Computation Theory

University of Cambridge

Ashwin Ahuja

Computer Science Tripos Part IB
Paper 6

April 2019

Contents

1 Algorithmically Undecidable Problems

Mathematical problems which can't be solved even given unlimited time and working space, for example:

- Hilbert's Entsheidungsproblem
- Halting Problem
- Hilbert's 10th Problem

1.1 Entsheidungsproblem (Decision Problem)

Is there an algorithm st when fed a statement in formal language of first-order arithmetic, determines in a finite number of steps whether or not the statement is provable from Peano's axioms for arithmetic using the usual rules of first-order logic. This could help solve things like the Goldbach Conjecture (every even integer strictly greater than two is the sum of two primes) using the following:

$$\forall k > 1 \exists p, q(2k = p + q \wedge prime(p) \wedge prime(q)) \tag{1}$$

More formally, the associated problem is given:

1. A set **S** whose elements are finite data structures of some kind (formulas of first-order logic)
2. A property **P** of elements of **S** (property of a formulat that it has a proof)

the associated problem it: *to find an algorithm which terminates with result 0 or 1 when fed an element $s \in S$ and yields the result 1 when fed s iff s has property P*

Algorithm: First issue is that there was no precise definition of an algorithm, just examples of what are algorithms. The features they included were that:

1. **Finite** description of the procedure in terms of elementary operations
2. **Deterministic**: next step uniquely determined if there is one
3. Can recognise when it terminates and what the result it (does not necessarily terminate)

Negative Solutions to Entsheidungsproblem: Given by Turing and Church in 1935/36. It is composed of:

1. Precise, mathematical definition of algorithm (*Turing: Turing Machines* and *Church: lambda-calculus*)
2. Regard algorithms as data on which algorithms can act and reduce the problem to.
3. Construct an algorithm encoding instances (A, D) of the Halting Problem as arithmetic statements $\Phi_{A,D}$ with the property that $\Phi_{A,D} \leftrightarrow A(D) \downarrow$

1.2 The Halting Problem

Decision problem with:

- Set **S** consisting of all pairs **(A, D)**, where A is an algorithm and D is a datum on which it is designed to operate
- Property **P** holds for **(A, D)** if algorithm A when applied to datum D eventually produces a result (that is, it halts - **A(D)** $\downarrow$

Turing and Church's work shows that the Halting Problem is **undecidable: there is no algorithm H st** $\forall$ **(A, D)** $\in$ **S**

$$H(A, D) = \begin{cases} 1 & if\, A(D) \downarrow \\ 0 & otherwise \end{cases} \tag{2}$$

Proofs that Halting Problem is Undecidable: If there were such an H, let C be the following algorithm

```
input A; compute H(A, A); if H(A, A) = 0 then return 1, else loop forever
```

Since H is total and by definition of H:

$$\forall A(C(A)\downarrow \leftrightarrow H(A,A)=0) \tag{3}$$

$$\forall A(H(A,A)=0 \leftrightarrow \neg A(A)\downarrow) \tag{4}$$

$$So\ \forall A(C(A)\downarrow \leftrightarrow \neg A(A)\downarrow) \tag{5}$$

Taking A to be the algorithm C:

$$C(C)\downarrow \leftrightarrow \neg C(C)\downarrow \tag{6}$$

which is a contradiction, therefore there exists no such algorithm H.

This doesn't quite work as we've taken A as a datum on which A is designed to operate - we've therefore assumed that we can pass a function to a function - not a first-order function then.

This can be used to negatively prove the Entsheidungsproblem by showing that any algorithm deciding provability of arithmetic statements could be used to decide the Halting Problem - therefore no such exists.

1.3 Hilbert's 10th Problem

Given an algorithm, which, when started with a Diophantine equation, determines in a finite number of operations whether or not there are natural numbers satisfying the equation.

Diophantine Equations: $p(x_1,\ldots,x_n)=q(x_1,\ldots,x_n)$ where p and q are polynomials in unknowns x_1, ..., x_n with coefficients from $\mathbb{N}=\{0,1,2,\ldots\}$

Posed in 1900, but proved undecidable by reduction from the Halting Problem by Y MatijaseviÄĐ, J Robinson, M Davis and H Putnam. The original proof used Turing machines, but a later, simpler proof used register machines (concept made by Minsky and Lambek).

2 Register Machines

Operate on $\mathbb{N}$ stored in finite (idealised) registers $(R_0, R_1, ..., R_n)$ each storing a natural numbers, using the following elementary operations:

1. **Add 1** to the contents of a register
2. Test whether the contents of a register is 0
3. **Subtract 1** from the contents of a register if it is non-zero
4. **Jumps**
5. **Conditionals** (if, then, else)

And a program consisting of a finite list of instructions of the form *label : body* where for i = 0, 1, 2, ..., the $(i+1)^{th}$ instruction has label L_i. The instruction body takes one of the three forms:

1. $\mathbf{R}^+ \to L'$

   Add 1 to the contents of register R and jump to instruction labelled L'
2. $\mathbf{R}^- \to L', L''$

   If the contents of R > 0, then subtract 1 from it and jump to L', else jump to L''
3. **HALT**

   Stop executing instructions

Configuration: $c=(\ell, r_0, \ldots, r_n)$ where ℓ is the current label and r_i is the current contents of R_i. R_i = x [in configuration c] means $c=(\ell, r_0,\ldots,r_n)$ with r_i=x.
Initial Configuration c_0 = (0, r_0, r_1, ..., r_n) where r_i = initial contents of register R_i

Computation: finite sequence of configurations $c_0, c_1, c_2, \ldots$ where c_0 is an initial configuration and each c in the sequence determines the next configuration in the sequence by carrying out the program instruction labelled L_ℓ

Halting: For a finite computation, the last configuration is a halting configuration, where the instruction is either HALT **(proper halt)** or another instruction involving going to an instruction that doesn't exist (**erroneous halt**).

- Erroneous halts can always be turned into proper halts by adding extra HALT instructions to the list with appropriate labels

2.1 Graphical Representation

One node in the graph for each instruction, with arcs representing jumps between instructions.

instruction	representation
$R^+ \to L$	$R^+ \longrightarrow [L]$
$R^- \to L, L'$	R^- $\to [L]$, $\twoheadrightarrow [L']$
HALT	HALT
L_0	START $\longrightarrow [L_0]$

2.2 Partial Functions

Relation between initial and final register contents which is defined by a register machine program is a partial function. This is because register machine computation is deterministic - in any non-halting configuration, the next configuration is uniquely determined by the program.

Partial function from set X to set Y is specified by any subset f $f \subseteq X \times Y$ (ordered pairs) satisfying:

$$(x, y) \in f \wedge (x, y') \in f \to y = y' \tag{7}$$

$\forall x \in X \; and \; y, y' \in Y$

For all x ∈ X, there is at most one y ∈ Y, with (x, y) ∈ f

2.2.1 Notation

- f(x) = y means (x, y) ∈ f
- f(x) $\downarrow$ means $\exists y \in Y(f(x) = y)$
- f(x) $\uparrow$ means $\neg\exists y \in Y(f(x) = y)$
- X $\rightharpoonup$ means set of all partial functions from X to Y
- X $\to$ Y means set of all total functions from X to Y

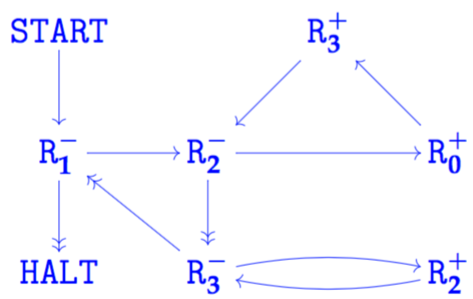
Partial function from set X to set Y is total if it satisfies f(x) $\downarrow \forall x \in X$

2.3 Computable Functions

$f \in \mathbb{N}^n \to \mathbb{N}$ is (register machine) computable if there is a register machine M with at least n+1 registers such that $(x_1, ..., x_n) \in \mathbb{N}$ and all y ∈ ℕ, with the computation of M starting with $R_0 = 0$, $R_1 = x_1$, ..., $R_n = x_n$ and all other registers set to 0, and halts with R_0=y iff $f(x_1, ..., x_n) = y$

