Foundations of Computer Science
Logic, models, and computations

Chapter: A few other models of computation

Course INF412

of l'Ecole Polytechnique

Olivier Bournez

bournez@lix.polytechnique.fr

Version of July 16, 2023

A few other models of computation

1 RAM

The Turing machine model may seem extremely rudimentary. However, it is in fact extremely powerful, and it is able to capture the notion of *computability* in computer science.

The objective of this chapter is to argue the following Church-Turing thesis: Every computation that can be programmed by a digital computational device, such as a modern computer, can be simulated by a Turing machine. In order to make this plausible, we will introduce first a model very close (actually the closest that I know) to the way that today's computers work: The *RAM model*.

1.1 RAM model

The model of the *RAM (Random Access Machine)* is a computational model that is close to today's machine languages, and the way the processors of today work.

A RAM has registers that each contain a natural number (null when initialized, i.e. if not yet used). The machine is assumed to have infinitely many registers, indexed by integers. The authorized instructions depend of the processor that one wants to model[1] but in general they include the following:

1. copying the content of a register into another;
2. doing indirect addressing: Get/Write the content of a register whose index is given by the value of some other register;
3. doing some very basic elementary operation on some particular register such as for example adding 1, subtract 1 or test equality to 0 of a register;
4. doing some other operation on a or several register(s), for example addition, subtraction, multiplication, division, binary shifts or bitwise binary operations.

[1] And actually, also of the reference book that one takes to formally describe the model.

In the following, we will limit the discussion to the *SRAM (Successor Random Access Machine)* model that only has instructions of type 1., 2. and 3. We will see later that this does not really change things, as long as each of the allowed operations of type 4. can be simulated by a Turing machine (and this is the case for all operations mentioned above).

1.2 Simulation of a RISC machine by a Turing machine

We are going to show that any (S)RAM can be simulated by a Turing machine.

To help the understanding of the proof, we will reduce the set of instructions of the RAM to a reduced set of instructions (*RISC reduced instruction set*) by using a unique register x_0 as an accumulator.

Definition 1 *A RISC machine is a (S)RAM whose instructions are (only) of the form:*

1. $x_0 \leftarrow 0$;
2. $x_0 \leftarrow x_0 + 1$;
3. $x_0 \leftarrow x_0 \ominus 1$;
4. ***if*** $x_0 = 0$ ***then*** *go* ***to*** *instruction number* j ;
5. $x_0 \leftarrow x_i$;
6. $x_i \leftarrow x_0$;
7. $x_0 \leftarrow x_{x_i}$;
8. $x_{x_0} \leftarrow x_i$.

Clearly, every SRAM program with instructions of type 1., 2. and 3. can be converted to an equivalent RISC program, by replacing every instruction by instructions that do the equivalent operation by systematically using the accumulator x_0 (if needed).

Example 1 *For example: the instruction $x_i \leftarrow x_j$ can be replaced by the two instructions $x_0 \leftarrow x_j$ and then $x_i \leftarrow x_0$.*

We will start with the following simulation:

Theorem 1 *Every RISC machine can be simulated by a Turing machine.*

Proof: We describe how to construct a Turing machine that simulates the RISC machine. The Turing machine has 4 tapes. The first two tapes are encoding the pairs (i, x_i) for x_i non null. The third tape encodes the accumulator x_0 and the fourth tape is used as a scratch tape.

More concretely, for every integer i, denote by $\langle i\rangle$ its binary representation. The first tape encodes a word of the form

$$\ldots\mathbf{BB}\langle i_o\rangle\mathbf{B}\langle i_1\rangle\cdots\mathbf{B}\ldots\langle i_k\rangle\mathbf{BB}\cdots.$$

The second tape encodes a word of the form

$$\ldots\mathbf{BB}\langle x_{i_o}\rangle\mathbf{B}\langle x_{i_1}\rangle\cdots\mathbf{B}\ldots\langle x_{i_k}\rangle\mathbf{BB}\cdots$$

The heads of the first two tapes are on the second **B**. The third tape encodes $\langle x_0\rangle$, the head being at the left. We call this position of the heads the *standard position.*

The simulation is described for three examples. The reader will find it easy to complete the other cases:

1. $x_0 \leftarrow x_0 + 1$: We increment the content of the third tape which by convention contains the binary encoding of x_0. To do so, the head of tape 3 moves right until it reads a **B** symbol. It then moves left once. It then replaces the **1**'s by by **0**'s, while moving to the left as long as as it reads **1**'s. When a **0** or a **B** is found, it is changed into a **1** and the head moves left until it comes back to the standard position.

