Regular languages: From automata to logic and back

Madhavan Mukund
Chennai Mathematical Institute
Siruseri 603103, India
Email: madhavan@cmi.ac.in

1 Introduction

The mathematical foundations of computer science predate the development of the first electronic computers by a few decades. The early 20th century saw pioneering work by Alonzo Church, Alan Turing and others towards the formulation of an abstract definition of computability. This effort was in the context of one of David Hilbert's challenges, the *Entscheidungsproblem*, which asked whether all mathematical truths could be deduced "mechanically" from the underlying axioms. Church and Turing showed, independently, that it is impossible to decide algorithmically whether statements in arithmetic are true or false, and thus a general solution to the Entscheidungsproblem is impossible.

This early research on computability yielded the Turing machine, a compelling abstraction that captures the essence of an automatic computational device. A Turing machine has a finite set of control states that guide its behaviour, but it has access to an infinite "tape" on which it can record intermediate results during a computation. This makes a Turing machine unrealizable in practice, since it requires an external storage device with unbounded capacity.

If we restrict our computing devices to use an arbitrarily large, but fixed, amount of external storage, the situation changes drastically. The number of distinct configurations of such a machine—all combinations of control states and contents of the external storage—now becomes bounded. This yields a new abstract model, finite automata, whose properties were investigated by Stephen Cole Kleene, Michael Rabin, Dana Scott and others in the decades following Turing's work.

Initially, the focus was on characterizing the computational power of this restricted class of machines. However, around the late 1950's, a deep connection was identified between finite automata and mathematical logic, which has since played a central role in computer science.

The aim of this article is to provide a quick introduction to this elegant interplay between finite automata and mathematical logic. We begin with a self-contained presentation of finite automata. In Section 3, we describe regular expressions, an alternative notation for describing the computations that finite automata can perform. Some shortcomings of regular expressions motivate us to turn to logical specifications in monadic second order logic, introduced in Section 4. The next two sections prove the central theorem due to Julius Richard

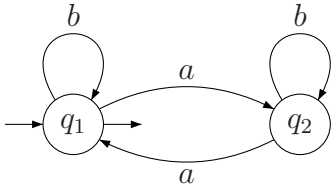


Figure 1: A finite automaton

Büchi, Calvin Elgot and Boris Trakhtenbrot, showing that finite automata are expressively equivalent to specifications in monadic second order logic. We conclude with a discussion where we provide some pointers to the literature.

2 Finite automata

A finite automaton $\mathcal{A}$ consists of a finite set of states Q with a distinguished *initial* state $q_{in} \in Q$ and a set of *accepting* or *final* states $F \subseteq Q$.

An automaton $\mathcal{A}$ processes sequences of abstract actions drawn from a finite *alphabet* Σ. The operation of $\mathcal{A}$ is given by a *transition function* $\delta : Q \times \Sigma \to Q$. A transition of the form $\delta(q, a) = q'$ means that whenever $\mathcal{A}$ is in state q, the action a takes it to state q'.

Thus, an automaton $\mathcal{A}$ over Σ is fully described by its four components (Q, q_{in}, F, δ). A useful way to visualize an automaton is to represent it as a directed graph, where the vertices are the states and the labelled edges correspond to the transition function, as shown in Figure 1. In our pictures, we denote the initial state with an unlabelled incoming arrow and final states with unlabelled outgoing arrows, so $q_{in} = q_1$ and $F = \{q_1\}$ in this example.

The sequences of actions processed by finite automata are typically called *input words* and we usually say that an automaton *reads* such an input word. To process an input word $w = a_1 a_2 \dots a_n$, the automaton $\mathcal{A}$ starts in the initial state q_{in} and traces out a path labelled by w in the state-transition graph. More formally, a *run* of $\mathcal{A}$ on $w = a_1 a_2 \dots a_n$ is an alternating sequence of states and actions $q_0, a_1, q_1, a_2, q_2, \dots, q_{n-1}, a_n, q_n$ such that $q_0 = q_{in}$ and for each $1 \leq i \leq n$, $\delta(q_{i-1}, a_i) = q_i$.

The automata we have defined are *deterministic*: for each state q and action a, $\delta(q, a)$ fixes uniquely the state to which $\mathcal{A}$ moves on reading a in state q. This enforces that $\mathcal{A}$ has exactly one run on each input word. We will relax this condition later.

An automaton $\mathcal{A}$ *accepts* or *recognizes* an input w if it reaches a final state after processing w—in other words, the last state of the run of $\mathcal{A}$ on w belongs to the set F. The set of words accepted by $\mathcal{A}$ is called the *language* of $\mathcal{A}$ and is denoted $L(\mathcal{A})$.

Example 2.1 *Consider the automaton in Figure 1. On input aba, the automaton goes through the sequence of states $q_1q_2q_2q_1$, while on input babb, the automaton goes through the sequence of states $q_1q_1q_2q_2q_2$. Since $F = \{q_1\}$, the automaton accepts aba and does not accept babb. This automaton accepts all words with an even number of a's—we can show by induction that after reading any word with an even (respectively, odd) number of a's, the automaton will be in state q_1 (respectively, q_2).*

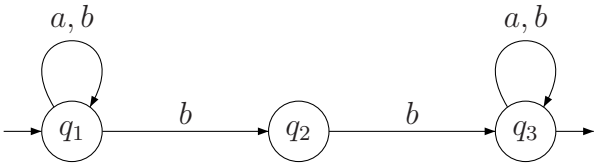


Figure 2: A nondeterministic finite automaton

2.1 Properties of regular languages

A set of words X is said to be a *regular language* if there exists a finite automaton $\mathcal{A}$ such that $X = L(\mathcal{A})$. Regular languages are closed with respect to boolean operations: complementation, union and intersection. Complementation is defined with respect to the set of all words over Σ—this set is normally denoted Σ^*. Thus, given a set of words X, its complement $\overline{X}$ is the set $\Sigma^* \setminus X$. Note that the set Σ^* includes the empty word, ε, of length zero.

