Modelling of Computing: First Lectures

Marc Bezem

Fall 2020

Contents

Chapter 1

Introduction and Overview

Welcome to the course INF210, Modelling of Computing! All information can be found on `mitt.uib.no/courses/24954`, including these lecture notes and the textbook. In this course you will learn about:

- Various machine models;
- What can/not be computed by this and this model;
- What can/not be computed at all;
- Formal languages;
- Regular expressions, search patterns;
- Parsing fundamentals.

On the way, you will get insight in:

- The origins of 'modern' concepts such as:
  - abstract machine state,
  - interpretation (1936, universal Turing Machine),
  - virtual machine,
  - emulation;
- Computing as a science;
- How mathematical analysis has led to better programs, and that
- 'nothing is more practical than a good theory'.

Chapter 2

State Transition Systems

In these lectures you will learn about state transition systems (`en.wikipedia.org/wiki/Transition_system`), a.k.a. directed graphs, or abstract rewriting systems. State transition systems form a simple setting in which many important notions of computing can be introduced, defined and illustrated. These notions are: state, terminal state, termination (weak and strong), reachable state, non/determinism, confluence.

2.1 State Transition Systems

A *state* is just an element of some given set, an abstraction of a 'situation'. Examples are:

- Chess state: the position of all pieces on the chess board, plus the information which player should make the next move. In a timed chess game the amount of time left for each player is part of the state. [1]
- Computer state: an assignment of values to all registers and memory locations.

A *State Transition System* (STS) is a pair (S, R) where S is a set of states, and R a relation on S, that is, a subset of $S \times S$. The relation R is called

[1]Remark by Åsmund Kløvstad: A little known rule of Chess says that a certain move called roccade, involving a King and one of the Rooks, is only allowed when the pieces involved have not moved before. This means that this information should be included in the state, since otherwise the legal moves would not depend on the state only, but also on the history of the game.

the *transition relation*. We often write $a \to b$ for $R(a, b)$ and call such a pair a *transition* or a *step*.

One example is Chess with states as above and (allowed) moves defining the relation between these states. Mathematical examples are:

- Loop: $S = \{0\}$, $R = \{(0, 0)\}$.
- Switch: $S = \{0, 1\}$, $R = \{(0, 1), (1, 0)\}$.
- Collatz: $S = \{1, 2, 3, 4, \ldots\}$,

$R = \{(n, m) \in S \times S \mid (n \text{ even and } m = n/2) \text{ or } (n > 1 \text{ odd and } m = 3n + 1)\}$.

The first example is the simplest possible non-empty STS, modelling a loop. The second example models a switch between two states. The third example is more interesting and will be elaborated after a few more definitions.

2.2 Termination and Reachability

A *terminal state* of an STS (S, R) is a state $x \in S$ such that there exist no $y \in S$ such that $x \to y$. A terminal state 'cannot be left', but should not be confused with a *looping state*, such as the state 0 in the first example Loop.

The example Switch has no terminal state, and no looping states either. The example Collatz has 1 as the only terminal state.

An important notion, both in graph theory and in STS, is the *reachability relation*. Reachability of a state s' from a state s means that there is a sequence of zero or more steps, starting in s and ending in s', each step ending in the state from which the next step departs. In the world of graphs, the reachability relation is called the *path relation*. Depending on your background in mathematics, you may pick one of the following three equivalent definitions:

(1) For all $x, z \in S$: $R^*(x, z)$ iff there are $y_1, \ldots, y_n \in S$ $(n > 0)$ such that $x = y_1$, $z = y_n$ and $y_i \to y_{i+1}$ for all $i = 1, \ldots, n - 1$.

(2) R^* is the smallest reflexive and transitive relation containing R.

(3) R^* is inductively defined by two clauses:

- For all $x \in S$: $R^*(x, x)$;
- For all $x, z \in S$: $R^*(x, z)$ if $R^*(x, y)$ and $R(y, z)$ for some $y \in S$.

An interesting open problem of mathematics is en.wikipedia.org/wiki/Collatz_conjecture: is state 1 reachable from any other state? Click on the link for the amazing stuff behind this seemingly simple STS!

Given an STS (S, R), a state $x \in S$ is called *strongly terminating* if every sequence of steps starting from x is finite, and *weakly terminating* if a terminal state can be reached from x. The STS itself is called weakly/strongly terminating if every $x \in S$ is weakly/strongly terminating.

For the examples Loop and Switch and Collatz, the distinction between weak and strong termination is not relevant, since we always do only one step in each state. The examples Loop and Switch are not terminating, and for Collatz it is unknown. Note that if 1 is reachable from any state, then Collatz is terminating, and conversely. This is because 1 is the only terminal state of Collatz. A very simple example of weak termination that is not strong is the STS $S = \{0, 1\}$, $R = \{(0, 0), (0, 1)\}$. In the following section we see more interesting examples of terminating STSs.

2.3 Determinism and Confluence

An STS (S, R) is *deterministic* if for every state $x \in S$ there is at most one state $y \in S$ such that $R(x, y)$. In other words, if R is a (partial) function. (In pure set theory this is true as per definition; in other fields one would call R the graph of a function.)

Chess is, of course, non-deterministic. Loop, Switch and Collatz are all deterministic. For Loop and Switch the transition relation is even a total function; for Collatz this function is partial since 1 is a terminal state.

Expression evaluation is an example of a terminating, non-deterministic STS that is relevant for computer science. The set of states S consists of all fully parenthesized, well-formed arithmetical expressions with integers and the operations $+$ and $*$. The transitions are as follows. Given such an arithmetical expression with a subexpression $(n + m)$ or $(n * m)$ with n and m integers, replace that subexpression by its value. For example:

$$(2 + 3) * (-2 + 6) \to 5 * (-2 + 6) \to 5 * 4 \to 20$$

Terminal states are integers. The STS is strongly terminating since any transition leads to a smaller expression. One interesting aspect (and a critical one in the operational semantics of programming languages) is that there may be more than one subexpression that one can evaluate. For example, we also have:

$$(2+3)*(-2+6) \to (2+3)*4 \to 5*4 \to 20$$

This aspect is called non-determinism, and the STS in this example is not deterministic. However, in this case the terminal state is the same for both ways of evaluating $(2+3)*(-2+6)$. It is possible to make expression evaluation deterministic, for example, by prescribing that the leftmost subexpression has to be evaluated first. That, however, would limit some potential benefit of parallel evaluation.

The above example leads to the important notion of confluence, which guarantees that, for any given start state, the terminal state reached is independent of the chosen steps.

An STS (S, R) is *confluent* if, for any states $x, y, z \in S$, if y and z are reachable from x, then there is a state $u \in S$ that is reachable from both y and z. (Make a picture.)

Remark: any deterministic STS is trivially confluent. In the following paragraph we show that if a non-deterministic STS is confluent, then for every state, if it has a reachable terminal state, then the latter is unique.

In a confluent STS, if t and t' are both terminal states reachable from state s, then $t = t'$, since u reachable from both t and t' must be equal to both t and t', since the latter two are terminal. (Make a picture.)

Example: Chess has different terminal states reachable from the start state, and hence Chess is not confluent.