2. $x_{23} \leftarrow x_0$: The heads scan the tapes 1 and 2 to the right, block by block (we call block a word delimited by two **B**'s), in parallel (i.e if head of tape 1 reads block number i, then this is true for head of tape 2 and conversely), until the head of tape 1 (and also of tape 2) reaches the end of tape 1, or until a block **B10111B** (**10111** is 23 in binary) is found on tape 1.

   If the end of tape 1 is reached, this means that memory position 23 has never been seen previously. One adds it by writing **10111** at the end of tape 1, and one copies the content of tape 3 (the value of x_0) onto tape 2. Afterwards, all heads are moved back to the standard position.

   Otherwise we have found **B10111B** on tape 1. By construction, the head of tape 2 then points at the **B** after $\langle x_{23}\rangle$. In that case, we must modify the part of tape 3 containing $\langle x_{23}\rangle$ which is done in the following way:

   (a) One copies the content at the right of the head of tape 2 onto tape 4.

   (b) One overwrites the content of x_{23} on tape 2 by the the content of tape 3 (the value of x_0).

   (c) One writes **B**, and one copies the content of tape 4 at the right of the head of tape 2, in order to restore the rest of tape 2.

   (d) One returns to the standard position.

3. $x_0 \leftarrow x_{x_{23}}$: Starting from the left of tapes 1 and 2, one scans the tapes 1 and 2 going to the right, block by block, in parallel, until one reaches the end of tape 1, or a block **B10111B** (**10111** is 23 in binary) is read.

   If the end of tape 1 has been reached, one does nothing, since x_{23} values 0 and tape 3 already contains $\langle x_0\rangle$.

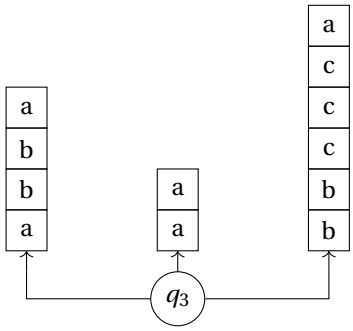

Figure 1: A machine with 3 stacks.

Otherwise, this means that we found **B10111B** on tape 1. One then reads $\langle x_{23} \rangle$ on tape 2, that one copies on tape 4. As above, one scans tapes 1 and 2 in parallel until one finds $\mathbf{B}\langle x_{23} \rangle \mathbf{B}$ or the end of tape 1 is reached. If the end of tape 1 is reached, then one writes **0** on tape 3, since $x_{x_{23}} = x_0$. Otherwise, one copies the block corresponding to tape 2 on tape 3, since the block on tape 2 contains $x_{x_{23}}$, and one returns to the standard position.

□

1.3 Simulation of a RAM by a Turing machine

Let us come back to the fact that we have reduced the set of possible operations of an *SRAM* to instructions of type 1., 2. and 3. In fact, it is easy to see that one can deal with all instructions of type 4., as long as the underlying operation can be computed by a Turing machine: Every operation $x_0 \leftarrow x_0$ ''operation'' x_i can be simulated as above, as soon as *"operation"* corresponds to some computable operation.

2 Rudimentary models

The Turing machine model is extremely rudimentary. It turns out that one can consider models that are even more rudimentary, and that are still able to simulate them.

2.1 Machines with $k \geq 2$ stacks

A k-stack machine, has k stacks $r_1, r_2, \ldots, r_k$. Each of this stacks corresponds to some stack of elements of the finite alphabet Σ. The instructions of the machine permit only to push a symbol on one of the stacks, read the symbol at the top of a stack, or pop the symbol at the top of stack.

If one prefers, one can see each stack r_i of elements of the finite alphabet Σ as a word w_i over alphabet Σ. Pushing (written $push(i,a)$) a symbol $a \in \Sigma$ on this stack

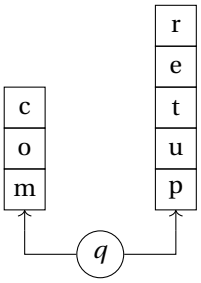

Figure 2: The Turing machine from Example 7.3 seen as a 2-stacks machine.

consists in replacing w_i by aw_i. Reading (written $a \leftarrow top(r,i)$) the symbol at the top of this stack consists in reading the first letter of w_i. Doing a pop on (written $pop(i)$) this stack consists in deleting the first letter of w_i.

