Computation Theory

12 lectures for
University of Cambridge
2022 Computer Science Tripos, Part IB
by Prof. Andrew Pitts

Introduction

Algorithmically undecidable problems

Computers cannot solve all mathematical problems, even if they are given unlimited time and working space.

Three famous examples of computationally unsolvable problems are sketched in this lecture.

- Hilbert's *Entscheidungsproblem*
- The Halting Problem
- Hilbert's 10th Problem.

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?

Such an algorithm would be useful! For example, by running it on

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

(where $prime(p)$ is a suitable arithmetic statement that p is a prime number) we could solve *Goldbach's Conjecture* ("every even integer strictly greater than two is the sum of two primes"), a famous open problem in number theory.

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?

Posed by Hilbert at the 1928 International Congress of Mathematicians. The problem was actually stated in a more ambitious form, with a more powerful formal system in place of first-order logic.

In 1928, Hilbert believed that such an algorithm could be found. A few years later he was proved wrong by the work of Church and Turing in 1935/36, as we will see.

Decision problems

Entscheidungsproblem means “decision problem”. Given

- a set S whose elements are finite data structures of some kind

  (e.g. formulas of first-order arithmetic)
- a property P of elements of S

  (e.g. property of a formula that it has a proof)

the associated decision problem is:

find an algorithm which
terminates with result $\mathbf{0}$ or $\mathbf{1}$ when fed an element $s \in S$
and
yields result $\mathbf{1}$ when fed s if and only if s has property P.

Algorithms, informally

No precise definition of “algorithm” at the time Hilbert posed the *Entscheidungsproblem*, just examples, such as:

- Procedure for multiplying numbers in decimal place notation.
- Procedure for extracting square roots to any desired accuracy.
- Euclid’s algorithm for finding highest common factors.

Algorithms, informally

No precise definition of "algorithm" at the time Hilbert posed the *Entscheidungsproblem*, just examples.

Common features of the examples:

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

Algorithms, informally

No precise definition of "algorithm" at the time Hilbert posed the *Entscheidungsproblem*, just examples.

Common features of the examples:

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

e.g. multiply two decimal digits by looking up their product in a table

Algorithms, informally

No precise definition of “algorithm” at the time Hilbert posed the *Entscheidungsproblem*, just examples.

In 1935/36 Turing in Cambridge and Church in Princeton independently gave negative solutions to Hilbert’s *Entscheidungsproblem*.

- First step: give a precise, mathematical definition of “algorithm”.

  (Turing: Turing Machines; Church: lambda-calculus.)
- Then one can regard algorithms as data on which algorithms can act and reduce the problem to...

The Halting Problem

is the decision problem with

- 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;
- property P holds for (A, D) if algorithm A when applied to datum D eventually produces a result (that is, eventually halts—we write $A(D)\downarrow$ to indicate this).

The Halting Problem

is the decision problem with

- 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;
- property P holds for (A, D) if algorithm A when applied to datum D eventually produces a result (that is, eventually halts—we write $A(D)\downarrow$ to indicate this).

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

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

There's no H such that $H(A, D) = \begin{cases} 1 & \text{if } A(D)\downarrow \\ 0 & \text{otherwise.} \end{cases}$ for all (A, D).

Informal proof, by contradiction. If there were such an H, let C be the algorithm:

> "input A; compute $H(A, A)$; if $H(A, A) = 0$ then return 1, else loop forever."

There's no H such that $H(A,D) = \begin{cases} 1 & \text{if } A(D){\downarrow} \\ 0 & \text{otherwise.} \end{cases}$ for all (A,D).

Informal proof, by contradiction. If there were such an H, let C be the algorithm:

> "input A; compute $H(A,A)$; if $H(A,A) = 0$ then return 1, else loop forever."

So $\forall A\,(C(A){\downarrow} \leftrightarrow H(A,A) = 0)$ (since H is total)

and $\forall A\,(H(A,A) = 0 \leftrightarrow \neg A(A){\downarrow})$ (definition of H).

There's no H such that $H(A,D) = \begin{cases} 1 & \text{if } A(D)\downarrow \\ 0 & \text{otherwise.} \end{cases}$ for all (A,D).

Informal proof, by contradiction. If there were such an H, let C be the algorithm:

> "input A; compute $H(A,A)$; if $H(A,A) = 0$ then return 1, else loop forever."

So $\forall A\,(C(A)\downarrow \leftrightarrow H(A,A) = 0)$ (since H is total)

and $\forall A\,(H(A,A) = 0 \leftrightarrow \neg A(A)\downarrow)$ (definition of H).

So $\forall A\,(C(A)\downarrow \leftrightarrow \neg A(A)\downarrow)$.

There's no H such that $H(A,D) = \begin{cases} 1 & \text{if } A(D){\downarrow} \\ 0 & \text{otherwise.} \end{cases}$ for all (A,D).

Informal proof, by contradiction. If there were such an H, let C be the algorithm:

> "input A; compute $H(A,A)$; if $H(A,A) = 0$ then return 1, else loop forever."

So $\forall A\,(C(A){\downarrow} \leftrightarrow H(A,A) = 0)$ (since H is total)

and $\forall A\,(H(A,A) = 0 \leftrightarrow \neg A(A){\downarrow})$ (definition of H).

So $\forall A\;(C(A){\downarrow} \leftrightarrow \neg A(A){\downarrow})$.

Taking A to be C, we get $C(C){\downarrow} \leftrightarrow \neg C(C){\downarrow}$, contradiction!

There's no H such that $H(A,D) = \begin{cases} 1 & \text{if } A(D)\downarrow \\ 0 & \text{otherwise.} \end{cases}$ for all (A,D).

Informal proof, by contradiction. If there were such an H, let C be the algorithm:

> "input A; compute $H(A,A)$; if $H(A,A) = 0$ then return 1, else loop forever."

So $\forall A\,(C(A)\downarrow \leftrightarrow H(A,A) = 0)$ (since H is total)

and $\forall A\,(H(A,A) = 0 \leftrightarrow \neg A(A)\downarrow)$ (definition of H).

So $\forall A\,(C(A)\downarrow \leftrightarrow \neg A(A)\downarrow)$.

Taking A to be C, we get $C(C)\downarrow \leftrightarrow \neg C(C)\downarrow$,

contradiction!

$p \leftrightarrow \neg p$ is always false

There's no H such that $H(A,D) = \begin{cases} 1 & \text{if } A(D){\downarrow} \\ 0 & \text{otherwise.} \end{cases}$ for all (A,D).

Informal proof, by contradictio n H, let C be the algorithm:

> "input A; compute $H(A,A)$; if $H(A,A) = 0$ then return 1, else loop forever."

why is A "a datum on which A is designed to operate"?

So $\forall A\,(C(A){\downarrow} \leftrightarrow H(A,A) = 0)$ (since H is total)

and $\forall A\,(H(A,A) = 0 \leftrightarrow \neg A(A){\downarrow})$ (definition of H).

So $\forall A\,(C(A){\downarrow} \leftrightarrow \neg A(A){\downarrow})$.

Taking A to be C, we get $C(C){\downarrow} \leftrightarrow \neg C(C){\downarrow}$, contradiction!

From HP to *Entscheidungsproblem*

Final step in Turing/Church proof of undecidability of the *Entscheidungsproblem*: they constructed an algorithm encoding instances (A, D) of the Halting Problem as arithmetic statements $\Phi_{A,D}$ with the property

$$\Phi_{A,D} \textit{ is provable } \leftrightarrow A(D){\downarrow}$$

Thus any algorithm deciding provability of arithmetic statements could be used to decide the Halting Problem—so no such exists.

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?

With hindsight, a positive solution to the *Entscheidungsproblem* would be too good to be true. However, the algorithmic unsolvability of some decision problems is much more surprising. A famous example of this is. . .

Hilbert's 10th Problem

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

One of a number of important open problems listed by Hilbert at the International Congress of Mathematicians in 1900.

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 $\mathbb{N} = \{0, 1, 2, \ldots\}$.

Named after Diophantus of Alexandria (c. 250AD).

Example: “find three whole numbers x_1, x_2 and x_3 such that the product of any two added to the third is a square”
[Diophantus’ *Arithmetica*, Book III, Problem 7].

In modern notation: find $x_1, x_2, x_3 \in \mathbb{N}$ for which there exists $x, y, z \in \mathbb{N}$ with

$$(x_1x_2 + x_3 - x^2)^2 + (x_2x_3 + x_1 - y^2)^2 + (x_3x_1 + x_2 - z^2)^2 = 0$$

Diophantine equations

$$p(x_1, \ldots, x_n) = q(x_1, \ldots, x_n)$$

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

Named after Diophantus of Alexandria (c. 250AD).

Example: "find three whole numbers x_1, x_2 and x_3 such that the product of any two added to the third is a square"
[Diophantus' *Arithmetica*, Book III, Problem 7].

In modern notation: find $x_1, x_2, x_3 \in \mathbb{N}$ for which there exists $x, y, z \in \mathbb{N}$ with

$$x_1^2 x_2^2 + x_2^2 x_3^2 + x_3^2 x_1^2 + \cdots = x^2 x_1 x_2 + y^2 x_2 x_3 + z^2 x_3 x_1 + \cdots$$

[One solution: $(x_1, x_2, x_3) = (1, 4, 12)$, with $(x, y, z) = (4, 7, 4)$.]

Hilbert's 10th Problem

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

- Posed in 1900, but only solved in 1970: Y Matijasevič, J Robinson, M Davis and H Putnam show it undecidable by reduction from the Halting Problem.

Hilbert's 10th Problem

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

- Posed in 1900, but only solved in 1970: Y Matijasevič, J Robinson, M Davis and H Putnam show it undecidable by reduction from the Halting Problem.
- Original proof used Turing machines. Later, simpler proof [JP Jones & Y Matijasevič, J. Symb. Logic 49(1984)] used Minsky and Lambek's register machines

Hilbert's 10th Problem

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

- Posed in 1900, but only solved in 1970: Y Matijasevič, J Robinson, M Davis and H Putnam show it undecidable by reduction from the Halting Problem.
- Original proof used Turing machines. Later, simpler proof [JP Jones & Y Matijasevič, J. Symb. Logic 49(1984)] used Minsky and Lambek's register machines—we will use them in this course to begin with and return to Turing and Church's formulations of the notion of "algorithm" later.

Register machines

Algorithms, informally

No precise definition of “algorithm” at the time Hilbert posed the *Entscheidungsproblem*, just examples.

Common features of the examples:

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

Register Machines, informally

They operate on natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ stored in (idealized) registers using the following "elementary operations":

- add 1 to the contents of a register
- test whether the contents of a register is 0
- subtract 1 from the contents of a register if it is non-zero
- jumps ("goto")
- conditionals ("if_then_else_")

Definition. A register machine is specified by:

- finitely many registers $\mathtt{R}_0$, $\mathtt{R}_1$, . . . , $\mathtt{R}_n$
  (each capable of storing a natural number);
- a program consisting of a finite list of instructions of the form $label : body$, where for $i = 0, 1, 2, \ldots$, the $(i+1)^{\text{th}}$ instruction has label $\mathtt{L}_i$.

Definition. A register machine is specified by:

- finitely many registers $\mathtt{R}_0$, $\mathtt{R}_1$, . . . , $\mathtt{R}_n$ (each capable of storing a natural number);
- a program consisting of a finite list of instructions of the form $label : body$, where for $i = 0, 1, 2, \ldots$, the $(i+1)^{\text{th}}$ instruction has label $\mathtt{L}_i$.

Instruction body takes one of three forms:

$R^+ \to L'$	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 from it and jump to L', else jump to L''
$\mathtt{HALT}$	stop executing instructions

Example

registers:

$R_0\ R_1\ R_2$

program:

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

$L_1 : R_0^+ \to L_0$

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

$L_3 : R_0^+ \to L_2$

$L_4 : \mathtt{HALT}$

example computation:

L_i	R_0	R_1	R_2
0	**0**	**1**	**2**

Example

registers:

$\mathtt{R}_0\ \mathtt{R}_1\ \mathtt{R}_2$

program:

$\mathtt{L}_0 : \mathtt{R}_1^- \to \mathtt{L}_1, \mathtt{L}_2$

$\mathtt{L}_1 : \mathtt{R}_0^+ \to \mathtt{L}_0$

$\mathtt{L}_2 : \mathtt{R}_2^- \to \mathtt{L}_3, \mathtt{L}_4$

$\mathtt{L}_3 : \mathtt{R}_0^+ \to \mathtt{L}_2$

$\mathtt{L}_4 : \mathtt{HALT}$

example computation:

$\mathtt{L}_i$	$\mathtt{R}_0$	$\mathtt{R}_1$	$\mathtt{R}_2$
0	0	1	2

Example

registers:

$\mathtt{R_0}\ \mathtt{R_1}\ \mathtt{R_2}$

program:

$\mathtt{L_0} : \mathtt{R_1^-} \to \mathtt{L_1}, \mathtt{L_2}$

$\mathtt{L_1} : \mathtt{R_0^+} \to \mathtt{L_0}$

$\mathtt{L_2} : \mathtt{R_2^-} \to \mathtt{L_3}, \mathtt{L_4}$

$\mathtt{L_3} : \mathtt{R_0^+} \to \mathtt{L_2}$

$\mathtt{L_4} : \mathtt{HALT}$

example computation:

L_i	R_0	R_1	R_2
0	0	1	2
1	0	0	2

Example

registers:

R_0 R_1 R_2

program:

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

$L_1 : R_0^+ \to L_0$

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

$L_3 : R_0^+ \to L_2$

$L_4 : \mathtt{HALT}$

example computation:

L_i	R_0	R_1	R_2
0	0	1	2
1	0	0	2
0	1	0	2

Example

registers:

$R_0 \; R_1 \; R_2$

program:

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

$L_1 : R_0^+ \to L_0$

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

$L_3 : R_0^+ \to L_2$

$L_4 : \mathtt{HALT}$

example computation:

L_i	R_0	R_1	R_2
0	0	1	2
1	0	0	2
0	1	0	2
2	1	0	2

Example

registers:

$\mathtt{R}_0\ \mathtt{R}_1\ \mathtt{R}_2$

program:

$\mathtt{L}_0 : \mathtt{R}_1^- \to \mathtt{L}_1, \mathtt{L}_2$

$\mathtt{L}_1 : \mathtt{R}_0^+ \to \mathtt{L}_0$

$\mathtt{L}_2 : \mathtt{R}_2^- \to \mathtt{L}_3, \mathtt{L}_4$

$\mathtt{L}_3 : \mathtt{R}_0^+ \to \mathtt{L}_2$

$\mathtt{L}_4 : \mathtt{HALT}$

example computation:

$\mathtt{L}_i$	$\mathtt{R}_0$	$\mathtt{R}_1$	$\mathtt{R}_2$
0	0	1	2
1	0	0	2
0	1	0	2
2	1	0	2
3	1	0	1

Example

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

example computation:

L_i	R_0	R_1	R_2
0	0	1	2
1	0	0	2
0	1	0	2
2	1	0	2
3	1	0	1
2	2	0	1

Example

registers:

$\mathtt{R}_0\ \mathtt{R}_1\ \mathtt{R}_2$

program:

$\mathtt{L}_0 : \mathtt{R}_1^- \to \mathtt{L}_1, \mathtt{L}_2$

$\mathtt{L}_1 : \mathtt{R}_0^+ \to \mathtt{L}_0$

$\mathtt{L}_2 : \mathtt{R}_2^- \to \mathtt{L}_3, \mathtt{L}_4$

$\mathtt{L}_3 : \mathtt{R}_0^+ \to \mathtt{L}_2$

$\mathtt{L}_4 : \mathtt{HALT}$

example computation:

$\mathtt{L}_i$	$\mathtt{R}_0$	$\mathtt{R}_1$	$\mathtt{R}_2$
0	0	1	2
1	0	0	2
0	1	0	2
2	1	0	2
3	1	0	1
2	2	0	1
3	2	0	0

Example

registers:

R_0 R_1 R_2

program:

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

$L_1 : R_0^+ \to L_0$

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

$L_3 : R_0^+ \to L_2$

$L_4 : \mathtt{HALT}$

example computation:

L_i	R_0	R_1	R_2
0	0	1	2
1	0	0	2
0	1	0	2
2	1	0	2
3	1	0	1
2	2	0	1
3	2	0	0
2	3	0	0

Example

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

example computation:

L_i	R_0	R_1	R_2
0	0	1	2
1	0	0	2
0	1	0	2
2	1	0	2
3	1	0	1
2	2	0	1
3	2	0	0
2	3	0	0
4	3	0	0

Register machine computation

Register machine configuration:

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

where ℓ = current label and r_i = current contents of $\mathtt{R}_i$.

Notation: "$\mathtt{R}_i = x$ [in configuration c]" means $c = (\ell, r_0, \dots, r_n)$ with $r_i = x$.

Register machine computation

Register machine configuration:

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

where ℓ = current label and r_i = current contents of $\mathtt{R}_i$.

Notation: "$\mathtt{R}_i = x$ [in configuration c]" means $c = (\ell, r_0, \dots, r_n)$ with $r_i = x$.

Initial configurations:

$$c_0 = (0, r_0, \dots, r_n)$$

where r_i = initial contents of register $\mathtt{R}_i$.

Register machine computation

A computation of a RM is a (finite or infinite) sequence of configurations

$$c_0, c_1, c_2, \ldots$$

where

- $c_0 = (0, r_0, \ldots, r_n)$ is an initial configuration
- each $c = (\ell, r_0, \ldots, r_n)$ in the sequence determines the next configuration in the sequence (if any) by carrying out the program instruction labelled $\mathtt{L}_\ell$ with registers containing r_0,. . . ,r_n.