Examples of Computable Functions

1. Multiplication



2. Projection: $p(x, y) \triangleq x$
3. Constant: $c(x) \triangleq n$
4. Truncated Subtraction: $x \dot{-} y \triangleq \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{if } y > x \end{cases}$
5. Integer Division: $x \operatorname{div} y \triangleq \begin{cases} \text{integer part of } x/y & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases}$
6. Integer Remainder: x mod $y \triangleq x \dot{-} y(x\ div\ y)$
7. Exponentiation base 2: $e(x) \triangleq 2^x$
8. Logarithm base 2: $\log_2(x) \triangleq \begin{cases} \text{greatest } y \text{ such that } 2^y \leq x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$
9. Sequential Composition: M1; M2

10. IF R=0 THEN M1 ELSE M2

11. WHILE R≠0 DO M

3 Coding Programs as Numbers

Turing / Church solutions for Entscheidungsproblem use the idea that the algorithms can be the data on which the algorithms act - therefore need to be able to code Register Machines as numbers. In general, these codings are called **GÃűdel Numberings**

Aim: Coding st RM program and initial contents of the registers can be coded into a number and that can be decoded back into the RM programs and initial contents of the registers.

3.1 Numerical Coding of Pairs

A possible numerical coding of pairs is as follows:

$$\text{For } x, y \in \mathbb{N}, \text{ define } \begin{cases} \langle\langle x, y\rangle\rangle & \triangleq 2^x(2y+1) \\ \langle x, y\rangle & \triangleq 2^x(2y+1) - 1 \end{cases} \tag{8}$$

- <-, -> gives a bijection (one-one correspondence) between $\mathbb{N} \times \mathbb{N} and \mathbb{N}$
- «-, -» gives a bijection between $\mathbb{N} \times \mathbb{N}$ and $\{n \in \mathbb{N} | n \neq 0\}$

3.1.1 Numerical Coding of Lists

For l ∈ list ℕ (set of all finite lists of natural numbers), define $\ulcorner l \urcorner \in \mathbb{N}$ by induction on the length of list l:

$$\begin{cases} \ulcorner [] \urcorner \triangleq 0 \\ \ulcorner x :: l \urcorner \triangleq \ll x, \ulcorner l \urcorner \gg = 2^x(2\ulcorner l \urcorner + 1) \end{cases} \tag{9}$$

Thus, $\ulcorner [x_1, x_2, ..., x_n] \urcorner = \ll x_1, \ll x_1, ..., \ll x_n, 0 \gg ... \gg \gg$

$$\boxed{0b\ulcorner[x_1, x_2, \ldots, x_n]\urcorner} = \boxed{1}\underbrace{\boxed{0\cdots0}}_{x_n\ 0s}\boxed{1}\underbrace{\boxed{0\cdots0}}_{x_{n-1}\ 0s}\cdots\boxed{1}\underbrace{\boxed{0\cdots0}}_{x_1\ 0s}$$

3.2 Numerical Coding of Programs

$$P \text{ is the RM program } \begin{bmatrix} \mathrm{L}_0 : \mathrm{body}_0 \\ \mathrm{L}_1 : \mathrm{body}_1 \\ \vdots \\ \mathrm{L}_n : \mathrm{body}_n \end{bmatrix}$$

then the numerical code is: $\ulcorner P \urcorner \triangleq \ulcorner [\ulcorner body_0 \urcorner, ..., \ulcorner body_n \urcorner] \urcorner$

$$\text{where } \ulcorner body \urcorner \text{ is defined by: } \begin{cases} \ulcorner R_i^+ \to L_j \urcorner \triangleq \ll 2i, j \gg \\ \ulcorner R_i^- \to L_j, L_k \urcorner \triangleq \ll 2i+1, < j, k > \gg \\ \ulcorner HALT \urcorner \triangleq 0 \end{cases}$$

Decoding

```
if x=0 then body(x) is HALT,
else (x>0 and) let x = <<y, z>> in
    if y=2i, then body(x) is Ri+ -> Lz
    else y=2i+1 let z = <j, k> in body(x) is Ri- -> Lj, Lk
```

So any e ∈ ℕ decodes to a unique program prog(e), called the program with index e:

$$\mathrm{prog}(e) \triangleq \begin{bmatrix} \mathrm{L}_0 : \mathrm{body}(x_0) \\ \vdots \\ \mathrm{L}_n : \mathrm{body}(x_n) \end{bmatrix} \text{ where } e = \ulcorner [x_0, \ldots, x_n] \urcorner \tag{10}$$

=> prog(0) is the program with an empty list of instructions, which by convention is a Register Machine that does nothing - halts immediately

4 Universal Register Machine

Universal Register Machine U carries out the following (starting with R_0=0, R_1=e (code of a program), R_2=a (code of a list of arguments) and all other registers zeroed:

1. Decode e as a Register program P
2. Decode a as a list of register values $a_1, ..., a_n$
3. Carry out the computation of the Register Machine program P starting with $R_0 = 0, R_1 = a_1, ..., R_n = a_n$ (and any other registers occurring in P set to 0)

4.1 Register Usage

- **R_1**: P - code of the RM to be simulated
- **R_2**: A - code of current register contents of simulated RM
- **R_3**: PC program counter - number of the current instruction
- **R_4**: N code of the current instruction body

- $\mathbf{R}_5$: C type of the current instruction body
- $\mathbf{R}_6$: R current value of the register to be incremented by current instruction
- $\mathbf{R}_7$, $\mathbf{R}_8$ and $\mathbf{R}_9$ are auxiliary registers

4.2 Structure of URM Program

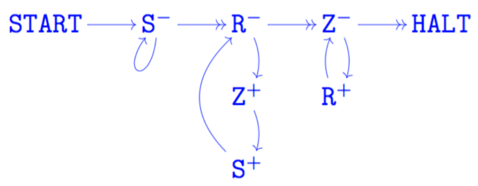
1. Copy PCth item of the list in P to N (halting if PC > length of list); goto 2
2. If N=0, then copy 0th item of list in A to R_0 and halt, else (decode N as $\ll y, z \gg$; C ::= y; N ::= z; goto 3
    - C = 2i and current instruction is $R_i^+ \rightarrow L_z$
    - OR
    - C = 2i + 1 and current instruction is $R_i^- \rightarrow L_j, L_k$ where z = <j, k>
3. Copy **i**th item of list in A to R; goto 4
4. Execute current instruction on R; update PC to next label; restore register values to A; goto 1

In order to do this, need to define Register Machines for manipulating codes of lists of numbers.