2.4 Making an STS deterministic

Let (S, R) be an STS. Let $\mathcal{P}S$ be the power set of S, that is, the set of all subsets of S. For any $x \in S$, define R_x be the set of $y \in S$ such that $R(x, y)$. For any $X \in \mathcal{P}S$, define $r(X) = \bigcup_{x \in X} R_x$. Then $r(X)$ is a subset of S. Moreover, r is a function. Now define a transition relation $R_{\mathcal{P}}$ on $\mathcal{P}S$ by $R_{\mathcal{P}}(X, r(X))$ for all $X \in \mathcal{P}S$. In other words, $R_{\mathcal{P}}$ is the graph of the function r. Hence $(\mathcal{P}S, R_{\mathcal{P}})$ is a deterministic STS.

Example. States $S = \{a, b, c\}$ and steps $a \to b$, $a \to c$, $b \to a$. Then $\mathcal{P}S = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$. We see that we get an exponential number of states. Make a picture for the steps! Here they come (easy ones first):

$$\begin{aligned}
\emptyset &\longrightarrow \emptyset \\
\{c\} &\longrightarrow \emptyset \\
\{b\} &\longrightarrow \{b,c\} \\
\{a\} &\longrightarrow \{b,c\} \\
\{b,c\} &\longrightarrow \{a\} \\
\{a,c\} &\longrightarrow \{b,c\} \\
\{a,b\} &\longrightarrow \{a,b,c\} \\
\{a,b,c\} &\longrightarrow \{a,b,c\}
\end{aligned}$$

The STSs (S, R) and $(\mathcal{P}S, R_{\mathcal{P}})$ are closely related. Their precise relationship is expressed by the following statements.

(1) For all states $x, y \in S$, if y is reachable from x in (S, R) then there exists a state $Y \in \mathcal{P}S$ such that $y \in Y$ and Y is reachable from $\{x\}$ in $(\mathcal{P}S, R_{\mathcal{P}})$. More formally:

$$R^*(x, y) \quad \text{implies} \quad R^*_{\mathcal{P}}(\{x\}, Y) \text{ for some } Y \text{ with } y \in Y.$$

(2) For all states $x \in S$ and $Y \in \mathcal{P}S$, if Y is reachable from $\{x\}$ in $(\mathcal{P}S, R_{\mathcal{P}})$, then any $y \in Y$ is reachable from x in (S, R). More formally:

$$R^*_{\mathcal{P}}(\{x\}, Y) \quad \text{implies} \quad R^*(x, y) \text{ for any } y \in Y.$$

Proofs by induction on the length of sequences for $R^*(x, y)$ and $R^*_{\mathcal{P}}(\{x\}, Y)$, respectively, adopting the first definition of R^* in 2.2.

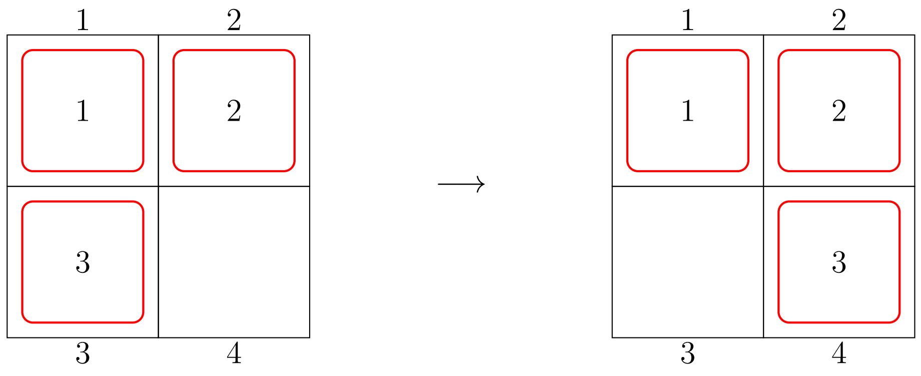
Remark: in this stage it is not clear why making an STS deterministic is interesting. We will later generalise this construction to making various machines deterministic. Deterministic machines are easier to implement: the next state can be computed and need not be guessed. This often outweighs the fact that the states become more complex and that the state space (= set) becomes (much) larger. An example is the unix command `grep`, that builds a deterministic machine searching for text patterns that are specified in a non-deterministic way. We will explore this further in the lectures about regular expressions.

2.5 Exercises

(1) For the STS Collatz above, show that state 1 is reachable from state 17. Write (in your favourite language) a program that checks reachability of state 1 from any state up to 1.000.000. (How do you achieve efficiency?)

(2) (Only if you know chess, or are interested to read `en.wikipedia.org/wiki/Chess` and `en.wikipedia.org/wiki/Fool's_mate`.) For the STS Chess, give two different terminal states (one checkmate and one stalemate). Show that from the initial state one particular final state is reachable in four moves.

(3) Look up `en.wikipedia.org/wiki/15_puzzle`. You will have no difficulty imagining the 8-puzzle. In this exercise we consider the even simpler 3-puzzle. Moves consist in moving an adjacent red square to the empty slot. Here is an example:



In order to analyse this game we need some notation.[2] We have numbered the positions in the frame from 1 to 4. We have numbered the red squares from 1 to 3. It is convenient to give the empty slot number 4. Then a state is just a permutation of $\{1, 2, 3, 4\}$. The left state above is now denoted 1234, and the right state 1243. There are 24 states, a move is now swapping 4 with a number at an adjacent position.

(a) In this representation, figure out the positions adjacent to 4 in $xyz4$, $xy4z$, $x4yz$, and $4xyz$, respectively ($\{x, y, z\} = \{1, 2, 3\}$).

(b) Describe the transition relation in the notation of the previous point.

[2]This is an essential step in modelling of computing!

(c) Give the set of states that is reachable from 1234.

(d) Verify that 2134 is not reachable from 1234.

(e) Argue that this STS is confluent, but not terminating.

(f) (Optional, only if you know/want to learn `en.wikipedia.org/wiki/Parity_of_a_permutation` and `en.wikipedia.org/wiki/Taxicab_geometry`) Note that the parity of the number of steps plus the taxicab distance (also called Manhattan distance) of 4 to its original position is invariant. Use this insight to explain why 2134 is not reachable from 1234.

(4) Complete the proofs by induction of the statements in the last paragraph before the exercises.

Quiz

All answers can be found in the text (by close reading).

(1) What is a terminal state?

(2) Give an example of a weakly but not strongly terminating state.

(3) What is the difference between confluence and non-determinism?

(4) Can there be a state without reachable states?

(5) Does confluence mean that each state has exactly one reachable state?

(6) Is the following true? In any given STS (S, R), if any state is reachable from a given state s, then every state of the deterministic STS $(\mathcal{P}S, R_{\mathcal{P}})$ is reachable from $\{s\}$.

Chapter 3

Labelled Transition Systems

State transition systems are often too abstract to capture essential aspects of a system or game that we want to model. For example, in Chess one is interested in the actual moves and not just in the terminal state (for example, checkmate or stalemate). In fact, in some cases we are more interested in the steps themselves. Think of reaching airport B from airport A. Apart from reachability, one would be interested in the stopovers, how many and which. Imperative programming means specifying the steps that change the machine state of the computer.

