Constructing
Minimal Acyclic Deterministic
Finite Automata

Bruce William Watson
bruce@bruce-watson.com
FASTAR Research Group
www.fastar.org

Professor Extraordinary
Computer Science
University of Pretoria

Full Professor
Information Systems
Stellenbosch University

Submitted in partial fulfillment of the requirements for the degree
Philosophiae Doctor (Computer Science) in the
Faculty of Engineering, Built Environment and Information Technology
University of Pretoria, Pretoria, South Africa

November 2010

© University of Pretoria
Constructing
Minimal Acyclic Deterministic
Finite Automata

Author: Prof. Dr. Bruce W. Watson
Supervisor: Prof. Dr. Derrick G. Kourie

University of Pretoria, Pretoria, South Africa
To my family, especially Liam and Keira (I almost finished this thesis as you arrived)
Abstract

This thesis is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Ph.D) in the FASTAR group of the Department of Computer Science, University of Pretoria, South Africa. I present a number of algorithms for constructing minimal acyclic deterministic finite automata (MADFs), most of which I originally derived/designe‌d or co-discovered. Being acyclic, such automata represent finite languages and have proven useful in applications such as spell-checking, virus-searching and text indexing. In many of those applications, the automata grow to billions of states, making them difficult to store without using various compression techniques — the most important of which is minimization. Results from the late 1950’s show that minimization yields a unique automaton (for a given language), and later results show that minimization of acyclic automata is possible in time linear in the number of states. These two results make for a rich area of algorithmics research; automata and algorithmics research are relatively old fields of computing science and the discovery/invention of new algorithms in the field is an exciting result.

I present both incremental and nonincremental algorithms. With nonincremental techniques, the unminimized acyclic deterministic finite automaton (ADFA) is first constructed and then minimized. As mentioned above, the unminimized ADFA can be very large indeed — often even too large to fit within the virtual memory space of the computer. As a result, incremental techniques for minimization (i.e. the ADFA is minimized during its construction) become interesting. Incremental algorithms frequently have some overhead: if the unminimized ADFA fits easily within physical memory, it may still be faster to use nonincremental techniques.

The presentation used in this thesis has a few unusual characteristics:

• Few other presentations follow a correctness-by-construction style for presenting and deriving algorithms. The presentations given here include correctness arguments or sketches thereof.

• The presentation is taxonomic — emphasizing the similarities and differences between the algorithms at a fundamental level.

• While it is possible to present these algorithms in a formal-language-theoretic setting, this thesis remains somewhat closer to the actual implementation issues.

• In several chapters, new algorithms and interesting new variants of existing algorithms are presented.

• It gives new presentations of many existing algorithms — all in a common format with common examples.

• There are extensive links to the existing literature.
# Contents

Abstract \hspace{2cm} i

Preface \hspace{2cm} vi

1 Introduction 1
   1.1 Problem statement \hspace{2cm} 1
   1.2 To the reader \hspace{2cm} 2
   1.3 Related work and a short history \hspace{2cm} 2
   1.4 Links to the literature \hspace{2cm} 4
   1.5 Future work \hspace{2cm} 4

2 Preliminaries 7
   2.1 General definitions \hspace{2cm} 7
   2.2 Algorithm presentation \hspace{2cm} 8
   2.3 Strings and languages \hspace{2cm} 8
   2.4 Automata \hspace{2cm} 10
   2.5 Minimality of automata \hspace{2cm} 17
      \hspace{2cm} 2.5.1 Computing $E$ \hspace{2cm} 21
      \hspace{2cm} 2.5.2 Computing $\neg E$ in ADFAs \hspace{2cm} 21
      \hspace{2cm} 2.5.3 Computing $E(p, q)$ pointwise \hspace{2cm} 21
      \hspace{2cm} 2.5.4 A more efficient computation of $E$ \hspace{2cm} 21

3 A MADFA-construction skeleton 23
   3.1 Specific instantiations \hspace{2cm} 24
      \hspace{2cm} 3.1.1 Choosing a structural invariant \hspace{2cm} 24
      \hspace{2cm} 3.1.2 Function $\text{add} \_ \text{word}$ \hspace{2cm} 25
      \hspace{2cm} 3.1.3 Function $\text{cleanup}$ \hspace{2cm} 25
   3.2 Commentary \hspace{2cm} 25

4 Trie intermediate ADFA 27
   4.1 Procedure $\text{add} \_ \text{word}_T$ \hspace{2cm} 27
      \hspace{2cm} 4.1.1 Adding only prefix words \hspace{2cm} 28
      \hspace{2cm} 4.1.2 Adding a nonprefix word in a trie \hspace{2cm} 29
   4.2 Procedure $\text{cleanup}_T$ \hspace{2cm} 30
      \hspace{2cm} 4.2.1 Selecting $N : N \subseteq T \land N \neq \emptyset \land \text{Inequiv}(N)$ \hspace{2cm} 31
         \hspace{2cm} 4.2.1.1 Selecting a single state \hspace{2cm} 32
         \hspace{2cm} 4.2.1.2 Selecting a path of states \hspace{2cm} 34
      \hspace{2cm} 4.2.2 Selecting $N : N \subseteq T \land N \neq \emptyset \land \text{Pairwise}_\text{inequiv}(D, N)$ \hspace{2cm} 35
   4.3 An example \hspace{2cm} 38
4.4 Time and space performance ........................................ 41
  4.4.1 Improvements .................................................... 41
4.5 Commentary ........................................................... 41

5 Arbitrary intermediate ADFA ........................................... 43
  5.1 Procedure add_word_N ............................................. 43
  5.2 Procedure cleanup_N ............................................. 46
  5.3 Time and space performance ...................................... 46
  5.4 Commentary ........................................................ 46

6 Minimal intermediate ADFA ........................................... 49
  6.1 Procedure add_word_I ............................................. 49
    6.1.1 Recursive helper procedure visit_min ..................... 51
  6.2 Procedure cleanup_I ............................................. 54
  6.3 An example ........................................................ 54
  6.4 Time and space performance ...................................... 59
  6.5 Commentary ........................................................ 59

7 Reversed trie intermediate ADFA ...................................... 61
  7.1 Procedure add_word_R ............................................. 61
  7.2 Procedure cleanup_R ............................................. 61
  7.3 An example ........................................................ 62
  7.4 Time and space performance ...................................... 63
  7.5 Commentary ........................................................ 63

8 Avoiding cloning while adding words ................................ 65

9 Words in lexicographic order ......................................... 67
  9.1 Procedure add_word_S ............................................. 67
    9.1.1 A minor problem in using visit_min ..................... 70
  9.2 Procedure cleanup_S ............................................. 71
  9.3 An example ........................................................ 71
  9.4 Time and space performance ...................................... 73
    9.4.1 Improvements .................................................. 73
  9.5 Commentary ........................................................ 74

10 Minimizing depth layers ............................................. 75
  10.1 Procedure add_word_D ........................................... 75
  10.2 Procedure cleanup_D ........................................... 77
  10.3 An example ........................................................ 77
  10.4 Time and space performance ...................................... 81
  10.5 Commentary ........................................................ 81

11 Minimizing semi-incrementally ..................................... 83
  11.1 Procedure add_word_W ........................................... 83
    11.1.1 Procedure semi_min ......................................... 85
      11.1.1.1 Refining S'_{11.1.1} ..................................... 86
      11.1.1.2 Refining S''_{11.1.1} .................................... 86
      11.1.1.3 A final version of semi_min ......................... 87
CONTENTS

11.2 Procedure \( \text{cleanup}_W \) .................................................. 88
11.3 An example ................................................................. 88
11.4 Time and space performance ....................................... 92
11.5 Commentary ............................................................... 92

Bibliography ........................................................................ 93

Index ................................................................................... 99

Colophon ............................................................................. 101
Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Ph.D) in the FASTAR group of the Department of Computer Science, University of Pretoria, South Africa. I present a number of algorithms for constructing minimal acyclic deterministic finite automata, most of which I personally originally derived/designer or discovered. In certain applications, these automata are a compact representation of a large finite number of words — for example in a spelling checker or a computer virus detector. As such, efficient new algorithms also have commercial value in addition to their intrinsic scientific value.

In the time leading up to the presentation of my first Ph.D [Wat95] in Eindhoven, I was actively implementing and using finite automata and transducers in industry (primarily in the fields of compilation and text indexing). During and subsequent to my Eindhoven Ph.D research, I have remained very active in the field in several ways:

- I have served as a consultant on automata construction, minimization, and implementation, particularly for applications in computational linguistics, network security and text indexing.

- From 1996, I served on the program committee of the International Workshop on Implementing Automata. In 2000, the workshop was upgraded to the international Conference on Implementations and Applications of Automata. During 23–25 July 2001, the sixth such event was chaired by me and hosted at the University of Pretoria, in South Africa [WW01a, WW01b]. As a side-effect of that conference, I co-edited a special issue of the journal Theoretical Computer Science [WW04]. In 2007, I presented one of the keynote talks at the conference, in Prague.

- From its origins in 1996 onwards, I have been involved (served in the program committee) of the annual Prague Stringology Conference (formerly a workshop).

- Also since 1996, I have actively been involved in Finite State Methods in Natural Language Processing workshop (not held annually). It serves as a venue for work at the intersection of computing science and linguistics — including the work presented in this thesis.

- The first annual FASTAR symposium (on algorithmics, data-structures and applications of finite state techniques) was held in Eindhoven in 2004. Having an algorithmics and data-structures slant, the FASTAR conference complemented the Conference on Implementations and Applications of Automata.

- I have actively published work in this field. For an overview, see the reference list at the end of this thesis.

Although the common thread in my research relates to automata, a more recent specific thread is the work on constructing acyclic deterministic finite automata. It is this work that is reported in this thesis.
Acknowledgements  Particular thanks go to Derrick Kourie for being an outstanding promotor, supervisor and motivator, and to Loek Cleophas for his detailed feedback. For their technical inputs over several years, I am grateful to (in random order)

- The FASTAR and ESPRESSO research groups, especially Fritz Venter, Ernest Ketcha, Tinus Strauss, Lorraine Liang, …

- The original Program Implementation (later Software Construction) group in Eindhoven, especially Frans Kruseman Aretz (my first promotor), Kees Hemerik (my first copromotor), Gerard Zwaan, Michiel Frishert, Tom Verhoeff, Rik van Geldrop, …

- The Prague Stringology group, particularly Bořivoj Melichar, Jan Holub, František Franěk, Jan Janoušek, Jan Žďárek, …

- The StringMasters, including Lynette van Zijl, Bill Smyth, Costas Iliopoulos, Maxime Crochemore, Jackie Daykin, Brink van der Merwe, Laurent Mouchard, …

- Computational linguists, especially Andre Kempe, Thomas Hanneforth, Johannes Bubenzer, Lauri Karttunen, Anssi Yli-Jyrä, Kimmo Koskenniemi, Jan Daciuk, Krister Lindén, Tamás Gaál, Stoyan Mihov, …

- Researchers in related fields, such as Blaine Kubesh, Jeffrey Shallit, John Brzozowski, Richard Watson, Stefan Gruner, Judith Bishop, Roelf van den Heever, …

(My apologies for anyone I have overlooked.) Last but certainly not least, thanks go to my entire family for their support — in particular to Nanette who (as always) also proofread this thesis.
Chapter 1

Introduction

In this thesis, we present algorithms for building minimal acyclic deterministic finite automata, also known as MADFAs. By their acyclic nature, they represent finite languages and are therefore useful in applications such as storing words for spell-checking (among other computational linguistics applications), computer and biological virus searching, program verification, text indexing and searching, and XML tag lookup. In each of these applications, the automata can grow extremely large (sometimes having more than $10^9$ states) and are difficult to store without first applying a minimization procedure. The specific applications are not discussed further here, but can be found in literature on stringology [Smy03, CR03, CR94], computational linguistics [JM00], information retrieval [BYRN99], data-structures [GBY91], computational biology [Gus97, Pev00] and compilers [ALSU07].

We consider both incremental and nonincremental algorithms. With nonincremental techniques, the unminimized acyclic deterministic finite automaton (ADFA) is first constructed and then minimized. As mentioned above, the unminimized ADFA can be very large indeed — often even too large to fit within the virtual memory space of the computer. As a result, incremental techniques for minimization (i.e. the ADFA is minimized during its construction) become interesting. Incremental algorithms frequently have some overhead: if the unminimized ADFA fits easily within physical memory, it may still be faster to use nonincremental techniques. On the other hand, with very large ADFAs, using an incremental technique may be the only option.

Although our main computational problem in this thesis is ‘building MADFAs’, we often refer to a particular subproblem: modify a MADFA to additionally accept a specific word $w$. By default, when we refer to an ‘algorithm’, we assume it to solve one of these two problems.

1.1 Problem statement

Although a precise definition will be given later, we can informally state the problem:

Given an alphabet $\Sigma$ and some finite set of words $W \subseteq \Sigma^*$, construct a minimal acyclic deterministic finite automaton $M$ which accepts exactly the words in $W$.

Although all of the algorithms in this thesis are presented as imperative programs, it is clear that they could also have been derived using other paradigms — for example as functional programs. Imperative programs are used to ease the translation directly to C++.

In this thesis, we are primarily interested in acyclic deterministic finite automata. The algorithms can be extended to work with acyclic deterministic transducers (automata with outputs on transitions).
CHAPTER 1. INTRODUCTION

1.2 To the reader

This thesis presents several original research contributions. As a result, there are several compelling reasons to read it:

- Very few other presentations follow a correctness-by-construction style for presenting and deriving algorithms. The presentations given here include correctness arguments or sketches thereof.

- The presentation is taxonomic — emphasizing the similarities and differences between the algorithms at a fundamental level.

- While it is possible to present these algorithms in a formal-language-theoretic setting (as was done in [Wat02b]), this thesis remains somewhat closer to the actual implementation issues.

- In several chapters, new algorithms and interesting new variants of existing algorithms are presented.

- It gives new presentations of many existing algorithms — all in a common format with common examples.

- There are extensive links to the existing literature.

The structure and style of this thesis deserves some explanation:

- In several cases, the simplest algorithms are presented in the main part of the chapter, while refinements for a practical implementation are only mentioned at the end of the chapter.

- For most of the algorithms presented in this thesis, we only mention the running times and memory requirements. No attempt is made to rigorously derive them, as this is presented elsewhere in the literature.

- There is no single chapter with global conclusions. In a sense, the algorithms (and their corresponding derivations) are themselves the ‘conclusions’ of this work. In each chapter, there are some closing comments.

1.3 Related work and a short history

A great deal of practical work on constructing and minimizing ADFAs has been done. Unfortunately, much of the research is of a proprietary nature and thus forms part of the folklore of automata algorithmics. Some of the algorithms may even have been known for some years and remained unpublished.

In the early-1990s, Dominique Revuz derived one of the first known efficient (linear in time and space) ADFA minimization algorithms [Rev91, Rev92]. The primary algorithm presented by Revuz uses an ordering of the words to quickly compress the endings of the words within the dictionary. Further work by Revuz has also yielded algorithms which correspond rather closely to some of the algorithms in this thesis\(^1\). A version of Revuz’s algorithm appears in §4.2. Recent derivations by Johannes Bubenzer (in Thomas Hanneforth’s group at Universität Potsdam) have yielded efficient

\(^1\)All minimization algorithms show strong similarities, as can be seen from the taxonomy in [Wat95]. The subtle differences between the algorithms can lead to domain-specific performance advantages for each algorithm.
new algorithms bearing a resemblance to Revuz’s [Bub11]; that work is not explicitly included in this thesis.

By the mid-1990s, several groups were working independently on incremental algorithms — most of which are the same or very similar. In Greece, Sgarbas et al derived an algorithm\textsuperscript{2} and presented it in [SFK95]. In Japan, Park et al were also deriving a related generalized algorithm [PAMS94]. At Marne-la-Valée in France, a group (including Revuz) was continuing work on related algorithms. In 1996, Richard E. Watson and I (both working at Ribbit Software Systems Inc. in Canada) completed work on the implementation of a generalized incremental algorithm for a division of Novell Corporation. (The Novell group using this particular algorithm was later acquired by Lernout & Hauspie, the now-defunct Belgian speech technology company.) Unlike many of the other derivations of related algorithms, our implementation also provides facilities for removing words from the language accepted by the automaton, while maintaining minimality\textsuperscript{3}. Owing to its commercial value, the algorithm was not published at that time. Also in 1996–1997, Jan Daciuk was completing his Ph.D. research (independently) involving the generalized algorithm. In addition, Daciuk derived a new incremental algorithm which adds the words in lexicographic order. (This is known as the \textit{sorted} algorithm.) Daciuk approached us during his literature search and we decided to combine efforts, publishing the generalized and sorted algorithms at the First Workshop on Finite State Machines in Natural Language Processing [DWW98] in Ankara, Turkey. Several papers (including ours) at that workshop were invited for submission to the Journal of Computational Linguistics. While typesetting that paper, we discovered the work of Stoyan Mihov, then a Ph.D. student in Bulgaria who had also derived the sorted algorithm, publishing it as [Mih99a]. Again, we combined efforts — with Daciuk, Mihov, Watson & Watson publishing the journal article [DMWW00]. Daciuk and Mihov also published the algorithms in their dissertations as [Dac98] and [Mih99b] respectively. Independently, in the field of program verification, Gerard Holzmann and Anuj Puri [HP98] discovered a restricted form of the algorithm, in which all words accepted by the automaton are the same length. Also independently, Marcin Ciura and Sebastian Deorowicz discovered the sorted algorithm, benchmarked it by building automata for several dictionaries and published the results as a technical report [CD99]. In early 2000, Daciuk re-examined the literature, discovering the work of Sgarbas et al and Ciura & Deorowicz. At the 2000 Conference on Implementations and Applications of Automata (CIAA\textsuperscript{4}), Dominique Revuz presented essentially the generalized algorithm [Rev00] — though he also sketched word deletion algorithms similar to those previously derived by Watson & Watson for Novell. At the 2001 CIAA, Jorge Graña et al summarized some of the current results and made improvements to several of the algorithms [GBA01]. Recently, the generalized algorithm has been straightforwardly extended by Rafael Carrasco and Mikel Forcada to handle \textit{cyclic} automata [CF02]. In this thesis, the generalized algorithm is given in Chapter 6 while the sorted algorithm is given in Chapter 9.

In early 1998, I presented the generalized incremental algorithm from memory in a seminar at Sheng Yu’s group at the University of Western Ontario, in Canada. During the presentation, I made some alternative derivation choices, arriving at a semi-incremental algorithm\textsuperscript{5}, which was then presented at the 1998 Workshop on Implementing Automata held in Rouen, France [Wat98b]. That paper was subsequently revised as a journal article in Science of Computer Programming [Wat03a] and a simplified version is given in this thesis as Chapter 11. While preparing this thesis, I derived

\textsuperscript{2}which we call the \textit{generalized} incremental algorithm, since it can add words to the automaton in any order whatsoever.

\textsuperscript{3}In [Rev00], Revuz sketches algorithms related to the word deletion ones presented in this thesis.

\textsuperscript{4}Formerly the Workshop on Implementations and Applications of automata.

\textsuperscript{5}A \textit{semi-incremental} algorithm in this context is one which does much of the minimization work incrementally (as words are added), but still requires a final ‘cleanup’ phase.
a simplified version of the semi-incremental algorithm — also based on adding words in any order of decreasing length. That simplified algorithm is not previously known from the literature and is presented in Chapter 10.

By 1999, I had started work on a taxonomy of the known algorithms. It was subsequently presented at the 1999 Workshop on Implementing Automata (the predecessor of CIAA) in Potsdam, Germany [Wat99c] and as a journal article in the South African Computer Journal [Wat01e]. One of the algorithms presented in that taxonomy can be derived by combining an automata construction and a minimization algorithm — both by Brzozowski [Wat00a, Wat02a]. An alternative derivation of the same algorithm is given in [Wat02b]. The resulting algorithm appears in this thesis as Chapter 7.

Also in 2000, I wrote a book chapter covering an elegant new recursive algorithm [Wat03b]. An elaborated version is presented as [Wat01d]. That algorithm can be viewed as a recursive rendition of the one in Chapter 6.

1.4 Links to the literature

The relationships between literature and the algorithms presented here is given in the following table:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Algorithm</th>
<th>Literature</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Revuz</td>
<td>[Rev91, Rev92] and new. Recent work by Bubenzer [Bub11] is not explicitly included in this thesis</td>
</tr>
<tr>
<td>5</td>
<td>Naïve</td>
<td>New</td>
</tr>
<tr>
<td>6</td>
<td>Fully incremental</td>
<td>[SFK95, PAMS94, DWW98, Mih99a, DMWW00, Dac98, Mih99b, HP98, Rev00, GBA01]</td>
</tr>
<tr>
<td>7</td>
<td>Double reversal (Brzozowski)</td>
<td>[Wat99c, Wat01e, Wat00a, Wat02a, Wat02b]</td>
</tr>
<tr>
<td>9</td>
<td>Words in lexicographic order</td>
<td>[Mih99a, Dac98, Mih99b, CD99]</td>
</tr>
<tr>
<td>10</td>
<td>Decreasing length (depth layers)</td>
<td>New</td>
</tr>
<tr>
<td>11</td>
<td>Decreasing length (semi-incremental)</td>
<td>[Wat98b, Wat03a]</td>
</tr>
</tbody>
</table>

Typically, the algorithms presented here include more detailed correctness arguments than their previous presentations in the literature.

1.5 Future work

Like all work in algorithmics, the results presented here are only the beginning. There are several promising directions for future work:

- Construct a toolkit (C++ library) of the algorithms and benchmark them in the style of [Wat95, Cle08]. This work has already begun, though is not advanced enough to report here.
- Derive and present the algorithms here in another paradigm, for example, functionally. This may lead to cleaner derivations and further insights.
• Explore parallelization of these algorithms, or the derivation of entirely new types of parallel algorithms solving the same problem. Recent related work in [SKW08] indicates promising directions.
Chapter 2

Preliminaries

In this chapter, we present the necessary mathematical preliminaries (including many properties not often given in the literature), assuming readers have a working knowledge of automata and formal languages. For such a background, see [HU79, Wat95, Smy03, CR03].

