Register Machines

Klaus Sutner
Carnegie Mellon University

Fall 2011

Outline

1. Register Machines
2. Coding
3. Universality

Computability

We need a formal definition of computability. We would like a definition that

- is easy to understand and apply, and
- matches our intuitive notion of computability.

There are many plausible approaches, we'll start with a model that is slightly reminiscent of assembly language programming, only that our language is much, much simpler than real assembly languages.

Why not C?

- It might be tempting to try a "definition" along the lines of: a function is computable if it can be implemented as a C program.
- Alas, there are lots of problems: why C rather than Java or APL, which version of C, which compiler, which operating system, which hardware?
- Your most likely answer: it does not matter!
  OK, but then: why does it not matter?
- Because there is an invariant, an underlying notion of computability–which we are trying to pin down, precisely.

Register Machine

Definition

A register machine (RM) consists of a finite number of registers and a control unit.

We write $R_0, R_1, \ldots$ for the registers.

Each register contains a natural number: $[R_i]$ denotes the content of the ith register.

Note: there is no bound on the size of the numbers stored in our registers, any number of bits is fine.

The control unit is capable of executing certain instructions that manipulate the register contents.

Instruction Set

Our instruction set is very, very primitive:

- `inc r k`
  increment register R_r, goto k,
- `dec r k l`
  if $[R_r] > 0$ decrement register R_r and goto k, otherwise goto l,
- `halt`
  well . . .

Note that there is no indirect addressing; these machines are sometimes called counter machines. The gotos refer to line numbers in the program.

RM Programs

Definition

A register machine program (RMP) is a sequence of RM instructions $P = I_0, I_1, \dots, I_{n-1}$.

For example, the following program performs addition:

```
// addition   R1 R2 --> R0
  0:   dec 1  1  2
  1:   inc 0  0
  2:   dec 2  3  4
  3:   inc 0  2
  4:   halt
```


RM-Computability

Definition

A function is RM-computable if there is some RMP that implements the function.

This is a bit wishy-washy: we really need to fix

- a register machine program P,
- input registers I, and
- output registers O.

Then (P, I, O) determines a partial function $f \colon \mathbb{N}^k \hookrightarrow \mathbb{N}^\ell$ where $k = |I|$ and $\ell = |O|$.

Reasonable I/O Conventions

- Given input arguments $a_0, \dots, a_{k-1} \in \mathbb{N}$, set the input registers: $[R_i] = a_i$.
- All other registers in P are initialized to 0.
- Then run the program.
- If it terminates read off the values of R_j, $j \in O$, producing the result $(b_1, \dots, b_\ell) = f(a_1, \dots, a_k)$.
- If P does not terminate, $f(\mathbf{a})$ is undefined.

Run the Program?

Note that we may safely assume that $P = I_0, I_1, \dots, I_{n-1}$ uses only registers R_i, $i < n$, so all numbers in the instructions are bounded by n.

To describe a computation of P we need to explain what a snapshot of a computation is, and how get from one snapshot to the next. Clearly, for RMPs we need the current instruction and the contents of all registers.

Definition

A configuration of P is a pair $C = (p, \mathbf{x}) \in \mathbb{N} \times \mathbb{N}^n$.

Steps in a Computation

Configuration $(p, \mathbf{x})$ evolves to configuration $(q, \mathbf{y})$ in one step under P iff

- $I_p = \texttt{inc r k}$:
  $q = k$ and $\mathbf{y} = \mathbf{x}[x_r \mapsto x_r + 1]$.
- $I_p = \texttt{dec r k l}$:
  $x_r > 0$, $q = k$ and $\mathbf{y} = \mathbf{x}[x_r \mapsto x_r - 1]$ or
  $x_r = 0$, $q = l$ and $\mathbf{y} = \mathbf{x}$.

Notation: $(p, \mathbf{x}) \vdash_P^1 (q, \mathbf{y})$.

Note that if $(p, \mathbf{x})$ is halting (i.e. $I_p = \texttt{halt}$) there is no next configuration. Ditto for $p \geq n$.