Therefore we now introduce a refinement of the notion STS called labelled transition systems and you will soon learn their formal definition (`en.wikipedia.org/wiki/Transition_system`). In a labelled transition system one can keep track of a sequence of steps. The reachability relation is still there, but we can now also analyse, for example, which sequences lead from one given state to a set of states. (This will later turn out to be an important mechanism for defining languages.)

3.1 Labelled Transition Systems

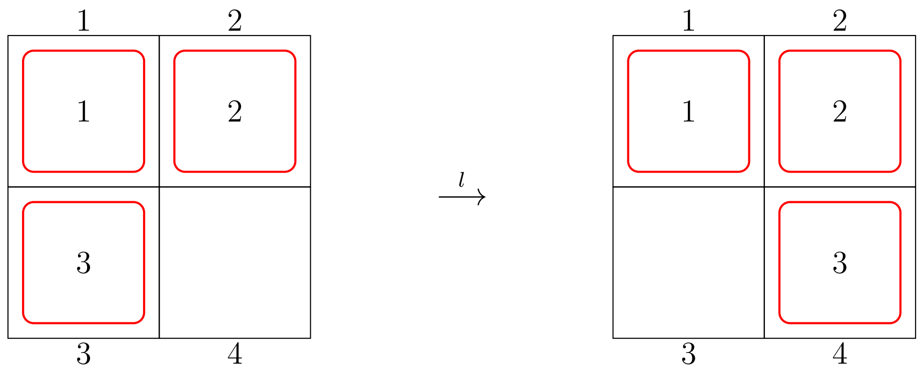
A *Labelled Transition System* (LTS) is a triple (L, S, R) where L is a set of *labels*, S a set of states, and R a ternary relation between two states and label, that is, a subset of $S \times L \times S$. The relation R is called a *(labelled) transition relation*. We often write $a \xrightarrow{l} b$ for $R(a, l, b)$ and call such a triple a *(labelled) transition* or a *(labelled) step*. We have put the word 'labelled' in parenthesis, since we will often leave it out.

One example is Chess with and notation as described in en.wikipedia.org/wiki/Chess_notation.

The mathematical examples can be labelled as well, for example:

- Loop: $L = \{\circlearrowleft\}$, $S = \{0\}$, $R = \{(0, \circlearrowleft, 0)\}$.
- Switch: $L = \{on, off\}$, $S = \{0, 1\}$, $R = \{(0, on, 1), (1, off, 0)\}$.

The 3-puzzle can be extended to an LTS as well. We can view the steps as moving the empty slot (with number 4) either left, up, right, or down. These four possibilities are abbreviated by their first letter and we define the set of labels L to be $\{l, u, r, d\}$. In each state only two of the four steps are possible. For example, in the state below on the left, only l and u are allowed. In the state below on the right, only u and r are allowed. Here is an example of a labelled transition:



In the simplified notation, the above step reads $1234 \xrightarrow{l} 1243$. The other possible step is $1234 \xrightarrow{u} 1432$.

Clearly, sequences of labelled steps lead to sequences of labels. In order to avoid confusion between these different types of sequences we develop in the next section some terminology for sequences of labels. Much of this terminology, as well as the notation, comes from formal language theory, and we will use this vocabulary in later lectures.

3.2 Letters, words and languages

For element, set and sequence of elements we use the following synonyms and notations:

- letter = symbol = label, various notations;

- alphabet = set of letters (or symbols, or labels), often denoted Σ;
- word = string = finite sequence of letters (or symbols, or labels), often denote in boldface, $\mathbf{w}$ or $\vec{w}$, on the blackboard;
- infinite word = infinite sequence = function $\mathbb{N} \to \Sigma$ (without the explicit qualifier 'infinite', words are assumed to be finite);
- language = set of words over an alphabet.

Consider an alphabet Σ, for example, $\Sigma = \{a, b, c\}$.

- A *word over* Σ is a finite sequence of letter from Σ.
- The *empty word* (no letters at all) will be denoted by λ.
- The *length* of a word is its number of letters, for example, $|aba| = 3$.
- The length of $\mathbf{w} = w_1 \cdots w_k$, all $w_i \in \Sigma$, is $|\mathbf{w}| = k$.
- Given $a \in \Sigma$ and $k \geq 0$, we let a^k denote the word of k times the same letter a. (We make no distinction between letters and words consisting of one letter.)

We have the following notations for sets of words, finite and infinite.

- Σ^k is the set of words over Σ of length k.
- Σ^* is the set of all (finite!) words over Σ.
- Σ^+ is the set of all non-empty (finite!) words over Σ.
- Σ^∞ is the set of all infinite words over Σ.

Thus a *language over* Σ is a subset of Σ^*.

Words and languages can be concatenated:

- Concatenation: $(a_1 \cdots a_k) \cdot (b_1 \cdots b_l) = (a_1 \cdots a_k b_1 \cdots b_l)$.
- Words with concatenation form a *monoid*:
  - Concatenation is associative $(\mathbf{x} \cdot \mathbf{y}) \cdot \mathbf{z} = \mathbf{x} \cdot (\mathbf{y} \cdot \mathbf{z})$;
  - Concatenation has unit $\lambda : \mathbf{x} \cdot \lambda = \lambda \cdot \mathbf{x} = \mathbf{x}$.

- Concatenation of languages $S, T \subseteq \Sigma^*$: $S \cdot T = \{\mathbf{s} \cdot \mathbf{t} \mid \mathbf{s} \in S, \mathbf{t} \in T\}$.
- Iteration of language $S \subseteq \Sigma^*$: $S^* = \{\lambda\} \cup S \cup (S \cdot S) \cup (S \cdot S \cdot S) \cup \cdots$.
- Example: $\{00, 1\} \cdot \{\lambda, a, 1\} = \{00, 1, 00a, 1a, 001, 11\}$.

Since concatenation is associative, we may leave out parentheses. If no confusion can arise, we even leave out the symbol $\cdot$ when concatenating words. If the labels are words themselves, we add space between the labels. For example, *off on* in the LTS Switch is a sequence of labels of length two; it is not a word of five letters. When we wish to stress that it is the concatenation of two one-letter words we write *off* $\cdot$ *on*. Another example of a two-letter words is 11 above, not to be confused with the number 11.

3.3 Labelled reachability

Given an LTS (L, S, R), its *labelled reachability relation* is an extension of the labelled transition relation, just like for STSs the reachability relation is an extension of the transition relation.

For defining the labelled reachability relation we have several equivalent options (like for the reachability relation), and we choose to follow the textbook (Chapter 3, p. 46) here. The idea is that we can modify the LTS (L, S, R) into an STS by including sequences of labels in the states of the STS. Thus we consider an STS (S_L, R_L), where the set of states $S_L = S \times L^*$ and the transition relation R_L is defined by $(s, l \cdot \mathbf{w}) \to (s', \mathbf{w})$ if and only if $s \xrightarrow{l} s'$ in R. We denote the reachability relation R_L^* by (infix) $\vdash_R^*$, following the textbook. Note that we have (why?):

$$s, \mathbf{w} \vdash_R^* s', \mathbf{v} \quad \text{iff} \quad s, \mathbf{wu} \vdash_R^* s', \mathbf{vu} \text{ for all } \mathbf{u} \in L^*.$$

Define: s' is *reachable* from s with word $\mathbf{w}$ in (L, S, R) if (s', λ) is reachable from $(s, \mathbf{w})$ in (S_L, R_L), denoted by $s, \mathbf{w} \vdash_R^* s', \lambda$. This elegant but subtle definition deserves some explanation, which we give in the next paragraphs. We also say simply s' is *reachable* from s, without mentioning the word, if s' is *reachable* from s with some word $\mathbf{w}$.