§2.1 and §2.2 lay notational foundations, including the style of algorithm presentation; §2.3 gives numerous definitions related to strings and languages, while §2.4 introduces automata and their key properties, and finally §2.5 details the minimality of automata and algorithms for minimizing them.

2.1 General definitions

In this section, we present some general notational definitions.

**Notation 2.1 (Quantification)** We assume a basic understanding of the meaning of quantification. We use the notation

\[
\langle \oplus a : R(a) : f(a) \rangle
\]

where \( \oplus \) is an associative and commutative operator (to be quantified) with unit 1\( \oplus \), \( a \) is a dummy variable, \( R \) is a range predicate, and \( f(a) \) is the quantification expression. By definition, when the range is empty (the predicate is false), the entire quantification evaluates to the unit 1\( \oplus \). For example, \( a \forall \) quantification over an empty range evaluates to true because the quantified operator is \( \land \) which has unit true.

**Notation 2.2 (Conditional Boolean operators)** We use \( \text{cand} \) and \( \text{cor} \) to refer to the conditional (also known as ‘short circuit’) equivalents of \( \land \) and \( \lor \) (respectively).

**Notation 2.3 (Powerset)** For any set \( A \), we use \( \mathcal{P}(A) \) to denote the set of all subsets (including the empty set, \( \emptyset \)) of \( A \).

**Notation 2.4 (Set difference)** For any sets \( A, B \), we use \( A - B \) to denote set difference instead of the \( A \setminus B \) sometimes seen in the literature.

**Notation 2.5 (Derivation)** We adopt the ‘Eindhoven’ style of derivational proof, in which each derivation step appears on its own line, separated by a derivation operator (typically \( \equiv, \Rightarrow, =, \text{etc.} \)) and ‘hint’ of why the step is valid. For example
\[ P_0 \equiv \text{"some reason why } P_1 \text{ should obviously be equivalent to } P_0 \text{"} \]

Note of course that this step could have involved implication, or any of the other derivation operators.

### 2.2 Algorithm presentation

In this thesis, the abstract algorithms are presented in Dijkstra’s guarded command language [Dij76, Gri80], while the concrete implementation details are presented in C++. For ease of presentation, we take a number of notational shortcuts in the abstract algorithm presentations:

- We use the as-sa statement: many if-fi statements involve one branch guarded by a predicate \( R \) and another skip branch guarded by \( \neg R \); in that case, we simplify to an as-sa statement with the single \( R \)-guarded statement.

- Since all of the algorithms presented in this thesis operate on one finite automaton (constructing it from a finite set of words), we simply assume the automaton and set of words are global variables. This is in contrast to explicitly passing them to program functions, procedures, predicates (as used in pre- and post-conditions and invariants), etc. The aim of this shortcut is to reduce typesetting and presentation clutter. Naturally, a real implementation would limit the scope of such variables for software engineering reasons.

- We often use shadow variables in predicates (preconditions, postconditions, invariants, etc.) to capture an old value of some variable or property, allowing us to later reason about the ‘old’ value. Shadow variables are typeset the same as program variables.

- The notation \( x : S \) designates a statement \( S \) to be refined in such a way that it may modify only variable \( x \). Since the majority of the statements may modify the global automaton (and its subcomponents), we do not explicitly mention it in this ‘frame’.

Work on a C++ toolkit of the algorithms is ongoing; it will be available via www.fastar.org when complete.

### 2.3 Strings and languages

In this section, we present definitions and properties related to strings and languages.

**Definition 2.6 (Alphabet)** An alphabet is a finite nonempty set of symbols — also known as letters. Throughout this thesis, we assume a fixed alphabet \( \Sigma \).

**Notation 2.7** \( \Sigma^* \) denotes the set of all words over \( \Sigma \) — including the empty string, written as \( \varepsilon \). Furthermore, we define \( \Sigma^+ = \Sigma^* - \{\varepsilon\} \).

**Definition 2.8 (String operators)** Define string head and tail operators \( \text{head} \in \Sigma^+ \to \Sigma \) and \( \text{tail} \in \Sigma^+ \to \Sigma^* \) (for \( a \in \Sigma, v \in \Sigma^* \)) as

\[ \text{head}(av) = a \]
and
\[ \text{tail}(av) = v \]

Without explicitly defining them, we can extend these two operators as needed to sequences of other types.

**Definition 2.9 (String and language reversal)** Given string \( w \), define \( w^R \) to be the reversal of \( w \), i.e. in which the letters appear in reverse order. Inductively (for \( a \in \Sigma \)), \( \varepsilon^R = \varepsilon \) and \( (aw)^R = w^Ra \). For a language \( L \), define \( L^R = \{ w^R \mid w \in L \} \).

**Definition 2.10 (Alphabet ordering)** We assume a total ordering \( \leq \) on alphabet \( \Sigma \). (This is typically the ASCII ordering.) By extension, we also have the ordering \( < \) on \( \Sigma \).

Some of the algorithms in this thesis require the lexicographic ordering (also known as the ‘telephone book’ ordering) on words in \( \Sigma^* \).

**Definition 2.11 (Lexicographic ordering of \( \Sigma^* \))** For simplicity, we begin with the ordering \( \sqsubseteq_1 \) on \( \Sigma^* \). For all \( a, b \in \Sigma \) and \( v, w \in \Sigma^* \)
\[ \varepsilon \sqsubseteq_1 av \]
and
\[ av \sqsubseteq_1 bw \equiv \begin{cases} v \sqsubseteq_1 w & \text{if } a = b \\ a < b & \text{otherwise} \end{cases} \]

The lexicographic ordering \( \sqsubseteq_1 \) is a total ordering on \( \Sigma^* \), defined as
\[ v \sqsubseteq_1 w \equiv (v = w \lor v \sqsubseteq_1 w) \]

**Example 2.12 (Lexicographic order)** The words had, hard, he, head, heard, her, herd, and here are in lexicographic order.

**Definition 2.13 (Longest common prefix)** Given two words \( v, w \in \Sigma^* \), define \( v \triangleleft w \) as the longest common prefix of \( v \) and \( w \). This can also be given inductively (where additionally \( a, b \in \Sigma : a \neq b \))
\[ \varepsilon \triangleleft v = v \triangleleft \varepsilon = \varepsilon \]
and
\[ av \triangleleft bw = \varepsilon \]
and
\[ av \triangleleft aw = a(v \triangleleft w) \]

**Example 2.14 (Longest common prefix)**
head \( \triangleleft \) heard = hea

**Definition 2.15 (Left derivative)** Given two strings \( v, w \) such that \( v \) is a prefix of \( w \), we define the left \( v \)-derivative of \( w \), written \( v^{-1}w \), as the unique string such that
\[ w = v(v^{-1}w) \]

**Example 2.16 (Left derivative)**
her\(^{-1}\) herd = d

Derivatives were introduced by Brzozowski in [Brz62b]. There is a symmetrical notion of right derivatives, though we have no need for them in this thesis.
2.4 Automata

In this section, we present automata and related properties.

**Definition 2.17 (Deterministic finite automata)** A deterministic finite automaton (a DFA — also used to denote the set of all such automata) is a quadruple \((Q, \delta, s, F)\) where

- \(Q\) is a finite nonempty set of states.
- \(\delta \in Q \times \Sigma \xrightarrow{\text{partial}} Q\) is the (possibly partial) transition function. We use \(\perp\) to denote the invalid destination state of a transition, thus the signature could have been written as a total function\(^1\) \(\delta \in Q \times \Sigma \rightarrow Q \cup \{\perp\}\).
- \(s \in Q\) is the start state.
- \(F \subseteq Q\) is the set of final states.

In the literature, a common interpretation is to view \(\perp\) as a special state, such as a sink state. In this thesis, \(\perp\) rather indicates an undefined part of \(\delta\). The sink state interpretation would lead to cyclic automata — undesirable in this thesis.

Throughout this thesis, we assume a specific DFA \(M = (Q, \delta, s, F)\). Many functions and predicates take \((Q, \delta, s, F)\) as their only argument. To avoid notational clutter, where the meaning is clear we will omit the argument and assume it implicitly.

In this thesis, we will not need nondeterministic finite automata.

**Definition 2.18 (Abstract program procedures and types for automata)** Program type \(\text{STATE}\) is a universe of states. We also assume a number of program functions/procedures\(^2\):

- create() : \(\text{STATE}\)
  Create a new state (without out-transitions) in \(M\) (taken from \(\text{STATE}\)).
- clone(p : \(\text{STATE}\)) : \(\text{STATE}\)
  Create and return a new state with the same out-transitions and 'finality' as \(p\).
- merge(p, q : \(\text{STATE}\))
  Assume \(p, q : p \neq q\) are equivalent (discussed later); redirect all of \(p\)'s in-transitions to \(q\), and delete \(p\).

**Notation 2.19 (Drawing DFAs)** We draw the automata in the standard way, depicting states as ellipses, start states having an in-edge from nowhere and final states being two concentric ellipses. Transitions are depicted as labeled directed edges, as in

---

\(^1\)Note that numerous other isomorphic signatures are possible, for example \(\delta \in Q \xrightarrow{\text{partial}} \mathcal{P}(\Sigma \times Q)\).

\(^2\)Note that these procedures implicitly take our DFA \(M\) as an additional argument, and we assume that they update the components of \(M = (Q, \delta, s, F)\) as needed.
Definition 2.20 (ADFA) ADFA is the set of all DFAs with acyclic transition graphs.

Definition 2.21 (Size of a DFA) The size of \( M \), written \( |M| \), is defined as \( |Q| \).

Other notions of size are possible, for example, involving the total number of transitions. We do not consider them here.

Notation 2.22 For a state \( p \), \( \Sigma_p \) denotes the subset of \( \Sigma \) on which \( p \) has out-transitions. That is,

\[
\Sigma_p = \{ a \mid a \in \Sigma \land \delta(p, a) \neq \bot \}
\]

Definition 2.23 (Confluence state) A state \( p \) is a confluence state, written \( Is_{\text{confl}}(p) \), iff it has more than one in-transition. In the literature, these are sometimes also known as ‘re-entrant’ states, though we avoid that term. In Notation 2.19, state 3 is a confluence.

Definition 2.24 A set of states \( X \) is confluence-free, written \( Confl_{\text{free}}(X) \), iff

\[
(\forall p : p \in X : \neg Is_{\text{confl}}(p))
\]

Property 2.25 (Merging states) For states \( p, q : p \neq q \) where both have in-transitions, merge\((p, q)\) leaves \( q \) as a confluence state. (Recall that \( p \) is deleted.)

Definition 2.26 (Useless state) A state \( p \) is useless if there is no path from the start state to \( p \), or there is no path from \( p \) to a final state.

In this thesis, most algorithms will be designed to not introduce useless states; similarly, most of our definitions and properties will require the assumption of no useless states. We have, however, taken one minor short-cut: in a DFA accepting the empty language, the start state \( s \) is not final and is therefore, strictly speaking, useless. To keep the algorithms simple, we ignore this corner case.

Example 2.27 In the following (unconnected) DFA, both states are useless.

Property 2.28 Since we have no useless states, in an ADFA the start state \( s \) is never a confluence state.

Definition 2.29 (Trie) \( M \) is a trie, written \( Is_{\text{trie}} \) (we assume \( M \) in our global scope), iff its transition graph is a tree rooted at start state \( s \).

Property 2.30 (Tries) Tries have no confluence states.

Example 2.31 The following DFA is a trie:
Property 2.32 If a trie has no useless states then all leaves are final states.

Definition 2.33 (Extending $\delta$) We extend $\delta$ to $\delta^* : Q \times \Sigma^* \rightarrow Q$ as

$$\delta^*(p, \varepsilon) = p$$

and (for $a \in \Sigma, v \in \Sigma^*$)

$$\delta^*(p, av) = \begin{cases} \delta^*(\delta(p, a), v) & \text{if } a \in \Sigma_p \\ \bot & \text{otherwise} \end{cases}$$

Definition 2.34 (Right language of a state) The right language of a state $p$, denoted $\overrightarrow{L}(p)$, is defined by

$$\overrightarrow{L}(p) = \{ w | \delta^*(p, w) \in F \}$$

That is, $\overrightarrow{L}(p)$ is the set of strings on paths from $p$ to any final state.

Property 2.35 $q \in F \equiv \varepsilon \in \overrightarrow{L}(q)$.

Definition 2.36 (Left language of a state) The left language of a state $p$, denoted $\overleftarrow{L}(p)$, is defined by

$$\overleftarrow{L}(p) = \{ w | \delta^*(s, w) = p \}$$

That is, $\overleftarrow{L}(p)$ is the set of strings on paths from start state $s$ to state $p$. 
Property 2.37 For a state \( p \) which is not useless, we have \( \leftarrow L(p) \neq \emptyset \) and \( \rightarrow L(p) \neq \emptyset \).

Example 2.38 In Notation 2.19, \( \rightarrow L(0) = \leftarrow L(3) = \{ab, ba\} \).

Property 2.39 (Recursive definition of \( \rightarrow L \)) The recursive definition of \( \delta^* \) can be used to give a recursive definition for \( \rightarrow L \) as follows:

\[
\rightarrow L(q) = \bigcup_{a \in \Sigma} (\rightarrow L(\delta(q, a))) \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{if } q \notin F \end{cases}
\]

Phrased differently, a string \( v \) is in \( \rightarrow L(q) \) iff

- \( v \) is of the form \( aw \) where \( a \in \Sigma \) is a label of an out-transition from \( q \) to \( \delta(q, a) \) (i.e. \( a \in \Sigma_q \)) and \( w \) is in the right language of \( \delta(q, a) \), or
- \( v = \varepsilon \) and \( q \) is a final state.

A recursive definition of \( \leftarrow L \) is not required in this thesis.

Definition 2.40 (Language of a DFA) The language accepted by DFA \( M \), denoted \( \mathcal{L} \), is defined by

\[
\mathcal{L} = \rightarrow L(s)
\]

Note that we could also have defined \( \mathcal{L} \) using left languages as

\[
\langle \bigcup f : f \in F : \leftarrow L(f) \rangle
\]

Definition 2.41 (Path through a DFA) For state \( p \) and \( w \in \Sigma^* \),

\[
[p \xrightarrow{w}]
\]

is the sequence of states \( p, \ldots, \delta^*(p, \nu) \) where \( \nu \) is the longest prefix of \( w \) such that \( \delta^*(p, \nu) \neq \bot \). We refer to this as the single “\( w \)-path from state \( p \).”

Notation 2.42 (Path through a DFA) We use the standard parentheses notation to denote state sequences which are open at the beginning or end — for example \( [p \xrightarrow{w}] \) does not include \( p \) but does include the rest of \( [p \xrightarrow{w}] \). In some contexts, we may pass a path \( [p \xrightarrow{w}] \) as an argument to a predicate or function which expects a set, thereby implicitly treating the path as a set of states.

Property 2.43 We can give a recursive definition for \( [p \xrightarrow{w}] \):

\[
[p \xleftarrow{w}] = p
\]

and, for all \( a \in \Sigma, w \in \Sigma^* \) (where \( \cdot \) is sequence concatenation and \( \varepsilon \) is the empty sequence which some authors write as \( [] \))

\[
[p \xrightarrow{aw}] = p \cdot \begin{cases} [\delta(p, a) \xrightarrow{w}] & \text{if } a \in \Sigma_p \\ \varepsilon & \text{otherwise} \end{cases}
\]
Definition 2.44 (Reachability of states) \( \text{Succ} \) is a binary relation on states defined as
\[
\text{Succ}(p, q) \equiv \exists a : a \in \Sigma_p : \delta(p, a) = q
\]
Note that \( \text{Succ} \) is essentially \( \delta \) with the \( \Sigma \) component projected away.

Definition 2.45 (\( \text{Succ}^+ \) and \( \text{Succ}^* \)) \( \text{Succ}^* \) (respectively \( \text{Succ}^+ \)) is the reflexive and transitive (respectively reflexive-only) closure of \( \text{Succ} \). \( \text{Succ}^*(p, q) \) iff there is a path from \( p \) to \( q \) in the transition graph.

It follows that \( \text{Succ}^* \) (respectively \( \text{Succ}^+ \)) is essentially \( \delta^* \) (respectively \( \delta^+ \)) with the \( \Sigma^* \) (respectively \( \Sigma^+ \)) component projected away.

Notation 2.46 We will also use \( \text{Succ} \) as a function, mapping a state to its successor states. In this context,
\[
\text{Succ}(p) = \{ \delta(p, a) \mid a \in \Sigma_p \}
\]
We extend \( \text{Succ} \) to taking a set of states by distributing \( \text{Succ} \) over \( \cup \). We similarly extend the signatures of \( \text{Succ}^* \) and \( \text{Succ}^+ \).

Property 2.47 (Recursive forms of \( \text{Succ}^+ \) and \( \text{Succ}^* \)) For any state \( p \), we have
\[
\text{Succ}^+(p) = "definition of reflexive closure"
\]
\[
\text{Succ}^*(\text{Succ}(p)) = "Definition 2.44"
\]
\[
\text{Succ}^*\langle \cup a : a \in \Sigma_p : \delta(p, a) \rangle = "\text{Succ} and \text{Succ}^* \text{ distribute over } \cup"
\]
\[
\text{Succ}^*(\delta(p, a)) = \langle \cup a : a \in \Sigma_p : \text{Succ}^*(\delta(p, a)) \rangle
\]
Using the above derivation we have
\[
\text{Succ}^+(p) = "definition of reflexive and transitive closure"
\]
\[
\{p\} \cup \text{Succ}^+(p) = "derivation above"
\]
\[
\{p\} \cup \langle \cup a : a \in \Sigma_p : \text{Succ}^*(\delta(p, a)) \rangle
\]
(These are well-formed because our automata are acyclic and finite. Similar definitions are valid for cyclic automata, and are then based on a fixed-point of such recursive equations.)

Property 2.48 (Confluence-free state paths) A confluence-free path \( [s \leadsto^\sim] \) allows us to characterize the successors of the remaining states (not on that path):
\[
\text{Confl\_free}([s \leadsto^\sim]) \equiv "\text{definitions of } [s \leadsto^\sim] \text{ and confluence state; state } s \text{ has no in-transitions}"
\]
\[
each state in (s \leadsto^\sim) has a single in-transition from its predecessor, which is in [s \leadsto]
\]
\[
\Rightarrow "\text{transitions from other states } Q - [s \leadsto] \text{ cannot go to } (s \leadsto)"
\]
2.4. AUTOMATA

\[ \text{Succ}(Q - [s \xrightarrow{u}]) \cap (s \xrightarrow{u}) = \emptyset \]
\[ \equiv \ "\text{set calculus}" \]
\[ \text{Succ}(Q - [s \xrightarrow{u}]) \subseteq Q - (s \xrightarrow{u}) \]
\[ \equiv \ "\text{state s has no in-transition (and is not a successor of any state)}" \]
\[ \text{Succ}(Q - [s \xrightarrow{u}]) \subseteq Q - [s \xrightarrow{u}] \]

Definition 2.49 (Longest right word length function) For an ADFA only\(^3\), define \( \overrightarrow{L}_{\max} \in Q \rightarrow \mathbb{N} \) as
\[ \overrightarrow{L}_{\max}(p) = \langle \text{MAX } w : w \in \overrightarrow{L}(p) : |w| \rangle \]
\( \overrightarrow{L}_{\max}(p) \) is the length of the longest path from \( p \) to any final state in an ADFA. In [Rev92], Revuz calls \( \overrightarrow{L}_{\max}(p) \) the ‘height’ of \( p \).

Property 2.50 (Function \( \overrightarrow{L}_{\max} \)) The recursive definition of \( \overrightarrow{L} \) can be used to give a recursive definition for \( \overrightarrow{L}_{\max} \):
\[ \overrightarrow{L}_{\max}(p) = (\langle \text{MAX } a : a \in \Sigma_p : \overrightarrow{L}_{\max}(\delta(p, a)) \rangle + 1) \max \begin{cases} 0 & \text{if } p \in F \\ -\infty & \text{if } p \not\in F \end{cases} \]
The above expression deserves some explanation: in the event that \( p \) has no out-transitions, the \( \max \) quantification has an empty range and evaluates to the unit of \( \max \), namely \(-\infty\). The righthand-side of the infix \( \max \) considers the case that \( p \) is a final state, in which case its right language contains \( \varepsilon \), of length 0. Since we usually have no useless states in this thesis, we cannot have a non-final state without out-transitions, and therefore we could simply use 0 as the second operand of the infix \( \max \).

Example 2.51 In Example 2.31, \( \overrightarrow{L}_{\max}(3) = 0 \), \( \overrightarrow{L}_{\max}(6) = 3 \) and \( \overrightarrow{L}_{\max}(0) = 5 \).

Property 2.52 It follows from Property 2.50 that, for all states \( p \)
\[ \langle \forall a : a \in \Sigma_p : \overrightarrow{L}_{\max}(p) \geq \overrightarrow{L}_{\max}(\delta(p, a)) + 1 \rangle \]

Definition 2.53 (Height levels) In an ADFA, for each \( k \in \mathbb{N} \) we define a set of states
\[ \text{HL}_k = \{ p \mid p \in Q \land \overrightarrow{L}_{\max}(p) = k \} \]
State layer sets \( \text{HL}_k \) are a partition of \( Q \). (That is, they are disjoint, and every state appears in some layer.)

Example 2.54 In the trie of Example 2.31,
\[ \text{HL}_0 = \{3, 5, 8, 10, 12, 13\} \]
\[ \text{HL}_1 = \{4, 9, 11\} \]
\[ \text{HL}_2 = \{2, 7\} \]
\[ \text{HL}_3 = \{6\} \]
\[ \text{HL}_4 = \{1\} \]
\[ \text{HL}_5 = \{0\} \]

---

\(^3\)This restriction is placed because a cyclic DFA may have arbitrarily long paths (following a cycle) from a state to a final state.
Property 2.55 (Procedure merge) Merging state $p$ into $q$ with an invocation $\text{merge}(p, q)$ (which is only valid when $\vec{L}(p) = \vec{L}(q)$) does not change $\vec{L}(q)$. Consequently, $\vec{L}_{\text{max}}(q)$ remains unchanged and $q$ remains in the same height level as before the invocation of $\text{merge}$.