Whole Computation

Define

$$(p, \mathbf{x}) \vdash_P^0 (q, \mathbf{y}) \iff (p, \mathbf{x}) = (q, \mathbf{y})$$

$$(p, \mathbf{x}) \vdash_P^t (q, \mathbf{y}) \iff (p, \mathbf{x}) \vdash_P^{t-1} (q', \mathbf{y}') \vdash_P^1 (q, \mathbf{y})$$

$$(p, \mathbf{x}) \vdash_P (q, \mathbf{y}) \iff \exists t \; (p, \mathbf{x}) \vdash_P^t (q, \mathbf{y})$$

A computation (or a run) of P is a sequence of configurations C_0, C_1, C_2, ... where $C_i \vdash_P^1 C_{i+1}$.

Finite versus Infinite

Note that a computation may well be infinite:

```
0:  inc 0 0
```

has no terminating computations at all. More generally, for some particular input a computation on a machine may be finite, and infinite for other inputs.

We will write

$$(C_i)_{i<\omega} \quad \text{and} \quad (C_i)_{i<n}$$

for infinite versus finite computations.

Termination

We really want terminating computations: finite and ending in a halt instruction.

Alas, computations may get stuck:

```
0: inc 0 1
```

trivially cannot reach a halting instruction.

Bad labels are easy to discover, but even if all labels are good and there is a halting instruction it's not clear whether one ever gets there.

Computing a Function

The initial configuration for input $\mathbf{a} \in \mathbb{N}^k$ is $I_\mathbf{a} = (0, (\mathbf{a}, \mathbf{0}))$.

Definition

A RMP P computes the partial function $f\colon \mathbb{N}^k \hookrightarrow \mathbb{N}^\ell$ if for all $\mathbf{a} \in \mathbb{N}^k$ we have:

- If $\mathbf{a}$ is in the domain of f then the computation of P on $\mathbf{x}$ terminates in configuration $C_n = (p, \mathbf{y})$ where $f(\mathbf{a}) = (y_k, \ldots, y_{k+\ell-1})$ and $I_p = \texttt{halt}$.
- If $\mathbf{a}$ is not in the domain of f then the computation of P on input $\mathbf{a}$ fails to terminate.

A Subtlety

According to our definition, it is not admissible that a RMP get stuck (e.g., because a label denotes a non-existing instruction).

More precisely, the RMP simply does not define a function.

Exercise

Modify the definition so "getting stuck" is allowed and show that we obtain exactly the same class of partial functions this way.

Time Complexity

The number of steps in a finite computation provides a measure of complexity, in this case time complexity.

Given a RM P and some input $\mathbf{x}$ let $(C_i)_{i<N}$ where $N \leq \omega$ be the computation of P on $\mathbf{x}$.

We write

$$T_P(\mathbf{x}) = \begin{cases} N-1 & \text{if } N < \omega, \\ \infty & \text{otherwise.} \end{cases}$$

for the time complexity of P.

Named Registers

To make RMPs slightly easier to read we use names such as X, Y, Z and so forth for the registers.

This is just a bit of syntactic sugar, if you like you can always replace X by R_0, Y by R_1 and so forth.

And we will be quite relaxed about distinguishing register X from its content $[X]$.

Digression: Notation

There is actually something very important going on here: we are trying to produce notation that works well with the human cognitive system.

Humans are exceedingly bad at dealing with fully formalized systems; in fact, we really cannot read formal mathematics except in the most trivial (and useless) cases.

The current notation system in mathematics evolved over centuries and is very carefully fine-tuned to work for humans.

Computers need an entirely different presentation and it is very difficult to move between the two worlds.

Example: Multiplication

Here is a program that multiplies registers X and Y, and places the product into Z.

