Efficient Text Processing

Constructing and Applying Deterministic Finite-State Devices for Text Rewriting Tasks

Stoyan Mihov and Klaus U. Schulz

October 13, 2015

Contents

Part I

Formal Background

Chapter 1

Formal Preliminaries

The aim of this chapter is twofold. First, we collect a set of basic mathematical notions that are needed for the discussions of the following chapters. Second, we have a first - still purely mathematical - look at the central topics of the book: languages, relations and functions between strings, as well as important operations on languages, relations and functions. We also introduce monoids, which later serve to give an abstract view on strings, languages and relations.

1.1 Sets, functions and relations

Sets and Boolean operations for sets (i.e. union, intersection, difference, complement) are introduced as usual. The cardinality of a set A is written $|A|$. Let $n \geq 1$, let $M_1, \ldots, M_n$ be sets. The *Cartesian product* $\prod_{i=1}^{n} M_i$ is introduced as usual. We write M^n if $M_1 = \ldots = M_n = M$.

Special notation conventions for tuples. In what follows, n-tuples $\langle m_1, \ldots, m_n \rangle$ are often written as $\bar{m}$. If $\bar{a} = \langle a_1, \ldots, a_m \rangle$ and $\bar{b} = \langle b_1, \ldots, b_n \rangle$ are tuples, then $\langle \bar{a}, \bar{b} \rangle$ is shorthand for the $(m+n)$-tuple $\langle a_1, \ldots, a_m, b_1, \ldots, b_n \rangle$. Furthermore, if $n \geq 2$ and $\bar{m}$ is as above, then $\bar{m}_{\times i}$ is the $(n-1)$-tuple obtained from $\bar{m}$ by deleting the i-th component.

Definition 1.1.1 Let $M_1, \ldots, M_n$ be sets. Then any set $R \subseteq \prod_{i=1}^{n} M_i$ is called an *n-ary relation*. If $M_1 = \ldots = M_n = M$, then R is called an *n-ary relation over M*. The set $\{\langle m, m \rangle \mid m \in M\}$ is called the *identity* on M and denoted Id_M.

If M is an n-ary relation we often write $R(\bar{m})$ to indicate that $\bar{m} \in R$. In the special case of a binary relation R we also use the notation xRy to express that $\langle x, y \rangle \in R$.

Definition 1.1.2 If $R_1 \subseteq \prod_{i=1}^{k} M_i$ and $R_2 \subseteq \prod_{i=1}^{l} N_i$ are two relations, then

$$R_1 \times R_2 := \{\langle \bar{m}_1, \bar{m}_2 \rangle \mid R_1(\bar{m}_1), R_2(\bar{m}_2)\}$$

is the *Cartesian product* of R_1 and R_2. If $k, l > 1$, then

$$R_1 \circ R_2 := \{\langle \bar{m}_1, \bar{m}_2 \rangle \mid \exists m : R_1(\bar{m}_1, m), R_2(m, \bar{m}_2)\}$$

is the *composition* of R_1 and R_2. Note that $R_1 \circ R_2$ is a $(k+l-2)$-place relation.

Definition 1.1.3 [Inverse relation] Let R be a binary relation. Then $R^{-1} := \{\langle m, n\rangle \mid \langle n, m\rangle \in R\}$ is called the *inverse relation* of R.

Definition 1.1.4 [Reflexive and transitive closure] For a binary relation R over a given set M we define

- $R^{\langle 0\rangle} := Id_M$,
- $R^{\langle i+1\rangle} := R^{\langle i\rangle} \circ R$,
- $R^{\langle *\rangle} := \bigcup_{i=0}^{\infty} R^{\langle i\rangle}$.

$R^{\langle *\rangle}$ is called the reflexive and transitive closure or the (relational) *Kleene star* of R.

- $R^{\langle +\rangle} = \bigcup_{i=1}^{\infty} R^{\langle i\rangle}$ is called the transitive closure of R.

In the literature, the relational Kleene star is often denoted R^*, and relations $R^{\langle i\rangle}$ are denoted R^i. In the present context, these symbols will be reserved for other notions. For example, we will use the symbol R^* to denote the Kleene star of a string relation with respect to n-way concatenation. These notions will be introduced later (cf. Remark 1.4.6).

Definition 1.1.5 [Relation projection] Let $R \subseteq \prod_{i=1}^{n} M_i$ where $n \geq 2$. The relation $R_{\times i}$ defined as

$$R_{\times i} := \{\bar{m}_{\times i} \mid R(\bar{m})\}$$

is called the *projection of R to the set of coordinates* $\{1, \ldots, i-1, i+1, \ldots, n\}$. Let $\emptyset \neq \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$. Then

$$Proj(\langle i_1, \ldots, i_k\rangle, R) := \{\langle m_{i_1}, \ldots, m_{i_k}\rangle \mid \langle m_1, \ldots, m_n\rangle \in R\}$$

is called the *generalized projection of R to the coordinates* $\langle i_1, \ldots, i_k\rangle$.

Note that in the above situation we have $R_{\times i} = Proj(\langle 1, \ldots, i-1, i+1, \ldots, n\rangle, R)$. If $n = 2$ we have $R^{-1} = Proj(\langle 2, 1\rangle, R)$. In what follows, generalized projections are simply called projections. As a matter of fact we also use the *Boolean operations* (union, intersection, difference, complement) for relations.

Definition 1.1.6 [Equivalence relation, index] Let $R \subseteq M \times M$ be a binary relation.

1. R is *reflexive* iff $\forall m \in M$ we have $R(m, m)$,
2. R is *symmetric* iff $\forall m, n \in M$ always $R(m, n)$ implies $R(n, m)$.
3. R is *transitive* iff $\forall m, m', m'' \in M$ always $R(m, m')$ and $R(m', m'')$ implies $R(m, m'')$.

R is an *equivalence relation* iff R is reflexive, symmetric, and transitive. If R is an equivalence relation and $m \in M$ we write $[m]_R$ for the set (equivalence class) $\{n \in M \mid R(m,n)\}$. Furthermore $M/R := \{[m]_R \mid m \in M\}$ denotes the set of all equivalence classes. The *index* of R is the number (cardinality) $|M/R|$.

Definition 1.1.7 Let M and N be sets. A subset $f \subseteq M \times N$ is called a *partial function* from M to N if $\langle m,n\rangle \in f$ and $\langle m,n'\rangle \in f$ implies $n = n'$. The notation $f : M \to N$ indicates that f is a partial function from M to N. As usual we write $n = f(m)$ if $\langle m,n\rangle \in f$. The element n is called the image of m under f. The *domain* $dom(f)$ and the *codomain* $codom(f)$ of a partial function $f : M \to N$ are respectively defined as

$$\begin{aligned} dom(f) &:= \{m \in M \mid \exists n \in N : n = f(m)\} \\ codom(f) &:= \{n \in N \mid \exists m \in M : n = f(m)\}. \end{aligned}$$

A *total function* from M to N is a partial function with domain M.

In what follows, by a function we always mean a partial function if not mentioned otherwise. As a general convention, when writing an expression $f(m)$ we always mean that f is defined for m.

Definition 1.1.8 Let $f : M \to N$ be a function. The "lifted" version of f is the function $\hat{f} : 2^M \to 2^N$ defined pointwise:

$$\hat{f}(T) = \{f(t) \,|\, t \in T\}, \text{ for } T \subseteq M.$$

The function $\hat{f}$ is total, for any $T \subseteq M$ the set $\hat{f}(T)$ contains all defined images $f(t)$ for $t \in T$. Later we will use f to denote $\hat{f}$ if this does not lead to confusions.

Proposition 1.1.9 *Let $f : M \to N$ be a function, let $T_\alpha \subseteq M$ for every $\alpha \in A$. Then*

$$f(\bigcup_{\alpha \in A} T_\alpha) = \bigcup_{\alpha \in A} f(T_\alpha).$$

Definition 1.1.10 Let $f : A \to B$ and $g : B \to C$ be functions. Then

$$f \circ g := \{\langle a, g(f(a))\rangle \mid a \in A\}$$

is called the (functional) composition of f and g. Note that $f \circ g : A \to C$.

As a matter of fact, composition of functions is a special case of the more general composition of relations described above in Definition 1.1.2.

Definition 1.1.11 Let $f : A \to B$ be a function.

1. f is *injective* iff $\forall a, a' \in A : f(a) = f(a')$ implies $a = a'$.
2. f is *surjective* iff $\forall b \in B \exists a \in A : b = f(a)$.
3. f is *bijective* iff f is surjective and injective.

Bijective functions $f : A \to A$ are also called *permutations* of the set A.

1.2 Monoids

We present a brief introduction to monoid theory.

Basic concept and examples

Definition 1.2.1 A *monoid* $\mathcal{M}$ is a triple $\langle M, \circ, e\rangle$ where

- M is a non-empty set, the set of monoid elements;
- $\circ : M \times M \to M$ is the monoid operation (we will use infix notation);
- $e \in M$ is the monoid unit element,

and the following conditions hold:

- $\forall a, b, c \in M : a \circ (b \circ c) = (a \circ b) \circ c$ (associativity of "$\circ$"),
- $\forall a \in M : a \circ e = e \circ a = a$ (e is a unit element).

Later we often use the set M to denote the monoid $\mathcal{M} = \langle M, \circ, e\rangle$. It is easy to show that for any monoid $\mathcal{M} = \langle M, \circ, e\rangle$ the triple $\widehat{\mathcal{M}} = \langle 2^M, \widehat{\circ}, \{e\}\rangle$ is again a monoid called the "lifted" version of the monoid $\mathcal{M}$. Note that $\circ$ and $\widehat{\circ}$ respectively operate on the level of monoid elements and monoid subsets. Later we often use the same symbol for the basic monoid operation and for the "lifted" version.

Example 1.2.2 We list some examples of monoids. Let X be a set.

1. The set of natural numbers $\mathbb{N}$ with addition $+$ as operation and 0 as unit element is a monoid.
2. Let M be a set. The power set 2^M with union as operation and $\emptyset$ as unit is a monoid.
3. The set of all relations $R \subseteq X \times X$ with composition as monoid operation and the identity function over X as unit element is a monoid $\mathit{Rel}(X)$.
4. The set of all (partial) functions $f : X \to X$ is a submonoid of $\mathit{Rel}(X)$ written ($\mathit{pFun}(X)$) $\mathit{Fun}(X)$.
5. The set of all bijections of X defines a submonoid of $\mathit{Rel}(X)$, which is a group called the permutation group of X.

Operations on monoidal languages

Definition 1.2.3 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid. A *monoidal language over* $\mathcal{M}$ is a subset of M.

Obviously, monoidal languages can be combined using the Boolean operations union, intersection, set difference, and set complement with respect to M. We now present two further important operations on monoidal languages.

Definition 1.2.4 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid. For $T_1, T_2 \subseteq M$ the set

$$T_1 \circ T_2 := \{t_1 \circ t_2 \,|\, t_1 \in T_1, t_2 \in T_2\}.$$

is called the *product* of the monoidal languages T_1, T_2.

Definition 1.2.5 [Monoidal Kleene star] Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid and $T \subseteq M$. For $n \in \mathbf{N}$ we define T^n inductively:

1. $T^0 = \{e\}$,
2. $T^{k+1} = T^k \circ T$.

Furthermore we define $T^+ = \bigcup_{k=1}^{\infty} T^k$ and $T^* = \bigcup_{k=0}^{\infty} T^k$. We call T^* the *iteration* or (monoidal) *Kleene star* of T.

Definition 1.2.6 A subset $T \subseteq M$ of a monoid $\mathcal{M}$ is a *submonoid* of $\mathcal{M}$ iff $e \in T$ and $T^2 \subseteq T$.

Proposition 1.2.7 *For any subset $T \subseteq M$ of a monoid $\mathcal{M}$, the set T^* is the least submonoid of $\mathcal{M}$ containing T.*

In Algebra, T^* is often called the submonoid generated by the set T.

Monoid homomorphisms

Definition 1.2.8 Let $\mathcal{M}_1 = \langle M_1, \circ, e_1\rangle$ and $\mathcal{M}_2 = \langle M_2, \bullet, e_2\rangle$ be monoids. A function $h : M_1 \to M_2$ is a *monoid homomorphism* iff the following conditions hold:

- $h(e_1) = e_2$,
- $\forall a, b \in M_1 : h(a \circ b) = h(a) \bullet h(b)$.

Proposition 1.2.9 *The composition of two homomorphisms is again a homomorphism.*

Example 1.2.10 Let $\mathcal{M}_1$ denote the set of real numbers with addition as operation and 0 as unit element. Let $\mathcal{M}_2$ denote the set of strictly positive real numbers with multiplication as operation and 1 as unit element. Then $exp : r \mapsto e^r$ is a (bijective) monoid homomorphism.

Proposition 1.2.11 *Let $h : M_1 \to M_2$ be a monoid homomorphism.*

1. *For all $T_1, T_2 \subseteq M_1$ we have $h(T_1 \circ T_2) = h(T_1) \bullet h(T_2)$.*
2. *For all $T \subseteq M_1$ we have $h(T^*) = h(T)^*$.*

Proof. The proof of Point 1 is straightforward. The proof of Point 2 is based on a simple induction, using Point 1 and Proposition 1.1.9. □

Using Point 1 of the above proposition it is easy to show that the lifted version of $h : 2^{M_1} \to 2^{M_2}$ induces a monoid homomorphism between the lifted versions of the monoids $\mathcal{M}_1$ and $\mathcal{M}_2$.

Remark 1.2.12 Let $h : \mathcal{M}_1 \to \mathcal{M}_2$ be a monoid homomorphism, let $T \subseteq M_1$. Then $h(T)$ is called the *homomorphic image* of the monoid language T. As a matter of fact, $h(T) \subseteq M_2$ is a monoidal language.

Cartesian product of monoids

Definition 1.2.13 Let $n \geq 1$. For $1 \leq i \leq n$ let $\mathcal{M}_i = \langle M_i, \circ_i, e_i \rangle$ be a monoid. Let $\circ : (\prod_{i=1}^n M_i) \times (\prod_{i=1}^n M_i) \to \prod_{i=1}^n M_i$ denote the function defined as

$$\langle u_1, \ldots, u_n \rangle \circ \langle v_1, \ldots, v_n \rangle := \langle u_1 \circ_1 v_1, \ldots, u_n \circ_n v_n \rangle .$$

Then the triple $\prod_{i=1}^n \mathcal{M}_i := \langle \prod_{i=1}^n M_i, \circ, \langle e_1, \ldots, e_n \rangle \rangle$ is a monoid called the *Cartesian product* of the monoids $\mathcal{M}_i$.

In the special situation where all component monoids $\mathcal{M}_i$ represent the same monoid $\mathcal{M}$ the Cartesian product is written $\mathcal{M}^n$.

