Imperial College
London

COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Title of course

Author:
Your Name (CID: your college-id number)

Date: March 29, 2018

1 Introduction

1.1 Algorithm

1.2 Hilbert's Entscheidungsproblem

Is there an algorithm which, when fed any statement in the formal language of first-order logic, determines in a finite number of steps whether or not the statement is provable, using the usual rules of first-order logic?

Entscheidungsproblem means 'decision problem'. Given

1. A set S whose elements are finite data structures of some kind

2. a property P of elements of S

the associated decision procedure is to find an algorithm which terminates with result 0 or 1 when given an element $s \in S$, and yields result 1 when given s if and only if s has property P.

People failed to find such an algorithm to solve Hilbert?s Entscheidungsproblem.

A natural question was then to ask whether it was possible to prove that such an algorithm did not exist. To ask this question properly, it was necessary to provide a formal definition of algorithm. Any formal definition of an algorithm should be:

1. **Precise**, with no implicit assumptions, so we know what we are talking about; for maximum precision, it should be phrased in the language of mathematics

2. **Simple** and without extraneous details, so we can reason easily with it

3. **General** so that all algorithms are covered

Turing and Church gave a formal definition to algorithm: Turing invented **Turing machines** and Church invented **lambda calculus**. Then they regarded algorithms as data on which algorithms can act, and reduced the problem to the **Halting Problem**.

1.3 The Halting Problem

The Halting Problem is the decision problem with:

1. the set S of all pairs (A, D), where A is an algorithm and D is some input datum on which the algorithm is designed to operate

2. the property $A(D)\downarrow$ holds for $(A, D) \in S$ if algorithm A when applied to D eventually produces a result: that is, eventually halts.

Turing and Church's work shows that the Halting Problem is unsolvable (undecidable): that is, there is no algorithm H such that, for all $(A, D) \in S$:

$$\begin{aligned} H(A,D) &= 1 && A(D)\downarrow \\ &= 0 && otherwise \end{aligned}$$

Thus there is no such algorithm to Hilbert's Entscheidungsproblem

1.4 Program Semantics

Denotational Semantics: a program?s meaning is described compositionally using mathematical objects (called denotations): the denotation of a program phrase should be built out of denotations of its subphrases.

Operational Semantics: a program?s meaning is given in terms of the steps of computation the program makes when it runs.

2 WHILE Language

The language While of simple while programs has a grammar consisting of three syntactic categories: **numeric expressions**; **booleans**; and **commands**.

2.1 Numeric expressions

The syntax of simple expression is given by:

$E \in SimpleExp ::= n \mid E + E \mid E \times E ...$

Where $n \in \mathbb{N}$ and $+$, $\times$ denote operators to form expressions. We can add more operations if we need to.

An operational semantics for SimpleExp will tell us how to evaluate an expression to get a result. This can be done by **small-step** or **big-step**

2.1.1 Big Steps Semantics

The big step semantics is an operational semantics ignores the intermediate steps and gives the result immediately. In big step semantics, we use the notation $E \Downarrow n$ to mean that expression E evaluates to n.

These are the rules we use to compute simple expressions using big step semantics.

$$\text{(B-NUM)}\ \frac{}{n \Downarrow n}$$

$$\text{(B-ADD)}\ \frac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3}\ n_3 = n_1 \pm n_2$$

A Proof that $3 + (2 + 1) \Downarrow 6$

$$\dfrac{\dfrac{}{3 \Downarrow 3}\,(\text{B-NUM}) \qquad \dfrac{\dfrac{}{2 \Downarrow 2}\,(\text{B-NUM}) \qquad \dfrac{}{1 \Downarrow 1}\,(\text{B-NUM})}{2 + 1 \Downarrow 3}\,(\text{B-ADD})}{3 + (2 + 1) \Downarrow 6}\,(\text{B-ADD})$$

There are two natural properties to big step semantics:

1. **Determinacy**: For all E, n_1 and n_2, if $E \Downarrow n_1$ and $E \Downarrow n_2$ then $n_1 = n_2$. which says that an expression can evaluate to at most one answer.

2. **Totality**: For all E, there exists a n such that $E \Downarrow n$. which says that an expression must evaluate to at least one answer.

