Universal Register Machine Writeup

Ian Griswold
$\wedge \times \succ - \times \wedge$
igriswol@andrew.cmu.edu
May 13, 2019

1: `interpreter.py`

This is just a naive interpreter for register machines, with some quality-of-life features added. By default, it simply prompts the user for a file containing register machine code and the initial register state, then displays the final state of the registers, if it exists. There are options for a step-by-step trace of the running of the machine, a stylized display with labels for each register, and outputting the trace to a file for easier analysis. To add to the stylized display, you can also input an "annotation file" that specifies labels for registers, comments displayed during the execution of the program, and whether or not to interpret the value of a register as a sequence encoding. This was a mostly experimental feature that exists as a proof-of-concept that register machines can be made to be relatively user-friendly.

When the RM code is parsed, any tokens past where an argument would be expected to be are disregarded by the interpreter. With some slight modifications, this could be used to create annotation files, where RM code with comments could be split into a raw RM code file and an annotation file that contains the original comments. Since RM code is extremely nonlinear and can be difficult to follow, comments could be used to mark the beginnings of loops and key parts of the program, even during runtime. When an annotation file is used, every time a given line is run in the interpreter, its associated comment appears beside it. I originally planned to leverage this in the compiler, but I cut these features for time.

2: `diagram.py`

To make RM code easier to analyze, I decided to express it as a graph. Each line of the code is a vertex, with edges leading to each line that can be immediately reached from it. `halt` instructions have outdegree 0, `inc` instructions have outdegree 1, and `dec` instructions have outdegree 2, but these out-edges have an important distinction. One edge represents a successful decrement, and the other represents an unsuccessful decrement. It is useful to consider the unsuccessful decrement edges separately from all other edges. I will refer to the unsuccessful decrements as "zgotos", and all other edges as "gotos".

This file also includes a way to convert these diagrams into RM code. I implemented this with the hope that I'd be able to find useful optimizations that could be done on the graph before writing it back to code, but I was unable to find any significant ones. The vertices keep track of the instruction, target register, and next instruction(s) to execute, but do not have associated line numbers. When writing to a file, line numbers are assigned by marking the starting vertex as 1, then performing a DFS from it, searching along gotos before zgotos. This improves the readability of the resulting code, since gotos tend to be used more frequently than zgotos, and this would lead to better spatial locality (and thus readability) in the code. The DFS also effectively removes any dead code from the final file.

3: `smart_interp.py`

While the naive interpreter had some useful features but was incredibly slow, this version is quite bare-bones but significantly faster. The other features are still applicable and relatively simple to implement, but were cut for time.

This interpreter starts by converting the code to a diagram representation of it, and then starts running it, keeping a list of all the vertices it's executed so far. If it encounters a vertex already in this list, we've found a cycle, and this means we can leverage some convenient properties of the RM language. For short, I will refer to an `inc` instruction as I, a successful `dec` as D, and an unsuccessful `dec` as Z. It's quite clear that multiplication distributes over each of these instructions. If we can find a simple way to express what happens in a cycle, then we can just multiply by some constant to effectively run the cycle that number of times.

Once a cycle is detected, the interpreter tests its effects on a set of dummy registers that are allowed to reach negative values, while quietly running the actual code normally as well. If the code ends up leaving the path of the cycle (for example, by doing a D instead of a Z, or vice versa), we resume normal behavior and clear our list of vertices. If the code completes the cycle a second time, we use the values of the dummy registers after one cycle and the current state of the registers to inform what we do next.

Since `dec` instructions are the only ones with outdegree greater than 1, they are the only way we can break out of the cycle. To determine how many times we follow the cycle before breaking out, we only need to look at those registers which are the targets of `dec` instructions, more specifically D instructions (we've already handled Z by checking that the code stayed in the cycle the second time). The dummy registers allowed us to determine the net effect of the cycle on the values of the registers, so for each register, we divide its current value by the amount it decreases per cycle to determine how many passes we can make before that register causes us to leave the cycle. Since we break out of the cycle at the first change in behavior, taking the minimum of these values tells us how many passes we can "safely" do automatically. If all of the decrement-targeted registers are increasing in the cycle, this means the program diverges, so the interpreter throws an error and halts execution. If we can safely perform any laps around the cycle, the interpreter multiplies the net effect by the number of safe iterations, then adds these changes to the current register state. After doing this, the interpreter resumes normal operation to determine how exactly it will leave the cycle. The list of seen vertices is cleared, and the interpreter continues to add to it, in search for the next cycle.

