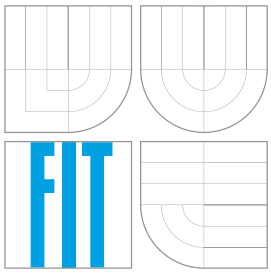
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SIMULACE KONEČNÝCH PŘEVODNÍKŮ

SIMULATION OF FINITE TRANSDUSERS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — BARBORA MICENKOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ROMAN LUKÁŠ, Ph.D.
SUPERVISOR

BRNO 2008

Zadání bakalářské práce

Řešitel:	Barbora Micenková
Obor:	Informační technologie
Téma:	Simulace konečných převodníků
Kategorie:	Překladače
Vedoucí:	Roman Lukáš, Ing., Ph.D., UIFS FIT VUT
Datum zadání:	1. listopadu 2007
Datum odevzdání:	14. května 2008

Pokyny:

1. Seznamte se podrobně s problematikou konečných převodníků.
2. Navrhněte algoritmus, který efektivně odsimuluje činnost konečného převodníku (obecně nedeterministického).
3. Vytvořte program, jehož vstupem bude vhodně popsaný konečný převodník a jeho výstupem vygenerovaný program v jazyce C, který po kompilaci a spuštění odsimuluje činnost konečného převodníku ze vstupu.
4. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Licenční smlouva je svázána s výtiskem, který je uložen v knihovně FIT.

Abstrakt

K rychlému překladu mezi strojovým kódem a assemblerem za účelem simulace je možné použít speciální abstraktní model – tzv. párový konečný automat. Jeho vnitřní uspořádání nás přivádí k problematice konečných převodníků. Vzhledem k tomu, že simulace deterministických převodníků je efektivnější, musíme se procesem determinizace zabývat. Existující algoritmy jsou bohužel aplikovatelné pouze na převodníky provádějící překlad konečných jazyků, zatímco my na vstupu očekáváme obecně nekonečný jazyk. Proto je nutné nalézt způsob, jak rychle rozpoznat, je-li převodník na vstupu determinizovatelný. V této bakalářské práci jsou shrnuty doposud publikované poznatky z oblasti determinizace konečných převodníků a rovněž navržen nový algoritmus determinizace převodníků provádějících překlad obecně nekonečných jazyků. Nedeterminizovatelné převodníky na vstupu jsou detekovány.

Klíčová slova

konečný převodník, determinizace, determinizovatelnost, párový konečný automat, HW/SW co-design

Abstract

A quick translation between binary code and assembler for the purpose of simulation can be done by a special abstract model – a two-way coupled finite automaton. Its inner representation brings us to the question of finite transducers. As we know that simulation of deterministic transducers is more efficient, we have to concern ourselves with that process. Unfortunately, the existing algorithms are applicable just for transducers translating finite languages while we expect generally infinite language on input. Therefore it is important to find a way how to quickly detect the determinizability of the input transducer. In this bachelor's thesis, so far published works on the determinizability of finite transducers are brought together and a new algorithm of determinization of transducers translating generally infinite languages is presented. Undeterminizable transducers on input are detected.

Keywords

finite transducer, determinization, determinizability, two-way coupled finite automaton, HW/SW co-design

Citace

Barbora Micenková: Simulation of Finite Transdusers, bakalářská práce, Brno, FIT VUT v Brně, 2008

Simulation of Finite Transdusers

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing. Romana Lukáše, Ph.D.

Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Barbora Micenková
May 13, 2008

Poděkování

Velmi děkuji panu Ing. Romanu Lukáši, Ph.D. za ochotnou pomoc při vytváření této práce, zajímavé konzultace a podnětné připomínky.

Věnováno rodině a přátelům.

Contents

List of Figures

Chapter 1

Introduction

There are many methods how to describe a formal language. One of them is defining it by means of an *abstract machine*. In this work I would like to present a classical computational model which defines translation – a finite transducer. A finite transducer can be viewed as an extended finite automaton that can read from the input tape and also write on the output tape. We can say it *transduces* the contents of its input tape to its output tape. In general it computes a relation between two formal languages. Finite transducers also possess computational abilities like addition, subtraction or modular division and memory [10]. They are important for investigation of computation because they underlie many computer programs which translate input data to output data. Typically, they are used in morphological analysis in natural language processing.

1.1 Motivation

Particularly, this work is intended to contribute to research in the field of hardware/software co-design that is in progress at the Faculty of Information Technology, Brno University of Technology. A partial goal of the project is to create and implement a language named *Instruction Set Architecture C* (ISAC) which would be used for description of microprocessor architecture. Apart from that it is necessary to create a development environment which would provide development of both software tools and simultaneously microprocessor hardware. Due to concurrent work on hardware and software (hardware/software co-design), the total time of the development will be reduced. The project consists of several parts. The work on the project is concentrated on the design of typical constructions of the description language and on implementation of software tools (compiler, universal assembler, disassembler, linker and simulator).

Significant results have already been achieved in the part of the project based on the formal languages theory. A new model for assembler and disassembler – *two-way coupled finite automata* – has been invented by Dr. Roman Lukáš and further described in [13].

Two way coupled finite automata

The model is composed of two finite automata – an input and an output one. They are coupled in the following way: The input automaton reads the input string and controls the output automaton. The output automaton finally generates the output string. More exactly, the way in the output automaton is set according to the sequence of rules used during reading the input by the input automaton. This means we have to handle the input

automaton as a transducer because we need to execute some semantic actions attached to each rule. Here we approach the title of this thesis – simulation of finite transducers.

1.2 Deterministic translation

Generally, there are two possibilities of the simulation of a finite transducer:

1. to simulate the original transducer by state space search
2. to determinize the transducer and then simulate it.

Deterministic translation is always more efficient and that is why we will require it. Therefore, before the simulation we have to run an algorithm of determinization. There has been published such an algorithm but serious deficiencies have been found. I will try to draft out and solve these problems in this thesis.

1.3 Structure of the thesis

Let me describe the chapters of this bachelor's thesis briefly. *First chapter* is an introduction to the problems brought up by this thesis and explains my motivation to have written it. *Second chapter* apprises the reader of the fundamental models of the theory of formal languages and brings necessary definitions. *Third chapter* gives a summary of the algorithm for determinization of finite transducers and figures out its deficiencies. In *fourth chapter*, there is shown an existing algorithm to determine whether a finite transducer is determinizable. *Fifth chapter* presents my own suggestions on determining the determinization. *Sixth chapter* briefly informs on implementation of the functional determinization algorithm and the last *seventh chapter* recapitulates the results of the work and presents new challenges.

Chapter 2

Preliminaries

In this chapter I would like to sum up the basic ideas of the theory of formal languages as well as some more difficult principles necessary for further comprehension of the text. Nevertheless I assume the reader is acquainted with the fundamentals of the theory and wield terms like *alphabet*, *string*, string operations *(length, concatenation, power, reversal, prefix, suffix, substring)* and automata. Most of the definitions were adopted from [6, 11] and adjusted to the context.

2.1 Lazy finite automata

The only difference between finite automata and lazy finite automata is in the form of transitions. Note that finite automata can read only one symbol from the input tape in one step while lazy finite automata are able to read a string in one step.

Definition 2.1. *A* lazy finite automaton *is a quintuple,* $M = (Q, \Sigma, R, s, F)$ *where*

- Q *is a finite set of states,*
- Σ *is an input alphabet,*
- $R \subseteq Q(\Sigma^*) \times Q$ *is a transition relation,*
- $s \in Q$ *is the start state of* M*,*
- $F \subseteq Q$ *is a set of final states.*

Members of R are called *rules* and thus R a *finite set of rules.* The rule is usually written as $px \to q$, where $p, q \in Q$ and $x \in \Sigma^*$.

Definition 2.2. *A* configuration *of a lazy finite automaton*$M = (Q, \Sigma, R, s, F)$ *is a string* $\chi = px$ *where* $p \in Q$ *and* $x \in \Sigma^*$.

Definition 2.3. *Let* pxy *and* qy *be two configurations of a lazy finite-automaton* M *where* $p, q \in Q$ *and* $x, y \in \Sigma^*$*. Let* $r = px \to q \in R$ *be a rule. Then* M *makes a* move *from* pxy *to* qy *according to* r*, written as* $pxy \vdash qy\,[r]$ *or, simply,* $pxy \vdash qy$.

Definition 2.4. *Let* M *be a lazy finite automaton:*

- *Let* χ *be a configuration,* M *makes* zero moves *from* χ *to* χ*; in symbols* $\chi \vdash \chi[\varepsilon]$ *or, simply,* $\chi \vdash \chi$.

- *Let* $\chi_0, \chi_1, ..., \chi_n$ *be a sequence of configurations, where* $n \geq 1$*, and* $\chi_{i-1} \vdash \chi_i\,[r_i]$*, where* $r_i \in R$*, for all* $i = 1, ..., n$*. Then* M *makes* n moves *from* χ_0 *to* χ_n*, written as* $\chi_0 \vdash^n \chi_n\,[r_1 ... r_n]$ *or, simply,* $\chi_0 \vdash^n \chi_n$*.*
- *If* $\chi_0 \vdash^n \chi_n[\varrho]$ *for some* $n \geq 1$*, then we write* $\chi_0 \vdash^+ \chi_n[\varrho]$*.*
- *If* $\chi_0 \vdash^n \chi_n[\varrho]$ *for some* $n \geq 0$*, then we write* $\chi_0 \vdash^* \chi_n[\varrho]$*.*

Definition 2.5. *Let* $M = (Q, \Sigma, R, s, F)$ *be a lazy finite automaton. The* language $L(M)$ *accepted by* M *is defined as* $L(M) = \{w : w \in \Sigma^*, sw \vdash^* f, f \in F\}$*.*

Definition 2.6. *Let* M *be a lazy finite automaton. We say* M *is* unambiguous *if for each* $x \in L(M)$ *there exists exactly one sequence of rules* ϱ *such that* $sx \vdash^* f[\varrho], f \in F$*.*

2.2 Lazy finite transducers

A finite transducer is a generalization of the Mealy machines [12]. It can be seen as a finite-state automaton with output. Each edge is labeled by a pair of symbols.

