Chapter 2.

X-machines - a computational model framework.

This chapter has three aims:

- To examine the main existing computational models and assess their computational power.
- To present the X-machines as a unified computational model framework of which the above mentioned models are particular cases and to highlight its potential as a specification tool.
- To investigate two natural classes of X-machines (i.e. straight-move stream X-machines and stream X-machines) and discuss them from two points of view:
  - how general they are as specification tools
  - to what extent they can provide a basis for a theoretical specification based strategy.

2.1. Computational models survey.

The emergence of the concept of *computable function* over sixty years ago marked the birth of a new branch in mathematics. A main purpose of *computability theory* is to make precise the intuitive idea of a computable function; that is, a function whose values can be calculated in some kind of automatic or effective way. Certain classes of computable functions (or languages) can be obtained using certain types of procedures; these are the *computational models* corresponding to those classes of functions or languages. These models can be classified according to their *computational power*, i.e. how large is the class of functions and languages they can cope with. This hierarchy of languages and computational models is known as the Chomsky hierarchy.

In this section we shall be presenting the main existing computational models and describe their power in terms of the languages or functions they compute.

2.1.1. Automata.

The finite automaton is a mathematical model of a system with discrete inputs and outputs. The system can be in any of a finite number of internal states. The state of the system summarises the information concerning past inputs that is needed to determine the behaviour of the system on subsequent inputs.

Definition 2.1.1.1.
Let Σ be a finite alphabet. A *finite automaton* (or *finite state machine*) denoted *FA*, over Σ consists of the following components (see Cohen [8], Kain [40] or Hopcroft & Ullman [33, 34]):

1. A finite set Q of elements called *states*.
2. A subset I of Q containing the *initial* states.
3. A subset T of Q containing the *terminal* states.
4. A finite set of *transitions* that give, for each state and for each letter of the input alphabet, which state (or states), if any, to go to next. This can be represented as a partial function $F: Q \times \Sigma \rightarrow \wp Q$, where $F(q, \sigma)$ contains the states the machine is allowed to go to from state q when it receives the input σ.

Alternatively, each letter $\sigma \in \Sigma$ can be regarded as a partial function $\sigma: Q \rightarrow \wp Q$. Each triple $(q, \sigma, q') \in Q \times \Sigma \times Q$, where $q' \in \sigma(q)$, is called an *edge* of the automaton. We frequently use the notation $q \xrightarrow{\sigma} q'$ to indicate the edge. A sequence of edges is called a *path* .

The automaton is presented with an input string of letters (or characters) that it reads letter by letter starting at the leftmost letter. Beginning at one of the initial states, the letters determine a sequence of states. The sequence may be interrupted when there is no edge corresponding to the letter read and, in this case, the input string is rejected. Otherwise, the sequence ends when the last input letter has been read. If the last state of the sequence is a terminal one, the input string is accepted, otherwise it is rejected. The set $L \subseteq \Sigma^*$ of all the strings accepted is called the *language accepted (or recognised) by the automaton.*

Note: $\Sigma^* = \Sigma^+ \cup \{1\}$, where Σ^+ is the set of all finite strings of characters from Σ and 1 is the empty (or null) string.

The automaton is called *deterministic* if there is only one initial state (i.e. $I = \{q_0\}$) and for each state and each letter there is at most one single next state (i.e. F can be regarded as a partial function $F: Q \times \Sigma \rightarrow Q$). Otherwise, the automaton is called *nondeterministic*. Obviously, the path through a deterministic automaton is fully determined by the input string, whereas for the nondeterministic case, the choices of the initial state and the next state selected for each letter and current state are made

randomly from a set of possible options. In this latter case, a string is accepted if at least one path determined by it through the machine ends in a final state.
Although nondeterministic automata appear to have extra power over the deterministic ones, it has been proved that any language accepted by a nondeterministic automaton can be also accepted by a deterministic one. In other words, the nondeterminism does not increase the power of the automata (see Cohen [8], or Hopcroft & Ullman [34]).

However, in order to evaluate the computational power of the automata and to determine the languages accepted by them, we have to introduce the concept of regular expressions.

Definition 2.1.1.2.
The set of *regular expressions* over an alphabet Σ is defined by the following rules:

1. Every letter of Σ is a regular expression; the empty string, 1, is a regular expression.
2. If r and r' are regular expressions, then so are

rr', r + r', r*

Note: The symbols used have the following meanings:

rr'	r concatenated with r'
r + r'	r or r'
r*	r concatenated with itself any finite number of times

In other words, the set of regular expressions is closed under concatenation, Boolean, or and *.

Definition 2.1.1.3.
A language that can be defined by a regular expression is called a *regular language*.

The following theorem gives the measure of the computational power of the automata (Cohen [8]):

Theorem 2.1.1.4. (Kleene)
1. Any regular language can be accepted by an automaton.
2. Any language accepted by an automaton is regular.

The theorem above shows that the class of the languages accepted by automata is restricted to those which can be defined as regular expressions. However, this class is limited and there are situations where the rules of automata are not sufficient to describe certain computationally interesting behaviour. This idea can be illustrated better by the following example.

Example 2.1.1.5.
Let Σ be an alphabet, a and b be two distinct characters of Σ and $c \in \Sigma^*$ be a string. Then the language

$L = \{a^n c b^n | \ n \in N\}$

is not regular (see Cohen [8]). Therefore there is no finite automaton that will accept it.

The language above can be used, for example, to describe a very simple compiler which will accept arithmetic expressions having the same number of opening and closing brackets.

The reason why no finite automaton could recognise the language L is that, for large enough n, the a^n part has to run around in a circuit and the machine cannot keep track of how many times it had looped around. It cannot therefore distinguish between $a^n c b^n$ and $a^{n+m} c b^n$, for some positive integers n and m. From this example, it appears obvious that the machine has to have an unlimited memory capacity. The following two models increase the computational power of the finite state machine model by adding an unlimited memory structure.

2.1.2. Pushdown automata.

The *pushdown automaton, PDA*, can be regarded as a finite automaton with a primitive memory unit, its stack. A rigorous definition is given below (see Cohen [8]).

Definition 2.1.2.1.

A pushdown automaton, is a tuple having eight elements:

1. An alphabet Σ of input letters; the blank symbol $\delta \notin \Sigma$.
2. An input TAPE (infinite in one direction). Initially, the string of input letters is placed on the TAPE starting in cell i. The rest of the TAPE is filled with δ's.
3. An alphabet Ω of stack characters.
4. A pushdown STACK (infinite in one direction). Initially, the stack is empty (it contains only a special symbol Δ called the bottom of stack character).
5. One START state that has only out-edges and no in-edges.
6. Halt states of two kinds: ACCEPT and REJECT. They have in-edges and no out-edges.
7. Finitely many non branching PUSH states that introduce characters onto the top of the stack.
8. Finitely many branching states of two kinds:
   (i) States that read the next unused letter from the tape which may have out-edges labelled with letters from Σ or the blank character δ.
   (ii) States that read the top character of the STACK which may have out-edges labelled with letters from Σ or the blank character δ.

We further require that the states be connected so as to produce a connected directed graph.

To run a string of input letters on a PDA means to begin from the start state and follow the unlabelled edges and those labelled edges that are appropriate (making choices of edges when necessary) to produce a path through the graph. This path will end either in a halt state or the automaton will crash in a branching state when there is no edge corresponding to the letter read/popped. When the graph contains loops consisting only of PUSH or POP instructions, the input string may loop forever on the machine. An input string with a path that ends in ACCEPT is said to be accepted by the PDA; otherwise it is said to be rejected.

At this point we should discuss the possibility of nondeterminism. A *deterministic PDA* is one for which every string determines a unique path through the machine. A *nondeterministic PDA* is one for which at certain times we may have to choose among possible paths through the machine. We say that an input string is accepted by such a machine if at least one choice leads us to an ACCEPT state.

Obviously, any language that can be accepted by a FA can be also accepted by a PDA (we just translate the formalism of FA into that of PDA and we obtain a PDA without the STACK identical to the FA). In fact PDAs are more powerful that FAs due to the unlimited memory available. For example, the language

$L = \{a^n c b^n | \ n \in N\}$

can be accepted by a *deterministic* PDA (see Cohen [8]).

Unlike the case of the FA, the nondeterminism gives extra power to a PDA, as illustrated by the following example (Cohen [8]).

Example 2.1.2.2.

Let $\Sigma = \{a, b\}$ be an alphabet and

PALINDROME = $\{s \ reverse(s) | \ s \in \Sigma^*\}$ be a language over Σ.

Then:

1. PALINDROME cannot be accepted by any deterministic PDA.
2. PALINDROME can be accepted by a nondeterministic PDA.

[**Note**: reverse(s) denotes the reverse of string s].

In order to evaluate the power of PDA, we have to introduce the notions of grammar in general and context-free grammar in particular.

Definition 2.1.2.3.

A *phrase-structure grammar*, is a way of defining a language and it is specified by four elements:

1. The set of nonterminal symbols, V.
2. The set of terminal symbols, Σ.
3. The start symbol S, $S \in V$.
4. A set of productions of the form

   $\alpha \rightarrow \beta$,

   where $\alpha \in (V \cup \Sigma)^* \ V \ (V \cup \Sigma)^*$, $\beta \in (V \cup \Sigma)^*$.

Note: $XY = \{xy \mid x \in X, y \in Y\}$ where xy is x concatenated with y.
The *language generated* by a phrase-structure grammar is the set of all strings of terminals that can be produced from the start symbol S using the productions as substitutions.

The particular phrase-structure grammars of interest to us are:

1. *Context-free grammar* - all productions have the form
   $A \rightarrow \beta$,
   where $A \in V$, $\beta \in (V \cup \Sigma)^*$.

2. *Regular grammar* - all productions have the form
   $A \rightarrow s\,B$ or $A \rightarrow s$,
   where $A, B \in V$, $s \in \Sigma^*$.

Obviously, the regular grammars are also context-free. Furthermore, there exist context-free languages which are not regular (the languages described by example 2.1.1.5 above belongs to this category of languages (see Cohen [8]).

The following two theorems illustrate the relationship between these grammars and the machines presented above (see Cohen [8] or Hopcroft & Ulman [33, 34]):

Theorem 2.1.2.4.
1. Any regular language can be generated by a regular grammar.
2. Any language generated by a regular grammar is regular.

Theorem 2.1.2.5.
1. Any language accepted by a PDA (deterministic or not) can be generated by a context-free grammar.
2. Any language generated by a context-free grammar can be accepted by a *nondeterministic* PDA.
This theorem states in fact that the class of languages accepted by *nondeterministic* PDAs is identical to that generated by context-free grammars and strictly includes the set of languages accepted by deterministic PDAs. The languages accepted by deterministic PDAs are called *deterministic context-free languages.*

Therefore, the PDA is a more powerful machine than a FA. However, because of the primitive type of memory that it uses (i.e. a stack), its power is too limited to model all real computer systems.

Example 2.1.2.6.
Let $\Sigma = \{a, b\}$ be an alphabet and L be a language over Σ,
$L = \{a^n b^n a^n \mid n \in N\}$.
Then L cannot be accepted by any PDA (deterministic or not) (see Cohen [8]).

A more powerful type of machine will be described in the next section.

2.1.3. Turing machines.

In this section we investigate a third type of recognising device, the Turing machine. The Turing machine has been proposed as a mathematical model for describing procedures. Since our notion of a procedure as a finite sequence of instructions which can be mechanically carried out is not mathematically precise, we can never hope to show formally that it is equivalent to the precise notion of Turing machine. However, from the definition of a Turing machine it will become readily apparent that any computation that can be described by means of a Turing machine can be mechanically carried out. It can be also be shown that any computation that can be carried out on a computer can be described by means of a Turing machine. This strengthens the belief that the Turing machine model is general enough to encompass the intuitive notion of a procedure. It has been hypothesised by Church that any process which could be naturally called a procedure can be realised by a Turing machine. Consequently, computability by a Turing machine has become the accepted definition of a procedure.
The Turing machine model has been defined in various ways in the literature, all of these models being proved to be equivalent. We adopt the following definition due to Cohen, [8].

Definition 2.1.3.1.
A *Turing machine, denoted TM*, is a tuple having six elements:

1. An alphabet Σ of input letters; the blank symbol $\delta \notin \Sigma$.

2. A TAPE divided into a sequence of numbered cells each containing one character or blank. The input word $w \in \Sigma^*$ is presented to the machine one letter per cell beginning in the left-most cell, called i. The rest of the tape is initially filled with blanks, δ's.

3. A TAPE HEAD that can, in one step, read the contents of a cell on the TAPE, replace it with some other character, and reposition itself to the next cell to the right or to the left of the one it has just read. At the start of the processing, the TAPE HEAD always begins by reading the input in cell i. The TAPE HEAD never moves left from cell i; if it is instructed to do so, the machine crashes.

4. An alphabet, Ω, of characters that can be printed on the TAPE by the TAPE HEAD. This can include Σ. We allow the TAPE head to print a δ on the TAPE but we call this erasing and we consider that $\delta \notin \Omega$.

5. A finite set of states including exactly one START state from which we begin execution (and which we may reenter during execution) and some (maybe none) HALT states that cause execution to terminate when we enter them. The other states have no functions, only names.

6. A program, which is a set of rules that tell us, on the basis of the letter the TAPE has just read, how to change states, what to print and where to move the TAPE HEAD. We depict the program as a collection of directed edges connecting the states. Each edge is labelled with a triple of information:

(letter, letter, direction)

The first letter is the one that the TAPE HEAD reads. The second is the letter that the TAPE HEAD prints in the cell before it leaves it. The third component tells the TAPE HEAD whether to move one cell to the right (R) or one cell to the left (L).

We shall consider that all Turing machines are deterministic (nondeterministic Turing machines have also been investigated, but nondeterminism does not add extra power to the model). This means that there is no state that has two or more edges leaving it with the same first letter.

Unlike the FA, every Turing machine T over an alphabet Σ divides the set of strings into the classes:

1. The set of all strings in Σ* that cause T to enter a HALT state. This is called the *language accepted by T.*
2. The set of all strings that cause the machine to crash during execution by moving left from cell i or by being in a state that has no exit edge that wants to read the character the TAPE HEAD is reading.
3. The set of all other strings; that is, strings that cause T to loop forever.

However, for an arbitrary TM, there is no way to determine algorithmically whether some arbitrary input string is accepted by it or not. This is known as the unsolvability of the *Halting Problem for Turing machines* (Cohen [8] or Hopcroft & Ullman [33, 34]). If T is a TM, we say that "T halts when given the input w" if w does not cause T to loop forever (i.e. it either enters a HALT state or crashes during execution). Then the Halting Problem can be formulated as: "given some arbitrary TM, T, and some arbitrary string w, is there an algorithm to decide whether T halts when given the input w?". Since our intuitive definition of an algorithm is a procedure that terminates and we have accepted the TM as the mathematical model of a procedure, it is natural to consider the model of an algorithm to be a TM which halts for any input in Σ*. Then the following theorem states the unsolvability of the Halting Problem (see Cohen [8], Kain [40] or Hopcroft & Ullman [33, 34]).

Theorem 2.1.3.2.
There is no algorithm (i.e. TM that halts for any input string) to determine whether an arbitrary Turing machine T halts when given an arbitrary input w.

Definition 2.1.3.3.
A language L over the alphabet Σ is called *recursively enumerable* if there is a Turing machine that accepts every word in L and either rejects or loops for every word that is not in L.

The following theorem describes the set of recursively enumerable languages in terms of grammars that generate them (see Cohen [8] or Hopcroft & Ullman [33, 34]).

Theorem 2.1.3.4.
1. Any recursively enumerable language can be generated by a phrase-structure grammar.
2. Any language generated by a phrase-structure grammar can be accepted by some Turing machine.

From the theorem above, it is clear that the class of languages accepted by PDAs is included in the set of recursive enumerable languages. The inclusion is strict (we can, for example, prove that the language described by the example 2.1.2.6 is recursively enumerable, (see Cohen [8])).

The class of languages that are accepted by TM that halt for any input are called *recursive languages* and are strictly included in the class of recursively enumerable languages (see Cohen [8] or Hopcroft & Ulman [33, 34]). In view of our intuitive definition of an algorithm, a language L over Σ^* is *recursive* if for any $w \in \Sigma^*$ there exists an algorithm that decides whether $w \in L$ or not.

Another important class of languages is the class of *context-sensitive languages*. A language is called context-sensitive if it can be generated by a phrase grammar in which in any production $\alpha \rightarrow \beta$ (see definition 2.1.2.3), β is at least as long as α. It has been proved that a language is context-sensitive if and only if it is accepted by a nondeterministic Turing machine which, instead of having a potentially infinite tape on which to compute, is restricted to the portion of the tape containing the input $w \in \Sigma^*$ together with the first square that contains a blank (i.e. if the input contains n symbols, the machine uses n+1 squares). This model is called a *linear bounded automaton*. Any context-sensitive language is recursive and there are recursive languages that are not context sensitive (see Cohen [8] or Hopcroft & Ullman [33, 34]).

The class of languages accepted by PDAs is included in the set of context-sensitive languages. The inclusion is strict (we can, for example, prove that the language described by the example 2.1.2.6 is context-sensitive (see Cohen [8])).

At this point, we can draw a diagram (figure 2.1) showing the hierarchy of the machines presented so far, according to their computational power. This diagram can be expanded by adding extra intermediate levels, but this is not the purpose of this chapter.

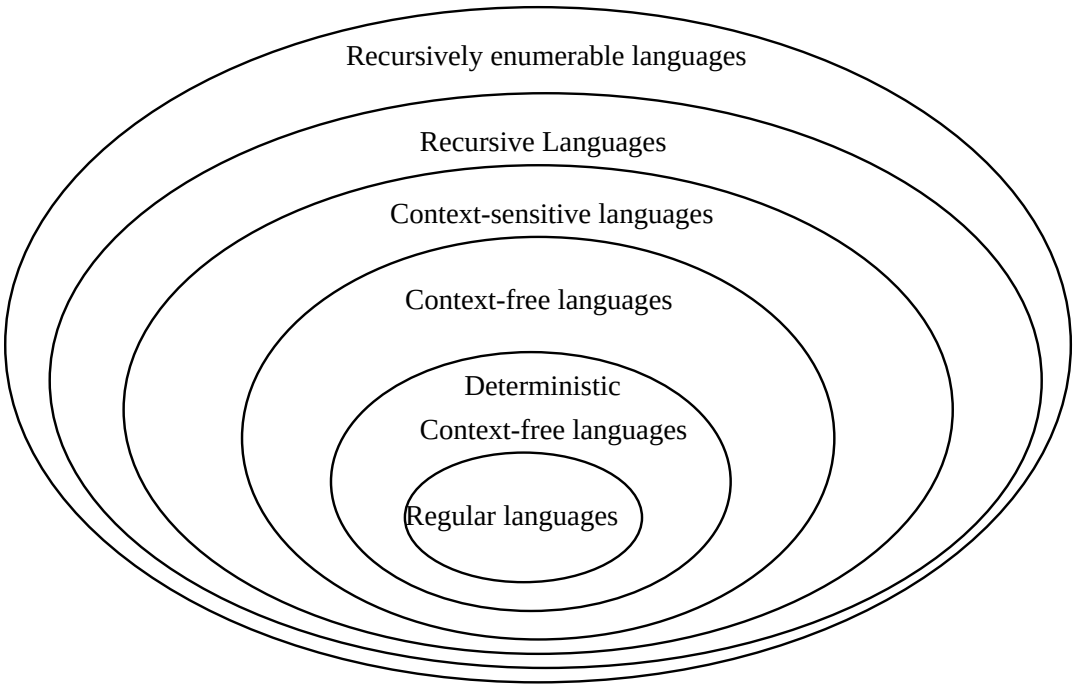


Figure 2.1. Languages Hierarchy

In addition to being a language acceptor, the Turing machine may be viewed as a computer for evaluating partial functions from Σ^* to Γ^*, where $\Gamma \subset \Sigma \cup \Omega$ is called the output alphabet (see Mal'cev [42]). Let T be a TM as defined in definition 2.1.3.1. We shall make the simplifying assumption that whenever T enters a HALT state, the TAPE will contain only symbols from Γ and δ's (i.e. the machine can, for example, erase all the symbols from $\Sigma \cup \Omega$ - Γ before it enters a HALT state). Hence T computes a function f: $\Sigma^* \rightarrow \Gamma^*$, where f(w) will be defined for an input $w \in \Sigma^*$ if and only if T accepts w and f(w) = y, and $y \in \Gamma^*$ is the string obtained from the final value of the tape by deleting all occurrences of the blank symbol δ. The set of partial functions computed by Turing machines are called *partial recursive functions*. If f is a (total) function rather than a partial one, then f is called a *total recursive function*. In a sense, the partial recursively functions are analogous to the recursive enumerable languages, since they are computed by Turing machines that may or may not halt on a given input. The total recursive functions correspond to the recursive languages, since they are computed by TMs that always halt.

The definitions above can be extended to functions

$$f: \Sigma_1^* \times ... \times \Sigma_k^* \rightarrow \Gamma^*,$$

where $\Sigma_1, ..., \Sigma_k, \Gamma$ are finite alphabets. This can be done by initialising the tape of the Turing machines with $w_1 \in \Sigma_1^*, ..., w_k \in \Sigma_k^*$ separated by δ's.

An important class of functions that can be computed in this way are functions from positive integers to positive integers,

$f: N \rightarrow N$ (or $f: N^k \rightarrow N$).

The traditional approach is to represent integers in unary (i.e. Σ contains a single element, say 1); the integer $n \geq 0$ is represented by the string 1^n.

2.1.4. Computational models as specification tools.

The usage of finite state machines is a widely followed practice, particularly in sequential hardware design (Clements [7]). In some types of software system design, they are also used, for example in user interface design (Wasserman et al. [56]). But the model is rather simple and has several drawbacks.

Firstly, it is difficult to model any *non-trivial* data structure using the finite state machine model. This makes modelling large systems with complex data structures very difficult.

Secondly, its computational power is too limited. One could argue that, in practice, any system is a finite state machine (this is because the memory used will be always finite). For example the language presented in example 2.1.1.5 will be approximated in practice by

$L_k = \{a^n cb^n | \ n < k\}$

for some sufficiently large $k \in N$. Since

$L_k = \{a^n cb^n | \ n < k\}$

is regular, it can be accepted by some finite automaton. However, if k is large, the number of states of the automaton accepting the language will be huge and such a model will become too complicated.

The Pushdown Automaton and Turing Machines are more powerful models obtained by augmenting the finite state machine model by using some form of unlimited memory. But the models are too restrictive and low level to be used as vehicles for description and analysis of serious applications. Indeed, in practice one might want to use more complex memory structures than a tape or a stack and allow more complicated functions to be performed on this memory structures. For example, imagine a simple system which uses a set of registers as its memory, the value of a register being a natural number. In this case it is natural to assume that a function which compares the values of two registers exists and can be used to build the specification of the system. Furthermore, the specification can use even more complicated functions (for example arithmetic functions such as multiplication, factorial, etc.), provided that these have been already specified. It is therefore obvious that in practice the process of building a system specification is hierarchical, i.e. a complex system is specified using simpler functions that in turn can be specified by simpler machines, rather then always getting back to the lowest level (i.e. the push's,

pop's, remove symbol from the tape, write symbol onto the tape, etc.) used by the traditional computational models. This approach can be accommodated by the X-machine model.