2.1.2 Small Steps Semantics

Small step semantics is an operational semantics gives a method for evaluating an expression step-by-step. We define the relationship $E \to E'$ which describes the one step evaluation of expression E.

These are the rules we use to compute simple expressions using small step semantics.

$$(\text{S-LEFT}) \quad \frac{E_1 \to E_1'}{E_1 + E_2 \to E_1' + E_2}$$

$$(\text{S-RIGHT}) \quad \frac{E \to E'}{n + E \to n + E'}$$

$$(\text{S-ADD}) \quad \frac{}{n_1 + n_2 \to n_3}\; n_3 = n_1 \underline{+} n_2$$

These rules say: to evaluate an addition, first evaluate the left-hand argument; when you get to a number, evaluate the right-hand argument; when you get to a number there too, add the two together to get a number.

For example to evaluate the expression $3 + (2 + 1)$. Using Axiom S-RIGHT and S-ADD, we have $3 + (2 + 1) \to 3 + 3$. Then using Axiom S-ADD, we get $3 + 3 = 6$

Given the relation $\to$, we can define $\to^*$ as the reflexive transitive closure of $\to$.

$E \to^* E'$ holds if and only if $E = E'$ or there is a finite sequence $E \to E_1 \to E_2 \to \ldots \to E'$. For the expressions, we say n is the final answer if $E \to^* n$

An Expression is in its **Normal Form** if is it irreducible, there does not exist a E' such

that $E \to E'$.

There are few properties to small step semantics:

1. **Determinacy**: For all E, E_1, E_2, if $E \to E_1$ and $E \to E_2$ then $E_1 = E_2$. Which means that each expression can only evaluate (for one step) in at most one way.

2. **Confluence**: For all E, E_1, E_2, if $E \to^* E_1$ and $E \to^* E_2$ then there exists E' such that $E_1 \to^* E'$ and $E_2 \to^* E'$. which states that if we choose two different evaluation paths for an expression then they can both be extended so that they eventually converge.

3. **Normalisation**: Every evaluation path eventually reaches a normal form.

4. **Theorem**: For all E, n_1, n_2, if $E \to^* n_1$ and $E \to^* n_2$ then $n_1 = n_2$.

2.1.3 States

A **state** is a partial function from variables to numbers such that $s(x)$ is defined for finitely many x. E.g. $s = (x \mapsto 4, y \mapsto 5, z \mapsto 6)$ describes a partial function where variable x has value 4, y has value 5, and z has value 6.

Our small-step semantics for WHILE will be defined using configurations of the form $\langle E, s\rangle$, $\langle B, s\rangle$ and $\langle C, s\rangle$. The idea is that we evaluate E, B and C with respect to state s.

The rules we use to compute simple expressions with states becomes

$$\text{(W-EXP.LEFT)} \quad \frac{\langle E_1, s\rangle \to_e \langle E_1', s'\rangle}{\langle E_1 + E_2, s\rangle \to_e \langle E_1' + E_2, s'\rangle}$$

$$\text{(W-EXP.RIGHT)} \quad \frac{\langle E, s\rangle \to_e \langle E', s'\rangle}{\langle n + E, s\rangle \to_e \langle n + E', s'\rangle}$$

$$\text{(W-EXP.VAR)} \quad \frac{}{\langle x, s\rangle \to_e \langle n, s\rangle}\ s(x) = n$$

$$\text{(W-EXP.ADD)} \quad \frac{}{\langle n_1 + n_2, s\rangle \to_e \langle n_3, s\rangle}\ n_3 = n_1 \underline{+} n_2$$

2.2 Booleans

These are the rules use for evaluating booleans in big step semantics

$$B.TRUE\ \frac{}{\langle true,s,h\rangle \Downarrow_b \langle true,s,h\rangle} \qquad B.NOT.TRUE\ \frac{\langle B,s,h\rangle \Downarrow_b \langle BVal,s',h'\rangle \qquad BVal = false}{\langle \neg B,s,h\rangle \Downarrow_b \langle true,s',h'\rangle}$$

$$B.FALSE\ \frac{}{\langle false,s,h\rangle \Downarrow_b \langle false,s,h\rangle} \qquad B.NOT.FALSE\ \frac{\langle B,s,h\rangle \Downarrow_b \langle BVal,s',h'\rangle \qquad BVal = true}{\langle \neg B,s,h\rangle \Downarrow_b \langle false,s',h'\rangle}$$