Proposition 2.2 *Let X and Y be regular languages. Then $\overline{X}$, $X \cup Y$ and $X \cap Y$ are also regular languages.*

Proof To show that $\overline{X}$ is regular, let $\mathcal{A}$ be an automaton that accepts X. If we interchange the final and non-final states of $\mathcal{A}$ we obtain an automaton $\mathcal{B}$ that accepts every input word w that $\mathcal{A}$ does not accept and vice versa. Thus $L(\mathcal{B}) = \overline{X}$.

To show that $X \cap Y$ is regular, let $\mathcal{A}_X = (Q_X, q^X_{in}, F_X, \delta_X)$ and $\mathcal{A}_Y = (Q_Y, q^Y_{in}, F_Y, \delta_Y)$ be automata that accept X and Y respectively. We construct the direct product $\mathcal{A}_{XY}$ with states $Q_X \times Q_Y$, initial state (q^X_{in}, q^Y_{in}), final states $F_X \times F_Y$ and transition function δ_{XY}, such that for all $q_x \in Q_X$, $q_y \in Q_Y$ and $a \in \Sigma$, $\delta_{XY}((q_x, q_y), a) = (\delta_X(q_x, a), \delta_Y(q_y, a))$. The automaton $\mathcal{A}_{XY}$ simulates $\mathcal{A}_X$ and $\mathcal{A}_Y$ in parallel and accepts an input w provided both $\mathcal{A}_X$ and $\mathcal{A}_Y$ accept w. Hence, $L(\mathcal{A}_{XY}) = X \cap Y$.

Since $X \cup Y = \overline{\overline{X} \cap \overline{Y}}$, it follows that $X \cup Y$ is also regular. We can also give a direct construction for this case by setting the final states of $\mathcal{A}_{XY}$ to $(F_X \times Q_Y) \cup (Q_X \times F_Y)$. □

2.2 Nondeterminism

An important generalization of our definition of finite automata is to relax the restriction that state transitions are deterministic. In a *nondeterministic* finite automaton, given a state q and an action a, the automaton may change to one of several alternative new states. Formally, a nondeterministic finite automaton (NFA) is a tuple $\mathcal{A} = (Q, q_{in}, F, \Delta)$ where Q, q_{in} and F are as before and the transition function $\Delta : Q \times \Sigma \to 2^Q$ maps each pair (q, a) to a subset of states. Notice that it is possible for $\Delta(q, a)$ to be empty, in which case $\mathcal{A}$ gets "stuck" on reading a in state q and cannot finish processing the input, and hence cannot accept it. We can still represent automata as directed graphs with labelled edges—an example of a nondeterministic automaton is shown in Figure 2. Here, for instance, $\Delta(q_1, b) = \{q_1, q_2\}$ and $\Delta(q_2, a) = \emptyset$.

As with deterministic automata, a nondeterministic automaton $\mathcal{A}$ processes an input w by tracing out a path labelled by w in the state transition graph: a *run* of $\mathcal{A}$ on $w = a_1 a_2 \ldots a_n$

is an alternating sequence of states and actions $q_0, a_1, q_1, a_2, q_2, \ldots, q_{n-1}, a_n, q_n$ such that $q_0 = q_{in}$ and for each $1 \leq i \leq n$, $q_i \in \Delta(q_{i-1}, a_i)$. Notice that a run *must* read the entire input.

Clearly, nondeterminism permits an automaton to have multiple runs (or even no run at all!) on an input. To accept an input, it is sufficient to have one good run: an automaton $\mathcal{A}$ accepts an input w if there exists a run on which $\mathcal{A}$ reaches a final state after processing w.

Example 2.3 *Consider the automaton in Figure 2. This automaton starts in q_1 and must reach q_3 to accept an input. The automaton can stay in states q_1 and q_3 indefinitely on any sequence of a's and b's. To get from q_1 to q_3, the automaton must read two consecutive b's. Thus, this automaton accepts any word w of the form w_1bbw_2 with two consecutive b's. On such an input, the automaton "guesses" after w_1 that it should move to q_2 on the next b, and eventually ends up in q_3 where it can remain till the end of w_2. If w does not have this form, it cannot make such a guess: after any b that takes the automaton from q_1 to q_2, the next input is an a, on which there is no transition from q_2, so the automaton gets "stuck" and there is no accepting run.*

It turns out that nondeterminism does not add any computational power to finite automata. Given a nondeterministic finite automaton, we can construct an equivalent deterministic automaton that simulates all possible runs of the original nondeterministic automaton on each input. This is known as the subset construction because the deterministic simulation maintains the set of states that the NFA can be in at each point. We do not need the details of the subset construction: all we need is the fact that nondeterministic automata also accept only regular languages, so we can demonstrate that a language is regular by exhibiting either a deterministic or a nondeterministic finite automaton that accepts it.

