

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 $\mathbb{N} = \{0, 1, 2, \dots\}$ 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** R_0, R_1, \dots, 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 $i = 0, 1, 2, \dots$, the $(i + 1)^{\text{th}}$ instruction has label L_i .

Definition. A register machine is specified by:

- ▶ finitely many registers R_0, R_1, \dots, 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 $i = 0, 1, 2, \dots$, the $(i + 1)^{\text{th}}$ instruction has label L_i .

Instruction *body* takes one of three forms:

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

Example

registers:

R_0 R_1 R_2

program:

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

$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

example computation:

L_i	R_0	R_1	R_2
0	0	1	2

Example

registers:

R_0 R_1 R_2

program:

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

$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

example computation:

L_i	R_0	R_1	R_2
0	0	1	2

Example

registers:

R_0 R_1 R_2

program:

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

$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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:

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

$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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:

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

$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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:

R_0 R_1 R_2

program:

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

$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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

Example

registers:

R_0 R_1 R_2

program:

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

$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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:

R_0 R_1 R_2

program:

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

$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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

Example

registers:

R_0 R_1 R_2

program:

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

$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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:

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

$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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 R_i .

Notation: “ $R_i = x$ [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 R_i .

Notation: “ $R_i = x$ [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 R_i .

Register machine computation

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

$$c_0, c_1, c_2, \dots$$

where

- ▶ $c_0 = (0, r_0, \dots, r_n)$ is an initial configuration
- ▶ each $c = (\ell, r_0, \dots, r_n)$ in the sequence determines the next configuration in the sequence (if any) by carrying out the program instruction labelled L_ℓ with registers containing r_0, \dots, r_n .

Halting

For a finite computation c_0, c_1, \dots, c_m , the last configuration $c_m = (\ell, r, \dots)$ 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 L_ℓ (an “**erroneous halt**”)

Halting

For a finite computation c_0, c_1, \dots, c_m , the last configuration $c_m = (\ell, r, \dots)$ 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 L_ℓ (an “**erroneous halt**”)

E.g.

$L_0 : R_0^+ \rightarrow L_2$
$L_1 : \text{HALT}$

 halts erroneously.

Halting

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

Note that **computations may never halt**. For example,

$L_0 : R_0^+ \rightarrow L_0$ $L_1 : \text{HALT}$
--

 only has infinite computation sequences

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

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
$R^+ \rightarrow L$	$R^+ \longrightarrow [L]$
$R^- \rightarrow L, L'$	$R^- \begin{array}{l} \nearrow [L] \\ \searrow [L'] \end{array}$
HALT	HALT
L_0	$\text{START} \longrightarrow [L_0]$

Example

registers:

R_0 R_1 R_2

program:

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

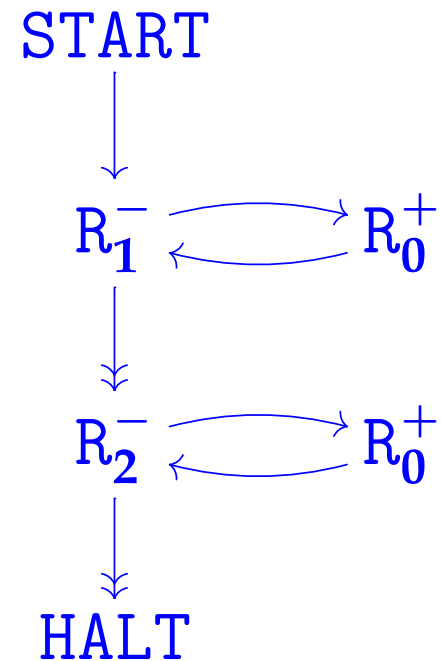
$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

graphical representation:



Example

registers:

R_0 R_1 R_2

program:

