IIT CS443: Compiler Construction

Project 5: Dataflow Analysis and Optimization

Prof. Stefan Muller

Out: Thursday, Oct. 31
Due: Tuesday, Nov. 12, 11:59pm CDT

Total: 45 points (Programming: 35 points, Test cases: 10 points)

Logistics

Submission Instructions

Please read and follow these instructions carefully.

- The starter code for Project 5 has been distributed through a pull request to your Project 2 GitHub repo. Merge this pull request into your main branch to start working. (This shouldn't cause merge conflicts, but if it does and you aren't sure how to handle them, ask.)
- When you want to submit, commit the latest changes to your GitHub repo **with a commit message that clearly indicates this is your Project 5 submission** (e.g., "Project 5 Submission" would be a good commit message). This is so I know which commit to grade. If you resubmit later, just use a similar commit message. I'll grade the last commit that clearly indicates it's a Project 5 submission (and count your late days based on the time stamp of this submission).
- Compile (by running `make`) before submitting. **Submissions that don't compile will not get credit.**

Collaboration and Academic Honesty

You may work in groups of at most 2 on this project. Read the policy on the website and be sure you understand it.

A Quick Note on Grading

Grading for the programming tasks in this project will be primarily based on safety, i.e., whether your optimizations preserve the original behavior of the program. You can still get full credit even if your implementation misses some opportunities for optimization, as long as it is a reasonably complete attempt at the intended optimization (e.g., if you leave the compiler as is so that it does not alter the behavior but also does not optimize the program, you will not get credit :-).)

1 Programming Tasks: Optimizations

You will complete the programming tasks of the project in `opt.ml`. You will need to add your own functions in this file to perform various optimizations, and also edit one or both of the following functions:

`opt_body : typ LLVM.Typecheck.LLVarmap.t -> string -> inst list -> inst list`
and
`opt: typ LLVM.Typecheck.LLVarmap.t -> prog -> prog`

Currently, `opt_body` just returns the body of a function unchanged, and `opt` calls `opt_body` on the body of each function, rebuilding the (currently unchanged) program from the function bodies returned by `opt_body`. You'll be changing these functions to actually optimize the program! (**Note:** All of the optimizations mentioned in this project operate only on function bodies, and therefore only require editing `opt_body`; you'll only (maybe) need to edit `opt` if you add your own other optimizations.)

The arguments to `opt_body` are as follows:

- `ts`: LLVM type information. This is useful for calling some of the convenience functions I provide.
- `fname`: The name of the function being optimized.
- `body`: The body of the function as a list of LLVM instructions.

You can assume that the LLVM code passed to your functions is in SSA format.

1.1 Task 1: Common Subexpression Elimination (20 points)

Your first task is to implement common subexpression elimination (CSE). Recall that CSE requires performing a dataflow analysis to determine the available expressions at every point in the program. Lucky for you, I've given you code that performs a generic iterative dataflow analysis. You just have to instantiate it with the right gen and kill functions for available expressions.

Some notes on how to use the dataflow analysis framework:

- The dataflow analysis is implemented in the module `ExpDataflow`[1]. The main function in this module is `ExpDataflow.compute`. It takes the following arguments:
  - `cfg : DFG.t` : A dataflow graph. See below for how to compute this.
  - `gen: inst -> ExpSet.t` : The definition of *gen*, as a function that returns a set of expressions genned by the given expression.
  - `kill: inst -> ExpSet.t -> ExpSet.t` : The definition of `kill`, as a function that takes an instruction and a set of expressions and returns the set of expressions *remaining* after killing expressions killed by the given expression.
  - `fwd: bool`. If this is a forward (as opposed to backward) dataflow analysis.
  - `must: bool`. If this is a "must" (as opposed to "may") analysis.

  and returns a pair `(in, out)`, where both components are of type `ExpSet.t NodeMap.t`.

- `ExpSet.t` is a set of LLVM instructions. It is an instantiation of OCaml's `Set` module. See `https://v2.ocaml.org/api/Set.S.html` for the API of this module (where `elt` is instantiated here to `inst`.)
- `NodeMap.t` is a map from DFG nodes to `ExpSet.t`s. It is an instantiation of OCaml's `Map` module. See `https://v2.ocaml.org/api/Map.S.html` for the API of this module (where `key` is instantiated here to `DFG.G.node`, the type of CFG nodes.)
- The module `DFG` has all of the facilities for working with dataflow graphs. The function you'll need is

  `DFG.cfg_of_insts : string -> inst list -> (DFG.t * DFG.G.node) * DFG.node list`

  which builds a dataflow graph from a function name and body. It returns a nested pair ((Dataflow graph, entry node), list of all nodes). The nodes of the dataflow graph correspond one-to-one to the provided LLVM instructions, and the list of all nodes (the last component of the returned pair) gives the nodes in the *same order* as the list of LLVM instructions passed to `cfg_of_insts`. This is useful

[1] If you're curious and not scared off by OCaml functor syntax, this instantiates the functor in `dataflow.ml` by setting the sets of "facts" to lists of LLVM instructions. You're welcome to reuse this functor in other ways for other optimizations in Task 2 if you want, but you don't have to.

since, after doing dataflow analysis, you can pair the `body` together with this list to do something for each instruction in the function. For example, if you call `cfg_of_insts` on `body` and `nodes` is the list of all nodes you get back, you can use

```
List.map2 (fun inst node ->
               let available_in = ExpDataflow.NodeMap.find node in\_fs
               in ...)
           body
           nodes
```

where `in_fs` is returned by `ExpDataflow.compute`, to map over the body of the function and do something (the ... above) with the "in" set for each instruction.