Remark 1.2.14 Consider a Cartesian product of the form $\mathcal{M}^n$ where $\mathcal{M} = \langle M, \circ, e \rangle$. For every function $f : M \to M$ there exists a canonical lifted version

$$f_{\uparrow n} : M^n \to M^n : \langle m_1, \ldots, m_n \rangle \mapsto \langle f(m_1), \ldots, f(m_n) \rangle.$$

More generally, for every $k \geq 1$ and every k-ary function $f : M^k \to M$ we have a canonical lifted version $f_{\uparrow n} : (M^n)^k \to M^n$ which maps the k arguments $\langle m_{1,1}, \ldots, m_{1,n} \rangle, \ldots, \langle m_{k,1}, \ldots, m_{k,n} \rangle \rangle$ to

$$\langle f(m_{1,1}, \ldots, m_{k,1}), \ldots, f(m_{1,n}, \ldots, m_{k,n}) \rangle.$$

Similarly, for every relation $R \subseteq M \times M$ there exists a canonical lifted version $R_{\uparrow n} \subseteq M^n \times M^n$ where

$$R_{\uparrow n}(\langle a_1, \ldots, a_n \rangle, \langle b_1, \ldots, b_n \rangle) \quad \Leftrightarrow \quad R(a_i, b_i) \text{ for } i = 1, \ldots, n.$$

Remark 1.2.15 Let $\mathcal{M} := \prod_{i=1}^n \mathcal{M}_i$ be a Cartesian product of monoids where $n > 1$, let $1 \leq i \leq n$. Let $\mathcal{M}_{\times i}$ denote the projection of $\mathcal{M}$ to the set of coordinates $1, \ldots, i-1, i+2, \ldots, n$ as introduced in Def. 1.1.5. The mapping $p_i : \mathcal{M} \to \mathcal{M}_{\times i} : \bar{m} \mapsto \bar{m}_{\times i}$ is called the *i-th projection mapping*. The i-th projection mapping is a monoid homomorphism.

1.3 Words and free monoids

An *alphabet* Σ is a set of symbols.

Definition 1.3.1 A *word* w over an alphabet Σ is an n-tuple

$$w = \langle a_1, \ldots, a_n \rangle$$

where $(n \geq 0)$ and $a_i \in \Sigma$ for $i = 1, \ldots, n$. The integer n is called the *length* of w and is denoted $|w|$. The (unique) tuple of length 0, written ε, is called the *empty word*. By Σ^*, we denote the set of all words over Σ. The *concatenation* of two words $u = \langle a_1, \ldots, a_n \rangle$ and $v = \langle b_1, \ldots, b_m \rangle \in \Sigma^*$ is

$$u \cdot v = \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle .$$

Clearly, $u \cdot v \in \Sigma^*$ and $|u \cdot v| = |u| + |v|$. Later we may write $w = a_1 \dots a_n$ for $w = \langle a_1, \dots, a_n \rangle$. Recall that words represent finite strings.

Definition 1.3.2 The set Σ^* with concatenation as monoid operation and the empty word ε as unit element is called the *free monoid* $\langle \Sigma^*, \cdot, \varepsilon \rangle$ for alphabet Σ.

As usual, we often write Σ^* for the monoid. Clearly, the only submonoid of Σ^* containing Σ is Σ^*, and for each element of Σ^* there exists a unique representation as a finite concatenation of elements of Σ.

Remark 1.3.3 Let $n \geq 2$, for $i = 1, \dots, n$ let $\mathcal{M}_i$ be a free monoid $\langle \Sigma_i^*, \cdot, \varepsilon \rangle$. Let $\prod_{i=1}^n \mathcal{M}_i$ be as in Definition 1.2.13. Then $\prod_{i=1}^n \mathcal{M}_i$ is *not* a free monoid. The elements of the form $\langle \varepsilon, \dots, \varepsilon, \sigma_i, \varepsilon, \dots, \varepsilon \rangle$ (where $\sigma_i \in \Sigma_i$ occupies the i-th position) generate the Cartesian product monoid, however, monoid elements $\langle u_1, \dots, u_n \rangle$ can be represented in multiple ways as products of such generators.

Proposition 1.3.4 *Let $\mathcal{M} = \langle M, \circ, e \rangle$ be a monoid, let Σ be an alphabet and $f : \Sigma \to M$ be a function. Then the* natural extension h_f *of f over Σ^*, inductively defined as*

1. $h_f(\varepsilon) = e$
2. $h_f(\sigma \cdot a) = h_f(\sigma) \circ f(a)$, *where $\sigma \in \Sigma^*, a \in \Sigma$,*

is a homomorphism between the monoids Σ^* and $\mathcal{M}$ and the only homomorphism extending f.

Definition 1.3.5 Let $s \in \Sigma^*$. Then $v \in \Sigma^*$ is an *infix* of s if s can be represented in the form $s = u \cdot v \cdot w$ for $u, w \in \Sigma^*$. Similarly v is a *prefix* of s if s can be represented in the form $s = v \cdot w$ for $w \in \Sigma^*$. In this situation, w is called a *suffix* of s. The prefix (suffix) u of s is a *proper* prefix (suffix) of s if $u \neq s$. The notation $u \leq s$ ($u < s$) expresses that u is a (proper) prefix of s. The expression $u^{-1}s$ denotes the word v if u is a prefix of $s = u \cdot v$, otherwise $u^{-1}s$ is undefined.

Definition 1.3.6 The *reverse function* $\rho : \Sigma^* \to \Sigma^*$ is defined by induction as

$$\rho(\varepsilon) := \varepsilon, \quad \forall a \in \Sigma : \rho(a) := a, \quad \forall u, v \in \Sigma^* : \rho(u \cdot v) := \rho(v) \cdot \rho(u).$$

1.4 Languages, string relations and string functions

We now introduce the fundamental concepts of languages, string relations, and string functions. A key insight in text and document processing is that many tasks can be traced back to suitable operations on languages, string functions and n-ary string relations. In this section we collect the most important operations. Larger parts of the remainder of the book will be devoted to the questions how to efficiently realized these operations, using appropriate algorithms and computational devices.

Definition 1.4.1 Let Σ be an alphabet. A subset $L \subseteq \Sigma^*$ is called a *language* over Σ.

Recall that each subset of a monoid is called a monoidal language. When we want to stress that we focus a subset of Σ^* (as opposed to a subset of some arbitrary monoid) we sometimes say that L is a *classical* language. If not mentioned otherwise, by a language we always mean a classical language.

Definition 1.4.2 Let $n \geq 1$. For each $i = 1, \ldots, n$, let Σ_i be an alphabet. Then each subset R of $\prod_{i=1}^{n} \Sigma_i^*$ is called an *n-ary string relation*. By $\bar{\varepsilon}$ we denote the n-tuple $\langle \varepsilon, \ldots, \varepsilon \rangle$.

Note that any n-ary string relation is a monoidal language, for $n = 1$ an n-ary string relation is just a classical language. Hence all the operations for monoidal languages described in Section 1.2 (set union, intersection, difference, complement, concatenation, Kleene star) can be used for n-ary string relations.

Definition 1.4.3 A binary string relation that is a function $f : \Sigma_1^* \to \Sigma_2^*$ is called a *string function*.

Definition 1.4.4 Let L be a classical language over the alphabet Σ. Then

$$\rho(L) := \{\rho(w) \mid w \in L\}$$

is called the *reverse language*.

For convenience of the reader, we have a closer look at concatenation and Kleene star for n-ary string relations. As a preliminary remark, note that *concatenation of n-tuples of words* (or *n-way concatenation*) defined as

$$\langle u_1, \ldots, u_n \rangle \cdot \langle v_1, \ldots, v_n \rangle := \langle u_1 \cdot v_1, \ldots, u_n \cdot v_n \rangle$$

represents the canonical operation of the product monoid $\prod_{i=1}^{n} \Sigma_i^*$, and $\bar{\varepsilon}$ is the unit element. Also note that reversal of words naturally generalizes to *reversal of n-tuples of words* (or *n-way reversal*):

$$\rho(\langle u_1, \ldots, u_n \rangle) := \langle \rho(u_1), \ldots, \rho(u_n) \rangle.$$

Example 1.4.5 Let $\Sigma = \{a, b\}$. Then $\langle aaa, bb \rangle \cdot \langle ba, a \rangle = \langle aaaba, bba \rangle$ and $\rho(\langle aaab, bba \rangle = \langle baaa, abb \rangle$.

Remark 1.4.6 Let R_1, R_2 be n-ary string relation. The *concatenation* of R_1 and R_2 is

$$R_1 \cdot R_2 := \{\bar{u} \cdot \bar{v} \mid \bar{u} \in R_1, \bar{v} \in R_2\}.$$

We call it *n-way concatenation* in order to stress that the concatenation boils down to the string concatenation of the n components. The Kleene star for an n-ary string relation R has the following form

- $R^0 = \{\bar{\varepsilon}\}$,
- $R^{k+1} = R^k \cdot R$,
- $R^* = \bigcup_{k=0}^{\infty} R^k$.

We also define $R^+ = \bigcup_{k=1}^{\infty} R^k$. Finally, the n-way reverse relation of R is

$$\rho(R) := \{\rho(\bar{u}) \mid \bar{u} \in R\}.$$

Example 1.4.7 We illustrate these notions using two examples from the field of translation. Let Σ_1 and Σ_2 denote two alphabets.

1. Let $u_1, u_2 \in \Sigma_1^*$ and $v_1, v_2 \in \Sigma_2^*$. Let $R_1 := \{\langle u_1, v_1\rangle\}$ and $R_2 := \{\langle u_2, v_2\rangle\}$. The entries $\langle u_i, v_i\rangle$ can be considered as single string translation rules expressing that u_i is translated into v_i $(i = 1, 2)$. We may depict these rules in the form $\rightarrow_{v_1}^{u_1}$ and $\rightarrow_{v_2}^{u_2}$. In this situation the relational concatenation $R_1 \cdot R_2 = \{\langle u_1u_2, v_1v_2\rangle\}$ describes the joint translation of u_1u_2 into v_1v_2, written as $\rightarrow_{v_1v_2}^{u_1u_2}$. Later we introduce transducers, devices with transition rules of the form $\rightarrow_{v_i}^{u_i}$, each rule representing a "translation pair" of the above form. The combination of transducer transition rules is defined as relational concatenation and leads to the joint translation of a sequence of strings of the form u_i to the corresponding sequence of strings v_i.

2. Let $R \subseteq \Sigma_1^* \times \Sigma_2^*$. The relation R can be considered as a set of translation rules $\rightarrow_v^u$ where $R(u, v)$. Then R^* describes the set of all rules $\rightarrow_{v_1 \ldots v_n}^{u_1 \ldots u_n}$ for some $n \geq 0$ where $R(u_i, v_i)$ for $i = 1, \ldots n$.

It is important to note that n-way concatenation of relations $R_1 \cdot R_2$ is distinct from the usual relational composition $R_1 \circ R_2$ of binary relations R_1 and R_2 as defined in Definition 1.1.2. Similarly the reader should not confuse relational Kleene-Star and monoidal Kleene-Star.

1.5 Summing up

Table 1.1 summarizes central operations for languages and n-ary string relations. For the sake of completeness we mention *functional composition* as an important operation for string functions. The complete family of these operations represents a kind of calculus with many interesting applications in text processing. In the following section we show how to realize these operations for specific classes of input languages or relations in a procedural way.

	Mon. lang.	Languages	String relations
Union	$T_1 \cup T_2$	$L_1 \cup L_2$	$R_1 \cup R_2$
Intersection	$T_1 \cap T_2$	$L_1 \cap L_2$	$R_1 \cap R_2$
Difference	$T_1 \setminus T_2$	$L_1 \setminus L_2$	$R_1 \setminus R_2$
Complement	$M \setminus T$	$\Sigma^* \setminus L$	$(\Sigma^* \times \ldots \times \Sigma^*) \setminus R$
Concatenation	$T_1 \cdot T_2$	$L_1 \cdot L_2$	$R_1 \cdot R_2$
(Concatenation) Kleene-Star	T_1^*	L^*	R^*
Homomorphism	$h(T)$	$h(L)$	$h(R)$
Cartesian product	$T_1 \times T_2$		$R_1 \times R_2$ (Def. 1.1.2)
Projection			$R_{\times i}$ (Def. 1.1.5)
Generalized projection			$Proj(\langle i_1, \ldots, i_k \rangle, R)$ (Def. 1.1.5)
Composition			$R_1 \circ R_2$ (Def. 1.1.2)
Relational Kleene-Star			$R^{\langle * \rangle}$ (Def. 1.1.4)
Reversal	—	$\rho(L)$	$\rho(R)$

Table 1.1: Important operations for monoidal languages, classical languages and n-ary string relations. In Column 2, M represents the underlying set of the monoid. Union, intersection, difference, complement, concatenation, (concatenation) Kleene star and homomorphic images for languages and string relations are just special cases of the corresponding operations for monoidal languages. For languages and string relations, reversal represents an additional operation. Furthermore, Cartesian product, composition, relational Kleene star and projection are additional operations for string relations.

Chapter 2

The $C(M)$ Language

In this chapter we introduce the $C(M)$ language, a new programming language created by the first co-author of this book. The language will be used throughout the rest of the book for implementing and presenting algorithms. $C(M)$ statements and expressions closely resemble the notation commonly used for the presentation of formal constructions in a Tarskian style set theoretical language. The usual set theoretic objects such as sets, functions, relations, tuples etc. are naturally integrated in the language. In contrast to imperative languages such as C or Java, $C(M)$ is in a declarative programming language. When implementing the solution to a problem, instead of specifying how to achieve it, we specify the goal itself. In practice, we just formally describe the kind of mathematical object we want to obtain. This allows us to focus on the high-level construction structure and not on the low-level details. The $C(M)$ compiler translates an appropriate construction into an efficient C program, which can be executed after compilation. Since it is easy to read $C(M)$ programs, a pseudo-code description becomes obsolete. In the first section we start with a simple algorithm showing the flavour of the language. Afterwards we provide a more comprehensive overview of the language elements. At the end a small library of commonly used types and functions for automata constructions is given.

2.1 Basics and two simple examples

Perhaps the best introduction to $C(M)$ is a short example.

Example 2.1.1 Recall that the composition of two binary relations R_1 and R_2 is defined as $R_1 \circ R_2 := \{\langle a, c\rangle \mid \exists b : \langle a, b\rangle \in R_1, \langle b, c\rangle \in R_2\}$ (Def. 1.1.2). In $C(M)$, the usual elements for describing sets are available - we may define this composition directly as

$$\text{compose}(R_1, R_2) := \{(a, c) \mid (a, b) \in R_1, (b, c) \in R_2\};$$

Given the two relations R_1 and R_2, the above function returns the set of pairs (a, c), where (a, b) runs over R_1 and (b, c) runs over R_2.

