Tree algorithms: two taxonomies and a toolkit

Citation for published version (APA):

DOI:
10.6100/IR633481

Document status and date:
Published: 01/01/2008

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
PROPOSITIONS

accompanying the dissertation

Tree Algorithms

Two Taxonomies
and a Toolkit

Loek Gerard Willem Antoine Cleophas
1. The taxonomies and toolkit described in this dissertation show that the uniform presentation of algorithms in a taxonomy helps to implement the algorithms in a uniform toolkit, confirming earlier results in [1].

*Chapter 8 of this dissertation.*


2. The good practical performance of two new filter functions compared to that of existing filter functions show that comparing and classifying existing algorithms can lead to theoretically and practically interesting new results.

*Chapters 5, 6 and 8 of this dissertation.*

3. The more time and effort is spent on comparing and classifying existing algorithms, the smaller and more uniform the resulting taxonomy of these algorithms becomes.

4. Comparing algorithms and classifying them in the form of a taxonomy makes existing publications on such algorithms superfluous.

*Chapters 5 and 6 of this dissertation, as well as [1, 2].*


5. Computer scientists should spend more time and effort on searching and classifying published results, as this often yields interesting new results and prevents ‘reinventing the wheel’.
6. Pressure to publish makes the preceding recommendation hard to follow, yet the fact that many give in to this pressure makes it all the more interesting to follow.

7. The content of the slides to be used for a conference presentation should be peer reviewed in advance of the presentation, making it more worthwhile to attend such a presentation.

8. Software engineering traditionally looks at other engineering disciplines for inspiration yet plays an increasingly important role in other engineering disciplines. This causes a circular dependency.

9. Given the lack of monitoring and quality assurance in Dutch secondary education over the past twenty years [3], students should be required to pass a basic Dutch language and basic math test before being allowed to enroll at university.


10. Given increasing human population and decreasing fish population, the ancient Chinese proverb ‘Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime’ may no longer hold true.

11. In view of Proposition 3, it is wise to stop comparing and classifying in time, lest readers no longer be impressed by the work performed.
Tree Algorithms

*Two Taxonomies and a Toolkit*

**PROEFSCHRIFT**

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 1 april 2008 om 16.00 uur

door

Loek Gerard Willem Antoine Cleophas

geboren te Roermond
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. M.G.J. van den Brand
en
prof.dr. B.W. Watson

Copromotor:

dr.ir. C. Hemerik
Tree Algorithms

Two Taxonomies
and a Toolkit

Loek G.W.A. Cleophas
Eerste Promotor: prof.dr. M.G.J. van den Brand  
(Technische Universiteit Eindhoven) 
Tweede Promotor: prof.dr. B.W. Watson  
(University of Pretoria) 
Copromotor: dr.ir. C. Hemerik  
(Technische Universiteit Eindhoven) 

Overige Leden Kerncommissie: 
prof.Ing. B. Melichar, DrSc. (Czech Technical University in Prague) 
prof.dr. M. de Berg (Technische Universiteit Eindhoven) 
dr. F. Neven (Universiteit Hasselt)

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2008-10.

© Loek G.W.A. Clephas, 2008.

Printing: Printservice Technische Universiteit Eindhoven 
Cover design: Oranje Vormgevers 
Cover background photograph: Coast live oaks at California Memorial Stadium, University of California, Berkeley, by Ingrid Stromberg,  
http://www.flickr.com/photos/ingorrr/

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Clephas, Loek Gerard Willem Antoine

NUR 980 
Subject headings: programming ; formal methods / taxonomy / trees and graphs / formal languages / finite automata / algorithms / pattern recognition / software development 
Voor mijn ouders:
Louise Cleophas-Boots †
Har Cleophas
Karin Cleophas-Thissen
Acknowledgments

While I worked on the research that resulted in this dissertation, a number of people were very helpful in providing support and guidance to me.

First of all, I thank Bruce Watson for taking me on as a PhD student, being enthusiastic about my ideas, giving me the freedom to explore many sidetracks, and being such enjoyable company on many occasions. I am glad you are my \textit{tweede promotor}.

\textit{Copromotor} Kees Hemerik provided me with the ever important day-to-day supervision. Your many suggestions, anecdotes, and words of wisdom were certainly helpful. You also took upon you the perhaps ungrateful task of getting me back onto the main track whenever I was pursuing various sidetracks too far. My sincere thanks for this. Undoubtedly, without your advice and gentle admonitions, I would have taken a lot longer to finish.

Halfway through the journey, I was fortunate enough to have Mark van den Brand agree to travel along in the field of regular tree languages and tree algorithms. Despite my tendency to write elaborate sentences, you were always willing to read and discuss my drafts. Your comments and questions certainly improved the readability and structure of this dissertation. I also thank you for agreeing to be my \textit{eerste promotor}, and for the pleasant and collegial atmosphere you managed to create in the Software Engineering and Technology research group.

Mark de Berg, Bořivoj Melichar and Frank Neven were kind enough to serve on my doctoral committee (\textit{kerncommissie}) and read my dissertation during the busy month of december. Bořivoj Melichar additionally read and extensively commented on early versions of a number of chapters last summer.

The Software Engineering and Technology group and its predecessor, the Software Construction group, provided a friendly atmosphere to work in. I thank its members, past and present, for this. The groups’ secretary Hanneke Driever provided a lot of support in handling administrative issues. Gerard Zwaan gave many detailed technical comments on countless drafts. I am confident that this greatly improved the quality of this dissertation. I also thank the members of the Eindhoven Tuesday Afternoon Club for the instructive and interesting meetings.

In the past years, I supervised a number of students during an internship or Master’s
project: Jan Ouwend, Remi Bosman, Martijn van de Rijdt, and Rudi Bruns were a
pleasure to work with. Roger Stroblenbger deserves a special mention, working with
me on the practical parts of the research both during his Master’s project and during
a temporary appointment after his graduation.

The Espressa/FASTAR research group members—both staff and students, in partic-
ular Derrick Kourie, Ernest Ketcha Ngassan and Morkel Theumissen—always pro-
vided a nice and stimulating atmosphere when I was visiting the University of Preto-
ria or meeting them elsewhere. I also thank the members of the Prague Stringology
Club for organizing the interesting and enjoyable Prague Stringology Conferences.
Its members—and Jan Holub in particular—always provided a friendly atmosphere
and pleasant company in Prague and at conference venues elsewhere.

I am very grateful to my friends and family for their support. I thank my youngest
brother Björn Cleophas for agreeing to serve as paraninf during my PhD defense.
Michiel Frisheit was willing to do the same, even crossing the Atlantic for it. Jeroen
Heijmans and Felix Ogg & Tera Uijtdewilligen in particular provided a lot of neces-
sary and enjoyable distraction and relativization.

Bregje, our relationship was just ten days old when my PhD research project started,
and it’s been over four years of hearing the words tree, taxonomy, and toolkit far too
often, yet you never complained. I am immensely grateful for your love and support.

Finally, this work would not have been possible without the care and support of
my parents: my mom Louise Cleophas-Boots who unfortunately could not see me
get this far, and my dad Har Cleophas and mom Karin Cleophas-Thissen, who
fortunately both could see me get this far. All three of you provided me with a safe
environment in which to grow up and always supported me and the choices I made.
I am most grateful for this and therefore dedicate this work to you.

Lock Cleophas
Breda, 8 February 2008
Contents

1 Prologue 1

1 Introduction 3
  1.1 Problem area ........................................... 3
  1.2 Problem statement ........................................ 5
  1.3 Solution method and contributions .......................... 6
  1.4 Dissertation structure ..................................... 7
  1.5 Reader’s guide ............................................. 8

2 Preliminaries 11
  2.1 Notation .................................................... 11
  2.2 Basic definitions .......................................... 12
  2.3 Alphabets, strings and languages ............................ 14
  2.4 Warshall’s algorithm and a variant .......................... 15
  2.5 Reachability under n-ary relations .......................... 17

3 Regular tree language theory 21
  3.1 Trees and tree languages ................................... 23
    3.1.1 Trees ................................................. 23
      3.1.1.1 Functions and operations on trees ................. 26
    3.1.2 Tree pattern matching ................................ 30
    3.1.3 Tree languages ....................................... 33
  3.2 Regular tree languages ..................................... 36
  3.3 Regular tree grammars .................................... 37
II Taxonomies

4 Taxonomy construction

4.1 Taxonomies and taxonomy construction

4.2 A taxonomy of pattern matching algorithms

4.3 Taxonomies and feature models

4.4 Advantages of taxonomy construction

5 Tree acceptance

5.1 Taxonomy overview

5.2 Running example

5.3 The problem and three naive solutions

5.4 Using root-to-frontier tree acceptors

5.4.1 Using more specific root-to-frontier tree acceptors

5.5 Using frontier-to-root tree acceptors

5.5.1 Using more specific frontier-to-root tree acceptors

5.6 Constructing tree acceptors

5.6.1 A first construction for undirected tree automata

5.6.1.1 Relation to match set computation

5.6.2 A construction without ε-transitions
5.6.3 A construction without unreachable states .................................. 103
5.6.4 A construction including all nonterminals as states .................. 106
5.6.5 Constructions for root-to-frontier tree acceptors ..................... 107
      5.6.5.1 A first construction ........................................ 107
      5.6.5.2 A construction without ε-transitions .......................... 108
      5.6.5.3 A construction including all nonterminals as states ....... 109
      5.6.5.4 Deterministic constructions ................................. 110
5.6.6 Constructions for frontier-to-root tree acceptors .................... 110
      5.6.6.1 A first construction ........................................ 111
      5.6.6.2 A construction without ε-transitions .......................... 112
      5.6.6.3 A construction including all nonterminals as states ....... 112
      5.6.6.4 Deterministic constructions ................................. 113
5.7 Recursive match set computation ......................................... 116
      5.7.1 Relation to DFRTA computation ................................ 121
      5.7.2 Computing auxiliary function values ............................ 123
      5.7.3 Using tabulated match set values ............................... 124
          5.7.3.1 Reachability-based tabulation ............................ 126
      5.7.4 Filtering match sets ........................................... 131
      5.7.5 Using tabulated match set values with filtering ................ 136
          5.7.5.1 Reachability-based tabulation with filtering ............. 138
5.8 Stringpath-based match set computation ................................ 145
5.9 Conclusions .................................................................. 149

6 Tree pattern matching ................................................. 153
6.1 Taxonomy overview .................................................. 153
       6.1.1 Related work .................................................. 157
6.2 The problem and some naive solutions ................................. 158
6.3 Using root-to-frontier pattern matchers ............................... 163
6.4 Using frontier-to-root pattern matchers ............................... 164
       6.4.1 Using deterministic frontier-to-root tree pattern matchers .. 166
6.5 Constructing tree pattern matchers ................................... 167
       6.5.1 A construction for undirected tree automata ................. 168
II Toolkits

7 TABASCO

7.1 Domain engineering .............................................. 214
7.2 Generative programming ....................................... 216
7.3 TABASCO .......................................................... 216
7.4 Toolkit design .................................................... 219
  7.4.1 The design of SPARE Time .............................. 220
  7.4.2 Advantages of taxonomy-based toolkit design ....... 223
7.5 DSL design and implementation ............................... 224
7.6 Evolution as part of TABASCO ................................ 228
7.7 Related work ...................................................... 229
7.8 Final remarks ..................................................... 230

8 Forest FIRE

8.1 Introduction ....................................................... 233
8.2 Related work ...................................................... 236
8.3 High-level design .............................................. 237
8.3.1 Taxonomy algorithms ..................................... 237
8.3.2 Automata constructions ................................. 239
  8.3.2.1 Deterministic and nondeterministic automata ... 240
  8.3.2.2 Tree acceptance and tree pattern matching .... 240
  8.3.2.3 Encapsulating direction ............................ 240
  8.3.2.4 Encapsulating filtering ............................. 241
  8.3.2.5 Different item types and item sets .............. 241
  8.3.2.6 Constructing stringpath matching automata .... 241
8.3.3 Automata, states, and items ............................ 242
  8.3.3.1 States ............................................... 243
  8.3.3.2 Items and item set providers .................... 243
  8.3.3.3 Stringpath automata .............................. 244
8.4 Basic data structures and algorithms .................... 245
8.5 Abstract algorithms versus implementations .......... 246
  8.5.1 Example 1: Algorithm (T-ACCEPTOR, FR, DET) ... 246
  8.5.2 Example 2: Transformation step RED-U ............ 247
8.6 Experiments .................................................. 249
  8.6.1 Acceptance automata constructions ................ 249
  8.6.2 Pattern matching automata constructions .......... 253
  8.6.3 Acceptance and pattern matching algorithms ...... 256
8.7 Experiences and conclusions ............................... 258

IV Epilogue ......................................................... 261

9 Conclusions .................................................... 263
  9.1 Contributions ............................................... 263
  9.2 Future work ................................................ 266

References ....................................................... 269

Summary .......................................................... 289
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samenvatting</td>
<td>291</td>
</tr>
<tr>
<td>Curriculum Vitae</td>
<td>293</td>
</tr>
</tbody>
</table>

viii
Part I

Prologue
Chapter 1

Introduction

This dissertation deals with the construction of two taxonomies and a toolkit of algorithms solving two different but closely related problems from the domain of regular tree languages: tree acceptance and tree pattern matching. The domain has a rich theory, has broad applicability, and contains many algorithms, yet it suffers from inaccessibility and difficulty in reasoning about and comparing the algorithms. Furthermore, it suffers from difficulty in comparing and choosing between the algorithms’ implementations. The taxonomies and the toolkit developed based on them serve as solutions to the domain deficiencies by classifying the algorithms according to their essential details and providing an implementation of (a subset of) the algorithms.

In this chapter, we introduce the problem statement, the solution method, and the contributions in more detail. We also give an overview of the dissertation structure.

1.1 Problem area

After the development of a substantial amount of formal language theory for (one-dimensional) string languages in the 1950s and early 1960s, a number of researchers started to look at generalizations of the string case. The generalization to trees was one of them. In the late 1960s and early 1970s, many theoretical results were published, particularly regarding regular tree languages on ordered, ranked trees.

The area of regular tree languages has a rich theory, with many results that are generalizations of regular string languages, and many relations between the two areas. Parts of this theory have broad (potential) applicability in a number of areas:

1. Code generation in compilers, particularly for instruction selection or optimization [AG85, HC86, Tur86, Din87, Mee88, AGT89, HK89, WW89, BDB90, FSW94, WM95].
2. Term rewriting [Kro75, HO82a, HO82b, O’D85].

3. Genetics, in particular RNA structure analysis [Gie98, SZ90].

4. XML document processing [Mur99, BKMW01, Nev02a, Nev02b, MLMK05, Sch07].

5. Verification, in particular for cryptography and network protocols [GK00, GL00, Mon03].

In this dissertation, the focus will be on algorithmic problems underlying applications of regular tree languages on ordered, ranked trees. Such trees occur in the first three application areas. We will from here on refer to regular tree languages on ordered, ranked trees simply as regular tree languages. We focus on this type of trees and the algorithmic problems using them because the theory is mature, and many algorithms solving these problems exist, yet a number of deficiencies related to them exist, which will be detailed in the next section.

The last two areas may involve different kinds of trees and extensions of the theory of regular tree languages to such trees. XML documents without attributes can be represented by ordered, unranked trees. For XML-related applications, related yet different and somewhat less mature theory on ordered, unranked trees has been developed in the last decade, with roots in earlier work in this area (see [Mur99, BKMW01, Nev02a, Nev02b, MLMK05, Sch07] for details). Applications of regular tree language theory in verification often extend this theory to deal with e.g. associativity and commutativity of symbols [LM94, Ols01, OT02a, OT02b].

In particular, we focus on two important algorithmic problem areas related to regular tree language theory that underlie some of the practical applications:

1. Tree acceptance. Given a regular tree grammar and an input tree, determine whether the input tree can be generated by the regular tree grammar i.e. is part of the language denoted by the regular tree grammar.

2. Tree pattern matching. Given a finite, non-empty set of trees (the pattern set) and an input tree, find the set of all occurrences of the patterns in the input tree.

The two problems are related and their solutions involve many of the same algorithmic ingredients, as will become apparent in this dissertation.

In term rewriting, tree pattern matching may be applied to find occurrences of rewrite rules’ left hand sides. In instruction selection, a regular tree grammar may be used, in which every production rule of a regular tree grammar is associated with an instruction of a target processor. Instruction selection then corresponds to solving the tree parsing problem for an intermediate representation tree. The tree parsing problem is an extension of the tree acceptance problem, in which one is also interested in how an input tree can be generated by a regular tree grammar. Due to time constraints, this extension will not be considered in this thesis.
1.2 Problem statement

Since the 1960s, many algorithms solving the aforementioned two algorithmic problems have been described in the literature. Related to these solutions, unfortunately, a number of deficiencies exist:

1. Inaccessibility of the theory and algorithms, which are scattered over the literature, and for which few overview publications—none of which is algorithm oriented—exist.

