CS 4240 - Project 1
Design Document

Matthew Sklar & Ohad Rau

17 February, 2020

1 Architecture

The compiler is made up of a number of stages, which come together to implement dead-code elimination. At a high level, the stages are as follows:

1. Lex/tokenize the input code
2. Parse the input code into a list of typed function, with each function containing a stream of instructions known as the "body"
3. Create a control-flow graph (CFG) for each function, where each instruction is represented by one vertex
4. Copy Propagation
   (a) Initialize the "in" set of the first node to $\emptyset$ and the "in" set for all other nodes to the universal set and initialize the "gen" and "kill" sets for each node in the CFG as defined by the copy propagation dataflow equations
   (b) Recursively update the "in" and "out" sets of each node in the CFG until you find a fixpoint where the previous and current iterations of the CFG's dataflow equations converge
   (c) Apply the copy propagation algorithm to each node in the CFG. This algorithm checks to see if any operands in the node can be replaced with a copied variable, and if so updates the instruction
5. Dead Code Elimination
   (a) Using "def" and "use" functions, initialize the "gen" and "kill" sets for each node in the CFG as defined by the dead code elimination dataflow equations
   (b) Find a fixpoint for the dataflow equations on the CFG, recursively updating the "in" and "out" sets until they are solved

(c) Using reaching definitions (to determine dependencies in the CFG) and a set of basic critical operations, apply the worklist algorithm for deadcode elimination

(d) Remove all vertices from the graph that were not reached by the algorithm, replacing all the now-deleted edges with an equivalent edge

6. Repeat copy propagation and dead code elimination until the outputted code converges

7. For each function, convert the CFG back into a stream of instructions

8. Pretty-print the final syntax tree into a new .ir file

9. Optionally, print the before/after CFGs into Graphviz DOT files to allow the user to inspect the optimizations applied

2 Implementation Decisions

We chose to implement our compiler in OCaml. The reasoning for this was largely based on OCaml's features that assist in compiler design. Specifically, we find that pattern matching, algebraic data types, and strong static type systems greatly aid in the development of a compiler. Further, we find the existing set of compiler tools written in OCaml to be a useful reference to work with.

Specifically, a number of OCaml-specific tools greatly assisted in the design of the compiler. The built-in `ocamllex` tool greatly simplifies the work required to tokenize the Tiger IR code and the `Menhir` add-on automates much of the parsing phase. Using EBNF-like syntax, we were able to design a very declarative front-end. Note that while simply scanning through the token stream was a simpler solution, we chose to implement a full lexer/parser front-end in order to offer a better structure for pattern matching in later stages and support future extensions that may inspect function type declarations or other meta-data in the Tiger IR code.

In addition, the `OCamlgraph` library served as a great basis for many of the operations we implemented on the CFG. Specifically, it gave us the basic tools needed to build the graph, perform graph traversals, etc. but it also provided modules for rendering graphs via Graphviz, creating dominator trees from arbitrary graphs, and searching for fixpoints on a graph. Both of these prove immensely useful for designing the middle-end of a compiler and made `OCamlgraph` a solid choice to work with.

Unlike what compilers typically do, we choose not to create maximal basic blocks. Instead, each node in the CFG represents a single instruction. While this inflates the graph to be much larger, it's conceptually very simple to reason about the graph operations. Specifically, we can more easily reason about the reaching definitions for an individual instructions because they map 1:1 with

vertices on the graph. However, this also complicates some parts of the program. When deleting from the CFG, we have to delete entire vertices rather than just an array cell. This complicates deletion, since we have to reconstruct the lost edges in a logical way.

3 Challenges Encountered

1. CFG → code algorithm is much harder than code → CFG algorithm:

   Converting each node to a vertex is very simple and only requires us to inspect the branch targets and next instruction to create edges. However, it's quite difficult to perform the reverse transformation.

