Automata and Formal Languages

October 14, 2019

1 Register Machines and Computability

Books: PTJ (Chapter 4)

Note: Here $\mathbb{N} = \{0, 1, 2, \ldots\}$

A ***register machine (RM)*** consists of:

1. A sequence of ***registers*** $R_1, R_2, R_3, \ldots$ where at discrete time steps $t = 0, 1, 2, \ldots$ have $R_i(t) \in \mathbb{N}$, In fact, we only have finitely many registers, and regard $R_i \equiv 0$ for all $i \geq I$.
2. A finite ***program*** consisting of a fixed number of ***states*** S_0 (Halt), S_1 (Start), $S_2, \ldots, S_n$. Each state comes with a fixed instruction performed when in state S_i. When the computer reaches Halt, we get the output from R_1. Otherwise, for $1 \leq i \leq n$ we have 2 types of ***commands***:
   (a) Increment R_j, then move to state S_k. We write this $S_i : (j, +, k)$.
   (b) If $R_j \neq 0$ then decrement R_j, then move to state S_k. Otherwise move to state S_l. We write this $S_i : (j, -, k, l)$.

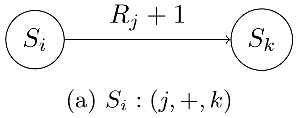
A ***sequence of instructions*** for a RM is the ordered list of the instructions for the program.
An ***input*** for a RM is, for some $k \geq 1$, a finite k-tuple $(n_1, \ldots, n_k) \in \mathbb{N}^k$ which are the initial values of $R_1, \ldots, R_k$. The other registers are set to 0.

A ***program diagram*** for a RM is a directed graph with vertices being the states of the machine and the labelled arrows denote the instructions: $S_i : (j, +, k)$

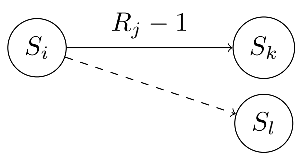
We can then use these to describe programs:

For any $k > 0$ a program P ***halts*** on input $(m_1, m_2, \ldots, m_k) \in \mathbb{N}^k$ if it ever reaches state S_0, written $P(m_1, \ldots, m_k) \downarrow$
The ***halting set*** $\Omega(P)$ is a set of inputs on which P halts.

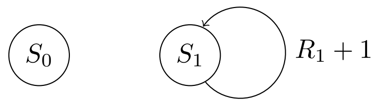
$$\Omega(P) = \cup_{k>0}\{(m_1, \ldots, m_k) : P(m_1, \ldots, m_k) \downarrow\}$$



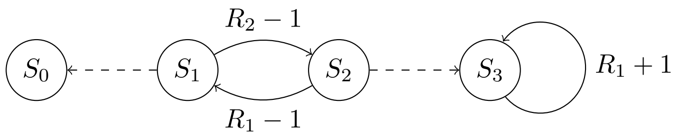
(a) $S_i : (j, +, k)$



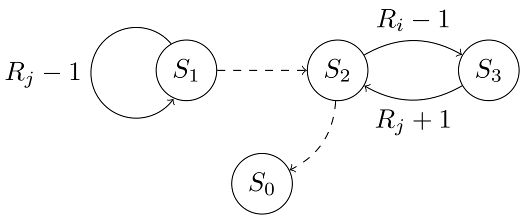
(b) $S_i : (j, -, k, l)$



(a) Repeatedly increment R_1, never halting



(b) For input (n_1, n_2) returns $n_1 - n_2$ if $n_1 \geq n_2$, else never halt



(c) Transfer R_i to R_j, emptying R_i

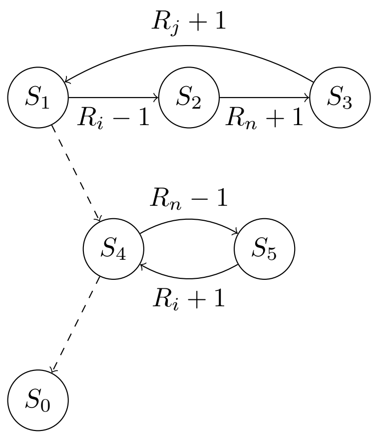
If P does not halt, we write $P(m_1, \ldots, m_k) \uparrow$.