2. Difficulty of comparing the algorithms due to differences in presentation style and level of formality.

3. Lack of reference to the theory and lack of correctness arguments in publications of practical algorithms.

4. Lack of a large and coherent collection of implementations of the algorithms.

5. Difficulty of choosing between different algorithms for practical applications.

We mainly focus our attention on the first three deficiencies in this dissertation, as the solution method we propose below uses the results of solutions to these three deficiencies to solve the last two deficiencies. The first three deficiencies give rise to two important research questions which we aim to answer in this dissertation:

RQ1 How are the algorithms solving these algorithmic problems—found in the literature or as variations of those found in the literature—related, i.e. what are their commonalities and differences?

RQ2 How can the algorithms be presented together and in a common style such that their relations become clear and their correctness becomes apparent?

In this dissertation we present taxonomies of tree acceptance and tree pattern matching algorithms to answer these questions and hence as solutions to the first three deficiencies mentioned. The construction of such taxonomies is an important part of the TABASCO method, which we propose to apply to improve the situation with respect to the deficiencies mentioned.

Regarding the last two of the five deficiencies, the main research question is:

RQ3 How can the taxonomies—with the formal description of the tree algorithms, constructions and basic data structures and algorithms involved—be used in the design and implementation of a collection of implementations?

After presenting the two taxonomies, we consider a toolkit of tree algorithms that was developed based on the formal description of (many of) the algorithms in the taxonomies, the data structures involved, and more fundamental algorithms involved.
The taxonomy-based construction of toolkits forms another important step of TA-
BASCO and helps to solve the last two deficiencies.

Taxonomy construction and taxonomy-based construction of toolkits had already
been used successfully in the domain of regular string language theory. In [Wat95],
they were used to solve the above five deficiencies for the domain of regular string
language theory, for the problems of keyword pattern matching, string automaton
construction and string automaton minimization. In this dissertation, we apply the
approach to the related area of regular tree language theory and the problems of
tree acceptance and tree pattern matching.

We do not consider all algorithms solving the tree pattern matching or tree ac-
ceptance problems. We focus our attention on algorithms using tree automata or
(to a lesser extent) string automata, constructed from the pattern set or regular
tree grammar. We do not consider algorithms that preprocess the subject tree by
transforming it into a different data structure, as the intended applications of the
algorithms—particularly in code generation—usually deal with many or frequently
changing subject trees (yet relatively stable pattern sets or regular tree grammars).
(See e.g. [CH97, CH99, DGM94, Kos89, Cha02] for different approaches, to the tree
pattern matching problem in particular.)

1.3 Solution method and contributions

TABASCO (TAxonomy-BAsed Software COstruction) is a domain modeling and
domain engineering method for algorithmic domains [CWK+06].\footnote{We will inter-
changeably use the terms domain, field and problem area.} Its contributions
are the solution of the first three deficiencies by classifying algorithms in the form
of a taxonomy, and the solution of the last two deficiencies by creating a toolkit and
domain specific language based on this domain model.

The TABASCO process involves a number of steps, which are summarized below
(and will be considered in more detail in Chapter 7).

1. Selection of a specific algorithmic problem domain. A domain is chosen based
on its richness (existence of many algorithms and data structures), maturity
(availability of a rich theory that can be used to reason about the problems and
their solutions), and applicability (algorithms should have broad applicability
in practical software systems).

2. Literature survey. Once a domain has been chosen, a literature research and
survey is performed to gather as many related algorithms and data structures
as possible.

3. Taxonomy construction. A taxonomy is a classification of problems and solu-
tions. As with biological taxonomies, one can create a classification according
to essential details of algorithms and data structures from a certain domain. Such a classification makes the field more accessible and may lead to the discovery of new algorithms. Since we aim at taxonomies based on a formal representation of the problems and their solutions, a taxonomy gives us correctness arguments as well.

4. Toolkit design and implementation. The availability of a taxonomy simplifies the toolkit design process. The systematic use of formal specifications from the taxonomy provides guidance for the toolkit architecture. Due to the taxonomy-basedness, the toolkit will be more coherent and easier to understand and implement than toolkits based on an ad hoc design.

5. Benchmarking. Given the toolkit, one can perform benchmarking to determine the practical performance of the algorithms. Domain experts can then select toolkit components based on their knowledge of the domain, the theoretical complexity analysis included in the taxonomy, and the performance data obtained in the benchmarking process.

6. DSL and GUI design and implementation. A Domain Specific Language (DSL) may be developed to allow both novices and experts to obtain those components best suited for them. The mapping from domain specific description to toolkit component in the DSL can be implemented based on the theoretical complexity information in the taxonomy, as well as the data obtained from benchmarking. (Alternatively, the components may be integrated into a Graphical User Interface (GUI) for a development environment, allowing users to use the components by setting some properties. The property values then form the domain specific description, determining the toolkit component to be used.)

As mentioned, we mainly focus our attention on the steps up to and including taxonomy construction, i.e. solving the first three of the five deficiencies mentioned. The main contributions of this dissertation thus consist of two algorithm taxonomies and the knowledge gained about the correctness and relations of the algorithms in the taxonomies. The toolkit design and implementation steps of the method will be considered as well. Furthermore, some practical results on algorithm efficiency will be given, based on experiments performed with the toolkit. In our discussion of TABASCO we discuss all steps of the approach and use the domain of keyword pattern matching algorithms as an additional case study of the entire method.

1.4 Dissertation structure

This dissertation consists of four parts. The first part contains this chapter, as well as the next two chapters. In Chapter 2, we discuss the mathematical and notational preliminaries necessary for reading the remainder of this dissertation. That chapter
may be skipped and referred back to as necessary. In Chapter 3 the domain of regular tree languages is considered. The chapter gives an overview of the theory of such languages on ordered, ranked trees. The overview focuses on those concepts most important for the two algorithmic problems considered in this dissertation.

The second part contains three chapters. Chapter 4 briefly introduces taxonomies and TABASCO’s taxonomy construction, using a taxonomy of keyword pattern matching algorithms as a brief example. In Chapters 5 and 6 we discuss the taxonomies of tree acceptance and tree pattern matching algorithms respectively, which resulted from applying the taxonomy construction step to the tree acceptance and tree pattern matching problems.

Part III consists of Chapters 7 and 8. The first chapter discusses the TABASCO approach used in this dissertation to improve upon the current state of the domain with respect to the deficiencies mentioned. Apart from the two algorithmic problems on trees serving as (partial) case studies throughout this dissertation, a case study of TABASCO’s application to the domain of keyword pattern matching algorithms is treated. In Chapter 8 the toolkit of (a subset of the) algorithms in the tree acceptance and tree pattern matching taxonomies is considered. Particular attention will be paid to the influence on its design and implementation of the taxonomies (and of the formal description of the tree algorithms, constructions and basic data structures and algorithms involved). Some experimental results related to implementations of many of the algorithms are discussed in Chapter 8 as well.

Part IV concludes this dissertation. In Chapter 9, we give an overview of the main results and conclusions reached, while a list of open problems that are suitable for future investigation is discussed as well.

1.5 Reader’s guide

To make it easier for a reader to find his or her way in this dissertation, we provide a brief reader’s guide to Chapters 3 through 8, indicating which (parts) of those chapters particular readers might focus their attention on.

- The reader interested in getting an overview of the taxonomies should read Chapter 4 as well as the overview sections at the beginning of Chapters 5 and 6. To get an overview of the taxonomies, the reader should then consider the parts of those chapters indicated by ‘Detail’, ‘Algorithm’, and ‘Construction’. When necessary or desired, he or she may read other parts of the chapters for more details related to the taxonomies and refer back to Chapter 3 for definitions related to regular tree language theory.

- The reader interested in the details of one of the two taxonomies and algorithms in them might first want to read parts of Chapter 3 and read Chapter 4, discussing TABASCO’s taxonomy construction part. In the former,
– for tree acceptance, Sections 3.1 (except Section 3.1.2) through 3.4 of
Chapter 3 are most relevant, discussing theory related to trees, tree lan-
guages, tree grammars, and tree automata;
– for tree pattern matching, Sections 3.1 and 3.4 of Chapter 3 are most
relevant, discussing theory related to trees, tree patterns, matching, and
tree automata.

• For the reader interested in the TABASCO method, Chapters 4 and 7 may
be read on their own, as they are more or less self-contained (although some of
the notation used in examples might require the reader to consult Chapter 2).

• The reader interested in the implementation or performance of tree algorithms
may restrict his or her attention to Chapter 8 on the toolkit, although it
may be useful to read the introduction to Chapter 3 and the introductions
and conclusions to Chapters 5 and/or 6 to get an overview of the concepts
underlying the implementations in the toolkit.
Chapter 2

Preliminaries

In this chapter, basic notation, definitions and properties needed throughout this dissertation and not related to regular tree language theory are presented. The chapter can be skipped on first reading and referred back to as necessary.

2.1 Notation

Since much of this dissertation is concerned with taxonomies of existing algorithms, we will often use names for sets, relations, functions etc. as they occur in the literature. In addition, names will often be given that are suggestive of their use. Apart from the use of such names, we will use the following general naming conventions:

- $U, \ldots, Z$ for arbitrary sets.
- $\Sigma$ for (terminal) alphabets, $N$ for nonterminal alphabets.
- $a, \ldots, e$ for arbitrary set elements and for terminal alphabet symbols.
- $A, \ldots, E$ and $S$ for nonterminal alphabet symbols.
- $v, w, x, y, z$ for sequences of alphabet symbols (i.e. words or strings, elements of $(N \cup \Sigma)^*$).
- $\alpha, \beta, \gamma$ for trees whose nodes are labeled by alphabet symbols (i.e. for elements of $Tr(N \cup \Sigma, r)$, introduced in Chapter 3).
- $s, t, u$ for trees whose nodes are labeled by terminal alphabet symbols (i.e. for elements of $Tr(\Sigma, r)$, introduced in Chapter 3).
- $\xi, \lambda, \mu, \nu$ for tree nodes.
• $G$, $H$ for grammars.
• $K$, $L$ for languages.
• $i, \ldots, n$ for integer variables.
• $M$ for finite (tree) automata.
• $p$, $q$, $s$ for states, and $Q$ for state sets; note that $s$ is also used for trees.
• $R$ for relations and (particularly) tree automata transition relations, and $\delta$ for string automata transition functions.

Names will also be used with a subscript ($U_1$), superscript ($U^n$), prime symbol ($U'$) or tilde ($\tilde{U}$) attached.

### 2.2 Basic definitions

We use $\mathbb{B}$, $\mathbb{N}$ and $\mathbb{N}_+$ to denote the booleans, the set of all natural numbers, and $\mathbb{N}\setminus\{0\}$—the set of positive natural numbers—respectively.

We use the notation $\langle Q_{\oplus} a : R(a) : E(a) \rangle$ for quantifications where $Q_{\oplus}$ is the quantifier symbol associated with an associative and commutative binary operator $\oplus$ (with unit $e_{\oplus}$), $a$ is the quantified variable introduced, $R$ is the range predicate on $a$, and $E$ is the quantified expression. By definition, we have $\langle Q_{\oplus} a : false : E(a) \rangle = e_{\oplus}$. The following table lists some commonly quantified operators, their quantifier symbols, and their units:

<table>
<thead>
<tr>
<th>Operator</th>
<th>$\lor$</th>
<th>$\land$</th>
<th>$\cup$</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantifier symbol</td>
<td>$\exists$</td>
<td>$\forall$</td>
<td>$\cup$</td>
<td>$\text{MAX}$</td>
</tr>
<tr>
<td>Unit</td>
<td>false</td>
<td>true</td>
<td>$\emptyset$</td>
<td>$-\infty$</td>
</tr>
</tbody>
</table>

We use $\langle \text{Set} a : R(a) : E(a) \rangle$ for the set $\langle \cup a : R(a) : \{E(a)\} \rangle$.

**Example 2.2.1.** We give some examples of the quantification notation used:

\[
\langle \text{Set} i : 1 \leq i \leq 3 : i^2 \rangle = \langle \cup i : 1 \leq i \leq 3 : \{i^2\} \rangle = \{1, 4, 9\}
\]

\[
\langle \forall i : 1 \leq i \leq 3 : i \leq i^2 \rangle \equiv \text{true}
\]

Using more conventional notations, the quantifications would be represented as $\{i^2|1 \leq i \leq 3\}$, $\bigcup_{i=1}^{3} \{i^2\}$ and $\bigwedge_{i=1}^{3} i \leq i^2$ or similarly. Our notation has the advantage of making the quantified variables more explicit and separating them from the range predicates.

For any set $U$, the set of all subsets of $U$ is denoted $\mathcal{P}(U)$ and called the powerset of $U$. We use $U^*$ to denote the set of (possibly empty) sequences of elements of a set $U$, and $U^n$ ($n \geq 0$) for the set of all such sequences of length $n$. 
Given sets $U_1, \ldots, U_n$ ($n \geq 2$), any subset of $U_1 \times \ldots \times U_n$ is an $n$-ary relation. For $n = 2$, the term binary relation is used.

For sets $U$ and $V$, we use $f \in U \to V$ to denote a (total) function $f$ from $U$ to $V$. Sets $U$ and $V$ are called the domain and codomain of $f$. We will sometimes represent functions as sets of pairs, e.g. $(a,b) \in f \equiv f(a) = b$.

**Convention 2.2.2** (Function generalization to a set of arguments). For a function $f \in U \to V$, we often use the common generalization to a function $f \in \mathcal{P}(U) \to \mathcal{P}(V)$ obtained by taking as the function value for $a \in \mathcal{P}(U)$ the subset of the codomain consisting of the function values for the elements of $a$.

**Convention 2.2.3** (Relations as functions). Given sets $U$ and $V$ and $R \subseteq U \times V$, relation $R$ can be interpreted as a function $R \in U \to \mathcal{P}(V)$ defined for all $a \in U$ by

$$R(a) = \{b : (a,b) \in R\}.$$ 

Alternatively, relation $R$ can be interpreted as a function $R \in U \times V \to \mathbb{B}$ defined for all $a, b \in U$ by

$$R(a,b) \equiv (a,b) \in R.$$ 

Note that this convention easily extends to $n$-ary relations with $n > 2$. For brevity, we do not explicitly present such an extension.

**Definition 2.2.4.** Given sets $U$, $V$, and $W$ and relations $R \subseteq U \times V$ and $S \subseteq V \times W$, we define infix relation composition operator $\circ$ by

$$R \circ S = \{ (a,c) : (a,b) \in R \land (b,c) \in S \}.$$ 

Note that $R \circ S \subseteq U \times W$, i.e. $R \circ S$ is a relation on $U$ and $W$.

**Definition 2.2.5.** Given sets $U$, $V$, and $W$ and functions $f \in U \to V$ and $g \in V \to W$, we define infix function composition operator $\circ$ for all $a \in U$ by

$$(g \circ f)(a) = g(f(a)).$$ 

Note that $g \circ f \in U \to W$.

Note that relation composition when viewing relations as functions (as per Convention 2.2.3) is different from function composition. When this might cause confusion, it will be clear from the the context which kind of composition is meant.

**Definition 2.2.6** (Relation exponentiation and closure). Let $U$ be a set and $R \subseteq U \times U$ a relation, then

- $R^0 = I_U$, the identity relation on $U$,
- $R^i = R \circ R^{i-1}$ for $1 \leq i$,
- $R^* = \{ \bigcup i : 0 \leq i : R^i \}$, and
- $R^+ = \{ \bigcup i : 1 \leq i : R^i \}$.
An algorithm to compute \( R^+ \), the transitive closure of a relation \( R \), is presented in Section 2.4.

When \( n \) is clear from the context, we take \( \overrightarrow{a} \) to be the tuple \( (a_1, \ldots, a_n) \). Given a tuple \( \overrightarrow{a} = (a_1, \ldots, a_n) \) we use the tuple projection to element \( i \), \( \pi_i(\overrightarrow{a}) \), (for \( 1 \leq i \leq n \)) to denote \( a_i \). We define \( \Pi(\overrightarrow{a}) = \{a_1, \ldots, a_n\} \), i.e. \( \Pi \) flattens a tuple into the set containing the tuple elements.

For a function taking a single tuple as argument, we often omit one pair of parentheses in a function application, e.g. we use \( f(a_1, \ldots, a_n) \) for \( f((a_1, \ldots, a_n)) \).

We use predicate calculus in derivations [DS90] and present algorithms in an extended version of (part of) the guarded command language [Dij76]. In that language, \( x, y := X, Y \) is used for multiple-variable assignment, ; for sequential composition, if \( b_1 \rightarrow S_1 \mid \cdots \mid b_n \rightarrow S_n \) \( \Phi \) represents selection i.e. executing one of the \( S_i \) for which \( b_j \) evaluates to true (and aborting if none of them is true), and \( \textbf{do} \ b \rightarrow S \ \textbf{od} \) represents a repetition i.e. executing \( S \) repeatedly as long as \( b \) is true. The extensions of the basic language as \( \textbf{as} \ b \rightarrow S \ \textbf{sa} \) as a shortcut for \( \textbf{if} \ b \rightarrow S \ \textbf{1} \ \neg b \rightarrow \textbf{skip} \ \Phi \), and \( \textbf{for} \ x : R \rightarrow S \ \textbf{rof} \) for executing statement list \( S \) once for each value of \( x \) initially satisfying \( R \) (assuming there is a finite number of such values for \( x \)), in arbitrarily chosen order [Eij92].