3 Regular Expressions

As we have seen, the overall behaviour of an automaton is defined in terms of the language that it accepts. Thus, two automata that accept the same language are essentially equivalent from our point of view, though they may vary greatly in their internal structure.

Since languages are the basic notion of behaviour, it is useful to be able to abstractly specify them, rather than having to present concrete implementations in terms of automata. For instance, it would be useful to have a precise notation to write descriptions of languages corresponding to the informal descriptions we have seen, such as "all words with an even number of a's", or "all words with two consecutive b's"

Ideally, the specifications that we write in this manner should be realizable—that is, they should describe regular languages and there should be an effective algorithm to construct an automaton from such a specification. Even better would be to show that every regular language can be described within such a specification language.

One way of abstractly defining languages is to use algebraic expressions. The set of *regular expressions* over an alphabet Σ is defined inductively as follows:

- The symbols $\emptyset$ and ε are regular expressions.
- Every letter $a \in \Sigma$ is a regular expression.

- If e and f are regular expressions, so are $e + f$, $e \cdot f$ and e^*.

The operator $+$ denotes union while $\cdot$ denotes concatenation. We usually omit $\cdot$ and write ef for $e \cdot f$. The operation * represents iteration and is called the Kleene star.

Each regular expression corresponds to a language. To describe how to associate languages with regular expressions, we need to define some operations on languages.

Let X and Y be languages. Then:

- $XY = \{xy \mid x \in X, y \in Y\}$, where xy represents concatenation of words: if $x = a_1 a_2 \dots a_m$ and $y = b_1 b_2 \dots b_n$ then $xy = a_1 a_2 \dots a_m b_1 b_2 \dots b_n$.
- $X^* = \bigcup_{i \geq 0} X^i$, where $X^0 = \{\varepsilon\}$ and for $i > 0$, $X^i = XX^{i-1}$.

We can now map regular expressions to languages inductively, using the structure of the regular expression. For an expression e, let $L(e)$ denote the language associated with e. We then have the following:

- $L(\emptyset) = \emptyset$ and $L(\varepsilon) = \{\varepsilon\}$.
- For each $a \in \Sigma$, $L(a) = \{a\}$.
- $L(e + f) = L(e) \cup L(f)$, $L(ef) = L(e)L(f)$ and $L(e^*) = (L(e))^*$.

It is not difficult to show that if X and Y are regular languages, then XY and X^* are also regular. Thus, every regular expression describes a regular language.

The converse is also true, but the translation from automata to expressions is harder: finite automata do not have any natural inductive structure and the global behaviour of an automaton cannot easily be decomposed into properties of individual states and transitions. We will not prove this result here. The final section lists standard references where the details can be found.

Together, these translations between regular expressions and finite automata yield the following result.

Theorem 3.1 (Kleene) *A language is regular if and only if it is described by a regular expression.*

Example 3.2 *Recall the automaton in Figure 1 that accepts words with an even number of a's. Any word w with an even number of a's can be decomposed into blocks $w_1 w_2 \dots w_k$ where each w_i has exactly two a's. The expression $b^*ab^*ab^*$ describes all words with exactly two a's: before the first a, between the two a's and after the second a we can have zero or more b's, which is captured by b^*. We use the Kleene star to allow an arbitrary number of blocks of this form, resulting in the expression $(b^*ab^*ab^*)^*$ for this language.*

The regular expression for the automaton in Figure 2 is $(a + b)^*bb(a + b)^*$: any word in the language starts and ends with an arbitrary sequence of a's and b's and contains the sequence bb in between.

We have seen that regular languages are closed under boolean operations such as negation and conjunction. Hence, regular expressions should also be able to cope with these operations. However, for instance, the process of going from a regular expression for a language L to an expression describing the complement of L is far from transparent. Complementation involves a switch of quantifiers: for example, the language of Figure 2 asks if *there exist* two consecutive positions labelled b while the complement language would demand *for every* position labelled b, the positions before and after this position are not labelled b. In general, regular expressions are convenient for expressing *existential* properties but are clumsy for describing *universal* properties—as another example, consider a variant of the language from Figure 2 where we ask that *every* position labelled b occurs as part of a pair of adjacent positions labelled b.

4 Logical Specifications

Another approach to describing sets of words over an alphabet is to use mathematical logic. We will write formulas that are interpreted over words: in other words, the formulas we write can be assigned either the value true or the value false when applied to a given word. For instance, if we write a formula that expresses the fact that *"the first letter in the word is a"*, then this formula will evaluate to true on words of the form *abab* or *aaaa* or *abbbb*, but will be false for words such as ε, *bbba* or *baaaa* that do not start with an a. Thus, a formula φ defines in a natural way a language of words $L(\varphi)$: those words over which φ evaluates to true. This gives us a way to specify languages using logic. Our main goal is to identify a formal logical notation whose formulas capture precisely the regular languages.

4.1 Formulating the logic

We begin with an informal presentation of our logical language. The basic entities we deal with in our logic are positions in words. Thus, the variables that we use in our logic, such as x and y, denote positions. We use predicates to describe the letters that appear at positions. For each $a \in \Sigma$, we have a predicate P_a such that $P_a(x)$ is true in a word w if and only if the letter at position x in w is a. We can compare positions: $x < y$ is true if the position denoted by x is earlier than the position denoted by y.