Prerequisite RMs

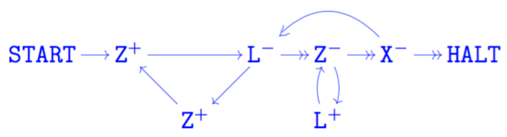
- **Copy Contents from R to S**

  $START \rightarrow S ::= R \rightarrow HALT$



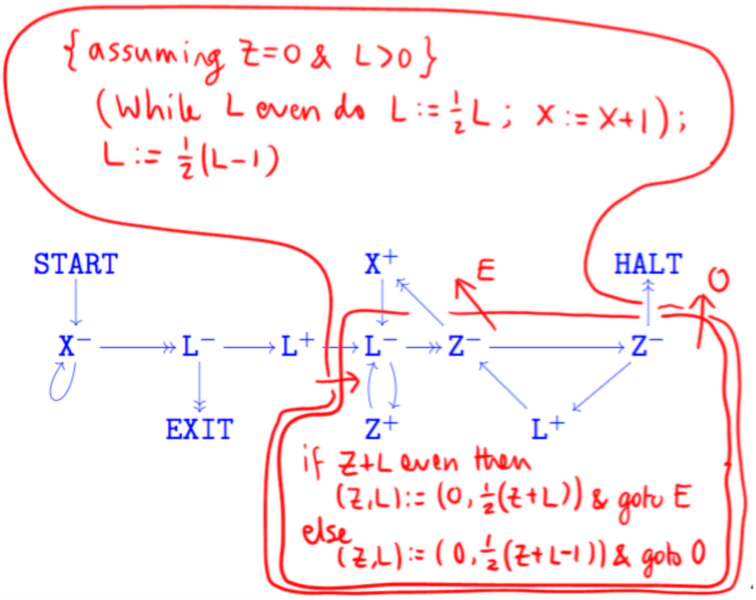
- **Push X to L**

  $START \rightarrow (X, L) ::= (0, X :: L)(2^X(2L+1)) \rightarrow HALT$

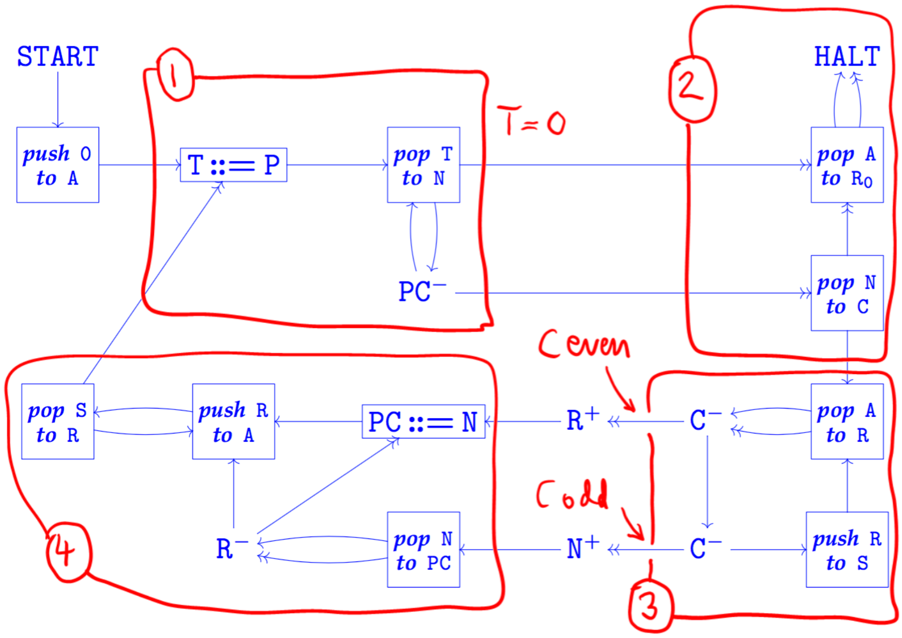


- **Pop L to X**:

```
if L = 0 then (X::= 0; goto EXIT) else
let L = <<x, l>> in (X ::= x, L ::= l; goto HALT)
```



4.3 Program for U



5 Halting Problem

A register machine H decides the Halting Problem if for all e, $a_1, ..., a_n \in \mathbb{N}$, starting H with R_0=0, R_1=e and R_2=$\ulcorner [a_1, ..., a_n] \urcorner$ and all other registers zeroed, the computation of H always halts with R_0 containing 0 or 1; moreover when the computation halts, R_0 iff the register machine program with index e eventually halts when started with R_0=0, $R_1 = a_1, ..., R_n = a_n$ and all other registered zeroed.

Theorem: No such register machine H can exist

5.1 Proof of Theorem

Assume ∃ a RM H that decides the Halting Problem and derive a contradiction as follows:

1. Let H' be obtained from H by replacing 'START →' with 'START → Z ::= R_1 → push Z to R_2 → where Z is a register not mentioned in H's program

2. Let C be obtained from H' by replacing each HALT (and each erroneous halt) by

$$\begin{tikzcd} {} \arrow[r] & R_0^- \arrow[r, bend left] \arrow[d] & R_0^+ \arrow[l, bend left] \\ & \text{HALT} & \end{tikzcd}$$

3. Let $c \in \mathbb{N}$ be the index of C's program
4. C started with R_1 = c eventually halts
5. $\iff$ H' started with R_1=c halts with R_0=0
6. $\iff$ H started with R_1=c, $R_2 = \ulcorner [c] \urcorner$ halts with $R_0 = 0$
7. $\iff$ prog(c) started with R_1=c does not halt
8. $\iff$ C started with R_1=c does not halt - this is a contradiction

5.2 Enumerating Computable Functions

For each e $\in \mathbb{N}$, let $\varphi_e \in \mathbb{N} \rightharpoonup \mathbb{N}$ be the unary partial function computed by the Register Machine with program prog(e). So, for all x, y $\in \mathbb{N}$:

$\varphi_e(x) = y$ holds iff computation of prog(e) starts with R_0=0, R_1=x and all other registers zeroed eventually halts with $R_0 = y$

Therefore, $e \mapsto \varphi_e$ defines an onto function from $\mathbb{N}$ to the collection of all computation partial functions from $\mathbb{N}$ to $\mathbb{N}$ - therefore this collection is countable. Therefore $\mathbb{N} \rightharpoonup \mathbb{N}$

Example Uncomputable Function: f $\in \mathbb{N} \rightharpoonup \mathbb{N}$ is the partial function with graph (x, 0) | $\varphi_x(x) \uparrow$

$$f(x) = \begin{cases} 0 & \text{if } \varphi_x(x) \uparrow \\ \text{undefined} & \text{if } \varphi_x(x) \downarrow \end{cases} \tag{11}$$

- f is not computable, as if it were, then f = φ_e for some $e \in \mathbb{N}$ and hence
- If $\varphi_e(e) \uparrow$, then f(e) = 0; so $\varphi_e(e)$=0 (since f = φ_e) hence $\varphi_e(e) \downarrow$
- If $\varphi_e(e) \downarrow$ then $f(e) \downarrow$ (since f = φ_e) so $\varphi_e(e) \uparrow$ (by definition of f)
- Therefore this is a contradiction, therefore f cannot be computable

5.3 Undecidable Sets of Numbers

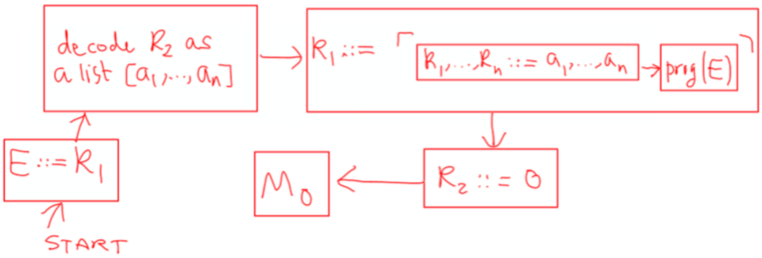
Set ($S \subseteq \mathbb{N}$) is RM decidable if

- Its characteristic function $\chi_s \in \mathbb{N} \to \mathbb{N}$ is a register machine computable function.
- $\equiv$ iff there is a RM M with the property that: (1) $\forall x \in \mathbb{N}$, M started with $R_0 = 0, R_1 = x$ and all other registers zeroed eventually halts with R_0 containing 1 or 0 and (2) $R_0 = 1$ on halting iff $x \in S$

Otherwise it is called undecidable. In order to prove undecidability, generally try to prove that that decidability of S would imply decidability of the Halting Problem.

Claim: $S_0 \triangleq e | \varphi_e(0) \downarrow$ is undecidable
Proof: Suppose M_0 is a RM computing χ_{S_0}. From M_0s program (using the same techniques as for constructing a universal Register Machine), we can construct a Register Machine to carry out.

```
let e = R1 and ⌜[a1,...,an]⌝ = R2 in
    R1 ::= ⌜(R1 ::= a1); ...; (Rn ::= an); prog(e)⌝
R2 ::= 0;
run M0
```



Therefore, by assumption on M_0, H decides the Halting Problem - this is a contradiction. Therefore no such M_0 exists, therefore χ_{S_0} is uncomputable => S_0 is undecidable

6 Turing Machines

Register Machines computation abstracts away particular, concrete abstractions of numbers and the associated elementary operations of increment / decrement / zero-test.

Turing Machines are more concrete: even numbers have to be represented in terms of a fixed finite alphabet of symbols and increment / decrement / zero-test programmed in terms of more elementary symbol manipulating operations

6.1 Features

1. Linear tape, unbounded to right, divided into cells containing a symbol from a finite alphabet of tape symbols. Only finitely many cells contain non-blank symbols
2. The machine is in one of a finite set of states
3. The tape symbol is being scanned by a tape head
4. The machine computes in discrete steps, each of which depends on the current state and the symbol being scanned by the tape head.