$$B.OR.TRUE\ \frac{\langle B_1,s,h\rangle \Downarrow_b \langle BVal_1,s',h'\rangle \qquad \langle B_2,s',h'\rangle \Downarrow_b \langle BVal_2,s",h"\rangle \qquad (BVal_1 = true) \vee (BVal_2 = true)}{\langle B_1 \vee B_2,s,h\rangle \Downarrow_b \langle true,s",h"\rangle}$$

$$B.OR.FALSE\ \frac{\langle B_1,s,h\rangle \Downarrow_b \langle BVal_1,s',h'\rangle \qquad \langle B_2,s',h'\rangle \Downarrow_b \langle BVal_2,s",h"\rangle \qquad (BVal_1 = false) \wedge (BVal_2 = false)}{\langle B_1 \vee B_2,s,h\rangle \Downarrow_b \langle false,s",h"\rangle}$$

This is an example of AND rule in small step semantics

$$\frac{B_1 \to B_1'}{B_1 \,\&\, B_2 \to B_1' \,\&\, B_2}$$

$$\frac{}{\texttt{false} \,\&\, B_2 \to \texttt{false}}$$

$$\frac{}{\texttt{true} \,\&\, B_2 \to B_2}$$

An operation is called **strict** in one of its arguments if it always needs to evaluate that argument. The above definition for AND is a **left-strict operator**, it is **non-strict** in its right argument.

2.3 Commands

The commands in WHILE Language includes:

1. The noop command, $skip$, which does nothing. A configuration $\langle skip, s\rangle$ is said to be an **answer configuration**. Since there is no rule for executing skip, answer configurations are normal forms.

For expressions, the answer configurations are $\langle n, s\rangle$ for number n.

For booleans, the answer configurations are $\langle true, s\rangle$ and $\langle false, s\rangle$.

2. The assignment command, $x := E$

$$\text{(W-ASS.EXP)}\quad \frac{\langle E, s\rangle \to_e \langle E', s'\rangle}{\langle x := E, s\rangle \to_c \langle x := E', s'\rangle}$$

$$\text{(W-ASS.NUM)}\quad \frac{}{\langle x := n, s\rangle \to_c \langle \texttt{skip}, s[x \mapsto n]\rangle}$$

3. The conditional command, $if\ B\ then\ C_1\ else\ C_2$

$$\text{(W-COND.TRUE)}\ \frac{}{\langle \texttt{if true then } C_1 \texttt{ else } C_2, s\rangle \to_c \langle C_1, s\rangle}$$

$$\text{(W-COND.FALSE)}\ \frac{}{\langle \texttt{if false then } C_1 \texttt{ else } C_2, s\rangle \to_c \langle C_2, s\rangle}$$

$$\text{(W-COND.BEXP)}\ \frac{\langle B, s\rangle \to_b \langle B', s'\rangle}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \to_c \langle \texttt{if } B' \texttt{ then } C_1 \texttt{ else } C_2, s'\rangle}$$

4. Sequential composition, $C_1; C_2$

$$\text{(W-SEQ.LEFT)}\ \frac{\langle C_1, s\rangle \to_c \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \to_c \langle C_1'; C_2, s'\rangle}$$

$$\text{(W-SEQ.SKIP)}\ \frac{}{\langle \texttt{skip}; C_2, s\rangle \to_c \langle C_2, s\rangle}$$

5. The loop constructor, *while B do C*

$$\text{(W-WHILE)}\ \frac{}{\langle \texttt{while } B \texttt{ do } C, s\rangle \to_c \langle \texttt{if } B \texttt{ then } (C; \texttt{while } B \texttt{ do } C) \text{ else } \texttt{skip}, s\rangle}$$

2.4 Stuck Configuration

Apart from answer configuration there are other normal forms for configurations, called **stuck configurations**.

For example, $\langle y, (x \mapsto 3)\rangle$ is a normal expression configuration, since y cannot be evaluated in the state $(x \mapsto 3)$.This configuration is stuck.

2.5 Normalisation

The evaluations of expressions and booleans are normalising. However the evaluations of commands are not normalising.

For example, the program (while true do skip) loops forever.