We use the usual logical connectives $\neg$ (negation), $\vee$ (disjunction) and $\wedge$ (conjunction) to combine formulas, as also derived connectives like $\rightarrow$ (implication), where $\varphi \rightarrow \psi \equiv (\neg\varphi \vee \psi)$ and $\leftrightarrow$ (if and only if), where $\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$. To these logical connectives, we add, as usual, the quantifiers $\exists$ (there exists) and $\forall$ (for all).

For example, consider the language described by the automaton in Figure 2. A formula for this language is

$$\exists x.\exists y.P_b(x) \wedge P_b(y) \wedge \mathit{next}(x,y), \text{ where } \mathit{next}(x,y) \stackrel{\triangle}{=} x < y \wedge \neg\exists z.(x < z \wedge z < y),$$

which asserts that there are two consecutive positions labelled b in the word—the formula $\mathit{next}(x,y)$ expresses that position y occurs immediately after x by demanding that there are no other positions strictly between the two.

The first example we saw, Figure 1, is more complicated. We need to assert that the number of positions labelled a is even. Any formula we write has a fixed set of variables $x_1, x_2, \ldots, x_k$ and can only describe the relationship between k positions in the word at a time. Intuitively speaking, for words that have more than k a's, we cannot expect this formula to assert a property about *all* positions labelled a. This argument can be formalized, but the net result is that we have to enrich our language to directly refer to *sets of positions*.

We introduce a new class of variables, written $X, Y, \ldots$ that represent sets of positions. We write $X(y)$ to denote that the position y belongs to the set X. We are allowed to use the quantifiers $\exists$ and $\forall$ for set variables, just as we do for individual variables.

Let us consider a simpler language than the one in Figure 1—let L_e be the set of all words of even length. To describe this language, we can use a set variable X and restrict the members of X to precisely the odd positions. To do this, we demand that the first position be in X and, for any pair of consecutive positions x and y, x is in X if and only if y is not in X. This captures the fact that X contains every alternate (odd) position, starting from the first one.

Notice that we can use our logical language to define a constant *min* to refer to the first position, for use in such a formula. A formula $\varphi(min)$ that asserts some property involving the first position *min* can be rewritten as $\exists x.\varphi(x) \wedge (\neg\exists z.z < x)$.

Finally, to say that the overall numbers of positions is even, we just assert that the last position is not in X—like *min*, we can define *max*: the position x such that $\neg\exists z.x < z$. Putting this together, we have the following formula,

$$\exists X.X(min) \wedge \forall y, z.(next(y, z) \rightarrow (X(y) \leftrightarrow \neg X(z))) \wedge \neg X(max),$$

where we use the abbreviations *min*, *max* and *next* defined earlier.

From this, it is only a small step to describing the language in Figure 1. We relativize *min*, *max* and *next* to positions labelled a, so $min_a \stackrel{\triangle}{=} P_a(x) \wedge \neg(\exists z.z < x \wedge P_a(z))$ is the smallest a-labelled position, $max_a \stackrel{\triangle}{=} P_a(x) \wedge \neg(\exists z.x < z \wedge P_a(z))$ is the largest a-labelled position and $next_a(x, y) \stackrel{\triangle}{=} P_a(x) \wedge P_a(y) \wedge x < y \wedge \neg(\exists z.x < z \wedge z < y \wedge P_a(z))$ describes when two a-labelled positions are consecutive. We can then modify the formula above to use min_a, max_a and $next_a$ in place of *min*, *max* and *next* and capture the fact that the number of a-labelled positions in a word is even.

To illustrate why logical specifications are more convenient than regular expressions, let us revisit some of the examples from the end of the previous section. Complementing logical specifications is easy—we just negate the specification: if φ describes a set of words, $\neg\varphi$ describes the complement. How about languages that implicitly incorporate universal quantifiers, such as "*every* b occurs as part of a pair of adjacent b's"? Since we have quantifiers at our disposal, we can describe this quite easily, as follows.

$$\forall x.\left(P_b(x) \rightarrow \exists y.\left[(next(y, x) \wedge P_b(y)) \vee (next(x, y) \wedge P_b(y))\right]\right)$$

This formula asserts that every position labelled b is either immediately preceded by or followed by another position labelled b.

4.2 Monadic second order logic

Formally, the logic we have looked at is called monadic second order logic, or MSO. Second order refers to the fact that we can quantify over predicates, not just individual elements, and monadic says that we restrict our attention to 1-place predicates, which are equivalent to sets.

As we have seen, our formulas are built from the following components.

- Individual variables x, y, ... and set variables X, Y, ... denoting positions and sets of positions, respectively. Recall that we have derived constants *min* and *max* that can be used wherever we use individual variables such as x, y, ...
- Atomic formulas:
  - $x = y$ and $x < y$ over positions, with the natural interpretation.
  - $X(x)$, denoting that position x belongs to set X.
  - $P_a(x)$ for each $a \in \Sigma$, denoting that position x is labelled a.
- Complex formulas are built using the usual boolean connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$, together with the quantifiers $\exists$ and $\forall$.

