Computation Theory

Simon

April 5, 2025

Contents

1 Introduction **3**
1.1 Hilbert's Entscheidungsproblem . . . 3
1.2 Decision Problems . . . 3
1.3 Halting Problem . . . 4

2 Register Machines **5**
2.1 Definitions . . . 5
2.2 Graphical Representation . . . 6
2.3 Partial Functions . . . 6
2.4 Computable Functions . . . 7

3 Coding Programs as Numbers **8**
3.1 Numerical Coding of Pairs . . . 8
3.2 Numerical Coding of Lists . . . 8
3.3 Numerical Coding of Programs . . . 9

4 Universal Register Machine **10**
4.1 High-Level Specification . . . 10
4.2 Universal Register Machine Inner Functions . . . 11
4.2.1 Assignment . . . 11
4.2.2 Pushing an element to a list (Cons) . . . 11
4.2.3 Popping an element from a list . . . 11

5 The Halting Problem **12**
5.1 Computable Functions . . . 13
5.1.1 Enumerating Computable Functions . . . 13
5.1.2 An Uncomputable Function . . . 13
5.1.3 (Un)decidable Sets of Numbers . . . 13

6 Turing Machines **14**
6.1 Turing Machine Configurations . . . 14
6.2 Turing Machine Computation . . . 15
6.3 Turing Computability . . . 17

7 Notions of Computability **18**
7.1 Computable Partial Functions . . . 18

8 Partial Recursive Functions **19**
8.1 Examples of Partial Recursive Functions . . . 19
8.2 Primitive Recursion . . . 19
8.3 Minimisation . . . 20
8.4 Partial Recursive Functions . . . 20
8.5 Ackermann's Function . . . 21

9 Lambda-Calculus **22**
9.1 λ-Terms . . . 22
9.1.1 Bound and Free Variable . . . 22
9.2 α-Equivalence . . . 23
9.3 β-Reduction . . . 23
9.3.1 Substitution . . . 23
9.3.2 Reduction . . . 23
9.3.3 β-Conversion $M =_\beta N$ . . . 24
9.4 β-Normal Forms . . . 24

10 Lambda-Definable Functions **26**
10.1 Encoding Data in λ-Calculus . . . 26
10.2 λ-Definable Functions . . . 27
10.2.1 Representing Basic Functions . . . 27
10.2.2 Representing Composition . . . 27
10.2.3 Representing Predecessor . . . 28
10.3 Primitive Recursion . . . 28
10.3.1 Y Combinator . . . 28
10.3.2 Representing Primitive Recursion . . . 29
10.3.3 Representing Minimisation . . . 30
10.4 λ-Definability . . . 30

1 Introduction

1.1 Hilbert's Entscheidungsproblem

Definition 1.1: Hilbert's Entscheidungsproblem

Is there an algorithm which when fed any statement in the 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?

Useful as for example could run it on the following:

$$\forall k > 1 : \exists p, q : (2k = p + q \land prime(p) \land prime(q))$$

Which would solve Goldbach's Conjecture. The algorithm wouldn't tell us the proof steps just whether or whether not it is a true statement.

Entscheidungsproblem means "decision Problem".

1.2 Decision Problems

Given:

- a set S whose elements are finite data structures of some kind
- a property P of elements S (i.e. a function $P : S \to \{\bot\top\}$)

The associated decision problem is to determine whether an element $s \in S$ has property P or not:

> Find an **algorithm** which terminates with a result 0 or 1 when fed an element $s \in S$ and yields result 1 when fed s if and only if s has property P.

At the time Hilbert posed this idea there was no precise formal defintion of an algorithm just examples for example:

- Euclids algorithm
- Multiplying numbers in decimal place notation
- Extracting square roots to any desired accuracy

However these examples share certain common features:

- A **finite** description of the procedure in terms of elementary operations
- It is **deterministic** (the next step is uniquely determined if there is one)
- The procedure may not terminate on some input data, but we can recognise when it does terminate and what the result is.

Negative solutions to the Entscheidungsproblem were proposed by Church who used Lambda-Calculus, and Turing who useed Turing Machines, both used different representations of algorithms which could be regarded as data. This means that other algorithms could be used on other algorithms.

1.3 Halting Problem

Definition 1.2: Halting Problem

The **Halting Problem** is a decision problem where:

- The set S consists of all pairs (A, D), where A is an algorithm and D is a datum on which it is designed to operate on.
- The property P holds for (A, D) if the algorithm A, when applied to datum D, eventually produces a result (that is, eventually **halts** - we write $A(D)\downarrow$ to denote this).

Both Turing and Church showed the Halting Problem is Undecidable, that is, there is no algorithm H such that for all $(A, D) \in S$:

$$H(A, D) = \begin{cases} 1 & \text{If } A(D)\downarrow \\ 0 & \text{Otherwise} \end{cases}$$

Proof 1.0: Informal Proof by contradiction of the Halting Problem

Assume there is an algorithm H, and let C be the algorithm:

"Given an input A, compute $H(A, A)$; if $H(A, A) = 0$ then return 1, else loop forever (don't halt)"

So:

$$\begin{aligned} &\forall A : (C(A)\downarrow \leftrightarrow H(A, A) = 0) && \text{Since H is total} \\ &\forall A : (C(A) \leftrightarrow \neg A(A)\downarrow) && \text{Definition of H} \end{aligned}$$

Taking A to be C, we get $C(C)\downarrow \leftrightarrow \neg C(C)\downarrow$, which is a contradiction so there must not be an algorithm H.

2 Register Machines

2.1 Definitions

Inofrmally, a register machine operates on $\mathbb{N} = \{0, 1, 2, ...\}$ stored in (idealised) registers using the following "elementary operations":

- Addition of 1 to the contents of a register
- Test whether the contents of a register is 0
- Subtract 1 frin the contents of a register if it is non-zero
- Jumps/gotos
- Conditionals

Definition 2.1: Register Machines

A Register Machine is specified by:

- Finitely many registers $R_0, R_1, \cdots, R_n$ (each capable of storing a natural number)
- 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 three forms:

$R^+ \to L'$	Add 1 to the contents of register R and jump to instruction labelled with L'
$R^- \to L', L''$	if contents of R is larger than 0, then subtract 1 from it and jump to L', else jump to L''.
HALT	Stop executing instructions

Definition 2.2: Register Machine Configuration

A **Register Machine Configuration** is a tuple:

$$c = (\ell, r_0, \cdots, r_n)$$

Where L_ℓ is the current label and r_i is the current content of register R_i.

We will write "$R_i = x$ [in configuration c]" to mean that $c = (\ell, r_0, \cdots, r_n)$ with $r_i = x$.

An initial configuration is given by $c_0 = (0, r_0, \cdots, r_n)$.

Definition 2.3: Register Machine Computation

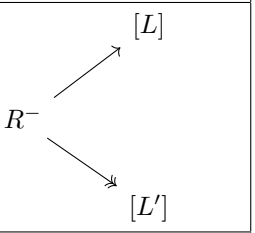
A **Computation** of a register machine is a (finite or infinite) sequence of configurations

$$c_0, c_1, c_2, ...$$

c_0 being the initial configuration.

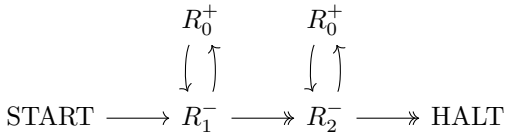
Each $c = (\ell, r_0, \cdots, r_n)$ in the sequence determines the next configuration in the sequence, if any, by carrying out the program instruction labelled L_ℓ with registers containing $r_0, ..., r_n$

2.2 Graphical Representation

Program Code	Graphical Representation
$R^+ \to L$	$R^+ \longrightarrow [L]$
$R^- \to L, L'$	
HALT	HALT
L_0	START $\longrightarrow [L_0]$

Example program with R_0, R_1, R_2:

L_0: $R_1^- \to L_1, L_2$
L_1: $R_0^+ \to L_0$
L_2: $R_2^- \to L_3, L_4$
L_3: $R_0^+ \to L_2$
L_4: `HALT`



2.3 Partial Functions

A register machine computation is deterministic: in any non-halting configuration, the next configuration is uniquely determined by the program. So the relation between initial and final register contents defined by a register machine program is a partial function.

Definition 2.4: Partial Function

A partial function from a set X to a set Y is specified by any subset $f \subseteq X \times Y$ satisfying

$$\in Y \quad (x, y) \in f \wedge (x, y') \in f \implies y = y'$$

For all $x \in X$ and for all $y, y' \in Y$

The cartesian product $X \times Y$ is the set of all ordered pairs $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$.

The defintion of a partial function is to say that for all $x \in X$ there is at most one $y \in Y$ with $(x, y) \in f$. For a (total) function, in contrast, each $x \in X$ has exactly one $y \in Y$ with $(x, y) \in f$.