3 Register Machines

A Registered Machine RM is specified by:

1. Finitely many registers $R_0, R_1, R_2 ...$ each capable of storing a natural number

2. A program consisting of a finite list of instructions of the form label : body. The body takes the form:

$R^+ \to L'$	add 1 to contents of register R and jump to instruction labelled L'
$R^- \to L', L''$	if contents of R is > 0, then subtract 1 and jump to L', else jump to L''
$HALT$	stop executing instructions

The **register machine configuration** has the form $c = (l, r_0, r_1, r_2, ..., r_n)$, where l is the current label and r_i stores the content of register R_i.

The **initial configuration** of the register machine is $c_0 = (0, r_0, r_1, r_2, ..., r_n)$.

The **computation** of the register machine is a sequence of configuration $c_0, c_1, c_2....$ Where c_0 is the initial configuration and each c in the sequence determines the next configuration by carrying out the program instruction labelled L_i with registers containing $r_0, ..., r_n$.

For a finite computation, the last configuration is a **halting configuration**. That means, the instruction label is either the HALT instruction or the instruction jumps to a label that is not defined.

3.1 Graphical Representation

Instructions can be represented graphically.

Instruction	Representation
$R^+ \to L$	$R^+ \longrightarrow [L]$
$R^- \to L, L'$	$R^- \nearrow [L]$, $R^- \searrow [L']$
$HALT$	$HALT$
L_0	$START \longrightarrow [L_0]$

Registers

$R_0\ R_1\ R_2$

Program

$L_0 : R_1^- \to L_1, L_2$

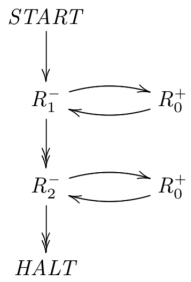
$L_1 : R_0^+ \to L_0$

$L_2 : R_2^- \to L_3, L_4$

$L_3 : R_0^+ \to L_2$

$L_4 : HALT$

Graphical Representation



3.2 Computable Functions

The partial function $f \in \mathbb{N}^n \to \mathbb{N}$ is computable if there is a register machine M with at least $n+1$ registers $R_0, R_1, ..., R_n$ such that for all $(x_1, ..., x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$, the computation of M starting with $R_0 = 0, R_1 = x_1, ..., R_n = x_n$ and all other registers set to 0, halts with $R_0 = y$ if and only if $f(x_1, x_2, ..., x_n) = y$

3.3 Coding Programs as Numbers

3.3.1 Coding Instructions

$$\text{For } x, y \in \mathbb{N}, \text{ define } \begin{cases} \langle\langle x, y \rangle\rangle & \triangleq & 2^x(2y+1) \\ \langle x, y \rangle & \triangleq & 2^x(2y+1) - 1 \end{cases} \qquad \begin{aligned} \ulcorner R_i^+ \to L_j \urcorner &\triangleq \langle\langle 2i, j \rangle\rangle \\ \ulcorner R_i^- \to L_j, L_k \urcorner &\triangleq \langle\langle 2i+1, \langle j, k \rangle \rangle\rangle \\ \ulcorner HALT \urcorner &\triangleq 0 \end{aligned}$$

Any $x \in \mathbb{N}$ decodes to a unique instruction body(x).

If $x = 0$, body(x) is HALT.

Else if $x > 0$, let $x = \langle\langle y, z \rangle\rangle$
If $y = 2i$ is even, body(x) is $R_i^+ \to L_z$
If $y = 2i+1$ is odd, let $z = \langle j, k \rangle$. body(x) is $R_i^- \to L_j, L_k$

3.3.2 Coding List

For $\ell \in List\,\mathbb{N}$, define $\ulcorner \ell \urcorner \in \mathbb{N}$ by induction on the length of the list

$$\ell\text{:} \quad \begin{cases} \ulcorner [] \urcorner & \triangleq & 0 \\ \ulcorner x :: \ell \urcorner & \triangleq & \langle\langle x, \ulcorner \ell \urcorner \rangle\rangle = 2^x(2 \cdot \ulcorner \ell \urcorner + 1) \end{cases}$$

Thus, $\ulcorner [x_1, x_2, \ldots, x_n] \urcorner = \langle\langle x_1, \langle\langle x_2, \cdots \langle\langle x_n, 0 \rangle\rangle \cdots \rangle\rangle \rangle\rangle$

$\ulcorner [3] \urcorner = \ulcorner 3 :: [] \urcorner = \langle\langle 3, 0 \rangle\rangle = 2^3(2 \cdot 0 + 1) = 8$

$\ulcorner [1, 3] \urcorner = \langle\langle 1, \ulcorner [3] \urcorner \rangle\rangle = \langle\langle 1, 8 \rangle\rangle = 34$

$\ulcorner [2, 1, 3] \urcorner = \langle\langle 2, \ulcorner [1, 3] \urcorner \rangle\rangle = \langle\langle 2, 34 \rangle\rangle = 276 =$

A useful property to find out elements of the list is to use its binary representation.

