INSTITUT FÜR INFORMATIK
UND PRAKTISCHE MATHEMATIK

Report on the Program AMoRE

O. Matz, A. Miller, A. Potthoff,
W. Thomas, E. Valkema

Bericht Nr. 9507
October 1995

CHRISTIAN-ALBRECHTS-UNIVERSITÄT
KIEL

Report on the Program AMoRE

O. Matz, A. Miller, A. Potthoff,
W. Thomas, E. Valkema

October 1995[1]

Bericht 9507
Institut für Informatik und Praktische Mathematik
der Christian-Albrechts-Universität zu Kiel

[1] modified January 1997 for LaTeX2e, minor differences to previous versions of this document.

Preface

This report describes a program which offers a number of procedures for computations related to finite automata. AMoRE stands for "Automata, Monoids, and Regular Expressions".

The program was developed since 1986, first at the Technical University of Aachen, from 1990 onwards (after the move of the group to Kiel) at the University of Kiel.

Many people have contributed to this work, not only the persons listed as authors of this report. Until 1990, contributions were made in Aachen also by Albert Maier, Michael Fahn, Herbert Grochtmann, Vera Jansen, Detlef Lippert, and Udo Wermuth. In Kiel several students have worked on AMoRE and experimental extensions, among them Kais Haddar, Laurent Hirsinger, Thomas Kahlert, Pierre Manches, Savvas Nikolaides, Peter Stach, Michael Steinmann, Jens Vöge. Special thanks are due to Heidi Luca-Gottschalk, the AMoRE system administrator. By her efforts AMoRE has been opened to an efficient use by many external users around the world.

Financial support was provided by Deutsche Forschungsgemeinschaft (projects Th 352/2-1 and 3-1) as well as by the ESPRIT Working Group No. 3166/6317 ASMICS ("Algebraic and Syntactic Methods in Computer Science").

Ongoing work aims at a wider scope of the system, including procedures involving logical formulas and automata on infinite words, and other related objects like finite trees instead of words. It turned out that the architecture of AMoRE has to be modified considerably to allow a convenient inclusion of such procedures. This was a motivation to finish the work on the existing "kernel" by the present detailed documentation.

For the accessibility and use of AMoRE see the Copyright Note at the end of this volume. Even after a long time of development, some errors will still remain in the presently 27,000 lines (about 1 MByte) of C-Code. We kindly ask users to communicate queries, problems, and detected errors to the authors. Nevertheless, we hope that the program will be useful to all who are interested in automata theoretic algorithms and their application.

W. Thomas

Contents

Chapter 1

Introduction

1.1 The Purpose of AMoRE

The theory of regular languages and finite automata involves a number of basic and interesting algorithms. Examples are the algorithms which convert regular expressions into finite automata and vice versa, transform nondeterministic automata into deterministic ones and produce the minimal deterministic finite automaton for a given regular language.

The semigroup theoretical approach to formal languages, which connects regular languages with (varieties of) finite monoids, leads to further algorithms, among them the construction of the syntactic monoid of a regular language and the decomposition of this monoid according to Green's relations.

The program AMoRE (short for "Automata, Monoids, and Regular Expressions") consists of procedures which implement these and related algorithms, including a generation of automaton state graphs on the screen.

There were several motivations to set up such a program. First, it is meant to be a tool which supports researchers in the analysis of language examples that are relevant to either theoretical questions or some application. If it is necessary to calculate the minimal automaton or the syntactic monoid of a regular language, the task is often tedious or even infeasible when done by hand. Secondly, the system can be used in teaching automata theory. For example, an application of the semigroup decomposition by Green's relations is the decidability of language properties like "starfree" or "piecewise testable". This can be illustrated nicely in nontrivial examples when a suitable program is available.

The questions involved in making the algorithms practical and efficient were another motivation. In the literature (e.g., [AHU74]) algorithmic results of automata theory are taken as key examples for the design of computer algorithms. In some cases, it turned out that even algorithms with an exponential worst case behavior could be used in most practical examples. At this point, an implementation supports case studies for theoretical investigations on the applicability of algorithms. Also some deeper problems of automata theory might be attacked

on the basis of such case studies, e.g. the unsolved question of simplification of regular expressions.

There are several tracks in which the scope of the procedures of AMoRE could be extended: the integration of diverse applications (e.g. in pattern matching or code optimization), the extension to automata theoretic algorithms over generalized domains (besides finite words also infinite words, trees, or even graphs), and the combination with logical methods (e.g., temporal logic, and the application to program verification and synthesis). Ongoing work is concerned with the latter two fields but not included in the present version of AMoRE.

1.2 Use of the Report and Functions of AMoRE

The purpose of this report is threefold: Firstly, it shall serve the user as a kind of manual to enable her or him to use AMoRE's features properly. Secondly, it is meant to be also a programmer's manual that describes how to install AMoRE and provides help in case problems occur or one wishes to adapt and extend AMoRE to new purposes. Thirdly, this report explains the theoretical background of the implemented algorithms.

However, for a complete understanding of this background and also for proofs, the reader will have to read the original papers. See for example [Pe90] or [Pi86]. The latter is recommended especially for semigroup theory.

The Figure 1.1 summarizes the main functions of AMoRE. Each of the boxes contains a language representation. The edges between them stand for the possible conversions AMoRE can perform and are labelled with the chapters of this report that deal with the respective functions. The doubly framed boxes mark those language representations that are possible inputs. Arrows into one of the ovals named "tests" and "operations" mean that tests or operations require that language representation as input. Labels on these edges, again, give the chapters in which the corresponding functions are explained in detail. (The conversion into starfree expressions is, of course, only available if the language is starfree, i.e. if such a representation exists.)

AMoRE offers also some tests for certain language properties as well as some operations that create new languages from previously defined ones. Figure 1.2 gives an overview of these tests and operations and the language representations they work on.

Both of these diagrams, however, are of no concern for the user since AMoRE will perform necessary intermediate calculations without explicit command. For example, if a user has defined a language by a regular expression and asks AMoRE whether it is starfree, then AMoRE will first compute an equivalent NFA, afterwards a DFA, then the minimal DFA and finally the syntactic monoid, on which it will perform the starfreeness test.

AMoRE's functions are comparable and complementary to other existing soft-

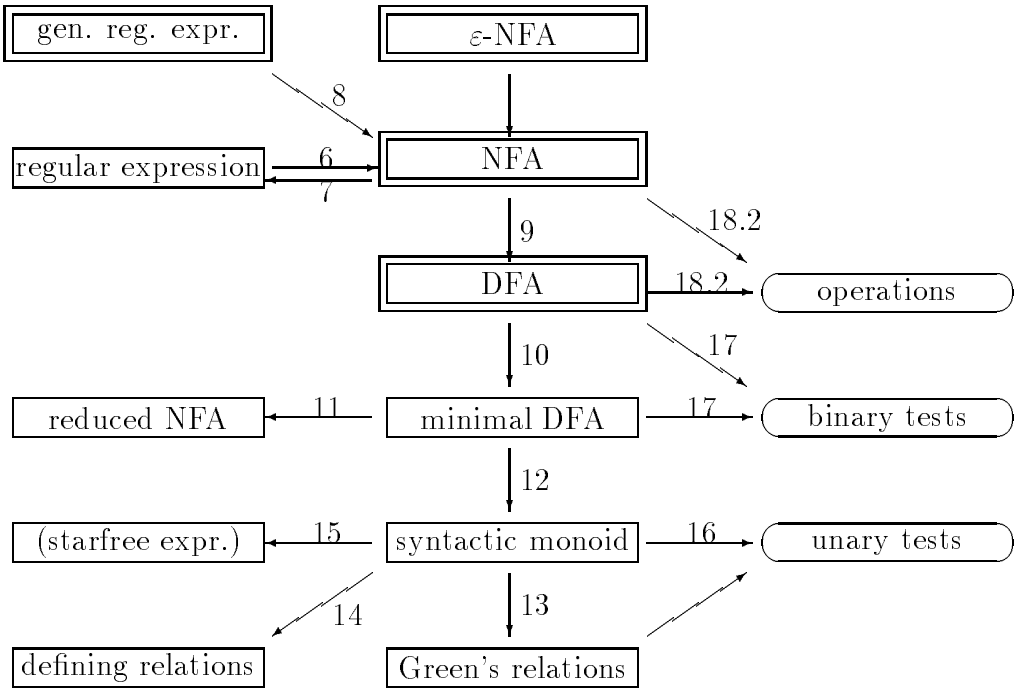


Figure 1.1: AMoRE's Functions in a Diagram.

ware packages for automata theory, such as AUTOMATE [ChH91] and Grail [RW93]. Specific features of AMoRE are its treatment of generalized and starfree expressions, the reduction of nondeterministic automata, its monoid decomposition part, and the display of transition graphs.

tests on minDFA	
unary	binary
empty full	equiv.

tests on DFA
binary
disjointness inclusion

tests on monoid/Green's relations
unary
starfree in $\mathrm{FOL}_{\mathrm{U}}$ ([MNP72]) dot-depth one locally testable piecewise testable generalized definite definite reverse definite

operations on NFA	
unary	binary
star reverse	union concatenation left quotient right quotient shuffle product

operations on DFA	
unary	binary
minimum maximum prefix	intersection right quotient set minus

Figure 1.2: AMoRE's Tests and Operations

1.3 Some Terminology

In this section we will introduce some notations that will be used throughout the report. We assume that the reader is familiar with most of them and also with the basic results of the theory of regular (recognizable, rational) languages, concerning for example the equivalence of the expressive power of the different representations of regular languages.

1.3.1 Sets, Mappings and Words

For a set M, we denote by 2^M the powerset of M. If f is a total mapping into B defined on the set A, we will write $f : A \longrightarrow B$ and call A the domain of f.

The notation $f : A \dashrightarrow B$ indicates a partial mapping, where the domain of f is a subset of A. We will identify this f with the (total) mapping from A to $B \cup \{\bot\}$, given by the assignment

$$a \mapsto \begin{cases} f(a) & \text{if } a \text{ is in the domain of } f \\ \bot & \text{else,} \end{cases}$$

where $\bot$ is a special symbol (called "undefined" or "bottom").

We identify $f : A \dashrightarrow B$ with the function from 2^A to 2^B, given by $M \mapsto \{f(a) \mid a \in M \text{ is in the domain of } f\}$. We will denote this function again by f.

Then f^{-1} is function from 2^B to 2^A, given by $M \mapsto \{a \in A \mid f(a) \in M\}$. If $b \in B$, we will sometimes identify b and $\{b\}$ and simply write $f^{-1}(b)$ instead of $f^{-1}(\{b\})$

If $f : A \longrightarrow B$ is injective, then f^{-1} will also denote the function $B \dashrightarrow A$,

$$b \mapsto \begin{cases} a & \text{if } f(a) = b \\ \bot & \text{if } \neg\exists a \in A : f(a) = b \end{cases}$$

Everywhere in this paper, we consider Σ as non-empty finite set called *alphabet*, whose elements are *letters*. Finite sequences of letters will be called *words*. The empty word is denoted by the reserved symbol ε. Σ^* denotes the set of all finite sequences of letters of Σ, and $\Sigma^+ := \Sigma^* \backslash \{\varepsilon\}$. For a word u we will denote its length by $|u|$.

As usual, for words $u, v \in \Sigma^*$ we use the juxtaposition uv for the concatenation of u and v.

Σ^+ together with the juxtaposition forms a semigroup, and Σ^* together with the juxtaposition forms a monoid with the neutral element ε.

If there is a total ordering $<$ on the set Σ, we define the canonical ordering $<$ on Σ^* as follows: For u,v we let

$$u < v \overset{\text{def}}{\iff} |u| < |v| \vee (|u| = |v| \wedge \exists x, y, z \in \Sigma^* \, \exists a, b \in \Sigma : u = xaz, v = xby \wedge a < b).$$

So a word u is smaller than a word v, if it is either shorter or of the same length as v but for the first distinct letters a and b of u and v, respectively, we have $a < b$.

This ordering will be used for the description of equivalence relations, where classes are usually given by their smallest representatives. As in practical implementation the letters of Σ are identified with integers, the ordering of Σ is always fixed in a canonical way.

1.3.2 Automata

A *nondeterministic finite automaton (NFA)* over the alphabet Σ is a quintuple $A = (\Sigma, Q, I, \Delta, F)$, where Q is a nonempty finite set of *states*, $I \subseteq Q$ is the nonempty set of *initial states*, $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation* and $F \subseteq Q$ is the set of *final states*. An NFA is called *deterministic* (and is therefore denoted by DFA), iff the following conditions hold:

- $\forall p \in Q, a \in \Sigma \, |\{q \,|\, (p, a, q) \in \Delta\}| = 1$
- $|I| = 1$

In this case we will write $(\Sigma, Q, q_0, \delta, F)$ rather than $(Q, \Sigma, \{q_0\}, \Delta, F)$, where $\delta : Q \times \Sigma \to Q$ is the function with $(q, a, \delta(q, a)) \in \Delta$ for all $(q, a) \in Q \times \Sigma$. We will also use the symbol δ for the natural extension from δ to a function $2^Q \times \Sigma^* \to 2^Q$.

Sometimes it is convenient to regard the transition relation Δ of an NFA as a function into the powerset of Q, so we will sometimes use the symbol δ also for NFA's with the meaning $\delta : Q \times \Sigma \longrightarrow 2^Q$, $(q, a) \mapsto \{p \in Q \,|\, (q, a, p) \in \Delta\}$. This δ can also be extended to the domain $2^Q \times \Sigma^*$ as above.

In some cases we will, given $w \in \Sigma^*$, write δ_w for the mapping $Q \mapsto Q$ (in case of DFA's) or $Q \mapsto 2^Q$ (in case of NFA's) with $\delta_w(q) = \delta(q, w)$.

On the other hand, it might be convenient to fix a certain state $q \in Q$ and write $\delta_q(w)$ for $\delta(q, w)$.

For an NFA $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$, the language recognized by $\mathcal{A}$ is $L(\mathcal{A}) := \{w \in \Sigma^* \,|\, \delta(I, w) \cap F \neq \emptyset.\}$ For a DFA $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, the language recognized by $\mathcal{A}$ is $\delta_{q_0}^{-1}(F) = \{w \in \Sigma^* \,|\, \delta(q_0, w) \in F\}$.

We call two automata *equivalent* iff they accept the same language.

We call two NFA's $\mathcal{A}_1 = (\Sigma, Q_1, I_1, \Delta_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, I_2, \Delta_2, F_2)$ *isomorphic* iff there is an *automaton isomorphism*, i.e. a bijection $\phi : Q_1 \longrightarrow Q_2$, such that $\forall p, q \in Q_1 \, \forall a \in \Sigma : (p, a, q) \in \Delta_1 \Leftrightarrow (\phi(p), a, \phi(q)) \in \Delta_2$, $\phi(F_1) = \phi(F_2)$ and $\phi(I_1) = \phi(I_2)$.

1.3.3 Semigroups and Monoids

A *semigroup* is a pair $S = (A, \cdot)$, where A is a set and $\cdot$ is a binary associative operation on A. By abuse of notation, we will identify S with its carrier A if no

ambiguity arises. Moreover, we will usually omit $\cdot$ when denoting elements of a semigroup.

A *monoid* is a semigroup M with a neutral element 1 (that is, we have $\forall m \in M : 1m = m = m1$).

A language $L \subseteq \Sigma^*$ is said to be recognized by a finite monoid M, if there exists a subset $P \subseteq M$ and a monoid homomorphism $h : \Sigma^* \longrightarrow M$ (that is a mapping satisfying $\forall u, v \in \Sigma^* : h(uv) = h(u)h(v)$), such that $h^{-1}(P) = L$.

The following definition describes how to obtain for any regular language an accepting monoid.

Definition 1.1 Let L be a language. The *syntactic congruence* $\cong_L$ induced by L is given by $u \cong_L v$ iff $\forall x, y \in \Sigma^* : xuy \in L \Leftrightarrow xvy \in L$.

The *syntactic monoid* and the *syntactic semigroup* of L are given by $M(L) = \Sigma^*/\cong_L$ and $S(L) = \Sigma^+/\cong_L$, together with the multiplication $[u] \cdot [v] = [uv]$, where for $u \in \Sigma^*$, $[u]$ denotes the equivalence class of u wrt. $\cong_L$.

In this context we call $[a]$ a *generator* for all $a \in \Sigma$.

The proof that the syntactic congruence is indeed a congruence (and that therefore the multiplication is well defined) is very simple, and thus the quotient $M(L)$ is a monoid and $S(L)$ is a semigroup. We remark that $M(L)$ is finite iff L is regular. The syntactic congruence is a refinement of the Nerode congruence used for the minimization of DFA's. (Two words u, v are Nerode congruent modulo $L \subseteq \Sigma^*$ if $uy \in L \Leftrightarrow vy \in L$ holds for all $y \in \Sigma^*$.)

Usually we will omit the index L from $\cong$ if the language is clear from the context.

Note that every element of the monoid is a product of generators, but the set of generators need not be minimal with this property.

1.3.4 Regular Expressions

Regular expressions are terms over a one sorted signature with the binary operation symbols $\cup, \cdot$ and the unary operation symbol * and the constants $\emptyset$ and a for all $a \in \Sigma$. We will adopt the usual conventions concerning the omission of brackets and $\cdot$. For convenience, we will not distinguish between the operation symbols and the corresponding operations.

The language $L(r)$ represented by a regular expression r is defined inductively over the structure of regular expressions, that is:

- $L(\emptyset) := \emptyset$, $L(a) := \{a\}$
- For regular expressions r, s, we define $L((r \cup s)) := L(r) \cup L(s)$, $L((r \cdot s)) := L(r) \cdot L(s)$, $L(r^*) := (L(r))^*$

From now on, we will allow the symbol Σ for the regular expression $a_1 \cup \ldots \cup a_n$, where $\Sigma = \{a_1, \ldots, a_n\}$, and we write ε for the regular expression $\emptyset^*$.

For example $L((a \cup b)^*ba \cup b^*)$ is the language consisting of all words over the alphabet $\{a, b\}$ that have the suffix ba or consist entirely of b's.

Generalized regular expressions are like regular expressions, but with the additional operation symbols $\cap, \setminus, \sim$ for intersection, set difference and complement with respect to Σ^*, respectively.

Part I

How to Use AMoRE

Chapter 2

User Manual

2.1 Operating the System

This section is a summary of the commands by which the user can run the system. Additional facilities, not described here, are explained on the screen while the system is operating.

The Figures 2.1, 2.2, 2.3 show three menus. The user may enter a command when one of them is displayed. Each of these menus consists of five windows that have a headline surrounded by "="'s. To activate a command of such a window, the user has to type the letter which is capital in its headline, followed by the number of the desired command. (This letter can be typed as capital or lower case.)

The line above the system prompt usually shows a message from the system, e.g. an error message or what to type in next.

The rightmost window in all menus is called "Languages". It displays a list (initially empty) of languages defined or loaded by the user, with an arrow pointing to the so-called "current language". This language is the one which is currently handled by the system and which the commands will operate upon. The current language can be changed by the command "L" followed by the new number. In the following, we will call this list the "language list".

We will now give a short description of the commands. We remark that it is not necessary to change to a particular menu before giving a certain command, but each command is accessible from any of the three menus.

2.1.1 General Commands

The most important command to remember is <CONTROL>-C. With this command you activate a certain interrupt that discontinues the current calculation. You can use it safely without loss of data to exit either a long computation or the (automaton-/expression-)editor.

```
       L A N G U A G E   R E P R E S E N T A T I O N S

====== Expressions =======  ======= Automata =========   ==== Languages ====
|                        |  |                        |   |                  |
| 1. regular expression  |  | 1. det. automaton      |   |->1.   qwerty     |
| 2. generalized expr.   |  | 2. ndet. automaton     |   |  2.   extree     |
| 3. starfree expression |  | 3. epsilon automaton   |   |  3.              |
|                        |  |                        |   |  4.              |
|________________________|  | 4. min. det. automaton |   |  5.              |
=== syntactic Monoid =====  | 5. red. ndet. automaton|   |  6.              |
|                        |  |________________________|   |  7.              |
| 1. elements,relations  |                               |  8.              |
| 2. Green's relations   |                               |  9.              |
| 3. multiplication table|                               |__________________|
|                        |
============================ General commands ==============================
|                                                                           |
| 1. load     3. modify      5. print      7. test menu    9. save         |
| 2. create   4. delete      6. display    8. assign menu  0. exit AMORE   |
|___________________________________________________________________________|
Enter uppercase letter and number
INPUT >>>
```

Figure 2.1: The main menu. It is entered first and opens the computation of different language representations

G1 : load Read a language from the disk. A special menu named "CONTENTS" is entered. If there are more languages on the disk than can be shown, the user can trace the file lists downwards by hitting the space bar. To load a certain language, simply type in its name. If you type in a name that does not exist, AMoRE will return to the menu with the error message "cannot open file; quit."

If you type in a name that already is among the nine ones of the RAM-list displayed in the "Languages"-menu, it will be rejected. You will have to delete this language from the language list before you can load the equally named one from disk.

After a language has been loaded successfully, it receives the lowest available number in the language list and becomes the current language.

G2 : create Define a new language. A special version of the "representation" menu is entered, which offers you only the different language representation suitable for input. To choose one of them, simply type the letter followed by the number. In case of a wrong input, AMoRE immediately returns to the ordinary representation, displaying an error message.

After the choice how to define a language, AMoRE will ask you for the name of the language, for a "reference", for the size of the alphabet and (if the

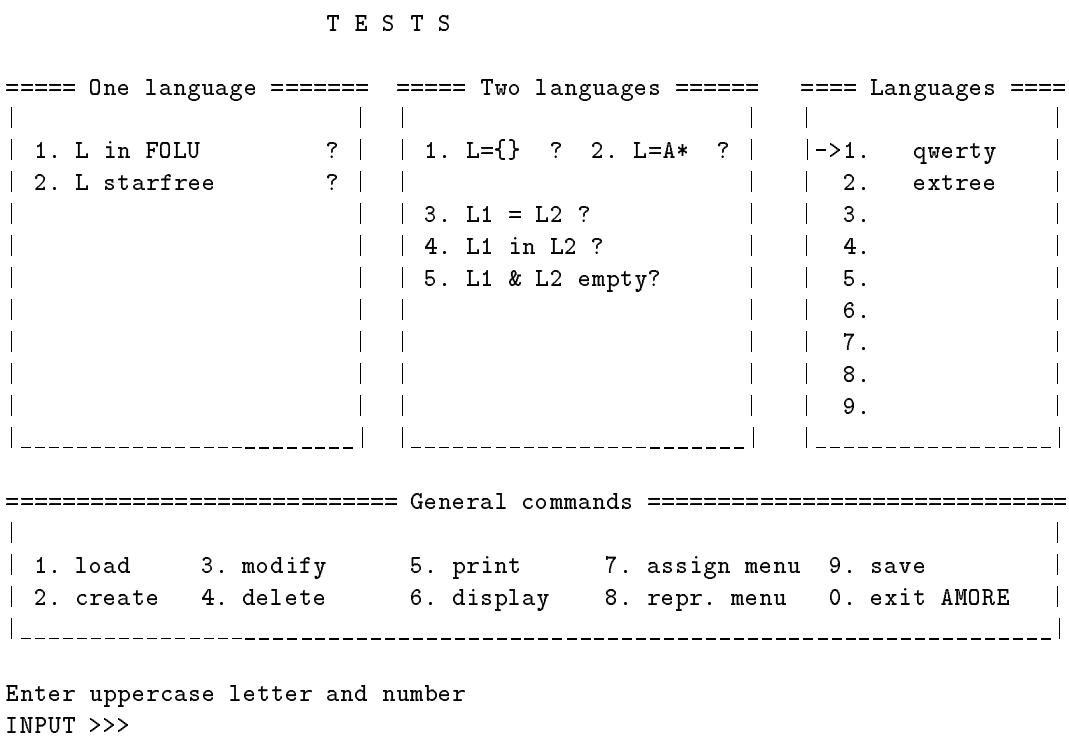


Figure 2.2: The test menu. It is entered from the representation menu via the command "G7". It offers different tests. Note that more tests on one language are offered if the test for starfreeness for the current language has been answered positively.

chosen way of input is automaton) for the number of states. The name (max. 8 characters) will be both the one that is displayed in the language list and the one used for the save file. The reference is any arbitrary string of max. 60 characters. It will be displayed in the "CONTENTS" menu, so it can be used for some extra information.

G3 : modify Define a new language by modifying one of the known languages (from the language list).

G4 : delete Delete a language from the language list. If the language has not been saved after the most recent computation on it, the system will ask the user for permission before actually deleting the language.

After a language is deleted, the language list is renumbered, so that the numbers used will be consecutive and starting at one.

G5 : print Print a language. The system will not actually print but rather create a printable file. See section 2.2 for more information.

G6 : display Generate a graphical display of an automaton. The system asks for the automata type that you wish to be displayed. See section 2.3

```
                A S S I G N M E N T S

==== Unary operation =====  === Binary operation =====   ==== Languages ====
|                        |  |                        |   |                 |
| 1. L2 := L1*           |  | 1. L3 := L1 U L2       |   |->1.   qwerty    |
| 2. L2 := L1+           |  | 2. L3 := L1 & L2       |   |  2.   extree    |
| 3. L2 := ~L1           |  | 3. L3 := L1 . L2       |   |  3.             |
| 4. L2 := rev(L1)       |  | 4. L3 := L1 - L2       |   |  4.             |
| 5. L2 := min(L1)       |  | 5. L3 := L1 W L2       |   |  5.             |
| 6. L2 := max(L1)       |  | 6. L3 := L1 LQ L2      |   |  6.             |
| 7. L2 := pref(L1)      |  | 7. L3 := L1 RQ L2      |   |  7.             |
| 8. L2 := suff(L1)      |  |                        |   |  8.             |
|                        |  |                        |   |  9.             |
|________________________|  |________________________|   |_________________|

============================ General commands ==============================
|                                                                           |
| 1. load     3. modify      5. print      7. repr. menu   9. save          |
| 2. create   4. delete      6. display    8. test menu    0. exit AMORE    |
|___________________________________________________________________________|

Enter uppercase letter and number
INPUT >>>
```

Figure 2.3: The assignment menu. It is entered from the representation menu via "G8" and offers several commands to compute a new language out of one or two old ones.

An extra (X-Windows –) window is opened that shows the automaton. AMoRE cannot continue before this window has been closed.

This feature does not work on all kinds of terminals.

G7,G8 : Switch to the representation resp. test menu.

G9 : save Write a language to disk. AMoRE writes all known information of a specified language to a file named with the name of the language followed by the extension .amr. If the language is re-loaded later, all the language representations and test results will be available without an extra computation.

If an older version of this language (with less information) or an equally named, but different language exists, it will be deleted!

If you wish to avoid this, you may proceed as follows: first, modify the language by typing G3. When asked for, enter a name that does not occur on the disk. Instead of actually modifying the language, just leave it unchanged. Then you can save this (freshly created) language under its new name. But note that all the information about the language except for the

representation you have chosen for the "modification" will be lost and will have to be computed again.

G0: exit AMORE AMoRE is exited after prompting for confirmation. Note that remaining unsaved languages do not cause an additional question. They will get lost.

2.1.2 Viewing and Editing Expressions

E1,E2 : (generalized) regular expressions

View the (generalized) regular expression of the current language. If you ask for a generalized regular expression and the language has been defined any other way, the system will show you a *standard* regular expression anyway. The expression will be computed automatically if necessary. If the input has been a regular expression or some former computation yielded a regular expression, then this one is shown without further computation.

You enter an editor as shown in Figure 2.4. This editor is used both for viewing a regular expression and for editing one. The latter case is activated, if "E1" or "E2" was typed after the "G2" or "G3" command for "create" or "modify", respectively. In these cases, the editor looks different from the normal (viewing) mode.

As you can see in Figure 2.4, the top half of the screen has the headline "DEFINITIONS". These definitions are abbreviations that the user may use to edit more complex expressions. One abbreviation generated by the system is "A" for the whole alphabet. For further abbreviations the following capital letters will be used.

The syntax of regular expressions is rather convenient. Expressions are typed in infix notation with the usual conventions concerning priority of operands and omitting of brackets. The operators are in detail (in descending priority):

`*` for the Kleene-star *. (unary postfix operator)

`+` for the Kleene-plus $^+$. (unary postfix operator)

$\sim$ for the complement wrt. A^* (unary prefix operator, only in case of generalized expressions).

`.` for the concatenation $\cdot$ (may be omitted).

`&` for the intersection $\cap$ (only in case of generalized expressions).

`-` for the set difference $\setminus$ (only in case of generalized expressions).

`U` (a capital u) for union $\cup$.

Spaces inside regular expressions are ignored, so they can be used to improve the readability.

For example,

```
~ a*b & (aab)* U bb
```

is as well as

```
(((~(a*)).b) & (aab)*) U (b.b)
```

a correct input for the regular expression $(((\Sigma^* \setminus (a*)) \cdot b) \cap ((a \cdot a \cdot b)^*)) \cup (b \cdot b)$. [1]

Note that AMoRE will remember the way you have defined a certain language and therefore keep the regular expression you used as input exactly as you typed it in. So if you ask for the display of an expression, all the abbreviations will still be present, as well as omitted brackets, and so on.

The expression editor In the two bottom lines, the editor offers you some commands that enable you to scroll through longer regular expressions and move the cursor to a particular position, namely the commands "h", "j", "k" and "l". Their meaning is the same as in the UNIX-editor "vi" (namely "left", "down", "up" and "right'', respectively).

- The command "d" changes into the definition editor. The cursor will change into the top half of the screen. You may then define a new abbreviation. The definition editor works very similar to the expression editor. Again, the displayed commands will differ. In particular, press "q" if you finished editing the abbreviation and wish to re-enter the expression editor. Note that the abbreviations may contain abbreviations themselves, but only previously defined ones. The abbreviations will not make AMoRE's computations any faster or slower, so whether you use them is only a question of convenience.

- The command "i" causes a change to the insert mode. You may then insert text at the current cursor position or delete text left from the current cursor position. In the insert mode, the commands offered in the bottom line are different, also depending on whether you are editing a *generalized* expression or not. In particular, you have to press <ESC> (and not "q") to leave the insert mode.

- The command "q" exits the (definition- or expression-) editor. If you were editing an abbreviation or an expression, its syntax will be checked. If it is not correct, the system will ask you to correct your mistake. If it is, your new abbreviation (or language, respectively), is defined.

[1] Invisibly for the user, it will be converted into the postfix notation `a*~ b.aa.b.*&bb.U`. We observe that AMoRE chooses left-associativity, so `aaa` is the same as `(a.a).a`, but this is of no concern for the user.

```
      G E N E R A L I Z E D   R E G U L A R   E X P R E S S I O N

Size of alphabet: 2

====================== D E F I N I T I O N S ======================
  A: (a U b)
  B: (a*b*)
  C: B U aaa
  D: A U C



====================== E X P R E S S I O N ======================

B+ U (AC*) U aabb*



h: left, j: down, k: up, l: right  q: quit
i: insert mode  s: scroll definitions  d: define
```

Figure 2.4: The editor for regular expressions. The bottom line may offer different commands, depending on whether you wish to view, to create or to modify a language

2.1.3 E3 : Starfree Expression

A starfree expression is a generalized regular expression that does not use any stars. If you wish to define a language by some kind of regular expression, you have to choose either "E1" or "E2", depending on whether you need the non-standard operator for intersection, complement and set difference. The menu item "E3 : starfree expression" is reserved only for output.

If asked for a starfree expression, AMoRE first tests whether the language is starfree i. e. whether such an expression exists. If the answer is negative, a corresponding error message will be displayed. If the answer is positive, AMoRE computes a starfree expression for a language. Both the test and the computation run on the syntactic monoid and its D-class decomposition, which will be computed first, if necessary. The corresponding algorithms are described in [Pe90].

Because of the implemented algorithm, the computed starfree expression will always have a certain structure. It is always a union of (possibly several) subexpressions, each of which describes an equivalence class of the syntactic congruence.

These subexpressions are always of the form $(r\Sigma^* \cap \Sigma^* s)\backslash\Sigma^* t\Sigma^*$, where r, s, t are expressions themselves, namely

- r is a union of several expressions of the form $L(u)a$, $(u \in \Sigma^*,\ a \in \Sigma)$,
- s is a union of several expressions of the form $aL(u)$, $(u \in \Sigma^*,\ a \in \Sigma)$,
- t is a union of several expressions of the form $aL(u)b$, $(u \in \Sigma^*,\ a, b \in \Sigma)$,

where the abbreviation $L(u)$ is used for the starfree expression that describes the equivalence class with the minimal[2] representative u.