Definition 2.7. *A* lazy finite transducer *is a sextuple,* $M = (Q, \Sigma, \Omega, R, s, F)$ *where*

- Q *is a finite set of states,*
- Σ *is an input alphabet,*
- Ω *is an output alphabet,*
- $R \subseteq Q(\Sigma^*) \times (\Omega^*)Q$ *is a transition relation,*
- $s \in Q$ *is the start state of* M*,*
- $F \subseteq Q$ *is a set of final states.*

Members of R are called *rules* and thus R a *finite set of rules.* The rule is usually written as $px \rightarrow yq$ where $p, q \in Q$, $x \in \Sigma^*$ and $y \in \Omega^*$.

Definition 2.8. *A* configuration *of a lazy finite transducer* $M = (Q, \Sigma, R, s, F)$ *is a string* vpu*, where* $p \in Q$*,* $u \in \Sigma^*$ *and* $v \in \Omega^*$*.*

Definition 2.9. *Let* $vpxu$ *and* $vyqu$ *be two configurations of* M *where* $p, q \in Q$*,* $x, u \in \Sigma^*$ *and* $v, y \in \Omega^*$*. Let* $r = px \rightarrow yq \in R$ *be a rule. Then* M *makes a* move *from* $vpxu$ *to* $vyqu$ *according to* r*, written as* $vpxu \vdash vyqu[r]$ *or, simply,* $vpxu \vdash vyqu$*.*

Definition of the *sequence of moves* is the same as for lazy finite automata (2.4).

Definition 2.10. *A* translation $T(M)$ *defined by a lazy finite transducer* M *is*

$$T(M) = \{(x, y) : sx \vdash^* yf, x \in \Sigma^*, y \in \Omega^*, f \in F\}.$$

An input language corresponding to $T(M)$ *is defined as*

$$L_I(M) = \{x \colon (x, y) \in T(M) \text{ for some } y \in \Omega^*\}.$$

An output language corresponding to $T(M)$ *is defined as*

$$L_O(M) = \{y \colon (x, y) \in T(M) \text{ for some } x \in \Sigma^*\}.$$

2.3 Special types of finite transducers

Here I am going to introduce some useful types of finite transducers that will be discussed later on.

Definition 2.11. *Let M be a lazy finite transducer. M is* unambiguous *if for each $(x, y) \in T(M)$ there exists exactly one sequence of rules ϱ such that $sx \vdash^* yf[\varrho], f \in F$ and there exists no z and ϱ' such that $sx \vdash^* zf'[\varrho'], f' \in F$ and $\varrho \neq \varrho'$.*

In other words, a transducer is unambiguous if for any string $x \in \Sigma^*$, there is at most one successful path – it means a path from the start state to one of the final states.

Definition 2.12. *Let $M = (Q, \Sigma, \Omega, R, s, F)$ be a lazy finite transducer. M is a* finite-state transducer *if for each $px \to yq \in R$ holds $x \in \Sigma \cup \{\varepsilon\}$.*

Theorem 2.13. *For every lazy finite transducer M_1 there exists a finite transducer M_2 such that $T(M_1) = T(M_2)$.*

Definition 2.14. *Let $M = (Q, \Sigma, \Omega, R, s, F)$ be a finite-state transducer. M is an* ε-free finite-state transducer *if $card(F) = 1$ and each rule from R is of the form $pa \to yq$, where $p, q \in Q - F$, $a \in \Sigma$, $y \in \Omega^*$ or $p \to yf$, where $p \in Q - F$, $y \in \Omega^*$, $f \in F$.*

Theorem 2.15. *For every unambiguous finite transducer $M_1 = (Q_1, \Sigma, \Omega, R_1, s, F_1)$ there exists an ε-free finite transducer $M_2 = (Q_2, \Sigma, \Omega, R_2, s, F_2)$ such that $T(M_1) = T(M_2)$.*

The usual definition of a deterministic finite transducer is not convenient for us so we will use a different one:

Definition 2.16. *Let $M = (Q, \Sigma, \Omega, R, s, F)$ be an ε-free finite-state transducer. M is a* strictly deterministic finite transducer *if for each $a \in \Sigma$ and $p \in Q$ there exists no more than one rule of the form $pa \to yq$, where $p, q \in Q - F$, $y \in \Omega^*$, and for each $p \in Q$ there exists no more than one rule of the form $p \to yf$, where $f \in F$ and $y \in \Omega^*$.*

Theorem 2.17. *For every unambiguous ε-free finite transducer M_1 where $T(M_1)$ is a finite relation there exists a strictly deterministic finite transducer M_2 such that $T(M_1) = T(M_2)$.*

The three previous definitions describe transducers that will be used during the determinization process. According to theorems 2.13, 2.15, 2.17 we can subsequently convert a lazy finite transducer to a strictly deterministic finite transducer which will be shown in the next chapter.

Chapter 3

Determinization of finite transducers

As outlined above, the simulation of a general transducer could be very inefficient if we used a method of state space search. That is why we want the input transducer to be deterministic and thus easier to simulate. In this way my work reassumes bachelor's thesis of Eva Zámečníková [13] dealing with the determinization of finite transducers. I am going to outline her work in this chapter and afterwards show the deficiencies of the algorithm she presented.

The *process of determinization* has three phases:

1. conversion from a lazy finite transducer to a finite transducer,
2. conversion from a finite transducer to an ε-free finite transducer,
3. conversion from an ε-free finite transducer to a strictly deterministic finite transducer.

In the following sections I offer descriptions of the single phases.

3.1 Conversion from a lazy finite transducer to a finite transducer

With regard to the character of the problem, we have to assume a lazy finite automaton on input. Fortunately, we can conveniently convert a lazy finite transducer to a finite transducer without any loss of power by theorem 2.13. According to their definitions, the difference between a lazy finite transducer and a finite transducer is in the way they read the input. A lazy finite transducer reads a whole string in one step while a finite transducer can only read one symbol at most. As a result, after the conversion there can be more states and rules in the new transducer.

The idea of the conversion is simple: Assume the original transducer has a rule $px \rightarrow yq$, $x \in \Sigma^*$, $|x| > 1$, then we only read the first symbol of x, move to a new state and generate an empty string ε on output. As soon as we read the last symbol of x we can write y on the output and move to the state q.

Provided $|x| = 1$, we can make a normal move.

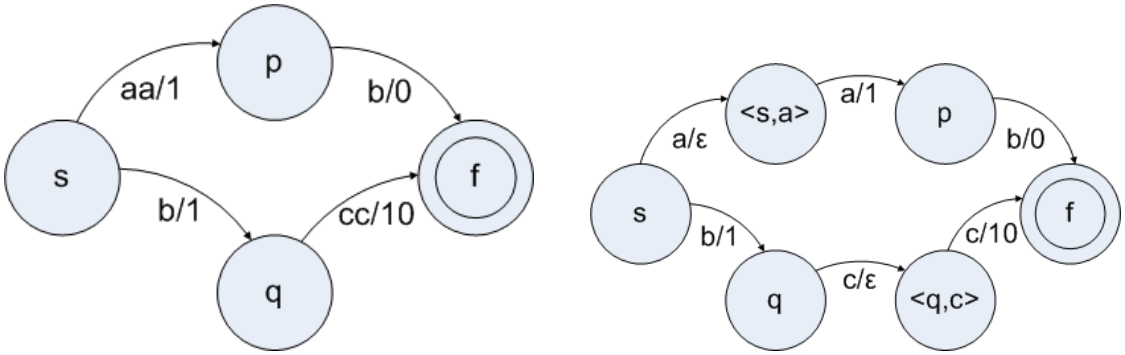


Figure 3.1: Lazy finite transducer and its corresponding finite transducer

The formal algorithm of conversion

- Input: Lazy finite transducer $M_1 = (Q_1, \Sigma, \Omega, R_1, s, F)$
- Output: Finite transducer $M_2 = (Q_2, \Sigma, \Omega, R_2, s, F)$, such that $T(M_1) = T(M_2)$
- Method:
  $Q_2 := Q_1$;
  for each $px \to yq \in R_1$ **do**
  if $|x| \leq 1$ **then**
  add $px \to yq$ to R_2
  else
  let $x = a_1 a_2 \dots a_n$, where $n \geq 2$:
  add $\langle p, a_1 \rangle$, $\langle p, a_1 a_2 \rangle$, $\dots$, $\langle p, a_1 a_2 \dots a_{n-1} \rangle$ to Q
  add $pa_1 \to \langle p, a_1 \rangle$, $\langle p, a_1 \rangle a_2 \to \langle p, a_1 a_2 \rangle$, $\dots$, $\langle p, a_1 a_2 \dots a_{n-2} \rangle a_{n-1} \to \langle p, a_1 a_2 \dots a_{n-1} \rangle$,
  $\langle p, a_1 a_2 \dots a_{n-1} \rangle a_n \to yq$ to R_2

3.2 Conversion from a finite transducer to an ε-free finite transducer

Definition 2.16 does not allow ε-transitions leading to any states except the final state in a strictly deterministic finite transducer. We just do not need them for deterministic translation. On the other hand, they are sometimes necessary for the transitions leading to the final states as we will see later in Figure 3.5. The figure depicts a transducer that would not be determinizable if we didn't use ε-transitions to the final state.

Computation of an ε-closure

Definition 3.1. *Let* $M = (Q, \Sigma, \Omega, R, s, F)$ *be a finite transducer and* $q \in Q$. *Then,* ε*-closure*(q) *is defined as:*

$$\varepsilon\text{-}closure(q) = \{(p, y) : q \vdash^* yp, p \in Q, y \in \Omega^*\}.$$

In other words, ε-closure(q) is a set of all states which are accessible from q by ε-transitions.