Mathematically, the *N-th order composition* $R^{\langle N\rangle}$ of a binary relation R is defined in an inductive manner:

- $R^{\langle 1\rangle} := R$,
- $R^{\langle i+1\rangle} := R^{\langle i\rangle} \circ R$,

Explicit inductive constructions are a core element of $C(M)$. These definitions start with a "step 1" where we define a base form of the objects to be constructed. Then "step i+1" explains how to obtain variant $i+1$ of the objects from variant i. An "until" clause explains when the construction is finished. Following this scheme, the N-th order composition may be introduced in $C(M)$ as the object "$\text{compose}_{\mathbb{N}}(R, N)$" in the following way:

$\text{compose}_{\mathbb{N}}(R, N) := R'$, **where**
  $R' :=$ **induction**
    step 1 :
      $R'^{(1)} := R$;
    step $i+1$:
      $R'^{(i+1)} := \text{compose}(R'^{(i)}, R)$;
    until $i = N$
    ;
  ;

Mathematically, the transitive closure of a binary relation R is defined (see Definition 1.1.4) as the relation

$$C_R := \bigcup_{i=1}^{\infty} R^{\langle i\rangle}.$$

To construct C_R we may proceed inductively, defining $C_R^{(n)} = \bigcup_{i=1}^{n} R^{\langle i\rangle}$. For a finite relation the inductive step has to be performed until $R^{\langle n+1\rangle} \subseteq C_R^{(n)}$. In $C(M)$ we may use exactly the same procedure and construct C_R using the following induction:

$C_R :=$ **induction**
  step 1 :
    ${C_R}^{(1)} := R$;
  step $n+1$:
    ${C_R}^{(n+1)} := {C_R}^{(n)} \cup \text{compose}_{\mathbb{N}}(R, n+1)$;
  until $\text{compose}_{\mathbb{N}}(R, n+1) \subseteq {C_R}^{(n)}$
  ;

Program 2.1.2 In order to complete this $C(M)$ program we have to specify the type of each object. The resulting program has the following form:

1 $\mathcal{REL}$ **is** $2^{\mathbb{N}\times\mathbb{N}}$;
2 $\text{compose} : \mathcal{REL} \times \mathcal{REL} \to \mathcal{REL}$;
3 $\text{compose}(R_1, R_2) := \{(a, c) \mid (a, b) \in R_1, (b, c) \in R_2\}$;

```
 4  compose_N : RECL × IN → RECL;
 5  compose_N(R, N) := R', where
 6     R' := induction
 7        step 1 :
 8           R'^(1) := R;
 9        step i + 1 :
10           R'^(i+1) := compose(R'^(i), R);
11        until i = N
12        ;
13     ;
14  transitiveClosure : RECL → RECL;
15  transitiveClosure(R) := C_R, where
16     C_R := induction
17        step 1 :
18           C_R^(1) := R;
19        step n + 1 :
20           C_R^(n+1) := C_R^(n) ∪ compose_N(R, n + 1);
21        until compose_N(R, n + 1) ⊆ C_R^(n)
22        ;
23     ;
```

In Line 1 we define the type $\mathcal{RECL}$ as the sets of pairs of natural numbers. Line 2 defines the type of "compose" as a function that takes two relations and returns a relation. Line 3 is the actual definition of the function "compose". Line 4 defines the type of the N-th order composition "compose$_N$" as a function that takes a relation and a natural number and returns a relation. Lines 5-13 present the definition of the function "compose$_N$". Line 14 defines the type of "transitiveClosure" as a function from relations to relations. Lines 15-23 present the inductive definition of the function "transitiveClosure".

How to write, compile and use $C(M)$ programs

From a graphical point of view, the above $C(M)$ program is presented using bookstyle mathematical notion and formatted using LaTeX. As a matter of fact, when writing the real program one has to specify the description in plain text first. Yet, $C(M)$ comes with a nice feature: after writing a $C(M)$ program as plain text, the compiler can translate it into LaTeX for producing the above layout. The plain text description for our program looks as follows:

```
REL is 2^(IN*IN);

compose in REL * REL -> REL;
compose(R_1,R_2) := {(a,c) | (a,b) in R_1, (b,c) in R_2};

compose_N in REL * IN -> REL;
compose_N(R,N) := R', where
   R' := induction
      step 1 :
         R'@1 := R;
      step i+1 :
```

```
         R'@i+1 := compose(R'@i,R);
      until i=N
      ;
   ;

transitiveClosure in REL -> REL;
transitiveClosure(R) := C_R, where
   C_R := induction
      step 1:
         C_R@1 := R;
      step n+1:
         C_R@n+1 := C_R@n \/ compose_N(R,n+1);
      until compose_N(R,n+1) subset C_R@n
      ;
   ;
```

The $C(M)$ compiler takes as input the plain text of the program and outputs a C source code (or a LaTeX code). Let the above program be stored as a file named `closure.cm`. Then the compiler is invoked by

```
cm closure.cm -o closure.c
```

The file `closure.c` will contain the corresponding C source code. Afterwards the C code has to be compiled with the `gcc` compiler for producing an executable. Currently the C code generated by $C(M)$ contains nested functions, which are supported by `gcc` but are not ANSI C compatible. The compilation is invoked by the following command

```
gcc -fnested-functions -o closure closure.c
```

In the newer versions (e.g 4.7) of `gcc` the option `-fnested-functions` has to be omitted. The LaTeX layout is generated by the compiler in the following way:

```
cm -L closure.cm -o closure.tex
```

Afterwards the closure.tex file has to be compiled with LaTeX.

The way HOW we describe objects in programs matters

In mathematics, a fixed object can be described in many distinct ways. Ignoring matters of transparency, it is not important how we specify the object. However, in a computational context the way how we describe an object may have a strong influence on the time needed to compute it. Under this perspective, the above construction and program can easily be improved. Although it produces the correct transitive closure when specifying a concrete finite relation R, it has two sources of inefficiency. First, the definition of the composition in Line 3 runs through all pairs in R_1, all pairs in R_2, and then it checks whether the selected pairs are of the form (a, b) and (b, c). This can be optimised by explicitly constructing a function F_R mapping a given b to the set $\{c \,|\, (b, c) \in R_2\}$. We can then define the composition by running through the pairs (a, b) in R_1 and the elements c in $F_R(b)$. Second, in Line 20 of the above construction we construct for each n the $n + 1$-st order composition. As a side effect we repeat

the computation of the n-th, $n-1$-th, ..., 2-nd order compositions, which have been already computed in previous steps. Moreover, if we have a pair, (a, b), which is in $R^{\langle i_1 \rangle}$, $R^{\langle i_2 \rangle}$, ..., then the extensions $\{(a, c) \mid \exists b : (b, c) \in R\}$ will be in $R^{\langle i_1+1 \rangle}$, $R^{\langle i_2+1 \rangle}$, ... Since it is not relevant in which step (a, b) is generated we can optimise our construction by generating the extensions of (a, b) only once.

Program 2.1.3 The following improved construction avoids the two deficiencies. First, it explicitly builds the function F_R and the composition is performed in the optimised way. And second, the induction is performed by considering at each step one new pair from the set C_R. In this way, starting from ${C_R}^{(0)} := R$, in step $n+1$ we generate the composition of the $n+1$-st element in $C_R^{(n)}$ with the relation R until the set $C_R^{(n)}$ is exhausted.

The program below presents two new features of $C(M)$, the use of a subcase analysis in definitions (Line 9) and the use of the "functionalize"-operator $\mathcal{F}$ in Line 3: to "functionalise" the binary relation R in the form $1 \to 2$ means (s.b.) that we build the function that maps each element x in the first projection of R to the set of all elements y in the second projection such that $(x, y) \in R$.

```
1   transitiveClosure : 2^(IN×IN) → 2^(IN×IN);
2   transitiveClosure(R) := C_R, where
3      F_R := F_{1→2}(R);
4      C_R := induction
5         step 0 :
6            C_R^(0) := R;
7         step n+1 :
8            (a,b) := (C_R^(n) as (IN × IN)*)_{n+1};
9            C_R^(n+1) := { C_R^(n) ∪ {(a,c) | c ∈ F_R(b)}   if !F_R(b)
                          { C_R^(n)                          otherwise   ;
10           until n = |C_R^(n)|
11           ;
12     ;
```

The plain text version of the above program has the following form:

```
transitiveClosure in 2^(IN*IN) -> 2^(IN*IN);
transitiveClosure(R) := C_R, where
   F_R := Func(1,2,R);
   C_R := induction
      step 0:
         C_R@0 := R;
      step n+1:
         (a,b) := (C_R@n as (IN*IN)^*)[n+1];
         C_R@n+1 := ? C_R@n \/  {(a,c) | c in F_R(b)} if !F_R(b)
                    ? C_R@n otherwise;
      until n=|C_R@n|
      ;
   ;
```

In Line 3 the function F_R is defined. The closure C_R of R is defined in the

induction starting from Line 4. In Line 6 the base value $C_R^{(0)}$ is defined to be R. In step n+1 of the construction, the pair (a, b) is defined as the $n+1$-st element of $C_R^{(n)}$ (Line 8). The expression $(C_R^{(n)}$**as** $(\mathbb{N} \times \mathbb{N})^*)$ means that we treat the set of pairs $C_R^{(n)}$ as a list of pairs. In this case the order of the elements in the list is the order in which the elements have been added to the set. In Line 9 we define the value $C_R^{(n+1)}$ using a subcase analysis. If $F_R(b)$ is defined, we join $C_R^{(n)}$ with the composition of (a, b) with R. Otherwise, the composition is empty and we define $C_R^{(n+1)} = C_R^{(n)}$. The induction ends when the condition in Line 10 is met, i.e., when the set $C_R^{(n)}$ is exhausted.

2.2 Types, terms, and statements in $C(M)$

We will now give a more thorough description of the language elements. In this section we describe types, terms and statements.

Types

$C(M)$ is a strongly typed language. The type of every object has to be either explicitly stated or implicitly derived from its definition. There is no universal type. The types in $C(M)$ are inductively defined. *Basic types* in $C(M)$ are:

- $\mathbb{N}$ natural number,
- $\mathbb{Z}$ integer number,
- $\mathbb{R}$ real number, and
- $\mathbb{B}$ Boolean values.

In plain text, these types are respectively written `IN`, `IZ`, `IR`, `IB`. $C(M)$ supports the following ways to build *composite types*:

- Tuples: if $T_1, T_2, \ldots, T_n$ are types, then $T_1 \times T_2 \times \ldots \times T_n$ is the type of n-tuples where the i-th projection is of type T_i.
- Lists: if T is any type, then T^* is the type of lists with elements of type T.
- Sets: if T is any type, then 2^T is the type of sets with elements of type T.
- Functions: if T_1 and T_2 are types, then $T_1 \to T_2$ is the type of functions with domain T_1 and range T_2.

In plain text, these types are respectively written `T_1 * T_2 * ... * T_n`, `T^*`, `T_1 -> T_2`. Parentheses can be used for type grouping. For example $(A \times B) \times C$ represents the type of pairs where the first projection is a pair of type $A \times B$ and the second projection is of type C. Some complex types are predefined: the type $\mathcal{STRING}$ corresponds to $\mathbb{N}^*$. The type of matrixes of real numbers is $\mathcal{M}(\mathbb{R})$. More complex types can be named for more convenient

notation. For example we may define the type of regular relations - which are sets of pairs of strings - as

$\mathcal{REGREL}$ **is** $2^{\mathcal{STRING}\times\mathcal{STRING}}$; (`REGREL is 2^(STRING * STRING);`).

The type of an object is specified with an 'in'-statement as in

$A, B \in \mathcal{REGREL}$; (`A, B in REGREL;`)

Any two objects of the same type can be checked for equality and respectively for inequality in the usual way: $A = B$ and $A \neq B$ (`A ~= B`). Numerical expressions can be compared as usual: $A < B$, $A \leq B$ (`A <= B`), $A > B$, $A \geq B$ (`A >= B`).

Identifiers, constants, and simple terms

The most basic expressions to name objects are identifiers and constants.

Identifiers in $C(M)$ have to start with a letter, can contain letters and digits and can end with apostrophes. Identifiers may also have indices, which are again identifiers. Greek letter names used in the plain text version are displayed in the mathematical layout as the corresponding Greek symbols. For example, possible identifiers of a more complex kind in display and textual form are

$X'_3, \Psi_{F''}, P_{\lambda_0}$ (`X'_3, Psi_F'', P_lambda_0`)

In an induction statement the inductively defined identifiers are followed by the index of the induction step written in parentheses, as in the expressions

$A^{(0)}, A^{(k+1)}, F'^{\,(n+1)}_{\Delta}$ (`A@0, A@k+1, F'_Delta@n+1`)

See also program Lines 8, 10, 18, 20 in Program 2.1.2.

Constants. In $C(M)$ there are constants for:

Booleans	$true$, $false$
Natural numbers	301, 2014
Real numbers	-25.349, 2.1234E$-$23
Strings	"Example", "This is a sentence."

Simple terms. $C(M)$ allows the grouping of identifiers/constants into tuples for supporting parallel assignments or multiple definitions. Tuples of identifiers/constants can be recursively grouped into subterms. Examples are:

$(a^{(k)}, b^{(k)})$ (`(a@k,b@k)`)

$(a, ((1, c', SE1), f_1))$ (`(a,((1,c',SE1),f_1))`)

If T is a tuple we obtain its projections using the predefined Proj operator. The second projection of T is $\mathrm{Proj}_2(T)$, written `Proj(2,T)`. We can specify the pair consisting of the second and fourth element of T as $\mathrm{Proj}_{(2,4)}(T)$. Simple terms can be used for running arguments in set builder and quantifier expressions. The term can contain constants for element building and for constraining running arguments. An example is the assignment

$M := \{(a, 0, y) \mid (a, 0, y) \in S \;\&\; a > y\}$

(`M := {(a,0,y) | (a,0,y) in S & a > y}`)

Complex terms

Set construction. There are several operators for set construction. Sets can be constructed by simply listing its elements like in $\{a, b, c\}$ (written `{a,b,c}`) or by specifying a range of numbers as in $\{n_1, \ldots, n_2\}$ (`{n_1..n_2}`). Similarly as in standard mathematical notation sets can be specified with set abstraction as in

$$\{f(x,z) \mid (x,y) \in A, z \in B\} \qquad (\texttt{\{f(x,z) | (x,y) in A, z in B\}})$$

Here the simple term (x, y) runs through the elements of A. In case that e.g. y is already defined before, it will act as a constraint – only those x will be considered for which $(x, y) \in A$. A set builder can be augmented with an additional condition as in

$$\{f(x,z) \mid (x,y) \in A, z \in B \;\&\; x+y \leq z\}$$
`{f(x,y) | (x,y) in A, z in B & x+y <= z}`

The usual set operations union $A \cup B$, intersection $A \cap B$, set difference $A \setminus B$ are provided (written `A \/ B`, `A /\ B`, `A \ B`). If A is a set of sets then the union of its elements is $\bigcup(A)$, written `union(A)`. The Cartesian product of the sets A, B and C is $A \times B \times C$ (written `A * B * C`).

Relations. Subsets of Cartesian product are relations. As for tuples, if R is a relation, then we can get its projections with the Proj operator. The second projection of R is $\mathrm{Proj}_2(R)$, written `Proj(2,R)`. We can specify the sub-relation of the second and fourth coordinates of R as $\mathrm{Proj}_{(2,4)}(R)$. Since relations are sets of tuples, they can be constructed with the set builder construction as well.

Membership and non-membership check of an element a in a set (or list) A can be expressed as $a \in A$ (`a in A`) and $a \notin A$ (`a ~in A`). Inclusion check of a set A in B is denoted as $A \subset B$ and $A \not\subset B$ (`A subset B, A ~subset B`).