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

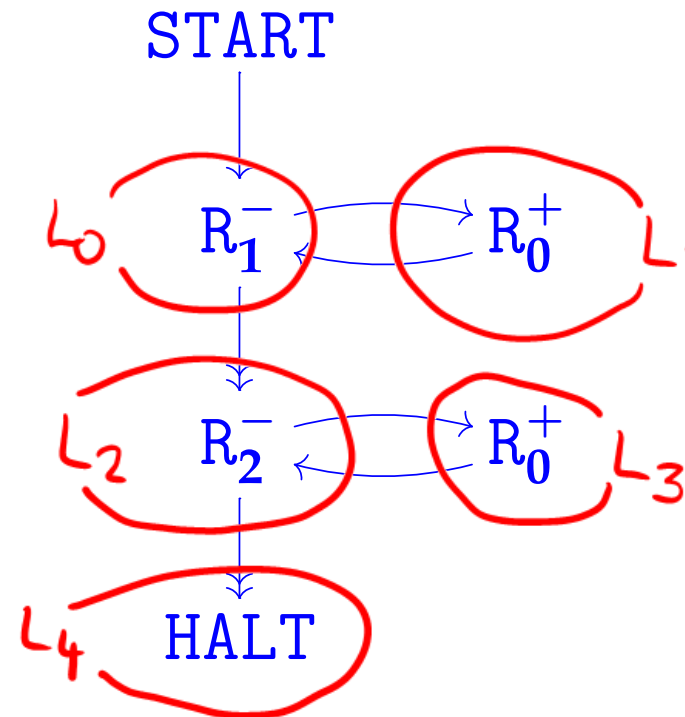
$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

graphical representation:



Graphical representation is helpful for seeing what function a machine computes...

Example

registers:

R_0 R_1 R_2

program:

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

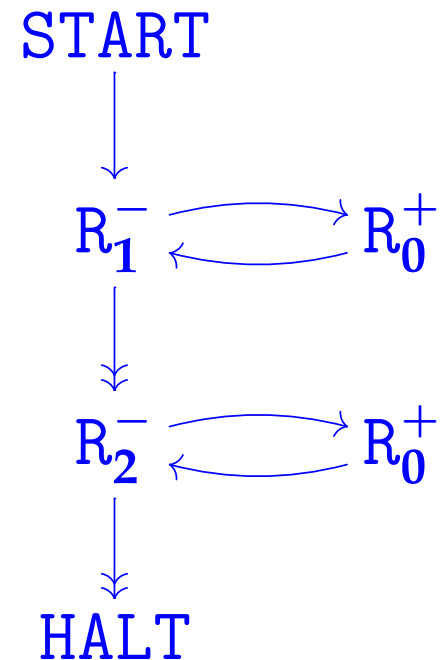
$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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:

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

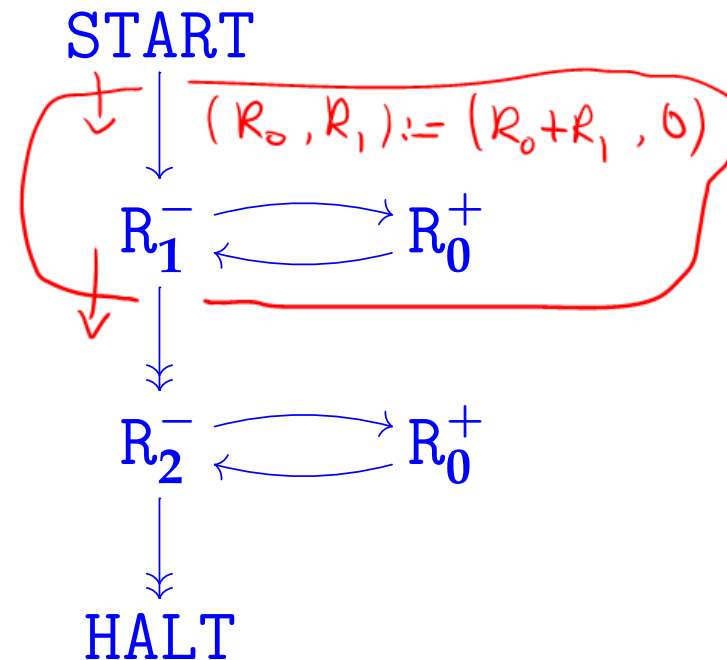
$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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:

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

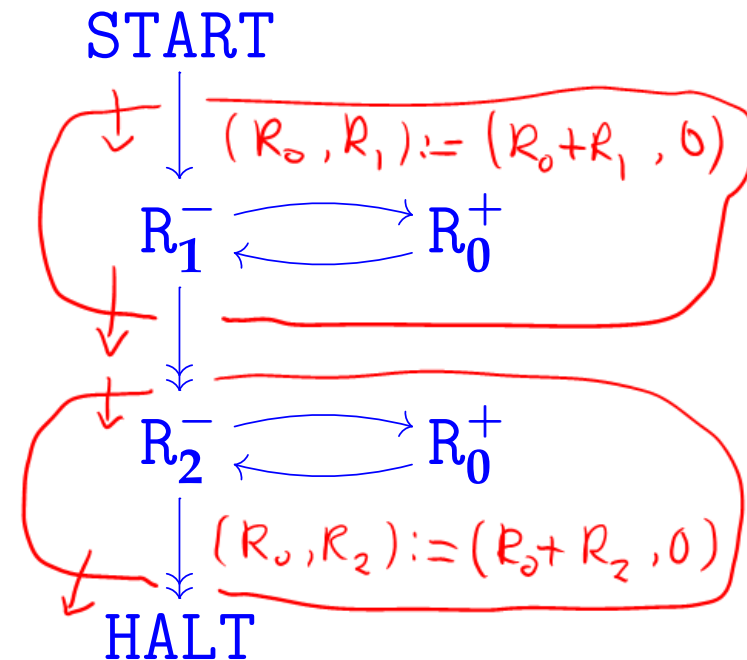
$L_1 : R_0^+ \rightarrow L_0$

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

$L_3 : R_0^+ \rightarrow L_2$

$L_4 : \text{HALT}$

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 $\neg \exists y \in Y (f(x) = y)$
- ▶ $X \dashrightarrow Y$ = 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 $\neg \exists y \in Y (f(x) = y)$
- ▶ $X \dashrightarrow Y$ = 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. $f \in \mathbb{N}^n \rightarrow \mathbb{N}$ is (**register machine**) **computable** if there is a register machine M with at least $n + 1$ registers R_0, R_1, \dots, R_n (and maybe more) such that for all $(x_1, \dots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$,
the computation of M starting with $R_0 = 0$,
 $R_1 = x_1, \dots, R_n = x_n$ and all other registers set to 0 , halts with $R_0 = y$
if and only if $f(x_1, \dots, x_n) = y$.

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

Computable functions

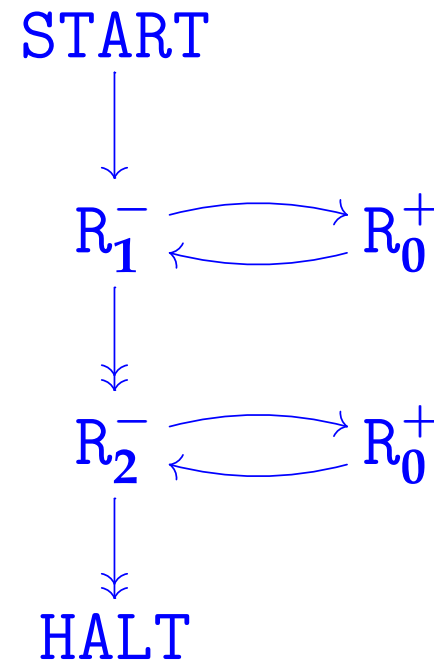
Definition. $f \in \mathbb{N}^n \rightarrow \mathbb{N}$ is (**register machine**) **computable** if there is a register machine M with at least $n + 1$ registers R_0, R_1, \dots, R_n (and maybe more) such that for all $(x_1, \dots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$,
the computation of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

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

Example

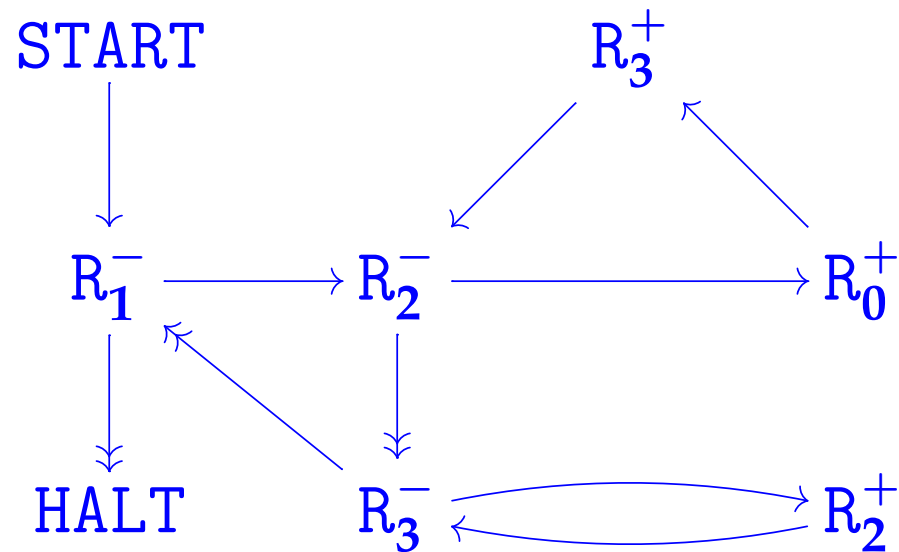
registers:
 $R_0 R_1 R_2$
program:
 $L_0 : R_1^- \rightarrow L_1, L_2$
 $L_1 : R_0^+ \rightarrow L_0$
 $L_2 : R_2^- \rightarrow L_3, L_4$
 $L_3 : R_0^+ \rightarrow L_2$
 $L_4 : \text{HALT}$

graphical representation:

