Theory of Computation 9 Models of Computation

Frank Stephan

Department of Computer Science

Department of Mathematics

National University of Singapore

fstephan@comp.nus.edu.sg

Repetition 1

Pushdown Automaton $(\mathbf{Q}, \mathbf{\Sigma}, \mathbf{N}, \delta, \mathbf{s}, \mathbf{S}, \mathbf{F})$
$\mathbf{Q}$ are states with start state $\mathbf{s}$ and accepting states $\mathbf{F}$;
$\mathbf{N}$ are stack symbols with start symbol $\mathbf{S}$;
$\mathbf{\Sigma}$ is terminal alphabet;
δ gives choices what to do in cycle; δ maps $(\mathbf{state}, \mathbf{current\ input}, \mathbf{top\ stack\ symbol})$ to choices for $(\mathbf{new\ state}, \mathbf{new\ top\ of\ stack})$ where the input is from $\{\varepsilon\} \cup \mathbf{\Sigma}$ and the new top of stack is from $\mathbf{N}^*$.

In each cycle, the pushdown automaton follows an option of δ what it can do.

A run is successful iff all input gets processed and an accepting state gets reached (acceptance by state); acceptance by empty stack requires in addition that the stack is empty.

Repetition 2

Theorem

The following are equivalent for a language $\mathbf{L}$:

(a) $\mathbf{L}$ is context-free;

(b) $\mathbf{L}$ is recognised by a pushdown automaton accepting by state;

(c) $\mathbf{L}$ is recognised by a pushdown automaton accepting by empty stack;

(d) $\mathbf{L}$ is recognised by a pushdown automaton accepting by empty stack and having only one state which is accepting;

(e) $\mathbf{L}$ is recognised by a pushdown automaton accepting by state which reads in every cycle exactly one input symbol.

Items (b) – (d): Translate CNF into PDA;
Item (e): Translate GNF into PDA.

Repetition 3

Example 9.17: Deterministic PDA

- $\mathbf{Q} = \{\mathbf{s}\}$; $\mathbf{F} = \{\mathbf{s}\}$; start state $\mathbf{s}$;
- $\mathbf{N} = \{\mathbf{S}\}$; start symbol $\mathbf{S}$;
- $\mathbf{\Sigma} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$;
- $\delta(\mathbf{s}, \mathbf{0}, \mathbf{S}) = \{(\mathbf{s}, \varepsilon)\}$;
  $\delta(\mathbf{s}, \mathbf{1}, \mathbf{S}) = \{(\mathbf{s}, \mathbf{S})\}$;
  $\delta(\mathbf{s}, \mathbf{2}, \mathbf{S}) = \{(\mathbf{s}, \mathbf{SS})\}$;
  $\delta(\mathbf{s}, \mathbf{3}, \mathbf{S}) = \{(\mathbf{s}, \mathbf{SSS})\}$;
  $\delta(\mathbf{s}, \varepsilon, \mathbf{S}) = \emptyset$;
- Acceptance mode is by empty stack.

The PDA recognises $\{\mathbf{w} : \mathbf{digitsum}(\mathbf{w}) < |\mathbf{w}|$ and all proper prefixes $\mathbf{v}$ of $\mathbf{w}$ satisfy $\mathbf{digitsum}(\mathbf{v}) \geq |\mathbf{v}|\}$ deterministically.

Repetition 4

Definition 8.19

A deterministic pushdown automaton is given as $(\mathbf{Q}, \mathbf{\Sigma}, \mathbf{N}, \delta, \mathbf{s}, \mathbf{S}, \mathbf{F})$ and has the acceptance mode by state with the additional constraint that for every $\mathbf{A} \in \mathbf{N}$ and every $\mathbf{a} \in \mathbf{\Sigma}$ and every $\mathbf{q} \in \mathbf{Q}$, only one of the sets $\delta(\mathbf{q}, \varepsilon, \mathbf{A}), \delta(\mathbf{q}, \mathbf{a}, \mathbf{A})$ and if non-empty, the set contains one pair $(\mathbf{p}, \mathbf{w})$. The languages recognised by a deterministic pushdown automaton are called deterministic context-free languages.