Theorem 2 *Every Turing machine can be simulated by a* 2 *stacks machine.*

Proof: According to the formalization of Page 104, a configuration of a Turing machine corresponds to $C = (q,u,v)$, where u and v denote the content respectively on left and on right of the head of tape i. One can see u and v as stacks: See Figure 2. If one reads the formalization Page 104 carefully, one sees that the operations done by the program of a Turing machine to go from configuration C to its successor configuration C' coincide with operations that can be trivially coded by $push$, pop, and top: One can construct a machine with 2 stacks, each stack encoding u or v (the content on the right and on the left of the tape) and that simulate the Turing machine step by step. For example, moving the head to the right, consists in reading the top of the second stack ($a \leftarrow top(2)$), pushing this symbol a on the first stack ($push(1,a)$), and doing a pop on the second stack ($pop(2)$). Moving the head to the left, consists in reading the top of the first stack ($a \leftarrow top(1)$), pushing this symbol a on the second stack ($push(2,a)$), and doing a pop on the first stack $pop(1)$. Changing the symbol in front of the head into symbol a consists in doing a pop on the second stack ($pop(2)$), and then pushing symbol a on second stack ($push(2,a)$). □

2.2 Counter machines

We introduce now a model even more rudimentary: A *counter machine* has a finite number k of counters $r_1, r_2, \cdots, r_k$, which contain natural numbers. The instructions of a counter machine allow only to test equality of a given counter to 0, increment a given counter or decrement a given counter. Initially all the counters are set to 0, except for the one encoding the input.

Remark 1 *The machine is usually considered as computing functions over the integers, or as recognizing languages defined as subsets of the integers. If one wants to compute over words, say words over the alphabet* Σ *is* $\{0,1\}$*, this requires to encode words into integers, for example by considering binary expansions.*

Remark 2 *We consider in this document that machines either halt or loop. Rejection is encoded in the coming simulation by the fact that the machine does not halt. It would be possible also to consider counter machines with* Accept *and* Reject *instructions, to simulate in a fine way acceptance and rejection of Turing machines.*

Remark 3 *This is hence a (S)RAM, but with a extremely reduced set of instructions, and with furthermore a finite number of registers.*

More formally, all the instructions of a counter machine are of the following 4 types:

- Inc(c, j): counter c is incremented and then one goes to instruction j;
- Decr(c, j): counter c is decremented (if non null, unchanged otherwise) and then one goes to instruction j;
- IsZero(c, j, k): one tests whether counter c is 0 and one goes to instruction j if this is the case, or to instruction k otherwise;
- Halt: the computation is halted.

Example 2 *For example, the following program with* 3 *counters*

1. IsZero$(1,5,2)$
2. Decr$(1,3)$
3. Inc$(3,4)$
4. Inc$(3,1)$
5. Halt

transforms $(n,0,0)$ *into* $(0,0,2n)$*: If one starts with* $r_1 = n$*,* $r_2 = r_3 = 0$*, then when instruction* Halt *is reached, we have* $r_3 = 2n$*, and all other counters set to* 0*.*

Exercise 1 *For every of the following conditions, describe some counters machine that reaches instruction* Halt *if and only if the following condition is true on the initial condition:*

1. $r_1 \geq r_2 \geq 1$;
2. $r_1 = r_2$ *or* $r_1 = r_3$;
3. $r_1 = r_2$ *or* $r_1 = r_3$ *or* $r_2 = r_3$.

Theorem 3 *Every machine with k-stacks can be simulated by a machine with $k+1$ counters.*

Proof: The idea is to see a stack $w = a_1 a_2 \cdots a_n$ on an alphabet Σ of size $r-1$ as an integer i in basis (radix) r: Without loss of generality, we can consider Σ to be $\Sigma = \{0, 1, \cdots, r-1\}$. The word w can be interpreted as the integer $i = a_n r^{n-1} + a_{n-1} r^{n-2} + \cdots a_2 r + a_1$.

One uses a counter i for every stack. A $k+1$th counter, that we will call *additional counter*, is used to adjust the value of the counters and simulate every operation (push, pop, reading the top) of one of the stacks.

Popping is replacing i by i div r, where div denotes integer division: Starting with the additional counter set to 0, one iteratively in a loop decrements the counter i by r (in r steps) and increments the additional counter by 1. This operation is repeated until counter i reaches value 0. One then copies the additional counter to counter i: one iteratively decrements the additional counter by 1 while incrementing counter i by 1 until the additional counter reaches 0. At this moment, counter i contains the correct result.