When the starfree expression has been computed, you are asked if you wish to see the expression expanded or not. If you choose "no", then the subexpressions for the equivalence classes are displayed separately.

If you choose "yes" for the expansion of the starfree expression, then you see the complete expression without any abbreviations.

The starfree expressions computed by AMoRE are mostly much too long to be readable. Especially the expanded version is even in very trivial examples not recommendable as an output. Again, it would be a great benefit if someone discovered some reasonable way to minimize starfree expressions.

2.1.4 A1, A2, A3 : DFA, NFA, ε-NFA

View the transition table of the desired automaton type. It will be computed first, if necessary.

You enter an editor as shown in Figure 2.5. This editor is used both for viewing an automaton and for editing one. The latter case is activated, if you typed "A1", "A2" or "A3" after the "G2" or "G3" command for "create" or "modify", respectively.

The editor is almost the same for NFA's and for DFA's. In case of DFA's the headline says "DFA Transitiontable".

The leftmost column lists all states. Initial states are indicated by a star and final states have a capital "Q". In Figure 2.5 the state 1 is the only initial state and state 0 is the only final one. The table entry in the line i and the column for b is the list of all states to which there exists a transition from state q_i with letter b.

The cursor movement commands are again the same as the ones in the UNIX-editor `vi`. The command "g" is only useful for viewing/editing large automata: "g" followed by a state number scrolls the screen so that its corresponding row is displayed (in the topmost line, if possible).

To make a state initial (final, resp.) move the cursor into the corresponding row and then type "i" ("f" resp.).

[2] *minimal* wrt. the ordering defined in Section 1.3

```
                N F A   T R A N S I T I O N T A B L E

  Number of states: 3          Number of letters: 2

                  a            b

  Q0              q1           q1,q2

 *q1              q0           q2

  q2              q0,q1,*      q0



h: left  j: down  k: up  l: right  g: goto                  q: quit
f: switch final ('Q')    i: switch initial ('*')  0-9: new state list
s: show all states
```

Figure 2.5: The Automaton Editor.

Editing DFA's To enter a new transition, say from state 3 to state 6 via the letter b, move into the row 3 and the column corresponding to b and type "6". Then the entry "q6" will be generated in this table position. The system will reject state numbers that are too high. Type "q" if you finished editing the DFA. The system will reject your automaton if there are transitions missing or if there is not exactly one initial state.

Editing NFA's In the transition table of an NFA, there might be several states in one table position. If not all of them fit in the table, there is a star displayed in order to indicate that there are states left (see Figure 2.5, transition list for state 2 and letter a). To see all of them, move the cursor into this table position and type "s".

If the cursor is at one table position and you type a digit, the transition list for this position is completely deleted and replaced by the single state number you type in. If you want to add a state to a transition list, you have to type "a" when the cursor is at the corresponding table position. Then the system will ask you to type in that state in the bottom line.

To delete a state from a transition list, move the cursor to the corresponding table position and then type "d".

Type "q" to finish editing the NFA. The system will reject your automaton if no state is marked as initial.

2.1.5 M1 : Syntactic Monoid

Viewing the syntactic monoid of a language. You enter a display as shown in

```
                    S Y N T A C T I C   M O N O I D

 Number of elements: 7       On page: 1. - 7. Number of states: 3

          0        1        2

 *1       0        1        2

 *a       0        2        2

  b       1        0        2

  ab      1        2        2

  ba      2        0        2

 *0       2        2        2

 *bab     2        1        2



h: left  j: down  k: up  l: right  g: goto                    q: quit
r: show defining relations
'*' marks idempotent transformations
```

Figure 2.6: The Monoid Display

Figure 2.6. Before this, the system will compute the syntactic monoid from the minimal DFA, if necessary.

The leftmost column lists the least representatives of the equivalence classes that build the syntactic monoid. (See Chapter 12 and also subsection 1.3.3) for a detailed explanation.) The star marks the idempotent elements. The neutral element of the monoid — the equivalence class of the empty word — is always displayed in the topmost row and is denoted by `1`. If the monoid contains a zero (like the example monoid of Figure 2.6) it is denoted by `0`.

The remaining columns show for each monoid element the state transformation induced on the MDFA.

Again, the commands "h", "j", "k", "l" and "g" can be used similarly to the automaton editor. With the command "r" you can ask the system to list a

(minimal) set of *defining relations*. See next subsection.

2.1.6 Defining Relations

You enter the display for the defining relations by typing "r" while you are in the display for the syntactic monoid, which can be entered via "M1" from the main menu. With "relation" we mean (in this context) any pair of two words that are syntactically congruent (i.e. they are in the same equivalence class of the syntactic congruence, see Chapter 12).

A set of such pairs is called a *defining set of relations* if the syntactic congruence is the smallest congruence relation on Σ^* that includes all these pairs. The pairs computed by the system have the additional property that the right hand side is always "smaller" than the left hand side (wrt. the ordering defined in 1.3.1).

These pairs form a Semi-Thue-System which is minimal with the property to define the syntactic congruence and to be Church-Rosser.

This means that if you start with two syntactically congruent words and repeatedly replace one of their infixes that is the left-hand side of such a pair with its corresponding right-hand side, then you will finally finish with one and the same word, which will be the least representative of its equivalence class; and no proper subset of this set of pairs has this convenient property.

For the example monoid of Figure 2.6, these relations are

```
aba   = 0
aa    = a
bb    = 1.
```

Press any key to return to the monoid display.

2.1.7 M2 : Green's Relations

You enter a display for the equivalence classes wrt. Green's relations. See [Pi86] and Chapter 13 for the definition of these relations. Figures 2.7 and 2.8 show AMoRE's display for the example monoid of Figure 2.6.

Each of the large boxes shows one D-class. (There are three in Figure 2.7.) Each D-class-box is divided in rows and columns, corresponding to $\mathcal{R}$-classes or L-classes, respectively. The H-classes correspond to the small boxes. (In the example, there are six H-classes, one of size 2 and five of size 1.) The monoid elements itself are given, as usual, by there "smallest" representatives (wrt. the ordering given in Chapter 1.3). The idempotents are indicated by stars.

On the left one finds, for each $\mathcal{R}$-class, the *kernel* of the elements of that $\mathcal{R}$-class. That is the partition of the state set of the minimal automaton such that all states from one set of this partition are mapped onto one and the same state

```
                    D - C L A S S   S T R U C T U R E

  D-Classes: 1. - 3. of 3


          0 1 2
         --------
0/ 1/ 2 |*1     |
         | b    |
         --------

          0 2    1 2
         ---------------
0/ 1 2  |*a     | ab    |
         ---------------
0 2/ 1  | ba    |*bab   |
         ---------------

          2
         --------
0 1 2    |*0     |
         --------


h:left, j:down, k:up, l:right,          q:quit
n,p: next,previous dclass(es),          s: show structure
'*' marks idempotent elements
```

Figure 2.7: The D-Class Decomposition of the Example Monoid

via all of the transformation corresponding [3] to the elements of the $\mathcal{R}$-classes of that row.

You can use the commands "h", "j", "k", "l", "n" and "p" to scroll through the complete decomposition. Figure 2.8 shows the summary of the D-class decomposition that the system gives when you type "s" in the D-class structure display.

2.1.8 M3 : Multiplication Table

You enter a display that shows the multiplication table of the syntactic monoid. Figure 2.9 shows the multiplication table of the monoid from Figure 2.6. The

[3] in the manner of Lemma 12.1 in Chapter 12

```
    G L O B A L  S T R U C T U R E  O F  G R E E N ' S  R E L A T I O N S

                              R-classes  L-classes  Elements per H-class

1  reg.   D-class   rank 3  :      1          1             2
1  reg.   D-class   rank 2  :      2          2             1
1  reg.   D-class   rank 1  :      1          1             1

          End global structure: press any key to continue
```

Figure 2.8: Summary of the D-Class Structure

```
                M U L T I P L I C A T I O N   T A B L E

Number of elements: 7       On page: 1. - 7.

                  1        2        3        4        5        6        7

*1       1        1        2        3        4        5        6        7

*a       2        2        2        4        4        6        6        6

 b       3        3        5        1        7        2        6        4

 ab      4        4        6        2        6        2        6        4

 ba      5        5        5        7        7        6        6        6

*0       6        6        6        6        6        6        6        6

*bab     7        7        6        5        6        5        6        7

h: left  j: down  k: up  l: right  g: goto                 q: quit

'*' marks idempotent transformations
```

Figure 2.9: The Multiplication Table of the Example Monoid

leftmost column contains, again, least representatives. The second column shows a number that is given to every monoid element as an abbreviation. The remaining part of the table shows in the i-th row and j-th column the result of the multiplication of the the i-th and j-th element.

For example, one can see from Figure 2.9 that $[b] \cdot [ba] = [a]$, because $[b]$ has got number 3 in the monoid, $[ba]$ has got number 5 and the entry in row 3 and column 5 is 2, the number of $[a]$. (For the notation see 1.3.) The same information could have been drawn from the set of defining relations given in 2.1.6 , since $bba \approx 1a = a$ due to the third relation.

2.2 Printing

The command for printing is "G5". Afterwards you have to choose a language representation (e.g. type "E1" for regular expression).

But instead of actually printing, AMoRE will create a printable file in the directory `printdir`, one new file for each representation you choose. The name of the file is composed of the name of the language and an extension, which depends on the representation. The Table 2.1 gives a summary of the possible extensions.

representation	command	filename extension
Expressions		
regular expression	E1	.rex
generalized reg. expression	E2	.grx
starfree expression	E3	.sfx
Automata		
deterministic aut.	A1	.dfa
nondeterministic aut.	A2	.nfa
epsilon aut.	A3	.efa
minimal det. aut.	A4	.mfa
reduced nondet. aut.	A5	.mna
Syntactic Monoid		
elements, relations	M1	.mon and .rel
Green's relations	M2	.dcl
multiplication table	M3	.mul

Table 2.1: Filename Extension for the Print Files

The Figures 2.10 and 2.11 show the two pages of a print file generated by the "G5" command applied to the NFA for a language named "large1". The print files for expressions are very similar to the displays of the editor.

```
AMORE  Name: large1  -->
                              Page 1    PID (1,1)

    N F A   T R A N S I T I O N T A B L E
                  a              b              c
   *q0            q0,q1,q2,*     q3,q4,q5,*     q7
   *q1            q2             q3             q6
    q2                                          q0,q1,q3,*
    q3
    q4                           q7
    q5                           q0,q1,q2,*
    Q6            q6                            q1
    Q7                                          q7
```

Figure 2.10: An Example Print File, First Page

```
 AMORE  Name: large1  -->
                               Page 2    PID (2,1)

     L I S T   O F   * - T R A N S I T I O N S
     (q0,a): q0 q1 q2 q3 q7

     (q0,b): q3 q4 q5 q6

     (q2,c): q0 q1 q3 q5 q7

     (q5,b): q0 q1 q2 q3 q4 q5 q6 q7
```

Figure 2.11: An Example Print File, Second Page

As you can see in Figure 2.10, the transition lists in this table show up to three transitions per state and letter explicitly (in contrast to the NFA-editor). The transition lists that are longer are listed on an extra page like in Figure 2.11. The maximal length of one line is 80 characters, the maximal number of lines per page is 66.

In the section 5.2 the interested programmer finds some hints how to adapt these default values to his (and his printer's) needs.

If you intend to print out the graphical display you have to use the print command of the display program AMD, see next section.

2.3 Displaying Transition Graphs

The graph display algorithm implemented in AMoRE is based on the approach of Sugiyama et al. [STT81, RDM*86]: A breadth first search of the transition graph is carried out, starting with the initial state. For a back edge or a cross edge from some level of the breadth first tree to a non-neighbour level, auxiliary points are introduced on each level which is crossed by the edge. Subsequently the states and the auxiliary points within the levels of the breadth first tree are permuted in order to minimize edge crossings. In the displayed graph, the levels of the tree are arranged vertically, successively from left to right.

The output is presented in essentially two versions, called "Dir" and "Nodir" in the final display, depending on whether the breadth first tree is built up by referring to the directed edges as given by the automaton or whether these edges are assumed symmetric (i.e., non-directed). It depends on the specific example which strategy yields a "better" graph display; usually, the directed version leads to less states on the individual levels and a larger number of levels, and the converse holds for the undirected version. In a second choice between two options "Sym" and "Nosym", the user can influence the location of the states on the levels (in comparison with the states on the respective succeeding levels); the first option tries to establish better symmetry by minimizing the length difference of the corresponding edges. This choice has less effect on the actual output than the choice between the options "Dir" and "Nodir". An example of the graph display is given in Figure 2.12. (It represents the minimal DFA for the language given by the regular expression $(((aa)^*ab)^+((aa)^*b)^+((aa)^*b)^+((aa)^*b)^+)^*$ suggested in [Th81] as a candidate language for generalized star-height 2, and recently shown to be of generalized star-height 1 by M. Robson.) With the command "G6" you create a display-file in the directory `displaydir`. The display program AMD reads this file and shows it as a graphic on the screen. The name of the file is composed of the name of the language and a file extension that depends on the automaton type as shown in Table 2.2.

With AMD, you may also print, but similarly to AMoRE, AMD just creates a printable file. It is written into the directory `displaydir`. AMD produces HPGL-Code. The name of the file is yielded by appending an extension to the automaton name. This extension depends on the chosen display option as shown in Table 2.3.

Since the present version of the display program is experimental, we do not give further details on the implementation.

	filename extension
deterministic aut.	.d
nondeterministic aut.	.n
epsilon aut.	.en
minimal det. aut.	.md
reduced nondet. aut.	.mn

Table 2.2: Filename Extensions for AMD Display Files

	filename extension
directed & symmetric	.ds.hp
directed & assymmetric	.da.hp
undirected & symmetric	.us.hp
undirected & asymmetric	.ua.hp

Table 2.3: Filename Extensions for AMD Print Files

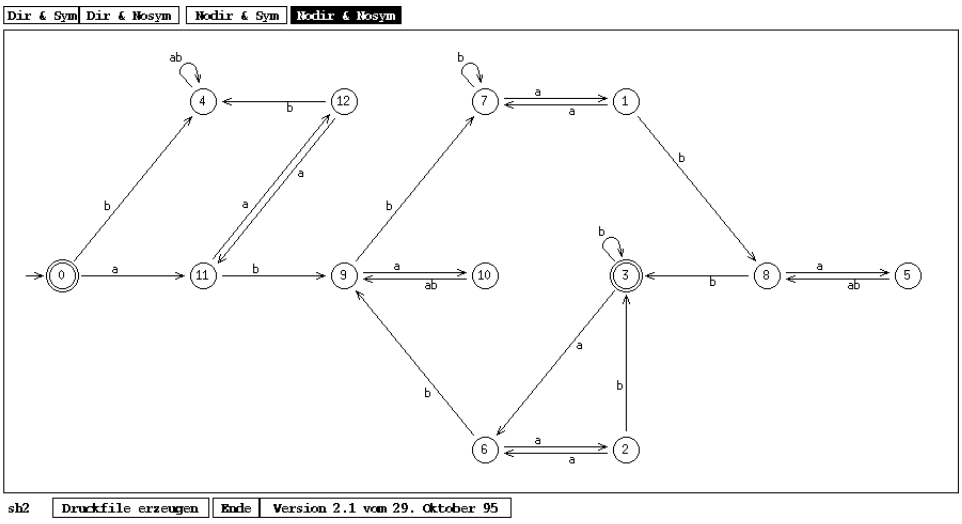


Figure 2.12: A sample graph display

Part II

General Remarks on Implementation

Chapter 3

The Structure of AMoRE

3.1 Basic Data Structures

This section describes the data structures for DFA's, (ε-) NFA's, (generalized) regular expressions, monoids and regular languages. These structures are included in `types.h`.

3.1.1 Elements vs. Indices

In practical implementations of finite sets (like the state set of an automaton or the alphabet Σ), we consider these sets as subsets of $\mathbb{N}$. Thus in the program AMoRE, the integer `1` may represent either the first letter of the alphabet, i.e. a, or one state of an automaton or one element of a monoid and so on.

This might lead to some troubles and misunderstandings in the terminology. If we talk about one "state" in general, we will mean an element of some arbitrary finite set that happens to be the state set of an automaton. If we use the word "state" in the context of AMoRE-code, we will mean a positive integer less than or equal to the maximal state number of the current automaton. To avoid misunderstandings, we will in the latter case sometimes refer to "the state with index `i`", which is actually nothing else but the integer `i`.

We thus follow the idea that the elements of a set of states Q are indexed, that is $Q = \{p_0, \ldots, p_n\}$ and $p_i \neq p_j$ for $i \neq j$. Note that the state with index 0, identified with the state $p_0 \in Q$, may be different from the initial state q_0 of the considered automaton.

The same problem occurs when talking about monoids. A "monoid element" in the context of AMoRE-code is a positive integer, whereas in mathematical sense, the elements of the monoids we consider will be equivalence classes of Σ^*.

A consequence of this index convention is the total ordering introduced in this way. Thus it makes sense to talk about the "next" or the "following" state, letter or monoid element.

Moreover, this ordering on Σ induces in a very natural way an ordering on Σ^*, namely the one defined in Section 1.3. That is for $a_1 \ldots a_n, b_1 \ldots b_m \in \Sigma^*$, we have $a_1 \ldots a_n < b_1 \ldots b_m$ iff $n < m$ or ($n = m$ and $\exists 1 \leq i < n : a_1 \ldots a_i = b_1 \ldots b_i \wedge a_{i+1} < b_{i+1}$).

This small example shows again a difficulty: In the above definition, a_i is the i-th letter in a particular word, which does not mean "the letter with index `i`" — the `i`-th letter of the alphabet, which is denoted by the integer `i`.

Another remark about the alphabet: in the AMoRE-program and its user surface, the underlying alphabet of a language (which can be different for different languages) is always abbreviated by A instead of the notation Σ we often use in this report.

3.1.2 DFA

The C-structure `dfauto` is used for representing DFA's. A brief discussion of the structure elements will follow there.

- The variable `qno` holds the *maximal state number* of the automaton. The states of the automaton are numbered from `0` to `qno`.
- The variable `init` holds the state number of the *initial state* of the automaton.
- The *size of the alphabet* is stored in the variable `sno`. The letters of the alphabet are numbered from `1` to `sno`.
- The array `final` holds for each state a boolean value that indicates if the state is a *final state*, i.e. `final[i] == true` $\Leftrightarrow$ the state with index `i` is a final state.
- The field `delta` holds the *transition table* of the automaton. For each pair `(l,q)` the successor of the state `q` with the letter `l` is stored in the variable `delta[l][q]`.
- If the boolean variable `minimal` holds the value `true`, then the DFA is known to be *minimal*. After applying the minimization procedure for deterministic automata, `minimal` is set to `true`. The variable is used to avoid applying the minimization procedure more than once.

3.1.3 (ε-)NFA

The C-structure `nfauto` is used for representing (ε-)NFA's. The type of an ε-NFA is only used as an input for regular languages, none of the implemented algorithms has an ε-NFA as the output.

- The variable `qno` holds the *maximal state number* of the automaton. The states of the automaton are numbered from `0` to `qno`.

- The *size of the alphabet* is stored in the variable `sno`. The letters of the alphabet are numbered from `0` to `sno`, where the letter `0` is reserved for the empty word ε.

- The array `infin` holds for each state the information, whether the state is a *final state or an initial state*.

  The C-macros `isfinal`, `setfinalT`, `setfinalF`, `isinit`, `setinit` and `rminit`, which are defined in file `ext.h`, allow access to this information. The macro `isinit(infin[q])` returns `true` iff the state with number `q` is marked as an initial state, `isfinal(infin[q])` returns `true` iff the state with number `q` is marked as a final state. `setinit(infin[q])` marks the state `q` as an initial state, `setfinalT(infin[q])` marks the state `q` as a final state. `rminit(infin[q])` removes the "initial mark" from the state `q`, `setfinalF(infin[q])` removes the "final mark" from the state `q`.

- The field `delta` holds the *transition table* of the automaton. For each triple `(l,f,t)` a single bit in `delta` indicates if there is a transition from the state `f` to the state `t` with the letter `l`.

  The C-macros `connect`, `disconnect` and `testcon` allow access to the transition table. `connect(d,l,t,f)` adds the transition from the state `f` to the state `t` with the letter `l` to the transition table `d` of an NFA, `disconnect(d,l,t,f)` removes this transition from the table.

  `testcon(d,l,t,f)` returns `true` iff there is a transition from `f` to `t` with the letter `l` in the table.

- The boolean variable `minimal` holds the value `true` only if the *NFA is reduced*. After applying the reduction procedure for nondeterministic automata, `minimal` is set to `true`.

- The boolean variable `iseps` holds the value `true` if there are $\varepsilon-$*transitions* in the transition table of the NFA, i.e. transitions with the letter `0`.

3.1.4 Regular Expression

The C-structure `rex` is used for (generalized) regular expressions.

- The variable `sno` holds the *size of the alphabet*. The letters of the alphabet are numbered from `0` to `sno`, where the letter `0` is reserved for the empty word ε.

- The *regular expression* itself is stored in the variable `rex`. The expression is stored exactly as it was typed by the user, i.e. in infix notation and with the abbreviations defined by the user.

- The array `abbr` holds the *abbreviations* defined by the user, i.e. the abbreviation with the name `A` is stored in the variable `abbr[0]`, the abbreviation with the name `B` is stored in the variable `abbr[1]` etc. The maximal number of abbreviations is given by the constant `NABBR`, which is defined in the file `cons.h` (currently as 12).

- The *lengths of the abbreviations* are stored in the array `abbl`, i.e. for `i` $\leq$ `NABBR`, `abbl[i]` is the length of the string `abbr[i]`.

- The *number of used abbreviations* is stored in the variable `useda`. We note that `useda` is at least `1` because the abbreviation `A` is predefined by the system.

- The variable `exprex` is used to hold the *postfix notation* of the regular expression. Additionally, possible abbreviations are expanded to regular expression.

- The variable `rexl` holds the *length of the expression stored in* `rex`.

- The variable `erexl` holds the *length of the expression stored in* `exprex`.

- The boolean variable `grex` holds the value `true` if the regular expression is generalized, i.e. the expression contains one of the operators $\sim, \cap, -$.

3.1.5 Monoid and Defining Relations

The data structure for monoids is the C-structure `*monoid`, and it has the following descriptors:

`qno,sno,mno` for the highest state number in the minimal DFA, the number of letters of the underlying alphabet Σ and the number of elements, respectively. The monoid elements are internally indexed from `0` to `mno-1` whereas in the output visible for the user, the element numbers are one higher. The elements of the monoid are numbered wrt. the ordering of their least representatives, so the (internal) element number `0` is always the neutral element, whose least representative is the empty word.

`gno` for the number of generators (see Definition 1.1), so the elements `1,...,gno` are those monoid elements whose least representative is a letter of Σ.

`generator` is an array that is indexed from `1` to `gno`. For every `1`≤`i`≤`gno`, `generator[i]` contains the least letter that is a representative of the monoid element `i`.

Then every transformation induced by some letter of Σ is induced by exactly one letter appearing in this array, and the letters appearing in the array are the least ones with that property. If `generator[letter] == letter`, then the letters from `1` to `letter` induce pairwise different transformations.

The value `generator[0]` is needed only for the output, since `0` represents the "letter" ε, the empty word.

`let2gen` is an array indexed from `1` to `sno`. For `letter`, `let2gen[letter]` holds the monoid element that induces the same transformation as `letter`. The arrays `generator` and `let2gen` are related as follows:
`let2gen[generator[i]] == i` for all `1`≤`i`≤`gno`.

`no2trans` is an array indexed from `0` to `mno-1` of arrays indexed from `0` to `qno`. For any `i`, `no2trans[i]` will be the state transformation of the `i`-th element of the monoid.

`gensucc` is an array `[0..mno-1]` of arrays `[0..gno]` with two purposes:

1. `gensucc[no][0]` holds the number of the predecessor of the monoid element `no`, that is the number of the monoid element induced by the word you obtain when you omit the last letter in the smallest[1] representative of the element with number `no` For the identity, which will always have the number zero, the descriptor `gensucc[0][0]` will always hold zero.
2. For `1`≤`i`≤`gno`, `gensucc[no][i]` will hold the number of the transformation that results by multiplying the element with index `no` with the `i`-th generator.

`lastletter` is an array indexed from `0` to `mno-1` that holds for any monoid element the number of the monoid element induced by the last letter of the smallest word that induces the transformation. The values of `lastletter` are always in `0,...,gno`, where `0` is reserved for the identity monoid element.

`no2length` is an array `[0..mno-1]` that holds for every monoid element the length of the smallest word that induces this state transformation.

Thus if $w \in \Sigma^*$ is the smallest word that induces the state transformation of the monoid element with number `i`, `no` is a number of another monoid element and `gensucc[no][0]==i`, `lastletter[no][i]==j`

[1] wrt. the ordering of 1.3.1

states	letters			
	a	b	c	d
i 0	1	1	2	0
1	2	2	0	1
f 2	0	0	1	2

representative	induced transformation		
	0	1	2
$*1$	0	1	2
a	1	2	0
c	2	0	1

Figure 3.1: Example MDFA and its Syntactic Monoid.
i and f are supposed to mark the initial and final state.

and `generator[j]==`a, then wa will be the least word that induces the state transformation of the element `no`.

`no2length[no]` will hold $|wa|$.

`no2rang` is only defined when the $\mathcal{D}$-class is already computed, and in this case it is an array `[0..mno-1]` that holds for any monoid element the rank of its state transformation (see Chapter 13 for the definition of rank).

`zero` holds the number of the element that represents the zero of the monoid if such an element exists, otherwise it holds `mno`, a non-existent element index.

`mequals` is a boolean that is true iff the syntactic semigroup equals the syntactic monoid. This is true iff there is a word $w \in \Sigma^+$ that induces the identity in the underlying minimal DFA.

`dclassiscomputed` is a boolean that is true iff the $\mathcal{D}$-class has already been computed.

`ds` is a pointer to the $\mathcal{D}$-class, if it has already been computed.

`rno` holds the number of defining relations.

`lside,rside` are two arrays indexed `0...rno-1` keeping the defining relations. Let for example `i`$\leq$`rno-1`. The `i`-th relation has the form $u * a = v$, where $u, v \in \Sigma^*$, $a \in \Sigma$. Then `lside[i]` holds the number of the monoid element for the transformation induced by u (thus `0`$\leq$ `lside[i]`$\leq$ `mno-1`), and `rside` holds the number of the generator induced by a (thus `1`$\leq$ `rside[i]`$\leq$ `gno-1`). v is not stored explicitly because it can be computed quickly.

`word` is an auxiliary storage used in the `dfamon.c` for the computation of a least representative.

Since this data structure is a bit tricky, we give another example. Consider the MDFA over $\Sigma = \{a, b, c, d\}$ given in Figure 3.1 by its transition table, and its syntactic monoid. Obviously, a and b induce the same transformations.

Descriptor	Content	Explanation
`qno`	2	states $0, 1, 2$
`sno`	4	size of alphabet
`mno`	3	size of monoid
`gno`	2	elements 1 and 2 are generators
`generator`	$\begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 3 \end{pmatrix}$	$\{[a], [c]\}$ is the set of generators
`let2gen`	$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 1 & 2 & 0 \end{pmatrix}$	
`no2trans`		see Figure 3.1
`gensucc[i][0]`	0	for `i=0,1,2`, since all transformations are induced by single letters
`gensucc[1]`	$\begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix}$	$\delta_{aa} = \delta_c$, $\delta_{ac} = \delta_\varepsilon$
`gensucc[2]`	$\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$	$\delta_{ca} = \delta_\varepsilon$, $\delta_{cc} = \delta_a$
`lastletter`	$\begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix}$	all transformations are induced by single letters

Table 3.1: Example Data Structure

Table 3.1 shows some descriptors of the C-structure `mono` that AMoRE uses to store this monoid. We admit that the example is a bit boring because there are no state transformations in the monoid which are not induced by single letters. But note especially the usage of the descriptors `let2gen` and `generator` and how they work together with the remaining descriptors.

3.1.6 $\mathcal{D}$-Class Decomposition

The type for the storage of the $\mathcal{D}$-class decomposition of a monoid is `darray`. `darray` is an array of C-structures `*d_class`, each of which holds the information about one $\mathcal{D}$-class D. The descriptors of `*d_class` are in detail:

`org` one element in the upper left $\mathcal{H}$-class.

`hsize` the number of elements in an $\mathcal{H}$-class contained in D.

`hclass` an array indexed `0...hsize-1` containing the element stored in `org` and all the other elements of its $\mathcal{H}$-class.

`rno, lno` the numbers of $\mathcal{R}$-classes and $\mathcal{L}$-classes, respectively.

`rrep, lrep` array indexed `0...rno-1` (`0...lno-1`, resp.) of so-called "prefixes" ("suffixes", resp.) of representatives of each $\mathcal{R}$-class ($\mathcal{L}$-class, resp.)

An element of the $\mathcal{H}$-class which is the intersection of the `i`-th $\mathcal{R}$-class with the `j`-th $\mathcal{L}$-class can be obtained uniquely by `rrep[i] * org * llrep[j]`, where `*` denotes the monoid multiplication, and all of the `hsize` elements of this $\mathcal{H}$-class are: `rrep[i] * hclass[k] * lrep[j]` with `0`$\leq$`k`$\leq$`hsize-1`.

`regular` boolean value, which is true iff the $\mathcal{D}$-class is regular.

`rang` the rank of the $\mathcal{D}$-class.

`maxlen` an upper bound for the number of characters in the display of a least representative of an element in this $\mathcal{D}$-class. (Useful for the screen-display!) If the size of the alphabet is larger than `ALPHSIZE`, so that letters are displayed by strings like `a27, a255` etc. `maxlen` will be $l \cdot m$, where l is the number of letters in least representative and l is the maximal length of the display of one character.

3.1.7 Regular Language

The C-structure `regset` is used for regular languages. In AMoRE a language is represented by one C-structure that contains pointers to all language representations and flags for additional information that have been computed so far and might be of interest later on. If a particular item has not been computed yet, the corresponding pointer holds `NULL`. When a language is saved, all these representations and flags are stored together.

- The *name of the language* is stored in the variable `name`.
- The *reference of the language* is stored in the variable `ref`. This reference is nothing but an arbitrary string.
- The boolean variable `saved` indicates, whether the language is *saved to a physical file* or not. The variable is set to the value `true` after the language was saved to disk, and it is set to the value `false` whenever a new representation or information of the language is computed. The variable is used to prevent saving a language more than once.
- The variable `inputis` indicates *how the regular language was defined.* The constants for this purpose (`REXINP etc.`) are defined in file `cons.h`
- The *size of the alphabet* is stored in the variable `sizeal`. The letters of the alphabet are numbered from `0` to `sizeal`, where the letter `0` is reserved for the empty word ε.
- The structure `lrex` holds a pointer to a *regular expression* for the language.

- The structure `orex` holds the generalized regular expression that was used to define the language if such a regular expression exists and AMoRE has already computed another one (namely a non-generalized one). This new expression is then stored in `lrex`.

  The idea with `lrex` and `orex` is that `lrex` shall always point to the "best" regular expression that has been computed, but if a new one is computed, AMoRE must not forget the original expression.

- The structure pointed to by `sexp` holds a *starfree expression* that represents the language. This element of the structure `regset` is filled if the system is asked to compute a starfree expression from a given language and if the given language is starfree.

- The structures pointed to by `ldfa` and `odfa` hold DFA's for the language. The usage of these two pointers is in the same spirit as explained in above with `lrex` and `orex`. That means that `orex` is used to remember the DFA originally typed in by the user as soon as another (namely minimal) DFA has been computed.

- The structure pointed to by `lnfa` and `onfa` hold NFA's for the language. The difference of these pointers is analogous to the pointers for DFA's.

- The structure pointed to by `oenfa` holds the *ε-NFA* that was used to define the language.

- The structure pointed to by `lmon` holds the *syntactic monoid* of the language.

- Additionally, there are various flags that indicate algebraic properties of the language. The flags hold values from the set {`true`, `false`, `unknown`}, where `true` and `false` indicate whether the language has a certain property or not and `unknown` indicates that it has not yet been tested whether the property holds or not. The flags are typically set after the system is asked to perform some tests on the given language.