For each program P, the ***upper register index*** $\text{upper}(P)$ is the largest index of a register appearing in the instructions for P. So if $i > \text{upper}(P)$ then R_i never changes.

A ***partial function*** $f : \mathbb{N}^k \to \mathbb{N}$ is one where the domain of f is a subset of $\mathbb{N}^k$, and undefined otherwise. If f is defined everywhere then we call it a ***total function***. This lets us define these programs as functions - we say f is ***partial computable*** by a program P such that $\forall (m_1, \ldots, m_k) \in \text{dom}(f)$ have $P(m_1, \ldots, m_k) \downarrow$ with $f(m_1, \ldots, m_k) = R_1$ on halting, and $\forall (m_1, \ldots, m_k) \notin \text{dom}(f)$ we have $P(m_1, \ldots, m_k) \uparrow$. Hence any program P and $k > 0$ gives a partial function $f : \mathbb{N}^k \to \mathbb{N}$.

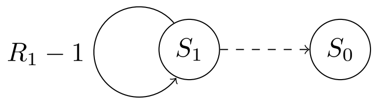
Lemma 1.1. *We can add R_i to R_j leaving R_i unchanged.*

Proof.



Thus by setting $(i = 2, j = 1)$ we see that $(n_1, n_2) \mapsto n_1 + n_2$ is total computable. □

We have already seen that the function $n \mapsto 0$ is also computable. This can be done with the machine:



Corollary 1.2. *There exists a routine which can copy R_i to R_j leaving R_i unchanged.*

Proof. First empty R_j, then use **1.1** to add R_i to R_j. $\square$

We can use these as subroutines to join with other programs P. Use registers R_n s.t. $n >$ upper(P) and largest input register. Then replace the halt state of P with the start state of the subroutine. In fact we have already done this - if you look carefully at the adding machine, you can see that the middle section is the same as the machine in (c) of the examples - this is the part where we replace the value in R_i from its temporary location in R_n.

Partial Recursive Functions

Partial computable functions have good closure properties.

Theorem 1.3.

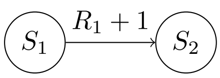
1. *For $i \leq k$, the* **projection function** $(n_1, \ldots, n_k) \mapsto n_i$ *is computable.*
2. *The zero function $n \mapsto 0$ and* **successor function** $n \mapsto n+1$ *are computable*
3. *(Composition) If $f : \mathbb{N}^k \to \mathbb{N}$ and $g_1, \ldots, g_k : \mathbb{N}^l \to \mathbb{N}$ are all partial computable then so is the composition function $h(n_1, \ldots, n_l) = f(g_1(n_1, \ldots, n_l), \ldots, g_k(n_1, \ldots, n_l))$ where defined. If $f, g_1, \ldots, g_k$ are total functions, so is h.*
4. *(Recursion) If f on k variables and g on $k+2$ variables are partial computable, then so is the partial function $h : \mathbb{N}^{k+1} \to \mathbb{N}$ defined inductively as:*

$$\begin{aligned} h(n_1, \ldots, n_k, 0) &= f(n_1, \ldots, n_k) \\ h(n_1, \ldots, n_k, n_{k+1} + 1) &= g(n_1, \ldots, n_{k+1}, h(n_1, \ldots, n_{k+1})) \end{aligned}$$

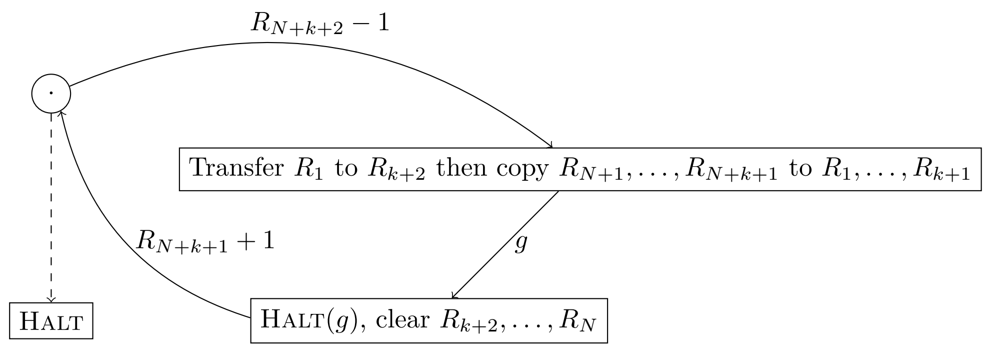
   Moreover, f, g total $\implies$ h total.
5. *(Minimisation) If f on $k+1$ variables is partial computable then so is the partial function $g : \mathbb{N}^k \to \mathbb{N}$ defined by $g(n_1, \ldots, n_k) = n$ if $f(n_1, \ldots, n_k, n) = 0$ and $f(n_1, \ldots, n_k, m) > 0$ for all $m < n$, and is undefined if no zero is ever found. Note that f total $\not\Rightarrow$ g total.*