We will need to compute an ε-closure for all states to form an ε-free finite transducer. The algorithm for computing it is very simple. Actually, we just check all edges going from p and search for ε-transitions. If there is some, e.g. leading to state q, we add the pair $\langle q$, output string$\rangle$ to ε-closure(p) and continue in searching for ε-transitions coming out from q. If we find some again, we attach the output to the output string saved in q and put the pair $\langle$new state, concatenated output$\rangle$ to ε-closure(p) and continue in the same way. During computation we have to check unambiguity. The transducer is ambiguous if there exists $p \in Q$ and different $u, v \in \Omega^*$ such that $q \vdash^* up$ and $q \vdash^* vp$.

The formal algorithm of computation of an ε-closure

- Input: Lazy finite transducer $M_1 = (Q_1, \Sigma, \Omega, R_1, s, F)$; $q \in Q$
- Output: ε-closure(q) or error of ambiguity
- Method:
  $S_{undone} := \{(q, \varepsilon)\}$;
  $S_{done} := \emptyset$;
  while $S_{undone} \neq \emptyset$ **do begin**
    let $(p, v) \in S_{undone}$:
    if exists $w \in \Omega^*, w \neq v$ **such that** $(p, w) \in S_{done}$ **then**
      error(ambiguity)
    $S_{undone} := S_{undone} - \{(p, v)\}$;
    $S_{done} := S_{done} \cup \{(p, v)\}$;
    $S_{undone} := S_{undone} \cup \{(t, vy) : p \to yt \in R\}$;
  end
  ε-closure(q) := S_{done};

The formal algorithm of conversion

Conversion from finite transducer to ε-free finite transducer

- Input: Finite transducer $M_1 = (Q_1, \Sigma, \Omega, R_1, s, F_1)$, ε-closure(q) for all $q \in Q$
- Output: ε-free finite transducer $M_2 = (Q_2, \Sigma, \Omega, R_2, s, F_2)$ such that $T(M_1) = T(M_2)$
- Method:
  $Q_2 := Q_1 \cup \{f_2\}$;
  $R_2 := \emptyset$;
  $F_2 := \{f_2\}$;
  for each $q \in Q$ **do**
    $R_2 := R_2 \cup \{qa \to xyt : pa \to yt \in R_1, (p, x) \in \varepsilon\text{-closure}(q), a \in \Sigma, t \in Q\} \cup$
      $\{q \to xf_2 : (f, x) \in \varepsilon\text{-closure}(q), f \in F\}$;

3.3 Conversion from an ε-free finite transducer to a strict deterministic transducer

Finally, I am going to explain the most difficult part of the whole conversion. We can say there are three different situations that must be faced up during the determinization process.

- The simpliest one is that there is only one edge coming out of a state with a particular label. The situation turns up when reading symbol a in Figure 3.2. Then we just read the symbol and execute the semantic action attached to that edge.
- Second situation occurs if there are two or more edges with the same label coming out of one state. In that case we create a new state which contains a set of pairs ⟨state, semantic action⟩ and do not generate any output, just an empty string. The situation is demonstrated in Figure 3.2 by reading symbol b.
- Third and most difficult situation comes up if there are more states reachable from two or more "substates" of a composed state by reading the same symbol. In Figure 3.2 it is illustrated by label c. If c is read, there cannot be generated any output as we do not know which state we accessed. We have to remember the saved "to-be-generated" string plus the new output. Therefore we create a new composed state that consists of pairs ⟨state, concatenated string⟩.

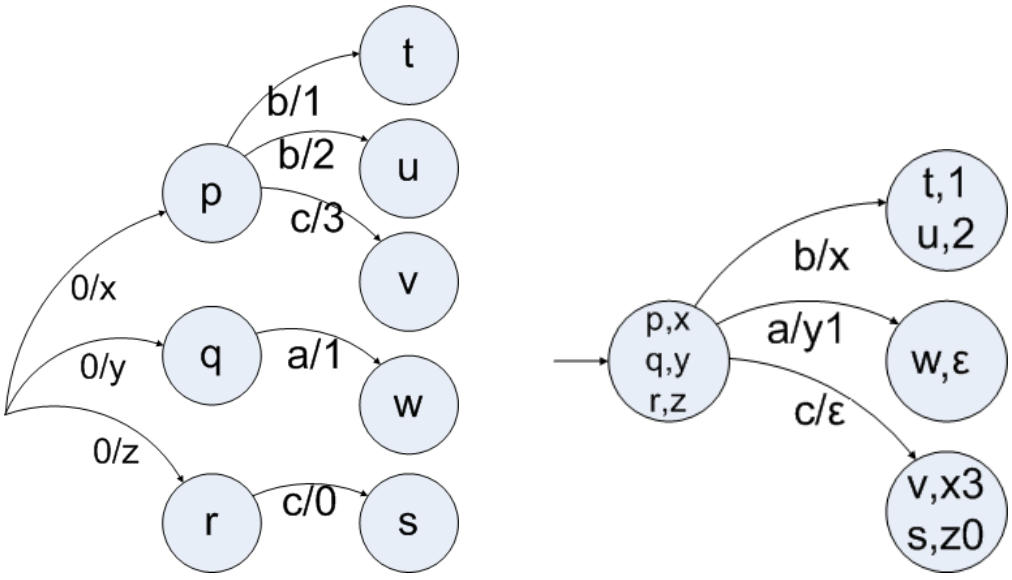


Figure 3.2: Determinization process.

The formal algorithm of determinization

Now I will present the formal algorithm of determinization but remember that it is not applicable to all finite transducers.

- Input: ε-free finite transducer $M_1 = (Q_1, \Sigma, \Omega, R_1, s_1, \{f_1\})$
- Output: Deterministic finite transducer $M_2 = (Q_2, \Sigma, \Omega, R_2, s_2, \{f_2\})$ such that $T(M_1) = T(M_2)$, or error if M_1 is ambiguous
- Method:
  $s_2 := \{\langle s_1, \varepsilon\rangle\}$;
  $f_2 := \{\langle f_1, \varepsilon\rangle\}$;
  $S_{undone} := \{s_2\}$;
  $S_{done} := \emptyset$;
  while $S_{undone} \neq \emptyset$ **do begin**
    let $X \in S_{undone}$:
    $S_{undone} := S_{undone} - \{X\}$;
    $S_{done} := S_{done} \cup \{X\}$;
    for each $a \in \Sigma \cup \{\varepsilon\}$ **do**
      $Q_{in} := \{\langle p, z\rangle : \langle p, z\rangle \in X, pa \to yq \in R_1, y, z \in \Omega^*, q \in Q_1\}$;
      if $|Q_{in}| = 1$ **then**
        let $\langle p, z\rangle \in Q_{in}$:
        $Q_{out} := \{\langle q, y\rangle : q \in Q_1, y \in \Omega^*, pa \to yq \in R_1\}$;
        if $|Q_{out}| = 1$ **then**
          let $\langle q, y\rangle \in Q_{out}$:
          $Q_{out} := \{\langle q, \varepsilon\rangle\}$;
          add $Xa \to zyQ_{out}$ to R_2;
        if $|Q_{out}| > 1$ **then**
          add $Xa \to zQ_{out}$ to R_2;
      if $|Q_{in}| > 1$ **then**
        $Q_{out} := \{\langle q, zy\rangle : \langle p, z\rangle \in X, pa \to yq \in R_1, y, z \in \Omega^*, q \in Q_1\}$;
        add $Xa \to Q_{out}$ to R_2;
      $S_{undone} := S_{undone} \cup \{Q_{out}\}$;
      if exists $q \in Q_1$ **such that** $|\{y : \langle q, y\rangle \in Q_{out}\}| > 1$ **then** error(ambiguity)
  end
  $Q_2 := S_{done}$;

The algorithm was invented by Eva Zámečníková and presented in her bachelor's thesis [13]. It can detect ambiguity of the input transducer and also prefix code and it is very quick, but unfortunately we cannot use it in practise to convert a general transducer.

Finitness of the algorithm

The problem is that the algorithm as presented above is functional just for conversions of transducers translating finite languages. If there was a transducer accepting infinite language on input, the algorithm might fail. In Figure 3.3, there is a transducer that cannot be determinized and, moreover, when we try to determinize it by the outlined algorithm, it starts to cycle. Figure 3.4 depicts an attempt to determinize the transducer.

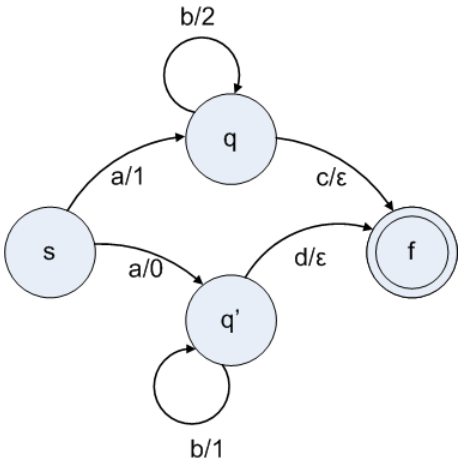


Figure 3.3: An undeterminizable finite transducer.

Example 3.1: Let's try to simulate the algorithm of determinization for the transducer in Figure 3.3. I will write down the contents of the sets $S_{undone}, S_{done}, Q_{in}$ and Q_{out} in each of the first few steps.

1. step:
   $S_{undone} = \{< s, \varepsilon >\}$
   $S_{done} = \{\}$
   $Q_{in} = \{< s, \varepsilon >\}$
   $Q_{out} = \{< q, 1 >, < q', 0 >\}$
2. step:
   $S_{undone} = \{\{< q, 1 >, < q', 0 >\}\}$
   $S_{done} = \{< s, \varepsilon >\}$
   $Q_{in} = \{< q, 1 >, < q', 0 >\}$
   $Q_{out} = \{< q, 12 >, < q', 01 >\}$
3. step:
   $S_{undone} = \{\{< q, 12 >, < q', 01 >\}\}$
   $S_{done} = \{\{< q, 1 >, < q', 0 >\}, < s, \varepsilon >\}$
   $Q_{in} = \{< q, 12 >, < q', 01 >\}$
   $Q_{out} = \{< q, 122 >, < q', 011 >\}$
4. step:
   $\vdots$

As early as in the third step we can see that the process will never end!