### 2.3 Alphabets, strings and languages

An alphabet is a finite, non-empty set. The elements of an alphabet are called symbols. Given an alphabet \( \Sigma \), we call elements of \( \Sigma^* \) strings over \( \Sigma \). For string concatenation, we use \( \cdot \) or —when it is clear from the context—juxtaposition. Any subset of \( \mathcal{P}(\Sigma^*) \) is a (string) language.

We use \( \varepsilon \) to denote the empty string. For a string \( w \), we use \( |w| \) to denote its length, defined by \( |\varepsilon| = 0 \) and \( |av| = 1 + |v| \) (for every \( a \in \Sigma, v \in \Sigma^* \)).

**Definition 2.3.1** (String operators \(|,|,|,|\)). Assuming alphabet \( \Sigma \), we define four infix operators \(|,|,|,|\) \( \in \Sigma^* \times \mathbb{N} \rightarrow \Sigma^* \) for \( w \in \Sigma^* \) and \( i \in \mathbb{N} \) as follows:

- \( w|i \) is the string consisting of the \( i \) \( \min \) \( |w| \) leftmost symbols of \( w \)
- \( w|i \) is the string consisting of the \( (|w| - i) \) \( \max \) \( 0 \) rightmost symbols of \( w \)
- \( w|i \) is the string consisting of the \( i \) \( \min \) \( |w| \) rightmost symbols of \( w \)
- \( w|i \) is the string consisting of the \( (|w| - i) \) \( \max \) \( 0 \) leftmost symbols of \( w \)

The operators \(|,|,|,|\) are called left take, left drop, right take and right drop respectively.
Property 2.3.2 (String operators $|, |, |,$). For string operator $|, |, |$ and $|,$

\[(w|i)(w|i) = w
(w|i)(w|i) = w\]

for every $w \in \Sigma^*$ and $i \in \mathbb{N}$. \hfill \Box

Example 2.3.3 (String operators $|, |, |,$). $(abcd)|3 = abc$, $(abcd)|1 = bcd,$ $(abcd)|5 = abed$ and $(abcd)|10 = \varepsilon$. \hfill \Box

Definition 2.3.4 (String operator $|)$. Assuming an alphabet $\Sigma$, we define infix operator $\downarrow \in \Sigma^* \times \mathcal{P}(\Sigma) \rightarrow \Sigma^*$ for $w \in \Sigma^*$ and $\Sigma' \subseteq \Sigma$ by

\[
\varepsilon \downarrow \Sigma' = \varepsilon \\
(aw) \downarrow \Sigma' = a(w \downarrow \Sigma') \text{ if } a \in \Sigma' \\
(aw) \downarrow \Sigma' = w \downarrow \Sigma' \text{ if } a \notin \Sigma'
\]

This operator projects a string onto a (sub-)alphabet. \hfill \Box

Definition 2.3.5 (Functions pref and suff). For a given alphabet $\Sigma$, define $\text{pref} \in \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ and $\text{suff} \in \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ for any language $L$ as

\[
\text{pref}(L) = \langle \text{Set } v, w : vw \in L : v \rangle \\
\text{suff}(L) = \langle \text{Set } v, w : vw \in L : w \rangle
\]

Informally, $\text{pref}(L)$ ($\text{suff}(L)$) is the set of all strings which are (not necessarily proper) prefixes (suffixes) of strings in $L$. For string $w \in \Sigma^*$, we will write $\text{pref}(w)$ and $\text{suff}(w)$ instead of $\text{pref}(\{w\})$ and $\text{suff}(\{w\})$ respectively.

Furthermore, we use $\text{pref}_\neq(w)$ for $\text{pref}(w)\setminus\{w\}$, i.e. for the proper prefixes of $w$.

2.4 Warshall’s algorithm and a variant

We present a solution to the following problem:

Given a finite set $U$ and a relation $R \in U \times U \rightarrow \mathbb{B}$, determine $R^+$, the transitive closure of $R$.

A solution to this problem can be used to compute the so-called nonterminal closure for a regular tree grammar. We will encounter this nonterminal closure in Sections 3.3.3 and 5.7.2. The solution we present here is based on [Zwa05]. Originally, the problem was solved by Warshall [War62], whom the resulting algorithm was named after.
For transitive closure $R^+$ we have, for every $a, b \in U$,
\[ a R^+ b \equiv a R b \lor (\exists c : c \in U : a R c \land c R^+ b) . \]
Replacing $U$ in the existential quantification by any $V$ such that $V \subseteq U$ gives relation $R_V$, and we have
\[
R_{\emptyset} = R \\
R_U = R^+
\]
and, for $a, b \in U$ and $c \in U \setminus V$,
\[ a R_{V \cup \{c\}} b \equiv a R_V b \lor (a R_V c \land c R_V b) \]
and (derivation omitted)
\[ a R_{V \cup \{c\}} c \equiv a R_V c \\
c R_{V \cup \{c\}} b \equiv c R_V b. \]
Using these properties we obtain Warshall’s algorithm:

\[
\begin{array}{l}
\begin{array}{l}
\textbf{var} \ r : U \times U \rightarrow \mathbb{B}; \\
\quad V : \mathcal{P}(U); \\
\quad c : U \\
\textbf{let} \ V := \emptyset; \\
\textbf{for} \ a, b : a, b \in U \rightarrow \\
\quad r(a, b) := a R b \textbf{ rof}; \\
\quad \{ \textbf{ inv } \emptyset \subseteq V \subseteq U \land r = R_V \} \\
\textbf{do} \ V \neq U \rightarrow \\
\quad \textbf{let} \ c \in U \setminus V; \\
\quad \textbf{for} \ a, b : a, b \in U \rightarrow \\
\quad \quad r(a, b) := r(a, b) \lor (r(a, c) \land r(c, b)) \textbf{ rof}; \\
\quad \{ r = R_{V \cup \{c\}} \} \\
\quad V := V \cup \{c\} \\
\textbf{od} \\
\quad \{ r = R_U = R^+ \} \\
\end{array}
\end{array}
\]

By eliminating the explicit use of $V$ we obtain:

\[
\begin{array}{l}
\begin{array}{l}
\textbf{var} \ r : U \times U \rightarrow \mathbb{B} \\
\quad \textbf{for} \ a, b : a, b \in U \rightarrow r(a, b) := a R b \textbf{ rof}; \\
\quad \textbf{for} \ c : c \in U \rightarrow \\
\quad \quad \textbf{for} \ a, b : a, b \in U \rightarrow \\
\quad \quad \quad r(a, b) := r(a, b) \lor (r(a, c) \land r(c, b)) \textbf{ rof}; \\
\quad \quad \textbf{rof} \\
\textbf{rof}
\end{array}
\end{array}
\]
Since the inner of the nested for-loops is $O(|U|^2)$, the complete algorithm has $O(|U|^3)$ running time.

**Example 2.4.1.** Given set $\{S,T,U\}$ and relation $R = \{(S,T),(T,U)\}$, the algorithm results in $r(S,T) = \text{true}$, $r(T,U) = \text{true}$ (both by the first for-loop), $r(S,U) = \text{true}$ (by the second one) and $r$’s value being false for other element pairs.

We can obtain an alternative version of Warshall’s algorithm by representing relation $R$ by a function $f_R$ instead, as per Convention 2.2.3. Similar definitions can be given for $f_{R^+}$ and $f_{R_v}$.

For every $a \in U$ and $c \in U \setminus V$,

$$f_{R_{V\cup\{c\}}} (a) = f_{R_v}(a) \cup f_{R_v}(c) \text{ if } c \in f_{R_v}(a)$$

$$f_{R_{V\cup\{c\}}} (a) = f_{R_v}(a) \text{ if } c \notin f_{R_v}(a)$$

and hence (calculations omitted) $f_{R_{V\cup\{c\}}} (c) = f_{R_v}(c)$.

This results in the following version of Warshall’s algorithm:

```
| var $f_r : U \rightarrow \mathcal{P}(U)$
| for $a : a \in U \rightarrow f_r(a) := (\text{Set } b : b \in U \land a R b : b)$ rof;
| \{ $f_r = f_R$ \}
| for $c : c \in U \rightarrow$
| \{ $f_r = f_R^+$ \}
```

When using a representation of sets by bitvectors, this version is usually more efficient in practice, as the set operations can be implemented using operations such as bitwise OR and AND, and zero testing.

**Example 2.4.2.** Based on Example 2.4.1, $f_R(A) = \{B\}$, $f_R(B) = \{C\}$, and $f_R(C) = \emptyset$. The algorithm results in $f_r(S) = \{T,U\}$, $f_r(T) = \{U\}$, and $f_r(U) = \emptyset$.

### 2.5 Reachability under $n$-ary relations

In this section we present a solution to the following problem:
Given a finite set $U$, relations $R_1 \subseteq U^{n_1} \times U, \ldots, R_m \subseteq U^{n_m} \times U$ respectively (with $n_1, \ldots, n_m \in \mathbb{N}_+$), and an initial set $U_0 \subseteq U$, determine the subset of $U$ reachable from $U_0$ under (repeated) application of any of the $R_i$ ($1 \leq i \leq m$), i.e. determine the smallest $Z$ such that

1. $U_0 \subseteq Z$ and
2. $(\overrightarrow{a}, b) \in R_i \land \overrightarrow{a} \in Z^{n_i} \Rightarrow b \in Z$ (for $1 \leq i \leq m$, $b \in U$).

Instances of this problem will be encountered in Sections 5.7 and 6.6, where they are used to tabulate reachable states of certain tree automata. Since examples will be presented there, we focus on the presentation of the algorithm and its invariants here, omitting examples.

**Remark 2.5.1.** When considering a single, binary relation $R \subseteq U \times U$, the problem can be and often is formulated as one on graphs. Usually set $U$ and relation $R$ are then called $V$ (for vertices) and $E$ (for edges) respectively. One well-known algorithm to solve this problem uses a breadth-first search and partitions the set $U$ into three sets during execution, consisting of white, grey and black nodes. The algorithm we present below is a generalization from a single binary relation to a set of (not necessarily binary) relations. □

The algorithm presented here works by partitioning set $U$ into three parts, called $W$ (for white), $G$ (for grey) and $Z$ (for ‘zwart’, the Dutch word for black) and consisting of so-called white, grey and black elements. The partitioning is such that

- elements in $G \cup Z$ are in $U_0$ or are reachable using one of the $R_i$ from (i.e. neighbors of) a tuple of elements of $Z$, and
- elements in $W$ are not directly reachable from tuples of elements in $Z$. (Since the three sets form a partitioning of $U$, this implies that for each tuple of elements of $Z$ its neighbors are in $G \cup Z$.)

Formally, we have invariants:

$P0 : U = Z \cup G \cup W \land Z \cap G = G \cap W = W \cap Z = \emptyset,$

$P1 : \forall b : b \in Z \cup G : \left( \exists i : 1 \leq i \leq m \land \overrightarrow{a} \in Z^{n_i} : (\overrightarrow{a}, b) \in R_i \right),$  

$P2 : \left( \forall i : 1 \leq i \leq m : (Z^{n_i} \times W) \cap R_i = \emptyset \right).$

Initializing $Z, G, W$ to $\emptyset, U_0, U \setminus U_0$ establishes $P0 \land P1 \land P2$, while $G = \emptyset \land P0 \land P1 \land P2$ imply the desired postcondition.

Hence $G \neq \emptyset$ becomes the guard of a repetition, in which an element (say $c$) is selected and removed from $G$. To keep $P0$ and $P1$ invariant, $c$ is added to $Z$. We calculate the effect on $P2$:
2.5 Reachability under n-ary relations

\[ P2(G, Z := G \setminus \{c\}, Z \cup \{c\}) \]
\[ = \{ \text{definition of } P2, \text{substitution } \} \]
\[ \langle \forall i : 1 \leq i \leq m : ((Z \cup \{c\})^{n_i} \times W) \cap R_i = \emptyset \rangle \]
\[ = \{ \text{set calculus, distributivity of } \land \text{ over } \forall \} \]
\[ \langle \forall i : 1 \leq i \leq m : (Z^{n_i} \times W) \cap R_i = \emptyset \rangle \]
\[ \land \langle \forall i : 1 \leq i \leq m : (((Z \cup \{c\})^{n_i} \setminus Z^{n_i}) \times W) \cap R_i = \emptyset \rangle \]
\[ = \{ P2 \} \]
\[ \langle \forall i, \bar{q}, d : 1 \leq i \leq m \land \bar{q} \in ZVN \land d \in W : (\bar{q}, d) \notin R_i \rangle \]
\[ = \{ \text{set calculus } \} \]
\[ \langle \forall d : d \in W : \langle \forall i, \bar{q} : 1 \leq i \leq m \land \bar{q} \in ZVN : (\bar{q}, d) \notin R_i \rangle \rangle \]

We therefore choose to remove from \( W \) all elements \( d \) for which a \( j \) and a tuple \( \bar{e} \in (Z \cup \{c\})^{n_j} \setminus Z^{n_j} \) exist with \((\bar{e}, d) \in R_j\), and move them to \( G \). Clearly, this does not invalidate \( P0(G, Z := G \setminus \{c\}, Z \cup \{c\}) \). As for its effect on \( P1(G, Z := G \setminus \{c\}, Z \cup \{c\}) \) we derive (abbreviating \( Z \cup \{c\} \) by \( ZV \)):

\[ P1(G, Z := G \setminus \{c\}, ZV)(W, G := W \setminus \{d\}, G \cup \{d\}) \]
\[ = \{ \text{definition } P1, \text{substitution (twice), } c \neq d \} \]
\[ \langle \forall b : b \in Z \cup G : \langle \exists i, \bar{a} : 1 \leq i \leq m \land \bar{a} \in ZV^{n_i} : (\bar{a}, b) \in R_i \rangle \} \cup\{d\} \]
\[ \lor b \in U_0 \]
\[ = \{ \text{distributivity, one-point-rule } \} \]
\[ \langle \forall b : b \in Z \cup G : \langle \exists i, \bar{a} : 1 \leq i \leq m \land \bar{a} \in ZV^{n_i} : (\bar{a}, b) \in R_i \rangle \} \lor (\exists i, \bar{a} : 1 \leq i \leq m \land \bar{a} \in ZV^{n_i} : (\bar{a}, d) \in R_i) \lor b \in U_0 \]
\[ \land (\langle \exists i, \bar{a} : 1 \leq i \leq m \land \bar{a} \in ZV^{n_i} : (\bar{a}, d) \in R_i \rangle \lor b \in U_0) \]
\[ \leftarrow \{ P1, Z^{n_i} \subseteq ZV^{n_i} \} \]
\[ \langle \exists i, \bar{a} : 1 \leq i \leq m \land \bar{a} \in ZV^{n_i} : (\bar{a}, d) \in R_i \rangle \lor b \in U_0 \]
\[ \leftarrow \{ (\bar{e}, d) \in R_j, \bar{e} \in ZV^{n_j} \setminus Z^{n_j} \subseteq ZV^{n_j} \} \]
\[ \text{true} \]

As a variant function, we have \(|W| + |G|\) i.e. \(|U| - |Z|\). The resulting algorithm is:
Chapter 2 Preliminaries

\[
\begin{align*}
\text{var } c &: U \\
Z, G, W &: := \emptyset, U_0, U \setminus U_0; \\
\{ & \text{ inv } P_0 \land P_1 \land P_2, \text{ vf } |W| + |G| \} \\
do \text{ G } \neq \emptyset \rightarrow \\
& \text{ let } c \in G; \\
& \text{ for } j : 1 \leq j \leq m \rightarrow \\
& \quad \text{ for } \vec{e} : \vec{e} \in (Z \cup \{c\})^{n_j} \setminus Z^{n_j} \rightarrow \\
& \quad \quad \text{ for } d : (\vec{e}, d) \in R_j \land d \in W \rightarrow \\
& \quad \quad \quad W, G := W \setminus \{d\}, G \cup \{d\} \\
& \quad \text{ rof} \\
& \text{ rof}; \\
& G, Z := G \setminus \{c\}, Z \cup \{c\} \\
& \text{ od}
\end{align*}
\]

When considering a set \( V \), initial set \( V_0 \subseteq V \) and binary relation \( E \subseteq V \times V \), the algorithm becomes the familiar one for graphs:

\[
\begin{align*}
\text{var } c &: V \\
Z, G, W &: := \emptyset, V_0, V \setminus V_0; \\
do \text{ G } \neq \emptyset \rightarrow \\
& \text{ let } c \in G; \\
& \text{ for } d : (c, d) \in E \land d \in W \rightarrow \\
& \quad W, G := W \setminus \{d\}, G \cup \{d\} \\
& \text{ rof}; \\
& G, Z := G \setminus \{c\}, Z \cup \{c\} \\
& \text{ od}
\end{align*}
\]
Chapter 3

Regular tree language theory

This chapter introduces the problem domain considered in this dissertation: the domain of regular tree languages on ordered ranked trees. It attempts to provide an overview of regular tree language theory, which will be used in the discussion of algorithmic problems and algorithms in the taxonomy chapters of the dissertation.

