Introduction to Theoretical Computer Science

Note 1 – two registers suffice

In this note, we will prove the statement that two registers are enough.

1 Three into two does go

We start by showing that we can easily encode a three-register machine program into a two-register machine program, given the three instructions INC, DECJZ, GOTO. (We don't actually need the CLEAR mentioned in Minsky's paper, as it's easily implemented as on the slides; and in fact we won't even use it, *provided* we assume that R_1 is initially zero.)

For this, we will use the 'obvious' tripling function:

$$\langle x, y, z \rangle = 2^x 3^y 5^z$$

The reason for this choice is that it is particularly easy to manipulate the three components by manipulating the code directly.

The target machine has three registers, which we call T_0, T_1, T_2; our interpreting machine has two, R_0, R_1. The principle of the encoding is that R_0 always contains $\langle T_0, T_1, T_2 \rangle$, while R_1 is our work register.

Each instruction of the target machine is translated to a macro of the interpreting machine; we will now define these macros.

First, consider the instruction INC T_0. Incrementing T_0 corresponds to multiplying $\langle T_0, T_1, T_2 \rangle$ by 2:

```
# r1 is assumed zero
# move double r0 into r1
start:  decjz r0 next
        inc r1
        inc r1
        goto start
# move r1 to r0
next:   decjz r1 end
        inc r0
        goto next
# note this leaves r1 zero
```

Equally easily, we can translate INC T_1 and INC T_2. The decrement instructions are a little trickier – to encode DECJZ T_0, *dest*, we need to divide $\langle T_0, T_1, T_2 \rangle$ by 2, but back out and jump if it turns out not to be a multiple of 2. Note, by the way, that R_0 can never contain zero on entry, as this is not a valid triple code.

```
# r1 assumed zero
# by repeated subtraction, move half r0 into r1
# if we hit zero at this point, that's good -
# it means we've halved r0
start:  decjz r0 next
# if we hit zero here, r0 wasn't a multiple
# of two, so we need to recover
        decjz r0 not2
        inc r1
        goto start
# move r1 back to r0
next:   decjz r1 end
        inc r0
        goto next
# recovery: need to restore r0
# first undo the dec at start before we jumped here
not2:   inc r0
# now add double r1 on to r0;
# when finished, jump to branch target
loop:   decjz r1 dest
        inc r0
        inc r0
        goto loop
# note that either way, r1 is zero when we
# leave this macro
```

Similarly, though rather more tediously, particularly with the recovery, we can write macros to decrement T_1 and T_2.

The target GOTO is translated directly.

Modulo correctness of the above macros, we have now encoded a 3-register machine into a 2-register machine.

It is obvious that for any given value of n, this encoding can be extended to encode n-register machines into 2-register machines, and so in some sense we're done: we've shown two registers are enough to encode any given machine, and so any RM-computable function can be simulated by a 2-register RM.

It is an interesting quirk that while a 2-register RM can simulate a general RM, and can therefore compute a suitable *encoding* of any computing function, it is *not* true that a 2-register RM can compute any computable function, in the sense of taking the input in R_0 and leaving the answer in R_0. Such simple functions as n^2, 2^n cannot be computed (in this sense) by a 2-register machine.

2 Unbounded register numbers

However, it would be more satisfactory if we had a single encoding that would translate machines with arbitrary numbers of registers into 2-register machines. This is rather harder, because it means we have to treat the target register index (i in T_i) as a variable quantity, instead of hard-wiring each different instance. Moreover, using the arbitrary extension of the prime-powers pairing function would involve us generating the ith prime on the fly every time we touched T_i. While this is possible, it's pretty hideous (and certainly not obviously doable with two registers).

So in order to sketch the proof, I'll assume some pairing function $\langle , \rangle_2 : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \setminus \{0\}$ (note that $2^x 3^y$ is such a function), and then define the following coding of sequences:

$$\langle \rangle = 0$$

$$\langle x, s \rangle = \langle x, \langle s \rangle \rangle_2 \text{ where } s \text{ is a sequence}$$

Hence, extracting the i element of a coded sequence is simply a matter of pulling out the first component of a pair, i times. Assuming the unpairing function is reasonable, this can be done with a couple of auxiliary registers.

Similarly, updating the ith element is routine, though a little more tedious. Try writing the pseudo-code for it, and see how many auxiliary registers you need.

Given that, then the rest of the translation process is very simple: if interpreting machine R_0 holds the coded registers, then target machine instruction INC(i) turns into
LOAD R_1 with i
EXTRACT T_i into R_2, saving $\langle T_0, \ldots, T_{i-1} \rangle$ in R_3 and leaving $\langle T_{i+1}, \ldots \rangle$ in R_0
INC (R_2)
PACK $T_0, \ldots, T_{i-1}, R_2, T_{i+1}, \ldots$ into R_0

The macros are a bit fiddly, especially the re-packing (if you are a Lisp programmer, you may remember basic exercises that involved a lot of list reversing), but not difficult.

Similarly for DECJZ and GOTO.

While I haven't counted, I'd guess that ten registers are enough to do all that conveniently – possibly fewer.

But we wanted just *two* registers.

No problem! If we can produce an arbitary-register-RM interpreter using ten registers, we can encode it into a two-register machine using the technique of the previous section.

Note, however, that if we do all this with the primes coding function, the machine has zilch chance of doing anything useful in the lifetime of the universe. What is the code of the sequence $(2, 2, 2, 2, 2)$?

For the purpose of this section, any coding funtion will do, so we can code more efficiently using the diagonal pairing function (or to be precise, 1 + the diagonal pairing function).

For the construction of section 1, I don't know any tupling function that can be calculated with only one scratch register, and is not exponential in the inputs. I also don't know that no such function exists.

I believe that the GOTO is necessary, in that without it we cannot write a general RM simulator using just two registers. So far, the proof escapes me.