   Actions

   - Overwrite the current tape cell with a symbol
   - Move left or right one cell
   - Stay stationary
   - Change state
5. **Alphabet**
   - $\triangleright$: left endmarker symbol (start symbol)
   - $\llcorner\lrcorner$: blank symbol

More accurately specified by:

1. **Q**: finite set of machine states
2. Σ: finite set of tape symbols (disjoint from Q)
3. $s \in Q$, an initial state
4. $\delta \in (Q \times \Sigma \to (Q \cup \{acc, rej\}) \times \Sigma \times \{L, R, S\}$: transition function
   - Specifies for each state and symbol a next state (or accept or reject)
   - Symbol to overwrite the current symbol
   - And direction for the tape head to move (left, right or stationary)
   - $\forall q \in Q\ \exists q' \in Q \cup \{\text{accept, reject}\}$ with $\delta(q, \triangleright) = (q', \triangleright, R)$ (left endmarker is never overwritten and machine always moves to the right when scanning it)

6.2 Configuration

(q, w, u) where:

1. $q \in Q \cup \{acc, rej\}$ = current state
2. w = non-empty string (w = va) of tape symbols under ad to the left of the tape head, whose last element (a) is contents of cell under the tape head.
3. u = (possibly empty) string of tape symbols to the left of the tape head (upto some point beyond which all symbols are ␣)
4. Hence, wu $\in \Sigma^*$ represents the current tape contents
5. The initial configuration is (s, $\triangleright$, u)

6.3 Representing Transitions

Given a TM = (Q, Σ, s, *delta*), we write:

$$(q, w, u) \rightarrow_M (q', w', u') \tag{12}$$

to mean: (1) q $\neq$ acc, rej, (2) w = va (for some v and a) and:

1. Either $\delta(q, a) = (q', a', L)\,, w' = v,$ and $u' = a'u$
2. OR $\delta(q, a) = (q', a', S)\,, w' = va'$ and $u' = u$
3. OR $\delta(q, a) = (q', a', R)\,, u = a''u''$ is non-empty, $w' = va'a''$ and $u' = u''$
4. OR $\delta(q, a) = (q', a', R)\,, u = \varepsilon$ is empty, $w' = va'$ ␣and $u' = \epsilon$

6.4 Computation

Computation of a TM M is a (finite or infinite) sequence of configurations where (1) $c_0 = (s, \triangleright, u)$ is an initial configuration and (2) $c_i \rightarrow_M c_{i+1}$ holds for i $\in \mathbb{Z}^+$. The computation:

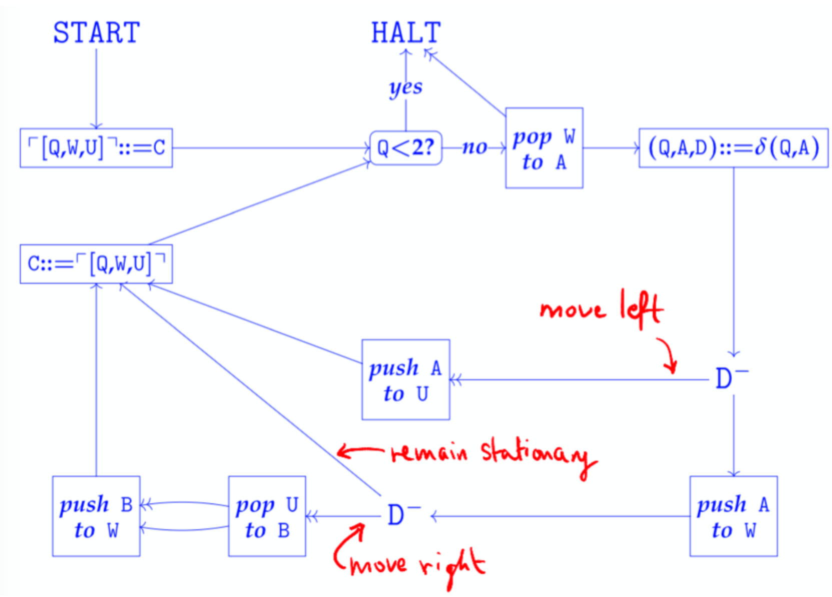
1. Does not halt if the sequence is infinite
2. Halts if the sequence is finite or if its last element is of the form (acc, w, u) or (rej, w, u)

6.5 Computation of a Turing Machine (M) can be implemented by a Register Machine

Proof

1. **Fix a numerical encoding of M's states, tape symbols, tape contents and configurations**
   (a) Identify states and type symbols with specific numbers: (1) acc=0, rej=1, Q=2, 3, ..., n and (2) ␣=0, $\triangleright$=1, Σ=0,1, ..., m
   (b) Code configurations c = (q, w, u) is given by:
   $\ulcorner c \urcorner = \ulcorner [q, \ulcorner [a_n, ..., a_1] \urcorner, \ulcorner [b_1, ..., b_m] \urcorner] \urcorner$ where w = $a_1 ... a_n$ (n > 0) and u = $b_1 ... b_m$ (m $\geq$ 0)
   We reverse the w to make it easier to use our Register Machine programs for list manipulation
2. **Implement M's transition function (finite table) using RM instructions on codes**
   (a) We use registers to represent (1) Q - current state, (2) A - current tape symbol, (3) D - current direction of the tape head (L = 0, R = 1, S = 2)
   (b) Can turn the finite table of (argument, result)-pairs specifying δ into a Register Machine program $\rightarrow (Q, A, D) ::= \delta(Q, A) \rightarrow$ such that starting the program with Q=q, A=a, D=d and all other registers zero, it halts with Q=q', A=a', D=d' where (q', a', d') = $\delta(q, a)$
3. **Implement a Register Machine to repeatedly carry out $\rightarrow_M$**
   (a) Uses registers to store: (1) C - code of current configuration, (2) W - code of tape symbols at and left of tape head, (3) U - code of tape symbols right of tape head

(b) Starting with C containing the code of an initial configuration (and all other registers zeroed), the RM halts iff M halts and in that case C holds the code of the final configuration



6.6 Computation of a Register Machine can be implemented by a Turing Machine

Can be reasonably easily proven, just need to show how to carry out the action of each type of the RM.

6.6.1 Tape encoding of list of numbers

A tape over $\Sigma = \triangleright, \sqcup, 0, 1$ codes a list of numbers if precisely two cells contain 0 and the only cells containing 1 occur between these

$$\triangleright \sqcup \cdots \sqcup 0 \underbrace{1\cdots1}_{n_1} \sqcup \underbrace{1\cdots1}_{n_2} \sqcup \underbrace{\cdots}_{\cdots} \sqcup \underbrace{1\cdots1}_{n_k} 0 \underbrace{\sqcup\cdots}_{\text{all } \sqcup\text{'s}}$$

corresponds to the list $[n_1, n_2, ..., n_k]$

7 Notions of Computability

Church defined computability using λ-calculus and Turing used Turing machines - though Turing showed that the two approaches determine the same class of computable functions.

Church-Turing Thesis: Every algorithm can be realised as a Turing machine

Further evidenced by:

- Goedel and Kleene (1936): partial recursive functions
- Post (1943) and Markov (1951): canonical systems for generating the theorems of a formal system
- Lambek and Minsky (1961): register machines

7.1 Turing Computability

f $\in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is Turing computable iff $\exists$ a Turing machine M with the following property:

1. Starting M from initial state with tape head on the left endmarker of a tape coding $[0, x_1, ..., x_n]$, M halts iff f$(x_1, ..., x_n)\downarrow$ and in that case the final tape codes a list whose first element is y where f$(x_1, ..., x_n) =$ y

7.2 Aim

A more abstract, machine-independent description of the collection of computable partial functions than provided by register / Turing machines. They form the smallest collection of partial functions containing some basic functions and closed under some fundamental operations for forming new functions from old - composition, primitive recursion and minimisation.

7.3 Functions

Kleene Equivalence of possibly-undefined expressions: Either both LHS and RHS are undefined or they are both defined and equal.