  The flags are in detail: `folu`, `empty`, `full`, `starfree`, `localtest`, `definite`, `revdefinite`, `gendefinite`, `dotdepth1`, `piecetest` and `nilpotent`. The
  names of the flags describe the meaning of the according property very clearly, for example `revdefinite == true` iff the language is reverse definite, or `folu == true` iff the language is in $\mathrm{FOL}_{\mathrm{U}}$ (s. [MNP72]).

- The variable `localdegree` holds the *local degree* of the language if the degree is already computed.

A language is called *locally testable* iff it is a (finite) boolean combination of languages of one of the forms $w\Sigma^*$, Σ^*w and $\Sigma^*w\Sigma^*$. The *local degree* is defined as the minimal number k, such that words with length $\leq k$ can be found that enable such a representation.

Chapter 4

Memory Management

AMoRE has its own memory management, which enables quick and comfortable allocation and deallocation. This chapter is to give the reader hints how to add new procedures to the program.

4.1 Memory for Two Purposes

A typical function for AMoRE needs memory for two basically different purposes:

Firstly, to store a new language representation that it has been asked to compute. This memory must continue to exist after the function has terminated until the user gives an explicit command to deallocate this memory. Let us call the memory for this purpose *resident*.

Secondly, to store intermediate results like search trees etc. This memory serves as auxiliary space for calculations and has to be deallocated after the function that needed this extra memory has terminated. This type of memory will be called *temporary* in the following.

The file `buffer.c` provides some routines that handle temporary memory. The main observation is that a function will often require additional space during a computation. If any function was responsible for deallocating this, then it would need some more or less complicated mechanism to remember all space it has allocated.

Besides, in file `mem.c` there are some routines that allocate and deallocate resident memory for a regular language or any representation of it.

4.2 Temporary Memory

4.2.1 The Concept

The main idea is that the memory management anticipates by allocating a comfortably large slice of memory (called one *buffer*) at one time. Afterwards, any

(1) perform necessary computations, allocating new temporary space when needed.
(2) allocate resident memory for the result of 1.
(3) transport the information into the resident memory.
(4) deallocate all of the temporary memory.

Figure 4.1: Typical AMoRE Computation

time new temporary space is needed, it is taken from this slice. When the memory is not needed any more, the whole buffer is deallocated at one time.

One advantage of this method becomes evident with the following observation: For intermediate results programs often use chained lists, trees etc., i.e. memory of unbounded size. These consist of very small pieces of memory, which might lead easily to a memory fragmentation. AMoRE's management for temporary memory avoids this problem.

However, this strategy causes troubles as soon as the memory available in the current slice is too small for the requested new space. So in this case we need a new buffer. It will have either the standard size for a buffer (if the requested size fits in it) or exactly the requested size. Thus several buffers are needed, and each time new temporary space is needed, a buffer offering sufficiently much space has to be searched for. For reasons of efficiency, the number of available buffers is bounded.

Of course, the performance of this strategy depends very much on the choice of the default size for new buffers and on the number of buffers available at all. As a slight improvement of the explained method, the default size for a new buffer increases when more than a particular number of buffers are already in use. The constants for these purposes can be found in file `buffer.c` as well. See Table 4.1.

Constant	Default Value DOS	UNIX	Purpose
`BUFPOS`	6	16	Number of available buffers
`BUFSBSWITCH`	2	8	
`BUFS_BIG`	20000	4000000	Default size for buffers whose number is above `BUFSBSWITCH`
`BUFS_SMALL`	10000	524288	Default size for buffers whose number is smaller than `BUFSBSWITCH`

Table 4.1: Constants for Memory Management

A function performing any calculation using AMoRE's memory management could look like shown in Figure 4.1. Note that the step 3 has another advantage, apart from the necessity caused by the distinction between temporary and

resident memory: the data obtained in intermediate calculations is often stored in structured data of unbounded size, e.g. chained lists, whereas for later usage, the storage in compact and more easily accessible arrays etc. is advantageous. Therefore, this step would have to be done anyway.

4.2.2 Semi-Resident Memory

Let us have a closer look at Figure 4.1. Consider the situation that in step 1 another function of AMoRE, say `help()`, is called to perform an intermediate calculation. What if `help()` follows the same convention as Figure 4.1, i.e. it performs a calculation that needs auxiliary memory that it deallocates after usage? Then in its last step, `help()` will deallocate all of the temporary memory, including the one used by the calling function. This is certainly not intended, since the calling function might have to go on with its calculation after `help()` has terminated.

This sketched situation will occur as soon as the programmer abuses a function for an intermediate calculation though it was written for computing some resident language representation and needs auxiliary memory for this.

For this particular situation, which only occurs in function `grexnf()`, there is a special feature of the memory management. It is a mark that any function can set. Instead of step 4 of Figure 4.1, the last step of a function will be to deallocate the temporary memory allocated *after the mark was set the last time.*

4.2.3 The Functions for Temporary Memory

The functions for the handling of the temporary memory can be found in file `buffer.c`. There are:

`initbuf()` has to be called once to initialize the memory management.

`newbuf()` receives two positive integers `num` and `size` as parameters. `newbuf()` allocates `num*size` bytes of temporary memory and returns a pointer of type `*char`. Usually this pointer will have to be converted by a cast command. So the parameters and return types are just the same as for the standard command `calloc()`. The memory allocated by a `newbuf()` command is always initialized with `0`. There is a constant `MULTOF` in `buffer.c` with default value `4`. `newbuf()` makes sure that the address of the return pointer is always a multiple of `MULTOF`. This method causes the C program verifier `lint` to produce warnings you should not care for.

For example, if you need temporary space for an array `[0..i-1]` of objects of type `obj`, the appropriate command would be

```
pointer = (obj *)newbuf(i,sizeof(obj)),
```

where `pointer` is of type `obj*`. It points to the desired array afterwards.

`bufmark()` sets the mentioned mark in order to protect the temporary memory used so far from the next `freebuf()` command.

`buffree()` deletes this mark. The next `freebuf()` command will deallocate all of the temporary memory.

`freebuf()` deallocates all of the memory "above the mark".

`switchbuf()` begins a new buffer. This function is static, i.e. not visible from outside.

Note that the file `ext.h`, which every AMoRE-file includes, provides C-macros for frequently used `newbuf()` commands, such as `newarray(i)` for the allocation of an array indexed `0..i-1` of positive integers.

We remark that `newbuf()` uses internally the `calloc()`-command. So the AMoRE-management is not an alternative for the memory management of the operating system, but uses it.

4.3 Resident Memory

The resident memory is dealt with via the usual three operating system commands `calloc()`, `malloc()` and `dispose()`. For the programmer's convenience, there are several functions that allocate memory for special purposes, e.g. a DFA-structure, and also report errors in case the operating system cannot satisfy this request. Besides, there are functions that deallocate the memory for a complete C-structure, including the arrays and substructures contained in it. All of these functions can be found in file `mem.c`. Tables 4.2 and 4.3 lists some of them.

Function	Parameter Type	Return Type	allocates memory for:
`newlang()`	`int idx`	`language`	complete new language with index `idx` in the RAM-list
`newddelta()`	`posint s,q`	`ddelta`	DFA transition table with letters `1..s` and states `0..q`
`newndelta()`	`posint s,q`	`ndelta`	NFA transition table with letters `1..s` and states `0..q`
`newendelta()`	`posint s,q`	`ndelta`	ε-NFA transition table with letters `0..s` and states `0..q`
`newrexstr()`	`posint strl`	`string`	string of length `strl`, e.g. for regular expression
`newfinal()`	`posint q`	`mrkfin`	NFA/DFA table for final and initial states of an automaton with states `0..q`
`newdfa()`	none	`dfa`	structure for DFA
`newnfa()`	none	`nfa`	structure for NFA
`newrex()`	none	`regex`	structure for regular expression
`newmon()`	none	`monoid`	structure for monoid
`newar()`	`posint i`	`posint*`	array `0..i-1` of integers
`newbarray()`	`posint i`	`boole*`	array `0..i-1` of booleans
`newarray1()`	`posint i`	`posint**`	array `0..i-1` of arrays
`newdclass()`	none	`d_class*`	structure for one $\mathcal{D}$-class
`newdstruct()`	none	`dstruct`	structure for $\mathcal{D}$-class decomposition
`newsfexp()`	none	`starfreeexp`	structure for starfree expression
`newdarray()`	`posint i`	`darray`	array `0..i-1` of `d_class*`

Table 4.2: Functions of `mem.c` for Allocating Resident Memory
Note that the functions that allocate memory for structures and many-dimensional arrays do not allocate space for substructures or subarrays.

Function	Parameters	deallocates memory of:
`freedfa()`	`dfa d`	complete DFA pointed to by `d`
`freenfa()`	`nfa n`	complete NFA pointed to by `n`
`freerex()`	`regex r`	complete regular expression pointed to by `r`
`freesf()`	`starfexp sf;` `boole flag;` `posint mno`	complete starfree expression pointed to by `sf`. — `flag` must be `TRUE` iff memory for certain subarrays has already been allocated. `mno` must contain the maximal monoid element number.
`freedcl()`	`dstruct dcl;` `posint i`	complete $\mathcal{D}$-class structure. `i` must contain the number of $\mathcal{D}$-classes.
`freemon()`	`monoid mon`	complete monoid structure pointed to by `mon`

Table 4.3: Functions for Deallocating Resident Memory
Note that these functions also deallocate the memory used in subarrays and substructures.

Chapter 5

Installing and Customizing AMoRE

5.1 Files, Directories & Environment Variables

Each AMoRE-user should have one directory named `amore` with the three subdirectories `amorefile`, `printfiles`, `displayfiles` for language files, print files or display files, respectively. The environment variables `AMOREDIR`, `PRINTDIR`, `DISPLAYDIR` should contain the complete path names to these directories[1]. In the following, we will assume such a standard installation. If there is an environment variable named `AMORELPR`, this file will be used to put in the non-default values for the number of lines and columns on pages generated by the print command.

5.2 Important Constants

There are several constants which the user may modify in order to adapt his AMoRE-version to his needs. Most of the constants are defined in the file `cons.h`. We give a list of some of the constants and their meanings in Table 5.1. Besides, there are three compiler options worth mentioning:

`DOS` If this compiler option is set, DOS-code is generated. `MULTOF` might have to be adapted. See Section 4.2.3.

`UNIX` If this compiler option is set, UNIX-code is generated. Another Compiler Option, either `SYS5` or `BSD43` has to be set to choose the appropriate UNIX-Version.

`DEBUG` If set, a debugging version of AMoRE is generated.

[1] Actually, these environment variables may contain four arbitrary directories, not necessarily different; but the suggested path names are convenient.

Name	in File	Default	Meaning
General Constants			
MULTOF	buffer.c	4	Pointers to memory allocated by newbuf are always multiples of this constant.
ALPHSIZE	cons.h	26	Maximal size of alphabet, for which letter are represented by a, b, ... If the size of the alphabet of a language is larger, the letters will be named a1, a2, ...
Constants for Printing			
DEFAULT_PLINES	cons.h	66	Default value for the variable that determines the number of lines on a page generated by the print command "G5".
DEFAULT_PCOL	cons.h	80	Default value for the variable that determines the number of columns on a print page.
nmax	prtcomp.c	4	(Maximal length of a transitionlist displayed in the NFA-printfile)+1
Constants for Automaton Editors			
STDDCOLJUMP	cons.h	9	The width of one column in the DFA-editor
STDNCOLJUMP	cons.h	15	The width of one column in the NFA-editor
MAXNDSPLIT	cons.h	3	(Maximal length of a transitionlist displayed in the NFA-editor)+1

Table 5.1: Some Constants of AMoRE

Note that the values for the number of lines and columns can also be read from a configuration file determined by the environment variable AMORELPR.

Part III

The Algorithms

Chapter 6

From Regular Expression to NFA

6.1 Introduction

It is a well known fact that for any regular expression there exists an NFA accepting the same language. The proof for this fact is easier than the one for the converse statement and it usually contains an ε-NFA-construction that follows the inductively defined structure of the regular expression.

Unfortunately, this simple approach often yields superfluously large automata, which do even contain ε-transitions. The algorithm due to [BS86], which is implemented in AMoRE, is more sophisticated and yields an NFA whose number of states exceeds the sum of occurrences of letters in the input regular expression only by one.

6.2 The Main Idea

The main idea of the algorithm is to regard the different occurrences of letters in the input regular expression temporarily as different symbols, i. e. $(a \cdot b \cup b)^+ \cdot b \cdot a$ becomes $(1 \cdot 2 \cup 3)^+ \cdot 4 \cdot 5$. So we introduce a new alphabet Γ so that every letter from Γ corresponds to one occurrence of a Σ-letter in the input expression. Then these so-called "marked symbols" are the states for the automaton. The intuitive meaning of such a state is that the lastly read symbol was the corresponding symbol of Σ. Two states are connected by an automaton transition iff the second can follow the first in a matching word, i. e. a word that matches the marked regular expression. Similarly, the final states are those that can be the last ones in such a word. The initial state is an extra state, which is connected to all those states that can be the first ones in a matching word.

The transition labels are always the unmarked symbols from Σ that correspond to the state in which the transition ends.

The idea should be fairly clear from the Figure 6.1, which shows the automaton that the algorithm produces from of the above example expression.

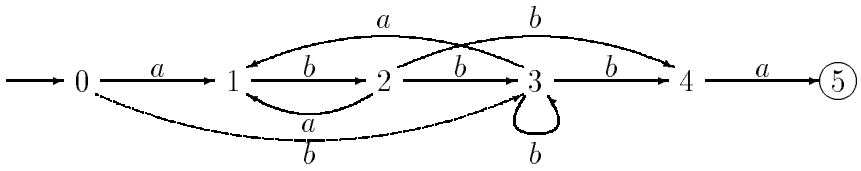


Figure 6.1: Automaton for $(a \cdot b \cup b)^+ \cdot b \cdot a$

Obviously, this algorithm produces quite directly an NFA that works the same way the regular expression does: Each state of the automaton corresponds directly to one position in the regular expression. It scans a word from left to right and matches each input symbol with one symbol in the regular expression. The automaton guesses non-deterministically the next position whenever the next (unmarked) input symbol enables more then one match with one of the (distinct) symbols in the regular expression.

Note that this algorithm cannot be extended in a straightforward way to cope also with intersection and complement. The intersection would require the ability to match two or more expressions simultaneously, and the complement would require some mechanism to reject words that match a certain regular expression.

6.3 The Algorithm

Before we give the algorithm explicitly we will introduce some helpful definitions.

Definition 6.1 Let r be a regular expressions over some alphabet Γ, let $b \in \Gamma$. Then define

$$\begin{aligned} \mathit{first}(r) &:= \{a \in \Gamma \mid \exists v \in \Gamma^* : av \in L(r)\} \\ \mathit{last}(r) &:= \{a \in \Gamma \mid \exists v \in \Gamma^* : va \in L(r)\} \\ \mathit{follow}_r(b) &:= \{a \in \Gamma \mid \exists u, v \in \Gamma^* : ubav \in L(r)\} \end{aligned}$$

If r is clear from the context, we will omit this index and simply write *follow*.

That is, for a fixed regular expression *first* (*last*, *follow* resp.) is the set of symbols that can be the first one (or the last one or the successor of a specified symbol, resp.) in a matching word.

Provided we have three functions compfirst, complast and compfollow that compute those sets, we can now formulate the algorithm as described so far (see Figure 6.2).

{ input: regular expression r with symbols from Σ }
(1) mark the letters in r and obtain r' with distinct symbols $q_1, \ldots, q_n$ and a mapping $\phi : \{q_1, \ldots q_n\} \longrightarrow \Sigma$.
(2) $F :=$ complast(r');
(3) $\Delta := \emptyset$.
For all $q \in$ compfirst(r')
$\Delta := \Delta \cup \{(q_0, \phi(q), q)\}$
(4) compute $follow(q_i)$ for all $i \in \{1, \ldots, n\}$.
(5) For all $i \in \{1, \ldots, n\}$
For all $q \in follow(q_i)$
$\Delta := \Delta \cup \{(q_i, \phi(q), q)\}$
{output: NFA $(\{q_0, \ldots q_n\}, \Sigma, q_0, \Delta, F)$}

Figure 6.2: The Algorithm

The problems that remain to be solved now are the functions compfirst, complast and the implementation of step 4. For these problems we give two Lemmas that will allow the recursive implementation of these functions. Note that we are only interested in marked regular expressions, i. e. regular expressions with distinct symbols.

Lemma 6.2 Let t_1, t_2 be two regular expressions, let $a \in \Sigma$.

$$\begin{aligned} first(a) &= \{a\} \\ first(t_1 \cup t_2) &= first(t_1) \cup first(t_2) \\ first(t_1 \cdot t_2) &= \begin{cases} first(t_1) & \text{if} \quad \varepsilon \notin L(t_1) \\ first(t_1) \cup first(t_2) & \text{if} \quad \varepsilon \in L(t_1) \end{cases} \\ first(t_1^*) &= first(t_1) \end{aligned}$$

Analogue relations hold for *last*.

Lemma 6.2 tells intuitively how the *first*-set (*last*-set) of an expression looks like, provided you know the *first*-set (*last*-set) of the subexpressions. It can be used to implement the two recursive functions compfirst and complast.

Before we state the mentioned second Lemma we will extend $follow_r$ to arbitrary subexpressions of r:

Definition 6.3 Let r be a regular expression over Γ and s be a subexpression of r. Then

$$follow_r(s) := \{a \in \Gamma \mid \exists u, v, w \in \Gamma^* : r \text{ generates } uwav \text{ with } w \text{ generated by } s\}.$$

```
procedure compfollowsets(r,f,Var Fol):
{ r subexpression of the input expression, f = follow(r). }
  if r = a ∈ Γ then Fol[a] := f
  elseif r = s · t then
    compfollowsets(t,f,Fol)
    if ε ∉ L(t) then
      compfollowsets(s,first(t),Fol)
    else
      compfollowsets(s,first(t) ∪ f,Fol)
    endif
  elseif r = s ∪ t then
    compfollowsets(t,f,Fol)
    compfollowsets(s,f,Fol)
  elseif f = s* or f = s+ then
    compfollowsets(s,f ∪ first(s),Fol)
  endif
```

Figure 6.3: How to Compute the *follow*-Sets

Now we have:

Lemma 6.4 Let r be a regular expression with distinct symbols from Γ, let s be a subexpression of r and let $f = follow_r(s)$. The following implications hold:

$$
\begin{array}{rcl}
follow_r(r) = \emptyset & & \\
s = t_1 \cup t_2 & \Longrightarrow & follow_r(t_1) = follow_r(t_2) = f \\
s = t_1 \cdot t_2 & \Longrightarrow & follow_r(t_2) = f \\
 & & follow_r(t_1) = \begin{cases} first(t_2) & \text{if} \quad \varepsilon \notin L(t_2) \\ first(t_2) \cup f & \text{if} \quad \varepsilon \in L(t_2) \end{cases} \\
s = t^* & \Longrightarrow & follow_r(t) = f \cup first(t)
\end{array}
$$

Lemma 6.4 tells intuitively how the *follow*-sets of the subexpressions look like, provided you know the *follow*-set of the whole expression.

Figure 6.3 shows a recursive procedure that will, called with $f = \emptyset$, perform the step 4 of the algorithm of Figure 6.2 by filling an array Fol such that $\forall i \in \{1, \ldots, n\} : Fol[q_i] = follow(q_i)$.

The difference between the recursive algorithms for compfirst and compfollowsets is that in the syntax tree representing the regular expression, compfirst starts at the leaves and moves conceptionally upwards, whereas compfollowsets starts at the root and moves downwards.

6.4 About the Implementation

Three Difficulties A small difficulty in the straightforward implementation of the algorithm of Figure 6.2 is to deal with sets of marked symbols. However, slight modifications avoid dealing with sets.

Another difficulty is not that obvious: the actual representation of regular expressions as postfix strings is very appropriate for the computation of *last*, but for the computation of *first* the prefix notation is more comfortable. This is due to the case distinction in the calculation of $first(t_1 \cdot t_2)$: the regular expression t_1 should be dealt with at first so that we know whether $\varepsilon \in L(t_1)$ before we treat t_2. This direction of processing is hard to implement if you only have the postfix notation.

Moreover, we would like to modify the algorithm to accept also regular expressions that contain a Σ such as $\Sigma^* ab \Sigma^*$.

Solutions The conversion into prefix notation can be done simultaneously to the computation of the *last*-sets out of the postfix notation.

We make the observation that we can describe the sets on the right hand side of the relations in Lemma 6.4 as *first*-sets of certain regular expressions, in particular:

Let s' be a regular expression with $follow_r(s) = first(s')$, then

$$
\begin{array}{rcl}
s = t_1 \cup t_2 & \Longrightarrow & follow_r(t_1) = follow_r(t_2) = first(s') \\
s = t_1 \cdot t_2 & \Longrightarrow & follow_r(t_2) = first(s') \\
 & & follow_r(t_1) = \begin{cases} first(t_2) & \text{if} \quad \varepsilon \notin L(t_2) \\ first(t_2 \cup s') & \text{if} \quad \varepsilon \in L(t_2) \end{cases} \\
s = t^* & \Longrightarrow & follow_r(t) = first(t \cup s')
\end{array}
$$

So instead of computing the follow sets we generate those regular expressions and apply compfirst afterwards. The generation of such a regular expression does not necessarily require extra space or time: When the postfix notation is converted into a prefix notation, it almost appears as a subexpression. In case it is not actually, the temporary insertion of one symbol yields the desired subexpression. Sets are not needed as return values either: the function compfirst inserts the necessary transitions into the transition table as well and the function complast marks all states in $last(r)$ as final.

The occurrences of Σ in the input expression are dealt with the same way as single letters, that means that we firstly replace them with distinct symbols. But each time we insert a transition leading to a state that corresponds to Σ, we will actually insert $|\Sigma|$ transitions instead, one for each symbol.

6.5 Details about the Implementation

The function visible from outside is `rex2nfa()`. It receives a value of type `regex`, that is a pointer to a structure that represents a regular expression, but only the postfix notation contained in this structure is relevant. The regular expression must be standard , i. e. without the operations intersection, complement or set difference.

The three functions that do the main work are `compfirst()`, `complast()` and `compff()`. Additionally, there is an auxiliary function `getexpr()` that either copies a whole regular (sub-)expression at once from `rex` to `prerex` or counts the number of symbols.

The function `complast()` receives a value of type `mrkfin`, that is a pointer to an array that shall keep the information for the final and initial states afterwards. If this pointer is not `NULL`, `complast()` will trace the postfix notation of the regular expression that begins at `lastrex` backwards and mark the states corresponding to the symbols inside final if necessary. Simultaneously, this function creates a prefix notation of the regular expression. Moreover, `complast()` returns `TRUE` iff the empty word matches this regular (sub-)expression.

The function `compfirst()` receives a value of type `ndelta`, that is a pointer to an array that shall keep the information for the transition function. It traces the prefix notation starting at `firstprerex` from left to right and connects the state `constate` to the states corresponding to those symbols inside that can be the first in a matching word. `compfirst()` returns `TRUE` iff the empty word matches the subexpression starting in `firstprerex`.

The function `compff()` receives a number `f` and the pointer of type `mrkfin`. `compff()` treats the regular subexpression in postfix notation that begins at `ffrex`. Let r denote this expression. When `compff()` is called, `f` must contain the position in the prefix notation where the subexpression starts that represents the *follow*-set of r. Then it traces the postfix notation and inserts all necessary transitions starting in states that correspond to symbols in r. Doing so, a modified prefix notation is created in order to prepare the recursive calls of `compff()`.

The counters necessary for the tracing of `rex` and `prerex` are all global, since they are used by recursive functions. There are `lastrex` and `lastprerex` (both used in `complast()`), `firstprerex` (used in `compfirst()`), `getrex` and `getprerex` (both used in `getexpr()`) and finally `ffrex` and `ffprerex` (both used in `compff()`).

Tricky Data Structures The global string `rex` holds the input regular expression in postfix notation, the global string `prerex` is filled twice with its prefix notation: Once by `complast()` so that `compfirst()` can work on it, and the second time by the function `compff()`, which recursively computes the *follow*-sets. This second time, special characters `skipch` and `unskipch` (= '!' and '?') are inserted in the prefix notation. The meaning of `unskipch` is intuitively that it

can be omitted from the expression. The meaning of `skipch` is that the currently intended regular expression does not continue at the following position in `prerex` but at the position that the array `skipval[]` holds. Thus if `prerex[i]==skipch` one may regard `prerex[skipval[i]]` as the next symbol in the prefix notation; and if `prerex[i]==unskipch` the next symbol simply is `prerex[i+1]`.

Let us have a look at the following example from the beginning: $c \cup (ab)^*$. Its postfix notation is $cab.*U.$When `compff()` is called first, the relevant part of the string `prerex` (that is starting at the position `ffprerex`) will contain the empty string @ and also the local variable f will point to this position. `rex[ffrex]` will be the outermost union character, which is the root of the tree representation of the hole expression. This situation can be interpreted as: The current subexpression which has to be dealt is the hole input and the *follow*-set of it is $first(\varepsilon)=\emptyset$.

Now `compff` will decrease `ffrex`, recognize the union and will recursively call itself twice: firstly with `ffrex` pointing to the star (meaning the regular expression $(c \cdot b)^*$) and afterwards with `ffrex` pointing to the c (meaning the subexpression c).

The prefix notation that is generated before the first one of these calls is !@, and the entry in `skipval[]` for this '!' points to the '@'.

When `compff()` reads this '*', it has to build an expression whose *first*-set is the *follow*-set of the symbols of the expression $a \cdot b$. For this aim, this subexpression is copied in front by `getexpr()` (without any further treatment, the states corresponding to the symbols inside are not connected to anything) and a union symbol U is put in front afterwards. Note that this union does not actually appear in the input expression; it builds temporarily the union of the expression under the Kleene star $(a \cdot b)$ together with the expression that could come afterwards in a matching word (@).

So before `compff()` is called with `ffrex` pointing to the concatenation dot, the part of `prerex` stating at `f` will be $U.ab!!@$, where skipval points to the '@' for both of the '!'.

Now let us have a look at this call. The concatenation requires two recursive calls: one with `ffrex` pointing to the b and the other one with `ffrex` pointing to the a. The first one of these is already well prepared because `f` points correctly to $U.ab!!@$; when it ends, it will leave `f` pointing to $b!U.ab!!@$, where `skipval[]` points to the union for the fresh '!'. If you read this as a prefix notation, it is simply a b, and that is indeed an expression representing the *follow*-set of a as required for the call with `ffrex` pointing to it. But when `compff()` has been called for both the a and the b and the call for the concatenation dot is to be finished, this concatenation dot is copied in front and it completes the prefix notation of $(a \cdot b)$, so it can be used for further calls.

When the call for the star is to be completed, the temporary union in replaced by this star, thereby producing a correct prefix notation of $(a \cdot b)^*$.

What remains to be done is the second call of `compff()` inside the outermost

call. There, `ffrex` points to the c and f still points to the @. When this call is left finally, the string `prerex` will contain $?Ua*.ab??@$, which is again a correct prefix notation of the expression dealt with inside this call (namely the whole one).

Summarizing, the function `compff` prepends to the current `prerex` a prefix notation of the postfix notation starting in `ffrex`. It may use special symbols to generate the expressions required by the algorithm, but when it ends, it has to overwrite them by '?' or the original symbols.

Chapter 7

From NFA to Regular Expressions

7.1 The Algorithm

For the purpose of the transformation it is useful to introduce the notion of a *generalized transition graph (GTG)*. That is something similar to an NFA, but the edges may be labelled with arbitrary standard regular expressions instead of just letters.

The main idea of the algorithm (as presented e.g. in [Ma74]) is to delete nodes in a GTG consecutively and to modify at each step the transition labels properly to make sure that the represented language remains the same. Let us begin with a precise definition of a GTG.

Definition 7.1 A *generalized transition graph (GTG)* is given by a quintuple $\mathcal{A} = (Q, \Sigma, q_0, \gamma, q_f)$, where

- Q is a set of states,
- Σ is an alphabet,
- $q_0, q_f \in Q$, $q_0 \neq q_f$ are the unique start and final states.
- $\gamma : Q \times Q \longrightarrow R_\Sigma$, where R_Σ is the set of all regular expressions over Σ.

A word $w \in \Sigma^*$ is *accepted* by $\mathcal{A}$ iff there exists a finite sequence of states $q_0 = p_0, p_1, \ldots, p_n = q_f \in Q$, $(n \geq 1)$, and a decomposition of w into words $w = u_1 \ldots u_n$ such that for any $1 \leq i \leq n$ $u_i \in L(\gamma(p_{i-1}, p_i))$, that means u_i is in the language generated by the label of the transition form p_{i-1} to p_i.

In the subsections 7.1.1 to 7.1.3 we show three simple constructions necessary for the algorithm.

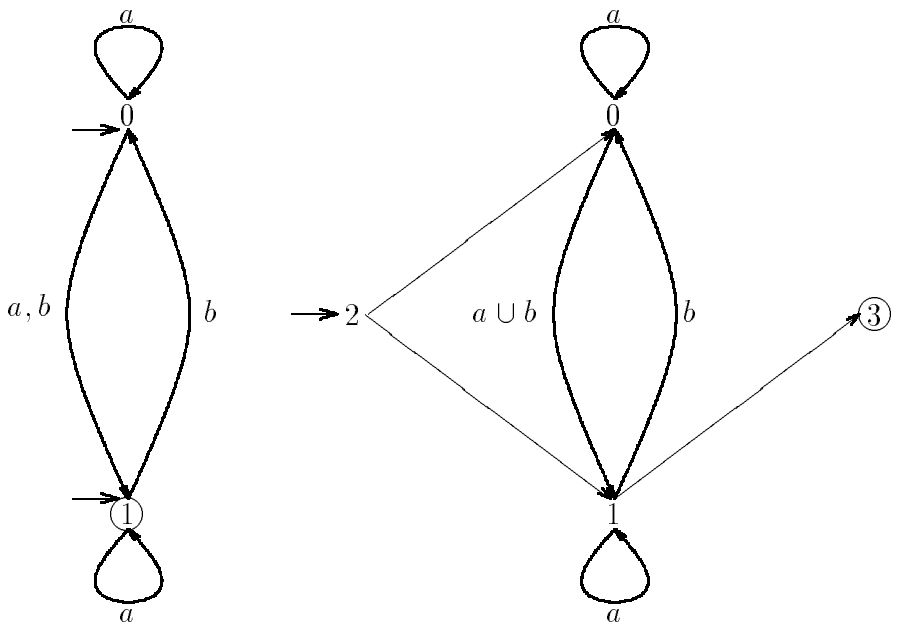


Figure 7.1: An example NFA and its corresponding GTG. As usual, ε – labels are omited. In the GTG also the transitions labeled with $\emptyset$ are dropped.

7.1.1 Construction of a GTG from an NFA

Obviously it is easy to construct a GTG accepting the same language as a given NFA. Let $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ be some NFA. Then the GTG $\mathcal{A}' = (Q \cup \{q_0, q_f\}, \Sigma, q_0, \gamma, q_f)$ where q_0, q_f are distinct new objects not in Q and

$$\gamma(p,q) = \begin{cases} a_1 \cup \ldots \cup a_n & \text{if } p,q \in Q \text{ and} \\ & \{a_1, \ldots, a_n\} = \{a \in \Sigma \mid (p,a,q) \in \Delta\} \neq \emptyset \\ \varepsilon & \text{if } (p,q) \in (\{q_0\} \times I) \cup (F \times \{q_f\}) \\ \emptyset & \text{else} \end{cases}$$

accepts the same language as $\mathcal{A}$. Note that the images under γ are always meant to be regular expressions.

7.1.2 Deletion of a Loop in a GTG

Let $\mathcal{A} = (Q, \Sigma, q_0, \gamma, q_f)$ be a GTG and let $\overline{q} \in Q \backslash \{q_0, q_f\}$. Then the GTG $\mathcal{A}' = (Q, \Sigma, q_0, \gamma', q_f)$ where

$$\gamma'(p,q) = \begin{cases} \gamma(p,q) & \text{if } p \neq \overline{q} \\ \gamma(\overline{q},\overline{q})^*\gamma(\overline{q},q) & \text{if } p = \overline{q}, q \neq \overline{q} \\ \emptyset & \text{if } p = q = \overline{q} \end{cases}$$

accepts the same language as $\mathcal{A}$. Note that the second case causes the resulting regular expression to get long.