Proof.

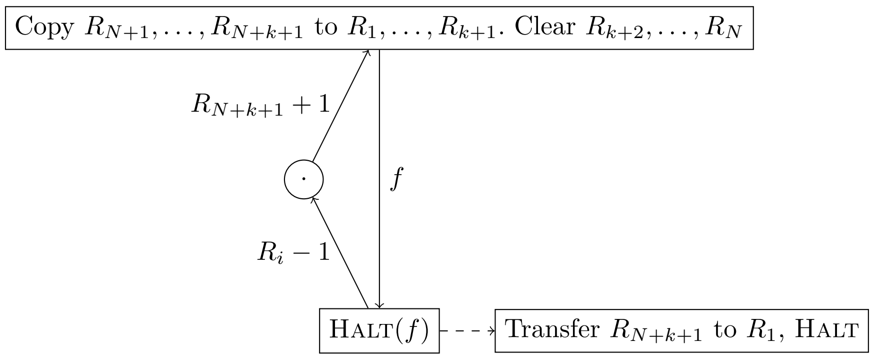
1. We can use the program Transfer R_i to R_1, Halt.
2. Zero function has already been seen. For successor function, use:



3. First transfer $R_1, \ldots, R_l$ to $R_{N+1}, \ldots, R_{N+l}$ where N is large enough to not be needed in other subroutines. Then for each $1 \leq i \leq k$ in turn, copy $R_{N+1}, \ldots, R_{N+l}$ to $R_{k+1}, \ldots, R_{k+l}$, perform g_i but with all registers shifted up by k and then transfer answer from R_{k+1} to R_i, then clear $R_{k+2}, \ldots, R_N$. Finally, apply f.
4. Copy $R_1, \ldots, R_k$ to $R_{N+1}, \ldots, R_{N+k}$, transfer R_{k+1} to R_{N+k+2} ("counts down"), then do f. Then:



5. Copy $R_1, \ldots, R_k$ to $R_{N+1}, \ldots, R_{N+k}$. Then



□

The class of ***partial recursive functions*** is the smallest class of partial functions from $\mathbb{N}^k$ to $\mathbb{N}$ over all $k \geq 1$ closed under the operations **1.3** (1) to (5). That is, f can be constructed from basic functions and applications of (3), (4), (5) a finite number of times.

So **1.3** says that partial recursive $\implies$ partial computable.

A partial function is ***primitive recursive*** if we never use **1.3** (5) its construction. Note that primitive recursive $\implies$ total recursive, as (5) was the only construction that breaks the totality of the function. ⟦The converse implication is not true: the Ackermann function. ⟧

Example: $+$ and $\times$ are primitive recursive:

$+$: Let $h(m, 0) = m$, $h(m, n+1) = h(m, n) + 1 = g(m, n, h(m, n))$, where $g(x, y, z) = z + 1$.

$\times$: $H(m, 0) = 0$, $H(m, n+1) = H(m, n) + m = g(m, n, H(m, n))$ for $g(x, y, z) = x + z$.

Example: $(m, n) \mapsto m^n$ is primitive recursive - left as exercise.

We need to be able to "encode" finite sequences of arbitrary length in $\mathbb{N}$. For $n > 0$ and $i \in \mathbb{N}$, write p_i for the $(i+1)th$ prime (so $p_0 = 2$). Write $(n)_i$ for the largest power of the prime p_i that divides n.