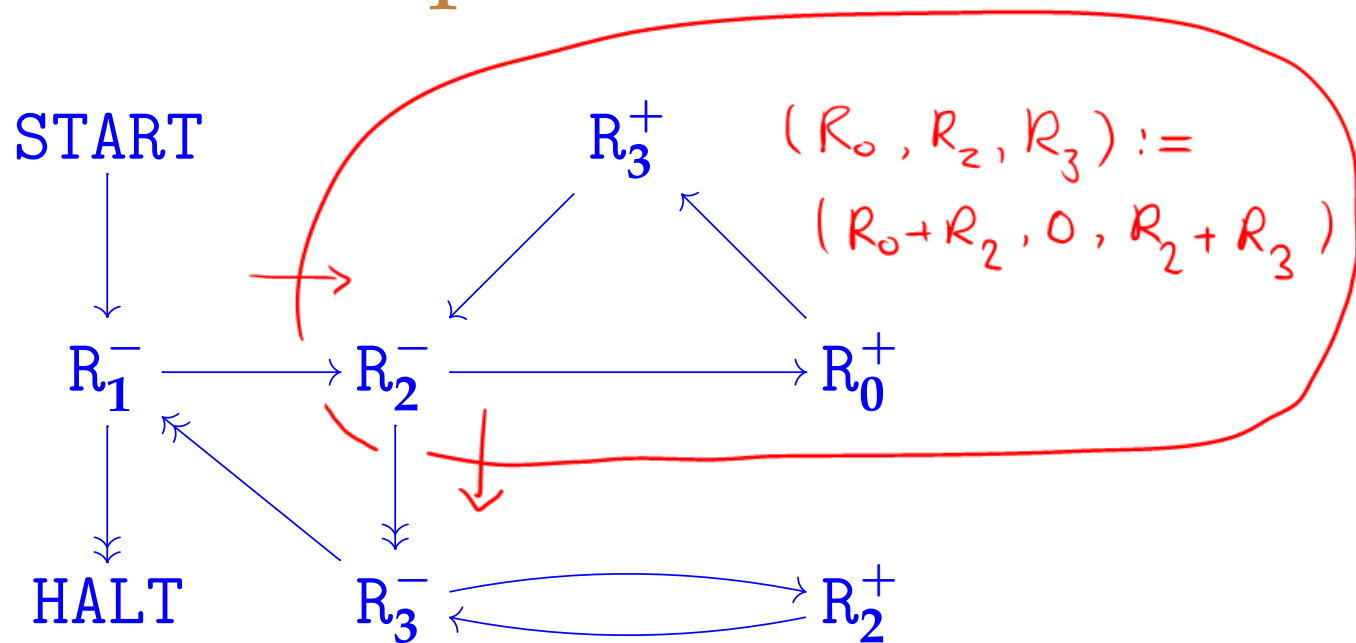


Claim: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(4, x + y, 0, 0)$. So $f(x, y) \triangleq x + y$ is computable.