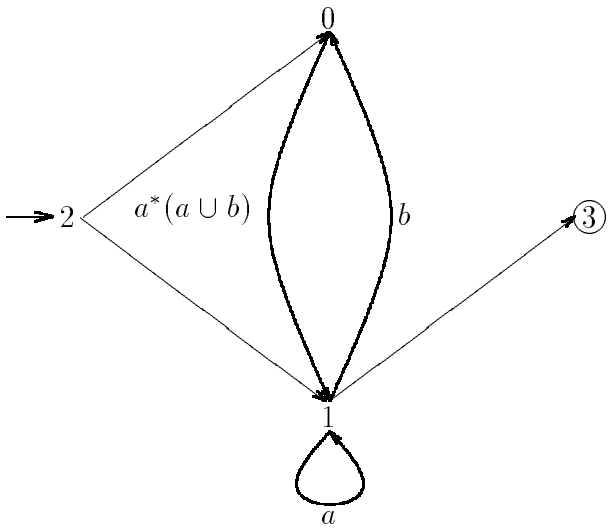


Figure 7.2: The example GTG after applying `destroyloop(0)`.

7.1.3 Deletion of a State

Let $\mathcal{A} = (Q, \Sigma, q_0, \gamma, q_f)$ be a GTG and let $\overline{q} \in Q \backslash \{q_0, q_f\}$ be such a state that $\gamma(\overline{q}, \overline{q}) = \emptyset$. Then the GTG $\mathcal{A}' = (Q \backslash \{\overline{q}\}, \Sigma, q_0, \gamma', q_f)$ where

$$\gamma'(p, q) = \gamma(p, \overline{q})\gamma(\overline{q}, q) \cup \gamma(p, q)$$

accepts the same language as $\mathcal{A}$ but has one node less.

In the implementation both of 7.1.2 and of 7.1.3 the modification of γ is "optimized" in the sense that trivial rules like $r \cup \emptyset = r$, $r \cdot \varepsilon = r$, $r^* \cdot r = r^+$ and so on are applied in order to prevent the resulting regular expression from getting more complicated than necessary.

7.1.4 Strategy of the Algorithm

The strategy of the algorithm now is simply to transform a given NFA $\mathcal{A}$ into a GTG like in section 7.1.1 and then apply 7.1.2 and 7.1.3 alternately until q_0 and q_f are the only states left.

Compute a GTG $\mathcal{A}'$ from $\mathcal{A}$ like in Construction 7.1.1
while $Q' \neq \{q_0, q_f\}$ do
    select any state $\overline{q}$ from $\mathcal{A}'$
    destroy the loop of $\overline{q}$ in $\mathcal{A}'$ like in 7.1.2
    delete $\overline{q}$ from $\mathcal{A}'$ like in 7.1.3

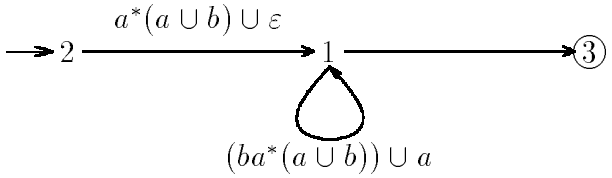


Figure 7.3: The example GTG after applying `deletenode(0)`.

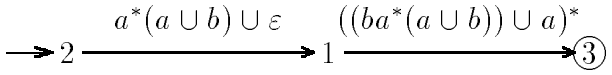


Figure 7.4: The example GTG after applying `destroyloop(1)`.

7.2 Details about the Implementation

7.2.1 Data Structures

The GTG is stored in the two dimensional array `area[]` of C-structures `edgelbl`, which contain selectors `s` for a string and `l` for its length. That means if `q` and `p` are two numbers representing states of the GTG, then `area[p][q].s` contains the label of the edge from `p` to `q`, that is a string that is a regular expression in postfix notation; and `area[p][q].l` contains the length of that string. This length is stored explicitly to make it easier to allocate memory for the labels that are built.

The array `area` has indices from `qst` to `qno`. `qst` is initialized with 0 and is incremented up to `qno` -2 when the states are successively eliminated. `qno` is the number of states of the NFA plus two. The numbers $0 \ldots$ `qno` -2 correspond to the numbers of the input NFA. `qno-1` and `qno` are the numbers of the additionally introduced initial and final state.

The C-macros `CCC` and `CCS` and `test` are helpful to manipulate strings. For a string `r1` you have `test(r1)==TRUE` iff `r1` represents the empty string (denoted by @ in AMoRE).

For strings `r1` and `r2`, `CCC(str,l,r1,r2)` allocates memory for a string `str` of length `l` and then concatenates `r1` and `r2` and stores the result in `str`.

`CCS(str,l,r1,r2,target)` allocates memory for the string `str` of length `l` and then concatenates `r1` and `r2` and stores the result in `target`.

Both `CCC` and `CCS` leave the strings `r1` and `r2` unchanged. Note that in both cases `str` must be a pointer different from `r1` and `r2` and `l` must be large enough for the concatenation, otherwise you get troubles.

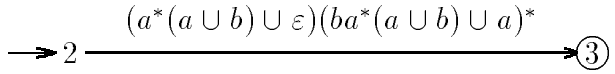


Figure 7.5: The example GTG after applying `deletenode(1)`. The resulting regular expression is the label transition from 2 to 3.

7.2.2 The Functions

The function visible from outside is `nfa2regex()`. It receives a pointer to an NFA and returns a pointer to the resulting regular expression.

The constructions 7.1.1, 7.1.2 and 7.1.3 are implemented by the functions `init()`, `destroyloop()` and `deletenode()`. The function `init()` receives a pointer to the input NFA and then copies the information to the array `area[]`. The memory needed for this array is allocated in `init()` as well. The function `destroyloop()` receives the number `qst` as a parameter and eliminates the possible loop of this state in the GTG that the array `area[]` holds. The function `deletenode()` receives the number `qst` and deletes this state from the GTG in `area[]` by applying the required modifications.

The algorithm 7.1.4 is implemented simply by initializing `qst=0` and calling `destroyloop(qst)` and `deletenode(qst)` in alternation and incrementing `qst` until `qst=qno-2`.

Chapter 8

From Generalized Expressions to NFA

Generalized regular expressions may include the operators $\cap$, $\sim$, $-$ denoting intersection, complement and set difference, respectively. In the following we will introduce an inductive method of computing an NFA $\mathcal{A}$ from a generalized regular expression f with $L(\mathcal{A}) = L(f)$. The maximal "standard-regular"–subexpressions of f (i.e. the maximal subformulas of f not containing one of the operators $\cap, \sim$ or $-$) will serve as the start of the induction; we use the method described in [BS86] to compute a NFA from these subformulas. A general description of the algorithm is given in 8.1, the sections 8.1.1 – 8.1.4 will describe the inductive step for each operator.

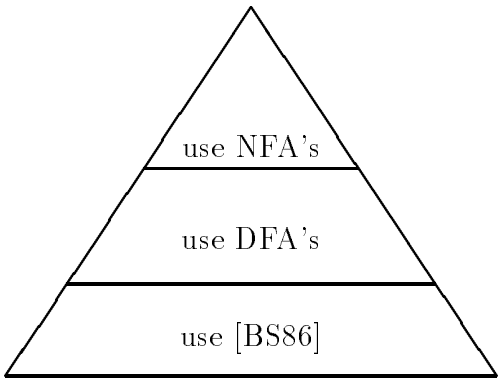


Figure 8.1: Conceptual cuts through the tree

8.1 The Algorithm

There is a natural correspondence of a given regular expression f to a tree, whose nodes are labelled with the operators or f. The principal idea is to analyse the tree in a bottom–up manner and at the same time construct automata for the subtrees. Figure 8.1 shows how the tree representing the generalized regular expression can conceptually be cut into three levels, each of which is dealt with differently.

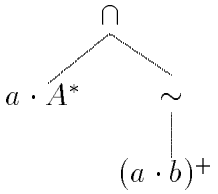
The lowest level consists of subtrees that are labelled only with the standard–regular operators, which can be dealt with by use of [BS86] very efficiently. See Chapter 6 for the implementation of this algorithm.

The middle level contains all nodes that are not among those in the first level and have nodes labelled with one of the non–standard operators $\cap$, $\sim$, $-$ above them. These nodes require the use of DFA's. This fact makes this level the most problematic one.

The top level contains all nodes that are not inside the other two levels but above which there are only nodes labelled with standard operators. For these nodes the use of NFA's will do.

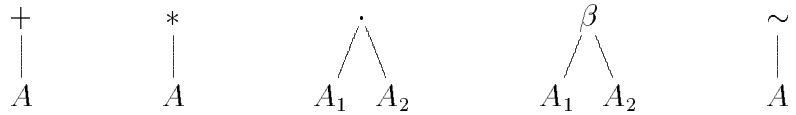
The first step is to compute from a given generalized expression f a tree T that represents f.The leaves of T will be the maximal standard-regular subexpressions of f. These nodes represent the subtrees of the lowest level in the conceptual tree of Figure 8.1.

For example, if $f = (a \cdot A^*) \cap \sim (a \cdot b)^+$, then T would be



The top level consists only of the $\cap$-labelled node, the middle level contains only the $\sim$-labelled one.

In order to construct an automaton for a given expression, we have to deal with the following types of subtrees:



where β is a binary boolean operator. We now apply the methods described in 8.1.1 – 8.1.4 to T' and obtain an automaton A that accepts exactly the language of the subformula of f that is represented by T'. By replacing T' in T with A, we obtain a tree T''. T'' is of the same form as T (i.e. the inner nodes of T'' are elements of the set $\{\sim, \cap, \cup, -, +, *, \cdot\}$ and the leaves are NFA's), but has less nodes than T. The repeated application of the method "computation of an

automaton for a subtree and replacing the subtree with the automaton" finally leads to a tree that consists only of one node. This is of course an automaton A with $L(A) = L(f)$.

We note that the implementation applies method 8.1.4 to the *maximal boolean subtrees*, i.e. the maximal subtrees of T with inner nodes from the set $\{\sim, \cap, \cup, -\}$.

In analogy, the implementation applies method 8.1.3 to those maximal subtrees of T that have nothing but concatenation dots as inner nodes.

Now we show how to obtain automata for a language L of the form $L = L_1^+$ or $L = L_1^*$ or $L = L_1 \cdot L_2$, provided we have got automata for L_1 and L_2.

These constructions make essential use of nondeterminism, but in the middle level of the conceptual tree of Figure 8.1 DFA's are needed to handle the boolean operators.

So we proceed like this:

1. Convert the input DFA into an NFA. This is trivial and only a question of data structures.

2. Apply the construction.

3. Convert the output NFA into a DFA with the powerset construction.

4. Minimize the DFA.

8.1.1 case "+"

Let $A_1 = (\Sigma, Q_1, I_1, \Delta_1, F_1)$ be some NFA. Then, for the NFA $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ defined by $Q := Q_1$, $I := I_1$, $F := F_1$ and $\Delta := \Delta_1 \cup \{(p, a, q) \in Q \times \Sigma \times Q | \, p \in F$, and $(q_0, a, q) \in \Delta_1$ for a $q_0 \in I_1\}$, we have $L(\mathcal{A}) = (L(\mathcal{A}_1))^+$. The implementation of this method is done in `grexnfa.c->plus_star_dfa`.

8.1.2 case "*"

Let $\mathcal{A}_1 = (\Sigma, Q_1, I_1, \Delta_1, F_1)$ be some NFA. Then, for the NFA $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ defined by $Q := Q_1 \cup \{\hat{q}\}$ (for a $\hat{q} \notin Q_1$), $I := \{\hat{q}\}$, $F := F_1 \cup \{\hat{q}\}$ and $\Delta := \Delta_1 \cup \{(p, a, q) \in Q \times \Sigma \times Q | \, p \in F$, and $(q_0, a, q) \in \Delta_1$ for a $q_0 \in I_1\} \cup \{(\hat{q}, a, q) \in Q \times \Sigma \times Q | \, (q_0, a, q) \in \Delta_1$ for a $q_0 \in I_1\}$, we have $L(\mathcal{A}) = L(\mathcal{A}_1)^*$. The implementation of this method is done in `grexnfa.c->plus_star_dfa` as well.

8.1.3 case "."

Let $\mathcal{A}_1 = (\Sigma, Q_1, I_1, \Delta_1, F_1)$, $\mathcal{A}_2 = (\Sigma, Q_2, I_2, \Delta_2, F_2)$ be some NFA's. We assume $Q_1 \cap Q_2 = \emptyset$. Then, for the NFA $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ defined

by $Q := Q_1 \cup Q_2$, $I := I_1, \Delta := \Delta_1 \cup \Delta_2 \cup \{(p,a,q) \in Q \times \Sigma \times Q | \, p \in F_1$ and $(q_0,a,q) \in \Delta_2$ for a $q_0 \in I_2\}$ and

$$F := \begin{cases} F_1 \cup F_2 & \text{if } I_2 \cap F_2 \neq \emptyset \\ F_2 & \text{otherwise} \end{cases},$$

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

We note that this method can easily be generalized for languages of the form $L(A_1) \cdot \ldots \cdot L(A_n)$. `grexnfa.c->conc_dfa` implements this method.

8.1.4 Boolean Operators $\cap, \cup, -$ and $\sim$

This method requires DFA's as input, that is why any NFA that is below maximal boolean subtree (and therefore in the middle level of the conceptual tree in Figure 8.1) is converted to a DFA by using the classic subset construction [RS59] and the minimization procedure of [Ho71]. Fortunately, the output automata of this method are deterministic, too, so there is no problem when boolean operators appear in the middle level.

Let L be a boolean combination of languages $L_1, \ldots, L_n$ and let $\mathcal{A}_i = (\Sigma, Q_i, q_{0\,i}, \delta_i, F_i)$ be DFA's with $L(A_i) = L_i$ (for $i = 1, \ldots, n$). In a natural way a boolean function $\beta : \{0,1\}^n \longrightarrow \{0,1\}$ can be obtained by defining $\beta(x_1, \ldots, x_n)$ as the (usual boolean) evaluation of the boolean expression that is obtained from the boolean combination for L by replacing every occurrence of an L_i with the boolean variable x_i (for $i = 1, \ldots, n$). For example, for the boolean combination $L = L_1 \cap (\Sigma^* \backslash L_2)$ the corresponding function is $(x_1, x_2) \mapsto x_1 \wedge \neg x_2$.

Then, for the DFA $A = (\Sigma, Q, q_0, \delta, F)$ defined by $Q := Q_1 \times \ldots \times Q_n$, $I := I_1 \times \ldots \times I_n$, $\delta : Q \times \Sigma$, $((p_1, \ldots, p_n), a)) \mapsto (\delta(p_1), a), \ldots, (\delta(p_n), a))$ and $(q_1, \ldots, q_n) \in F :\Leftrightarrow \beta_f(x_1, \ldots, x_n) = 1$ with $x_i := 1 \Leftrightarrow q_i \in F_i$ (for $i = 1, \ldots, n$), we have $L(\mathcal{A}) = L$. Of course, we can restrict $\mathcal{A}$ to its reachable states.

The implementation of this method is done in `grexnfa.c->boole_dfa`, but the situation is a bit simpler there because in the boolean combination none of the languages $L_1, \ldots, L_n$ appears more than once.

8.2 Details about the Implementation

8.2.1 Optimization Strategy

Firstly, the tree representing the regular expression is computed in function `postfix2tree()`. The leaves of this tree are labelled with standard regular expressions and represent the subtrees of the lowest conceptual level illustrated in Figure 8.1. Afterwards it is optimized in the sense that trivial rules like $\sim\sim L = L$ or $\emptyset \cup L = L$ are applied in order to reduce the number of nodes. This optimization is performed in `optimize_tree()`.

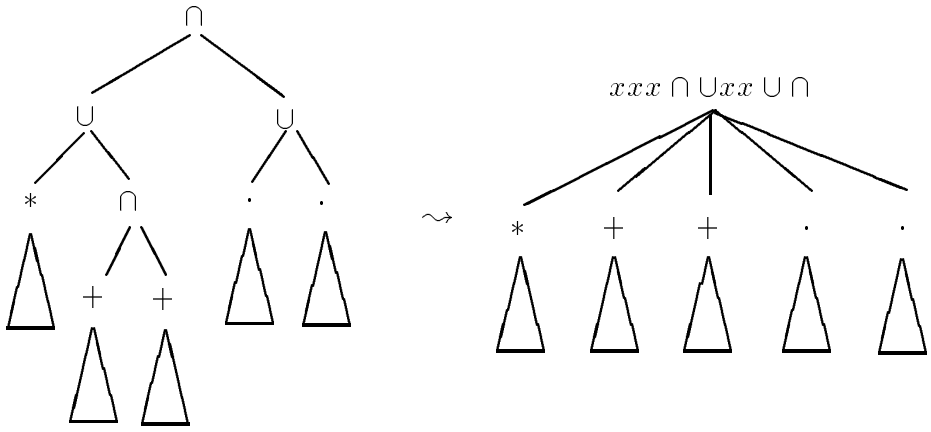


Figure 8.2: `reorder_node()`

Afterwards, function `mark_tree()` marks all those nodes of the tree (by setting the flag `is_reg_ab`), above which all nodes are labelled only with one of the standard operators $+, *, \cdot, \cup$. These are the nodes of the top level of the conceptual tree in Figure 8.1.

The transformation of these nodes into automata is done by using NFA's in function `second_run()` whereas all the other nodes require the use of DFA's (function `first_run`).

The next step is to simplify the structure of the tree by building nodes that have several sons whenever the tree has either several consecutive nodes labelled with boolean operators, or several ones labelled with a concatenation symbol. In the first case, a string consisting of `x`'s and the operators is built up and inserted as the label. This string is a postfix notation of the subtree in which every subtree whose root label is none of the boolean ones corresponds to an '`x`'.

This simplification is done in function `reorder_node()`, which is called by `reorder_tree` and is illustrated in Figure 8.2:

Again, this is only done in case there are nodes labelled with non-standard-symbols above, which will be indicated by the node's flag `is_reg_ab`. Finally, the desired NFA is computed in two steps, namely `first_run()` and `second_run()`.

The main difference is that `first_run` produces DFA's for each node in the middle conceptual level, whereas `second_run()` deals with NFA's for the top level.

Inside `first_run()`, the functions `boole_dfa()` and `conc_dfa()` are used to compute the DFA accepting a language defined by a boolean expression (or a concatenation resp.) of possibly more than two languages, each of which is defined by a DFA. This can be done easily by using the strings generated in `reorder_tree()`.

8.2.2 Data Structures of `genrex2nfa`

The trees representing regular expressions are chained lists, built up by structures `*t_elem` as defined in `types.h`.

`op` holds one of the constants `concatch, boolech, regch, starch, plusch`, which mean that the label of the node is a concatenation symbol, a boolean expression containing $\cup, \cap, \sim, -$, a regular expression, a Kleene-star or a Kleene-plus, respectively.

`expr` holds the label itself. Initially, the leaves are labelled with standard regular expressions in postfix notation, every inner node holds its operator.

`sons_no` holds the number of sons. Leaves have `sons_no==0`.

`son` holds an array of pointers to the sons. This array is indexed beginning with zero, so if `node` is of type `t_elem`, then `node->son[1]` is a pointer to its second son.

`father` holds a pointer to the father.

`son_passed` will usually indicate the number of sons that have already been visited during the depth-first-search inside one of the functions `postfix2tree(), optimize_tree(), reorder_tree(), first_run(), second_run()`. By convention, functions have to re-initialize `son_passed` to zero after usage.

`is_reg_ab` *(is regular above)* is a boolean flag set to `TRUE` if there is no node labelled with $\cap, \sim$ or $-$ among the predecessors, i.e. the node is in the top conceptual level.

`da` or `na` may contain a pointer to a DFA (NFA, resp.) accepting the "subtree language" that is the language represented by the subtree with this node. Both are `NULL` in case these have not been computed so far.

`q_no` may hold the number of states required for an NFA that accepts the subtree language.

`init` may hold the single initial state of the NFA for the subtree language.

Note that the descriptors `son_passed, is_reg_ab, da, na, q_no` and `init` are not necessary for representing the regular expression, but are used by some of the functions as auxiliary storage.

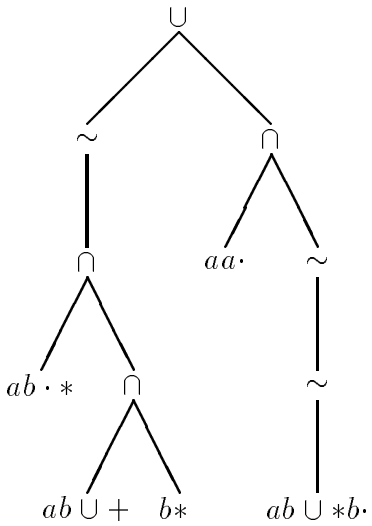


Figure 8.3: `postfix2tree()`

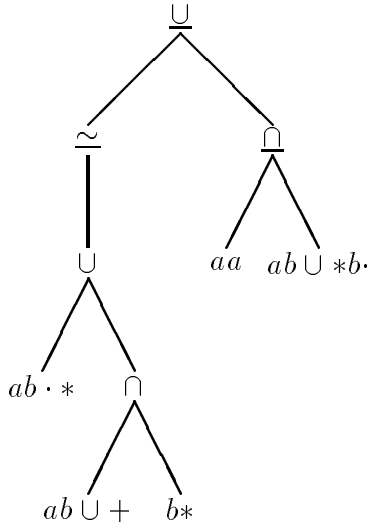


Figure 8.4: `optimize_tree()`, `mark_tree()`

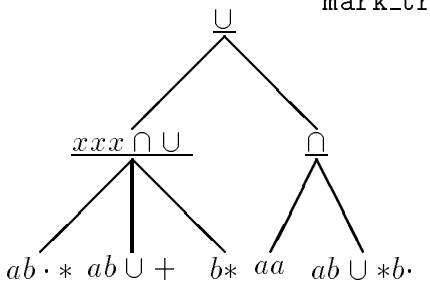


Figure 8.5: `reorder_tree()`

8.2.3 The Functions in genrex2nfa

- `nfa genrex2nfa(regex)` receives a pointer to a regular expression in postfix notation and returns a pointer to an NFA.

- `t_elem postfix2tree(regex)` receives a pointer to a regular expression in postfix notation and returns a pointer to the root of the tree that will be constructed. This root is an auxiliary node labelled with a blank.

- ```
  void  optimize_tree(t_elem)
        mark_tree(t_elem)
        reorder_tree(t_elem)
        first_run(t_elem)
  ```

  receive a pointer to the root of the tree, which is afterwards modified.

- `second_run(t_elem,nfa)` receives a pointer to the root of the tree and a pointer to an NFA, which must have been initialized properly with the
  ```

number of states required to deal with the nodes that have not been treated in `first_run()`.

Consider, for example, the following generalized regular expression:

$$\sim((ab)*\cup((a\cup b)^{+}\cap b^{*}))\cup(aa\cap\sim\sim((a\cup b)^{*}b))$$

In the Figures 8.3, 8.4 and 8.5 you can see how the functions mentioned above modify the tree stepwise. Note that in Figures 8.4 and 8.5 the underlined node labels indicate that the corresponding node has been marked with the `is_reg_ab` flag.

After these modifications, the function `first_run()` will compute DFA's for the nodes that are not underlined and at the same time count the states that will be needed to build NFA's for the underlined nodes. This is finally done by `second_run()`.

Chapter 9

From NFA to DFA

The well–known powerset construction from [RS59] shows that for any NFA there is an equivalent DFA. AMoRE's implementation of this construction is described in this section.

9.1 The Formal Method

The main idea of the construction is to consider sets of states of the NFA as states of the DFA. The DFA uses these sets to remember the set of states that the NFA could have reached after reading the same input.

We start with the definition of the powerset automaton for a given NFA. Recall that for an NFA $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ we let always $\delta : Q \times \Sigma \longrightarrow 2^Q$, $(q, a) \mapsto \{p \mid (q, a, p) \in \Delta\}$.The extension of this function to $2^Q \times \Sigma \longrightarrow 2^Q$ will be denoted by δ as well.

Definition/Lemma 9.1 Let $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ be an NFA. The *powerset automaton* $\mathcal{P}(\mathcal{A})$ of $\mathcal{A}$ is the DFA $(\Sigma, Q^{\mathcal{P}}, q_0^{\mathcal{P}}, \delta^{\mathcal{P}}, F^{\mathcal{P}})$, where
$Q^{\mathcal{P}} := \{\delta(I, w) \mid w \in \Sigma^*\}$,
$\delta^{\mathcal{P}} : Q^{\mathcal{P}} \times \Sigma \longrightarrow Q^{\mathcal{P}}$, $(P, w) \mapsto \delta(P, w)$,
$F^{\mathcal{P}} := \{P \in Q^{\mathcal{P}} \mid P \cap F \neq \emptyset\}$.
Then $\mathcal{P}(\mathcal{A})$ is always well defined and we have $L(\mathcal{P}(\mathcal{A})) = L(\mathcal{A})$.

For the correctness of this Lemma we remark that a simple induction shows that for all $w \in \Sigma^*$ we have $\delta(I, w) = \delta^{\mathcal{P}}(I, w)$.

Note that for an NFA $\mathcal{A}$, $\mathcal{P}(\mathcal{A})$ has only reachable states. This is not essential; one could also use 2^Q as the set of states of the DFA instead of $Q^{\mathcal{P}}$. For the actual implementation, however, the implicit restriction to reachable states is reasonable and more efficient.

The algorithm of Figure 9.1 corresponds to the construction in Lemma 9.1. For the correctness we remark that the following condition is an invariant for the outermost while-loop starting in line 3.