Halting

For a finite computation $c_0, c_1, \ldots, c_m$, the last configuration $c_m = (\ell, r, \ldots)$ must be a halting configuration, i.e. ℓ must satisfy:

either ℓ^{th} instruction in program has body `HALT` (a "proper halt")

or ℓ is greater than the number of instructions in program, so that there is no instruction labelled $\mathtt{L}_\ell$ (an "erroneous halt")

Halting

For a finite computation $c_0, c_1, \ldots, c_m$, the last configuration $c_m = (\ell, r, \ldots)$ must be a halting configuration, i.e. ℓ must satisfy:

either ℓ^{th} instruction in program has body HALT (a "proper halt")

or ℓ is greater than the number of instructions in program, so that there is no instruction labelled $\mathtt{L}_\ell$ (an "erroneous halt")

E.g. $\boxed{\begin{array}{l} \mathtt{L}_0 : \mathtt{R}_0^+ \to \mathtt{L}_2 \\ \mathtt{L}_1 : \mathtt{HALT} \end{array}}$ halts erroneously.

Halting

For a finite computation $c_0, c_1, \ldots, c_m$, the last configuration $c_m = (\ell, r, \ldots)$ must be a halting configuration, i.e. ℓ must satisfy:

either ℓ^{th} instruction in program has body `HALT` (a "proper halt")

or ℓ is greater than the number of instructions in program, so that there is no instruction labelled $\mathtt{L}_\ell$ (an "erroneous halt")

N.B. can always modify programs (without affecting their computations) to turn all erroneous halts into proper halts by adding extra `HALT` instructions to the list with appropriate labels.

Halting

For a finite computation $c_0, c_1, \ldots, c_m$, the last configuration $c_m = (\ell, r, \ldots)$ must be a halting configuration.

Note that computations may never halt. For example,

$$\boxed{\begin{array}{l} \mathtt{L}_0 : \mathtt{R}_0^+ \to \mathtt{L}_0 \\ \mathtt{L}_1 : \mathtt{HALT} \end{array}}$$

only has infinite computation sequences

$$(0, r), (0, r+1), (0, r+2), \ldots$$

Graphical representation

- one node in the graph for each instruction
- arcs represent jumps between instructions
- lose sequential ordering of instructions—so need to indicate initial instruction with START.

instruction	representation
$R^+ \to L$	$R^+ \longrightarrow [L]$
$R^- \to L, L'$	$\begin{tikzcd} & {[L]} \\ R^- \arrow[ur] \arrow[dr, two heads] & \\ & {[L']} \end{tikzcd}$
HALT	HALT
L_0	$\mathrm{START} \longrightarrow [\mathrm{L}_0]$

Example

registers:

$\mathtt{R}_0\ \mathtt{R}_1\ \mathtt{R}_2$

program:

$\mathtt{L}_0 : \mathtt{R}_1^- \to \mathtt{L}_1, \mathtt{L}_2$

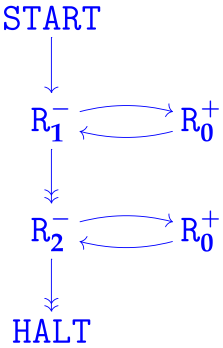
$\mathtt{L}_1 : \mathtt{R}_0^+ \to \mathtt{L}_0$

$\mathtt{L}_2 : \mathtt{R}_2^- \to \mathtt{L}_3, \mathtt{L}_4$

$\mathtt{L}_3 : \mathtt{R}_0^+ \to \mathtt{L}_2$

$\mathtt{L}_4 : \mathtt{HALT}$

graphical representation:



Example

registers:

R_0 R_1 R_2

program:

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

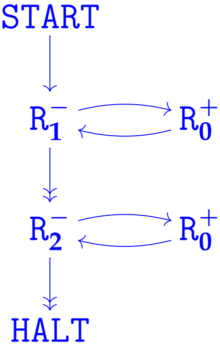
$L_1 : R_0^+ \to L_0$

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

$L_3 : R_0^+ \to L_2$

$L_4 : \mathtt{HALT}$

graphical representation:



Claim: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(4, x + y, 0, 0)$.

Example

registers:

$R_0 \; R_1 \; R_2$

program:

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

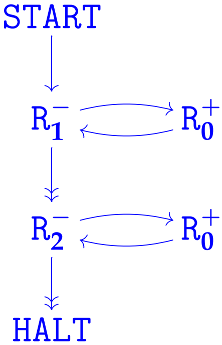
$L_1 : R_0^+ \to L_0$

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

$L_3 : R_0^+ \to L_2$

$L_4 : \mathtt{HALT}$

graphical representation:



Claim: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(4, x + y, 0, 0)$.

Example

registers:

$\mathtt{R}_0\ \mathtt{R}_1\ \mathtt{R}_2$

program:

$\mathtt{L}_0 : \mathtt{R}_1^- \to \mathtt{L}_1, \mathtt{L}_2$

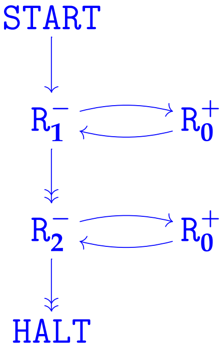
$\mathtt{L}_1 : \mathtt{R}_0^+ \to \mathtt{L}_0$

$\mathtt{L}_2 : \mathtt{R}_2^- \to \mathtt{L}_3, \mathtt{L}_4$

$\mathtt{L}_3 : \mathtt{R}_0^+ \to \mathtt{L}_2$

$\mathtt{L}_4 : \mathtt{HALT}$

graphical representation:



Claim: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(4, x + y, 0, 0)$.

Partial functions

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. . .

Partial functions

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

$$(x, y) \in f \wedge (x, y') \in f \rightarrow y = y'$$

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

Partial functions

ordered pairs $\{(x, y) \mid x \in X \wedge y \in Y\}$

i.e. for all $x \in X$ there is at most one $y \in Y$ with $(x, y) \in f$

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

$$(x, y) \in f \wedge (x, y') \in f \rightarrow y = y'$$

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

Partial functions

Notation:

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

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

$$(x, y) \in f \wedge (x, y') \in f \rightarrow y = y'$$

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

Partial functions

Notation:

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

Definition. A partial function from a set X to a set Y is total if it satisfies

$$f(x)\downarrow$$

for all $x \in X$.

Computable functions

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is (register machine) computable if there is a register machine M with at least $n+1$ registers R_0, R_1, . . . , R_n (and maybe more) such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ 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 $f(x_1, \ldots, x_n) = y$.

Note the [somewhat arbitrary] I/O convention: in the initial configuration registers R_1, . . . , R_n store the function's arguments (with all others zeroed); and in the halting configuration register R_0 stores it's value (if any).

Computable functions

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is (register machine) computable if there is a register machine M with at least $n+1$ registers R_0, R_1, . . . , R_n (and maybe more) such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ 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 $f(x_1, \ldots, x_n) = y$.

N.B. there may be many different M that compute the same partial function f.

Example

registers:

$\mathtt{R}_0\ \mathtt{R}_1\ \mathtt{R}_2$

program:

$\mathtt{L}_0 : \mathtt{R}_1^- \to \mathtt{L}_1, \mathtt{L}_2$

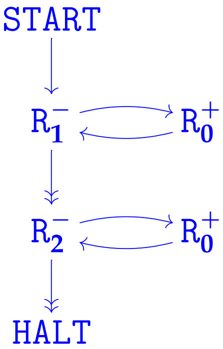
$\mathtt{L}_1 : \mathtt{R}_0^+ \to \mathtt{L}_0$

$\mathtt{L}_2 : \mathtt{R}_2^- \to \mathtt{L}_3, \mathtt{L}_4$

$\mathtt{L}_3 : \mathtt{R}_0^+ \to \mathtt{L}_2$

$\mathtt{L}_4 : \mathtt{HALT}$

graphical representation:



Claim: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(4, x+y, 0, 0)$. So $f(x, y) \triangleq x + y$ is computable.

Computable functions

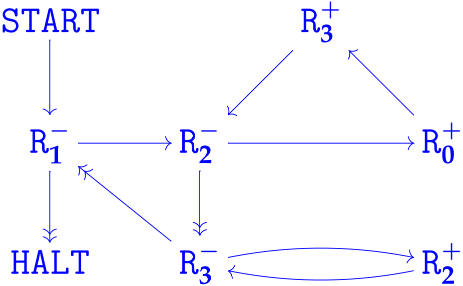
Recall:

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is (register machine) computable if there is a register machine M with at least $n + 1$ registers R_0, R_1, . . . , R_n (and maybe more) such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$,

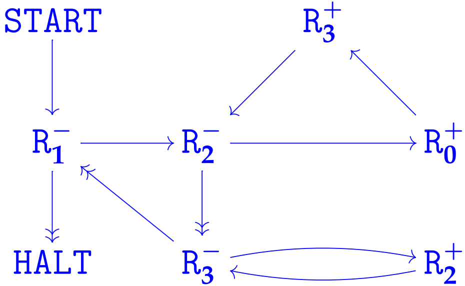
> the computation of M starting with $\mathrm{R}_0 = 0$, $\mathrm{R}_1 = x_1$, . . . , $\mathrm{R}_n = x_n$ and all other registers set to 0, halts with $\mathrm{R}_0 = y$

if and only if $f(x_1, \ldots, x_n) = y$.

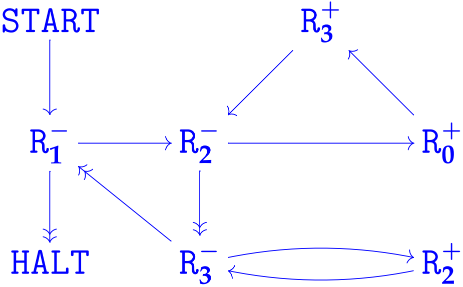
Multiplication $f(x,y) \triangleq xy$ is computable



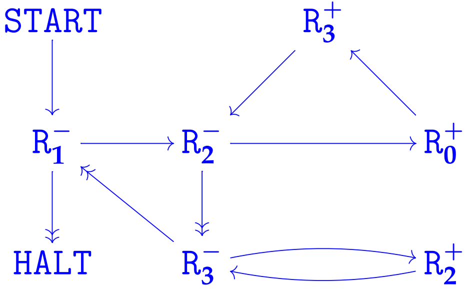
Multiplication $f(x,y) \triangleq xy$ is computable



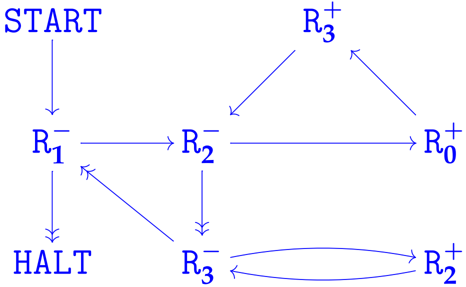
Multiplication $f(x, y) \triangleq xy$ is computable



Multiplication $f(x,y) \triangleq xy$ is computable



Multiplication $f(x, y) \triangleq xy$ is computable



If the machine is started with $(R_0, R_1, R_2, R_3) = (0, x, y, 0)$, it halts with $(R_0, R_1, R_2, R_3) = (xy, 0, y, 0)$.

Further examples

The following arithmetic functions are all computable. (Proof—left as an exercise!)

projection: $p(x, y) \triangleq x$

constant: $c(x) \triangleq n$

truncated subtraction: $x \dot{-} y \triangleq \begin{cases} x - y & \text{if } y \le x \\ 0 & \text{if } y > x \end{cases}$

Further examples

The following arithmetic functions are all computable. (Proof—left as an exercise!)

integer division:

$$x \, \mathit{div} \, y \triangleq \begin{cases} \mathit{integer\ part\ of\ } x/y & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases}$$

integer remainder: $x \, \mathit{mod} \, y \triangleq x \dot{-} y(x \, \mathit{div} \, y)$

exponentiation base 2: $e(x) \triangleq 2^x$

logarithm base 2:

$$\log_2(x) \triangleq \begin{cases} \mathit{greatest\ } y \mathit{\ such\ that\ } 2^y \leq x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

Coding programs as numbers

Turing/Church solution of the Entscheidungsproblem uses the idea that (formal descriptions of) algorithms can be the data on which algorithms act.

To realize this idea with Register Machines we have to be able to code RM programs as numbers. (In general, such codings are often called Gödel numberings.)

To do that, first we have to code pairs of numbers and lists of numbers as numbers. There are many ways to do that. We fix upon one. . .

Numerical coding of pairs

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

So

$$\begin{aligned} \boxed{0\mathrm{b}\langle\!\langle x, y \rangle\!\rangle} &= \begin{array}{|c|c|c|}\hline 0\mathrm{b}y & 1 & 0\cdots0 \\ \hline \end{array} \\ \boxed{0\mathrm{b}\langle x, y \rangle} &= \begin{array}{|c|c|c|}\hline 0\mathrm{b}y & 0 & 1\cdots1 \\ \hline \end{array} \end{aligned}$$

(Notation: $0\mathrm{b}x \triangleq x\ \textit{in binary}$.)

E.g. $27 = 0\mathrm{b}11011 = \langle\!\langle 0, 13 \rangle\!\rangle = \langle 2, 3 \rangle$

Numerical coding of pairs

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

So

$$\begin{array}{|c|}\hline 0\mathrm{b}\langle\!\langle x, y \rangle\!\rangle \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline 0\mathrm{b}y & 1 & 0\cdots0 \\ \hline\end{array}$$

$$\begin{array}{|c|}\hline 0\mathrm{b}\langle x, y \rangle \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline 0\mathrm{b}y & 0 & 1\cdots1 \\ \hline\end{array}$$

$\langle -, - \rangle$ gives a bijection (one-one correspondence) between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$.

$\langle\!\langle -, - \rangle\!\rangle$ gives a bijection between $\mathbb{N} \times \mathbb{N}$ and $\{n \in \mathbb{N} \mid n \neq 0\}$.

Numerical coding of lists

$\mathit{list}\,\mathbb{N} \triangleq$ set of all finite lists of natural numbers, using ML notation for lists:

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

Numerical coding of lists

$\mathit{list}\,\mathbb{N} \triangleq$ set of all finite lists of natural numbers, using ML notation for lists.

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

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

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

Numerical coding of lists

$\mathit{list}\,\mathbb{N} \triangleq$ set of all finite lists of natural numbers, using ML notation for lists.

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

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

$$\boxed{0\mathrm{b}\ulcorner[x_1, x_2, \ldots, x_n]\urcorner} = \begin{array}{|c|c|}\hline \mathbf{1} & \mathbf{0}\cdots\mathbf{0} \\ \hline\end{array}\ \begin{array}{|c|c|}\hline \mathbf{1} & \mathbf{0}\cdots\mathbf{0} \\ \hline\end{array}\cdots\begin{array}{|c|c|}\hline \mathbf{1} & \mathbf{0}\cdots\mathbf{0} \\ \hline\end{array}$$

Numerical coding of lists

$\mathit{list}\,\mathbb{N} \triangleq$ set of all finite lists of natural numbers, using ML notation for lists.

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

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

$$\boxed{0\mathrm{b}\ulcorner[x_1, x_2, \ldots, x_n]\urcorner} = \boxed{\mathbf{1}}\boxed{\mathbf{0}\cdots\mathbf{0}}\ \boxed{\mathbf{1}}\boxed{\mathbf{0}\cdots\mathbf{0}}\cdots\boxed{\mathbf{1}}\boxed{\mathbf{0}\cdots\mathbf{0}}$$

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

Numerical coding of lists

$\mathit{list}\,\mathbb{N} \triangleq$ set of all finite lists of natural numbers, using ML notation for lists.

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

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

For example:

$\ulcorner [3] \urcorner = \ulcorner 3 :: [] \urcorner = \langle\!\langle 3, 0 \rangle\!\rangle = 2^3(2 \cdot 0 + 1) = 8 = 0\mathrm{b}1000$

$\ulcorner [1,3] \urcorner = \langle\!\langle 1, \ulcorner [3] \urcorner \rangle\!\rangle = \langle\!\langle 1, 8 \rangle\!\rangle = 34 = 0\mathrm{b}100010$

$\ulcorner [2,1,3] \urcorner = \langle\!\langle 2, \ulcorner [1,3] \urcorner \rangle\!\rangle = \langle\!\langle 2, 34 \rangle\!\rangle = 276 = 0\mathrm{b}100010100$

Numerical coding of lists

$\mathit{list}\,\mathbb{N} \triangleq$ set of all finite lists of natural numbers, using ML notation for lists.