List construction. Lists are constructed in an analogous way – by simply listing elements as in $\langle a, b, c\rangle$ (written `[a,b,c]`), by specifying a range of numbers as in $\langle 1, \ldots, n_2\rangle$ (`[1..n_2]`), or using list builders as in $\langle f(x,z) \mid (x,y) \in A, z \in B\rangle$. The latter form can again be augmented with an additional condition as in

$$\langle f(x,z) \mid (x,y) \in A, z \in B \;\&\; x+y \leq z\rangle$$
`[f(x,y) | (x,y) in A, z in B & x+y <= z]`

Concatenation of two lists A and B is denoted $A.B$. If A is a list of lists, then $\bigodot(A)$ (written `flatten(A)`) is the concatenation of the elements of A. The number of elements in a set or a list A is denoted as $|A|$. The i$^{\text{th}}$ element of a list L is $(L)_i$, written `L[i]`. The sublist from the i$^{\text{th}}$ to the j$^{\text{th}}$ element of a list L is $(L)_{i,\ldots,j}$, written `L[i..j]`. With $\#_L(a)$ (written `#(a,L)`) we denote the index of a in L. If a is not an element of L, then 0 is returned. If there are more that one occurrence of a in L, then one of the corresponding indices is returned. The set of elements of a list L is $\mathrm{set}(L)$. The minimal, maximal, sum and product of elements of a set or list A of appropriate type are denoted as $\min(A)$, $\max(A)$, $\sum(A)$ (`sum(A)`), $\prod(A)$ (`prod(A)`), respectively.

Functions. Functions in $C(M)$ are either finite or specified with an expression over the arguments as in

$$f(x,y) := x + 2 \times y; \qquad \texttt{f(x,y) := x+2*y;}$$

Finite functions are sets of pairs and can be constructed with the usual set and relation constructors. If f is a function, then as usual $f(a)$ denotes the image of a and $!f(a)$ checks whether f is defined for a. If f is a function of type $T_1 \to T_2$ and A is a set of type 2^{T_1}, then $f(A)$ is the image of the set A through f. If f is a function of type $T_1 \times T_2 \to T_3$ and a is of type T_1, then $f(a)$ is a function of type $T_2 \to T_3$, such that $f(a)(b) = f(a,b)$. Functions (both finite and infinite) are treated as normal objects and can be passed as parameters, returned by other functions, added as elements in sets etc. If R is a relation of type $2^{T_1 \times T_2}$ then we can "functionalize" it using the Func operator. $\mathrm{Func}_{1\to 2}(R)$ (written `Func(1,2,R)`) denotes the finite function $\{(a, \{b \,|\, (a,b) \in R\}) \,|\, a \in \mathrm{Proj}_1(R)\}$. More generally, if R is an n-ary relation and $\{i_1, \ldots, i_k\}, \{j_1, \ldots, j_l\}$ are disjoint and nonempty subsets of $\{1, \ldots, n\}$, then $Func_{\langle i_1,\ldots,i_k\rangle \to \langle j_1,\ldots,j_l\rangle}(R)$ is the function with domain $Proj(\langle i_1, \ldots, i_k\rangle, R)$ that maps an element $\langle a_{i_1}, \ldots, a_{i_k}\rangle$ to the set $\{\langle a_{j_1}, \ldots, a_{j_l}\rangle | \langle a_1, \ldots a_n\rangle \in R\}$.

Arithmetic expressions. The usual arithmetic expressions are provided for the numerical types – addition $a+b$, subtraction $a-b$, negative value $-a$, absolute value $|a|$, multiplication $a \times b$, division a/b, power a^b and reminder $a \,\mathrm{rem}\, b$.

Boolean expressions include

- negation $\neg P$, (`~P`)
- conjunction $P \wedge Q$, (`P /\ Q`)
- disjunction $P \vee Q$, (`P \/ Q`)
- implication $P \to Q$, (`P -> Q`)
- equivalence $P \leftrightarrow Q$, (`P <-> Q`)

In addition $C(M)$ supports bounded quantifiers like

$\forall x \in X : (\exists y \in Y : p(x,y))$ (`forall x in X: (exists y in Y: p(x,y))`)

Conditional expressions can be specified as:

$$f(a) := \begin{cases} a-1 & \textbf{if } a > 5 \\ a & \textbf{if } (a>0) \wedge (a \le 5) \\ 0 & \textbf{otherwise;} \end{cases}$$

```
f(a) := ? a-1 if a > 5
        ? a if (a>0) /\ (a <= 5)
        ? 0 otherwise;
```

Matrix calculation. A one row matrix specified as $[a,b,c]$. Rows can be appended like $[1,2,3] \setminus\setminus [4,5,6]$. Another way to constructing matrices is by using a matrix builder: $[f(i,j) \mid i = 1, \ldots, 2,\ j = 1, \ldots, 3]$. If M is a matrix, then $M_{i,j}$ is the i$^{\text{th}}$ row j$^{\text{th}}$ column element of M. The usual matrix operations like addition, subtraction, multiplication and division are implemented. Transposition is denoted as M^T.

Statements

Each $C(M)$ program is a list of statements. There are four kinds of statements – type definitions, declarations, assignments and supplementary actions. Every statement must end with a semicolon.

Type definitions. A type definition is used for naming a complex type. For example in Line 1 of Program 2.1.2 we have the following type definition:

$$\mathcal{REL} \textbf{ is } 2^{\mathbb{N}\times\mathbb{N}};$$

Declarations. Declaration statements are used for declaring the types of identifiers and terms used. In cases where the type of an object can be directly derived from the expression the declaration can be omitted. But in most cases when an object is defined by induction or when a function is defined the type can hardly be inferred and a type declaration is needed. The statement in Line 7 of Program 2.1.2 declares the type of the function "compose":

$$\text{compose} : \mathcal{REL} \times \mathcal{REL} \to \mathcal{REL};$$

Assignments are the most commonly used statements in $C(M)$. There are four types of assignments – simple assignments, assignments with where block, case assignments and inductive assignments.

Simple assignments. Simple assignments consist of an expression only. They are used to define the value of a term. Line 8 of Program 2.1.3 gives an example:

$$(a, b) := (C_R{}^{(n)} \textbf{ as } (\mathbb{N} \times \mathbb{N})^*)_{n+1};$$

Function values can be defined refering to arguments as in Line 3 of Program 2.1.2:

$$\text{compose}(R_1, R_2) := \{(a, c) \mid (a, b) \in R_1, (b, c) \in R_2\};$$

Assignments with where block Assignment statements with a 'where'-block appear as simple assignments that are followed by a block of additional statements after the 'where' keyword. The additional statements define the objects used in the expression and may include further statements. The statements below give an illustration:

$$\begin{aligned} &\text{compose}_{2\text{N}}(R, N) := \text{compose}(R', R'), \textbf{ where} \\ &\quad R' := \text{compose}_{\text{N}}(R, n); \\ &\quad ; \end{aligned}$$

Note that the semicolon on the last line closes the whole statement – as discussed above each statement has to end with a semicolon.

Case assignment The case assignment consists of a list of pairs and expressions. The expression of the first condition evaluated to true is assigned to the term. An "otherwise" case expression is optional and used if none of the previous conditions holds. The case assignment differs from the conditional expression by the option to include a 'where' block for each of the cases. Instead of

using a conditional expression in Line 9 of Program 2.1.3 we could alternatively use a case assignment:

$$
\begin{array}{l}
{C_R}^{(n+1)} := \\
\quad \textbf{case } !\,\mathrm{F_R}(b) : {C_R}^{(n)} \cup A', \textbf{where} \\
\quad\quad A' := \{(a, c) \mid c \in \mathrm{F_R}(b)\} \\
\quad \textbf{otherwise } : {C_R}^{(n)} \\
\quad ;
\end{array}
$$

Inductive assignment The inductive assignment is used for inductive constructions of terms. Lines 4-11 from Program 2.1.3 present an inductive assignment.

$$
\begin{array}{l}
C_R := \textbf{induction} \\
\quad \textbf{step } 0 : \\
\quad\quad {C_R}^{(0)} := R; \\
\quad \textbf{step } n+1 : \\
\quad\quad (a, b) := ({C_R}^{(n)} \textbf{ as } (\mathbb{N} \times \mathbb{N})^*)_{n+1}; \\
\quad\quad {C_R}^{(n+1)} := \left\{ \begin{array}{ll} {C_R}^{(n)} \cup \{(a, c) \mid c \in \mathrm{F_R}(b)\} & \textbf{if } !\,\mathrm{F_R}(b) \\ {C_R}^{(n)} & \textbf{otherwise} \end{array} \right. ; \\
\quad \textbf{until } n = \left|{C_R}^{(n)}\right| \\
\quad ;
\end{array}
$$

The base of the induction is defined by the statements after the step 0 clause. The inductive step is defined by the statements after the step $n+1$ clause. The inductive steps are repeated until the condition of after the until clause holds.

Supplementary actions

Besides definitions, declarations and assignments, $C(M)$ supports supplementary actions for input and output operations and similar issues. In contrast to all other statements, those actions can have side effects. For example we can dump out the relation resulting by the transitive closure of $\{(1, 2), (2, 3), (3, 5), (5, 10)\}$ with the following statement:

$$\text{dump} \leftarrow \text{transitiveClosure}(\{(1, 2), (2, 3), (3, 5), (5, 10)\});$$

```
dump <- transitiveClosure({(1,2),(2,3),(3,5),(5,10)});
```

Some of the implemented supplementary actions are

- dump: dumps the expression specified to the standard output;

- print: the expression (which must be a STRING) is printed on the standard output;
- assert: if the expression of type $\mathbb{B}$ does not hold, the program halts.

Chapter 3

Monoidal finite-state automata

In the previous chapter we introduced a large set of operations on strings, languages, and string relations. We now look at the procedural side. As a starting point we use finite-state automata, which represent the simplest devices for recognizing languages. The theory of finite-state automata has been described in numerous textbooks (e.g., [**?**, **?**, **?**, **?**, **?**]). The notes below offer a brief introduction to finite-state automata. However, we immediately introduce the more general concept of a monoidal finite-state automaton. Similar generalized perspectives are found in [**?**, **?**]. Many constructions and properties of finite-state automata directly generalize to the case of monoidal finite-state automata. Later we shall see that more complex concepts such as multi-tape automata, which are treated separately in classical descriptions on automata, also represent natural examples of monoidal finite-state automata.

3.1 Basic concept and examples

We introduce the central concept of this chapter.

Definition 3.1.1 A *monoidal finite-state automaton* (FSA) is a tuple of the form $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ where

- $\mathcal{M} = \langle M, \circ, e \rangle$ is a monoid,
- Q is the set of states,
- $I \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ is the set of final states, and
- $\Delta \subseteq Q \times M \times Q$ is the transition relation.

Triples $\langle p, a, q \rangle \in \Delta$ are called *transitions*. The transition $\langle p, a, q \rangle$ *begins* at p, *ends* at q and has the *label* a.

If Δ is clear from the context, transitions $\langle p, a, q\rangle$ are also denoted in the form $p \to^a q$.

Definition 3.1.2 *Classical finite-state automata* are monoidal finite-state automata where the underlying monoid is the free monoid over a finite alphabet, Σ and the transition relation labels are in $\Sigma^\varepsilon = \Sigma \cup \{\varepsilon\}$.

Classical finite-state automata with monoid Σ^* are also denoted in the form $\langle \Sigma, Q, I, F, \Delta\rangle$.

Example 3.1.3 Let $\mathcal{M} = \mathcal{M}_1^2$ be the Cartesian product of two copies of the free monoid $\mathcal{M}_1 = \{a, b\}^*$, let $Q = I = F = \{q_0\}$, let $\Delta = \{\langle q_0, \langle a, b\rangle, q_0\rangle\}$. Then $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta\rangle$ is a monoidal finite-state automaton. Using terminology introduced below (cf. Definition **??**), $\mathcal{A}$ is a two-tape automaton.

Definition 3.1.4 Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta\rangle$ be a monoidal finite-state automaton where $\mathcal{M} = \langle M, \circ, e\rangle$. $\mathcal{A}$ is called *e-free* iff $\Delta \subseteq Q \times (M \setminus \{e\}) \times Q$.

An important notion for classical finite-state automata is determinism.

Definition 3.1.5 Let $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta\rangle$ be a classical finite-state automaton. $\mathcal{A}$ is *deterministic* iff the transition relation is a (partial) function $\delta : Q \times \Sigma \to Q$ and $\mathcal{A}$ has exactly one initial state.

We do not introduce this concept for monoidal finite-state automata since there are distinct options.

Definition 3.1.6 Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta\rangle$ be a monoidal finite-state automaton. A *proper path* in $\mathcal{A}$ is a finite sequence of $k > 0$ transitions

$$\pi = \langle q_0, a_1, q_1\rangle \langle q_1, a_2, q_2\rangle \ldots \langle q_{k-1}, a_k, q_k\rangle$$

where $\langle q_{i-1}, a_i, q_i\rangle \in \Delta$ for $i = 1 \ldots k$. The number k is called the *length* of π, we say that π starts in q_0 and ends in q_k. The monoid element $w = a_1 \circ \ldots \circ a_k$ is called the *label* of π. We may denote the path π as

$$\pi = q_0 \to^{a_1} q_1 \ldots \to^{a_k} q_k.$$

The *null path* of $q \in Q$ is 0_q starting and ending in q with label e. A *successful path* is a path starting in an initial state and ending in a final state.

Definition 3.1.7 Let $\mathcal{A}$ be as above. The *generalized transition relation* Δ^* is defined as the smallest subset of $Q \times M \times Q$ with the following closure properties:

- for all $q \in Q$ we have $\langle q, e, q\rangle \in \Delta^*$.
- For all $q_1, q_2, q_3 \in Q$ and $w, a \in M$: if $\langle q_1, w, q_2\rangle \in \Delta^*$ and $\langle q_2, a, q_3\rangle \in \Delta$, then also $\langle q_1, w \circ a, q_3\rangle \in \Delta^*$.

Triples $\langle p, u, q\rangle \in \Delta^*$ are called *generalized transitions*. The generalized transition $\langle p, u, q\rangle$ *begins* at p, *ends* at q and has the *label* u.

Clearly, generalized transitions are simplified descriptions of (proper or null) paths, representing the beginning, label and ending and abstracting from intermediate states.

Definition 3.1.8 Let $\mathcal{A}$ be as above. For $q \in Q$ the set

$$L_{\mathcal{A}}(q) := \{w \in M \mid \exists q_1 \in F : \langle q, w, q_1\rangle \in \Delta^*\}$$

is called the *language of state q in $\mathcal{A}$*. For $S \subseteq Q$ we define

$$L_{\mathcal{A}}(S) := \cup_{q\in S} L_{\mathcal{A}}(q).$$

Definition 3.1.9 Let $\mathcal{A}$ be as above. A state $q \in Q$ is called *reachable* from a starting state iff there exists a path starting in an initial state and ending in q i.e. there exists $w \in \Sigma^*$ and $q_0 \in I$ such that $\langle q_0, w, q\rangle \in \Delta^*$. From a state $q \in Q$ a final state is *reachable* iff there exists a path starting in q and ending in a final state i.e. $L_{\mathcal{A}}(q) \neq \emptyset$.