1. **Projection**: $proj_i^n \in \mathbb{N}^n \to \mathbb{N}$

   $proj_i^n(x_1, ..., x_n) \triangleq x_i$

   START $\to [\mathrm{R}_0 ::= \mathrm{R}_i] \to$ HALT

2. **Constant with value 0**: $\mathrm{zero}^n \in \mathbb{N}^n \to \mathbb{N}$

   $z\,\mathrm{ero}^n\,(x_1, \ldots, x_n) \triangleq 0$

   START $\to$ HALT

3. **Successor**: $\mathrm{succ} \in \mathbb{N} \to \mathbb{N}$

   $\mathrm{succ}(x) \triangleq x + 1$

   START $\to \mathrm{R}_1^+ \to [\mathrm{R}_0 ::= \mathrm{R}_1] \to$ HALT

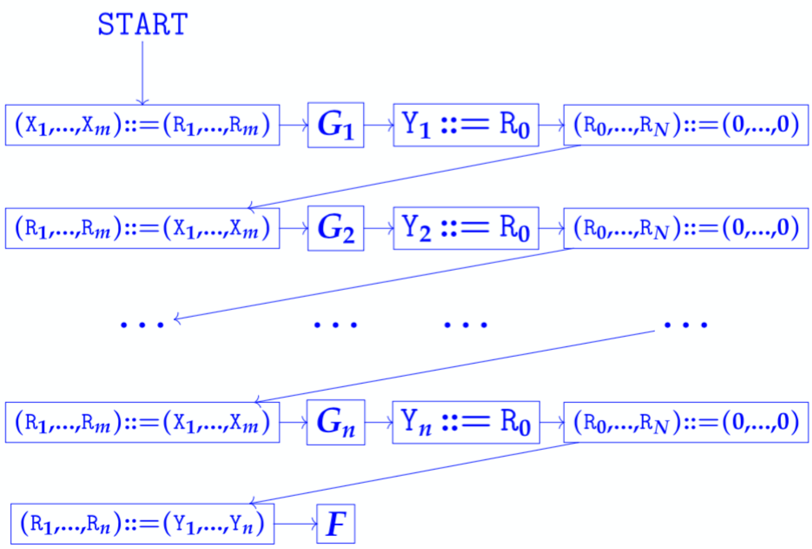
4. **Composition** of $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ with $g_1, ..., g_n \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ is $f \circ [g_1, \ldots, g_n] \in \mathbb{N}^m \to \mathbb{N}$ satisfying for all $x_1, \ldots, x_m \in \mathbb{N}$:

   $f \circ [g_1, \ldots, g_n]\,(x_1, \ldots, x_m) \equiv f\,(g_1\,(x_1, \ldots, x_m)\,, \ldots, g_n\,(x_1, \ldots, x_m))$

   Therefore, $f \circ [g_1, \ldots, g_n]\,(x_1, \ldots, x_m) = z$ iff $\exists\ y_1, \ldots, y_n$ with $\boldsymbol{g}_i\,(\boldsymbol{x_1}, \ldots, \boldsymbol{x}_m)$ (for i = 1, ..., n) and $f\,(y_1, \ldots, y_n) = z$

   Idea is that $f \circ [\boldsymbol{g}_1, \ldots, \boldsymbol{g}_n]$ is computable if f and $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_n$ are

   Proof: *Given RM programs* $\begin{cases} F \\ G_i \end{cases}$ *computing* $\begin{cases} f\,(y_1, \ldots, y_n) \\ g_i\,(x_1, \ldots, x_m) \end{cases}$ *in* R_0 *starting with* $\begin{cases} \mathrm{R}_1, \ldots, \mathrm{R}_n \\ \mathrm{R}_1, \ldots, \mathrm{R}_m \end{cases}$ *set to* $\begin{cases} y_1, \ldots, y_n \\ x_1, \ldots, x_m \end{cases}$, *then we can define a RM program computing the composition* $(f \circ [g_1, \ldots, g_n]\,(x_1, \ldots, x_m))$ *starting with* $\mathrm{R}_1, \ldots, \mathrm{R}_m$ *set to* $x_1, \ldots, x_m$



8 Primitive Recursion

Partial Function f is primitive recursive ($\in PRIM$) if it can be built in finitely many steps from the basic functions by use of the operations of composition and primitive recursion. PRIM is the smallest set (with respect to subset including) of partial functions containing the basic functions and closed under the operations of composition and primitive recursion.

Theorem: Given $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $g \in \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$, $\exists! h \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ which satisfies $\forall \vec{x} \in \mathbb{N}^n$ and $x \in \mathbb{N}$:

$$\begin{cases} h(\vec{x}, 0) & \equiv f(\vec{x}) \\ h(\vec{x}, x+1) & \equiv g(\vec{x}, x, h(\vec{x}, x)) \end{cases} \tag{13}$$

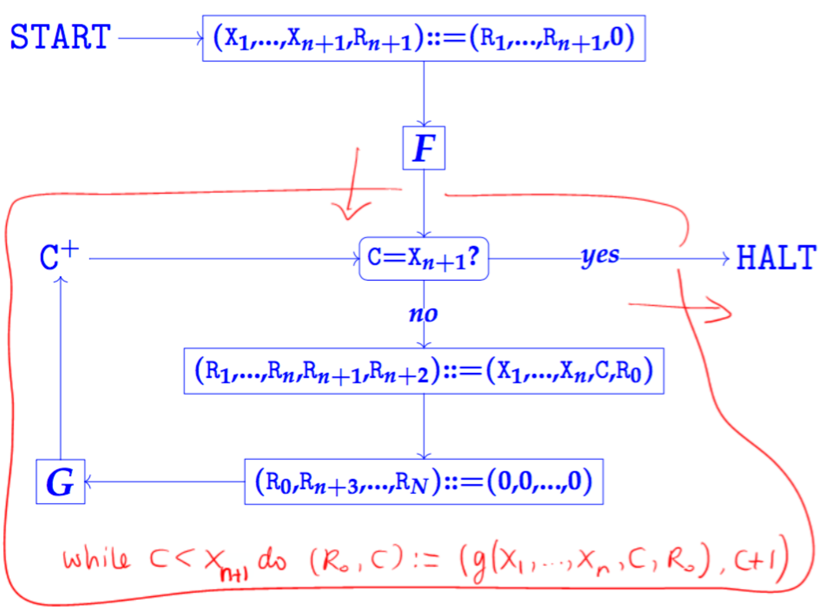
This h is written as $\rho^n(f, g)$ and it is called the partial function defined by primitive recursion from f and g.

Theorem: All functions f $\in PRIM$ are computable
Proof:

- Basic functions are computable, composition preserves computability as already proved, therefore must show:

$$\rho^n(f, g) \in \mathbb{N}^{n+1} \to \mathbb{N}\ computable\ if\ f \in \mathbb{N}^n \to \mathbb{N}\ and\ g \in \mathbb{N}^{n+2} \to \mathbb{N}are \tag{14}$$

- Suppose f and g are computed by RM programs F and G, then this RM computes $\rho^n(f, g)$



All functions f $\in PRIM$ are all total as:

1. All basic functions are total
2. If f, $\boldsymbol{g_1}, \ldots, \boldsymbol{g_n}$ are total, then so is $f \circ (\boldsymbol{g}_1, \ldots, \boldsymbol{g}_n)$
3. If f and g are total, then so is $\rho^n(f, g)$

8.1 Examples of Primitive Recursive Functions

Addition

$$add \in \mathbb{N}^2 \to \mathbb{N} = \begin{cases} add\,(x_1, 0) & \equiv x_1 \\ add\,(x_1, x+1) & \equiv add\,(x_1, x) + 1 \end{cases} \tag{15}$$

Therefore $add = \rho^1(f, g)$ where $\begin{cases} f\,(x_1) & \triangleq x_1 \\ g\,(x_1, x_2, x_3) & \triangleq x_3 + 1 \end{cases}$
f $= proj_1^1$ and g $=$ succ $\circ$ proj$_3^3$ so add can be made from basic functions and so add $\in PRIM$

Predecessor

$$\text{pred } \in \mathbb{N} \to \mathbb{N} = \begin{cases} \text{pred}(0) & \equiv 0 \\ \text{pred}(x+1) & \equiv x \end{cases} \tag{16}$$