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

$$786432 = 2^{19} + 2^{18} = 0\mathtt{b}11\underbrace{0 \ldots 0}_{18\ \text{"0"}s} = \ulcorner [18, 0] \urcorner$$

3.3.3 Coding Program

Using the coding for list and instructions, any $e \in \mathbb{N}$ decodes to a unique program $prog(e)$, called the register machine **program with index** e.

Where $e = \ulcorner [x_0, x_1, ..., x_n] \urcorner$, and each x_i is an instruction with label L_i.

Notice that when $e = 0$, we have $\ulcorner [] \urcorner$, so $prog(0)$ is a program with an empty list of instructions which by convention we regard as a RM that does nothing.

4 Universal Register Machine

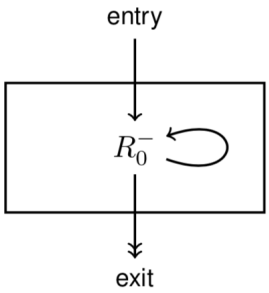
4.1 Gadgets

To construct register machines which perform complex operations, we intro- duce gadgets which are components used to perform specific operations.

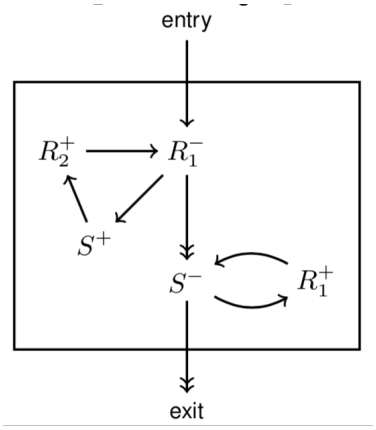
The gadget may use other registers, called scratch registers, for temporary storage.

The gadget assumes the scratch registers are initially set to 0, and must ensure that they are set back to 0 when the gadget exits.

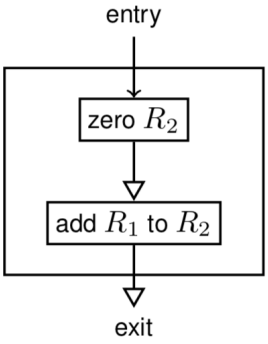
1. Zero R_0: sets register R_0 to be zero, whatever its initial value



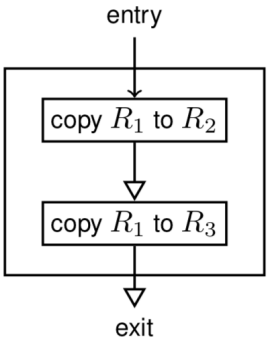
2. Add R_1 to R_2: adds the initial value of R_1 to register R_2, storing the result in R_2 but restoring R_1 to its initial value.



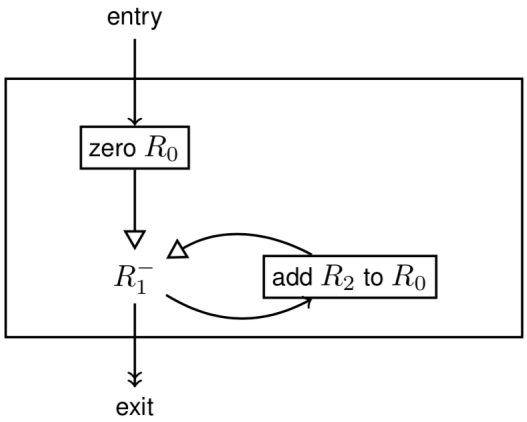
3. Copy R_1 to R_2: copies the value of register R_1 into register R_2, leaving R_1 with its initial value.