Property 2.56 (Height levels) For $k \geq 0$

$$\text{HL}_k = \emptyset \Rightarrow \text{HL}_{k+1} = \emptyset$$

This follows from the following contrapositive argument

$$\Rightarrow \text{ "Definition 2.53" }$$

$$\langle \exists p : p \in Q : \vec{L}_{\text{max}}(p) = k + 1 \rangle$$

$$\Rightarrow \text{ "Property 2.50; $M$ has no useless states" }$$

$$\langle \exists p : p \in Q : (\langle \text{MAX } a : a \in \Sigma_p : \vec{L}_{\text{max}}(\delta(p, a)) \rangle + 1) \text{ max } 0 = k + 1 \rangle$$

$$\Rightarrow \text{ "drop max } 0 \text{ because } k \geq 0 \text{ from assumption and so } k + 1 \geq 1"$$

$$\langle \exists p : p \in Q : \langle \text{MAX } a : a \in \Sigma_p : \vec{L}_{\text{max}}(\delta(p, a)) \rangle + 1 = k + 1 \rangle$$

$$\Rightarrow \text{ "arithmetic" }$$

$$\Rightarrow \text{ "quantify states } \delta(p, a)"$$

$$\langle \exists q : q \in Q : \vec{L}_{\text{max}}(q) = k \rangle$$

$$\Rightarrow \text{ "Definition 2.53" }$$

$$\text{HL}_k \neq \emptyset$$

Definition 2.57 (State depth function) Function $\vec{L}_{\text{min}} : Q \rightarrow \mathbb{N}$ is defined as

$$\vec{L}_{\text{min}}(p) = \langle \text{MIN } w : w \in \vec{L}(p) : |w| \rangle$$

$\vec{L}_{\text{min}}(p)$ is the length of the shortest path from $s$ to $p$. In the literature, $\vec{L}_{\text{min}}(p)$ is also known as the ‘depth’ of $p$. Note the asymmetry of functions $\vec{L}_{\text{max}}$ and $\vec{L}_{\text{min}}$. We will not need a recursive definition of $\vec{L}_{\text{min}}$.

Definition 2.58 (Depth levels) In an ADFA, for each $k \in \mathbb{N}$ we define a set of states at ‘depth level $k$’

$$\text{DL}_k = \{ p \mid p \in Q \wedge \vec{L}_{\text{min}}(p) = k \}$$

Property 2.59 The depth levels form a partition of $Q$.

Notation 2.60 (Depth levels) We use the following notational short-hand for $k, l \in \mathbb{N}$

$$\text{DL}_{> k} = \langle \cup j : j > k : \text{DL}_j \rangle$$

and

$$\text{DL}_{\leq k} = \langle \cup j : 0 \leq j \leq k : \text{DL}_j \rangle$$

We can analogously define $\text{DL}_{(k, 1)}$, $\text{DL}_{(k, 1]}$, etc.
2.5. MINIMALITY OF AUTOMATA

**Definition 2.61 (Shortest word length of a DFA)** Function minlen is the length of the shortest word accepted by $M$. Formally,

$$\text{minlen} = \langle \text{MIN} f : f \in F : \mathcal{L}_{\text{min}}(f) \rangle$$

Clearly, minlen is the depth of a final state closest (in terms of path-length) to start state $s$.

**Definition 2.62 (Lexicographically greatest word)** Define lexmax as the lexicographically greatest word in $\mathcal{L}$. Formally,

$$\text{lexmax} = \langle \text{MAX} w : w \in \mathcal{L} : w \rangle$$

**Property 2.63 (Lexicographically greatest word)** lexmax can be found structurally in $M$ by beginning at start state $s$ and always following the out-transition on the highest ranked letter (under $\leq$) until no further out-transitions are possible.

**Property 2.64** lexmax is unique since $\subseteq$ is total.

**Property 2.65** Note that if $\mathcal{L} = \emptyset$ then lexmax = $\varepsilon$ since $\varepsilon$ is the unit of $\max \subseteq$.

2.5 Minimality of automata

In this section, we present definitions and properties related to the minimality of automata, and follow this with two algorithms to determine the equivalence of states. Many of these properties and definitions do not appear explicitly in the literature.

**Definition 2.66 (Minimality of a DFA)** $M$ is minimal, written $\text{Min}(M)$ or simply $\text{Min}$, iff it is the smallest (as measured by the number of states — see Definition 2.21) DFA accepting $\mathcal{L}$.

**Property 2.67** A minimal DFA is unique up to isomorphism — see [HU79, §3.4].

**Definition 2.68 (MADFA)** MADFA is the set of all minimal ADFAs.

**Definition 2.69 (State equivalence)** Define $E$ as an equivalence relation on states where

$$E(p, q) = (\mathcal{L}^*(p) = \mathcal{L}^*(q))$$

If two states $p, q$ are equivalent under $E$, $p$ can be merged into $q$ using procedure merge — including some redirection of transitions — giving a smaller equivalent automaton.

**Property 2.70** Assuming no useless states, start state $s$ is unique — that is, it is not equivalent to any other state.

**Property 2.71 (Recursive definition of $E$)** The recursive definition of $\mathcal{L}^*$ (Property 2.39) gives rise to a recursive definition of $E$ as follows
\[ E(p, q) \equiv \text{"definition of } E \text{"} \]
\[ \overline{L}(p) = \overline{L}(q) \]
\[ \equiv \text{"definition of language equality"} \]
\[ (\forall v : v \in \Sigma^* : v \in \overline{L}(p) \equiv v \in \overline{L}(q)) \]
\[ \equiv \text{"split domain } \Sigma^* \text{ into } \{\epsilon\} \cup \Sigma^+ \text{"} \]
\[ (\forall v : v \in \{\epsilon\} \cup \Sigma^+ : v \in \overline{L}(p) \equiv v \in \overline{L}(q)) \]
\[ \equiv \text{"split quantification"} \]
\[ (\forall v : v \in \{\epsilon\} : v \in \overline{L}(p) \equiv v \in \overline{L}(q)) \]
\[ \land (\forall v : v \in \Sigma^+ : v \in \overline{L}(p) \equiv v \in \overline{L}(q)) \]
\[ \equiv \text{"one-point rule on the first universal quantification"} \]
\[ (\epsilon \in \overline{L}(p) \equiv \epsilon \in \overline{L}(q)) \]
\[ \land (\forall v : v \in \Sigma^+ : v \in \overline{L}(p) \equiv v \in \overline{L}(q)) \]
\[ \equiv \text{"introduce dummies } a \in \Sigma, w \in \Sigma^* \text{ such that } v = aw \text{ in second quantification"} \]
\[ (\epsilon \in \overline{L}(p) \equiv \epsilon \in \overline{L}(q)) \]
\[ \land (\forall a, w : a \in \Sigma, w \in \Sigma^* : aw \in \overline{L}(p) \equiv aw \in \overline{L}(q)) \]
\[ \equiv \text{"in context, } aw \in \overline{L}(p) \equiv (a \in \Sigma_p \text{ cand } w \in \overline{L}(\delta(p, a))) \"
\[ (p \in F \equiv q \in F) \land (\forall a, w : a \in \Sigma, w \in \Sigma^* : aw \in \overline{L}(p) \equiv aw \in \overline{L}(q)) \]
\[ \equiv \text{"split universal quantifier; cand no longer needed with } a \in \Sigma_p \cap \Sigma_q \text{ in quantifier range"} \]
\[ (p \in F \equiv q \in F) \land (\forall a : a \in \Sigma : a \in \Sigma_p \equiv a \in \Sigma_q) \]
\[ \land (\forall a, w : a \in \Sigma_p \cap \Sigma_q, w \in \Sigma^* : w \in \overline{L}(\delta(p, a)) \equiv w \in \overline{L}(\delta(q, a))) \]
\[ \equiv \text{"definition of alphabet equality"} \]
\[ (p \in F \equiv q \in F) \land \Sigma_p = \Sigma_q \]
\[ \land (\forall a : a \in \Sigma_p \cap \Sigma_q, w \in \Sigma^* : w \in \overline{L}(\delta(p, a)) \equiv w \in \overline{L}(\delta(q, a))) \]
\[ \equiv \text{"definition of language equality"} \]
\[ (p \in F \equiv q \in F) \land \Sigma_p = \Sigma_q \land (\forall a : a \in \Sigma_p \cap \Sigma_q : E(\delta(p, a), \delta(q, a))) \]
\[ \equiv \text{"definition of } E \text{"} \]
\[ (p \in F \equiv q \in F) \land \Sigma_p = \Sigma_q \land (\forall a : a \in \Sigma_p \cap \Sigma_q : E(\delta(p, a), \delta(q, a))) \]

**Corollary 2.72** In an ADFA, for all states p, q
\[ \overline{L}_{\text{max}}(p) > \overline{L}_{\text{max}}(q) \]
\[ \Rightarrow \text{"} \overline{L}(p) \text{ has a word of length } \overline{L}_{\text{max}}(p) \text{ that is not in } \overline{L}(q) \text{"} \]
\[ (\exists v : v \in \overline{L}(p) \land |v| = \overline{L}_{\text{max}}(p) : v \not\in \overline{L}(q)) \]
2.5. MINIMALITY OF AUTOMATA

⇒ “language equality”
\[ \text{L}(p) \neq \text{L}(q) \]
⇒ “definition of E”
\[ \neg E(p, q) \]

Furthermore
\[ \text{L}_{\text{max}}(p) \neq \text{L}_{\text{max}}(q) \]
≡ “definition of ≠”
\[ \text{L}_{\text{max}}(p) > \text{L}_{\text{max}}(q) \lor \text{L}_{\text{max}}(p) < \text{L}_{\text{max}}(q) \]
⇒ “derivation above”
\[ \neg E(p, q) \lor \neg E(q, p) \]
≡ “without loss of generality; E is an equivalence relation”
\[ \neg E(p, q) \]

Corollary 2.73 In an ADFA, for all states p
\[ \langle \forall u : u \in \Sigma^+ \land \delta^*(p, u) \neq \bot : \neg E(p, \delta^*(p, u)) \rangle \]
This follows from Property 2.52 and Corollary 2.72.

Corollary 2.74 In an ADFA, for all states p, q
\[ \text{Succ}^+(p, q) \]
≡ “Property 2.47”
\[ \langle \exists u : u \in \Sigma^+ : \delta^*(p, u) = q \rangle \]
⇒ “Corollary 2.73”
\[ \neg E(p, q) \]

Definition 2.75 (Pairwise inequivalent pair of sets) Given two state sets X, Y \( \subseteq Q \), X is said to be pairwise inequivalent (PI) against Y, written
\[ \text{Pairwise}_\text{inequiv}(X, Y) \]
iff all pairs of states (taken respectively from X, Y) are inequivalent. Formally,
\[ \text{Pairwise}_\text{inequiv}(X, Y) \equiv \langle \forall p, q : p \neq q \land p \in X \land q \in Y : \neg E(p, q) \rangle \]

Property 2.76 (Pairwise_inequiv) For three state sets X, Y, Z
\[ \text{Pairwise}_\text{inequiv}(X \cup Y, Z) \equiv \text{Pairwise}_\text{inequiv}(X, Z) \land \text{Pairwise}_\text{inequiv}(Y, Z) \]

Definition 2.77 (Pairwise inequivalent states) Define predicate
\[ \text{Inequiv}(X) \equiv \langle \forall p, q : p \neq q \land p \in X \land q \in X : \neg E(p, q) \rangle \]
When Inequiv(X) holds, we can take a notational shortcut and say that states X are ‘minimized’.

Property 2.78 Note that Inequiv(X) \( \equiv \text{Pairwise}_\text{inequiv}(X, X) \). These two predicates could have been combined; they are separately defined for readability.
Corollary 2.79 Thanks to Corollaries 2.73 and 2.74 and the definition of \( \sim \), we have (for any state \( p \) and string \( w \)) \( \text{Inequiv}(\llbracket p \sim^w \rrbracket) \). (Here, we view \( \llbracket p \sim^w \rrbracket \) as a set.) Note that this holds even when \( w = \epsilon \), because \( \llbracket p \sim^\epsilon \rrbracket = p \).

**Property 2.80 (Inequivalence of levels)** From Definition 2.53, Corollary 2.72, and Definition 2.75, we have (for \( i, j : i \neq j \))

\[
\text{Pairwise}_\text{inequiv}(\text{HL}_i, \text{HL}_j)
\]

**Property 2.81** From Property 2.50, we have for all \( k > 0 \)

\[
\text{Succ}(\text{HL}_k) \subseteq \langle \cup j : 0 \leq j < k : \text{HL}_j \rangle
\]

Note that, in general, we do not have

\[
\text{Succ}(\text{HL}_k) \subseteq \text{HL}_{k-1}
\]

since states in \( \text{HL}_k \) may have some successors in levels strictly less than \( k - 1 \).

**Property 2.82 (Minimality of a DFA)** \( \text{Min} \) is equivalent to:

- all states in \( Q - \{s\} \) are useful, and
- \( \text{Inequiv}(Q) \).

This is shown in [HU79, §3.4].

**Property 2.83** Given sets of states \( X, Y \),

\[
\text{Inequiv}(X \cup Y) \\
\equiv \text{“definition of Inequiv”} \\
\langle \forall p, q : p \neq q \land p, q \in (X \cup Y) : \neg E(p, q) \rangle \\
\equiv \text{“elaborate range”} \\
\langle \forall p, q : p \neq q \land (p, q \in X \lor p, q \in Y \lor (p \in X \land q \in Y) \lor (p \in Y \land q \in X)) : \neg E(p, q) \rangle \\
\equiv \text{“without loss of generality, symmetry of } E \text{”} \\
\langle \forall p, q : p \neq q \land (p, q \in X \lor p, q \in Y \lor (p \in X \land q \in Y)) : \neg E(p, q) \rangle \\
\equiv \text{“split range”} \\
\langle \forall p, q : p \neq q \land p, q \in X : \neg E(p, q) \rangle \\
\land \langle \forall p, q : p \neq q \land p, q \in Y : \neg E(p, q) \rangle \\
\land \langle \forall p, q : p \neq q \land p \in X \land q \in Y : \neg E(p, q) \rangle \\
\equiv \text{“definition of Inequiv”} \\
\text{Inequiv}(X) \land \text{Inequiv}(Y) \land \langle \forall p, q : p \neq q \land p \in X \land q \in Y : \neg E(p, q) \rangle \\
\equiv \text{“definition of Pairwise}_\text{inequiv”} \\
\text{Inequiv}(X) \land \text{Inequiv}(Y) \land \text{Pairwise}_\text{inequiv}(X, Y)
\]
2.5. MINIMALITY OF AUTOMATA

2.5.1 Computing $E$

Relation $E$ can be computed using one of several known algorithms — see [Wat95, WD03, Wat01c]. While those algorithms have widely varying worst-case running times, the best known (in terms of asymptotic worst-case running time) algorithm is that of Hopcroft [Hop71, Gri73], with time $O(|Q| \log |Q|)$.

**Conjecture 2.84** The general minimization algorithms (such as Hopcroft’s, etc.) have linear worst-case running time on acyclic automata.

2.5.2 Computing $\neg E$ in ADFAs

Instead of computing (and using) $E$, we can compute its complement $\neg E$ — known as the *distinguishability* relation — directly from $Succ$. Relation $Succ$ is easily computed from transition function $\delta$. Thanks to Corollary 2.74, we can then use $Succ$ as a starting point to compute $\neg E$. We do not elaborate the algorithm here — see [Wat95, §7.3.2, 7.4.1–7.4.3] for more.

2.5.3 Computing $E(p, q)$ pointwise

In [Wat95, Wat01c, WD03], the recursive form of $E$ (Property 2.71) is transformed into a program which computes it pointwise (i.e. for each pair of states). That program avoids endless recursion in cyclic DFAs by extra book-keeping. In an ADFA, such endless recursion is not an issue and the definition can be used directly in the following program (in which we have omitted the invariant):

**Algorithm 2.1:**

```plaintext
func eq(in p, q : Q) : B →
  var e : B
  e := (p ∈ F ≡ q ∈ F) ∧ Σ_p = Σ_q;
  var a : Σ
  for a : a ∈ Σ_p ∩ Σ_q →
    as e → e := eq(δ(p, a), δ(q, a)) sa
  rof
  { e ≡ E(p, q) }
return e
```

This algorithm has exponential worst-case running time — see [Wat95, §15.4] for a pathological example. *Memoization* (caching previously computed results) improves this significantly as is shown in [WD03].

2.5.4 A more efficient computation of $E$

The equivalence of two states can be determined more cheaply if we make a simple restriction on the states being tested (for equivalence).
Property 2.85 Given two states \( p, q \)

\[
\text{Inequiv}(\text{Succ}([p, q]))
\]

\[
\equiv \quad \text{“definitions of Inequiv and } E \text{”}
\]

\[
\langle \forall \ m, n : m, n \in \text{Succ}([p, q]) \rangle : E(m, n) \equiv (m = n)
\]

\[
\equiv \quad \text{“Notation 2.46 rewritten in Eindhoven-style quantification”}
\]

\[
\langle \forall \ m, n : m, n \in (\cup a : a \in \Sigma_p : \{\delta(p, a)\}) \cup (\cup b : b \in \Sigma_q : \{\delta(q, b)\}) \rangle : E(m, n) \equiv (m = n)
\]

\[
\equiv \quad \text{“combine inner quantifications”}
\]

\[
\langle \forall \ m, n : m, n \in (\cup a, b : a \in \Sigma_p, b \in \Sigma_q : \{\delta(p, a), \delta(q, b)\}) \rangle : E(m, n) \equiv (m = n)
\]

\[
\Rightarrow \quad \text{“restrict range and drop one dummy in inner quantification”}
\]

\[
\langle \forall \ m, n : m, n \in (\cup a : a \in \Sigma_p \cap \Sigma_q : \{\delta(p, a), \delta(q, a)\}) \rangle : E(m, n) \equiv (m = n)
\]

\[
\equiv \quad \text{“one-point rule; change of dummy”}
\]

\[
\langle \forall a : a \in \Sigma_p \cap \Sigma_q : (E(\delta(p, a), \delta(q, a)) \equiv (\delta(p, a) = \delta(q, a))
\]

Property 2.86 Given two states \( p, q \) such that Inequiv(\text{Succ}([p, q])), we have a more easily computed form of \( E(p, q) \):

\[
E(p, q)
\]

\[
\equiv \quad \text{“recursive definition of } E \text{”}
\]

\[
(p \in F \equiv q \in F) \land \Sigma_p = \Sigma_q \land \langle \forall a : a \in \Sigma_p \cap \Sigma_q : E(\delta(p, a), \delta(q, a))\rangle
\]

\[
\equiv \quad \text{“assumption Inequiv(\text{Succ}([p, q])) and Property 2.85”}
\]

\[
(p \in F \equiv q \in F) \land \Sigma_p = \Sigma_q \land \langle \forall a : a \in \Sigma_p \cap \Sigma_q : \delta(p, a) = \delta(q, a)\rangle
\]

This can be evaluated directly with the following function:

Algorithm 2.2:

```haskell
func eq'(in p, q : Q) : \mathbb{B} \rightarrow
{ [ pre Inequiv(Succ([p, q])) ]
| var e : \mathbb{B} |
| e := (p \in F \equiv q \in F) \land \Sigma_p = \Sigma_q; |
| var a : \Sigma |
| for a : a \in \Sigma_p \cap \Sigma_q -> |
| as e \rightarrow e := (\delta(p, a) = \delta(q, a)) sa |
| rof |
| ]; |
| post e \equiv E(p, q) |
return e |
}
cnuf
```

In [WD03], Watson and Daciuk show how \( eq' \) can be implemented to have \( O(|\Sigma|) \) (constant) running-time, typically by using hashing and caching \( \Sigma_p \) for all states \( p \). This is surprising, given that \( eq \) can be worst-case exponential in \( |Q| \).
Chapter 3

A MADFA-construction algorithm skeleton

In this chapter, we present a general algorithm skeleton for constructing a MADFA from a set of words. One of the main contributions of this thesis is that all of the known algorithms for constructing MADFAs are presented using this common algorithm skeleton — it therefore forms the basis of a taxonomy of such algorithms, clearly illustrating where they differ and what they have in common. A less rigorous (and slightly differently structured) taxonomy of MADFA construction algorithms is given in [Wat01e].

The structure of the algorithm we have chosen will add the words one-by-one. In some cases, the order in which the words are added is important — and so we assume some partial order \( \leq \) on the words. As we present the general algorithm, we will leave a number of things undetailed:

1. A structural invariant, \( \text{Struct}(D) \) (for the set of words \( D \) already processed), maintained on the ADFA; that is, \( \text{Struct}(D) \) holds both before and after a word \( w \) is added to the ADFA. Examples of such invariants are: the ADFA has a trie structure, the ADFA is minimal, etc.

2. The body of a procedure, \( \text{add}_\text{word} \), used to add individual words.

3. The partial order \( \leq \) on the words.

4. The body of a cleanup procedure, \( \text{cleanup} \), applied to the ADFA after the words have been added, yielding the desired MADFA.

All of these are, in some sense, meta-parameters of the algorithm.

The algorithm skeleton adds a single word (from \( W \)) at a time to the words accepted by the automaton. With some of the \( \text{add}_\text{word} \) procedures, the intermediate automaton may not yet be minimal, requiring the corresponding \( \text{cleanup} \) procedure to further manipulate the ADFA, finally resulting in a MADFA. For this reason, we actually use the more general type ADFA for the automaton.

In the following algorithm, we maintain a partition of \( W \subset \Sigma^* \) into \( D \) (for ‘done’) and \( T \) (for ‘to-do’) and assume word set \( W \) and ADFA \( M = (Q, \delta, s, F) \) are global variables:

Algorithm 3.1:

---

1As Gerard Zwaan pointed out in personal communication [Zwa01], we could consider other structures — for example where several words are added at once. An alternative ‘divide-and-conquer’ approach would be to recursively halve set \( W \), building a MADFA for each such smaller set, and combining the resulting MADFAs. Although such algorithms may be interesting (and remain as future work), they are not considered in this thesis.