This means that for any cycle, we will perform at most 3 step-by-step passes of it (and some multiplication in Python). This lets us take something that was originally $O(n)$ in the value of the register inputs and do it in $O(\log n)$, with evidently quite good constants. A useful consequence of this is that we can now effectively treat functions like addition, moving a register to another, copying the contents of a register, multiplication, division, and modulo as if they're built-in instructions. Most importantly, this gives us a feasible way to handle sequences of integers.

4: `misc.py`

This is basically just a library of functions used in several different parts of the program, but most importantly, it handles the sequence encoding used by the URM. Since the smart interpreter allows for fast multiplication, division, and mod, we can feasibly work with a sequence encoding based on those functions. Let $\mathsf{rbin}(n)$ denote some $n \in \mathbb{N}$ written as a reverse binary string and let $[x]_3$ denote some $x \in \{0,1,2\}^*$ interpreted as a number in base 3. For our sequence encoding, I decided on:

$$\langle n_1, n_2, \ldots n_k \rangle = [2\,\mathsf{rbin}(n_k)\,2\,\ldots\,2\,\mathsf{rbin}(n_2)\,2\,\mathsf{rbin}(n_1)]_3$$

Intuitively, this is useful because we can pull off digits from a number from lowest to highest and insert them from highest to lowest. To extract a number from the front of the sequence, we just pull digits off the end using repeated mod and division by 3, and then build the number by repeated doubling and adding the digits we pull off. Putting a number at the front of the sequence works basically the same way. Thus, these sequences most effectively function as stacks, and that's how they'll be used in the URM.

There are certainly sequence encodings that would produce smaller natural numbers as a result, but that wasn't much of a concern here, the focus was more on the squences being easy to manipulate quickly with the smart interpreter, but I chose the smallest bases possible for an encoding like this one. The only real sacrifice is human-readability, but hey, that's what the encoding/decoding functions are for.

5: `compiler.py`

This file defines a register-machine-with-macro (RMM) language and provides a simple compiler from RMM code to raw RM code. It does very little to check validity or safety of the code, but properly handles correctly-written code.

The top part of an RMM file is RM code, but with additional instructions that can be defined later in the file. These lines look much like raw RM code, with an alphabetical instruction name and arguments delimited by spaces.

After the augmented RM code, there must be a line that simply reads `MACROS` as a delimiter. From here on, there are only macro definitions. A macro definition starts with a line as follows:

```
macro name :  r1 r2 ...  rn | l1 l2 ...  ln # comment
```

where `name` is the name of the macro, the `r`x`s` are alphabetical names of register arguments, the `l`x`s` are alphabetical names of line arguments, and `comment` is an optional comment. This line must contain `macro`, the `:` delimiter, at least one register argument, the `|` delimiter, at least one line argument, and the `#` delimiter, in that order. There is a short list of protected names in the code that cannot be used as argument or macro names. Optionally, this line can be followed by a line of the format `temp t1 t2 ...` that declares "temporary" registers that can be used inside the macro but must be zeroed out at the end of the macro. The compiler does not yet automatically check that these registers are properly cleared, but it can be fairly easily checked by analyzing a graph representation of the macro. Finally, below this line is the actual macro definition. It is effectively composed of augmented RM code, and it can use other macros, as long as there are no circular definitions. Additionally, the first token of each line is ignored, allowing the user to put their own line numbers. The macro definition cannot include explicit line references, only arguments, but it

can include explicit register references (but this is not recommended, and if used, you must be very careful about what registers are in use at what times).