Each (nonempty) word $w = b_1 b_2 \dots b_m$ over the alphabet $\Sigma = \{a_1, a_2, \dots, a_n\}$ provides us a *word model*

$$\langle w \rangle = (\{1, 2, \dots, m\}, <^w, P^w_{a_1}, P^w_{a_2}, \dots, P^w_{a_n})$$

where

- $pos(w) = \{1, 2, \dots, m\}$ is the set of positions in w,
- $<^w$ is the (natural) ordering on the positions in $pos(w)$,
- for each $a_i \in \Sigma$, $P^w_{a_i} = \{j \in pos(w) \mid b_j = a_i\}$ describes the positions where a_i occurs.

We restrict our attention to nonempty words to avoid dealing with some boundary cases that occur when $pos(w) = \emptyset$.

A variable x or X that occurs within the scope of a quantifier is said to be *bound*. Variables that are not bound are said to be *free*. We write $\varphi(x_1, x_2, \dots, x_m, X_1, X_2, \dots, X_n)$ to denote that the set of free variables in φ is at most $\{x_1, x_2, \dots, x_m, X_1, X_2, \dots, X_n\}$.

The meaning of a bound variable is determined by its binding quantifier: for instance, if we write $\forall x.\varphi(x)$, this formula is true if $\varphi(j)$ evaluates to true for all possible choices $j \in pos(w)$ for x. However, for free variables we need to additionally fix how the variable is to be interpreted. Hence, to give meaning to a formula $\varphi(x_1, x_2, \dots, x_m, X_1, X_2, \dots, X_n)$, we need

- A word model $\langle w \rangle$.
- Positions $k_1, k_2, \dots, k_m$ as interpretations of $x_1, x_2, \dots, x_m$.

- Sets of positions $K_1, K_2, \ldots, K_n$ as interpretations of $X_1, X_2, \ldots, X_n$.

We write

$$(\langle w \rangle, k_1, k_2, \ldots, k_m, K_1, K_2, \ldots, K_n) \models \varphi(x_1, x_2, \ldots, x_m, X_1, X_2, \ldots, X_n)$$

to express the fact that φ holds in $\langle w \rangle$ if each x_i is interpreted as k_i and each X_j as K_j. As an abbreviation, we write $\langle w \rangle \models \varphi[k_1, k_2, \ldots, k_m, K_1, K_2, \ldots, K_n]$. We also write $\langle w \rangle \not\models \varphi[k_1, k_2, \ldots, k_m, K_1, K_2, \ldots, K_n]$ to indicate that a formula evaluates to false under an interpretation.

Example 4.1 *Let $w = babaa$ with $\Sigma = \{a, b\}$ and*

1. $\varphi_1(x_1, X_1) \stackrel{\triangle}{=} P_b(x_1) \wedge \forall y.(x_1 < y \wedge X_1(y) \rightarrow P_a(y))$
2. $\varphi_2 \stackrel{\triangle}{=} \exists x.\exists y.(P_b(x) \wedge P_b(y) \wedge \text{next}(x, y))$

Then

1. $\langle w \rangle \models \varphi_1[1, \{2, 4, 5\}]$.
2. $\langle w \rangle \not\models \varphi_1[1, \{2, 3, 4, 5\}]$.
3. $\langle w \rangle \not\models \varphi_2$.

Notice that φ_2 has no free variables, so it can be interpreted directly on a word model $\langle w \rangle$.

A formula with no free variables is called a *sentence*. Sentences can be interpreted directly on word models. We can thus associate with each sentence φ, a language of words $L(\varphi) = \{w \mid \langle w \rangle \models \varphi\}$. Such a language is said to be MSO-definable.

If we restrict our formulas to not use set variables or set quantifiers, the corresponding logic is called first-order logic, or FO. For instance, φ_2 in Example 4.1 is an FO sentence. A language described by an FO sentence is said to be FO-definable.

The main result we are interested in is that MSO-definability is equivalent to regularity.

Theorem 4.2 (Büchi, Elgot, Trakhtenbrot) *A language of nonempty words over Σ is regular if and only if it is MSO-definable. The transformation in both directions is effective.*

To prove this result, we need to extend our definitions so that we associate languages with arbitrary formulas, not just sentences. Let $\varphi(x_1, x_2, \ldots, x_m, X_1, X_2, \ldots, X_n)$ be a formula with free variables. We have seen that models for such formulas are of the form $(\langle w \rangle, k_1, k_2, \ldots, k_m, K_1, K_2, \ldots, K_n)$. For each position $j \in pos(w)$, we record whether $j = k_1, j = k_2, \ldots, j = k_m, j \in K_1, j \in K_2, \ldots, j \in K_n$. This gives us a bit vector of length $m + n$ for each position j. Formally, we can represent $(\langle w \rangle, k_1, k_2, \ldots, k_m, K_1, K_2, \ldots, K_n)$ as a word over the alphabet $\Sigma \times \{0, 1\}^{m+n}$. Clearly, for a sentence, $m = n = 0$, so the word associated with a model is just the word itself.

Example 4.3 *Let $w = babaa$, $k_1 = 1$, $K_1 = \{3\}$ and $K_2 = \{1, 2, 4, 5\}$. Then, we represent $(\langle w \rangle, k_1, K_1, K_2)$ as a word over $\Sigma \times \{0, 1\}^3$ as follows.*

$$\begin{array}{c} w \\ k_1 \\ K_1 \\ K_2 \end{array} \begin{bmatrix} b \\ 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} a \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} b \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} a \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} a \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

5 From automata to logic ...

One half of proving the Büchi-Elgot-Trakhtenbrot theorem consists of associating an MSO-formula with each regular language. In general, let us assume that our regular language is described by an NFA. Given an NFA $\mathcal{A}$, we have to construct a sentence $\varphi_{\mathcal{A}}$ such that $\langle w \rangle \models \varphi_{\mathcal{A}}$ if and only if $w \in L(\mathcal{A})$. To do this, we write a formula that describes successful runs of $\mathcal{A}$.

