Tsz Hang Kiang, Colten Webb, Wooseok Kim

CS 4240 SP 22

Dr. Qirun Zhang

February 8, 2022

Project 1 Optimizer Design Document

1. High-Level Optimizer Architecture

Our optimizer is composed of three major components: parser, optimizer, and pretty printer. First, the parser parses raw instructions into appropriate data type, encapsulated in Function type. Then, the optimizer performs dead code elimination with reaching definitions by generating a control flow graph and identifying the Gen, Kill, In, and Out sets. We decided to use maximal blocks in our CFG over single-instruction basic blocks for performance reasons. Finally, the optimized function is pretty printed into a valid optimized IR file.

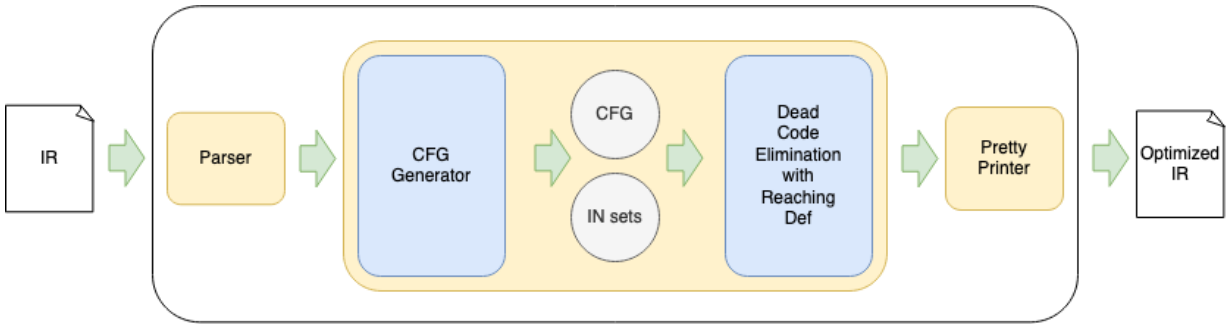

Figure 1. Optimizer architecture composed of parser, optimizer, and pretty printer

2. Low-level design decisions

We implemented our optimizer in Haskell for two reasons, for fun, and because functional programming languages are known to be good for compiler design work. Specifically, this is because compilers have minimal IO: there is a single well-defined input at the beginning of process, and a single output. That means we can use clean, declarative functions to describe the transformations we make to the IR, without the side-effects that plague imperative PLs. Haskell also has a powerful type system that makes our code modular and robust.

3. Software engineering challenges

We had to write our own parser since we decided to use our own language Haskell. Although Haskell has strong support for writing parsers, the hard part was ensuring equivalence of the provided reference parser, and our own. For example, our own parser was initially unable to handle CRLF-style line endings.

We randomly generated a lot of IR code and fed it through our parser, which helped us catch multiple bugs in our parser, and gave us a greater peace of mind.

Another issue was the correctness of the CFG generation algorithm in the lecture slides. There were many difficult edge cases originally left out that were thankfully spotted by other students on Piazza. As a result, we as a team dedicated a lot more time scrutinizing our CFG generation algorithm in comparison to other parts of our code base.

4. Known Outstanding Bugs / Deficiencies

No known bugs were discovered before project submission. If we had more time, we would have wanted to implement other optimizations.

5.Installation / Build steps

1. Make sure GHC and cabal are installed, easiest way is via **`ghcup`**. See https://www.haskell.org/ghcup/
2. To build the program, **`cd`** into the project directory (the one containing the file `cabal.project`), and then run

   ```
   $ cabal new-build project1-app
   ```

3. To run the optimizer, which will take in the path to an IR file to optimize as an argument, and print to stdout the resulting optimized IR, run

   ```
   $ cabal new-run project1-app -- $IR_FILE_TO_OPTIMIZE
   ```

6. Results on Public Test Cases

Testing our optimizer on quicksort and sqrt, we found that our performance matches the baseline deadcode elimination with reaching defs implementation. The exact dynamic instruction counts are included below.

Outputs on **quicksort.ir**:

Outputs match, DIC 482 -> 434

Outputs match, DIC 1124 -> 1026

Outputs match, DIC 1806 -> 1658

Outputs match, DIC 2520 -> 2322

Outputs match, DIC 3126 -> 2878

Outputs match, DIC 4180 -> 3882

Outputs match, DIC 4654 -> 4306

Outputs match, DIC 5544 -> 5146

Outputs match, DIC 6274 -> 5826

Outputs match, DIC 7012 -> 6514

Outputs on **sqrt.ir**:

Outputs match, DIC 94 -> 83

Outputs match, DIC 70 -> 61

Outputs match, DIC 16 -> 11

Outputs match, DIC 101 -> 89

Outputs match, DIC 133 -> 119

Outputs match, DIC 155 -> 139

Outputs match, DIC 168 -> 151

Outputs match, DIC 272 -> 247

Outputs match, DIC 155 -> 139

Outputs match, DIC 16 -> 11