Theorem

If $\mathbf{L}$ is deterministic context-free and $\mathbf{H}$ is regular then $\mathbf{H} - \mathbf{L}$, $\mathbf{L} - \mathbf{H}$, $\mathbf{L} \cap \mathbf{H}$ and $\mathbf{L} \cup \mathbf{H}$ are also deterministic context-free. The intersection or union of two deterministic context-free languages does not need to be deterministic context-free.

Turing Machine

Example 9.2

The following Turing machine maps each binary number $\mathbf{x}$ to binary $\mathbf{x+1}$. States $\mathbf{s}, \mathbf{t}, \mathbf{u}$, start $\mathbf{s}$, halting $\mathbf{u}$, tape symbols $\sqcup, \mathbf{0}, \mathbf{1}$, input/output symbols $\mathbf{0}, \mathbf{1}$.

state	symbol	new state	new symbol	movement
s	0	s	0	right
s	1	s	1	right
s	$\sqcup$	t	$\sqcup$	left
t	1	t	0	left
t	0	u	1	left
t	$\sqcup$	u	1	left

Input $\ldots \sqcup \sqcup \mathbf{101} \sqcup \sqcup \ldots$, Output $\ldots \sqcup \sqcup \mathbf{110} \sqcup \sqcup \ldots$;
TM $(\{\mathbf{s}, \mathbf{t}, \mathbf{u}\}, \{\mathbf{0}, \mathbf{1}, \sqcup\}, \sqcup, \{\mathbf{0}, \mathbf{1}\}, \delta, \mathbf{s}, \{\mathbf{u}\})$.
Head starts under first input symbol.

Working of Turing Machine

In each cycle, Turing machine reads symbol under head, updates state and symbol according to table and moves the head according to table left or right. If the new state is halting then the Turing machine stops to work and the tape content is the output (provided that it uses only input/output symbols between the blanks $\sqcup$). The start position is the first input symbol (or anywhere for empty input word).

Exercise 9.3
Construct a Turing machine to compute $\mathbf{x} \mapsto \mathbf{3x}$. Input and output are binary numbers.

Exercise 9.4
Construct a Turing machine to compute $\mathbf{x} \mapsto \mathbf{x} + \mathbf{5}$. Input and output are binary numbers.

Words versus Numbers

One can represent numbers by binary, decimal or just follow the words in length-lexicogrphaphical order.

decimal	binary	bin words	ternary	ter words
0	0	ε	0	ε
1	1	0	1	0
2	10	1	2	1
3	11	00	10	2
4	100	01	11	00
5	101	10	12	01
6	110	11	20	02
7	111	000	21	10

One can translate the inputs accross various representations and look at functions as from $\mathbb{N}$ to $\mathbb{N}$.

Register Machines

Programs with registers $\mathbf{R_1, R_2, \ldots, R_n}$ as variables which can take all values from $\mathbb{N}$. Permitted operations:

- $\mathbf{R_i = c}$ for a number $\mathbf{c}$;
- $\mathbf{R_i = R_j + c}$ for a number $\mathbf{c}$;
- $\mathbf{R_i = R_j + R_k}$;
- $\mathbf{R_i = R_j - c}$ for a number $\mathbf{c}$;
- $\mathbf{R_i = c - R_j}$ for a number $\mathbf{c}$;
- $\mathbf{R_i = R_j - R_k}$;
- If $\mathbf{R_i = c}$ Then Goto Line $\mathbf{k}$; (also with other comparisons)
- Goto Line $\mathbf{k}$;
- Return$\mathbf{(R_i)}$; finish the computation with content of Register $\mathbf{R_i}$.

Here subtraction does not give negative values, $\mathbf{3 - 5 = 0}$.

Multiplication

Multiplication can be done naively by repeated addition.

Line 1: Function Mult$(\mathbf{R_1}, \mathbf{R_2})$;
Line 2: $\mathbf{R_3} = \mathbf{0}$;
Line 3: $\mathbf{R_4} = \mathbf{0}$;
Line 4: If $\mathbf{R_3} = \mathbf{R_1}$ Then Goto Line 8;
Line 5: $\mathbf{R_4} = \mathbf{R_4} + \mathbf{R_2}$;
Line 6: $\mathbf{R_3} = \mathbf{R_3} + \mathbf{1}$;
Line 7: Goto Line 4;
Line 8: Return$(\mathbf{R_4})$.

Remainder

The remainder is computed by adding up in steps of $\mathbf{R_2}$ until one reaches the target value and then one takes the difference to the multiple of $\mathbf{R_2}$.