Pushing symbol a is replacing i by $i * r + a$: First, one multiplies the counter i by r: Starting with the additional counter set to 0, one decrements the counter i by 1 and one increments the additional counter by r (in r steps) until counter i reaches 0. One then decrements the additional counter by 1 while incrementing counter i until the former reaches 0. At this moment, one reads counter i contains $i * r$. One then increments counter i by a (using a incrementation operations).

Reading the top of stack i is computing $i \ mod \ r$, where $i \ mod \ r$ denotes the remainder of the Euclidean division of i by r: We start by moving the content of i to the additional counter as follows. Starting with additional counter set to 0, one decrements counter i by 1 and one increments the additional counter by 1. When counter i reaches 0 one stops. One then decrements the additional counter by 1 while incrementing counter i until the former reaches 0. While doing this, one memorizes in parallel the number of operations incrementations to counter i performed modulo r in the internal state. When the loop is done, the internal state thus contains i modulo r. □

Theorem 4 *Every machine with $k \geq 3$ counters can be simulated by a machine with 2 counters.*

Proof: Suppose first that $k = 3$.

The idea is to encode the three counters i, j and k by integer $m = 2^i 3^j 5^k$. One of the counters stores this integer. The other counter is used to do multiplications, divisions, and compute remainders modulo 2, 3, and 5.

To increment i, j or k by 1, it is sufficient to multiply m by 2, 3 or 5 by using the techniques of the previous proof.

To test whether i, j or $k = 0$, it is sufficient to test whether m is divisible by 2, 3 or 5, by using the techniques of the previous proof.

To decrement i, j or k of 1, it is sufficient to divide m by 2, 3 or 5 using the techniques of the previous proof.

For $k > 3$, we use the same approach, but with the first k prime numbers instead of simply 2, 3, and 5. □

Exercise 2 *Reconsider the previous exercise but by using systematically at most 2 counters.*

By combining the previous results, we obtain:

Corollary 1 *Every Turing machine can be simulated by a 2 counters machine.*

Remark 4 *Observe that the simulation is particularly inefficient: The simulation of a time t of the Turing machine requires an exponential time by a 2 counter machine.*

3 Church-Turing thesis

3.1 Equivalence of all considered models

In this chapter, we have introduced various models, and we have shown that they can all be simulated by Turing machines, or simulate Turing machines.

Actually, all these models are equivalent in terms of what they are able to compute: We already proved that Turing machines can simulate RAMs. We could also easily prove the contrary: One can simulate a Turing machine using a RAM.

We have also shown that the counter machines and the stack machines with 2 or more stacks can simulate Turing machines. It is easy to see that the contrary holds: One can simulate the evolution of a stack machine or of counter machine by a Turing machine.

Consequently, all these models are equivalent at the level of what they can compute.

3.2 Church-Turing thesis

The equivalences we observed before and many others led to the Church-Turing thesis, expressed historically by Alonzo Church, Alan Turing and later also formalized by Stephen Kleene. This thesis states that "what is effectively *calculable* is computable by a Turing machine".

In this formulation, the first notion of "*calculable*" makes reference to a intuitively given notion, while the second notion of "computable" means "computable by a Turing machine", i.e., a formal notion.

Since it is not possible to formally capture the first notion, this is a thesis in the *philosophical* sense of this term: It is not possible to prove it.

However, given two (sufficiently expressive) models, we can mathematically prove, as we did, that everything that can be computed by the first can be simulated, and hence computed, by the second (and conversely). This gives evidence that the notion of "computable" as defined with Turing machines is matching the intuitive notion of "computable" (or calculable).

While it cannot be proved, the Church-Turing thesis is widely assumed to be true.

4 Bibliographic notes

Suggested readings To go further with all the mentioned notions in this chapter, we suggest to read [Sipser, 1997],[Hopcroft et al., 2001].

Other formalisms equivalent to Turing machines exist, in particular the notion of recursive functions that is presented for example in [Dowek, 2008], [Stern, 1994] or in [Cori and Lascar, 1993].

Bibliography (S)RAM is inspired by [Papadimitriou, 1994] and [Jones, 1997].

Index

Bibliography

[Cori and Lascar, 1993] Cori, R. and Lascar, D. (1993). *Logique Mathématique, volume II*. Masson.

[Dowek, 2008] Dowek, G. (2008). *Les démonstrations et les algorithmes*. Polycopié du cours de l'Ecole Polytechnique.

[Hopcroft et al., 2001] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition.

[Jones, 1997] Jones, N. (1997). *Computability and complexity, from a programming perspective*. MIT press.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company.

[Stern, 1994] Stern, J. (1994). Fondements mathématiques de l'informatique. *Ediscience International, Paris*.