propagation in `src/code_gen/cp_builder.rs` The structure is very similar to `cse_builder.rs`. When we build the in and out hash for each basic block (with only one statement), our operations are very simple. For any `LinearizedStatement` we remove whatever the result/destination temp is for the operation from our set because its value is being overwritten, and then for `Loads` and `Stores` which propagate a value, we then insert this into our out hash set. When actually doing the elimination, for every non-result temp in the `LinearizedStatement` we switch the temp with its corresponding copy propagation temp if possible. Note that our hashes map temps to temps they should be replaced with. So `a = b` is inserted as `a` maps `b`, and then if `a` is ever seen later, it is swapped to `b`. We do this recursively though, so when we do the propagation, we swap `a` with `b` and then also check if we can swap `b` with anything. Lastly, we continue to do BFS, setting the in and out hashes and then doing the elimination, until we see no change in the CFG.

Due to the amount of time it took copy propagation to run (our tests timed out), we did not include it in our final optimizations, nor have time to properly merge it in, so this code exists on branch `dataflow-cleanup`.

V.IV Constant Elimination

We also implement constant elimination, with the same structure as `cp_builder.rs`. Our conversion from the in hash to the out hash similarly removes any result temps from the hashset, and only inserts when traversing over a `LinearizedStatement::LoadImm`, if `z = 4`, then `z` maps to `4` in the hashset. Our `do_elimination` substitutes constants in `BinStatement`, `ArrayLoad`, and `ArrayStore`. This is because these make up most of our use cases, and for `CmpStatement`, we ran into issues generating assembly for this as we could have something like `cmpq $4, $7`. This is an illegal instruction, but due to time constraints we couldn't resolve this.

One prominent issue we ran into during implementation was that we had no current infrastructure to replace a temp with a constant. To work around this, we took the path of least resistance and simply added another field to temp which was `is_imm`. If this boolean was true, then the temp ID would be interpreted as a constant value, not an ID. This forced us to rework our code generation in `two_addres.rs`, which originally just loaded each temp into %rax. To make our code run faster as well as to further simplify our `BinStatements`, we added a `do_math()` function to constant_elimination. This function iterates through statements and reduces `BinStatement` of two immediates to the result of that operation, since we can do that math at compile time, and replaced it with a Store. We also added the function `write_asm_with_imm(..)` , which consolidated the logic of checking whether a temp is an immediate or not. This resulted in pretty significant reworkings of our assembly generation.

The Minus, Div, and Mod operations were more difficult to get working, since order of operands mattered, and the compare operation for array bounds checking also had to be reworked to accommodate immediates. These changes were made alongside the copy propagation changes, and we were not able to be merged in because they took too long to run in Gradescope.

VI. Register Allocation

The main optimization we implemented for phase 5 was register allocation, which provided the most significant performance improvement of any one optimization we tried. The bulk of the code for this portion lives in `src/optimizations/register_allocation.rs` and `src/optimizations/reaching_defs.rs`.

VI.I. Building webs

To build the webs, we used information from both reaching definitions and liveness analysis conducted during our dataflow optimizations. Using reaching and liveness information, we constructed live ranges for each temp variable in the `get_live_range()` function. Since we split the CFG into single statement blocks before doing this analysis, we can associated each defined temp variable with the block it is defined in, which we use to combine the reaching definition information with the liveness information. After we calculate all live ranges, we then iterate through them and merge any live ranges whose def reach overlapping uses, creating `Web` objects that we use for the rest of our register allocation.

VI.II Building the interference graph

After building webs, we iterate through the webs and check for conflicts between them. We maintain a data structure called `web_conflicts` that stores a mapping between each web and every other web it has a conflict with. An issue that we are aware of with this is that statements of the form `a = b` get converted to `def a; use b`, rather than `use b; def a`. If `b` is never used again after `a`, we should be able to allocate them both to the same register, but our scheme will not do this. We believe that this would result in further performance benefits.

VI.III Coloring the interference graph

We use **four callee-saved registers**, `%r12`, `%r13`, `%r14`, and `%r15`. We compute the coloring in `do_coloring()`, which loops through registers, popping registers with fewer than 4 edges to a stack, and spilling once we cannot pop anymore. We end up with a HashMap `web_to_register`, which maps webs to their associated register or the string “spill”.

VI.IV Heuristics

Before coloring the interference graph, we sorted the webs by a heuristic in order to spill things with the lowest spill cost first. We tried a few different heuristics: live ranges, use counts, and loop appearances. Surprisingly, we saw the best performance from live ranges, despite it being the simplest of the heuristics, although it is possible that this is due to implementation errors in the other two.

VI.IV.I Live Ranges

For the live ranges heuristic, we spill webs that are live for the fewest number of blocks first. This is to approximate the duration of time a value is used, although it’s not very accurate since it doesn’t take into account the number of uses, or if the blocks that a value is present in belong to a loop.

VI.IV.II Use Counts

For the use counts heuristic, we spill webs with the fewest number of uses first. We calculate uses using the liveness dataflow analysis info, which stores a use set for each block. Surprisingly, this performed slightly worse than live ranges.