2.2. X-machines.

The X-machine is a natural generalisation of the computational devices presented in section 2.1. Recall that in the previous chapter we described the X-machine as consisting of a state diagram in which the transitions are labelled with a set of basic processing functions, Φ, which operate on a basic data set X. This fits exactly the description of the computational models presented above. Unlike these though, the set Φ is not restricted to a certain class of functions. Any function φ can be used as a label in the state transition diagram as long as it is computable by some procedure (one can also use non-computable functions, but this is obviously not desirable if the X-machine is to be used to model computer systems; however, the use of noncomputable φ's could be useful in areas such as biocomputing, but this beyond the scope of this chapter).

The following definition is due to Holcombe, [27].

Definition 2.2.1.
An *X-Machine* is a 10-tuple $\mathcal{M} = (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T)$, where

1. X is the *fundamental data set* that the machine operates on.
2. Y and Z are the *input* and the *output sets*, respectively.
3. α and β are the *input* and the *output relations* respectively, used to convert the input and the output sets into, and from, the fundamental set, i.e.

$$\alpha: Y \leftrightarrow X, \quad \beta: X \leftrightarrow Z$$

4. Q is the (finite) *set of states*.
5. Φ is the *type* of $\mathcal{M}$, a set of non-empty relations on X, i.e.

$$\Phi: \mathcal{P}(X \leftrightarrow X)$$

[**Note:** For a set A, $\mathcal{P}A$ is the powerset of A].
The type of the machine is the class of relations (usually partial functions) that constitute the elementary operations that the machine is capable of performing. Φ is viewed as an abstract alphabet. Φ may be infinite, but only a *finite* subset Φ' of Φ is used (this is because $\mathcal{M}$ has a *finite* number of edges despite the infinite number of labels available).

6. F is the *'next state' partial function*

$$F: Q \rightarrow (\Phi \rightarrow \mathcal{P}Q)$$

So, for state $q \in Q$, $F(q): \Phi \rightarrow \mathcal{P}Q$ is a partial function.

However, when it is convenient, F can be treated like a partial function with two arguments, i.e.

$F(q, \varphi) = (F(q))(\varphi)$.

F is often described by means of a *state-transition diagram*.

7. I and T are the sets of *initial* and *terminal states* respectively.

$I \subseteq Q, T \subseteq Q$

An example of an X-machine will be given later (see example 2.3.5). Before we continue, we give some basic definitions.

Definition 2.2.2.

If $q, q' \in Q$, $\varphi \in \Phi$ and $q' \in F(q, \varphi)$, then

$$q \xrightarrow{\varphi} q',$$

is the *arc* from q to q'.

Definition 2.2.3.

If $q, q' \in Q$ are such that there exist $q_1,..., q_n \in Q$ and $\varphi_1,..., \varphi_{n+1} \in \Phi$ with $q_1 \in F(q, \varphi_1)$, $q_2 \in F(q_1, \varphi_2)$, ..., $q_n \in F(q_{n-1}, \varphi_n)$, $q' \in F(q_n, \varphi_{n+1})$, then

$$q \xrightarrow{\varphi_1} q_1 \xrightarrow{\varphi_2} q_2 \ldots q_n \xrightarrow{\varphi_{n+1}} q'$$

is the *path* from q to q'. Each path p is labelled with $|p|$, where

$|p| = \varphi_1 \ldots \varphi_{n+1}: X \leftrightarrow X$

is the *relation computed* by the machine when it follows that path.

When the state sequence is not relevant we shall refer to a path as the sequence of relations, i.e.

$p = \varphi_1 \ldots \varphi_{n+1}$.

A *successful path* is one that starts in an initial state and ends in a terminal one.

A *loop* is a path whose initial state is also terminal (i.e. a path from a state to itself).

Definition 2.2.4.

The *behaviour* of $\mathcal{M}$ is the relation

$|\mathcal{M}|: X \leftrightarrow X$

defined as

$|\mathcal{M}| = \cup |p|$

with the union extending over all the successful paths p in $\mathcal{M}$.

Given $y \in Y$, the operations of the X-machine $\mathcal{M}$ on Y consist of:

1. Picking a path p, from a start state q_i ($q_i \in I$), to a terminal state q_t ($q_t \in T$) i.e. $q_i \xrightarrow{p} q_t$.
2. Apply α to the input to convert it to the internal type X.
3. Apply $|p|$, if it is defined for $\alpha(y)$. Otherwise, go back to step 1.
4. Apply β to get the output.

Therefore, the operation can be summarised as $\beta(|p|(\alpha(y)))$.

Definition 2.2.5.
The composite relation

$$f = Y \xleftarrow{\alpha} X \xleftarrow{|\mathcal{M}|} X \xleftarrow{\beta} Z$$

is called the *relation computed* by $\mathcal{M}$.

A deterministic X-machine is an X-machine whose type Φ is a set of partial functions rather than relations and in which there is at most one possible transition for any state q and any $x \in X$. This will always be the case in practical applications considered in this thesis.

Definition 2.2.6.
A X-machine $\mathcal{M}$ is called *deterministic* if:

1. α and β are partial functions, not relations:
   $\alpha: Y \to X, \ \beta: X \to Z$
2. Φ contains only partial functions on X rather than relations:
   $\Phi: \mathcal{P}(X \to X)$
3. F maps each pair $(q, \varphi) \in Q \times \Phi$ onto at most a single next state:
   $F: Q \to (\Phi \to Q)$

A partial function is used because each $\varphi \in \Phi$ will not necessarily be defined as the label to an edge in every state. If F is treated as a function with two arguments, then F is a partial function

$$F: Q \times \Phi \to Q.$$

4. I contains only one element (i.e. $I = \{q_0\}$, where $q_0 \in Q$).
5. If $q \xrightarrow{\varphi} p$ and $q \xrightarrow{\varphi'} p'$ are distinct arcs emerging from the same state q, then

$$\text{dom } \varphi \cap \text{dom } \varphi' = \varnothing.$$

Note: dom φ denotes the domain of φ.

If we consider F as a function with two arguments, the condition 5 can be written as:

5'. $\forall q \in Q, \varphi, \varphi' \in \Phi$, if $(q, \varphi), (q, \varphi') \in$ dom F then
$\text{dom } \varphi \cap \text{dom } \varphi' = \varnothing.$

The input and output sets will almost always comprise sequences of symbols from some alphabets Σ and Γ. Therefore $Y = \Sigma^*$, $Z = \Gamma^*$, where Σ is called the *input alphabet* and Γ the *output alphabet*. Thus relations

$$f: \Sigma^* \leftrightarrow \Gamma^*$$

will be computed. The set X will have the form

$$X = \Gamma^* \times M \times \Sigma^*,$$

where M is a monoid called *memory*. The first component of the above cartesian product (i.e. Γ^*) will be called the *output register*. The last component (i.e. Σ^*) will be called the *input register*. In what follows we shall only be referring to X-machines having this form.

If $\mathcal{M}$ is a X-machine acceptor (i.e. $\Gamma = \varnothing$, hence $\Gamma^* = \{1\}$, where 1 is the empty sequence), then

$X = M \times \Sigma^*$

and $\mathcal{M}$ will compute a function f with only one output value, i.e.

$$f(x) = \begin{cases} c, \text{ if } x \in \text{dom } f \\ \varnothing, \text{ otherwise} \end{cases}$$

with $c = \beta(1)$.

We call L = dom f the *language accepted* by the machine.

Before we continue, we introduce the following notation:

Notation 2.2.7.
Given an X-machine $\mathcal{M}$ with $X = \Gamma^* \times M \times \Sigma^*$ we define the following functions:

Out: $X \to \Gamma^*$, Out(g, m, s) = g
In: $X \to \Sigma^*$, In(g, m, s) = s
Mem: $X \to M$, Mem(g, m, s) = m

$\forall\, g \in \Gamma^*, m \in M, s \in \Sigma^*$.

If $\mathcal{M}$ is a machine acceptor (i.e. $\Gamma = \varnothing$), then In: $X \to \Sigma^*$ and Mem: $X \to M$ are defined by

In(m, s) = s, Mem(m, s) = m.

If $\mathcal{M}$ is a deterministic X-machine, one would expect $\mathcal{M}$ to compute a partial function $f: \Sigma^* \to \Gamma^*$ rather than a relation. However, this is not always the case, an additional condition being required.

Definition 2.2.8.
A path $|p| = \varphi_1 \ldots \varphi_{n+1}: X \to X$ is called *trivial*, if $\exists\, x \in \text{dom } |p|$ such that

In(|p|(x)) = In(x).

In other words, a trivial path is one along which the machine does not change the value of the input register for some values of X, while possibly changing the output register and memory.

Then, we have the following straightforward result (see Eilenberg [12]):

Proposition 2.2.9.
If $\mathcal{M}$ is a deterministic X-machine in which no trivial path connects two terminal states, then $\mathcal{M}$ computes a partial function.

The model presented so far is slightly too general and we now consider two natural classes of these machines.

2.3. Straight move stream X-machines.

Consider a program (or a hardware device) that receives some external inputs from an alphabet Σ and produces some outputs from an alphabet Γ. The program will have a memory structure M and will consist of a set of 'basic' moves or instructions performing one of the following types of 'basic' operations:

- · update the memory;
- · update the memory and produce an output $\gamma \in \Gamma$.
- · read an input $\sigma \in \Sigma$ and update the memory.
- · read an input $\sigma \in \Sigma$, update the memory and produce an output $\gamma \in \Gamma$.

Also, a special input symbol that indicates when the program has finished reading all the necessary inputs might be required. We denote this by δ and we call it the *blank* or *end marker*. For the sake of simplicity we shall consider that $\delta \notin \Sigma$.

Obviously, more than one output can be produced at one time, but this can be dealt with by a convenient augmentation of the output set Γ (i.e. we replace Γ with the appropriate $\Gamma' \subseteq \Gamma^*$).

Therefore, it looks as though a very wide class of applications can be represented in the way described above. Let us translate this model into an X-machine. First, we introduce some notation that we shall be needing in what follows:

Definition 2.3.1.
Let Σ be a set. Then we define the functions

$$\text{head}: \Sigma^* \to \Sigma^*, \text{front}: \Sigma^* \to \Sigma^*, \text{tail}: \Sigma^* \to \Sigma^*, \text{rear} : \Sigma^* \to \Sigma^*$$

by:

$\text{head}(\sigma s) = \sigma$, $\forall \sigma \in \Sigma$, $s \in \Sigma^*$; $\text{head}(1) = 1$;
$\text{front}(s\sigma) = s$, $\forall \sigma \in \Sigma$, $s \in \Sigma^*$; $\text{front}(1) = 1$;
$\text{tail}(\sigma s) = s$, $\forall \sigma \in \Sigma$, $s \in \Sigma^*$; $\text{tail}(1) = 1$;
$\text{rear}(s\sigma) = \sigma$, $\forall \sigma \in \Sigma$, $s \in \Sigma^*$; $\text{rear}(1) = 1$.

Note: If $s \in \Sigma^*$, $t \in \Sigma^*$,st is s concatenated with t.

We can now define formally the particular type of X-machine described at the beginning of this section.

Definition 2.3.2.
Let Σ and Γ two alphabets, and let $\delta \notin \Sigma \cup \Gamma$ and $\Sigma' = \Sigma \cup \{\delta\}$. Then, an X-machine $\mathcal{M}$ with

$$X = \Gamma^* \times M \times \Sigma'^*$$

is called a *straight-move stream X-machine* (denoted *SMS X-machine)* if:

1. The input and output codes

$$\alpha: \Sigma^* \to X, \quad \beta: X \to \Gamma^*$$

are defined by

$$\alpha(s) = (1, m_0, s\delta), \forall\, s \in \Sigma^*,$$

$$\beta(g, m, s) = \begin{cases} g, & \text{if } s = 1 \\ \varnothing, & \text{otherwise} \end{cases}$$

$$\forall\, g \in \Gamma^*,$$

where $m_0 \in M$ is called the *initial memory value*.

2. The type is

$$\Phi = \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \Phi_4,$$

where

i) $\forall\, \varphi_1 \in \Phi_1$, $\varphi_1: X \leftrightarrow X$ is a relation of the form:

$$\varphi_1(g, m, s) = \begin{cases} (g\, \rho_1(m, \text{head}(s)), \mu_1(m, \text{head}(s)), \text{tail}(s)), & \text{if } s \neq 1 \\ \varnothing, & \text{otherwise} \end{cases}$$

where

$$\mu_1: M \times \Sigma' \leftrightarrow M,\ \rho_1: M \times \Sigma' \leftrightarrow \Gamma \text{ are relations;}$$

i.e. any $\varphi_1 \in \Phi_1$ reads the head of the input string (possibly δ) and adds an output character to the end of the output string while updating the memory. Φ_1 is called the set of *non-empty input and non-empty output* operations.

ii) $\forall\, \varphi_2 \in \Phi_2$, $\varphi_2: X \leftrightarrow X$ is a relation of the form:

$$\varphi_2(g, m, s) = \begin{cases} (g, \mu_2(m, \text{head}(s)), \text{tail}(s)), & \text{if } s \neq 1 \\ \varnothing, & \text{otherwise} \end{cases}$$

where

$$\mu_2: M \times \Sigma' \leftrightarrow M \text{ is a relation;}$$

i.e. any $\varphi_2 \in \Phi_2$ reads the head of the input string (possibly δ) and leaves the output string unchanged while updating the memory. Φ_2 is called the set of *non-empty input and empty output* operations.

iii) $\forall \varphi_3 \in \Phi_3$, $\varphi_3: X \leftrightarrow X$ is a relation of the form:

$$\varphi_3(g, m, s) = (g\, \rho_3(m), \mu_3(m), s),$$

where

$\mu_3: M \leftrightarrow M$, $\rho_3: M \leftrightarrow \Gamma$ are relations;

i.e. any $\varphi_3 \in \Phi_3$ leaves the input string unchanged and adds an output character to the end of the output string while updating the memory. Φ_3 is called the set of *empty input and non-empty output* operations.

iv) $\forall \varphi_4 \in \Phi_4$, $\varphi_4: X \leftrightarrow X$ is a relation of the form:

$$\varphi_4(g, m, s) = (g, \mu_4(m), s),$$

where

$\mu_4: M \leftrightarrow M$ is a relation;

i.e. any $\varphi_4 \in \Phi_4$ leaves both the input and output strings unchanged while updating the memory. Φ_4 is called the set of *empty input and empty output* operations.

3. We further assume that $\forall q \in T$, $\forall \varphi \in \Phi_3 \cup \Phi_4$ then $(q, \varphi) \notin \text{dom } F$. Therefore, no empty input transition is allowed from a terminal state.

A SMS X-machine will be denoted as a tuple $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, Q_0, T, m_0)$.

Basically, a SMS X-machine can process the head of the input string or no input at all and add one output character or no output at all at the end of the output string. Furthermore, no transition is allowed to use information from the tail of the input or any of the output. It is fairly clear that a SMS X-machine computes a relation

$f: \Sigma^* \leftrightarrow \Gamma^*$, $f = \alpha\, |\mathcal{M}|\, \beta$.

Observation 2.3.3.

If $\mathcal{M}$ is a SMS X-machine acceptor (i.e. $\Gamma = \varnothing$) then

$$X = M \times \Sigma^*$$

and the type can be written as

$$\Phi = \Phi_1' \cup \Phi_2',$$

where

i) $\forall \varphi_1' \in \Phi_1', \varphi_1': X \leftrightarrow X$ is a relation of the form:

$$\varphi_1'(g, m, s) = \begin{cases} (\mu_1'(m, \text{head}(s)), \text{tail}(s)), & \text{if } s \neq 1 \\ \varnothing, & \text{otherwise} \end{cases}$$

where

$\mu_1': M \times \Sigma' \leftrightarrow M$ is a relation;

ii) $\forall \varphi_2' \in \Phi_2', \varphi_2' : X \leftrightarrow X$ is a relation of the form:

$\varphi_2'(g, m, s) = (\mu_2'(m), s)$,

where

μ_2': $M \leftrightarrow M$ is a relation.

From condition 3 of definition 2.3.2, it follows that no trivial paths start from a terminal state in a deterministic SMS X-machine. Hence we have the following result.

Proposition 2.3.4.
Any deterministic straight-move stream X-machine computes a partial function
$f: \Sigma^* \rightarrow \Gamma^*$.

In practice Σ and Γ will be almost always finite. However, our model does not make any such assumption. A simple SMS X-machine example is given below.

Example 2.3.5.
Let $\Sigma = \{a, b\}$, $\Gamma = \{x, y\}$. Then $\mathcal{M}$ is a SMS X-machine defined as follows.

1. The set of states is

$Q = \{q_0, q_1, q_2, q_3\}$.

q_0 is the initial state; q_3 is the terminal state.

2. The memory is the set of positive integers, M = N. Therefore, the fundamental data set is

$X = \Gamma^* \times N \times \Sigma'^*$,

where $\Sigma' = \Sigma \cup \{\delta\}$. The initial memory value is $m_0 = 0$.

3. $\Phi = \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \Phi_4$

with

$\Phi_1 = \{\varphi_2\}$, $\Phi_2 = \{\varphi_1, \varphi_5\}$, $\Phi_3 = \{\varphi_4\}$, $\Phi_4 = \{\varphi_3\}$,

where

$\varphi_1, \varphi_1, \varphi_3, \varphi_4, \varphi_5: \Gamma^* \times N \times \Sigma'^* \rightarrow \Gamma^* \times N \times \Sigma'^*$

are partial functions defined as follows:

$\text{dom } \varphi_1 = \Gamma^* \times N \times \{a\}\Sigma'^*$,
$\varphi_1(g, n, as) = (g, n+1, s)$, $\forall\ g \in \{x, y\}^*, n \in N, s \in \{a, b, \delta\}^*$;

$\text{dom } \varphi_2 = \Gamma^* \times N \times \{b\}\Sigma'^*$,
$\varphi_2(g, n, bs) = (gy, n+1, s)$, $\forall\ g \in \{x, y\}^*, n \in N, s \in \{a, b, \delta\}^*$;

dom $\varphi_3 = \Gamma^* \times N \times \Sigma'^*$,
$\varphi_3(g, n, s) = (g, n+1, s), \forall\ g \in \{x, y\}^*, n \in N, s \in \{a, b, \delta\}^*$;

dom $\varphi_4 = \Gamma^* \times (N - \{0\}) \times \Sigma'^*$,
$\varphi_4(g, n, s) = (gx, n-1, s), \forall\ g \in \{x, y\}^*, n \in N - \{0\}, s \in \{a, b, \delta\}^*$;

dom $\varphi_5 = \Gamma^* \times \{0\} \times \{\delta\}\Sigma'^*$,
$\varphi_5(g, 0, \delta s) = (g, 0, s), \forall\ g \in \{x, y\}^*, s \in \{a, b, \delta\}^*$.

4. The next state function F follows from the diagram in figure 2.2.

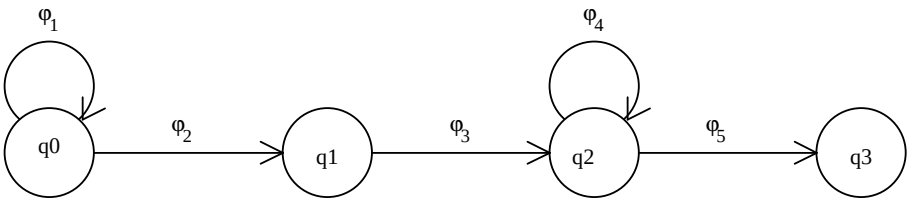


Figure 2.2.

It is clear that $\mathcal{M}$ is deterministic and it computes a partial function

$f: \{a, b\} \rightarrow \{x, y\}$

defined by

dom $f = \{a^n b \mid n \in N\}$ and $f(a^n b) = yx^{n+1}, \forall\ n \in N$.

At the beginning of this section we claimed the SMS X-machine could be used as a general model of a program or of a hardware device. We can now support this claim by showing that the computational models presented in section 2.1 are all particular types of SMS X-machines. We show that finite state machines, pushdown automata and Turing machines can be described as SMS X-machines. First, introduce some notation that we shall be using later.

Definition 2.3.6.
Let Σ be an alphabet and let $s \in \Sigma^*$. Then we define the functions $L_s, R_s: \Sigma^* \rightarrow \Sigma^*$ by

$L_s(x) = sx, \qquad R_s(x) = xs, \forall\ x \in \Sigma^*$

and the partial functions $L_{-s}, R_{-s}: \Sigma^* \rightarrow \Sigma^*$ by

$L_{-s}(x) = s^{-1}x$ (i.e. dom $L_{-s} = \{s\}\Sigma^*$ and $L_{-s}(sx) = x, \forall\ x \in \Sigma^*$)

$R_{-s}(x) = xs^{-1}$ (i.e. dom $R_{-s} = \Sigma^*\{s\}$ and $R_{-s}(xs) = x, \forall\ x \in \Sigma^*$)

Obviously, $L_s L_t = L_{ts}$, $R_s R_t = R_{st}$, $L_{-s} L_{-t} = L_{-st}$, $R_{-s} R_{-t} = R_{-ts}, \forall\ s, t \in \Sigma^*$.
We shall denote by $I: \Sigma^* \rightarrow \Sigma^*$ the identity function.