To a large extent, regular tree language theory generalizes regular (string) language theory. After the development of a substantial amount of formal language theory for (one-dimensional) string languages in the 1950s and early 1960s, a number of researchers started to look at generalizations and extensions of the string case. One of these was the generalization and extension to trees.

There are many different overviews of and textbooks discussing regular (string) language theory [RS97, HMU01, Lin01], all of which share all or most of the following important elements:

- Strings, languages, and operations on them are introduced.
- Regular sets or languages are defined using finite sets and the union, concatenation, and closure operators.
- Different characterizations of regular languages are given:
  - Regular grammars as a generating formalism for regular languages.
  - Regular expressions as another generating formalism, forming a more compact syntax to represent the regular languages.
  - Finite automata as an accepting mechanism for regular languages.
- Theoretical results on regular languages and their different characterizations are discussed: the equivalence of the different characterizations, closure of the regular languages under a number of operations, and decidability of certain decision problems related to them.
Since regular tree language theory is a generalization of regular (string) language theory, we aim to give an overview of regular tree language theory in a similar way as above:

- Trees, tree languages, and operations on them are introduced in Section 3.1. In addition, that section includes a definition of tree patterns and what it means for such a pattern to match a tree.
- Regular tree languages are defined in Section 3.2.
- Different characterizations of regular tree languages are given in that section and in Sections 3.3 and 3.4:
  - Regular tree grammars are extensively treated in Section 3.3.
  - Finite tree automata as an accepting mechanism for regular tree languages are treated in detail in Section 3.4.
  - In addition, regular tree systems—forming a slightly different generating formalism—and regular tree expressions are briefly touched upon in Section 3.2. No explicit definitions are given, as they do not play a role in most practical applications of regular tree languages. This in contrast to regular (string) expressions, which are often encountered in practical applications (cf. Perl, Python, Ruby, sed, awk), far more often than regular (string) grammars.
- Theoretical results on regular tree languages and their different characterizations are briefly discussed in Section 3.5. We mention results comparing the different characterizations presented in Sections 3.3 through 3.4, closure of the regular tree languages under a number of different operations, and decidability of certain decision problems. We do not discuss proofs and other details, as these are not that important for this dissertation.
- Finally, in Section 3.6, the various links that exist between (regular) string language theory and regular tree language theory are discussed. With the exception of some links that are of particular importance for some algorithms in this dissertation, we only discuss such links briefly, omitting proofs and details.

As indicated, we only give an overview of regular tree language theory here. More details and missing proofs can be found in e.g. [Tha73, Eng75b, GS84, GS97, CDG+07]. A brief, early survey of tree automata theory is given in [Tha73], while [Eng75b] offers a more detailed treatment of regular tree language theory as a whole in the form of lecture notes, despite being over thirty years old. The hard-to-find [GS84] gives an overview of regular tree language theory as a whole up to the early 1980s. A more recent handbook chapter by the same authors [GS97] compresses this book into a single chapter, updating the results as well. The so-called TATA book [CDG+07] is a more recent work on tree automata and their applications (in logic and program verification), but has been in statu nascendi for a number of years now.
Remark 3.0.2. After the first theoretical research on regular tree languages, theoretical researchers soon turned their attention to generalizing and extending this theory as well. This lead to work on e.g. context-free tree language theory [Ron69, Rou70a, Rou70b], tree transducers [Tha69, Tha70, LJ71, Bak73, Eng75a], and pushdown tree automata [Gue81] from the late 1960s onward. An overview of literature and results related to them can be found there and in [Eng75b, GS97, CDG+07], with the latter providing (references to) many more recent results. Since our focus in this dissertation is on regular tree languages—forming the basis for tree acceptors, tree pattern matchers and tree parsers—we do not discuss context-free tree languages any further. The same holds for generalizations of the theory to ( (directed) acyclic) graphs [Roz97].

3.1 Trees and tree languages

In contrast to strings, for which a simple definition suffices, trees require a more elaborate definition. In generalizing from strings to trees, concepts like symbol rank and sibling order can start to play a role. We first define various kinds of trees and operations on them. Given these, we define tree languages and some operations on tree languages.

3.1.1 Trees

In the literature, two definitions of trees frequently appear. One is based on tree domains, the other on a view of trees as terms. We introduce both definitions and three ways of representing trees. For the type of trees we consider, the definitions turn out to be equivalent.

In this dissertation, we focus on ordered, ranked, node labeled trees. We introduce the concepts of a tree domain, node labeling, orderedness, and rankedness sequentially instead of concurrently, since they are more or less independent.

We use \( E \) for a set of edge labels and \( \cdot \) to indicate concatenation of elements of \( E \). Unless explicitly noted otherwise, we assume the edge label set \( E \) to be \( \mathbb{N}_+ \), the positive natural numbers.

Definition 3.1.1 (Tree domain). Given a set of edge labels \( E \), a tree domain is a finite non-empty subset \( D \) of \( E^* \) such that \( \text{pref}(D) \subseteq D \), i.e. \( D \) is prefix-closed (note that \( D \subseteq \text{pref}(D) \) due to Definition 2.3.5 for \( \text{pref} \)). In particular, \( \varepsilon \in D \) for any tree domain \( D \).

The tree domain based notation for trees was introduced in [Gor65]. The intuition behind a tree domain is that it represents the structure of a tree. Note that the above definition defines \( D \) to be non-empty i.e. does not allow for empty trees. This is in contrast to string language theory, where the empty string is often encountered.
The elements of a tree domain are called nodes and are usually denoted using Frakturschrift, e.g. as \( n \), except when explicitly using elements of \( \mathbb{N}_+ \) as edge labels. We use \( \cdot \) to concatenate edge labels. Nodes \( n \) in a tree domain \( D \) such that \( \neg \langle \exists i : i \in E : n \cdot i \in D \rangle \) are called leaf nodes or leaves.

**Example 3.1.2** (Tree domain). Set \( \{ \varepsilon, 1, 3 \cdot 2 \} \) is a tree domain, while sets \( \{ 1, 3 \cdot 2 \} \) and \( \{ \varepsilon, 3 \cdot 2 \} \) are not since they are not prefix-closed. Note that it is not necessary for the example tree domain set to contain a node \( 2 \), though the result would be a (different) tree domain as well.

**Definition 3.1.3** (Tree). Given a tree domain \( D \) and an alphabet \( \Sigma \), a (node labeled) tree \( t \) is a function \( t \in D \rightarrow \Sigma \). We use \( t(n) \) for the label of a node \( n \in D \).

We use \( D_t \) for the tree domain of a tree \( t \). We often do not explicitly mention the underlying tree domain of a tree.

The size of a tree \( t \), denoted \( |t| \), is defined as the number of nodes of the tree, i.e. as \( |D_t| \)—the size of the underlying tree domain.

Before defining rankedness and orderedness formally, we present some example trees and their tree domains.

**Example 3.1.4** (Tree). (Note that \( E = \mathbb{N}_+ \) with order \( < \) forms a totally ordered edge label set with minimal element \( 1 \).)

Set \( \{ (\varepsilon, a), (1, a), (2, c), (3, b), (1 \cdot 1, b), (1 \cdot 2, c), (1 \cdot 1 \cdot 1, c) \} \) forms an ordered, unranked tree: nodes labeled by symbols \( a \) and \( b \) each occur with different numbers of child nodes, so it is not possible to assign a single fixed rank to either symbol. The tree and its domain can be represented graphically as in the left of Figure 3.1.1.

Set \( \{ (\varepsilon, a), (1, c), (2, a), (2 \cdot 1, b), (2 \cdot 2, c), (2 \cdot 1 \cdot 1, c) \} \) forms an ordered, ranked tree, with symbol \( a \) having rank or arity 2, \( b \) having rank 1, and \( c \) having rank 0. This tree and its domain can be represented graphically as in the right of Figure 3.1.1.

**Definition 3.1.5** (Ranked alphabet). A ranked alphabet is a pair \( (\Sigma, r) \) such that \( \Sigma \) is an alphabet (a finite, non-empty set of symbols) and \( r \in \Sigma \rightarrow \mathbb{N} \) is a ranking function. For \( a \in \Sigma \), we call \( r(a) \) the rank or arity of \( a \).

**Notation 3.1.6.** Given a ranked alphabet \( (\Sigma, r) \), we use \( \Sigma_n \) \( (n \geq 0) \) for the symbols
of rank $n$, i.e.\[\Sigma_n = \{\text{Set } a : a \in \Sigma \land r(a) = n : a\}\], we use $r_{\max}$ to denote the maximum rank of the symbols in $\Sigma$, i.e.\[r_{\max} = \{\text{MAX } a : a \in \Sigma : r(a)\}\], and we use $\mathbb{N}_{\leq r}$ for the natural numbers up to and including $r_{\max}$. \hfill \Box

**Definition 3.1.7** (Ranked Tree). A ranked tree is a node labeled tree $t$ whose alphabet is a ranked alphabet $(\Sigma, r)$ and for which, for all $n \in D$,\[r(t(n)) = \{\# i : i \in E \land n \cdot i \in D_t : i\}\], i.e. the rank of the symbol labeling a node corresponds to the number of children of that node. Note that the rank of the symbol labeling a node has nothing to do with the values of the $i$ but merely with the number of $i$ such that $i \in E \land n \cdot i \in D_t$. \hfill \Box

Note that for finite ranked trees over $\Sigma$ to exist, the set of symbols for labeling leaf nodes should be nonempty i.e. $\Sigma_0 \neq \emptyset$ should hold.

**Definition 3.1.8** (Ordered tree domain, ordered tree). A tree domain $D$ is ordered if and only if the underlying edge label set $E$ is well ordered (i.e. has a minimal element and is totally ordered) and, for all $n \in D$ and $i \in E$, $n \cdot i \in D \Rightarrow \{\forall j : j \in E \land j < i : n \cdot j \in D\}$ holds. An ordered tree is a tree $t$ whose tree domain $D_t$ is ordered. \hfill \Box

Note that the definition is stronger than one would normally expect of ordered: it assumes $E$ to be well ordered and requires a list of sibling nodes to be consecutive and to start with the minimal element of $E$. We nevertheless use the terms ordered tree domain and ordered tree, as they are commonly used in the literature.

From this point onward, we only consider ordered, ranked node labeled trees, unless indicated otherwise. We also assume that $\Sigma_0 \neq \emptyset$ for the remainder of this document. In particular, in most examples involving trees, we use alphabet $\{a, b, c, d\}$ with $r(a) = 2$, $r(b) = 1$, and $r(c) = r(d) = 0$. As mentioned before, we assume the edge label set $E$ to be $\mathbb{N}_+$, the positive natural numbers, with minimal element 1. (As we deal with ranked trees, the edge labels in fact have to be from $\mathbb{N}_{\leq r}$.) To avoid confusion we also assume that $\Sigma \cap \mathbb{N}_+ = \emptyset$.

**Convention 3.1.9.** In definitions, lemmas, etc., whenever an unspecified symbol $a$ is used, $n$ represents its rank. \hfill \Box

We denote ordered, ranked trees over alphabet $(\Sigma, r)$ with edge labels from $E$ by $Tr(E, \Sigma, r)$. We use $Tr(\Sigma, r)$ as an abbreviation for $Tr(\mathbb{N}_+, \Sigma, r)$.

**Convention 3.1.10.** We often identify a single symbol $a$ with the single node tree whose node is labeled by that symbol (for all $a \in \Sigma_0$). In cases where we want to refer to such a tree explicitly, we use $\bar{a}$. \hfill \Box
We can also define trees inductively, i.e. as terms over a ranked alphabet.

**Definition 3.1.11 ((Ordered, Ranked) Term).** Set $Te(\Sigma, r)$ is the smallest set satisfying

1. $\Sigma_0 \subseteq Te(\Sigma, r)$
2. $a(t_1, \ldots, t_n) \in Te(\Sigma, r)$ for all $a \in \Sigma \setminus \Sigma_0, t_1, \ldots, t_n \in Te(\Sigma, r)$

It is not hard to see that ordered, ranked terms correspond precisely to ordered, ranked trees with edge labels from $\mathbb{N}_+$. Since we can convert a tree defined using a tree domain to one defined using terms and vice versa, we will use $Tr(\Sigma, r)$ and $Te(\Sigma, r)$ interchangeably without being explicit about this. As a result, we use functions defined on either trees or terms on both, even though we do not explicitly define them for both. Whether the definition for terms or for trees is used depends on which is more suitable in a particular situation.

**Example 3.1.12 ((Ordered, Ranked) Term).** The ordered, ranked tree from Example 3.1.4 corresponds to term $a(c, a(b(c), c))$.

**Remark 3.1.13.** There is a close relation between strings over an alphabet $\Sigma$ and ranked trees over the extension of that alphabet (all whose symbols are assumed to have rank 1) by a termination symbol, say $\#$, of rank 0: a string $w = w_1w_2 \ldots w_n$ corresponds to a tree $w_1(w_2(\ldots(w_n(\#))\ldots))$.

**Definition 3.1.14.** Let $t \in Tr(\Sigma, r)$ and $n \in D_t$, then the pair $(t, n)$ is a **dotted tree**. We use $DottedTr(t)$ to indicate the set of all dotted trees for a tree $t$.

**Example 3.1.15.** Let $u = a(b(c), c)$, then $DottedTr(u) = \{(u, c), (u, 1), (u, 1 \cdot 1), (u, 2)\}$.

The notion of dotted trees rarely appears in publications related to tree automata and algorithms. Grune et al. [GBJL00] use them informally in the context of bottom-up tree pattern matching. Dotted **items** representing tree linearizations sometimes appear in literature on left-to-right tree processing. In the field of Tree Adjoining Grammars, a notion of dotted trees similar to the one used here is used [SVS90, SW93, Sar96]. We will use dotted trees in a particular tree automaton construction in Section 6.7.

### 3.1.1 Functions and operations on trees

**Definition 3.1.16 (//).** (Infix) partial function $/ \in Tr(\Sigma, r) \times \mathbb{N}_{\leq r}^* \rightarrow Tr(\Sigma, r)$ is defined for $t \in Tr(\Sigma, r)$ and $n \in D_t$ by

$$t/n = \langle \text{Set } m, a : (n \cdot m, a) \in t : (m, a) \rangle,$$

i.e. $t/n$ represents the subtree of $t$ starting at node $n$ and is pronounced $t$ at $n$. 

Note that $t/\varepsilon = t$.

Apart from the notations used before, a tree is also uniquely characterized by its set of \textit{stringpaths}, which represent its labeled root to leaf paths.

\textbf{Definition 3.1.17} (Tree stringpaths). Function $\textit{SPaths} \in \textit{Tr}(\Sigma, r) \rightarrow \mathcal{P}((\Sigma \cdot \mathbb{N}_{\leq r})^* \cdot \Sigma)$ is defined for $t \in \textit{Tr}(\Sigma, r)$ by

$$\textit{SPaths}(t) = \{t(\varepsilon)\} \quad \text{if } r(t(\varepsilon)) = 0,$$

$$\textit{SPaths}(t) = \{t(\varepsilon)\} \cdot (\bigcup i : 1 \leq i \leq r(t(\varepsilon)) : \{i\} \cdot \textit{SPaths}(t/i)) \quad \text{if } r(t(\varepsilon)) > 0$$

(where string concatenation operator $\cdot$ is extended to operate on sets of strings).

\textbf{Example 3.1.18.} For $t = a(b(c), a(c))$, $\textit{SPaths}(t) = \{a1b1c, a2a1c, a2a2c\}$ and $\textit{SPaths}(t/2) = \{a1c, a2c\}$.

Related to the definition of stringpaths, we define a function yielding the \textit{rootpath} for a node, i.e. the labeled path from the tree root to the given node:

\textbf{Definition 3.1.19.} Partial function $\textit{RPath} \in \textit{Tr}(\Sigma, r) \times \mathbb{N}_{\leq r}^* \rightarrow (\Sigma \cdot \mathbb{N}_{\leq r})^* \cdot \Sigma$ is defined by

$$\textit{RPath}(t, \varepsilon) = t(\varepsilon),$$

$$\textit{RPath}(t, n \cdot i) = \textit{RPath}(t, n) \cdot i \cdot t(n \cdot i) \text{ for } n \cdot i \in D_t.$$

Note that a rootpath $\textit{RPath}(t, n)$ always ends with symbol $t(n)$ and that stringpaths are rootpaths ending in a symbol of rank 0, i.e. rootpaths to leaf nodes.