VI.IV.III Loop count

Our most complicated attempt at a heuristic was loop count, the code for which can be found in `src/optimizations/loop_optim.rs`. For this, we wanted to spill values that were not present in loops (or present in the fewest number of loops) first. In order to do this we had to identify loops first, then compare each web’s live range to the list of identified loops. To identify loops, we begin by computing the dominator tree with the `dominator()` function. After constructing the tree, we use `identify_loops()` to discover back edges, loop headers, and associated loop bodies. The output of this is a list of `Loop` objects, which each store a header, back edge, and set of blocks in the loop. This also did not perform as well as live ranges, to our surprise, although given more time we could have experimented more with how we convert the loop appearances to a numerical spill cost.

VI.V Generating code using the registers

After generating assignments of webs to registers, we had to modify the rest of our code to be able to handle the new generation possibilities. The main changes occurred in our TempManager (which lives in `src/codegen/cfg_builder.rs`), which was modified to map

temps to stack offsets or registers at different points in the code (given that now, a temp could be mapped to different things depending on where we are accessing it).

In `src/code_gen/gen.rs`, we also had to modify our function prologue and epilogue code to push and pop the callee-saved registers from the stack before and after the function. Right now, we push/pop all four registers at the beginning and end of every function. For further performance optimization, we could analyze each function body and only push/pop registers that are used within that function.

VII. All Optimizations

Our `-all` flag includes `dce`, `cse`, and `regalloc`. We found that `dce` and `cse` generally did not improve performance that much, but in some cases they produced small improvements, which is why we chose to include them in our final `-all`. We ran `cse`, then `dce`, then `regalloc`, since CSE generates extra temps that DCE can clean up, but register allocation does not result in extra code generation that DCE would need to clean up. Deeper explanation of the individual optimization implementations can be found above, but generally, since our framework takes in a CFG and outputs a modified CFG with preserved correctness, running the optimizations sequentially preserves correctness.

You can find examples of our optimizations in the `doc` folder at the root of our repository.

Although copy prop and constant elimination did not make it into our final compiler, we also spent a lot of time looking at the order we wanted to run these optimizations in. We did two loops of copy prop, CSE, DCE, then constant elim, before doing register allocation. We found that this was necessary for preserving correctness of the outputted code, and ensuring that we didn't actually introduce performance hits when trying to optimize. The code can be found in `main.rs` the `dataflow-cleanup` branch.

In addition to implementing these optimizations, we did some peephole optimizations and jump improvements as a part of this phase of the project. For the peephole optimizations, we modified generation code for addition, subtraction, and multiplication, to remove 1 extraneous instruction when a register is involved in the computation. For jump optimization, we found that we often had unnecessary jumps between sequential blocks due to splitting our CFG for register allocation. We couldn't merge the CFG after doing register allocation, since we tied registers to a temp and its associated block, so instead we did a pass through the generated assembly and got rid of any instructions of the form `` `jmp {name}_block_{id}` `` directly followed by `` `{name}_block_{id}` ``. We saw noticeable improvement from this optimization: when we tested noise_median.dcf on our machine with this optimization included in `-all`, it went from taking around 46 ms to run to 41 ms.

Here are some general statistics from our `derby_bench` script (explained more in the Extras section) that demonstrate the effectiveness of our optimizations. It is clear that register allocation is doing most of the heavy lifting here when it comes to improving performance.

	-all	-none	-regalloc	-dce	-cse
noise_median.dcf	**41.2 ms ± 0.5 ms**	59.8 ms ± 10.9 ms	41.4 ms ± 8.1 ms	57.3 ms ± 8.0 ms	61.2 ms ± 12.9 ms
segovia_blur.dcf	170.0 ms ± 16.9 ms	186.0 ms ± 16.8 ms	**168.0 ms ± 16.0 ms**	185.7 ms ± 16.8 ms	184.9 ms ± 17.1 ms
saman_sl.dcf	**28.4 ms ± 0.9 ms**	35.4 ms ± 10.8 ms	32.2 ms ± 7.2 ms	31.8 ms ± 11.0 ms	32.5 ms ± 8.9 ms

II. Extras

For phase 5, we added a few things to our test suite. First off, we added `cargo test all_opt` and `cargo test derby`, the first of which runs tests on all of our codegen inputs with whatever the `-all` flag is currently set to, and the second of which runs the three public derby tests. For the derby tests, the test case structure is slightly different to accommodate the linking of libraries, but otherwise it's the same.

```
test tests::test_all_opt_legal_files_tests_codegen_input_x_10_many_args_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_25_index_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_23_nested_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_26_math3_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_28_sideeffectparams_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_29_stack_arrays_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_31_scope_shadowing_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_32_assign_order_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_30_short_circ_methods_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_34_chained_if_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_33_binop_order_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_35_const_assignments_dcf ... ok
test tests::test_derby_files_tests_derby_input_saman_sl_dcf ... ok
test tests::test_all_opt_legal_files_tests_codegen_input_x_33_initializers_dcf ... ok
test tests::test_derby_files_tests_derby_input_segovia_blur_dcf ... ok
test tests::test_derby_files_tests_derby_input_noise_median_dcf ... ok

test result: ok. 1018 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 144.73s
```

✨last test case screenshot for this class, 1018 test cases passing ✨

We also added a quality-of-life script, `./compile.sh`, which takes in an input file (-i), a list of optimizations (-O), and an output file (-o). It will then build our rust code, run our compiler and generate an assembly file, use gcc to compile the assembly, and run the executable, displaying the output to the screen. We should've done this earlier; it was nice to not have to remember the gcc command.