Line 1: Function Remainder$(\mathbf{R_1}, \mathbf{R_2})$;
Line 2: $\mathbf{R_3 = 0}$;
Line 3: $\mathbf{R_4 = 0}$;
Line 4: $\mathbf{R_5 = R_4 + R_2}$;
Line 5: If $\mathbf{R_1 < R_5}$ Then Goto Line 8;
Line 6: $\mathbf{R_4 = R_5}$;
Line 7: Goto Line 4;
Line 8: $\mathbf{R_3 = R_1 - R_4}$;
Line 9: Return$(\mathbf{R_3})$.

Division

This algorithm is very similar to the one for the remainder. One has only keep track on how often one adds up.

Line 1: Function Divide$(\mathbf{R_1}, \mathbf{R_2})$;
Line 2: $\mathbf{R_3} = \mathbf{0}$;
Line 3: $\mathbf{R_4} = \mathbf{0}$;
Line 4: $\mathbf{R_5} = \mathbf{R_4} + \mathbf{R_2}$;
Line 5: If $\mathbf{R_1} < \mathbf{R_5}$ Then Goto Line 9;
Line 6: $\mathbf{R_3} = \mathbf{R_3} + \mathbf{1}$;
Line 7: $\mathbf{R_4} = \mathbf{R_5}$;
Line 8: Goto Line 4;
Line 9: Return$(\mathbf{R_3})$.

Exercises

Write register machine programs following the basic form given above; so adding and subtracting is permitted.

Exercise 9.7

Write a program $\mathbf{P}$ which computes for input $\mathbf{x}$ the value $\mathbf{y} = \mathbf{1} + \mathbf{2} + \mathbf{3} + \ldots + \mathbf{x}$.

Exercise 9.8

Write a program $\mathbf{Q}$ which computes for input $\mathbf{x}$ the value $\mathbf{y} = \mathbf{P}(\mathbf{1}) + \mathbf{P}(\mathbf{2}) + \mathbf{P}(\mathbf{3}) + \ldots + \mathbf{P}(\mathbf{x})$ for the program $\mathbf{P}$ from the previous exercise.

Exercise 9.9

Write a program $\mathbf{O}$ which computes for input $\mathbf{x}$ the factorial $\mathbf{y} = \mathbf{1} \cdot \mathbf{2} \cdot \mathbf{3} \cdot \ldots \cdot \mathbf{x}$. Here the factorial of $\mathbf{0}$ is $\mathbf{1}$.

Subprograms (Macros)

Register machines come without a management for local variables. When writing subprograms, they behave more like macros: One replaces the calling text with a code of what has to be executed at all places inside the program where the subprogram is called. Value passing into the function and returning back is implemented; registers inside the called function are renumbered in case of clashes.

Line 1: Function Power$(\mathbf{R_5}, \mathbf{R_6})$;
Line 2: $\mathbf{R_7} = \mathbf{0}$;
Line 3: $\mathbf{R_8} = \mathbf{1}$;
Line 4: If $\mathbf{R_6} = \mathbf{R_7}$ Then Goto Line 8;
Line 5: $\mathbf{R_8} = \text{Mult}(\mathbf{R_8}, \mathbf{R_5})$;
Line 6: $\mathbf{R_7} = \mathbf{R_7} + \mathbf{1}$;
Line 7: Goto Line 4;
Line 8: Return$(\mathbf{R_8})$.

Translated Program

Line 1: Function Power$(\mathbf{R_5}, \mathbf{R_6})$;
Line 2: $\mathbf{R_7} = \mathbf{0}$;
Line 3: $\mathbf{R_8} = \mathbf{1}$;
Line 4: If $\mathbf{R_6} = \mathbf{R_7}$ Then Return$(\mathbf{R_8})$;
Line 5: $\mathbf{R_1} = \mathbf{R_5}$; // Initialising the Variables used
Line 6: $\mathbf{R_2} = \mathbf{R_8}$; // in the subfunction
Line 7: $\mathbf{R_3} = \mathbf{0}$; // Subfunction starts
Line 8: $\mathbf{R_4} = \mathbf{0}$;
Line 9: If $\mathbf{R_3} = \mathbf{R_1}$ Then Goto Line 13;
Line 10: $\mathbf{R_4} = \mathbf{R_4} + \mathbf{R_2}$;
Line 11: $\mathbf{R_3} = \mathbf{R_3} + \mathbf{1}$;
Line 12: Goto Line 9;
Line 13: $\mathbf{R_8} = \mathbf{R_4}$; // Passing value back, subfunction ends
Line 14: $\mathbf{R_7} = \mathbf{R_7} + \mathbf{1}$;
Line 15: Goto Line 4.