2The order could also be ‘unordered’ for those algorithms in which words may be added in arbitrary orders.
3.1 Specific instantiations

In each of the subsequent chapters, we will co-derive specific versions of: add_word, cleanup, Struct(D) and \( \leq \). The general process is outlined in the following subsections.

3.1.1 Choosing a structural invariant

We begin with a choice of our structural invariant as one of:

1. \( \text{Struct}_T(D) \equiv \text{Is}
\_\text{trie} \wedge \mathcal{L} = D \), leading to nonincremental (i.e. those which first build an
ADFA then minimize afterwards) trie-based algorithms in Chapter 4 on page 27.

2. \( \text{Struct}_N(D) \equiv \mathcal{L} = D \), leading to more nonincremental algorithms in Chapter 5 on page 43.

3. \( \text{Struct}_I(D) \equiv \text{Min} \wedge \mathcal{L} = D \), leading to incrementally minimizing algorithms in Chapter 6 on
page 49.

4. \( \text{Struct}_R(D) \equiv \text{Is}
\_\text{trie} \wedge \mathcal{L} = D^R \), leading to an algorithm related to Brzozowski’s minimization
algorithm. This is presented in Chapter 7 on page 61.

5. \( \text{Struct}_S(D) \equiv \text{Inequiv}(Q - [s \stackrel{\text{lexmax}}{\sim}]) \wedge \text{Confl}
\_\text{free}([s \stackrel{\text{lexmax}}{\sim}]) \wedge \mathcal{L} = D \). This leads to the sorted
algorithm by Daciuk, Mihov, and others, appearing in Chapter 9 and page 67.
6. \( \text{Struct}_D(D) \equiv \text{Inequiv}(DL_{\geq \text{minlen}}) \land \text{Confl\_free}(DL_{\leq \text{minlen}}) \land L = D \). This leads to a new state-depth based algorithm, presented in Chapter 10 on page 75.

7. \( \text{Struct}_W(D) \equiv \text{Inequiv}(\text{Succ}^*(F)) \land \text{Confl\_free}(Q - \text{Succ}^*(F)) \land L = D \). This leads to a simplified version of the semi-incremental algorithm by Watson, given here in Chapter 11 page 83.

See Section 1.4 for the mapping from specific structural invariants to the literature.

### 3.1.2 Function add_word

We now have a specification for add_word (for a given Struct):

\[
\{ \text{Struct}(D) \} \\
\text{add\_word}(w) \\
\{ \text{Struct}(D \cup \{ w \}) \}
\]

For each such body and Struct, we get a precondition. In some cases, the precondition can be established by adding the words in a particular order (a choice of \( \leq \)). For clarity, the versions of add_word will be given names of the form add\_word\_X where \( X \) is the corresponding subscript of Struct.

### 3.1.3 Function cleanup

Given Struct, we also have a specification for cleanup:

\[
\{ \text{Struct}(W) \} \\
\text{cleanup} \\
\{ \text{Min} \land L = W \}
\]

The versions of cleanup will be given names of the form cleanup\_X where \( X \) is the corresponding subscript of Struct.

### 3.2 Commentary

The common algorithm skeleton is a key aspect of the algorithm presentation in this thesis. All of the presently known algorithms have been successfully cast in this framework, and there is every reason to believe that newly discovered algorithms will also fit within this or a similar taxonomy.
Chapter 4

Trie intermediate ADFA

In this chapter, we maintain $M$ as a trie during construction — using structural invariant

$$\text{Struct}_T(D) \equiv \text{Is}_\text{trie} \land \mathcal{L} = D$$

Following the construction of the trie using $\text{add}_\text{word}_T$, procedure $\text{cleanup}_T$ merges equivalent states.

4.1 Procedure $\text{add}_\text{word}_T$

For $\text{add}_\text{word}_T$, we get specification (using shadow variable $L$ to express the postcondition)

\[
\text{proc} \quad \text{add}_\text{word}_T(\text{in} \ w : \Sigma^*) \rightarrow \\
\quad \{ \text{ pre } \text{Is}_\text{trie} \land \mathcal{L} = L \} \\
\quad S_{4.1} \\
\quad \{ \text{ post } \text{Is}_\text{trie} \land \mathcal{L} = L \cup \{w\} \}
\]

To make the final implementation $\text{add}_\text{word}_T$ and $S_{4.1}$ reusable in other chapters, we weaken the precondition $\text{Is}_\text{trie} \land \mathcal{L} = L$ to

$$\text{Confl}_\text{free}([s \xrightarrow{\omega}]) \land \mathcal{L} = L$$

(This is a weakening because no state in a trie is a confluence, so no state in the $[s \xrightarrow{\omega}]$ path is a confluence.) For the same reusability reasons, we will derive $S_{4.1}$ such that it only affects $[s \xrightarrow{\omega}]$. This allows us to weaken the postcondition to express the following: if $\text{Is}_\text{trie}$ holds before $\text{add}_\text{word}_T$ is invoked then $\text{Is}_\text{trie}$ will hold after. This postcondition gives us the inductive property that $\text{Is}_\text{trie}$ holds after each word in $W$ is added when $\text{add}_\text{word}_T$ is used to construct $M$ starting with the empty ADFA. (Note that the empty ADFA is also a trie.)

This gives us the following specification in which we use shadow variable $M'$ to express the postcondition

\[
\text{proc} \quad \text{add}_\text{word}_T(\text{in} \ w : \Sigma^*) \rightarrow \\
\quad \{ \text{ pre } \text{Confl}_\text{free}(\{s \xrightarrow{\omega}\}) \land \mathcal{L} = L \land M = M' \} \\
\quad S_{4.1} \\
\quad \{ \text{ post } (\text{Is}_\text{trie}(M') \Rightarrow \text{Is}_\text{trie}) \land \text{Confl}_\text{free}(\{s \xrightarrow{\omega}\}) \land \mathcal{L} = L \cup \{w\} \}
\]

corp
The simplest way to proceed in refining $S_{4,1}$ is to introduce a new state variable $q$, establish $q = \delta^*(s, w) \land q \neq \bot$ and then make $q$ a final state (so that the ADFA accepts $w$), as in the following example.

**Example 4.1 (Adding a prefix word)** Assume we initially have the following ADFA accepting *herd*:

![Diagram of ADFA accepting *herd*]

We wish to add the word *her*, which is a prefix of *herd*. This results in state 3 becoming a final one, as in:

![Diagram of ADFA with state 3 as a final state]

We therefore have the following procedure

```plaintext
proc add_word_T (in w : Σ*) →
\{ pre Confl_free([s \sim w]) \land \mathcal{L} = L \land M = M' \}
|| var q : STATE
\| S'_{4,1};
\{ q = \delta^*(s, w) \land q \neq \bot \}
F := F \cup \{q\}
||
\{ post (Is_trie(M') ⇒ Is_trie) \land Confl_free([s \sim w]) \land \mathcal{L} = L \cup \{w\} \}
corp
```

We can continue our derivation with $S'_{4,1}$.

### 4.1.1 Adding only prefix words

In this section only, we assume that $w$ is a prefix of a word already accepted by $(Q, \delta, s, F)$ — that is $\delta^*(s, w) \neq \bot$. Clearly, this is an unrealistic assumption — it is rarely applicable — but it forms a good starting point for a simple algorithm. We also introduce two additional variables $l, r : w = lr$ and maintain invariant $q = \delta^*(s, l)$, giving the following for $S'_{4,1}$

```plaintext
;
\{ \delta^*(s, w) \neq \bot \}
|| var l, r : Σ*
| l, r, q := ε, w, s;
\{ invariant: w = lr \land q = \delta^*(s, l) \land q \neq \bot
\land variant: |r| \}
do r \neq \epsilon →
\{ q = \delta^*(s, w) \land q \neq \bot \}
;  
```

Of course, precondition $\delta^*(s, w) \neq \bot$ cannot always be established just by adding the words in a certain order (i.e. by choosing $\leq$) and so we generalize this algorithm in the next section.
4.1.2 Adding a nonprefix word in a trie

In the case $\delta^*(s, w) = \perp$, we begin by finding the longest prefix $l$ of $w$ such that $\delta(s, l) \neq \perp$, then build additional states and transitions if required, as in the following example:

Example 4.2 (Adding a word causing a create) Initially, we have the following ADFA accepting her:

![Diagram of initial ADFA accepting her]

We wish to add the word had. The (longest common) prefix $h$ (of had and her) lies on a path to state 1, at which point we are stuck and new states 4 and 5 must be created, eventually giving:

![Diagram of ADFA after adding had]

To express that 'l is the longest prefix on a path reachable from s,' we use the following (using the invariant $q = \delta^*(s, l)$)

$q \neq \perp \land (r = \varepsilon \text{ cor } \delta(q, \text{head}(r)) = \perp)$

Intuitively, this means that there is a full l-path from the start state s, and that either we have run out of symbols to consider (that is $r = \varepsilon$) or no further transitions are possible and we are stuck in state q.

Instead of our previous refinement of $S_{4,1}'$, we obtain

\[
\begin{align*}
\{ & \text{ var } l, r : \Sigma^* \\
\text{ S}_{4,1}'' ; & \\
\{ q = \delta^*(s, l) \land q \neq \perp \land (r = \varepsilon \text{ cor } \delta(q, \text{head}(r)) = \perp) \} \\
\text{ S}_{4,1}''' & \\
\{ q = \delta^*(s, w) \land q \neq \perp \} \\
\} \\
\end{align*}
\]

Statement $S_{4,1}'''$ simply follows the w-path through M until no further transition is possible, then statement $S_{4,1}'''$ extends M as necessary with new states and transitions. The final procedure is

```plaintext
proc add_word_T\(\text{in } w : \Sigma^*\) \rightarrow \\
{ \text{ pre Confl_free([s \sim w]) } \land L = L \land M = M' \} \\
{ \text{ var } q : \text{STATE} } \\
{ \text{ var } l, r : \Sigma^* } \\
{ l, r, q := \varepsilon, w, s; } \\
{ \text{ invariant: } w = lr \land q = \delta^*(s, l) \land q \neq \perp }
```
This algorithm corresponds closely to most trie-construction algorithms — including that sketched by Fredkin, the inventor of tries [Fre60]. A more complete example follows on page 38 in §4.3.

4.2 Procedure cleanupCOMM

In this section, we consider minimization procedures with specification:

\[
\text{proc } \text{cleanup}_T() \rightarrow \\
\{ \text{pre } \text{Is}_\text{trie} \land \mathcal{L} = \mathcal{L} \} \\
S_{4.2} \\
\{ \text{post } \text{Min} \land \mathcal{L} = \mathcal{L} \}
\]

corp

To derive an implementation which will be usable in subsequent chapters as well, we relax the precondition to $\mathcal{L} = \mathcal{L}$ — by dropping the first conjunct $\text{Is}_\text{trie}$, and allowing $M$ to have confluence states.

We maintain a partition of our states $Q$ into $D$, $T$, where $D$ is a set of pairwise inequivalent states (that is $\text{Inequiv}(D)$) which is not shrinking\(^1\). In each iteration, we choose a set of states $N$ from $T$ (which is shrinking), make $D \cup N$ pairwise inequivalent against $D$, then add $N$ to $D$:

Algorithm 4.1:

\[
\text{proc } \text{cleanup}_T() \rightarrow \\
\{ \text{pre } \mathcal{L} = \mathcal{L} \} \\
\| \text{var } D, T : \text{set of STATE} \\
| D, T := \emptyset, Q;
\]

\(^1\)Note that we do not state that $D$ is growing, since it may in fact remain the same size for many iterations when redundant states are being merged.
4.2. PROCEDURE CLEANUP

\{ invariant: \text{Inequiv}(D) \land \mathcal{L} = \mathcal{L} \}
\text{variant: } |T|

\textbf{do } T \neq \emptyset \to
\begin{align*}
\quad & \text{\begin{var}
\quad & N : \text{set of STATE}
\quad & \text{let } N : N \subseteq T \land N \neq \emptyset;
\quad & T := T - N;
\quad & \{ N \neq \emptyset \} \tag{4.2.1}
\quad & N : S_{4.2};
\quad & \{ \text{Inequiv}(D \cup N) \}
\quad & D := D \cup N
\quad & \{ \text{Inequiv}(D) \}
\end{var}\}
\end{align*}

\textbf{od}

\{ post Min \land \mathcal{L} = \mathcal{L} \}

This gives a specification for statement $S_{4.2}$: establish $\text{Inequiv}(D \cup N)$ while changing only $N$ (and implicitly $M$). Thanks to Property 2.83, we can rewrite $\text{Inequiv}(D \cup N)$ as

\[
\underbrace{\text{Inequiv}(D)}_{\text{in invariant}} \land \underbrace{\text{Inequiv}(N)}_{\text{let in §4.2.1}} \land \underbrace{\text{Pairwise}_-\text{inequiv}(D, N)}_{\text{let in §4.2.2}}
\]

Conjunct $\text{Inequiv}(D)$ is already in the repetition invariant, so we ignore it in refining $S_{4.2}$ as our refined statement cannot change $D$. Of the remaining two conjuncts, we can move one of them into the \textbf{let} statement which selects $N$ in the first place, thereby simplifying $S_{4.2}$. In the following sections, we consider those two possibilities (i.e. which conjunct is moved into the \textbf{let} statement) by focusing on the body of the repetition in Algorithm 4.1.

4.2.1 Selecting $N : N \subseteq T \land N \neq \emptyset \land \text{Inequiv}(N)$

If we place $\text{Inequiv}(N)$ in the \textbf{let} statement condition (that is, we select state set $N$ such that they are already known to be pairwise inequivalent), we obtain the following refinement in Algorithm 4.1:

\begin{align*}
\textbf{proc} & \quad \text{cleanup}_{T, \text{Inequiv}(N)}(\) \to \{ \text{pre } \mathcal{L} = \mathcal{L} \} \tag{4.2.1}
\begin{align*}
& \begin{var}
& \begin{align*}
& D, T : \text{set of STATE}
& | D, T := \emptyset, Q;
& \{ \text{invariant: } \text{Inequiv}(D) \land \mathcal{L} = \mathcal{L} \}
& \text{variant: } |T| \}
& \begin{align*}
& \text{do } T \neq \emptyset \to
& \begin{var}
& \begin{align*}
& \quad & N : \text{set of STATE}
& \quad & \text{let } N : N \subseteq T \land N \neq \emptyset \land \text{Inequiv}(N);
& \quad & T := T - N;
& \quad & \{ N \neq \emptyset \land \text{Inequiv}(N) \}
& \quad & N : S_{4.2.1};
& \quad & \text{Inequiv}(D \cup N)
\end{var}\}
\end{align*}
\end{var}\}
\end{align*}
\end{var}\}
\end{align*}
\end{align*}

\[
\underbrace{\text{Inequiv}(D)}_{\text{invariant}} \land \underbrace{\text{Inequiv}(N)}_{\text{let statement}} \land \underbrace{\text{Pairwise}_-\text{inequiv}(D, N)}_{\text{establish in } S_{4.2.1}}
\]
\[ D := D \cup N \]
\[ \{ \text{Inequiv}(D) \} \]
\[ \]}
\[ \od \]
\[ \]}
\[ \{ \text{post } \text{Min} \land L = L \} \]
\[ \]}
\[ corp \]

The easiest implementation of the \texttt{let} statement is to choose \( N : \text{Inequiv}(N) \) as a single state \( p : p \in T \), in which case \( \text{Inequiv}(\{p\}) \) holds trivially. An alternative is to choose a path of states \([r \sim x] \subseteq T\) (for some state \( r \) and string \( x \)). We consider each of these two possibilities in the next sections.

4.2.1.1 Selecting a single state

When selecting a single state, the resulting statement is:

\[ D := D \cup N \]
\[ \{ \text{Inequiv}(D) \} \]

\[ \text{for clarity, we add a local state variable } p : N = \{p\} \text{ and use it almost everywhere}^2 \text{ in place of } N. \]

\[ D := D \cup N \]
\[ \{ \text{Inequiv}(D) \} \]

\[ \od \]
\[ : \]

\[ \]}
\[ \{ \text{invariant: } \text{Inequiv}(D) \land L = L \}
\[ \text{variant: } |T| \} \]
\[ \text{do } T \neq \emptyset \rightarrow \[ \text{var } N : \text{set of } \text{STATE} \]
\[ | \text{let } N : N \subseteq T \land |N| = 1; \]
\[ T := T - N; \]
\[ \{ N \neq \emptyset \land \text{Inequiv}(N) \} \]
\[ N : S_{4.2.1.1}; \]
\[ \{ \text{Inequiv}(D) \land \text{Inequiv}(N) \land \text{Pairwise}_\text{inequiv}(D, N) \} \]
\[ D := D \cup N \]
\[ \{ \text{Inequiv}(D) \} \]
\[ \]}
\[ \od \]
\[ : \]

\[ \]}
\[ \{ \text{invariant: } \text{Inequiv}(D) \land L = L \}
\[ \text{variant: } |T| \} \]
\[ \text{do } T \neq \emptyset \rightarrow \[ \text{var } N : \text{set of } \text{STATE}; \]
\[ \text{var } p : \text{STATE} \]
\[ | \text{let } N, p : N \subseteq T \land N = \{p\}; \]
\[ T := T - \{p\}; \]
\[ \{ \{p\} \neq \emptyset \land \text{Inequiv}(\{p\}) \} \]
\[ p : S'_{4.2.1.1}; \]
\[ \{ \text{Inequiv}(D) \land \text{Inequiv}(N) \land \text{Pairwise}_\text{inequiv}(D, N) \} \]
\[ D := D \cup N \]
\[ \{ \text{Inequiv}(D) \} \]
\[ \]}
\[ \od \]

\[ ^2 \text{We still use } N \text{ in the postcondition of } S'_{4.2.1.1} \text{ because at that point } N \text{ may be } \emptyset \text{ if } p \text{ is equivalent to a state in } D. \]
To establish \( \text{Pairwise\_inequiv}(D, N) \), statement \( S'_{4.2.1.1} \) looks for a \( q \in D \) equivalent to \( p \); there can be at most one such \( q \) since \( \text{Inequiv}(D) \):

\[
\begin{align*}
\text{invariant: } & \quad \text{Inequiv}(D) \land \mathcal{L} = L \\
\text{variant: } & \quad |T|
\end{align*}
\]

\[
\begin{align*}
do & \quad T \neq \emptyset \\
& \quad \textbf{[} \quad \text{var } N : \text{set of STATE;}
& \quad \text{var } p : \text{STATE}
& \quad \mid \quad \text{\textbf{let} } N, p : N \subseteq T \land N = \{ p \};
& \quad T := T - \{ p \};
& \quad \{ (p) \neq \emptyset \land \text{Inequiv}(\{ p \}) \} \\
& \quad \text{as } (\exists q : q \in D : E(p, q)) \rightarrow
& \quad \text{\textbf{let} } q : q \in D \land E(p, q);
& \quad \text{merge}(p, q);
& \quad N := \emptyset
& \quad \textbf{sa;}
& \quad \{ \text{Inequiv}(D) \land \text{Inequiv}(N) \land \text{Pairwise\_inequiv}(D, N) \}
& \quad D := D \cup N
& \quad \{ \text{Inequiv}(D) \}
\end{align*}
\]

\[
\begin{align*}
od
\end{align*}
\]

The now-superfluous \( N \) can be removed, changing the update of \( D \):

\[
\begin{align*}
\text{invariant: } & \quad \text{Inequiv}(D) \land \mathcal{L} = L \\
\text{variant: } & \quad |T|
\end{align*}
\]

\[
\begin{align*}
do & \quad T \neq \emptyset \\
& \quad \textbf{[} \quad \text{var } p : \text{STATE}
& \quad \mid \quad \text{\textbf{let} } p : p \in T;
& \quad T := T - \{ p \};
& \quad \{ \text{Inequiv}(\{ p \}) \} \\
& \quad \text{if } (\exists q : q \in D : E(p, q)) \rightarrow
& \quad \text{\textbf{let} } q : q \in D \land E(p, q);
& \quad \text{merge}(p, q)
& \quad \mid \quad \neg (\exists q : q \in D : E(p, q)) \rightarrow
& \quad \{ \text{Inequiv}(D \cup \{ p \}) \}
& \quad D := D \cup \{ p \}
& \quad \textbf{fi}
& \quad \{ \text{Inequiv}(D) \}
\end{align*}
\]

\[
\begin{align*}
od
\end{align*}
\]

The guard \( (\exists q : q \in D : E(p, q)) \) can be directly evaluated using function \( \text{eq} \) (see page 21). Our algorithm would be more efficient if we could use \( \text{eq}' \) (page 22). According to Property 2.86, we
can use eq’ iff Inequiv(Succ((p, q))). If we add Succ(p) ⊆ D as a conjunct of the let statement selecting p, we inductively get Succ(D) ⊆ D as an additional invariant conjunct since D monotonically grows and is built-up state-by-state from such p : Succ(p) ⊆ D. Thanks to this new conjunct and our other invariant conjunct, Inequiv(D), we have the required Inequiv(Succ((p, q))) for all q ∈ D. The resulting procedure is (where subscript ss stands for ‘single state’):

\[
\text{proc cleanup}_{T, \text{Inequiv}(N), \text{ss}}() \rightarrow \\
\{ \text{pre } L = L \} \\
\| \text{var } D, T : \text{set of STATE} \\
| D, T := \emptyset, Q; \\
| \{ \text{invariant: Inequiv}(D) \land \text{Succ}(D) \subseteq D \land L = L \} \\
| \text{variant: } |T| \} \\
\text{do } T \neq \emptyset \rightarrow \| \text{var } p : \text{STATE} \\
| \text{let } p : p \in T \land \text{Succ}(p) \subseteq D; \\
| T := T - \{p\}; \\
| \{ \text{Inequiv}((p)) \land \text{Succ}(p) \subseteq D \land \text{Inequiv}((\text{Succ}(p))) \} \\
| \text{if } \langle \exists q : q \in D : \text{eq’}(p, q) \rangle \rightarrow \\
| \text{let } q : q \in D \land \text{eq’}(p, q); \\
| \text{merge}(p, q) \\
| \{ \text{Inequiv}(D \cup \{p\}) \} \\
| D := D \cup \{p\} \} \\
\text{fi} \\
\{ \text{Inequiv}(D) \} \\
\} \\
\text{od} \\
\| \{ \text{post } \text{Min} \land L = L \} \\
\text{corp}
\]

We now consider whether such a state p : Succ(p) ⊆ D can always be selected in the let. From the invariant, we have Inequiv(D) \land \text{Succ}(D) \subseteq D and by the repetition guard T \neq \emptyset. There are two (mutually exclusive) cases:

1. HL₀ \not\subseteq D — there is a leaf state not yet in D, and Succ(HL₀) \subseteq D holds trivially since leaves have no successors, in which case we can select p from HL₀ \setminus D as such a leaf state; or

2. HL₀ \subseteq D — let k > 0 be the smallest k such that HLₖ \not\subseteq D. We can select p from HLₖ since

\[
\text{Succ}(p) \\
\subseteq \begin{cases} 
\text{“Property 2.81”} \\
\langle \cup j : 0 \leq j < k : \text{HL}_j \rangle \\
\text{“assumption that } k \text{ is the smallest such that HL}_k \not\subseteq D \end{cases} \\
D
\]

4.2.1.2 Selecting a path of states

From Corollary 2.79, we can select any path of states for N in the repetition of Algorithm 4.1:
4.2. PROCEDURE CLEANUPₜ

: 
  { invariant: Inequiv(D) ∧ L = L 
    variant: |T| } 
  do T ≠ ∅ → [ [ var N : set of STATE 
      | let N : N ⊆ T ∧ N ≠ ∅ ∧ N = [r ~ x] for some r, x; 
      T := T − N; 
      { N ≠ ∅ ∧ Inequiv(N) } 
      N : S₄₊₂.₁.₂ 
      { Inequiv(D) ∧ Inequiv(N) ∧ Pairwise_inequiv(D, N) } 
      D := D ∪ N 
      { Inequiv(D) } ] ] 
  od 
:

An implementation of $S₄₊₂.₁.₂$ will consider the individual states in $[r ~ x]$, comparing them for equivalence (using eq) against states in $D$; those found to be equivalent will be merged, while the inequivalent ones are added to $D$. The details of such an implementation resemble those in the previous section and are not discussed further here. When $x = ε$, we have the degenerate path $[r ~ ε] = [r ~ ε] = r$, yielding the single-state algorithm of §4.2.1.1.