Finally, we created a benchmarking script, `./derby_bench.sh`, using `hyperfine`. `derby_bench.sh` takes in a derby input file (-i), optional number of runs (-r), and optional output file name (-o). The script will use hyperfine to benchmark the derby file with different optimizations enabled ("all", "dce", "cse", "regalloc", and "none"), and print the output to the screen.

```
~/desktop/61100/lulew-mrebholz-hkimura-lmwilson git:(main)±31 (1m 1.71s)
./derby_bench.sh -i ./tests/derby/input/noise_median.dcf
Benchmark 1: ./out_all.o
  Time (mean ± σ):      40.4 ms ±   1.2 ms    [User: 36.0 ms, System: 2.3 ms]
  Range (min … max):    39.5 ms …  48.9 ms    63 runs


Benchmark 1: ./out_dce.o
  Time (mean ± σ):      56.2 ms ±   1.5 ms    [User: 50.8 ms, System: 2.5 ms]
  Range (min … max):    54.7 ms …  62.2 ms    46 runs

Benchmark 1: ./out_cse.o
  Time (mean ± σ):      57.8 ms ±   6.1 ms    [User: 51.7 ms, System: 2.6 ms]
  Range (min … max):    55.4 ms …  97.8 ms    47 runs


Benchmark 1: ./out_regalloc.o
  Time (mean ± σ):      46.1 ms ±  16.2 ms    [User: 37.3 ms, System: 3.3 ms]
  Range (min … max):    40.7 ms … 146.5 ms    62 runs


Benchmark 1: ./out_none.o
  Time (mean ± σ):      58.2 ms ±   8.4 ms    [User: 51.9 ms, System: 2.8 ms]
  Range (min … max):    55.7 ms … 113.9 ms    46 runs
```

Example of derby_bench script running 🏃‍♂️💨

As in Phase 4, our code still uses `is_mac` in `src/code_gen/gen.rs` to generate both normal x86 assembly and Mac-friendly x86 assembly. While this made debugging sometimes a nightmare, it's still cool that our compiler can do both!

III. Difficulties

Fixing test cases

From phase 4, both our CSE and DCE implementations did not interact with global variables correctly. We were failing one test case, `cse_07`, for the phase 4 deadline, but we found that our DCE implementation was also nondeterministically failing one DCE test case. We identified that both of these issues were with handling of globals by each dataflow optimization, and we corrected these by modifying the in/out set logic for globals.

Testing Problems

Our benchmarking suite provided comprehensive information regarding our efficiency and performance, but we had certain edge cases for our correctness which our prior testing

procedures failed to pick up on. Specifically, our test suite compared the files running each of the derby tests created, but it did not necessarily check if the derby tests ran to completion and actually generated those files. This became an issue as originally, our test suite did not delete the output files after completion, so if we had a round of successful tests but then created erroneous changes to our code, it would not always pick them up. This detection failure was not noticeable at first, and led to situations where particular branches had hours of code written which were built on a faulty foundation.

Extra Registers Problems

We spent substantial time trying to increase the amount of registers we could allocate past four, especially since `%r10` did not have any other uses. However, this ended up posing substantial issues we are still trying to get a grapple on. Specifically, when adding an additional register, certain test cases would begin to fail. These test cases corresponded to arrays, and had to do with our register allocator incorrectly allocating array uses to registers. Yet, when we resolved this, we found an error in one of the derby test cases, `segovia_blur`. In the end, we were not able to find a means of including a 5th register without causing issues with the derby test cases (compounded by difficulties in testing which gave us some unfortunate red herrings). We think the issue may still lie in how we are processing arrays using register allocation, but we lacked the time to thoroughly resolve the issue.

Debugging optimizations, unused optimizations

We did a lot of work that didn't make it into the final compiler, most notably copy propagation, constant elimination, and loop detection. Part of this was due to issues with debugging these optimizations (or seeing surprisingly negligible performance benefits from them), and part of it was due to issues with merging together diverging versions of the compiler. Given more time, we could have definitely put everything together a bit better.

IV. Contribution

- Register Allocation: Hannah, Luca, Lili
- Copy propagation: Maya
- Constant elimination: Maya
- Fixing dataflow optimization from phase 4: Maya
- Loop optimization: Lili
- Testing and benchmarking: Lili

Thank you to the TAs for a great semester and all of the help in office hours!