```
    {input : NFA (Σ, Q, I, Δ, F)}
(1) Q̄ := {I}; rest := {I}; q0 := I;
(2) if I ∩ F ≠ ∅ then F̄ := {I} else F̄ := ∅ fi;
(3) while rest ≠ ∅ do
(4)    select and delete any P1 ∈ rest;
(5)    for all a ∈ Σ
(6)       P2 := ∅;
(7)       for all q1 ∈ P1
(8)          P2 := P2 ∪ δ(q1, a);
(9)       endfor
(10)      if P2 ∉ Q̄ then
(11)         rest := rest ∪ {P2};
(12)         Q̄ := Q̄ ∪ {P2}
(13)         if F ∩ P2 ≠ ∅ then F̄ := F̄ ∪ {P2} fi
(14)      fi
(15)      δ̄(P1, a) := P2
(16)   endfor
(17) endwhile
    { output : DFA (Σ, Q̄, q̄0, δ̄, F̄) }
```

Figure 9.1: The Powerset Construction

$$(*)\ \overline{Q} \cup \{\delta(P,w) \mid P \in rest\} = Q^{\mathcal{P}} \text{ and } \overline{\delta} = \delta^{\mathcal{P}}\big|_{Q^{\mathcal{P}} \setminus rest} \text{ and } \overline{F} = F^{\mathcal{P}} \cap Q^{\mathcal{P}}$$

The dealing with sets is of course the main problem for the implementation. In AMoRE, the sets of NFA-states are represented by bitmaps.

The set $\overline{Q}$ is represented by a balanced tree. These are binary search trees with the property that in every node the difference of maximal lengths of paths in the subtrees starting in the successors must be at most one. The ordering on 2^Q required for this search tree is the one obtained from regarding the bitmaps as binary representation of numbers. For a detailed explanation of balanced trees see [Knu73].

For the representation of *rest* a stack is used. The components of both the tree and the stack are pointers to structures that contain all the information for one DFA-state.

For the complexity we note that there are actually examples in which the outermost loop is performed 2^n times, where $n := |Q|$. For each of these DFA-states and for each letter, the δ-image has to be computed and searched for in the balanced tree. Since the path length in this tree is bounded by n and each comparison needs at most n steps, this leads to a worst case complexity of $\mathcal{O}(2^n \cdot n^2 \cdot |\Sigma|)$.

9.2 The Implementation

9.2.1 Data Structures

The balanced tree is built up by structures of type `*treenode`. Each node represents one state in the powerset automaton $\mathcal{P}(\mathcal{A})$. Let $P \in Q^{\mathcal{P}}$ be a state of $\mathcal{P}(\mathcal{A})$. Such a structure contains the following descriptors:

`dfastate` – a char array that keeps P as a bitmap.

`number` – 0 if $P = \emptyset$, otherwise one more than the number P as a state in the output DFA. That is, if P will become the DFA-state with index `i`, then `number == i+1`.

`delta` – an array that keeps for every letter a the number of $\delta^{\mathcal{P}}(P, a)$ (if already computed).

`final` – a boolean flag that is `TRUE` iff P is final in $\mathcal{P}(\mathcal{A})$.

`bal` – a number that indicates how the tree is balanced in this node: if `bal == 0` then the maximal path lengths going left or right are equal, if `bal == 1` then the right subtree contains a path that is one node longer than the longest path of the left subtree, if `bal == 2` then the left subtree contains a path that is one node longer than the longest path of the right subtree.

`tr,tl` – pointers to the right and left son in the balanced tree.

The stack used for the implementation of *rest* is a chained list of structures of type `*stacknode`. Such a structure contains a descriptor `info` of type `treenode`, that is a pointer to the structure explained above, and a pointer to the next stack entry.

The global pointers `bottom` and `top` point to the first entry (last entry resp.) in this stack. Since in the run time the size of the stack can increase and decrease alternatingly, the memory that once has been used for the stack but is temporarily not occupied is kept in a second stack to which the global pointers `freestack` and `freebot` belong. Memory is only allocated when `freebot == freestack` and something is pushed onto the stack. The situation looks like visualized in Figure 9.2.

The operations push and pop on the stack are performed by the functions `push()` and `pop()`.

9.2.2 The Functions

There are two functions visible from outside: firstly, the function `nfa2dfa()`, which receives a pointer to an NFA and returns a pointer to the created DFA. Secondly, the function `modnfa2dfa()`, which additionally receives a pointer to an

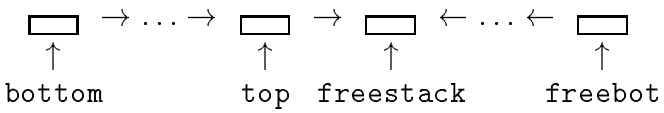
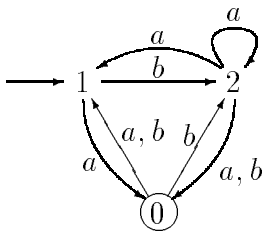


Figure 9.2: The Stack Convention



P	$\delta(P,a)$	$\delta(P,b)$
$\emptyset$	$\emptyset$	$\emptyset$
$\{0\}$	$\{1\}$	$\{1,2\}$
$\{1\}$	$\{0\}$	$\{2\}$
$\{2\}$	$\{0,1,2\}$	$\{0\}$
$\{0,1\}$	$\{0,1\}$	$\{1,2\}$
$\{0,2\}$	$\{0,1,2\}$	$\{0,1,2\}$
$\{1,2\}$	$\{0,1,2\}$	$\{0,2\}$
$\{0,1,2\}$	$\{0,1,2\}$	$\{0,1,2\}$

Figure 9.3: Example NFA

array `transformation` whose entries are bitmaps (= arrays of `char`). Thus this second parameter has the type `***char`. After `modnfa2dfa(na,&tra)` has been called (`tra` being a variable of type `**char`), `tra[q]` will be an array of `char` (of size `nb`) that is the bitmap representation of the set corresponding to the state with number `q` of the output DFA.

This modified version of the algorithm is needed only for the computation of the reduced NFA.

Both of these functions call the function `nnfa2dfa()`, which receives a pointer to an NFA and returns a pointer to the DFA. The global boolean variable `with` is set to `TRUE` only if this function is called from `modnfa2dfa()` and determines whether the transformation table is relevant after the computation. The memory needed for the output DFA and (if `with == TRUE`) for the transformation table

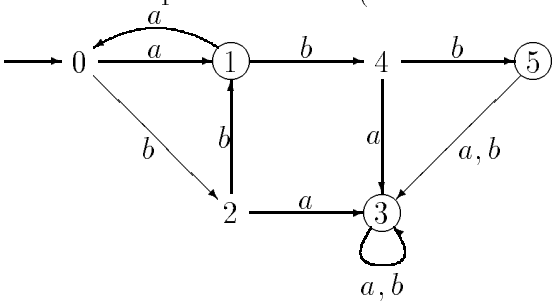


Figure 9.4: Resulting DFA

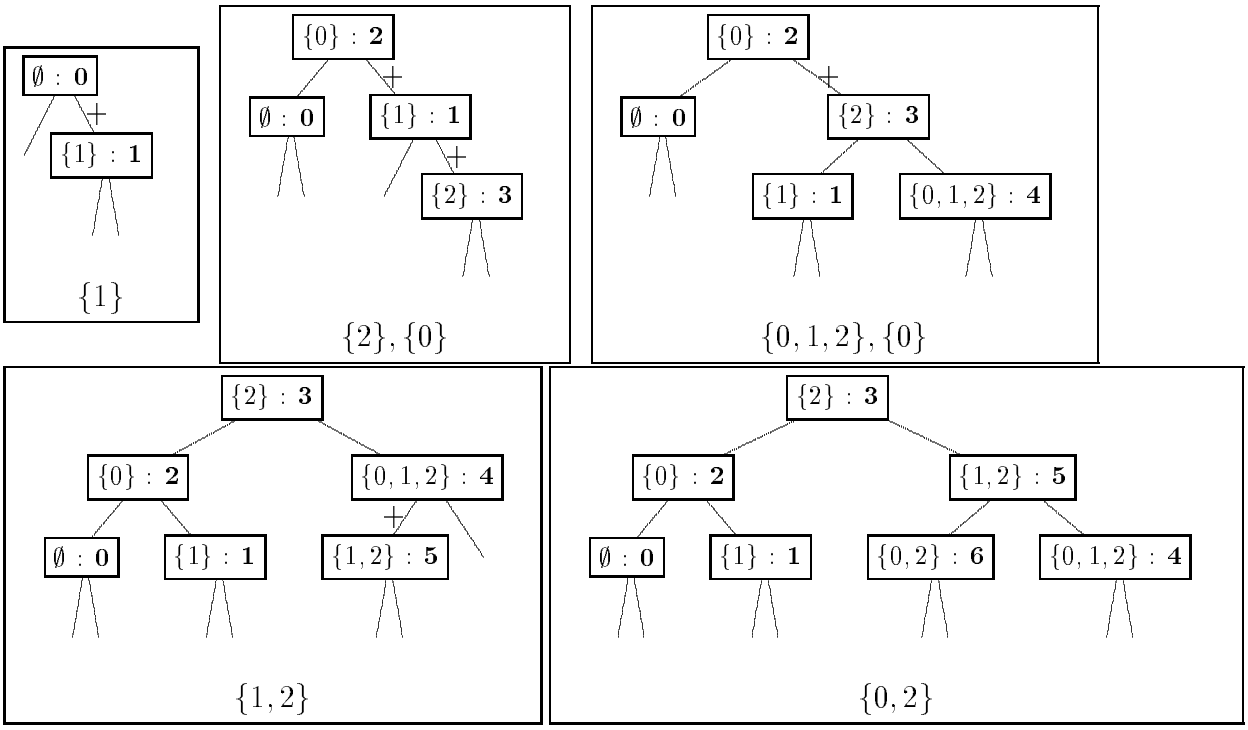


Figure 9.5: An Example Run

is allocated inside `nnfa2dfa()` as well.

The function `nnfa2dfa()` has three main steps: first, to initialize the DFA, second, to compute the new DFA-states and the DFA-transitions by creating the balanced tree, and third, to copy the information into the DFA data structure.

The second step is done by the function `compnewstates()`, and the third one is done by the function `cpinfo()`.

`compnewstates()` receives a pointer to the input NFA and returns a number, which is the highest used state number in the output DFA. Note that for technical reasons, there is another treenode with undefined values that is not meant to belong to $\overline{Q}$. It has no left successor, and its right successor is the root of the tree (the set of initial NFA-states).

`compnewstates()` works about the same way as the algorithm from Figure 9.1. One difference is that the balanced tree (corresponding to the set $\overline{Q}$) is initialized with $\{\emptyset, I\}$ instead of just $\{I\}$, so we actually have not $\mathit{rest} \subseteq \overline{Q}$ during the while-loop.

The function with the name `search()` is used to search for a certain set of NFA-states in the balanced tree. It inserts this set in case it has not been contained yet. It receives a `treenode` and returns `TRUE` iff the set `treenode->dfastate` has been in the balanced tree before. `search()` uses the auxiliary function `cmp()`, which compares two treenodes by the bitmap induced order on the NFA-state subsets.

The function `cpinfo()` mentioned above receives a pointer to the output DFA, for which memory has to be allocated already. It returns `TRUE` iff the empty set is reachable. `cpinfo()` investigates the balanced tree in a depth first search, reusing the stack. It copies the information of the tree into the DFA-structure and, in case `with == TRUE`, to the transformation table `new2old`. Note that before `cpinfo()` is called, `compnewstates()` has made sure that all the information for the transition function is contained in the treenodes' descriptor `delta`.

Chapter 10

From DFA to minimal DFA

10.1 Introduction

It is a natural question to ask for a DFA that accepts a certain regular language with a minimal number of states. It turns out that there is only one DFA (up to isomorphism) that matches this minimal size. It is called the *minimal DFA (MDFA)* and is of great importance for various algorithms and theorems in formal language theory.

The purpose of this section is to describe the algorithm of [Ho71, AHU74] that computes this MDFA from an arbitrarily given DFA with n states in time $\mathcal{O}(n \log n)$, and its implementation in AMoRE.

10.2 Formal Background

In the following, let $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ be a DFA accepting the language L. We start with the following definition.

Definition/Lemma 10.1 Consider the equivalence relation $\approx$ on Q defined by

$$q \approx p \text{iff} \forall w \in \Sigma^* : \delta(q, w) \in F \Leftrightarrow \delta(p, w) \in F$$

For $q \in Q$ the equivalence class of Q containing q is denoted by $[q]$. Then the *minimal DFA* for L is given by $\mathcal{MA} = (\Sigma, \overline{Q}, \overline{q_0}, \overline{\delta}, \overline{F})$, where

- $\overline{Q} = Q/\approx$
- $\overline{q_0} = [q_0]$
- $\overline{F} = \{[f] \,|\, f \in F\}$
- $\overline{\delta}([q], a) = [\delta(q, a)]$ for all $q \in Q$ and all $a \in \Sigma$

Note that δ is well–defined since for $p, q \in Q$ with $p \approx q$ one has $\forall a \in \Sigma : \delta(p,a) \approx \delta(q,a)$.

All DFA's that accept the same language as a given minimal DFA are either isomorphic to it or have more states.

So the computation of the MDFA from a given DFA boils down to determining the equivalence relation $\approx$ or rather its induced partition into equivalence classes. The following Definition and Lemma are helpful.

Definition 10.2 For all $a \in \Sigma$ let $\delta_a : Q \to Q, q \mapsto \delta(q,a)$.

Call a partition π of Q *compatible with* δ, iff

$$\forall C \in \pi \, \forall a \in \Sigma \, \exists B \in \pi : C \subseteq \delta_a^{-1}(B)$$

Note that this is exactly the criterion required for a partition induced by an equivalence relation to form a well–defined transition function like in Definition 10.1.

In fact, the following Lemma holds.

Lemma 10.3 The partition induced by $\approx$ is the coarsest partition compatible with δ that refines $\{F, Q \backslash F\}$.

10.3 About the Minimization Algorithm

Lemma 10.3 allows us to develop an interesting algorithm whose main idea is to refine the initial partition $\{F, Q \backslash F\}$ stepwise until it is compatible.

Let π be a partition of Q that we intend to make compatible by refinement. We call the elements of π *blocks*. Let B be such a block and $a \in \Sigma$.

We call a block $C \in \pi$ *compatible wrt. B and a* iff $C \cap \delta_a^{-1}(B) = \emptyset$ or $C \subseteq \delta_a^{-1}(B)$.

If there is a block C not compatible wrt. B and a, then this block C must not stay in the partition because it is not compatible with any refinement of π. So we split C up into C_1 and C_2 with $C_1 := \delta_a^{-1}(B) \cap C$ and $C_2 := C \backslash \delta_a^{-1}(B)$. Afterwards, both C_1 and C_2 are compatible wrt. B and a, and the new partition π obtained from the old one by $\pi \backslash \{C\} \cup \{C_1, C_2\}$.

We have to repeat this "splitting wrt. B and a" until every block is compatible wrt. every other block and every letter in Σ.

Once all blocks have been treated wrt. B and a (and have been split if necessary), we need not split wrt. B and a again until B itself is split. This is true because any subset of a block compatible wrt. B and a keeps that property.

The created new blocks C_1 and C_2 might make it necessary to split further blocks (wrt. C_i and some $a' \in \Sigma$), so we will have to use some mechanism to remember them.

(1) $\pi := \{F, Q\backslash F\}$
(2) for all $a \in \Sigma$ do
(3) if $|F| \leq |Q\backslash F|$ then
(4) $W(a) = \{F\}$
(5) else
(6) $W(a) = \{Q\backslash F\}$
(7) while $\bigcup_{a\in\Sigma} W(a) \neq \emptyset$ do
(8) select one $b \in \Sigma$ and delete any B from $W(b)$
(9) $INV := \delta_b^{-1}(B)$
(10) for each $C \in \pi$ not compatible wrt. B and b do
(11) $C_1 := C \cap INV$
(12) $C_2 := C\backslash INV$
(13) $\pi := \pi\backslash\{C\} \cup \{C_1, C_2\}$
(14) for all $a \in \Sigma$ do
(15) if $C \in W(a)$ then
(16) $W(a) := W(a)\backslash\{C\} \cup \{C_1, C_2\}$
(17) elseif $|C_1| < |C_2|$ then
(18) $W(a) := W(a) \cup \{C_1\}$
(19) else
(20) $W(a) := W(a) \cup \{C_2\}$

Figure 10.1: The Minimization Algorithm

If C had to be remembered before we split it, we may forget C and concentrate on C_1 and C_2 instead since any block that is compatible wrt. C_1 and C_2 and $a \in \Sigma$ is compatible wrt. $C = C_1 \cup C_2$ as well.

If C had not been one of the blocks to remember, this is due to the fact that all other blocks were compatible to C. We observe that for all blocks D compatible with C we have that D is compatible with C_1 iff D is compatible with C_2. Therefore if suffices to remember either C_1 or C_2 because splitting wrt. to one of the C_i will make sure that the resulting blocks are compatible with the other one as well. Of course, it is convenient to choose to remember the smaller of C_1 and C_2.

The algorithm described so far is shown in Figure 10.1. It works in time $\mathcal{O}(n \cdot t \cdot \log n)$ where n is the number of states of the input DFA and t is the size of the alphabet. This and the correctness of the algorithm is proved in [Ho71].

10.4 Details about the Implementation

In this section we will examine the data structures and functions of the algorithm as implemented in AMoRE. The reader may verify that these data structures indeed enable the handling of blocks and partitions as quickly as maintained in the complexity analysis.

For the implementation it is useful to regard the partition rather as a sequence of blocks than as a set. This sequence is indexed from `0` to the variable `numberofblocks`. The index usually used for its items is `bno`.

Each block itself is represented by a chain of states. This doubly connected cyclic list is established by the arrays `startblock[]`, `nextinblock[]`, `previnblock[]` and `blocksize[]`. For each blocknumber `bno`, `startblock[bno]` is the first state in the block `#bno` and `blocksize[bno]` is the number of states in this block.

For each state `state`, `nextinblock[state]` (and `previnblock[state]`) are the next (resp. the previous) state that is in the same block as `state` (only in case there is such a state; otherwise the entry is the first state in the same block, i.e. `nextinblock[state] == startblock[inblock[state]]`).

These four arrays enable the tracing of blocks and adding or deleting states quickly. In order to be able to find out for a state which block it is in, an additional array `inblock[]` is used.

The array `jlist[]` is filled every time `INV` (corresponding to INV in Figure 10.1) is computed by the function `computeinverse()`. It keeps a list of all those blocknumbers that contain a state in `INV`. The length of this list is stored in `sizeofjlist`. `computeinverse()` also fills the array `sizeofintersection[]`, which holds for every blocknumber `bno` in that list the size of the intersection of `INV` and block #`bno`.

The family of sets $(W(a))_{a\in\Sigma}$ in Figure 10.1 is replaced by a cyclicly linked list of structures `*wlist`. These structures contain a letter and a blocknumber that corresponds to an element in W. Pointers to the beginning and end of this list are stored in `waiting` and `wlast`.

The array `inverse[]` has nothing to do with the set `INV` above, but it holds the transposed transition table of the automaton in order to enable a quick computation of `INV`. For a state `state1` and a letter `letter`, `state1` is in the array `inverse[letter][state2]` iff `d[letter][state1] = state2`.

10.5 The Functions

The function `dfamfa()` is the function visible from outside. It receives a pointer to the input DFA and a boolean parameter that indicates whether the memory of the input DFA may finally be deallocated. It returns the minimal DFA for the input DFA.

The main algorithm however is performed by the function `mindfa()`. It starts by deleting the non-reachable states with function `delstates()` if function `dfamdfa()` has been called with the second parameter set `FALSE`. Afterwards `mindfa()` treats two trivial cases: Firstly, it returns the current DFA if it has only one state (possibly due to the deletion of non-reachable states). The second trivial case is that in initializing all static variables, arrays etc. (with function `initstatic()`) it turns out that either all states are final or there are no final states. In this case `initstatic()` returns `TRUE` and the function `makenewautomaton()` is advised to create a simple one-state DFA with one (or no, resp.) final state, which is then returned.

Note that differing from the algorithm given in Figure (10.1) the initialization for $W(a)$ is always $W(a) := \pi$, which costs only little more time.

The remainder of the function `mindfa()` implements exactly the lines (7) to (20) of Figure 10.1.

Note that the difference of the used data structures and their redundancy requires sometimes several statements for one step of the algorithm of Figure 10.1. For example, line (8) is implemented by dequeuing a blocknumber and a letter from the cyclic chain `WAITING` and additionally setting the corresponding flag in the array `inwaiting[]` to `FALSE`.

For another example consider the statement $\pi := \pi \backslash \{C\} \cup \{C_1, C_2\}$ in line (13). To achieve this effect by dealing only with blocknumbers instead of sets, function `split()` creates a new block by incrementing `blocknumber` and removes states of C (here the bi-directional chain is helpful) and adds them to the new block.

The function `putinwaiting()` receives a letter `letter` and a blocknumber `bno` and adds this pair to the chain `WAITING` representing the family of sets $(W(a))_{a \in \Sigma}$.

The function `makenewautomaton()` finally creates a new automaton out of the partition of the set of states of the input DFA and its transition table. The states of the resulting DFA correspond to the blocknumbers in the natural way. For the transition table the function traces all blocknumbers and picks one representative out of each. For each transition leaving this representative one transition to the block containing the target is inserted.

Figure 10.2 shows an example for a DFA and how the algorithm runs on it. Figure 10.3 shows the result.

	a	b
q0	q6	q7
*q1	q4	q5
q2	q6	q7
q3	q4	q8
q4	q3	q8
Q5	q8	q0
Q6	q9	q2
Q7	q8	q0
Q8	q9	q8
Q9	q8	q8

WAITING	Partition
0a 1a 0b 1b	$[5,6,7,8,9][0,1,2,3,4]$
split wrt. $[5,6,7,8,9]$ and letter a	
1a 0b 1b 2a 2b 0b 1b 2a 2b	$[5,6,7,8,9][1,3,4][2,0]$
split wrt. $[5,6,7,8,9]$ and letter b	
1b 2a 2b 3a 3b 2a 2b 3a 3b 2b 3a 3b 3a 3b 3b	$[5,6,7][1,3,4][2,0][9,8]$
split wrt. $[9,8]$ and letter b	
1a 1b 1b	$[5,6,7][1][2,0][9,8][4,3]$

Figure 10.2: An example DFA and how the algorithms works on it

	a	b
Q0	q3	q2
*q1	q4	q0
q2	q0	q0
Q3	q3	q3
q4	q4	q3

Figure 10.3: The resulting MDFA

Chapter 11

From DFA to Reduced NFA

In this section we describe a heuristic algorithm to reduce NFA's wrt. the number of states. We only give an excerpt of a paper dealing with the theoretical background of the implemented algorithm. For the complete paper see [MP95].

11.1 Introduction

Minimization of nondeterministic finite automata ("NFA") is more difficult than that of deterministic finite automata. The problem is PSPACE-complete (except for the case of one-letter-alphabets) [GJ79], whereas deterministic finite automata can be minimized in time $O(n \cdot \log n)$ [Ho71]. Moreover, there is in general not a unique minimal NFA recognizing a given regular language. Procedures for minimization of nondeterministic finite automata are investigated in several papers, e.g. Kameda and Weiner [KW70] (see also Indermark [In70]) and Kim [Ki74]. Further references are given in Brauer's monograph [Br84].
We describe a general technique of constructing minimal NFA which is implicit in older papers like [KW70] and has been suggested in a different version in a recent contribution of Arnold, Dicky and Nivat [ADN92]. The idea is to construct a nondeterministic "canonical automaton" in which any NFA recognizing the considered language occurs (via a homomorphism) as a subautomaton. This means that the construction of the minimal NFA can be done in two steps: construction of the canonical automaton and search for a minimal subautomaton therein which accepts the same language.
The second issue is the search of a minimal NFA within a given (canonical) automaton. We outline a heuristic which improves the one of [Ki74]. It yields an algorithm with worst-case-behaviour time $\mathcal{O}(2^{(n^2)})$, where n is the number of states of the minimal deterministic automaton of the considered language. The algorithm is practical in the sense that in many non-trivial examples the actual implementation works in acceptable time. However, we show that the heuristic fails in some cases to produce a minimal NFA. It is open how to sharpen the

heuristic in order to get an algorithm which always outputs a proper minimal NFA and is as "practical" as our heuristic.
An implementation of the heuristic is part of AMoRE. For a more detailed analysis see [MP95].

11.2 Background

We assign two languages to every state of an NFA.

Definition 11.1 ([KW70],[HU79])
Let $\mathcal{A}$ be an NFA and $q \in Q$. Then the *pre-language of* q and the *post-language of* q are given by $pre(\mathcal{A}, q) = L((Q_\mathcal{A}, \Sigma, I_\mathcal{A}, \delta_\mathcal{A}, \{q\}))$ and by $post(\mathcal{A}, q) = L((Q_\mathcal{A}, \Sigma, \{q\}, \delta_\mathcal{A}, F_\mathcal{A}))$, respectively. q is *reachable* if $pre(\mathcal{A}, q) \neq \emptyset$ and q is *productive* if q is reachable and $post(\mathcal{A}, q) \neq \emptyset$. Two states $p, q \in Q$ are *equivalent*, written $p \sim q$, if $post(\mathcal{A}, p) = post(\mathcal{A}, q)$. $[q] = \{\, p \,|\, p \sim q \,\}$ denotes the equivalence class to which q belongs.

In [ADN92], pre- and post-language are called *history* and *prophecy*.
In the sequel all NFA have productive states only.
Now we introduce three classical operations on automata and analyse pre- and post-languages of the resulting automata.

Definition 11.2 ([KW70])
Let $\mathcal{A}$ be an NFA. Its *reversed NFA* is $\overline{\mathcal{A}} = (Q, \Sigma, F, \overline{\delta}, I)$ with $\forall a \in \Sigma\ \forall p, q \in Q$: $p \in \overline{\delta}(q, a) \Longleftrightarrow q \in \delta(p, a)$.
Given a word $u = a_1 \ldots a_n$ the word $\overline{u} = a_n \ldots a_1$ is its *reversed word*. Given a language L the language $\overline{L} = \{\, \overline{u} \,|\, u \in L \,\}$ is its *reversed language*.

Obviously $L(\overline{\mathcal{A}}) = \bigcup_{q \in F} post(\overline{\mathcal{A}}, q) = \overline{\bigcup_{q \in F} pre(\mathcal{A}, q)} = \overline{L(\mathcal{A})}$ for all NFA $\mathcal{A}$. Furthermore $\mathcal{A}$ is a minimal NFA accepting L iff $\overline{\mathcal{A}}$ is a minimal NFA accepting $\overline{L}$.
The next operation is the wellknown powerset construction.

Definition 11.3 ([RS59])
Let $\mathcal{A}$ be an NFA. The *subset automaton for* $\mathcal{A}$, denoted $D(\mathcal{A})$, is the DFA $\mathcal{B} = (Q_\mathcal{B}, \Sigma, I_\mathcal{B}, \delta_\mathcal{B}, F_\mathcal{B})$, where $Q_\mathcal{B} = \{\, \delta_\mathcal{A}(I_\mathcal{A}, u) \,|\, u \in \Sigma^* \,\} \subseteq 2^Q$, $I_\mathcal{B} = \{\, I_\mathcal{A} \,\}$, $F_\mathcal{B} = \{\, P \in Q_\mathcal{B} \,|\, P \cap F_\mathcal{A} \neq \emptyset \,\}$ and with $\delta_\mathcal{B}(P, a) = \delta_\mathcal{A}(P, a)$ for all $P \in Q_\mathcal{B}$ and for all $a \in \Sigma$.

Obviously $L(\mathcal{A}) = \bigcup_{q \in I} post(\mathcal{A}, q) = post(\mathcal{B}, I_\mathcal{B}) = L(D(\mathcal{A}))$.
Next we give the constructive definition of the minimal DFA for a given NFA.

Definition/Lemma 11.4 *([HU79],[KW70])*
Let $\mathcal{A}$ be an NFA and $\mathcal{B} = D(\mathcal{A})$. The *minimal DFA* $\mathcal{D} = (Q_\mathcal{D}, \Sigma, I_\mathcal{D}, \delta_\mathcal{D}, F_\mathcal{D})$ *for* $\mathcal{A}$, denoted $M(\mathcal{B})$ or $MD(\mathcal{A})$, can be constructed the following way: $Q_\mathcal{D} = \{[\,P\,]|P \in Q_\mathcal{B}\}$, $I_\mathcal{D} = [I_\mathcal{B}]$, $F_\mathcal{D} = \{[\,P\,]|P \in F_\mathcal{B}\}$ and for all states P of $Q_\mathcal{B}$ and for all $a \in \Sigma$, $\delta_\mathcal{D}([P], a) = [\delta_\mathcal{B}(P, a)]$.
Let q be a state of $\mathcal{A}$ and $[P]$ be a state of $MD(\mathcal{A})$. We write $q < [P]$ as an abbreviation for: $\exists P' \in [P]\ q \in P'$. We give some relations that hold between post- and pre-languages in $\mathcal{A}$ and $\mathcal{D}$.

$$q \in F \wedge q < [P] \quad \Rightarrow \quad [P] \in F_\mathcal{D} \tag{11.1}$$

$$q \in I \quad \Rightarrow \quad q < [P] \text{ where [P] is the initial state of } \mathcal{D} \tag{11.2}$$

$$p \xrightarrow[\mathcal{A}]{a} q \wedge p < [P] \quad \Rightarrow \quad q < \delta_\mathcal{D}([P], a) \tag{11.3}$$

Remark 11.5
For every regular language L, there is a unique (up to isomorphism) minimal DFA $\mathcal{D}$ with $L(\mathcal{D}) = L$, which we call "the" minimal DFA for a language L.
The analogue is not true for minimal NFA since there are regular languages for which several non-isomorphic minimal NFA accepting these languages exist.
All states of a minimal DFA are reachable and at most one state can be non-productive. If it exists, this state is called the sink state or — because its post-language is the empty set — the empty state. Pre-languages of different states of a DFA are disjoint and post-languages of different states in a minimal DFA are different. The minimal DFA $MD(\mathcal{A})$ for an NFA $\mathcal{A}$ with n states can have up to 2^n states.

The next theorem is interesting because it offers a way to compute the minimal DFA for a given language without computing the equivalence classes of states (as it is suggested by 11.4).

Theorem 11.6 (Brzozowski [Brz63])
Let L be a regular language and $\mathcal{D}$ an arbitrary DFA accepting L. Then $\mathcal{E} := D(\overline{\mathcal{D}})$ is the minimal DFA accepting $\overline{L}$.

Definition 11.7 Let $\mathcal{A} = (Q, \Sigma, I, \delta, F)$ and $\mathcal{B}$ be nondeterministic automata. A mapping $h : Q \to Q_\mathcal{B}$ is called a *morphism* if $h(I) \subseteq I_\mathcal{B}$, $h(F) \subseteq F_\mathcal{B}$ and $h(\delta(q, a)) \subseteq \delta_\mathcal{B}(h(q), a)$ for all $q \in Q$ and $a \in \Sigma$.
$\mathcal{A}$ is called a *subautomaton* of $\mathcal{B}$ if $Q \subseteq Q_\mathcal{B}$, $F = F_\mathcal{B} \cap Q$, $I = I_\mathcal{B} \cap Q$ and $\delta(q, a) \subseteq \delta_\mathcal{B}(q, a) \cap Q$ for all $q \in Q$ and $a \in \Sigma$. If $\delta(q, a) \subseteq \delta_\mathcal{B}(q, a) \cap Q$, we call $\mathcal{A}$ the *subautomaton of* $\mathcal{B}$ *induced by* Q.
If h is a morphism from $\mathcal{A}$ to $\mathcal{B}$ then we denote by $h(\mathcal{A})$ the subautomaton of $\mathcal{B}$ induced by the set $h(Q)$ of states.

Remark 11.8 Let $\mathcal{A}$ and $\mathcal{B}$ nondeterministic automata and h a morphism from $\mathcal{A}$ to $\mathcal{B}$. Then $L(\mathcal{A}) \subseteq L(h(\mathcal{A})) \subseteq L(\mathcal{B})$.

Proof We have $p \xrightarrow[\mathcal{A}]{w} q \Rightarrow h(p) \xrightarrow[h(\mathcal{A})]{w} h(q) \Rightarrow h(p) \xrightarrow[\mathcal{B}]{w} h(q)$ and $p \in F(I) \Rightarrow h(p) \in F_{\mathcal{B}}(I_{\mathcal{B}})$. Thus an accepting path in $\mathcal{A}$ is mapped to an accepting path in $h(\mathcal{A})$ which is an accepting path of $\mathcal{B}$ by definition. □

11.3 The Fundamental Automaton

In this section we construct for every regular language L an NFA $\mathcal{F}$, called the fundamental automaton of L, such that for every NFA $\mathcal{A}$ accepting L there exists a morphism from $\mathcal{A}$ in $\mathcal{F}$. We will use this fact to show that a minimal automaton accepting L can be found as a subautomaton of $\mathcal{F}$.
In order to simplify the notation we fix the language L. Let $\mathcal{A} = (Q, \Sigma, I, \delta, F)$ be an arbitrary NFA accepting L, $\mathcal{D} = (Q_{\mathcal{D}}, \Sigma, I_{\mathcal{D}}, \delta_{\mathcal{D}}, F_{\mathcal{D}})$ the minimal DFA accepting L and $\mathcal{E} = (Q_{\mathcal{E}}, \Sigma, I_{\mathcal{E}}, \delta_{\mathcal{E}}, F_{\mathcal{E}}) = D(\overline{\mathcal{D}})$. (Because of 11.6, $\mathcal{E}$ is the minimal DFA accepting L.) If $\mathcal{E}$ has a sink state then this state is not productive in the reversed automaton $\overline{\mathcal{E}} = (Q_{\overline{\mathcal{E}}}, \Sigma, I_{\overline{\mathcal{E}}}, \delta_{\overline{\mathcal{E}}}, F_{\overline{\mathcal{E}}})$ and we assume that this state and all transitions from this state are deleted in $\overline{\mathcal{E}}$.
$\overline{\mathcal{E}}$ has several important properties described in the following remark.

Remark 11.9
$\overline{\mathcal{E}}$ is a reversed automaton to a deterministic finite automaton. Therefore there exists for all words in L a unique accepting path in $\overline{\mathcal{E}}$, and deletion of any transition in $\delta_{\overline{\mathcal{E}}}$ yields an automaton which accepts a proper subset of L. Moreover, a post-language of an arbitrary NFA accepting L is a subset of a disjoint union of post-languages of $\overline{\mathcal{E}}$. $\mathcal{E}$ is isomorphic to $MD(\overline{\mathcal{A}})$ and thus we can associate to every state r of $\mathcal{E}$ an equivalence class of sets of states of $\mathcal{A}$ (cf. Definition 11.4). For every state q of $\mathcal{A}$ we denote by $\overset{\bullet}{q}$ the set of states r of $\mathcal{E}$ with $q < r$.

Now we arrive at the definition of the fundamental automaton, given $\mathcal{E} = D(\overline{\mathcal{D}})$.

Definition 11.10 (The Fundamental Automaton)
The *fundamental automaton* $\mathcal{F} = (Q_{\mathcal{F}}, \Sigma, I_{\mathcal{F}}, \delta_{\mathcal{F}}, F_{\mathcal{F}})$ is defined as follows:

$$Q_{\mathcal{F}} = \{\, P \subseteq Q_{\overline{\mathcal{E}}} \mid \bigcap P \neq \emptyset \}$$
$$I_{\mathcal{F}} = \{\, P \in Q_{\mathcal{F}} \mid P \subseteq I_{\overline{\mathcal{E}}} \}$$
$$F_{\mathcal{F}} = \{\, P \in Q_{\mathcal{F}} \mid P \cap F_{\overline{\mathcal{E}}} \neq \emptyset \}$$
$$\delta_{\mathcal{F}}(P, a) = \{\, P' \in Q_{\mathcal{F}} \mid P' \subseteq \delta_{\overline{\mathcal{E}}}(P, a) \} \text{ for all } P \text{ in } Q_{\mathcal{F}} \text{ and } a \text{ in } \Sigma.$$

We proceed in three steps. First, we show that the fundamental automaton accepts L. Then we verify that for each automaton $\mathcal{A}$ accepting L there exists a morphism from $\mathcal{A}$ into the fundamental automaton of L. This proves that a minimal automaton accepting L can be found as a subautomaton of $\mathcal{F}$. We conclude with a condition that describes how to find certain subautomata of $\mathcal{F}$ that accept L.

Lemma 11.11 The fundamental automaton $\mathcal{F}$ is equivalent to $\overline{\mathcal{E}}$.

Proof $L(\mathcal{F}) \supseteq L(\overline{\mathcal{E}})$ because $\overline{\mathcal{E}}$ is a subautomaton of $\mathcal{F}$ (take all subsets of $Q_{\mathcal{F}}$ of size 1). The following fact on the transition function of $\mathcal{F}$ can be shown easily by an induction on the length of words w:

$$\forall P, P' \in Q_{\mathcal{F}}\ \forall w \in \Sigma^*\ P \xrightarrow[\mathcal{F}]{w} P'\ \Rightarrow\ \forall p' \in P'\ \exists p \in P\ p \xrightarrow[\overline{\mathcal{E}}]{w} p' \qquad (11.4)$$

Then the definition of initial and final states of $\mathcal{F}$ yields: if $P \xrightarrow[\mathcal{F}]{w} P'$ is an accepting path in $\mathcal{F}$ then there exists a state $p' \in P' \cap F_{\overline{\mathcal{E}}}$ and a state $p \in P \subseteq I_{\overline{\mathcal{E}}}$ with $p \xrightarrow[\overline{\mathcal{E}}]{w} p'$. Thus $L(\mathcal{F}) \subseteq L(\overline{\mathcal{E}})$. □

Lemma 11.12
Let $\mathcal{A}$ be an NFA accepting L and $\mathcal{F}$ the fundamental automaton of L. The mapping $q \mapsto \overset{\bullet}{q}$ is a morphism from $\mathcal{A}$ to $\mathcal{F}$.

Proof
Because of Theorem 11.6, we may regard $\mathcal{E}$ as $D(\overline{D(\mathcal{A})})$. The states of $\mathcal{E}$ are thus sets of states of $\overline{D(\mathcal{A})}$. For a state q' of $\overline{D(\mathcal{A})}$) and a state r of $\mathcal{E}$, we have then $q' < r$ iff $q' \in r$.
We will begin by showing that $q \mapsto \overset{\bullet}{q}$ is a mapping from $Q_{\mathcal{A}}$ in $Q_{\mathcal{F}}$.

Let q be a state of $\mathcal{A}$ and $r \in \overset{\bullet}{q}$. Since all states of $\mathcal{A}$ are supposed to be productive, there is a state q' of $D(\mathcal{A})$ such that $q \in q'$. Since $q < r$ and $\mathcal{E}$ is isomorphic to $MD(\overline{\mathcal{A}})$, there is a state R of $D(\overline{\mathcal{A}})$ such that $q \in R$ and $post(D(\overline{\mathcal{A}}), R) = post(\mathcal{E}, r)$. (Each state of $\mathcal{E}$ corresponds to a non-empty set of states of $D(\overline{\mathcal{A}})$ whose elements have the same post-language.)
Now we have
$post(\overline{D(\mathcal{A})}, q') = pre(D(\mathcal{A}), q') \subseteq pre(\mathcal{A}, q) = post(\overline{\mathcal{A}}, q) \subseteq \bigcup_{p \in R} post(\overline{\mathcal{A}}, p)$
$= post(D(\overline{\mathcal{A}}), R) = post(\mathcal{E}, r) = post(D(\overline{D(\mathcal{A})}), r) = \bigcup_{p' \in r} post(\overline{D(\mathcal{A})}, p')$
Since different pre-languages of a DFA are disjoint, so are different post-language of $\overline{D(\mathcal{A})}$. Thus we may conclude that $q' \in r$, i.e. $r \in \overset{\bullet}{q}{}'$. Since r was chosen arbitrarily from $\overset{\bullet}{q}$, we have shown $\overset{\bullet}{q} \subseteq \overset{\bullet}{q}{}'$.

We have $\overset{\bullet}{q}' = \{r \in Q_{\mathcal{E}} \mid q' \in r\}$, thus $q' \in \bigcap \overset{\bullet}{q}' \subseteq \bigcap \overset{\bullet}{q}$, showing that $\overset{\bullet}{q}$ is a state of $\mathcal{F}$. So we have shown that $q \mapsto \overset{\bullet}{q}$ is a mapping into $\mathcal{F}$.

To show that it is a morphism, we have to verify $q \in F(I) \Rightarrow \overset{\bullet}{q} \in F_{\mathcal{F}}(I_{\mathcal{F}})$ and $p \xrightarrow[\mathcal{A}]{a} q \Rightarrow \overset{\bullet}{p} \xrightarrow[\mathcal{F}]{a} \overset{\bullet}{q}$.

$$
\begin{array}{lll}
q \in I_{\mathcal{A}} & q \in F_{\mathcal{A}} & p \xrightarrow[\mathcal{A}]{a} q \\
& \Downarrow & \text{Definition of the reversed automaton} \\
q \in F_{\overline{\mathcal{A}}} & q \in I_{\overline{\mathcal{A}}} & q \xrightarrow[\overline{\mathcal{A}}]{a} p \\
& \Downarrow & \text{Equations (11.1,11.2,11.3)} \\
\overset{\bullet}{q} \subseteq F_{\mathcal{E}} & \overset{\bullet}{q} \cap I_{\mathcal{E}} \neq \emptyset & \overset{\bullet}{p} \supseteq \delta_{\mathcal{E}}(\overset{\bullet}{q}, a) \\
& \Downarrow & \text{Definition of the reversed automaton} \\
\overset{\bullet}{q} \subseteq I_{\overline{\mathcal{E}}} & \overset{\bullet}{q} \cap F_{\overline{\mathcal{E}}} \neq \emptyset & \overset{\bullet}{q} \subseteq \delta_{\overline{\mathcal{E}}}(\overset{\bullet}{p}, a) \\
& \Downarrow & \text{Definition of the fundamental automaton} \\
\overset{\bullet}{q} \in I_{\mathcal{F}} & \overset{\bullet}{q} \in F_{\mathcal{F}} & \overset{\bullet}{p} \xrightarrow[\mathcal{F}]{a} \overset{\bullet}{q}
\end{array}
$$

□

Theorem 11.13 A minimal automaton accepting L can be found as a subautomaton of $\mathcal{F}$.

Proof Let $\mathcal{A}$ be a minimal NFA accepting L and let $h : q \mapsto \overset{\bullet}{q}$ be the morphism of Lemma 11.12. The number of states of $h(\mathcal{A})$ is less or equal to the number of states of $\mathcal{A}$. Furthermore we have: $L = L(\mathcal{A}) \subseteq L(h(\mathcal{A})) \subseteq L(\mathcal{F}) = L$. Thus $L(h(\mathcal{A})) = L$ and $h(\mathcal{A})$ is a minimal NFA accepting L and a subautomaton of $\mathcal{F}$. □

With the next lemma we give a criterion for equivalent subautomata of $\mathcal{F}$ that improves the one given by Kim [Ki74] and was suggested by Kahlert [Ka91]. Kim claimed that his criterion was necessary, which is not true in general.

We start with an auxiliary definition. Let Q be a set, $\mathcal{P} \subseteq 2^Q$ and $R \subset Q$. We say $\mathcal{P}$ *covers* R *with subsets* iff $R = \bigcup\{P \in \mathcal{P} \mid P \subseteq R\}$.

Lemma 11.14
Let $\mathcal{F}_{\mathcal{P}}$ be a subautomaton of $\mathcal{F}$ induced by the set $\mathcal{P} \subseteq Q_{\mathcal{F}}$ of states. Assume

(i) $\mathcal{P}$ covers $I_{\overline{\mathcal{E}}}$ with subsets and

(ii) $\mathcal{P}$ covers $\delta_{\overline{\mathcal{E}}}(P, a)$ with subsets for all $P \in \mathcal{P}$, $a \in \Sigma$.

Then $\mathcal{F}_{\mathcal{P}}$ is equivalent to $\mathcal{F}$.

Proof Let $\mathcal{P}$ fulfill conditions (i) and (ii) and let $p_0 \xrightarrow[\overline{\mathcal{E}}]{a_1} \dots \xrightarrow[\overline{\mathcal{E}}]{a_n} p_n$ be an accepting path in $\overline{\mathcal{E}}$. We show that there exists an accepting path $P_0 \xrightarrow[\mathcal{F}_\mathcal{P}]{a_1} \dots \xrightarrow[\mathcal{F}_\mathcal{P}]{a_n} P_n$ in $\mathcal{F}_\mathcal{P}$ with $p_i \in P_i$. By $p_0 \in I_{\overline{\mathcal{E}}}$ and condition (i) there is a set $P_0 \in \mathcal{P}$ with $P_0 \subseteq I_{\overline{\mathcal{E}}}$ and $p_0 \in P_0$. Furthermore P_0 is initial in $\mathcal{F}_\mathcal{P}$. We find $P_1 \dots P_n$ by induction in the following way: if $p_i \in P_i$ then $p_{i+1} \in \delta_{\overline{\mathcal{E}}}(P_i, a_{i+1})$; by condition (ii) there exists $P_{i+1} \subseteq \delta_{\overline{\mathcal{E}}}(P_i, a_{i+1})$, $P_{i+1} \in \mathcal{P}$ with $p_{i+1} \in P_{i+1}$. By definition of $\mathcal{F}$ we get $P_i \xrightarrow[\mathcal{F}]{a_{i+1}} P_{i+1}$. P_n is final in $\mathcal{F}$ because $p_n \in P_n$ and p_n is final in $\overline{\mathcal{E}}$. Thus $L(\overline{\mathcal{E}}) \subseteq L(\mathcal{F}_\mathcal{P})$. □

11.4 An Example

In this section we present a simple example that shows that the criterion given in 11.14 is not necessary for subautomata of the fundamental automaton which are equivalent to the given NFA. Thus it does not yield an algorithm for determining a minimal NFA in general.

Consider the NFA $\mathcal{A}$ and its reversed NFA $\overline{\mathcal{A}}$ given by the following transition tables. Initial and final states are marked with "i" or "f", respectively.

$\mathcal{A}$

	a	b
f0	0, 2	
1	3	2, 3
i2	1	0, 1
3	3	1

$\overline{\mathcal{A}}$

	a	b
i0	0	2
1	2	2, 3
f2	0	1
3	1, 3	1

The following tables show the minimal DFA $\mathcal{E}$ equivalent to $\overline{\mathcal{A}}$, the reverse of this DFA and the mapping $q \mapsto \overset{\bullet}{q}$ from $Q_\mathcal{A}$ into $2^{Q_\mathcal{E}}$. The leftmost column of the first table contains for each state of $\mathcal{E}$ the corresponding states of $\overline{\mathcal{A}}$.

		a	b
$\{0\}$	ie_0	e_0	e_1
$\{2\}$	fe_1	e_0	e_2
$\{1\}$	e_2	e_1	e_3
$\{2,3\}$	fe_3	e_4	e_2
$\{0,1,3\}$	e_4	e_5	e_5
$\{0,1,2,3\}, \{1,2,3\}$	fe_5	e_5	e_5

	a	b
fe_0	e_0, e_1	
ie_1	e_2	e_0
e_2		e_1, e_3
ie_3		e_2
e_4	e_3	
ie_5	e_4, e_5	e_4, e_5

$h(0) = \{e_0, e_4, e_5\}$
$h(1) = \{e_2, e_4, e_5\}$
$h(2) = \{e_1, e_3, e_5\}$
$h(3) = \{e_3, e_4, e_5\}$

The states of the fundamental automaton are certain subsets of $Q_\mathcal{E}$. In fact, a look at the correspondence between $\mathcal{E}$ and $D(\overline{D(\mathcal{A})})$, which is not presented here, would show that the fundamental automaton contains actually *all* subsets of $Q_\mathcal{E}$. That is why we do not present it here. But we show the transition table of the

image of $\mathcal{A}$ under the morphism h.

	a	b
f$\{e_0, e_4, e_5\}$	$\{e_0, e_1, e_3, e_4, e_5\}$	$\{e_4, e_5\}$
$\{e_2, e_4, e_5\}$	$\{e_3, e_4, e_5\}$	$\{e_1, e_3, e_4, e_5\}$
i$\{e_1, e_3, e_5\}$	$\{e_2, e_4, e_5\}$	$\{e_0, e_2, e_4, e_5\}$
$\{e_3, e_4, e_5\}$	$\{e_3, e_4, e_5\}$	$\{e_2, e_4, e_5\}$

This transition table has to be interpreted like this: if $Q \subseteq Q_{\mathcal{E}}$ is in the line corresponding to $P \subseteq Q_{\mathcal{E}}$ and letter a, then in the fundamental automaton there is a transition from P with a to each subset of Q. So the subautomaton of the fundamental automaton induced by the image of h is:

	a	b
f$h(0)$	$h(0), h(2), h(3)$	
$h(1)$	$h(3)$	$h(2), h(3)$
i$h(2)$	$h(1)$	$h(0), h(1)$
$h(3)$	$h(3)$	$h(1)$

The automaton is (up to isomorphism) the same as $\mathcal{A}$, except for one superfluous transition.
The chosen set of states $\mathcal{P} = \{h(0), h(1), h(2), h(3)\}$ of the fundamental automaton covers indeed the set of initial states of $\overline{\mathcal{E}}$ with subsets, so the first criterion of 11.14 holds. But the transition table shows that the set $\delta_{\overline{\mathcal{E}}}(\{e_0, e_4, e_5\}, b) = \{e_4, e_5\}$ is not covered with subsets; so the second property does not hold. Thus $h(\mathcal{A})$ is a subautomaton of the fundamental automaton that accepts L, but does not fulfill the conditions (i) and (ii) of Lemma 11.14.
In fact, there is no set of states with the property of Lemma 11.14 that has less than 5 states, so for this example language the suggested heuristic fails to find a minimal NFA.

Chapter 12

From DFA to Monoid

12.1 Introduction

Given some DFA, every letter induces some mapping on the set of states. These mappings form (together with the composition operation) a monoid. If the underlying DFA is the minimal DFA accepting a regular language, then this monoid is isomorphic to its syntactic monoid, which is of great importance for several theoretical results which establish certain correspondences between properties of languages and their syntactic monoids (see [Pi86]).

In this section AMoRE's algorithm for the computation of the syntactic monoid is described. It follows [Pi86], and its implementation can be found in the file `dfamon.c`.

12.2 Formal Background

We recall that the syntactic monoid of a regular language is given by $\Sigma^*/\cong_L$, where $\cong_L$ denotes the syntactic congruence wrt. L. (See Definition 1.1 in Subsection 1.3.3.)

The mentioned different characterization of the syntactic monoid and semigroup of a regular language is provided by the following Lemma:

Lemma 12.1 Let L be some regular language. Let $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ be the minimal DFA accepting L. For a word $w \in \Sigma^*$ let $\delta_w : Q \to Q$, $q \mapsto \delta(q, w)$.

The mapping $\Sigma^* \to Q^Q$, $w \mapsto \delta_w$ is a monoid homomorphism, that is, we have $\forall u, v \in \Sigma^* : \delta_u \delta_v = \delta_{uv}$.

$M := \{\delta_w \,|\, w \in \Sigma^*\}$ and $S := \{\delta_w \,|\, w \in \Sigma^+\}$ together with the composition forms a sub-monoid (a sub-semigroup resp.) of Q^Q isomorphic to $M(L)$ (to $S(L)$ resp.) via the isomorphism $\delta_w \mapsto [w]$.

Recall that $M(L)$, the syntactic monoid of L, is $\Sigma^*/\cong_L$, and $S(L)$, the syntactic semigroup of L, is $\Sigma^+/\cong_L$.

This Lemma enables the computation of the syntactic monoid of a language.