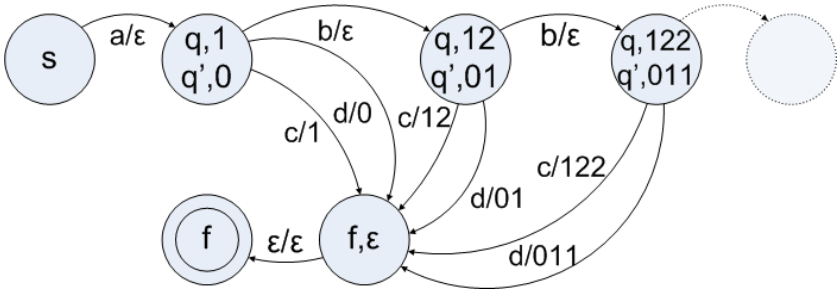


Figure 3.4: An attempt to determinize the transducer from Figure 3.3. Determinization process will never end.

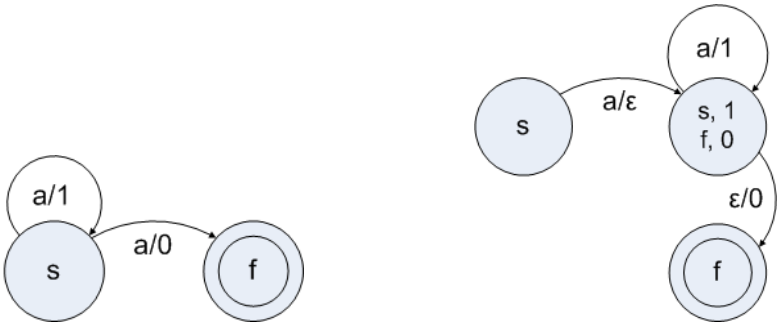


Figure 3.5: Illustration of the need of an ε-transition leading to the final state. The right transducer is a deterministic version of the left one.

3.4 Determinizability

As our algorithm is not powerful enough to determinize all ε-free finite transducers, it is necessary to either *increase its power* or find a way how to *detect whether an input finite transducer is determinizable* by the presented algorithm and thereby guarantee finitness of the program.

In this work I will deal with the option of determining the determinizability of an input transducer with consideration to further extension and increase of the power of the own determinization algorithm.

In Chapter 5 I will describe some new approaches to the determination of determinizability, but first I would like to outline the latest method published by Cyril Allauzen and Mehryar Mohri.

Chapter 4

Determining the determinizability of finite transducers

The complexity of the known methods for determining the determinizability is always on the order of the n-th power of the number of states, mostly for high n. These methods are not guaranteed to give an answer in a satisfactory period of time and therefore are not useful. It would be more convenient to run the process of determinization and declare the transducer undeterminizable if it didn't halt within a required period of time. Such a method is a obvious waste of computer resources as well.

So far, the most reasonable solution was given by Cyril Allauzen and Mehryar Mohri in their latest works [1, 8, 2] concerning determinization of weighted finite automata and weighted and unweighted finite transducers. They present a new algorithm deciding the determinizability *before the own process of determinization is run* on the base of determining so called *"twins property"*.

4.1 The twins property

The twins property has been known since 1970s [4] and it is a necessary and sufficient condition for functional finite transducers to be determinizable.

A necessary but not sufficient condition for the input transducer is to be *functional* which means it associates at most one string to any input string. Unambiguous transducers are functional. But note it may be the case that the transducer is ambiguous and still functional. We will not concern ourselves with the problem of *functionality/sequentiality/ambiguity* in detail as it is well described in [3]. We will deal with the test of functionality later on.

Definition 4.1. *A* semiring *is a system* $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$ *such that*

1. $(\mathbb{K}, \oplus, \bar{0})$ *is a commutative monoid with* $\bar{0}$ *as the identity element for* $\oplus$,
2. $(\mathbb{K}, \otimes, \bar{1})$ *is a monoid with* $\bar{1}$ *as the identity element for* $\otimes$,
3. $\otimes$ *distributes over* $\oplus$: *for all* a, b, c *in* $\mathbb{K}$,
$$\begin{aligned}(a \oplus b) \otimes c &= (a \otimes c) \oplus (b \otimes c),\\ c \otimes (a \oplus b) &= (c \otimes a) \oplus (c \otimes b),\end{aligned}$$
4. $\bar{0}$ *is an annihilator for* $\otimes$: $\forall a \in \mathbb{K}$, $a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$ [9]

A semiring $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$ is said to be *commutative* when the multiplicative operation $\otimes$ is commutative:

$$\forall a, b \in \mathbb{K}, a \otimes b = b \otimes a.$$

A *left semiring* is a semiring where $\otimes$ distributes over $\oplus$ in a following way:

$$c \otimes (a \oplus b) = (c \otimes a) \oplus (c \otimes b).$$

The *string semiring* is a left semiring $(\Sigma^* \cup \{\infty\}, \wedge, \cdot, \infty, \varepsilon)$ where $\wedge$ denotes the longest common prefix operation, $\cdot$ concatenation and ∞ a new element such that for any string $w \in (\Sigma^* \cup \{\infty\}), w \wedge \infty = \infty \wedge w = w$ and $w \cdot \infty = \infty \cdot w = \infty$.

A semiring is said to be *left divisible* if for any $x \neq \bar{0}$, there exists $y \in \mathbb{K}$ such that $y \otimes x = \bar{1}$, that is if all elements of $\mathbb{K}$ admit a left inverse.

Now I will present some new terminology so that the next text is easier to read. According to the definition of finite transducer 2.7, $r \in R$ is a transition. We denote its input label by $i[r]$ and its output label by $o[r]$. A *path* π is an element of R^* with consecutive transitions. We denote $P(q, x, q')$ the set of paths from q to q' with input string $x \in \Sigma^*$.

Definition 4.2. *Let T be a finite transducer defined over a divisible string semiring. Two states q and q' are said to be siblings if there exist two strings $y, x \in \Sigma^*$ such that both q and q' can be reached from the initial state by paths labeled with y and there is a cycle at q and a cycle at q' both labeled with x. Two sibling states are said to be twins iff for any paths $\pi_1 \in P(s, y, q)$, $\pi_2 \in P(q, x, q)$, $\pi_1' \in P(s, y, q')$, $\pi_2' \in P(q', x, q')$,*

$$o[\pi_1]^{-1} o[\pi_1'] = (o[\pi_1] o[\pi_2])^{-1} o[\pi_1'] o[\pi_2']. \tag{4.1}$$

Finite transducer has a twin property as long as every two siblings of the transducer are twins.

According to the definition, an acyclic transducer has the twins property [8].

Example 4.1: Figure 4.1(a) describes a simple transducer with siblings states (the output is not important here). The second Figure, 4.1(b), shows a transducer with twins property. By Definition 4.2 $o[\pi_1]^{-1} o[\pi_1'] = 1^{-1}1 = \varepsilon$ and $(o[\pi_1] o[\pi_2])^{-1} o[\pi_1'] o[\pi_2'] = (12)^{-1}12 = \varepsilon$. The expressions are equal and therefore the siblings states are twins.

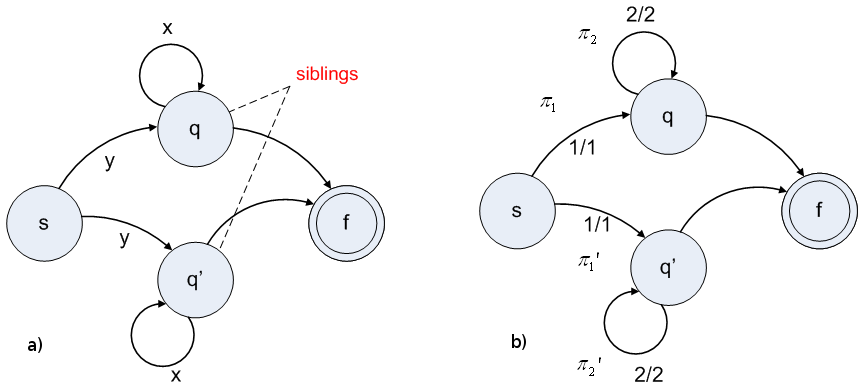


Figure 4.1: Transducers with siblings and twins

4.2 Use of composed transducers

Mohri and Allauzen present an algorithm [1] comparing all the paths that share the same input, by building a transducer, which is composed of a finite transducer with its inverse. The loops in the construction have a special property and that is the fact the power of the algorithm lies in.

The *inverse* T^{-1} of a transducer T is obtained by swapping its input and output labels of each transition. The states remain the same.

Now we will look at the definition of a *composed transducer* [3, 2].