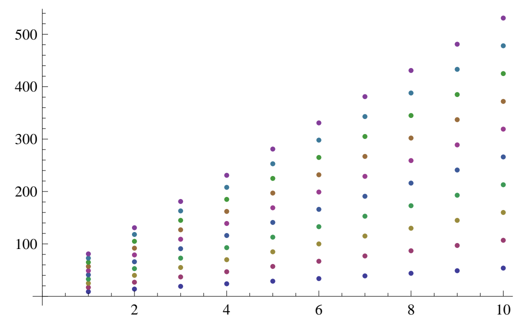
```
// multiplication   X Y --> Z
  0:   dec X  1  6
  1:   dec Y  2  4
  2:   inc Z  3
  3:   inc U  1
  4:   dec U  5  0
  5:   inc Y  4
  6:   halt
```


A Computation

0	$(2,2,0,0)$	1	$(0,2,2,0)$
1	$(1,2,0,0)$	2	$(0,1,2,0)$
2	$(1,1,0,0)$	3	$(0,1,3,0)$
3	$(1,1,1,0)$	1	$(0,1,3,1)$
1	$(1,1,1,1)$	2	$(0,0,3,1)$
2	$(1,0,1,1)$	3	$(0,0,4,1)$
3	$(1,0,2,1)$	1	$(0,0,4,2)$
1	$(1,0,2,2)$	4	$(0,0,4,2)$
4	$(1,0,2,2)$	5	$(0,0,4,1)$
5	$(1,0,2,1)$	4	$(0,1,4,1)$
4	$(1,1,2,1)$	5	$(0,1,4,0)$
5	$(1,1,2,0)$	4	$(0,2,4,0)$
4	$(1,2,2,0)$	0	$(0,2,4,0)$
0	$(1,2,2,0)$	6	$(0,2,4,0)$

```
// multiplication   X Y --> Z
  0:   dec X  1  6
  1:   dec Y  2  4
  2:   inc Z  3
  3:   inc U  1
  4:   dec U  5  0
  5:   inc Y  4
  6:   halt
```

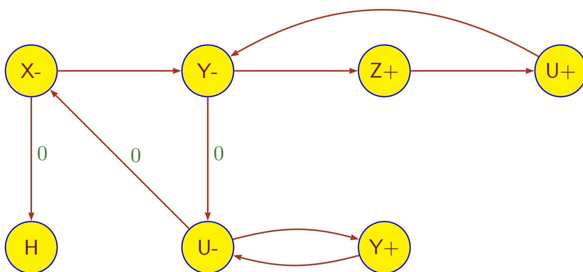

Time Complexity?



Exercise

Determine the time complexity of the multiplication RM.

Flowgraph for Multiplication

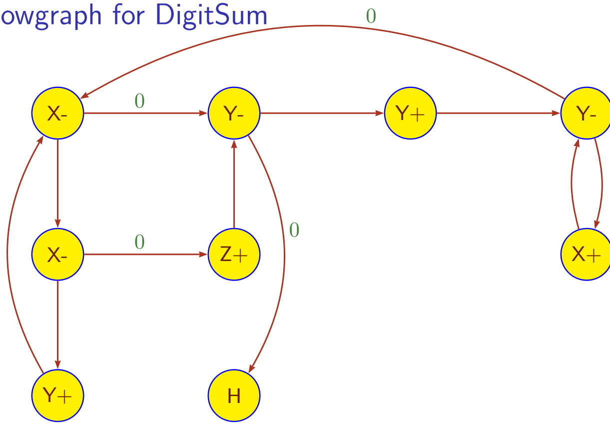


Example: Binary Digit Sum

The following RMP computes the number of 1's in the binary expansion of X, the so-called binary *digit sum* of x.

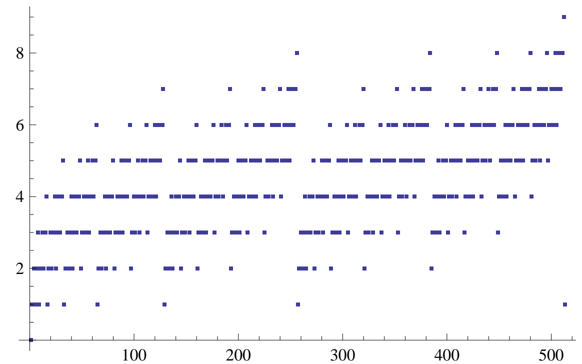
```
// binary digitsum of X --> Z
  0:   dec X  1  4
  1:   dec X  2  3
  2:   inc Y  0
  3:   inc Z  4
  4:   dec Y  5  8
  5:   inc Y  6
  6:   dec Y  7  0
  7:   inc X  6
  8:   halt
```

Flowgraph for DigitSum



Digit Sum

The (binary) digit sum is actually quite useful in some combinatorial arguments.