Notation:

- $f(x) = y$ means $(x, y) \in f$
- $f(x) \downarrow$ means $\exists y \in Y \,.\, (f(x) = y)$ (i.e. $f(x)$ is defined)
- $f(x) \uparrow$ means $\neg\exists y \in Y \,.\, (f(x) = y)$ (i.e. $f(x)$ is undefined)
- $X \rightharpoonup Y$ is the set of all partial functions from X to Y
- $X \to Y$ is the set of all (total) functions from X to Y

Definition 2.5: Total function

A partial function $f \in X \rightharpoonup Y$ is total if it satisfies

$$\forall x \in X \quad f(x) \downarrow$$

2.4 Computable Functions

In essence, a partial function is computable if you can design a register machine that computes the partial function. We thereby require that the register machine fully implements the partial function: if the partial function is undefined for some input x, the register machine will not halt for that input x either.

Definition 2.6: Computable

$f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is (register machine) computable if there is a register machine M with at least $n+1$ registers $R_0, \cdots, R_n$ (and maybe more) such that for all $(x_1, \cdots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$, the computation of M with $c_0 = (0, x_1, \cdots, x_n, 0, ..., 0)$ halts with $R_0 = y$ if and only if $f(x_1, \cdots, x_n) = y$

3 Coding Programs as Numbers

3.1 Numerical Coding of Pairs

Definition 3.1: Coding of Pairs

For $x, y \in \mathbb{N}$, define:

$$\begin{aligned} \langle\!\langle x, y \rangle\!\rangle &\triangleq 2^x(2y+1) \\ \langle x, y \rangle &\triangleq 2^x(2y+1) - 1 \end{aligned}$$

$\langle\, .\, ,\, .\, \rangle$ provides a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$

$\langle\!\langle\, .\, ,\, .\, \rangle\!\rangle$ provides a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}_{>0}$
So we use $\langle\, .\, ,\, .\, \rangle$ when we want to encode every natural number as a possible pair and $\langle\!\langle\, .\, ,\, .\, \rangle\!\rangle$ when we need a special marker (that is not a pair) for "null" or "end of data".

3.2 Numerical Coding of Lists

Let $list\mathbb{N}$ be the set of all finite lists of natural numbers. We will use ML notation for lists:

- empty list: $[] \in list\mathbb{N}$
- list-cons: $x :: \ell \in list\mathbb{N}$ (given $x \in \mathbb{N}$ and $\ell \in list\mathbb{N}$)
- $[x_1, \cdots, x_n] \triangleq x_1 :: (x_2 :: \cdots\ (x_n :: []))$

Definition 3.2: Coding Lists

For $\ell \in list\mathbb{N}$ defin $\ulcorner \ell \urcorner$ by induction on the length of the list ℓ:

$$\ulcorner [] \urcorner \triangleq 0$$

$$\ulcorner x :: \ell \urcorner \triangleq \langle\!\langle x, \ulcorner \ell \urcorner \rangle\!\rangle \triangleq 2^x(2\ulcorner \ell \urcorner + 1)$$

The binary representation of a list thus looks like:

$$\mathrm{bin}(\ulcorner [x_1, \cdots, x_n] \urcorner) = 0\mathrm{b}1 \mid \underbrace{0\cdots0}_{x_n} 1 \underbrace{0\cdots0}_{x_{n-1}} 1 \cdots \underbrace{0\cdots0}_{x_1}$$

$\ell \mapsto \ulcorner \ell \urcorner$ gives a bijection from $list\mathbb{N}$ to $\mathbb{N}$

Note:

We need 0 as a marker for the empty list and therefore use $\langle\!\langle\, .\, ,\, .\, \rangle\!\rangle$ and not $\langle\, .\, ,\, .\, \rangle$.

3.3 Numerical Coding of Programs

Definition 3.3: Coding of Programs

If P is a register machine with n lines then its numerical code is represented as:

$$\ulcorner P \urcorner \triangleq \ulcorner [\ulcorner body_0 \urcorner, ..., \ulcorner body_n \urcorner] \urcorner$$

Where each $\ulcorner body \urcorner$ is encoded by the following:

$$\begin{aligned} \ulcorner R_i^+ \to L_j \urcorner &\triangleq \langle\!\langle 2i, j \rangle\!\rangle \\ \ulcorner R_i^- \to L_j, L_k \urcorner &\triangleq \langle\!\langle 2i+1, \langle j, k \rangle \rangle\!\rangle \\ \ulcorner \text{HALT} \urcorner &\triangleq 0 \end{aligned}$$

So every $e \in \mathbb{N}$ decodes to a unique program $\text{prog}(e)$, called the register machine with index e.

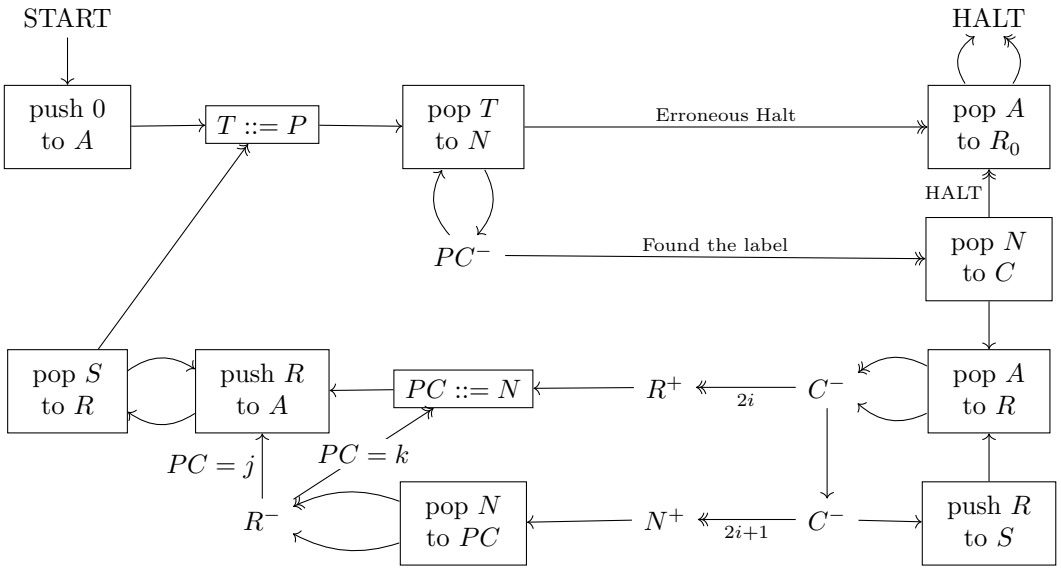
4 Universal Register Machine

4.1 High-Level Specification

The universal register machine U carries out the following computation, starting with $R_0 = 0$, $R_1 = e$, $R_2 = a$ and all other registers zero-ed it performs:

1. Decode e as a register machine program P.
2. Decode a as a list of register values $a_1, \cdots, a_n$.
3. Carry out the computation of the register machine program P with $R_0 = 0, R_1 = a_1, ..., R_n = a_n$

The structure of U is below:



We first push 0 to A, which adds a R_0 register to A with value of 0.

Then assign T to P so that we can pop off T without messing up P.

Then pop T to N in order to get the instruction that the PC points to.

Then N is the instruction coded as described in 3.3 Numerical Coding of Programs

Get the register that the operation acts on from A, pushing all the registers we pop off before to S, so we can push them back onto A. This ensures that R is updated in A.

If C is even ($C = 2i$) then we increment the simulated register and then assign PC to the value stored in N still which is j.

If C is odd ($C = 2i + 1$) then we increment N since N now is equal to $\langle j, k \rangle$ so we convert it to $\langle\langle j, k \rangle\rangle$ in order to be able to pop. Then perform the subtraction, if R is 0 then we set $PC = k$ by setting it to what is left in N.

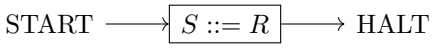
Push back all the registers that we popped off looking for R.

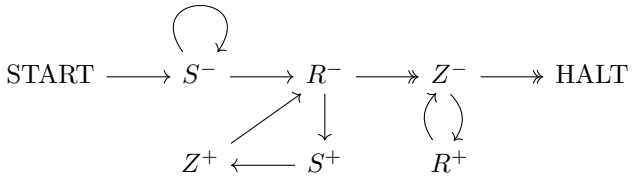
Then reassign $T ::= P$ and repeat from there.

4.2 Universal Register Machine Inner Functions

As seen above we need to have certain functions such as assigning a value in register to another register without affecting the original. The structure of these will be shown below.