One advantage of using the STS (S_L, R_L) is that we inherit some good properties without having to reprove them, for example, that reachability is transitive. See Exercise (6) below.

(Digression) Using the STS (S_L, R_L) to define reachability in the LTS (L, S, R) stipulates the role that the labels play here. Technically, an LTS is an edge-labelled directed graph. However, in the latter the labels often play a different role, for example, labelling the edge with a number expressing the cost of a transition. In so-called edge-weighted digraphs, the weight of a path is the sum of the weights of the edges. Let's say we have a set of weights $W = L$, usually numbers, with $0 \in W$ and a binary operation $+$ for weights. Then we could define the STS (S_W, R_W) with $S_W = S \times W$ and a step relation R_W defined by $(s, x) \to (s', x + l)$ if and only if $s \xrightarrow{l} s'$ in R. Furthermore, s' is reachable from s with *weight* x in (W, S, R) if (s', x) is reachable from $(s, 0)$ in (S_W, R_W). The pattern of the definition with weights is the same as with words, but serves a different purpose. Both definitions are clean and avoid dot-dot-dot (provided one avoids dot-dot-dot style in STS reachability; some people prefer dot-dot-dot).

In the LTS example Switch above, we have the following sequence of labelled steps:

$$0 \xrightarrow{on} 1 \xrightarrow{off} 0 \xrightarrow{on} 1$$

In the corresponding STS, which we call Switch$_L$ we have the following sequence of unlabelled steps:

$$(0, on \cdot off \cdot on) \longrightarrow (1, off \cdot on) \longrightarrow (0, on) \longrightarrow (1, \lambda)$$

This shows that 1 is reachable from 0 with the word $on \cdot off \cdot on$. Other examples are: 0 is reachable from 0 with the word λ; 0 is reachable from 0 with the word $on \cdot off$, but not with the word $off \cdot on$. Nothing is reachable from 0 with the word *off* in the LTS Switch, although $(0, off)$ is trivially reachable from $(0, off)$ in Switch$_L$.

The above example is not a coincidence, but reveals an equivalent, dot-dot-dot style definition of labelled reachability. For all $s, s' \in S$ we have:

$$s = y_1 \xrightarrow{l_1} y_2 \longrightarrow \cdots \longrightarrow y_{n-1} \xrightarrow{l_{n-1}} y_n = s'$$

for some $n > 0$, $y_1, \ldots, y_n \in S$ and $l_1, \ldots, l_{n-1} \in L$, if and only if

$$s, l_1 \cdots l_{n-1} \vdash_R^* s', \lambda.$$

This can be proved by induction on n (see exercise), and provides a less elegant, but for some people more intuitive definition of labelled reachability.

3.4 Making an LTS deterministic

An LTS (L, S, R) is *deterministic* if for every state $x \in S$ and every label $l \in L$, there is at most one state $y \in S$ such that $R(x, l, y)$. In other words, if R is a (partial) function from $S \times L$ to S.

We will refine the construction of a deterministic STS from a given STS to LTSs. Let (L, S, R) be an LTS. Again we use $\mathcal{P}S$, the power set of S, as the set of states of a new LTS. Thus a state of the new LTS is a set of states of the old LTS. For any $x \in S$ and $l \in L$, define $R_{x,l}$ be the set of $y \in S$ such that $R(x, l, y)$. For any $X \in \mathcal{P}S$ and $l \in L$, define $r(X, l) = \bigcup_{x \in X} R_{x,l}$. Then $r(X, l)$ is a subset of S. Moreover, r is a function. Now define a transition relation $R_{\mathcal{P}}$ by $R_{\mathcal{P}}(X, l, r(X, l))$ for all $X \in \mathcal{P}S$. In other words, $R_{\mathcal{P}}$ is the graph of the (total) function r. Hence $(L, \mathcal{P}S, R_{\mathcal{P}})$ is a deterministic LTS.

The LTSs (L, S, R) and $(L, \mathcal{P}S, R_{\mathcal{P}})$ are closely related. Their precise relationship is expressed by the following statements.

(1) For any state $x \in S$ and word $\mathbf{w} \in L^*$, there exists a unique state $Y_{x,\mathbf{w}} \in \mathcal{P}S$ such that $Y_{x,\mathbf{w}}$ is reachable from $\{x\}$ in $(L, \mathcal{P}S, R_{\mathcal{P}})$ with $\mathbf{w}$.

(2) For any state $x \in S$ and word $\mathbf{w} \in L^*$, $Y_{x,\mathbf{w}}$ is the subset of all states $y \in S$ that are reachable from x in (L, S, R) with $\mathbf{w}$.

Proofs are by induction on the length of $\mathbf{w}$. Examples: $Y_{x,\lambda} = \{x\}$; $Y_{x,l} = r(\{x\}, l)$; $Y_{x,ll'} = r(r(\{x\}, l), l')$. Note how r is iterated here.

3.5 Example: a pocket calculator

In this section we model a simple pocket calculator as an LTS. The pocket calculator has 12 keys: $0, 1, \ldots, 9, \oplus$ and C for 'clear'. It can thus only add natural numbers. Our LTS, let's call it Cosia, has these 12 keys as labels. It has following states: $n?$ (entering first operand) and $n, m?$ (entering second operand) for all $n, m \in \mathbb{N}$. Thus there are infinitely many states. The labelled transitions are ($d = 0, 1, \ldots, 9$):

$$\begin{aligned}
n? &\xrightarrow{C} 0? \\
n? &\xrightarrow{d} (10n + d)? \\
n? &\xrightarrow{\oplus} n, 0? \\
n, m? &\xrightarrow{C} n? \\
n, m? &\xrightarrow{d} n, (10m + d)? \\
n, m? &\xrightarrow{\oplus} (m + n), 0?
\end{aligned}$$

For example, the state 60? is reachable from the state 0? with the word $12{\oplus}3{\oplus}45{\oplus}C$:

$$\begin{array}{rll}
0?, & 12{\oplus}3{\oplus}45{\oplus}C & \xrightarrow{1} \\
1?, & 2{\oplus}3{\oplus}45{\oplus}C & \xrightarrow{2} \\
12?, & {\oplus}3{\oplus}45{\oplus}C & \xrightarrow{\oplus} \\
12, 0?, & 3{\oplus}45{\oplus}C & \xrightarrow{3} \\
 & \cdots & \xrightarrow{C} \\
60?, & \lambda &
\end{array}$$

3.6 Exercises

(1) Make sure you understand each step in the sequence of steps in Cosia above. Fill out the missing steps.