\textbf{Definition 3.1.20 (Subtrees).} Function $\textit{Subtrees} \in \textit{Tr}(\Sigma, r) \rightarrow \mathcal{P}(\textit{Tr}(\Sigma, r))$ is defined for $t \in \textit{Tr}(\Sigma, r)$ by

$$\textit{Subtrees}(t) = \langle \text{Set } n : n \in D_t : t/n \rangle.$$

\textbf{Definition 3.1.21 (ProperSubtrees).} Function $\textit{ProperSubtrees} \in \textit{Tr}(\Sigma, r) \rightarrow \mathcal{P}(\textit{Tr}(\Sigma, r))$ is defined for $t \in \textit{Tr}(\Sigma, r)$ by

$$\textit{ProperSubtrees}(t) = \langle \text{Set } n : n \in D_t \setminus \{\varepsilon\} : t/n \rangle.$$

Note that $\textit{ProperSubtrees}(t) = \textit{Subtrees}(t) \setminus \{t\}$.

We will use $\textit{Subtrees}(U)$ and $\textit{ProperSubtrees}(U)$ for $U$ a set of trees as well, as per Convention 2.2.2. Note that for $U$ a set of trees, $\textit{ProperSubtrees}(U) \neq \textit{Subtrees}(U) \setminus U$ may hold: Let $U = \{b(c), c\}$ for example, then $\textit{ProperSubtrees}(U) = \{c\} \neq \emptyset = \textit{Subtrees}(U) \setminus U$.

Having defined trees, we define different kinds of \textit{tree substitution}, of which example instances are depicted in Figure 3.1.2:
• Substitution of a single subtree occurrence, indicated by its root node, by another subtree (see Definition 3.1.22).

• Substitution of all occurrences of a single subtree by occurrences of another subtree (see Definition 3.1.23).

• Concurrent substitution at multiple leaf symbols, where all occurrences of the same leaf symbol are replaced by the same subtree (see Definition 3.1.24). This operation is also called tree concatenation [Eng75b] in the literature. In particular, the restriction to a single leaf symbol is called tree product.

• Substitution of occurrences of a leaf symbol by different subtrees, where all occurrences of a single leaf symbol are replaced by possibly distinct subtrees (see Definition 3.1.25).

**Definition 3.1.22** (Substitution of a single subtree occurrence). Given trees $s, t \in Tr(\Sigma, r)$, $n \in D_t$, the tree substitution in $t$ at node $n$ of subtree $t/n$ by tree $s$, denoted $u = t[t/n \leftarrow s]$ or $u = t[\cdot n \leftarrow s]$, is defined by $D_u = (D_t \setminus (n \cdot D_{t/n})) \cup n \cdot D_s$ and

1. $u(m) = t(m)$ if $m \in D_t \setminus (n \cdot D_{t/n}),$
2. $u(n \cdot l) = s(l)$ if $l \in D_s.$

**Definition 3.1.23** (Substitution of all occurrences of a single subtree). Given $s, t, u \in Tr(\Sigma, r)$, the substitution in $t$ of all occurrences of a tree $u$ by a tree $s$, denoted $t[u \leftarrow s]$, is defined by

1. $s$ if $t = u,$
2. $a$ if $t \neq u$ and $t = a \in \Sigma_0,$
3. $a(t_1[u \leftarrow s], \ldots, t_n[u \leftarrow s])$
   if $t \neq u$ and $t = a(t_1, \ldots, t_n)$ for $a \in \Sigma \setminus \Sigma_0, t_1, \ldots, t_n \in Tr(\Sigma, r).$

**Definition 3.1.24** (Concurrent substitution at leaf symbols, tree concatenation). Given $t \in Tr(\Sigma, r), c_1, \ldots, c_m \in \Sigma_0$ all different, $s_1, \ldots, s_m \in Tr(\Sigma, r),$ the tree substitution of leaf symbols $c_1, \ldots, c_m$ by $s_1, \ldots, s_m$ in $t$, denoted $t[c_1 \leftarrow s_1, \ldots, c_m \leftarrow s_m]$, is defined by

1. $s_i$ if $t = c_i$ for some $i$, $1 \leq i \leq m,$
2. $a$ if $t \neq c_i$ for all $i$ ($1 \leq i \leq m$) and $t = a \in \Sigma_0.$
Figure 3.1.2 Examples of the four different kinds of tree substitution. Situation before substitution shown on the left, after on the right.
3. \[ a(t_1[c_1 \leftarrow s_1, \ldots, c_m \leftarrow s_m], \ldots, t_n[c_1 \leftarrow s_1, \ldots, c_m \leftarrow s_m]) \]
if \( t = a(t_1, \ldots, t_n) \) for \( a \in \Sigma \setminus \Sigma_0, t_1, \ldots, t_n \in \text{Tr}(\Sigma, r) \).

As mentioned, this operation is also called tree concatenation [Eng75b] in the literature. The restriction to substituting a single leaf symbol (called tree product) is also denoted by \( t \cdot s \) instead of \( t[a \leftarrow s] \). We extend these operations to tree languages in the next section.

Notation \( \cdot \) can be seen as a generalization from the string case as well: there, the dot operator \( \cdot \) is used to indicate concatenation by replacing the implicit \( \varepsilon \) occurrence at the right end of one string by another string. Since we deal with trees here, the notion needed to allow replacement of particular leaf symbols.

**Definition 3.1.25** (Substitution of all occurrences of a leaf symbol by different subtrees). Given \( t \in \text{Tr}(\Sigma, r) \) with \( k \) occurrences of a symbol \( a \in \Sigma_0 \) at \( l_1, \ldots, l_k \in D_l \) in lexicographical order, and \( s_1, \ldots, s_k \in \text{Tr}(\Sigma, r) \), the tree substitution in \( t \) of the occurrences in lexicographical order by \( s_1, \ldots, s_k \) respectively, denoted \( u = t[s_1, \ldots, s_k] \), is defined by

\[ D_u = D_t \cup \langle \text{Set } m, i : 1 \leq i \leq k \wedge m \in D_{s_i} : l_i \cdot m \rangle \]

and

1. \( u(n) = t(n) \) if \( n \in D_t \setminus \{l_1, \ldots, l_n\} \),
2. \( u(l_i \cdot m) = s_i(m) \) if \( m \in D_{s_i} \), for all \( i, 1 \leq i \leq k \).

### 3.1.2 Tree pattern matching

We define tree patterns, tree pattern matching, and stringpath matching, which will all play a role in Chapter 6. Tree patterns should be defined to allow them to match inside subject trees, i.e. with their root matching a node possibly different from a subject tree’s root node, and their leaves matching nodes possibly different from a subject tree’s leaves. The former is already possible, and for the latter we extend the alphabet with a special variable or ‘wildcard’ symbol, indicating a match of any tree from \( \text{Tr}(\Sigma, r) \).

**Definition 3.1.26.** Given ranked alphabet \((\Sigma, r)\), ranked alphabet \((\Sigma', r')\) is defined by

\[ \Sigma' = \Sigma \cup \{v\}, \]
\[ r'(v) = 0 \]
and \( r'(a) = r(a) \) for all \( a \in \Sigma \).
Trees in $Tr(\Sigma', r')$ are called tree patterns, pattern trees or patterns. Note that $\nu$ can only label leaf nodes.

We can now define what it means for a subtree of a tree to match a pattern, defining a function $Match$.

**Definition 3.1.27.** Partial function $Match \in Tr(\Sigma', r') \times Tr(\Sigma, r) \times \mathbb{N}^+_t \to \mathbb{B}$ is defined for every pattern $p \in Tr(\Sigma', r')$, subject tree $t \in Tr(\Sigma, r)$ and $n \in \mathbb{N}^+_t$ by $Match(p, t, n) =$

$$n \in D_t \land \langle \exists s_1, \ldots, s_k : s_1, \ldots, s_k \in Tr(\Sigma, r) : p[s_1, \ldots, s_k]^\nu = t/n \rangle,$$

where $p[s_1, \ldots, s_k]^\nu$ is obtained by substitution according to Definition 3.1.25. □

**Example 3.1.28.** Given trees $t = a(b(c), a(b(c), a(c, c)))$ and $p = a(b(c), \nu)$ (depicted in Figure 3.1.3), $Match(p, t, \varepsilon)$ holds for $n = \varepsilon$ and $n = 2$ (and for no other $n$). $Match(p, t, \varepsilon)$ holds since $t/\varepsilon = p[a(b(c), a(c, c))]^\nu$. $Match(p, t, 2)$ holds since $t/2 = p[a(c, c)]^\nu$. □

We call a tree matched by a given tree pattern an instance of the pattern.

We can give a recursive version of $Match$ as well:

**Definition 3.1.29.** Partial function $Match \in Tr(\Sigma, r') \times Tr(\Sigma, r) \times \mathbb{N}^+_t \to \mathbb{B}$ can be defined recursively for every pattern $p \in Tr(\Sigma', r')$, subject $t \in Tr(\Sigma, r)$ and $n \in \mathbb{N}^+_t$ by $Match(p, t, n) =$

$$n \in D_t \land \langle \exists s_1, \ldots, s_k : s_1, \ldots, s_k \in Tr(\Sigma, r) : p[s_1, \ldots, s_k]^\nu = t/n \rangle,$$

where $p[s_1, \ldots, s_k]^\nu$ is obtained by substitution according to Definition 3.1.25. □

**Lemma 3.1.30.** The definitions of $Match$ in Definition 3.1.27 and 3.1.29 are equivalent.

**Proof.** To prove the equivalence of the two definitions, we derive the latter from the former one:

Case $p = \nu$:

$$Match(p, t, n)$$
\[
\equiv \{ \text{Definition 3.1.27}\}
n \in D_t \land (\exists s_1 : s_1 \in Tr(\Sigma, r) : \nu[s_1]^{\nu} = t/n)
\equiv \{ \text{choose } s_1 = t/n\}
n \in D_t
\]

Case \( p = a \):

\[
\begin{align*}
\text{Match}(p, t, n) \\
\equiv \{ \text{Definition 3.1.27}\} \\
n \in D_t \land a = t/n \\
\equiv \{ n = 0 \} \\
n \in D_t \land a = t(n)
\end{align*}
\]

Case \( p = a(p_1, \ldots, p_n) \):

\[
\begin{align*}
\text{Match}(p, t, n) \\
\equiv \{ \text{Definition 3.1.27}\} \\
n \in D_t \land \left( \exists s_1, \ldots, s_k : s_1, \ldots, s_k \in \text{Tr}(\Sigma, r) : a(p_1, \ldots, p_n)[s_1, \ldots, s_k]^{\nu} = t/n \right) \\
\equiv \{ s_1, \ldots, s_k = s_{1,1}, \ldots, s_{1,k_1}, s_{2,1}, \ldots, s_{2,k_2}, \ldots, s_{n,1}, \ldots, s_{n,k_n}; \text{tree equality}\} \\
\land n \in D_t \land a = t(n)
\end{align*}
\]

\[
\equiv \{ \Rightarrow: n \in D_t \land a = t(n) \Rightarrow \langle \forall i : 1 \leq i \leq n : \nu[i] = t/(n \cdot i) \rangle, \text{Def. 3.1.27}; \} \\
\equiv: \text{Definition 3.1.27, weakening}\} \\
n \in D_t \land a = t(n) \land \langle \forall i : 1 \leq i \leq n : \text{Match}(p_i, t, n \cdot i) \rangle
\]

Using the operators \( \| \) and \( \| \) of Definition 2.3.1, we define a function indicating whether a stringpath matches starting at a given subject tree node:

**Definition 3.1.31.** Partial function \( SMatch \in (\Sigma^* \cdot N_{\leq r})^* \cdot \Sigma^* \times \text{Tr}(\Sigma, r) \times N_+^* \rightarrow \mathbb{B} \) is defined for every \( s \in (\Sigma^* \cdot N_{\leq r})^* \cdot \Sigma^* \), subject \( t \in \text{Tr}(\Sigma, r) \) and \( n \in N_+^* \) by \( SMatch(s, t, n) = \)

\[
n \in D_t \land (s \in SPaths(t/n) \lor (s[1 = \nu \land s[1 \in \text{pref}(SPaths(t/n)))))
\]

(\text{where function } \text{pref} \text{ is as in Definition 2.3.5}).
Example 3.1.32. As in Example 3.1.28 (and Figure 3.1.3), for pattern $p = a(b(c), \nu)$ and tree $t = a(b(c), a(b(c), a(c, c)))$, $\text{Match}(p, t, n)$ holds for $n = \varepsilon$ and $n = 2$ only. $\text{Match}(p, t, \varepsilon)$ holds since $a1b1c \in \text{SPaths}(t/\varepsilon)$ and $a2\nu|1 = \nu \land a2\nu|1 \in \text{pref}(\text{SPaths}(t/\varepsilon))$. $\text{Match}(p, t, 2)$ holds since $a1b1c \in \text{SPaths}(t/2)$ and $a2\nu|1 = \nu \land a2\nu|1 = a2 \in \text{pref}(\text{SPaths}(t/2))$.

The following lemma (proof omitted) relates the matching of a tree pattern $p$ at node $n$ of a subject tree $t$ to the matching of the strings from the stringpath set of $p$ starting at node $n$. This well-known result forms the justification for the use of stringpath matching in tree algorithms.

Lemma 3.1.33. Given pattern $p \in \text{Tr}(\Sigma', r')$, subject $t \in \text{Tr}(\Sigma, r)$ and $n \in \mathbb{N}^*_1$,

$$\text{Match}(p, t, n) \equiv \langle \forall s : s \in \text{SPaths}(p) : \text{SPMatch}(s, t, n) \rangle.$$  

We finally define a function indicating whether a rootpath matches starting at a given subject tree node, similar to function $\text{SPMatch}$ for stringpath matches:

Definition 3.1.34. Partial function $\text{RPMatch} \in (\Sigma' \cdot \mathbb{N}_{\leq_r})^{*} \cdot \Sigma' \times \text{Tr}(\Sigma, r) \times \mathbb{N}^*_1 \rightarrow \mathbb{B}$ is defined for every $s \in (\Sigma' \cdot \mathbb{N}_{\leq_r})^{*} \cdot \Sigma'$, subject $t \in \text{Tr}(\Sigma, r)$ and $n \in \mathbb{N}^*_1$ by

$$\text{RPMatch}(s, t, n) = n \in D_t \land (s \in \text{pref}(\text{SPaths}(t/n)) \lor (s|1 = \nu \land s|1 \in \text{pref}(\text{SPaths}(t/n))))$$

(Where function $\text{pref}$ is as in Definition 2.3.5).

The only difference between the definition of $\text{RPMatch}$ and $\text{SPMatch}$ (as in Definition 3.1.31) is in the presence and absence respectively of $\text{pref}$ in the first part of the disjunction in their respective definitions. Note that $\text{SPMatch}(s, t, n) \Rightarrow \text{RPMatch}(s, t, n)$ but not in general vice versa: $\text{RPMatch}(s, t, n) \Rightarrow \text{SPMatch}(s, t, n)$ if and only if $s|1 \in \Sigma_0$ i.e. $s$ ends in a symbol of rank 0.

The definitions and lemma given above will be used in Section 6.7 when discussing tree pattern matching algorithms based on stringpath matching.

### 3.1.3 Tree languages

We define tree languages, which will be used in Section 3.2 to define regular tree languages. Just as string languages are defined as sets of strings, we define tree languages as sets of trees. Some operations on tree languages however are inherently more complicated than corresponding operations on string languages.

Definition 3.1.35. A tree language is a subset of $\text{Tr}(\Sigma, r)$. The set of all tree languages, $\mathcal{P}(\text{Tr}(\Sigma, r))$, is denoted by $\text{TL}$.
Since this dissertation is mainly concerned with trees (versus strings), we use the terms ‘tree language’ and ‘language’ interchangeably. Whenever we talk about string languages, we explicitly call them so if not doing so could cause confusion.

As tree languages are subsets of $\text{Tr}(\Sigma, r)$, we can use the well-known set operators—such as union ($K \cup L$), intersection ($K \cap L$) and complement ($\neg L$)—to construct new tree languages. We now define concatenation for tree languages as a straightforward extension of the definition of tree concatenation given previously. We first give a definition of substituting tree languages in a single tree, and then use this to define the substitution of tree languages in a tree language. Before we can do so, the following definition is needed.