Therefore, $\text{pred} = \rho^0(f, g)$ where $\begin{cases} f() & \triangleq 0 \\ g(x_1, x_2) & \triangleq x_1 \end{cases} = \rho^0\left(\text{zero}^0, \text{proj}_1^2\right)$

Multiplication

$$mult \in \mathbb{N}^2 \to \mathbb{N} = \begin{cases} \text{mult}(x_1, 0) & \equiv 0 \\ \text{mult}(x_1, x+1) & \equiv \text{mult}(x_1, x) + x_1 \end{cases} \tag{17}$$

$$mult = \rho^1(zero^1, add \circ (proj_3^3, proj_1^3)) \tag{18}$$

Therefore, since mult can be made from composition and primitive recursion (since add can be)

9 Minimisation

Given a partial function $f \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$, define $\mu^n f \in \mathbb{N}^n \to \mathbb{N}$ by:

$$\mu^n f(\vec{x}) \triangleq$$

least x such that $f(\vec{x}, x) = 0$ and for each i=0, ..., x-1, $f(\vec{x}, i)$ is defined and > 0 OR

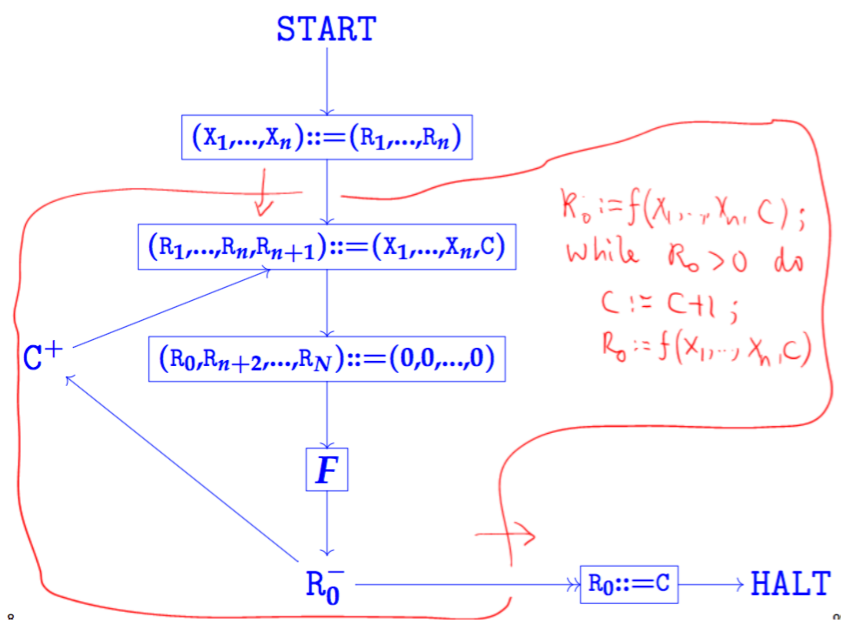
$$\mu^n f = \{(\vec{x}, x) \in \mathbb{N}^{n+1} | \exists y_0, \ldots, y_x \left(\bigwedge_{i=0}^{x} f(\vec{x}, i) = y_i\right) \wedge \left(\bigwedge_{i=0}^{x-1} y_i > 0\right) \wedge y_x = 0\} \tag{19}$$

10 Partial Recursion

Partial Function f is partial recursive ($\in PR$) if it can be built in finitely many steps from the basic functions by use of the operations of composition, primitive recursion **and minimisation**. PR is the smallest set (with respect to subset including) of partial functions containing the basic functions and closed under the operations of composition, primitive recursion **and minimisation**.

The members of PR that are total are called recursive functions - their are recursive functions that are not primitive recursive - eg Fibonacci Numbers

Theorem: All functions f $\in PR$ are computable
Proof: Suppose f is computer by RM program F. Then the following RM computes $\mu^n f$



Theorem: Every computable partial function is partial recursive
Proof:

- Let $f \in \mathbb{N}^n \to \mathbb{N}$ be computed by RM M with N $\geq$ n registers.
- construct primitive recursive functions lab, val_0, $\text{next}_M \in \mathbb{N} \to \mathbb{N}$, satisfying:
  1. $lab(\ulcorner [l, r_0, ..., r_N] \urcorner) = l$
  2. $val_0(\ulcorner [l, r_0, ..., r_N] \urcorner) = r_0$
  3. $next_M(\ulcorner [l, r_0, ..., r_N] \urcorner) =$ code of M's next configuration
- Writing $\vec{x}$ for $x_1, ..., x_n$, let $\text{config}_M(\vec{x}, t)$ be the code of M's configuration after t steps, starting with initial register values: $\text{R}_0 = 0, \text{R}_1 = x_1, \ldots, \text{R}_n = x_n, \text{R}_{n+1} = 0, \ldots, \text{R}_N = 0$. This is in PRIM because:

$$\begin{cases} \text{config}_M(\vec{x}, 0) & = \ulcorner [0, 0, \vec{x}, \vec{0}] \urcorner \\ \text{config}_M(\vec{x}, t+1) & = next_M\left(\text{config}_M(\vec{x}, t)\right) \end{cases}$$

- Assume M has a single HALT as last instruction. Let $halt_M(\vec{x})$ be the number of steps M takes to halt, when started with initial register values $\vec{x}$.
- Satisfies $\text{halt}_M(\vec{x}) \equiv$ least t st $I - \text{lab}\left(\text{config}_M(\vec{x}, t)\right) = 0$ and hence in PR (because lab, $config_M, I - () \in PRIM$.
- Therefore, $f(\vec{x}) \equiv val_0\left(\text{config}_M\left(\vec{x}, halt_M(\vec{x})\right)\right)$ and f $\in PR$

11 Ackermann's Function

$$ack \in \mathbb{N}^2 \to \mathbb{N}$$

$$ack(0, x_2) = x_2 + 1$$
$$ack(x_1 + 1, 0) = ack(x_1, 1)$$
$$ack(x_1 + 1, x_2 + 1) = ack(x_1, ack(x_1 + 1, x_2))$$

ack is (1) computable and therefore is recursive and (2) grows faster than any primitive recursive function $f \in \mathbb{N}^2 \to \mathbb{N}$:
$\exists N_F \forall x_1, x_2 \; > N_f(f(x_1, x_2) < ack(x_1, x_2))$
Hence ack is not primitive recursive

12 Lambda Calculus

Function Definition Notation

1. Named: let f be the function f(x) = x^2 + x + 1
2. Anonymous: f: x $\mapsto$ x^2 + x + 1
3. Lambda Notation: $\lambda x.x^2 + x + 1$

λ-Terms are built from a given, countable collection of variables (x, y, z) by operations for forming λ-terms:

1. λ-abstraction (λx.M) where x is a variable and M is a λ-term
2. Application which is left-associative (MM') where M and M are λ-terms

12.1 Notational Conventions

- $(\lambda x_1 x_2 \ldots x_n \cdot M) \equiv (\lambda x_1 \cdot (\lambda x_2 \ldots (\lambda x_n \cdot M) \ldots))$
- $(M_1 M_2 \ldots M_n) \equiv (\ldots \cdot (M_1 M_2) \ldots M_n)$
- Drop outermost parentheses and those enclosing the body of a λ-abstraction - eg: $(\lambda x.(x(\lambda y.(yx)))) \equiv \lambda x.x(\lambda y.yx)$
- x # M means that the variable x does not occur anywhere in the λ-term M

12.2 Free and Bound Variables

In $\lambda x.M$, x is the **bound variable** and M is the body of the λ-abstraction. Occurrence of x in a λ-term M is:

1. Binding if in between λ and .
2. Bound if in the body of a binding occurrence of x
3. Free if neither binding nor bound
4. Sets of free and bound variables:
   - FV(x) = x
   - FV($\lambda x.M$) = FV(M) - x
   - FV(MN) = FV(M) $\cup$ FV(N)
   - FV(M) = $\implies M$ is a closed term, or combinator
   - BV(x) =
   - BV($\lambda x.M$) = BV(M) $\cup$ x
   - BV(MN) = BV(M) $\cup$ BV(N)

12.3 α-Equivalence (M $=_\alpha$ M')

is the equivalence relation (reflexive, symmetric and transitive) inductively generated by the rules:

1. $\overline{x =_\alpha x}$
2. $\frac{z \# (MN) \quad M\{z/x\} =_\alpha N\{z/y\}}{\lambda x.M =_\alpha \lambda y.N}$
3. $\frac{M =_\alpha M' \quad N =_\alpha N'}{MN =_\alpha M'N'}$

where $M\{z/x\}$ is M with all occurrences of x replaced by z.

