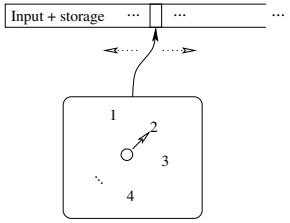
CS 467/567: Some models of (sequential) computation

Stefan D. Bruda

Winter 2020

Turing machines



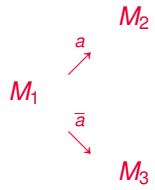
- Finite state control (program) + storage
- An infinite tape used as storage and also input
  - The head scans the tape, can read the current cell, can overwrite the current cell, or can move left or right
- Formally, $M = (K, \Sigma, \delta, s, h)$
- Finite set of states K, tape alphabet Σ
- $\triangleright, \# \in \Sigma$, $L, R \notin \Sigma$
- Special halt state $h \notin K$
- $\delta : K \times \Sigma \to (K \cup \{h\}) \times (\Sigma \cup \{L, R\})$ such that for all $q \in K$:
  - $\delta(q, \triangleright) = (p, b)$ implies $b = R$
  - $\delta(q, a) = (p, b)$ implies $b \neq \triangleright$
- Configuration: $K \times \Sigma^* \times (\Sigma^*(\Sigma \setminus \{\#\}) \cup \{\varepsilon\})$
  - Configuration (q, wa, w') commonly written as $(q, w\underline{a}w')$

Turing machines (cont'd)

- Yields in one step: $(q_1, w_1\underline{a_1}u_1) \vdash_M (q_2, w_2\underline{a_2}u_2)$ iff $\delta(q_1, a_1) = (q_2, b)$ for some $b \in \Sigma \cup \{L, R\}$ and either
  - $b \in \Sigma$, $w_1 = w_2$, $u_1 = u_2$, $a_2 = b$,
  - $b = L$, $w_1 = w_2 a_2$,
    - $u_2 = a_1 u_1$ if $a_1 \neq \#$ or $u_1 \neq \varepsilon$
    - $u_2 = \varepsilon$ if $a_1 = \#$ or $u_1 = \varepsilon$,
  - $b = R$, $w_2 = w_1 a_1$,
    - $u_1 = a_2 u_2$ if $a_2 \neq \#$
    - $u_1 = u_2 = \varepsilon$ if $a_2 = \#$
- Yields: $\vdash_M^*$, the reflexive and transitive closure of $\vdash_M$
- Yields in n steps: $C_0 \vdash_M^n C_n$ iff $C_0 \vdash_M C_1 \vdash_M \cdots \vdash_M C_{n-1} \vdash_M C_n$
- M computes $f : \Sigma^* \to \Sigma^*$ iff $(s, \#w\underline{\#}) \vdash_M^* (h, \#f(w)\underline{\#})$
- Computation of a Turing machine = sequence of configurations

Compositional notation for Turing machines

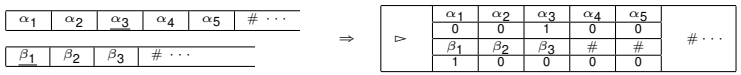
- Basic machines: a : $\forall b \in \Sigma : \delta(s, b) = (h, a)$
  L : $\forall b \in \Sigma : \delta(s, b) = (h, L)$ R : $\forall b \in \Sigma : \delta(s, b) = (h, R)$
- Combining machines:



  M_1 halts and then either M_2 or M_3 start, depending on whether a is read (or not) by the head when M_1 halts
- Supplementary, handy notations:
  - $M \to N$ or just MN for M followed immediately by N
  - $M \xrightarrow{x=\alpha} M_x$
  - R_x for $R \circlearrowleft_{\bar{x}}$; $R_{\bar{x}}$ for $R \circlearrowleft_x$; similar for L_x, $L_{\bar{x}}$

Extensions of Turing machines

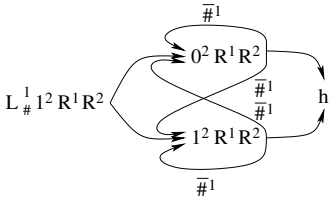
- Multiple tapes $\delta : K \times \Sigma^k \to (K \cup \{h\}) \times (\Sigma \cup \{L, R\})^k$
  - Can be readily simulated by a classical Turing machine
  - We put all the tapes as "tracks" on a single tape