(2) Give the complete labelled transition relation of the 3-puzzle. Hint: the 24 states can be divided in four groups of six states, of the form $xyz4$, $xy4z$, $x4yz$, $4xyz$.

(3) Make the deterministic LTS for (L, S, R), where $L = \{a, b\}$, $S = \{s_0, s_1, s_2\}$, $R = \{(s_0, a, s_0), (s_0, a, s_1), (s_1, b, s_2), (s_2, b, s_2)\}$. Hint: make pictures for both LTSs, like in Fig. 4.1. Which states are reachable from $\{s_0\}$ with some word? With which words is $\{s_2\}$ reachable from $\{s_0\}$? (The latter question prepares for the definition of a formal language using LTSs.)

(4) Extend the pocket calculator in 3.5 with a multiplication operation, so that you can compute left-associative expressions. Hint: extend the states so that they remember the operation at hand.

(5) Design an LTS with 11 states that describes the payment of the sum of (at least) 10 with units of 1, 2, 5 and 10. This LTS does not give change!

(6) Show that in any given LTS, if state s'' is reachable from state s' with word $\mathbf{v}$, and s' is reachable from state s with word $\mathbf{u}$, then s'' is reachable from s with word $\mathbf{uv}$.

(7) Prove the last statement of Section 3.3. Hint: $\vdash^*$ is transitive.

(8) Prove the last two statements of Section 3.4. Hint: the induction step goes from $\mathbf{w}$ to $\mathbf{w} \cdot l$.

Quiz

All answers can be found in the text (by close reading).

(1) What is the difference between an STS and an LTS?

(2) Do we have $\emptyset^* = \emptyset^+$? And $\emptyset^+ = \emptyset^\infty$?

(3) Is (S_L, R_L) above necessarily deterministic/confluent/terminating?

(4) In (S_L, R_L) above, is any state $(s, \mathbf{w})$ reachable from itself?

(5) Is the following true? In any given LTS (L, S, R) and $\mathbf{w} \in L^*$, every state has at least one state that is reachable with $\mathbf{w}$.

(6) Is the following true? In any given LTS (L, S, R) every state has at least one state that is reachable with some $\mathbf{w} \in L^*$.

Chapter 4

Machines

There is a large variety of machine models, for an overview of a selection, see `en.wikipedia.org/wiki/Automata_theory`. Almost all of them can be described as LTSs, which is the reason that we have started by introducing STSs and LTSs. Important concepts, such as reachability, termination and determinism have been defined for LTSs. Thus they apply to each machine model described as LTS, and need not be described for each machine model separately.

Machine model can be distinguished in several aspects. One is memory, how much memory they have. Another is how the machine can access the memory. Yet another aspect is determinism. In this introduction we give two machine models as LTSs. One is the Finite State Machine (FSM), also called the Finite Automaton (FA).

The other machine model is the well-known Turing Machine (TM), which has, in addition finitely many abstract states like the FSM, a memory in the form of an tape with finitely many symbols from a finite alphabet. There is no bound on how many symbols the tape can have, giving the TM unbounded memory. Any terminating run of the TM only uses a finite part of the tape.

Thus the FSM and the TM are in some sense extremes: finitely many states versus unbounded amount of memory. (There are devices with no memory at all: static Boolean circuits. They are interesting, but they are outside the scope of this course.) Both the FSM and the TM are idealised in that they have no bound on the runtime.

4.1 Finite Automata

A *finite automaton* consist of an LTS (L, S, R) where L and S are finite, plus a *start* state $s_0 \in S$ and a set of *acceptance states* $F \subseteq S$. The start state is also called the *initial* state and the acceptance states are also called *final states*. Thus the finite automaton can be denoted as a 5-tuple (L, S, R, s_0, F).

Notationally there is a lot of variation in the literature. The set of labels L is often called the alphabet and denoted by Σ. In $(Q, \Sigma, \delta, q_0, F)$ from `en.wikipedia.org/wiki/Deterministic_finite_automaton` the set of states Q comes first and R is denoted by δ, respectively by Δ for the non-deterministic variant from `en.wikipedia.org/wiki/Nondeterministic_finite_automaton`. The textbook uses $(\Sigma, Q, s_0, \Upsilon, F)$.

Finite automata can be depicted graphically with the states depicted by circles, and the transitions by labelled arrows between the states. The start state is indicated by an unlabelled incoming arrow, and final states are depicted by double circles. See Figure 4.1 for a simple example.

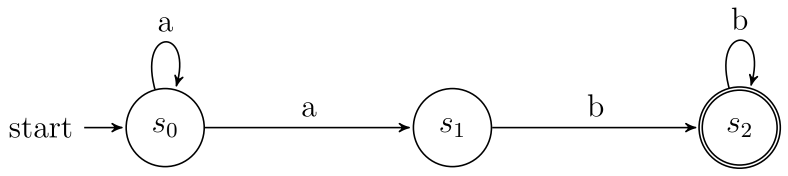


Figure 4.1: A simple finite automaton, see Exercise 3.6(3).

4.2 Turing Machines

The Turing Machine (`en.wikipedia.org/wiki/Turing_machine`) is named after A.M. Turing who introduced this mathematical model of computation in the seminal paper ”On Computable Numbers, with an Application to the Entscheidungsproblem”, `http://www.turingarchive.org/viewer/?id=466&title=01b`. This model became soon the standard for computability, with the so-called Church-Turing Thesis, stating that everything that is computable can be computed by a TM. See `en.wikipedia.org/wiki/History_of_the_Church-Turing_thesis`.

After Church and Turing there have been proposed numerous other models. None of them computes more than a TM, but some of them are surprising by their simplicity. However, Turing and Church were first, and Turing's analysis stands out as the most convincing, philosophically.

The Church-Turing Thesis cannot be proved, it is a postulate of the theory of computation. There is no (generally accepted) example of a computation that cannot be done on a TM. A TM cannot compute a *perfect* random generator, but this is not seen as a counterexample to the Church-Turing Thesis.

Assuming this thesis, we can now ask whether something can be computed or not. One famous problem that is not computable is the so-called Halting Problem. In the practice of computer science, a computer can do many very many useful things, like parsing a program, type-checking it, and generate machine code so that the program can run. However, a computer cannot determine whether a program will terminate or not (on the assumption that the program is run on a computer with unbounded memory).

There are very many variants of the TM; here we give a simple description. (More complicated variants can give some technical advantages.) The essential ingredients of TM are:

(1) A finite alphabet Σ that includes the blank symbol #;

(2) A finite set of states Q, including a start state q_0 and a set $H = \{h_0, h_1\}$ of two halting states;

(3) A tape with successive cells, infinite in one direction, say to the right;

(4) A read/write head on the tape, can also move one cell left or right;

(5) A transition *function* $\delta : (Q - H) \times \Sigma \longrightarrow Q \times \Sigma \times \{-1, 0, +1\}$.

See Figure 4.2 for an example. Thus we can describe a TM as a 5-tuple $(\Sigma, Q, \delta, q_0, H)$. We will now explain how a TM works by modelling it as a deterministic LTS. This technique is one application of the notion of LTS, namely defining the so-called operational semantics of a machine or formalism.