Interestingly, we could have chosen any sequence of states $p₀, \ldots, p_j : (∀ i : 0 ≤ i ≤ j : p_{l+1} ∈ \text{Succ}^+(p_i))$ which form a reachability chain. (Note that they need not be immediate successors.) Path $[r ~ x]$ is just one such reachability chain.

4.2.2 Selecting $N : N ⊆ T ∧ N ≠ ∅ ∧ Pairwise_inequiv(D, N)$

We return to the repetition of Algorithm 4.1 (page 30), instead placing Pairwise_inequiv(D, N) in the let statement — choose $N$ such that each state therein is inequivalent to the states $D$ already processed (though Inequiv(N) is still to be established in statement $S₄₊₂.₂$). This gives the following:

```
proc cleanupₜ,Pairwise_inequiv(D,N)() → 
  { pre L = L } 
  [ [ var D,T : set of STATE 
      | D, T := ∅, Q; 
      { invariant: Inequiv(D) ∧ L = L 
        variant: |T| } 
      do T ≠ ∅ → [ [ var N : set of STATE 
        | let N : N ⊆ T ∧ N ≠ ∅ ∧ Pairwise_inequiv(D, N); 
        T := T − N; 
        { N ≠ ∅ ∧ Pairwise_inequiv(D, N) } 
        N : S₄₊₂.₂; 
        { Inequiv(D ∪ N) } ] ] 
      ] ] 
```

We focus on selecting $N$ before returning to an implementation of $S_{4.2.2}$. From Property 2.80 (page 20), we know that (for $k \geq 0$) Pairwise_inequiv($H_{L_{k+1}}$, $H_{L_k}$). It follows that one way to select $N$ is to proceed in ascending levels (i.e. starting with the leaves $H_{L_0}$) within the ADFA. For this, we add variable $k$ and select $N = H_{L_k}$. Variable $D$ accumulates the states from all visited levels, that is $D = \langle \bigcup_{j: 0 \leq j < k} H_{L_j}\rangle$, while $T = \langle \bigcup_{j: j \geq k} H_{L_j}\rangle$. Clearly, we then have Pairwise_inequiv($D, N$), which we express as Pairwise_inequiv($D, H_{L_k}$).

From Property 2.81, we have

\[ \text{Succ}(H_{L_k}) \subseteq \langle \bigcup_{j: 0 \leq j < k} H_{L_j}\rangle \]

That is, \text{Succ}(H_{L_k}) \subseteq D. As in §4.2.1, we introduce invariant conjunct \text{Succ}(D) \subseteq D. (This is thanks to the monotonic growth of $D$ and because it is built up from height levels where \text{Succ}(H_{L_k}) \subseteq D.) With Inequiv($D$) from our invariant, we also have Inequiv(\text{Succ}(H_{L_k})). Since $N$ is now redundant, we can eliminate it in favour of $H_{L_k}$ everywhere\(^3\).

\[^3\text{Here, we assume we have some way of implementing } H_{L_k} \text{ easily and cheaply.}\]
4.2. PROCEDURE CLEANUP

\[ D = \langle \cup j : 0 \leq j < k : H_L j \rangle \]
\[ T = \langle \cup j : j \geq k : H_L j \rangle \]
\[ \text{Pairwise}_\text{inequiv}(D, H_L k) \]
\[ \text{Succ}(H_L k) \subseteq D \]
\[ \text{Succ}(D) \subseteq D \]
\[ \text{Inequiv}(\text{Succ}(H_L k)) \]

variant: \(|T| \}

\text{do} \ T \neq \emptyset \rightarrow
  \text{T := T} - H_L k;
  \{ \text{HL}_k \neq \emptyset \}
  H_L k : S''_{4,2,2};
  \{ \text{Inequiv}(D) \wedge \text{Inequiv}(H_L k) \wedge \text{Pairwise}_\text{inequiv}(D, H_L k) \}
  \text{invariant}
  D := D \cup H_L k;
  \{ \text{Inequiv}(D) \}
  k := k + 1
\text{od}

With our invariant conjunct \( T = \langle \cup j : j \geq k : H_L j \rangle \) and Property 2.56, we can change our repetition guard from \( T \neq \emptyset \) to \( H_L k \neq \emptyset \).

In \( S''_{4,2,2} \), we will iterate over \( p, q \in H_L k \), considering them for equivalence to each other. Since \( \text{Inequiv}(\text{Succ}(H_L k)) \) holds, we can directly use the simpler form for \( E(p, q) \) (see Property 2.86) and therefore eq'. Recall from Property 2.55 that merging states does not change their height levels. \( S''_{4,2,2} \) is now:

\[ \vdots \]
\[ \{ \text{HL}_k \neq \emptyset \} \]
\[ \{ \text{HL}_k \neq \emptyset \} \]
\[ \{ \text{HL}_k \neq \emptyset \} \]
\[ \{ \text{HL}_k \neq \emptyset \} \]
\[ \{ \text{HL}_k \neq \emptyset \} \]
\[ \text{as eq'}(p, q) \rightarrow \text{merge}(p, q) \text{ sa rof} \]
\[ \{ \text{Inequiv}(D) \wedge \text{Inequiv}(H_L k) \wedge \text{Pairwise}_\text{inequiv}(D, H_L k) \} \]
\[ \text{invariant} \]
\[ \text{invariant} \]
\[ \text{invariant} \]
\[ \text{invariant} \]
\[ \text{invariant} \]
\[ \text{invariant} \]
\[ \text{invariant} \]

The resulting algorithm, closely related to the one first presented by Revuz in [Rev92], is

**Algorithm 4.2 (Revuz-like):**

\text{proc} \ cleanup_{T, \text{Pairwise}_\text{inequiv}(D, H_L k)}() \rightarrow
\{ \text{pre } L = L \}
\[ \{ \text{var } D, T : \text{set of } \text{STATE}; k : \mathbb{N} \}
\{ D, T := \emptyset, Q; \]
\{ k := 0; \}
\{ \text{invariant: Inequiv}(D) \wedge L = L \]
\[ \wedge D = \langle \cup j : 0 \leq j < k : H_L j \rangle \]
\[ T = \{ j : j \geq k : HL_j \} \]
\[ \text{Pairwise}_\text{inequiv}(D, HL_k) \]
\[ \text{Succ}(HL_k) \subseteq D \]
\[ \text{Inequiv}(\text{Succ}(HL_k)) \]
\[ \text{Succ}(D) \subseteq D \]

variant: \(|T| \}
\[ \text{do} \quad \text{HL}_k \neq \emptyset \rightarrow \]
\[ T := T - \text{HL}_k; \]
\[ \{ \text{HL}_k \neq \emptyset \} \]
\[ \{ \quad \text{var} \ p, q : \text{STATE} \]
\[ \quad \text{for} \ p, q : p, q \in \text{HL}_k \rightarrow \]
\[ \quad \quad \text{as } \text{eq}'(p, q) \rightarrow \text{merge}(p, q) \text{ sa} \]
\[ \quad \text{rof} \]
\[ \} \]
\[ \text{D} := D \cup \text{HL}_k; \]
\[ \{ \text{Inequiv}(D) \} \]
\[ k := k + 1 \]
\[ \text{od} \]
\[ \{ \quad \text{post} \ \text{Min} \wedge \mathcal{L} = L \} \]

In [Rev92], Revuz shows that \text{eq}' can be implemented using some clever coding tricks. Graña et al [GBA01] and Bubenzer [Bub11] make further improvements to this algorithm.

### 4.3 An example

In this section, we present an extended example. After adding \text{had}, \text{hard}, \text{head}, \text{heard}, \text{herd}, \text{here}, \text{her}, \text{he} using \text{add\_word}_T', the resulting trie is
4.3. AN EXAMPLE

Now, consider Revuz’s algorithm applied (as cleanup\textsubscript{T,Pairwise_inequiv}\((D,N)\)) to the trie given above. Initially, we consider \(HL_0 = \{3, 5, 8, 10, 12, 13\}\), all of which can be merged, yielding
We now examine the states in $HL_1 = \{4, 9, 11\}$ against themselves, and 9 is merged into 4 (note that merging could have occurred vice-versa, with 4 merged into 9); state 11 is not merged since it is final and the other two are not. The resulting automaton is

Considering $HL_2 = \{2, 7\}$, we find that the two states are equivalent and they are merged, giving

After considering $HL_3 = \{6\}$, $HL_4 = \{1\}$, $HL_5 = \{0\}$, the above ADFA is minimal.
4.4 Time and space performance

For word $w$, invocation $\text{add}_\text{word}_T(w)$ adds $O(|w|)$ states in $O(|w|)$ time if we assume that $\delta(p, a)$ and $\text{create}$ take constant time. It follows that building $M$ from word set $W$ yields a trie of $O(\sum_{w\in W}|w|)$ states in the same order of time. Revuz’s version of $\text{cleanup}_T$ can be shown to take time $O(|M|)$ and can be implemented to take space corresponding to the longest word in $W$. It follows that the construction and minimization of $M$ takes $O(\sum_{w\in W}|w|)$ time and space — a surprising result given that minimization of an arbitrary DFA $Z$ takes time $O(|Z|\log|Z|)$.

4.4.1 Improvements

There are numerous improvements which can be made to the implementation of $\text{eq}'$ and all versions of $\text{cleanup}_T$, some of which are discussed in [Rev92, Dac98, GBA01, DMWW00, Bub11]:

- The states to be considered are sorted according to their level and whether or not they are final.
- The first iteration always merges all leaf states. This can be factored out to a separate statement before the iteration.
- The start state $s$ is never equivalent to any other state, and need not be considered at all.
- In the implementation selecting $N$ as a path (§4.2.1.2), there may be efficient orders of considering the states in $N$ which are as yet unknown, for example, from the head to the tail of the path.

Significant further improvements were made by Bubenzer and reported in [Bub11].

4.5 Commentary

The algorithms presented in this chapter are the most rudimentary ones. As a result, they are rarely used in industrial-strength applications, though the implementations of $\text{cleanup}_T$ are interesting in their own right for minimizing arbitrary ADFAs which have already been constructed.
Chapter 5

Arbitrary intermediate ADFA

In this chapter, we consider adding a word to an arbitrary ADFA — one in which confluences may be encountered (when adding word \( w \)) on the \( w \)-path — though we will make use of the fact that \( s \) (the start state) cannot be a confluence due to acyclicity\(^1\). The structural predicate is simply

\[
\text{Struct}_N(D) \equiv \mathcal{L} = D
\]

Following construction, procedure \( \text{cleanup}_N \) merges states in the same way as \( \text{cleanup}_T \) does in Chapter 4.

5.1 Procedure \( \text{add\_word}_N \)

Without modification, the algorithms of Chapter 4 (\( \text{add\_word}_T \) and variants) may add words accidentally if a confluence state is encountered. Consider the following example.

Example 5.1 (Adding words accidentally) Initially, we have the following ADFA accepting hard and herd (without useless states, at least two words are necessary to have a confluence state):

While adding the new word head, we arrive at confluence state 2. From state 2, there is no outgoing transition and we naively extend the automaton, yielding:

\(^1\)Strictly, it is possible for \( s \) to be a confluence state if useless states are introduced, though we have excluded that case in this thesis.
This ADFA incorrectly also accepts haad.

Clearly, a ‘cloning’ operation is required (see page 10 for the definition of cloning), as we see in the following corrected example.

**Example 5.2 (Adding a word causing a clone)** As in the previous example, we begin with the following MADFA accepting hard and herd:

![Diagram of MADFA accepting hard and herd]

While adding the new word head, we arrive at confluence state 2 which is cloned, yielding new state 5:

![Diagram showing confluence state 2 cloned to state 5]

Two additional states are then added — giving the final automaton:

![Diagram showing final automaton]

When modifying function \( \text{add}_r \) (from Chapter 4 on page 27), the algorithm needs to clone confluence states, and confluence states can only be encountered in the first repetition since the second repetition is only creating new states — none of which can be a confluence state. We can modify the first repetition accordingly yielding the procedure body:

```plaintext
proc add_word_N (in w : \( \Sigma^* \)) →
{ pre \( \mathcal{L} = \emptyset \)
∥ var q : STATE
∥∥ var l, r : \( \Sigma^* \); p : STATE
∥ l, r, q := \( \varepsilon, w, s \);
{ invariant: w = lr \land q = \delta^*(s, l) \land q \neq \bot \land \text{Confl\_free}([s \sim])
```
5.1. *PROCEDURE ADD_WORD*$_N$

variant: |r|

**do** r $\neq \varepsilon$ **cand** $\delta(q, \text{head}(r)) \neq \bot \rightarrow$
p := $\delta(q, \text{head}(r));$
**as** ls_confl(p) →
p := clone(p);
$\delta(q, \text{head}(r)) := p$

sa;
q := p;
l, r := l \cdot \text{head}(r), \text{tail}(r)
**od**;
{ Confl_free([s \sim w]) }
{ q = \delta^*(s, l) \land q \neq \bot \land (r = \varepsilon \textbf{ cor } \delta(q, \text{head}(r)) = \bot) }
{ invariant: w = lr \land q = \delta^*(s, l) \land q \neq \bot }
variant: |r|

**do** r $\neq \varepsilon \rightarrow \llbracket \textbf{ var } p : \text{STATE} \rrbracket$
p := create();
$\delta(q, \text{head}(r)), q := p, p;$
l, r := l \cdot \text{head}(r), \text{tail}(r)

\]

\]
{ q = \delta^*(s, w) \land q \neq \bot }
F := F \cup \{q\}
\]
{ post Confl_free([s \sim w]) \land L = L \cup \{w\} }

\]

corp

This, however, is also subject to improvement thanks to another observation:

Once we encounter a confluence state on the w path and clone it, all subsequent states on the path (other than newly created ones) will also be confluences and will have to be cloned.

To see why this holds, consider confluence state p on the w-path: state clone(p) has out-transitions with the same labels and destinations as p, making each of p’s successors also a confluence with in-transitions from at least p and clone(p).

For this reason, we can again split the first of the above repetitions into two sequentially composed repetitions, in our final algorithm:

\[
\text{proc } \text{add_word}_N(\text{in } w : \Sigma^*) \rightarrow
\{ \text{ pre } \mathcal{L} = L \} \\
\llbracket \textbf{ var } l, r : \Sigma^*; p : \text{STATE} \rrbracket \\
\llbracket \textbf{ var } l, r, q := \varepsilon, w, s; \rrbracket
\{ \text{ invariant: w = lr \land q = \delta^*(s, l) \land q \neq \bot \land Confl_free([s \sim l]) }
\text{ variant: |r| } \}
\textbf{ do } r \neq \varepsilon \textbf{ cand } \delta(q, \text{head}(r)) \neq \bot \textbf{ cand } \neg ls_confl(\delta(q, \text{head}(r))) \rightarrow 
\text{ l, r, q := l \cdot \text{head}(r), \text{tail}(r), \delta(q, \text{head}(r)) }
\textbf{ od};
\]


{ invariant: \(w = lr \land q = \delta^*(s, l) \land q \neq \bot \land \text{Confl\_free}([s \sim])\) 
variant: \(|r|\) 

do \ r \neq \epsilon \ \text{cand} \ \delta(q, \text{head}(r)) \neq \bot \rightarrow 
\{ \ \text{ls\_confi}(\delta(q, \text{head}(r))) \ \} 
\ p := \delta(q, \text{head}(r)); 
\{ \ \text{ls\_confi}(p) \ \} 
\ p := \text{clone}(p); 
\delta(q, \text{head}(r)), q := p, p; 
l, r := l \cdot \text{head}(r), \text{tail}(r) 
\text{od}; 
\{ \ \text{Confl\_free}([s \sim]) \ \} 
\{ \ q = \delta^*(s, l) \land q \neq \bot \land (r = \epsilon \ \text{cor} \ \delta(q, \text{head}(r)) = \bot) \ \} 
\{ \ invariant: \ w = lr \land q = \delta^*(s, l) \land q \neq \bot \ 
variant: \ |r| \ \} 

do \ r \neq \epsilon \rightarrow p := \text{create}(); 
\delta(q, \text{head}(r)), q := p, p; 
l, r := l \cdot \text{head}(r), \text{tail}(r) 
\text{od} 
\} 
\ | \ 
\{ \ q = \delta^*(s, w) \land q \neq \bot \ \} 
F := F \cup \{ q \} 
\} 
\{ \ post \ \text{Confl\_free}([s \sim]) \land L = L \cup \{ w \} \ \} 

corp

Implementation 5.3 This algorithm always clones confluences, which proves to be inefficient if they are subsequently found to be equivalent (and therefore merged). High-performance implementations of this algorithm perform a ‘lazy cloning’ (also known as ‘virtual cloning’) operation, substantially improving the performance [DMWW00].

5.2 Procedure cleanup\(_\text{N}\) 

As in Chapter 4, for cleanup\(_\text{N}\) we can use any one of the general minimization algorithms from [Wat95] or a version of cleanup\(_\text{T}\) from §4.2.

5.3 Time and space performance 

For word \(w\), invocation add\(_\text{word}_{\text{N}}(w)\) adds \(O(|w|)\) states in \(O(|w|)\) time. It follows that building \(M\) from word set \(W\) yields an ADFA in \(O(\sum_{w \in W} |w|)\) time and space. The most efficient implementations of cleanup\(_\text{T}\) in §4.2 take \(O(|M|)\) time and space. It follows that the construction and minimization of \(M\) takes \(O(\sum_{w \in W} |w|)\) time and space.

5.4 Commentary 

If the MADFA is built from scratch, add\(_\text{word}_{\text{N}}\) is uninteresting since the initial ADFA will be a trie in which no confluences occur. Procedure add\(_\text{word}_{\text{N}}\) is primarily interesting for adding words to
an ADFA in which some confluences already occur from previous minimization steps; \( add_{\text{word}_N} \) will also be used in Chapter 6 to derive an incremental algorithm. Interestingly, \( add_{\text{word}_N} \) works on cyclic DFA's.
Chapter 6
Minimal intermediate ADFA

This chapter presents an incremental algorithm. We maintain structural invariant

\[ \text{Struct}_i(D) \equiv \text{Min} \wedge L = D \]

Given that \( M \) is already minimal in the invariant (while adding words), \( \text{cleanup}_i \) is reduced to a \text{skip} statement.

6.1 Procedure \text{add\_word}_i

Our starting point for \text{add\_word}_i is

\begin{verbatim}
proc add_word_i( in \ w : \Sigma^* ) \to
{ pre Min \wedge L = L } 
S_{6.1} 
{ post Min \wedge L = L \cup \{w\} }
endproc
\end{verbatim}

In \( S_{6.1} \), we will first use \text{add\_word}_{N'} (from page 45) to add \( w \), since \( M \) is initially minimal (that is, \( \text{Inequiv}(Q) \)) and we may encounter confluences on the \( w \)-path. After \( w \) has been added, we consider those states whose right languages have changed and may now be equivalent to another — merging them and thereby restoring minimality.

Looking at Example 5.2 (page 44), it is easy to see that the only states with changed right languages are precisely those on the \( w \)-path, namely \( [s \sim^w] \), and the remaining states will still be inequivalent. In other words, after \( w \) has been added, we have

\[ \text{Inequiv}(Q - [s \sim^w]) \]

Actually, thanks to Property 2.70 we know that \( s \) is inequivalent to all other states, and we could have written \( \text{Inequiv}(Q - [s \sim^w]) \), which leads to a slightly more efficient algorithm; for simplicity, we initially ignore that in this chapter. We also have some important properties of the states on path \([s \sim^w]\):

1. \( \text{Inequiv}([s \sim^w]) \) — that is, no state on the path is equivalent to any other state on the path\(^1\).
   This is given in Corollary 2.79.

\(^1\)States on \([s \sim^w]\) may, however, be equivalent to others in \( M \).
2. Confl\_free([s ~\rightleftharpoons s]) — that is, no state on the path is a confluence. This follows from the postcondition of add\_word_N′ (see page 45).