   In order to create linear code from a CFG we need to utilize a depth-first search that prioritizes fallthrough edges over branches. This is important, because if a fallthrough exists for any instruction it must be placed immediately following the instruction, but the location of the branch code is unimportant. This becomes a difficult operation when we reach code that has an outgoing branch to a vertex that hasn't been placed yet, as it would normally try to place that code in the buffer immediately. Instead, we check to see whether that vertex has an incoming fallthrough edge and if so we yield (to allow the fallthrough edge to decide the placement of that vertex).

   This is further complicated by the fact that there's no inherent "first" instruction in a CFG (for example, the last instruction could jump back to the first instruction, so a topological sort wouldn't work). At the same time, we need to keep the first instruction at the top to preserve the original semantics. A simple solution is assigning a number to each vertex representing its position in the original function. We then pick the lowest-numbered vertex in the graph to be the first instruction. It's important to note that it's not always instruction #0 in the case where the first instruction is dead, so we have to actually iterate through the remaining vertices to find this min-vertex.

2. Limited support for custom DFS's and merges in `OCamlgraph`:

   When working on the compiler, we had to utilize depth-first searches a few times. This specifically came up when turning a CFG back into linear code and when solving data flow equations. While the `OCamlgraph` library included support for DFS, we quickly found that it was not expressive enough for us to use. Firstly, the provided DFS algorithm had no notion of a starting vertex, and would simply pick an arbitrary vertex to start the search from. This was unable to describe the semantics of our linearization algorithm, since it depended on starting at the initial vertex to preserve the semantics of the code.

   When defining the algorithm for deleting vertices (and experimenting with merging single-instruction blocks to true basic blocks), we wanted to use

the included Merge module to simplify this transformation. The module provided a function to combine two vertices, such that they would become a single vertex with all of the original edges retained. This was very appealing, but unfortunately did not give us a way to define the transformation on vertex labels. Specifically, there was no way to specify which label was kept or to specify that labels should be merged to create a longer code block. As a result, this was useless for our code and we had to reimplement our own merging functionality.

The challenge with implementing this from scratch is that when we merge or delete a vertex, all references to that vertex are now unresolvable, since it's been replaced by a wholly new vertex. To perform multiple merges/deletes safely, we had to create a mapping of each old vertex to its replacement. If two nodes are merged, we map them both to the same replacement vertex. Then we iterate through all the edges that are to be deleted and copy them to the replacement vertices. This effectively rebuilds the original graph's topology using the new merged nodes.

3. Fixpoint convergence (how to check for convergence?):

   While this was a relatively easy problem on paper, we encountered a few bugs when testing our compiler. Specifically, we noticed that we only ever had a single iteration and that the converged function would always seemingly return false. This was quickly pinpointed to us not properly comparing every set for every vertex, since we assumed that the `=` operator in OCaml had the correct equality implemented. We then realized that we had flipped the two branches in our fixpoint function, iterating only when the sets had *not* converged. However, this change led to an infinite loop. After some further digging, we realized that while we had copied the old set to a new set before modifying it, we had accidentally given the next iteration the old set. This meant that on every iteration we would do the same work again, since we didn't have access to the latest information.

   All of these were relatively quick fixes and by abstracting this behavior into a higher-order function we were able to provide a fixpoint primitive for use in other places in the compiler. Specifically, this single implementation is able to power the core of dead code elimination, copy propogation, and the repeated iteration of these two optimizations to find a global maximum.

4. Floating point parsing:

   Another simple challenge we encountered was correctly parsing floating point numbers. When the compiler was first in a working state, we ran the provided test cases to verify that everything worked as intended. While we passed all the tests for quicksort, some of the tests for sqrt failed. Curiously, this only happened for numbers that were not perfect squares and the results seemed like they were off by tiny floating point precision errors. After inspecting the before/after code side-by-side, we quickly realized that the number `0.000001` had become `0.1`. The reasoning for this became clear when inspecting the parser:

In an effort to simplify parsing, we had used integer tokens to represent each number. Then, the parser would look for the pattern `INT DOT INT` and convert that to the equivalent floating point number. While the code correctly realized that the number of zeroes on the right-hand side were significant (which meant that we needed to parse the number as a string), we had accidentally already converted the `INT` tokens into actual integers. This resulted in us losing the significant zeroes between the dot and the parsed integer. The quick fix here was to simply store each `INT` token as a string and then parse it later. Once this was resolved, the sqrt test cases all passed as expected.