Simulating Turing Machines

Register machines can simulate Turing machines.

The main challenge is to read and write the tape and to read the Turing table.

For simplicity, only a one-side infinite tape is implemented. This can be implemented as a natural number. If there are 10 tape symbols and 0 is the zero then one can say that the digit relating to 10^n could be the tape symbol number n on the tape. So 210 would represent the tape $0120000\ldots$ in this model.

This can be done for any base number including 10. One has to implement how to Read and Write the tape.

Furthermore, one has to implement the main simulation loop.

Reading the Tape / Turing Table

$\mathbf{R_1 = |\Gamma|}$, $\mathbf{R_2}$ is the tape, $\mathbf{R_3}$ is the position

Line 1: Function Read$(\mathbf{R_1, R_2, R_3})$;
Line 2: $\mathbf{R_4} =$ Power$(\mathbf{R_1, R_3})$;
Line 3: $\mathbf{R_5} =$ Divide$(\mathbf{R_2, R_4})$;
Line 4: $\mathbf{R_6} =$ Remainder$(\mathbf{R_5, R_1})$;
Line 5: Return$(\mathbf{R_6})$.

$\mathbf{R_6}$ is the symbol read. A coding of this type will also be used for the Turing table. In general, $\mathbf{R_1}$ is the basis and $\mathbf{R_6}$ the digit number $\mathbf{R_3}$ in the number $\mathbf{R_2}$.

If $\mathbf{R_1 = 10}$, $\mathbf{R_2 = 23842}$ and $\mathbf{R_3 = 2}$ then $\mathbf{R_4 = 100}$, $\mathbf{R_5 = 238}$ and $\mathbf{R_6 = 8}$.

Writing the Tape

$\mathbf{R_1} = |\mathbf{\Gamma}|$, $\mathbf{R_2}$ is the old tape, $\mathbf{R_3}$ is the position, $\mathbf{R_4}$ is the symbol to be written, $\mathbf{R_9}$ is the new tape content.

Line 1: Function Write$(\mathbf{R_1}, \mathbf{R_2}, \mathbf{R_3}, \mathbf{R_4})$; // $\mathbf{10, 23842, 2, 7}$
Line 2: $\mathbf{R_5} =$ Power$(\mathbf{R_1}, \mathbf{R_3})$; // $\mathbf{R_5 = 100}$
Line 3: $\mathbf{R_6} =$ Divide$(\mathbf{R_2}, \mathbf{R_5})$; // $\mathbf{R_6 = 238}$
Line 4: $\mathbf{R_7} =$ Remainder$(\mathbf{R_6}, \mathbf{R_1})$; // $\mathbf{R_7 = 8}$
Line 5: $\mathbf{R_6 = R_6 + R_4 - R_7}$; // $\mathbf{R_6 = 238 + 7 - 8 = 237}$
Line 6: $\mathbf{R_8} =$ Mult$(\mathbf{R_6}, \mathbf{R_5})$; // $\mathbf{R_8 = 23700}$
Line 7: $\mathbf{R_9} =$ Remainder$(\mathbf{R_2}, \mathbf{R_5})$; // $\mathbf{R_9 = 42}$
Line 8: $\mathbf{R_9 = R_9 + R_8}$; // $\mathbf{R_9 = 23742}$
Line 9: Return$(\mathbf{R_9})$. // Return New Tape

The tape is split into two parts and then the last digit of the first part is adjusted and afterwards the two parts are reassembled.