4.2.1 Assignment

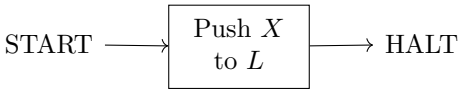


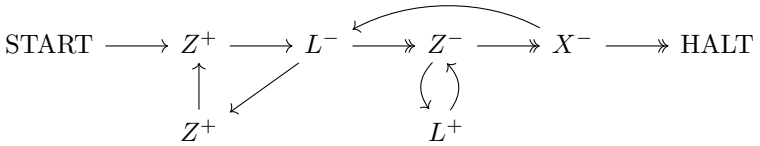


Preconditions:
$R = x,\ S = y,\ Z = 0$

Postconditions:
$R = x,\ S = x,\ Z = 0$

4.2.2 Pushing an element to a list (Cons)

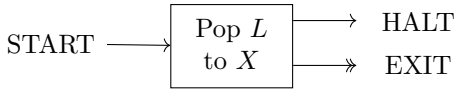


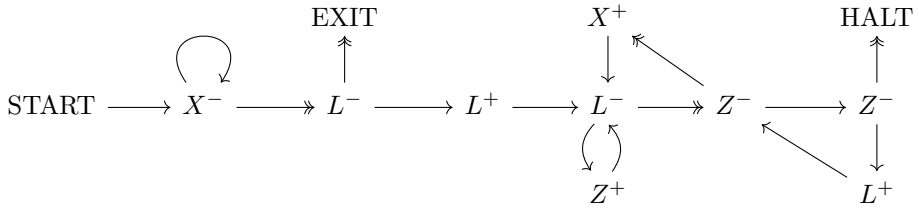


Preconditions:
$X = x,\ L = \ell,\ Z = 0$

Postconditions:
$X = 0,\ L = \langle\langle x, \ell \rangle\rangle = 2^x(2\ell + 1),\ Z = 0$

4.2.3 Popping an element from a list





Preconditions:
$X = n,\ L = \langle\langle x, \ell \rangle\rangle,\ Z = 0$

Postconditions:
$X = x,\ L = \ell,\ Z = 0$

5 The Halting Problem

Definition 5.1: The Halting Problem

A register machine H decides the Halting Problem if for all $e, a_1, \cdots, a_n \in \mathbb{N}$, starting H with:

$$R_0 = 0 \qquad R_1 = e \qquad R_2 = \ulcorner [a_1, \ldots, a_n] \urcorner$$

And all other registers zereod the computation of H always halts with R_0 containing 0 or 1; moreover **when** the computation halts, $R_0 = 1$ if and only if the register machine program with index e eventually halts when started with $R_0 = 0, \quad R_1 = a_1, \ \ldots, \ R_n = a_n$ and all other registers zereod.

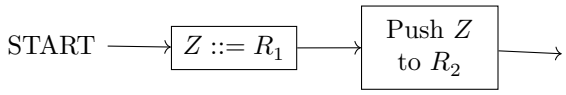
Theorem 5.1

No register machine H that decided the Halting Problem can exist.

Proof 5.1: No H can exist

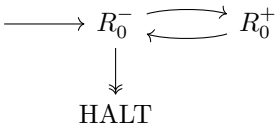
Assume that there exists a register machine H that decides the Halting Problem, and we will derive a contradiction to disprove this assumption.

Define H' as H with the START$\rightarrow$ with:



Where Z is a register not present in H's program.

Define C to be H' with every HALT (and erroneous halt) replaced with:



Let $c \in \mathbb{N}$ be the index of C's program.

$$\begin{aligned}
& C \text{ started with } R_1 = c \text{ eventually halts} \\
\iff & H' \text{ started with } R_1 = c \text{ halts with } R_0 = 0 \\
\iff & H \text{ started with } R_1 = c,\ R_2 = \ulcorner [c] \urcorner \text{ halts with } R_0 = 0 \\
\iff & \text{prog}(c) \text{ started with } R_1 = c \text{ does not halt} \\
\iff & C \text{ started with } R_1 = c \text{ does not halt.}
\end{aligned}$$

So we come to the conclusion:

$$C \text{ started with } R_1 = c \text{ eventually halts} \iff C \text{ started with } R_1 = c \text{ does not halt.}$$

Which is a contradiction which means that our assumption was incorrect so there cannot exist such a machine H.

5.1 Computable Functions

Recall a (partial) function is computable if it satisfies the defintion in 2.6 Computable Functions

The register machine M, that computes f could also be used to compute a unary function $n = 1$, or a binary function $n = 2$, etc. From now on we will concentrate on the unary case only.

5.1.1 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 prog(e). So for all $x, y \in \mathbb{N}:\ \varphi_e(x) = y$ holds iff the computation of prog(e) started with $R_0 = 0, R_1 = x$ and all other registers zeroed eventually halts with $R_0 = y$.

This $e \mapsto \varphi_e$ defines an onto function from $\mathbb{N}$ to the collection of all computable functions from $\mathbb{N}$ to $\mathbb{N}$.

5.1.2 An Uncomputable Function

Let $f \in \mathbb{N} \rightharpoonup \mathbb{N}$ be the partial function with graph $\{(x, 0) \mid \varphi_x(x) \uparrow\}$. Thus:

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

f is not computable, because if it were, then $f = \varphi_e$ for some $e \in \mathbb{N}$ and hence:

- if $\varphi_e(e) \uparrow$, then $f(e) = 0$, by definition, so $\varphi_e(e) = 0$, so $\varphi_e(e) \downarrow$
- if $\varphi_e(e) \downarrow$, then $f(e) \downarrow$, since $f = \varphi_e$, so $\varphi_e(e) \uparrow$

Which is a contradiction for both cases and so f cannot be computable.

5.1.3 (Un)decidable Sets of Numbers

Given a subset $S \subseteq \mathbb{N}$, its characteristic function $\chi_S \in \mathbb{N} \to \mathbb{N}$ is given by:

$$\chi_S(x) \triangleq \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

Definition 5.2: Decidable Sets

$S \in \mathbb{N}$ is called (register machine) **decidable** if its characteristic function $\chi_S \in \mathbb{N} \to \mathbb{N}$ is a register machine computable function. Otherwise it is called undecidable.

General Strategy:

To prove a set is undecidable try to show that the decidability of S would imply decidability of the Halting Problem which is a known undecidable problem and thus a contradiction of S is decidable.

6 Turing Machines

Informally, a Turing Machine is a finite state machine with a tape that is unbound "to the right". The tape is linear and divided into cells, each of which contains a symbol of a finte alphabet of tape symbol, including a special **blank** symbol. Only finitely many cells contain non-blank symbols. A tape head can read or write the symbol in one specific cell, be moved one step to the left (unless the end of the tap is reached), or move to the right.

Definition 6.1: Turing Machine

A **Turing Machine** is specified by a tuple (Q, Σ, s, δ) with:

- Q, a finite set of machine states.
- Σ, a finite set of tape symbols (disjoint from Q) containing distinguished symbols $\triangleright$ (left endmarker) and $\sqcup$ (blank).
- $s \in Q$, an initial state.
- $\delta \in (Q \times \Sigma) \to (Q \cup \{\text{acc}, \text{rej}\}) \times \Sigma \times \{L, R, S\}$, a transition function satisfiying:

  For all $q \in Q$, there exists $q' \in Q \cup \{\text{acc}, \text{rej}\}$ with $\delta(q, \triangleright) = (q', \triangleright, R)$

  (i.e the left endmarker is never overwritten and the machine always moves to the right when scanning it)

Example:
$M = (Q, \Sigma, s, \delta)$ with states $Q = \{s, q, q'\}$ the symbols $\Sigma = \{\triangleright, \sqcup, 0, 1\}$ and the transition function:

δ	$\triangleright$	$\sqcup$	0	1
s	$(s, \triangleright, R)$	$(q, \sqcup, R)$	$(\text{rej}, 0, S)$	$(\text{rej}, 1, S)$
q	$(\text{rej}, \triangleright, R)$	$(q', 0, L)$	$(q, 1, R)$	$(q, 1, R)$
q'	$(\text{rej}, \triangleright, R)$	$(\text{acc}, \sqcup, S)$	$(\text{rej}, 0, S)$	$(q', 1, L)$

6.1 Turing Machine Configurations

Definition 6.2: Turing Machine Configuration

A Turing Machine Configuration is a triple (q, w, u) where:

- $q \in Q \cup \{\text{acc}, \text{rej}\}$ is the current state.
- w is a non-empty string ($w = va$) of tape symbols under and to the left of the tape head, whose last element a is the contents of the cell under the head.
- u is a (possibly empty) string of tape symbols to the right of the tape head (up to some point beyond which all symbols are blanks $\sqcup$)

A Turing Machine starts in the initial configuration $(s, \triangleright, u)$

6.2 Turing Machine Computation

Given a Turing Machine $M = (Q, \Sigma, s, \delta)$ we write:

$$(q, w, u) \to_M (q', w', u')$$

To mean that $q \neq \text{acc}, \text{rej}, w = va$ (for some v, a) and exactly one of the following holds for $\delta(q, a)$:

- $\delta(q, a) = (q', a', L),\ w' = v,\ u' = a'u$
- $\delta(q, a) = (q', a', S),\ w' = va',\ u' = u$
- $\delta(q, a) = (q', a', R),\ u = a''u'',\ w' = va'a'',\ u' = u''$ (and u is non-empty)
- $\delta(q, a) = (q', a', R),\ u = \epsilon, w' = va'\textvisiblespace,\ u' = \epsilon$

Definition 6.3: Turning Machine Computation

A **computation** of a Turing Machine M is a (finite or infinite) sequence of configurations $c_0, c_1, c_2, ...$ where:

- $c_0 = (s, \triangleright, u)$ is an initial configuration
- $c_i \to_M c_{i+1}$ holds for each $i = 0, 1, ...$

The computation does not halt if the sequence is infinite. It halts if the sequence is finite and its last element is of the form (acc, w, u) or (rej, w, u) for some w and u.