**Definition 3.1.36.** Let $a \in \Sigma_n$, $L_1, \ldots, L_n \in TL$, then

$$a(L_1, \ldots, L_n) = \langle \text{Set} \ t_1, \ldots, t_n : t_1, \ldots, t_n \in L_1, \ldots, L_n : a(t_1, \ldots, t_n) \rangle.$$  

\[\square\]

**Definition 3.1.37.** Given $t \in \text{Tr}(\Sigma, r)$, $c_1, \ldots, c_m \in \Sigma_0$ all different, $L_1, \ldots, L_m \in TL$, the language substitution or language concatenation in tree $t$ of $c_1, \ldots, c_m$ by $L_1, \ldots, L_m$, denoted $t(c_1 \leftarrow L_1, \ldots, c_m \leftarrow L_m)$, is a tree language defined by

1. $L_i$ if $t = c_i$ for some $i$, $1 \leq i \leq m$,
2. $\{a\}$ if $t = a \in \Sigma_0$ yet $a \neq c_i$ for all $i$, $1 \leq i \leq m$,
3. $a(t_1(c_1 \leftarrow L_1, \ldots, c_m \leftarrow L_m), \ldots, t_m(c_1 \leftarrow L_1, \ldots, c_m \leftarrow L_m))$

if $t = a(t_1, \ldots, t_n)$ for some $a \in \Sigma \setminus \Sigma_0, t_1, \ldots, t_n \in \text{Tr}(\Sigma, r)$.

\[\square\]

**Definition 3.1.38.** Given $L \in TL$, $c_1, \ldots, c_m \in \Sigma_0$ all different, $L_1, \ldots, L_m \in TL$, the language substitution or language concatenation in $L$ of $c_1, \ldots, c_m$ by $L_1, \ldots, L_m$, denoted $L(c_1 \leftarrow L_1, \ldots, c_m \leftarrow L_m)$, is defined by

$$\left\langle \bigcup t : t \in L : t(c_1 \leftarrow L_1, \ldots, c_m \leftarrow L_m) \right\rangle.$$  

\[\square\]

**Remark 3.1.39.** The language concatenation as defined is ‘nondeterministic’: different elements of a language may be substituted at different occurrences of a symbol, i.e.

$$t(c_1 \leftarrow L_1, \ldots, c_m \leftarrow L_m) 
\neq \langle \text{Set} \ s_1, \ldots, s_m : s_1, \ldots, s_m \in L_1, \ldots, L_m : t(c_1 \leftarrow s_1, \ldots, c_m \leftarrow s_m) \rangle.$$  

One could also define ‘deterministic’ tree language concatenation [Eng75b, Remark 2.28].
3.1 Trees and tree languages

We also denote \( L(a \leftarrow K) \) by \( L \cdot aK \), for each \( a \in \Sigma_0 \) and \( K, L \in TL \).

When it is clear from the context which form is meant, we use substitution (concatenation) as a shorthand for tree substitution (concatenation), language substitution (concatenation) in a single tree, or language substitution (concatenation).

Now that we have defined concatenation, we can define language exponentiation and closure, which we need to define the regular tree languages in the next section.

**Definition 3.1.40** (Language Exponentiation). Given \( L \in TL, a \in \Sigma_0 \) and \( k \geq 0 \), \( L^k \in TL \) is defined recursively by

1. \( L^0 = \{a\} \) and
2. \( L^{(m+1)} = L^m \cdot_a (L \cup \{a\}) \) for \( m \geq 0 \).

Note that the second part of this definition might be different from what one would expect based on the definition of string language exponentiation. Defining \( L^{m+1} = L^m \cdot aL \) for \( m \geq 0 \) would mean that the exponentiation takes place at *every* occurrence of \( a \). Our definition allows the exponentiation to take place at any number of occurrences of \( a \), not necessarily all of them. We will give an example after defining language closure.

**Definition 3.1.41** (Language Closure). Given \( L \in TL \) and \( a \in \Sigma_0 \), \( L^* \in TL \) is defined by

\[
L^* = \left\{ \bigcup m : m \geq 0 : L^m \right\}.
\]

Note that defining \( L^* \) in a similar way is not useful, since it equals \( L^0 \subseteq L^1 \) (due to the second part of Definition 3.1.40).

**Example 3.1.42** (Language Exponentiation and Closure). Let \( L = \{a(c, c)\} \). Using Definition 3.1.40,

\[
\begin{align*}
L^0 &= \{c\}, \\
L^1 &= \{c, a(c, c)\}, \\
L^2 &= \{c, a(c, c), a(a(c, c), c), a(c, a(c, c)), a(a(c, c), a(c, c))\},
\end{align*}
\]

giving \( L^* = \{c, a(c, c), a(a(c, c), c), a(c, a(c, c)), a(a(c, c), a(c, c)), \ldots\} \).

If we replace \( L^{(m+1)} = L^m \cdot_a (L \cup \{a\}) \) by \( L^{(m+1)} = L^m \cdot aL \), we get

\[
\begin{align*}
L^0 &= \{c\}, \\
L^1 &= \{a(c, c)\}, \\
L^2 &= \{a(a(c, c), a(c, c))\},
\end{align*}
\]

giving \( L^* = \{c, a(c, c), a(a(c, c), a(c, c)), \ldots\} \). At least intuitively, this would not be considered a regular language.
Definition 3.1.43. Given \( L \in TL \), the path closure of \( L \), denoted \( \text{PathCl}(L) \), is defined by

\[
t \in \text{PathCl}(L) \equiv \text{SPaths}(t) \subseteq \left( \bigcup u : u \in L : \text{SPaths}(u) \right).
\]

We call \( L \in TL \) such that \( L = \text{PathCl}(L) \) a path closed language.

Example 3.1.44. Language \( L_1 = \{a(b,c), a(c,b), a(b,b), a(c,c)\} \) is path closed, while \( L_2 = \{a(b,c), a(c,b)\} \) is not.

The path closure of a language will play a role in Section 3.4.3.

3.2 Regular tree languages

Regular string languages are often defined using finite sets and the union, concatenation and closure operators. We can define regular tree languages in a similar fashion:

Definition 3.2.1 (Regular Tree Languages). The set of regular tree languages over alphabet \((\Sigma, r)\), denoted \( RTL(\Sigma, r) \) or \( RTL \) for short, is defined as the smallest set such that

1. \( \emptyset \in RTL \),
2. \( \{a\} \in RTL \) for all \( a \in \Sigma_0 \),
3. \( a(L_1, \ldots, L_n) \in RTL \) for all \( a \in \Sigma \setminus \Sigma_0 \), \( L_1, \ldots, L_n \in RTL \) (see Definition 3.1.36 for \( a(L_1, \ldots, L_n) \)),
4. \( L_1 \cup L_2 \in RTL \) for all \( L_1, L_2 \in RTL \),
5. \( L_1 \cdot aL_2 \in RTL \) for all \( L_1, L_2 \in RTL \) and \( a \in \Sigma_0 \),
6. \( L^{*a} \in RTL \) for all \( L \in RTL \) and \( a \in \Sigma_0 \).

As in the string case, multiple formalisms exist for generating regular tree languages. In the next section, we treat regular tree grammars in detail. We then treat finite tree automata as an accepting formalism. These two formalisms are most important for the applications and underlying algorithmic problems that are the topic of this dissertation.

Other generating formalisms exist, such as regular tree systems and regular tree expressions. Regular tree systems are a generalization of regular tree grammars that appears in the literature [Bra69, Bra67]. In regular tree systems, multiple axioms
may exist instead of a single start nonterminal, nonterminals may have rank greater
than 0, and a production LHS may be a tree instead of a nonterminal. In those two
publications and in [Din87] it is shown that such systems are in fact equivalent in
generating power to regular tree grammars.

(Context-free tree grammars—not considered here—are more powerful, allowing pro-
duction left hand sides with variables, for which any tree can be substituted [Rou69,
Rou70a, Rou70b].)

As regular tree systems do not appear in the applications of regular tree language
theory we consider, they are not treated any further in this dissertation. Similarly,
regular tree expressions can be defined [CDG+07], in a way similar to the definition
of regular (string) expressions. As they are seldom treated and scarcely used in
(applications of) regular tree language theory, they are not treated here.

After discussing regular tree grammars and finite tree automata in detail, we returns
to the definition of regular tree languages in Section 3.5, where we consider the
equivalence of the regular tree languages and the languages that can be generated
and accepted by these formalisms. There, we also briefly discuss closure properties
and decision problems for regular tree languages.

### 3.3 Regular tree grammars

Regular tree grammars play an important role in the algorithmic problems we con-
sider in Part II of this dissertation. We define regular tree grammars, derivation steps
and derivations, and the languages generated by such grammars. We give examples
and show how such grammars can be characterized based on certain properties of
their production rules.

**Definition 3.3.1 (RTG, Regular Tree Grammar).** A regular tree grammar (RTG) $G$
is a 5-tuple $(N, \Sigma, r, Prods, S)$ such that

- $N$ is an alphabet, the **nonterminals**, 
- $\Sigma$ is an alphabet, the **terminals**, 
- $N \cap \Sigma = \emptyset$, 
- $(N \cup \Sigma, r)$ is a ranked alphabet with $N = N_0$ (i.e. nonterminals have rank 0
  and thus may only appear as leaf symbols), 
- $Prods \subseteq N \times Tr(N \cup \Sigma, r)$, the finite set of **(production) rules or productions**, and 
- $S \in N_0$, the start symbol.
This definition of RTGs corresponds to the one used most frequently in the literature [Eng75b, GS97, WM95, CDG+07]. An example of an RTG will be given further on as Example 3.3.8.

We use LHS and RHS as an abbreviation for left hand side and right hand side (of a production), and use RHS(Prods) to indicate the set of right hand sides of productions in Prods.

**Remark 3.3.2.** Recall from e.g. [HMU01, Lin01] that regular string grammars in right-linear form have productions of the form $A \rightarrow a, A \rightarrow a \ldots bB$ or $A \rightarrow B$. Productions in regular tree grammars form the generalization to trees: their LHSs are nonterminals, while nonterminals can only appear at leaves of RHSs. As mentioned when briefly discussing regular tree systems in Section 3.2, the restriction of the productions in such a way is not necessary for a grammar to be equivalent to an RTG.

**Definition 3.3.3** ($\Rightarrow$, Derivation Step). Given a grammar $G$, for all $\beta_1, \beta_2 \in \text{Tr}(N \cup \Sigma, r)$:

- $\beta_1 \xrightarrow{A \Rightarrow \beta_2} \beta_2$ means $\beta_1(n) = A$ and $\beta_2 = \beta_1[A \Rightarrow \alpha]$ for all $n \in D_{\beta_1}, A \rightarrow \alpha \in \text{Prods}$. $\xrightarrow{A \Rightarrow \beta_2}$ is pronounced “derives (in one step) by replacement at $n$ using $A \rightarrow \alpha$.”

- $\beta_1 \Rightarrow \beta_2$ means there exists an $A \rightarrow \alpha \in \text{Prods}$ such that $\beta_1 \xrightarrow{A \Rightarrow \beta_2} \Rightarrow$ is pronounced “derives (in one step) by replacement at $n$”.

- $\Rightarrow$ is the reflexive and transitive closure of $\Rightarrow$ and $\Rightarrow$, and are pronounced “derives in one or more steps (by replacement at $n$).”

- $\Rightarrow$ are the reflexive and transitive closure of $\Rightarrow$ and $\Rightarrow$, and are pronounced “derives in zero or more steps (by replacement at $n$).”

(Notations $\Rightarrow$ and $\Rightarrow$ may not seem useful, but will be used in Lemma 3.3.39.)

**Definition 3.3.4.** Let $G$ be an RTG. A derivation in $G$ is a sequence $\alpha_0, \ldots, \alpha_m \in \text{Tr}(N \cup \Sigma, r)^*$ such that for all $i, 0 \leq i < m$, $\alpha_i \Rightarrow \alpha_{i+1}$.

**Lemma 3.3.5.** Let $G$ be an RTG, $\alpha \in \text{Tr}(N \cup \Sigma, r)$ and $t = a(t_1, \ldots, t_n) \in \text{Tr}(\Sigma, r)$, then

$$\alpha \Rightarrow t \equiv \exists \beta : \beta \in \text{Tr}(N \cup \Sigma, r) \land \beta(\varepsilon) = a : \alpha \Rightarrow \beta \land \beta \Rightarrow t.$$
3.3 Regular tree grammars

Proof idea. Clearly a tree $\beta \in Tr(N \cup \Sigma, r)$ such that $\beta(\varepsilon) = a$ and $\beta \Rightarrow t$ exists: $t$ is such a tree. Productions can only replace a nonterminal by another nonterminal or by a tree with a terminal at its root, so any steps in a derivation before the occurrence of the first such intermediate tree $\beta$ have to take place at $\varepsilon$. □

Definition 3.3.6. For RTG $G$ and $\alpha \in Tr(N \cup \Sigma, r)$,

$$L_{RTG}(G, \alpha) = \left\{ \text{Set } t : t \in Tr(\Sigma, r) \land \alpha \Rightarrow t \right\}.$$

□

Definition 3.3.7. The language of RTG $G$ is defined by

$$L_{RTG}(G) = L_{RTG}(G, S).$$

When it is clear that $G$ is an RTG, we often use $L(G)$ to denote $L_{RTG}(G)$. □

Example 3.3.8 (RTGs and their languages). Let $N = N_0 = \{S, A\}$. Then $G_1 = (N, \Sigma, r, \text{Prods}_1, S)$ is a regular tree grammar with production rules $\text{Prods}_1 = \{ S \rightarrow b(a(c, c)), \ S \rightarrow b(S) \}$ whose language is

$$L_{RTG}(G_1) = L = \{ b(a(c, c)), \ b(b(a(c, c))), \ b(b(b(a(c, c)))) \ldots \}.$$

For example, $S \Rightarrow b(S) \Rightarrow b(b(a(c, c)))$.

Let $G_2 = (N, \Sigma, r, \text{Prods}_2, S)$ where $\text{Prods}_2 = \{ S \rightarrow b(a(c, c)), \ S \rightarrow A, \ A \rightarrow b(A), \ A \rightarrow b(a(c, c)) \}$, then $L_{RTG}(G_2) = L$. □

In example grammar $G_2$, there are two essentially different derivation sequences generating the same tree: $S \Rightarrow A \Rightarrow b(a(c, c))$ and $S \Rightarrow b(a(c, c))$. This indicates that tree grammars, just as string grammars, may be ambiguous, i.e. multiple derivations may exist which cannot be transformed into one another by merely reordering derivation steps.

Definition 3.3.9 (Grammar equivalence). Grammars $G$ and $H$ are called equivalent if and only if $L(G) = L(H)$. We use $G \equiv H$ to denote that $G$ and $H$ are equivalent. □

3.3.1 Grammar characteristics

Although our definition of RTGs already restricts production rules to the form $A \rightarrow \alpha$, it is useful to be able to characterize such RTGs in somewhat more detail. We introduce grammar characteristics for this purpose:

Definition 3.3.10 (Grammar Characteristics). We use the following grammar characteristics, which can be either present (indicated by $+$) or absent $(-)$ in a certain RTG.
• \( z \) indicates whether non-root nodes in productions may be labeled by any symbol \((z_+)\), or by a nonterminal only \((z_-)\). Non-root nodes labeled by terminals are called Z-nodes. For type \((z_-)\) grammars, productions are restricted to the form \( A \rightarrow a(A_1, \ldots, A_n) \) (in particular, for \( n = 0, A \rightarrow a \)) or \( A \rightarrow B \).

• \( u \) indicates whether it is possible \((u_+)\) or not \((u_-)\) that the root node of the RHS of the same production are labeled by a nonterminal (i.e. whether unit productions are possible or not). For type \((u_-)\) grammars, productions are restricted to the form \( A \rightarrow a(t_1, \ldots, t_n) \) (in particular, for \( n = 0, A \rightarrow a \)).

Using these grammar characteristics, we can describe various RTG types that either explicitly appear in the literature or appear as variations of such types:

• Type \((z_+ u_+)\) grammars are called Regular Tree Grammars in [CDG⁺07] and Tree Normal Form (TNF) grammars in [Din87]. This grammar type does not restrict the production rule format of Definition 3.3.1. Production rules have the form \( A \rightarrow a \), i.e. (due to nonterminals having rank 0) \( A \rightarrow a(t_1, \ldots, t_n) \) (in particular, for \( n = 0, A \rightarrow a \)) or \( A \rightarrow B \).

• Type \((z_+ u_-)\) grammars are the TNF grammars without Unit Productions. Rules have the form \( A \rightarrow a(t_1, \ldots, t_n) \) (in particular, for \( n = 0, A \rightarrow a \)).

• Type \((z_- u_-)\) grammars, called Expansive Tree Grammars (ETGs) in [Bra67, Din87], Tree Grammars in Normal Form in [Eng75b], and Normalized Regular Tree Grammars in [CDG⁺07]. Rules have the form \( A \rightarrow a(A_1, \ldots, A_n) \) (in particular, for \( n = 0, A \rightarrow a \)).

• Type \((z_- u_+)\) grammars are ETGs allowing Unit Productions. Rules have the form \( A \rightarrow a(A_1, \ldots, A_n) \) (in particular, for \( n = 0, A \rightarrow a \)) or \( A \rightarrow B \).

**Example 3.3.11** (Regular Tree Grammar Types). Let \( N = N_0 = \{ S, A, B, C \} \). We consider \( G = (N, \Sigma, r, \text{Prods}, S) \) for various production rule sets \( \text{Prods} \), such that \( \mathcal{L}(G) = L \) as in Example 3.3.8.