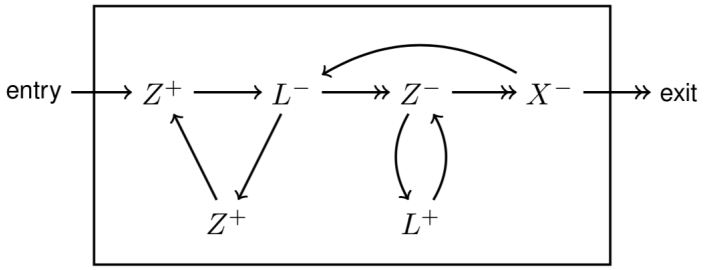
4. Copy R_1 to R_2 and R_3



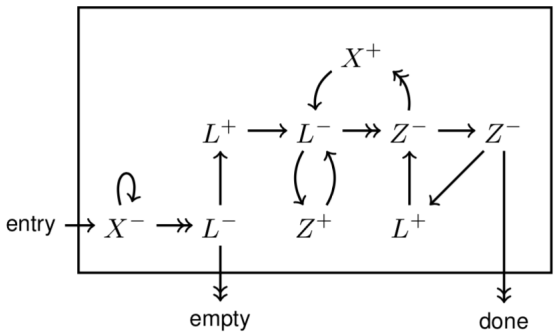
5. Multiply R_1 by R_2 to R_0



6. Push X to L: Given input values $X = x$, $L = l$ and $Z = 0$, it returns the output values $X = 0$, $L = \langle x, l \rangle = 2x(2l + 1)$ and $Z = 0$



7. Pop L to X: If $L = 0$ then return $X = 0$ and go to "empty". If $L = \langle x, l \rangle$ then return $X = x$ and $L = l$, and go to "done".



4.2 The Universal Register Machine

The universal register machine carries out the following computation, 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 RM program P

2. Decode a as a list of register values $a_1, ..., a_n$

3. Carry out the computation of the RM 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).

4.2.1 Overall Structure of Universal Register Machine

Registers used in Universal Register Machine:

R_0 stores the result of the computation

R_1 stores the Program Code (P)

R_2 stores the list of Arguments (A)

R_3 stores the label number of the current instruction, Program Counter (PC)

R_4 stores the label number of the Next instruction (N)

R_5 stores the Code of the current instruction (C)

R_6 stores the value in the Register used in the current instruction (R)

R_7, R_8 are auxiliaries registers

The overall **procedure** of universal register machine:

1. Copy PC item of list in P to N. Goto 2

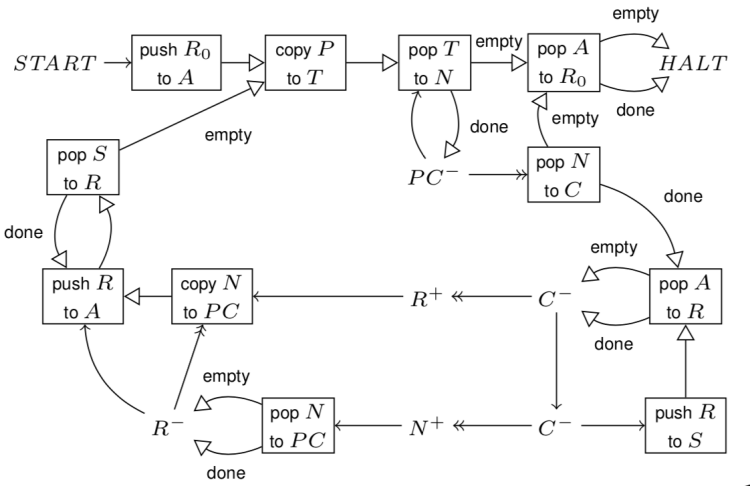
2. Halt if $N = 0$. Otherwise, decode N as $\langle\langle y, z \rangle\rangle$. Store y to C. Store z to N. Goto 3.

Either $C = 2i$ is even and the current instruction $R_i^+ \to L_z$

Or $C = 2i + 1$ is odd and the current instruction is $R_i^- \to L_j, L_k$ where $z = \langle j, k \rangle$

3. Copy the ith item from list A to R. Goto 4.

4. Execute current instruction on R. Update PC to next label. Restore registers values to A. Goto 1.

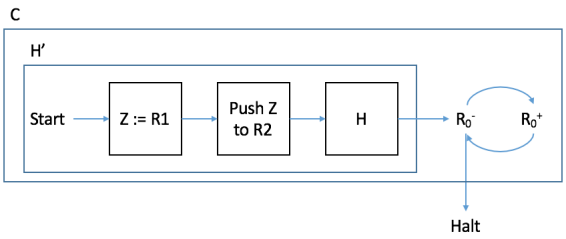


5 Halting Problem For Register Machines

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 halts with $R_0 = 1$ if and only if:

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 registers zeroed.