```
    {input: MDFA (Σ, Q, q0, δ, F)}
(1) M := id
(2) G := ∅
(3) for all a ∈ Σ do
(4)    if δ_a ∉ M then
(5)       M := M ∪ {δ_a}
(6)       G := G ∪ {a}
(7)    endif
(8) endfor
(9) W := G
(10) while W ≠ ∅ do
(11)   select and delete any w ∈ W
(12)   for all a ∈ G do
(13)      if δ_w δ_a ∉ M then
(14)         M := M ∪ {δ_w δ_a}
(15)         W := W ∪ {wa}
(16)      endif
(17)   endfor
(18) endwhile
    {output: M = {δ_w | w ∈ Σ}}
```

Figure 12.1: The Algorithm

For the correctness of the algorithm of Figure 12.1 we note firstly, that after line (9) for every transformation induced by some letter there is exactly one letter in G that induces the same, as it is required for the data structure described in Section 3.1.5.

Secondly, the condition

$$M(L) = \{\delta_w \delta_u \,|\, \delta_w \in M, u \in G^*\}$$

is an invariant for the while-loop of lines (10) to (18).

12.3 The Implementation

The difficulty of the implementation is the representation of the state sets. The data structure used for the representation of M in AMoRE is a binary search tree. The ordering required for this is the lexicographic ordering of the arrays

$\delta_w(0)\ldots\delta_w(n)$ induced by the ordering of states that you have naturally if the states are represented by the integers $0,\ldots,n$.

The set W (for waiting) is represented by a queue, and the set G (for generators) is represented by an array. In the actual implementation, the set W is built up in the first for-loop simultaneously with G instead of the assignment of line (9).

The algorithm can easily be modified to yield also the syntactic semigroup. Obviously, one has

$$S(L) = \begin{cases} M(L) & \text{if } \exists w \in \Sigma^+ : id = \delta_w \\ M(L)\backslash\{id\} & \text{else.} \end{cases}$$

The words w are stored instead of their induced transformations δ_w. The FIFO-mechanism of the set W and the convention to trace the elements of G in the correct order makes sure that these words are the least representatives in the equivalence classes of the syntactic congruence.

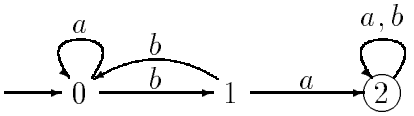


Figure 12.2: Example MDFA

		1	2	3	4	5	6	7
$*1$	1	1	2	3	4	5	6	7
$*a$	2	2	2	4	4	6	6	6
b	3	3	5	1	7	2	6	4
ab	4	4	6	2	6	2	6	4
ba	5	5	5	7	7	6	6	6
$*0$	6	6	6	6	6	6	6	6
$*bab$	7	7	6	5	6	5	6	7

Figure 12.3: The Resulting Monoid

12.4 Details about the Implementation

12.4.1 Data Structures

For the data structure `*monoid` see 3.1.5. Note that the descriptors `let2gen` and `generator` are only necessary for the input and output routines.

The data structure for one transformation is a struct `*melement`, which has the following descriptors:

`number` – the number that this transformation will have in the resulting monoid.

`lastletter` – same meaning as same descriptor of `*monoid`, see Section 3.1.5.

`transf` – an array with indices `0..qno` that holds the state transformation.

`gensucc` – same meaning as same descriptor of `*monoid`, see Section 3.1.5.

`rsucc, lsucc` – pointers of type `melement` to the right and left successor in the binary search tree.

The binary search tree used for the set M is established by these pointers.

`next` – a pointer of type `melement` that points to the next transformation that might yield new transformations.

The waiting queue W is established by these pointers.

The root of the tree and the beginning of the waiting chain are always the transformation *id* induced by the empty word. This structure is always pointed to by the static variable `root`. The global variable `nextno` holds one more than the number of transformations found so far.

12.4.2 Functions

The function visible from outside is `dfa2mon()` and can be found in the file `dfamon.c`. It receives a pointer to a DFA and returns a pointer to a monoid, i.e. a value of type `monoid`. It works like the way described above. Besides, `dfa2mon()` searches for a zero in the monoid and set the descriptor `zero` appropriately. There is one auxiliary function in this file, too, namely the function `searchtrans()`. It receives an array that represents a transformation and returns an `melement`, i.e. a pointer to a structure explained in 12.4.1.

`searchtrans` will search in the binary search tree for the received transformation and insert it into the tree correctly if necessary. If so, `searchtrans` allocates memory for this node and its descriptor `number` will be set to `nextno`. This counter has to be incremented by the calling function.

In any case, `searchtrans` returns a pointer to the node of the tree containing the transformation received.

12.4.3 Auxiliary Functions for Monoids

There are several useful auxiliary functions in the file `monoid.c`. We give a short explanation of them:

number	yielded by	trans	$\cdot a$	$\cdot b$
0*		**(0,1,2)**	**1**	**2**
1	**0** $\cdot a$	**(0,2,2)**		
2	**0** $\cdot b$	**(1,0,2)**		
0		**(0,1,2)**	**1**	**2**
1*	**0** $\cdot a$	**(0,2,2)**	**1**	**3**
2	**0** $\cdot b$	**(1,0,2)**		
3	**1** $\cdot b$	**(1,2,2)**		
0		**(0,1,2)**	**1**	**2**
1	**0** $\cdot a$	**(0,2,2)**	**1**	**3**
2*	**0** $\cdot b$	**(1,0,2)**	**4**	**0**
3	**1** $\cdot b$	**(1,2,2)**		
4	**2** $\cdot a$	**(2,0,2)**		
0		**(0,1,2)**	**1**	**2**
1	**0** $\cdot a$	**(0,2,2)**	**1**	**3**
2	**0** $\cdot b$	**(1,0,2)**	**4**	**0**
3*	**1** $\cdot b$	**(1,2,2)**	**5**	**0**
4	**2** $\cdot a$	**(2,0,2)**		
5	**3** $\cdot a$	**(2,2,2)**		
0		**(0,1,2)**	**1**	**2**
1	**0** $\cdot a$	**(0,2,2)**	**1**	**3**
2	**0** $\cdot b$	**(1,0,2)**	**4**	**0**
3	**1** $\cdot b$	**(1,2,2)**	**5**	**0**
4*	**2** $\cdot a$	**(2,0,2)**	**4**	**6**
5	**3** $\cdot a$	**(2,2,2)**		
6	**4** $\cdot b$	**(2,1,2)**		
0		**(0,1,2)**	**1**	**2**
1	**0** $\cdot a$	**(0,2,2)**	**1**	**3**
2	**0** $\cdot b$	**(1,0,2)**	**4**	**0**
3	**1** $\cdot b$	**(1,2,2)**	**5**	**0**
4	**2** $\cdot a$	**(2,0,2)**	**4**	**6**
5	**3** $\cdot a$	**(2,2,2)**	**5**	**5**
6*	**4** $\cdot b$	**(2,1,2)**	**5**	**4**

The first representatives and their transformations

	0	1	2
$*1$	0	1	2
$*a$	0	2	2
b	1	0	2
ab	1	2	2
ba	2	0	2
$*0$	2	2	2
$*bab$	2	1	2

Figure 12.4: An Example Calculation

The left-hand table shows some steps of the algorithm. Its first column holds the numbers of the monoid elements computed so far (the set M), and the elements below the one marked with a star form the set W of elements that might yield new transformations. The information of the second column show the contents of the descriptors `gensucc[0]` and `lastletter`. The third shows the array `trans` and the last contains the array `gensucc[1,2]` (if computed so far).

The right-hand table summarizes the resulting monoid. The star marks the idempotent elements.

`prword()` This function is useful for output routines. It receives three parameters: the number of a monoid element, a pointer to a monoid and a boolean flag. It generates a string that holds the least representative of that monoid element. The zero of the monoid (if there is one) is represented by `0`, the identity is represented by a `1`. If the boolean flag is true, the returned string starts with an additional star indicating that the monoid element is an idempotent. There is similar function `prword1()` that offers some more parameters. See file `monoid.c` for details.

`compword()` This function receives a pointer to a monoid and number of an element as parameters. It stores the least representative of that element as a sequence of generators (not as a sequence of letters !) into the descriptor `word` of the monoid. This function is only used inside `monoid.c`.

`mult()` This function receives a pointer to a monoid and two numbers `a,b` of elements. It returns the number of the element `a*b`, where `*` denotes the monoid multiplication. It works in time $\mathcal{O}(l(b))$, where $l(b)$ denotes the length of the minimal representative of b.

This function uses `compword()` to compute the least representative for b. Then the $a * b$ product is calculated using the array `mon->gensucc`.

A different way to compute $a * b$ is to build the composition of their mappings and find its image in the monoid. Even if the monoid was stored as a search tree, this method would take time $\mathcal{O}(q \log n)$, where q is the number of states in the underlying MDFA and n is the size of the monoid.

`comprestofmon()` This function computes for a monoid and its underlying MDFA the necessary information for the descriptors `no2trans`, `gensucc[..][0]`, `no2length`, `lastletter` from the descriptor `gensucc`.

`monmaxlen()` This function computes for a monoid an upper bound for the string length of its least representatives.

`multtable()` Output of the multiplication table of a monoid.

Chapter 13

D-Class Decomposition

13.1 Introduction

AMoRE's algorithm for the generation of this decomposition is probably its most complicated one. It makes use of several non-trivial results as presented in [Pi86].

This chapter will give an overview of the necessary definitions and facts and how they are used for the data structures and algorithms.

For another algorithm see [LM90]. It contains also a more sophisticated strategy for computing the *irregular* $\mathcal{D}$-classes, see Definition 13.4.

13.2 Mathematical Background

In the following, let M be a finite monoid. Let us start with the definition of Green's relations.

Definition 13.1 The three equivalence relations $\mathcal{D}, \mathcal{R}, \mathcal{L}, \mathcal{H}$ are defined as follows: For all $a, b \in M$ we let

$$\begin{aligned} a\mathcal{R}b &\text{ iff } aM = bM \\ a\mathcal{L}b &\text{ iff } Ma = Mb \\ a\mathcal{H}b &\text{ iff } a\mathcal{R}b \wedge a\mathcal{L}a \\ a\mathcal{D}b &\text{ iff } MaM = MbM \end{aligned}$$

These relations are called *Green's relations*.

One verifies easily that these are indeed equivalence relations. Moreover we remark that $\mathcal{R}$ and $\mathcal{L}$ are refinements of $\mathcal{D}$ and that $\mathcal{H}$ is the intersection of $\mathcal{R}$ and $\mathcal{L}$. So a syntactic monoid is partitioned into $\mathcal{D}$-classes, each of which is partitioned into $\mathcal{R}$-classes and $\mathcal{L}$-classes. The following Lemma gives alternative characterizations for these relations.

Lemma 13.2 The following conditions hold:

$$\begin{array}{lll} a\mathcal{R}b & \text{iff} & \exists u,v \in M : au = b \wedge a = bv \\ a\mathcal{L}b & \text{iff} & \exists u,v \in M : ua = b \wedge a = vb \\ a\mathcal{D}b & \text{iff} & \exists k \in M : a\mathcal{R}k \wedge k\mathcal{L}b. \end{array}$$

Besides, $\mathcal{R}$ and $\mathcal{L}$ commute, therefore $\mathcal{D} = \mathcal{R} \circ \mathcal{L} = \mathcal{L} \circ \mathcal{R}$.

It follows that a $\mathcal{D}$-class can be represented by the classical "egg-box" picture (see Figure 13.1), were each cell represents an $\mathcal{H}$-class, each row constitutes an $\mathcal{R}$-class and each column an $\mathcal{L}$-class.

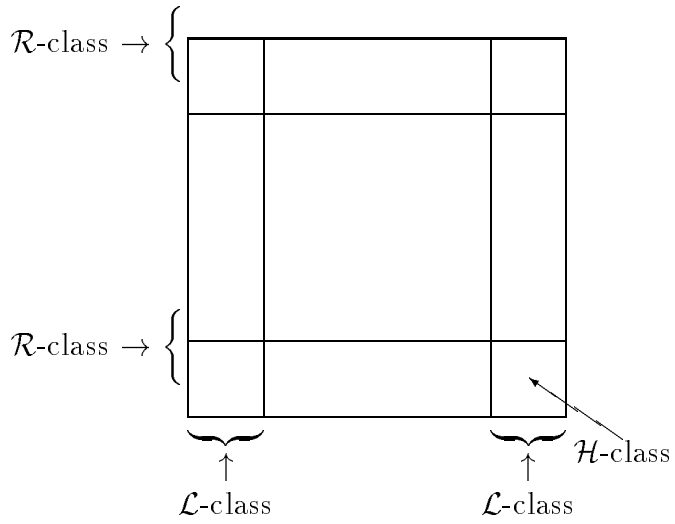


Figure 13.1: Egg-Box Diagram for $\mathcal{D}$-class

The following important lemma due to Green, [Gr51], leads to a data structure like AMoRE's, which contains indeed all information about one $\mathcal{D}$-class. It shows in particular that all $\mathcal{H}$-classes of one and the same $\mathcal{D}$-class have the same size.

Lemma 13.3 Let $a, b \in M$ with $a\mathcal{R}b$. Let L_a, L_b be the $\mathcal{L}$-classes containing a, b, respectively. Let $u, v \in M$ such that $au = b$ and $a = bv$ (see Lemma 13.2). Let $\rho_u : M \to M$, $m \mapsto mu$ and $\rho_v : M \to M$, $m \mapsto mv$.

Then ρ_u and ρ_v induce inverse bijections from L_a onto L_b, which preserve $\mathcal{H}$-classes; that means,

$$\forall m, n \in L_a : m\mathcal{H}n \Leftrightarrow mu\mathcal{H}nu,$$

$$\forall m, n \in L_b : m\mathcal{H}n \Leftrightarrow mv\mathcal{H}nv.$$

A dual statement holds if you replace $\mathcal{R}$ with $\mathcal{L}$ and right multiplication with left multiplication.

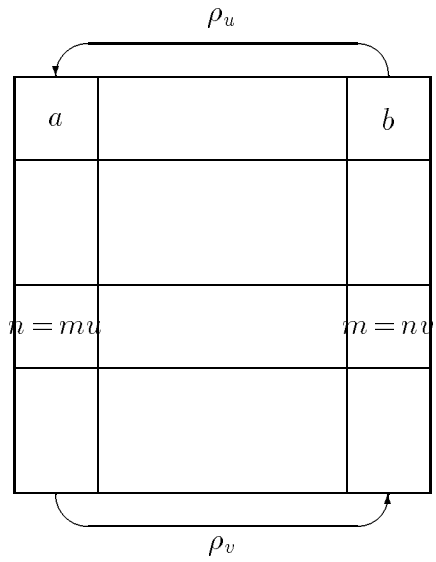


Figure 13.2: Illustration for Lemma 13.3

Figure 13.3 illustrates how Lemma 13.3 is used to represent a $\mathcal{D}$-class. It suffices to store the top left $\mathcal{H}$-class and the elements $s_1, \ldots, s_m$ and $r_1, \ldots, r_n$. These elements can be used to "transport" the original $\mathcal{H}$-class into each of the other $\mathcal{H}$-classes uniquely.

Definition 13.4 An element $a \in M$ is called *regular* iff $\exists m \in M : ama = a$. A D-class is *regular* iff all its elements are regular.

Lemma 13.5 Let D be a $\mathcal{D}$-class. The following conditions are equivalent:

1. D is regular.
2. D contains a regular element.
3. Each $\mathcal{R}$-class of D contains a least one idempotent.
4. Each $\mathcal{L}$-class of D contains a least one idempotent.
5. D contains at least one idempotent.
6. There exist $x, y \in D$ such that $xy \in D$.

Now we give some useful definitions and results for the practical calculation of the $r_1, \ldots, r_n$ and $s_1, \ldots, s_m$ of Figure 13.3.

We recall that we are interested in the syntactic monoid of a language. Because of Lemma 12.1 we may consider the syntactic monoid as a submonoid of the transformation monoid of the set of states of the minimal DFA. So let in the

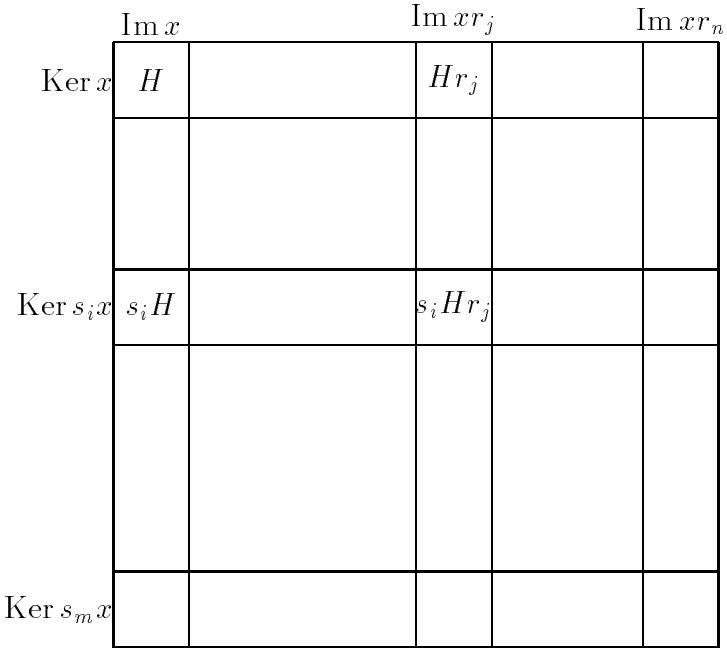


Figure 13.3: Egg-Box Diagram for $\mathcal{D}$-class

following Q be a finite set and M be a submonoid of the transformation monoid of the set Q, i.e. the elements of M are mappings $Q \to Q$ and the multiplication is the juxtaposition.

Definition 13.6 Let $a \in M$. We denote by $\mathrm{Im}\, a$ the image of a and by $\mathrm{Ker}\, a$ the partition of Q induced by the equivalence relation $\sim$ over Q defined by $q_1 \sim q_2 \Leftrightarrow q_1 a = q_2 a$.

We call $\mathrm{rg}\, a := |\mathrm{Im}\, a| = |\mathrm{Ker}\, a|$ the rank of a and $\mathrm{Ker}\, a$ the kernel of a.

A *transversal* of a partition $\mathcal{K}$ of Q is a subset $T \subseteq Q$ such that $\forall K \in \mathcal{K} : |K \cap T| = 1$.

For example, if $Q = \{0,1,2,3,4,5\}$

$$a = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 1 & 0 & 4 & 0 & 1 \end{pmatrix},$$

then we have $\mathrm{Im}\, a = \{0,1,4\}$, $\mathrm{Ker}\, a = \{\{0,1,5\},\{2,4\},\{3\}\}$ and $\mathrm{rg}\, a = 3$. A possible transversal of $\mathrm{Ker}\, a$ is $\{1,2,3\}$.

Lemma 13.7 Let $a, b \in M$. The following implications hold:

1. $a \mathcal{R} b \Rightarrow \mathrm{Ker}\, a = \mathrm{Ker}\, b$.
2. $a \mathcal{L} b \Rightarrow \mathrm{Im}\, a = \mathrm{Im}\, b$.