Notation 2.3.7.
Let $\mathcal{M}$ be an X-machine with $X = \Gamma^* \times M \times \Sigma^*$ and $M = A_1 \times ... \times A_k$ (k possibly 0). Let also ϕ_Γ: $\Gamma^* \to \Gamma^*$, ϕ_Σ: $\Sigma^* \to \Sigma^*$ and ϕ_i: $A_i \to A_i$, i = 1, ..., k be (partial) functions. Then we denote by

$$\varphi = (\phi_\Gamma, \phi_1, ..., \phi_k, \phi_\Sigma)$$

the (partial) function φ: $X \to X$ defined by:

$$\varphi(g, a_1, ..., a_k, s) = (\phi_\Gamma(g), \phi_1(a_1), ..., \phi_k(a_k), \phi_\Sigma(s))$$

$\forall\ g \in \Gamma^*, s \in \Sigma^*, a_1 \in A_1, ..., a_k \in A_k$
(i.e. $\varphi(g, a_1, ..., a_k, s)$ is defined iff $\phi_\Gamma(g)$, $\phi_1(a_1)$, ..., $\phi_k(a_k)$ and $\phi_\Sigma(s)$ are all defined).
Also, let Φ_Γ be a set of (partial) functions from Γ^* to Γ^*, Φ_Σ be a set of (partial) functions from Σ^* to Σ^* and Φ_i sets of (partial) functions from A_i to A_i, i = 1, ..., k. Then

$$\Phi_\Gamma \times \Phi_1 \times ... \times \Phi_k \times \Phi_\Sigma$$

denotes the set

$$\{\varphi = (\phi_\Gamma, \phi_1, ..., \phi_k, \phi_\Sigma) |\ \phi_\Gamma \in \Phi_\Gamma, \phi_\Sigma \in \Phi_\Sigma, \phi_i \in \Phi_i, i = 1, ..., k\}.$$

If $\mathcal{M}$ is an X-machine acceptor (i.e. $\Gamma = \emptyset$), then we denote by

$$\varphi = (\phi_1, ..., \phi_k, \phi_\Sigma)$$

the (partial) function φ: $X \to X$ defined by:

$$\varphi(a_1, ..., a_k, s) = (\phi_1(a_1), ..., \phi_k(a_k), \phi_\Sigma(s)),\ \forall\ s \in \Sigma^*, a_1 \in A_1, ..., a_k \in A_k.$$

Also

$$\Phi_1 \times ... \times \Phi_k \times \Phi_\Sigma$$

denotes the set

$$\{\varphi = (\phi_1, ..., \phi_k, \phi_\Sigma) |\ \phi_\Sigma \in \Phi_\Sigma, \phi_i \in \Phi_i, i = 1, ..., k\}.$$

A. Finite state machines

Let $M = \emptyset$, $\Gamma = \emptyset$, $X = \Sigma'^*$.
The type is given by

$$\Phi = \Phi_1' = \{L_{-\sigma} |\ \sigma \in \Sigma'\}.$$

If we consider finite state machines with outputs ($\Gamma \neq \emptyset$), then

$$X = \Gamma^* \times \Sigma'^*$$

and the type becomes

$$\Phi = \Phi_1 = \{R_\gamma |\ \gamma \in \Gamma\} \times \{L_{-\sigma} |\ \sigma \in \Sigma'\}.$$

B. Pushdown automata

Let $M = \Omega^*$, $\Gamma = \emptyset$, $X = \Omega^* \times \Sigma^*$, $m_0 = \Delta$, where Δ is the bottom of stack character (we consider that $\Delta \in \Omega$).

The type is given by

$$\Phi = \Phi_1' \cup \Phi_2',$$

with

$\Phi_1' = \{I\} \times \{L_{-\sigma} | \sigma \in \Sigma'\}$ and
$\Phi_2' = \{R_{-\omega} | \omega \in \Omega\} \times \{I\} \cup \{R_{\omega} | \omega \in \Omega\} \times \{I\}$.

The model can be generalised by considering SMS X-machine models whose memory structure is a stack or a finite set of stacks, the basic operations on stacks being the usual 'push' and 'pop'.

Definition 2.3.8.
Let $\mathcal{m} = (\Sigma, \Gamma, Q, M, \Phi, F, q_o, T, m_o)$ be a straight move stream X-machine with Σ and Γ finite. If

1. $M = \Omega_1^* \times ... \times \Omega_k^*$ (hence $X = \Gamma^* \times \Omega_1^* \times ... \times \Omega_k^* \times \Sigma'^*$,

where $\Omega_1, ... \Omega_k$ are finite alphabets.

2. $\Phi = \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \Phi_4$

with

$\Phi_1 = \{R_{\gamma} | \gamma \in \Gamma\} \times \Phi_1'' \times ... \times \Phi_k'' \times \{L_{-\sigma} | \sigma \in \Sigma'\}$,
$\Phi_2 = \{I\} \times \Phi_1'' \times ... \times \Phi_k'' \times \{L_{-\sigma} | \sigma \in \Sigma'\}$,
$\Phi_3 = \{R_{\gamma} | \gamma \in \Gamma\} \times \Phi_1'' \times ... \times \Phi_k'' \times \{I\}$,
$\Phi_4 = \{I\} \times \Phi_1'' \times ... \times \Phi_k'' \times \{I\}$,

where

$\Phi_i'' = \{R_{-\omega} | \omega \in \Omega_i\} \cup \{R_{\omega} | \omega \in \Omega_i\} \cup \{I\}$, i = 1, ..., k.

then $\mathcal{m}$ is called a *k-stack straight-move stream X-machine* (denoted *k-stack SMS X-machine*).

Obviously, a PDA is a 1-stack SMS X-machine acceptor.

Note: We can consider that Φ_i also includes a partial function

$E: \Omega_i^* \rightarrow \Omega_i^*$

that checks whether the stack is empty or not, i.e. E is defined by

dom E = {1} and E(1) = 1.

This follows since $E = R_{-\Delta} R_{\Delta}$, where Δ is the bottom of stack character.

C. Turing machines

This case is more complicated since the Turing machine uses only one tape both as an input and output device and moves in both directions are allowed. However, a Turing machine can be simulated by a 2-stack SMS X-machine.

Theorem 2.3.9.
Let Σ and Γ be two finite alphabets and let $f: \Sigma^* \rightarrow \Gamma^*$ be a partial recursive function. There exists then a deterministic 2-stack straight-move stream X-machine $\mathcal{m}$ which computes f.

Proof:
If f is recursively enumerable, then there exists a Turing machine $\mathcal{J}$ with the state set

$Q = \{q_1,..., q_n\}$

(q_1 is the Start state), $\Omega_{\mathcal{J}}$ the set of tape symbols, which computes f. Hence, if t is the initial value of the tape and t' its end value, then

$Rmb(t') = f(t)$,

where

$Rmb: (\Gamma \cup \{\delta\})^* \rightarrow \Gamma^*$

is a function which removes all occurrences of the blank symbol δ from the tape.

Any transition of $\mathcal{J}$ can be described as

$(q, a) \rightarrow (p, a', d)$,

where q is the state $\mathcal{J}$ currently is in, a the character read, p the next state, a' the replacement character and $d \in \{L, R\}$ is the direction the tape head moves in.

We can now simulate the Turing machine $\mathcal{J}$ on the following 2 stack SMS X-machine $\mathcal{M}$:

1. The set of states is

   $Q' = \{q_1', q_1'', ..., q_n', q_n''\} \cup Q''$.

The states set of $\mathcal{M}$ is obtained by duplicating each state from Q and adding some extra states (viz. Q''). The set Q'' will explicitly follow from the construction of $\mathcal{M}$. $\mathcal{M}$ will be in the state q_i', i = 1, ..., n if $\mathcal{J}$ is in the state q_i and it has not read a blank (δ) from the tape (therefore the Turing machine has not finished reading the input sequence); $\mathcal{M}$ will be in the state q_i'', i = 1, ..., n, if $\mathcal{J}$ is in the state q_i and it has read a blank (δ) from the tape (the Turing machine has read the whole input sequence).

2. The initial state is q_1' and the set of terminal states is

   $T' = \{q_i'' | q_i$ is a Halt state of $\mathcal{J}\}$.

3. The memory is

   $M = \Omega^* \times \Omega^*$, where $\Omega = \Sigma \cup \Omega_{\mathcal{J}}'$ and $\Omega_{\mathcal{J}}' = \Omega_{\mathcal{J}} \cup \{\delta\}$.

The values of the two stacks s and s' will hold the tape of the Turing machine up to the rightmost location of the tape that has been read by the tape head, i.e. if

$t = a_1 ...a_j$, $(a_1,..., a_j \in \Omega)$,

is the tape up to the rightmost location that has been read by the head tape and i is the current position of the tape head, $i \leq j$, then

$s = a_1 ...a_{i-1}$, $s' = a_j ...a_i$.

Hence t = s rev(s'), where rev(x) denotes the reverse of the string x.

The initial value of the memory is $m_0 = (1, 1)$.

4. F results by the following simulation of $\mathcal{J}$ on $\mathcal{M}$:

a. For a transition in $\mathcal{J}$ of the form

$(q, \sigma) \rightarrow (p, b, R)$, $\sigma \in \Sigma$, $b \in \Omega_{\mathcal{J}}$',

the corresponding transitions in $\mathcal{M}$ are:

$F(q', \varphi_1) = p'$, $F(q', \varphi_2) = p'$, $F(q'', \varphi_2) = p''$,

where

$\varphi_1 = (I, R_b, E, L_{-\sigma})$, $\varphi_2 = (I, R_b, R_{-\sigma}, I)$.

Therefore, if $\mathcal{J}$ has not finished reading the input string ($\mathcal{M}$ is in state q') and s' = 1 then $\mathcal{M}$ reads a new input character. Otherwise, no input is read and $\mathcal{M}$ only operates on its stacks.

b. For a transition in $\mathcal{J}$ of the form

$(q, a) \rightarrow (p, b, R)$, $a \in \Omega_{\mathcal{J}} - \Sigma$, $b \in \Omega_{\mathcal{J}}$',

the corresponding transition in $\mathcal{M}$ is:

$F(q'', \varphi_3) = p''$,

where

$\varphi_3 = (I, R_b, R_{-a}, I)$.

Since a is not an input character, $\mathcal{J}$ has finished reading the input string; therefore $\mathcal{M}$ operates only on its stacks.

c. For a transition in $\mathcal{J}$ of the form

$(q, \delta) \rightarrow (p, b, R)$, $b \in \Omega_{\mathcal{J}}$',

the corresponding transitions in $\mathcal{M}$ are:

$F(q', \varphi_4) = p''$, $F(q', \varphi_5) = p'$, $F(q'', \varphi_5) = p''$,

where

$\varphi_4 = (I, R_b, E, L_{-\delta})$, $\varphi_5 = (I, R_b, R_{-\delta}, I)$.

Therefore $\mathcal{M}$ can read the end marker of the input string only if a δ has not yet been read. Otherwise, the machine operates only on its stacks.

d. The transitions

$(q, \sigma) \rightarrow (p, b, L)$, $(q, a) \rightarrow (p, b, L)$, $(q, \delta) \rightarrow (p, b, L)$,

$\sigma \in \Sigma$, $a \in \Omega_{\mathcal{J}} - \Sigma$, $b \in \Omega_{\mathcal{J}}$',

can be obtained from those above by replacing φ_i, i = 1, ..., 5 by $\varphi_i' = \varphi_i \, Tf^2$, where Tf is the function which transfers any character from the first stack to the second. Such transitions can be transformed into a sequence of three SMS X-machine operations by adding two new states $r_1, r_2 \in Q''$ for each transition. For example

$F(q', \varphi_1 Tf^2) = p'$

is equivalent to

$F(q', \varphi_1) = r_1$, $F(r_1, \psi) = r_2$, $F(r_2, \psi) = p'$,

$\forall \psi \in \{(I, R_{-a}, R_a, I) | \, a \in \Omega\}$ (i.e. ψ takes all the values of the set $\{(I, R_{-a}, R_a, I) | \, a \in \Omega\}$).

In order to complete our construction, we have to address the following two problems:

e. $\mathcal{M}$ has to read the entire input sequence even if $\mathcal{I}$ halts earlier. This can be easily addressed by adding an extra state $r_i' \in Q''$ for each $i \in \{1, ..., n\}$ such that q_i is a Halt state of $\mathcal{I}$, and the following transitions:

$F(q_i', \xi_1) = q_i'$, $F(q_i', \xi_2) = r_i'$, $F(r_i', \xi_3) = r_i'$, $F(r_i', \xi_4) = q_i''$, i = 1,..., n,

$\forall\ \xi_1 \in \{(I, R_a, R_{-a}, I) | a \in \Omega\}$ and $\xi_3 \in \{(I, R_\sigma, I, L_{-\sigma}) | \sigma \in \Sigma\}$,

where

$\xi_2 = (I, I, E, I)$, $\xi_4 = (I, R_\delta, I, L_{-\delta})$.

Therefore if $\mathcal{I}$ has halted without finishing reading the input string, $\mathcal{M}$ will store the part of the tape already read in the first stack, read the remaining part (until a δ is reached) and store the remaining part of the tape into the first stack. Since no path can leave a Halt state in $\mathcal{I}$, $\mathcal{M}$ remains deterministic.

f. So far $\mathcal{M}$ does not produce any outputs. Therefore, any transition of the type

$F(q, \varphi) = q_i''$,

where $q \in Q'$ (Q' is the state set of $\mathcal{M}$ constructed so far) and $q_i'' \in T'$ (i.e. q_i'' is a terminal state) has to be replaced by

$F(q, \varphi GH) = q_i''$,

where G stores s' rev(s) into s' (therefore s' will hold the reverse of the tape value t), where s and s' are the values of the two stacks, and H outputs Rmb(rev(s')) (i.e. the string obtained by erasing all the blanks from the tape t). This can be achieved by adding 2 extra states $r_1'', r_2'' \in Q''$ for each $q_i'' \in T'$ and replacing each transition of the type $F(q, \varphi) = q_i''$ with the following sequence of transitions:

$F(q, \varphi) = r_1''$, $F(r_1'', \zeta_1) = r_1''$, $F(r_1'', \zeta_2) = r_2''$, $F(r_2'', \zeta_3) = r_2''$,

$F(r_2'', \zeta_4) = r_2''$, $F(r_2'', \zeta_5) = q_i''$,

$\forall\ \zeta_1 \in \{(I, R_{-a}, R_a, I) | a \in \Omega\}$, $\zeta_4 \in \{(R_\gamma, I, R_{-\gamma}, I) | \gamma \in \Gamma\}$,

where

$\zeta_2 = (I, E, I, I)$, $\zeta_3 = (I, I, R_{-\delta}, I)$, $\zeta_5 = (I, I, E, I)$.

From the construction above it is clear that $f = \alpha|\mathcal{M}|\beta$. Therefore $\mathcal{M}$ computes f. ⑥

Example 2.3.10.

Let $\mathcal{I}$ be the Turing machine described in figure 2.3, where:

$\Sigma = \{0, 1\}$, $\Gamma = \{0, 1\}$, $\Omega = \{0, 1\}$,

q_0 is the initial state and q_2 the Halt state.

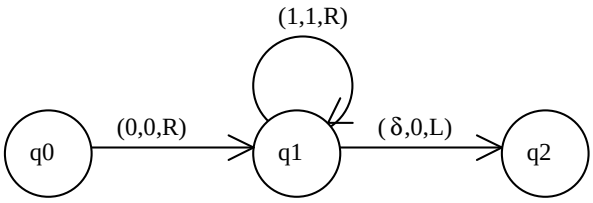


Figure 2.3.

$\mathcal{J}$ computes the partial function f: $\Sigma^* \rightarrow \Gamma^*$ with

dom f = $\{01^k | k \in N\}$

defined by

$f(01^k) = 01^k0$, $k \geq 0$.

By applying the transformations given by 4.a - d, 4.e and 4.f, $\mathcal{J}$ is transformed successively as described in figures 2.4, 2.5 and 2.6 respectively.

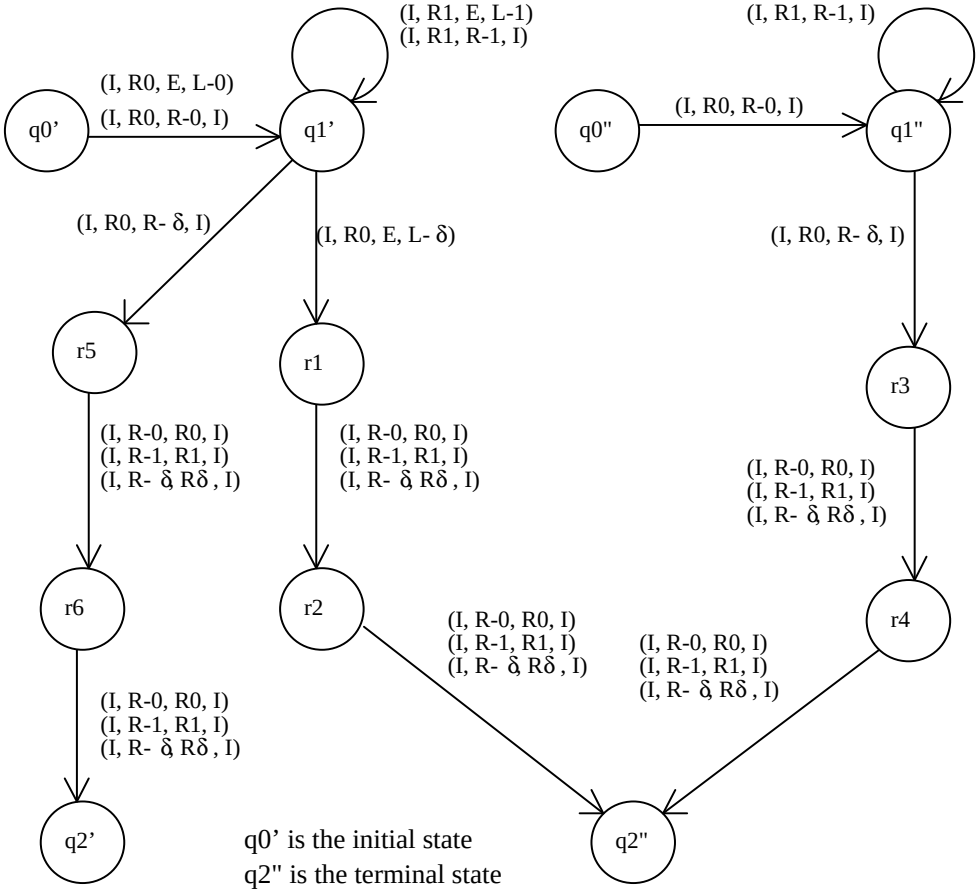


Figure 2.4.

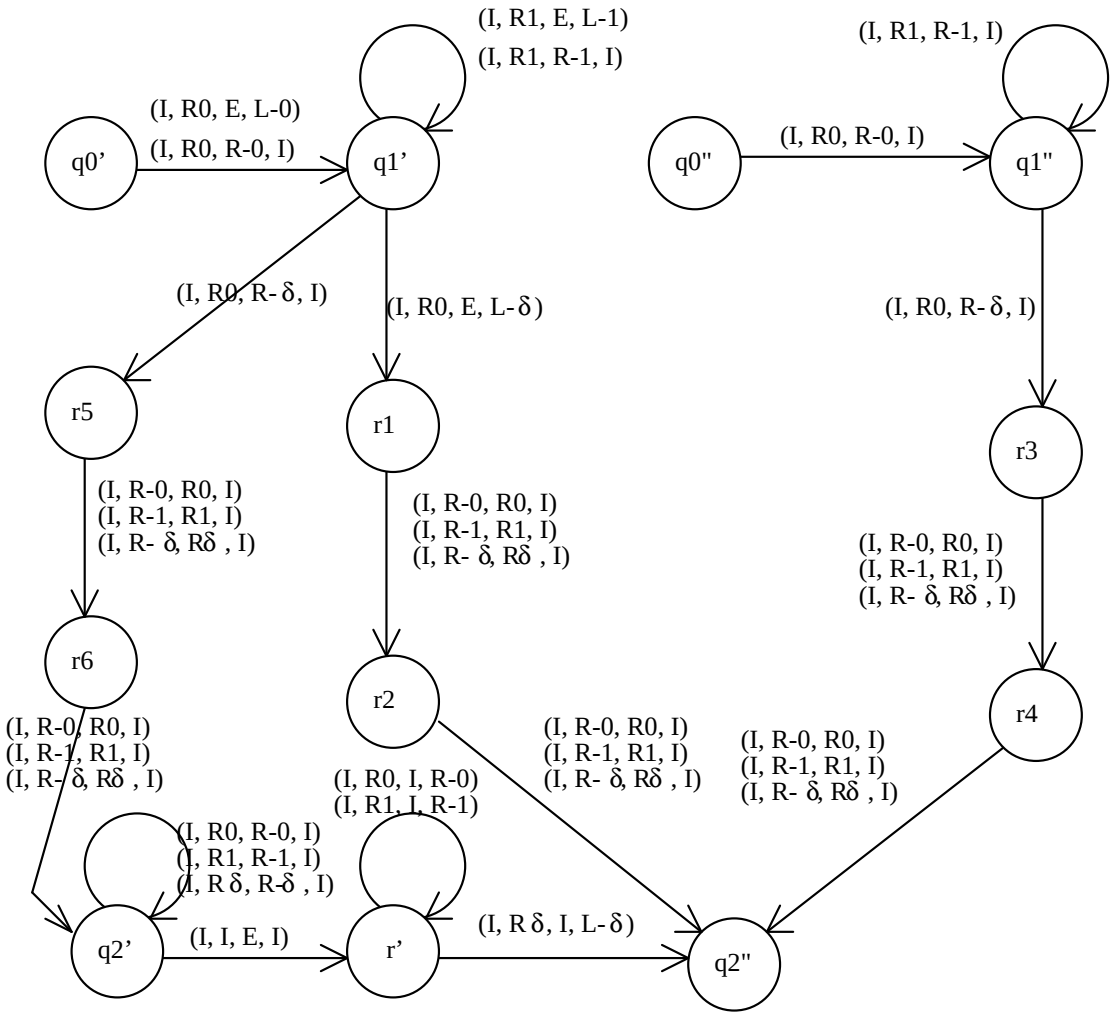

(I, R1, E, L-1)
(I, R1, R-1, I)
(I, R1, R-1, I)
(I, R0, E, L-0)
(I, R0, R-0, I)
(I, R0, R-0, I)
q0'
q1'
q0"
q1"
(I, R0, R- δ, I)
(I, R0, E, L- δ)
(I, R0, R- δ, I)
r5
r1
r3
(I, R-0, R0, I)
(I, R-1, R1, I)
(I, R- δ, Rδ , I)
(I, R-0, R0, I)
(I, R-1, R1, I)
(I, R- δ, Rδ , I)
(I, R-0, R0, I)
(I, R-1, R1, I)
(I, R- δ, Rδ , I)
r6
r2
r4
(I, R-0, R0, I)
(I, R-1, R1, I)
(I, R- δ, Rδ , I)
(I, R-0, R0, I)
(I, R-1, R1, I)
(I, R- δ, Rδ , I)
(I, R-0, R0, I)
(I, R-1, R1, I)
(I, R- δ, Rδ , I)
(I, R0, I, R-0)
(I, R1, I, R-1)
(I, R0, R-0, I)
(I, R1, R-1, I)
(I, R δ, R-δ , I)
q2'
(I, I, E, I)
r'
(I, R δ, I, L- δ)
q2"

Figure 2.5.