This effectively says that the name of the bound variable is immaterial and therefore if M' = Mx'/x is the result of taking M and changing all occurrences of x to some variable x' # M then $\lambda x.M$ and $\lambda x'.M'$ both represent the same function

12.4 β-Reduction

$\lambda x.M$ represented the function f st f(x) = M $\forall x$. Regard $\lambda x.M$ as a function on λ-terms via substitution: map each N to M[N/x] (result of substituting N for free x in M).

Substitution N[M/x]: Result of replacing all free occurrences of x in N with M, avoiding the capture of free variables in M by λ-binders in N

1. $x[M/x] = M$
2. $y[M/x] = y \quad$ if $y \neq x$
   - y does not occur in M and
   - y $\neq$ x
   - This makes substitution capture-avoiding
   - If $x \neq y$: $(\lambda y.x)[y/x] \neq \lambda y.y$
3. $(\lambda y.N)[M/x] = \lambda y.N[M/x] \quad$ if $y \# (Mx)$
4. $(N_1 N_2)[M/x] = N_1[M/x] N_2[M/x]$

N $\mapsto$ N[M/x] induces a totally defined function from the set of α-equivalence classes of λ-terms to itself

Natural notion of computation for λ-terms is given by stepping from: (1) β-redex *($\lambda x.M$)N* to (2) β-reduct *M[N/x]*

12.4.1 β-Reduction Rules

1. $\overline{(\lambda x.M)N \to M[N/x]}$
2. $\frac{M \to M'}{\lambda x \cdot M \to \lambda x.M'}$
3. $\frac{M \to M'}{MN \to M'N}$
4. $\frac{M \to M'}{NM \to NM'}$
5. $\frac{N =_\alpha M \quad M \to M' \quad M' =_\alpha N'}{N \to N'}$

12.4.2 β-Conversion (M $=_\beta$ N)

Holds if N can be obtained from M by performing zero or more steps of α-equivalence, β-reduction or β-expansion

Rules

1. $\frac{M =_\alpha M'}{M =_\beta M'}$
2. $\frac{M \to M'}{M =_\beta M'}$
3. $\frac{M =_\beta M'}{M' =_\beta M}$
4. $\frac{M =_\beta M' \quad M' =_\beta M"}{M =_\beta M"}$
5. $\frac{M =_\beta M'}{\lambda x \cdot M =_\beta \lambda x.M'}$
6. $\frac{M =_\beta M' \quad N =_\beta N'}{MN =_\beta M'N'}$

12.4.3 Church-Rosser Theorem

Theorem: $\twoheadrightarrow$ is confluent ie $M_1 \twoheadleftarrow M \twoheadrightarrow M_2 \implies \exists M' \; st \; M_1 \twoheadrightarrow M^1 \twoheadleftarrow M_2$
Corollary: $M_1 =_\beta M_2 \iff \exists M (M_1 \twoheadrightarrow M \twoheadleftarrow M_2)$
Proof:

- $=_\beta$ satisfies the rules generating $\twoheadrightarrow$, so $M \twoheadrightarrow M' \implies M =_\beta M'$
- Therefore $M_1 \twoheadrightarrow M \twoheadleftarrow M_2 \implies M_1 =_\beta M =_\beta M_2 \implies M_1 =_\beta M_2$

Conversely, relation $\{(M_1, M_2) \,|\, \exists M \, (M_1 \twoheadrightarrow M \twoheadleftarrow M_2)\}$ satisfies the rules generating $=_\beta$: the only difficult case is the closure of the relation under transitivity and for this, we use Church-Rosser Theorem:

$$\begin{tikzcd}
M_1 \arrow[r, two heads] & M \arrow[dr, two heads] & M_2 \arrow[l, two heads] \arrow[r, two heads] \arrow[d, phantom, "\text{C-R}"] & M' \arrow[dl, two heads] & M_3 \arrow[l, two heads] \\
 & & M_2' & &
\end{tikzcd}$$

Therefore, $M_1 =_\beta M_2 \implies \exists M (M_1 \twoheadrightarrow M' \twoheadleftarrow M_2)$

12.5 β-Normal Forms

λ-term is in β-normal form if it contains no β-redexes (no sub-terms of the form $(\lambda x.M)M'$).
M has β-nf N if M $=_\beta$N with N being a β-nf.
β-nf of M is unique upto α-equivalence if it exists (if $N_1 =_\beta N_2$ with N_1 and N_2 both being β-nfs then $N_1 =_\alpha N_2$)

Important to note that some λ terms have no β-nf and that a term can possess both a β-nf and infinite chains of reduction from it.

12.5.1 Normal-order Reduction

Deterministic strategy for reducing λ-terms: reduce the left-most, outer-most redex first where:

- left-most means reduce M before N in MN
- outer-most means reduce $(\lambda x.M)$N rather than either of M or N

This is guaranteed to reduce to the β-nf if it possesses one

12.6 Lambda-Definable Functions

In order to relate λ-calculus to register and Turing Machine computation, or to compute partial recursive functions, we need to encode numbers, pairs and lists as λ-terms.

12.6.1 Church's Numerals

$$\begin{aligned}
\underline{0} &\triangleq \lambda f\,x.x \\
\underline{1} &\triangleq \lambda f\,x.f\,x \\
\underline{2} &\triangleq \lambda f\,x.f(f\,x) \\
&\vdots \\
\underline{n} &\triangleq \lambda f\,x.\underbrace{f(\cdots(f}_{n \text{ times}}\,x)\cdots)
\end{aligned}$$

Notation: $\begin{cases} M^0 N & \triangleq N \\ M^1 N & \triangleq MN \\ M^{n+1} N & \triangleq M\,(M^n N) \end{cases}$ so we can write $\underline{n}$ as $\lambda f x.f^n x$ and we have $\underline{n}\boldsymbol{MN} =_\beta \boldsymbol{M}^n\boldsymbol{N}$

12.6.2 Definition

$f \in \mathbb{N}^2 \rightharpoonup \mathbb{N}$ is λ-definable if there is a closed λ-term F that represents it: $\forall (x_1, ..., x_n) \in \mathbb{N}^n$ and $y \in \mathbb{N}$

1. $f\,(x_1, \ldots, x_n) = y \implies F\vec{x_1}...\vec{x_n} =_\beta \vec{y}$
2. $f\,(x_1, \ldots, x_n) \uparrow \implies F\vec{x_1}...\vec{x_n}$ has no β-nf

   This condition can make it tricky to find a λ-term representing a non-total function

12.6.3 Computability

Partial function is computable iff it is λ-definable: gets split into:

1. Every partial recursive function is λ-definable
2. λ-definable functions are Register Machine computable

12.6.4 Showing elements of PRIM are λ-definable

1. $\text{proj}_i^n \in \mathbb{N}^n \to \mathbb{N}$ is represented by $\lambda x_1 \ldots x_n \cdot x_i$
2. zero $^n \in \mathbb{N}^n \to \mathbb{N}$ is represented by $\lambda x_1 \ldots x_n \cdot \underline{0}$
3. succ $\in \mathbb{N} \to \mathbb{N}$ is represented by $\lambda x_1 f x.f\,(x_1 f x)$ OR $\lambda x_1 f x.x_1 f(fx)$

12.7 Representations

12.7.1 Representing Composition

If total function f $\in \mathbb{N}^n \to \mathbb{N}$ is represented by F and total functions $g_1, \ldots, g_n \in \mathbb{N}^m \to \mathbb{N}$ are represented by $G_1, \ldots, G_n$, then the composition $(f \circ (g_1, \ldots, g_n) \in \mathbb{N}^m \to \mathbb{N})$ is represented by $\lambda x_1 \ldots x_m.F\,(G_1 x_1 \ldots x_m) \ldots (G_n x_1 \ldots x_m)$