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

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

For example:

$\ulcorner [3] \urcorner = \ulcorner 3 :: [] \urcorner = \langle\!\langle 3, 0 \rangle\!\rangle = 2^3 (2 \cdot 0 + 1) = 8 = 0\mathrm{b}1000$

$\ulcorner [1,3] \urcorner = \langle\!\langle 1, \ulcorner [3] \urcorner \rangle\!\rangle = \langle\!\langle 1, 8 \rangle\!\rangle = 34 = 0\mathrm{b}100010$

$\ulcorner [2,1,3] \urcorner = \langle\!\langle 2, \ulcorner [1,3] \urcorner \rangle\!\rangle = \langle\!\langle 2, 34 \rangle\!\rangle = 276 = 0\mathrm{b}100010100$

Numerical coding of programs

If P is the RM program $\boxed{\begin{array}{c} \mathtt{L}_0 : body_0 \\ \mathtt{L}_1 : body_1 \\ \vdots \\ \mathtt{L}_n : body_n \end{array}}$

then its numerical code is

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

where the numerical code $\ulcorner body \urcorner$ of an instruction body is defined by:

$$\begin{cases} \ulcorner \mathtt{R}_i^+ \to \mathtt{L}_j \urcorner & \triangleq \langle\!\langle 2i, j \rangle\!\rangle \\ \ulcorner \mathtt{R}_i^- \to \mathtt{L}_j, \mathtt{L}_k \urcorner & \triangleq \langle\!\langle 2i+1, \langle j, k \rangle \rangle\!\rangle \\ \ulcorner \mathtt{HALT} \urcorner & \triangleq 0 \end{cases}$$

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

if $x = 0$ then $body(x)$ is $\mathtt{HALT}$,
else ($x > 0$ and) let $x = \langle\!\langle y, z \rangle\!\rangle$ in
  if $y = 2i$ is even, then
    $body(x)$ is $\mathtt{R}_i^+ \to \mathtt{L}_z$,
  else $y = 2i + 1$ is odd, let $z = \langle j, k \rangle$ in
    $body(x)$ is $\mathtt{R}_i^- \to \mathtt{L}_j, \mathtt{L}_k$

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

$$prog(e) \triangleq \boxed{\begin{array}{c} \mathtt{L}_0 : body(x_0) \\ \vdots \\ \mathtt{L}_n : body(x_n) \end{array}} \text{ where } e = \ulcorner [x_0, \ldots, x_n] \urcorner$$

Example of $prog(e)$

- $786432 = 2^{19} + 2^{18} = 0\mathtt{b}11\underbrace{0\ldots0}_{18\ ''0''s} = \ulcorner[18, 0]\urcorner$
- $18 = 0\mathtt{b}10010 = \langle\!\langle 1, 4\rangle\!\rangle = \langle\!\langle 1, \langle 0, 2\rangle\rangle\!\rangle = \ulcorner \mathtt{R}_0^- \to \mathtt{L}_0, \mathtt{L}_2 \urcorner$
- $0 = \ulcorner \mathtt{HALT} \urcorner$

So $prog(786432) = \boxed{\begin{array}{l} \mathtt{L}_0 : \mathtt{R}_0^- \to \mathtt{L}_0, \mathtt{L}_2 \\ \mathtt{L}_1 : \mathtt{HALT} \end{array}}$

Example of $prog(e)$

- $786432 = 2^{19} + 2^{18} = 0\mathtt{b}11\underbrace{0\ldots0}_{18\ \text{"0"}s} = \ulcorner[18,0]\urcorner$
- $18 = 0\mathtt{b}10010 = \langle\!\langle 1, 4\rangle\!\rangle = \langle\!\langle 1, \langle 0, 2\rangle\rangle\!\rangle = \ulcorner \mathtt{R}_0^- \to \mathtt{L}_0, \mathtt{L}_2 \urcorner$
- $0 = \ulcorner \mathtt{HALT} \urcorner$

So $prog(786432) = \boxed{\begin{array}{l} \mathtt{L}_0 : \mathtt{R}_0^- \to \mathtt{L}_0, \mathtt{L}_2 \\ \mathtt{L}_1 : \mathtt{HALT} \end{array}}$

N.B. jump to label with no body (erroneous halt)

Example of $prog(e)$

- $786432 = 2^{19} + 2^{18} = 0\mathtt{b}11\underbrace{0\ldots0}_{18\ ''0''s} = \ulcorner[18,0]\urcorner$
- $18 = 0\mathtt{b}10010 = \langle\!\langle 1,4\rangle\!\rangle = \langle\!\langle 1,\langle 0,2\rangle\rangle\!\rangle = \ulcorner \mathtt{R}_0^- \to \mathtt{L}_0,\mathtt{L}_2\urcorner$
- $0 = \ulcorner\mathtt{HALT}\urcorner$

So $prog(786432) = \boxed{\begin{array}{l}\mathtt{L}_0 : \mathtt{R}_0^- \to \mathtt{L}_0,\mathtt{L}_2\\ \mathtt{L}_1 : \mathtt{HALT}\end{array}}$

N.B. In case $e = 0$ we have $0 = \ulcorner[]\urcorner$, so $prog(0)$ is the program with an empty list of instructions, which by convention we regard as a RM that does nothing (i.e. that halts immediately).

Universal register machine, U

High-level specification

Universal RM U carries out the following computation, starting with $\mathtt{R}_0 = 0$, $\mathtt{R}_1 = e$ (code of a program), $\mathtt{R}_2 = a$ (code of a list of arguments) and all other registers zeroed:

- decode e as a RM program P
- decode a as a list of register values a_1, . . . , a_n
- carry out the computation of the RM program P starting with $\mathtt{R}_0 = 0, \mathtt{R}_1 = a_1, \ldots, \mathtt{R}_n = a_n$ (and any other registers occurring in P set to 0).

Mnemonics for the registers of $\boldsymbol{U}$ and the role they play in its program:

$\mathtt{R}_1 \equiv \mathtt{P}$ code of the RM to be simulated

$\mathtt{R}_2 \equiv \mathtt{A}$ code of current register contents of simulated RM

$\mathtt{R}_3 \equiv \mathtt{PC}$ program counter—number of the current instruction (counting from $\mathbf{0}$)

$\mathtt{R}_4 \equiv \mathtt{N}$ code of the current instruction body

$\mathtt{R}_5 \equiv \mathtt{C}$ type of the current instruction body

$\mathtt{R}_6 \equiv \mathtt{R}$ current value of the register to be incremented or decremented by current instruction (if not $\mathtt{HALT}$)

$\mathtt{R}_7 \equiv \mathtt{S}$, $\mathtt{R}_8 \equiv \mathtt{T}$ and $\mathtt{R}_9 \equiv \mathtt{Z}$ are auxiliary registers.

Overall structure of U's program

$\boxed{1}$ copy $\mathtt{PC}$th item of list in $\mathtt{P}$ to $\mathtt{N}$ (halting if $\mathtt{PC} >$ length of list); goto $\boxed{2}$

$\boxed{2}$ if $\mathtt{N} = \mathbf{0}$ then copy $\mathbf{0}$th item of list in $\mathtt{A}$ to $\mathtt{R}_0$ and halt, else (decode $\mathtt{N}$ as $\langle\!\langle y, z \rangle\!\rangle$; $\mathtt{C} ::= y$; $\mathtt{N} ::= z$; goto $\boxed{3}$)

{at this point either $\mathtt{C} = 2i$ is even and current instruction is $\mathtt{R}_i^+ \to \mathtt{L}_z$, or $\mathtt{C} = 2i + 1$ is odd and current instruction is $\mathtt{R}_i^- \to \mathtt{L}_j, \mathtt{L}_k$ where $z = \langle j, k \rangle$}

$\boxed{3}$ copy ith item of list in $\mathtt{A}$ to $\mathtt{R}$; goto $\boxed{4}$

$\boxed{4}$ execute current instruction on $\mathtt{R}$; update $\mathtt{PC}$ to next label; restore register values to $\mathtt{A}$; goto $\boxed{1}$

Overall structure of U's program

$\boxed{1}$ copy $\mathtt{PC}$th item of list in $\mathtt{P}$ to $\mathtt{N}$ (halting if $\mathtt{PC} >$ length of list); goto $\boxed{2}$

$\boxed{2}$ if $\mathtt{N} = 0$ then copy 0th item of list in $\mathtt{A}$ to $\mathtt{R}_0$ and halt, else (decode $\mathtt{N}$ as $\langle\!\langle y, z\rangle\!\rangle$; $\mathtt{C} ::= y$; $\mathtt{N} ::= z$; goto $\boxed{3}$)

{at this point either $\mathtt{C} = 2i$ is even and current instruction is $\mathtt{R}_i^+ \to \mathtt{L}_z$,
or $\mathtt{C} = 2i + 1$ is odd and current instruction is $\mathtt{R}_i^- \to \mathtt{L}_j, \mathtt{L}_k$ where $z = \langle j, k\rangle$}

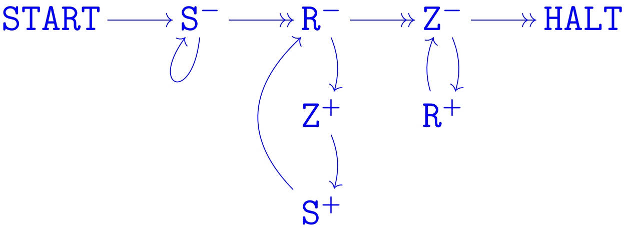
$\boxed{3}$ copy ith item of list in $\mathtt{A}$ to $\mathtt{R}$; goto $\boxed{4}$

$\boxed{4}$ execute current instruction on $\mathtt{R}$; update $\mathtt{PC}$ to next label; restore register values to $\mathtt{A}$; goto $\boxed{1}$

To implement this, we need RMs for manipulating (codes of) lists of numbers. . .

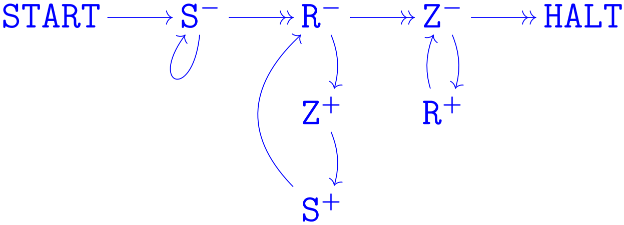
The program $\texttt{START}\longrightarrow\boxed{\texttt{S} ::= \texttt{R}}\longrightarrow\texttt{HALT}$

to copy the contents of $\texttt{R}$ to $\texttt{S}$ can be implemented by



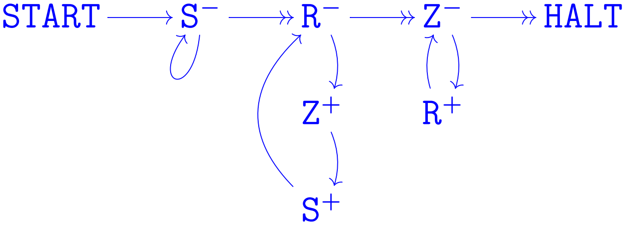
The program $\texttt{START}\longrightarrow\boxed{\texttt{S} ::= \texttt{R}}\longrightarrow\texttt{HALT}$

to copy the contents of $\texttt{R}$ to $\texttt{S}$ can be implemented by



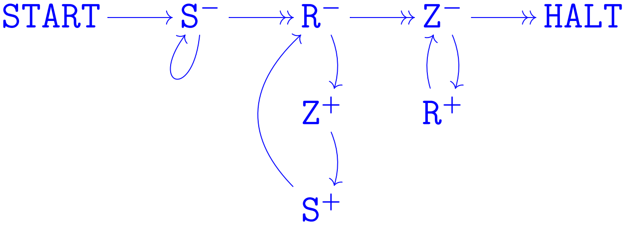
The program $\mathtt{START}\longrightarrow\boxed{\mathtt{S} ::= \mathtt{R}}\longrightarrow\mathtt{HALT}$

to copy the contents of $\mathtt{R}$ to $\mathtt{S}$ can be implemented by



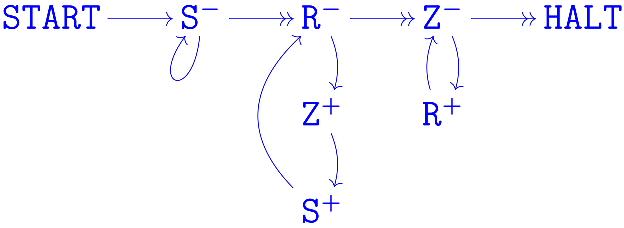
The program $\mathtt{START}\longrightarrow\boxed{\mathtt{S} ::= \mathtt{R}}\longrightarrow\mathtt{HALT}$

to copy the contents of $\mathtt{R}$ to $\mathtt{S}$ can be implemented by



The program $\texttt{START}\longrightarrow\boxed{\texttt{S} ::= \texttt{R}}\longrightarrow\texttt{HALT}$

to copy the contents of $\texttt{R}$ to $\texttt{S}$ can be implemented by



precondition:

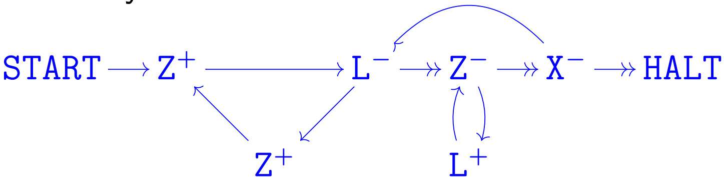
$\texttt{R} = \boldsymbol{x}$
$\texttt{S} = \boldsymbol{y}$
$\texttt{Z} = \mathbf{0}$

postcondition:

$\texttt{R} = \boldsymbol{x}$
$\texttt{S} = \boldsymbol{x}$
$\texttt{Z} = \mathbf{0}$

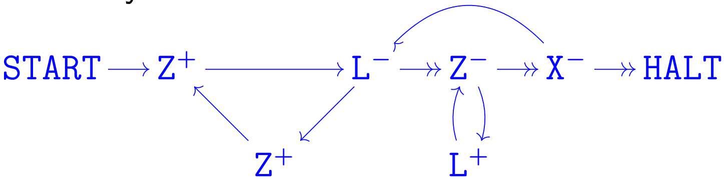
The program $\mathtt{START} \longrightarrow \boxed{\begin{array}{c}\textit{push}\ \mathtt{X}\\ \textit{to}\ \mathtt{L}\end{array}} \longrightarrow \mathtt{HALT}$ $\boxed{2^{\mathtt{X}}(2\mathtt{L}+1)}$

to carry out the assignment $(\mathtt{X}, \mathtt{L}) ::= (0, \mathtt{X} :: \mathtt{L})$ can be implemented by



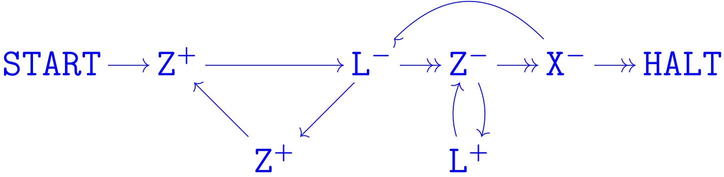
The program $\mathtt{START}\longrightarrow\boxed{\begin{array}{c}\textit{push}\ \mathtt{X}\\ \textit{to}\ \mathtt{L}\end{array}}\longrightarrow\mathtt{HALT}$

to carry out the assignment $(\mathtt{X},\mathtt{L}) ::= (0, \mathtt{X} :: \mathtt{L})$ can be implemented by



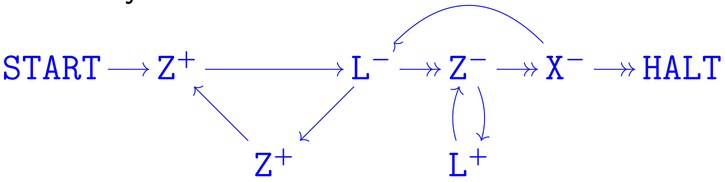
The program $\texttt{START}\longrightarrow\boxed{\begin{matrix}\textit{push}\ \texttt{X}\\ \textit{to}\ \texttt{L}\end{matrix}}\longrightarrow\texttt{HALT}$

to carry out the assignment $(\texttt{X},\texttt{L}) ::= (0, \texttt{X} :: \texttt{L})$ can be implemented by



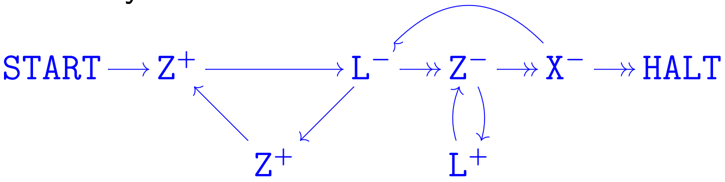
The program $\mathtt{START} \longrightarrow \boxed{\begin{matrix} \textbf{\textit{push}}\ \mathtt{X} \\ \textbf{\textit{to}}\ \mathtt{L} \end{matrix}} \longrightarrow \mathtt{HALT}$

to carry out the assignment $(\mathtt{X}, \mathtt{L}) ::= (0, \mathtt{X} :: \mathtt{L})$ can be implemented by



The program $\mathtt{START}\longrightarrow\boxed{\begin{array}{c}\textit{push } \mathtt{X}\\ \textit{to } \mathtt{L}\end{array}}\longrightarrow\mathtt{HALT}$

to carry out the assignment $(\mathtt{X},\mathtt{L}) ::= (0, \mathtt{X} :: \mathtt{L})$ can be implemented by



precondition:

$\mathtt{X} = x$
$\mathtt{L} = \ell$
$\mathtt{Z} = 0$

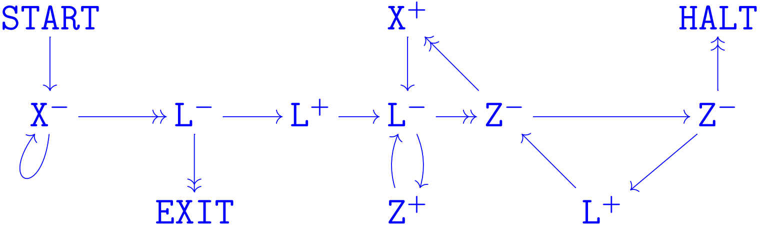
postcondition:

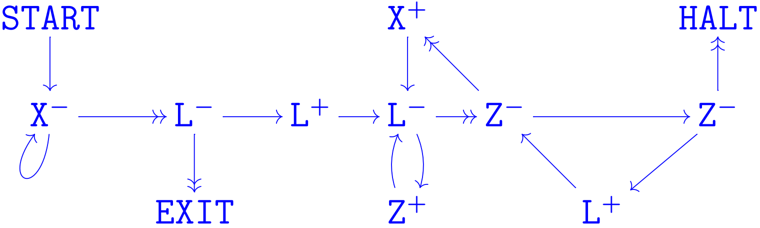
$\mathtt{X} = 0$
$\mathtt{L} = \langle\!\langle x, \ell \rangle\!\rangle = 2^x(2\ell + 1)$
$\mathtt{Z} = 0$