The following results, which gives an inductive definition for the language of a state, follows immediately.

Proposition 3.1.10 *Let $\mathcal{A}$ be as above, let $p \in Q$. Then*

$$L_{\mathcal{A}}(p) = E(p) \cup \bigcup_{\langle p,w,q\rangle\in\Delta} w \cdot L_{\mathcal{A}}(q)$$

where $E(p) = \{\varepsilon\}$ if $p \in F$ and $E(p) = \emptyset$ otherwise.

Definition 3.1.11 Let $\mathcal{A}$ as above. Then $L(\mathcal{A}) := L_{\mathcal{A}}(I)$ is called the monoidal language *accepted* (or *recognized*) by $\mathcal{A}$. $L(\mathcal{A}) := L_{\mathcal{A}}(I)$.

Hence, $L(\mathcal{A})$ is the set of labels of the successful paths of $\mathcal{A}$.

Definition 3.1.12 Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta\rangle$ as above. For $p \in Q$ and $w \in M$, the set of *w-successors of p* is

$$Suc_{\mathcal{A}}(p, w) := \{q \in Q \mid \langle p, w, q\rangle \in \Delta^*\}.$$

For $P \subseteq Q$ the set of *w-successors of P* is

$$Suc_{\mathcal{A}}(P, w) := \bigcup_{p\in P} Suc_{\mathcal{A}}(p, w).$$

The set of *active states* for input $w \in M$ is $Act_{\mathcal{A}}(w) = Suc_{\mathcal{A}}(I, w)$.

Definition 3.1.13 Two monoidal FSA $\mathcal{A}_1$ and $\mathcal{A}_2$ are *equivalent* if $L(\mathcal{A}_1) = L(\mathcal{A}_2)$.

Definition 3.1.14 A monoidal language over M is called an *monoidal automaton language* iff it is recognized by some monoidal finite-state automaton. A classical language is called a *classical automaton language* iff it is recognized by a classical finite-state automaton.

Remark 3.1.15 Let Σ be a finite alphabet. The transition labels of monoidal finite-state automata $\mathcal{A}$ over the free monoid Σ^* are arbitrary words over Σ. Any such automaton $\mathcal{A}$ can be converted to an equivalent classical finite-state automaton by introducing intermediate states for each label consisting of more than one symbol: we substitute each transition $\langle q', a_1 a_2 \dots a_n, q'' \rangle \in \Delta$ where $n > 1$ with a sequence of transitions of the form $\langle q', a_1, q'_1 \rangle$, $\langle q'_1, a_2, q'_2 \rangle$, ..., $\langle q'_{n-2}, a_{n-1}, q'_{n-1} \rangle$, $\langle q'_{n-1}, a_n, q'' \rangle$ where $q'_1, q'_2, \dots, q'_{n-1}$ are new non-final states.

Remark 3.1.16 If $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ is a deterministic classical finite-state automaton, the generalized transition relation is a function $\delta^* : Q \times \Sigma^* \to Q$. It is called the *generalized transition function.*

3.2 Closure properties of monoidal finite-state automata

We give constructions showing that the class of monoidal automaton languages is closed under some of the operations for monoidal languages mentioned in Section 1.2.

Proposition 3.2.1 (Closure properties of monoidal finite-state automata) *Let $\mathcal{A}_1 = \langle \mathcal{M}, Q_1, I_1, F_1, \Delta_1 \rangle$ and $\mathcal{A}_2 = \langle \mathcal{M}, Q_2, I_2, F_2, \Delta_2 \rangle$ be two monoidal finite-state automata and let $Q_1 \cap Q_2 = \emptyset$. Then the following holds:*

1. *(Union) For the monoidal finite-state automaton*

   $$\mathcal{A} = \langle \mathcal{M}, Q_1 \cup Q_2, I_1 \cup I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2 \rangle$$

   we have $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$.

2. *(Concatenation) For the monoidal finite-state automaton*

   $$\mathcal{A} = \langle \mathcal{M}, Q_1 \cup Q_2, I_1, F_2, \Delta_1 \cup \Delta_2 \cup \{\langle q_1, e, q_2 \rangle \mid q_1 \in F_1, q_2 \in I_2\} \rangle$$

   we have $L(\mathcal{A}) = L(\mathcal{A}_1) \circ L(\mathcal{A}_2)$.

3. *(Kleene-Star) Let q_0 be a new state, let*

   $$\Delta := \Delta_1 \cup \{\langle q_0, e, q_1 \rangle \mid q_1 \in I_1\} \cup \{\langle q_2, e, q_0 \rangle \mid q_2 \in F_1\}.$$

   For the monoidal finite-state automaton

   $$\mathcal{A} = \langle \mathcal{M}, Q_1 \cup \{q_0\}, \{q_0\}, F_1 \cup \{q_0\}, \Delta \rangle$$

   we have $L(\mathcal{A}) = L(\mathcal{A}_1)^*$.

4. *(Homomorphic images) Let $h : \mathcal{M} \to \mathcal{M}'$ be a monoid homomorphism. Let $\mathcal{A}' = \langle \mathcal{M}', Q_1, I_1, F_1, \Delta' \rangle$ where*

   $$\Delta' = \{\langle q_0, h(m), q_1 \rangle \mid \langle q_0, m, q_1 \rangle \in \Delta_1\}.$$

   Then we have $L(\mathcal{A}') = h(L(\mathcal{A}_1))$.

Classical finite-state automata additional closure properties, see Section 3.7.

3.3 Removal of e-transitions for monoidal finite-state automata

In contrast to the operations considered in the previous subsection, the operations considered here and in the next two subsections do not modify the monoidal language of the given automaton. The goal is rather to simplify automata from a structural point of view, thus improving efficiency of the recognition process.

Definition 3.3.1 The *forward e-closure* $C_e : Q \to 2^Q$ is defined as

$$C_e(q) = \{q' \in Q \mid \langle q, e, q' \rangle \in \Delta^*\}.$$

Proposition 3.3.2 *For any monoidal FSA $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ there exists an equivalent e-free monoidal FSA $\mathcal{A}$' with the same set of states.*

Proof. The monoidal FSA

$$\mathcal{A}' = \langle \mathcal{M}, Q, C_e(I), F, \Delta' \rangle ,$$

where

$$\Delta' = \{\langle q_1, a, q_2 \rangle \mid \exists \langle q_1, a, q' \rangle \in \Delta : q_2 \in C_e(q') \ \& \ a \neq e\}.$$

is e-free and equivalent to $\mathcal{A}$. □

It should be noted that several constructions for eliminating e-transitions exist. Instead of using a forward e-closure, other constructions are based on a backward (or backward-forward) e-closure. The backward e-closure is defined as $C'_e(q) = \{q' \in Q \mid \langle q', e, q \rangle \in \Delta^*\}$. A discussion of distinct constructions for removal of e-transitions and their computational properties can be found in [**?**].

Remark 3.3.3 The e-removal only makes sense in situations where the underlying monoid does not contain zero divisors, i.e., for monoids where we have $\forall m_1, m_2 \in M : m_1 \neq e \ \& \ m_2 \neq e \to m_1 \circ m_2 \neq e$. The most important examples are free monoids and Cartesian products of free monoids. If the underlying monoid does not contain zero divisors, then every proper path of an e-free automaton has a label different from e.

3.4 Monoidal regular languages and monoidal regular expressions

Monoidal regular languages are defined using a simple inductive definition of languages. Starting from the empty language and all "singleton languages" containing exactly one monoid element, inductive rules close under union, concatenation and Kleene star. We shall see that the monoidal regular languages are exactly the monoidal automaton languages. This yields another, less procedural characterization of monoidal automaton languages.

Definition 3.4.1 Let $\mathcal{M} = \langle M, \circ, e \rangle$ be a monoid. We define a *monoidal regular language* over $\mathcal{M}$ by induction:

1. $\emptyset$ is a monoidal regular language;
2. if $m \in M$, then $\{m\}$ is a monoidal regular language;
3. if $L_1, L_2 \subseteq M$ are monoidal regular languages, then
   - $L_1 \cup L_2$ is a monoidal regular language (union),
   - $L_1 \circ L_2$ is a monoidal language (product, cf. Def. 1.2.4),
   - L_1^* is a monoidal regular language (Kleene star, cf. Def. 1.2.5).
4. There are no other monoidal regular languages.

Definition 3.4.2 Let L be a monoidal regular language over the monoid $\mathcal{M}$. If $\mathcal{M}$ is the free monoid over a finite alphabet Σ, then L is called a *classical regular language*.

As in the classical case, monoidal regular languages can be represented by means of special expressions.

Definition 3.4.3 Let $\mathcal{M} = \langle M, \circ, e\rangle$ be a monoid. A *monoidal regular expression* over $\mathcal{M}$ is a word over $M \cup \{(,), *, +, \cdot, \emptyset\}$. The set of monoidal regular expressions over $\mathcal{M}$ is defined by induction:

1. $\emptyset$ is a monoidal regular expression over $\mathcal{M}$ (sometimes denoted 0);
2. if $m \in M$, then m is a monoidal regular expression over $\mathcal{M}$;
3. if E_1 and E_2 are monoidal regular expressions over $\mathcal{M}$, then
   - $(E_1 + E_2)$ is a monoidal regular expression,
   - $(E_1 \cdot E_2)$ is a monoidal regular expression,
   - (E_1^*) is a monoidal regular expression.
4. There are no other monoidal regular expressions over $\mathcal{M}$.

As usual, for each monoidal regular expression E over $\mathcal{M}$ we inductively define a monoidal language $L(E)$, using the clauses

$$\begin{aligned} L(\emptyset) &:= \emptyset, \\ L(m) &:= \{m\} \quad (m \in M), \\ L(E_1 + E_2) &:= L(E_1) \cup L(E_2), \\ L(E_1 \cdot E_2) &:= L(E_1) \circ L(E_2) \\ L(E^*) &:= L(E)^* \end{aligned}$$

This defines a natural correspondence between monoidal regular expressions and monoidal regular languages.

If $\mathcal{M}$ is the free monoid over a finite alphabet Σ, then monoidal regular expressions are called *classical regular expressions*.

3.5 Equivalence between monoidal regular languages and monoidal automaton languages

Proposition 3.5.1 *The following shows that the simplest monoidal regular languages can be represented by means of monoidal FSA.*

1. *(Empty language) For* $\mathcal{A}_\emptyset = \langle \mathcal{M}, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ *we have* $L(\mathcal{A}_\emptyset) = \emptyset$.
2. *(Single element languages) Let* $a \in M$. *For the monoidal FSA* $\mathcal{A}_a = \langle \mathcal{M}, \{q_0, q_1\}, \{q_0\}, \{q_1\}, \{\langle q_0, a, q_1 \rangle\} \rangle$ *we have* $L(\mathcal{A}_a) = \{a\}$.

The following theorem presents a deeper result for the correspondence between monoidal automaton languages and regular languages.

Theorem 3.5.2 (Kleene) *A monoidal language is regular if and only if it is a monoidal automaton language.*

Proof. ("⇒") This direction follows directly from the closure properties given in Propositions 3.5.1 and 3.2.1.

("⇐") Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \delta \rangle$ be a monoidal FSA. Let $Q = \{q_1, \ldots, q_n\}$, for $0 \leq k \leq n$ let and $Q^k := \{q_1, \ldots, q_k\}$. Let R_{ij}^k denote the set of all labels of paths π starting in q_i and ending q_j for which all intermediate states (beside the beginning and the ending states) are in Q_k. Note that there are no intermediate states for $k = 0$. Clearly

$$R_{ij}^0 = \begin{cases} \{a \in M \mid \langle q_i, a, q_j \rangle \in \delta\}, \text{ if } i \neq j \\ \{e\} \cup \{a \in \Sigma \mid \langle q_i, a, q_j \rangle \in \delta\}, \text{ if } i = j \end{cases}$$

and

$$R_{ij}^k = R_{ij}^{k-1} \cup (R_{ik}^{k-1} \circ (R_{kk}^{k-1})^* \circ R_{kj}^{k-1})$$

From the above presentation it follows by induction over k that for any $i, j, k \in \{0, \ldots, n\}$ the monoidal languages R_{ij}^k are regular. We have that

$$L(\mathcal{A}) = \bigcup_{\{i \mid q_i \in I\}\{j \mid q_j \in F\}} R_{ij}^n$$

Hence $L(\mathcal{A})$ is a monoidal regular language. □

3.6 Connection between monoidal regular languages and regular languages over finite alphabets

Theorem 3.6.1 *A subset A of a monoid is a monoidal regular language if and only if A is the homomorphic image of a regular language over some finite alphabet.*

Proof. ("⇒") Let A be a monoidal regular language over the monoid $\mathcal{M}$. Let $\mathcal{A} = \langle \mathcal{M}, Q, I, F, \Delta \rangle$ be a monoidal FSA over $\mathcal{M} = \langle M, \circ, e \rangle$ such that $L(\mathcal{A}) = A$. Let Σ be the finite alphabet

$$\Sigma := \{a_m \mid \exists \langle q_1, m, q_2 \rangle \in \Delta\}.$$

Consider the function $\phi : \Sigma \to M$ defined as $\phi(a_m) := m$. Following Property 1.3.4 there exists a unique extension of ϕ to a homorphism $\Phi : \Sigma^* \to M$. Let $\mathcal{A}' = \langle \Sigma, Q, I, F, \Delta' \rangle$ be the classical finite-state automaton where

$$\Delta' = \{\langle q_1, a_m, q_2 \rangle \mid \langle q_1, m, q_2 \rangle \in \Delta\}$$

. $L(\mathcal{A}')$ is a regular language and we have $\Phi(L(\mathcal{A}')) = L(\mathcal{A})$.

("⇐") Follows from a very similar construction. □

3.7 Determinization and further closure properties of classical finite-state automata

In Definition 3.1.5 we introduced deterministic classical finite-state automata. Deterministic FSA are important since they represent an efficient device for recognizing a given automaton language. The proof of the following theorem gives us an effective procedure for determinization of FSA.

Theorem 3.7.1 (Determinization of classical FSA) *For any classical finite-state automaton $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ there exists an equivalent deterministic finite-state automaton $\mathcal{A}_D$.*

Proof. Using Proposition 3.3.2 we may assume without loss of generality that $\mathcal{A}$ does not have ε-transitions, which means that $\Delta \subseteq Q \times \Sigma \times Q$. Consider the automaton

$$\mathcal{A}_D := \left\langle \Sigma, 2^Q, I, F_D, \delta \right\rangle$$

where $F_D := \{S \subseteq Q \mid S \cap F \neq \emptyset\}$ and

$$\delta(S, a) := \{q \in Q \mid \exists q_1 \in S : \langle q_1, a, q \rangle \in \Delta\}$$

for $S \subseteq Q, a \in \Sigma$. Clearly δ is a function $2^Q \times \Sigma \to 2^Q$ and $\mathcal{A}_D$ has exactly one initial state I. If a word $w \in \Sigma^*$ is accepted in $\mathcal{A}$ there exists a path with label w leading from a state in I to a state in the set F. The definitions of δ and F_D ensure that $\delta^*(I, w) \in F_D$, which shows that w is accepted in $\mathcal{A}_D$. Conversely, if $\delta^*(I, w) \in F_D$ then the set $\delta^*(I, w)$ contains a final state $f \in F$. Using the unique path of $\mathcal{A}_D$ with label w and the definition of δ it is easy to construct in a backward manner, starting from f, a path of $\mathcal{A}$ with label w that leads from an initial state to f. This shows that w is accepted in $\mathcal{A}$. □

It should be noted that the simple determinization procedure obtained from this proof can be considerably refined. The basic idea is to use the collection of all sets of active states (for arbitrary input) of the given non-deterministic classical FSA as the new set of states of the deterministic FSA. There are also determinization procedures that directly include removal of ϵ-transitions. Note also that the function δ built in the above proof is a total function.