```
    { M submonoid of a transition monoid }
(1) W := M
(2) while W contains an idempotent
(3)   select and delete any idempotent e from W
(4)   find proper elements r_1, ..., r_n and s_1, ..., s_m (like in Figure 13.3)
        and the H-class H of e.
(5)   delete all elements of the D-class of e from W.
(6) endwhile
    { all regular D-classes are computed }
(7) while W ≠ ∅
(8)   select and delete any m from W
(9)   find proper elements r_1, ..., r_n and s_1, ..., s_m (like in Figure 13.3)
        and the H-class of m.
(10)  delete all elements of the D-class of m from W.
(11) endwhile
```

Figure 13.4: Algorithm for the $\mathcal{D}$-Class Decomposition

3. $a\mathcal{D}b \Rightarrow \operatorname{rg} a = \operatorname{rg} b$.

The regular $\mathcal{D}$-classes are so convenient because they are computed easily. The reason for this is the following Lemma:

Lemma 13.8 Let $a, x, y \in M$, a regular. The following implications hold:

1. $\operatorname{Im} ax = \operatorname{Im} x \Rightarrow a\mathcal{H}ax$.

2. $\operatorname{rg} yax = \operatorname{rg} a \Rightarrow a\mathcal{D}yax \vee yax$ is irregular.

3. $\operatorname{Im} ax$ is a transversal of $\operatorname{Ker} ya \Rightarrow ax\mathcal{D}a \wedge ya\mathcal{D}a$.

13.3 The Algorithm

13.3.1 Main Algorithm

The strategy is to compute all regular $\mathcal{D}$-classes first, using Lemma 13.8, and then to compute the remaining (irregular) $\mathcal{D}$-classes, using only Lemma 13.3. Note that "computing a $\mathcal{D}$-class" means only finding proper elements $r_1, \ldots, r_n$ and $s_1, \ldots, s_m$ of Figure 13.3, and the upper left $\mathcal{H}$-class.

The strategy is given in Figure 13.4.

Lines 2 to 6 compute all regular $\mathcal{D}$-classes whereas the remaining lines compute all irregular $\mathcal{D}$-classes. The elements e and m chosen in lines 3 resp. 8 as

```
    { (M,*) submonoid of a transition monoid, e ∈ M idempotent, W ⊆ M }
(1) n' := 0
(2) for all y ∈ M
(3)   if rg y * e = rg e and Ker y * e ∉ {K'_1, ..., K'_n'} then
(4)     n' := n' + 1: K'_n' := Ker y * e; y_n' := y
(5) endfor
    { y_1 ..., y_n' have the pairwise different kernels K'_1, ... K'_n' }
(6) n := 0; m := 0; H := ∅
(7) for all z ∈ M
(8)   if Im e * z = Im e then
(9)     insert e * z into H
(10)  else if Im e * z ∉ {I_1, ..., I_m} then
(11)    for all K' ∈ {K'_1, ... K'_n'}
(12)      if Im e * z is a transversal of K' then
(13)        if K'_i ∉ {K_1, ... K_n} then
(14)          n := n + 1; K_n := K'_i; s_n := y_i
(15)        if Im e * z ∉ {I_1, ..., I_m} then
(16)          m := m + 1; I_m := Im e * z; r_m := z
(17)    endfor
(18) endfor
```

Figure 13.5: Computation of the Regular $\mathcal{D}$-Class of an Idempotent

representatives of the $\mathcal{D}$-class in the upper left corner of the egg-box-diagram are denoted *original elements* in the following. The procedure for the irregular $\mathcal{D}$-classes works for arbitrary $\mathcal{D}$-classes as well, but it is advantageous to treat the regular ones first, because the given results enable a quick computation.

13.3.2 Regular $\mathcal{D}$-Classes

The algorithm for the step 4 is given in Figure 13.5.

In this algorithm, the $K'_1, \ldots K'_{n'}$ computed in lines 1 to 5 are only candidates for the kernels of the $\mathcal{D}$-class of e. Among these, the remaining lines determine the "good" kernels that enable a transversal of the form $\operatorname{Im} ez$. See Lemma 13.8.

Note that for the loop tracing all elements of ye and ez in lines 2 and 7, respectively, it is not necessary to examine actually all $y \in M$ ($z \in M$, resp.).

Instead, for the for-loop of line 2 there is a FIFO queue Q that is initialized with the neutral element of M and examines instead of all y only the elements gq for $g \in G$, $q \in Q$, where $G \subseteq M$ is a set that generates M. If an element with the same rank as e is found, it is appended to Q.

This is justified because $\forall a, b : \operatorname{rg} ab \leq \operatorname{rg} b$, so if for some y we have $\operatorname{rg} ye \neq \operatorname{rg} e$

{ M finite transition monoid, $m \in M$, $W \subseteq M$ }
(1) Let R be the set of all elements that are together with m in a cycle in the graph (M, E_r).
(2) Let L be the set of all elements that are together with m in a cycle in the graph (M, E_l).
(3) Let $H := R \cap L$

Figure 13.6: Step 9 of the $\mathcal{D}$-Class Decomposition

then for all x we might consider later, we have automatically $\mathrm{rg}\, xye \neq \mathrm{rg}\, e$.

A symmetrical argument applies to the for-loop in line 7, so the FIFO queue for the line 7 is initialized with the neutral element and examines the elements qg (with $g \in G$, $q \in Q$).

But note that if we modify this loop (as it is done in the implementation), we can not be sure that in line 18 H does actually contain *all* elements of the $\mathcal{H}$-class of e, but all elements of this $\mathcal{H}$-class are successors of elements of H. Thus in the actual implementation, the complete $\mathcal{H}$-class is computed in an extra loop.

13.3.3 Irregular $\mathcal{D}$-Classes

For the computation of the irregular $\mathcal{D}$-classes we have no better procedure than the one suggested by Lemma 13.2. So we consider the directed graph (M, E_r), where $E_r = \{(x, xg) \,|\, x \in M, g \in G\}$, $G \subseteq M$ is a set generating M, i.e. there is an edge leading from one monoid element to another one iff the first one can be yielded by right multiplication with one of the generators. Then we have a path leading from x to y iff $\exists z \in M\, y = xz$. Therefore there is a cycle containing x and y iff x and y belong to the same $\mathcal{R}$-class (see Lemma 13.2). An analogue construction can be done for the $\mathcal{L}$-classes, considering the set of edges $E_l = \{(x, gx) \,|\, x \in M, g \in G\}$.

The test in this imaginary graph works like this: First, make a breadth first search and mark all elements that are reachable from the original element m. If a cycle is encountered, mark the elements on it with an extra mark. Secondly, search for other cycles and mark elements on them with an extra mark.

For these tests there are several tricks to make it fast: Firstly, elements that are already in one $\mathcal{D}$-class are marked (with yet another mark) so that the search for a cycle may be aborted if such an element is encountered. Secondly, if a cycle is found, the elements on it can be found quickly by a reverse chain that is built up during the breadth first search.

The strategy for the step 9 of Figure 13.4 is given in Figure 13.6:

Experience shows that fortunately the irregular $\mathcal{D}$-classes are mostly small.

13.4 The Implementation

13.4.1 Functions

The described algorithm is (with certain optimizations) realized in file `mondcl.c`. The function visible from outside is `mon2dcl()`. It computes the $\mathcal{D}$-classes for the current language and fills the C-structure `curlan->lmon->ds`, which contains an array of C-structures `dclass` that hold one $\mathcal{D}$-class.

Before `mon2dcl()` is called, the syntactic monoid of the language must have been computed.

Another function visible from outside is `idempotent()`. It receives the number of an element and a pointer to a structure `*monoid` and returns a boolean value that is `TRUE` iff the element is an idempotent in the monoid.

The algorithm proceeds as described in the above section. Besides, it orders the $\mathcal{D}$-classes after calculation wrt. the following ordering. Let D_1 and D_2 be $\mathcal{D}$-classes. Then $D_1 < D_2$ iff

1. $\mathrm{rg}\, D_1 < \mathrm{rg}\, D_2$ or
2. the above criterion gives no decision, but D_1 regular, D_2 irregular, or
3. the above criterion gives no decision, but D_1 has less $\mathcal{R}$-classes than D_2, or
4. the above criterion gives no decision, but D_1 has less $\mathcal{L}$-classes than D_2, or
5. the above criterion gives no decision, but D_1 has smaller $\mathcal{H}$-classes than D_2, or
6. the above criterion gives no decision, but the original element of D_1 (to be presented in the upper left corner of the display of D_1) is smaller (in the canonical ordering of words) than the one of D_2.

The functions performing the single steps are described below:

`mon2dcl()`: The complete algorithm of Figure 13.4.

`l_reg_class()`: Steps 1 to 5 of Figure 13.5.

`r_h_reg_class()`: Steps 7 to 18 of Figure 13.5.

`r_irreg_class()`: Step 1 of Figure 13.6.

`l_irreg_class()`: Step 2 of Figure 13.6.

`d_irreg_class()`: Step 3 of Figure 13.6.

`serchkernel()`: Test in line 3 of Figure 13.5.

`searchimage()`: The tests of lines 8, 10, 12 and 15 together. That is, `searchimage()` receives the number of an element and tests the following:

- Does it have the same image as the original element e? If so, `searchimage()` returns `3` to indicate this fact.
- If not, then check for every kernel K' if its image is a transversal of that kernel. (Thereby, the kernels are resorted so that the "good" kernels, which allow such a transversal, stand in the beginning of the `kernelarray`.)

  If there is no such kernel, return `0`. If there is one, return `2` if the image is known (i.e. $\in \{I_1, \ldots, I_m\}$), `1` if it is not known.

After calling `searchimage()`, the calling function `r_h_reg_class()` can continue appropriately, depending on the returned value.

There are several auxiliary functions, namely:

`imcompare()`: This function is required for the extra test necessary due to modification we mentioned at the end of Section 13.3.2. It receives an element as a parameter and compares its image with the current original element e.

`cpt_rang()`: This function receives an element and computes its rank.

`cpt_image()`: This function receives an element and computes its rank and its image.

`cpt_kernel()`: This function receives an element and computes its rank and its kernel.

The functions `cpt_rang()`, `cpt_image()` and `cpt_kernel()` store their results into the data structures `no2rang`, `kernelarray` and `imagearray` described below. To avoid unnecessary calls of these functions, the calling function usually looks inside these arrays before.

`dc1gedc2()`: This function is used to define an ordering on the set of $\mathcal{D}$-classes of one monoid.

`orderdclass()`: This function orders the $\mathcal{D}$-classes wrt. the ordering defined by `dc1gedc2()`.

13.4.2 Data Structures

Here follows a detailed explanation of the data structures and variables used. Remember not to get confused about what is meant with "element": when talking about C-code, it is a positive integer that receives its meaning only from the contents of the data structure representing the current monoid. Nevertheless,

if `x` and `y` are such elements, we will write `x * y` and mean the multiplication of monoid elements that correspond to the indices `x` and `y`, or rather the (C-code)-element that represents the result of this multiplication. AMoRE offers some C-functions for this purpose, but our notation is more readable.

Fortunately, the standard multiplication of integers is not needed here.

As often in AMoRE, as many of the required variables as possible are global in the file `mondcl.c` in order not to waste time in parameter passing and memory allocation for local variables. In the algorithm there is always one current $\mathcal{D}$-class. The information about this $\mathcal{D}$-class is stored in certain static variables, and most of the functions in this file find their input in them and store their output into them.

Similarly, in some situations of the algorithm, there is a particular kernel or image, that is interesting at the moment. This can typically be found at a particular position in some static array.

The original element (like e or m in Figure 13.4) is an exception because this element out of the upper left $\mathcal{H}$-class of the current $\mathcal{D}$-class is always passed as a parameter named `no`.

Some Static Variables

Some of the static variables in file `mondcl.c` are:

`dfirst, dlast` Pointers to the beginning and the end of a chain of C-structures `*d_list`. Each entry keeps a pointer to a structure `*d_list` for the information of one $\mathcal{D}$-class.

`currentrang` The rank of the $\mathcal{D}$-class currently dealt with.

`no2rang` An array indexed `0...mno-1` that holds for every monoid element its rank if it has been computed, otherwise `0`.

`countimage` An array used in some functions for the computation of the image of a monoid element. This array is only static due to reasons of efficiency; functions have to re-initialize it to zero after usage.

`countkernel` An array used for the computation of the kernel of a monoid element. See the remark for `kernelarray` below. Again, this array could very well be local in the functions that use it.

`mark` An array that marks certain elements as "already dealt with" in some way. The usage of this array is different in the computation of regular and irregular $\mathcal{D}$-classes.

In the function `r_h_reg_class()` performing steps 7 to 18 in Figure 13.5, `mark` is used to mark elements that are accidentally found out to be irregular

or in an already computed $\mathcal{D}$-class and therefore not interesting at the moment.

If later on in the computation of the regular $\mathcal{D}$-classes such an element is encountered again, it will be skipped without further test.

In the functions `r_irreg_class()` and `l_irreg_class()` performing the steps 1 and 2, the array `mark` is used to mark the visited elements in the breadth first search in the considered graph. There you have `mark[i]==2` only if `i` is an element of the same $\mathcal{R}$-class ($\mathcal{L}$-class, resp.) as the current original element, `mark[i]==1` if `i` is reachable in the considered graph, but unknown whether it is in a cycle, and `mark[i]==0` if `i` has not turned out to be reachable from the original element at all.

Static Variables for Regular Classes

Especially for the computation of the regular $\mathcal{D}$-classes, there are the following static variables in file `mondcl.c`:

`maximage, maxkernel` The maximal number of images (kernels) in one of the regular $\mathcal{D}$-classes treated so far. `imagearray` (`kernelarray`, resp.) are twodimensional arrays. They are sufficiently large in the first dimension, but memory for the second dimension is only allocated for the first index $\leq$`maximage` ($\leq$`maxkernel`, resp.).

`imagearray` An array keeping for every `0` $\leq$ `i` $\leq$ `maximage-1` an array indexed `0,...,currentrang` that represents one image that occurs in the currently computed $\mathcal{D}$-class.

`imagearray` corresponds to the $I_1, \ldots I_m$ in Figure 13.5.

`kernelarray` An array keeping for every `0` $\leq$ `i` $\leq$ `maxkernel-1` an array indexed `0,...,qno+currentrang+1`, where `qno` is the maximal state number in the minimal DFA. This array represents one kernel like in the following example:

Consider the example transformation of the state set $\{0,1,2,3,4,5\}$, (i.e. `qno==5`) $a = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 1 & 0 & 4 & 0 & 1 \end{pmatrix}$.

Its kernel is the partition $\{\{4,2\},\{3\},\{0,1,5\}\}$, stored as `( 0 1 5 6 2 4 6 3 6 )`.

So the number `qno+1` is used as a separator, and the sets of the partition are ordered with respect to their least elements, and the elements of each set of the partition are ordered itself.

`kernelarray` corresponds both to the $K'_1, \ldots, K'_{n'}$ and to the $K_1, \ldots, K_n$ of the algorithm in Figure 13.5. When the latter are computed, the array is just sorted.

In the computation of the kernels in function `cpt_kernel()`, the array `countkernel` is additionally used. It keeps for every monoid element out of the current image the index of the beginning of the corresponding partition set in its kernel, which is stored in `kernelarray[kernumber]`.

If, for example, `kernelarray[kernumber]` and `countarray` hold the above example, then `countarray[0]==5`, `countarray[1]==0`, `countarray[4]==8`, because the elements of Q that have the image 0 start at position 5 in the `kernelarray[kernumber]` and so on.

`imnumber, kernumber` The current number of images (kernels, resp.) in the $\mathcal{D}$-class currently dealt with. Both are initialized to zero, so at any time, `imnumber` (`kernumber`, resp.) holds the number of the image (kernel, resp.) currently dealt with.

These numbers correspond to m (n', resp.) of the algorithm in Figure 13.5.

`goodkernel` The number of kernels in the current $\mathcal{D}$-class that represent $\mathcal{R}$-classes. This number corresponds to n in Figure 13.5.

Static Variables for Irregular Classes

Finally, there are some static variables reserved for the computation of the irregular $\mathcal{D}$-classes, namely:

`rlqueue` The FIFO-storage required for the breadth first search in the considered graph.

`suffarray` Only used in function `r_irreg_class()`. All the elements in the array `rlqueue` are of the form `no*y`, and suffarray holds this `y`. That is, `rlqueue[i]==`
`no*y` iff `suffarray[i]==y`.

`prearray` Similar to `suffarray`, but used in function `l_irreg_class`. Therefore, `rlqueue[i]==y*no` iff `prearray[i]==y`.

`pred` An array, that holds for any position in the `rlqueue`, except for the `0`, the (smaller) position in `rlqueue` holding the element that caused `rlqueue[i]` to be inserted into the queue.

So in the function `r_irreg_class()` this means that there is for all `i` $\neq 0$ some generator `g` with `rlqueue[pred[i]] * g == rlqueue[i]`.

In function `l_irreg_class()` this means that there is for all `i` $\neq 0$ some generator `g` with `g * rlqueue[pred[i]] == rlqueue[i]`.

`prefix, suffix` These arrays hold the $s_1, \ldots, s_m$ and the $r_1, \ldots, r_n$, respectively, out of Figure 13.3 for the current $\mathcal{D}$-class, where the x from Figure 13.3 is the element with number `no`. These arrays are indexed the same way as `kernelarray` and `imagearray`, respectively.

So if, for example, `prefix[i]==y`, then the kernel of the element `y * no` is stored in `kernelarray[i]`: and if `suffix[i]==x` then the image of the element `no*x` is stored in `imagearray[i]`.

Chapter 14

From Syntactic Monoid to Defining Relations

14.1 Introduction

Since the syntactic congruence gives lots of information about a regular language, it is an interesting question how to make it visible. One way is to give a set of pairs of words such that the syntactic congruence is the smallest congruence including this set. We call the pairs of such a set "defining relations[1]".

AMoRE computes such a set of defining relations that has the additional property that the right hand side of a relation is always smaller than the left hand side. Besides, a consecutive application of these relations to infixes of a word always yields the least representative of its congruence class; and the set of these relations is minimal with this property.

In [Pi86] one finds the algorithm given in Chapter 12 for the computation of the syntactic monoid from the DFA in a version that also computes the defining relations at the same time. However, in AMoRE, the computation of these relations is done in an extra step immediately after the computation of the syntactic monoid.

14.2 Main Idea

In the following let L be a regular language and $\cong$ be the syntactic congruence wrt. L. For all $w \in \Sigma^*$ we will denote the least representative of its syntactic congruence class by $\langle w \rangle$.

Before we start to explain the algorithm for the computation of the defining relations, we will have a closer look at these relations and how to store them. The relations are of the form $wg \to u$, where w is a least representative of a

[1]The term "relation" is due to historical reasons and might be misleading because in fact one set of "defining relations" is simply one relation in the usual meaning.

congruence class and g is a least representative of a generator and $u = \langle wg \rangle$. (Recall that the generators are the $[a]$ for $a \in \Sigma$.) Obviously it is not necessary to store the right hand side u explicitly because it can be computed easily.

We note also that if $w \to u$ is a relation, then any congruence relation containing this pair also contains all pairs of the form (xwy, xuy) for $x, y \in \Sigma^*$.

Therefore it suffices to consider relations of the mentioned form because any word that is not minimal wrt. the syntactic congruence will not be minimal wrt. to a congruence including all relations of the mentioned form because the shortest non-minimal suffix of w will be a left hand side of one of the relations.

So the main idea is to compute all wg with w (g) being a least representative of a monoid element (a generator, resp.) and insert the relation $wg \to \langle wg \rangle$ into the list. To avoid superfluous relations we will not insert it if wg and $\langle wg \rangle$ are equal or if there is already a relation in the list whose left hand side is a suffix of w.

14.3 Theoretical Background

For a set of pairs $D \subseteq \Sigma^* \times \Sigma^*$ and a congruence $\sim$ we will say that D *generates* $\sim$ iff $\sim$ is the smallest congruence including D. For any such D, there is always one unique congruence generated by D, namely the intersection of all congruences that include D.

One may ask for an algorithm that computes a set D minimal among the sets that generate $\cong$. But the algorithm realized in AMoRE computes a set of pairs that is minimal in some other sense, namely it is minimal among the sets that generate $\cong$ and yield unique representatives when applied only in one direction.

To put this more formally, we need some more definitions.

Definition 14.1 Let $\to \subseteq \Sigma^* \times \Sigma^*$. We call $(\Sigma, \to)$ (or sometimes only $\to$) a *Semi-Thue-System*. For such a Semi-Thue-System, we denote by $\Rightarrow$ the relation on Σ^* defined by

$$u \Rightarrow v \iff \exists x, y, z_1, z_2 \in \Sigma^* : u = xz_1y, v = xz_2y, z_1 \to z_2.$$

We denote by $\overset{*}{\Rightarrow}$ the transitive and reflexive closure of $\Rightarrow$, that is

$$\overset{*}{\Rightarrow} := \bigcup_{i \geq 0} \Rightarrow^i,$$

where $\Rightarrow^i$ denotes the i-fold relation product and $\Rightarrow^0$ is the identity on Σ^*.

We denote by $\Leftrightarrow := (\Rightarrow \cup \Rightarrow^{-1})$ the smallest symmetrical relation containing $\Rightarrow$, and consequently we define

$$\overset{*}{\Leftrightarrow} := \bigcup_{i \geq 0} \Leftrightarrow^i .$$