Example:
$M = (Q, \Sigma, s, \delta)$ with states $Q = \{s, q, q'\}$ the symbols $\Sigma = \{\triangleright, \textvisiblespace, 0, 1\}$ and the transition function:

δ	$\triangleright$	$\textvisiblespace$	0	1
s	$(s, \triangleright, R)$	$(q, \textvisiblespace, R)$	$(\text{rej}, 0, S)$	$(\text{rej}, 1, S)$
q	$(\text{rej}, \triangleright, R)$	$(q', 0, L)$	$(q, 1, R)$	$(q, 1, R)$
q'	$(\text{rej}, \triangleright, R)$	$(\text{acc}, \textvisiblespace, S)$	$(\text{rej}, 0, S)$	$(q', 1, L)$

We claim that the computation of M starting from configuration $(s, \triangleright, \textvisiblespace 1^n 0)$ halts in the configuration $(\text{acc}, \triangleright\textvisiblespace, 1^{n+1}0)$.

$$\begin{aligned}
(s, \triangleright, \textvisiblespace 1^n 0) &\to_M (s, \triangleright\textvisiblespace, 1^n 0) \\
&\to_M (q, \triangleright\textvisiblespace 1, 1^{n-1}0) \\
&\vdots \\
&\to_M (q, \triangleright\textvisiblespace 1^n, 0) \\
&\to_M (q, \triangleright\textvisiblespace 1^n 0, \epsilon) \\
&\to_M (q, \triangleright\textvisiblespace 1^{n+1}\textvisiblespace, \epsilon) \\
&\to_M (q', \triangleright\textvisiblespace 1^{n+1}, 0) \\
&\vdots \\
&\to_M (q', \triangleright\textvisiblespace 1^{n+1}, 1^{n+1}0) \\
&\to_M (\text{acc}, \triangleright\textvisiblespace, 1^{n+1}0)
\end{aligned}$$

Theorem 6.1

The computation of a Turing Machine M can be implemented by a register machine.

Proof (sketch):

Step 1: Fix a numerical encoding of M's startes, tape symbols, tape contents and configurations.

Step 2: Implement M's transition function using Register Machine Instructions on codes

Step 3: Implement a Register Machine to repeatedly carry out the $\rightarrow_M$ operation.

Step 1. Indentify states and tape symbols with particular numbers:

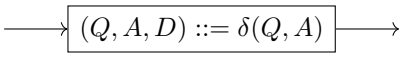
$$\begin{aligned} \text{acc} &= 0 \\ \text{rej} &= 1 \\ Q &= \{2, 3, ..., n\} \\ \text{␣} &= 0 \\ \triangleright &= 1 \\ \sigma &= \{2, 3, ..., m\} \qquad (\sigma = \Sigma \setminus \{\text{␣}, \triangleright\}) \end{aligned}$$

Encode configurations $c = (q, w, u)$ as:

$$\ulcorner c \urcorner = \ulcorner [q, \ulcorner [a_n, \ldots, a_1] \urcorner, \ulcorner [b_1, \ldots, b_m] \urcorner] \urcorner$$

where $w = a_1 \cdots a_n \;\; (n > 0)$ and $u = b_1 \cdots b_m \;\; (m \geq 0)$ say.

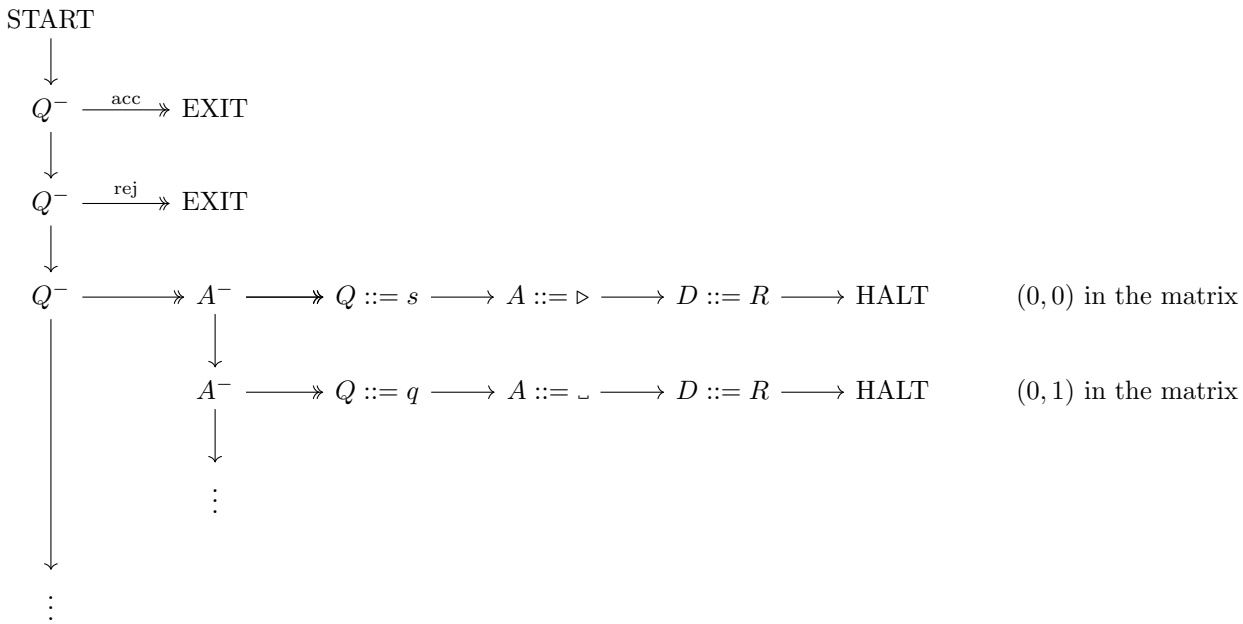
Step 2. Using the registers Q for the current state, A for the current tape symbol and D for the current direction of the tape head (with $L = 0,\; R = 1$ and $S = 2$) one can turn the finite table of pairs specifying δ into a register machine program:



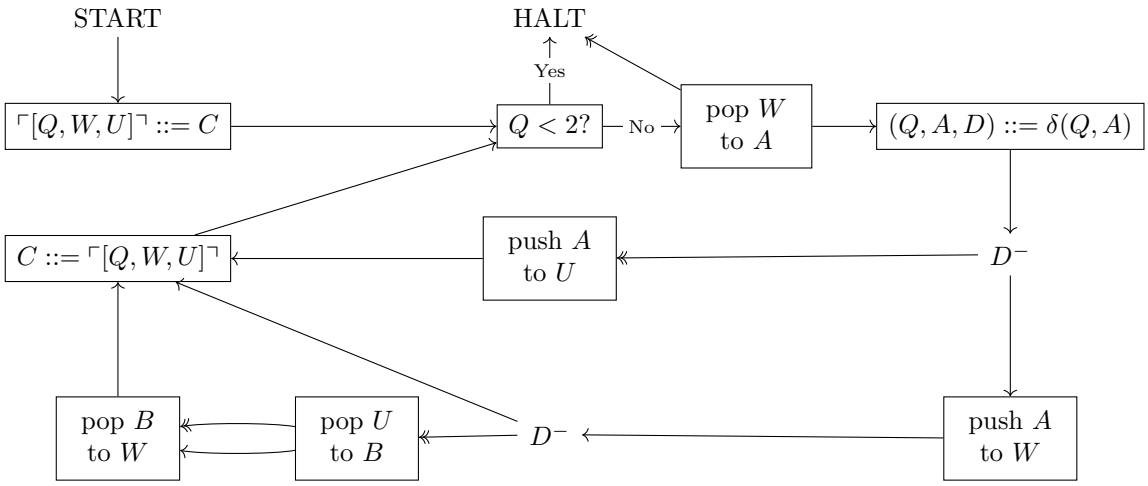
Preconditions:
$Q = q, \;\; A = a, \;\; D = d$

Postconditions:
$Q = q', \;\; A = a', \;\; D = d' \qquad$ where $(q', a', d') = \delta(q, a)$

Something similar to:



Step 3. The register machine shown below will carry ot M's computation. It uses the registers C for the code of the current configuration, W for the code of the tape symbols at and to the left of the tape head (reading right-to-left) and U for the code of tape symbols right of the tape head (reading left-to-right). Starting with C containing the code of an initial configuration (and all other registers zeroed), the register machine program halts if and only if M halts; and in that case C holds the code of the final configuration.



6.3 Turing Computability

Definition 6.4: Turing Computable Functions

A function $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is **Turing Computable** if and only if there is a Turing Machine M with the following property:

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

And to code a list of numbers onto a tape we can do the following:

Definition 6.5: Coding Lists in Turing Machines

A tape codes a list of numbers if precisely two cells contains 0 and the only cells containing 1 occur between these.