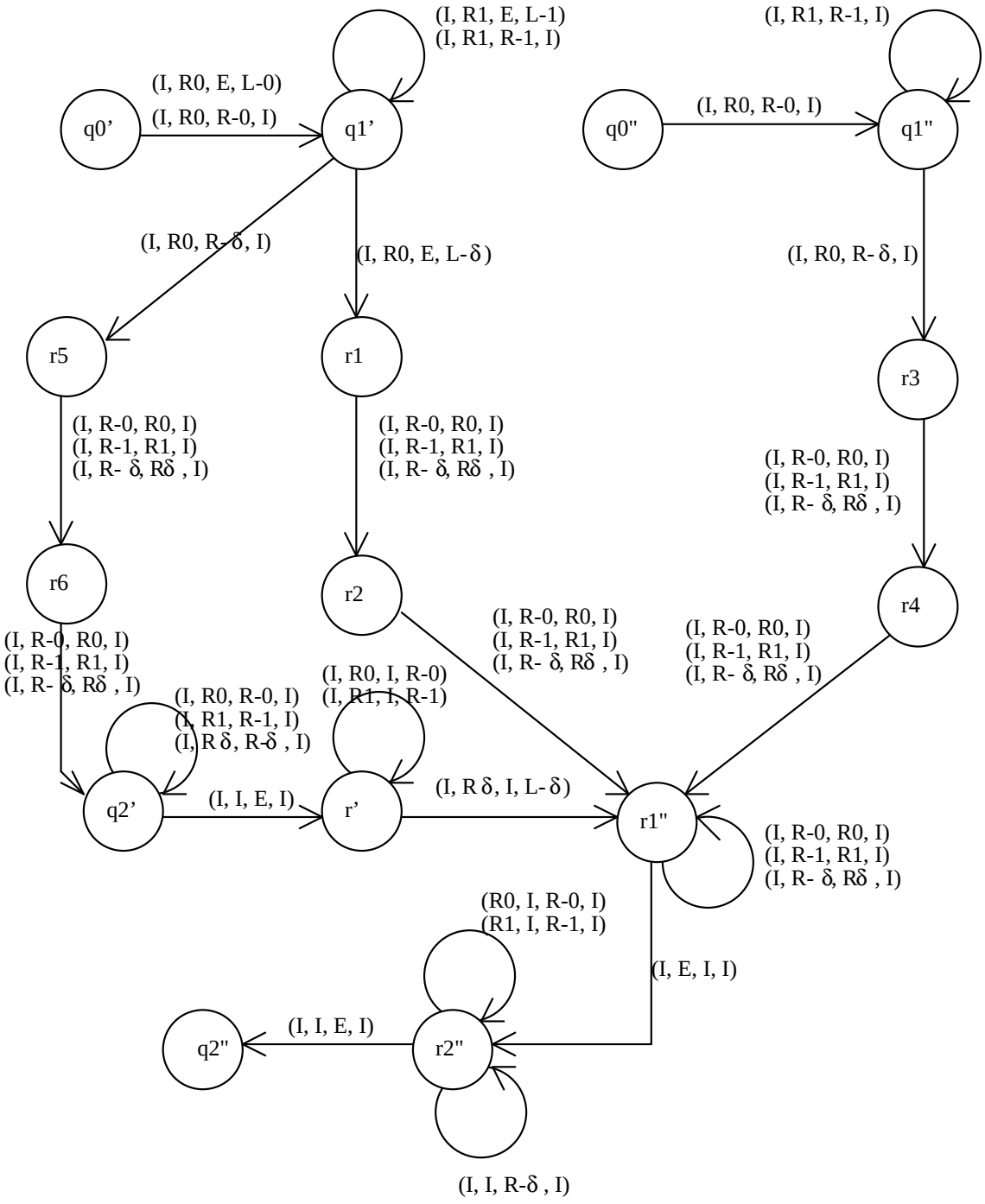


Figure 2.6.

Since q_0", q_1", r_3 and r_4 are not connected to the initial state q_0', they can be deleted together with the arcs that emerge from them or leave them. Hence, the 2-stack SMS X-machine obtained has the 'state transition' diagram given in figure 2.7, where q_0' is the initial state and q_2" is the terminal state.

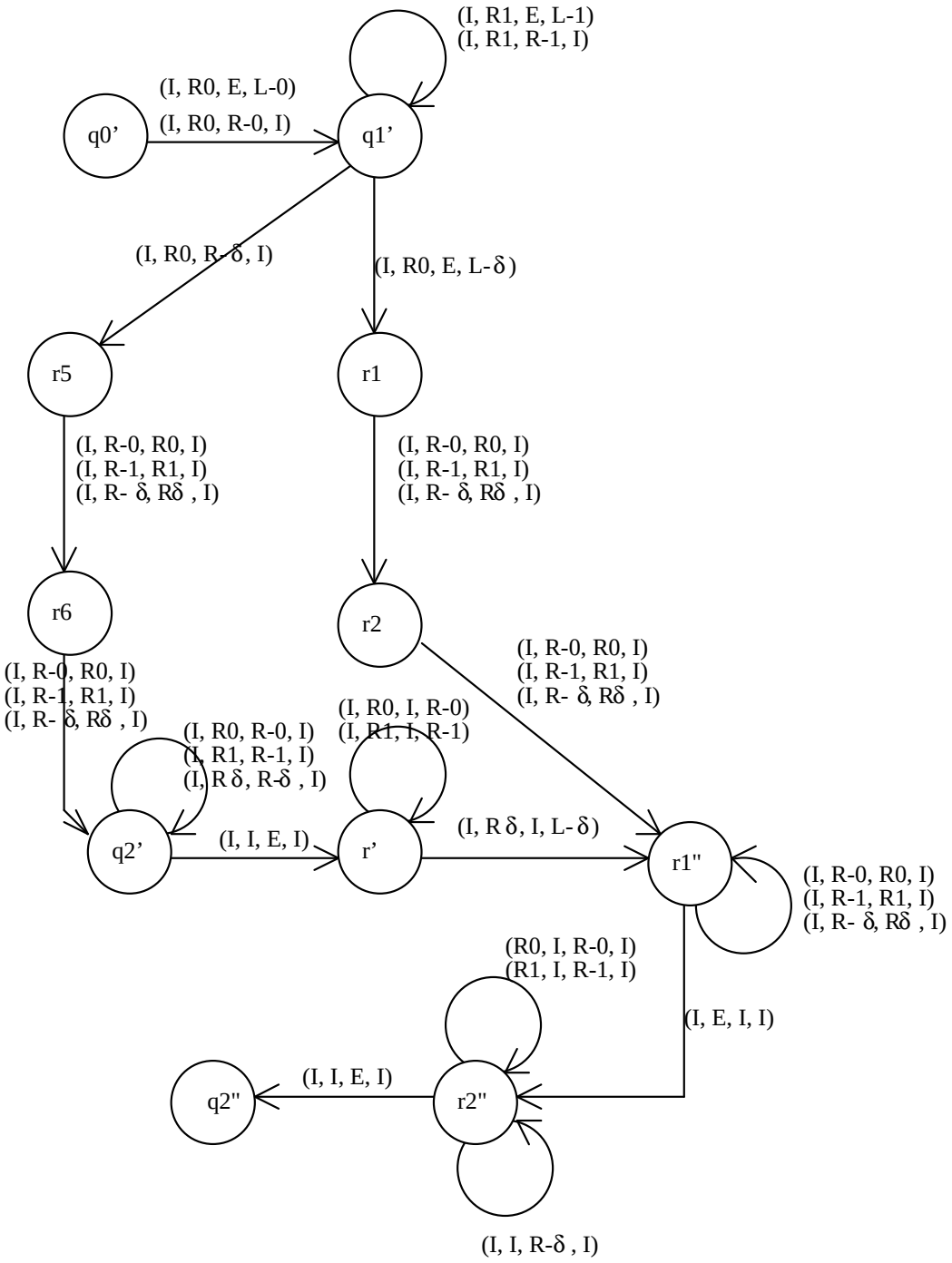


Figure 2.7.

Obviously, adding extra stacks to a 2-stack SMS X-machine (by constructing a n-stack SMS X-machine with n > 2) will not increase the power of the machine beyond that of a Turing machine (i.e. it can be shown fairly easily that any n-stack SMS X-machine can be simulated by a Turing machine by placing the input string, the n

stacks and the output string on the Turing machine tape separated by an extra symbol).

2.3.1. SMS X-machines as a basis for a specification and testing method.

The SMS X-machine model is general enough to model any computation performed by a Turing machine. Furthermore, the model is much more flexible and does not necessarily require the computation to be specified at the lowest level in the way that Turing machines do. Indeed, the type Φ of a top level SMS X-machine of a system can use fairly complex functions that can be defined by other means or can be even described themselves as SMS X-machines. Any function can be used as an arc label as long as we know that it is computable by some procedure.

However, our intention in investigating various X-machine models is to find a formal specification (i.e. a particular type of X-machine) that can be also used as basis for developing a theoretical testing strategy. The generality of the SMS X-machine model offers little hope in this direction. Indeed, a very simple SMS X-machine with two stacks together with the basic 'push' and 'pop' operations is as complex as a Turing machine. For instance, let $\mathcal{S}$ be a 2-stack SMS X-machine specification of a system and let $\mathcal{I}$ be its implementation. Then testing $\mathcal{I}$ against $\mathcal{S}$ would mean finding an algorithm which determines whether $\mathcal{I}$ and $\mathcal{S}$ compute the same function. This is impossible since $\mathcal{I}$ is an arbitrary Turing machine. Indeed there is no algorithm that establishes whether an arbitrary Turing machine halts for an arbitrary input sequence, let alone an algorithm that determines that two arbitrary Turing machines compute the same function. Since, in practice, a specification will be more complex than the 2 stack SMS X-machine model, the development of a well founded testing methodology based on this model appears to be impossible.

On the other hand, many real applications correspond to Turing machines that halt. Therefore, one might imagine a less general X-machine model that can be used to specify these systems and also provide a basis for a testing methodology. Also, it is fairly clear that the 'empty input' moves (i.e. when the machine does not consume an input) are the ones that cause all the problems in the SMS X-machine model. Indeed, an infinite loop caused by a certain input string *s* will contain only a finite number of 'non-empty input' moves but an infinite number of 'empty input' ones.

2.4. (Generalised) stream X-machines.

These are X-machines with no 'empty input' moves.

Definition 2.4.1.
Let Σ and Γ two alphabets, and let $\delta \notin \Sigma \cup \Gamma$ and $\Sigma' = \Sigma \cup \{\delta\}$. Then, an X-machine $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, Q_0, T, m_0)$ with

$X = \Gamma^* \times M \times \Sigma'^*$ is called a *stream X-machine* (denoted *S X-Machine*) if:

1. The input and output codes

$\alpha: \Sigma^* \to X, \quad \beta: X \to \Gamma^*$

are defined by

$\alpha(s) = (1, m_0, s\delta)\ \forall\ s \in \Sigma^*,$

$$\beta(g, m, s) = \begin{cases} g, & \text{if } s = 1 \\ \varnothing, & \text{otherwise} \end{cases}$$

$\forall\ g \in \Gamma^*,$

where $m_0 \in M$ is the *initial memory value*.

2. The type is Φ such that

$\forall\ \varphi \in \Phi$, $\varphi: X \leftrightarrow X$ is a relation of the form:

$$\varphi(g, m, s) = \begin{cases} (g\ \rho(m, \text{head}(s)), \mu(m, \text{head}(s)), \text{tail}(s)), & \text{if } s \neq 1 \\ \varnothing, & \text{otherwise} \end{cases}$$

where $\mu: M \times \Sigma' \leftrightarrow M$, $\rho: M \times \Sigma' \leftrightarrow \Gamma$ are relations.

If the relation ρ is defined as $\rho: M \times \Sigma' \leftrightarrow \Gamma^*$ instead of $\rho: M \times \Sigma' \leftrightarrow \Gamma$ then the X-machine is called a *generalised stream X-machine* (denoted *GS X-Machine*).

In any state $q \in Q$, a stream X-machine reads the first character σ of the input string s, removes it from s and adds a new character γ to the output string. The new value of the memory and γ depend on σ, but they are not affected by the rest of the input string. A generalised stream X-machine can add a string of characters (possibly the empty one) to the output string for each input character it processes. S X-machines and GS X-machines compute relations from Σ^* to Γ^*.

Obviously, S X-machines are special types of GS X-machines. Conversely, one could transform a generalised stream X-machine into a stream X-machine by augmenting the output alphabet to a (possibly infinite) set $\Gamma_1 \subseteq \Gamma^*$, the set of sequences from Γ^* produced by ρ (i.e. there exists an injection $h: \Gamma_1 \to \Gamma^*$). If $f: \Sigma^* \leftrightarrow \Gamma^*$ is the relation computed by the generalised stream X-machine and $f_1: \Sigma^* \leftrightarrow \Gamma_1^*$ is the relation computed by the stream X-machine obtained in this way, then $f = f_1 H$, where $H: \Gamma_1^* \to \Gamma^*$ is the morphism induced by h. Obviously, if only machine acceptors are considered (i.e. the output is not relevant) then the S X-machine and GS X-machine models are equivalent.

Obviously, no trivial paths exist in a (generalised) stream X-machine. Hence a deterministic (generalised) stream X-machine computes a function f: $\Sigma^* \rightarrow \Gamma^*$.

Obviously, the S X-machine model is more restrictive than the SMS X-machine model. However, the more restrictive nature of the model makes it more attractive as a basis for a specification based testing method. Indeed, if each basic function in Φ can be computed by a procedure that terminates in finite time (i.e. a Turing machine that halts) then, given any input $s \in \Sigma^*$, the computation determined by s through $\mathcal{M}$ also terminates in finite time (this is because the machine consumes an input symbol each time it performs a transition). Let us formalise this idea.

Definition 2.4.2.
A partial function φ is called *fully computable* if there exists an algorithm A (i.e. a Turing machine that halts) such that A computes φ. A type Φ is called *fully computable* if $\forall\ \varphi \in \Phi$, φ is *fully computable*.

Note: If $X = \Gamma^* \times \Omega_1^* \times ... \times \Omega_k^* \times \Sigma'^*$ with Γ, Ω_1, ..., Ω_k and Σ' finite, then let

$\varphi: X \rightarrow X$

and

$\varphi_o: X \rightarrow \Gamma^*$, $\varphi_1: X \rightarrow \Omega_1^*$, ..., $\varphi_k: X \rightarrow \Omega_k^*$, $\varphi_{k+1}: X \rightarrow \Sigma'^*$

be its projections on Γ^*, Ω_1^*, ..., Ω_k^* and Σ'^* respectively. Then φ is fully computable iff $\varphi_{o.t}$, $\varphi_{1.t}$, ..., $\varphi_{k.t}$, $\varphi_{k+1.t}$ are total recursive, where

$\varphi_{o.t}: X \rightarrow (\Gamma \cup \{c\})^*$ is defined by

$$\varphi_{o.t}(x) = \begin{cases} \varphi_o(x), \text{ if } x \in \text{dom } \varphi_o \\ c, \text{ otherwise} \end{cases}$$

where $c \notin (\Gamma \cup \Omega_1 \cup ... \cup \Omega_k \cup \Sigma)$; $\varphi_{1.t}$, ..., $\varphi_{k.t}$, $\varphi_{k+1.t}$ are defined similarly.

Proposition 2.4.3.
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, Q_o, T, m_o)$ be a deterministic (*generalised) stream* X-machine with Φ fully computable. Then, the partial function f: $\Sigma^* \rightarrow \Gamma^*$ computed by $\mathcal{M}$ is fully computable. Hence, the class of fully computable relations is closed under the (*generalised) stream* X-machine operator.

Proof:
Let $s = \sigma_1 ... \sigma_n$, with σ_1, ..., $\sigma_n \in \Sigma$. Hence

$\alpha(s) = (1, m_o, ..., \sigma_1 ... \sigma_n \delta)$.

Then, the path determined by $\alpha(s)$ consists of at most n+1 transitions. Therefore, $\varphi(s)$ is determined by applying at most n+1 algorithms (i.e. an algorithm for each φ which processes either σ_i or δ). Hence, $f = \alpha |\mathcal{M}| \beta$ is fully computable. ⑥

Obviously, if $\mathcal{M}$ is a stream X-machine acceptor with Φ fully computable, then the language accepted by $\mathcal{M}$ is recursive.

Thus, if each φ can be computed by a Turing machine that halts, then the whole machine can be represented as a Turing machine that halts. Then let us assume that we have specified a system as a stream X-machine $\mathcal{S}$ with a fully computable type Φ and that we can assume that the implementation $\mathcal{I}$ is also a stream X-machine with the same type. Then, $\mathcal{I}$ is guaranteed to halt. Hence if $\mathcal{I}$ is fed with an arbitrary input string *s*, then we are guaranteed to obtain the output in finite time. This is an important fact if the implementation is to be tested against the specification since we no longer have to solve the unsolvable halting problem.

Obviously, proposition 2.4.3 is not true for a SMS X-machine since it can contain loops consisting of only empty input operations which can cause the the machine to loop forever.

Obviously, if the stream X-machine is to be used as a specification tool, it is important to know how general this model is, i.e. how far can we get in the language hierarchy by using stream X-machines with a certain memory structure and a certain type Φ. Similarly to definition 2.3.8, we can define a k-stack stream X-machine.

Definition 2.4.4.
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, Q_0, T, m_0)$ be a stream X-machine with Σ and Γ finite. If

1. $M = \Omega_1^* \times ... \times \Omega_k^*$ (hence $X = \Gamma^* \times \Omega_1^* \times ... \times \Omega_k^* \times \Sigma'^*$), where $\Omega_1, ... \Omega_k$ are finite alphabets
2. $\Phi = \{R_\gamma | \gamma \in \Gamma\} \times \Phi_1 \times ... \times \Phi_k \times \{L_{-\sigma} | \sigma \in \Sigma'\}$,

where

$$\Phi_i = \{R_{-\omega} | \omega \in \Omega_i\} \cup \{R_\omega | \omega \in \Omega_i\} \cup \{I, E\}, i = 1, ..., k,$$

then $\mathcal{M}$ is called a *k-stack stream X-machine* (denoted *k-stack S X-machine*).

We saw that the class of 2-stack SMS X-machine was equivalent to the class of Turing machines. However, k-stack S X-machines are much more restrictive. Let us denote by L_k the class of languages accepted by k-stack S X-machines acceptors (i.e. $\Gamma = \varnothing$) and

$$L = \bigcup_{k=1}^{\infty} L_k.$$

We shall call L the class of *real time stack languages* (i.e. in this context *real time* refers to the fact that the machine does not have empty input moves, hence it decides whether to accept an input string immediately after it reads it). Obviously any L_k will

contain all regular languages. However, L does not contain all deterministic context-free languages. In fact we have the following result.

Proposition 2.4.5.
L_n and the class of deterministic context-free languages are incomparable $\forall\ n \in N$, $n \geq 2$.

Proof:
Let $\Sigma = \{a, b, c\}$ and

$$L = \{a^n b^n c^n \mid n \in N\}.$$

Obviously, L is not context-free (see Cohen [8]). It can be proven easily that $L \in L_2$.

Conversely, let

$$L' = \{a^{i_1} b a^{i_2} b \ldots a^{i_r} b a^{i_r} c^s a^{i_{r-s+1}} \mid r \geq 1,\ 1 \leq s \leq r,\ i_j \geq 1 \text{ for all } 1 \leq j \leq r\}.$$

Harrison, [24], proves that L' is deterministic context-free and $L' \notin L_n$, $\forall\ n \in N$.⑥

From the proof above it also follows that L_n and the class of (non-deterministic) context-free languages are incomparable $\forall\ n \in N$, $n \geq 2$. The position of L in the language hierarchy is shown by the diagram in figure 2.8.

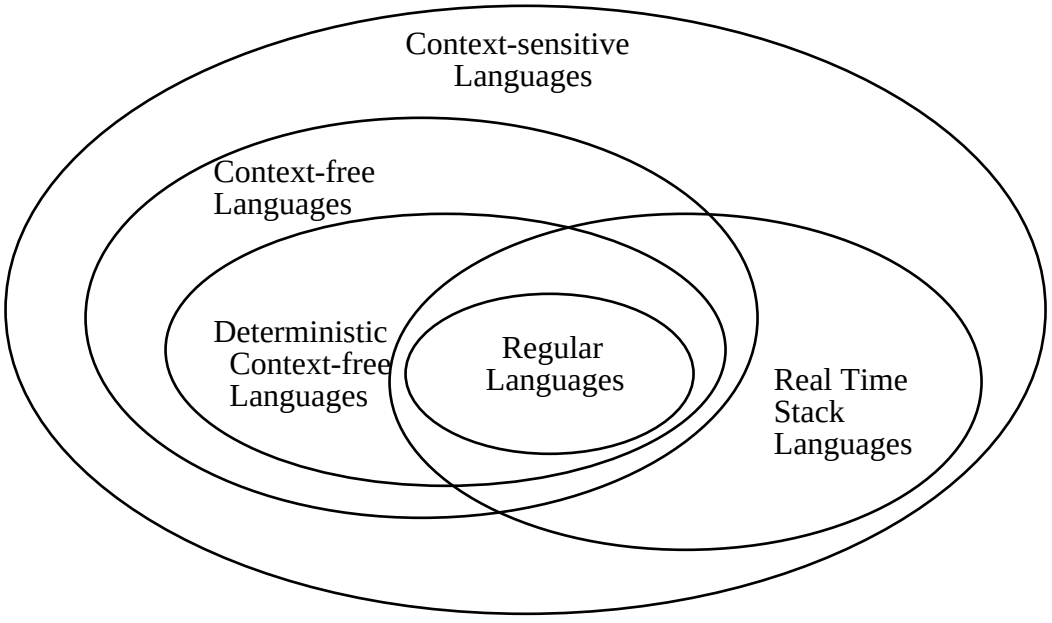


Figure 2.8.

It is clear that the k-stack S X-machines cannot cope with many applications. For example an arbitrary push-down automaton cannot be represented in this way. This is

because such a pushdown automaton might require an unbounded number of 'empty input' moves between two moves in which the machine consumes input symbols. However, we can 'hide' these empty moves by choosing more complex ϕ's which will allow the machine to perform an unbounded number of operations on its stack each time it reads an input symbol. Indeed, the fact that the X-machine does not restrict us to using only very low level functions such as push and pop is one of the main advantages of this model. We can use more complicated basic functions as long as we know that they can be computed by some computational models. Those models will, preferably, be simpler stream X-machines themselves, so the X-machine model can be used as a specification tool hierarchically. In this way we can have a high level stream X-machine model and several low-level models that specify each $\phi \in \Phi$. Since the finite automaton is the simplest stream X-machine, then we can try to construct a stream X-machine whose ϕ's are computed by finite automata.

2.4.1. Regular stack stream X-machines.

Before proceeding any further, we introduce some preliminary concepts.

Definition 2.4.1.1.
Let Ω be a finite alphabet and L be a subset of Ω^*. Then, following Eilenberg, [12], we call L a *prefix* if

$$s^{-1}L = \{1\}, \forall\, s \in L.$$

Example 2.4.1.2.
Let $\Sigma = \{a, b, c\}$. Then $L = \{ab^*c\}$ is a prefix.

In what follows we prove some properties of prefixes that we shall need later on. Proposition 2.4.1.3 is from Eilenberg, [12].

Proposition 2.4.1.3.
Let $L \subseteq \Omega^*$. Then the following conditions are equivalent:

1. L is a prefix.
2. If $s, st \in L$, then $t = 1$.
3. If $st = s't'$ with $s, s' \in L$, then $s = s'$ and $t = t'$.
4. $L = \varnothing$ or the minimal automaton (possibly infinite) of L has one terminal state t and no arcs leave t.
5. L is the behaviour of a deterministic (possibly infinite) automaton such that no arcs leave a terminal state.

Proposition 2.4.1.4.
1. If L is a prefix and $1 \in L$, then $L = \{1\}$.
2. Any subset of a prefix is a prefix.

Proof:
1 is obvious. 2 follows from proposition 2.4.1.3. ⑥

Lemma 2.4.1.5.
If g: $\Sigma^* \rightarrow \Omega^*$ is a morphism such that $g^{-1}(1) = \{1\}$ and $L \subseteq \Omega^*$ is a prefix, then $g^{-1}(L)$ is a prefix.