5. Set comparison in copy-propagation

   When implementing copy-propagation, we struggled to pass the sqrt test. After lots of debugging and hand-optimization, we realized it came down to a sequence of instructions where two identical defs would occur without being deadcode. Specifically, we had two definitions of `a = b` in the graph. When solving the dataflow equations, we initially chose to use the line number to decide equality. However, this led to a bug where the intersection of {`1.  a = b`} and {`2.  a = b`} was resolved to $\emptyset$. This was obviously incorrect, as either def would allow us to perform the same optimization. The fix we eventually came upon was simply ignoring the line number when comparing these defs.

4 Outstanding Bugs

At this time, there are no known bugs in the compiler. However, there are some places where we're less confident with our implementation. For example, our parser's implementation is substantially different from the provided Java parser. This is because of how we chose to design a full EBNF-based parser (rather than just reading from a stream of tokens). Because of this, some things in the parser were a bit hack-y, such as supporting instruction names as identifiers (e.g. `assign, add, 0`). While we haven't found any test cases where the parsers differ, there may be edge cases where our code is unable to parse the input or parses it incorrectly.

While this may not be a bug, our compiler currently doesn't do any type-checking, so we would attempt to optimize something like: `add a, 1, 0.5`. Further, our compiler would allow you to treat arrays as scalars and vice versa. The expectation is that the interpreter will perform the check here; if the program was invalid in the first place, the optimizer cannot make it "more" invalid.

In addition to the known issues, one edge case in our optimizer is unreachable code. If a segment of the code is its own component in the CFG and that component is separate from the entry component for that function, that code will be removed. This is a side-effect of how we do our depth-first search when rebuilding the IR from our CFG. While it should always preserve the correct semantics (and generate more optimal programs), our output may differ from

the expected output. Additionally, if there are any cases where the CFG is constructed incorrectly this could cause the output program to be invalid wherever edges are missing. For example, this previously caused a bug where the lack of "fallthrough" edges for `goto` and `return` meant that some code was missing in the final graph. To repair this, we added "unreachable" edges to preserve the original code structure without changing the semantics of the CFG, but this may lead to future bugs.

5 Instructions

If you have yet to install OCaml, follow the guide for installing OPAM. Upon a successful installation, make sure to run `opam init`. Depending on how your package manager handles installation, you may need to install the latest OCaml compiler on top of OPAM. To do so, run `opam switch 4.07.0`.

Once OPAM has successfully been installed, run `opam install dune menhir ocamlgraph` to get all the dependencies. Next, build the executable with `dune build src/optimize.exe`. With the executable built, you can run the compiler:

```
dune exec src/optimize.exe -- -i <input_file> -o <output_file>
```

You can also use `--gen-cfg` to create a Graphviz file for the input CFG and `--gen-opt-cfg` to create a Graphviz file for the optimized CFG.

By default, the optimizer will only run dead code elimination. To run it with copy propagation in addition to dead code elimination, add the `--copy-prop` flag.

6 Test Results

To test our program, we first went through the quicksort and sqrt tests provided in the starter code. We compiled both of these to programs that match the expected results and expected dynamic instruction counts of the provided examples for Copy-Propagation + Dead-Code. However, we weren't satisfied with this as our only test so we created a fuzzer in the compiler that generates random programs. Using this, we were able to generate long programs that should trigger any common errors in the code. After extensive testing, we patched out all the crashes we found. At this point in time we do not know of any remaining crashes/exceptions that can be thrown during compilation. Since we don't have a reference compiler to test against, it's difficult to determine whether our outputted code is always correct, but based on the test code and other examples that we hand-optimized we're fairly confident in our implementation.