Proposition 3.7.2 (Complementing deterministic FSA) *For any deterministic FSA $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ where δ is a total function the following holds: for the FSA $\mathcal{A}' = \langle \Sigma, Q, q_0, Q \setminus F, \delta \rangle$ we have $L(\mathcal{A}') = \Sigma^* \setminus L(\mathcal{A})$.*

Proof. This is an immediate consequence of the fact that for every input word $w \in \Sigma^*$ there exists a *unique* path of $\mathcal{A}$ starting at q_0 with label w. □

Above we have seen that the languages accepted by monoidal finite-state automata are closed under the "'regular operations" union, concatenation, and Kleene-star, and under homomorphic images. We now found that the set of languages accepted by classical finite-state automata is closed under complement. The following proposition collects additonal closure properties.

Proposition 3.7.3 *Let $\mathcal{A}_1 = \langle \Sigma, Q_1, I_1, F_1, \Delta_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q_2, I_2, F_2, \Delta_2 \rangle$ be two classical ε-free finite-state automata and let $Q_1 \cap Q_2 = \emptyset$. Then the following holds:*

1. *(Intersection for ε-free classical finite-state automata) If $\mathcal{A}_1$ and $\mathcal{A}_2$ are ε-free FSA, then for the FSA*

$$\mathcal{A} := \langle \Sigma, Q_1 \times Q_2, I_1 \times I_2, F_1 \times F_2, \Delta' \rangle$$

   where $\Delta' := \{\langle \langle q_1, q_2 \rangle, a, \langle r_1, r_2 \rangle \rangle \mid \langle q_1, a, r_1 \rangle \in \Delta_1 \ \& \ \langle q_2, a, r_2 \rangle \in \Delta_2\}$ we have $L(\mathcal{A}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.

2. *(Difference for deterministic classical finite-state automata) If $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic classical finite-state automata and the transition function of $\mathcal{A}_2$ is total, then for the finite-state automaton*

$$\mathcal{A} := \langle \Sigma, Q_1 \times Q_2, I_1 \times I_2, F_1 \times (Q_2 \setminus F_2), \Delta' \rangle$$

   where $\Delta' := \{\langle \langle q_1, q_2 \rangle, a, \langle r_1, r_2 \rangle \rangle \mid \langle q_1, a, r_1 \rangle \in \Delta_1 \ \& \ \langle q_2, a, r_2 \rangle \in \Delta_2\}$ we have $L(\mathcal{A}) = L(\mathcal{A}_1) \setminus L(\mathcal{A}_2)$.

3. *(Reverse language) For the classical FSA*

$$\mathcal{A} := \langle \Sigma, Q_1, F_1, I_1, \Delta' \rangle$$

   where $\Delta' = \{\langle q_2, a, q_1 \rangle \mid \langle q_1, a, q_2 \rangle \in \Delta_1\}$ we have $L(\mathcal{A}) = \rho(L(\mathcal{A}_1))$.

3.8 Minimization of classical finite-state automata and the Myhill-Nerod equivalence relation

The observations collected in this section are wellknown, hence we omit proofs. Let Σ be a finite alphabet.

Definition 3.8.1 An equivalence relation $R \subseteq \Sigma^* \times \Sigma^*$ is called *right invariant* if

$$\forall u, v \in \Sigma^* : u \ R \ v \Rightarrow (\forall w \in \Sigma^* : u \cdot w \ R \ v \cdot w).$$

Proposition 3.8.2 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a deterministic classical FSA. Then the relation*

$$R_{\mathcal{A}} = \{\langle u, v \rangle \in \Sigma^* \times \Sigma^* \mid \delta^*(q_0, u) = \delta^*(q_0, v)\}$$

is a right invariant equivalence relation and $|\Sigma^*/R_{\mathcal{A}}| = |\{[s]_{R_{\mathcal{A}}} \mid s \in \Sigma^*\}| \leq |Q|$.

Definition 3.8.3 Let $L \subseteq \Sigma^*$ be a language over Σ. Then the relation

$$R_L = \{\langle u, v \rangle \in \Sigma^* \times \Sigma^* \mid \forall w \in \Sigma^* : u \cdot w \in L \leftrightarrow v \cdot w \in L\}$$

is called the *Myhill-Nerode relation* for the language L.

Proposition 3.8.4 *Let $L \subseteq \Sigma^*$ be a language over Σ. Then the Myhill-Nerode relation is a right invariant equivalence relation.*

Proposition 3.8.5 *Let $L \subseteq \Sigma^*$ be a language over Σ such that the index of R_L is finite. Then L is an automaton language. For the deterministic classical FSA*

$$\mathcal{A}_L = \langle \Sigma, \{[s]_{R_L} \mid s \in \Sigma^*\}, [\varepsilon]_{R_L}, \{[s]_{R_L} \mid s \in L\}, \delta \rangle$$

with transition function $\delta = \{\langle [u]_{R_L}, a, [u \cdot a]_{R_L} \rangle \mid u \in \Sigma^*, a \in \Sigma\}$ we have $L(\mathcal{A}_L) = L$.

Proposition 3.8.6 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a deterministic classical FSA. Then $R_{\mathcal{A}} \subseteq R_{L(\mathcal{A})}$.*

Theorem 3.8.7 *For any deterministic classical FSA there exists a unique (up to state renaming) equivalent deterministic classical FSA that is minimal with respect to the number of states.*

Theorem 3.8.8 *Let $L \subseteq \Sigma^*$ be a classical language. Then L is a classical automaton language iff the index of R_L is finite.*

Combining Theorem 3.8.8 and Theorem 3.5.2 we obtain

Theorem 3.8.9 *Let $L \subseteq \Sigma^*$ be a classical language. Then the following are equivalent:*

1. *L is a classical automaton language,*
2. *L is a regular language,*
3. *The index of R_L is finite.*

Minimal deterministic classical FSA have characteristic structural properties.

Definition 3.8.10 Two states $q_1, q_2 \in Q$ are called *equivalent* iff $L_{\mathcal{A}}(q_1) = L_{\mathcal{A}}(q_2)$.

Remark 3.8.11 A deterministic classical finite-state automaton $\mathcal{A}$ is minimal iff each state is reachable and if distinct states of $\mathcal{A}$ are never equivalent.

3.9 Differences between classical and monoidal regular languages and finite-state automata

It is natural to ask which of the results obtained for classical regular languages and classical finite-state automata can be lifted to the more general monoidal case. A simple example (cf. [**?**]) shows that the set of monoidal regular languages is *not closed under intersection*: we consider the monoidal regular languages R_1, R_2 in the monoid $\langle \Sigma^* \times \Sigma^*, \cdot, \langle \varepsilon, \varepsilon \rangle \rangle$:

$$R_1 = \langle a, c \rangle^* \cdot \langle b, \varepsilon \rangle^* \qquad R_1 = \{ \langle a^n b^m, c^n \rangle \mid n, m \in \mathbb{N} \}$$
$$R_2 = \langle a, \varepsilon \rangle^* \cdot \langle b, c \rangle^* \qquad R_2 = \{ \langle a^m b^n, c^n \rangle \mid n, m \in \mathbb{N} \}$$

The intersection of R_1 and R_2 is

$$R_1 \cap R_2 = \{ \langle a^n b^n, c^n \rangle \mid n \in \mathbb{N} \}$$

which is not regular since the first projection is not a regular language - in the next chapter we show that monoidal regular languages are closed under projection, see Proposition **??**.

From the above property it follows directly that the set of monoidal regular languages is *not closed under complement and difference*. Otherwise using the de Morgan laws the monoidal regular languages would be closed under intersection.

Looking at the proof of Proposition 3.7.2 we see that the closure of classical regular languages an over alphabet Σ under complement is derived from the following observation. We may represent the language using a deterministic automaton with total transition function. Then for every word $w \in \Sigma^*$ there exist a unique path from the start state with label w. Since the monoidal regular languages are not closed under complement it is clear that there is no construction which ensures this "unique path" property for monoidal automata. This shows that the definition of a deterministic monoidal finite-state automaton is not natural. In the following chapters we shall discuss specialized notions of determinism for some classes of monoidal finite-state automata.

3.10 Summing up

We introduced monoidal finite-state automata and monoidal regular languages and showed that the former exactly recognize the monoidal regular languages (Theorem 3.5.2). We have seen that the class of monoidal languages recognized by monoidal finite-state automata is closed under union, concatenation, Kleene star, and homomorphic images (Proposition 3.2.1). Special cases of monoidal finite-state automata are classical finite-state automata, which are in addition closed under intersection, difference, complement, and reversion of languages (Proposition 3.7.3). Furthermore, for the latter class of automata there exists a determinization procedure (Theorem 3.7.1). Table 3.1 compares the general monoidal view with the classical (string based) view.

	Monoidal view	Classical (string) view
Device	Monoidal finite-state automaton	Classical finite-state automaton
Languages recognized	Monoidal automaton language	Classical automaton language
Closed under	Union, Concatenation, Kleene-Star Homomorphic images	Union, Concatenation, Kleene-Star Homomorphic images, Intersection, Complement, Difference, Reverse language
Equivalent lang. concept	Monoidal regular language	Classical regular language
Expressions	Monoidal regular expression	Classical regular expression
Operations for automata	Removal of e-transitions	Removal of ϵ-transitions Determinization, Minimization

Table 3.1: Finite-state automata and regular languages from a monoidal and a classical (string-based) view.

3.11 C(M) implementations for automata algorithms

This section describes the implementation of the constructions presented in this chapter in $C(M)$. Only algorithms for automata over the free monoid are presented.

C(M) programs for basic algorithms

Program 3.11.1 We start with the basic definitions and algorithms for finite-state automata. We note that the type of automaton used here has a *set* of initial states, and transition labels are arbitrary *words* over the input alphabet.

```
1  SYMBOL is ℕ;
2  ALPHABET is 2^SYMBOL;
3  WORD is SYMBOL*;
4  STATE is ℕ;
5  ATRANSITION is STATE × WORD × STATE;
6  FSA is ALPHABET × 2^STATE × 2^STATE × 2^STATE × 2^ATRANSITION;
7  newState : 2^STATE → STATE;
8  newState(Q) := |Q| + 1;
9  kNewStates : ℕ × 2^STATE → STATE*;
10 kNewStates(k, Q) := ⟨|Q| + 1, ..., |Q| + k⟩;
11 remap_FSA : FSA × 2^STATE → FSA;
12 remap_FSA((Σ, Q, I, F, Δ), Q') := (Σ, map(Q), map(I), map(F), Δ'), where
13     S := kNewStates(|Q|, Q');
14     map : STATE → STATE;
15     map(q) := (S)_q;
16     Δ' := {(map(q), c, map(r)) | (q, c, r) ∈ Δ};
17     ;
18 FSA_WORD : WORD → FSA;
19 FSA_WORD(a) := (set(a), {1, 2}, {1}, {2}, {(1, a, 2)});
```

In Lines 1-6 appropriate types are defined. Symbols are defined encoded as natural numbers, an $\mathcal{ALPHABET}$ as a set of symbols, a $\mathcal{WORD}$ as a list of symbols, and a $\mathcal{STATE}$ a natural number. For automaton transitions we introduce the type $\mathcal{ATRANSITION}$, each transition is a triple of source state, word and destination state. The type $\mathcal{FSA}$ for automata over the free monoid is defined to be a quintuple consisting of an alphabet, set of states, set of initial states, set of final states and set of transitions.

The function newState defined in Lines 7-8 takes a set of states (i.e., natural numbers) Q and returns the next new state outside Q. In this implementation we assume (and take care) that $Q = \{1, \ldots, |Q|\}$. Hence the new state can be simply defined as the number $|Q| + 1$. The function kNewState defined in Lines 9-10 returns the list of the next k new states $\langle |Q| + 1, \ldots, |Q| + k\rangle$ outside a given set Q.

The function $\text{remap}_{\text{FSA}}$ defined in Lines 11-17 takes as arguments an automaton $(\Sigma, Q, I, F, \Delta)$ and a set of states Q'. The function remaps all states in the

automaton to states outside Q'. In Line 13 S is defined as the list with $|Q|$ new states outside Q'. In Lines 14-15 we define the function map, which maps the state q (in Q) to the q-th state in the list S. Here again we assume that $Q = \{1, \ldots, |Q|\}$. In Line 16 the set of transitions Δ' is defined as variant of Δ where states have been renamed outside Q. Then in Line 12 the resulting automaton is defined as the quintuple consisting of the alphabet Σ, the images Q, I, F and the set of transitions Δ'.

The function $\mathrm{FSA}_{\mathrm{WORD}}$ constructs for a given word a the automaton recognizing $\{a\}$ as defined in Part 2 of Proposition 3.5.1.

Program 3.11.2 The following constructions present the regular operations union, concatenation, and Kleene star for finite-state automata.

20 $\mathrm{union}_{\mathrm{FSA}} : \mathcal{FSA} \times \mathcal{FSA} \to \mathcal{FSA}$;
21 $\mathrm{union}_{\mathrm{FSA}}((\Sigma_1, Q_1, I_1, F_1, \Delta_1), A_2) :=$
$(\Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2, I_1 \cup I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$, **where**
22 $(\Sigma_2, Q_2, I_2, F_2, \Delta_2) := \mathrm{remap}_{\mathrm{FSA}}(A_2, Q_1)$;
23 ;
24 $\mathrm{concat}_{\mathrm{FSA}} : \mathcal{FSA} \times \mathcal{FSA} \to \mathcal{FSA}$;
25 $\mathrm{concat}_{\mathrm{FSA}}((\Sigma_1, Q_1, I_1, F_1, \Delta_1), A_2) :=$
$(\Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2, I_1, F_2, (\Delta_1 \cup \Delta_2) \cup F_1 \times \{\varepsilon\} \times I_2)$, **where**
26 $(\Sigma_2, Q_2, I_2, F_2, \Delta_2) := \mathrm{remap}_{\mathrm{FSA}}(A_2, Q_1)$;
27 ;
28 $\mathrm{star}_{\mathrm{FSA}} : \mathcal{FSA} \to \mathcal{FSA}$;
29 $\mathrm{star}_{\mathrm{FSA}}(\Sigma, Q_1, I_1, F_1, \Delta_1) := (\Sigma, Q_1 \cup \{q_0\}, \{q_0\}, F_1 \cup \{q_0\}, \Delta)$, **where**
30 $q_0 := \mathrm{newState}(Q_1)$;
31 $\Delta := (\Delta_1 \cup \{(q_0, \varepsilon, q_1) \mid q_1 \in I_1\}) \cup \{(q_2, \varepsilon, q_0) \mid q_2 \in F_1\}$;
32 ;

Lines 20-23 describe the union of finite-state automata. In Line 22 we first remap the states of the second automaton in order to avoid common states. In Line 21 the union of two finite-state automata is defined exactly as in Part 1 of Proposition 3.2.1. In a very similar way the concatenation of two automata is defined in Lines 24-27, exactly following the description in Part 2 of Proposition 3.2.1. The Kleene star of an automaton is defined in Lines 28-32 in accordance with Part 3 Proposition 3.2.1.