Proof:
If $s, st \in g^{-1}(L)$, then $g(s), g(s)g(t) \in L$. Hence $g(t) = 1$ and $t = 1$. ⑥

Definition 2.4.1.6.
Let Ω be a finite alphabet and L be a subset of Ω^*. Then L is called a *regular prefix* if L is both a prefix and a regular language.

Proposition 2.4.1.7.
Let $L \subseteq \Omega^*$. Then the following conditions are equivalent:

1. L is a regular prefix.
2. $L = \varnothing$ or the minimal finite automaton of L has one terminal state t and no arcs leave t.
3. L is the behaviour of a deterministic finite automaton such that no arcs leave a terminal state.

Proof:
This follows from the fact that L is both a regular language and a prefix. ⑥

Lemma 2.4.1.8.
If g: $\Sigma^* \rightarrow \Omega^*$ is a morphism such that $g^{-1}(1) = \{1\}$, $L \subseteq \Omega^*$ is a regular prefix and $K \subseteq \Sigma^*$ is a regular language, then

$$g^{-1}(Lu) - K$$

is a regular prefix $\forall u \in \Omega^*$.

Proof:
Let $u \in \Omega^*$. Since L is a regular language, Lu is a regular prefix. Therefore $g^{-1}(Lu)$ is a regular language (this is because the class of regular languages is closed under inverse morphisms (see Hopcroft & Ulman [34]). Since K is also a regular language, it follows that

$$g^{-1}(Lu) - K$$

is a regular language (this is because the class of regular languages is closed under '-' (see Hopcroft & Ulman [34]). Using the minimal automata of L and Lu, it can be shown easily that Lu is a regular prefix. From lemma 2.4.1.5, it follows that $g^{-1}(Lu)$ is a prefix. Hence any subset of $g^{-1}(Lu)$ is a prefix. Therefore

$$g^{-1}(Lu) - K$$

is a prefix. ⑥

Having a non-empty regular prefix $L \subseteq \Omega^*$, we can define a partial function

$$R_{-L}: \Omega^* \to \Omega^*$$

as described bellow. This type of function will be used to construct our new type of stream X-machine.

Definition 2.4.1.9.
Let Ω be a finite alphabet and $L \subseteq \Omega^*$ be a non-empty prefix. Then we define a partial function

$$R_{-L}: \Omega^* \to \Omega^*$$

by:

$$R_{-L}(s) = \begin{cases} s\,rev(v)^{-1}, \text{ if } \exists\, v \in L \text{ such that } s\,rev(v)^{-1} \neq \emptyset, \\ \emptyset, \text{ otherwise} \end{cases}$$

Note: rev(x) denotes the reverse of x. For $s, u \in \Omega^*$, $su^{-1} \neq \emptyset$ means that $\exists\, t \in \Omega^*$ such that $s = tu$.

Observation 2.4.1.10.
Since L is a prefix, there exists at most one $v \in L$ such that $s\,rev(v)^{-1} \neq \emptyset$ (see proposition 2.4.1.3, 2). Therefore R_{-L} is well defined.

Example 2.4.1.11.
Let $\Omega = \{a, b, c\}$ and $L = \{ab^*c\}$.

1. If the value of the stack is s = aacbba, then $R_{-L}(s) = aa$.
2. If the value of the stack is s = aacbbab, then $R_{-L}(s)$ is not defined.

If L is a regular prefix, then $R_{-L}(s)$ removes symbols from the end of the input string s like a finite state machine in which no arcs leave a terminal state. Therefore R_{-L} can be computed using a finite automaton. We shall use such functions to construct a new type of stream X-machine.

Definition 2.4.1.12:
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, Q_0, T, m_0)$ be a stream X-machine. If:

1. $M = \Omega^*$ (hence $X = \Gamma^* \times \Omega^* \times \Sigma'^*$), where Ω is a finite alphabet
2. $\Phi = \{R_\gamma | \gamma \in \Gamma\} \times \Phi_M \times \{L_{-\sigma} | \sigma \in \Sigma'\}$,

where

$$\Phi_M = \{R_{-L}R_u | u \in \Omega^*, L \subseteq \Omega^* \text{ is a non-empty regular prefix}\},$$

then $\mathcal{M}$ is called a *regular stack stream X-machine* (denoted *RStack S X-machine*).
A *regular stack generalised stream X-machine* (denoted *RStack GS X-machine*) will have the type

$$\{R_g | g \in \Gamma^*\} \times \Phi_M \times \{L_{-\sigma} | \sigma \in \Sigma'\},$$

with Φ_M defined as above.

Note: $R_{-L}R_u$ denotes the functional composition of R_{-L} and R_u. $R_{-L}R_u$ removes symbols from the end of the memory stack like a finite state machine and then adds a *fixed* string u.

We shall prove that the class of *regular stack stream X-machine* acceptors accepts exactly the deterministic context-free languages. We shall do this by showing that a deterministic pushdown automaton (i.e. a deterministic 1-stack SMS X-machine acceptor) can be converted into a RStack S X-machine.

2.4.2. RStack S X-machines and 1-stack SMS X-machines acceptors are equivalent.

In this section we shall be referring to X-machine acceptors (i.e. $\Gamma = \varnothing$).

Let us consider a deterministic SMS X-machine acceptor,

$$\mathcal{M} = (\Sigma, \varnothing, Q, M, \Phi, F, q, T, m_0).$$

Then $\Phi = \Phi_1' \cup \Phi_2'$, where Φ_1' and Φ_2' are defined in Observation 2.3.3.

Let $q \in Q$ and $m \in M$. Let

$$\mathcal{M}'(q, m) = (\Sigma, \varnothing, Q, M, \Phi_2', F', q_0, Q, m)$$

be the deterministic X-machine obtained from $\mathcal{M}$ by removing all the arcs with labels in Φ_1' (i.e. $F'|Q \times \Phi_2' = F|Q \times \Phi_2'$ and $F'|Q \times \Phi_1'$ is null, where $F|Q \times \Phi_i'$ denotes the restriction of F to the set $Q \times \Phi_i'$, i = 1, 2) with q the initial state, m the initial memory value and the set of terminal states T' = Q.

Since $\mathcal{M}'(q, m)$ operates only on empty input moves, it is clear that the computation of $\mathcal{M}'(q, m)$ is independent of the value of the input register and it depends only on the initial state q and the initial memory value m.

We say that $\mathcal{M}'(q, m)$ *halts* if:

i) $\exists\, q' \in Q$, $m' \in M$ and a path $q \xrightarrow{p} q'$ in $\mathcal{M}'(q, m)$ such that

$$|p|(m, s) = (m', s), \forall\, s \in \Sigma'^*$$

($|p|$ is the partial function computed along p) and

ii) $\forall\, \varphi \in \Phi_2'$ with $(q', \varphi) \in \text{dom } F$, we have $\varphi(m', s) = \varnothing$, $\forall\, s \in \Sigma'^*$

(in other words p is the *maximal* path that $\mathcal{M}'(q, m)$ can follow).

We assume that $\mathcal{M}'(q, m)$ halts $\forall\, q \in Q$, $m \in M$ (i.e. there is such a *maximal* path). Then we can define a function

$$\tau: Q \times M \to Q \times M$$

by

$\tau(q, m) = (q', m')$,
where q' is the state in which $\mathcal{M}'(q, m)$ halts and m' is the final memory value.

Note: More rigorously, we can use an auxiliary function π: $Q \times M \rightarrow Q \times M$, which keeps track of the states and memory values of the computation of $\mathcal{M}$ on empty moves, i.e.

$$\pi(q, m) = \begin{cases} (F(q, \varphi), Mem(\varphi(m, 1))), \text{ if } \exists\, \varphi \in \Phi_2' \text{ such that } (q, \varphi) \in \text{dom } F \text{ and } \varphi(m, 1) \neq \varnothing \\ \varnothing, \text{ otherwise} \end{cases}$$

Hence $\tau(q, m) = \pi^n(q, m)$,
where

$n = \max\{k \in N \mid \pi^k(q, m) \neq \varnothing\}$.

Since we assumed that $\forall\ q \in Q, m \in M$, $\mathcal{M}'(q, m)$ halts, n is finite and τ is well defined.

Finally, we define the set of functions

$\{\psi_{q.q'}\}_{q,q' \in Q}$,
which describes the computation $\mathcal{M}$ on empty moves:

$$\psi_{q.q'}: M \times \Sigma'^* \rightarrow M \times \Sigma'^*$$

is defined by

$$\psi_{q.q'}(m, s) = \begin{cases} (m', s), \text{ if } \exists\, m' \in M \text{ such that } (q', m') = \tau(q, m) \\ \varnothing, \text{ otherwise} \end{cases}$$

Therefore, $\psi_{q.q'}(m, s)$ is defined if $\mathcal{M}'(q, m)$ halts in q'; $\psi_{q.q'}$ keeps the input register unchanged (this is because empty input moves do not affect this) while the machine transforms m into m', where m' is the final memory value of the computation of $\mathcal{M}'(q, m)$.

Let

$\Psi = \{\psi_{q.q'} \mid q, q' \in Q \text{ and } (q' \in T \text{ or } \exists\, \varphi \in \Phi_1' \text{ such that } F(q', \varphi) \neq \varnothing)\}$.
Thus Ψ is the set of those functions $\psi_{q.q'}$ for which q' is either a terminal state of $\mathcal{M}$ or there exists an arc emerging from q' labelled by a non-empty input function.

Then, we have the following result.

Proposition 2.4.2.1.
Let $\mathcal{M} = (\Sigma, \varnothing, Q, M, \Phi, F, q_0, T, m_0)$ be a deterministic SMS X-machine acceptor and L be the language accepted by it. If $\forall\ q \in Q, m \in M$, $\mathcal{M}'(q, m)$ halts, then there exists a deterministic S X-machine acceptor $\mathcal{M}''$ with the type $\Phi'' = \Phi_1' \Psi$, where

Φ_1'Ψ = {$\varphi\ \psi_{q.q'}$ | $\varphi \in \Phi_1$', q, q' $\in$ Q and (q' $\in$ T or $\exists\ \varphi \in \Phi_1$' such that $F(q',\varphi) \neq \emptyset$)},

which accepts the same language L.

Proof:

We define the deterministic S X-machine $\mathcal{M}$" as follows:

1. The set of states Q, the memory M and the input alphabet (i.e. Σ) remain unchanged. Obviously, the output alphabet is empty ($\Gamma = \emptyset$).
2. q_o" and m_o" the initial state and the initial value of the memory are chosen such that (q_o", m_o") = $\tau(q_o, m_o)$.
3. The set of terminal states is T (therefore unchanged).
4. The type is Φ" = Φ_1'Ψ.
5. The next state function F": Q $\times$ Φ" $\rightarrow$ Q is defined by

$$F''(p, \varphi\psi_{q.q'}) = \begin{cases} q', \text{ if } q = F(p,\varphi) \\ \emptyset, \text{ otherwise} \end{cases}$$

$\forall$ q, q', p $\in$ Q such that q' $\in$ T or $\exists\ \varphi \in \Phi_1$' such that $F(q',\varphi) \neq \emptyset$.

Let L and L" be the languages accepted by $\mathcal{M}$ and $\mathcal{M}$" respectively. Then, we have to prove that L = L".

Let s $\in$ L, s = $\sigma_1\sigma_2...\sigma_{n-1}$ and let

$$(q_o, m_o, \sigma_1 \dots \sigma_{n-1}\delta) \xrightarrow{\varphi'_{0,1} \dots \varphi'_{0,k_0}} (q_o'', m_o'', \sigma_1 \dots \sigma_{n-1}\delta) \xrightarrow{\varphi_1} (q_1, m_1, \sigma_2 \dots \sigma_{n-1}\delta) \xrightarrow{\varphi'_{1,1} \dots \varphi'_{1,k_1}}$$

$$(q_1'', m_1'', \sigma_2 \dots \sigma_{n-1}\delta) \dots \xrightarrow{\varphi_n} (q_n, m_n, 1) \xrightarrow{\varphi'_{n,1} \dots \varphi'_{n,k_n}} (q_n'', m_n'', 1)$$

be the computation determined by sδ through $\mathcal{M}$, where k_o, ..., $k_n \in$ N, q_o" and m_o" are the initial state and memory value of $\mathcal{M}$" and q_n" is a terminal state (obviously, $\varphi_i \in \Phi_1$', i = 1, ..., n and $\varphi'_{i,1}$ ... $\varphi'_{i,k_i} \in \Phi_2$', i = 0, ..., n).

Then

$\varphi'_{i,1}$ ... $\varphi'_{i,k_i}(m_i, x) = \psi_{q_i.q_i''}(m_i, x)$, $\forall\ x \in \Sigma'^*$, i = 0,..., n,

hence the computation determined through $\mathcal{M}''$ by sδ is

$$(q_o'', m_o'', \sigma_1 \dots \sigma_{n-1}\delta) \xrightarrow{\varphi_1\psi_{q_1.q_1''}} (q_1'', m_1'', \sigma_2 \dots \sigma_{n-1}\delta) \xrightarrow{\varphi_2\psi_{q_2.q_2''}} \dots \xrightarrow{\varphi_n\psi_{q_n.q_n''}} (q_n'', m_n'', 1)$$

Hence s $\in$ L".

Note: The notation $(q, m, t) \xrightarrow{\zeta_1 \dots \zeta_n} (q', m', t')$ used above denotes that the machine follows the path ζ_1 ... ζ_k and ζ_1 ... ζ_k(m, t) = (m', t'), with ζ_1 ... ζ_k being transition functions, m, m' memory values and t, t' input strings.

This follows from the following two facts:

i). $\varphi'_{i,1} \ldots \varphi'_{i,ki}\, \varphi_{i+1}(m_i, \sigma_{i+1} \ldots \sigma_{n-1}\delta) = \varphi_{i+1}(m_i'', \sigma_{i+1} \ldots \sigma_{n-1}\delta) \neq \varnothing$
for i = 0, ..., n-1.
Since

$\varphi'_{i,1} \ldots \varphi'_{i,ki}\, \varphi_{i+1}(m_i, \sigma_{i+1} \ldots \sigma_{n-1}\delta) \neq \varnothing$,

and

$(q_i'', \varphi_{i+1}) \in$ dom F,

it follows that $\forall\, \varphi \in \Phi_2'$ with $(q_i'', \varphi) \in$ dom F, we have

$\varphi'_{i,1} \ldots \varphi'_{i,ki}\, \varphi(m_i, \sigma_{i+1} \ldots \sigma_{n-1}\delta) = \varnothing$

(this is because $\mathcal{M}$ is deterministic).
Therefore

$\varphi'_{i,1} \ldots \varphi'_{i,ki}(m_i, \sigma_{i+1} \ldots \sigma_{n-1}\delta) = \psi_{q_i.q_i''}(m_i, \sigma_{i+1} \ldots \sigma_{n-1}\delta)$, i = 0,..., n-1

and hence

$\varphi'_{i,1} \ldots \varphi'_{i,ki}(m_i, x) = \psi_{q_i.q_i''}(m_i, x)$, $\forall\, x \in \Sigma'^*$, i = 0,..., n.

ii). Since q_n'' is a terminal state of $\mathcal{M}$, we have

$F(q_n'', \varphi) = \varnothing$, $\forall\, \varphi \in \Phi_2'$.

Thus

$\varphi'_{n,1} \ldots \varphi'_{n,kn}(m_n, x) = \psi_{q_n.q_n''}(m_n, x)$, $\forall\, x \in \Sigma'^*$.

The converse implication (i.e. if $s \in L''$, then $s \in L$) can be proven similarly since $\forall\, m \in M$ and $\psi_{q.q'} \in \Psi$, there is a set of functions $\varphi'_{j1}, .., \varphi'_{jk} \in \Phi_2'$, such that

$\psi_{q.q'}(m, x) = \varphi'_{j1} \ldots \varphi'_{jk}(m, x)$ $\forall\, x \in \Sigma'^*$

and $\varphi'_{j1} \ldots \varphi'_{jk}$ label a path from q to q'. ⑥

Therefore, if $\mathcal{M}'(q, m)$ halts $\forall\, q \in Q$, $m \in M$, $\mathcal{M}$ can be converted into a S X-machine with type $\Phi'' = \Phi_1' \Psi$. We show now that, given a 1-stack SMS X-machine $\mathcal{M}$, there exists a 1-stack SMS X-machine $\mathcal{M}_e$ equivalent to $\mathcal{M}$ (i.e. $\mathcal{M}$ and $\mathcal{M}_e$ accept the same language) such that $\mathcal{M}_e'(q, m)$ halts for any state q and memory value m. As a consequence, $\mathcal{M}_e$ can be converted into a S X-machine (we shall prove later that the S X-machine obtained is a RStack S X-machine).

Lemma 2.4.2.2.
Let $\mathcal{M} = (\Sigma, \varnothing, Q, M, \Phi, F, q_o, T, m_o)$ be a deterministic 1-stack SMS X-machine acceptor. There exists then a deterministic 1-stack SMS X-machine acceptor $\mathcal{M}_e = (\Sigma, \varnothing, Q_e, M, \Phi, F_e, q_{oe}, T_e, m_o)$ equivalent to $\mathcal{M}$ such that $\mathcal{M}_e'(q, m)$ halts $\forall\, q \in Q_e$, $m \in M$.

Proof:
In this case

$\Phi_1' = \Phi_1' \cup \Phi_2'$,

where

$\Phi_1' = \Phi_M \times \{L_{-\sigma} | \sigma \in \Sigma'\}, \Phi_2' = \Phi_M \times \{I\}$,

with

$\Phi_M = \{R_{-a} | a \in \Omega\} \cup \{R_a | a \in \Omega\} \cup \{I\}$,

where Ω is the stack alphabet.

Without loss of generality, we can assume that

$\Phi_1' = \{I\} \times \{L_{-\sigma} | \sigma \in \Sigma'\}$

(this is because $(R_{-a}, L_{-\sigma})$ can be written as the composition of two functions,

$(R_{-a}, L_{-\sigma}) = (I, L_{-\sigma})\ (R_{-a}, I)$;

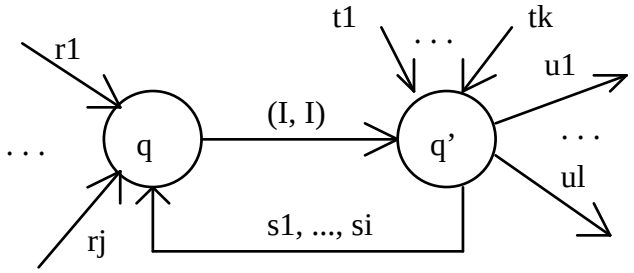
similarly,

$(R_a, L_{-\sigma}) = (I, L_{-\sigma})\ (R_a, I))$

Let G be the graphical representation of F (i.e. the state diagram associated with F). We shall transform G into a new diagram G' such that the computation of the machine $\mathcal{M}$ remains unaffected. The following two procedures are used:

1) If there is an arc $q \xrightarrow{(I, I)} q'$ (i.e. I is the identity function), then q and q' are merged. We have the following two cases:

a. There is no arc labelled (I, I) from q' to q (Fig. 2.9). By merging q and q', figure 2.9 is transformed into figure 2.10.



Note: r_1, .., r_j are arcs that are incident on q, s_1, .., s_i arcs from q' to q, u_1, .., u_l arcs from q' to any state other than q and q', t_1, .., t_k arcs that are incident on q'.

Figure. 2.9.

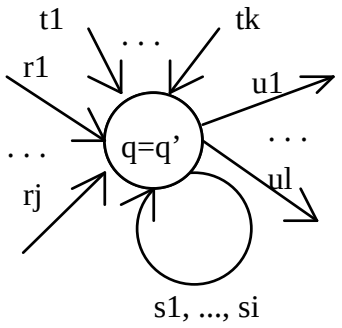


Figure 2.10.

b. If there is an arc labelled (I, I) from q' to q, then there is no other arc leaving q' (otherwise the machine will be non-deterministic) and, once the machine is in the state q (or q'), it will loop forever. Then, figure 2.11 is transformed into figure. 2.12.

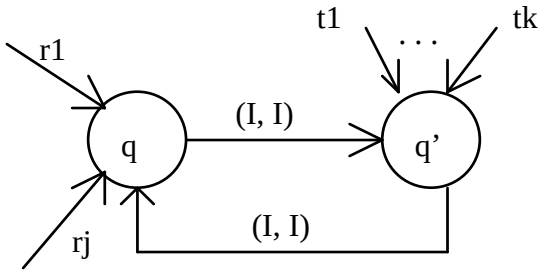


Figure 2.11.

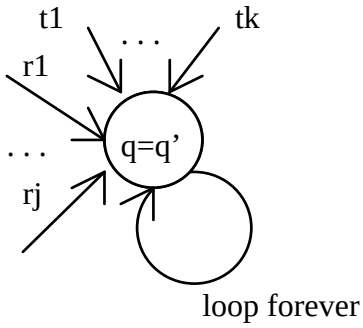


Figure 2.12.

We apply these two rules until all the edges labelled (I, I) bave been eliminated. Since their number is finite, the procedure will also be finite.

2) Since

$R_a R_{-a} = I, \forall a \in \Omega$

$R_a R_{-b} = \varnothing, \forall a, b \in \Omega, a \neq b,$

any (R_a, I) followed by an (R_{-b}, I) can be eliminated.

Let q and q' be two states such that:

i) there exists an arc labelled (R_a, I), $a \in \Omega$, that connects them, i.e.

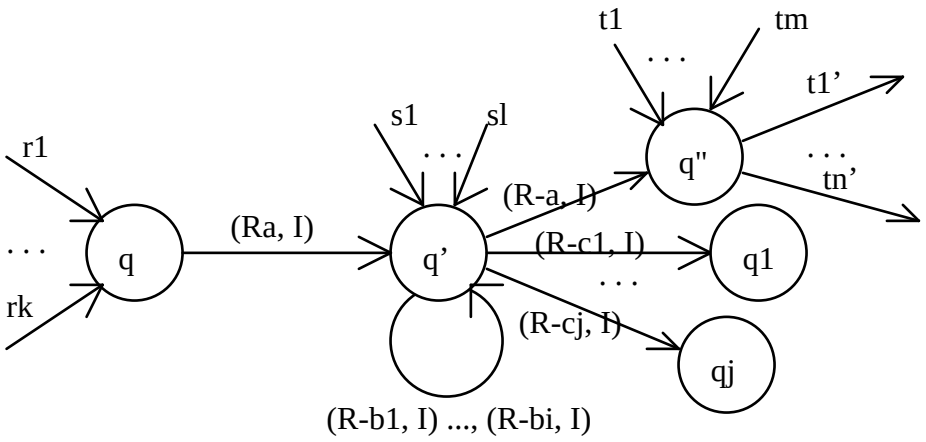
$q \xrightarrow{(R_a, I)} q'$;

ii) there exists at least one arc (R_{-b}, I), $b \in \Omega$, that leaves q'.