Definition 4.3. *Given a transducer* $T = (Q, \Sigma, \Omega, R, s, F)$*, the cross product of* T^{-1} *and* T *is a* composed transducer*:*

$$T^{-1} \times T = (Q \times Q, \Sigma', \Omega, R', (s, s), F \times F)$$

where $\Sigma' = \Omega$ *and the transition set* R' *is defined by*

$$R' = \{(p_1, p_2)a \to c(q_1, q_2) : p_1 b \to aq_1, p_2 b \to cq_2 \in R\}.$$

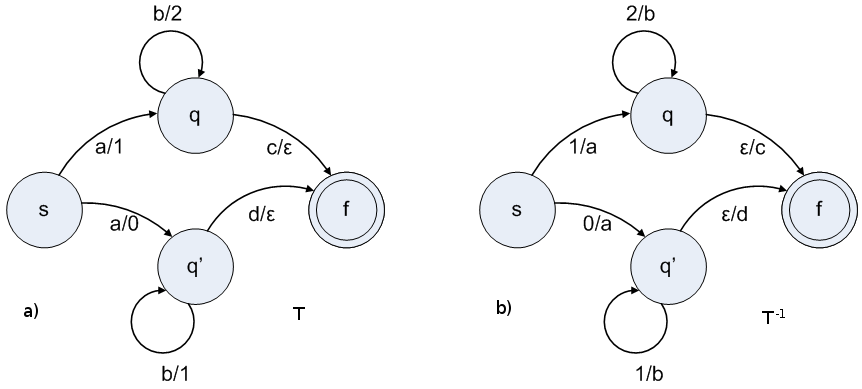


Figure 4.2: A finite transducer T (a) and its inverse transducer T^{-1} (b)

Now I will show how to test twins property in the composed transducer $T^{-1} \times T$. Let's go back to the Definition 4.2 of twins property in Section 4.1. Using a composed transducer, we can simplify it. We assume the same finite transducer T. T has the twins property if and only if the following condition holds for any state q, any path π from the initial state to q, and any cycle c at q in $T^{-1} \times T$:

$$i[\pi]^{-1} o[\pi] = i[\pi c]^{-1} o[\pi c]. \tag{4.2}$$

4.3 Residues

The test of condition 4.2 is based on the notion of "*residue*" of two paths. Let $\Sigma^{(*)}$ denote the *free group generated by* Σ. For two elements $x, y \in \Sigma^{(*)}$, the *residue of* x *by* y is defined as $y^{-1}x$. A residue is said to be *pure* if it is in $\Sigma^{(*)} \cup (\Sigma^{-1})^*$.

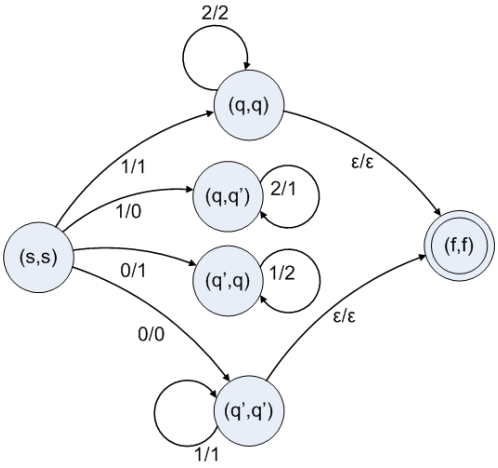


Figure 4.3: A composed transducer $T^{-1} \times T$

Commutativity

An element $x \in \Sigma^{(*)}$ is said to be *primitive* if it cannot be written as $x = y^n$, $n \in \mathbb{N}$ and $x \neq y$. The element y is called the *primitive root of* x.

Lemma 4.4. *Let x and y be two elements of $\Sigma^{(*)} - \{\varepsilon\}$, x and y commute iff x and y or x and y^{-1}, have the same primitive roots.*

When x and y commute, we write $x \equiv y$.

Example 4.2: Assume that $u = abab$. We can rewrite the string as $u = (ab)^2$, it means u is not primitive and ab is its primitive root. Let's take another string $v = ababab$. We can rewrite it as $v = (ab)^3$ which implies that strings u and v have the same primitive roots and therefore *commute*.

Properties of residues

The residue of path π is defined as the residue of its input and output labels: $\langle \pi \rangle = i[\pi]^{-1} o[\pi]$. A path π is *pure* if its residue $\langle \pi \rangle$ is pure. Simply we can say that a residue is pure only if all of its symbols are regular or all of them are inverted.

Example 4.3: In Figure 4.4 we can see a part of a finite transducer with a simple path π beginning in the initial state s and ending in the state q. The residue of the path is $\langle \pi \rangle = (12)^{-1} 1 \varepsilon = 2^{-1} 1^{-1} 1 = 2^{-1}$ and it is pure.

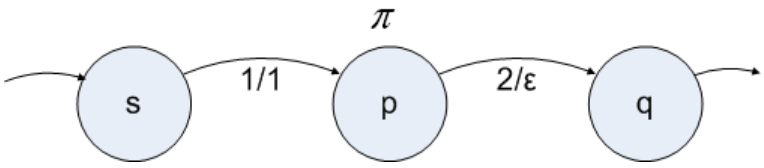


Figure 4.4: Computing the residue of the path π

The following properties of residues will be used in the algorithm for testing the twins property:

Lemma 4.5. *Let π_1, π_2, π_3 and π be four paths in a transducer T such that paths π_1, π_2, π_3 end in the same state where path π begins. Then,*

1. $\langle\pi_1\rangle = \langle\pi_2\rangle$ *iff* $\langle\pi_1\pi\rangle = \langle\pi_2\pi\rangle$;
2. $\langle\pi_1\rangle^{-1}\langle\pi_3\rangle \equiv \langle\pi_1\rangle^{-1}\langle\pi_2\rangle$ *iff* $\langle\pi_1\pi\rangle^{-1}\langle\pi_3\pi\rangle \equiv \langle\pi_1\pi\rangle^{-1}\langle\pi_2\pi\rangle$;
3. *if π_1 is not pure, then $\pi_1\pi$ is also not pure;*
4. *if π_1 is not pure and $\langle\pi_1\pi\rangle = \langle\pi_1\rangle$, then $i[\pi] = o[\pi] = \varepsilon$.*

Generally, $\equiv$ defines an equivalence relation on $\Sigma^{(*)} - \{\varepsilon\}$.

4.4 Test of functionality of finite transducers

It has been denoted that functionality is a necessary but insufficient condition for a transducer to be determinizable. Test of functionality must precede to the test of twins property. First of all, let's introduce some theory.

A *successful path* in a finite transducer is a path from an initial state s to a final state f. A state q is called *"accessible"* if that state can be reached from the initial state. That state is called *"terminating"* if one of the final states can be reached from the state q.

Example 4.4: Look at Figure 4.3. The states (q', q) and (q, q') are accessible but not terminating. By contrast, the states (s, s),(q', q') and (q, q) are accessible and terminating at the same time. The paths $(s, s)(q, q)(f, f)$ and $(s, s)(q', q')(f, f)$ are both successful.

Functionality of T can be determined by verifying that following condition holds for all successful paths in the composed transducer $T^{-1} \times T$ [3, 1]: the input string of the path π is equal to the output string of the path π, that is, $i[\pi] = o[\pi]$. This is further equivalent to that the residue for any successful path π is an empty string, $\langle\pi\rangle = \varepsilon$.

This also implies that for any two paths π and π' from one of the initial states to a particular terminating state q, the residue of π is equivalent to the residue of π': $\langle\pi\rangle = \langle\pi'\rangle$.

The composed transducer in Figure 4.3 is evidently functional.

In [1] has been shown that the complexity of the test is $O(|R|^2 + |\Omega||Q|^2)$.

4.5 Test of the twins property

Using that residue notion, we can rewrite Eq. 4.2 and the twins property condition:

Proposition 4.6. *A transducer T has the twins property iff the following condition holds for the composed transducer $T^{-1} \times T$: for any path π from an initial state to a cycle c,*

$$\langle\pi\rangle = \langle\pi c\rangle. \tag{4.3}$$

In particular, the condition outlined above implies that $|i[c]| = |o[c]|$ – that is, the length of the input and output string of the cycle c is equal. It further implies that $i[c] = \varepsilon$ iff $o[c] = \varepsilon$. Thus, when $i[c] = o[c] = \varepsilon$, the condition $\langle\pi\rangle = \langle\pi c\rangle$ holds for any path π. We say

that a state q is *"cycle-accessible"* if there exists a path from that state q to a non-ε cycle c ($i[c] \neq \varepsilon$ or $o[c] \neq \varepsilon$). In Figure 4.3 there are cycle-accessible states (q, q), (q, q'), (q', q) and (q', q').

Lemma 4.7. *If T has the twins property, then the residue of any path π_1 in $T^{-1} \times T$ from the initial state to a cycle-accessible state q is pure.*

Lemma 4.7 is directly derived from the statement 3 and 4 in Lemma 4.5.

By Proposition 4.6, testing the twins property is equivalent to testing that the condition holds for any path π from an initial state to a cycle c in the transducer $T^{-1} \times T$. We can limit ourselves to the cycle-accessible part of the composed transducer since the condition is otherwise always satisfied.

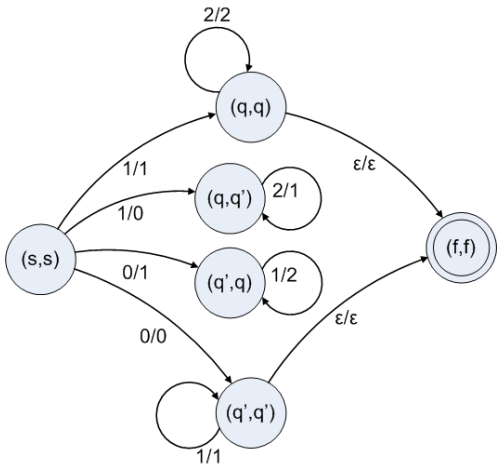


Figure 4.5: A composed transducer $T^{-1} \times T$

Example 4.5: To determine whether the transducer in Figure 4.2(a) is determinizable, we have to compute the residues for all paths leading to cycle-accessible states in the composed transducer $T^{-1} \times T$. The composed transducer is displayed again in Figure 4.5.