The program $\texttt{START}\longrightarrow\boxed{\begin{matrix}\textit{pop}\ \texttt{L}\\ \textit{to}\ \texttt{X}\end{matrix}}\begin{matrix}\longrightarrow\texttt{HALT}\\ \twoheadrightarrow\texttt{EXIT}\end{matrix}$ specified by

"$\textit{\textbf{if}}\ \texttt{L} = \mathbf{0}\ \textit{\textbf{then}}\ (\texttt{X} ::= \mathbf{0};\ \textit{\textbf{goto}}\ \texttt{EXIT})\ \textit{\textbf{else}}$
$\textit{\textbf{let}}\ \texttt{L} = \langle\!\langle x, \ell\rangle\!\rangle\ \textit{\textbf{in}}\ (\texttt{X} ::= x;\ \texttt{L} ::= \ell;\ \textit{\textbf{goto}}\ \texttt{HALT})$"

can be implemented by

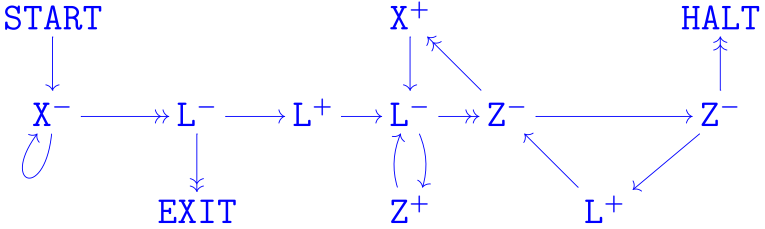




START X^+ HALT

X^- L^- L^+ L^- Z^- Z^-

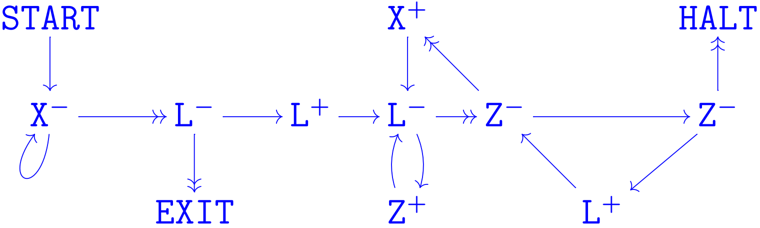
EXIT Z^+ L^+


START
X⁺
HALT
X⁻
L⁻
L⁺
L⁻
Z⁻
Z⁻
EXIT
Z⁺
L⁺

The program $\mathtt{START}\longrightarrow\boxed{\begin{matrix}\textit{pop}\ \mathtt{L}\\ \textit{to}\ \mathtt{X}\end{matrix}}\begin{matrix}\longrightarrow\mathtt{HALT}\\ \twoheadrightarrow\mathtt{EXIT}\end{matrix}$ specified by

"$\boldsymbol{\mathit{if}}\ \mathtt{L} = \mathbf{0}\ \boldsymbol{\mathit{then}}\ (\mathtt{X} ::= \mathbf{0};\ \boldsymbol{\mathit{goto}}\ \mathtt{EXIT})\ \boldsymbol{\mathit{else}}$
$\boldsymbol{\mathit{let}}\ \mathtt{L} = \langle\!\langle x, \ell\rangle\!\rangle\ \boldsymbol{\mathit{in}}\ (\mathtt{X} ::= x;\ \mathtt{L} ::= \ell;\ \boldsymbol{\mathit{goto}}\ \mathtt{HALT})$"

can be implemented by



Overall structure of U's program

$\boxed{1}$ copy $\mathtt{PC}$th item of list in $\mathtt{P}$ to $\mathtt{N}$ (halting if $\mathtt{PC} >$ length of list); goto $\boxed{2}$

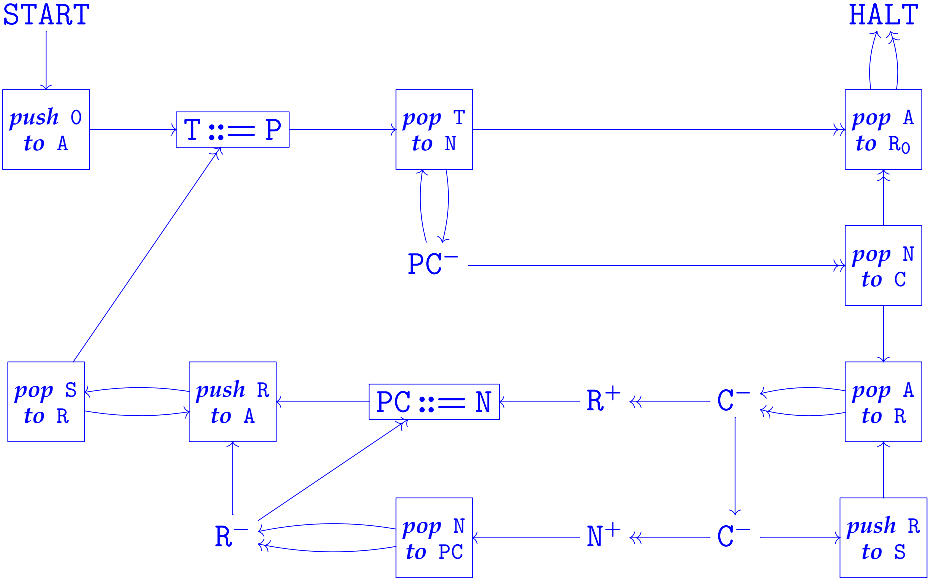
$\boxed{2}$ if $\mathtt{N} = 0$ then copy 0th item of list in $\mathtt{A}$ to $\mathtt{R}_0$ and halt, else (decode $\mathtt{N}$ as $\langle\!\langle y, z \rangle\!\rangle$; $\mathtt{C} ::= y$; $\mathtt{N} ::= z$; goto $\boxed{3}$)

{at this point either $\mathtt{C} = 2i$ is even and current instruction is $\mathtt{R}_i^+ \to \mathtt{L}_z$,
or $\mathtt{C} = 2i + 1$ is odd and current instruction is $\mathtt{R}_i^- \to \mathtt{L}_j, \mathtt{L}_k$ where $z = \langle j, k \rangle$}

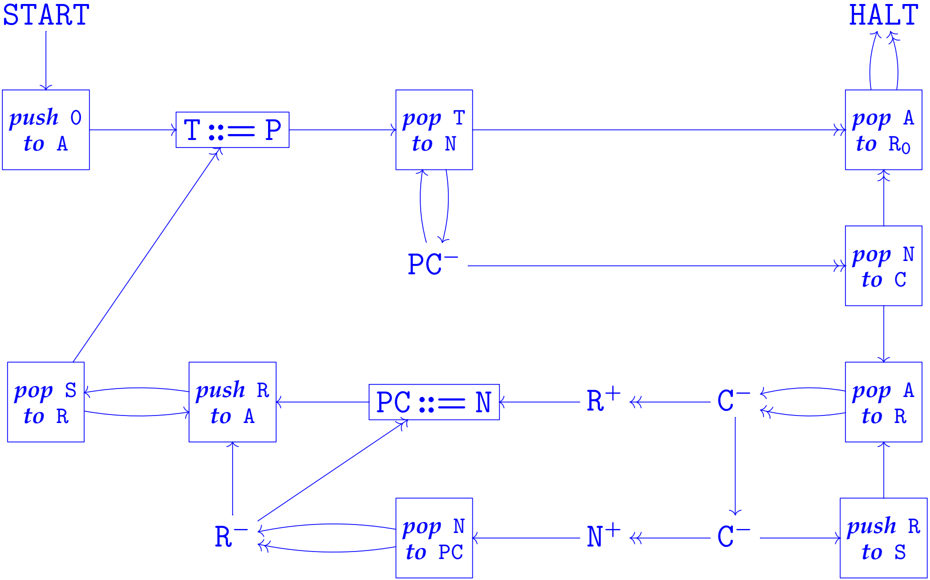
$\boxed{3}$ copy ith item of list in $\mathtt{A}$ to $\mathtt{R}$; goto $\boxed{4}$

$\boxed{4}$ execute current instruction on $\mathtt{R}$; update $\mathtt{PC}$ to next label; restore register values to $\mathtt{A}$; goto $\boxed{1}$

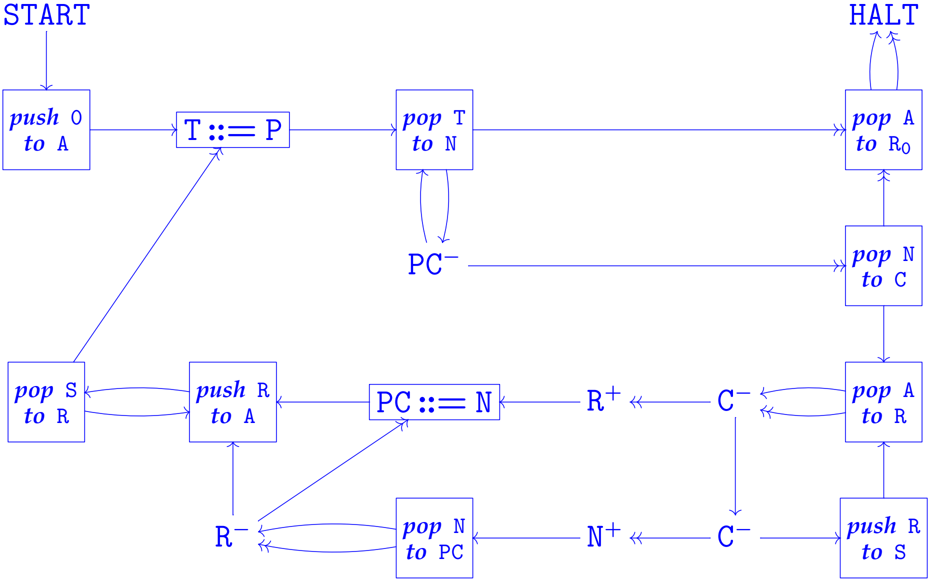
The program for U



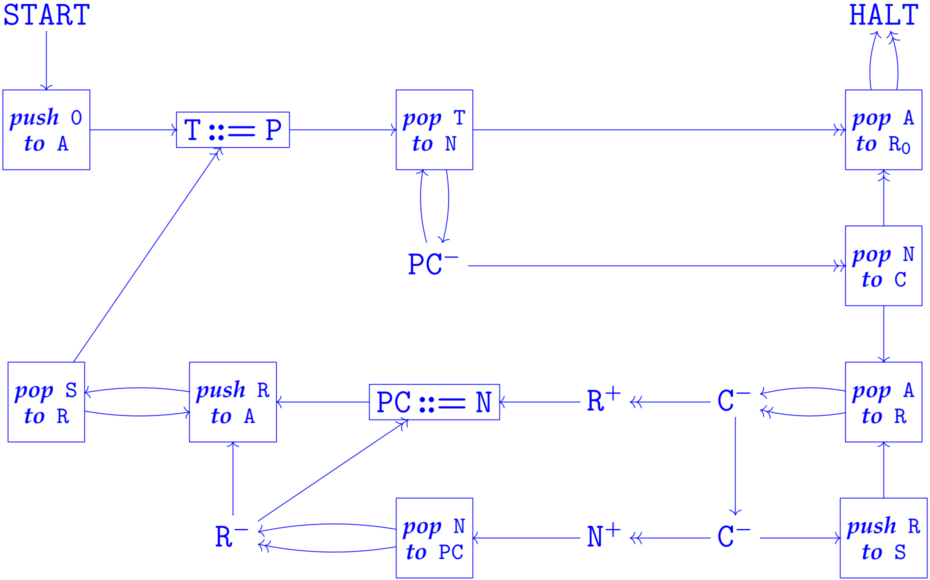
The program for *U*



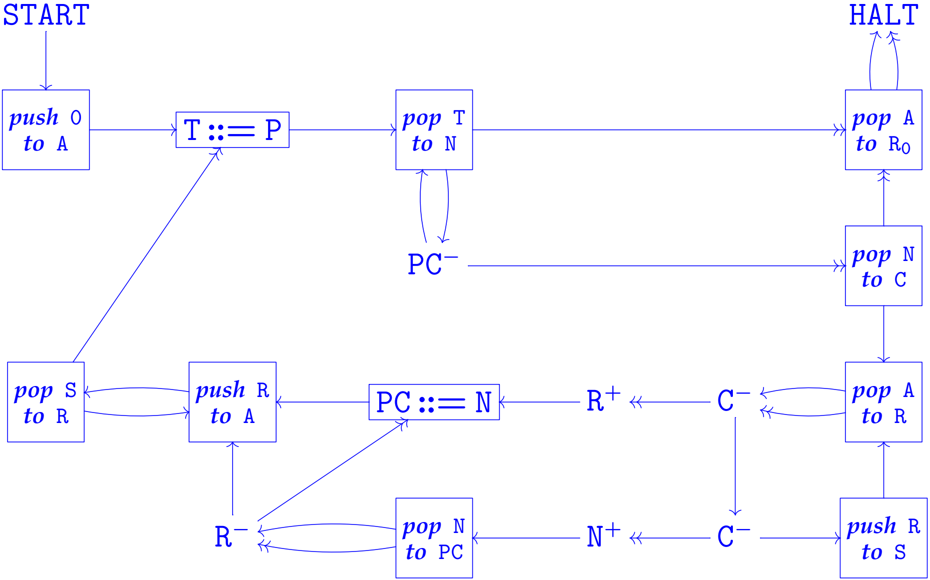
The program for ***U***



The program for $\boldsymbol{U}$



The program for U



The halting problem

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

$$\mathtt{R}_0 = 0 \qquad \mathtt{R}_1 = e \qquad \mathtt{R}_2 = \ulcorner [a_1, \dots, a_n] \urcorner$$

and all other registers zeroed, the computation of H always halts with $\mathtt{R}_0$ containing 0 or 1; moreover when the computation halts, $\mathtt{R}_0 = 1$ if and only if

> the register machine program with index e eventually halts when started with $\mathtt{R}_0 = 0, \mathtt{R}_1 = a_1, \dots, \mathtt{R}_n = a_n$ and all other registers zeroed.

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

$$\mathrm{R}_0 = 0 \qquad \mathrm{R}_1 = e \qquad \mathrm{R}_2 = \ulcorner [a_1, \ldots, a_n] \urcorner$$

and all other registers zeroed, the computation of H always halts with R_0 containing 0 or 1; moreover when the computation halts, $\mathrm{R}_0 = 1$ if and only if

> the register machine program with index e eventually halts when started with $\mathrm{R}_0 = 0, \mathrm{R}_1 = a_1, \ldots, \mathrm{R}_n = a_n$ and all other registers zeroed.

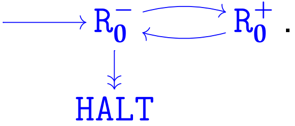
Theorem. No such register machine H can exist.

Proof of the theorem

Assume we have a RM H that decides the Halting Problem and derive a contradiction, as follows:

- Let H' be obtained from H by replacing $\texttt{START}\longrightarrow$ by

  $\texttt{START}\longrightarrow\boxed{\texttt{Z} ::= \texttt{R}_1}\longrightarrow\boxed{\begin{array}{c}\textit{push } \texttt{Z} \\ \textit{to } \texttt{R}_2\end{array}}\longrightarrow$

  (where $\texttt{Z}$ is a register not mentioned in H's program).
- Let C be obtained from H' by replacing each $\texttt{HALT}$ (& each erroneous halt) by  .

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

Proof of the theorem

Assume we have a RM H that decides the Halting Problem and derive a contradiction, as follows:

C started with $\mathtt{R}_1 = c$ eventually halts

if & only if

H' started with $\mathtt{R}_1 = c$ halts with $\mathtt{R}_0 = 0$

Proof of the theorem

Assume we have a RM H that decides the Halting Problem and derive a contradiction, as follows:

C started with $\mathtt{R}_1 = c$ eventually halts

if & only if

H' started with $\mathtt{R}_1 = c$ halts with $\mathtt{R}_0 = 0$

if & only if

H started with $\mathtt{R}_1 = c, \mathtt{R}_2 = \ulcorner [c] \urcorner$ halts with $\mathtt{R}_0 = 0$

Proof of the theorem

Assume we have a RM H that decides the Halting Problem and derive a contradiction, as follows:

C started with $\mathtt{R}_1 = c$ eventually halts

if & only if

H' started with $\mathtt{R}_1 = c$ halts with $\mathtt{R}_0 = 0$

if & only if

H started with $\mathtt{R}_1 = c, \mathtt{R}_2 = \ulcorner [c] \urcorner$ halts with $\mathtt{R}_0 = 0$

if & only if

$prog(c)$ started with $\mathtt{R}_1 = c$ does not halt

Proof of the theorem

Assume we have a RM H that decides the Halting Problem and derive a contradiction, as follows:

C started with $\mathtt{R}_1 = c$ eventually halts

if & only if

H' started with $\mathtt{R}_1 = c$ halts with $\mathtt{R}_0 = 0$

if & only if

H started with $\mathtt{R}_1 = c, \mathtt{R}_2 = \ulcorner[c]\urcorner$ halts with $\mathtt{R}_0 = 0$

if & only if

$prog(c)$ started with $\mathtt{R}_1 = c$ does not halt

if & only if

C started with $\mathtt{R}_1 = c$ does not halt

Proof of the theorem

Assume we have a RM H that decides the Halting Problem and derive a contradiction, as follows:

C started with $\mathtt{R}_1 = c$ eventually halts

if & only if

H' started with $\mathtt{R}_1 = c$ halts with $\mathtt{R}_0 = 0$

if & only if

H started with $\mathtt{R}_1 = c, \mathtt{R}_2 = \ulcorner [c] \urcorner$ halts with $\mathtt{R}_0 = 0$

if & only if

$prog(c)$ started with $\mathtt{R}_1 = c$ does not halt

if & only if

C started with $\mathtt{R}_1 = c$ does not halt

—contradiction!

