CDM Final Project: One-Instruction Languages

Abraham Riedel-Mishaan
ariedelm@andrew.cmu.edu

Description of System

Basically I adopt a slightly different instruction than most, but it is mostly the same, all instructions are of the form

$$l_1\ x\ y\ l_2$$

Where x, y are register numbers and inside the register can be any integer. I have it such that the 0th register is always constant 1 and you initialize the registers starting from register 1 onwards, this way there is always a way to make 1 which is enormously useful.
The instruction basically says: let $R_x = R_x - R_y$, and if $R_x < 0$ now then goto line l_1, if $R_x \geq 0$ goto line l_2. Also note that notationally I may elect to drop the R_x and instead just use x directly when it is clear from context whether I am referring to the register number or the value in the register. The normal instruction has the former as an equality and the latter as a not-equality, however I found this one to actually make much cleaner code when considering operations like divmod, since it is natural to not hit the 0 on the mark but more of a "closest without going over" paradigm of truncation. This actually made it possible for me to implement a much cleaner fast divmod (same efficiency as doing it in any language), fast multiplication, and somewhat fast exponentiation (if done straightforward, it is asymptotically similar to dumber exponentiation, but not repeated squaring). The full rules on how to use my project can be found in the info.txt, but this document is more to explain design decisions and do the computationally complete part.

Optimizing Efficiency

Essentially anytime I see one instruction referencing itself, I just autocompute the rest of it. If two point to each other, the registers used as x are distinct from the ones used as y, then I also contract this. I do this by first figuring out which instruction will end first (i.e. change sign first), and then computing both instructions this many steps (accounting for off-by-one if first instruction ends first, then no longer run second instruction on last loop). The exact method I use can be seen in the code

Computationally Complete

My system is actually really easily computationally complete. It can simulate a register machines inc, dec, and halt commands very easily. Assume all input to the register machine starts as a natural number (since register machine). Assume we start with a register with 1 in it and one with -1 in it that don't change, useful for our OIL. for increment we see
inc x line = line x -1 line
The first bit doesn't matter because, since we assume x is natural, $x - -1 = x + 1$ is always positive as well. For decrement it only gets slightly harder, for:
dec x l1 l2 = l2 x 1 l1
And then we need to add a command just before actually jumping to line l2, where we add 1 back to x (since technically our x drops down to -1, so we need to add back to 0). So add:
l2 x -1 l2

As an intermediary command we jump to then to l2.
for halt, we merely just don't alter any registers then jump past the end of the program.
Thus, we can easily simulate all of the register machine commands (very easily in fact), and as such we can convert and RM program into a OIL program. Since we know RM is computationally complete, that must also mean out OIL system is also computationally complete. Yay.