We will begin with the path π_1 $((s, s)(q, q))$ and cycle c_1 at (q, q):

$$\left.\begin{array}{l} \langle \pi_1 \rangle = (1)^{-1}1 = \varepsilon \\ \langle \pi_1 c_1 \rangle = (12)^{-1}12 = \varepsilon \end{array}\right\} \checkmark$$

And other paths in the same manner:

$$\left.\begin{array}{l} \langle \pi_2 \rangle = (1)^{-1}0 = 1^{-1}0 \\ \langle \pi_2 c_2 \rangle = (12)^{-1}01 = 2^{-1}1^{-1}01 \end{array}\right\} \neq$$

$$\left.\begin{array}{l} \langle \pi_3 \rangle = (0)^{-1}1 = 0^{-1}1 \\ \langle \pi_3 c_3 \rangle = (01)^{-1}12 = 1^{-1}0^{-1}12 \end{array}\right\} \neq$$

$$\left.\begin{array}{l} \langle \pi_4 \rangle = (0)^{-1}0 = \varepsilon \\ \langle \pi_4 c_4 \rangle = (01)^{-1}01 = \varepsilon \end{array}\right\} \checkmark$$

Only two conditions out of four have been satisfied which means that the original transducer is not determinizable (more exactly, the algorithm of determinization would never halt if we tried to run it).

One more important observation could have been made. Those two conditions were satisfied because the cycle-accessible states (q, q) and (q', q') are also terminating. It has been stated above that if a transducer is functional, the residue for any successful path or its part is an empty string.

Lemma 4.8. *Let π_1 and π_2 be two paths leading from the initial state to the same cycle-accessible state c. Assume that $\langle \pi_1 \rangle = \langle \pi_1 c \rangle$, then $\langle \pi_2 \rangle = \langle \pi_2 c \rangle$ iff $\langle \pi_1 \rangle^{-1} \langle \pi_2 \rangle \equiv o[c]$.*

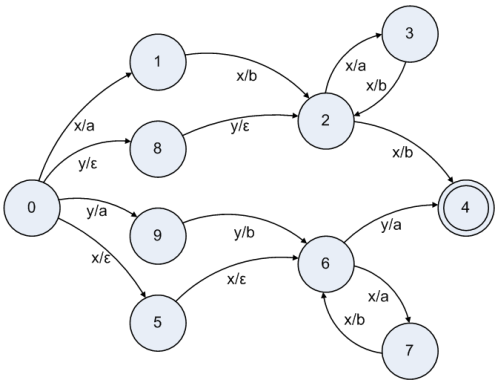


Figure 4.6: A finite transducer T

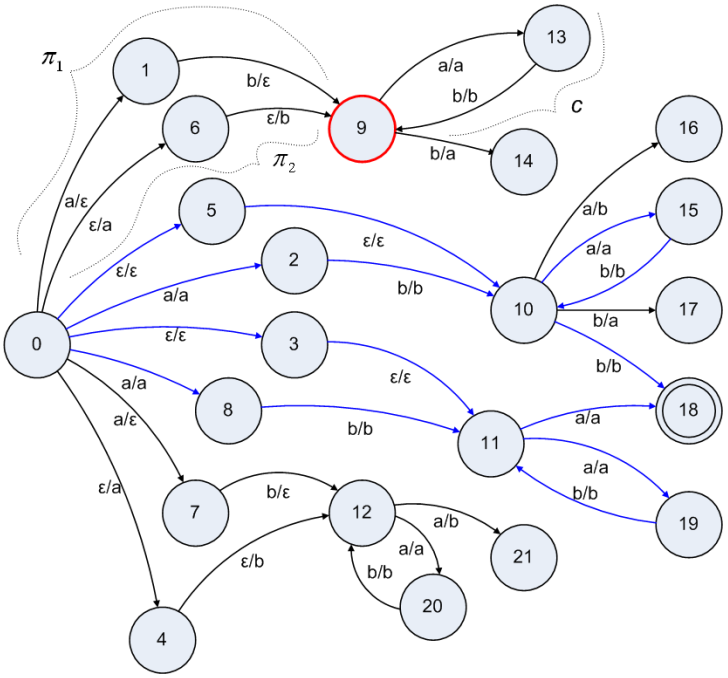


Figure 4.7: A composed transducer $T^{-1} \times T$

Example 4.6: Figures 4.6 and 4.7 display a finite transducer T and a composed transducer $T^{-1} \times T$. Let's explain how the condition brought in by Lemma 4.8 is used in testing the twins property when there are two distinct path to the same cycle-accessible state:

We will focus on the state 9 in Figure 4.7. It is cycle-accessible (cycle c) and there are two distinct paths π_1 and π_2 leading to that state. First, we have to calculate the residue of path π_1 and compare it with the residue $\pi_1 c$ according to the base condition. Then it is necessary to find out the residue of the path π_2:

$$\begin{aligned}\langle \pi_1 \rangle &= (ab)^{-1}\varepsilon = (ab)^{-1} \\ \langle \pi_1 c \rangle &= (abab)^{-1}ab = (ab)^{-1} \\ \langle \pi_2 \rangle &= \varepsilon(ab) = ab,\end{aligned}$$

$\langle \pi_1 \rangle = \langle \pi_1 c \rangle$. Now instead of testing the condition $\langle \pi_2 \rangle = \langle \pi_2 c \rangle$ we will test $\langle \pi_1 \rangle^{-1}\langle \pi_2 \rangle \equiv o[c]$ by Lemma 4.8:

$$\begin{aligned}\langle \pi_1 \rangle^{-1}\langle \pi_2 \rangle &= ((ab)^{-1})^{-1}ab = abab \\ o[c] &= ab\end{aligned}$$

The strings $abab$ and ab commute (Lemma 4.4) and therefore the condition is satisfied.

Corollary 4.9. *Let π_1, π_2 and π be three paths leading from the initial state to the same cycle-accessible state such that $\langle \pi_1 \rangle \neq \langle \pi_2 \rangle$. Assume that $\langle \pi_1 \rangle = \langle \pi_1 c \rangle$ and $\langle \pi_2 \rangle = \langle \pi_2 c \rangle$, then: $\langle \pi \rangle = \langle \pi c \rangle$ iff $\langle \pi_1 \rangle^{-1}\langle \pi_2 \rangle \equiv \langle \pi_1 \rangle^{-1}\langle \pi \rangle$.*

In particular, the condition $\langle \pi \rangle = \langle \pi c \rangle$ from Proposition 4.6 is used when there is only one path through the composed transducer to a cycle-accessible state q and for the first path when there are more distinct paths. In contrast, the condition of Lemma 4.8 is used for the second path and the condition brought in the Corollary 4.9 for the third and subsequent paths if there are more distinct path to the same cycle-accessible state q. Such testing is more efficient.

To sum it up, once the finite transducer is found to be functional, the algorithm is based on the computation of the residues of paths in $T^{-1} \times T$ as suggested by the proposition. However, we use some properties of residues to avoid redundant computations:

1. the property shown in Lemma 4.5(1) which applies to paths sharing the same suffix,
2. we can simplify the computation of the residue of the second path leading to the same state q by Lemma 4.8,
3. we only need to compute at most two distinct path residues for any state q, as shown in Corollary 4.9.

To come through the graph, it is reasonable to use the method of depth first search (DFS). The complexity of the algorithm testing the twins property is $O(|Q^2|(|Q|^2 + |E|^2))$. The most time-consuming operation is comparing the primitive roots of strings. As outlined above, the input transducer has to be functional and therefore the test of functionality must precede. Thereby the total time complexity increases by $O(|R|^2 + |\Omega||Q|^2)$.

Chapter 5

New approaches to determination of the determinizability

In this chapter I will present some new ideas about determination of the determinizability of finite transducers.

If you had read the previous chapter, you could have notice that the problem of deciding determinizability is not simple and the achieved results are very accurate but the method is too time-consuming and unnecessarily difficult for us. As far as translation is concerned, speed is the most important aspect. Our determinization algorithm is very quick and we want to keep its time complexity so we cannot afford to compute a complex predeterminization algorithm. Actually, there are two different approaches to determination of the determinizability:

1. *We can implement some pre-determinization computation and thus analyze the input transducer before we run the own determinization process.*

   There are pros and cons of such a method. It quickly detects an undeterminizable transducer but on the other hand the time of the whole process could distinctly lengthen as the computation would be done before each determinization.

2. *The second approach assumes checking the determinizability* "on the fly" *which means while the determinization process is running.*

   If we used this method, we would not have to go through the graph more then once, but, of course, there would be more calculations in each step. An undeterminizable transducer would probably be revealed later then by using the first method.

To sum up the requirements on our parsing method – we demand a quick method and suppose the most of the input transducers to be determinizable as there is only a few undeterminizable cases. Therefore the process of determination the determinizability should not slow down the whole method. Moreover, we do not mind so much if the computation is a little longer in the case that the input transducer is undeterminizable, but it must be finite.

I tried to work out the problem by several methods, especially by counting the states and edges in the input and output transducer, but this way I have not found any reasonable solution although there might exist. In the following sections I am going to bring up a new method of determining the determinizability by *detection of cycles* in the input transducer, but first of all I will give some information on the power of our determinization algorithm.

5.1 On the power of the determinization algorithm

There is one more reason to invent a new detection method – our determinization algorithm does not take into account the output strings by building new states whereas the method presented in Chapter 4 assumes we do it. For example, the determinization algorithm described in [13] interprets the following transducer as ambiguous and thus not determinizable. It is not a wrong behaviour as we assume there should not occur such case in our situation – recall that the outputs of the transducer are semantic actions that take control of another finite automaton.

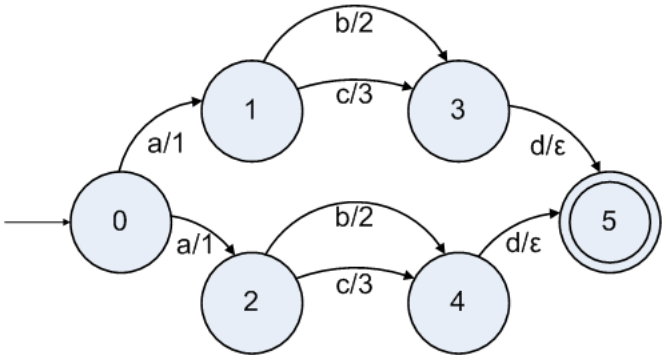


Figure 5.1: An ambiguous transducer

In the same manner Mohri's algorithm would mark the following transducer (Figure 5.2) determinizable, but we would not be able to determinize it as we do not check the output strings at all, so an attempt to determinize such a transducer would end up in cycle.