3. No state in Q − [s ~\rightleftharpoons s] has a transition to a state in [s ~\rightleftharpoons s]. This follows from the second property above and Property 2.48.

Our new algorithm (in which we have strengthened post-condition of add\_word_N′ based on the discussion above) is

\[
\text{proc add\_word}_I(\text{in } w: \Sigma^*) \rightarrow
\{ \text{pre } \text{Min} \land L = L \}
add\_word_N′(w);
\{ \text{Confl\_free([s ~\rightleftharpoons s])} \land L = L \cup \{w\} \}
\text{normal postcondition of add\_word_N′}
\{ \text{inequiv}(Q − [s ~\rightleftharpoons s]) \land \text{inequiv}([s ~\rightleftharpoons s]) \}
\text{discussion above}
S'_{6,1}
\{ \text{Min} \}
\{ \text{post } \text{Min} \land L = L \cup \{w\} \}
\]

corp

Given the precondition of S'_{6,1}

\text{inequiv}(Q − [s ~\rightleftharpoons s]) \land \text{inequiv}([s ~\rightleftharpoons s])

and Property 2.83, we need only establish

Pairwise\_inequiv(Q − [s ~\rightleftharpoons s], [s ~\rightleftharpoons s])

to equivalently have

\text{inequiv}((Q − [s ~\rightleftharpoons s]) \cup [s ~\rightleftharpoons s])

≡ \text{inequiv}(Q) \equiv \text{Min}. We can traverse the states [s ~\rightleftharpoons s] in several orders, among which from s to \delta^*(s, w) (‘top-down’) or vice-versa (‘bottom-up’); we will shortly see that it makes sense to consider them bottom-up. This is easily done with a recursive procedure (visit\_min to be derived in §6.1.1 starting on page 51), traversing them in post-order and merging states found to be equivalent. An invocation visit\_min(1, r) processes states [\delta^*(s, l) ~\rightleftharpoons s]. (Note the inclusion of the first and last states.) Both l and r are passed recursively for proper bookkeeping and to express our invariant. The specification (where the pre- and postcondition correspond to the context of S'_{6,1}) is:

\[
\text{proc visit\_min}(\text{in } l, r: \Sigma^*) \rightarrow
\{ \text{pre } \text{Confl\_free([s ~\rightleftharpoons s])} \land \text{inequiv}(Q − [s ~\rightleftharpoons s]) \land \delta^*(s, l) \neq \bot \land L = L \}
S'_{6,1}
\{ \text{post } \text{inequiv}(Q − [s ~\rightleftharpoons s]) \land L = L \}
\]

corp

Clearly, invocation visit\_min(\epsilon, w) satisfies the specification of S'_{6,1} since, in the postcondition

\[
Q − [s ~\rightleftharpoons \epsilon) = Q − \emptyset = Q
\]

and we have our final version of add\_word_I.
6.1. PROCEDURE ADD\_WORD\_1

---

**proc** add\_word\_1(in w : \(\Sigma^*\)) →

\{ **pre** Min \(\land\) \(\mathcal{L}\) = L \}  
add\_*L'*(w);

\{ Confl\_*L'*(\(s \downarrow\)) \(\land\) \(\mathcal{L}\) = L \(\cup\) \{ w \}
\(\land\) Inequiv(Q − \(s \downarrow\)) \(\land\) Inequiv([\(s \downarrow\)])

visit\_min(\(\varepsilon, W\))

\{ **post** Min \(\land\) \(\mathcal{L}\) = L \(\cup\) \{ w \} \}

---

### 6.1.1 Recursive helper procedure visit\_min

We can now focus on deriving an implementation for S\_*6.1.1* in visit\_min. We can rewrite the first conjunct of the visit\_min postcondition:

\[
\text{Inequiv}(Q - [s \downarrow])
\equiv
\text{“definition of open ended range [. . .] and set calculus”}
\]

\[
\text{Inequiv}((Q - [s \downarrow]) \cup \{\delta^*(s, l)\})
\equiv
\text{“Property 2.83”}
\]

\[
\text{Inequiv}(Q - [s \downarrow]) \land \text{Inequiv}([\delta^*(s, l)]) \land \text{Pairwise\_inequiv}(Q - [s \downarrow], \{\delta^*(s, l)\})
\equiv
\text{“Inequiv always holds on a single state, \(\delta^*(s, l)\) in this case”}
\]

\[
\text{Inequiv}(Q - [s \downarrow]) \land \text{Pairwise\_inequiv}(Q - [s \downarrow], \{\delta^*(s, l)\})
\]

(Note the similarity of this derivation to the reasoning surrounding the specification of visit\_min on page 50.) We establish each of these last two conjuncts separately:

---

**proc** visit\_min(in l, r : \(\Sigma^*\)) →

\{ **pre** Confl\_*L'*(\(s \downarrow\)) \(\land\) Inequiv(Q − \(s \downarrow\)) \(\land\) \(\delta^*(s, lr)\) \(\neq\) \(\bot\) \(\land\) \(\mathcal{L}\) = L \}

\(S'_{6.1.1}\);

\{ Confl\_*L'*(\(s \downarrow\)) \(\land\) Inequiv(Q − \(s \downarrow\)) \(\land\) \(\mathcal{L}\) = L \}

\(S'_{6.1.1}\)

\{ Confl\_*L'*(\(s \downarrow\)) \(\land\) Inequiv(Q − \(s \downarrow\)) \(\land\) \(\mathcal{L}\) = L \}

\(\equiv\) Inequiv(Q − \(s \downarrow\)) from derivation above

\(\land\) \(\mathcal{L}\) = L \}

\{ **post** Inequiv(Q − \(s \downarrow\)) \(\land\) \(\mathcal{L}\) = L \}

---

Consider \(S'_{6.1.1}\) : its postcondition’s first conjunct already holds when \(r = \varepsilon\), and no deeper recursion is required in this case, making this our recursion termination condition. This leads to the following refinement:

**proc** visit\_min(in l, r : \(\Sigma^*\)) →

\{ **pre** Confl\_*L'*(\(s \downarrow\)) \(\land\) Inequiv(Q − \(s \downarrow\)) \(\land\) \(\delta^*(s, lr)\) \(\neq\) \(\bot\) \(\land\) \(\mathcal{L}\) = L \}
as \(r \neq \varepsilon \rightarrow S''_{6.1.1}\) \(sa;\)

\{ Confl\_*L'*(\(s \downarrow\)) \(\land\) Inequiv(Q − \(s \downarrow\)) \(\land\) \(\mathcal{L}\) = L \}

\(S''_{6.1.1}\)
We can rewrite the postcondition conjunct of $S_{6.1.1}''$:

\[
\text{Inequiv}(Q - [s \mapsto l]) \land L = L
\]

This last line is established with a recursive invocation

\[
\text{visit}\_\text{min}(l \cdot \text{head}(r), \text{tail}(r))
\]

giving

\[
\textbf{proc} \text{ visit}\_\text{min}(\text{in } l, r : \Sigma^*) \rightarrow \\
\{ \textbf{pre} \text{ Confl}_\text{free}([s \mapsto l]) \land \text{Inequiv}(Q - [s \mapsto l]) \land \delta^*(s, l) \neq \bot \land L = L \} \\
\textbf{as} r \neq \varepsilon \rightarrow \text{visit}\_\text{min}(l \cdot \text{head}(r), \text{tail}(r)) \\
\{ \text{ Inequiv}(Q - [s \mapsto l]) \land L = L \} \\
\}
\]

We can now turn to $S_{6.1.1}'''$:

\[
S_{6.1.1}'''
\]

\[
\{ \text{ Confl}_\text{free}([s \mapsto l]) \land \text{Inequiv}(Q - [s \mapsto l]) \land L = L \} \\
\}
\]

An implementation of $S_{6.1.1}'''$ must check state $\delta^*(s, l)$ for equivalence against states in $Q - [s \mapsto l]$, eliminating it if an equivalent one is found.

Indeed, it suffices to check $\delta^*(s, l)$ for equivalence against $Q - [s \mapsto l]$ (all states except $[s \mapsto l]$) because $\delta^*(s, l)$ lies on $[s \mapsto l]$, and $\text{Inequiv}([s \mapsto l])$ thanks to Corollary 2.79. The resulting implementation of visit\_\text{min} is now (where we introduce local variable $p = \delta^*(s, l)$ and use function eq to check state equivalence):

\[
\textbf{proc} \text{ visit}\_\text{min}(\text{in } l, r : \Sigma^*) \rightarrow \\
\{ \textbf{pre} \text{ Confl}_\text{free}([s \mapsto l]) \land \text{Inequiv}(Q - [s \mapsto l]) \land \delta^*(s, l) \neq \bot \land L = L \} \\
\]
as \( r \neq \varepsilon \rightarrow \text{visit}_{\min}(l \cdot \text{head}(r), \text{tail}(r)) \)
\[
\{ \text{Inequiv}(Q - [s^{\text{head}(r)}]) \land \mathcal{L} = L \} \]
sa;
\[
\{ \text{Confl}_\text{free}([s \overset{\lambda}{\rightarrow}]) \land \text{Inequiv}(Q - [s \overset{\lambda}{\rightarrow}]) \land \mathcal{L} = L \} \]

\[\| \text{var } p, q : \text{STATE} \]
\[
\{ \text{Confl}_\text{free}([s \overset{\lambda}{\rightarrow}]) \land \text{Inequiv}(Q - [s \overset{\lambda}{\rightarrow}]) \land \mathcal{L} = L \} \]

\[
\{ \text{post} \text{Inequiv}(Q - [s \overset{\lambda}{\rightarrow}]) \land \mathcal{L} = L \}\]
corp

For efficiency reasons, we would prefer eq' over eq — see §2.5.1. We start with Confl\_free([s \overset{\lambda}{\rightarrow}]) \land \text{Inequiv}(Q - [s \overset{\lambda}{\rightarrow}]), which holds before the second as-sa statement:

\[
\text{Confl}_\text{free}([s \overset{\lambda}{\rightarrow}]) \land \text{Inequiv}(Q - [s \overset{\lambda}{\rightarrow}]) \]
\[
\Rightarrow \text{“Property 2.48 and first conjunct”} \]
\[
\text{Succ}(Q - [s \overset{\lambda}{\rightarrow}]) \subseteq Q - [s \overset{\lambda}{\rightarrow}] \land \text{Inequiv}(Q - [s \overset{\lambda}{\rightarrow}]) \]
\[
\Rightarrow \text{“second conjunct and set containment in the first conjunct”} \]
\[
\text{Inequiv(\text{Succ}(Q - [s \overset{\lambda}{\rightarrow}]))} \]
\[
\equiv \text{“calculus of sets and state paths”} \]
\[
\text{Inequiv(\text{Succ}((Q - [s \overset{\lambda}{\rightarrow}] \cup \{\delta^*(s, l)\})))} \]

This last line, with Property 2.86, implies that eq’ can be used to evaluate the guard in:

\[
\text{proc visit}_{\min}(\text{in } l, r : \Sigma^*) \rightarrow \]
\[
\{ \text{pre} \text{Confl}_\text{free}([s \overset{\lambda}{\rightarrow}]) \land \text{Inequiv}(Q - [s \overset{\lambda}{\rightarrow}]) \land \delta^*(s, l) \neq \bot \land \mathcal{L} = L \} \]
as \( r \neq \varepsilon \rightarrow \text{visit}\_\min(l \cdot \text{head}(r), \text{tail}(r)) \)
\[
\{ \text{Inequiv}(Q - [s^{\text{head}(r)}]) \land \mathcal{L} = L \} \]
sa;
\[
\{ \text{Confl}_\text{free}([s \overset{\lambda}{\rightarrow}]) \land \text{Inequiv}(Q - [s \overset{\lambda}{\rightarrow}]) \land \mathcal{L} = L \} \]
\[
\{ \text{Inequiv(\text{Succ}(Q - [s \overset{\lambda}{\rightarrow}] \cup \{\delta^*(s, l)\}))) \} \]
\\[\| \text{var } p : \text{STATE} \]
\[
\{ \text{Confl}_\text{free}([s \overset{\lambda}{\rightarrow}]) \land \text{Inequiv}(Q - [s \overset{\lambda}{\rightarrow}]) \land \text{Pairwise}_\text{inequiv}(Q - [s \overset{\lambda}{\rightarrow}], \{\delta^*(s, l)\}) \} \]
To make this algorithm more practical (by avoiding recomputing \( p = \delta^*(s, l) \)), we could make \( p \) a parameter instead of recomputing it. This gives us a new version of our procedure:

\[
\text{proc visit\_min}'(\text{in} \ p : \text{STATE}; \text{in} \ l, r : \sum^*) \rightarrow \\
\{ \text{pre} \ \text{Confl\_free}([s \sim_{\langle \rangle}]) \land \text{Inequiv}(Q - [s \sim_{\langle \rangle}]) \land p = \delta^*(s, l) \land p \neq \bot \land L = L \} \\
\text{as} \ r \neq \epsilon \rightarrow \text{visit\_min}'(\delta(p, \text{head}(r)), l \cdot \text{head}(r), \text{tail}(r)) \\
\{ \ \text{Inequiv}(Q - [s \sim_{\langle \rangle}]) \land L = L \} \\
\text{sa; } \\
\{ \ \text{Confl\_free}([s \sim_{\langle \rangle}]) \land \text{Inequiv}(Q - [s \sim_{\langle \rangle}]) \land L = L \} \\
\{ \ \text{Inequiv}(\text{Succ}(Q - [s \sim_{\langle \rangle}]) \cup \{p\}) \} \\
\text{as} \ \langle \exists \ q : q \in Q - [s \sim_{\langle \rangle}] : \text{eq}'(p, q) \rangle \rightarrow \\
\text{let} \ q : q \in Q - [s \sim_{\langle \rangle}] \land \text{eq}'(p, q); \\
\text{merge}(p, q) \\
\text{sa} \\
\{ \ \text{Confl\_free}([s \sim_{\langle \rangle}]) \land \text{Inequiv}(Q - [s \sim_{\langle \rangle}]) \land \text{Pairwise\_inequiv}(Q - [s \sim_{\langle \rangle}], \{\delta^*(s, l)\}) \\
\land L = L \} \\
\{ \ \text{post} \ \text{Inequiv}(Q - [s \sim_{\langle \rangle}]) \land L = L \} \\
\text{corp}
\]

### 6.2 Procedure cleanup\(_1\)

Since \( \text{add\_word}_1 \) maintains minimality, we trivially have the following procedure

\[
\text{proc cleanup}_1() \rightarrow \\
\{ \ \text{pre} \ \text{Min} \land L = L \} \\
\text{skip} \\
\{ \ \text{post} \ \text{Min} \land L = L \} \\
\text{corp}
\]

### 6.3 An example

Starting with the MADFA from §4.3 starting on page 38
we add word heal using $\text{add}_{\text{word}_N}$, giving (where state 12 is a clone of 2 and 13 is new)

We then move on to $\text{visit}_{\min}(\epsilon, \text{heal})$, which considers $[s_{\text{heal}}^\sim] = [0, 1, 6, 12, 13]$ bottom up, we see that state 13 can be merged into state 3, giving
This ADFA is minimal since nothing further is minimized on \([s \sim^{heh}] = [0, 1, 6, 12]\). Consider now adding word `hal` using `add\_word_N'`, giving
In the invocation of \texttt{visit\_min}' we consider $[s^\text{hal}] = [0, 1, 2, 14]$, state 14 is merged with state 3, giving
States 2 and 12 are then found to be equivalent, giving
This last automaton is minimal.

### 6.4 Time and space performance

From Chapter 5, procedure $\text{add\_word}_N'$ takes $\mathcal{O}(|w|)$ time and space when adding word $w$. Using a clever coding of $\text{eq}'$, an invocation $\text{visit\_min}'(s, \varepsilon, w)$ also takes $\mathcal{O}(|w|)$ time and space — see [WD03]. It follows that $\text{add\_word}_I$ can also be implemented in $\mathcal{O}(|w|)$ time and space.

There are three relatively easy improvements:

1. Given Property 2.70, we can use invocation $\text{visit\_min}'(\delta(s, \text{head}(w)), \text{head}(w), \text{tail}(w))$ when $w \neq \varepsilon$.

2. While adding $w$ with $\text{add\_word}_N'$, we already traverse path $[s \xrightarrow{w} \cdot]$ and need not compute it anew within $\text{visit\_min}'$.

3. Another minor improvement can be made in the invocation of $\text{add\_word}_N'$: we need not create new states if they will subsequently be merged by $\text{visit\_min}'$. This approach, which is used in practice, requires some additional book-keeping, and has been presented in [Wat03b, Wat01d].

### 6.5 Commentary

Procedure $\text{add\_word}_I$ is essentially the same (modulo presentation and derivation style) as in [PAMS94, SFK95, Dac98, Mih99a, Mih99b, CD99] — though the greatest similarity is with those given in [DWW98, DMWW00]. Another version of this algorithm was presented in [Wat03b].
Chapter 7

Reversed trie intermediate ADFA

In this chapter, we maintain $M$ as a trie corresponding to the reverse of the words added so far. Formally, the invariant is

$$\text{Struct}_R(D) \equiv \text{Is\_trie} \land L = D^R$$

The resulting ADFA accepts $W^R$. Minimality of $M$ is achieved by reversing $M$ (usually yielding a nondeterministic automaton) and determinizing it. The reversal and determinization steps are combined in this chapter into $\text{cleanup}_R$. We will only present the procedures and examples — a full derivation of this algorithm can be found in [Wat01e, Wat02a] and most recently in [Wat02b], where an alternative derivation is given.

7.1 Procedure $\text{add\_word}_R$

As our specification, we get:

$$\text{proc } \text{add\_word}_R(\text{in } w : \Sigma^*) \to$$

{  \begin{align*}
\text{pre} & & \text{Is\_trie} \land L = L \\
S_{7,1} & & \text{post} \text{ Is\_trie} \land L = L \cup \{w^R\} \\
\end{align*} \}

\text{corp}

Given the specification of $\text{add\_word}_T$ (in Chapter 4), if we assume that argument $w$ can be reversed as a primitive operation (it can be done in $O(|w|)$ time and constant space), we implement $\text{add\_word}_R$ as

$$\text{proc } \text{add\_word}_R(\text{in } w : \Sigma^*) \to$$

{  \begin{align*}
\text{pre} & & \text{Is\_trie} \land L = L \\
\text{add\_word}_T(w^R) & & \text{post} \text{ Is\_trie} \land L = L \cup \{w^R\} \\
\end{align*} \}

\text{corp}

Naturally, it would also be easy to specialize $\text{add\_word}_T$ explicitly by expanding its body.

7.2 Procedure $\text{cleanup}_R$

We now require a minimization procedure with specification:
Without discussing the details (which are given originally in [Brz62a] and again in alternative forms in [Wat95, Wat00b, Wat02a], with surprisingly concise derivations), cleanupR can be implemented by reversing M and determinizing the result using the automata determinization (also known as the ‘subset construction’) algorithm; see [HU79] for a detailed treatment of automata determinization.

**Implementation 7.1** The efficiency of this algorithm hinges on a good encoding of the ADFA which is reversible — usually by storing the reversed transitions in addition to the forward transitions — and an efficient implementation of the determinization algorithm. Aspects of efficient determinization implementations are discussed in [JW96].

### 7.3 An example

The reverse trie corresponding to he, her, had, head, hard, herd, here, heard is

```
proc cleanupR() →
   { pre Is_trie ∧ ℒ = L }
   Q, δ, s, F : S_{7,2}
   { post Min ∧ ℒ = L^R }
   corp
```

The MADFA resulting from applying cleanupR to the above reverse trie is
This ADFA is minimal.

### 7.4 Time and space performance

As a simple variant of procedure \texttt{add_word}_T (in Chapter 4), procedure \texttt{add_word}_R has the same running time and space, also yielding a trie of size $O(\sum_{w \in W} |w|)$. I conjecture that procedure \texttt{cleanup}_R takes time and space $O(|M|)$. In particular, I conjecture the determinization step to be linear; the reversal step is easily done in linear time. Assuming this conjecture, the construction and minimization takes $O(\sum_{w \in W} |w|)$ space and time.

### 7.5 Commentary

This algorithm is a specialization (to acyclic DFAs) of Brzozowski’s DFA minimization algorithm [Brz62a, Brz62b]. More recently, it is described in [Wat00a, Wat02a]. Brzozowski’s minimization algorithm has an interesting history, described in [Wat00b, Wat01a].
Chapter 8

Avoiding cloning while adding words

Performance profiling of an implementation of the algorithm presented in Chapter 6, shows that most of the execution time is spent on two operations:

2. Merging states found to be equivalent.

(Creating new states is a cheap operation in practice.) While the merging operation is generally unavoidable in constructing a MADFA, in the subsequent chapters, we focus on a performance improvement by eliminating cloning. Cloning is only applied to confluence states. In those chapters, the structural invariants and the word-orders will be chosen in such a way that the preconditions of the add_word variants are strengthenings of Confl_free[[s, w]] when adding word w. As a result, the body of each add_word variant will use add_word (from Chapter 4), followed by further operations to restore the appropriate structural invariant and prepare for the next word to be added.
Chapter 9

Words in lexicographic order

In this chapter, we avoid the cloning operation by adding the words in lexicographic order so that part of the automaton will never be visited again while adding a word later. After those states are last visited, we consider them for equivalence with other states and merge them where possible — thereby enlarging the ‘minimized’ portion of the automaton. (Recall from Property 2.25 on page 11 that the merge operation usually creates confluence states.)

The only part of the automaton which will be ‘unminimized’ and is guaranteed not to have any confluences is the path of the lexicographically greatest word accepted by the automaton (the last word added in our ordering). The structural invariant is therefore

\[
\text{Struct}_S(D) \equiv \text{Inequiv}(Q - [s_{\text{lexmax}}]) \land \text{Confl\_free}( [s_{\text{lexmax}}]) \land \mathcal{L} = D
\]

The lexicographic order of word-adding yields the following precondition when adding word \( w \)

\[
\text{lexmax} \sqsubseteq_1 w
\]

Thanks to the definition of \( \sqsubseteq_1 \), we also have

\[
\text{Confl\_free}([s \sim])
\]

in the pre- and postcondition. We do not add it there explicitly because \( w \) consists of two parts:

- A prefix shared with \( \text{lexmax} \), namely \( \text{lexmax}^{\Delta_p}w \). We already know that the path corresponding to this prefix is confluence-free because \( \text{Confl\_free}([s_{\text{lexmax}}]) \) holds from \( \text{Struct}_S \).