The ingredients of the LTS (L, S, R) modelling the TM are:

(1) The finite set of labels (the alphabet) $L = \Sigma$, the same as of the TM;

(2) The set of states $S = Q \times \mathbb{N} \times (\mathbb{N} \to \Sigma)$ explained below;

(3) The (deterministic) transition relation $R \subseteq S \times \Sigma \times S$ described below.

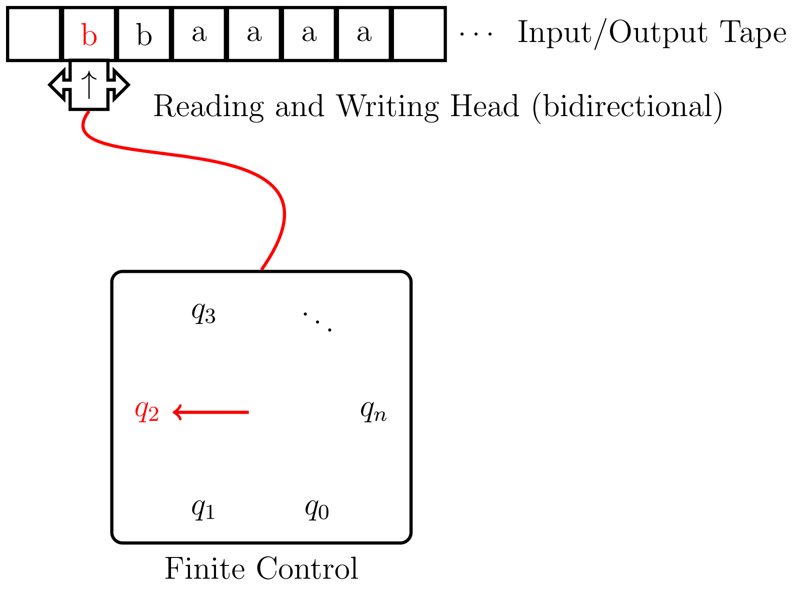


Figure 4.2: A state of a TM, one in which $\delta(b, q_2)$ applies.

Thus the states of this LTS are triples of the form (q, n, f) with $q \in Q$ a state of the TM, $n \in \mathbb{N}$ the position of the head, and $f : \mathbb{N} \to \Sigma$ representing the contents of the tape by $f(0), f(1), f(2), \ldots$.

In Figure 4.2, we would have: $q = q_2$, $n = 1$ $f(0) = \#$, $f(1) = f(2) = b$, $f(3) = f(4) = f(5) = a$, and $f(n) = \#$ for all $n > 5$, on the assumption that the part of the tape that is not shown is filled with blanks.

Now we turn to the labelled transitions of (L, S, R). In the state (q, n, f), the symbol that is read by the head is $f(n) \in \Sigma$. The general form of a transition step is therefore:

$$(q, n, f) \xrightarrow{f(n)} (q', n', f') \qquad \text{provided } q \notin H.$$

In order to define the new state (q', n', f') we first determine

$$\delta(q, f(n)) = (q', s, m),$$

where q' is the new TM state, s the symbol written, and $m \in \{-1, 0, +1\}$) the move of the read/write head. Then we define:

- If $n = 0$ and $m = -1$ there is no transition, meaning we terminate; Otherwise:

(1) q' is as given by $\delta(q, f(n)) = (q', s, m)$;

(2) $f'(x) = s$ if $x = n$ and $f'(x) = f(x)$ otherwise; this means that the symbol s is written on the tape at position n;

(3) $n' = n + m$, that is, the head moves as determined by m.

Remarks on the Turing Machine

1. Termination because of $\delta(q, s) = (q', s', -1)$ when the read/write head is in the leftmost position is irregular and can be considered a run-time error. Note that q cannot be in H in this case. There are variants of the TM that avoid this run-time error, for example, by letting the tape be infinite in both directions, or by always having a special symbol on the first position.

2. For Turing it was of importance that the state of his machine had a finite description. Therefore he imposed the restriction that the tape contains only finitely many non-blank symbols. Starting from such a state, one easily proves by induction on the number of steps that this invariant is maintained under any (finite) computation. We could also have imposed this restriction, but didn't do so, modelling the tape simply as an arbitrary function $\mathbb{N} \to \Sigma$.

3. There is also a non-deterministic variant of the TM, where the function δ is relaxed to a relation. This machine model has been extremely important for defining the famous P versus NP problem, see `en.wikipedia.org/wiki/Nondeterministic_Turing_machine`.

4. (Digression) The fact that a perfect random generator cannot be computed by a TM is actually very interesting, and has in the 1960's led to a whole new branch of algorithmic information theory. The idea is that a completely random string, say of booleans, has a high information content since you have to know it in full to be able to reproduce it. On the other hand, a very predictable string like 011011011011011, even if it would be much longer, can be described in a much shorter way, for example as $(011)^5$. The idea is now, roughly, to take the smallest TM computing a string as the measure of the randomness of that string. The more random a string is, the more information it contains, the larger the smallest TM computing it is (and the longer a lossless compression would be). For more information on this exciting subject, see `en.wikipedia.org/wiki/Kolmogorov_complexity`.

Chapter 5

Textbook Annotations

In this Chapter we list the material from the textbook (Tb) that is part of the syllabus. We add remarks, annotations, corrections, and fill gaps in the proofs. The numbering refers to the chapters and sections of the textbook.

5.1 Tb 2.1: Regular languages

Some of this material is in our Sec. 3.2. Important are the definitions of:

- The $*$ operation called the Kleene star;
- Regular expression (`en.wikipedia.org/wiki/Regular_expression`);
- Regular language (`en.wikipedia.org/wiki/Regular_language`).

5.2 Tb 3.1: Deterministic and nondeterministic automata

These automata are our (non-)deterministic FSMs in Sec. 4.1. Most important are:

- Definition that is not clearly stated in the textbook: the *language* L_M *accepted* by the FSM $M = (\Sigma, Q, s_0, \Upsilon, F)$ consist of all $\mathbf{w} \in \Sigma$ such that a final state $s \in F$ is reachable from s_0 with word $\mathbf{w}$. In one formula:

$$L_M = \{\mathbf{w} \in \Sigma \mid s_0, \mathbf{w} \vdash_{\Upsilon}^{*} s, \lambda \quad \text{for some } s \in F\};$$

This definition works equally well for LTSs with initial state and final states, without any finiteness requirement.

- Theorem: any language that is accepted by an FSM is also accepted by a deterministic FSM, constructed following the recipe given in our Sec. 3.4. The result holds in general for LTSs and has nothing to do with the finiteness of the FSM. The proof follows from the following equivalence based on (1) and (2) in Sec. 3.4:

  $$s_0, \mathbf{w} \vdash_R^* s, \lambda \quad \text{iff} \quad \{s_0\}, \mathbf{w} \vdash_{R_{\mathcal{P}}}^* Y_{s_0,\mathbf{w}}, \lambda \text{ and } s \in Y_{s_0,\mathbf{w}}.$$

  Now use that $Y_{s_0,\mathbf{w}}$ is final iff it contains a final state.