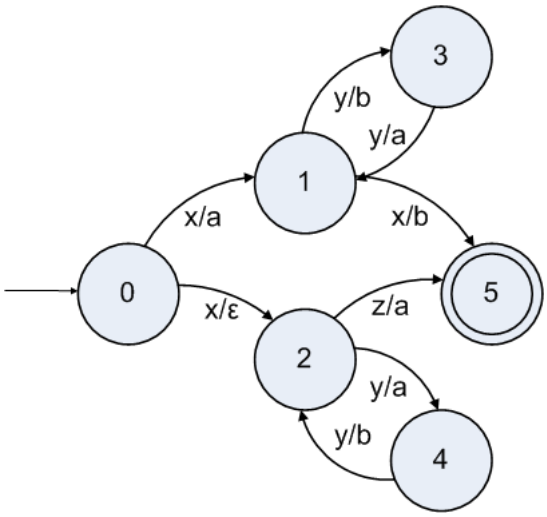


Figure 5.2: The transducer is determinizable in theory but actually we do not do it.

It implies that although the method of determining the determinizability by detection of cycles in input transducer, which will be presented immediately, is weaker than Mohri's method, it *does not reduce the power of the determinization algorithm we use.*

5.2 Determining the determinizability by detection of cycles

As we do not want to go through the graph more than once, we will use the second approach and check the determinizability while the determinization process is running. The additional calculations will not be very time-consuming, as you will see.

It is easy to imagine that only transducers that translate an infinite language can cause cycles in the determinization algorithm. The infinitness of both input and output languages is achieved by presence of loops in the input and output transducer. We can say a transducer is determinizable provided it is unambiguous and cycle-free. Obviously, not all loops can cause undeterminizability. They must have a special property and the property must be detected.

The algorithm of determining the determinizability by detection of cycles follows on testing the twins property but in a completely new way. We will not have to build a composed transducer or compare residues. The basic idea is in the *storage of the history* of states and searching for loops.

In the following picture I will show a typical transducer that cannot be determinized.

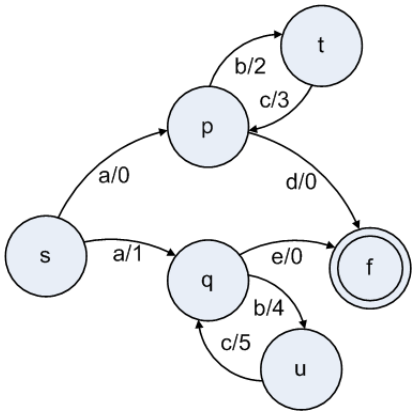


Figure 5.3: An undeterminizable transducer

It is obvious that the undeterminizability is caused by those two loops as there are two different paths with the same label (a) leading to the cycles marked with the same symbol (b). We could help determine undeterminizability by adding a special set to each composed state in the new transducer. In the set we would keep the history of all the previous composed states without its saved output strings. Actually, it is a set of the sets of substates labels. In the following example I will describe several phases of the determinization process of the transducer from Figure 5.3.

Example 5.1: Description of the Figure 5.4: In phase 1 we put the states p, q together and make one composed state. We have to remember the strings 0 and 1 in the state. Besides we put the set of labels of the substates into its own *history set*. In the next phase we do the same with states t, u – by our determinization algorithm – but as far as the history set is concerned, first we have to *copy* the contents of the previous state's history set and then *check* if it contains the actual state's substates labels' set. In phase 2 the test succeeded and we can continue to the phase 3. Here we make a new state and copy the contents of the previous state's history set $(\{p, q\}, \{t, u\})$ to its own history set. Now we finally get to the problem. Phase 3 ends up with failure when we try to put the set of actual substates $\{p, q\}$ to the own history set *for the second time*.

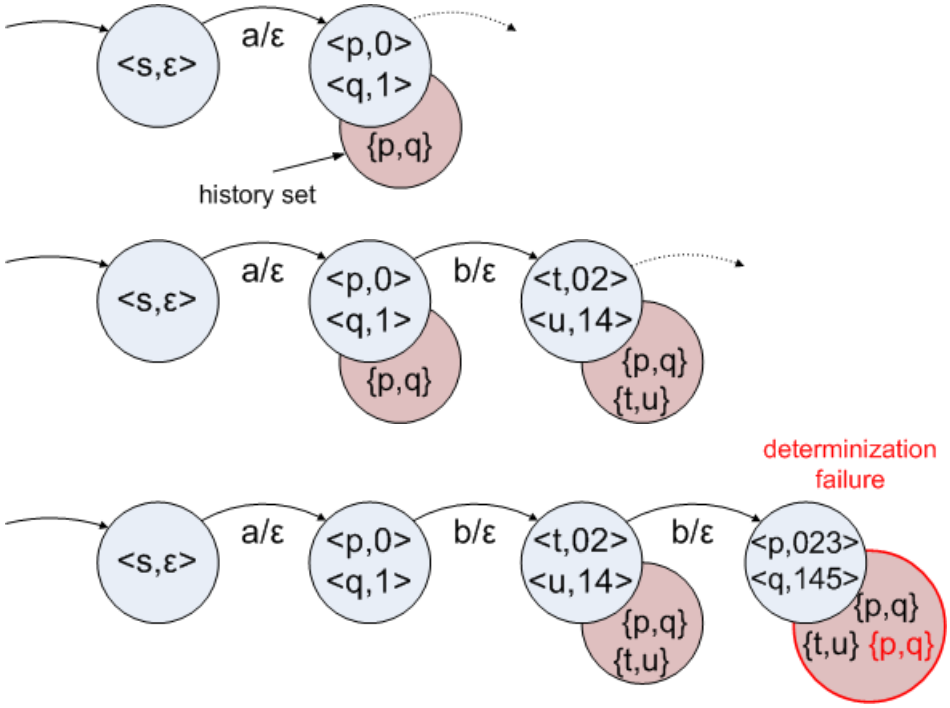


Figure 5.4: Three phases of the detection of undeterminizability.

Properties

Now the basic idea of the method of detection of cycles has been outlined and I will continue with more detailed description. The formal algorithm will be included in the next section (5.4).

To go through the graph we use the method of *depth first search* which means we put the new states onto the stack.

It is important that we only keep the history set by the states *we have not expanded yet* and by the actual state. Then the sets are immediately *deleted*. This ensures that they do not take up space any more.

Again we can distinguish three different situations by the determinization:

- The simpliest one is that there is only one edge coming out of a state with a particular label. Then we do not care for the history sets at all (or they are empty).
- Second situation comes up if there are two or more edges with the same label coming out from one state. By determinization we create a new composed state and put its own substates labels' set into the history set of the new composed state.
- The third situation we have to deal if there are more states reachable from two or more substates of a composed state by reading the same symbol. In such case we *copy* the history of the previous composed state into the history of the new state and *try* to add the set with its own substates labels. Before the union we have to *check* if we do not add it for the second time. If the history set already contains the set of the actual substates labels, it leads to the determinization failure and we can declare the transducer *undeterminizable*.

To sum it up, there are several new operations in the extended determinization algorithm:

insertion into a set,

duplication of a set,

searching in a set.

The cardinality of the set will be at most as great as the length of the longest *ambiguous path* in the transducer which is expected to be much less then the number of all states in the determinized transducer.

5.3 Proof

Proof of the algorithm can be based on the theory of testing the twins property. Let's recall some basic principles.

- We know that the twins property is a necessary and sufficient condition for functional finite-state transducers to be determinizable [4].
- The definition of the twins property 4.2 has been stated above.
- In Section 5.1 we explained why we do not have to check any outputs by determinization.

By definition of the twins property we know that two sibling states are said to be twins iff for any paths $\pi_1 \in P(s, y, q)$, $\pi_2 \in P(q, x, q)$, $\pi'_1 \in P(s, y, q')$, $\pi'_2 \in P(q', x, q')$:

$$o[\pi_1]^{-1} o[\pi'_1] = (o[\pi_1]o[\pi_2])^{-1} o[\pi'_1]o[\pi'_2].$$

As we do not want to check the particular output strings, we have to assume the equation has only one solution – both sides are equal to the empty string if we put $\pi_1 = \pi'_1$ and $\pi_2 = \pi'_2$. It implies that if for any paths $\pi_1 \in P(s, y, q)$, $\pi_2 \in P(q, x, q)$, $\pi'_1 \in P(s, y, q')$, $\pi'_2 \in P(q', x, q')$ holds that $\pi_1 = \pi'_1$ and $\pi_2 = \pi'_2$, the finite-transducer has the twins property and therefore is determinizable.