A tape looking like:

$$\triangleright \sqcup \cdots \sqcup 0 \underbrace{1 \cdots 1}_{n_1} \sqcup \underbrace{1 \cdots 1}_{n_2} \sqcup \cdots \sqcup \underbrace{1 \cdots 1}_{n_k} 0 \sqcup \cdots$$

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

Theorem 6.2: Turing Computability

A partial function is Turing Computable if and only if it is register machine computable.

We have shown that any Turing Machine can be implemented by a register machine. So we have proven the right direction. To prove the left direction we would show that any register machine can be implemented by a Turing machine, to do this we would have to show that each action of the register machine can be implemented on a Turing machine, which is tedious so is omitted.

7 Notions of Computability

Theorem 7.1

Every algorithm (in an intuitive sense) can be realised as a Turing machine.

7.1 Computable Partial Functions

Our aim now is to arrive at a more abstract, machine-independent description of the collection of computable partial functions, as opposed to Turing machines or register machines.

We start with a foundation of three basic functions, which are all register machine computable.

Projection $\text{proj}_i^n \in \mathbb{N}^n \to \mathbb{N}$:

$$\text{proj}_i^n(x_1, \ldots, x_n) \triangleq x_i$$

Constant (with value 0) $\text{zero}^n \in \mathbb{N}^n \to \mathbb{N}$:

$$\text{zero}^n(x_1, \ldots, x_n) \triangleq 0$$

Successor $\text{succ} \in \mathbb{N} \to \mathbb{N}$:

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

Composition. The composition of $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ with $g_1, \ldots, g_n \in \mathbb{N}^m \to \mathbb{N}$ is the partial function $f \circ [g_1, \ldots, g_n] \in \mathbb{N}^m \to \mathbb{N}$ satisfiying for all $x_1, \ldots, x_n \in \mathbb{N}$:

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

Where $\equiv$ is the 'Kleene equivalence' which means that either both the LHS and RHS are undefined, or both are defined and equal.

So $f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m) = z$ if and only if there exists $y_1, \ldots, y_n$ with $g_i(x_1, \ldots, x_m) = y_i$ for all $i = 1 \ldots n$ and $f(y_1, \ldots, y_n) = z$

Theorem 7.2

$f \circ [g_1, \ldots, g_n]$ is computable if f and $g_1, \ldots, g_n$ are.

8 Partial Recursive Functions

8.1 Examples of Partial Recursive Functions

- $f_1(x)$ is the sum of naturals upto and including x:

$$\begin{cases} f_1(0) & \equiv 0 \\ f_1(x+1) & \equiv f_1(x) + (x+1) \end{cases}$$

- $f_2(x)$ is the x^{th} Fibonnaci number:

$$\begin{cases} f_2(0) & \equiv 0 \\ f_2(1) & \equiv 1 \\ f_1(x+2) & \equiv f_2(x) + f_2(x+1) \end{cases}$$

- $f_3(x)$ is undefined unless $x = 0$:

$$\begin{cases} f_3(0) & \equiv 10 \\ f_3(x+1) & \equiv f_3(x+2) + 1 \end{cases}$$

8.2 Primitive Recursion

Theorem 8.1

Given $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $g \in \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$, there is a unique $h \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ satisfiying:

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

For all $\vec{x} \in \mathbb{N}^n$ and $x \in \mathbb{N}$.

$$h \equiv \rho^n(f, g)$$

We call h the partial function defined by primitive recursion from f and g. Recursion is done over the parameter x, we also allow for more parameters to be passed, namely $\vec{x}$. If additional parameters are not needed then $f = n$ where it acts just like a specific natural number.

Examples:

Addition: $add \in \mathbb{N}^2 \to \mathbb{N}$ satisfies:

$$\begin{cases} add(x_1, 0) & \equiv x_1 \\ add(x_1, x+1) & \equiv add(x_1, x) + 1 \end{cases}$$

So, $add = \rho^1(f, g)$ where $f(x_1) \triangleq x_1$ and $g(x_1, x, h) \triangleq h + 1$.

We can represent add in terms of the three basic functions we previously defined:

$$add = \rho^1(\text{proj}_1^1, \text{succ} \circ \text{proj}_3^3)$$

Multiplication: $mult \in \mathbb{N}^2 \to \mathbb{N}$ satisfies:

$$\begin{cases} mult(x_1, 0) & \equiv 0 \\ mult(x_1, x+1) & \equiv mult(x_1, x) + x_1 \end{cases}$$

$$mult = \rho^1(\text{proj}_1^1, add \circ [\text{proj}_3^3, \text{proj}_1^3])$$

Since add can be made up from basic functions so can $mult$

Definition 8.1: Primitive Recursive

A (partial) function f is primitive recursive ($f \in \text{PRIM}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition and primitive recursion.

In other words, the set PRIM of primitive recursive functions is the smallest set (with respect to subset inclusion) of partial functions containing the basic functions and closed under the operations of composition and primitive recursion.

All primitive recursive functions are total functions, as:

- all the basic functions are total
- if $f, g_1, \ldots, g_n$ are total, then so is $f \circ [g_1, \ldots, g_n]$
- if f and g are total, then so is $\rho^n(f, g)$

All primitive recursive functions are register machine computable.

8.3 Minimisation

Definition 8.2: Minimisation

Let $f \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ be a partial function. We then define the minimisation $\mu^n f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ as the least x such that $f(\vec{x}, x) = 0$ and for each $i < x$, $f(\vec{x}, i)$ is defined and larger than zero. We write $\mu^n f(\vec{x})$ for this value of x.

If no x exists (either because $f(\vec{x}, x)$ is never zero or because $f(\vec{x}, x')$ is undefined for an x' smaller than x), then $\mu^n f(\vec{x})$ is undefined.

8.4 Partial Recursive Functions

Definition 8.3: Partial Recursive

A partial function f is partial recursive ($f \in \text{PR}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition, primitive recursion, and minimisation.

In other words, the set PR of partial recursive functions is the smallest set (with respect to subset inclusion) of partial functions containing the basic functions and closed under the operations of composition, primitive recursion, and minimisation.

All partial recursive function are register machine computable.

Theorem 8.2

Not only is every $f \in \text{PR}$ computable, but conversly, every computable partial function is partial recursive.

8.5 Ackermann's Function

Ackermann's Function is a famous example of a (total) recursive function that is not primitive recursive.

To express Ackermann's function we will use a family of functions, rather than doing addition, multiplication with multiple arguments we will fix one to 2 for this explanation.

$f_0(x) = x + 2$	addition of x with the fixed value 2
$f_1(x) = 2 \cdot x$	multiplication
$f_2(x) = 2^x$	exponentiation

To show that exponentiation is repeated multiplication and multiplication is repeated addition we can use recursion.

$$f_1(x+1) = 2(x+1) = 2 \cdot x + 2 = f_0(f_1(x))$$
$$f_2(x+1) = \quad 2^{x+1} \quad = 2 \cdot (2^x) = f_2(f_2(x))$$

We can then define further functions, $f_{n+1}(x+1) = f_n(f_{n+1}(x))$. This idea is the core around which Ackermann's function is built. Instead of $f_n(x)$, Ackermann's function is usually written in the form $f(n, x)$. And if we use the increment by one as our base case, we get the following:

$$f(0, x) = x + 1 \qquad\qquad f(n+1, x+1) = f(n, f(n+1, x))$$

We just need to have a proper definition of $f_n(0) = f(n, 0)$ for all $n > 0$.

Definition 8.4: Ackermann's Function

There is a (unique) function $ack \in \mathbb{N} \to \mathbb{N}$ satisfiying:

$$\begin{aligned} ack(0, x_2) &= x_2 + 1 \\ ack(x_1 + 1, 0) &= ack(x_1, 1) \\ ack(x_+1, x_2 + 1) &= ack(x_1,\ ack(x_1 + 1, x_2)) \end{aligned}$$

The function ack is computable and hence recursive. However, ack grows facter than any primitive recursive function $f \in \mathbb{N}^2 \to \mathbb{N}$:

$$\exists N_f . \forall x_1, x_2 > N_f . f(x_1, x_2) < ack(x_1, x_2)$$

Which means that ack is not primitive recursive, although it is total recursive.

9 Lambda-Calculus

9.1 λ-Terms

λ-terms are built up from a given, countable collections of variables $x, y, z, \ldots$ by two operations for forming λ-terms:

- **λ-abstraction**: $(\lambda x.M)$ (where x is a variable and M is a λ-term)
- **Application**: (MM') (where M and M' are both λ-terms)

Instead of writing $\lambda x.\lambda y.M$ we shorten this to $\lambda x\, y.M$. Another notation is let, "let $x = N$ in M", which stands for $(\lambda x.M)\ N$.

$$\underbrace{\big(\lambda y.\ \underbrace{(x\ y)}_{\text{body}}\big)}_{\text{abstraction}}\ x' \quad\longrightarrow\quad x\ x'$$

Identity:
The λ-term $\lambda x.x$ stands for the identity and is usually abbreviated to I. For any λ-term t,

$$I\ t \equiv (\lambda x.x)\ t \longrightarrow t$$

Pairs:
If we wrote pairs (x, y) as (xy) in lambda calculus then that signifies application, so we need an alternative. We encode the pair as $\lambda f.f\ x\, y$.