Computable functions

Recall:

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is (register machine) computable if there is a register machine M with at least $n+1$ registers R_0, R_1, . . . , R_n (and maybe more) such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$,

> the computation of M starting with $\mathrm{R}_0 = 0$, $\mathrm{R}_1 = x_1$, . . . , $\mathrm{R}_n = x_n$ and all other registers set to 0, halts with $\mathrm{R}_0 = y$

if and only if $f(x_1, \ldots, x_n) = y$.

Note that the same RM M could 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. . .

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 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 $\mathtt{R}_0 = 0, \mathtt{R}_1 = x$ and all other registers zeroed eventually halts with $\mathtt{R}_0 = y$.

Thus

$$e \mapsto \varphi_e$$

defines an onto function from $\mathbb{N}$ to the collection of all computable partial functions from $\mathbb{N}$ to $\mathbb{N}$.

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 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 $\mathtt{R}_0 = 0, \mathtt{R}_1 = x$ and all other registers zeroed eventually halts with $\mathtt{R}_0 = y$.

Thus

$$e \mapsto \varphi_e$$

defines an onto function from $\mathbb{N}$ to the collection of all computable partial functions from $\mathbb{N}$ to $\mathbb{N}$.

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

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

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 def. of f); so $\varphi_e(e) = 0$ (since $f = \varphi_e$), hence $\varphi_e(e)\downarrow$
- if $\varphi_e(e)\downarrow$, then $f(e)\downarrow$ (since $f = \varphi_e$); so $\varphi_e(e)\uparrow$ (by def. of f)

—contradiction! So f cannot be computable.

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

(Un)decidable sets of numbers

Definition. $S \subseteq \mathbb{N}$ 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.

So S is decidable iff there is a RM M with the property: for all $x \in \mathbb{N}$, M started with $\mathtt{R}_0 = 0, \mathtt{R}_1 = x$ and all other registers zeroed eventually halts with $\mathtt{R}_0$ containing 1 or 0; and $\mathtt{R}_0 = 1$ on halting iff $x \in S$.

(Un)decidable sets of numbers

Definition. $S \subseteq \mathbb{N}$ 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.

So S is decidable iff there is a RM M with the property: for all $x \in \mathbb{N}$, M started with $\mathtt{R}_0 = 0, \mathtt{R}_1 = x$ and all other registers zeroed eventually halts with $\mathtt{R}_0$ containing 1 or 0; and $\mathtt{R}_0 = 1$ on halting iff $x \in S$.

Basic strategy: to prove $S \subseteq \mathbb{N}$ undecidable, try to show that decidability of S would imply decidability of the Halting Problem.

For example...

Claim: $S_0 \triangleq \{e \mid \varphi_e(0)\downarrow\}$ is undecidable.

Claim: $S_0 \triangleq \{e \mid \varphi_e(0){\downarrow}\}$ is undecidable.

Proof (sketch): Suppose M_0 is a RM computing χ_{S_0}. From M_0's program (using the same techniques as for constructing a universal RM) we can construct a RM H to carry out:

$$
\begin{array}{l}
\textit{let } e = \mathtt{R}_1 \textit{ and } \ulcorner [a_1, \dots, a_n] \urcorner = \mathtt{R}_2 \textit{ in} \\
\qquad \mathtt{R}_1 ::= \ulcorner (\mathtt{R}_1 ::= a_1) \,; \cdots ; (\mathtt{R}_n ::= a_n) \,; \textit{prog}(e) \urcorner \,; \\
\mathtt{R}_2 ::= 0 \,; \\
\textit{run } M_0
\end{array}
$$

Then by assumption on M_0, H decides the Halting Problem—contradiction. So no such M_0 exists, i.e. χ_{S_0} is uncomputable, i.e. S_0 is undecidable.

Claim: $S_1 \triangleq \{e \mid \varphi_e \textit{ a total function}\}$ is undecidable.

Claim: $S_1 \triangleq \{e \mid \varphi_e \textit{ a total function}\}$ is undecidable.

Proof (sketch): Suppose M_1 is a RM computing χ_{S_1}. From M_1's program we can construct a RM M_0 to carry out:

$$\begin{array}{l} \textit{let } e = \mathtt{R}_1 \textit{ in } \mathtt{R}_1 ::= \ulcorner \mathtt{R}_1 ::= 0\,; \textit{prog}(e) \urcorner\,; \\ \textit{run } M_1 \end{array}$$

Then by assumption on M_1, M_0 decides membership of S_0 from previous example (i.e. computes χ_{S_0})—contradiction. So no such M_1 exists, i.e. χ_{S_1} is uncomputable, i.e. S_1 is undecidable.

Turing machines

Algorithms, informally

No precise definition of "algorithm" at the time Hilbert posed the *Entscheidungsproblem*, just examples.

Common features of the examples:

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

e.g. multiply two decimal digits by looking up their product in a table

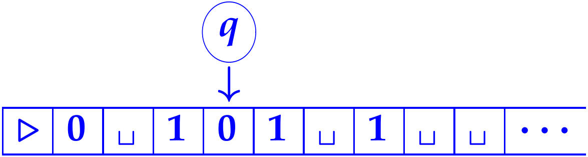
Register Machine computation abstracts away from any particular, concrete representation of numbers (e.g. as bit strings) and the associated elementary operations of increment/decrement/zero-test.

Turing's original model of computation (now called a Turing machine) is 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.

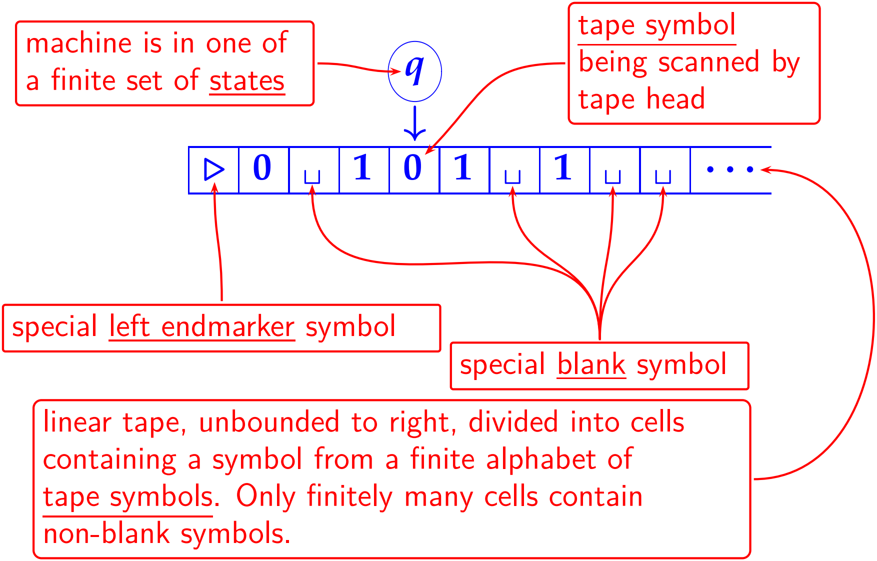
Register Machine computation abstracts away from any particular, concrete representation of numbers (e.g. as bit strings) and the associated elementary operations of increment/decrement/zero-test.

Turing's original model of computation (now called a Turing machine) is 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.

Turing machines, informally



Turing machines, informally



machine is in one of a finite set of <u>states</u>

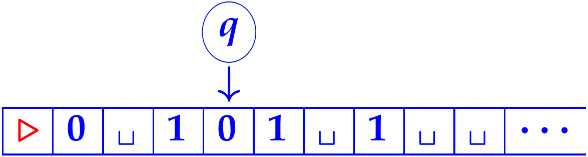
<u>tape symbol</u> being scanned by tape head

special <u>left endmarker</u> symbol

special <u>blank</u> symbol

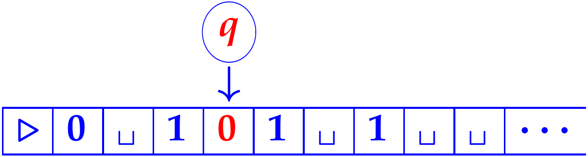
linear tape, unbounded to right, divided into cells containing a symbol from a finite alphabet of <u>tape symbols</u>. Only finitely many cells contain non-blank symbols.

Turing machines, informally



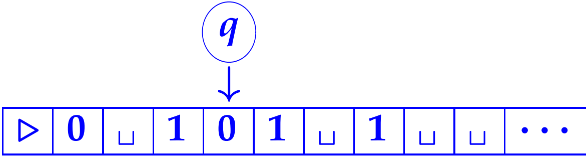
- Machine starts with tape head pointing to the special left endmarker $\triangleright$.

Turing machines, informally



- Machine starts with tape head pointing to the special left endmarker ▷.
- Machine computes in discrete steps, each of which depends only on current state (q) and symbol being scanned by tape head (**0**).

Turing machines, informally



- Machine starts with tape head pointing to the special left endmarker ▷.
- Machine computes in discrete steps, each of which depends only on current state (q) and symbol being scanned by tape head ($\mathbf{0}$).
- Action at each step is to overwrite the current tape cell with a symbol, move left or right one cell, or stay stationary, and change state.

Turing Machines

are specified by:

- Q, finite set of machine states
- Σ, finite set of tape symbols (disjoint from Q) containing distinguished symbols $\rhd$ (left endmarker) and $\sqcup$ (blank)
- $s \in Q$, an initial state
- $\delta \in (Q \times \Sigma) \to (Q \cup \{\mathtt{acc}, \mathtt{rej}\}) \times \Sigma \times \{L, R, S\}$, a transition function—specifies for each state and symbol a next state (or accept `acc` or reject `rej`), a symbol to overwrite the current symbol, and a direction for the tape head to move (L=left, R=right, S=stationary).

Turing Machines

are specified by:

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

  for all $q \in Q$, there exists $q' \in Q \cup \{\mathtt{acc}, \mathtt{rej}\}$
  with $\delta(q, \triangleright) = (q', \triangleright, R)$
  (i.e. left endmarker is never overwritten and machine always moves to the right when scanning it)

Example Turing Machine

$M = (Q, \Sigma, s, \delta)$ where

states $Q = \{s, q, q'\}$ (s initial)

symbols $\Sigma = \{\triangleright, \sqcup, 0, 1\}$

transition function

$\delta \in (Q \times \Sigma) \to (Q \cup \{\texttt{acc}, \texttt{rej}\}) \times \Sigma \times \{L, R, S\}$:

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

Turing machine computation

Turing machine configuration: (q, w, u)

where

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

(So $wu \in \Sigma^*$ represents the current tape contents.)

Turing machine computation

Turing machine configuration: (q, w, u)

where

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

Initial configurations: $(s, \triangleright, u)$

Turing machine computation

Given a TM $M = (Q, \Sigma, s, \delta)$, we write

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

to mean $q \neq \mathtt{acc}, \mathtt{rej}$, $w = va$ (for some v, a) and

either $\delta(q, a) = (q', a', L)$, $w' = v$, and $u' = a'u$

or $\delta(q, a) = (q', a', S)$, $w' = va'$ and $u' = u$

or $\delta(q, a) = (q', a', R)$, $u = a''u''$ is non-empty, $w' = va'a''$ and $u' = u''$

or $\delta(q, a) = (q', a', R)$, $u = \varepsilon$ is empty, $w' = va'\sqcup$ and $u' = \varepsilon$.

Turing machine computation

A computation of a TM M is a (finite or infinite) sequence of configurations $c_0, c_1, c_2, \ldots$

where

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

The computation

- does not halt if the sequence is infinite
- halts if the sequence is finite and its last element is of the form $(\mathtt{acc}, w, u)$ or $(\mathtt{rej}, w, u)$.

Example Turing Machine

$M = (Q, \Sigma, s, \delta)$ where

states $Q = \{s, q, q'\}$ (s initial)

symbols $\Sigma = \{\triangleright, \sqcup, 0, 1\}$

transition function

$\delta \in (Q \times \Sigma) \to (Q \cup \{\texttt{acc}, \texttt{rej}\}) \times \Sigma \times \{L, R, S\}$:

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

Claim: the computation of M starting from configuration $(s, \triangleright, \sqcup 1^n 0)$ halts in configuration $(\texttt{acc}, \triangleright \sqcup, 1^{n+1} 0)$.

Example Turing Machine

$M = (Q, \Sigma, s, \delta)$ where

states $Q = \{s, q, q'\}$ (s initial)

symbols $\Sigma = \{\triangleright, \sqcup, 0, 1\}$

transition function

$\delta \in (Q \times \Sigma) \to (Q \cup \{\mathtt{acc}, \mathtt{rej}\}) \times \Sigma \times \{L, R, S\}$:

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

a string of n 1s

Claim: the computation of M starting from configuration $(s, \triangleright, \sqcup 1^n 0)$ halts in configuration $(\mathtt{acc}, \triangleright\sqcup, 1^{n+1}0)$.

The computation of M starting from configuration $(s\,,\,\triangleright\,,\,\sqcup 1^n 0)$:

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

Theorem. 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 states, tape symbols, tape contents and configurations.

Step 2: implement M's transition function (finite table) using RM instructions on codes.

Step 3: implement a RM program to repeatedly carry out $\to_M$.

Step 1

- Identify states and tape symbols with particular numbers:

$$\begin{array}{rcl|rcl} \texttt{acc} & = & 0 & \sqcup & = & 0 \\ \texttt{rej} & = & 1 & \rhd & = & 1 \\ Q & = & \{2, 3, \ldots, n\} & \Sigma & = & \{0, 1, \ldots, m\} \end{array}$$

- Code configurations $c = (q, w, u)$ by:

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

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

Step 1

reversal of w makes it easier to use our RM programs for list manipulation

- Code configurations $c = (q, w, u)$ by:

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

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

Step 2

Using registers

$$\begin{aligned} \mathtt{Q} &= \text{current state} \\ \mathtt{A} &= \text{current tape symbol} \\ \mathtt{D} &= \text{current direction of tape head} \\ &\ (\text{with } \boldsymbol{L = 0},\ \boldsymbol{R = 1} \text{ and } \boldsymbol{S = 2}, \text{ say}) \end{aligned}$$

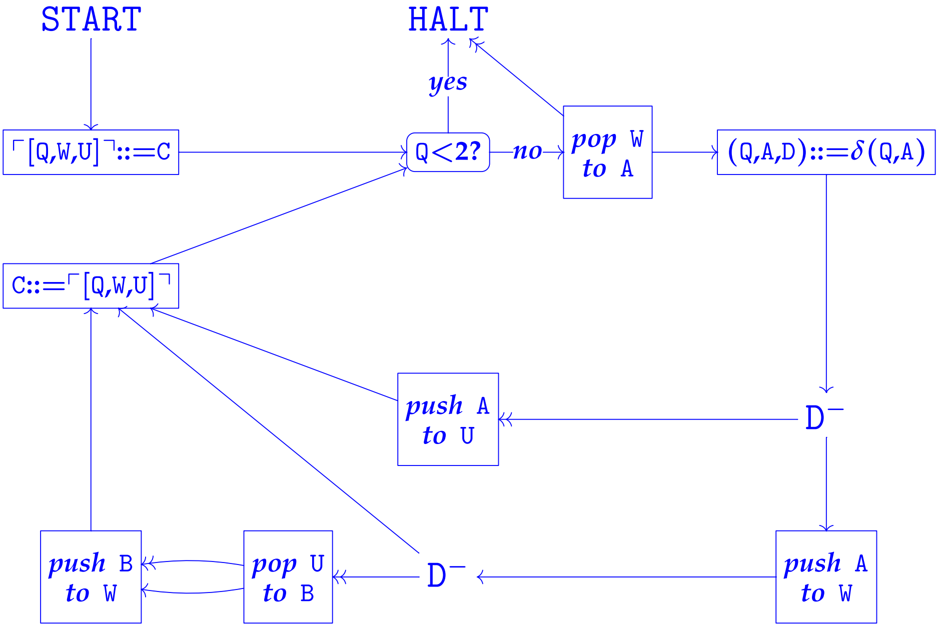
one can turn the finite table of (argument,result)-pairs specifying δ into a RM program $\rightarrow \boxed{(\mathtt{Q},\mathtt{A},\mathtt{D}) ::= \delta(\mathtt{Q},\mathtt{A})} \rightarrow$ so that starting the program with $\mathtt{Q} = q$, $\mathtt{A} = a$, $\mathtt{D} = d$ (and all other registers zeroed), it halts with $\mathtt{Q} = q'$, $\mathtt{A} = a'$, $\mathtt{D} = d'$, where $(q', a', d') = \delta(q, a)$.

Step 3

The next slide specifies a RM to carry out $\boldsymbol{M}$'s computation. It uses registers

- `C` = code of current configuration
- `W` = code of tape symbols at and left of tape head (reading right-to-left)
- `U` = code of tape symbols right of tape head (reading left-to-right)

Starting with `C` containing the code of an initial configuration (and all other registers zeroed), the RM program halts if and only if $\boldsymbol{M}$ halts; and in that case `C` holds the code of the final configuration.



START

HALT

yes

$\ulcorner$[Q,W,U]$\urcorner$::=C

Q<2?

no

pop W *to* A

(Q,A,D)::=δ(Q,A)

C::=$\ulcorner$[Q,W,U]$\urcorner$

push A *to* U

D^-

push B *to* W

pop U *to* B

D^-

push A *to* W

Computable functions

Recall:

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is (register machine) computable if there is a register machine M with at least $n + 1$ registers R_0, R_1, . . . , R_n (and maybe more) such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$,

> the computation of M starting with $\mathrm{R}_0 = 0$, $\mathrm{R}_1 = x_1$, . . . , $\mathrm{R}_n = x_n$ and all other registers set to 0, halts with $\mathrm{R}_0 = y$

if and only if $f(x_1, \ldots, x_n) = y$.

We've seen that a Turing machine's computation can be implemented by a register machine.