However, this does not necessarily work for partial functions

12.7.2 Representing Primitive Recursion

If $f \in \mathbb{N}^n \to \mathbb{N}$ is represented by λ-term F and $g \in \mathbb{N}^{n+2} \to \mathbb{N}$ is represented by λ-term G, want to show λ-definability of the unique $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ that satisfies h = $h = \Phi_{f,g}(h)$, where $\mathbf{\Phi}_{f,g} \in \left(\mathbb{N}^{n+1} \to \mathbb{N}\right) \to \left(\mathbb{N}^{n+1} \to \mathbb{N}\right)$ is given by:

$$\begin{aligned} h(\vec{a}, a) = \text{if } a = 0 \text{ then } f(\vec{a}) \\ \text{else } g(\vec{a}, a-1, h(\vec{a}, a-1)) \end{aligned} \tag{20}$$

OR

$$\begin{cases} h(\vec{a}, 0) & = f(\vec{a}) \\ h(\vec{a}, a+1) & = g(\vec{a}, a, h(\vec{a}, a)) \end{cases} \tag{21}$$

Strategy:

1. Show that $\Phi_{f,g}$ is λ-definable
2. Show that we can solve fixed point equations X = MX up to β-conversion in the λ-calculus

12.7.3 Representing Booleans

- True $\triangleq \lambda xy.x$
- False $\triangleq \lambda xy.y$
- If $\triangleq \lambda f$ xy. f xy

12.7.4 Representing Test-for-Zero

$$Eq_{\ 0} \triangleq \lambda x.x(\lambda y.False)\ True \tag{22}$$

12.7.5 Representing Ordered Pairs

$$Pair \triangleq \lambda xyf.fxy$$
$$Fst \triangleq \lambda f.f\ True$$
$$Snd \triangleq \lambda f.f\ False$$

12.7.6 Representing Predecessor

Has to satisfy:

1. Pred $n \vec{+} 1 =_\beta \vec{n}$
2. Pred $\vec{0} =_\beta \vec{0}$

$$Pred \triangleq \lambda yfx.Snd(y(Gf)(Pair\ x\ x)) \tag{23}$$

where

$$G \triangleq \lambda fp.Pair(f(Fst\ p))(Fst\ p) \tag{24}$$

12.7.7 Representing Primitive Recursion

$f \in \mathbb{N}^n \to \mathbb{N}$ represented by a λ-term F and $g \in \mathbb{N}^{n+2} \to \mathbb{N}$ is represented by λ-term G, we want to show λ-definability of the unique $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ that satisfies $h = \Phi_{f,g}(h)$ where $\Phi_{f,g} \in \left(\mathbb{N}^{n+1} \to \mathbb{N}\right) \to \left(\mathbb{N}^{n+1} \to \mathbb{N}\right)$ is given by:

$$\begin{aligned} \Phi_{f,g}(h)(\vec{a}, a) \triangleq \text{ if } a = 0 \text{ then } f(\vec{a}) \\ \text{else } g(\vec{a}, a-1, h(\vec{a}, a-1)) \end{aligned} \tag{25}$$

Strategy

1. Show that $\Phi_{f,g}$ is λ-definable

$$Y(\lambda z\overrightarrow{x}x.If(Eq_0\ x)(F\overrightarrow{x})(G\overrightarrow{x}(Pred\ x)(z\overrightarrow{x}(Pred\ x))))$$

2. Show that we can solve fixed point equations (X = MX) up-to β-conversion in the λ-calculus

Every $f \in$ PRIM is λ-definable: in order to expand this to all recursive functions, we have to consider how to represent the minimisation.

12.8 Examples

1. **Addition is λ-definable because represented by**: $P \triangleq \lambda x_1 x_2, \lambda fx \cdot x_1 f\,(x_2 fx)$

   $P\underline{mn} =_\beta \lambda fx.\underline{m}f(\underline{n}fx)$

   $P\underline{mn} =_\beta \lambda fx.mf\,(f^n x)$

   $P\underline{mn} =_\beta \lambda fx \cdot f^m\,(f^n x)$

   $= \lambda fx.f^{m+n}x$ (this is provable using induction on n)

   $m \vec{+} n$

12.9 Curry's Fixed Point Combinator Y

Name	**Naive Set Theory**	**λ calculus**
Russell Set	$R \triangleq \{x \mid \neg(x \in x)\}$	$not \triangleq \lambda b.If\ b\ False\ else\ True$ $R \triangleq \lambda x.not(xx)$
Russell's Paradox	$R \in R \iff \neg(R \in R)$	$RR =_\beta not(RR)$ $Ynot =_\beta RR = (\lambda x.not(xx))(\lambda x.not(xx))$ $Yf = (\lambda x.f(xx))(\lambda x.f(xx))$ $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

$$Y \triangleq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \tag{26}$$

This satisfied $YM \to (\lambda x.M(xx))(\lambda x.M(xx))$
$\to M((\lambda x.M(xx))(\lambda x.M(xx)))$

Hence, $YM \to M((\lambda x.M(xx))(\lambda x.M(xx))) \leftarrow M(YM)$

Therefore, for all $\lambda - terms$ M: $\mathbf{YM} =_\beta \boldsymbol{M}(\mathbf{Y}\boldsymbol{M})$

12.10 Turing's Fixed Point Combinator

$$A \triangleq \lambda xy.y(xxy)$$

$$\Theta \triangleq AA$$

$$\Theta M = AAM = (\lambda xy.y(xxy))AM \twoheadrightarrow M(AAM) = M(\Theta M)$$

12.11 Representing Minimisation

$$\mu^n f(\overrightarrow{x}) = g(\overrightarrow{x}, 0) \tag{27}$$

$$g(\overrightarrow{x}, x) = if\ f(\overrightarrow{x}, x) = 0\ then\ x\ else\ g(\overrightarrow{x}, x+1) \tag{28}$$

$\mu^n f$ can be expressed in terms of a fixed point equation: $\mu^n f(\vec{x}) \equiv g(\vec{x}, 0)$ where $g = \Psi_f(g)$ with $\Psi_f \in \left(\mathbb{N}^{n+1} \rightharpoonup \mathbb{N}\right) \to \left(\mathbb{N}^{n+1} \rightharpoonup \mathbb{N}\right)$ defined by:

$$\Psi_f(g)(\vec{x}, x) \equiv \text{ if } f(\vec{x}, x) = 0 \text{ then } x \text{ else } g(\vec{x}, x+1) \tag{29}$$

If a function f has a totally defined $\mu^n f$, $\forall \overrightarrow{a} \in \mathbb{N}^n, \mu^n f(\overrightarrow{a}) = g(\overrightarrow{a}, 0)$, with $g = \Psi_f(g)$ and $\Psi_f(g)(\overrightarrow{a}, a) = if(f(\overrightarrow{a}, a) = 0)\ then\ a\ else\ g(\overrightarrow{a}, a+1)$.
Hence, if f is represented by λ-term F, then $\mu^n f =$

$$\lambda \vec{x}.Y\ (\lambda z \vec{x} x.\,\text{If}\ (Eq_0(F\vec{x}x))\ x(z\vec{x}(\text{Succ}\,x))\vec{x}\underline{0} \tag{30}$$

Hence, every recursive function is λ-definable as they can be expressed in standard form as: $f = g \circ (\mu^n h)$ for some $g, h \in$ PRIM.

12.12 Computability

Theorem: A partial function is computable iff it is λ-definable. Prove this by showing we can:

1. Code λ-terms as numbers - ensuring that operations for constructing and deconstructing terms are RM computable
   (a) Fix an enumeration $x_0, x_1, ...$ of the set of variables
   (b) $\ulcorner x_i \urcorner = \ulcorner [0, i] \urcorner$
   (c) $\ulcorner \lambda x_i M \urcorner = \ulcorner [1, i, \ulcorner M \urcorner] \urcorner$
   (d) $\ulcorner MN \urcorner = \ulcorner [2, \ulcorner M \urcorner, \ulcorner N \urcorner] \urcorner$
2. Write a RM interpreter for β-reduction