To retrieve first or second item we can use the following terms, $\lambda x\, y.x$ or $\lambda x\, y.y$.

$$(\lambda f.f\ A\, B)(\lambda x\, y.\ x) \longrightarrow A$$

9.1.1 Bound and Free Variable

In $\lambda x.M$ we call the x the bound variable and M the body of the λ abstraction. An occurrence of x in a λ term M is called:

- **Binding** if it is between λ and the dot, e.g. $(\lambda \mathbf{x}.y\, x)x$
- **Bound** if it is in the body of a binding occurence of x, e.g. $(\lambda x.y\, \mathbf{x})x$
- **Free** if it is nether binding nor bound, e.g. $(\lambda x.y\, x)\mathbf{x}$

For a λ-term M we can define the sets of free and bound variables $FV(M)$ and $BV(M)$, respectively:

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.M) &= FV(M) \setminus \{x\} \\
FV(M\ N) &= FV(M) \cup FV(N) \\
BV(x) &= \emptyset \\
BV(\lambda x.M) &= BV(M) \cup \{x\} \\
BV(M\ N) &= BV(M) \cup BV(N)
\end{aligned}$$

We write $x \# M$ to mean that x does not occur in the term M.

If $FV(M) = \emptyset$, M is called a **closed term** or **combinator**. For instance, the identity is a **combinator**, as is $K = \lambda x\, y.x$

9.2 α-Equivalence

Structurally equivalent λ-terms that only differ in the names of bound variables are called α-equivalent. Changing the name of a bound variable is also known as α-conversion and is the simplest form of substitution.

α-equivalence $M =_\alpha M'$ is the binary relation inductively generated by the rules:

$$\frac{}{x =_\alpha x} \qquad \frac{z\#(M\ N) \quad M[z/x] =_\alpha N[z/y]}{\lambda x.M =_\alpha \lambda y.N} \qquad \frac{M =_\alpha M' \quad N =_\alpha N'}{M\ N =_\alpha M'\ N'}$$

Example:
Note that x is bound twice in the example, thus introducing completely different variables with the same name. When doing α-conversion, take care not to replace variables that only look alike.

$$\lambda x\, y.x\ (\lambda x.x\ y) =_\alpha \lambda z\, y.z\ (\lambda x.x\ y) =_\alpha \lambda z\, t.z\ (\lambda x.x\ t)$$

9.3 β-Reduction

9.3.1 Substitution

If we replace variable not only by other variables, but with any λ-term M, we arrive at the general concept of substitution, as determined by the following rules:

$$\begin{array}{rcll} x[M/x] & = & M & \\ y[M/x] & = & y & \text{if } y \neq x \\ (\lambda y.N)[M/x] & = & \lambda y.(N[M/x]) & \\ (N_1\ N_2)[M/x] & = & N_1[M/x]\ N_2[M/x] & \end{array}$$

The side-condition $y\#(M\ x)$ (y doesn not occur in M and $y \neq x$) makes substitution 'Capture-Avoiding', i.e. making sure that two distinct variables do not suddenly coincide and become one.

9.3.2 Reduction

Just as you can apply a function to an argument, you can apply a λ-abstraction to another λ-term. The notation $(\lambda x.M)\ a$ is thus intended to mean that all variables x in the term M shall be replaced by a, i.e. $(\lambda x.M)\ a = M[a/x]$. This substitution is at the heart of β-reduction.

The natural notation of computation for λ-terms is thus given by stepping from a β-redex $(\lambda x.M)\ N$ to the corresponding β-reduct $M[N/x]$ (Where 'redex' is short for 'reducible expression'). This is sometimes called contraction of $(\lambda x.M)\ N$ to $M[N/x]$.

One-step β-reduction, $M \to M'$.
The one-step β-reduction performs a single contraction, i.e. it replaces one λ-subterm with the pattern $(\lambda x.\, M)\ N$ by $M[N/x]$, leading to the following rules:

$$\frac{}{(\lambda x.\, M)\ N \to M[N/x]} \qquad \frac{M \to M'}{\lambda x.\, M \to \lambda x.\, M'}$$

$$\frac{M \to M'}{M\ N \to M'\ N} \qquad \frac{M \to M'}{N\ M \to N\ M'}$$

$$\frac{N =_\alpha M \quad M \to M' \quad M' =_\alpha N'}{N \to N'}$$

Many-step β-reduction, $M \twoheadrightarrow M'$.
It is often convenient to combine several single step of β-reduction and perform them as 'one' step. This leads to the many-step or 'big-step' reduction:

$$\frac{M =_\alpha M'}{M \twoheadrightarrow M'} \qquad \frac{M \to M'}{M \twoheadrightarrow M'} \qquad \frac{M \to M' \quad M' \to M''}{M \twoheadrightarrow M''}$$

9.3.3 β-Conversion $M =_\beta N$

Informally $M =_\beta N$ holds if N can be obtained from M by performing 0 or more steps of α-equivalence, β-reduction, or β-expansion (the inverse of β-reduction).

β-Conversion $M =_\beta N$ is the binary relation inductively generated by the rules:

$$\frac{M =_\alpha M'}{M =_\beta M'} \qquad \frac{M \to M'}{M =_\beta M'} \qquad \frac{M =_\beta M'}{M' =_\beta M} \qquad \frac{M =_\beta M' \quad M =_\beta M''}{M =_\beta M'}$$

$$\frac{M =_\beta M'}{\lambda x.\, M =_\beta \lambda x.M'} \qquad \frac{M =_\beta M' \quad N =_\beta N'}{M\ N =_\beta M'\ N'}$$

Theorem 9.1: Church-Rosser Theorem

$\twoheadrightarrow$ is confluent, that is, if $M \twoheadrightarrow M_1$, and $M \twoheadrightarrow M_2$, then there exists an M' such that $M_1 \twoheadrightarrow M'$ and $M_2 \twoheadrightarrow M'$.

Theorem 9.2: Corollary

$M_1 =_\beta M_2$ if and only if there is an M such that $M_1 \twoheadrightarrow M \twoheadleftarrow M_2$. In other words; M_1 and M_2 are β-equivalent if both can be reduced to a common term M'.

Proof:

> $=_\beta$ satisfies the rules generating $\twoheadrightarrow$;
>
> So $M \twoheadrightarrow M'$ implies $M =_\beta M'$. Thus if $M_1 \twoheadrightarrow M$ and $M_2 \twoheadrightarrow M$, then $M_1 =_\beta M =_\beta M_2$ and so $M_1 =_\beta M_2$
>
> Conversely, the relation $\{(M_1, M_2) \mid \exists.\, M_1 \twoheadrightarrow M \wedge M_2 \twoheadrightarrow M\}$ satisfies the rules generating $=_\beta$:
>
> The only difficult case is closure of the relation under transitivity and for this we use the Church-Rosser theorem. Hence $M_1 =_\beta M_2$ implies that there is an M such that $M_1 \twoheadrightarrow M$ and $M_2 \twoheadrightarrow M$.
>
> □

9.4 β-Normal Forms

Definition 9.1: β-Normal Form

A λ-term is in β-normal form (β-nf) if it contains no β-redexes. M has β-normal form N if $M =_\beta N$ with N a β-normal form.

Note that if N is a β-normal form and $N \twoheadrightarrow N'$, then it must be that $N =_\alpha N'$. Hence, if $N_1 =_\beta N_2$ with N_1 and N_2 both β-normal forms, then $N_1 =_\alpha N_2$. Because if $N_1 =_\beta N_2$, then by Church-Rosser $N_1 \twoheadrightarrow M' \twoheadleftarrow N_2$ for some M', so $N_1 =_\alpha M' =_\alpha N_2$.

So, the β-normal form of M is unique up to α-equivalence if it exists.

Non-termination:
Some λ-terms have no β-normal form. Every attempt to reduce such a λ-term introduces another β-redex, leading to an infinite sequence. With λ-terms as representations of algorithms, this means that the algorithm never terminates.

For example the Ω-combinator $\big(\Omega = (\lambda x.\, x\ x)(\lambda x.\, x\ x)\big)$ has no β-normal form, as:

$$\Omega \to (x\ x)[(\lambda x.\, x\ x)/x] = \Omega \qquad (\Omega \to \Omega)$$
$$\Omega \twoheadrightarrow M \text{ thus } \implies \Omega =_\alpha M$$

The picture is not always that clear: in fact, a term can possess both a β-normal form and infinite chains of reductions from it. In other words: depending on which β-redex you reduce first, you either arrive at a β-normal form, or you are caught in an 'infinite loop'.