• \( \text{Prods} = \{ S \rightarrow b(a(c, c)), S \rightarrow A, A \rightarrow b(A), A \rightarrow b(a(c, c)) \} \) results in a type \((z_+ u_+)\) grammar, i.e. a grammar in TNF. This is the second grammar of Example 3.3.8.

• \( \text{Prods} = \{ S \rightarrow b(a(c, c)), S \rightarrow b(S) \} \) results in a type \((z_+ u_-)\) grammar, i.e. a grammar in TNF without unit productions. This is the first grammar of Example 3.3.8.

• \( \text{Prods} = \{ S \rightarrow b(A), A \rightarrow b(A), A \rightarrow a(B, B), B \rightarrow c \} \) results in a type \((z_- u_-)\) grammar, i.e. an ETG.
• \( \text{Prods} = \{ S \rightarrow b(A), A \rightarrow b(A), A \rightarrow B, B \rightarrow a(C, C), C \rightarrow c \} \) results in a type \((z_-u_+)\) grammar, i.e. an RTG allowing unit productions.

Now that we have defined RTGs and grammar characteristics, we discuss three transformations on RTGs. The first of these removes useless symbols and productions from an RTG, while the other two restrict the form of a grammar by removing productions that violate characteristic value \(z_-\) or \(u_-\).

### 3.3.2 Removing useless symbols and productions

It may happen that an RTG contains so-called useless symbols and productions, just as e.g. a regular or context-free string grammar may. In cases where the original RTG does not contain them, such symbols and productions may be introduced by applying one of the other two grammar transformations (discussed in Section 3.3.3). The presence of useless symbols and productions may complicate proofs or influence the efficiency of certain algorithms operating on grammars. We therefore define what useless symbols and productions are and present a transformation for removing them from an RTG.

**Definition 3.3.12** (Start-reachable symbol). Let \( G \) be an RTG. A symbol \( X \in N \cup \Sigma \) is **start-reachable** if and only if

\[
\left\{ \exists \alpha : \alpha \in \text{Tr}(N \cup \Sigma, r) \land S \xrightarrow{\alpha} \alpha : X \in \alpha(D_\alpha) \right\}
\]

(Note that \( \alpha(D_\alpha) \) denotes the set of all node labels of tree \( \alpha \), using Convention 2.2.2 to generalize the tree function \( \alpha \) from a single node to a set of nodes, here the entire domain \( D_\alpha \) of \( \alpha \).)

**Definition 3.3.13** (Productive nonterminal, productive terminal). Let \( G \) be an RTG. A nonterminal \( B \in N \) is **productive** if and only if

\[
\left\{ \exists t : t \in \text{Tr}(\Sigma, r) : B \xrightarrow{t} t \right\}
\]

All terminals are productive (assuming \( \Sigma_0 \neq \emptyset \)).

**Definition 3.3.14** (Useful, useless symbol). Let \( G \) be an RTG. A symbol \( X \in N \cup \Sigma \) is **useful** if and only if

\[
\left\{ \exists \alpha, t : \alpha \in \text{Tr}(N \cup \Sigma, r) \land X \in \alpha(D_\alpha) \land t \in \text{Tr}(\Sigma, r) : S \xrightarrow{\alpha} \alpha \xrightarrow{t} \right\}
\]

A symbol is **useless** if and only if it is not useful.
Remark 3.3.15. As for context-free string grammars, useful symbols are both start-reachable and productive, but start-reachable and productive ones are not necessarily useful: for a productive and start-reachable symbol $X$, it may be the case that every tree $\alpha$ as in the $\exists$-quantification of Definition 3.3.14 contains one or more (other) nonterminal symbols which are not productive, causing $X$ to be useless as well.

Definition 3.3.16 (Useful, useless production). Given an RTG $G$, a production $A \to \alpha \in \text{Prods}$ is useful if and only if all symbols occurring in its LHS and RHS are useful. A production is useless if and only if it is not useful.

Any RTG which has useless symbols and productions may be transformed into an equivalent RTG without such symbols and productions. The approach we use for this is very similar to the one for context-free string grammars, e.g. as treated in [HMU01, Theorem 7.2] (in fact, the proof of Lemma 3.3.21 is similar to the proof of that theorem given in [HMU01]).

We will show how to compute the start-reachable symbols and productive nonterminals of an RTG, and how these computations can be used for such a transformation.

Definition 3.3.17. Given an RTG $G = (N, \Sigma, r, \text{Prods}, S)$, set $S\text{Reach} \subseteq N \cup \Sigma$ is the smallest set satisfying

$$S\text{Reach} = \{ S \} \cup \left\{ \bigcup B, \beta : B \to \beta \in \text{Prods} \land B \in S\text{Reach} : \beta(D_\beta) \right\} \tag{1}$$

Definition 3.3.18. Given an RTG $G = (N, \Sigma, r, \text{Prods}, S)$, set $\text{Productive} \subseteq N$ is the smallest set satisfying

$$\text{Productive} = \left( \bigcup A, \alpha : A \to \alpha \in \text{Prods} \land \alpha(D_\alpha) \cap N \subseteq \text{Productive} \right) \tag{2}$$

Note that recursive production rules (i.e. those in which the LHS nonterminal also appears in the RHS tree) are handled properly by the definitions. For RTG $G$, we use $\text{Productive}_G$ and $S\text{Reach}_G$ to indicate the corresponding sets $\text{Productive}$ and $S\text{Reach}$.

Property 3.3.19. Sets $S\text{Reach}$ and $\text{Productive}$ can be computed by computing the sequences of successive approximations $S\text{Reach}_n$ ($\text{Productive}_n$) for increasing $n$, starting with $n = 0$ and ending as soon as $S\text{Reach}_{n+1} = S\text{Reach}_n$ ($\text{Productive}_{n+1} = \text{Productive}_n$)
3.3 Regular tree grammars

Productive, where SReach and Productive are defined by:

\[
\begin{align*}
SReach_0 &= \{S\} \\
Productive_0 &= \emptyset \\
SReach_{n+1} &= SReach_n \\
&\quad \cup \left( \bigcup B, \beta : B \rightarrow \beta \in P \land B \in SReach_n : \beta(D_\beta) \right) \\
Productive_{n+1} &= Productive_n \\
&\quad \cup \left( \bigcup A, \alpha : A \rightarrow \alpha \in P \land \alpha(D_\alpha) \cap N \subseteq Productive_n : \{A\} \right)
\end{align*}
\]

Note that the above uses the standard approach for computing least fixed points by approximation from below, using monotonicity and finiteness of the sequences SReach and Productive.

**Transformation 3.3.20** (rem-useless, removing useless productions and symbols). Let \( G = (N, \Sigma, r, Prods, S) \) be an RTG. Assuming that \( S \) is productive—i.e. that \( \mathcal{L}_{\text{RTC}}(G) \neq \emptyset \), the following procedure can be used to transform \( G \) into a grammar \( G_2 \):

1. Construct \( G_1 = (Productive_G, \Sigma, r_1, Prods_1, S) \) where \( Prods_1 \) is the set of rules from \( Prods \) with nonterminals from \( Productive_G \) only and \( r_1 \) is the restriction of function \( r \) to symbols in \( Productive_G \cup \Sigma \).

2. Construct \( G_2 = (SReach_G \cap N, SReach_G \cap \Sigma, r_2, Prods_2, S) \) where \( Prods_2 \) is the set of rules from \( Prods_1 \) with symbols from \( SReach_G \) only and \( r_2 \) is the restriction of function \( r_1 \) to symbols in \( SReach_G \cap (N \cup \Sigma) \).

**Lemma 3.3.21.** For \( G \) and \( G_2 \) as in Transformation rem-useless, \( \mathcal{L}(G) = \mathcal{L}(G_2) \) and \( G_2 \) contains no useless symbols and productions.

**Proof.** Consider \( G_1 \) as resulting from step 1 of the transformation. Clearly, \( \mathcal{L}(G) = \mathcal{L}(G_1) \) since the only difference between \( G \) and \( G_1 \) is in removal of non-productive nonterminals and removal of productions they occur in, and \( \mathcal{L}(G_1) = \mathcal{L}(G_2) \) since the only difference between \( G_1 \) and \( G_2 \) is in removal of symbols that are not start-reachable and removal of productions involving such symbols. Hence we have \( \mathcal{L}(G) = \mathcal{L}(G_2) \).

Assume that \( G_2 \) contains a useless symbol or production, then it contains a useless symbol, say \( X \). By step 2 of the transformation, and using the definition of \( SReach \), there exists a derivation \( S \Rightarrow_{G_2} \alpha \) with \( X \in \alpha(D_\alpha) \). Since \( SReach_G \cap N \subseteq Productive_G \) and \( SReach_G \cap \Sigma \subseteq \Sigma \), we have \( S \Rightarrow_{G_1} \alpha \) and (by step 1 and using the definition of \( Productive \)) we have \( S \Rightarrow_{G_1} \alpha \Rightarrow_{G_1} t \) for some \( t \in Tr(\Sigma, r) \). Hence no
symbol in the derivation of \( \alpha \Rightarrow_{G_1} t \) is eliminated by step 2. Symbol \( X \) thus occurs in a derivation of a terminal tree in \( G_2 \), hence it is not useless.

**Example 3.3.22.** Let \( G = (N, \Sigma, r, \text{Prods}, S) \) with \( N = \{S, A, B\} \) and \( \text{Prods} = \{S \to A, S \to c, A \to a(A, B), B \to c\} \) be an RTG. Applying transformation \text{REM-USELESS} results in grammar

\[
\{(S), \{c\}, \{(c, 0)\}, \{S \rightarrow c\}, S\},
\]

where unused ranks have also been removed. Note that the only production with LHS \( A \) is recursive in \( A \), making \( A \) useless. As a result, \( B \) is useless as well.

**Remark 3.3.23.** Note that the order of the steps in the above transformation indeed is significant; applying the steps the other way around does not give the intended result. Instead, it results in grammar

\[
\{(S, B), \{c\}, \{(c, 0)\}, \{S \rightarrow c, B \rightarrow c\}, S\},
\]

which still contains a useless production and nonterminal.

**Convention 3.3.24** (Absence of useless symbols and productions). Now that we have given the above transformation, for each RTG \( G = (N, \Sigma, r, \text{Prods}, S) \), we assume \( N \) and \( \Sigma \) not to contain useless symbols and \( \text{Prods} \) not to contain useless productions. This simplifies our discussion of various transformations and proofs related to RTGs. In [CDG+07], such an RTG is called \textit{reduced}.

**Remark 3.3.25.** Since \text{REM-USELESS} \textit{removes} productions but does not add new ones, no chain rules or \( Z \)-nodes can be introduced by it. As a result, characteristic values \( u_-, u_+, z_- \) and \( z_+ \) are invariant under the transformation \text{REM-USELESS}. Note that it may reduce the number of \( Z \)-nodes and unit productions of grammars with characteristic value \( z_+ \) and \( u_+ \) to zero.

### 3.3.3 Removing \( z_- \) and \( u_- \) violating productions

Based on the tree grammar characteristics \( z \) and \( u \), transformations can be given to remove productions that cause an RTG not to have characteristic value \( z_- \) and \( u_- \) respectively: productions whose RHSs contain \( Z \)-nodes, and unit productions. We first describe \textit{transformation steps} to remove a single \( Z \)-node and a single unit production respectively. These steps can then be applied in a transformation as often as necessary to ensure an RTG has characteristic value \( z_- \) or \( u_- \).

In the discussion of the transformation steps, we use a single variant function to show the termination of a sequence of such steps, regardless of their order.

**Definition 3.3.26** (Variant function). We use the pair \((n_U, n_Z)\) as a variant function, where

- \( n_U = (\# A, B : A \rightarrow B \in \text{Prods}) \), i.e. \( n_U \) is the number of unit productions in the grammar,
3.3 Regular tree grammars

- \( n_Z = \langle \# A, \alpha, n : A \to \alpha \in \text{Prods} \land n \in D_\alpha \setminus \{\varepsilon\} : \alpha(n) \in \Sigma \rangle \), i.e. \( n_Z \) is the number of Z-nodes in the grammar,

and we use the standard lexicographical ordering on such pairs.

Example 3.3.27. For \( \text{Prods} = \{S \to c\} \), \( n_Z = 0 \) as there are no non-root terminal nodes, while for \( \text{Prods} = \{S \to a(b(c), B)\} \), \( n_Z = 2 \) since there are two non-root terminal nodes, labeled by \( b \) and \( c \) respectively.

We use the following RTG as a running example in this chapter. It is the same grammar used as a running example in [Din87].

Example 3.3.28 (Running example). Let \( G = (N, \Sigma, r, \text{Prods}, S) \) where

- \( N = \{S, B\} \),
- \( \Sigma = \{a, b, c, d\} \),
- \( r = \{(S, 0), (B, 0), (a, 2), (b, 1), (c, 0), (d, 0)\} \), and
- \( \text{Prods} = \{(1) S \to a, \ (2) S \to a, \ (3) S \to c, \ (4) B \to b, \ (5) B \to S, \ (6) B \to d\} \).

Note that we have numbered the production rules for ease of reference. Grammar \( G \) has characteristic values \( z_+ \) and \( u_+ \) and variant function \( (n_U, n_Z) = (1, 3) \).

Transformation step 3.3.29 (Red-Z-naive, Reducing the number of Z-nodes). Let \( G = (N, \Sigma, r, \text{Prods}, S) \) be an RTG with characteristic value \( z_+ \). Then there must be a production \( A \to \alpha \in \text{Prods} \) and an \( n \in D_\alpha \setminus \{\varepsilon\} \) such that \( \alpha(n) \in \Sigma \). Then \( G' = (N \cup \{A_F\}, \Sigma, r, \text{Prods}', S) \) where

- \( A_F \) is a fresh nonterminal (i.e. \( A_F \notin N \)) such that \( r(A_F) = 0 \) (i.e. \( A_F \) has rank 0) and
- \( \text{Prods}' = \text{Prods} \setminus \{A \to \alpha\} \cup \{A \to \alpha / \alpha / n \leftarrow A_F, A_F \to \alpha / n\} \)

is the resulting transformed grammar.

Informally, a Z-node is selected, and the subtree occurrence rooted at this node is removed by introducing a fresh nonterminal, replacing the production by one in which that subtree occurrence has been replaced by the nonterminal, and adding a production to replace the nonterminal by the subtree. Note that the selection of \( A \to \alpha \) and \( n \in D_\alpha \setminus \{\varepsilon\} \) with \( \alpha(n) \in \Sigma \) is nondeterministic.
Lemma 3.3.30. For $G$ and $G'$ as in transformation step RED-Z-NAIVE, $L(G) = L(G')$.

Proof. An application in a derivation in $G'$ of the first newly added rule has to be followed at some point by an application of the second newly added rule, corresponding to the application of the original rule in a derivation in $G$, and vice versa.

Remark 3.3.31. Applying transformation step RED-Z-NAIVE to an RTG with characteristic value $z_+$ reduces $n_Z$ in the variant function by 1. The transformation does not introduce unit productions, hence $n_U$ does not change. The transformation thus decreases $(n_U, n_Z)$, while characteristic value $u_-$ is invariant under the transformation.

Example 3.3.32 (RED-Z-NAIVE). Let $G$ be as in Example 3.3.28. Applying Transformation step 3.3.29 to the node labeled $d$ in production (1) $S \rightarrow a(B, d)$ replaces that production by (1a) $S \rightarrow a(B, D)$, (1b) $D \rightarrow d$. Note that $D$ is the fresh nonterminal added to $N$. (In Section 3.3.3.1 we will see an example of Z-node removal for every Z-node in the grammar.)

Definition 3.3.33 (Superfluous nonterminal). Given a grammar $G$, a nonterminal $A$ is called superfluous if and only if there is a nonterminal $B$ ($B \neq A$) such that $L(G, A) = L(G, B)$.

Transformation step RED-Z-NAIVE is naive, in that it may introduce superfluous nonterminals. There may be a nonterminal $B$ such that $B \rightarrow \alpha/n$ is the only production with LHS $B$. In such a case, instead of introducing a new nonterminal (which would make this new nonterminal and $B$ be superfluous), $B$ could be reused.1

We use this in the following version of the transformation step.

Transformation step 3.3.34 (RED-Z, Reducing the number of Z-nodes). Let $G = (N, \Sigma, r, Prods, S)$ be an RTG with characteristic value $z_+$. Then there must be a production $A \rightarrow \alpha \in Prods$ and an $n \in D_\alpha \setminus \{\varepsilon\}$ such that $\alpha(n) \in \Sigma$.