As a concrete example, let us consider the automaton Figure 2 that accepts all words with two consecutive b's. This automaton has three states $\{q_1, q_2, q_3\}$. For each state q_i, we associate a set variable X_i denoting the positions in which a run of the automaton assumes state q_i. For instance, on the inputs $abbab$, the accepting run $q_1, a, q_1, b, q_2, b, q_3, a, q_3, b, q_3$ would be encoded as follows, omitting the last state.

$$\begin{array}{c} w \\ X_1 \\ X_2 \\ X_3 \end{array} \begin{bmatrix} a \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} b \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} b \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} a \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} b \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

In general, if an automaton $\mathcal{A}$ has k states, we use k set variables $X_1, X_2, \ldots, X_k$ to describe runs of $\mathcal{A}$. We can describe in MSO the constraints that $X_1, X_2, \ldots, X_k$ should satisfy to constitute an accepting run.

(i) The X_i's partition the set of positions—at each position the automaton is in exactly one state.

(ii) At the first position, the automaton is in an initial state.

(iii) For each pair of consecutive positions x, y, the states at x and y and the letter at x describe a valid transition.

(iv) At the final position, there is a transition leading to an accepting state.

Concretely, here is how we write these properties in MSO for the automaton in Figure 2.

(i) $$\begin{aligned} \exists X_1, X_2, X_3.[\forall x.(X_1(x) \vee X_2(x) \vee X_3(x)) \ \wedge \ & \neg\exists x.(X_1(x) \wedge X_2(x)) \\ \wedge \ & \neg\exists x.(X_1(x) \wedge X_3(x)) \\ \wedge \ & \neg\exists x.(X_2(x) \wedge X_3(x))] \end{aligned}$$

In general, we would write

$$\exists X_1, X_2, \ldots, X_k.[\forall x.(X_1(x) \vee X_2(x) \vee \cdots \vee X_k(x)) \wedge \bigwedge_{i \neq j} \neg\exists x.(X_i(x) \wedge X_j(x))].$$

(ii) $X_1(min)$.

(iii) $$\begin{aligned}\forall x, y.next(x,y) \rightarrow\ & X_1(x) \wedge P_a(x) \wedge X_1(y) \ \vee\ X_1(x) \wedge P_b(x) \wedge X_1(y) \\ & X_1(x) \wedge P_b(x) \wedge X_2(y) \ \vee\ X_2(x) \wedge P_b(x) \wedge X_3(y) \\ & X_3(x) \wedge P_a(x) \wedge X_3(y) \ \vee\ X_3(x) \wedge P_b(x) \wedge X_3(y)\end{aligned}$$

In general, we would write

$$\forall x, y.next(x,y) \rightarrow \bigvee_{(q_i,a,q_j)\in\Delta} (X_i(x) \wedge P_a(x) \wedge X_j(y)).$$

(iv) $(X_2(max) \wedge P_b(max)) \vee (X_3(max) \wedge P_a(max)) \vee (X_3(max) \wedge P_b(max))$.

In general, we would write

$$\bigvee_{(q_i,a,q_j)\in\Delta, q_j\in F} X_i(max) \wedge P_a(max).$$

This establishes the first half of the Büchi-Elgot-Trakhtenbrot theorem. For every NFA $\mathcal{A}$ over Σ, we can write a sentence $\varphi_{\mathcal{A}}$ that describes all successful runs of $\mathcal{A}$ such that $L(\mathcal{A}) = L(\varphi_{\mathcal{A}})$. In other words, $\mathcal{A}$ accepts w if and only if $\langle w \rangle \models \varphi_{\mathcal{A}}$.

6 . . . and back

In the converse direction, for each formula $\varphi(x_1, x_2, \ldots, x_m, X_1, X_2, \ldots, X_n)$, we have to construct an automaton $\mathcal{A}_\varphi$ over $\Sigma \times \{0,1\}^{m+n}$ such that $L(\varphi) = L(\mathcal{A}_\varphi)$. We build up $\mathcal{A}_\varphi$ by induction on the structure of φ.

To reduce the number of cases we have to consider, we define a subset $\mathrm{MSO}_{\min}$ of MSO that is expressively equivalent to the original language. In $\mathrm{MSO}_{\min}$, we eliminate individual variables, so we only have set variables and set quantifiers. An individual position x will be represented as a singleton set $\{x\}$. The atomic formulas in $\mathrm{MSO}_{\min}$ are as follows.

- $X \subseteq Y$ and $X \subseteq P_a$.
- $singleton(X)$ — "X is a singleton set".
- $X < Y$ — "$X \neq \emptyset$, $Y \neq \emptyset$ and for each $x \in X$, $y \in Y$, $x < y$".

Once again, complex formulas are built using the boolean connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ and the quantifiers $\exists$ and $\forall$.

It is not difficult to see show that we can express every MSO formula in $\mathrm{MSO}_{\min}$.