To address this ambiguity, we introduce normal-order reduction. Normal-order reduction is a deterministic strategy for reducing λ-terms reduce the 'left-most, outer-most' redex first.

Definition 9.2: Normal-order Reduction

A redex is in head position in a λ-term M if M takes the form:

$$\lambda x_1, \ldots, x_n.\underline{(\lambda x.\, M')\ M_1}\ M_2\ \ldots\ M_m \qquad (n \geq 0, m \geq 1)$$

A λ-term is said to be in head normal form if it contains no redex in head position, in other words takes the form:

$$\lambda x_1, \ldots, x_n.\, x\ M_1\ M_2\ \ldots\ M_m \qquad (mn \geq 0)$$

Normal order reduction first continually reduces redexes in head position; if that process terminates then one has reached a head normal form and one continues applying head reduction in the subterms $M_1, M_2, \ldots$ from left to right.

Normal-order reduction of M **always** reaches the β-normal form of M if it possesses one.

10 Lambda-Definable Functions

10.1 Encoding Data in λ-Calculus

Computation within λ-calculus is given by β-reduction as previously discussed.

Definition 10.1: Church's Numerals

$$\begin{array}{lcll}
\underline{0} & \triangleq & \lambda f\, x.\, x & \equiv \lambda f\, x.\, f^0\ x \\
\underline{1} & \triangleq & \lambda f\, x.\, f\ x & \equiv \lambda f\, x.\, f^1\ x \\
\underline{2} & \triangleq & \lambda f\, x.\, f\ (f\ x) & \equiv \lambda f\, x.\, f^2\ x \\
 & \cdots & & \cdots \\
\underline{n} & \triangleq & \lambda f\, x.\, \underbrace{f\ (\cdots (f}_{n}\ x) \cdots) & \equiv \lambda f\, x.\, f^n\ x
\end{array}$$

With this notation we have $\underline{n}\ M\ N =_\beta M^n\ N$. Note that M^n alone has no meaning in this context. In particular $M\ M\ M = M^2\ M \neq M^3$

Definition 10.2: Church's Booleans

Like the numerals, the boolen values for **true** and **false** are λ-abstractions that may be applied to a λ-term.

(Note that **False** is actually α-equivalent to $\underline{0}$)

$$\begin{array}{rcl}
\textbf{True} & \triangleq & \lambda x\, y.\, x \\
\textbf{False} & \triangleq & \lambda x\, y.\, y \\
\textbf{If} & \triangleq & \lambda f\, x\, y.f\ x\ y \\
\textbf{Eq}_0 & \triangleq & \lambda x.\, x\ (\lambda y.\, \textbf{False})\ \textbf{True}
\end{array}$$

Definition 10.3: Church's Ordered Pairs

$$\begin{array}{rcl}
\textbf{Pair} & \triangleq & \lambda x\ y\, f.\, f\ x\ y \\
\textbf{Fst} & \triangleq & \lambda f.\, f\ \textbf{True} \\
\textbf{Scd} & \triangleq & \lambda f.\, f\ \textbf{False}
\end{array}$$

10.2 λ-Definable Functions

Definition 10.4: λ-Definable Functions

$f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is λ-definable if there is a closed λ-term F that represents it:

For all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and $y \in \mathbb{N}$:

- if $f(x_1, \ldots, x_n) = y$, then $F\ \underline{x_1}\ \ldots\ \underline{x_n} =_\beta \underline{y}$
- if $f(x_1, \ldots, x_n)\uparrow$, then $F\ \underline{x_1}\ \ldots\ \underline{x_n}$ has no β-normal form

10.2.1 Representing Basic Functions

Projection $\mathrm{proj}_i^n \in \mathbb{N}^n \to \mathbb{N}$:

$$\lambda x_1\ \ldots x_n.\, x_i$$

Constant (with value 0) $\mathrm{zero}^n \in \mathbb{N}^n \to \mathbb{N}$:

$$\lambda x_1\ \ldots\ x_n.\, \underline{0}$$

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

$$\lambda x'\, f\, x.\, f(x'\ f\ x)$$

10.2.2 Representing Composition

If the total functions $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ with $g_1, \ldots, g_n \in \mathbb{N}^m \to \mathbb{N}$ are represented by F and $G_1, \ldots, G_n$, respectively, then their composition $f \circ (g_1, \ldots, g_n) \in \mathbb{N}^m \to \mathbb{N}$ is represented simply by:

$$\lambda x_1 \ldots, x_m.F\ (G_1\ x_1\ \ldots\ x_m) \ldots (G_n\ x_1\ \ldots\ x_m)$$

$$\begin{aligned} F\ (G_1\ \underline{a_1}\ \ldots\ \underline{a_m}) \ldots (G_n\ \underline{a_1}\ \ldots\ \underline{a_m}) &=_\beta F\ \underline{g_1(a_1, \ldots, a_m)} \ldots \underline{g_n(a_1, \ldots, a_m)} \\ &=_\beta \underline{f\big(g_1(a_1, \ldots, a_m), \ldots, g_n(a_1, \ldots, a_m)\big)} \\ &= \underline{f \circ \big[g_1, \ldots, g_n\big](a_1, \ldots, a_m)} \end{aligned}$$

This does not necessarily work for partial functions. For instance, the totally defined function $u \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is represented by $\boldsymbol{U} \triangleq \lambda x_1.\Omega$ and $\mathrm{zero}^1 \in \mathbb{N} \to \mathbb{N}$ is represented by $\boldsymbol{Z} \triangleq \lambda x_1.\underline{0}$. But $\mathrm{zero}^1 \circ u$ is not represented by $\lambda x_1.\, \boldsymbol{Z}\ (\boldsymbol{U}\ x_1)$, because $\mathrm{zero}^1 \circ u(n)$ is undefined whereas:

$$\begin{aligned} \big(\lambda x_1.\, \boldsymbol{Z}\ (\boldsymbol{U}\ x_1)\big)\ \underline{n} &=_\beta \boldsymbol{Z}\ (\boldsymbol{U}\ \underline{n}) \\ &= \boldsymbol{Z}\ (\lambda x_1.\Omega\ \underline{n}) \\ &=_\beta \boldsymbol{Z}\ \Omega \\ &= \lambda\, x.\, \underline{0}\ \Omega \\ &=_\beta \underline{0} \end{aligned}$$

The composition of partial functions must make sure that each G_i is fully reduced even it does not contribute to the overall result.

One way to achieve this is:

$$(G\ a_1\ \ldots\ a_n)\ I\ I \ =_\beta\ \underline{n_G}\ I\ I \ =_\beta\ I^{n_G}\ I \ =_\beta\ I$$

Which β-reduces if and only if $(G\ a_1\ \ldots\ a_n)$ has β-nf of $\underline{n_G}$, otherwise no β-nf exists and thus cannot reduce and is then not defined.

$$\big((G_1\ \underline{a_1}\ \ldots\ \underline{a_m})\ I\ I\ \ldots (G_n\ \underline{a_1}\ \ldots\ \underline{a_m})\ I\ I\big)\ \big(F\ (G_1\ \underline{a_1}\ \ldots\ \underline{a_m}) \ldots (G_n\ \underline{a_1}\ \ldots\ \underline{a_m})\big)$$

10.2.3 Representing Predecessor

We want a λ-term **Pred** that satisfies:

$$\textbf{Pred}\ \underline{n+1}\ =_\beta\ \underline{n} \qquad \wedge \qquad \textbf{Pred}\ \underline{0}\ =_\beta\ \underline{0}$$

Our strategy is to take a pair $(0,0)$ together with a mapping $f\ :\ (a,b) \mapsto (a+1,a)$. Repeatedly applying f to the pair yields pairs of the form $(n, n-1)$, where n in the first field is the number of iterations and the second field naturally contains the predecessor of n.

$$\textbf{Pred} \triangleq \lambda n\, f\, x.\, \textbf{Snd}\big(n\ (G\ f)\ (\textbf{Pair}\ x\ x)\big)$$

$$G \triangleq \lambda f\, p.\, \textbf{Pair}\big(f\ (\textbf{Fst}\ p)\big)(\textbf{Fst}\ p)$$

Note that $n\ (G\ f)\ P$ in the **Pred** term above, whenever n reduces to a church numeral $\underline{n}$, this becomes $(G\ f)^n\ P$

10.3 Primitive Recursion

If $f \in \mathbb{N}^n \to \mathbb{N}$ is represented by a λ-term F and $g \in \mathbb{N}^{n+2} \to \mathbb{N}$ is represented by a λ-term G, we want to show λ-definability of the unique $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfiying:

$$\begin{cases} h(\vec{a}, 0) & \equiv \quad f(\vec{a}) \\ h(\vec{a}, b+1) & \equiv \quad g(\vec{a}, b, h(\vec{a}, b)) \end{cases}$$

That is, we want to show λ-definability of the unique $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfying

$$h = \Phi_{f,g}(h)$$

Where $\Phi_{f,g} \in \big(\mathbb{N}^{n+1} \to \mathbb{N}\big) \to (\mathbb{N}^{n+1} \to \mathbb{N})$ is given by:

$$\Phi_{f,g}(h)(\vec{a}, b) = \begin{cases} f(\vec{a}) & \text{if } b = 0 \\ g(\vec{a}, b-1, h(\vec{a}, b-1)) & \text{otherwise} \end{cases}$$

Our strategy is to first show that $\Phi_{f,g}$ is λ-definable and then to show that we can solve fixed point equations $(X = M\ X)$ up to β-conversion in the λ-calculus.