- Register Machines

2. Coding

- Universality

RMs and Data Structures

Since register machines operate only on natural numbers it is not clear how powerful they really are, compared to, say, C programs.

For example, could one concoct a RMP that computes shortest paths in a graph?

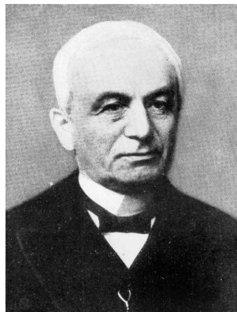
We would need to code the graph as a number. Plus all other needed data structures: lists, trees, matrices, ...

How?

Leopold Kronecker

Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk.

"Dear god" made the integers, everything else is the work of men.



Sequence Numbers

We would like to express a sequence $a_1, a_2, \ldots, a_n$ of natural numbers as a single number $\langle a_1, a_2, \ldots, a_n \rangle$. So we need a coding function, a polyadic map of the form

$$\langle . \rangle : \mathbb{N}^* \to \mathbb{N}$$

and a decoding function that extracts the components of a coded sequence

$$\text{dec} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

so that

$$a_i = \text{dec}(\langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle, i)$$

for $i = 0, \ldots, n-1$.

Length

Note that dec(x, i) need not be defined when i is too large or when x is not of the form $\langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle$.

One usually insists that the function is total and that there is a *length function*

$$\text{len}(\langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle) = n$$

The numbers that appear as codes of sequences are called sequence numbers.

Exercise

Show how to check if a number is a sequence number given dec *and* len.

Pairs

The first step is to select a pairing function, an injective map $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

There are many possibilities, for our purposes the following is the best choice:

$$\pi(x, y) = 2^x(2y + 1)$$

So

$$\pi(5, 27) = 1760 = 11011100000_2$$

In Binary

Note that the binary expansion of $\pi(x, y)$ looks like so:

$$y_k y_{k-1} \ldots y_0\, 1\, \underbrace{00 \ldots 0}_{x}$$

where $y_k y_{k-1} \ldots y_0$ is the binary expansion of y with y_k being the most significant digit.

This makes it easy to find the corresponding unpairing functions:

$$x = \pi_1(\pi(x, y)) \qquad y = \pi_2(\pi(x, y)).$$

Extending to Sequences

$$\langle \text{nil} \rangle = 0$$
$$\langle a_1, \ldots, a_n \rangle = \pi(a_1, \langle a_2, \ldots, a_n \rangle)$$

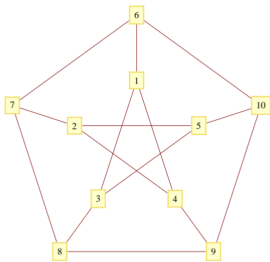
Here are some sequence numbers for this particular coding function:

$$\langle 10 \rangle = 1024$$
$$\langle 0, 0, 0 \rangle = 7$$
$$\langle 1, 2, 3, 4, 5 \rangle = 532754$$

That's It!

We can now code any discrete structure as an integer by expressing it as a nested list of natural numbers, and then applying the coding function.

For example, the so-called Petersen graph on the left is given by the nested list on the right.



$$((1,3),(1,4),(2,4),(2,5),(3,5),$$
$$(6,7),(7,8),(8,9),(9,10),(6,10),$$
$$(1,6),(2,7),(3,8),(4,9),(5,10))$$

Not So Fast

Of course, we also need to be able to operate on sequence numbers.

Exercise

Construct a RMP that computes the length of a sequence, given the code number as input.

Exercise

Construct a RMP that computes the two unpairing functions $x = \pi_1(\pi(x, y))$ *and* $y = \pi_2(\pi(x, y))$

Exercise

Construct a RMP that implements the join of two lists given as sequence numbers.