- You may find the `def_inst` and `use_inst` functions in `llvm/llvm_utils.ml` useful.
- There are a bunch of functions and definitions at the top of `opt.ml`. **Some of these are used in my solution.** Look through these and see if any of them might be useful before you start coding.

When you're done, don't forget to modify `opt_body` to call your code so it actually performs CSE on the program!

1.2 Task 2: Choose your own adventure (15 points)

Implement one of the following optimizations:

- Copy/constant propagation (you should propagate both constants and copies)
- Constant folding (you should perform arithmetic and logical operations, comparisons and constant jumps)
- Dead code elimination

If you want to implement a different optimization of similar complexity instead, contact me for approval first. You're welcome to implement additional optimizations if you wish, as long as you implement one of the above (or a different one with permission) and your additional optimizations don't interfere with its correctness. When you're done, don't forget to modify `opt_body` and/or `opt` to call your code so it actually performs the optimization on the program!

Some hints/additional information (ignore these at your peril!)

- There are a bunch of functions and definitions at the top of `opt.ml`. **Each of these is used in my solution for one or more of the optimizations above** (or for CSE). Look through these and see if any of them might be useful before you start coding.
- You may also find the `def_inst` and `use_inst` functions in `llvm/llvm_utils.ml` useful.
- Pay careful attention to the lecture notes on your chosen optimization, particularly the implementation details and caveats about safety.
- You may find you want maps from LLVM variables to other things. The `VMap` module is provided for this. Like `NodeMap`, it uses the OCaml Map API (see link in the previous task). You can use this to make your own maps, e.g. a `int VMap.t` maps LLVM variables to integers, and a `value VMap.t` maps LLVM variables to values.
- If you're putting an LLVM function (type `func`) together (not just returning a body as a list of instructions), make sure to call `make_func` (defined in `llvm/llvm_ast.ml`) rather than building a `func` record directly. This does some preprocessing on the function to maintain some invariants.

- The function signatures of functions you add to perform optimizations is up to you, but you may find it helpful to make your optimization functions have type

  ```
  typ LLVM.Typecheck.LLVarmap.t -> string -> inst list -> inst list
  ```

  (which is bound in the code as `func_optimizer`) for reasons discussed below. This matches the type of `opt_body`.

- If you're doing copy propagation or dead code elimination (or some combination of these and constant folding), you may want to call the optimization(s) in a loop since doing it once can enable more optimizations. I've provided the function `iterate` which takes an `int option`, a list of optimization functions of type `func_optimizer` (see the previous bullet point), and the same arguments `ts`, `fname`, and `body`, that are taken by `opt_body` (and by your optimizers if you chose to follow the advice in the previous point). The function applies the provided optimizations in a loop (in the order they're provided in the list, then repeating) until either the code stops getting shorter (if `cond` is `None`) or for `n` iterations (if `cond` is `Some n`).

2 Task 3: Two(!) test cases (10 points)

Each group should pick one of the optimizations they implemented (CSE or their chosen optimization in Task 2) and write at least two test cases:

- At least one test case that allows the optimization to occur.
- At least one test case that tests a corner case related to safety. This could be a test case in which the optimization *should not* be applied, or one where it must be applied carefully. (The exact meaning of this will depend on which optimization is targeted.)
- As in previous assignments, the test cases shouldn't be substantially the same as test cases provided by the instructor or other groups, and should be reasonable (but can be adversarial within reason).

Upload your test cases on the "Project 5 Test Cases" discussion board on Canvas. You can (and should!) test your compiler on other students' test cases if relevant to the optimizations you implemented. I may do so as well during grading. Feel free to ask clarification questions, note issues, etc., as replies in the threads created by other students.

Your test cases may be LLVM IR or MiniC, but must be accepted by the appropriate parser and type checker in `main`.

Note: There are, at least initially, no instructor-supplied test cases. As such, the `proj5_tests` folder and `p5tests` file are empty, and running `make test` won't do anything useful unless you write your own tests and add them to `p5tests`. Depending on how the test cases discussion is going, I may chime in with my own at some point. You can always test with the Project 2, 3, and 4 test cases if you enable the compiler to use MiniIITRAN, MiniC, and MiniML source code as inputs (see next section).

3 Testing

Compile your code using `make` (in the top level of the source tree). This will produce the binary `main`, which you can use as follows to compile test programs:

```
./main -interpllvm <path_to_test_case>
```

Assuming you've left your MiniIITRAN, MiniC, and MiniML compilers in the respective files, your compiler will also be able to take MiniIITRAN, MiniC, and MiniML source files as input. In any case, this will parse and type check the file, compile it to LLVM IR (if it isn't already), and run `opt` on the LLVM code. By default, it will output human-readable LLVM IR to `<same_path_as_input>/<file>.ll`.

Important: if using an LLVM IR test case, you should use the `-o <output_file>` flag to write the output to a different file.

Other command line options that might be helpful:

- `-cost`: If you use this along with `-interpllvm`, the interpreter will track the execution "time" of the LLVM code (in arbitrary units, assigning fixed costs to each opcode). You can use this to see what effect your optimizations are having.
- `-O0` (that's an uppercase letter O and a digit zero): Turns off optimizations (you can also use `-O1` to turn on the optimizations, but this is the default).

Note: You should **not** use the `-nossa` command line option, as your optimizations likely require SSA format for safety.