10.3.1 Y Combinator

To solve these sort of problems we use the concept of fixed points. It turn out that for any λ-term M there is a λ-term that y such that $M\ y\ =_\beta\ y$. In other words; for each λ-term there is a λ-term that remains entirely unaffacted by applying M. Such a y is called a fixed point.

So, how do we find the fixed point of any given λ-term M? We use a $\mathbf{Y}$ combinator:
A λ-term that, when applied to a λ-term M yields a representation of the fixed point. Hence, with $y = \mathbf{Y}\ M$ we get $M\ (\mathbf{Y}\ M) =_\beta \mathbf{Y}\ M$

To determine the form of $\mathbf{Y}$ we can try to 'solve' for **fact** in the below equation:

$$\textbf{fact} = \lambda n.\, \textbf{If}\ \big(\textbf{Eq}_0\ (n)\big)\ \ (1)\ \ (n \cdot \textbf{fact}\ (n-1))$$

Where multiplication of two values in λ-calculus can be defined below

$$\textbf{mult} = \lambda m\, n\, f\, x.\, m\ (n\ f)\ x$$

To help with 'solving' for fact we will define a $\textbf{fact}'$:

$$\textbf{fact}' = \lambda f\, n.\, \textbf{If}\ \big(\textbf{Eq}_0\ (n)\big)\ \ (1)\ \ (n \cdot f\ (n-1))$$

We want to find a **FIX** such that:

$$\textbf{FIX fact}' \ =\ \textbf{fact}'\ (\textbf{FIX fact}')$$

So that

$$\textbf{fact} \triangleq \textbf{FIX fact}'$$

This is because:

$$\begin{aligned}\mathbf{fact}\ n &= \mathbf{FIX}\ \mathbf{fact}'\ n \\ &= \mathbf{fact}'\ (\mathbf{FIX}\ \mathbf{fact}') \\ &= \big(\mathbf{Eq}_0\ (n)\big)\ \ (1)\ \ (n \cdot \underbrace{(\mathbf{FIX}\ \mathbf{fact}')}_{\mathbf{fact}}\ (n-1))\end{aligned}$$

Next we perform the "Hat Trick"

$$\hat{\mathbf{fact}} = \lambda f\, n.\, \mathbf{If}\ \big(\mathbf{Eq}_0\ (n)\big)\ \ (1)\ \ (n \cdot f\ f\ (n-1))$$

Let $\mathbf{fact} = \hat{\mathbf{fact}}\ \hat{\mathbf{fact}}$ so that:

$$\begin{aligned}\mathbf{fact}\ n &= \hat{\mathbf{fact}}\ \hat{\mathbf{fact}}\ n \\ &=_\beta \mathbf{If}\ \big(\mathbf{Eq}_0\ (n)\big)\ \ (1)\ \ (n \cdot \hat{\mathbf{fact}}\ \hat{\mathbf{fact}}\ (n-1)) \\ &=_\beta \mathbf{If}\ \big(\mathbf{Eq}_0\ (n)\big)\ \ (1)\ \ (n \cdot \mathbf{fact}\ (n-1))\end{aligned}$$

Next we want to find a G such that $G\ \mathbf{fact}' = \hat{\mathbf{fact}}$:

$$G = \lambda g\, f.\, g\ (f\ f)$$

Then

$$\begin{aligned}G\ \mathbf{fact}' &= (\lambda g\, f.\, g\ (f\ f))\mathbf{fact}' \\ &=_\beta \lambda f.\, \mathbf{fact}'\ (f\ f) \\ &= \lambda f\, n.\, \mathbf{If}\ \big(\mathbf{Eq}_0\ (n)\big)\ \ (1)\ \ (n \cdot (f\ f)\ (n-1)) \\ &= \hat{\mathbf{fact}}\end{aligned}$$

So putting all of this together we get

$$\begin{aligned}\mathbf{fact} &= \hat{\mathbf{fact}}\ \hat{\mathbf{fact}} \\ &= (G\ \mathbf{fact}')\ (G\ \mathbf{fact}') \\ &= (\lambda f.\, (G\ f)\ (G\ f))\ \mathbf{fact}' \\ &= \Big(\lambda f.\, \big(\lambda g.\, f\ (g\ g)\big)\ \big(\lambda g.\, f\ (g\ g)\big)\Big)\ \mathbf{fact}' \\ &= \boldsymbol{Y}\ \mathbf{fact}'\end{aligned}$$

This $\boldsymbol{Y}$ combinator satisfies the following:

$$\boldsymbol{Y}\ M \to \big(\lambda x.\, M\ (x\ x)\big)\ \big(\lambda x.\, M\ (x\ x)\big) \to M\ \Big(\big(\lambda x.\, M\ (x\ x)\big)\big(\lambda x.\, M\ (x\ x)\big)\Big)$$

and so:

$$\boldsymbol{Y}\ M \twoheadrightarrow M\ \Big(\big(\lambda x.\, M\ (x\ x)\big)\big(\lambda x.\, M\ (x\ x)\big)\Big) \twoheadleftarrow M\ (\boldsymbol{Y}\ M)$$

Which means for all λ-terms M we have

$$\boldsymbol{Y}\ M\ =_\beta\ M\ (\boldsymbol{Y}\ M)$$

10.3.2 Representing Primitive Recursion

If $f \in \mathbb{N}^n \to \mathbb{N}$ is represented by a λ-term F and $g \in \mathbb{N}^{n+2} \to \mathbb{N}$ is represented by a λ-term G, we want to show λ-definability of the unique $h \in \mathbb{N}^{n+1} \to \mathbb{N}$

$$h(\vec{a}, a) = \Phi_{f,g}(h)(\vec{a}, a) = \begin{cases} f(\vec{a}) & \text{if } a = 0 \\ g(\vec{a}, a-1, h(\vec{a}, a-1)) & \text{otherwise} \end{cases}$$

Using the $\boldsymbol{Y}$ combinator we can show that h can be represented as:

$$\boldsymbol{Y}\ \Big(\lambda z\, \vec{x}\, x.\ \textbf{If}\ (\textbf{Eq}_0\ x)\ (F\ \vec{x})\ \big(G\ \vec{x}\ (\textbf{Pred}\ x)\ (z\ \vec{x}\ (\textbf{Pred}\ x))\big)\Big)$$

10.3.3 Representing Minimisation

We can express $\mu^n f$ in terms of a fixed point equation:

$$\mu^n f(\vec{x}) \equiv g(\vec{x}, 0)$$

Where g satisfies $g = \Psi_f(g)$ with $\Psi_f \in (\mathbb{N}^{n+1} \rightharpoonup \mathbb{N}) \to (\mathbb{N}^{n+1} \rightharpoonup \mathbb{N})$ defined by

$$\Psi_f(g)(\vec{x}, x) \equiv \begin{cases} x & \text{if } f(\vec{x}, x) = 0 \\ g(\vec{x}, x+1) & \text{otherwise} \end{cases}$$

Suppose $f \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfies $\forall \vec{a}\ \exists a.\ \big(f(\vec{a}, a) = 0\big)$, so that $\mu^n f \in \mathbb{N}^n \to \mathbb{N}$

Thus for all $\vec{a} \in \mathbb{N}^n, \mu^n f(\vec{a}) = g(\vec{a}, 0)$ with $g = \Psi_f(g)$ and $\Psi_f(g)(\vec{a}, a)$ is given by

$$\Psi_f(g)(\vec{a}, a) \equiv \begin{cases} a & \text{if } f(\vec{a}, a) = 0 \\ g(\vec{a}, a+1) & \text{otherwise} \end{cases}$$

So if f is represented by a λ-term F, then $\mu^n f$ is represented by:

$$\lambda \vec{x}.\, \boldsymbol{Y}\Big(\lambda z\, \vec{x}\, x.\, \textbf{If}\ (\textbf{Eq}_0\ (F\ \vec{x}\ x))\ x\ (z\ \vec{x}\ (\textbf{Succ}\ x))\Big)\ \vec{x}\ \underline{0}$$

10.4 λ-Definability

Every partial recursive function is λ-definable, with matching $\uparrow$ with that there exists no β-nf makes the representations more complicated.

Theorem 10.1: Computability

A partial function is computable if and only if it is λ-definable.

We have already show that computable means partial recursive and hence is λ-definable. So it remains to see that λ-definablle functions are register machine computable. To show this we could:

- Code λ-terms as numbers (Ensuring that operations for constructing and deconstructing terms are given by register machine computable functions on codes)
- Write a register machine interpreter for (normal order) β-reduction