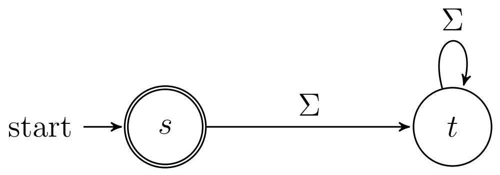
- In my version there is a mistake in l. 2, p. 48: the transition function Υ maps $Q \times (\Sigma \cup \{\lambda\})$ to Q. The idea of so-called λ-moves (also called ϵ-transitions/steps in the literature) is that you can do a step from state to state without 'consuming' a letter. In these paragraphs the textbook proves that FAs with λ-moves accept same languages as FAs without. In other words, λ-moves do not make finite automata more powerful from the point of view of accepting languages, only more convenient. We sketch an alternative, perhaps more intuitive proof of this fact in Section 5.4. Again, the result has nothing to do with the finiteness of FAs.

5.3 Tb 3.2: Kleene's Theorem

A classical result relating a language class to a machine class: a language is regular if and only if it is accepted by an FSM.

Here come some remarks on this section.

- Cryptic remark in 2nd sentence on p. 53 seems to refer to this FSM accepting the language $\{\lambda\}$:



The above FSM is deterministic and has a total transition function, the simpler one in the textbook has a partial transition function, actually the empty function. Note that we have used a new convention to label a step with Σ, meaning that we have this step for all letters in the alphabet. The state t can be considered to be a 'trash' collecting all words that are not accepted. This is sometimes a useful technique. With this technique you can show that deterministic FSMs with a total transition function accept the same languages as FSMs with a partial transition function (see Exercise (1) in Section 5.6).

- The sentence in the middle of p. 54 refers to a λ-move:

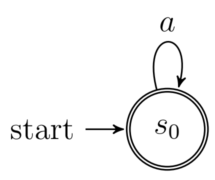
  "If state s_0 is an acceptance state, go to state s_0'."

  In Section 5.4 we explain how to deal with this.

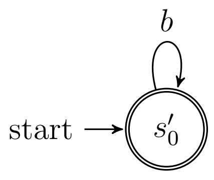
- The proof of Lemma 3.4, page 68, contains some typos. The last formula should read $L = \cup\{R(1, n+1, q_j) : q_j \in F\}$. Five lines above, the first m should be q_m. A few lines below the proof δ should be Υ.

5.4 ϵ-Transitions

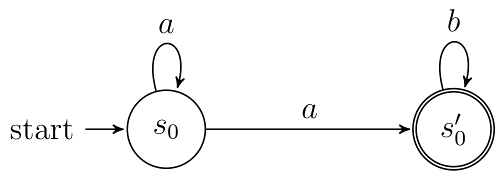
An *ϵ-transition*, or *ϵ-step*, is a transition that does not consume input, like ordinary transitions do. Here is an example why this can be useful. Consider the following FSMs M_1, M_2 and M:



M_1



M_2

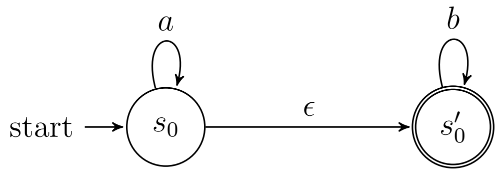


M

Here M_1 accepts a^*, M_2 accepts b^*, and M accepts a^+b^*. The FSM M is constructed following the recipe for the concatenation of the languages a^* and b^*. This does not work since a^+b^* is not the same language as a^*b^*. The problem is caused by the fact that in M_1 the initial state is also final, so that there is an incoming arrow of a final state of M_1 that is not accounted for.

There are several solutions to this problem. One is to let in M the state s_0' also be a start state. This requires the notion of FSM to be extended with multiple start states. Another solution, also requiring an extension of the notion of FSM, is to introduce so-called ϵ-steps, that is, transitions that change the state without consuming a letter from the input. In other words, an ϵ-step is an STS step in an LTS. If we denote the ϵ-step by $s \xrightarrow{\epsilon} s'$, then its meaning would be $s, \mathbf{w} \vdash^* s', \mathbf{w}$.

With ϵ-steps we can make the following ϵ-FSM accepting a^*b^*:



Since the goal is an FSM and not an ϵ-FSM, we explain the latter as a special notation of the former. Given an ϵ-FSM, every sequence

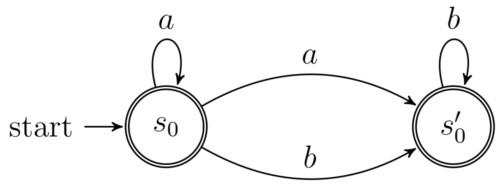
$$s \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} t \xrightarrow{a} t' \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} s'$$

with at least one ϵ-step in total, leads to a step $s \xrightarrow{a} s'$. Moreover, every state from which a final state can be reached by a sequence of ϵ-steps should also become a final state. Thereafter all ϵ-steps have become redundant and can be removed, yielding an FSM accepting the same language.

In the case of the above example with one single ϵ-step this operation is rather easy and the only such sequences are

$$s_0 \xrightarrow{\epsilon} s_0' \xrightarrow{b} s_0' \quad \text{and} \quad s_0 \xrightarrow{a} s_0 \xrightarrow{\epsilon} s_0'.$$

Thus we get the following FSM, indeed accepting a^*b^*:



Note that in this special case, coincidentally, the step $s_0 \xrightarrow{a} s_0'$ is redundant.

5.5 Tb 3.3: Minimal deterministic automata and syntactic monoids

The qualifier 'minimal' in the title refers to the minimal number of states of a deterministic FA accepting (DFA) a given language L over some alphabet Σ. In the literature, this topic is also called *state minimization*, see the webpage `en.wikipedia.org/wiki/DFA_minimization`.

The main result is an algorithm that transforms any given DFA into a minimal one that accepts the same language. This is quite useful and the algorithm is reasonably efficient. The problem is just that the given DFA can have a large number of states, and even the minimal DFA can have a large number of states. This just reflects the fact that some regular language are actually quite complicated.

The chapter works with two basic assumptions about the given DFA $M = (\Sigma, Q, s_0, \Upsilon, F)$:

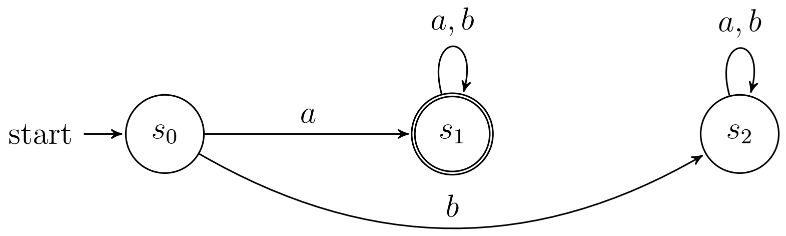
(1) All states are reachable from the initial state s_0 (unreachable states can be left out without changing the accepted language);

(2) The transition function Υ is total, and can be extended from $Q \times \Sigma \to Q$ to $\Upsilon^* : Q \times \Sigma^* \to Q$ in the standard way:

$$\Upsilon^*(x, \lambda) := x \qquad \Upsilon^*(x, \mathbf{w} \cdot l) := \Upsilon(\Upsilon^*(x, \mathbf{w}), l).$$

Here come some remarks on this section.