Then, since $\mathcal{M}$ is deterministic, all of the arcs that leave q' will be labelled by functions belonging to the set $\{(R_{-\omega}, I), \omega \in \Omega\}$. We show now that the arc $q \xrightarrow{(R_a, I)} q'$ can be eliminated. We have the following cases:

a. There is no arc labelled (R_{-a}, I) leaving q'. Then the arc $q \xrightarrow{(R_a, I)} q'$ can be deleted.

b. There is an arc labelled (R_{-a}, I) from q' to a third state q" (figure 2.13). By eliminating the arc $q \xrightarrow{(R_a, I)} q'$, q and q" are merged and figure 2.13 is modified as shown in figure 2.14.



$b_1, .., b_i, c_1, .., c_j \in \Omega - \{a\}$.

Figure 2.13.

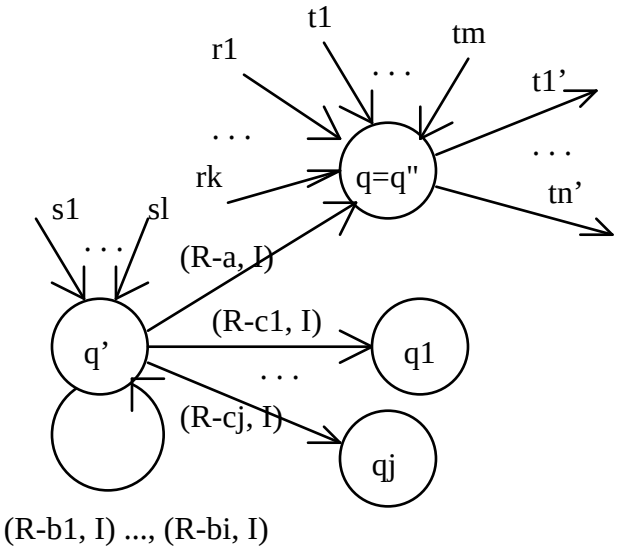
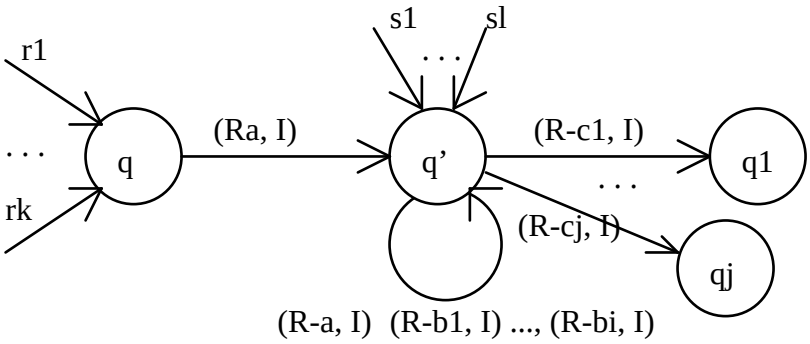


Figure 2.14.

c. There is an arc labelled (R_{-a}, I) from q' to itself (figure 2.15). By eliminating the arc $q \xrightarrow{(R_a, I)} q'$, q and q' are merged (figure 2.16).



$b_1, .., b_i, c_1, .., c_j \in \Omega - \{a\}$.

Figure 2.15.

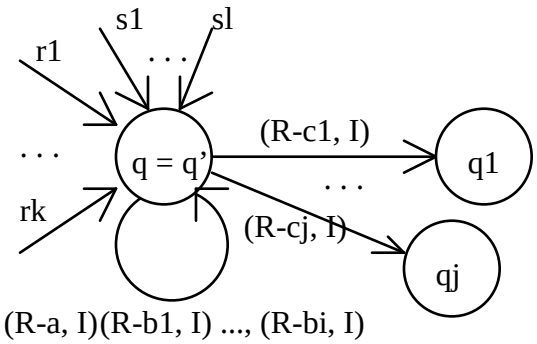
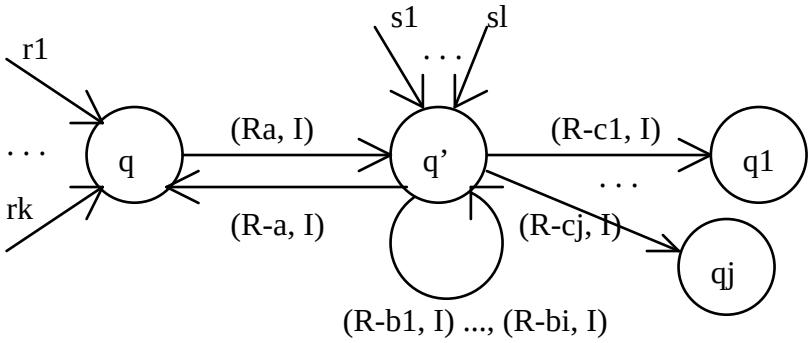


Figure 2.16.

d. There is an arc labelled (R_{-a}, I) from q' to q (figure 2.17). In this case, when the machine reaches state q, it will loop forever; therefore, figure 2.17 is transformed into figure 2.18.



$b_1, .., b_i, c_1, .., c_j \in \Omega - \{a\}$.

Figure 2.17.

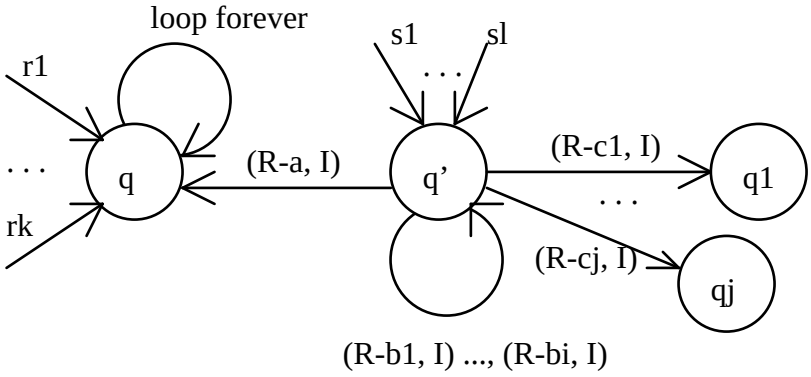


Figure 2.18.

Since the number of arcs of graph G labelled with functions of the type (R_a, I) is finite, the procedure which results from applying the rules above is finite. The resulting graph will contain no paths in which a (R_a, I) is followed by a (R_{-b}, I).

Since any input string which causes $\mathcal{M}$ to loop forever will not be accepted by this, we can further modify the graph G without affecting the computation of $\mathcal{M}$ as follows:

i). Any arc labelled 'loop forever' can be removed.

ii). Any loop formed only by arcs labelled with functions of type (R_a, I) can be opened by removing any of them.

Let G' be the graph resulting from transforming G as above. Then we can construct

$\mathcal{M}_e = (\Sigma, \emptyset, Q_e, M, \Phi, F_e, q_{oe}, T_e, m_o)$

a 1 stack SMS X-machine where:

1. Q_e is the state set of the graph G',
2. T_e is the set of all states $q \in Q_e$ such that either:

   $q \in T$ is a terminal state of $\mathcal{M}$ that has not been affected by the above transformation (i.e. it has not been merged with another state) or

   q is a state obtained by merging a number of states of which at least one was a terminal state of $\mathcal{M}$;
3. q_{oe} is either:

   q_o, if this has not been affected by the above transformations or

   the state that resulted from merging the initial state q_o of $\mathcal{M}$ with other states;
4. F_e is the next state function determined by the graph G'.

Obviously the memory ($M = \Omega^*$), the type Φ and the initial memory value remain the same as for $\mathcal{M}$.

From the construction of G', it is obvious that $\mathcal{M}$ and $\mathcal{M}_e$ are equivalent (i.e. they accept the same language). We can now prove that the computation of $\mathcal{M}_e'(q, m)$ halts $\forall\, q \in Q_e, m \in M$.

Let $N = card(Q_e)$. Let $q \in Q_e$ be a state of the graph G', $m \in M$ be a value of the stack and n be the length of m (i.e. the number of characters that m contains). Also, let p be an arbitrary path in G' that starts from the state q containing only elements of Φ_2' and let $|p|$ be the partial function computed along this path. We show that if $|p|(m, 1) \neq \emptyset$, then p contains at most n + N arcs. Indeed, since $|p|(m, 1) \neq \emptyset$ and because p cannot contain a function of type (R_a, I) followed by another of type (R_{-b}, I), p may be written as p = p'p", where:

$p' = (R_{a1}, I) .... (R_{-ai}, I)$, with $a_1, .., a_i \in \Omega$, $0 \leq i \leq n$,

$p" = (R_{b1}, I) .... (R_{bj}, I)$ with $b_1, .., b_j \in \Omega$.

Since G' does not contain any loops formed only of functions of type (R_a, I) it follows that $j \leq N$ (otherwise there will be a state which appears twice, and so G' will have such loops). Thus p has at most N + n arcs.

Therefore $\mathcal{M}_e'(q, m)$ halts $\forall\, q \in Q_e, m \in M$. ⑥

We can now prove that the classes of 1-stack SMS X-machines and RStack S X-machines acceptors are equivalent.

Proposition 2.4.2.3.
Let $\mathcal{M}$ be a deterministic 1-stack SMS X-machine acceptor. Then there exists a deterministic RStack S X-machine acceptor $\mathcal{M}$" such that $\mathcal{M}$" and $\mathcal{M}$ accept the same language.

Proof:
We shall refer to $\mathcal{M}_e$, the machine that results by applying lemma 2.4.2.2. As in the proof of this lemma, we shall assume (without loss of generality) that the set of non-empty input moves of $\mathcal{M}_e$ is

$\Phi_1' = \{I\} \times \{L_{-\sigma} | \sigma \in \Sigma'\}$.

From lemma 2.4.2.1 it follows that $\mathcal{M}_e$ can be converted into a S X-machine with the type $\Phi_1' \Psi$.

Let $q, q' \in Q$ be such that q' is either a terminal state of $\mathcal{M}_e$ or there exists an arc emerging from q' labelled by a non-empty input function. Then we prove that $\psi_{q.q'}$ can be written as a *finite* union of functions of the form $(R_{-L}R_u, I)$, where $L \subset \Omega^*$ is a non-empty regular prefix and $u \in \Omega^*$ a finite string.

Note: If f, g: $A \to B$ are two partial functions with dom f $\cap$ dom g = $\varnothing$, then their union f $\cup$ g is a (partial) function h: $A \to B$ defined by:

$$h(a) = \begin{cases} g(a), \text{ if } a \in \text{dom } g \\ h(a), \text{ if } a \in \text{dom } h \\ \varnothing, \text{ otherwise} \end{cases}$$

from the way in which q' is chosen it follows that

$\psi_{q.q'} = \cup |p|$,

where the union extends over all the paths p containing only empty input moves that start in q and end in q'. Let p be such a path. Then p = p' p", where

p' = (R_{a1}, I) .... (R_{-ai}, I), where $a_1, .., a_i \in \Omega$,

p" = (R_{b1}, I) .... (R_{bj}, I), where $b_1, .., b_j \in \Omega$.

We saw in the proof of the lemma above that $j \leq N$, where N is the number of states of the graph G'. Therefore the number of such p" is finite. We denote by $\{p_1", ..., p_n"\}$ the set of all such p". Then let $s \in \{1, ..., n\}$ be such that $p_s" = p"$ and let q_s be the initial state of $p_s"$. Then p can be written as p = p' $p_s"$, where p' is a path that starts from q and finishes in q_s. The machine $\mathcal{M}_e$ removes symbols from the stack along p' and adds symbols to the stack along $p_s"$.

Now, for s = 1, ..., n, let H_s be the (partial) function defined by

$H_s = \cup |p'|$,

where the union extends over all the paths p' of the form

$p' = (R_{-a1}, I) .... (R_{-ai}, I)\ a_1, .., a_i \in \Omega$, starting in q and ending in q_s.

Then

$$\psi_{q.q'} = \bigcup_{s=1}^{n} H_s|p_s''|.$$

It is clear that there is no arc from q_s labelled with a function of the type (R_{-a}, I), $a \in \Omega$ (otherwise the machine would be non-deterministic). Therefore H_s removes symbols from the end of the stack like a finite state machine with a single terminal state (q_s) and without any arc emerging from the terminal state. Hence, there is a regular prefix L_s such that H_s can be written as (R_{-Ls}, I). If we denote by (R_{us}, I) the computation along p_s'', then we have

$$\psi_{q.q'} = (\bigcup_{j=1}^{n} \{R_{-Lj}R_{uj}\}, I) .\ ⑥$$

Obviously, the converse implication is also true (since an arc labelled by a function of the form $(R_{-L}R_u, L_{-\sigma})$ can be replaced by a diagram whose arcs are labelled by 1-stack SMS X-machine type functions). Thus proposition 2.4.2.3. has been established.

Corollary 2.4.2.4.
The class of regular stack stream X-machine acceptors accept exactly the deterministic context-free languages.

Example 2.4.2.5.
Let $\Sigma = \{a, b, c\}$ and

$$L = \{a^{i1}ba^{i2}b \ldots a^{ir-1}ba^{ir}c^s a^{ir-s+1} |\ r \geq 1,\ 1 \leq s \leq r,\ i_j \geq 1 \text{ for all } 1 \leq j \leq r\}.$$

We know that L is deterministic context-free, but it is not a real time stack language.

A deterministic regular stack stream X-machine that accepts L is the following.

1. $\Omega = \{x, y\}$
2. $m_o = y$
3. The state transition diagram is presented in figure 2.19; q_o is the initial state and q_4 the terminal state and $\varphi_1, \ldots \varphi_6$ are defined as follows:

$\varphi_1 = (R_x, L_{-a})$
$\varphi_2 = (R_y, L_{-b})$
$\varphi_3 = (I, L_{-c})$
$\varphi_4 = (R_{-L}; L_{-c})$
$\varphi_5 = (R_{-x}, L_{-a})$
$\varphi_6 = (R_{-y}, L_{-\delta})$,

where $L' = \{x^n y |\ n \geq 0\}$.

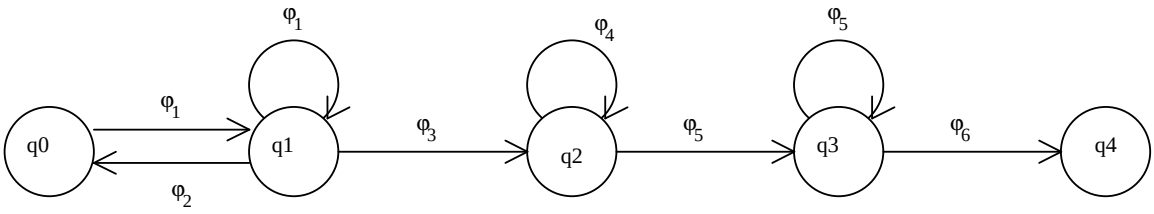


Figure 2.19.

We saw that a regular stack stream X-machine used stack functions of the form $R_{-L}R_u$, where R_{-L} removes symbols from the stacks like a finite state machine and R_u adds a finite number of symbols at the end of the stack. An alternative (but similar in principle) approach would be to use an additional set of symbols (called the set of markers) to mark each stack location and to allow the machine to remove all the stack symbols until a certain marker is encountered. We will call these stream X-machine with markers and we will show that they are equivalent to the class of regular stack stream X-machines.

2.4.3. Stack stream X-machines with markers.

The *stack stream X-machine with markers* (denoted *MStack S X-machine*) is a generalisation of the tabulator machine introduced by Cole, [9]. The model is somewhat similar to a regular stack stream X-machine, since the machine can remove in a single move an unlimited number of characters from the stack. Unlike the regular stack stream X-machine though, the stack stream X-machine with markers uses an additional set of symbols to mark the locations where the machine stops removing characters from the stack. Therefore, each stack symbol will be a pair (a, B_i), where a is a pushdown symbol and B_i is a set of markers.
Cole, [9], claims that a tabulator machine is equivalent to a pushdown automaton. However, there are some apparent gaps in the proof provided. We prove this result by showing the equivalence between MStack S X-machines and RStack S X-machines.

Definition 2.4.3.1.
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, Q_0, T, m_0)$ a stream X-machine. If

1. $M = \Omega^*$ (hence $X = \Gamma^* \times \Omega^* \times \Sigma'^*$),

with $\Omega = A \times \mathcal{P}(B)$, where A and B are finite alphabets and $\mathcal{P}(B)$ is the powerset of B. A is called the set of *push-down symbols* and B the set of *markers*.

2. $\Phi = \{R_\gamma | \gamma \in \Gamma\} \times \Phi_M \times \{L_{-\sigma} | \sigma \in \Sigma'\}$,

where each $\phi \in \Phi_M$ is determined by a pair of partial functions (z, w),

$z: A \times \mathcal{P}(B) \rightarrow B$,

$w: A \times \mathcal{P}(B) \rightarrow ((A \times \mathcal{P}(B)) \cup \{1\})$ (1 is the empty string)

such that

$\forall u \in \Omega^*$, $u = ((a_0, B_0) \dots (a_k, B_k))$, ϕ is defined by:

$$\phi(u) = \begin{cases} (a_0, B_0) \dots (a_m, B_m)\, w(a_k, B_k), \text{ if } w(a_k, B_k) \neq \varnothing \text{ and } z(a_k, B_k) \neq \varnothing \\ \varnothing, \text{ otherwise} \end{cases}$$

where

$$m = \max \{n \leq k \mid z(a_k, B_k) \in B_n\},$$

then $\mathcal{M}$ is called a *stack stream X-machine with markers*.

A *stack generalised stream X-machine with markers* (denoted *MStack GS X-machines*) will have the type

$$\{R_g \mid g \in \Gamma^*\} \times \Phi_M \times \{L_{-\sigma} \mid \sigma \in \Sigma'\},$$

with Φ_M defined as above.

Therefore, z indicates the location where the machine stops removing symbols from the stack (i.e. when the marker indicated by z is found) and w indicates the pushdown symbol and the set of markers which have to be added on top of the stack; if $w(a_k, B_k) = 1$ nothing is added on top of the stack.

To prevent the stacks becoming empty, the bottom-most location is required to contain (a_o, B_o), with $B_o = B$, the entire set of markers.

Example 2.4.3.2.

Let $\Sigma = \{a, b, c\}$ and

$$L = \{a^{i1}ba^{i2}b \dots a^{ir-1}ba^{ir}ca^{ir-s+1} \mid r \geq 1,\ 1 \leq s \leq r,\ i_j \geq 1 \text{ for all } 1 \leq j \leq r\}.$$

A deterministic stack stream X-machine with markers that accepts L is the following.

1. $\Sigma = \{a, b, c\}$, $\Gamma = \varnothing$. Hence $X = \Omega^* \times \Sigma^*$, where Ω is the stack alphabet.
2. $A = \{x, y\}$; $B = \{s, t, u, v\}$.

Hence $\Omega = \{x, y\} \times \{\varnothing, \{s\}, \{t\}, \{u\}, \{v\}, \{s, t\}, \{s, u\}, \{s, v\}, \{t, u\}, \{t, v\}, \{u, v\}, \{s, t, u\}, \{s, t, v\}, \{t, u, v\}, \{s, t, u, v\}\}$.

3. $m_o = (y, B)$.
4. The transition diagram is presented in figure 2.20; q_o is the initial state and q_4 the terminal state and $\varphi_1, \dots \varphi_8$ are defined as follows:

$\varphi_1 = (\phi_1, L_{-a})$
$\varphi_2 = (\phi_2, L_{-a})$
$\varphi_3 = (\phi_3, L_{-b})$
$\varphi_4 = (\phi_4, L_{-c})$
$\varphi_5 = (\phi_5, L_{-c})$
$\varphi_6 = (\phi_6, L_{-a})$
$\varphi_7 = (\phi_7, L_{-a})$

$\varphi_8 = (\phi_8, L_{-\delta})$

with ϕ_i determined by z_i and w_i, i = 1, ..., 8.

$z_i: \Omega \to B$ and $w_i: \Omega \to (\Omega \cup \{1\})$ are partial functions defined by:

Note: For the sake of simplicity the values for which the partial functions above are not defined will not be listed.

$z_1(y, \{s, t, u, v\}) = s$	$w_1(y, \{s, t, u, v\}) = (x, \{s, u\})$
$z_1(y, \{s, u\}) = s$	$w_1(y, \{s, u\}) = (x, \{t, u\})$
$z_1(y, \{t, u\}) = t$	$w_1(y, \{t, u\}) = (x, \{s, u\})$
$z_1(y, \{s, v\}) = s$	$w_1(y, \{s, v\}) = (x, \{t, v\})$
$z_1(y, \{t, v\}) = t$	$w_1(y, \{t, v\}) = (x, \{s, v\})$
$z_2(x, \{s, u\}) = s$	$w_2(x, \{s, u\}) = (x, \{t, u\})$
$z_2(x, \{t, u\}) = t$	$w_2(x, \{t, u\}) = (x, \{s, u\})$
$z_2(x, \{s, v\}) = s$	$w_2(x, \{s, v\}) = (x, \{t, v\})$
$z_2(x, \{t, v\}) = t$	$w_2(x, \{t, v\}) = (x, \{s, v\})$
$z_3(x, \{s, u\}) = t$	$w_3(x, \{s, u\}) = (y, \{s, v\})$
$z_3(x, \{t, u\}) = s$	$w_3(x, \{t, u\}) = (y, \{t, v\})$
$z_3(x, \{s, v\}) = t$	$w_3(x, \{s, v\}) = (y, \{s, u\})$
$z_3(x, \{t, v\}) = s$	$w_3(x, \{t, v\}) = (y, \{t, u\})$
$z_4(x, \{s, u\}) = t$	$w_4(x, \{s, u\}) = 1$
$z_4(x, \{t, u\}) = s$	$w_4(x, \{t, u\}) = 1$
$z_4(x, \{s, v\}) = t$	$w_4(x, \{s, v\}) = 1$
$z_4(x, \{t, v\}) = s$	$w_4(x, \{t, v\}) = 1$
$z_5(x, \{s, u\}) = v$	$w_5(x, \{s, u\}) = 1$
$z_5(x, \{t, u\}) = v$	$w_5(x, \{t, u\}) = 1$
$z_5(x, \{s, v\}) = u$	$w_5(x, \{s, v\}) = 1$
$z_5(x, \{t, v\}) = u$	$w_5(x, \{t, v\}) = 1$
$z_6(x, \{s, u\}) = s$	$w_6(x, \{s, u\}) = 1$
$z_6(x, \{t, u\}) = t$	$w_6(x, \{t, u\}) = 1$
$z_6(x, \{s, v\}) = s$	$w_6(x, \{s, v\}) = 1$
$z_6(x, \{t, v\}) = t$	$w_6(x, \{t, v\}) = 1$
$z_7(x, \{s, u\}) = t$	$w_7(x, \{s, u\}) = 1$
$z_7(x, \{t, u\}) = s$	$w_7(x, \{t, u\}) = 1$
$z_7(x, \{s, v\}) = t$	$w_7(x, \{s, v\}) = 1$
$z_7(x, \{t, v\}) = s$	$w_7(x, \{t, v\}) = 1$
$z_8(y, \{s, t, u, v\}) = s$	$w_8(y, \{s, t, u, v\}) = 1$

$z_8(y, \{s, u\}) = s$ $\quad w_8(y, \{s, u\}) = 1$
$z_8(y, \{t, u\}) = t$ $\quad w_8(y, \{t, u\}) = 1$
$z_8(y, \{s, v\}) = s$ $\quad w_8(y, \{s, v\}) = 1$
$z_8(y, \{t, v\}) = t$ $\quad w_8(y, \{t, v\}) = 1$

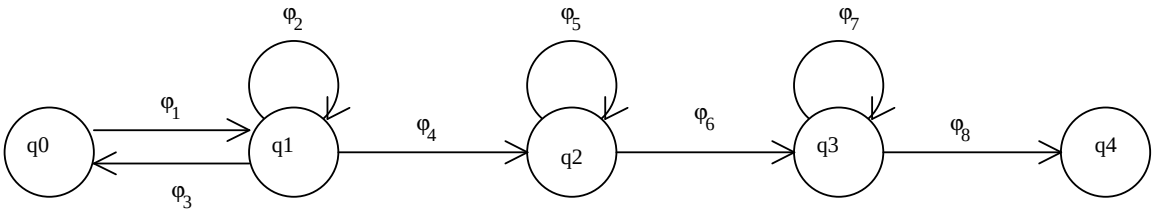
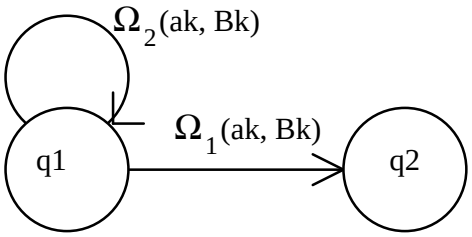


Figure 2.20.