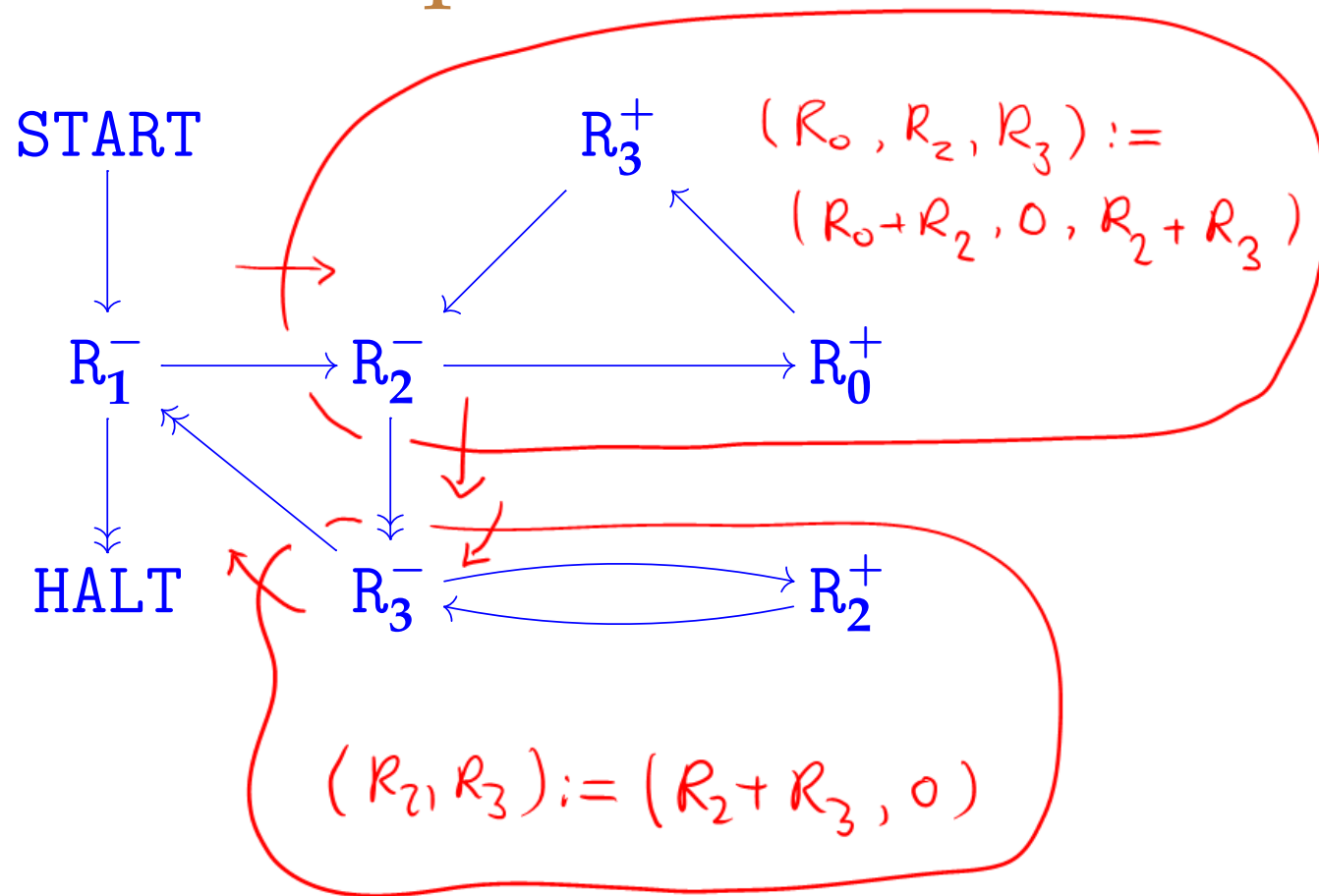
Multiplication $f(x, y) \triangleq xy$
is computable