The converse holds: the computation of a register machine can be implemented by a Turing machine.

To make sense of this, we first have to fix a tape representation of RM configurations and hence of numbers and lists of numbers. . .

Tape encoding of lists of numbers

Definition. 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.

Such tapes look like:

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

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

Tape encoding of lists of numbers

Definition. 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.

Such tapes look like:

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

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

Tape encoding of lists of numbers

Definition. 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.

Such tapes look like:

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

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

Turing computable function

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is Turing computable if and only if there is a Turing machine M with the following property:

> Starting M from its initial state with 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 $f(x_1, \ldots, x_n) = y$.

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

Proof (sketch). We've seen how to implement any TM by a RM. Hence

$$f \text{ TM computable implies } f \text{ RM computable.}$$

For the converse, one has to implement the computation of a RM in terms of a TM operating on a tape coding RM configurations. To do this, one has to show how to carry out the action of each type of RM instruction on the tape. It should be reasonably clear that this is possible in principle, even if the details (omitted) are tedious.

Notions of computability

- Church (1936): λ-calculus [see later]
- Turing (1936): Turing machines.

Turing showed that the two very different approaches determine the same class of computable functions. Hence:

Church-Turing Thesis. Every algorithm [in intuitive sense of Lect. 1] can be realized as a Turing machine.

Notions of computability

Church-Turing Thesis. Every algorithm [in intuitive sense of Lect. 1] can be realized as a Turing machine.

Further evidence for the thesis:

- Gödel and Kleene (1936): partial recursive functions
- Post (1943) and Markov (1951): canonical systems for generating the theorems of a formal system
- Lambek (1961) and Minsky (1961): register machines
- Variations on all of the above (e.g. multiple tapes, non-determinism, parallel execution. . .)

All have turned out to determine the same collection of computable functions.

Notions of computability

Church-Turing Thesis. Every algorithm [in intuitive sense of Lect. 1] can be realized as a Turing machine.

In rest of the course we'll look at

- Gödel and Kleene (1936): partial recursive functions
  ($\leadsto$ branch of mathematics called recursion theory)
- Church (1936): λ-calculus
  ($\leadsto$ branch of CS called functional programming)

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 minimization.

The characterization is due to Kleene (1936), building on work of Gödel and Herbrand.

Basic functions

- Projection functions, $\mathtt{proj}_i^n \in \mathbb{N}^n \to \mathbb{N}$:

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

- Constant functions with value $\mathbf{0}$, $\mathtt{zero}^n \in \mathbb{N}^n \to \mathbb{N}$:

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

- Successor function, $\mathtt{succ} \in \mathbb{N} \to \mathbb{N}$:

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

Basic functions

are all RM computable:

- Projection $\mathtt{proj}_i^n$ is computed by

$$\mathtt{START} \longrightarrow \boxed{R_0 ::= R_i} \longrightarrow \mathtt{HALT}$$

- Constant $\mathtt{zero}^n$ is computed by

$$\mathtt{START} \longrightarrow \mathtt{HALT}$$

- Successor $\mathtt{succ}$ is computed by

$$\mathtt{START} \longrightarrow R_1^+ \longrightarrow \boxed{R_0 ::= R_1} \longrightarrow \mathtt{HALT}$$

Composition

Composition of $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ with $g_1, \ldots, g_n \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ is the partial function $f \circ [g_1, \ldots, g_n] \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ satisfying for all $x_1, \ldots, x_m \in \mathbb{N}$

$$
\begin{aligned}
f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m) \equiv& \\
& f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))
\end{aligned}
$$

where $\equiv$ is "Kleene equivalence" of possibly-undefined expressions: $\mathbf{LHS} \equiv \mathbf{RHS}$ means "either both $\mathbf{LHS}$ and $\mathbf{RHS}$ are undefined, or they are both defined and are equal."

Composition

Composition of $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ with $g_1, \ldots, g_n \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ is the partial function $f \circ [g_1, \ldots, g_n] \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ satisfying for all $x_1, \ldots, x_m \in \mathbb{N}$

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

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

Composition

Composition of $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ with $g_1, \ldots, g_n \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ is the partial function $f \circ [g_1, \ldots, g_n] \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ satisfying for all $x_1, \ldots, x_m \in \mathbb{N}$

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

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

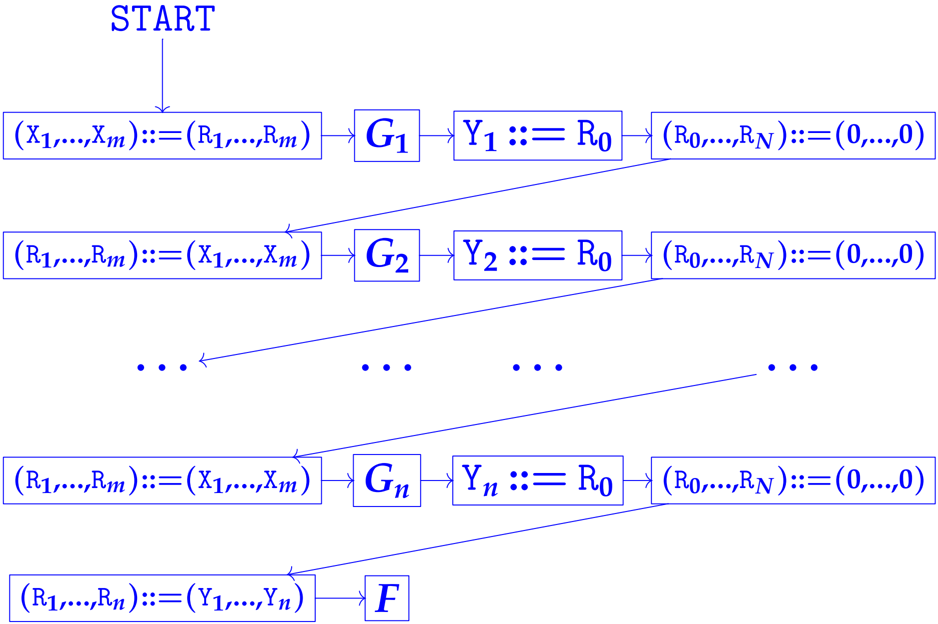
N.B. in case $n = 1$, we write $f \circ g_1$ for $f \circ [g_1]$.

Composition

$f \circ [g_1, \ldots, g_n]$ is computable if f and $g_1, \ldots, g_n$ 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 $\mathtt{R}_0$ starting with $\begin{cases} \mathtt{R}_1, \ldots, \mathtt{R}_n \\ \mathtt{R}_1, \ldots, \mathtt{R}_m \end{cases}$ set to $\begin{cases} y_1, \ldots, y_n \\ x_1, \ldots, x_m \end{cases}$, then the next slide specifies a RM program computing $f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m)$ in $\mathtt{R}_0$ starting with $\mathtt{R}_1, \ldots, \mathtt{R}_m$ set to $x_1, \ldots, x_m$.

(**Hygiene** [caused by the lack of *local names* for registers in the RM model of computation]: we assume the programs $F, G_1, \ldots, G_n$ only mention registers up to $\mathtt{R}_N$ (where $N \geq \mathbf{max}\{n, m\}$) and that $\mathtt{X}_1, \ldots, \mathtt{X}_m, \mathtt{Y}_1, \ldots, \mathtt{Y}_n$ are some registers $\mathtt{R}_i$ with $i > N$.)



START

$(X_1,\dots,X_m)::=(R_1,\dots,R_m)$

G_1

$Y_1 ::= R_0$

$(R_0,\dots,R_N)::=(0,\dots,0)$

$(R_1,\dots,R_m)::=(X_1,\dots,X_m)$

G_2

$Y_2 ::= R_0$

$(R_0,\dots,R_N)::=(0,\dots,0)$

$\cdots \qquad \cdots \qquad \cdots \qquad \cdots$

$(R_1,\dots,R_m)::=(X_1,\dots,X_m)$

G_n

$Y_n ::= R_0$

$(R_0,\dots,R_N)::=(0,\dots,0)$

$(R_1,\dots,R_n)::=(Y_1,\dots,Y_n)$

F

Partial recursive functions

Examples of recursive definitions

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

$f_1(x) =$ sum of $0, 1, 2, \ldots, x$

Examples of recursive definitions

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

$f_1(x) =$ sum of $0, 1, 2, \ldots, x$

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

$f_2(x) = x$th Fibonacci number

Examples of recursive definitions

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

$f_1(x) =$ sum of $0, 1, 2, \ldots, x$

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

$f_2(x) = x$th Fibonacci number

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

$f_3(x)$ undefined except when $x = 0$

Examples of recursive definitions

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

$f_1(x) =$ sum of $0, 1, 2, \ldots, x$

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

$f_2(x) = x$th Fibonacci number

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

$f_3(x)$ undefined except when $x = 0$

$$\begin{aligned} f_4(x) \equiv \ & \mathit{if}\ x > 100\ \mathit{then}\ x - 10 \\ & \mathit{else}\ f_4(f_4(x+11)) \end{aligned}$$

f_4 is McCarthy's "91 function", which maps x to 91 if $x \leq 100$ and to $x - 10$ otherwise

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 minimization.

The characterization is due to Kleene (1936), building on work of Gödel and Herbrand.

Primitive recursion

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

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

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

We write $\rho^n(f, g)$ for h and call it the partial function defined by primitive recursion from f and g.

Primitive recursion

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

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

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

Proof (sketch). *Existence*: the set

$$h \triangleq \{(\vec{x}, x, y) \in \mathbb{N}^{n+2} \mid \exists y_0, y_1, \ldots, y_x \\ f(\vec{x}) = y_0 \wedge (\textstyle\bigwedge_{i=0}^{x-1} g(\vec{x}, i, y_i) = y_{i+1}) \wedge y_x = y\}$$

defines a partial function satisfying $(*)$.

Uniqueness: if h and h' both satisfy $(*)$, then one can prove by induction on x that $\forall \vec{x}\ (h(\vec{x}, x) \equiv h'(\vec{x}, x))$.

Example: addition

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

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

So $\mathit{add} = \rho^1(f, g)$ where $\begin{cases} f(x_1) & \triangleq x_1 \\ g(x_1, x_2, x_3) & \triangleq x_3 + 1 \end{cases}$

Note that $f = \mathtt{proj}^1_1$ and $g = \mathtt{succ} \circ \mathtt{proj}^3_3$; so add can be built up from basic functions using composition and primitive recursion: $\mathit{add} = \rho^1(\mathtt{proj}^1_1, \mathtt{succ} \circ \mathtt{proj}^3_3)$.

Example: predecessor

Predecessor $\mathit{pred} \in \mathbb{N}\to\mathbb{N}$ satisfies:

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

So $\mathit{pred} = \rho^0(f,g)$ where $\begin{cases} f() & \triangleq 0 \\ g(x_1,x_2) & \triangleq x_1 \end{cases}$

Thus pred can be built up from basic functions using primitive recursion: $\mathit{pred} = \rho^0(\mathtt{zero}^0, \mathtt{proj}^2_1)$.

Example: multiplication

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

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

and thus $\mathit{mult} = \rho^1(\mathtt{zero}^1, \mathit{add} \circ (\mathtt{proj}^3_3, \mathtt{proj}^3_1))$.

So mult can be built up from basic functions using composition and primitive recursion (since add can be).

Definition. A [partial] function f is primitive recursive ($f \in \mathbf{PRIM}$) 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 $\mathbf{PRIM}$ 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.

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

Every $f \in \mathbf{PRIM}$ is a total function, because:

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

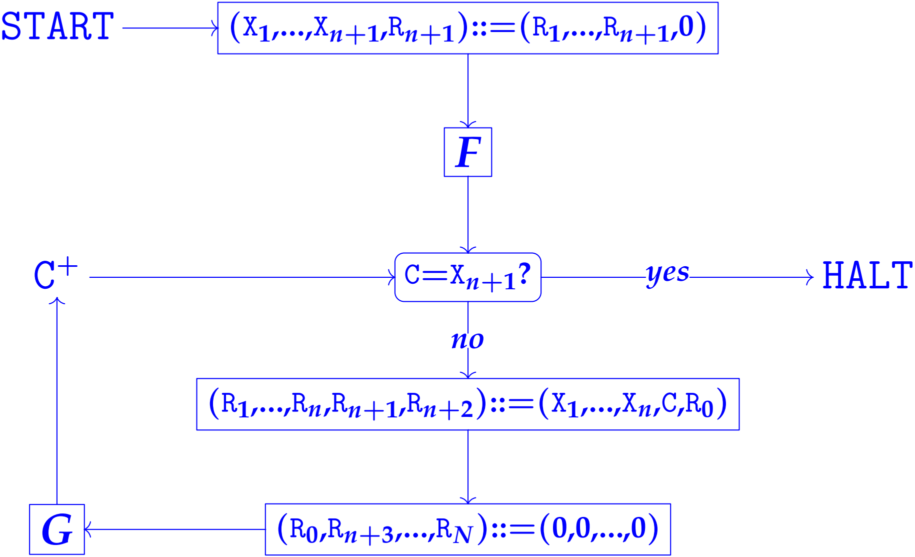
Definition. A [partial] function f is primitive recursive ($f \in \mathbf{PRIM}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition and primitive recursion.

Theorem. Every $f \in \mathbf{PRIM}$ is computable.

Proof. Already proved: basic functions are computable; composition preserves computability. So just have to show:

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

Suppose f and g are computed by RM programs F and G (with our usual I/O conventions). Then the RM specified on the next slide computes $\rho^n(f,g)$. (We assume $\mathtt{X}_1, \ldots, \mathtt{X}_{n+1}, \mathtt{C}$ are some registers not mentioned in F and G; and that the latter only use registers $\mathtt{R}_0, \ldots, \mathtt{R}_N$, where $N \geq n+2$.)



START

$(\mathtt{X}_1,\ldots,\mathtt{X}_{n+1},\mathtt{R}_{n+1})::=(\mathtt{R}_1,\ldots,\mathtt{R}_{n+1},0)$

F

$\mathtt{C}=\mathtt{X}_{n+1}?$

yes

HALT

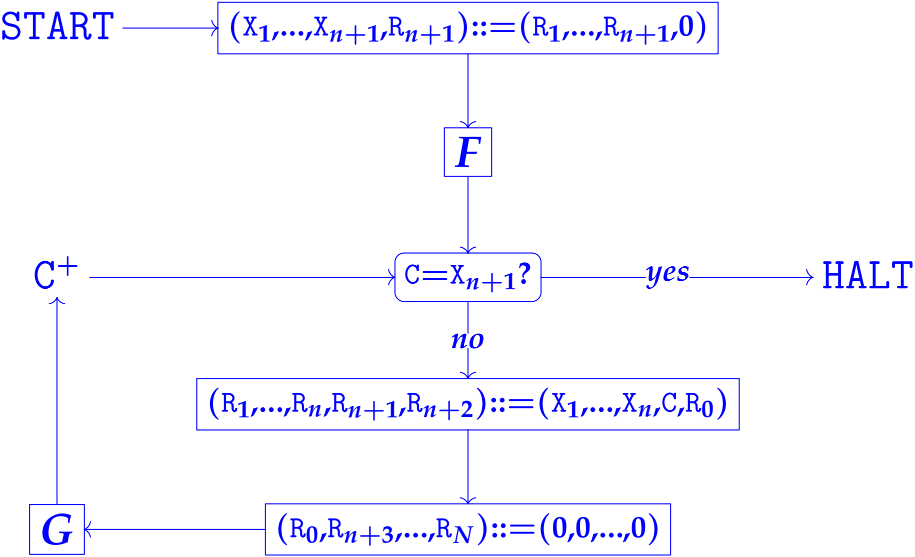
no

$(\mathtt{R}_1,\ldots,\mathtt{R}_n,\mathtt{R}_{n+1},\mathtt{R}_{n+2})::=(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{C},\mathtt{R}_0)$

$(\mathtt{R}_0,\mathtt{R}_{n+3},\ldots,\mathtt{R}_N)::=(0,0,\ldots,0)$

G

$\mathtt{C}^+$



START

$(X_1,\dots,X_{n+1},R_{n+1}) ::= (R_1,\dots,R_{n+1},0)$

F

$C = X_{n+1}?$

yes

HALT

no

C^+

$(R_1,\dots,R_n,R_{n+1},R_{n+2}) ::= (X_1,\dots,X_n,C,R_0)$

$(R_0,R_{n+3},\dots,R_N) ::= (0,0,\dots,0)$

G

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 minimization.

The characterization is due to Kleene (1936), building on work of Gödel and Herbrand.

Minimization

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

$$\mu^n f(\vec{x}) \triangleq \begin{array}{l} \text{least } x \text{ such that } f(\vec{x}, x) = 0 \text{ and} \\ \text{for each } i = 0, \ldots, x-1,\ f(\vec{x}, i) \\ \text{is defined and } > 0 \\ \text{(undefined if there is no such } x\text{)} \end{array}$$

In other words

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

Example of minimization

integer part of x_1/x_2 $\equiv$ least x_3 such that

(undefined if $x_2{=}0$) $x_1 < x_2(x_3+1)$

(In fact, if we make the 'integer part of x_1/x_2' function total by defining it to be $\mathbf{0}$ when $x_2 = \mathbf{0}$, it can be shown to be in **PRIM**.)

Example of minimization

$$\begin{array}{rcl} \text{integer part of } x_1/x_2 & \equiv & \text{least } x_3 \text{ such that} \\ \text{(undefined if } x_2{=}0\text{)} & & x_1 < x_2(x_3+1) \\ & & \\ & \equiv & \mu^2 f(x_1, x_2) \end{array}$$

where $f \in \mathbb{N}^3 \to \mathbb{N}$ is

$$f(x_1, x_2, x_3) \triangleq \begin{cases} 1 & \text{if } x_1 \geq x_2(x_3+1) \\ 0 & \text{if } x_1 < x_2(x_3+1) \end{cases}$$

(In fact, if we make the 'integer part of x_1/x_2' function total by defining it to be $\mathbf{0}$ when $x_2 = \mathbf{0}$, it can be shown to be in **PRIM**.)