Program 3.11.3 The next "supplementary constructions" are given for convenience. They can be expressed by our basis functions but help to keep the description of complex automata constructions transparent. We introduce the positive Kleene closure of a given automaton, optionality (adding the empty word) for an automaton, the automaton recognising a given set of symbols, and the automaton recognising all words over a given alphabet.

33 $\mathrm{plus}_{\mathrm{FSA}} : \mathcal{FSA} \to \mathcal{FSA}$;
34 $\mathrm{plus}_{\mathrm{FSA}}(\Sigma, Q_1, I_1, F_1, \Delta_1) := (\Sigma, Q_1 \cup \{q_0\}, \{q_0\}, F_1, \Delta)$, **where**
35 $q_0 := \mathrm{newState}(Q_1)$;
36 $\Delta := (\Delta_1 \cup \{(q_0, \varepsilon, q_1) \mid q_1 \in I_1\}) \cup \{(q_2, \varepsilon, q_0) \mid q_2 \in F_1\}$;
37 ;
38 $\mathrm{option}_{\mathrm{FSA}} : \mathcal{FSA} \to \mathcal{FSA}$;

```
39  option_FSA(Σ, Q, I, F, Δ) := (Σ, Q ∪ {q_0}, I ∪ {q_0}, F ∪ {q_0}, Δ), where
40      q_0 := newState(Q);
41      ;
42  symbolSet2FSA : 2^(SYMBOL) → FSA;
43  symbolSet2FSA(S) := (S, {1,2}, {1}, {2}, {(1, ⟨c⟩, 2) | c ∈ S});
44  all : ALPHABET → FSA;
45  all(Σ) := star_FSA(symbolSet2FSA(Σ));
```

Lines 33-37 define the positive Kleene closure in a very similar way like the Kleene star. The only difference is that the new starting state q_0 is not final. Lines 38-41 realize optionality by adding a new initial and final state q_0 to a given automaton in order to add the empty word to its language. Lines 42-43 define the function symbolSet2FSA which takes a set of symbols S and builds an automaton with two states recognizing S. If the set S is empty, the function symbolSet2FSA returns the automaton for the empty language. Lines 44-45 present the function all(Σ), which returns an automaton recognising all words over a given alphabet.

C(M) programs for ε-removal and further constructions

Program 3.11.4 The C(M) program for ε-removal closely follows Proposition 3.3.2.

```
46  removeEpsilon_FSA : FSA → FSA;
47  removeEpsilon_FSA(Σ, Q, I, F, Δ) := (Σ, Q, ⋃(C_ε(I)), F, Δ'), where
48      C := transitiveClosure({(q, r) | (q, a, r) ∈ Δ & a = ε});
49      C_ε := F_{1→2}(C ∪ {(q, q) | q ∈ Q});
50      Δ' := {(q_1, a, q_2) | (q_1, a, q') ∈ Δ & a ≠ ε, q_2 ∈ C_ε(q')};
51      ;
```

In Line 48, using Program 2.1.3, we define the transitive closure C of the ε-transitions in Δ. Then in Line 49 we construct the ε-closure by functionalizing the relation $C \cup \{(q, q) \mid q \in Q\}$. The resulting function C_ϵ maps each state $q \in Q$ to the set of all states that can be reached from q with a series of ε-transitions. In Line 50 and 47 Δ' and the resulting automaton are defined in accordance with the construction in Proposition 3.3.2.

Program 3.11.5 To "trim" an automaton means to remove all states that are not reachable from the initial states and all states from which no final state can be reached (cf. Definition 3.1.9).

```
52  trim_FSA : FSA → FSA;
53  trim_FSA(Σ, Q, I, F, Δ) :=
    (Σ, map(Q'), map(I ∩ Q'), map(F ∩ Q'), Δ'), where
54      R := transitiveClosure(Proj_(1,3)(Δ));
55      Q' := (I ∪ {r | (i, r) ∈ R & i ∈ I}) ∩ (F ∪ {r | (r, f) ∈ R & f ∈ F});
56      map : STATE → STATE;
57      map(q) := #_Q' as STATE*(q);
58      Δ' := {(map(q), c, map(r)) | (q, c, r) ∈ Δ & (q ∈ Q') ∧ (r ∈ Q')};
59      ;
```

In Line 54 we define the set R consisting of all pairs of states that are connected by a path of length ≥ 1. Here again we make use of Program 2.1.3. In Line 55 we define the set Q' of all states which are reachable from an initial state and from which at the same time a final state can be reached. Then in Lines 56-57 the mapping of the states in Q' to $\{1, \ldots, |Q'|\}$ is defined. Each state $q \in Q'$ is mapped to the index of q in Q', the latter set is here regarded as a list of states (Line 57). In Line 58 the new set of transitions Δ' is defined as the renamed variant of all transitions in Δ between states in Q'. In Line 53 we define the resulting automaton. The set of (resp. initial, final) states is the renamed variant of Q' (resp. $I \cap Q'$,$F \cap Q'$) and the set of transitions Δ'.

Program 3.11.6 Given an arbitrary finite-state automaton, our next program constructs a *classical* finite-states automaton, i.e., an automaton where each transition label is a word of length ≤ 1. Following Remark 3.1.15 we expand each transition of the form $\langle q, a_1 a_2 \ldots a_l, r\rangle$ with a transition label of length $l > 1$ into a sequence of l transitions $\langle q' = t_1, a_1, t_2\rangle \langle t_2, a_2, t_3\rangle \ldots \langle t_l, a_l, t_{l+1} = r\rangle$ by introducing $l - 1$ new intermediate states $t_2, t_3, \ldots, t_l$.

```
60  expand_FSA : FSA → FSA;
61  expand_FSA(Σ, Q, I, F, Δ) := (Σ, Q', I, F, Δ'), where
62      L := ⟨(q', a', r') | (q', a', r') ∈ Δ & |a'| > 1⟩;
63      (Q', Δ') := induction
64          step 0 :
65              (Q'^(0), Δ'^(0)) := (Q, Δ);
66          step n + 1 :
67              (q, a, r) := (L)_{n+1};
68              l := |a| ;
69              t := ⟨q⟩ · kNewStates(l − 1, Q'^(n)) · ⟨r⟩;
70              Q'^(n+1) := Q'^(n) ∪ set(t);
71              Δ'^(n+1) := Δ'^(n) \ {(q, a, r)} ∪ {((t)_i, ⟨(a)_i⟩, (t)_{i+1}) | i ∈ {1, ..., l}};
72          until n = |L|
73          ;
74      ;
```

The algorithm builds the resulting automaton by inductively defining the new set of states Q' and the new set of transitions Δ'. At each step we eliminate one transition $(q, a, r) \in \Delta$ such that $|a| > 1$. First, in Line 62 we extract from Δ the list L of such transitions. The induction starts with the original set of states and transitions in Line 65. In the inductive step in Line 67 we select the next transition (q, a, r) from the list L. In Line 69 t is defined as the sequence of states with first element q, followed by $l - 1$ new states and ending with r. In Line 70 $Q'^{(n+1)}$ is defined as the union of $Q'^{(n)}$ with the states in t. In the transition set Δ' we replace (q, a, r) by the transitions (t_i, a_i, t_{i+1}) for $i = 1, \ldots, l$ (Line 71). The induction ends when the list L is exhausted. The resulting automaton is then defined as the automaton with set of states Q' and set of transitions Δ'. Initial and final states are as in the source automaton (Line 61).

C(M) programs for determinization, intersection and difference of automata

In our description, the type of *deterministic* finite-state automata is distinct from the type of general finite-state automata: deterministic finite-state automata have a single initial state (as opposed to a set of initial states) and a transition function (as opposed to a relation). We now introduce the new types needed and a variant of the trimming construction for deterministic finite-state automata.

Program 3.11.7 As in the general case (cf. Program 3.11.5), to "trim" a deterministic automaton means to remove all states that are not reachable from the initial states and all states from which no final state can be reached.

```
75  DTRANSITIONS is STATE × SYMBOL → STATE;
76  DFSA is ALPHABET × 2^STATE × STATE × 2^STATE ×
    DTRANSITIONS;
77  trim_DFSA : DFSA → DFSA;
78  trim_DFSA(Σ, Q, q0, F, Δ) :=
      { (Σ, map(Q'), map(q0), map(F ∩ Q'), Δ')   if Q' ≠ ∅
      { (Σ, {1}, 1, ∅, ∅)                        otherwise   , where
79      R := transitiveClosure({(s, d) | ((s, a), d) ∈ Δ});
80      Q' := ({q0} ∪ {r | (q0, r) ∈ R}) ∩ (F ∪ {r | (r, f) ∈ R & f ∈ F});
81      map : STATE → STATE;
82      map(q) := #_Q' as STATE*(q);
83      Δ' := {((map(q), c), map(r)) | ((q, c), r) ∈ Δ & (q ∈ Q') ∧ (r ∈ Q')};
84      ;
```

In Line 75 the type $\mathcal{DTRANSITIONS}$ for the transition function of a deterministic finite-state automaton is defined. Then in Line 76 a $\mathcal{DFSA}$ is defined as a quintuple consisting of an alphabet, set of states, initial state, set of final states and a transition function. The trim function $\text{trim}_{\text{DFSA}}$ defined in Lines 77-84 is almost identical to the corresponding function for non-deterministic automata (Program 3.11.5). The only difference is in Line 78 that in case of the empty automata we return an automaton with one state (instead of no states).

Program 3.11.8 In Theorem 3.7.1 we presented a simple determinization construction for finite-state automata. This construction always builds an automaton with $2^{|Q|}$ states, not taking into account that many of the states may not be reachable from the resulting new initial state. Below we present a more efficient construction. It builds a deterministic automaton where all states are reachable from the new initial state. As a first preparation step it computes a non-deterministic automaton where all transition labels have length 1. Then the deterministic automaton is built inductively, starting from the set of initial states (acting as the new initial state). While the construction proceeds, a list of new states introduced is maintained. Each new state is a set of old states. At each induction step we treat the next element in the list and compute, for each symbol σ of the input alphabet, the sets of states which are reachable from the current set with σ. Each new set of states obtained in this way is added as a new state to the tail of the list of new states. The induction ends when all sets

in the list have been treated.

```
 85  determ_FSA : FSA → DFSA;
 86  determ_FSA(A) := trim_DFSA(Σ, {1, ..., |P|}, 1, F', δ'), where
 87      (Σ, Q, I, F, Δ) := expand_FSA(removeEpsilon_FSA(A));
 88      Δ' := F_{1→(2,3)}({(q, (α)_1, r) | (q, α, r) ∈ Δ});
 89      (P, δ') ∈ (2^STATES)* × DTRANSITIONS;
 90      (P, δ') := induction
 91          step 0 :
 92              P^(0) := ⟨I⟩;
 93              δ'^(0) := ∅;
 94          step n + 1 :
 95              N := ⋃(Δ'((P^(n))_{n+1}));
 96              N' := F_{1→2}(N);
 97              P^(n+1) := P^(n) · ⟨q | q ∈ Proj_2(N') & q ∉ P^(n)⟩;
 98              δ'^(n+1) := δ'^(n) ∪ {((n + 1, c), #_{P^(n+1)}(q)) | (c, q) ∈ N'};
 99          until n = |P^(n)|
100          ;
101      F' := {q | q ∈ {1, ..., |P|} & F ∩ (P)_q ≠ ∅};
102      ;
```

The preliminary first step is described in Line 87: ε-transitions are removed using Program 3.11.4 and transitions labels of length ≥ 2 are replaced by sequences of transitions with labels of length 1 using Program 3.11.6. Then the main step starts. In Line 88 we define Δ' as the functionalization $1 \to (2,3)$ of Δ. Δ' maps a state $q \in Q$ to the set of pairs (a,r), such that $(q,a,r) \in \Delta$. Only the source states of transitions appear in the domain of Δ', hence images are always non-empty sets. In Lines 89-100 the states of the deterministic automaton are constructed by induction. Each new state is a set of states of the source automaton (in the final version, each new state is translated into a natural number, see below). New states to be introduced are maintained in the list P. In parallel the transition function δ' is constructed. In Lines 92-93 the base of the induction is defined. $P^{(0)}$ contains only the set I (new initial state) and $\delta'^{(0)}$ is the empty function. In the inductive step in Line 95 we first take the next ($n+1$-st) element $P^{(n)})_{n+1}$ from the current list $P^{(n)}$. N is defined as the set of all pairs (c,q) found in the images of elements of $P^{(n)})_{n+1}$ under Δ'. At this point, if $P^{(n)})_{n+1}$ is empty or Δ' is undefined for all elements of $P^{(n)})_{n+1}$ we obtain the empty set. In Line 96 N' is defined as the function which maps a label c to the set of states which are reached with a c-transition from states in $(P^{(n)})_{n+1}$. In Line 97 we add to the current list of new states $P^{(n)}$ all state sets in the range of N' that yet do not occur in $P^{(n)}$. In Line 98 the function δ' is extended with the transitions from the current set of states to the corresponding sets defined with N'. At this point each set of states (new state) is mapped to its index in the list P. In this way, new states are now numbers. The induction ends when the list P is exhausted (Line 99). In Line 101 the set of final states F' is defined as the set of the indices of sets with non empty intersection with F. Finally in Line 86 the resulting automaton is defined. The set of states in the translated representation is the set of indices $\{1,\ldots,|P|\}$, the initial state is 1. The automaton is trimmed using Program 3.11.7.

The construction for intersection and difference of automata proceeds by building the Cartesian product automaton (cf. Proposition 3.7.3). We again optimise the process by inductively constructing only those (pairs of) states which can be reached from the new initial state. For both constructions we start from the pair of the two initial states of the input automata (new initial state) and then proceed by extending the automaton with the states and transitions which are directly reachable from states already obtained. The following auxiliary construction is used.