Simulating the Turing Machine

Line 1: Function Simulate$(\mathbf{R_1}, \mathbf{R_2}, \mathbf{R_3}, \mathbf{R_4})$;
Line 2: $\mathbf{R_5} = \mathbf{0}$;
Line 3: $\mathbf{R_7} = \mathbf{0}$;
Line 4: $\mathbf{R_9} = \text{Mult}(\text{Mult}(\mathbf{2}, \mathbf{R_2}), \mathbf{R_1})$;
Line 5: $\mathbf{R_6} = \text{Read}(\mathbf{R_1}, \mathbf{R_4}, \mathbf{R_5})$;
Line 6: $\mathbf{R_8} = \text{Read}(\mathbf{R_9}, \mathbf{R_3}, \text{Mult}(\mathbf{R_1}, \mathbf{R_7}) + \mathbf{R_6})$;
Line 7: $\mathbf{R_{10}} = \text{Divide}(\mathbf{R_8}, \text{Mult}(\mathbf{R_2}, \mathbf{2}))$;
Line 8: $\mathbf{R_4} = \text{Write}(\mathbf{R_1}, \mathbf{R_4}, \mathbf{R_5}, \mathbf{R_{10}})$;
Line 9: $\mathbf{R_7} = \text{Remainder}(\text{Divide}(\mathbf{R_8}, \mathbf{2}), \mathbf{R_2})$;
Line 10: If $\mathbf{R_7} = \mathbf{1}$ Then Goto Line 13;
Line 11: $\mathbf{R_5} = \mathbf{R_5} + \text{Mult}(\mathbf{2}, \text{Remainder}(\mathbf{R_8}, \mathbf{2})) - \mathbf{1}$;
Line 12: Goto Line 5;
Line 13: Return$(\mathbf{R_4})$.

Here $\mathbf{R_1} = |\mathbf{\Gamma}|$, $\mathbf{R_2} = |\mathbf{Q}|$, $\mathbf{R_3}$ is Turing Table, $\mathbf{R_4}$ is Turing Tape, $\mathbf{R_5}$ is position, $\mathbf{R_7}$ is state ($\mathbf{0}$ is start, $\mathbf{1}$ is halting).

Turing Machine Simulation

Theorem 9.12
Every Turing machine can be simulated by a register machine and there is even one single register machine which simulates for input (e, x) the Turing machine described by e; if this simulation ends with an output y in the desired form then the register machine produces this output; if the Turing machine runs forever, so does the simulating register machine.

Exercise 9.13
Explain how one has to change the simulation of the Turing machine in order to have a tape which is in both directions infinite.

Theorem 9.14 [Alan Turing 1936]
There is a single Turing machine $TM(e, x)$ which simulates the behaviour of the e-th Turing machine on input x.

Primitive Recursive Functions

Constant Function: The function producing the constant $\mathbf{0}$ without any inputs is primitive recursive.

Successor Function: The function $\mathbf{x \mapsto x+1}$ is primitive recursive.

Projection Function: Each function of the form $\mathbf{x_1, \ldots, x_n \mapsto x_m}$ for some $\mathbf{m \in \{1, \ldots, n\}}$ is primitive recursive.

Composition: If $\mathbf{f : \mathbb{N}^n \to \mathbb{N}}$ and $\mathbf{g_1, \ldots, g_n : \mathbb{N}^m \to \mathbb{N}}$ are primitive recursive, so is $\mathbf{x_1, \ldots, x_m \mapsto f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))}$.

Recursion: If $\mathbf{f : \mathbb{N}^n \to \mathbb{N}}$ and $\mathbf{g : \mathbb{N}^{n+2} \to \mathbb{N}}$ are primitive recursive then there is also a primitive recursive function $\mathbf{h}$ with $\mathbf{h(0, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)}$ and $\mathbf{h(y+1, x_1, \ldots, x_n) = g(y, h(y, x_1, \ldots, x_n), x_1, \ldots, x_n)}$.

Example 9.17

$\mathbf{pred(x) = x - 1}$ is primitive recursive via $\mathbf{pred(0) = 0}$ and $\mathbf{pred(y+1) = g(y, pred(y)) = y}$.

$\mathbf{h(x, y) = x - y}$ is primitive recursive via $\mathbf{h(x, 0) = x}$ and $\mathbf{h(x, y+1) = pred(h(x, y))}$. Recursion over wrong parameter, so define $\mathbf{\tilde{h}(y, x) = h(x, y)}$ and prove that $\mathbf{\tilde{h}}$ is primitive recursive and then $\mathbf{h(x, y)}$ is concatenation of $\mathbf{\tilde{h}}$ with $\mathbf{(second, first)}$ where $\mathbf{second(x, y) = y}$ and $\mathbf{first(x, y) = x}$.

$\mathbf{equal(x, y) = 1 - (x - y) - (y - x)}$ is $\mathbf{1}$ when $\mathbf{x = y}$ and $\mathbf{0}$ when $\mathbf{x \neq y}$.