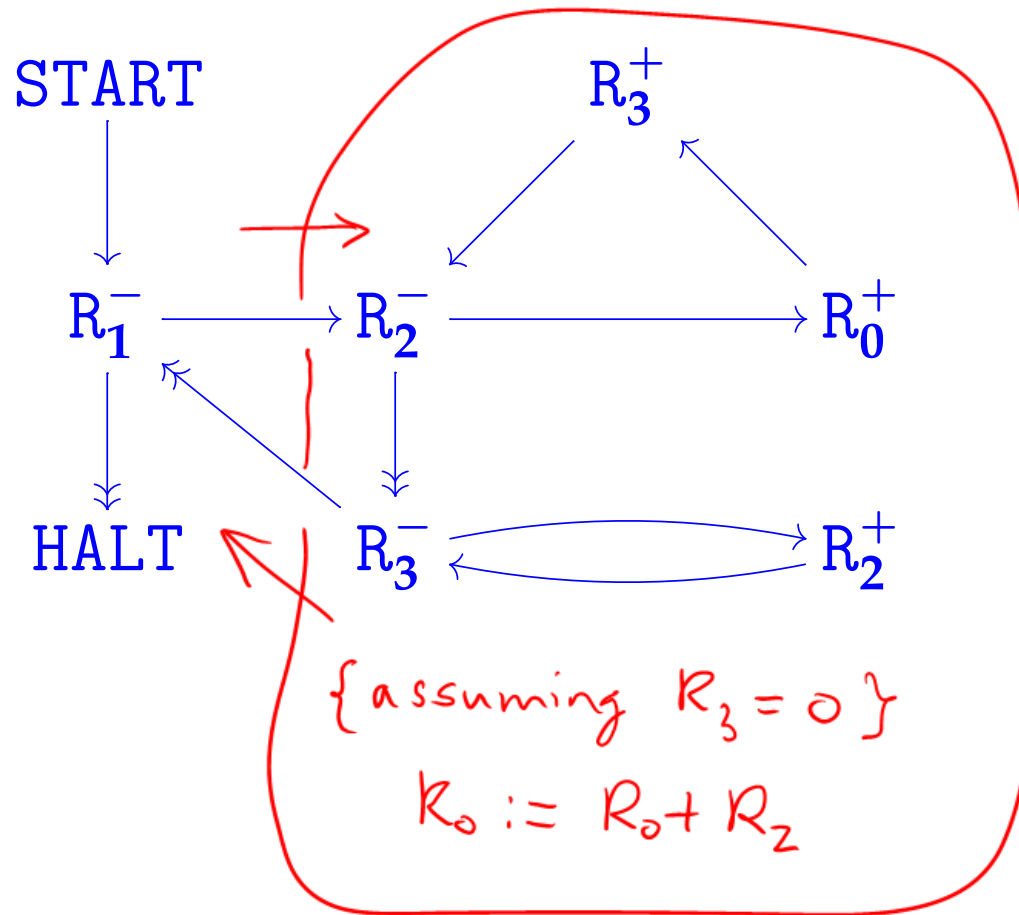
Multiplication $f(x, y) \triangleq xy$ is computable



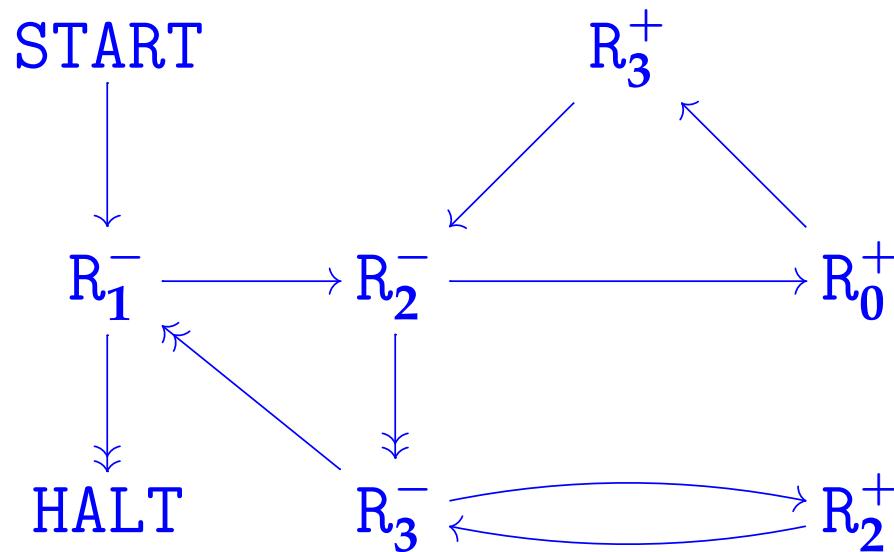
Multiplication $f(x, y) \triangleq xy$ is computable



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

Further examples

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

integer division:

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

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

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

logarithm base 2:

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

W.l.o.g. can use RMs with only one HALT

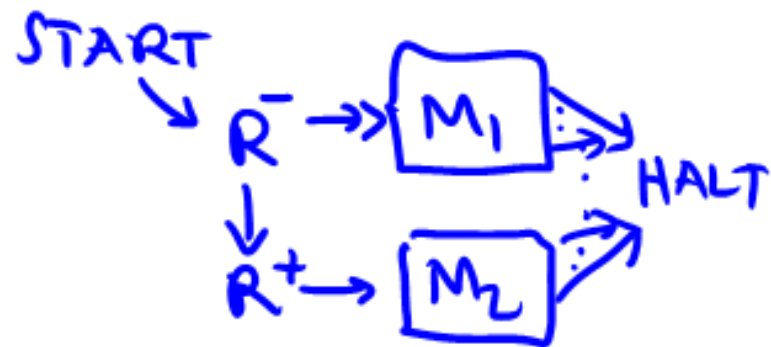


N.B.
interference

Sequential composition $M_1 ; M_2$



IF $R = 0$ THEN M_1 ELSE M_2



WHILE $R \neq 0$ DO M