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

In other words, the set $\mathbf{PR}$ 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 minimization.

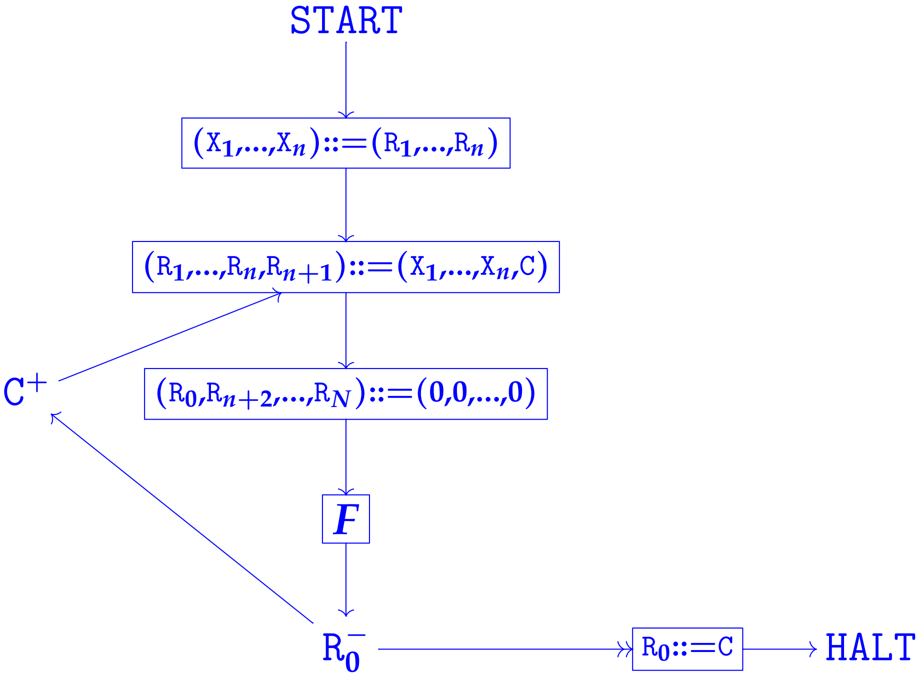
Definition. A partial function f is partial recursive ($f \in \mathbf{PR}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition, primitive recursion and minimization.

Theorem. Every $f \in \mathbf{PR}$ is computable.

Proof. Just have to show:

$\mu^n f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is computable if $f \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ is.

Suppose f is computed by RM program F (with our usual I/O conventions). Then the RM specified on the next slide computes $\mu^n f$. (We assume $\mathtt{X}_1, \ldots, \mathtt{X}_n, \mathtt{C}$ are some registers not mentioned in F; and that the latter only uses registers $\mathtt{R}_0, \ldots, \mathtt{R}_N$, where $N \geq n + 1$.)



START

$(\mathtt{X}_1,\ldots,\mathtt{X}_n) ::= (\mathtt{R}_1,\ldots,\mathtt{R}_n)$

$(\mathtt{R}_1,\ldots,\mathtt{R}_n,\mathtt{R}_{n+1}) ::= (\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{C})$

$(\mathtt{R}_0,\mathtt{R}_{n+2},\ldots,\mathtt{R}_N) ::= (0,0,\ldots,0)$

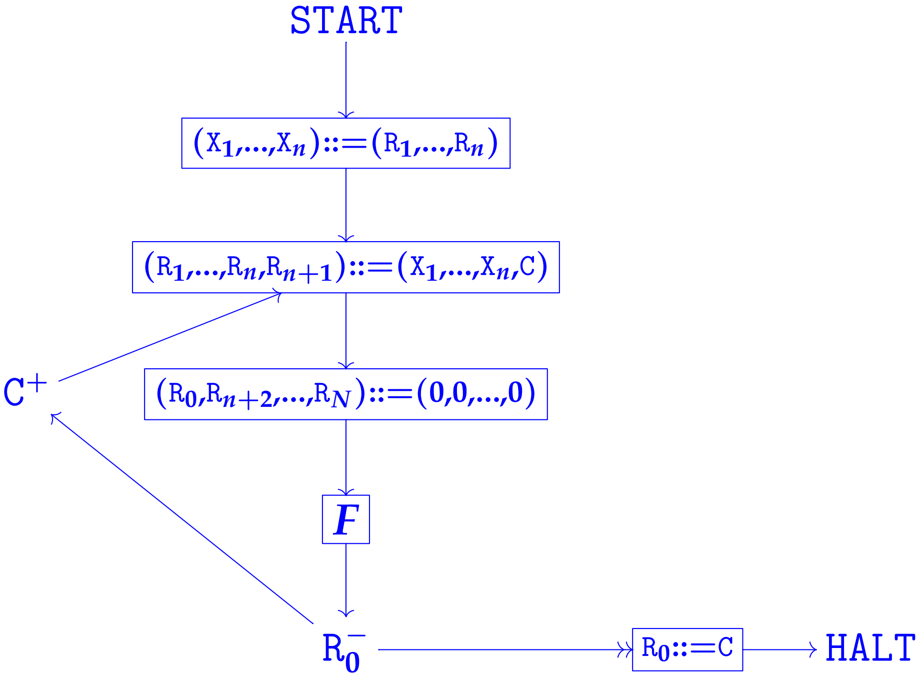
F

$\mathtt{R}_0^-$

$\mathtt{C}^+$

$\mathtt{R}_0 ::= \mathtt{C}$

HALT



START

$(X_1,\dots,X_n) ::= (R_1,\dots,R_n)$

$(R_1,\dots,R_n,R_{n+1}) ::= (X_1,\dots,X_n,C)$

C^+

$(R_0,R_{n+2},\dots,R_N) ::= (0,0,\dots,0)$

F

R_0^-

$R_0 ::= C$

HALT

Computable = partial recursive

Theorem. Not only is every $f \in \mathbf{PR}$ computable, but conversely, every computable partial function is partial recursive.

Proof (sketch). Let $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ be computed by RM M with $N \geq n$ registers, say. Recall how we coded instantaneous configurations $c = (\ell, r_0, \ldots, r_N)$ of M as numbers $\ulcorner [\ell, r_0, \ldots, r_N] \urcorner$. It is possible to construct primitive recursive functions $lab, val_0, next_M \in \mathbb{N} \to \mathbb{N}$ satisfying

$$
\begin{aligned}
lab(\ulcorner [\ell, r_0, \ldots, r_N] \urcorner) &= \ell \\
val_0(\ulcorner [\ell, r_0, \ldots, r_N] \urcorner) &= r_0 \\
next_M(\ulcorner [\ell, r_0, \ldots, r_N] \urcorner) &= \text{code of } M\text{'s next configuration}
\end{aligned}
$$

(Showing that $next_M \in \mathbf{PRIM}$ is tricky—proof omitted.)

Proof sketch, cont.

Writing $\vec{x}$ for $x_1, \ldots, x_n$, let $config_M(\vec{x}, t)$ be the code of M's configuration after t steps, starting with initial register values $\mathtt{R}_0 = 0, \mathtt{R}_1 = x_1, \ldots, \mathtt{R}_n = x_n, \mathtt{R}_{n+1} = 0, \ldots, \mathtt{R}_N = 0$. It's in **PRIM** because:

$$\begin{cases} config_M(\vec{x}, 0) & = \ulcorner [0, 0, \vec{x}, \vec{0}] \urcorner \\ config_M(\vec{x}, t+1) & = next_M(config_M(\vec{x}, t)) \end{cases}$$

Proof sketch, cont.

Writing $\vec{x}$ for $x_1, \ldots, x_n$, let $config_M(\vec{x}, t)$ be the code of M's configuration after t steps, starting with initial register values $\mathtt{R}_0 = 0, \mathtt{R}_1 = x_1, \ldots, \mathtt{R}_n = x_n, \mathtt{R}_{n+1} = 0, \ldots, \mathtt{R}_N = 0$. It's in **PRIM** because:

$$\begin{cases} config_M(\vec{x}, 0) & = \ulcorner [0, 0, \vec{x}, \vec{0}] \urcorner \\ config_M(\vec{x}, t+1) & = next_M(config_M(\vec{x}, t)) \end{cases}$$

Can assume M has a single HALT as last instruction, Ith say (and no erroneous halts). Let $halt_M(\vec{x})$ be the number of steps M takes to halt when started with initial register values $\vec{x}$ (undefined if M does not halt). It satisfies

$$halt_M(\vec{x}) \equiv \text{least } t \text{ such that } I - lab(config_M(\vec{x}, t)) = 0$$

and hence is in **PR** (because $lab, config_M, I - (\ ) \in$ **PRIM**).

Proof sketch, cont.

Writing $\vec{x}$ for $x_1, \ldots, x_n$, let $\mathit{config}_M(\vec{x}, t)$ be the code of M's configuration after t steps, starting with initial register values $\mathtt{R}_0 = 0, \mathtt{R}_1 = x_1, \ldots, \mathtt{R}_n = x_n, \mathtt{R}_{n+1} = 0, \ldots, \mathtt{R}_N = 0$. It's in $\mathbf{PRIM}$ because:

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

Can assume M has a single $\mathtt{HALT}$ as last instruction, Ith say (and no erroneous halts). Let $\mathit{halt}_M(\vec{x})$ be the number of steps M takes to halt when started with initial register values $\vec{x}$ (undefined if M does not halt). It satisfies

$$\mathit{halt}_M(\vec{x}) \equiv \text{least } t \text{ such that } I - \mathit{lab}(\mathit{config}_M(\vec{x}, t)) = 0$$

and hence is in $\mathbf{PR}$ (because $\mathit{lab}, \mathit{config}_M, I - (\ ) \in \mathbf{PRIM}$).

So $f \in \mathbf{PR}$, because $f(\vec{x}) \equiv \mathit{val}_0(\mathit{config}_M(\vec{x}, \mathit{halt}_M(\vec{x})))$.

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

The members of $\mathbf{PR}$ that are total are called recursive functions.

Fact: there are recursive functions that are not primitive recursive. For example. . .

Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying

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

Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying

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

- ► ack is computable, hence recursive [proof: exercise].

Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying

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

- ack is computable, hence recursive [proof: exercise].
- **Fact:** ack 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.

Lambda calculus

Notions of computability

- Church (1936): λ-calculus
- Turing (1936): Turing machines.

Turing showed that the two very different approaches determine the same class of computable functions. Hence:

Church-Turing Thesis. Every algorithm [in intuitive sense of Lect. 1] can be realized as a Turing machine.

λ-Terms, M

are built up from a given, countable collection 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: $(M\,M')$
  (where M and M' are λ-terms).

λ-Terms, M

are built up from a given, countable collection 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: $(M\,M')$
  (where M and M' are λ-terms).

Some random examples of λ-terms:

$$x \quad (\lambda x.x) \quad ((\lambda y.(x\,y))x) \quad (\lambda y.((\lambda y.(x\,y))x))$$

λ-Terms, M

Notational conventions:

- $(\lambda x_1\, x_2 \ldots x_n.M)$ means $(\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.M) \ldots))$
- $(M_1\, M_2 \ldots M_n)$ means $(\ldots (M_1\, M_2) \ldots M_n)$ (i.e. application is left-associative)
- drop outermost parentheses and those enclosing the body of a λ-abstraction. E.g. write $(\lambda x.(x(\lambda y.(y\, x))))$ as $\lambda x.x(\lambda y.y\, x)$.
- $x \# M$ means that the variable x does not occur anywhere in the λ-term M.

Free and 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

- binding if in between λ and $.$
  (e.g. $(\lambda x.y\,x)\,x$)
- bound if in the body of a binding occurrence of x
  (e.g. $(\lambda x.y\,x)\,x$)
- free if neither binding nor bound
  (e.g. $(\lambda x.y\,x)x$).

Free and bound variables

Sets of free and bound variables:

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(\lambda x.M) & = & FV(M) - \{x\} \\ FV(M\,N) & = & FV(M) \cup FV(N) \end{array}$$

$$\begin{array}{rcl} BV(x) & = & \emptyset \\ BV(\lambda x.M) & = & BV(M) \cup \{x\} \\ BV(M\,N) & = & BV(M) \cup BV(N) \end{array}$$

If $FV(M) = \emptyset$, M is called a closed term, or combinator.

α-Equivalence $M =_\alpha M'$

$\lambda x.M$ is intended to represent the function f such that

$$f(x) = M \text{ for all } x.$$

So the name of the bound variable is immaterial: if $M' = M\{x'/x\}$ 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.

For example, $\lambda x.x$ and $\lambda y.y$ represent the same function (the identity function).

α-Equivalence $M =_\alpha M'$

is the binary relation inductively generated by the rules:

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

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

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

α-Equivalence $M =_\alpha M'$

For example:

$$\begin{array}{rc}
 & \lambda x.(\lambda x x'.x)\, x' =_\alpha \lambda y.(\lambda x\, x'.x) x' \\
\text{because} & (\lambda z\, x'.z) x' =_\alpha (\lambda x\, x'.x) x' \\
\text{because} & \lambda z\, x'.z =_\alpha \lambda x\, x'.x \text{ and } x' =_\alpha x' \\
\text{because} & \lambda x'.u =_\alpha \lambda x'.u \text{ and } x' =_\alpha x' \\
\text{because} & u =_\alpha u \text{ and } x' =_\alpha x'.
\end{array}$$

α-Equivalence $M =_\alpha M'$

Fact: $=_\alpha$ is an equivalence relation (reflexive, symmetric and transitive).

We do not care about the particular names of bound variables, just about the distinctions between them. So α-equivalence classes of λ-terms are more important than λ-terms themselves.

- Textbooks (and these lectures) suppress any notation for α-equivalence classes and refer to an equivalence class via a representative λ-term (look for phrases like "we identify terms up to α-equivalence" or "we work up to α-equivalence").
- For implementations and computer-assisted reasoning, there are various devices for picking canonical representatives of α-equivalence classes (e.g. de Bruijn indexes, graphical representations, . . .).

Substitution $N[M/x]$

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

Substitution $N[M/x]$

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

Side-condition $y \# (M\ x)$ (y does not occur in M and $y \neq x$) makes substitution "capture-avoiding".

E.g. if $x \neq y$

$$(\lambda y.x)[y/x] \neq \lambda y.y$$

Substitution $N[M/x]$

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

Side-condition $y \mathrel{\#} (M\ x)$ (y does not occur in M and $y \neq x$) makes substitution "capture-avoiding".

E.g. if $x \neq y \neq z \neq x$

$$(\lambda y.x)[y/x] =_\alpha (\lambda z.x)[y/x] = \lambda z.y$$

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

β-Reduction

Recall that $\lambda x.M$ is intended to represent the function f such that $f(x) = M$ for all x. We can regard $\lambda x.M$ as a function on λ-terms via substitution: map each N to $M[N/x]$.

So the natural notion of computation for λ-terms is given by stepping from a

β-redex $\qquad (\lambda x.M)N$

to the corresponding

β-reduct $\qquad M[N/x]$

β-Reduction

One-step β-reduction, $M \to M'$:

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

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

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

β-Reduction

E.g.

$$\begin{tikzcd}
 & ((\lambda y.\lambda z.z)u)y \arrow[dr] & & \\
(\lambda x.x\,y)((\lambda y.\lambda z.z)u) \arrow[ur] \arrow[dr] & & (\lambda z.z)y \arrow[r] & y \\
 & (\lambda x.x\,y)(\lambda z.z) \arrow[ur] & &
\end{tikzcd}$$

β-Reduction

E.g.

$$\begin{tikzcd}
 & ((\lambda y.\lambda z.z)u)y \arrow[dr] & & \\
(\lambda x.x\,y)((\lambda y.\lambda z.z)u) \arrow[ur] \arrow[dr] & & (\lambda z.z)y \arrow[r] & y \\
 & (\lambda x.x\,y)(\lambda z.z) \arrow[ur] & &
\end{tikzcd}$$

β-Reduction

E.g.

$$\begin{tikzcd}
 & ((\lambda y.\lambda z.z)u)y \arrow[dr] & & \\
(\lambda x.x\,y)((\lambda y.\lambda z.z)u) \arrow[ur] \arrow[dr] & & (\lambda z.z)y \arrow[r] & y \\
 & (\lambda x.x\,y)(\lambda z.z) \arrow[ur] & &
\end{tikzcd}$$

β-Reduction

E.g.

$$\begin{tikzcd}
 & ((\lambda y.\lambda z.z)u)y \arrow[dr] & & \\
(\lambda x.x\,y)((\lambda y.\lambda z.z)u) \arrow[ur] \arrow[dr] & & (\lambda z.z)y \arrow[r] & y \\
 & (\lambda x.x\,y)(\lambda z.z) \arrow[ur] & &
\end{tikzcd}$$

β-Reduction

E.g.

$$\begin{tikzcd}
 & ((\lambda y.\lambda z.z)u)y \arrow[dr] & & \\
(\lambda x.x\,y)((\lambda y.\lambda z.z)u) \arrow[ur] \arrow[dr] & & (\lambda z.z)y \arrow[r] & y \\
 & (\lambda x.x\,y)(\lambda z.z) \arrow[ur] & &
\end{tikzcd}$$

β-Reduction

E.g.

$$\begin{tikzcd}
 & ((\lambda y.\lambda z.z)u)y \arrow[dr] & & \\
(\lambda x.x\,y)((\lambda y.\lambda z.z)u) \arrow[ur] \arrow[dr] & & (\lambda z.z)y \arrow[r] & y \\
 & (\lambda x.x\,y)(\lambda z.z) \arrow[ur] & &
\end{tikzcd}$$

E.g. of "up to α-equivalence" aspect of reduction:

$$(\lambda x.\lambda y.x)y =_{\alpha} (\lambda x.\lambda z.x)y \to \lambda z.y$$

Many-step β-reduction, $M \twoheadrightarrow M'$:

$$\frac{M =_{\alpha} M'}{M \twoheadrightarrow M'}$$

(no steps)

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

(1 more step)

E.g.

$$(\lambda x.x\,y)((\lambda y\,z.z)u) \twoheadrightarrow y$$

$$(\lambda x.\lambda y.x)y \twoheadrightarrow \lambda z.y$$

β-Conversion $M =_\beta N$

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

E.g. $u\,((\lambda x\,y.\,v\,x)y) =_\beta (\lambda x.\,u\,x)(\lambda x.\,v\,y)$

because $(\lambda x.\,u\,x)(\lambda x.\,v\,y) \to u(\lambda x.\,v\,y)$

and so we have

$$\begin{array}{lcll}
u\,((\lambda x\,y.\,v\,x)y) & =_\alpha & u\,((\lambda x\,y'.\,v\,x)y) & \\
 & \to & u(\lambda y'.\,v\,y) & \text{reduction} \\
 & =_\alpha & u(\lambda x.\,v\,y) & \\
 & \leftarrow & (\lambda x.\,u\,x)(\lambda x.\,v\,y) & \text{expansion}
\end{array}$$

β-Conversion $M =_\beta N$

is the binary relation inductively generated by the rules:

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

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

$$\frac{M =_\beta M' \qquad N =_\beta N'}{M\,N =_\beta M'\,N'}$$

Church-Rosser Theorem

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

[Proof omitted.]