- $x = y$ translates as $singleton(X) \wedge singleton(Y) \wedge X \subseteq Y \wedge Y \subseteq X$.
- $x < y$ translates as $singleton(X) \wedge singleton(Y) \wedge X < Y$.
- $X(y)$ translates as $singleton(Y) \wedge Y \subseteq X$.
- $P_a(x)$ translates as $singleton(X) \wedge X \subseteq P_a$.

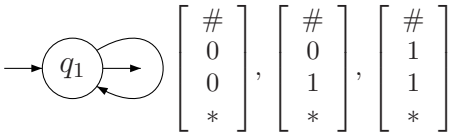
Since we no longer have individual variables, an $\mathrm{MSO}_{\min}$ formula is of the form $\varphi(X_1, X_2, \ldots, X_n)$ and a word model of φ can be encoded as a word over $\Sigma \times \{0,1\}^n$.

Recall that our aim is to associate with each such formula $\varphi(X_1, X_2, \ldots, X_n)$, an automaton $\mathcal{A}_\varphi$ over $\Sigma \times \{0,1\}^n$ such that $L(\varphi) = L(\mathcal{A})$, where $\mathcal{A}_\varphi$ is built up inductively, based on the structure of φ.

We begin with automata corresponding to the atomic formulas in $\mathrm{MSO}_{\min}$.

- $X \subseteq Y$

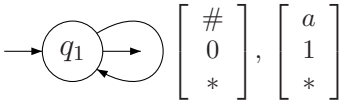
  Given a word w over $\Sigma \times \{0,1\}^n$, our automaton checks that whenever a position belongs to X, it also belongs to Y. Without loss of generality, let $X = X_1$ and $Y = X_2$. Our automaton checks that whenever the X_1 component is 1, so is the X_2 component.



  Here $\#$ denotes any letter in Σ and $*$ denotes an arbitrary bit vector in $\{0,1\}^{n-2}$. This automaton accepts an input provided it does not contain $\begin{bmatrix} \# \\ 1 \\ 0 \\ * \end{bmatrix}$, which would denote a position in the input that belongs to X_1 but not X_2.

- $X \subseteq P_a$

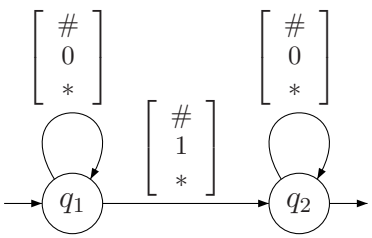
  Again, without loss of generality, let $X = X_1$. Our automaton then looks like this.



  This automaton checks that whenever $X_1 = 1$, the letter read is a. If $X_1 = 0$, the letter read is irrelevant.

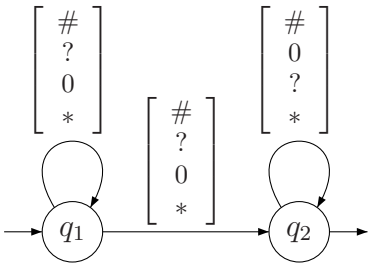
- $\mathit{singleton}(X)$

  As before, we assume that $X = X_1$. We check that there is exactly one 1 along the component corresponding to X_1.



- $X < Y$

  As usual, let $X = X_1$ and $Y = X_2$. We demand that each $x \in X$ comes before each $y \in Y$. For this, we check that all 1's in the X_1 component of the input come before any 1 in the X_2 component. We loop in q_1, insisting that the X_2 component remain 0. We "guess" when we have seen the last 1 in the X_1 component and make a transition to q_2, from which point we insist that we see no more 1's in the X_1 component. In this automaton, ? denotes 0 or 1.



Having translated the atomic formulas, we must look at the connectives used to build up complex formulas. We have already seen that regular languages are closed with respect to Boolean operations, so given automata corresponding to φ and ψ, we can build automata for $\neg\varphi$ and $\varphi \vee \psi$, and hence for all boolean combinations of φ and ψ.

There is one minor complication to take care of in the case of a binary connective such as $\vee$. When we write $\varphi \vee \psi$, the two components will typically have different sets of free variables. In fact, we can always rename the variables so that they have disjoint sets of free variables, so the overall formula is of the form $\varphi(X_1, X_2, \ldots, X_m) \vee \psi(Y_1, Y_2, \ldots, Y_n)$. By induction, we would have automata $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ over $\Sigma \times \{0,1\}^m$ and $\Sigma \times \{0,1\}^n$, respectively, corresponding to φ and ψ. We have to reconcile the alphabets of these two automata when we combine them. We need an alphabet $\Sigma \times \{0,1\}^{m+n}$ that can describe membership in $\{X_1, X_2, \ldots, X_m, Y_1, Y_2, \ldots, Y_n\}$, the set of free variables of the compound formula. Fortunately, it is a simple matter to pad out the alphabets of $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ with extra empty tracks to create equivalent automata over $\Sigma \times \{0,1\}^{m+n}$. We omit the details.

This leaves us with the quantifiers $\exists$ and $\forall$. Since $\forall X.\varphi(X) \equiv \neg\exists X.\neg\varphi(X)$, it suffices to consider just $\exists$.