5.4 Formal algorithm

- Input: ε-free finite transducer $M_1 = (Q_1, \Sigma, \Omega, R_1, s_1, \{f_1\})$
- Output: Deterministic finite transducer $M_2 = (Q_2, \Sigma, \Omega, R_2, s_2, \{f_2\})$ such that $T(M_1) = T(M_2)$, or error if M_1 is ambiguous or undeterminizable
- Method:

$s_2 := \{\langle s_1, \varepsilon\rangle\}$;
$f_2 := \{\langle f_1, \varepsilon\rangle\}$;
$S_{undone} := \{(s_2, \{\emptyset\})\}$;
$S_{done} := \emptyset$;
while $S_{undone} \neq \emptyset$ **do begin**
  let $(X, S_{cycle}) \in S_{undone}$:
  $S_{undone} := S_{undone} - \{X\}$;
  $S_{done} := S_{done} \cup \{X\}$;
  for each $a \in \Sigma \cup \{\varepsilon\}$ **do begin**
    $Q_{in} := \{\langle p, z\rangle : \langle p, z\rangle \in X, pa \to yq \in R_1, y, z \in \Omega^*, q \in Q_1\}$;
    if $|Q_{in}| = 1$ **then**
      let $\langle p, z\rangle \in Q_{in}$:
      $Q_{out} := \{\langle q, y\rangle : q \in Q_1, y \in \Omega^*, pa \to yq \in R_1\}$;
      if $|Q_{out}| = 1$ **then**
        let $\langle q, y\rangle \in Q_{out}$:
        $Q_{out} := \{\langle q, \varepsilon\rangle\}$;
        add $Xa \to zyQ_{out}$ to R_2;
        $Q_{cycle} := \emptyset$;
      if $|Q_{out}| > 1$ **then**
        add $Xa \to zQ_{out}$ to R_2;
        if exists $q \in Q_1$ **such that** $|\{y : \langle q, y\rangle \in Q_{out}\}| > 1$ **then**
          error(ambiguity)
        $Q_{cycle} := \{q\colon \langle q, y\rangle \in Q_{out}\}$;
      if $Q_{out} \notin S_{done}$ **then**
        $S_{undone} = S_{undone} \cup \{(Q_{out}, \{Q_{cycle}\})\}$;
    if $|Q_{in}| > 1$ **then**
      $Q_{out} := \{\langle q, zy\rangle : \langle p, z\rangle \in X, pa \to yq \in R_1, y, z \in \Omega^*, q \in Q_1\}$;
      add $Xa \to Q_{out}$ to R_2;
      if exists $q \in Q_1$ **such that** $|\{y : \langle q, y\rangle \in Q_{out}\}| > 1$ **then**
        error(ambiguity)
      $Q_{cycle} := \{q\colon \langle q, y\rangle \in Q_{out}\}$;
      if $Q_{cycle} \in S_{cycle}$ **then**
        error(cycle)
      if $Q_{out} \notin S_{done}$ **then**
        $S_{undone} = S_{undone} \cup \{(Q_{out}, S_{cycle} \cup \{Q_{cycle}\})\}$;
  end
end
$Q_2 := S_{done}$;

Actually, S_{undone} is implemented as a stack, but generally it can be a set.

5.5 Space for further extensions

The presented method guarantees finitness of the process of determinization nevertheless it does not solve the fact that there are some transducers we cannot convert to deterministic. It is only a small subset of general transducers, and mostly we do not need to work with them. Therefore it would be worthless for us to invent a completely new determinization algorithm which would be definitely more complex and demanding but able to determinize all (or a greater subset of) general transducers.

In spite of that, there is a suitable solution. The algorithm of determining the determinizability by detection of cycles was designed with reference to further extensions. As it checks determinizability "on the fly", it is able to exactly detect the cause of undeterminizability (the actual loop). So far we have put up with simple detection and notification of the problem, but in future there could be some special component added – such as a queue or a stack – and in that manner the power of the determinization algorithm would be *increased*. It would be an ideal solving if we did not have to use the extra component during the whole determinization process (as it would unnecessarily slow down the computations) but just in case it was required.

Chapter 6

Practical results

The algorithm of determining the determinizability by detection of cycles has been implemented as an extension to the program BISON-- made by Dr. Roman Lukáš and Eva Zámečníková within the scope of her bachelor's thesis [13].

6.1 Principles of the compiler

The compiler accepts just right linear grammar on input and thus is weaker than *Yacc* or *Bison* which are able to make a syntax analyzer based on LR-grammars, but on the other hand, LR-syntax analysis uses difficult LR-tables and stack and therefore is much slower. BISON-- was designed for a quick and efficient translation from assembly language to binary code and reversely.

The compiler is based on the model of two-way coupled finite automaton $\Gamma = (M_1, M_2, h)$, where M_1 is a lazy finite automaton for assembly format's description of instructions, M_2 is a lazy finite automaton for binary format's description of instructions, and h describes translating relations between rules of M_1 and M_2. Lazy finite automaton M_1 is converted to lazy finite transducer $\overline{M_1}$ by adding the label of each rule from M_1 to the same rule as a generated output. Lazy finite transducer $\overline{M_1}$ is converted to the equivalent strictly deterministic finite transducer $\overline{M_d}$ by using the determinization algorithm presented in Chapter 3.

By simulation of the strictly deterministic finite transducer (as an assembler-translator reads the assembly code of instruction), the output sequence of symbols is not generated to the output, but this sequence activates corresponding edges in M_2. Then M_2 generates the corresponding output by using the activated edges. Analogically, deterministic translation from binary code to assembly language and automatic construction of this translator is realized.

6.2 Implementation

As in this work I deal with detection of the determinizability of an input finite transducer, the second part of translation (by the coupled finite-transducer) is not interesting for us. Therefore the program attached to this thesis is reduced to simulation of an input lazy finite transducer. It is written in C++ language and it expects a transducer described in the form of *right linear grammar* on input. We will be further concerned with the syntax of the input.

For transparency, the output of the simulation is a description of the determinized finite transducer which can be visualized by the open source program Graphviz [5]. If we assign some semantic actions to the output – instead of just labels – we can generate a program in language C simulating the activity of the transducer. This function has also been implemented.

Specification of the input

Here I will give an example of the input of the program. Assume a transducers with the following rules:

$sa \rightarrow 0p$
$pb \rightarrow 1p$
$pc \rightarrow f$

The input of the program:

```
s0 -> "a" s1 {0}
    ;
s1 -> "b" s1 {1}
    | "c" s2 {}
    ;
s2 -> "" {}
    ;
```

Note the parallelism with right linear grammars in the notation. The states can be viewed as nonterminal symbols. The nonterminal symbol on the left side of the first rule is the starting symbol. Nonterminal symbols on the right side and semantic actions are not compulsory.

Specification of the output

As I have implemented an extension guaranteeing that the simulation of all transducers will be finite, the output will interest us. If program ends with success, a structure with the determinized transducer is written on output.

Here is an example:

```
digraph G {
0 -> 1[label = "a/0"];
1 -> 1[label = "b/1"];
1 -> 2[label = "c/"];
2 -> 3[label = "$/"];
}
```

On the other hand, if the program detects *ambiguity* or *undeterminizability*, it writes a note on `stderr` and just put the input transducer into output without any changes.

The method of determining the determinizability by detection of cycles is implemented right according to the formal algorithm introduced above.

Chapter 7

Conclusion

As it is obvious that the most efficient is simulation of *deterministic finite transducers*, I focused on the own determinization process in this thesis.

There has been given much effort in the field of the determinization of finite automata, unlike finite transducers. The process of the determinization of finite transducers is more complex as we have to take care to preserve the sequence of applied rules. A method of determinization has been shown in [7] and also dealt by Eva Zámečníková in her bachelor's thesis [13]. Nevertheless, the algorithms were inapplicable as they had a serious deficiency – they were not able to determinize a general unambiguous finite transducer on input. From the theory of formal languages we know that some transducers are undeterminizable. In order to apply the determinization algorithm we have to know if the input transducer is determinizable, otherwise the method fails.

Theoretical benefit

In this thesis I investigated several works concerning finite transducers and their determinization, and also outlined the only one published method of determining the determinizability of an input transducer [1]. Afterwards I showed why this method is not reasonable with reference to translation by two-way coupled finite automata and consequently offered a new method of determining the determinizability by detection of cycles.

Actually I extended the algorithm presented by [13] with the test of determinizability of input transducers and therefore guaranteed the finitness of the algorithm even for translation of infinite languages. It was designed for the simulation of finite transducers as a part of translation by two-way coupled finite automata but the method can be generally used to check determinizability of finite transducers.

Practical benefit

Within the scope of this thesis, a program simulating the function of a finite transducer has been implemented. Deterministic transducers are the basis of two-way coupled finite automata which can be used for more general deterministic parsing. More exactly, they were invented as a model for assembler and disassembler with the idea to simulate their behaviour and thus contribute to concurrent development of hardware and software.

The method of detection of the determinizability was designed with reference to further extension if needed. One of the potential extensions could be an attempt to increase the power of the determinization algorithm by adding an extra component, for example a queue.

Bibliography

[1] Allauzen, C., and Mohri, M. Efficient algorithms for testing the twins property. *Journal of Automata, Languages and Combinatorics 8*, 2 (2003), 117–144.

[2] Allauzen, C., and Mohri, M. An optimal pre-determinization algorithm for weighted transducers. *Theoretical Computer Science 328*, 1-2 (2004), 3–18.

[3] Béal, M.-P., Carton, O., Prieur, C., and Sakarovitch, J. Squaring transducers: an efficient procedure for deciding functionality and sequentiality. *Theoretical Computer Science 292*, 1 (2003), 45–63.

[4] Choffrut, C. Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationelles. *Theoretical Computer Science 5* (1977), 325–338.

[5] Ellson, J., and Gansner, E. Graphviz: Graph visualization software [online]. `http://www.graphviz.org`, 2004. [cit. 2008-05-07].

[6] Meduna, A. *Automata and Languages: Theory and Applications.* Springer Verlag, London, GB, 2000.

[7] Mohri, M. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering 2*, 1 (1996), 61–80.

[8] Mohri, M. Finite-state transducers in language and speech processing. *Computational Linguistics 23*, 2 (June 1997), 269–311.

[9] Mohri, M. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics 7*, 3 (2002), 321–350.

[10] Parkes, A. P. *Introduction to languages, machines and logic.* Springer Verlag, London, GB, 2002.

[11] Roche, E., and Schabes, Y., Eds. *Finite-State Language Processing.* MIT Press, Cambridge, MA, USA, 1997.

[12] Rozenberg, G., and Salomaa, A., Eds. *Handbook of formal languages, vol. 1: word, language, grammar.* Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[13] Zámečníková, E. Syntaktická analýza založená na speciálních modelech. Master's thesis, FIT VUT v Brně, Brno, 2007.

Appendix A

Contents of the attached CD

On the attached compact disk you can find an electronic version of this document and the complete program part. To find out how to run it, begin with file `/program/Readme`. There are also testing data including visualizations enclosed.