- On p. 73, it is a bit hard to understand what co-accessible means. Accessible = reachable states are states that you can reach from the intial state. Co-accessible states are states from which you can reach a final state. You can define them as the states that are from a final state if you reverse all arrows. Thus states that are not co-accessible are states from which you cannot reach a final state. They could be removed without changing the accepted language. However, there is a complication, illustrated by:



This FA accepts exactly all words starting with the letter a. The state s_2 is not co-accessible from the final state s_1. However, removing s_2 would make the transition function partial, as there is not longer a step from s_0 with the letter b. In fact, the minimal number of states depends on whether the transition function may be partial or not, the difference is at most one 'garbage state' like s_2.

- In Def. 3.5, the initial state 1 of the minimal automaton is meant to be $[\lambda]$. Note that M is an LTS, a 'directed arrow-labelled graph'.
- In Def. 3.6, the unit = identity element of the monoid is not mentioned, but is meant to be $[[\lambda]]$. Furthermore, not all statements are proved. For example, using the notations from the textbook, it is true that $[[x]][[y]] = [[xy]]$ is well-defined, but it requires an argument that $[[x'y']] = [[xy]]$ if $[[x]] = [[x']]$ and $[[y]] = [[y']]$. One of the cases goes like this: if $uxyv \in L$, then $ux'(yv) \in L$ as $[[x]] = [[x']]$, so $(ux')y'v \in L$, and hence $ux'y'v \in L$ as $[[y]] = [[y']]$. And so on. And then one understands that $[[\lambda]]$ is indeed the unit, since $x = \lambda x = x\lambda$.
- You may ask, on page 75, why $LR(x) = LR(y)$ implies $R(x) = R(y)$. This is not difficult: assume $LR(x) = LR(y)$ and $xv \in L$. Then $\lambda xv \in L$, so $(\lambda, v) \in LR(x)$, so $(\lambda, v) \in LR(y)$, so $\lambda yv \in L$, so $yv \in L$. This implies that the intrinsic automaton is finite if the syntactic monoid is finite.
- In the next paragraph there is a typo: $y \in \Sigma^*$ should be $x \in \Sigma^*$, and the function defined by this x is $[u] \mapsto [ux]$. This function is well-defined since $[u] = [u']$ means $R(u) = R(u')$ which implies $R(ux) = R(u'x)$, that is, $[ux] = [u'x]$. In other words, the function value does not depend on the choice of the element in the class $[u]$. If there are finitely many such classes, there are finitely many such functions. Moreover, the following are equivalent: (1) x and y define the same function; (2) for all $u \in \Sigma^*$, $[ux] = [uy]$; (3) for all $u \in \Sigma^*$, $R(ux) = R(uy)$; (4) $LR(x) = LR(y)$. This

implies that the syntactic monoid is finite if the intrinsic automaton is finite.

- Page 78 contains the proof of the fact that all reduced automata are isomorphic to the intrinsic automaton. It is hard to read since the original FA is not made explicit, and a reduced one is called $M = (\Sigma, Q, s_0, \Upsilon', F)$ and there is a lot of confusion about states versus classes of states. Also: what is Υ, x_0? Further $= x$ with $x \in [x]$ should probably be $= y$ with $y \in [x]$ since $x \in [x]$ is always true.
- In the transformation monoid, the order of the words is reversed, as demonstrated by $\bar{a}\bar{b}(s) = \bar{a}(\bar{b}(s)$. The formula $\bar{a}\bar{b} = \bar{ab}$ is not informative. Better take the definition in my StateMinimization.pdf.

5.6 Exercises on FSMs

Here come some more challenging exercise projects (with hints).

(1) Show that deterministic FSMs with a total transition function accept the same languages as FSMs with a partial transition function. Hint: use a garbage state.

(2) A monoid morphism φ from a monoid $(M, 1, *)$ to a monoid $(M', 1', *')$ is a function $\varphi : M \to M'$ such that $\varphi(1) = 1'$ and $\varphi(x * y) = \varphi(x) *' \varphi(y)$. In other words, φ preserves the unit and the operation. We have seen in Section 3.2 that $(\Sigma^*, \lambda, \cdot)$ is a monoid. Now let $(M, 1, *)$ be a *finite* monoid and φ a morphism from $(\Sigma^*, \lambda, \cdot)$ to $(M, 1, *)$. Show first that $\varphi(l_1 \cdots l_n) = \varphi(l_1) * \cdots * \varphi(l_n)$. In other words, the image of a word is given by the images of its letters. Then show for any subset F of M that $L = \{\mathbf{w} \in \Sigma^* \mid \varphi(\mathbf{w}) \in F\}$ is regular. Hint: build an FSM accepting L with set of states M. Figure out what the transition function is, as well as the initial state and the set of final states.

As a refinement, show that $\{a^{|\mathbf{w}|} \mid \mathbf{w} \in L\}$ is regular, with a fixed (but arbitrary) and L as above. In other words, the set of lengths of words in L in unary notation is a regular language.

Section 3.3 of the textbook shows that all regular languages are the inverse image of a morphism from Σ^* to a finite monoid. Hence the results of this exercise hold for all regular languages.

(3) Let L be a language over Σ. For words $\mathbf{v}, \mathbf{w} \in \Sigma^*$, define the relation $\mathbf{v} \sim_L \mathbf{w}$ by $\mathbf{vu} \in L$ if and only if $\mathbf{wu} \in L$ for all $\mathbf{u} \in \Sigma^*$.

Show first that $\sim_L$ is an equivalence relation (reflexive, symmetric and transitive). Let $[\mathbf{w}]$ denote the equivalence class of $\mathbf{w}$ modulo $\sim_L$. The set of all equivalence classes is denoted $\Sigma^*/\sim_L$.

As an example, consider the language $\{a^n b^n \mid n \geq 0\}$. Show that all equivalence classes $[a^n]$, $n \geq 0$, are different. Show that all equivalence classes $[ba^n]$, $n \geq 0$, are equal.

Consider the LTS $(\Sigma, \Sigma^*/\sim_L, R_L)$ with R_L defined by $R_L([\mathbf{w}], l, [\mathbf{w} \cdot l])$. Show that R_L is well-defined, that is, if $\mathbf{v} \sim_L \mathbf{w}$ then $R_L([\mathbf{v}], l, [\mathbf{v} \cdot l])$ iff $R_L([\mathbf{w}], l, [\mathbf{w} \cdot l])$. Show that the LTS is deterministic.

Assume now that L is accepted by an FSM $(\Sigma, Q, q_0, \Upsilon, F)$. Show that, for any $q \in Q$ and words $\mathbf{v}, \mathbf{w} \in \Sigma^*$, if q is reachable from q_0 with both $\mathbf{v}$ and $\mathbf{w}$, then $\mathbf{v} \sim_L \mathbf{w}$. Show that $\Sigma^*/\sim_L$ is finite.

Use the results obtained in this exercise to prove that the language $\{a^n b^n \mid n \geq 0\}$ is not accepted by an FSM.