Consider a formula of the form $\psi(X_2, X_3, \ldots, X_n) \equiv \exists X_1.\varphi(X_1, X_2, \ldots, X_n)$. This formula is true if we can associate some set of positions with X_1 such that $\varphi(X_1, X_2, \ldots, X_n)$ is true. By the induction hypothesis, we already have an automaton $\mathcal{A}_{\varphi(X_1,X_2,\ldots,X_n)}$ over $\Sigma \times \{0,1\}^n$ whose language is the same as that of $\varphi(X_1, X_2, \ldots, X_n)$. From this automaton, we must construct an automaton $\mathcal{A}_{\psi(X_2,X_3,\ldots,X_n)}$ over $\Sigma \times \{0,1\}^{n-1}$ whose language is the same as that of $\psi(X_2, X_3, \ldots, X_n)$.

Let $\mathcal{A}_{\varphi(X_1,X_2,\ldots,X_n)} = (Q, q_{in}, F, \Delta)$. For convenience, we will represent input letters from $\Sigma \times \{0,1\}^k$ as row vectors rather than as column vectors.

We retain (Q, q_{in}, F) and build our new automaton $\mathcal{A}_{\psi(X_2,X_3,\ldots,X_n)} = (Q, q_{in}, F, \Delta')$ such that $(q, [a, \overline{x}], q') \in \Delta'$ if and only if either $(q, [a, 0, \overline{x}], q') \in \Delta$ or $(q, [a, 1, \overline{x}], q') \in \Delta$. In other words, the new automaton "guesses" a value for the X_1 component at each step. It is not difficult to show that there is an accepting run q_{in}, $[a_1, \overline{x_1}]$, q_1, $[a_2, \overline{x_2}]$, q_2, $\ldots$, q_{n-1}, $[a_n, \overline{x_n}]$, q_n of $\mathcal{A}_{\psi(X_2,X_3,\ldots,X_n)}$, where $q_n \in F$, if and only if there is some accepting run q_{in}, $[a_1, b_1, \overline{x_1}]$,

q_1, $[a_2, b_2, \overline{x_2}]$, q_2, $\ldots$, q_{n-1}, $[a_n, b_n, \overline{x_n}]$, q_n of $\mathcal{A}_{\varphi(X_1,X_2,\ldots,X_n)}$, where each $b_j \in \{0,1\}$. The sequence $b_1 b_2 \ldots b_n$ describes the structure of the "witness" X_1 that is required to make $\psi(X_2, X_3, \ldots, X_n)$ true.

This establishes the second half of the Büchi-Elgot-Trakhtenbrot theorem—every MSO formula $\varphi(x_1, x_2, \ldots, x_m, X_1, X_2, \ldots, X_n)$ can be effectively transformed into an automaton $\mathcal{A}_\varphi$ over $\Sigma \times \{0,1\}^{m+n}$ such that $L(\varphi) = L(\mathcal{A}_\varphi)$.

7 Discussion

We have seen that monadic second order logic provides an appealing way to abstractly specify regular languages. Though MSO offers many advantages with respect to other expressively equivalent notations such as regular expressions, there is one glaring disadvantage: the cost of translating MSO formulas to automata. Larry Stockmeyer has shown that there is an unbounded family of MSO formulas such that each formula with n symbols in the family blows up into an automaton with $2^{2^{\cdot^{\cdot^{2}}}}$ states where the height of the tower of exponentials is proportional to n.

Despite this, the connection between MSO and automata has laid the foundation for an important area of computer science called automated verification. The idea is to use logical specifications to describe abstract properties of computational systems and then use automata theory to check if a proposed implementation conforms to the specification. If the specification is given by a formula φ, we can use the theory developed in the chapter to construct an automaton $\mathcal{A}_\varphi$ that accepts $L(\varphi)$, the set of *all* words consistent with φ. Given a potential implementation, described as another automaton $\mathcal{B}$, to check that $\mathcal{B}$ exhibits only "legal" behaviours with respect to φ, it suffices to verify that $L(\mathcal{B}) \subseteq L(\mathcal{A}_\varphi)$. In practice, we move from MSO to other logical formalisms such as temporal logic for which the translation from logic to automata is more tractable.

Further reading

We do not provide pointers to the original papers in these areas by Turing, Kleene, Rabin and Scott, Büchi and others. The original references are standard and the material is so classical by now that it appears in a number of textbooks and surveys.

All standard textbooks on automata theory cover finite automata and their connection to regular expressions in full detail—examples include [2] and [3]. Textbooks that cover the connection between MSO and finite automata are more rare. However, there are excellent survey articles such as [5], which present this connection between MSO and finite automata in a richer context, over infinite words. There are now numerous books that deal with automated verification in the context that we have discussed here, including [1]. For a more concise and accessible introduction, see [4].

Acknowledgment This article borrows heavily from Wolfgang Thomas's lecture notes on *Applied Automata Theory*, informally available from various sources on the Internet. I thank S.P. Suresh for reading through an earlier draft of this article and pointing out omissions and inconsistencies.

References

[1] Edmund M. Clarke, Orna Grumberg and Doron A. Peled: *Model Checking*, MIT Press (2000).

[2] John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman: *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley (2007).

[3] Dexter Kozen: *Automata and Computability*, Springer-Verlag (1997).

[4] Stephan Merz: Model Checking: A Tutorial Overview, in *Modeling and Verification of Parallel Processes*, Lecture Notes in Computer Science, Volume 2067, Springer-Verlag (2001) 3–38.

[5] Wolfgang Thomas: Automata on infinite objects, in Jan van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Volume B*, North-Holland, Amsterdam (1990) 133–191.