Another interpretation of each $\phi \in \Phi_M$ is the following.
For each topmost symbol of the stack (a_k, B_k), each $\phi \in \Phi_M$ removes symbols from the memory stack like a finite state machine with two states and with no arc leaving the terminal state (see figure 2.21) and adds the last symbol removed and another (possibly empty) symbol $w(a_k, B_k)$.



$\Omega_1(a_k, B_k) = \{(c, D) \in \Omega | z(a_k, B_k) \in D\}$, $\Omega_2(a_k, B_k) = \Omega - \Omega_1(a_k, B_k)$.
q_1 is the initial state, q_2 is the terminal state.

Figure 2.21.

Thus, as with the regular stack model, finite automata can be used to compute each $\phi \in \Phi_M$; unlike these, though, the MStack S X-machine uses more than one (a finite number, one for each stack symbol) finite automaton to compute each $\phi \in \Phi_M$, but these automata are simpler.

We now prove that deterministic MStack S X-machines and RStack S X-machines are equivalent.

2.4.4. RStack S X-machines and MStack S X-machines are equivalent.

Proposition 2.4.4.1.
For any deterministic stack stream X-machine with markers $\mathcal{M}$ there exists a deterministic regular stack stream X-machine $\mathcal{M}'$ such that $\mathcal{M}$ and $\mathcal{M}'$ are equivalent (i.e. they compute the same function).

Proof:
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, T, m_0)$ be a MStack S X-machine with $M = \Omega^*$ and $\Omega = A \times \mathcal{P}(B)$.
Then $\forall\ \varphi \in \Phi$, φ will have the form

$$\varphi = (\phi_\Gamma, \phi, \phi_\Sigma),$$

with $\phi_\Gamma \in \{R_\gamma | \gamma \in \Gamma\}$, $\phi_\Sigma \in \{L_{\text{-}\sigma} | \sigma \in \Sigma'\}$ and $\phi \in \Phi_M$ (Φ_M is as defined in definition 2.4.3.1).

We show that $\mathcal{M}$ can be simulated by the RStack S X-machine

$$\mathcal{M}' = (\Sigma, \Gamma, Q, M, \Phi', F', q_0, T, m_0)$$

in which each $\varphi \in \Phi$, $\varphi = (\phi_\Gamma, \phi, \phi_\Sigma)$ is replaced by a set of functions

$$\{\varphi_1', ..., \varphi_k'\} \subseteq \Phi' \text{ (i.e. } \Phi' \text{ is the type of the RStack S X-machine } \mathcal{M}'),$$

with $\varphi_i' = (\phi_\Gamma, \phi_i', \phi_\Sigma)$, i = 1, ...k.
Therefore the 'next state' function of $\mathcal{M}'$ will be defined by

$$F'(q, \varphi_i') = F(q, \varphi),\ \forall\ q \in Q,\ i = 1, ..., k.$$

The mapping $\phi \rightarrow \{\phi_1', ... \phi_k'\}$ is defined below.

Let ϕ: $(A \times \mathcal{P}(B))^* \rightarrow (A \times \mathcal{P}(B))^*$ defined by

$$z: A \times \mathcal{P}(B) \rightarrow B \text{ and } w: A \times \mathcal{P}(B) \rightarrow A \times \mathcal{P}(B).$$

Let $(a, C) \in A \times \mathcal{P}(B)$ and $\phi|(a, C)$ be the restriction of ϕ to $\Omega^*\{(a, C)\}$. If $z(a, C) = \varnothing$ or $w(a, C) = \varnothing$ then $\phi|(a, C)$ is the empty function. Otherwise, we prove that $\phi|(a, C)$ can be written as a finite union of functions of the form $R_{\text{-}L}R_u$ ($L \subseteq \Omega^*$ is a non-empty regular prefix and $u \in \Omega^*$). We have the following two cases.

i) Let $b = z(a, C)$. If $b \in C$, then let $L = \{a, C\}$ and $u = w(a, C)$. Then

$$\phi|(a, C) = R_{\text{-}L}R_u.$$

ii) Otherwise let

$$D = \{B' \subseteq B |\ b \notin B'\} \text{ and } E = \{B' \subseteq B |\ b \in B'\}.$$

Then, we define

$$L_{xy} = (a, C)\ (\sum_{a' \in A, B' \in D} (a', B'))^*(x, y)\ \forall\ x \in A,\ y \in E.$$

It is clear that L_{xy} is a regular prefix. We also define

$$u_{xy} = (x, y)\ w(a, C).$$

It is easy to verify that $\phi|(a, C)$ can be written as $\bigcup_{x \in A, y \in E} R_{\text{-}L_{xy}}R_{u_{xy}}$.

Since $\phi = \bigcup_{a\in A, C\in\wp(B)} \phi|(a, C)$, ϕ can be simulated by a finite set $\{\phi_1', ... \phi_k'\}$ of functions of the form $R_{-L}R_u$.

From the construction of $\mathcal{M}'$ it is clear that if $\mathcal{M}$ is deterministic, $\mathcal{M}'$ is deterministic. ⑥

The converse implication is also true. First we need the following technical result.

Lemma 2.4.4.2.
Let $\mathcal{M}$ be a regular stack stream X-machine. Then, by a suitable enrichment of the stack alphabet, Φ_M can be considered as

$\Phi_M = \{R_{-L}R_u | u \in \{1\} \cup \Omega \cup \Omega^2$, where $L \subseteq \Omega^*$ is a regular prefix, $L \neq \emptyset, \{1\}\}$
$\cup \{R_u | u \in \{1\} \cup \Omega\}$.

Also, we can consider that the initial memory value of a $\mathcal{M}$ can be chosen as $m_o \in \{1\} \cup \Omega$.

Proof:
Without loss of generality we shall consider that Φ_M is finite (this is because only a finite number of functions are used to label the arcs in $\mathcal{M}$). Then

$\Phi_M = \{R_{-L_i}R_{ui} | i = 1, ..., n\}$.

Let $m = \max(\{|u_i| \mid i = 1, ..., n\} \cup \{|m_o|\})$. Then let Ω' be an alphabet with the same number of elements as the set $\Omega \cup ... \cup \Omega^m$ and

$h: \Omega \cup ... \cup \Omega^m \rightarrow \Omega'$

be a bijective function.
In what follows h(x) will be denoted by [x], $\forall x \in \Omega \cup ... \cup \Omega^m$.

The idea we shall employ is to transform $\mathcal{M}$ into a RStack S X-machine $\mathcal{M}'$ with the stack alphabet Ω'. The state set, the initial state and the set of terminal states will remain unchanged. The initial memory value of $\mathcal{M}'$ will be $[m_o]$. The next state function of $\mathcal{M}'$ will be obtained from the next state function of $\mathcal{M}$ by replacing any arc

$q \xrightarrow{(\phi, I)} q', \phi \in \Phi_M$,

with a number of arcs

$q \xrightarrow{(\phi_1, I)} q', ..., q \xrightarrow{(\phi_k, I)} q'$

such that $\psi = \bigcup_{i=1}^{k} \phi_k$ simulates ϕ.

Then, for each

$\phi \in \{R_{-L_i}R_{ui} | i = 1, ..., n\}$

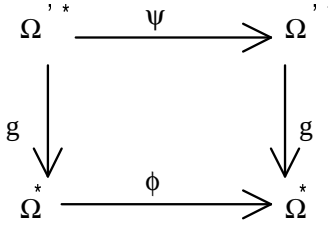
we have to find the corresponding function ψ such that ψ simulates ϕ and ϕ can be written as a finite union of partial functions of the form $R_{-L'}R_{u'}$, with $L' \subseteq \Omega'^*$ a non-empty regular prefix and $u' \in \{1\} \cup \Omega'$. In other words, let

$g: \Omega'^* \rightarrow \Omega^*$

be the surjective morphism induced by

$g([u]) = u, \forall\ u \in \Omega'$.

Then for each ϕ as above, we have to find ψ such that the diagram



commutes and ψ can be written as a finite union of the functions as described above.

Let $\phi = R_{-L}R_u$. Then we have the following two cases.

i) If $L = \{1\}$, then

$\psi = R_{[u]}$, $[u] \in \{1\} \cup \Omega'$ (i.e. if $u = 1$, $[u]$ denotes the empty string 1).

ii) Otherwise $L \neq \{1\}$. Then let $s' \in \Omega'^*$, $s' = [y_1] \ldots [y_k]$. Then we define ψ by

$\psi = h\, R_{[u]}$,

where

$$h(s') = \begin{cases} [y_1] \ldots [y_{i-1}]\,[y_i'], \text{ if } \exists\ v \in L \text{ such that } y_i \ldots y_k = y_i'\, \mathrm{rev}(v) \\ \qquad\qquad \text{with } |y_i'| < |y_i| \\ \varnothing, \text{ otherwise} \end{cases}$$

It is clear that h is well defined and $gf = f'g$.

Therefore h removes the string $[y_{i+1}] \ldots [y_k]$; $[y_i]$ is either removed (i.e. if $y_i \ldots y_k = \mathrm{rev}(v)$) or replaced by $[y_i']$, $0 < |y_i'| < |y_i|$, if $y_i \ldots y_k = y_i'\, \mathrm{rev}(v)$.

Since $|y_i'| \in \{0, \ldots, m-1\}$, h can be written as a union of functions of the form

$$\bigcup_{|y|<m} R_{-L'y}R_{[\mathrm{rev}(y)]},$$

where

$L'_1 = g^{-1}(L)$

. . . . . .

$L'_{x1 \ldots x_j} = g^{-1}(Lx_1 \ldots x_j) - (g^{-1}(Lx_1 \ldots x_{j-1})[x_j] \cup g^{-1}(Lx_1 \ldots x_{j-2})[x_{j-1}][x_j] \cup$
$g^{-1}(Lx_1 \ldots x_{j-2})[x_{j-1}x_j] \cup g^{-1}(Lx_1 \ldots x_{j-3})[x_{j-2}][x_{j-1}][x_j] \cup$

$g^{-1}(Lx_1 \ ... \ x_{j-3})[x_{j-2}x_{j-1}][x_j] \cup g^{-1}(Lx_1 \ ... \ x_{j-3})[x_{j-2}][x_{j-1}x_j] \cup$
$g^{-1}(Lx_1 ... x_{j-3})[x_{j-2}x_{j-1}x_j] \cup ... \cup g^{-1}(L)[x_1 ... x_j])$

$\forall x_1,... x_j \in \Omega$, $j < m$.

Aside. An example will illustrate the construction.

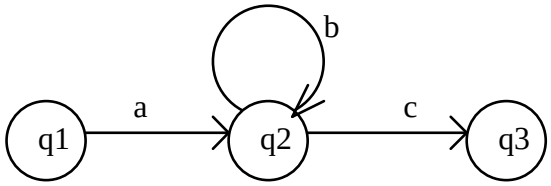
Let $\Omega = \{a, b, c\}$ and $m = 2$. Then

$\Omega' = \{[a], [b], [c], [aa], [ab], [ac], [ba], [bb], [bc], [ca], [cb], [cc]\}$.

If $L = \{ab^*c\}$, then

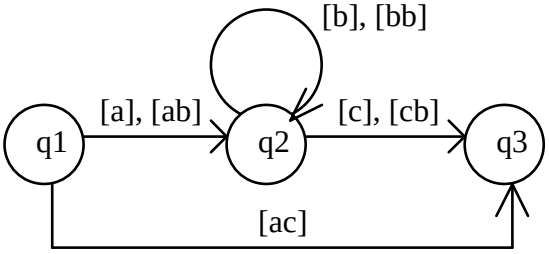
$g^{-1}(L) = ([a] + [ab]) ([b] + [bb])^* ([c] + [bc]) + [ac]$.

The minimal automata $\mathcal{A}$ and $\mathcal{A}'$ of L and $L_1' = g^{-1}(L)$, respectively, are represented in figures 2.22 and 2.23.



The minimal automaton of L

Figure 2.22.



The minimal automaton of L_1'

Figure 2.23.

Then

$L_a' = g^{-1}(\{ab^*ca\}) - g^{-1}(\{ab^*c\})[a]$
$L_b' = g^{-1}(\{ab^*cb\}) - g^{-1}(\{ab^*c\})[b]$
$L_c' = g^{-1}(\{ab^*cc\}) - g^{-1}(\{ab^*c\})[c]$ •

Since L is a regular prefix, all the sets L'_y are regular prefixes (this follows from lemma 4.2.1.7). Therefore, the function ψ can be written as

$$\psi = (\bigcup_{|y|<m} R_{-L'_y} R_{[rev(y)]}) R_{[u]},$$

where all of the sets L'_y are regular prefixes.

The equivalence of the two machines ($\mathcal{M}$ and $\mathcal{M}'$) follows since for all input sequences $x \in \Sigma^*$, x takes $\mathcal{M}$ from its initial state and initial memory value into the state q and memory value s iff x takes $\mathcal{M}'$ from its initial state and its memory value into the state q and the memory value s', such that $s' \in g^{-1}(s)$; also, the output sequences produced

by the two machines when they receive x are the same (this follows using a simple inductive argument). ⑥

Proposition 2.4.4.3.
For any deterministic regular stack stream X-machine $\mathcal{M}$, there exists a deterministic stack stream X-machine with markers $\mathcal{M}$' such that $\mathcal{M}$ and $\mathcal{M}$' are equivalent.

Proof:
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, T, m_0)$ be a deterministic RStack S X-machine with stack alphabet Ω. Hence

$$\Phi = \{\varphi = (\phi_\Gamma, \phi, \phi_\Sigma) | \ \phi_\Gamma: \Gamma^* \to \Gamma^*, \phi_\Sigma: \Sigma^* \to \Sigma^*, \phi: \Omega^* \to \Omega^*,$$
$$\phi_\Gamma \in \{R_\gamma | \ \gamma \in \Gamma\}, \phi_\Sigma \in \{L_{-\sigma} | \ \sigma \in \Sigma'\}, \phi \in \Phi_M\}.$$

Without loss of generality we shall consider that Φ_M is finite. From lemma 2.4.4.2, it follows that Φ_M will have the form

$$\Phi_M = \{R_{-L_i} R_{u_i} | \ L_i \neq \{1\}, u_i \in \{1\} \cup \Omega \cup \Omega^2, i = 1,..., n\} \cup \{R_u | \ u \in \{1\} \cup \Omega\}.$$

We also assume that $m_0 \in \{1\} \cup \Omega$.

Let

$$\ell_i = \{L_i x^{-1} | \ x \in \Omega^*, L_i x^{-1} \neq \emptyset\}, i = 1, ..., n.$$

Since L_i is a regular language, ℓ_i is finite. Let $k_i = card(\ell_i)$, i = 1, ..., n.

We shall prove that $\mathcal{M}$ can be simulated by a MStack S X-machine $\mathcal{M}$'. The proof will employ the following ideas:

Let $y \in \Omega^*$ be a stack value and $L \subseteq \Omega^*$ be a non-empty regular prefix. Then $R_{-L}(y) \neq \emptyset$ iff $\exists\ t \in \Omega^*, \xi \in L$ such that $y = t\ rev(\xi)$. Hence $y = t\varpi$, with $\varpi = rev(\xi)$. Therefore the end part of the stack is the reverse of an element of L. Hence $R_{-L}(y) \neq \emptyset$ iff y can be written as $y = t\varpi$, with $rev(\varpi) \in L$ (or $\{1\} \in L\ rev(\varpi)^{-1}$).

On the other hand, since $\forall\ x, x' \in \Omega^*$, $L\ rev(xx')^{-1} = L\ rev(x)^{-1} rev(x')^{-1}$, we can keep track of all values $L\ rev(\varpi)^{-1} \neq \emptyset$, where ϖ is the end part of the stack (i.e. $\exists\ t \in \Omega^*$ such that $s = t\varpi$).

Construction of $\mathcal{M}$'.
We construct $\mathcal{M}' = (\Sigma, \Gamma, Q', M', \Phi', F', q_0', T', m_0')$ a MStack S X-machine as follows.

1. $Q' = Q \times (\{1\} \cup \Omega \cup \Omega^2)$, $q_0' = (q_0, m_0)$, $T' = T \times (\{1\} \cup \Omega \cup \Omega^2)$.

2. $M' = \Omega'^*$, with the stack alphabet $\Omega' = A \times \mathcal{P}(B)$, where the set of pushdown symbols and the set of markers are defined as follows.
Let $\{B_i\}_{i=1, ..., n}$ be a family of sets such that

$$card(B_i) = k_i + 1 \text{ and } \bigcap_{i=1}^{n} B_i = \emptyset.$$

Then the set of markers is

$$B = \bigcup_{i=1}^{n} B_i.$$

The set A will be

$$A = A_1 \times \ldots \times A_n,$$

where

$$A_i = \{g_i |\ g_i: \mathcal{L}_i \to B_i \text{ is a partial function}\}.$$

Notation: In order to simplify the notation in what follows we define, for an arbitrary finite set X, the random function Rd[X]: ($\mathcal{P}$(X) - X) $\to$ X, where for each proper subset of X, X', Rd[X](X') = x, where x$\in$ X and x $\notin$ X' is chosen at random.

3. The initial memory value is

$$m_0' = ((g_{10}, \ldots, g_{n0}), B),$$

where g_{i0} is the initial value of g_i, i = 1, ..., n, defined by:

$$g_{i0}(L_i) = Rd[B_i](\varnothing) \text{ and undefined for } \mathcal{L}_i - \{L_i\}.$$

The initial value of the markers set is $B_0 = B$.

4. The next state function F' of $\mathcal{M}$' follows from the next state function F of $\mathcal{M}$ by applying the following transformations:

a) Each arc from $\mathcal{M}$ of the form

$$q \xrightarrow{\varphi} q', \text{ with } \varphi = (\phi_\Gamma, \phi, \phi_\Sigma),\ \phi = R_{-L_i}R_{u_i},\ L_i \neq \{1\},\ u_i \in \{1\} \cup \Omega \cup \Omega^2,$$

is replaced in $\mathcal{M}$' by the set of arcs

$$(q, v) \xrightarrow{\varphi_v'} (q', u_i),\ v \in \{1\} \cup \Omega \cup \Omega^2.$$

b) Each arc from $\mathcal{M}$ of the form

$$q \xrightarrow{\varphi} q', \text{ with } \varphi = (\phi_\Gamma, \phi, \phi_\Sigma),\ \phi = R_u,\ u \in \{1\} \cup \Omega,$$

is replaced in $\mathcal{M}$' by two sets of arcs as follows:

$$(q, v) \xrightarrow{\varphi_v'} (q', u),\ v \in \{1\} \cup \Omega,$$

$$(q, v) \xrightarrow{\varphi_v'} (q', \text{rear}(v)\ u),\ v \in \Omega^2.$$

Note: head and rear are defined in definition 2.3.1.

For $\varphi = (\phi_\Gamma, \phi, \phi_\Sigma)$ and $v \in \{1\} \cup \Omega \cup \Omega^2$ we denoted by φ_v' the function $\varphi_v' = (\phi_\Gamma, \phi_v', \phi_\Sigma)$, where each ϕ_v' is determined by a pair of functions (z, w),

$$w: A \times \mathcal{P}(B) \to ((A \times \mathcal{P}(B)) \cup \{1\}) \text{ and } z: A \times \mathcal{P}(B) \to B.$$

The definitions of these functions (and therefore the definition of ϕ_v') depend on the following two cases.

Case A. $\phi = R_{-L_i}R_{u_i},\ L_i \neq \{1\},\ u_i \in \{1\} \cup \Omega \cup \Omega^2.$

Let $a \in A$ and $C \in \mathcal{P}(B)$, $a = (g_1, .., g_n)$, $C = \bigcup_{i=1}^{n} C_i$ with $C_i \subseteq B_i$, the pushdown symbol and the markers set respectively of the topmost location of $\mathcal{M}$'.

i). We define w by:

$$w(a, C) = \begin{cases} ((g_1' .., g_n'), b'), \text{ if } v \in \Omega^2 \text{ and } rear(v) \in L_i \\ 1, \text{ if } \exists L \in dom(g_i), \text{ such that } rev(v) \in L \\ \varnothing, \text{ otherwise} \end{cases} \quad (1)$$

where $b' = \{b_1', ..., b_n'\}$ with $b_j' \in B_j$, $j = 1, ..., n$.
The expressions of g_j', b_j', $j = 1, ..., n$, are given below.

Let $dom(g_j) = \{L_{j1}, ..., L_{jr}\}$, $r \leq k_j$. Then

$$dom(g_j') = \{L_{j1}head(v)^{-1}, ..., L_{jr}head(v)^{-1}\} - \varnothing \cup \{L_j\}, \quad (2)$$

$$g_j'(L_{js}head(v)^{-1}) = g_j(L_{js}), \text{ if } L_{js}head(v)^{-1} \neq \varnothing, s = 1, ..., r, \quad (3)$$

$$g_j'(L_j) = Rd[B_j](\{g_j(L_{js}) | L_{js}head(v)^{-1} \neq \varnothing, s = 1, ..., r\}), \quad (4)$$

$$b_j' = g_j'(L_j). \quad (5)$$

ii). z is defined by:

$$z(a, C) = \begin{cases} Rd[C](\varnothing), \text{ if } v \in \Omega^2 \text{ and } rear(v) \in L_i \\ g_i(L), \text{ if } \exists L \in dom(g_i) \text{ such that } rev(v) \in L \\ \varnothing, \text{ otherwise} \end{cases} \quad (6)$$

Case B. $\phi = R_u$, $u \in \{1\} \cup \Omega$.
Then ϕ_v' is determined by the pair of functions z, w as follows.

i). We define w by:

$$w(a, C) = \begin{cases} ((g_1' .., g_n'), b'), \text{ if } v \in \Omega \cup \Omega^2 \\ 1, \text{ otherwise} \end{cases} \quad (1')$$

where $b' = \{b_1', ..., b_n'\}$ with $b_j' \in B_j$, $j = 1, ..., n$.
The expressions of g_j', b_j', $j = 1, ..., n$, are given below.