$\mathbf{h(x, y) = x + y}$ is primitive recursive by $\mathbf{h(0, y) = 0 + y = y}$ and $\mathbf{h(x+1, y) = (x+1) + y = (x+y) + 1 = h(x, y) + 1}$.

Exercises

Exercise 9.18

Prove that every linear function $\mathbf{h(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \ldots + a_nx_n + b}$ is primitive recursive, where the parameters $\mathbf{a_1, a_2, \ldots, a_n, b}$ are in $\mathbb{N}$.

Exercise 9.19

Prove that the function $\mathbf{h(x) = 1 + 2 + \ldots + x}$ is primitive recursive.

Exercise 9.20

Prove that the multiplication $\mathbf{h(x, y) = x \cdot y}$ is primitive recursive.

Not Primitive Recursive

There is no primitive recursive function $\mathbf{f(e,x)}$ such that for each primitive recursive function $\mathbf{g(x)}$ there is an $\mathbf{e}$ with $\forall \mathbf{x}\, [\mathbf{f(e,x)} = \mathbf{g(x)}]$.

Assume that such an $\mathbf{f}$ exists. Consider

$$\mathbf{g(x) = 1 + f(0,x) + f(1,x) + \ldots + f(x,x)}.$$

The function $\mathbf{g}$ grows faster than $\mathbf{x} \mapsto \mathbf{f(e,x)}$ for any constant $\mathbf{e}$. So there is no universal primitive recursive function.

Further Example, Ackermann Function:

- $\mathbf{f(0,y) = y + 1}$;
- $\mathbf{f(x+1,0) = f(x,1)}$;
- $\mathbf{f(x+1,y+1) = f(x,f(x+1,y))}$.

Partial Recursive Functions

Definition 9.21
If $\mathbf{f}(\mathbf{y}, \mathbf{x_1}, \ldots, \mathbf{x_n})$ is a function then the μ-minimalisation $\mathbf{g}(\mathbf{x_1}, \ldots, \mathbf{x_n}) = \mu\mathbf{y}\,[\mathbf{f}(\mathbf{y}, \mathbf{x_1}, \ldots, \mathbf{x_n})]$ is the first value $\mathbf{y}$ such that $\mathbf{f}(\mathbf{y}, \mathbf{x_1}, \ldots, \mathbf{x_n}) = \mathbf{0}$. The function $\mathbf{g}$ can be partial, since $\mathbf{f}$ might at certain combinations of $\mathbf{x_1}, \ldots, \mathbf{x_n}$ not take the value $\mathbf{0}$ for any $\mathbf{y}$ and then the search for the $\mathbf{y}$ is undefined.

The partial recursive or μ-recursive functions are those which are formed from the base functions by concatenation, primitive recursion and μ-minimalisation. If a partial recursive function is total, it is just called a recursive function.

Theorem 9.22
Every partial recursive function can be computed by a register machine.

Primitive Recursion

By structural induction. All base functions are linear and just be computed by a register machine.

Let $\mathbf{h(y, x_1, x_2)}$ be defined by primitive recursion from $\mathbf{f}$ and $\mathbf{g}$. Let register programs F and G for $\mathbf{f}$ and $\mathbf{g}$ be given. Now $\mathbf{h}$ has the following register program.

Line 1: Function H$(\mathbf{R_1, R_2, R_3})$;
Line 2: $\mathbf{R_4 = 0}$;
Line 3: $\mathbf{R_5 =}$ F$(\mathbf{R_2, R_3})$;
Line 4: If $\mathbf{R_4 = R_1}$ Then Goto Line 8;
Line 5: $\mathbf{R_5 =}$ G$(\mathbf{R_4, R_5, R_2, R_3})$;
Line 6: $\mathbf{R_4 = R_4 + 1}$;
Line 7: Goto Line 4;
Line 8: Return$(\mathbf{R_5})$.

Minimalisation

Let $\mathbf{h} = \mu\mathbf{y}\,[\mathbf{f}(\mathbf{y}, \mathbf{x_1}, \mathbf{x_2}) = \mathbf{0}]$ and let a program F for $\mathbf{f}$ be given. The following program is for $\mathbf{h}(\mathbf{x_1}, \mathbf{x_2})$.