We want to proof by contradiction that no such register machine H can exist.



Assume we have a RM H that decides the Halting Problem.

Assume that C started with $R_1 = c$ eventually halts.

This means H' started with $R_1 = c$ and halts with $R_0 = 0$

This means H started with $R_1 = c$, $R_2 = \ulcorner[c]\urcorner$ and halts with $R_0 = 0$

From definition of H, this means that $prog(c)$ does not halt

This means C started with $R_1 = c$ does not halt. Hence contradiction.

5.1 Uncomputable Function

For each $e \in \mathbb{N}$, let $\varphi_e \in \mathbb{N} \rightharpoonup \mathbb{N}$ be an unary partial function computed by the RM with program $prog(e)$.

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 $R_0 = y$.

Let $f \in \mathbb{N} \rightharpoonup \mathbb{N}$ be the partial function $\{(x, 0) | \varphi_x(x) \uparrow\}$

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

$f(x) \downarrow$ means $\exists y \in Y(f(x) = y)$

$f(x) \uparrow$ means $\neg\exists y \in Y(f(x) = y)$

f is not computable because if $f = \varphi_e$:

1. $\varphi_e(e) \uparrow$, then $f(e) = 0$. This means $\varphi_e(e) = 0$ hence $\varphi_e(e) \downarrow$.
2. $\varphi_e(e) \downarrow$, then $f(e) \uparrow$. This means $\varphi_e(e) \uparrow$.

In both cases, there is a contradiction.

6 Lambda Calculus

In λ-notation, we have special notation which dispenses with the need to give a name to a function (as in f or add) and easily scales up to more complicated function definitions.

We write $\lambda x.x + x$ for the function which takes in one argument x and returns the result $x + x$. The application of the function $\lambda x.x + x$ to the argument 2 is written $(\lambda x.x + x)\ 2$

Likewise, we write $\lambda xz.x + z$ for the function which takes in two arguments x, z and returns the result $x + z$. The application is then $(\lambda xz.x + z)\ 2\ 3$.

We can also take other functions as input. $(\lambda z.z3)(\lambda y.2 + y)$. This returns the term $(\lambda y.2 + y)\ 3$ and evaluates to 5.

6.1 λ-terms

λ-terms are constructed from a countable set of variables $x, y, z \in Var$ by:

$M ::= x \mid \lambda x.M \mid MM$

The term $\lambda x.M$ is called a λ-abstraction and MM an application. Application is left-associative.

6.2 α equivalence

6.2.1 Bound Variables

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

1. **binding** occurrence if x is in between λ and .
2. **bound** if in the body of a binding occurrence of x
3. **free** if neither binding nor bound

The **set of free variables** $FV(M)$ is defined as:

1. $FV(x) = \{x\}$
2. $FV(\lambda x.M) = FV(M) - \{x\}$
3. $FV(MN) = FV(M) \cup FV(N)$

6.2.2 Substitution

$x\ [y\ /\ x]$	equals	y
$z\ [y\ /\ x]$	equals	z
$(x\ y)(y\ z)\ [y\ /\ x]$	equals	$(y\ y)(y\ z)$
$(\lambda z.xz)\ [y\ /\ x]$	equals	$\lambda z.yz$
$(\lambda x.xy)\ [y\ /\ x]$	equals	$\lambda x.xy$
$(\lambda y.xy)\ [y\ /\ x]$	equals	$\lambda z.yz$

We cannot substitute a bound variable. Shown by the 5th example.

To substitute a bind variable, we must find another unused variables and rename the bound variable before doing the substitution. Shown by the 6th example.

6.2.3 α equivalence

α-equivalence is the binary relation on λ-terms defined by the rules:

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

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

We say M and N are α equivalent, $M =_\alpha N$ if and only if one can be obtained from the other by renaming the bound variables.

If $M =_\alpha N$, then they must have the same set of free variables.