- The corresponding suffix \( w \), namely \( (\text{lexmax}^{\Delta_p}w)^{-1}w \). The path of this suffix is not yet in the automaton and corresponds to states still to be added.

We return to the structure of \( w \) in the next section.

9.1 Procedure \( \text{add\_word}_S \)

Our starting point is

\[
\text{proc} \quad \text{add\_word}_S \left( \text{in} \ w : \Sigma^* \right) \rightarrow \\
\{ \quad \text{pre} \quad \text{lexmax} \sqsubseteq_1 w
\]
\[ \wedge \text{Inequiv}(Q - [s \xrightarrow{\text{lexmax}}]) \]
\[ \wedge \text{Confl\_free}([s \xrightarrow{\text{lexmax}}]) \]
\[ \wedge L = L \}

\emph{S\textsubscript{9.1}}
\[ \{ \text{post} \text{ lexmax} = w \]
\[ \wedge \text{Inequiv}(Q - [s \xrightarrow{\text{lexmax}}]) \]
\[ \wedge \text{Confl\_free}([s \xrightarrow{\text{lexmax}}]) \]
\[ \wedge L = L \cup \{w\} \}

\textit{corp}

We will use \texttt{add\_word}_T to add \textit{w}, since we are guaranteed not to encounter any confluences on \([s \xrightarrow{\text{w}}]\). Adding \textit{w} will traverse the longest common prefix of \textit{lexmax} and \textit{w}, namely \(\textit{lexmax} \triangleq \textit{w}\), before adding new states for the remainder of \textit{w}: \((\textit{lexmax} \triangleq \textit{w})^{-1} \textit{w}\), as mentioned earlier. We introduce shadow variable \textit{z} to capture the previous \textit{lexmax} before the invocation of \texttt{add\_word}_T. After \textit{w} has been added, we have

\[ \text{Confl\_free}([s \xrightarrow{\text{z}}] \cup [s \xrightarrow{\text{w}}]) \]

while the remainder of the automaton is minimized:

\[ \text{Inequiv}(Q - ([s \xrightarrow{\text{z}}] \cup [s \xrightarrow{\text{w}}])) \]

Our refined procedure is:

\texttt{proc add\_word}_S (\texttt{in} \textit{w} : \Sigma^*) \rightarrow 
\[ \{ \texttt{pre} \text{ lexmax} \sqsubseteq_1 \textit{w} \wedge z = \text{lexmax} \]
\[ \wedge \text{Inequiv}(Q - [s \xrightarrow{\text{lexmax}}]) \]
\[ \wedge \text{Confl\_free}([s \xrightarrow{\text{lexmax}}]) \]
\[ \wedge L = L \}
\]
\texttt{add\_word}_T(\textit{w});

\[ \{ \text{lexmax} = \textit{w} \]
\[ \wedge \text{Inequiv}(Q - ([s \xrightarrow{\text{z}}] \cup [s \xrightarrow{\text{w}}])) \]
\[ \wedge \text{Confl\_free}([s \xrightarrow{\text{z}}] \cup [s \xrightarrow{\text{w}}]) \]
\[ \wedge L = L \cup \{w\} \}

\emph{S\textsubscript{9.1}}
\[ \{ \text{post} \text{ lexmax} = \textit{w} \]
\[ \wedge \text{Inequiv}(Q - [s \xrightarrow{\text{lexmax}}]) \]
\[ \wedge \text{Confl\_free}([s \xrightarrow{\text{lexmax}}]) \]
\[ \wedge L = L \cup \{w\} \}

\textit{corp}

Considering the specification (pre- and postcondition) of \emph{S\textsubscript{9.1}}, we will have to minimize states \([s \xrightarrow{\text{z}}] - [s \xrightarrow{\text{w}}]\) — that is, all of the states on the \textit{z}-path after the split from the \textit{w}-path. In this case, ‘minimizing’ means comparing the states for equivalence against the already inequivalent states \(Q - ([s \xrightarrow{\text{z}}] \cup [s \xrightarrow{\text{w}}])\).

Since the common path (which may be visited again) is \([s \xrightarrow{\text{z}}] \triangleq \textit{w}\), the states \([s \xrightarrow{\text{z}}] - [s \xrightarrow{\text{w}}]\) to be minimized can also be written

\[ (s^*(s, \triangleq \textit{w}) \triangleright^{-1} \textit{z}) \]
9.1. **PROCEDURE ADD_WORD**

(Note the open/noninclusive beginning of this path.) If this path is empty (that is, \(z^{\hat{p}}w^{-1}z = \varepsilon\), which occurs when \(z\) is a prefix of \(w\)), no states need to be minimized. Otherwise, the minimization step can be accomplished using procedure \(\text{visit}_{\text{min}}\) from §6.1.1, beginning on page 51. Recall that \(\text{visit}_{\text{min}}\) takes two arguments \(l, r\) and minimizes \([\delta^*(s, l) \sim^r]\). Note the closed/inclusive beginning of this path, so that the naïve invocation

\[
\text{visit}_{\text{min}}(z^{\hat{p}}w, (z^{\hat{p}}w)^{-1}z)
\]

would therefore accidentally also minimize state \(\delta^*(s, z^{\hat{p}}w)\). What is needed instead is

\[
\text{visit}_{\text{min}}(z^{\hat{p}}w \cdot \text{head}((z^{\hat{p}}w)^{-1}z), \text{tail}((z^{\hat{p}}w)^{-1}z))
\]

The resulting algorithm (with correct invocation of \(\text{visit}_{\text{min}}\) and in which \(z\) is now a program variable\(^1\) capturing the previous value of \(\text{lexmax}\) and \(u, v\) are two variables to aid in readability):

```plaintext
proc add_word_s (in \(w : \Sigma^\ast\)) →
{ pre \text{lexmax} \sqsubseteq 1 \text{w}
  \land \text{Inequiv}(Q - [s^{\text{lexmax}}])
  \land \text{Confl_free}(s^{\text{lexmax}})
  \land L = L
} [ var u, v, z : \Sigma^\ast
| z := \text{lexmax};
  \text{add_word}_T(w);
  { \text{lexmax} = w
    \land \text{Inequiv}(Q - ([s^{\text{z}}] \cup [s^{\text{w}}]))
    \land \text{Confl_free}([s^{\text{z}}] \cup [s^{\text{w}}])
    \land L = L \cup \{w\} }
  u := z^{\hat{p}}w;
  v := u^{-1}z;
  \text{as } v \neq \varepsilon \rightarrow \text{visit}_{\text{min}}(u \cdot \text{head}(v), \text{tail}(v)) \text{ sa}
]} { post \text{lexmax} = w
  \land \text{Inequiv}(Q - [s^{\text{lexmax}}])
  \land \text{Confl_free}([s^{\text{lexmax}}])
  \land L = L \cup \{w\} }

corp

Implementation 9.1 The test in the \text{as-sa} statement can be simplified as follows:

\[
v \neq \varepsilon \\
\equiv \text{“} v = u^{-1}z \text{ and } u = z^{\hat{p}}w \text{”}
\]
\[
(z^{\hat{p}}w)^{-1}z \neq \varepsilon \\
\equiv \text{“} definition \text{ of derivatives } \text{”}
\]
\[
(z^{\hat{p}}w) \neq z
\]

\(^1\)As opposed to a shadow variable used for expressing pre- and postconditions.
\[ z \text{ is not a prefix of } w \]

This last form is easy to test: when it holds, \( \text{add} \textunderscore T \) will not have passed through state \( \delta^*(s, z) \) while adding word \( w \), and \( \text{add} \textunderscore T \) can be modified to note whether or not state \( \delta^*(s, z) \) was visited.

9.1.1 A minor problem in using \( \text{visit} \ _\text{min} \)

In \( \text{add} \textunderscore T \), we have taken a shortcut in our invocation

\[ \text{visit} \ _\text{min}(u \cdot \text{head}(v), \text{tail}(v)) \]

Before the invocation, we have

\[ \text{Inequiv}(Q \mathord{\mathcal{L}} (\{s \sim^T_z\} \cup \{s \sim^W_w\})) \]

By contrast, we can instantiate \( \text{visit} \ _\text{min} \text{'}s \) precondition (taken from §6.1.1 on page 51) conjunct using our invocation above:

\[ \text{Inequiv}(Q \mathord{\mathcal{L}} (s \sim^T_z)) \]

≡ “in our invocation of \( \text{visit} \ _\text{min} \): \( l = u \cdot \text{head}(v) \) and \( r = \text{tail}(v) \)”

\[ \text{Inequiv}(Q \mathord{\mathcal{L}} (s \sim^{u \cdot \text{head}(v)}_z \sim^{\text{tail}(v)}_z)) \]

≡ “definition of head and tail”

\[ \text{Inequiv}(Q \mathord{\mathcal{L}} (s \sim^{u \cdot v}_z)) \]

≡ “substituting variables’ values \( u = z^P \cdot w \) and \( v = u^{-1}z \)”

\[ \text{Inequiv}(Q \mathord{\mathcal{L}} (s \sim^z)) \]

≡ “string calculus, definition of common prefix and derivatives”

\[ \text{Inequiv}(Q \mathord{\mathcal{L}} (s \sim^z)) \]

Procedure \( \text{visit} \ _\text{min} \) therefore expects

\[ \text{Inequiv}(Q \mathord{\mathcal{L}} (s \sim^z)) \]

whereas we are only guaranteed the weaker

\[ \text{Inequiv}(Q \mathord{\mathcal{L}} (\{s \sim^z\} \cup \{s \sim^W\})) \]

before it is invoked in \( \text{add} \textunderscore T \). Indeed, at the point of invocation, we have \( \text{Confl} \ _\text{free}(\{s \sim^W\}) \) — those states are likely not inequivalent to states in \( Q \mathord{\mathcal{L}} (s \sim^z) \). In the body of \( \text{visit} \ _\text{min} \), this discrepancy affects in the second \( \text{as} \textunderscore \text{sa} \) (taken from page 53):

\[
\begin{align*}
\text{as} \ (\exists \ q : q \in Q - [s \sim^T_z] : \text{eq}')(p, q) \rightarrow \\
\text{let} \ q : q \in Q - [s \sim^T_z] \land \text{eq}')(p, q); \\
\text{merge}(p, q) \\
\text{sa} \\
\end{align*}
\]
The range of the quantification (and the let statement) should be narrowed to exclude \([s \sim w]\). This is most easily done by adding a parameter to visit\_min and passing in \(w\) while specializing that procedure's body. We do not do that here.

### 9.2 Procedure cleanup\(_S\)

After the last word has been added, a final minimization step is required to deal with states \([s \sim \text{lexmax}]\). This is trivially done using visit\_min'.

```plaintext
proc cleanup\(_S\) () →
  { pre Inequiv(Q − [s \sim \text{lexmax}])
    ∧ Confl\_free([s \sim \text{lexmax}])
    ∧ L = L }
  visit\_min(\(\varepsilon\), \text{lexmax})
  { post Min ∧ L = L }

corp
```

### 9.3 An example

Consider adding the words (using add\_word\(_S\)) in the lexicographic order had, hard, he, head, heard, her, herd, here. After adding he using add\_word\(_T\) (invoked from add\_word\(_S\)), but before minimizing, we have

![Diagram](image1.png)

The minimization step considers the \((\text{hard} \leftarrow \text{he})^{-1}\text{hard} = \text{ard}\) path from state 1 (state path \((1, 2, 4, 5)\)) and merges state 5 into 3, yielding

![Diagram](image2.png)

Adding head (an extension of he) and then heard and minimizing, we have
After adding her, but before minimizing, we have

Consider \( (\text{heard} \, \hat{\text{her}})^{-1} \text{heard} = \text{ard} \) from state 6 (state path \((6, 7, 9, 10)\)) shows that state 10 can be merged into state 3; subsequently state 9 can be merged into state 4, and finally state 7 can be merged into 2, giving

Adding the last two words herd and here yields
The cleanup\textsubscript{S} step considers \{0, 1, 6, 11, 13\} and only serves to merge states 3 and 13, giving the MADFA

### 9.4 Time and space performance

Sorting the words requires up to $O(|W| \log |W|)$ space and time; this is typically not taken into account in the MADFA construction time as $W$ can be kept sorted in real-life applications. From Chapter 4, procedure add\_word$\tau$ is time and space linear in the length of the word being added. As noted in Chapter 6, visit\_min can be implemented to take linear time and space. It follows that add\_word\textsubscript{S} and cleanup\textsubscript{S} are linear in the total lengths of the words.

#### 9.4.1 Improvements

Numerous improvements are possible, most of which are noted in the literature relating to visit\_min. They include the following:

1. Construct the new transitions and states (for $(z_{\hat{w}^\tau}w)^{-1}z$) lazily, since some of them may subsequently be merged in visit\_min.
2. [DMWW00] gives a way of finding state $\delta^*(s, z^p_w)$ without precomputing $z^p_w$.

9.5 Commentary

This algorithm was simultaneously derived by Daciuk and Mihov in their respective Ph.D. dissertations [Dac98] and [Mih99b]. Another presentation of the algorithm is given in [DMWW00].
Chapter 10

Words by decreasing length: minimizing depth layers

In this chapter, we derive a simple semi-incremental algorithm which, like the one in Chapter 9, depends upon an ordering of the words to avoid the relatively expensive cloning operation. The words are added in any order of decreasing length. No confluence states are encountered while adding a word \( w \), and states below depth\(^1 \) \( |w| \) are maintained pairwise inequivalent, while those at or above depth \( |w| \) are not necessarily confluence states. As in Chapter 9, this will enable us to use \( \text{add}_T \). Our first structural invariant is

\[
\text{Struct}_D(D) \equiv \text{Inequiv}(DL_{\geq \text{minlen}}) \land \text{Confl\_free}(DL_{\leq \text{minlen}}) \land \mathcal{L} = D
\]

\[
\text{Struct}_D(D) \equiv \text{Inequiv}(DL_{\geq \text{minlen}}) \land \text{Confl\_free}(DL_{\leq \text{minlen}}) \land \mathcal{L} = D
\]

10.1 Procedure \( \text{add}_D \)

Our starting point is

\[
\text{proc add}_D(w : \Sigma^*) \rightarrow
\{
\text{ pre } |w| \leq \text{minlen} \\
\text{ post } |w| = \text{minlen}
\}
\]

Clearly, in \( S_{10.1} \) we can use \( \text{add}_T \). (Here we also introduce shadow variable \( k \) to capture minlen.)

\[
\text{proc add}_D(w : \Sigma^*) \rightarrow
\{
\text{ pre } |w| \leq \text{minlen} \\
\text{ post } |w| = \text{minlen}
\}
\]

\(^1\)Not height; recall from Definition 2.57 that depth is a state's minimum path-distance from \( s \).
\[ L = L \]

\{ Confl\_free([s \sim w]) \}

\text{add\_word}_T(w);
\{ |w| = \text{minlen} \}

\\land \text{Inequiv}(DL_{>k})
\\land \text{Confl\_free}(DL_{\leq k})
\\land L = L \cup \{w\} \}

\text{S'}_{10.1}
\{ \text{post} \}
|w| = \text{minlen}
\\land \text{Inequiv}(DL_{>\text{minlen}})
\\land \text{Confl\_free}(DL_{\leq \text{minlen}})
\\land L = L \cup \{w\} \}

\text{corp}

Given that the word-lengths are monotonically decreasing, while adding \(w\) we are assured that no states deeper than \(|w|\) will be visited during future word-adding operations. After adding \(w\), we can minimize all states in \(DL_{>|w|}\); states \(DL_{>k}\) have already been done, so we need only consider the difference \(DL_{>|w|} - DL_{>k} = DL_{(|w|,k]}\). This gives us the following procedure, which implements \(S'_{10.1}\) straightforwardly:

\begin{verbatim}
proc \text{depths\_min}(\text{in i, j : } \mathbb{N}) \rightarrow \\
\{ \text{post} \ \text{Inequiv}(DL_{>j}) \\
\land L = L \}
\| \text{var} p, q : \text{STATE} \\
| \text{for} p : p \in DL_{(i,j]} \rightarrow \\
\as (\exists q : q \in DL_{>j} : eq(p, q)) \rightarrow \\
\let q : q \in DL_{>j} \land eq(p, q); \\
merge(p, q) \\
\sa \\
\| \\
\{ \text{post} \ \text{Inequiv}(DL_{>\text{min}(i,j)}) \\
\land L = L \}
\end{verbatim}

\text{corp}

Our final procedure is:

\begin{verbatim}
proc \text{add\_word}_D(\text{in w : } \Sigma^*) \rightarrow \\
\{ \text{pre} \ |w| \leq \text{minlen} \land k = \text{minlen} \\
\land \text{Inequiv}(DL_{>\text{minlen}}) \\
\land \text{Confl\_free}(DL_{\leq \text{minlen}}) \\
\land L = L \}
\{ \text{Confl\_free}([s \sim w]) \}
\text{add\_word}_T(w);
\{ |w| = \text{minlen} \\
\land \text{Inequiv}(DL_{>k}) \\
\land \text{Confl\_free}(DL_{\leq k}) \\
\land L = L \cup \{w\} \}
\text{depths\_min}(|w|, k)
\{ \text{post} \ |w| = \text{minlen} \\
\end{verbatim}
10.2 Procedure cleanup\(_D\)

Once the last word has been added, a final minimization step deals with the states \(DL_{(0,\text{minlen})}\) in:

\[
\text{proc \ cleanup}_D() \rightarrow \\
\{ \text{pre Struct}_D(L) \} \\
\text{depths}_\text{min}(0, \text{minlen}) \\
\{ \text{post Min } \land L = L \}
\]

Note that we do not explicitly consider \(DL_0 = \{s\}\) for minimization; that would have been unnecessary according to Property 2.70.

10.3 An example

Consider adding the words (using \(\text{add\_word}_D\)) in the order heard, herd, here, head, hard, her, had, he. After adding herd, we have

We can now minimize \(DL_{(4,5)} = \{5\}\) against \(DL_{(5)} = \emptyset\), and the automaton remains unchanged. After adding here, head, hard, we have
Once her is added (state 6 is made final), but before minimization, we have

We can minimize states $DL_{\{3,4\}} = \{4, 7, 8, 9, 12\}$ against $DL_{>4} = \{5\}$ and we see that all except 4 are final and without out-transitions and can be merged into state 5, giving
Adding had gives

and depths_min has nothing to minimize. Finally, adding he gives
This means that we can minimize states $DL_{(2,3)} = \{3, 6, 11, 12\}$ against $DL_{>3} = \{4, 5\}$. We see that state 11 can be merged into 4 and 12 into 5, giving

Our cleanup phase (cleanup$_D$ invoking $depths_{\min}(0, 2)$) minimizes $DL_{(0,2)} = \{1, 2, 10\}$ against $DL_{>2} = \{3, 4, 5, 6\}$. We can then merge state 10 into 3 giving our minimal automaton
In this particular example, each step (except for the invocation of cleanup_D) considered states at a single depth; this is not generally the case and holds in this example only because our set W consists of words of lengths 5, 4, 3, 2 — each possible length in [5, 2].

### 10.4 Time and space performance

Due to the simple implementation of depths_min, this algorithm is not particularly efficient. Word set W can be sorted into some order of decreasing length in time and space $O(|W|)$. Procedures add_word_D and cleanup_D potentially require exponential time and space (due to eq) while adding w.

**Conjecture 10.1** There is a strengthening of the invariant which would allow the use of eq', which is much more efficient.

More straightforward improvements are immediately possible, including maintaining minlen in a global variable instead of recomputing it.

### 10.5 Commentary

The algorithm presented here is a new one, not previously appearing in the literature.
Chapter 11

Words by decreasing length: minimizing semi-incrementally

In this chapter, we derive another algorithm which (as in Chapter 10) relies on the words being added in any order of decreasing length. Similarly, to avoid the cloning operation (and therefore allow us to use add_word_T), we require the to-be-traversed paths to remain confluence free. Since the words will be added in order of decreasing length, we are guaranteed never to encounter a final state while adding some word w. Intuitively, we could minimize that part of the automaton reachable from final states — a tighter invariant than in Chapter 10. Formally, we maintain invariant

\[
\text{Struct}_W(D) \equiv \text{Inequiv}(\text{Succ}^*(F)) \land \text{Confl\_free}(Q - \text{Succ}^*(F)) \land L = D
\]

\[\text{won't visit again} \land \text{may partly visit}\]

11.1 Procedure add_word_W

Our starting point is

\[
\text{proc add_word}_W(\text{in } w : \Sigma^*) \rightarrow \\
\{ \text{pre } |w| \leq \text{minlen} \\
\quad \land \text{Inequiv(Succ}^*(F)) \land \text{Confl\_free}(Q - \text{Succ}^*(F)) \\
\quad \land L = L \} \\
S_{11.1} \\
\{ \text{post } |w| = \text{minlen} \\
\quad \land \text{Inequiv(Succ}^*(F)) \land \text{Confl\_free}(Q - \text{Succ}^*(F)) \\
\quad \land L = L \cup \{w\} \}
\]

corp

After adding word w using add_word_T, we will have made state \(\delta^*(s,w)\) final. Since we are adding words in order of decreasing length, and will never pass through state \(\delta^*(s,w)\) or any other final state while adding another word, we can minimize states \(\text{Succ}^*(\delta^*(s,w))\), keeping in mind that states \(\text{Succ}^*(F - \delta^*(s,w))\) will already be minimized according to Struct_W. Our revised version (where we introduce shadow variable F' to capture F for expressing our postconditions) is

\[
\text{proc add_word}_W(\text{in } w : \Sigma^*) \rightarrow \\
\{ \text{pre } |w| \leq \text{minlen} \\
\quad \land \text{Inequiv(Succ}^*(F)) \land \text{Confl\_free}(Q - \text{Succ}^*(F)) \}
\]
\[ \mathcal{L} = L \]
\[ F' = F \]
\begin{align*}
\text{add\_word}_T(w); \\
\{ \ F = F' \cup \{ \delta^*(s, w) \} \land \delta^*(s, w) \notin F' \} \\
\{ \ \text{Inequiv}(\text{Succ}^*(F')) \land \text{Confl\_free}(Q - \text{Succ}^*(F')) \land \mathcal{L} = L \cup \{ w \} \}
\end{align*}

\[ S'_{11.1} \]
\begin{align*}
\{ \ \textbf{post} \ |w| = \minlen \\
\land \text{Inequiv}(\text{Succ}^*(F')) \land \text{Confl\_free}(Q - \text{Succ}^*(F')) \\
\land \mathcal{L} = L \cup \{ w \} \}
\end{align*}

corp

To implement \( S'_{11.1} \), we rewrite two of our postcondition conjuncts:

\[
\begin{align*}
\text{Inequiv}(\text{Succ}^*(F')) \land \text{Confl\_free}(Q - \text{Succ}^*(F')) \\
\equiv \quad \text{"value of } F \text{ after add\_word}_T \text{ invocation: } F = F' \cup \{ \delta^*(s, w) \}"
\end{align*}
\]

\[
\text{Inequiv}(\text{Succ}^*(F' \cup \{ \delta^*(s, w) \})) \land \text{Confl\_free}(Q - \text{Succ}^*(F' \cup \{ \delta^*(s, w) \}))
\]

We can establish this using a helper procedure specified as

\begin{verbatim}
proc semi_min(in p : STATE; in U : set of STATE) \rightarrow
{ \ pre \text{Inequiv}(\text{Succ}^*(U)) \land \text{Confl\_free}(Q - \text{Succ}^*(U)) \\
\land p \notin \text{Succ}^*(U) \land \mathcal{L} = L \}
\end{verbatim}

\[ S_{11.1.1} \]
\begin{align*}
\{ \ \textbf{post} \ \text{Inequiv}(\text{Succ}^*(U \cup \{ p \})) \land \text{Confl\_free}(Q - \text{Succ}^*(U \cup \{ p \})) \\
\land \mathcal{L} = L \}
\end{align*}

corp

(Note the similarities between this procedure’s specification and the specification of statement \( S'_{11.1} \) above, as well as the similarity to procedure visit\_min given in §6.1.1.)

In §11.1.1, we will derive an implementation of semi\_min. Using semi\_min, we can complete our implementation of add\_word\_w, where we have changed \( F' \) into a program variable for use in our invocation of semi\_min:

\begin{verbatim}
proc add\_word\_w(in w : \Sigma^*) \rightarrow
{ \ pre \ |w| \leq \minlen \\
\land \text{Inequiv}(\text{Succ}^*(F)) \land \text{Confl\_free}(Q - \text{Succ}^*(F)) \\
\land \mathcal{L} = L \}
|| \var F' : set of STATE \\
| F' := F; \\
add\_word\_T(w); \\
{ \ F = F' \cup \{ \delta^*(s, w) \} \land \delta^*(s, w) \notin F' \} \\
{ \ \text{Inequiv}(\text{Succ}^*(F')) \land \text{Confl\_free}(Q - \text{Succ}^*(F')) \\
\land \mathcal{L} = L \cup \{ w \} \}
semi\_min(\delta^*(s, w), F')
}\]
\{ \ \textbf{post} \ |w| = \minlen \\
\land \text{Inequiv}(\text{Succ}^*(F)) \land \text{Confl\_free}(Q - \text{Succ}^*(F)) \\
\land \mathcal{L} = L \cup \{ w \} \}
\end{verbatim}

corp
11.1. **PROCEDURE ADD\_WORD**

### 11.1.1 Procedure semi\_min

Recall semi\_min’s postcondition

\[
\text{Inequiv(Succ}^*(U \cup \{p\}) \land \text{Confl\_free}(Q - \text{Succ}^*(U \cup \{p\}))
\]

We rewrite this into a form which will be more easily established by two sequential statements, beginning with the first conjunct:

\[
\text{Inequiv(Succ}^*(U \cup \{p\}))
\]

\[
\equiv \text{“Succ}^*\text{ distributes over } U; \text{ notational shortcut: Succ}^*(p) \text{ for Succ}^*(\{p\})”
\]

\[
\text{Inequiv(Succ}^*(U \cup \text{Succ}^+(p)) \cup \{p\})
\]

\[
\equiv \text{“Property 2.47”}
\]

\[
\text{Inequiv}((\text{Succ}^*(U) \cup \text{Succ}^+(p)) \cup \{p\})
\]

\[
\equiv \text{“Property 2.83; re-order conjuncts w.r.t. the form of that property”}
\]

\[
\text{Inequiv(Succ}^*(U \cup \text{Succ}^+(p)) \land \text{Pairwise\_inequiv}((\text{Succ}^*(U) \cup \text{Succ}^+(p)) \cup \{p\}) \land \text{Inequiv}([p])
\]

\[
\equiv \text{“Inequiv always holds on a single state, p in this case”}
\]

\[
\text{Inequiv(Succ}^*(U \cup \text{Succ}^+(p))) \land \text{Pairwise\_inequiv}((\text{Succ}^*(U) \cup \text{Succ}^+(p)) \cup \{p\})
\]

establish in \(S'_{11.1.1}\) below  

establish in \(S''_{11.1.1}\) below

Intuitively

- \(S'_{11.1.1}\) ‘minimizes’ \(\text{Succ}^+(p)\) with respect to our already-minimized states \(\text{Succ}^*(U)\). Notably, state \(p\) is not to be minimized by \(S'_{11.1.1}\). Recall that \(\text{Confl\_free}(Q - \text{Succ}^*(U))\) is a precondition conjunct of \(\text{semi\_min}\). After \(S'_{11.1.1}\), additionally states \(\text{Succ}^+(p)\) may not be confluences. To capture this we add

\[
\text{Confl\_free}(Q - (\text{Succ}^*(U) \cup \text{Succ}^+(p)))
\]

as a \(S'_{11.1.1}\) postcondition conjunct.

- \(S''_{11.1.1}\) is to minimize \(p\) against states \(\text{Succ}^*(U) \cup \text{Succ}^+(p)\).

This gives us two statements establishing the conjuncts in the derivation above:

```plaintext
proc semi_min(in p : STATE; in U : set of STATE) ->

{ pre Inequiv(Succ*(U)) \land Confl_free(Q \minus Succ*(U)) \land p \not\in Succ*(U) \land L = L }

S'_{11.1.1} see derivation above

{ Inequiv(Succ*(U) \cup Succ+(p)) \land Confl_free(Q \minus (Succ*(U) \cup Succ+(p))) \land L = L }

S''_{11.1.1} see derivation above

{ Pairwise\_inequiv(Succ*(U) \cup Succ+(p), [p]) }

{ post Inequiv(Succ*(U \cup \{p\}) \land Confl_free(Q \minus Succ*(U \cup \{p\})) \land L = L }

corp
```

In the next two sections, we refine \(S'_{11.1.1}\) and \(S''_{11.1.1}\) respectively.
11.1.1.1 Refining $S'_{11.1.1}$

We can rewrite the first two postcondition conjuncts for $S'_{11.1.1}$

\[
\text{Inequiv}(\text{Succ}^*(U) \cup \text{Succ}^+(p)) \land \text{Confl\_free}(Q - (\text{Succ}^*(U) \cup \text{Succ}^+(p)))
\]

≡

"Property 2.47 twice, once in each conjunct"

\[
\text{Inequiv}(\text{Succ}^*(U) \cup \text{Succ}^*(\text{Succ}(p))) \land \text{Confl\_free}(Q - (\text{Succ}^*(U) \cup \text{Succ}^*(\text{Succ}(p))))
\]

≡

"Succ\_free distributes over \lor"

\[
\text{Inequiv}(\text{Succ}^*(U \cup \text{Succ}(p))) \land \text{Confl\_free}(Q - (\text{Succ}^*(U \cup \text{Succ}(p))))
\]

Note the similarity of the last derivation line to the postcondition conjuncts of semi\_min, namely

\[
\text{Inequiv}(\text{Succ}^*(U \cup \{p\})) \land \text{Confl\_free}(Q - \text{Succ}^*(U \cup \{p\}))
\]

We can most easily establish our predicate by considering the states Succ(p) one-by-one in recursive invocations of semi\_min. We accumulate those states of Succ(p) in local variable V for later recursive invocations (since they are by then part of the minimized states) in our $S'_{11.1.1}$ refinement (where local variable q is used for iterating over Succ(p))

\[
\begin{array}{l}
\{ \text{pre } \text{Inequiv}(\text{Succ}^*(U)) \land \text{Confl\_free}(Q - \text{Succ}^*(U)) \\
\quad \land p \notin \text{Succ}^*(U) \land \mathcal{L} = L \} \\
\end{array}
\]

\[
\begin{array}{l}
| q : \text{STATE}; \ V : \text{set of STATE} \\
| V := \emptyset; \\
\{ \text{invariant: } V \subseteq \text{Succ}(p) \land \text{Inequiv}(\text{Succ}^*(U \cup V)) \land \text{Confl\_free}(Q - \text{Succ}^*(U \cup V)) \\
\text{variant: } (\text{Succ}(p) - V) \} \\
\text{for } q : q \in \text{Succ}(p) \rightarrow \\
\quad \text{as } q \notin U \rightarrow \\
\quad \text{semi\_min}(q, U \cup V) \\
\quad \{ \text{Inequiv}(\text{Succ}^*(U \cup V \cup \{q\})) \land \text{Confl\_free}(Q - \text{Succ}^*(U \cup V \cup \{q\})) \} \\
\quad \text{sa;} \\
\quad V := V \cup \{q\} \\
\text{rof}; \\
\{ \text{Inequiv}(\text{Succ}^*(U) \cup \text{Succ}^+(p)) \land \text{Confl\_free}(Q - (\text{Succ}^*(U) \cup \text{Succ}^+(p))) \\
\quad \land \mathcal{L} = L \} \\
\end{array}
\]

11.1.1.2 Refining $S''_{11.1.1}$

We can rewrite the postcondition for $S''_{11.1.1}$ (see page 85 for how we obtained it in the first place)

\[
\text{Pairwise\_inequiv}(\text{Succ}^*(U) \cup \text{Succ}^+(p), \{p\})
\]

≡

"Property 2.76"

\[
\text{Pairwise\_inequiv}(\text{Succ}^*(U), \{p\}) \land \text{Pairwise\_inequiv}(\text{Succ}^+(p), \{p\})
\]

≡

"second conjunct, Corollary 2.74: p is inequivalent to all states Succ\_free (p)"

\[
\text{Pairwise\_inequiv}(\text{Succ}^*(U), \{p\})
\]
In our refinement of $S''_{11.1.1}$, we consider $p$ against $\text{Succ}^*(U)$, looking for an equivalent state $q$.

The seemingly redundant assignment of $q$ to $p$ in the above as-sa statement has a purpose: without it, after the merge of $p$ into $q$, variable/parameter $p$ does not refer to a valid state\(^1\), making its use in our postcondition dubious. State $q$ is of course equivalent, and the assignment allows us to leave the postcondition untouched; in an implementation it would be omitted.

Recall that $\text{eq}$ is less efficient than $\text{eq}'$ (see §2.5.3 and 2.5.4). In the precondition of $S''_{11.1.1}$ above, we have conjunct $\text{Inequiv}(\text{Succ}^*(U) \cup \text{Succ}^+(p))$. Thanks to Property 2.86, we also have $\text{Inequiv}(\text{Succ}(p))$ — allowing us to use $\text{eq}'$.

11.1.1.3 A final version of $\text{semi\_min}$

The procedure is:

```
proc \text{semi\_min}(\text{in} p : \text{STATE}; \text{in} U : \text{set of STATE}) \rightarrow 
{ \text{pre} \text{Inequiv}(\text{Succ}^*(U)) \land \text{Confl\_free}(Q \land (\text{Succ}^*(U) \cup \text{Succ}^+(p))) 
\land \mathcal{L} = L } 
\| \text{q : STATE; V : set of STATE} 
| V := \emptyset; 
{ \text{invariant: V} \subseteq \text{Succ}(p) \land \text{Inequiv}(\text{Succ}^*(U \cup V)) \land \text{Confl\_free}(Q \land (\text{Succ}^*(U \cup V))) 
\text{variant: |Succ(p) - V|} } 
\text{for} q : q \in \text{Succ}(p) \rightarrow 
\text{as} q \not\in U \rightarrow 
\text{semi\_min}(q, U \cup V) 
{ \text{Inequiv}(\text{Succ}^*(U \cup V \cup \{q\})) \land \text{Confl\_free}(Q \land (\text{Succ}^*(U \cup V \cup \{q\})) ) } 
\text{sa} ; 
V := V \cup \{q\} 
\text{rof} 
{ V = \text{Succ}(p) } 
}; 
{ \text{Inequiv}(\text{Succ}^*(U) \cup \text{Succ}^+(p)) \land \text{Confl\_free}(Q \land (\text{Succ}^*(U) \cup \text{Succ}^+(p))) 
\land \mathcal{L} = L } 
\| \text{var q : STATE} 
```

\(^1\)Recall from Definition 2.18 that invocation $\text{merge}(p, q)$ deletes $p$. 

| as $(\exists q : q \in \text{Succ}^*(U) : \text{eq}'(p, q)) \rightarrow$
| let $q : q \in \text{Succ}^*(U) \land \text{eq}'(p, q)$;
| merge($p, q$); $p := q$
| sa
| } \llbracket
| \{ \text{Pairwise}_\text{inequiv}(\text{Succ}^*(U) \cup \text{Succ}^+(p), \{p\}) \} \}
| \{ \text{post} \text{Inequiv}(\text{Succ}^*(U \cup \{p\})) \land \text{Confl}_\text{free}(Q - \text{Succ}^*(U \cup \{p\})) \land \mathcal{L} = L \} \}
|

11.2 Procedure cleanup$_W$

Once the last word has been added, our cleanup step deals with states $\text{Succ}^*(s) - \text{Succ}^*(F)$. We can rewrite the postcondition for cleanup$_W$

Min
\[ \equiv \quad \text{“definition of Min”} \]
\[ \text{Inequiv}(Q) \]
\[ \equiv \quad \text{“no useless states, so all are reachable from } s \text{”} \]
\[ \text{Inequiv}(\text{Succ}(s)) \]
\[ \equiv \quad \text{“set calculus, keeping in mind the postcondition of semi_min”} \]
\[ \text{Inequiv}(\text{Succ}^*(F - \{s\} \cup \{s\})) \]

This is easily established with invocation semi_min($s, F - \{s\}$). Our simple cleanup procedure is:

\[
\text{proc cleanup}_W() \rightarrow \\
\{ \text{pre Struct}_W(L) \} \\
\text{semi_min}(s, F - \{s\}) \\
\{ \text{post Min } \land \mathcal{L} = L \} \\
\text{corp}
\]

11.3 An example

Consider adding the words (using add_word$_W$) in the order heard, herd, here, head, hard, her, had, he. After adding herd with add_word$_T$, we have

The minimization step considers states $\text{Succ}^*(7) = \{7\}$ against the already-unique set $\text{Succ}^*(5) = \{5\}$ in invocation semi_min($7, \{5\}$), giving
After adding here, head, hard and minimizing, we have

While adding her with $\text{add\_word}_T$, we make state 6 final, yielding an automaton that remains the same after the $\text{semi\_min}(6,[5])$ minimization step (states 6 and 5 are inequivalent)
After adding had we get a new state and

Our minimization step considers $\text{Succ}^*(13) = \{13\}$ against $\text{Succ}^*(\{5, 6\}) = \{5, 6\}$ in $\text{semi_min}(13, \{5, 6\})$, allowing us to merge 13 into 5.
Finally, after adding `he`, we make state 2 final.

The minimization invocation `semi_min(2, {5, 6})` considers $\text{Succ}^*(2) - \text{Succ}^*({5, 6}) = \{2, 3, 4\}$ against $\text{Succ}^*({5, 6}) = \{5, 6\}$, leaving our automaton unchanged.

The cleanup invocation `semi_min(0, {2, 5, 6})` and considers

$$\text{Succ}^*(0) - \text{Succ}^*({2, 5, 6}) = \{0, 1, 10, 11\}$$

(bottom-up) against

$$\text{Succ}^*({2, 5, 6}) = \{2, 3, 4, 5, 6\}$$

and state 11 is merged into 4 and 10 into 3 giving our minimal result.
11.4 Time and space performance

Word set \( W \) can be sorted into some order of decreasing length in time and space \( O(|W|) \). As is shown in [Wat98b, Wat03a], procedure \( \text{add}_W \) requires time and space \( |w| \) while adding \( w \), and \( \text{cleanup}_W \) takes time and space \( |M| \). In [Wat03a], further performance improvements are discussed.

A simple implementation of this algorithm has proven to be efficient in practice [Wat03a]. In that paper, several other optimizations are discussed.

11.5 Commentary

As mentioned in Chapter 1, this algorithm was a new algorithm presented with a different derivation in [Wat98b] and in [Wat03a].
Bibliography


[Wat00a] Bruce W. Watson. Directly constructing minimal DFAs: Combining two algorithms by Brzozowski. In Yu and Păun [YP00], pages 242–249.

[Wat00b] Bruce W. Watson. A history of Brzozowski’s DFA minimization algorithm. In Yu and Păun [YP00].


Index


ADFA, see finite automata, deterministic, acyclic
alphabet, 1, 8–16, 18, 19, 21–24, 27–29, 44, 45, 49–54, 61, 67–69, 75, 76, 83, 84, 100

‘big-oh’, 21, 22, 41, 46, 59, 61, 63, 73, 81, 92

Booleans, 21, 22

⊥, see undefined

Confl_free, see confluence-free
confluence, 11, 45, 46, 99

δ, see transition function

depth level, 16, 25, 75–78, 80
depth-sandwich minimization procedure, 76, 77, 79–81

DFA, see finite automata, deterministic

empty string, 8, 9, 12, 13, 15, 17, 18, 20, 28–30, 35, 44–46, 50–55, 59, 69, 71
eq, see equivalent state function
equivalence, of states, 17–22, 33, 37
equivalent state function, 21, 22, 33–35, 37, 38, 41, 52–54, 59, 70, 76, 81, 87, 88, 99

false, 7

finite automata
deterministic, 10, 11, 13, 15, 17, 20, 21, 41, 47, 63, 99
acyclic, i, 1, 2, 11, 15–19, 21, 23, 24, 27, 28, 36, 40, 41, 43, 46, 47, 49, 56, 61–63, 99
minimal acyclic, i, 1, 17, 23, 44, 46, 54, 62, 65, 73, 99


function, cleanup, 23–25, 27, 30, 31, 34, 35, 37, 39, 41, 43, 46, 49, 54, 61–63, 71, 73, 77, 80, 81, 88, 91, 92

function, state cloning, 10, 44–46

function, state creation, 10, 24, 29, 30, 41, 45, 46

function, state merging, 10, 11, 16, 17, 33, 34, 37, 38, 53, 54, 67, 70, 76, 87, 88

head of a string, 8, 28–30, 45, 46, 52–54, 59, 69, 70

height level, 8, 28–30, 45, 46, 52–54, 59, 69, 70

Inequiv, see pairwise inequivalent states
ls_confl, see confluence
ls_trie, see trie


left language, 12, 13, 16, 17

lexicographic ordering, 9, 17, 67–69

lexicographically greatest word, 17, 24, 67–69, 71

longest common prefix, 9, 67–74

longest right word length function, 15, 16, 18, 19

MADFA, see finite automata, deterministic, minimal acyclic

max, 15–19

min, 16, 17, 76

minimality, 17, 20, 24, 25, 30–32, 34, 36, 38, 49–51, 54, 62, 71, 77, 88

naturals, 15, 16, 37, 76

≤, see partial order on words

pairwise inequivalent sets, 19, 20, 31–33, 35–39, 50–54, 85–88, 100


partial order on words, 23–25, 28, 99

99
path minimization procedure, 50–55, 57, 59, 69–71, 73, 84, 100
Pairwise_inequiv, see pairwise inequivalent sets
powerset, 7, 10
quantification
  existential, 14, 16, 18, 19, 33, 34, 53, 54, 70, 76, 87, 88
  maximum, 15, 16
  minimum, 16, 17
  union, 13, 14, 16, 20, 22, 34, 36–38
  universal, 11, 15, 18–20, 22, 35
right language, 12, 13, 15–19
semi-incremental minimization procedure, 84–91
set cardinality, 11, 15, 16, 18, 21, 22, 24, 28, 30–38, 41, 45, 46, 59, 61, 63, 73, 75, 76, 81, 83, 84, 86, 87, 92
shortest left word length function, 16, 17
shortest word length, 17, 25, 75–77, 81, 83, 84
Σ, see alphabet
state universe, 10, 28–38, 44, 45, 53, 54, 76, 84–87
Struct, see structural invariant
structural invariant, 23–25, 100
structural invariant, Daciuk-Mihov, 24, 67
structural invariant, depth-based, 25, 75, 77
structural invariant, double-reversal, 24, 61
structural invariant, incremental, 24, 49
structural invariant, nonincremental, 24, 43
structural invariant, nonincremental, trie structures, 24, 27
structural invariant, Watson, 25, 83, 88
tail of a string, 8, 9, 28, 30, 45, 46, 52–54, 59, 69, 70
transition function, 10–16, 18, 19, 21–24, 28–30, 41, 44–46, 50–54, 59, 62, 68–70, 74, 83, 84, 99
trie, 11, 24, 27, 28, 30, 61, 62, 99
true, 7
undefined, 10–12, 28–30, 45, 46, 99
visit_min, see path minimization procedure
Colophon

This thesis was produced on a variety of Apple Macintosh Powerbook and Macbook laptops, running MacOS X (the latest being Snow Leopard). All of the text, algorithmic and mathematics typesetting was done in $\TeX$ version 2\epsilon in the Charter text font with the Euler math font. The automata diagrams were generated using the graphviz graph-drawing tool.