Let $dom(g_j) = \{L_{j1}, ..., L_{jr}\}$, $r \leq k_j$. Then

$$\text{dom } (g_j') = \{L_{j1}\text{head}(v)^{-1}, ..., L_{jr}\text{ head}(v)^{-1}\} - \varnothing \cup \{L_j\}, \quad (2')$$

$$g_j'(L_{js}\text{head}(v)^{-1}) = g_j(L_{js}), \text{ if } L_{js}\text{head}(v)^{-1} \neq \varnothing, s = 1, ..., r, \quad (3')$$

$$g_j'(L_j) = Rd[B_j](\{g_j(L_{js})|\ L_{js}\text{head}(v)^{-1} \neq \varnothing, s = 1, ..., r\}), \quad (4')$$

$$b_j' = g_j'(L_j). \quad (5')$$

ii). z is defined by:

$$z(a, C) = Rd[C](\varnothing) \quad (6')$$

From the construction of $\mathcal{M}'$ it follows that the set of markers of each location except the bottom-most will contain n markers, one from each B_i.

Let us now explain how $\mathcal{M}'$ simulates $\mathcal{M}$. If a certain input sequence will take $\mathcal{M}$ into the state q with the stack value y, then the same input sequence will take $\mathcal{M}'$ into a state (q, v), $v \in \{1\} \cup \Omega \cup \Omega^2$, such that v is the end part of the stack value s, i.e. ($\exists\ \tau \in \Omega^*$ such that $y = \tau v$). The function g_i of the topmost location will associate a marker with each non-empty set $L_i x^{-1}$, where x is the end part of τ reversed (i.e. $\exists\ t \in \Omega^*$ such that $\tau = t\ \text{rev}(x)$). Also, each stack location (apart from the bottommost) will be marked by n markers, one from each B_i. When $\mathcal{M}$ adds or removes symbols from its stack, $\mathcal{M}'$ will add updated versions of the functions g_i and a new set of markers or removes symbols from its stack such that at any time $|\upsilon| = |\tau| + 1$, where υ is the stack value of $\mathcal{M}'$.

The fact that $\mathcal{M}$ is well defined and that $\mathcal{M}$ and $\mathcal{M}'$ are equivalent follows from the following three lemmas.

Lemma 2.4.4.3.1 (inside proof).
Let $\upsilon \in \Omega'^*$ the stack value of $\mathcal{M}'$, $\upsilon = \omega_0\omega_1... \omega_k$, with $\omega_0, \omega_1,..., \omega_k \in \Omega'$ (i.e. ω_0 is the value of the bottommost location of the stack). Let $(g_1^j,..., g_n^j)$ be the pushdown symbol of ω_j and $C^j = \{b_1^j,..., b_n^j\}$, $b_i^j \in B_i$, the set of markers of ω_j. Then, for i = 1, ..., n we have:
1. $\exists\ \tau \in \Omega^*$, $|\tau| = |\upsilon| - 1$ (i.e. the length of υ without the bottommost location is the length of τ), such that

$$\text{dom } g_i^j = \{L_i(\text{rev}(x))^{-1} \neq \varnothing |\ \exists\ t \in \Omega^* \text{ such that } x_j = tx\},\ j = 0, ..., k,$$

where $x_j \in \Omega^*$ satisfies $|x_j| = j$ and $x_j^{-1}\tau \neq \varnothing$ (i.e. $\exists\ d_j \in \Omega^*$ such that $\tau = x_j d_j$).
2. g_i^j is well defined, j = 0, ..., k.
3. g_i^j is injective, j = 0, ..., k.
4. Let $j \in$ 0, ..., k-1 and let $d_j = x_j^{-1}\tau$. If $L_i(\text{rev}(d_j))^{-1} \neq \varnothing$, then $b_i^j \notin C^r$, r = j+1,.., k. Also, $b_i^j = g_i^k(L_i(\text{rev}(d_j))^{-1})$.

Proof:
1. It follows by induction on k using the definition of ϕ_V' (i.e. the expressions of g_j' function of g_j, as given in relations (3), (4) and (3'), (4')). We use the fact that

$L_i\ rev(x\alpha)^{-1} = L\ rev(x)^{-1} rev\ (\alpha)^{-1}$, $\forall\ x \in \Omega^*$, $\alpha \in \Omega$.

2. Since L_i is a prefix, the sets $\{L_i(rev(x))^{-1} \neq \emptyset | \exists\ t \in \Omega^*$ such that $x_j = tx\}$ will be disjoint. Hence g_i^j is well defined.
3. It follows by induction on k using the same definition and relations as 1.
4. Let $d_j = \alpha_1 ... \alpha_{k-j}$, with $\alpha_1,... \alpha_{k-j} \in \Omega$. From 1 it follows that

$L_i(\alpha_1)^{-1}... (\alpha_{r-j})^{-1} \in$ dom g_i^r, r = j+1, ..., k.

Since g_i^r is injective it follows that

$g_i^r(L_i(\alpha_1)^{-1}... (\alpha_{r-j})^{-1}) \neq g_i^r(L_i)$.

Hence, since $b_i^r = g_i^r(L_i)$ (relations (5) and (5')) it follows that $b_i^j \notin C^r$, r = j+1, ..., k. By induction on the length of d_j it follows that $g_i^k(L_i(rev(d_j))^{-1}) = g_i^j(L_i)$. Hence $b_i^j = g_i^k(L_i(rev(d_j))^{-1})$. □

From lemma 2.4.4.3.1, 2, it follows that g_i^j is well defined for any stack location. Hence w is well defined. Also, z is well defined since the two conditions from relation (6) yield disjoint domains.

We can now prove the equivalence of $\mathcal{M}$ and $\mathcal{M}$'. This is done using the following two lemmas.

Lemma 2.4.4.3.2 (inside proof).
Let $s \in \Sigma^*$ be an input sequence. If s takes $\mathcal{M}$ from its initial state and memory value to the state $q \in Q$ and stack value $y \in \Omega^*$, while producing an output sequence $g \in \Gamma^*$, then there exist $v \in \{1\} \cup \Omega \cup \Omega^2$, $\tau \in \Omega^*$ and $\upsilon \in \Omega'^*$ such that:
1. $\tau v = y$.
2. s takes $\mathcal{M}$' from its initial state and memory value to the state (q, v) and the memory value υ while producing the same output sequence g.
3. υ satisfies $|\upsilon| = |\tau| + 1$.
4. Let $\upsilon = \omega_o\omega_1... \omega_k$, with $\omega_0, \omega_1,..., \omega_k \in \Omega'$ (i.e. ω_o is the value of the bottommost location of the stack). Let $j \in \{0, ..., k\}$ and $(g_1^j .., g_n^j)$ be the pushdown symbol of ω_j. Then:

dom $g_i^j = \{L_i(rev(x))^{-1} \neq \emptyset | \exists\ t \in \Omega^*$ such that $x_j = tx\}$, i = 1,..., n,

where $x_j \in \Omega^*$ satisfies $|x_j| = j$ and $x_j^{-1}\tau \neq \emptyset$ (i.e. $\exists\ d_j \in \Omega^*$ such that $\tau = x_jd_j$).

Proof:
We prove this lemma by induction on the length of the input sequence s. For s = 1 statements 1 - 4 are true (i.e. we take $\tau = 1$, $v = m_o$ (the initial memory value of $\mathcal{M}$) and υ containing only the bottommost location of the stack of $\mathcal{M}$').

Let us assume that 1 - 4 are true for s and let $\sigma \in \Sigma$ be such that there exists an arc

$$q \xrightarrow{\varphi} q', \varphi = (R_\gamma, \phi, L_{\neg\sigma}), \gamma \in \Gamma, \phi \in \Phi_M$$

such that $\phi(y) \neq \emptyset$. Then let $\phi(y) = y'$. We prove that there exist $v' \in \{1\} \cup \Omega \cup \Omega^2$, $\tau' \in \Omega^*$ and $\upsilon' \in \Omega'^*$ such that:

1. $\tau' v' = y'$.
2. There exists an arc

$$(q, v) \xrightarrow{\varphi_{v'}} (q', v'), \varphi_{v'} = (R_\gamma, \phi_{v'}, L_{\neg\sigma}),$$

in $\mathcal{M}'$ such that $\phi_{v'}(\upsilon) = \upsilon'$.

3. υ' satisfies $|\upsilon'| = |\tau'| + 1$.
4. Let $\upsilon' = \omega_0'\omega_1' \ldots \omega_k'$, with $\omega_0', \omega_1', \ldots, \omega_k' \in \Omega'$. Let $j \in \{0, \ldots, k'\}$ and $(g_1^j, \ldots, g_n^j)$ the pushdown symbol of ω_j'. Then:

$$\text{dom } g_i^j = \{L_i(\text{rev}(x))^{-1} \neq \emptyset \mid \exists\, t \in \Omega^* \text{ such that } x_j' = tx\},\ i = 1, \ldots, n,$$

where $x_j' \in \Omega^*$ satisfies $|x_j'| = j$ and $x_j^{-1}\tau' \neq \emptyset$ (i.e. $\exists\, d_j \in \Omega^*$ such that $\tau' = x_j' d_j$).

We have the following cases:

A. $\phi = R_u$, $u \in \{1\} \cup \Omega$.

i) If $v = 1$, then $v' = u, \tau' = \tau$ and $\upsilon' = \upsilon$.

ii) If $v \in \Omega$, then $v' = u$ and $\tau' = \tau v$. $\upsilon' = \upsilon\omega$, where $\omega \in \Omega'$ has the pushdown symbol $(g_1, \ldots, g_n)$ with

$$\text{dom } g_i = \{L_i(\text{rev}(x))^{-1} \neq \emptyset \mid \exists\, t \in \Omega^* \text{ such that } \tau v = tx\},\ i = 1, \ldots, n.$$

iii) If $v \in \Omega^2$, then $v' = \text{rear}(v)\, u$ and $\tau' = \tau\, \text{head}(v)$. $\upsilon' = \upsilon\omega$, where $\omega \in \Omega'$ has the pushdown symbol $(g_1, \ldots, g_n)$ with

$$\text{dom } g_i = \{L_i(\text{rev}(x))^{-1} \neq \emptyset \mid \exists\, t \in \Omega^* \text{ such that } \tau\, \text{head}(v) = tx\},\ i = 1, \ldots, n.$$

It can be easily verified that the statements 1 - 4 are true.

B. $\phi = R_{\neg L_i} R_{u_i}$, $L_i \neq \{1\}$, $u_i \in \{1\} \cup \Omega \cup \Omega^2$.

In this case $R_{\neg L_i}(y) \neq \emptyset$ iff $\exists\, \varpi \in L_i$, $a \in \Omega^*$ such that $y = a\, \text{rev}(\varpi)$. Hence $\tau v = a\, \text{rev}(\varpi)$. So we have the following cases:

i) $v \in \Omega^2$ and $\text{rear}(v) = \text{rev}(\varpi)$. Hence $\text{rear}(v) \in L_i$. In this case we have $v' = u_i$ and $\tau' = \tau\, \text{head}(v)$. $\upsilon' = \upsilon\omega$, where $\omega \in \Omega'$ has the pushdown symbol $(g_1, \ldots, g_n)$ with

$$\text{dom } g_i = \{L_i(\text{rev}(x))^{-1} \neq \emptyset \mid \exists\, t \in \Omega^* \text{ such that } \tau\, \text{head}(v) = tx\},\ i = 1, \ldots, n.$$

It can be easily verified that the statements 1 - 4 are true.

ii) $\exists\, b \in \Omega^*$ such that $\text{rev}(\varpi) = bv$ and $\tau = ab$. Hence $\text{rev}(v)\, \text{rev}(b) \in L_i$. Therefore $\text{rev}(v) \in L_i\, \text{rev}(b)^{-1}$. Let $L = L_i\, \text{rev}(b)^{-1}$. Then $\phi_{v'}$ removes symbols from υ until the marker $g_i^k(L)$ is encountered. Using lemma 2.4.4.3.1, 4, and relation (6) it follows that $\phi_{v'}(\upsilon) = \upsilon'$, such that υ' is obtained by removing $|b|$ symbols from the end of υ. Therefore conditions 1 - 4 are satisfied for $\tau' = a$ and $v' = u_i$. □

Lemma 2.4.4.3.3 (inside proof).

Let $s \in \Sigma^*$ be an input sequence. We assume that s takes $\mathcal{M}'$ from its initial state and memory value to the state $(q, v) \in Q \times (\{1\} \cup \Omega \cup \Omega^2)$ and stack value $\upsilon \in \Omega^*$ while producing an output sequence $g \in \Gamma^*$. Then there exists $\tau \in \Omega^*$ such that:
1. s takes $\mathcal{M}'$ from its initial state and memory value to the state q and the memory value τv while producing the same output sequence g.
2. $|\upsilon| = |\tau| + 1$.
3. Let $\upsilon = \omega_0\omega_1 ... \omega_k$, with $\omega_0, \omega_1, ..., \omega_k \in \Omega'$. Let $j \in \{0, ..., k\}$ and $(g_1^j .., g_n^j)$ the pushdown symbol of ω_j. Then:

$$\text{dom } g_i^j = \{L_i(\text{rev}(x))^{-1} \neq \varnothing \mid \exists\, t \in \Omega^* \text{ such that } x_j = tx\},\ i = 1, ..., n,$$

where $x_j \in \Omega^*$ satisfies $|x_j| = j$ and $x_j^{-1}\tau \neq \varnothing$ (i.e. $\exists\, d_j \in \Omega^*$ such that $\tau = x_j d_j$).

Proof:
By induction, in a manner similar to lemma 2.4.4.3.2. The induction step consists of showing that if $\mathcal{M}'$ follows an arc

$$(q, v) \xrightarrow{\varphi_v'} (q', v'),\ \varphi_V' = (R_\gamma, \phi_V', L_{-\sigma}), \text{ with } \phi_V'(\upsilon) = \upsilon'$$

then $\mathcal{M}'$ will follow the arc

$$q \xrightarrow{\varphi} q',\ \varphi = (R_\gamma, \phi, L_{-\sigma}) \text{ with } \phi(\tau v) = \tau' v',$$

where τ' satisfies conditions 1-3 with respect to υ' and v'. ⑥

Corollary 2.4.4.4.
The class of stack stream X-machines with markers accept exactly the deterministic context-free languages.

2.4.5. Assessment of the stream X-machine model

At first sight the stream X-machine model appeared to be too restrictive to cope with many common applications. We investigated the k-stack stream X-machines and we saw that there were context-free languages that could not be accepted by such machines.

But the power of the model increases by using more complex ϕ's. We have presented two classes of stream X-machines (i.e. RStack S X-Machines and MStack S X-machines) that accept exactly the deterministic context-free languages. These stream X-machines use ϕ's that can be computed using simpler stream X-machines (i.e. finite automata). In this way we can build even more complex stream X-machine models. For example, we can define a *k regular stack (generalised) stream X-machine* (denoted k-RStack (G)S X-machine) like a regular stack machine but with k stacks instead of one. These machines will accept a larger class of languages (i.e. it will include both the deterministic context-free language and real time stack languages).

Also, the hierarchy of (G)S X-machines can be continued further using the idea employed in the construction of RStack S X-machines. Following this approach we can construct (G)S X-machines with one stack (i.e. $M = \Omega^*$) and the set of functions that operate on that stack

$$\Phi_M = \{R_{-L}R_u \mid u \in \Omega^*, L \subseteq \Omega^* \text{ is a non-empty prefix}\},$$

where L is a language for which we already have a computational model (e.g. L is a language accepted by a RStack S X-machine or a k-RStack S X-machine).

2.5. Conclusions and further work.

We have investigated two particular classes of X-machines. Firstly, a very general class, the straight-move stream X-machines, that can model any type of Turing computation. However, the generality of the model makes it an unlikely basis for a theoretical testing methodology.

Secondly, we investigated a more restrictive class, the stream X-machines. These are the machines on which our testing method will be based. We have seen that, although more restrictive than the straight-move stream X-machine, the stream X-machine can be used to build models of complex computational devices. The approach employed was a hierarchical one, in which a machine uses basic operations that have been specified by simpler S X-machines. This appears to fit the approach used in practice for developing specifications of complex systems (i.e. a complex specification is built in terms of simpler ones). On the other hand, the use of the stream X-machine as a specification tool has been tried on numerous systems, some of them fairly complex (see Laycock [41], Howe [35], Chiu [5]) and the model appears to cope successfully with a wide range of applications.

Further work could concentrate on extending the hierarchy of stream X-machines. Following the approach used in this chapter, if $\mathcal{L}$ is a class of languages accepted by a certain class of (G)S X-machines, then the next level in the hierarchy will be the class of (G)S X-machines with one stack (i.e. $M = \Omega^*$) and the set of functions that operate on that stack

$$\Phi_M = \{R_{-L}R_u \mid u \in \Omega^*, L \subseteq \Omega^* \text{ is a non-empty prefix}, L \in \mathcal{L}\}.$$

An alternative (and more powerful) approach we can employ for building hierarchies of GS X-machines could be as follows. Let Σ a finite alphabet and $f: \Sigma^* \to \Sigma^*$ a partial function such dom f is a non-empty prefix. We denote by R_{-f} a (partial) function $R_{-f}: \Sigma^* \to \Sigma^*$
defined by:

$$\lceil \; s\,rev(v)^{-1}rev(f(v)), \text{ if } \exists\, v \in \text{dom } f \text{ such that } s\,rev(v)^{-1} \neq \varnothing,$$

$$R_{-f}(s) = \begin{cases} \\ \varnothing, \text{ otherwise} \end{cases}$$

If f is a function computed by a GS X-machine $\mathcal{M}$ and $s \in \Sigma^*$, then $R_{-f}(s)$ operates as though $\mathcal{M}$ starts removing inputs /adding outputs from the end of the sequence s. This is similar to the definition of R_{-L}, but in this case $\mathcal{M}$ does not only remove input symbols, it replaces them with outputs.

For example, let $\Sigma = \{a, b, c, x, y, z\}$ and $f: \Sigma^* \to \Sigma^*$ be the partial function with

dom f = {ab*c}

defined by

$f(ab^ic\} = xy^iz$, $i \geq 0$.

Then:

1. If the value of the stack is s = aacbba, then $R_{-f}(s)$ = aazyyx.
2. If the value of the stack is s = aacbbab, then $R_{-f}(s)$ is not defined.

Then, we can say that a GS X-machine is a *(n+1)-complex Stack GS X-machine*, $n \geq 0$, if its memory is $M = \Omega^*$ and its type is

$$\Phi = \{R_\gamma | \gamma \in \Gamma\} \times \Phi_M \times \{L_{-\sigma} | \sigma \in \Sigma'\},$$

where

$\Phi_M = \{R_{-f} | f: \Omega^* \to \Omega^*$ is a partial function computed by a n-complex Stack GS X-machine such that dom f is a non-empty regular set$\}$.

We shall also define a *0-complex Stack GS X-machine* to be any machine that computes the partial function $f: \Sigma^* \to \Sigma^*$ with

dom f = {1}, defined by f(1) = 1.

Let us denote by M_n the class of n-complex Stack GS X-machines and F_n the class of functions they compute. Then M_1 will be the class of finite state machines with outputs (and with the property that when an input symbol is read the machine can produce a sequence of output symbols, not only one). However, F_2 is larger then the class of functions computed by RStack GS X-machines. In fact F_2 includes all of the functions computed by k-RStack GS X-machines. This follows easily from the following two remarks.

1). Let $L \subseteq \Sigma^*$ a non-empty regular prefix and $u \in \Sigma^*$. Then $R_{-L}R_u = R_{-f}$, where $f: \Sigma^* \to \Sigma^*$ is a partial function with dom f = L defined by

$f(s) = rev(u), \forall s \in L$.

It is easy to show that $f \in F_1$.

2). Let $\phi: (\Sigma^*)^k \to (\Sigma^*)^k$ a partial function defined by

$\phi = (R_{-L1}R_{u1}, ..., R_{-Lk}R_{uk})$,

with $L_1, ..., L_k \subseteq \Sigma^*$ non-empty regular prefixes and $u_1, ..., u_k \in \Sigma^*$. Using 1) it follows that ϕ can be written as

$$\phi = (R_{-f_1}, ..., R_{-f_k})$$

for some partial functions $f_i: \Sigma^* \to \Sigma^*$, $f_i \in F_1$, i = 1, ..., k. Then the function ϕ operating on k stacks can be simulated by a function $\psi = R_{-g}$, where $g: \Omega^* \to \Omega^*$ is a function that operates on a single stack. The values of the k stacks will be placed on a single stack separated by some special symbols $a_1, ..., a_k$ such that $a_i \notin \Sigma$, i = 1, ..., k (i.e. if $s_1, s_2, ..., s_k$ are the values of the k stacks, then the value of the new stack will be $a_1s_1a_2s_2...a_ks_k$).
Then the stack alphabet will be $\Omega = \Sigma \cup \{a_1, ..., a_k\}$. The partial function g will have

$$\text{dom } g = \text{dom } f_k \, \Sigma^* \{a_k\} \, ... \, \text{dom } f_2 \, \Sigma^* \{a_2\} \, \text{dom } f_1 \, \Sigma^* \{a_1\}$$

and will be defined by

$$g(x_ky_ka_k...x_1y_1a_1) = f_k(x_k)y_ka_k...f_1(x_1)y_1a_1, \forall \, x_1 \in \text{dom } f_1, ..., x_k \in \text{dom } f_k, y_1, ..., y_k \in \Sigma^*.$$

Since dom f_1, ..., dom f_k are prefixes, g is well defined. It can be shown easily that $g \in F_1$.

Obviously, this hierarchical approach requires much further investigation. Two interesting questions that arise are:

How far can we get using this approach? We know that any language accepted by a n-complex stack GS X-machine is recursive (see proposition 2.4.3). But how close to the set of recursive languages can we get?

Does this hierarchy continue infinitely, is there any n such that
{n-complex Stack GS X-machines} = {(n+1)-complex Stack GS X-machines}?
This approach could provide us with a natural way of classifying context-sensitive (or even recursive) languages.

Another interesting area to explore is whether the X-machine model could be used to classify non-Turing computable functions, i.e. having a certain set Φ of noncomputable functions and $\Xi(\Phi)$ the set of functions computed by the X-machines with type Φ, under what conditions Φ is strictly included in $\Xi(\Phi)$?

Acknowledgement:
Theorem 2.3.9 is a generalisation of a result stating that a Turing machine *acceptor* can be simulated by a 2-PDA (i.e. a PDA with two stacks) (see Cohen [8]). The concept of S X-machine was introduced by Laycock, [41]. The concept of MStack S X-machine is a generalisation of the tabulator machine introduced by Cole, [9].