Church-Rosser Theorem

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

Corollary. To show that two terms are β-convertible, it suffices to show that they both reduce to the same term. More precisely: $M_1 =_\beta M_2$ iff $\exists M\,(M_1 \twoheadrightarrow M \twoheadleftarrow M_2)$.

Church-Rosser Theorem

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

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 $M =_\beta M'$. Thus if $M_1 \twoheadrightarrow M \twoheadleftarrow M_2$, then $M_1 =_\beta M =_\beta M_2$ and so $M_1 =_\beta M_2$.

Conversely, the relation $\{(M_1, M_2) \mid \exists M\,(M_1 \twoheadrightarrow M \twoheadleftarrow M_2)\}$ 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: $M_1 \longrightarrow\!\!\!\!\!\rightarrow M \twoheadleftarrow\!\!\!\!\!\longleftarrow M_2 \longrightarrow\!\!\!\!\!\rightarrow M' \twoheadleftarrow\!\!\!\!\!\longleftarrow M_3$

Church-Rosser Theorem

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

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 $M =_\beta M'$. Thus if $M_1 \twoheadrightarrow M \twoheadleftarrow M_2$, then $M_1 =_\beta M =_\beta M_2$ and so $M_1 =_\beta M_2$.

Conversely, the relation $\{(M_1, M_2) \mid \exists M\,(M_1 \twoheadrightarrow M \twoheadleftarrow M_2)\}$ 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:

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

Church-Rosser Theorem

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

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 $M =_\beta M'$. Thus if $M_1 \twoheadrightarrow M \twoheadleftarrow M_2$, then $M_1 =_\beta M =_\beta M_2$ and so $M_1 =_\beta M_2$.

Conversely, the relation $\{(M_1, M_2) \mid \exists M \, (M_1 \twoheadrightarrow M \twoheadleftarrow M_2)\}$ 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 $\exists M \, (M_1 \twoheadrightarrow M \twoheadleftarrow M_2)$.

β-Normal Forms

Definition. A λ-term N is in β-normal form (nf) 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 a β-nf.

β-Normal Forms

Definition. A λ-term N is in β-normal form (nf) 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 a β-nf.

Note that if N is a β-nf and $N \twoheadrightarrow N'$, then it must be that $N =_\alpha N'$ (why?).

Hence if $N_1 =_\beta N_2$ with N_1 and N_2 both β-nfs, then $N_1 =_\alpha N_2$. (For 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-nf of M is unique up to α-equivalence if it exists.

Non-termination

Some λ terms have no β-nf.

E.g. $\Omega \triangleq (\lambda x.x\,x)(\lambda x.x\,x)$ satisfies

- $\Omega \to (x\,x)[(\lambda x.x\,x)/x] = \Omega$,
- $\Omega \twoheadrightarrow M$ implies $\Omega =_\alpha M$.

So there is no β-nf N such that $\Omega =_\beta N$.

Non-termination

Some λ terms have no β-nf.

E.g. $\Omega \triangleq (\lambda x.x\,x)(\lambda x.x\,x)$ satisfies

- $\Omega \to (x\,x)[(\lambda x.x\,x)/x] = \Omega$,
- $\Omega \twoheadrightarrow M$ implies $\Omega =_\alpha M$.

So there is no β-nf N such that $\Omega =_\beta N$.

A term can possess both a β-nf and infinite chains of reduction from it.

E.g. $(\lambda x.y)\Omega \to y$, but also $(\lambda x.y)\Omega \to (\lambda x.y)\Omega \to \cdots$.

Non-termination

Normal-order reduction is a deterministic strategy for reducing λ-terms: reduce the "left-most, outer-most" redex first. More specifically:

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

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

where the redex is the underlined subterm. 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 $\lambda x_1 \ldots \lambda x_n. x\, M_1\, M_2 \ldots M_m$ $(m, n \geq 0)$.
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.

Fact: normal-order reduction of M always reaches the β-nf of M if it possesses one.

Lambda-Definable Functions

Encoding data in λ-calculus

Computation in λ-calculus is given by β-reduction. To relate this to register/Turing-machine computation, or to partial recursive functions, we first have to see how to encode numbers, pairs, lists, ... as λ-terms.

We will use the original encoding of numbers due to Church...

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: $\begin{cases} M^0 N & \triangleq N \\ M^1 N & \triangleq M\, N \\ M^{n+1} N & \triangleq M(M^n N) \end{cases}$

so we can write $\underline{n}$ as $\lambda f\, x.f^n x$ and we have $\boxed{\underline{n}\, M\, N =_\beta M^n\, N}$.

λ-Definable functions

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ 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 $F\,\underline{x_1} \cdots \underline{x_n} =_\beta \underline{y}$
- if $f(x_1, \ldots, x_n)\uparrow$, then $F\,\underline{x_1} \cdots \underline{x_n}$ has no β-nf.

For example, addition is λ-definable because it is represented by $P \triangleq \lambda x_1\, x_2.\lambda f\, x.\, x_1\, f(x_2\, f\, x)$:

$$\begin{aligned} P\,\underline{m}\,\underline{n} &=_\beta \lambda f\, x.\,\underline{m}\,f(\underline{n}\,f\,x) \\ &=_\beta \lambda f\, x.\,\underline{m}\,f(f^n x) \\ &=_\beta \lambda f\, x.\, f^m(f^n x) \\ &= \lambda f\, x. f^{m+n} x \\ &= \underline{m+n} \end{aligned}$$

Computable = λ-definable

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

We already know that

$$\begin{aligned} & \text{Register Machine computable} \\ = \; & \text{Turing computable} \\ = \; & \text{partial recursive.} \end{aligned}$$

Using this, we break the theorem into two parts:

- every partial recursive function is λ-definable
- λ-definable functions are RM computable

λ-Definable functions

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ 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 $F\,\underline{x_1} \cdots \underline{x_n} =_\beta \underline{y}$
- if $f(x_1, \ldots, x_n)\uparrow$, then $F\,\underline{x_1} \cdots \underline{x_n}$ has no β-nf.

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

For now, we concentrate on total functions. First, let us see why the elements of **PRIM** (primitive recursive functions) are λ-definable.

Basic functions

- Projection functions, $\mathtt{proj}_i^n \in \mathbb{N}^n \to \mathbb{N}$:

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

- Constant functions with value $\mathbf{0}$, $\mathtt{zero}^n \in \mathbb{N}^n \to \mathbb{N}$:

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

- Successor function, $\mathtt{succ} \in \mathbb{N} \to \mathbb{N}$:

$$\mathtt{succ}(x) \triangleq x + \mathbf{1}$$

Basic functions are representable

- $\mathtt{proj}_i^n \in \mathbb{N}^n{\to}\mathbb{N}$ is represented by $\lambda x_1 \ldots x_n.x_i$
- $\mathtt{zero}^n \in \mathbb{N}^n{\to}\mathbb{N}$ is represented by $\lambda x_1 \ldots x_n.\underline{0}$
- $\mathtt{succ} \in \mathbb{N}{\to}\mathbb{N}$ is represented by

$$\mathbf{Succ} \triangleq \lambda x_1\, f\, x.f\,(x_1\, f\, x)$$

since

$$\begin{aligned}\mathbf{Succ}\,\underline{n} &=_\beta \lambda f\, x.\, f\,(\underline{n}\, f\, x)\\ &=_\beta \lambda f\, x.\, f\,(f^n\, x)\\ &= \lambda f\, x.\, f^{n+1}\, x\\ &= \underline{n+1}\end{aligned}$$

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 their composition $f \circ (g_1, \ldots, g_n) \in \mathbb{N}^m \to \mathbb{N}$ is represented simply by

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

because

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

Representing composition

If total function $f \in \mathbb{N}^n \rightarrow \mathbb{N}$ is represented by F and total functions $g_1, \dots, g_n \in \mathbb{N}^m \rightarrow \mathbb{N}$ are represented by $G_1, \dots, G_n$, then their composition $f \circ (g_1, \dots, g_n) \in \mathbb{N}^m \rightarrow \mathbb{N}$ is represented simply by

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

This does not necessarily work for partial functions. E.g. totally undefined function $u \in \mathbb{N} \rightharpoonup \mathbb{N}$ is represented by $U \triangleq \lambda x_1 . \Omega$ (why?) and $\mathtt{zero}^1 \in \mathbb{N} \rightarrow \mathbb{N}$ is represented by $Z \triangleq \lambda x_1 . \underline{0}$; but $\mathtt{zero}^1 \circ u$ is not represented by $\lambda x_1 . Z(U\,x_1)$, because $(\mathtt{zero}^1 \circ u)(n){\uparrow}$ whereas $(\lambda x_1 . Z(U\,x_1))\,\underline{n} =_\beta Z\,\Omega =_\beta \underline{0}$. (What is $\mathtt{zero}^1 \circ u$ represented by?)

Primitive recursion

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

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

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

We write $\rho^n(f, g)$ for h and call it the partial function defined by primitive recursion from f and g.

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 $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfying

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

or equivalently

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

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 $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfying $\boxed{h = \Phi_{f,g}(h)}$

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

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

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 $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfying $\boxed{h = \Phi_{f,g}(h)}$

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

Strategy:

- show that $\Phi_{f,g}$ is λ-definable;
- show that we can solve fixed point equations $\boxed{X = M\,X}$ up to β-conversion in the λ-calculus.

Representing booleans

$$\begin{aligned}\textbf{True} &\triangleq \lambda x\, y.\, x\\ \textbf{False} &\triangleq \lambda x\, y.\, y\\ \textbf{If} &\triangleq \lambda f\, x\, y.\, f\, x\, y\end{aligned}$$

satisfy

- $\textbf{If}\ \textbf{True}\ M\ N =_\beta \textbf{True}\ M\ N =_\beta M$
- $\textbf{If}\ \textbf{False}\ M\ N =_\beta \textbf{False}\ M\ N =_\beta N$

Representing test-for-zero

$$\mathsf{Eq}_0 \triangleq \lambda x.\, x(\lambda y.\, \mathsf{False})\, \mathsf{True}$$

satisfies

- $\mathsf{Eq}_0\, \underline{0} \;=_\beta\; \underline{0}\,(\lambda y.\, \mathsf{False})\, \mathsf{True}$
  $\qquad =_\beta\; \mathsf{True}$
- $\mathsf{Eq}_0\, \underline{n+1} \;=_\beta\; \underline{n+1}\,(\lambda y.\, \mathsf{False})\, \mathsf{True}$
  $\qquad =_\beta\; (\lambda y.\, \mathsf{False})^{n+1}\, \mathsf{True}$
  $\qquad =_\beta\; (\lambda y.\, \mathsf{False})((\lambda y.\, \mathsf{False})^{n}\, \mathsf{True})$
  $\qquad =_\beta\; \mathsf{False}$

Representing ordered pairs

$$\begin{aligned}
\mathbf{Pair} &\triangleq \lambda x\, y\, f.\, f\, x\, y \\
\mathbf{Fst} &\triangleq \lambda f.\, f\, \mathbf{True} \\
\mathbf{Snd} &\triangleq \lambda f.\, f\, \mathbf{False}
\end{aligned}$$

satisfy

- $\mathbf{Fst}(\mathbf{Pair}\, M\, N) =_\beta \mathbf{Fst}(\lambda f.\, f\, M\, N)$
  $=_\beta (\lambda f.\, f\, M\, N)\, \mathbf{True}$
  $=_\beta \mathbf{True}\, M\, N$
  $=_\beta M$
- $\mathbf{Snd}(\mathbf{Pair}\, M\, N) =_\beta \cdots =_\beta N$

Representing predecessor

Want λ-term **Pred** satisfying

$$\begin{aligned} \mathbf{Pred}\,\underline{n+1} &=_\beta \underline{n} \\ \mathbf{Pred}\,\underline{0} &=_\beta \underline{0} \end{aligned}$$

Have to show how to reduce the "$n+1$-iterator" $\underline{n+1}$ to the "n-iterator" $\underline{n}$.

Idea: given f, iterating the function

$$g_f : (x, y) \mapsto (f(x), x)$$

$n+1$ times starting from (x, x) gives the pair $(f^{n+1}(x), f^n(x))$. So we can get $f^n(x)$ from $f^{n+1}(x)$ *parametrically in* f *and* x, by building g_f from f, iterating $n+1$ times from (x, x) and then taking the second component.

Hence. . .

Representing predecessor

Want $\boldsymbol{\lambda}$-term **Pred** satisfying

$$\begin{aligned} \mathbf{Pred}\,\underline{n+1} &=_\beta \underline{n} \\ \mathbf{Pred}\,\underline{0} &=_\beta \underline{0} \end{aligned}$$

$$\mathbf{Pred} \triangleq \lambda y\,f\,x.\,\mathbf{Snd}(y\,(G\,f)(\mathbf{Pair}\,x\,x))$$

where

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

has the required $\boldsymbol{\beta}$-reduction properties. [Exercise]

Curry's fixed point combinator $\mathbf{Y}$

$$\mathbf{Y} \triangleq \lambda f.\,(\lambda x.\, f(x\,x))(\lambda x.\, f(x\,x))$$

satisfies $\mathbf{Y}\,M \;\to\; (\lambda x.\, M(x\,x))(\lambda x.\, M(x\,x))$

$\qquad\qquad\quad\;\to\; M((\lambda x.\, M(x\,x))(\lambda x.\, M(x\,x)))$

hence $\mathbf{Y}\,M \twoheadrightarrow M((\lambda x.\, M(x\,x))(\lambda x.\, M(x\,x))) \twoheadleftarrow M(\mathbf{Y}\,M)$.

So for all λ-terms M we have

$$\boxed{\mathbf{Y}\,M =_\beta M(\mathbf{Y}\,M)}$$

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
$h \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfying

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

or equivalently

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

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 $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfying $\boxed{h = \Phi_{f,g}(h)}$

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

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

We now know that h can be represented by

$Y(\lambda z \vec{x} x.\, \mathbf{If}\, (\mathbf{Eq}_0\, x)\, (F\, \vec{x})\, (G\, \vec{x}\, (\mathbf{Pred}\, x)\, (z\, \vec{x}\, (\mathbf{Pred}\, x))))$.

Representing primitive recursion

Recall that the class **PRIM** of primitive recursive functions is the smallest collection of (total) functions containing the basic functions and closed under the operations of composition and primitive recursion.

Combining the results about λ-definability so far, we have: **every $f \in$ PRIM is λ-definable**.

So for λ-definability of all recursive functions, we just have to consider how to represent minimization. Recall. . .

Minimization

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

$$\mu^n f(\vec{x}) \triangleq \begin{array}{l} \text{least } x \text{ such that } f(\vec{x}, x) = 0 \text{ and} \\ \text{for each } i = 0, \ldots, x-1, \ f(\vec{x}, i) \\ \text{is defined and } > 0 \\ (\text{undefined if there is no such } x) \end{array}$$

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 $\boxed{g = \Psi_f(g)}$

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

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

Representing minimization

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

Thus for all $\vec{a} \in \mathbb{N}^n$, $\mu^n f(\vec{a}) = g(\vec{a}, 0)$ with $g = \Psi_f(g)$ and $\Psi_f(g)(\vec{a}, a)$ given by
$\textit{if}\ (f(\vec{a}, a) = 0)\ \textit{then}\ a\ \textit{else}\ g(\vec{a}, a+1)$.

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

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

Recursive implies λ-definable

Fact: every partial recursive $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ can be expressed in a standard form as $f = g \circ (\mu^n h)$ for some $g, h \in \mathbf{PRIM}$. (Follows from the proof that computable = partial-recursive.)

Hence every (total) recursive function is λ-definable.

More generally, every partial recursive function is λ-definable, but matching up $\uparrow$ with $\nexists\beta\mathbf{-nf}$ makes the representations more complicated than for total functions: see [Hindley, J.R. & Seldin, J.P. (CUP, 2008), chapter 4.]

Computable = λ-definable

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

We already know that computable = partial recursive $\Rightarrow$ λ-definable. So it just remains to see that **λ-definable functions are RM computable**. To show this one can

- code λ-terms as numbers (ensuring that operations for constructing and deconstructing terms are given by RM computable functions on codes)
- write a RM interpreter for (normal order) β-reduction.

The details are straightforward, if tedious.