Flowgraph dec(x, i)

Prepend b to x

Coding RMPs

Note that a single instruction of an RMP can also be coded as a number:

- `halt` $\langle 0 \rangle$
- `inc r k` $\langle r, k \rangle$
- `dec r k l` $\langle r, k, l \rangle$

And a whole program can be coded as the sequence number of these numbers.

Example: Addition

For example, the simplified addition program

```
// addition   R0 + R1 --> R1
  0:   dec 0  1  2
  1:   inc 1  0
  2:   halt
```

has code number

$$\langle \langle 0, 1, 2 \rangle, \langle 1, 0 \rangle, \langle 0 \rangle \rangle = 88098369175552.$$

Note that this code number does not include I/O conventions, but it is not hard to tack these on if need be.

- Register Machines
- Coding
3. Universality

Simulating Random Access Memory

According to the last section, we can code a RMP $P = I_0, I_1, \ldots, I_{n-1}$ as an integer e.

Moreover, we can access the instructions in the program by performing a bit of arithmetic on the RM. Note that we can do this non-destructively by making copies of the original values.

So, if e is stored in register E and some index stored in p (for program counter) we can retrieve instruction I_p and place it into register I.

Simulating a RM

Suppose we are given a sequence number C that codes some RMP P requiring one input x.

We claim that there is a universal register machine (URM) $\mathcal{U}$ that, on input C and x, simulates program P on x.

Alas, writing out $\mathcal{U}$ as a RMP is too messy, we need to use a few "macros" that shorten the program.

Of course, one has to check that all the macros can be removed and replaced by corresponding RMPs.

Macros

- `copy r s k`
  Non-destructively copy the contents of R_r to R_s, goto k.
- `zero r k l`
  Test if the content of R_r is 0; if so, goto k, otherwise goto l.
- `pop r s k`
  Interpret R_r as a sequence number $a = \langle b, c \rangle$; place b into R_s and c into R_r, goto k. If $[R_r] = 0$ both registers will be set to 0.
- `read r t s k`
  Interpret R_r as a sequence number and place the $[R_t]$th component into R_s, goto k. Halt if $[R_t]$ is out of bounds.
- `write r t s k`
  Interpret R_r as a sequence number and replace the $[R_t]$th component by $[R_s]$, goto k. Halt if $[R_t]$ is out of bounds.

URM

```
// universal RM
  0:   copy   C  R  1             // R = C
  1:   write  R  p  x   2         // R[0] = x
  2:   read   C  p  I   3         // I = C[p]
  3:   pop    I  r  4             // r = pop(I)
  4:   zero   I 13  5             // if( I == 0 ) halt
  5:   pop    I  p  6             // p = pop(I)
  6:   read   R  r  x   7         // x = R[r]
  7:   zero   I  8  9             // if( I != 0 ) goto 9
  8:   inc    x 12                // x++; goto 12
  9:   zero   x 10 11             // if( x != 0 ) goto 11
 10:   pop    I  p  2             // p = pop(I)
 11:   dec    x 12 12             // x--
 12:   write  R  r  x   2         // R[r] = x; goto 2
 13:   halt
```


Who Cares?

Using similar ideas, one can show that any "intuitively computable" function is already RM-computable. So the universal RM can compute all computable functions.

It has become fashionable in some quarters to talk about "breaking through the Turing barrier," Turing machines being another model of computation. This is complete rubbish, there is no indication whatsoever that physical computation can go beyond the "Turing limit." And mathematics can trivially go beyond it.

No Coding

One very pleasant feature of register machines is that the do not require any input/output coding for arithmetic functions.

In general this is emphatically not the case. We will shortly introduce Turing machines that naturally operate on strings, so numbers have to be coded (say, using binary notation).

Things get worse if on looks at more exotic models of computation such as cellular automata. In fact, any physics-like model tends to produce headaches when it comes to I/O conventions.

- Register machines are a simple intuitive model of computation.
- Sequence numbers can be used to code sequences of natural numbers as single numbers.
- As a consequence, all finite discrete structures can be expressed as natural numbers.
- There is a universal register machine capable of simulating all register machines. In fact, this machine can simulate all possible computations.