Line 1: Function H$(\mathbf{R_1}, \mathbf{R_2})$;
Line 2: $\mathbf{R_3} = \mathbf{0}$;
Line 3: $\mathbf{R_4} = \mathsf{F}(\mathbf{R_3}, \mathbf{R_1}, \mathbf{R_2})$;
Line 4: If $\mathbf{R_4} = \mathbf{0}$ Then Goto Line 7;
Line 5: $\mathbf{R_3} = \mathbf{R_3} + \mathbf{1}$;
Line 6: Goto Line 3;
Line 7: Return$(\mathbf{R_3})$.

Furthermore, concatenation of functions computed by register machines can also be computed by register machines. Thus all partial recursive functions can be computed by register machines.

Church's Thesis

Theorem 9.23

For a partial function f, the following are equivalent:

- f as a function from strings to strings can be computed by a Turing machine;
- f as a function from natural numbers to natural numbers can be computed by a register machine;
- f as a function from natural numbers to natural numbers is partial recursive.

Church's Thesis

All reasonable models of computation over Σ^* and $\mathbb{N}$ are equivalent and give the same notion as the partial recursive functions.

Complexity

One measures the size $\mathbf{n}$ of the input in the number of its symbols or by $\mathbf{log(x)} = \min\{\mathbf{n} \in \mathbb{N} : \mathbf{x} \leq \mathbf{2^n}\}$.

Theorem 9.25
A function $\mathbf{f}$ is computable by a Turing machine in time $\mathbf{p(n)}$ for some polynomial $\mathbf{p}$ iff $\mathbf{f}$ is computable by a register machine in time $\mathbf{q(n)}$ for some polynomial $\mathbf{q}$.

Theorem 9.26
A function $\mathbf{f}$ is computable by a Turing machine in space $\mathbf{p(n)}$ for some polynomial $\mathbf{p}$ iff $\mathbf{f}$ is computable by a register machine in such a way that all registers take at most the value $\mathbf{2^{q(n)}}$ for some polynomial $\mathbf{q}$.

The notions in Complexity Theory are also relatively invariant against changes of the model of computation; however, one has to interpret the word “reasonable” of Church in a stronger way than in recursion theory.

Example 9.27

The $O(n^2)$ Algorithm.

Line 1: Function Polymult(R_1, R_2);
Line 2: $R_3 = 0$;
Line 3: $R_4 = 0$;
Line 4: If $R_3 = R_1$ Then Goto Line 13;
Line 5: $R_5 = 1$;
Line 6: $R_6 = R_2$;
Line 7: If $R_3 + R_5 > R_1$ Then Goto Line 4;
Line 8: $R_3 = R_3 + R_5$;
Line 9: $R_4 = R_4 + R_6$;
Line 10: $R_5 = R_5 + R_5$;
Line 11: $R_6 = R_6 + R_6$;
Line 12: Goto Line 7;
Line 13: Return(R_4).

Example 9.27

The $\mathbf{O(n)}$ Algorithm.

Line 1: Function Binarymult$(\mathbf{R_1, R_2})$;
Line 2: $\mathbf{R_3 = 1}$; $\mathbf{R_4 = 1}$; $\mathbf{R_5 = 0}$; $\mathbf{R_6 = R_2}$;
Line 3: If $\mathbf{R_3 > R_6}$ Then Goto Line 5;
Line 4: $\mathbf{R_3 = R_3 + R_3}$; Goto Line 3;
Line 5: $\mathbf{R_6 = R_6 + R_6}$; $\mathbf{R_5 = R_5 + R_5}$; $\mathbf{R_4 = R_4 + R_4}$;
Line 6: If $\mathbf{R_6 < R_3}$ Then Goto Line 8;
Line 7: $\mathbf{R_5 = R_5 + R_1}$; $\mathbf{R_6 = R_6 - R_3}$;
Line 8: If $\mathbf{R_4 < R_3}$ Then Goto Line 5;
Line 9: Return$(\mathbf{R_5})$.

The first $\mathbf{O(n)}$ algorithm was given by Floyd and Knuth in 1990.

Exercises

Exercise 9.28
Write a register program which computes the remainder in polynomial time.

Exercise 9.29
Write a register program which divides in polynomial time.

Exercise 9.30
Let an extended register machine have the additional command which permits to multiply two registers in one step. Show that an extended register machine can compute a function in polynomial time which cannot be computed in polynomial time by a normal register machine.