Register Machine*

Graeme Taylor

May 6, 2004

Motivation

The classic model of computation is the Turing Machine, an idealised mathematical construct developed before modern digital computers. The Register (or Abacus) Machine model is of equal power to the Turing Machine, but bears a greater resemblance to modern computing based on random-access storage. As any Register Machine can be simulated by a Turing Machine, this makes it much easier to prove that functions are Turing-computable simply by showing that it can be computed by a Register Machine.

A third, and more mathematical, flavour of computation is the concept of (primitive) recursive functions. These can be simulated by Register Machines (see later), but crucially can themselves simulate Turing Machines, giving an equivalence in computational power between all three models- which lends weight to the Church-Turing Thesis that this level of power is sufficient for all effectively computable functions.

Programming Register Machines

A Register Machine program is a finite list of commands, which operate on a finite collection of non-negatively valued registers, referenced by number. Fortunately ($\rightarrow$), there are only two commands available:

- `i+; goto l`
  This instructs the program to increase the value of register number i by 1, then move to line l.
- `if i=0 then {goto` l_1`} else {i-; goto` l_2`}`
  If register with number i is already zero, move to line l_1; otherwise subtract one from that register and move to l_2.

*First appeared on Everything2, at *http://www.everything2.com/index.pl?node_id=1535830*

Further, program flow is assumed to start at line 1, and to halt if sent to a non-existant line. Output is assumed to be placed in register 1. (or the first k registers for multiple output) A flow chart notation is often employed, but this gets messy fast in plaintext.

Some Examples

The Register Machine makes no assumptions about initial values of registers, as generally you'll want to bolt different register machines together to implement a complicated function. Thus one of the simplest operations is also one of the most useful: setting a register to zero:

```
0:  if n=0 then {goto 99} else {n-; goto 0}
```

Hopefully it should be clear what this does- it tests for zero, and if this fails it decreases the value by one until it is successful. Once this state is met, line 99 is executed. If there is no such line, then the program halts and we have the value zero in register n; otherwise we continue with the instructions at line 99 which may make use of this now empty register.

Sometimes however zeroing the register may be redundant. For instance, the easiest way to add the value of two registers together is to leave one alone and transfer the contents of the other into it:

```
0:  if m=0 then {goto 99} else {m-; goto 1}
1:  n+; goto 0
```

Perhaps you can spot the downside of this simple addition program- in the process of adding m to n, m is itself annihilated. In effect what this program does is drain m into n one step at a time; you might want to try coding a Register Machine to copy m into n without destroying (or whilst simultaneously rebuilding) m.

Let us suppose we (you!) have created such a register machine. Then we could call it $(m) + (n) \rightarrow n$, say: meaning that it sets the value of register n to the sum of the values in m and n whilst preserving m. Equipped with this, we can easily define multiplication in a macro form, given here for $m \times p$ (destroying m and putting the answer into n, which should be initialised to zero):

```
0:  if m=0 then {goto 99} else {m-; goto 1}
1:  (p) + (n) → n; goto 0
```

This of course isn't strictly speaking a register machine, as 'line 1' is really several lines of valid register machine code. Nonetheless, once we are confident that the desired functionality of this line can be coded in the form of several others, we can speed up proofs of computability by taking this macro or modular approach to writing program lists.

Simulation of Register Machines by Turing Machines

Whilst this is entirely possible, the precise details depend on the formulation of Turing Machines and tapes you wish to employ, and once decided still takes much effort to rigorously describe the various stages of initialisation, simulation and cleanup required. Thus I hope you will take it on trust that any function computed by a Register Machine can be computed by a Turing Machine, or follow it up in one of the many books on the subject!

Simulation of Recursive Functions by Register Machines

This, however, is fairly easy to do if you accept a description of each building block of recursive functions via a macro-style Register Machine, and then consider any complicated recursive function as being a larger Machine using those as macros.

The Successor function

This merely needs to increase Register 1 by 1, so our Machine is simply

```
0:  1+; goto 99
```

The Zero function

The code to zero a register was given as the first example above.

The Projection function

The projection π_i^k depends on the value of i. If it's 1, then the result is already in register 1, so the 'empty program' with no command is used! Otherwise, we wish to move register i into register 1:

```
0:  if R1=0 then {goto 1} else {(1)-; goto 0}
1:  if Ri=0 then {goto 99} else {(i)-; goto 2}
2:  (1)+; goto 1
```

Composition

This will vary from function to function- consider an example where H is the composition of f,g, and h of the form $H(x_1, x_2, x_3) = f(g(x_1, x_2, x_3), h(x_1, x_2, x_3))$:

(In macro form)
(1) $\rightarrow x_1$ //Saving the value of register 1
(2) $\rightarrow x_2$ //Save register 2
(3) $\rightarrow x_3$ //Save register 3

`Register program for g` //puts answer in R_1
`(1)` $\rightarrow$ `aux` //store that answer elsewhere
`(x1)` $\rightarrow$ `1` //Restores R_1 for next function evaluation
`(x2)` $\rightarrow$ `2` //Restores R_2
`(x3)` $\rightarrow$ `3` //Restores R_3
`Register program for h` //puts answer in R_1
`(1)` $\rightarrow$ `2` //moves it to R_2
`(aux)` $\rightarrow$ `1` //Puts the first argument in R_1
`Register program for f` //acts on R_1, R_2 as desired.

Primitive Recursion

If h is defined from f and g by primitive recursion ($h = Pr(f, g)$) then we can code this under the following assumptions- z stores the result whilst y acts as a counter; x, y, z, and i are registers that shoudln't be acted on by f or g, and register 2 takes the initial value y_0.

(In macro form)
`(1)` $\rightarrow$ `x;`
`(2)` $\rightarrow$ `y;`
`Program for f;`
`(1)` $\rightarrow$ `z`
`0` $\rightarrow$ `i` //Now $z = h(x, i)$ and $i + y = y_0$
`if (y)=0 then {goto` **B**`} else {y-; goto` **A**`}`
A `(x)` $\rightarrow$ `1;`
`(y)` $\rightarrow$ `2;`
`(z)` $\rightarrow$ `3;`
`Program for g;`
`(1)` $\rightarrow$ `z;`
`i+; goto` **A**
B `(z)` $\rightarrow$ `1;` //return the answer z

Minimisation

So far we have defined all the building blocks for primitive recursive functions. The operation of minimisation offers all recursive functions. So let f be the function we seek to find the minimisation of, and suppose the program for f doesn't alter registers x or y:

(In macro form)
`(1)` $\rightarrow$ `x;`
`0` $\rightarrow$ `y;`
A `x` $\rightarrow$ `1;`

```
y → 2;
program for f;
if (1)=0 then goto C else goto B; //this will need to be a macro to undo the usual subtraction on else conditions
B y+; goto A
C (y) → 1;
```