```
    {M syntactic monoid }
(1) R := ∅
(2) for all least representatives w of congruence classes
(3)   for all least representatives g of generators
(4)     if wg ≠ ⟨wg⟩ then
(5)       if wg has no suffix in R then
(6)         insert wg into R
(7)   endfor
(8) endfor
    {{(r, ⟨r⟩) | r ∈ R} generates ≅}
```

Figure 14.1:

It is easy to see that for a Semi-Thue-System $\rightarrow$, $\overset{*}{\Leftrightarrow}$ is the congruence generated by $\rightarrow$.

Definition 14.2 Let $\rightarrow$ be a Semi-Thue-System. An element $w \in \Sigma^*$ is *irreducible* if there is no $u \in \Sigma^*$ such that $w \Rightarrow u$ (i.e. w contains no left hand side of $\rightarrow$ as an infix.

We say that $\rightarrow$ *yields unique representatives* if for all irreducible $u, w \in \Sigma$ with $u \overset{*}{\Leftrightarrow} w$ we have $u = w$.

The next section will describe an algorithm that computes for a syntactic monoid a minimal Semi-Thue-System that generates the syntactic congruence and yields unique representatives.

14.4 The Algorithm

The algorithm as described so far is shown in Figure 14.1.

The following Lemma states the fact we have mentioned above.

Lemma 14.3 Let $\forall a, b \in \Sigma : a \ncong b$. Let R' be the set of left hand sides of pairs produced by the algorithm in Figure 14.1. Let $\rightarrow := \{(r, \langle r \rangle) \mid r \in R'\}$. Then $\cong$ is indeed generated by $\rightarrow$, i.e.:

1. $\rightarrow \subseteq \cong$.

2. For all congruences $\sim$ with $\rightarrow \subseteq \sim$ it holds $\cong \subseteq \sim$.

Moreover, $\rightarrow$ yields unique representatives (namely the least representatives) and it is minimal with this property.

The proof is straightforward.

For the complexity we observe that if n is the number of elements in the monoid and t is the number of generators, then line 5 of Figure 14.1 will be reached $n \cdot (t-1)$ times, since the line before will be reached $n \cdot t$ times and the `if`-test yields false in n of these times.

The test in line 5 and the insertion into the set R, which is implemented as a binary search tree, will be performed together. These lines require at most to trace the longest path in that tree and to trace of the maximal representation length of letters in a least representative. Let l be that size. Since the path length in the tree has an average case bound of $\mathcal{O}(n)$, we obtain an average case complexity bound of $\mathcal{O}(nt(l + \log n))$.

14.5 The Implementation

14.5.1 A Modification of the Algorithm

The function performing the algorithm of Figure 14.1 is named `mon2rel()` and can be found in file `monrel.c`. One modification concerning the output is made. If the monoid contains a zero, i.e. an element z such that $\forall m \in M : zm = mz = z$, then the relation $w \to w$, where w is the least representative of the zero, will be inserted. Every time w appears on the right hand side of a relation, it will be replaced with 0 in the output. This convention justifies to suppress the output of relations of the form $uw \to w$ and $wu \to w$.

14.5.2 Data Structures

The ordering $<_{\text{rlex}}$ on Σ^* needed for the binary search tree representing the set R of Figure 14.1 is the lexicographic ordering of the reverted words, i.e.

$$x <_{\text{rlex}} y :\Longleftrightarrow \exists u, v, w \in \Sigma^* \, \exists a, b \in \Sigma : x = uaw \wedge y = vbw \wedge a < b.$$

In order to implement the modification described above, this tree will only contain those nodes of the tree whose left hand side is not a non-minimal representative of the zero.

Additionally, the nodes of the tree form a chained list used for the output. The relations will be put out in the order given by this list. Both the search tree and this chain consist of C-structures `*reltree`, which contain the following descriptors to encode a relation of the form $wg \to \langle wg \rangle$:

`left`: The number of the monoid element $[w]$.

`gen`: The number of the monoid element $[g]$ (which is a generator).

`lsucc,rsucc`: Pointers to left and right successors in the search tree. In the tree, all elements greater than wg will be in the subtree starting at the left successor, whereas the smaller ones will be found below the right successor.

`next` Pointer to the next element in the list of relations for the output.

14.5.3 Functions

We remark that the loops starting in lines 2 and 3 trace their elements upwards, i.e. smaller[2] words are always dealt with before larger ones.

This ensures that if a potentially new element wg is to be inserted into the tree and there is a suffix of wg in the tree, then this suffix must be on the path to the potentially new position of wg in the tree. This observation enables us to perform the test of line 5 and the insertion in line 6 simultaneously: In order to determine the position of the new wg in the search tree, the nodes on the path to this position are compared with wg. If a suffix of wg is encountered the search is aborted and wg is not inserted. Otherwise it is continued either at the left or at the right successor, depending on whether wg is larger or smaller than the current node (wrt. $<_{\text{rlex}}$).

The function `inrel()` performs steps 5 and 6 of Figure 14.1 in this manner. It uses function `revlex()`, which receives two pointers `rel1`, `rel2` to structures `*reltree`. The smallest representative of `rel1->left` must be longer than the smallest representative of `rel2->left`, which is made sure tracing the w's in line 2 upwards.

Let u_1a_1 and u_2a_2 be the words represented by `rel1->left`, `rel1->gen` and `rel2->left`, `rel2->gen`, that means u_1 is the least representative of the monoid element `rel1->left` and so on. Then `revlex()` returns

`1` if u_1a_1 is a suffix of u_2a_2,

`0` if $u_2a_2 <_{\text{rlex}} u_1a_1$ and

`2` if $u_1a_1 <_{\text{rlex}} u_2a_2$.

The function `mon2rel()` performs the whole algorithm of Figure 14.1 with the additional convention mentioned in Subsection 14.5.1: After line 4 it is tested if wg is a zero. If so, a new node `r` is inserted into the tree and prepended to the list of the output relations. The descriptors `r->left` and `r->gen` will represent the decomposition va of the least representative into $v \in \Sigma^*$ and $a \in \Sigma$. (This will be the only left hand side that is a least representative.) Once such a zero has been discovered, the lines 5 and 6 will be skipped for elements wg where $[w]$ or $[g]$ is the zero.

[2] wrt. the canonical order (by length, and lexicographic for fixed length).

Note that since the "zero"-relation is prepended to the list whereas all other relations are added to the end of the list, the "relation" $w \to 0$ (with w being the least representative of the zero) will, if it exists, always be the first in the output. The remaining relations will be ordered wrt. their left hand sides and the usual ordering[3], among them the possible relations of the form $ug \to 0$ with $[u]$ not being the zero.

Besides, there are two more functions in file `monrel.c`, namely `editrel()` for the display of defining relations and `prtrel()` for their printing.

[3]defined in Section 1.3

Chapter 15

From Monoid to Starfree Expression

Considering generalized regular expressions, i.e. expressions in which all boolean operators are allowed, one observes that in many cases one can find expressions that do not contain any Kleene stars. Such expressions are called *starfree expressions*, and the languages that enable a representation by such an expression are called starfree as well. In Chapter 16 we cite an effectively testable criterion for starfreeness of a regular language. The proof for this criterion yields also an algorithm how to find a starfree expression. This proof can be found in [Pe90]. See also [Sch65]. This chapter is to give only a short summary of the main facts used in the implementation of that algorithm in AMoRE. This conversion has, however, only little practical application, since the starfree expressions computed this way tend to be long and unreadable. It remains a deep unresolved problem to find short starfree expressions for a given language.

15.1 Definitions and Examples

Definition 15.1 Let $L \subseteq \Sigma^*$. L is *starfree* if there is a generalized regular expression for it with no Kleene star in it, i.e. it can be obtained from subsets of Σ by a finite number of set-concatenations, unions and complementations.

For example, if $\Sigma = \{a, b\}$, then the language $L((ab)^*)$ is starfree, since

$$L((ab)^*) = L(\varepsilon \cup (a\Sigma^* \cap \Sigma^* b) \backslash \Sigma^*(aa \cup bb)\Sigma^*)$$

and $\Sigma^* = \sim \emptyset$.

The following property turns out to be equivalent to being starfree.

Definition 15.2 Let $L \subseteq \Sigma^*$. L is called *aperiodic* if there is an $n \in \mathbb{N}$ such that for all $x, y, z \in \Sigma^*$ one has

$$xy^n z \in L \Longleftrightarrow xy^{n+1} z \in L.$$

The following Lemma summarizes some characterizations proved in [Sch65] and [MNP72],cf. also [Pi86].

Lemma 15.3 Let $L \subseteq \Sigma^*$ be a regular language. Then the following statements are equivalent.

1. L is starfree.
2. L is aperiodic.
3. The syntactic Monoid of L contains no non-trivial subgroups.
4. The syntactic Monoid of L contains no non-trivial $\mathcal{H}$-classes.

15.2 The Algorithm

Let L be an aperiodic language, M its syntactic monoid, $\varphi : \Sigma^* \longrightarrow M$, $w \mapsto [w]$ be the canonical homomorphism. The main observation for the algorithm is the following fact:

$$\varphi^{-1}(m) = (U\Sigma^* \cap \Sigma^* V) \backslash \Sigma^* W \Sigma^*$$

where the sets U, V, W are defined as follows:

- $U := \bigcup\{\varphi^{-1}(n)a \,|\, (n,a) \in M \times A : n\varphi(a)M = mM, n \notin mM\}$.
- $V := \bigcup\{a\varphi^{-1}(n) \,|\, (a,n) \in A \times M : M\varphi(a)n = Mm, n \notin Mm\}$.
- $W := \{a \in \Sigma \,|\, m \notin M\varphi(a)M\} \cup \bigcup\{a\varphi^{-1}(n)b \,|\, (a,n,b) \in \Sigma \times M \times \Sigma, m \in M\varphi(a)nM \cap Mn\varphi(b)M, m \notin M\varphi(a)n\varphi(b)M\}$.

Clearly, $L = \bigcup_{w \in L}[w] = \{\varphi^{-1}(m) \,|\, n \in M, \varphi^{-1}(m) \subseteq L\}$, so we may construct a starfree expression using this formula inductively and the fact that for all the $n \in M$ appearing inside the definitions of U, V, W on the right hand side of the formula, we have $|MmM| < |MnM|$. For details of this proof see [Pe90], where this algorithm has been suggested first.

In AMoRE, Perrin's method is slightly adapted: for the tests if $n\varphi(a)M = mM$ and so on, the $\mathcal{D}$-class decomposition is used.

The algorithm using the above remarks is implemented in file `monsfx.c`.

Chapter 16

Tests on the Monoid

16.1 List of Tests on Monoid

There are several language properties for which there exist effective tests that run either on the syntactic monoid or its decomposition according to Green's relations. These tests (and the names of AMoRE's functions for them) are listed below:

$L \in \mathrm{FOL_U}$?	`folutest()`
L starfree ?	`sftest()`
Has L dotdepth one ?	`testdd1()`
L definite ?	`proptest()`
L reverse definite ?	
L generalized definite ?	
L locally testable ?	
L finite or co-finite ?	
L piecewise testable ?	`testpwt()`

16.2 Comments on Background

The class $\mathrm{FOL_U}$ contains the languages definable in first-order logic with the "U-predicate" in the sense of [MNP72]; this predicate allows a weak form of recursion in defining formulars and thus represents a proper extension of first-order logic (in which, however, only special regular languages are definable). The test for $L \in \mathrm{FOL_U}$ follows the algorithm presented in [MNP72].

Note that `folutest()` receives a pointer to the minimal DFA of the input language. The test itself runs on the syntactic monoid of a DFA that has been modified in a particular way.

The test whether the given language L is starfree uses Schützenberger's theorem [Sch65] (for the complexity see also [St85]):

Lemma 16.1 L is starfree iff the syntactic monoid of L contains only trivial groups.

In terms of Green's relations this means that each H-class of the syntactic monoid consists of only one element.

The property of piecewise testability can be checked using Simon's Theorem [Si75]:

Lemma 16.2 L is piecewise testable iff each $\mathcal{D}$-class of its syntactic monoid consists of only one element.

The test for dot-depth 1 implements a highly non-trivial (and also highly inefficient) criterion due to Knast [Kna84]:

Lemma 16.3 Let L be a starfree language, S its syntactic semigroup, $E \subseteq S$ the set of idempotents of S and $n := |S|$. Then

$$L \text{ is of dot-depth } 1 \iff$$
$$\forall e, f \in E \, \forall x, y, u, v \in S : (exfy)^n exfvx(ufve)^n = (exfy)^n e(ufve)^n$$

The remaining language properties are checked via the following equivalences:

Let L be a regular language, S its syntactic semigroup, $E \subseteq S$ the set of idempotents of S. Then

$$\begin{aligned}
L \text{ finite or co-finite} &\iff \forall e \in E : eS = Se = e, \\
L \text{ definite} &\iff \forall e \in E : eS = e \\
L \text{ reverse definite} &\iff \forall e \in E : Se = e \\
L \text{ generalized definite} &\iff \forall e \in E : eSe = e \\
L \text{ locally testable} &\iff \forall e \in E : eSe \text{ contains only idempotents} \\
&\qquad\qquad\qquad\qquad \text{and is commutative}
\end{aligned}$$

For proofs of these characterizations see [Pi86] or [Ei76].

Chapter 17

Tests on Automata

All binary tests are collected in the file `win_two.c`. The functions performing these tests are:

- `equiv()` for the test of equality of languages and
- `inclusion()` for either the test of inclusion or of disjointness of languages.
- `empty_full_lan()` for the test if the current language is empty or equal to Σ^*.

17.1 Tests Concerning One Language

Emptiness/Fullness Test

This test is very simple: Check if the minimal DFA has only one state. If not, the language is neither full (i.e. equal to Σ^*) nor empty. If so, it is full if this state is final, and it is empty if this state is not final. The function performing this test is `empty_full_lan()`. It receives no parameters and returns `TRUE` if the language is full, `FALSE` it it is empty and `UN_KNOWN` if it is neither full nor empty.

Note that this return value convention is an abuse of the third boolean value `UN_KNOWN`.

17.2 Tests Concerning Two Languages

17.2.1 Equality Test

The equality test implemented in AMoRE has two steps : First, compute the minimal DFA's of the input languages, using function `compmdfa()`. Then check if these are isomorphic, using function `equiv()`. This condition is sufficient because for any regular language the minimal DFA is unique up to isomorphism.

Initialize π as in (17.1).
For all $q \in \mathrm{dom}(\pi)$
  For all $a \in \Sigma$
    $p_2 := \delta_2(\pi(q), a))$
    $p_1 := \delta_1(q, a)$
    if $p_1 \in \mathrm{dom}(\pi)$ then
      if $\pi(p_1) \neq p_2$, then return `false`;
    else
      if $p_2 \in \mathrm{range}(\pi)$ then return `false`;
      otherwise define $\pi(p_1) := p_2$;
      (thereby adding p_1 to $\mathrm{dom}(\pi)$ and p_2 to $\mathrm{range}(\pi)$)

Figure 17.1: The Algorithm for the Equivalence Test

Strategy of `equiv()`

Let $\mathcal{A}_1 = (\Sigma, Q_1, q_{01}, \delta_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, q_{02}, \delta_2, F_2)$ be the two minimal DFA's to be compared. Note, that for the case of minimal DFA's we may assume that all states are reachable.

`equiv()` first compares the number of states of the two DFA's and returns false if they are not the same.

In the second phase, `equiv` tries to compute a bijection $\pi : Q_1 \to Q_2$ that commutes with the two transition functions in the following sense:

$$\forall a \in \Sigma \, \forall q \in Q_1 : \pi(\delta_1(q, a)) = \delta_2(\pi(q), a).$$

The algorithm starts with the partial mapping

$$\pi : q \mapsto \begin{cases} q_{02} & \text{if } q = q_{01} \\ \perp & \text{else} \end{cases} \tag{17.1}$$

and then successively increments the domain of π according to the needs of the transition function δ_1, making sure that π remains an injection that commutes with the transition functions.

The algorithm for this is shown in Figure 17.1. Note that after execution of the outermost loop, $\mathrm{dom}(\pi)$ is a subset of Q that contains the initial state and is closed under the transition function. Since in every minimal DFA all states are reachable, the algorithm will not terminate before $\mathrm{dom}(\pi) = Q$ or it has realized that there exists no bijection that commutes with the transition functions.

In the third phase, `equiv()` checks whether $\pi(F_1) = F_2$ holds for the bijection π constructed in the second phase. Since the property $\pi(q_{01}) = q_{02}$ is made sure by the initialization, this final tests suffices to verify that π is an isomorphism.

Details about `equiv()`

The mapping π is held in the array `bijection[]` (though it is not a bijection initially). The states of $\mathrm{dom}(\pi)$ (resp. $\mathrm{range}(\pi)$) are marked with `true` in the boolean array `mark1[]` (resp. `mark2[]`).

The states of Q_1 are traced in a breadth-first manner which requires a FIFO storage to keep those states that have been visited but might be connected to ones which have not. This storage is a chained list of elements of type `list`. This list is initialized with `listinit()` to be the list only containing the initial state of $\mathcal{A}_1$, and it is enlarged using `listinsert()` each time a new state is added to $\mathrm{dom}(\pi)$. The functions `listinit()` and `listinsert()` can be found in `listop.c`.

The pointer `first` is used to indicate the position in this list where the next state of $\mathrm{dom}(\pi)$ is found whose outgoing transitions have not been dealt with yet.

Note that before all these computations, `equiv()` checks whether the size of the alphabets of the input languages is the same.

Complexity

The test of language equality by the algorithm described above is performed in time $\mathcal{O}(nt\log(n))$, where t denotes the number of letters in Σ and n denotes the number of states of an input DFA, which does not have to be minimal. This is the time needed for the computation of the minimal DFA's.

The test of isomorphism as described above takes only time $\mathcal{O}(mt)$, m denoting the number of states in the minimal DFA, which is, of course, less or equal as in the input DFA. To see this, note that every pair $(q,a) \in Q \times \Sigma$ requires exactly one loop of constant time.

Note that there is an algorithm for the equivalence test which is even faster, using the Union-Find algorithm. It works on arbitrary DFA's with n states in time $\mathcal{O}(nt\alpha(n))$, α denoting the inverse of the Ackermann function. The reason why AMoRE uses the above algorithm is the fact that it works in linear time if the minimal DFA's have already been computed, which will be very often the case when the equivalence test is activated.

17.2.2 Inclusion and Disjointness Tests

The main idea for these tests is to compute all reachable states in the product of the two DFA's accepting the languages and to test simultaneously whether they satisfy a particular condition concerning their final states, namely:

- if the first automaton reaches a final state, then so does the second (in case of inclusion test), or
- if the first automaton reaches a final state, then the second does not and vice versa (in case of disjointness test).

More formally, let $\mathcal{A}_1 = (\Sigma, Q_1, q_{01}, \delta_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, q_{02}, \delta_2, F_2)$ be DFA's accepting the languages L_1 and L_2, respectively. The *product automaton* [1] is defined as $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, with

$$\begin{aligned} Q &= \{(\delta_1(q_{01}, w), \delta_2(q_{02}, w)) \in Q_1 \times Q_2 \,|\, w \in \Sigma^*\} \\ \delta &: Q \times \Sigma \to Q,\ ((p,q),a) \mapsto (\delta_1(p,a), \delta_2(q,a)) \\ F &: F_1 \times F_2 \end{aligned}$$

Sometimes we will simply write $\mathcal{A}_1 \times \mathcal{A}_2$ for this automaton. Note that δ is indeed a function in Q, so $\mathcal{A}$ is a well–defined DFA, and it has only reachable states. We have:

$$L_1 \subseteq L_2 \iff \forall (p,q) \in Q : p \in F_1 \Rightarrow q \in F_2 \tag{17.2}$$

$$L_1 \cap L_2 = \emptyset \iff \forall (p,q) \in Q : p \in F_1 \Leftrightarrow q \notin F_2 \tag{17.3}$$

The AMoRE-function `inclusion()`, which performs these tests, is described now.

Strategy of `inclusion()`

The strategy of this function is simply to visit all reachable states in a breadth-first-search and check if the right hand side of the equivalences (17.2) and (17.3) is fulfilled. A FIFO-mechanism is used to implement the search through the product automaton.

Details about `inclusion()`

For reasons of efficiency, `inclusion()` uses a coding of $Q_1 \times Q_2$ into integers. Therefore the product states can be accessed quickly and easily, the disadvantage is that this method only copes with DFA's with $|Q_1| \cdot |Q_2| <$ `maxint`, where `maxint` is the greatest integer usable in C. The coding for $(p,q) \in Q_1 \times Q_2$ is given by $\alpha(p,q) = 1 + p + q \cdot |Q_1|$.

The array `mark[]` of size $|Q_1| \cdot |Q_2|$ is used in two ways: firstly, to mark the product states that have already been visited, and secondly, to use these marks to keep the queue of states that have to be visited later on. So `mark`$[\alpha(p,q)] = 0$ indicates that (p,q) has not been visited yet (and might be unreachable). `mark`$[\alpha(p,q)] = \alpha(p',q')$ indicates that this state is reachable, and the next state in the FIFO-queue to be visited is (p',q'). Note that this implies by no means that (p',q') can be reached starting in (p,q). The indices `last` and `actuel` hold the positions in the array `mark` where the next state should be inserted, or where to continue respectively .

Therefore, this array is initialized with zero, except for the state (q_{01}, q_{02}) (or rather its coding), which is initialized with a value larger than the codings

[1] To be precise, this is rather the restriction of the product automaton to its reachable states

of possible states. This value serves as an end marker. `actuel` and `last` are initialized with the $\alpha(q_{01}, q_{02})$ to indicate that the FIFO-queue consists of that single value.

Then, all elements of this queue are traced. For every pair of states in the queue, the function computes all pairs of states that are reachable from there. In case they have not been visited yet, it checks the mentioned property concerning the final states and appends it to the queue using the index `last`, which indicates the current tail of the queue.

Chapter 18

The Language Operations

18.1 Simple Language Operations

The AMoRE-file `win_una.c` offers some functions that perform simple unary operations on regular languages. Let us start with their definition.

Definition 18.1 Let L be a language.

$$\begin{aligned}
L^* &= \bigcup_{i \geq 0} L^i \\
&= \{w_1 \dots w_n \,|\, n \in \mathbb{N}, w_1, \dots, w_n \in L\} \\
L^+ &= \bigcup_{i \geq 1} L^i \\
&= \{w_1 \dots w_n \,|\, n \geq 1, w_1, \dots, w_n \in L\} \\
L^c &= \Sigma^* \backslash L \\
rev(L) &= \{a_1 \dots a_n \,|\, n \in \mathbb{N}, a_n \dots a_1 \in L\} \\
pref(L) &= \{w \in \Sigma^* \,|\, \exists u \in \Sigma^* : wu \in L\} \\
suff(L) &= \{w \in \Sigma^* \,|\, \exists u \in \Sigma^* : uw \in L\} \\
min(L) &= \{w \in L \,|\, \forall u \in pref(\{w\}) \backslash \{w\} : u \notin L\} \\
max(L) &= \{w \in L \,|\, \forall u \in \Sigma^+ : wu \notin L\}
\end{aligned}$$

The next two Lemmas show that the class of regular languages is closed under these operations. Moreover they provide constructions for automata that accept the respective new language. We observe that for some of the operations the usage of NFA's is advantageous, whereas others require DFA's or even minimal DFA's. Recall that in an automaton a state q is called *sink state* iff $\delta(q, \Sigma) \subseteq \{q\}$. Since two non-final sink states are always equivalent, a minimal DFA cannot have more than one non-final sink state.

Lemma 18.2 Let $\mathcal{A} = (\Sigma, Q, I, \Delta, F)$ be an NFA. Define the NFA $(\mathcal{A})^* = (\Sigma, Q \cup \{q_{st}\}, \{q_{st}\}, \Delta', F')$, where q_{st} is a new state, $\Delta' = \Delta \cup \{(q_{st}, a, q) \mid \exists s \in I : (s, a, q) \in \Delta\} \cup \{(f, a, q) \in F \times \Sigma \times Q \mid \exists s \in I : (s, a, q) \in \Delta\}$ and $F' = F \cup \{q_{st}\}$. Then we have $L((\mathcal{A})^*) = (L(\mathcal{A}))^*$.

Lemma 18.3 Let $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ be a DFA.

Then for $\mathcal{A}^c := (Q, \Sigma, q_0, \delta, Q \backslash F)$ we have $L(\mathcal{A}^c) = (L(\mathcal{A}))^c$.

Let $min(\mathcal{A}) := (Q \backslash F \cup \{q_{fin}, q_{sink}\}, q_{st}, \delta_1, \{q_{fin}\})$ with new distinct states q_{fin} and q_{sink},

$$q_{st} = \begin{cases} q_0 & \text{if } q_0 \notin F \\ q_{fin} & \text{if } q_0 \in F. \end{cases}$$

$$\delta_1(q, a) = \begin{cases} \delta(q, a) & \text{if } q, \delta(q, a) \notin F, \\ q_{sink} & \text{if } q \in \{q_{sink}, q_{fin}\}, \\ q_{fin} & \text{if } \delta(q, a) \in F. \end{cases}$$

Then we have $L(min(\mathcal{A}) = min(L(\mathcal{A}))$.

Lemma 18.4 Let $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ be a minimal DFA. Let

$$max(\mathcal{A}) := \begin{cases} (\{q_0\}, \Sigma, q_0, q_0 \mapsto q_0, \emptyset) \\ \qquad \text{if there is no non-final sink state in } \mathcal{A} \\ (Q, \Sigma, q_0, \delta, \{q \in F \mid \delta(q, \Sigma) = \{q_{sink}\}\}) \\ \qquad \text{if there is a non-final sink state } q_{sink} \end{cases}$$

Then we have $L(max(\mathcal{A})) = max(L(\mathcal{A}))$.

If there is no non-final sink state in $\mathcal{A}$, then we have $pref(L(\mathcal{A})) = \Sigma^*$ and therefore we let $pref(\mathcal{A}) := (\{q_0, \Sigma, q_0, \{q_0\} \times \Sigma \times \{q_0\}, \{q_0\})$.

If there is a non-final sink state q_{sink} in $\mathcal{A}$ then we let $pref(\mathcal{A}) := (Q, \Sigma, q_0, \delta, F')$ with $F' = Q \backslash \{q_{sink}\}$

Then we have $L(pref(\mathcal{A})) = pref(L(\mathcal{A}))$.

Let $suff(\mathcal{A}) := (Q, \Sigma, Q, \Delta, F)$, that is the NFA obtained from $\mathcal{A}$ by making all states initial. We have $L(suff(\mathcal{A})) = suff(L(\mathcal{A}))$.

18.1.1 The Functions

All the above constructions are implemented in the AMoRE-file `win_una.c`. The implementation is in all cases straightforward. The functions that receive and return a pointer to an NFA are: `revnfa()` and `starnfa()`. The latter also receives a boolean parameter, that, if set to `FALSE`, makes the function perform the Kleene-plus operation.

The functions `compldfa()`, `minL()`, `maxL()` and `pref()` all receive and return a pointer to a DFA.

The function `suff()` receives a pointer to a DFA and returns a pointer to an NFA.

Additionally, `win_una.c` provides a primitive function `simpledfa()`, that receives a boolean parameter `full` and a number `sno`. `simpledfa()` returns a one-state DFA with `sno` letters in its alphabet, which accepts Σ^* iff `full==TRUE`. (Otherwise it accepts $\emptyset$).

These functions also allocate memory for the automata they produce.

18.2 Computations with two Languages

AMoRE provides several functions that build a new language out of two given languages. These functions are collected in the file `win_bin.c`. The binary operations are the following: union, concatenation, intersection, shuffle product, set difference and left/right quotient. These operations and the corresponding constructions will be described in this section.

18.2.1 Union

Let $A_1 = (\Sigma, Q_1, I_1, \Delta_1, F_1)$, $A_2 = (\Sigma, Q_2, I_2, \Delta_2, F_2)$ be some NFA's. We assume $Q_1 \cap Q_2 = \emptyset$. Then, for the NFA $A = (\Sigma, Q, I, \Delta, F)$ defined by $Q := Q_1 \cup Q_2$, $I := I_1 \cup I_2$, $\Delta := \Delta_1 \cup \Delta_2$, $F := F_1 \cup F_2$, we have $L(A) = L(A_1) \cup (A_2)$.

Implementation The implementation of this method is straightforward. In the resulting NFA the states of $\mathcal{A}_1$ have indices $0, \ldots, |Q_1| - 1$, the states of $\mathcal{A}_2$ have indices $|Q_1|, \ldots, |Q_1| + |Q_2| - 1$.

18.2.2 Concatenation

Let $A_1 = (\Sigma, Q_1, I_1, \Delta_1, F_1)$, $A_2 = (\Sigma, Q_2, I_2, \Delta_2, F_2)$ be some NFA's. We assume $Q_1 \cap Q_2 = \emptyset$. Then, for the NFA $A = (\Sigma, Q, I, \Delta, F)$ defined by $Q := Q_1 \cup Q_2$, $I := I_1$, $\Delta := \Delta_1 \cup \Delta_2 \cup \{(p, a, q) \in Q \times A \times Q \,|\, p \in F_1 \text{ and } (q_0, a, q) \in \Delta_2 \text{ for a } q_0 \in I_2\}$ and $F := \begin{cases} F_1 \cup F_2 & \text{if } I_2 \cap F_2 \neq \emptyset \\ F_2 & \text{otherwise} \end{cases}$, we have $L(A) = L(A_1) \cdot (A_2)$.

Implementation: This method is very simple, too. The states of the resulting NFA are indexed the same way as in the above section.

18.2.3 Intersection and Set Difference

In contrast to the above two methods, the calculation of the intersection and the set difference are implemented using DFA's. For the intersection, the determinism of the input automata is not essential, but useful because the output automaton is deterministic then, too. For treating the set difference, determinism is necessary.

Let $A_1 = (\Sigma, Q_1, q_{01}, \delta_1, F_1)$, $A_2 = (\Sigma, Q_2, q_{02}, \delta_2, F_2)$ be some DFA's. For the DFA $A = (\Sigma, Q, q_0, \delta, F)$ defined by $Q := Q_1 \times Q_2$, $I := I_1 \times I_2$, $\delta : (Q_1 \times Q_2) \times \Sigma \longrightarrow Q$, $((p,q),a) \mapsto (\delta_1(p,a), \delta_2(q,a))$, $F := F_1 \times F_2$, we have $L(A) = L(A_1) \cap L(A_2)$.

For the DFA $A' = (\Sigma, Q, q_0, \delta, F_2')$ with Q, I, δ as above and $F' := F_1 \times Q_2 \backslash F_2$ we have $L(A') = L(A_1) \backslash L(A_2)$.

This automaton $\mathcal{A}$ is the *product automaton*[1] of $\mathcal{A}_1$ and $\mathcal{A}_2$ and its transition function will sometimes be denoted simply by $\delta_1 \times \delta_2$.

Implementation: For the mapping of indices of pairs of states of the input automata into indices of the output automaton, `win_bin.c` contains a macro `pair()` that establishes the mapping $(i,j) \mapsto i * |Q_1| + j$.

The function `insecfa()` receives three parameters: the first and second are pointers to the input DFA's and the third is a boolean value `insec` that determines whether to compute the intersection (if `insec==TRUE`) or the set difference (otherwise).

18.2.4 Shuffle Product

For two languages L_1 and L_2, their shuffle product is

$$L_1 \sqcup\!\sqcup L_2 := \{u_1 w_1 \dots u_n w_n \,|\, n \geq 1,\ u_1 ... u_n \in L_1,\ w_1 ... w_n \in L_2,\ u_i, w_i \in \Sigma^*\}.$$

It is easy to see that the shuffle product is commutative, associative and distributes with the union.

The class of regular languages is closed under shuffle product. The following construction shows how to obtain an NFA for the shuffle product of two languages defined by two given NFA's.

Let $A_1 = (\Sigma, Q_1, I_1, \Delta_1, F_1)$, $A_2 = (\Sigma, Q_2, I_2, \Delta_2, F_2)$ be some NFA's. Let $A := (\Sigma, Q, I, \Delta, F)$ be the NFA defined by $Q := Q_1 \times Q_2$, $I := I_1 \times I_2$, $F := F_1 \times F_2$

$$\begin{aligned} \Delta \;&:=\; \{((p,q_1),a,(p,q_2)) \in (Q_1 \times Q_2) \times \Sigma \times (Q_1 \times Q_2) \,|\, (q_1,a,q_2) \in \Delta_2\} \\ &\cup\; \{((p_1,q),a,(p_2,q)) \in (Q_1 \times Q_2) \times \Sigma \times (Q_1 \times Q_2) \,|\, (p_1,a,p_2) \in \Delta_1\}, \end{aligned}$$

that is we allow the automaton to perform a transition in one of the input automata. With this definition we have $L(\mathcal{A}) = L(\mathcal{A}_1) \sqcup\!\sqcup L(\mathcal{A}_2)$.

Implementation: The implementation of this construction is as easy as the ones before. The indices of the resulting NFA are chosen the same way as in the product NFA in the implementation of 18.2.3. The function performing this construction is named `shuffle()`.

[1] Compare to section 17.2.2

18.2.5 Left Quotient

First we start with the definition for the left quotient.

Definition 18.5 For two languages L_1, L_2 the left quotient of L_2 by L_1 is defined as

$$L_1^{-1}L_2 := \{w \in \Sigma^* \mid \exists v \in L_1 : vw \in L_2\}.$$

The following construction shows that the class of regular languages is closed under left quotient.

Construction

Let $A_1 = (\Sigma, Q_1, I_1, \Delta_1, F_1)$, $A_2 = (\Sigma, Q_2, I_2, \Delta_2, F_2)$ be some NFA's. Let $A := (\Sigma, Q_2, I, \Delta_2, F_2)$, with

$$I := \{q \in Q_2 \mid \exists v \in \Sigma^* : v \in L(\mathcal{A}_1) \wedge q \in \delta_2(I, v)\} = \bigcup_{v \in L(\mathcal{A}_1)} \delta_2(I, v)$$

Here, δ denotes the multifunction associated with the transition relation as described in the section 1.3.

For this NFA $\mathcal{A}$ we have $L(\mathcal{A}) = (L(\mathcal{A}_1))^{-1}L(\mathcal{A}_2)$.

Algorithm

The above construction tells how to obtain an NFA for the left quotient out of two NFA's for the input languages: You simply have to modify the set of initial states. But the implementation of this construction is not as easy as in the cases before because for determining whether a particular state $q \in Q_2$ turns into an initial state you have to find out if there is a word $w \in L(\mathcal{A})$ with $q \in \delta_2(I, w)$.

The main idea of AMoRE's algorithm is to compute all reachable states in the product automaton $\mathcal{A}_1 \times \mathcal{A}_2$. Then the initial states of $\mathcal{A}$ are those states of $\mathcal{A}_2$ which are in a reachable pair together with a final state of $\mathcal{A}_1$. The algorithm is given in Figure 18.1.

To show that after termination of the first while-loop the set R contains the set of reachable states, we remark that the following condition is an invariant for this loop:

$$(\delta_1 \times \delta_2)(R \cup U, \Sigma^*) = (\delta_1 \times \delta_2)((I_1 \times I_2), \Sigma^*)$$

This approach would enable to compute an NFA for the left quotient out of two input NFA's with n_1 and n_2 states in time $\mathcal{O}(n_1 n_2 t k_1 k_2)$, where for $i = 1, 2$ k_i is the maximal number of states reachable from one state of $\mathcal{A}_i$ reading one fixed letter. Unfortunately, the data structure for the transition table of NFA's used in AMoRE does not allow to trace the neighbours of one state. All states have to be traced instead and checked if being connected appropriately.

$R := I_1 \times I_2$;
$U := I_1 \times I_2$;
while $U \neq \emptyset$ do
  for all $a \in \Sigma$ do
    select and delete any $(p, q) \in U$
    $U := U \cup ((\delta_1 \times \delta_2)((p, q), a) \backslash R)$
    $R := R \cup (\delta_1 \times \delta_2)((p, q), a)$
$I := \emptyset$
for all $(p, q) \in R$ do
  if $p \in F_1$ then
    $I := I \cup \{q\}$

Figure 18.1:

For this reason there are two implementations. One of them uses DFA's as inputs. Nevertheless, the resulting automaton might have several initial states and is therefore nondeterministic.

Details about the Implementation

Data Structures The implementation traces a breadth first strategy to compute the set of reachable states as in Figure 18.1. The FIFO storage for this purpose is established by a chained list of C-structures `*pairlist`. Each entry of such a list contains two numbers `info1` and `info2` that represent the indices of states in $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively. In order to be able to find out in constant time whether a certain pair of states already is in R the elements of R are not only in this list but there is also a boolean array `mark` with `mark[p][q]==TRUE` iff $(p, q) \in R$. [2]

During the algorithm, the variables `first` and `last` are pointers to the beginning and ending of the list. Conceptually, this list has two parts. The first part of this list holds the pairs in $R \backslash U$, the rest of the list contains the pairs in U. (Note that always $U \subseteq R$.) The variable `run` holds a pointer to the position in the list where U begins.

The variables `indfa1` and `indfa2` hold pointers to the input DFA's $\mathcal{A}_1$ and $\mathcal{A}_2$. The output NFA is pointed to by the variable `outnfa`.

Functions The file `win_bin.c` provides the function `leftquot()` that receives two pointers to the input DFA's and returns a pointer to the output NFA. The functions `pairinit()` and `pairinsert()` allow to deal comfortably with the mentioned list of pairs. `pairinit()` receives two addresses of pointers that are

[2] The reader may compare this data structure with the one used in 17.2.2 for the same purpose.

intended to indicate the beginning and ending of the list, and also two numbers. `pairinit()` then allocates memory for one C-structure `*list`, fills in the two numbers and makes both of the the pointers point to this item.

The function `pairinsert()` receives an address of a pointer to the end of the list and two numbers. It allocates memory for a new structure, fills in the two numbers and appends the new item to the list by modifying the former last item and updating the pointer to the end of the list afterwards.

18.2.6 Right Quotient

The definition of the right quotient is analogous to that of the left quotient:

Definition 18.6 For two languages L_1, L_2 the *right quotient* of L_2 by L_1 is

$$L_2 L_1^{-1} := \{w \in \Sigma^* \mid \exists v \in L_1 : wv \in L_2\}$$

The following Lemma states the close connection between left and right quotient and can be proved easily:

Lemma 18.7 Let L_1, L_2 be two languages. Then $L_2 L_1^{-1} = \overline{\overline{L_1}^{-1} \overline{L_2}}$, where $\overline{L}$ denotes the reverse language of L for any language L.

An immediate consequence of this Lemma is that the class of regular languages is closed also under right quotient because we know it is closed under left quotient and reversion. It yields also an easy algorithm for computing the right quotient effectively.

Bibliography

[AHU74] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, New York 1974.

[ADN92] A. Arnold, A. Dicky, M. Nivat, A note about minimal non-deterministic automata; in: Bulletin of the EATCS 47, June 1992, pp. 166-169.

[BS86] G.Berry, R.Sethi. From regular expressions to deterministic automata, Theor. Comput. Sci. 48 (1986), pp. 117–126.

[Br84] W. Brauer, Automatentheorie, Teubner, Stuttgart 1984.

[Brz63] J.A. Brzozowski: Canonical regular expressions and minimal state graphs for definite events, in: Proc. Symp. on Math. Theory of Automata, Vol. 12, Brooklyn, N. Y.: Brooklyn Polytechnic Institute, 1963, pp. 529 -561.

[ChH91] J.M. Champarnaud, G. Hansel, AUTOMATE, a computing package for automata and finite semigroups, J. Symbolic Computation **12** (1991), 197–220.

[Ei76] Eilenberg, Automata, Languages, and Machines, Academic Press, New York 1976.

[GJ79] M.R. Garey, D.S. Johnson: Computers and Intractability – A Guide to the Theory of NP-Completeness; Freeman, San Francisco, 1979.

[Gr51] J.A. Green, On the Structure of Semigroups, Annals of Math. 54 (1951), 163-172.

[Ho71] J.E.Hopcroft, An n log n algorithm for minimizing states in a finite automaton, Theory of Machines and Computations (Z.Kohavi, A.Paz, Eds.), Academic Press, New York 1971, pp. 189–196.

[HU79] J.E. Hopcroft, J.D. Ullman: Introduction to Automata Theory, Languages, and Computation; Addison-Wesley, Reading, Mass. 1979.

[In70] K. Indermark: Zur Zustandsminimierung nichtdeterministischer erkennender Automaten; GMD Seminarberichte Bd. 33, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin – Bonn, 1970.

[Ka91] T. Kahlert: Ein Verfahren für die Minimierung nichtdeterministischer endlicher Automaten und seine Implementierung; Diplomarbeit, Christian-Albrechts-Universität, Kiel, 1991.

[KW70] T. Kameda, P. Weiner: On the State Minimization of Nondeterministic Finite Automata; IEEE Trans. Comp. C–19 (1970), pp. 617-627.

[Ki74] J. Kim: State minimization of nondeterministic machines; IBM Thomas J. Watson Res. Center Rep. RC 4896, 1974.

[Kna84] R. Knast, A semigroup characterization of dot-depth one languages, RAIRO Inf. Theor. 17 (1983) 321-330.

[Knu73] D.E. Knuth, The Art of Computer Programming, Vol. III: Sorting and Searching, Addison-Wesley 1973, [pp. 455–457].

[LM90] G. Lallement, R. McFadden, On the determination of Green's Relations in finite transformation semigroups, J. Symbolic Computation (1990) 10, pp 481–498.

[Ma74] Z. Manna, Theory of Computation, McGraw-Hill, New York 1974.

[MP95] O. Matz, A. Potthoff, Computing small nondeterministic finite automata, Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Dpt. of CS., Univ. of Aarhus 1995, pp. 74–88

[MNP72] R.McNaughton, S.Papert, Counter-free automata, MIT Press, Cambridge, Mass. 1972.

[Pe90] D. Perrin, Finite Automata, in: Handbook of Theoretical Computer Science, Volume B (J.v.Leeuwen, Ed.), North Holland, Amsterdam, (1990), pp. 1–57.

[Pi86] J.E. Pin, Varieties of Formal Languages, Oxford Univ. Press, Oxford 1986.

[RW93] D.R. Raymond, D. Wood, Grail: Engineering Automata in C++, Rep. CS-93-01 (1993), Dpt. of CS. , Univ. of Waterloo.

[RDM*86] L.A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Sprirakis, A. Tuan, A browser for directed graphs, Software – Practice and Experience 17 (1986), 61–76.

[RS59] M.O. Rabin, D.Scott. Finite automata and their decision problems, IBM Journal of Research (1959), 3(2), pp. 115–125.

[Sch65] M.P. Schützenberger, On finite monoids having only trivial subgroups, Inform. Contr. 8 (1985) 190-194.

[Si75] I. Simon, Piecewise testable events, Proc. 2nd GI Conf., LNCS 33 (1975), 214-222.

[St85] J. Stern, Complexity of some problems from the theory of automata, Inform. Contr. 66 (1985), 163-176.

[STT81] K. Sugiyama, S. Tagawa, M. Toda, Methods for visual understanding of hierachical systems, IEEE Trans. Syst. Man Cybern. SMC-11 (1981), 109-125.

[Th81] W. Thomas, Remark on the star-height problem, Theor. Comput. Sci 13 (1981), 231-237.

Access to AMoRE and Copyright Note

The program AMoRE can be copied and used freely for any non-commercial purpose. Commercial use or modifications leading to commercial use are not allowed. If the user wishes to modify the code in any way, she or he is requested to supply a detailed documentation together with the changed code. AMoRE is distributed without any warranty. If a user publishes a result using the program AMoRE, we would appreciate AMoRE to be cited. You can obtain AMoRE per Anonymous FTP (`ftp.informatik.uni-kiel.de:pub/kiel/amore`).
For any questions or suggestions please write to

`amore@informatik.uni-kiel.de`

If you become user, please send your name and address to the above e-mail-address. Besides, we would like to inform users about future updates. Personal queries should be sent to

Heidi Luca-Gottschalk
System Administrator
Christian–Albrechts–Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Olshausenstraße 40
24098 Kiel
Germany
e-mail: `hlg@informatik.uni-kiel.de`

or

Prof. Dr. Wolfgang Thomas
Christian–Albrechts–Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Olshausenstraße 40
24098 Kiel
Germany
e-mail: `wt@informatik.uni-kiel.de`