The current implementation of the compiler starts with the augmented RM code at the beginning of the file, and scans through it, replacing macros with their definitions whenever found and replacing references to variables with the arguments passed in. To avoid having to constantly change line numbers while expanding the code, line numbers are replaced with "line labels" which allow us to refer to specific lines several macros deep in the code. For example, if line 2 was replaced with a 3-line macro, these lines would be called 2.1, 2.2, and 2.3. After all of the macros are expanded and replaced with raw RM code, the labels are padded with 1s until they are all the same length, then they are sorted lexicographically and replaced with whatever number they are in that ordering.

All that remains are the temporary registers. Whenever a macro being expanded causes a new temporary register to be introduced into the code, it gets a unique number appended to its name, making sure it's able to be identified separately from all other temporary registers. After all the code is expanded, the compiler just needs to allocate unused registers to each of these temporary registers. The current implementation does this naively, with each temp register getting its own register. But this leads to the existence of a lot of inert registers, and we can do much better.

As long as we ensure that every macro zeroes out their temp registers at the end, we can re-use a lot of allocated registers. Let the "referential depth" of a macro be the longest chain of macro definitions from a given macro to raw RM code. We can safely give every macro with referential depth 1 access to the same register as a temp register. Since none of these macros reference each other, only one of them will be using it at a time, and each will zero it out when done, making it usable by another macro. Then, we can give all the macros with referential depth 2 access to a different register, so they can safely reference the ref. depth 1 macros without a register being used concurrently in multiple contexts. Following this pattern, if any macro needs at most a constant number of temp registers, the number of registers we'll need to allocate for temporary registers overall is linear in the maximum referential depth of a macro. Again, this is not implemented by the compiler, but it is feasible, and I did this manually for the URM.

6: `urmsetup.py`

This effectively acts as a wrapper for the smart interpreter, specifically for running the URM. The URM itself is fairly well-documented in `urm.rmm`, so all that's left to explain is how the registers are organized and how the input is encoded.

Since the URM needs to potentially both increase and decrease the index of the current line as well as that of the target register, having an implementation of a stack on its own is not enough. Instead we use a pair of sequences to represent a pair of stacks, one with all lines/registers below the current one, and the other with all lines/registers above the current one. The registers are used as follows:

Register	Description
0	Lines of code above line being read
1	Current instruction
2	Current target register
3	Current goto
4	Current zgoto
5	Lines of code below line being read
6	Registers left of current
7	Current register
8	Registers right of current
9	Program counter (current line number)
10	Register counter (index of current register)
11+	Temp

Finally, we need to express RM code as an integer sequence. For simplicity, each line is expressed as four integers as follows:

$$\texttt{halt} \mapsto 2, 0, 0, 0$$
$$\texttt{dec r g z} \mapsto 1, r, g, z$$
$$\texttt{inc r g} \mapsto 0, r, g, 0$$

The overall behavior of the wrapper asks for an RM file and a starting register input, then encodes this as a URM state. It then runs the smart interpreter on `urm.rm` with that starting state, waits for it to finish, then decodes the final register state into the final register state of the program it was simulating.

7: Future Work

- What the smart interpreter gains in speed, it loses in intelligibility. Adding support for breakpoints in annotation files could make it easier to look at more complex programs step-by-step, for whatever definition of a "step" the user prefers.
- It was relatively simple to make the smart interpreter speed up simple cycles, but is there a way to speed up cycles nested to some given depth? Maybe finding a concise description of the behavior of cycles containing cycles could yield some easily computable function that distributes over them like multiplication distributed over RM instructions.
- While there were no obvious small-scale changes that could make RM programs shorter, there is a larger-scale one that has potential. If there is one snippet of code that gets repeated several times with the same instructions and registers, and with line numbers shifted all by the same amount, we can use a jump table to compress all of these snippets into one. Just set a specific "jump register", jump to the snippet, and case on its value to decide where to go afterwards.
- The concept of spatial locality came up when converting RM code graphs into RM code, but didn't seem to have much of a use for most RM programs. Interestingly though, it might be worth optimizing for it when running these programs on a URM, since it has to physically slide from one register or line of code to the next, and can take a significant amount of time to do so.