Program 3.11.9 The following algorithm (product_Δ) constructs an automaton which is the Cartesian product of two input automata. We assume that the source automata are deterministic. The arguments are the initial states and the transition functions of the two input automata. The result is the list of pairs of states and the transition function of the resulting automata. Here again the states of the resulting automaton are finally represented using the indices of the pairs of states in the list.

```
103  product_Δ : (STATE × DTRANSITIONS) × (STATE ×
     DTRANSITIONS) → (STATE × STATE)* × DTRANSITIONS;
104  product_Δ((s1, δ1), (s2, δ2)) := (P, δ), where
105      Δ'1 := F_{1→(2,3)}(δ1 as 2^{STATE×SYMBOL×STATE});
106      (P, δ) ∈ (STATE × STATE)* × DTRANSITIONS;
107      (P, δ) := induction
108          step 0 :
109              P^(0) := ⟨(s1, s2)⟩;
110              δ^(0) := ∅;
111          step n + 1 :
112              (p1, p2) := (P^(n))_{n+1};
113              N :=
         { {(c, (q1, δ2(p2, c))) | (c, q1) ∈ Δ'1(p1) & ! δ2(p2, c)}   if ! Δ'1(p1)
         { ∅                                                          otherwise   ;
114              P^(n+1) := P^(n) · ⟨p | p ∈ Proj_2(N) & p ∉ P^(n)⟩;
115              δ^(n+1) := δ^(n) ∪ {((n + 1, c), #_{P^(n+1)}(q)) | (c, q) ∈ N};
116          until n = |P^(n)|
117          ;
118      ;
```

In Line 105 of the construction we define Δ'_1 as the function that maps a state from the first source automaton to a set of pairs of label symbol and destination state in the transition function δ_1. Then, the list of pairs of states P and the resulting transition function δ are constructed by induction. In the base step in Lines 109-110, P is set to the list with the only pair (s_1, s_2), and δ is empty. Then, in the inductive step we consider (p_1, p_2) – the $n+1$-st element of the list P. In Line 113 we define the set N. If $\Delta'_1(p_1)$ is defined, then N is the set of pairs $(c, (q_1, q_2))$ such that (c, q_1) in $\Delta'_1(p_1)$ (which means that $q_1 = \delta_1(p_1, c)$) and $\delta_2(p_2, c) = q_2$ is defined. Otherwise N is empty. In Line 114 we add to P the pairs of states in the second projection of N that are not found in the current set $P^{(n)}$. In Line 115 the function δ is extended with the transitions from the current pair of states to the corresponding pairs defined with N. Here the pairs of states are mapped to their corresponding indices in the list P. The

induction ends when the list P is exhausted (Line 116).

The above function is the essential part of the programs for intersection and difference to be described now.

Program 3.11.10 The following C(M) program implements intersection of finite-state automata as described in Part 1 of Proposition 3.7.3.

```
119  intersect_DFSA : DFSA × DFSA → DFSA;
120  intersect_DFSA((Σ1, Q1, s1, F1, δ1), (Σ2, Q2, s2, F2, δ2)) :=
     trim_DFSA(Σ1 ∪ Σ2, Q, 1, F, δ), where
121      (P, δ) := product_Δ((s1, δ1), (s2, δ2));
122      Q := {1, ..., |P|};
123      F := {q | q ∈ Q & (Proj_1((P)_q) ∈ F1) ∧ (Proj_2((P)_q) ∈ F2)};
124      ;
125  FSA_DFSA : DFSA → FSA;
126  FSA_DFSA(Σ, Q, q0, F, δ) :=
     { (Σ, ∅, ∅, ∅, ∅)                                  if F = ∅
     { (Σ, Q, {q0}, F, {(q, ⟨a⟩, r) | ((q, a), r) ∈ δ})  otherwise ;
127  intersect_FSA : FSA × FSA → FSA;
128  intersect_FSA(A1, A2) :=
     FSA_DFSA(intersect_DFSA(determ_FSA(A1), determ_FSA(A2)));
```

The function $\text{intersect}_{\text{DFSA}}$ constructs the intersection of two input automata. In Line 121 the Cartesian product of the two automata, (P, δ), is constructed using Program 3.11.9. Then in Line 122 the states of the automaton are defined as the set $\{1, \ldots, |P|\}$. In Line 123 the final states are defined as the indices of the state pairs for which both states are final in their corresponding automata. After trimming in Line 120 the automaton is returned.

We also want to have an intersection construction for non-deterministic automata. In Lines 125-126 we define the function FSA_{DFSA}, which converts a deterministic finite-state automaton to an arbitrary finite-state automaton. In Lines 127-128 the function $\text{intersect}_{\text{FSA}}$ is defined. This function intersects two non-deterministic automata by first determinizing the input automata (Program 3.11.8) and then constructing the intersection using the above function for deterministic automata. The resulting deterministic automaton is finally converted to a non-deterministic automaton.

Program 3.11.11 The next C(M) program implements difference of finite-state automata as described in Part 1 of Proposition 3.7.3.

```
129  diff_DFSA : DFSA × DFSA → DFSA;
130  diff_DFSA((Σ1, Q1, s1, F1, δ1), (Σ2, Q2, s2, F2, δ2)) :=
     trim_DFSA(Σ1 ∪ Σ2, Q, 1, F, δ), where
131      δ2' ∈ DTRANSITIONS;
132      δ2'(q, c) := { δ2(q, c)   if !δ2(q, c)
                     { 0          otherwise ;
133      (P, δ) := product_Δ((s1, δ1), (s2, δ2'));
134      Q := {1, ..., |P|};
135      F := {q | q ∈ Q & (Proj_1((P)_q) ∈ F1) ∧ (Proj_2((P)_q) ∉ F2)};
```

```
136       ;
137   diff_FSA : FSA × FSA → FSA;
138   diff_FSA(A_1, A_2) := FSA_DFSA(diff_DFSA(determ_FSA(A_1), determ_FSA(A_2)));
```

The first function $\mathrm{diff}_{\mathrm{DFSA}}$ constructs the difference of two deterministic automata. Since the second automaton has to be total (see Proposition 3.7.3), we extend in Line 132 the transition function δ_2 by complementing it with transitions to a fail state (0). Then in Line 133 the Cartesian product of the two automata (P, δ) is constructed using again Program 3.11.9. In Line 134 the set of states of the automaton is defined as $\{1, \ldots, |P|\}$. In Line 135 the final states are defined to be the indices of those pairs of states for which the state of the first automaton is final and the state in the second automaton is not final. After trimming in Line 130 the automaton is returned as result.

In Lines 137-138 the function $\mathrm{diff}_{\mathrm{FSA}}$ is defined. This function computes the difference of two non-deterministic automata by first determinizing them (Program 3.11.8) and then constructing the difference using the above function for deterministic automata. The resulting deterministic automaton is converted to a non-deterministic one at the end.

C(M) programs for minimizing automata

In this subsection we present an algorithm which given a deterministic automaton constructs the equivalent minimal automaton as characterized in Section 3.8. Let us consider Definition 3.8.10 and Remark 3.8.11. To construct the minimal automaton we have to build the classes of equivalent states of the source automaton. Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a deterministic finite-state automaton. Clearly

$$\begin{aligned} q \equiv p &\Leftrightarrow L_{\mathcal{A}}(q) = L_{\mathcal{A}}(p) \\ &\Leftrightarrow \forall \alpha \in \Sigma^* : (\delta^*(q,\alpha) \in F \leftrightarrow \delta^*(p,\alpha) \in F) \\ &\Leftrightarrow \forall i \in \mathbb{N}\ \forall j \leq i\ \forall \alpha \in \Sigma^j : (\delta^*(q,\alpha) \in F \leftrightarrow \delta^*(p,\alpha) \in F). \end{aligned}$$

Let us define:

$$q\ R_i\ p \ :\Leftrightarrow\ \forall j \leq i\ \forall \alpha \in \Sigma^j : (\delta^*(q,\alpha) \in F \leftrightarrow \delta^*(p,\alpha) \in F).$$

This implies that:

$$q \equiv p \leftrightarrow \forall i \in \mathbb{N} : q\ R_i\ p.$$

This allows us to construct the relation inductively. For $i = 0$ we have

$$q\ R_0\ p \leftrightarrow (q \in F \leftrightarrow p \in F).$$

The inductive step is:

$$\begin{aligned} q\ R_{i+1}\ p &\Leftrightarrow \forall j \leq i+1\ \forall \alpha \in \Sigma^j : (\delta^*(q,\alpha) \in F \leftrightarrow \delta^*(p,\alpha) \in F) \\ &\Leftrightarrow \forall j \leq i\ \forall \alpha \in \Sigma^j : (\delta^*(q,\alpha) \in F \leftrightarrow \delta^*(p,\alpha) \in F) \wedge \\ &\qquad \forall \alpha \in \Sigma^{i+1} : (\delta^*(q,\alpha) \in F \leftrightarrow \delta^*(p,\alpha) \in F) \\ &\Leftrightarrow q\ R_i\ p \wedge \forall \alpha \in \Sigma^{i+1} : (\delta^*(q,\alpha) \in F \leftrightarrow \delta^*(p,\alpha) \in F) \\ &\Leftrightarrow q\ R_i\ p \wedge \forall a \in \Sigma\ \forall \alpha_0 \in \Sigma^i : \\ &\qquad (\delta^*(\delta(q,a),\alpha_0) \in F \leftrightarrow \delta^*(\delta(p,a),\alpha_0) \in F) \\ &\Leftrightarrow q\ R_i\ p \wedge \forall a \in \Sigma : \delta(q,a)\ R_i\ \delta(p,a). \end{aligned}$$

Corollary 3.11.12 *Let $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ be a deterministic finite-state automaton. Then the relation $R = \bigcap_{i=0}^{\infty} R_i$, where*

$$\begin{aligned} q\ R_0\ p &\Leftrightarrow (q \in F \leftrightarrow p \in F) \\ q\ R_{i+1}\ p &\Leftrightarrow q\ R_i\ p \wedge \forall a \in \Sigma : \delta(q,a)\ R_i\ \delta(p,a) \end{aligned}$$

coincides with the Myhill-Nerode equivalence relation (cf. Def. 3.8.3) for $L(\mathcal{A})$.

Definition 3.11.13 Let $g : Q \to X$. Then the equivalence relation $Ker_Q(g) \subseteq Q \times Q$ defined as

$$\langle p, q \rangle \in ker_Q(g) :\Leftrightarrow g(p) = g(q)$$

is called the *kernel* of g over Q.

We denote the intersection of the equivalence relations $R_1, R_2 \subset Q \times Q$ as $R_1 \cap R_2$. Obviously the intersection of regular relations is a regular relation.

Proposition 3.11.14 *Let us define $f : Q \to \{0, 1\}$*

$$f(q) := \begin{cases} 1 & \text{if } q \in F \\ 0 & \text{otherwise} \end{cases}$$

and for every $a \in \Sigma$ and $i \in \mathbb{N}$ the function $f_a^{(i)} : Q \to Q/R_i$ as

$$f_a^{(i)}(q) := [\delta(q,a)]_{R_i}.$$

Then

1. $R_0 = ker_Q(f)$
2. $R_{i+1} = \bigcap_{a \in \Sigma} ker_Q(f_a^{(i)}) \cap R_i$

Program 3.11.15 The algorithm implements a minimization algorithm for deterministic finite-state automata which is based on the inductive construction presented in Corollary 3.11.12 and uses the definitions given in Proposition 3.11.14.

```
139  ℰ𝒬ℛℰℒ is 𝒮𝒯𝒜𝒯ℰ → 𝒮𝒯𝒜𝒯ℰ;
140  ker : 2^(𝒮𝒯𝒜𝒯ℰ) × (𝒮𝒯𝒜𝒯ℰ → ℕ) → ℰ𝒬ℛℰℒ;
141  ker(Q, g) := {(q, #_I(g(q))) | q ∈ Q}, where
142      I := ⟨g(q) | q ∈ Q⟩;
143      ;
144  intersect_EQREL : 2^(𝒮𝒯𝒜𝒯ℰ) × ℰ𝒬ℛℰℒ × ℰ𝒬ℛℰℒ → ℰ𝒬ℛℰℒ;
145  intersect_EQREL(Q, R1, R2) := {(q, #_I((R1(q), R2(q)))) | q ∈ Q}, where
146      I := ⟨(R1(q), R2(q)) | q ∈ Q⟩;
147      ;
148  minimal_DFSA : 𝒟ℱ𝒮𝒜 → 𝒟ℱ𝒮𝒜;
149  minimal_DFSA(Σ, Q, q0, F, δ) := trim_DFSA(Σ, R(Q), R(q0), R(F), δ'), where
150      R ∈ ℰ𝒬ℛℰℒ;
151      k ∈ ℕ;
152      (R, k) := induction
153          step 0 :
```

154 $\quad\quad\quad\quad \mathrm{f} : \mathcal{STATE} \to \mathbb{N};$

155 $\quad\quad\quad\quad \mathrm{f}(q) := \begin{cases} 1 & \textbf{if } q \in F \\ 0 & \textbf{otherwise} \end{cases};$

156 $\quad\quad\quad\quad k^{(0)} := 0;$

157 $\quad\quad\quad\quad R^{(0)} := \ker(Q, f);$

158 $\quad\quad\quad$ **step** $n+1$:

159 $\quad\quad\quad\quad \mathrm{f} : \mathcal{SYMBOL} \times \mathcal{STATE} \to \mathbb{N};$

160 $\quad\quad\quad\quad \mathrm{f}(c, q) := \begin{cases} \mathrm{R^{(n)}}(\delta(q, c)) & \textbf{if } !\,\delta(q, c) \\ 0 & \textbf{otherwise} \end{cases};$

161 $\quad\quad\quad\quad k^{(n+1)} := \left|\mathrm{Proj}_2(R^{(n)})\right|;$

162 $\quad\quad\quad\quad R^{(n+1)} := \mathrm{intersect}_{\mathrm{EQREL}}(Q, R^{(n)}, (\mathrm{intersect}_{\mathrm{EQREL}}(Q))(\langle \ker(Q, \mathrm{f}(c)) \mid c \in \Sigma \rangle));$

163 $\quad\quad\quad$ **until** $k^{(n)} = \left|\mathrm{Proj}_2(R^{(n)})\right|$

164 $\quad\quad\quad$;

165 $\quad\quad \delta' \in \mathcal{DTRANSITIONS};$

166 $\quad\quad \delta' := \{((\mathrm{R}(q), \sigma), \mathrm{R}(r)) \mid ((q, \sigma), r) \in \delta\};$

167 $\quad\quad$;

In Line 139 we define the type of an equivalence relation as a function mapping a state to (the number of) its equivalence class. We assume that the equivalence classes are numbered sequentially starting from 1. In Lines 140-143 we construct the kernel equivalence relation for a function on the automata states. In Lines 144-147 we construct the intersection of two equivalence relations. The equivalence class of the intersection for a given state q is defined as the index of the pair of the two equivalence classes of q with respect to the two source relations in the list I of all pairs of classes. Afterwards in Lines 148-167 we present the minimization algorithm. We construct the equivalence relation R by induction. In $k^{(n+1)}$ we store the number of equivalence classes of the relation in the previous step – $R^{(n)}$. The base of the induction is defined in Lines 154-157 in accordance with Proposition 3.11.14 point 1. For the inductive step in Lines 159-162 we use the definition given in Proposition 3.11.14 point 2. The induction ends when the number of classes of $R^{(n)}$ is equal to $k^{(n)}$ which is the number of classes of $R^{(n-1)}$ (Line 163). In Line 165-166 we map the states of the transitions to the classes of R and in Line 149 The automaton states, start state and final states are mapped to the equivalence classes. The algorithm returns the resulting automaton after trimming.