  - On every move we must scan the whole non-blank tape ⇒ quadratic slowdown
  - Compositional notation: M^k = machine M operating on tape k
- Nondeterministic Turing machine: Same definition except that we have a transition relation $\Delta \subseteq (K \backslash H) \times \Sigma \times K \times (\Sigma \cup \{L, R\})$
  - Configuration, $\vdash_M$, etc. identical, but now a configuration can yield in one step more than one configuration
  - M computes $f : \Sigma^* \to \Sigma^*$ iff
    1. $\forall\, w \in \Sigma : (s, \#w\underline{\#}) \vdash_M^* (h, \#f(w)\underline{\#})$
    2. For some finite N (depending on M and w) there exists no configuration C such that $(s, \#w\underline{\#}) \vdash_M^N C$ (M always halts)
  - Computation of a nondeterministic Turing machine = tree or graph of configurations

Example of nondeterministic computation: Composite numbers

- Problem: given an input $x \in \{0\} \cup \{1\}\{0,1\}^*$ are there $p, q \in \{1\}\{0,1\}\{0,1\}^*$ such that $x = p \times q$?
- Solved by a simple and fast nondeterministic Turing machine with two tapes:
  1. First tape contains input x (a binary number) and does not change
  2. Guess nondeterministically on the second tape a number $p \leqslant x$ and again a number $q \leqslant x$



  3. Multiply p and q, compare the result with x, answer "true" if they are equal and "false" otherwise

Determinism versus nondeterminism

Theorem

If a nondeterministic Turing machine M computes f then there exists a deterministic Turing machine M' that does the same thing.

- Crux: Let $C \vdash_M C_1$, $C \vdash_M C_2$, ..., $C \vdash_M C_n$; is there an upper bound for n?
  - Sure: $n \leqslant r = |K| \times (|\Sigma| + 2)$
- We first construct a machine M_d that receives the input and (on a different tape) a path description $i_1 i_2 \ldots i_k$ for some k, with $1 \leqslant i_j \leqslant r$
  - M_d carries on k steps of the computation of M along the path given; it is deterministic
- Then M' will be a 3-tape machine; tape 1 contains input w and remains unchanged, tapes 2 and 3 are initially empty
  1. M' then generate the next path description on tape 3 in lexicographic order
  2. Then M' copies w on tape 2 and launches M_d
  3. If M_d is successful, then M' reports success; otherwise repeat from Step 1
- Exponential slowdown

The Random Access Machine

- The Random Access Machine (RAM) consists of an unbounded set of registers R_i, $i \geqslant 0$, one register PC, and a control unit
  - The size (i.e. the number of bits) of a register is $\log n$ for an input of size n
- The control unit executes a program consisting of a sequence of numbered statements
  - In each computation step the RAM executes one statement of the program; the execution start with the first statement
  - The register PC specifies the number of the statement that is to be executed
  - The program halts when the program counter takes an invalid value (i.e. there is no statement with the specified number in the program)
- To "run" a RAM we need to
  - Specify a program
  - Define an initial values for the registers R_i, $0 \leqslant i < n$ (input)
  - The output is the content of the registers upon halting

RAM STATEMENTS

Statement	Effect on registers	Program counter
$R_i \leftarrow R_j$	$R_i := R_j$	$PC := PC + 1$
$R_i \leftarrow R[R_j]$	$R_i := R_{R_j}$	$PC := PC + 1$
$R[R_j] \leftarrow R_i$	$R_{R_j} := R_i$	$PC := PC + 1$
$R_i \leftarrow k$	$R_i := k$	$PC := PC + 1$
$R_i \leftarrow R_j + R_k$	$R_i := R_j + R_k$	$PC := PC + 1$
$R_i \leftarrow R_j - R_k$	$R_i := \max\{0, R_j - R_k\}$	$PC := PC + 1$
GOTO m		$PC := m$
IF $R_i = 0$ GOTO m		$PC := \begin{cases} m & \text{if } R_i = 0 \\ PC + 1 & \text{otherwise} \end{cases}$
IF $R_i > 0$ GOTO m		$PC := \begin{cases} m & \text{if} R_i > 0 \\ PC + 1 & \text{otherwise} \end{cases}$

Customary extensions:

- Named registers (or variables), even arrays and structures
- All the usual arithmetic operations (multiplication, division, shift, etc.)
- Structured control statements (if-then-else statements, while loops, etc.)

DESCRIBING ALGORITHMS

- The Turing machine and the RAM are equivalent to each other within polynomial speedup/slowdown
  - These plus a lot of other models of computation (the Church-Turing thesis)
  - So it makes a lot of sense to use the RAM to express and analyze algorithms
- The RAM programming language: pseudocode
  - The golden standard for describing algorithms
- Often an English description of the algorithm will do just as well (or even better!)
  - Often more concise and more readable than pseudocode
  - One just needs to make sure that the English description does describe an algorithm (i.e., can be converted into pseudocode)