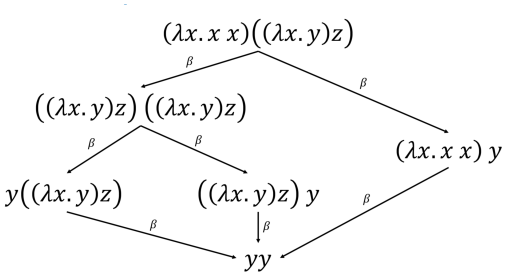
6.3 β-reduction

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

$$\frac{M \to_\beta M'}{M\,N \to_\beta M'\,N} \qquad \frac{N \to_\beta N'}{M\,N \to_\beta M\,N'}$$

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

An example of β-reduction is shown below



6.3.1 Multi-step β-reduction

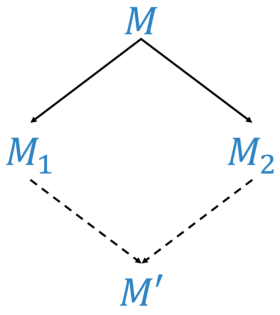
Given a relation $\to_\beta$, we define a new relation $\to_\beta^*$ by:

$$\frac{M =_\alpha M'}{M \to_\beta^* M'} \qquad \frac{M \to_\beta M'' \quad M'' \to_\beta^* M'}{M \to_\beta^* M'}$$

This relation $\to_\beta^*$ is called the **reflexive transitive closure** of $\to_\beta$.

6.3.2 Church-Rosser Theorem

$\to_\beta^*$ is confluent. if $M \to_\beta^* M1$ and $M \to_\beta^* M2$ then there exists M' such that $M1 \to_\beta^* M'$ and $M2 \to_\beta^* M'$.



6.3.3 β normal forms

A λ-term M is in β-normal form if it contains no β-redexes: that is, no subterms of the form $(\lambda x.M1)M2$.

M has β-normal form N if $M \rightarrow_\beta^* N$ and N is in β-normal form.

Uniqueness of β-normal forms. For all M, $N1$, $N2$, if $M \rightarrow_\beta^* N1$ and $M \rightarrow_\beta^* N2$ and $N1$, $N2$ are in β-normal form, then $N1 =_\alpha N2$.

Not all λ-terms have normal form.

6.3.4 *beta* equivalence

$M1 =_\beta M2$ if and only if there exists M such that $M1 \rightarrow_\beta^* M$ and $M2 \rightarrow_\beta^* M$.

6.4 Church's Numerals

$$\underline{n} \stackrel{\text{def}}{=} \lambda f.\lambda x.\ \underbrace{f(\ldots(f}_{n}\ x)\ldots)$$

$$\underline{0} \stackrel{\text{def}}{=} \lambda f.\lambda x.\ x$$
$$\underline{1} \stackrel{\text{def}}{=} \lambda f.\lambda x.\ f\ x$$
$$\underline{2} \stackrel{\text{def}}{=} \lambda f.\lambda x.\ f\ (f\ x)$$
$$\underline{3} \stackrel{\text{def}}{=} \lambda f.\lambda x.\ f\ (f\ (f\ x))$$

6.5 λ-definable functions

A function is λ-definable if there is a closed λ-term F that represents it:

1. If $f(x_1, x_2, \ldots, x_n) = y$ then $F\ x_1\ x_2\ \ldots\ x_n =_\beta y$
2. If $f(x_1, x_2, \ldots, x_n) \uparrow$ then $F\ x_1\ x_2\ \ldots\ x_n$ has no β normal form

Addition is λ-definable

Addition is represented by $P \stackrel{\text{def}}{=} \lambda x_1\ x_2.\lambda f\ x.\ x_1\ f(x_2\ f\ x)$:

$$\begin{aligned} P\ \underline{m}\ \underline{n} &\rightarrow^* \lambda f\ x.\ \underline{m}\ f(\underline{n}\ f\ x) \\ &\rightarrow^* \lambda f\ x.\ \underline{m}\ f(f^n x) \\ &\rightarrow^* \lambda f\ x.\ f^m(f^n x) \\ &\stackrel{\text{def}}{=} \lambda f\ x.f^{m+n} x \\ &\stackrel{\text{def}}{=} \underline{m+n} \end{aligned}$$

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