If there is a $B$ such that $B \rightarrow \alpha/n \in Prods$ is the only production with LHS $B$, let $A_{\text{split}}$ be the first such $B$ (according to a fixed ordering of $N$) and let $G' = (N, \Sigma, r, Prods', S)$, otherwise let $A_{\text{split}}$ be a fresh nonterminal of rank 0 (i.e. $A_{\text{split}} \notin N$, $r(A_{\text{split}}) = 0$) and let $G' = (N \cup \{A_{\text{split}}\}, \Sigma, r, Prods', S)$, where

$$Prods' = Prods \setminus \{A \rightarrow \alpha\} \cup \{A \rightarrow \alpha(n \mapsto A_{\text{split}}), A_{\text{split}} \rightarrow \alpha/n\}$$

Informally, a Z-node is selected, and the subtree occurrence rooted at this node is removed by introducing a fresh nonterminal if necessary, replacing the production by one in which that subtree occurrence has been replaced by the fresh nonterminal or a suitable existing terminal, and if necessary adding a production to replace the fresh nonterminal by the subtree. Note that the selection of $A \rightarrow \alpha$ and $n \in D_\alpha \setminus \{\varepsilon\}$ with $\alpha(n) \in \Sigma$ remains nondeterministic.

1More generally, a nonterminal $B$ such that $L_{\text{arc}}(G, B) = L_{\text{arc}}(G, \alpha/n)$ can be reused. This generalization is not treated here, as it is expected not to be applicable much more often than the special case treated here.
3.3 Regular tree grammars

It should be clear that this modified transformation step does not change the language generated by the grammar either. It does reduce the variant function in the same way the original transformation step RED-Z-NAIVE does.

Despite the reuse, the repeated application of RED-Z may still cause superfluous fresh nonterminals to be added. This is due to an unfortunate choice in the order of Z-nodes the step is applied to:

**Example 3.3.35.** Consider a grammar with start symbol \( A \) and production \( A \to a(b(c), b(c)) \). First applying RED-Z to the leftmost node labeled \( c \) results in productions \( A \to a(b(C), b(c)) \) and \( C \to c \) (with \( C \) a fresh nonterminal). By next applying the step to the rightmost node labeled \( b \), we obtain \( A \to a(b(C), B), C \to c \) and \( B \to b(c) \). If one then selects the remaining node labeled \( b \) in the production with LHS \( A \) and finally selects the remaining Z-node in \( B \to b(c) \), the final grammar without Z-nodes has productions \( A \to a(D, B), C \to c, B \to b(C) \) and \( D \to b(C) \), while a grammar with one less production and nonterminal is possible by selecting Z-nodes in a more appropriate order: \( A \to a(B, B), C \to c, B \to b(C) \).

The addition of superfluous fresh nonterminals is caused by the nondeterministic selection of Z-nodes in transformation step RED-Z. A number of choices exist for the selection of such Z-nodes, including:

- Using a purely nondeterministic selection.
- Using *shortest tree first* (STF) order, i.e. always selecting a Z-node at the root of a smallest subtree among all subtrees rooted in Z-nodes.
- Using *tallest tree first* (TTF) order, i.e. always selecting a Z-node at the root of a tallest subtree among all subtrees rooted in Z-nodes.

These three strategies for selecting a Z-node in a RED-Z step were experimentally compared by Stroemberg [Str07a] using the toolkit described in Chapter 8. The experiments in that thesis—with RTGs taken from code generation applications—seemed to indicate that the last two strategies both do not result in superfluous nonterminals, but Stroemberg gives a contrived example where TTF does result in superfluous nonterminals.

We show that using STF as a selection strategy can lead to superfluous fresh nonterminals, but only due to superfluous nonterminals in the original grammar. Afterwards, we therefore assume the use of that strategy to select a Z-node in RED-Z. Note that the use of this or the TTF strategy over a purely nondeterministic selection also has a disadvantage: a Z-node satisfying a certain characteristic has to be found, instead of simply any Z-node. The running time of a RED-Z step may thus increase.

Transformation RED-Z will often be applied to an RTG a number of times, i.e. until \( n_Z \) reduces to 0. For the order of Z-node replacement and the introduction of superfluous nonterminals, we have the following lemma:
Lemma 3.3.36. The result of applying transformation RED-Z to an RTG until \( n_{Z} \)
reduces to 0 does not depend (in terms of number of fresh nonterminals and corresponding
productions introduced) on the nondeterministic choice of \( \alpha/n \) in each
application of RED-Z with selection strategy STF, and the only superfluous nonterminals
created are caused by existing superfluous nonterminals in the original grammar.

Proof. To see that this is the case (modulo renaming of the fresh nonterminals),
suppose there are two different choices of \( \alpha/n \), say \( \beta \) and \( \gamma \). We have two cases for \( \beta \).

Case \( \beta = a \in \Sigma_0 \): due to the use of the STF strategy, \( \gamma = b \in \Sigma_0 \) as well.
If \( a \neq b \), then the order in which they are replaced does not matter.
If \( a = b \), either the same existing nonterminal will be reused to replace both (due to
the fixed ordering of \( N \)), or a fresh nonterminal will be introduced to replace one,
after which this fresh nonterminal will be reused to replace the other as well.

Case \( \beta = a(t_1, \ldots, t_n), a \in \Sigma \setminus \Sigma_0 \): due to the use of the STF strategy \( t_1, \ldots, t_n \)
must be single nonterminals, i.e. \( \beta = a(B_1, \ldots, B_n) \) with \( B_1, \ldots, B_n \) nonterminals.
Similarly, \( \gamma = b(C_1, \ldots, C_n) \) with \( b \in \Sigma \setminus \Sigma_0 \) and \( C_1, \ldots, C_n \) nonterminals.
If \( a \neq b \), the order in which they are replaced does not matter.
If \( a = b \), \( m = n \) holds. If \( B_1, \ldots, B_n = C_1, \ldots, C_m \) as well, then the same existing
nonterminal will be reused to replace both (due to the fixed ordering of \( N \)), or a fresh
nonterminal will be introduced to replace one, after which this fresh nonterminal will
be reused to replace the other as well.
If \( a = b \) (and hence \( m = n \)) yet there is an \( i, 1 \leq i \leq n = m \) such that \( B_i \neq C_i \),
then either \( L(G, B_i) = L(G, C_i) \) or not. If they are different, then the (reused or
introduced) nonterminals used to replace the two Z-nodes will not be superfluous
with respect to one another, regardless of the order in which the two Z-nodes are
replaced. If the two languages are equal, then \( B_i \) and \( C_i \) are superfluous, hence the
newly introduced nonterminals (if any) to replace the two Z-nodes are superfluous as
a result of existing superfluous productions, and the order in which the two Z-nodes
are replaced is irrelevant.

Newly introduced nonterminals thus can only be superfluous due to existing superfluous
ones, and the order in which Z-nodes are replaced is irrelevant.

Transformation step 3.3.37 (RED-U, removing a single unit production). Let
\( G = (N, \Sigma, r, Prods, S) \) be an RTG with characteristic value \( u_+ \). Then there must be
a production \( A \rightarrow B \in Prods \) with \( A, B \in N \). Then \( G' = (N, \Sigma, r, Prods', S) \) where

\[
Prods' = Prods \backslash \{ A \rightarrow B \} \cup \left\{ \begin{array}{l}
\text{Set } C, \gamma : B \xrightarrow{\gamma} C \land C \rightarrow \gamma \in Prods : A \rightarrow \gamma \\
\land \gamma \notin N
\end{array} \right\}
\]

is the resulting transformed grammar.
Informally, the unit production \( A \rightarrow B \) is removed and a rule \( A \rightarrow \gamma \) is added for each tree \( \gamma \) which is not a single nonterminal and which can be derived in one step from a nonterminal \( C \) which is reachable (in zero or more steps) from \( B \). Note that \( B \Rightarrow C \) in the set quantification can only involve unit productions, due to the restricted form of productions in an RTG.

**Remark 3.3.38.** Reflexive transitive closure \( \Rightarrow \) used in this transformation step can be computed using a variant of Warshall’s algorithm. Warshall’s algorithm and this variant were discussed in Section 2.4, while its use to compute relation \( \Rightarrow \) will be discussed in Section 5.7.2.

**Lemma 3.3.39.** For \( G \) and \( G' \) as in transformation RED-U, \( \mathcal{L}(G) = \mathcal{L}(G') \).

**Proof.** We first show that \( \mathcal{L}(G) \subseteq \mathcal{L}(G') \). We only need to consider those parts of a derivation in \( G \) of the form\(^2\) \( t_s \xrightarrow{A \rightarrow B} t_1 \xrightarrow{C \Rightarrow \gamma} t_f \) for \( A \rightarrow B \in \text{Prods} \). By construction, a rule \( A \rightarrow \gamma \) and thus a derivation \( t_s \xrightarrow{A \Rightarrow \gamma} t_f \) exists in \( G' \) and hence \( \mathcal{L}(G) \subseteq \mathcal{L}(G') \).

We now show that \( \mathcal{L}(G') \subseteq \mathcal{L}(G) \). The only new rules in \( G' \) are of the form \( A \rightarrow \alpha \) with \( \alpha \notin N \), hence we only need to consider parts of derivations in \( G' \) of the form \( t_s \xrightarrow{A \Rightarrow \alpha} t_f \) with \( \alpha \notin N \). Either \( A \rightarrow \alpha \) already was a production in \( G \), or it was not but there was a nonterminal \( C \) such that \( B \Rightarrow^* C \), \( C \rightarrow \alpha \) was a production in \( G \) and \( t_s \xrightarrow{A \Rightarrow B} t_3 \xrightarrow{\gamma} t_4 \) with \( t_4(n) = C \) was a possible derivation in \( G \). Thus, derivation \( t_s \xrightarrow{\gamma} t_4 \xrightarrow{C \rightarrow \alpha} t_f \) was possible in \( G \).

**Remark 3.3.40.** Applying transformation step RED-U to an RTG with characteristic value \( u_+ \) reduces the number of unit productions by 1, hence \( n_U \) decreases by 1. As a result, the value of the variant function \( (n_U, n_Z) \) also decreases. Note that the value of \( n_Z \) increases if and only if a RHS containing \( Z \)-nodes is duplicated. As a result, characteristic value \( z_- \) is invariant under this transformation.

**Example 3.3.41 (RED-U).** Let \( G \) be as in Example 3.3.28. Applying the above removal step to production (5) \( B \rightarrow S \), that rule is replaced by three new rules:

\[
(5a) \quad B \rightarrow a(B, d), \quad (5b) \quad B \rightarrow a(b(c), B), \quad (5c) \quad B \rightarrow c.
\]

Note that these rules are a result of rules (1), (2) and (3) respectively, and that the resulting grammar no longer has any unit productions.

\(^2\)Note that derivation steps at nodes other than \( n \) could occur in between. Since productions have a nonterminal LHS, steps are not contractive. Steps at nodes other than \( n \) therefore cannot take place at nodes on the path from the root to \( n \) between the application at \( n \) of \( A \rightarrow B \) and that of \( C \rightarrow \gamma \). Such steps can therefore safely be performed before or after the sequence of steps at node \( n \).
Applying transformation step RED-U may have two noticeable side effects:

- It may result in useless symbols or productions, depending on the order of application to different unit productions. For example, consider a grammar with start symbol $A$ and productions

  \[ A \rightarrow B, \quad B \rightarrow C, \quad C \rightarrow c. \]

  First removing $A \rightarrow B$ gives a grammar in which the remaining unit production is useless, while first removing $B \rightarrow C$ does not (although it does make $C \rightarrow c$ useless). The second case therefore always requires another application of RED-U to obtain an RTG with characteristic value $U$, while the first case does so only if REM-USELESS is not first applied.

- It may cause duplication of existing Z-nodes. For example, consider a grammar with start symbol $A$ and productions

  \[ A \rightarrow B, \quad B \rightarrow C, \quad C \rightarrow b(c), \quad A \rightarrow C. \]

  This grammar has a single Z-node. First removing $A \rightarrow C$ gives a grammar with newly added production $A \rightarrow b(c)$ and thus having one additional Z-node. This effect may be prevented by first applying a number of RED-Z steps before applying a RED-U step: first applying a RED-Z step to the example grammar replaces $C \rightarrow b(c)$ by productions $C \rightarrow b(D)$ and $D \rightarrow c$ using a new nonterminal $D$. This grammar no longer has any Z-nodes, so applying RED-U now cannot duplicate any Z-node.

### 3.3.3.1 Algorithms using the transformation steps

The transformation steps RED-U and either RED-Z-NAIVE or RED-Z are usually applied as often as necessary to remove all unit productions and Z-nodes. Using a notation similar to regular expressions, we use $*$ to denote repeated application (as often as necessary), $|$ for choice and $;$ for sequential composition.

Clearly, the order in which individual transformation steps RED-Z-NAIVE or RED-Z and RED-U are applied does not influence a grammar’s language (see Lemma 3.3.30 and the text following Transformation 3.3.34, as well as Lemma 3.3.39). A single nondeterministic algorithm (RED-Z-NAIVE|RED-U)* or (RED-Z|RED-U)* can therefore be used to obtain a type $(Z,U)$ RTG. It is clear that more deterministic versions may be used as well. Applying any of these algorithms to any kind of RTG results in a type $(Z,U)$ RTG—i.e., an ETG.

**Literature reference 3.3.42.** The algorithm described in [Din87, Construction 3.4.5] corresponds to algorithm RED-U*;RED-Z-NAIVE*. In [Din87] Example 3.4.6 hints at replacing transformation step RED-Z-NAIVE by RED-Z and applying this step concurrently to replace all occurrences of a single subtree. The algorithm is RED-Z-NAIVE*;RED-U* in [CDG+07, Section 2.1.1, Proposition 3] and in [Din87, Example 3.4.6].
We give the following examples to illustrate how applying the steps in different orders works. The reader should note that the resulting type \((Z_{-\cup -})\) RTGs indeed are the same regardless of the order of the steps.

**Example 3.3.43.** Applying RED-Z\(^\ast\) to the RTG of Example 3.3.28 results in grammar \(G_1 = (N_{\text{new}}, T, r, P_1, S)\) where

- \(N_{\text{new}} = N \cup \{ C, D, E \}\),
- \(P_1 = \{(1a)\ S \rightarrow a, \quad (1b)\ D \rightarrow d, \)
  \[
  \begin{array}{c}
  \includegraphics[width=2cm]{example1.png}
  \end{array}
  \]
  
  (2a) \(S \rightarrow a\), (2ba) \(E \rightarrow b\), (2bb) \(C \rightarrow c, \)
  \[
  \begin{array}{c}
  \includegraphics[width=2cm]{example2.png}
  \end{array}
  \]
  (3) \(S \rightarrow c, \quad (4)\ B \rightarrow b, \quad (5)\ B \rightarrow S, \quad (6)\ B \rightarrow d \).

It is clear that this is a type \((Z_{-\cup -})\) RTG.

**Example 3.3.44.** Applying RED-U\(^\ast\) to the RTG of Example 3.3.28 results in grammar \(G_2 = (N, T, r, P_2, S)\) where \(P_2\) is obtained from \(P\) by removing production (5) and adding productions (5a), (5b), and (5c) as in Example 3.3.41, i.e.

- \(P_2 = \{(1)\ S \rightarrow a, \quad (2)\ S \rightarrow a, \quad (3)\ S \rightarrow c, \)
  \[
  \begin{array}{c}
  \includegraphics[width=2cm]{example3.png}
  \end{array}
  \]
  (4) \(B \rightarrow b, \quad (5a)\ B \rightarrow a, \quad (5b)\ B \rightarrow b, \quad (5c)\ B \rightarrow c, \quad (6)\ B \rightarrow d \).

It is clear that this is a type \((Z_{+\cup -})\) RTG.

**Example 3.3.45.** Applying RED-U\(^\ast\) to the RTG of Example 3.3.43 results in grammar \(G_3 = (N_{\text{new}}, T, r, P_3, S)\) where \(N_{\text{new}}\) is as in that example and

- \(P_3 = \{(1a)\ S \rightarrow a, \quad (1b)\ D \rightarrow d, \)
  \[
  \begin{array}{c}
  \includegraphics[width=2cm]{example4.png}
  \end{array}
  \]
  (2a) \(S \rightarrow a\), (2ba) \(E \rightarrow b\), (2bb) \(C \rightarrow c, \)
  \[
  \begin{array}{c}
  \includegraphics[width=2cm]{example5.png}
  \end{array}
  \]
(3) \[ S \rightarrow c, \quad (4) \quad B \rightarrow b, \quad (5a) \quad B \rightarrow a, \quad (5b) \quad B \rightarrow a, \quad (5c) \quad B \rightarrow c, \quad (6) \quad B \rightarrow d \] 

It is clear that this is a type \((Z_{-}, U_{-})\) RTG i.e. an ETG.

**Example 3.3.46.** Applying RED-Z* to the RTG of Example 3.3.44 results in grammar \(G_{3}\).

Even though the order of the individual steps does not influence the language of the resulting grammar, the running time of the transformation may be affected:

- As indicated earlier while discussing step RED-U, duplication of Z-nodes may occur due to this step. It is therefore better to first perform RED-Z steps.

- Useless productions may be present in the original grammar or may be created by RED-U (as indicated before). Applying either transformation steps to such productions is useless, but removing useless productions from intermediate grammars may take more time than only doing so after all RED-U and RED-Z steps have been applied. This tradeoff depends on the grammar under consideration and it is hard to give general guidelines for it.

We claim that the set of productions of the resulting grammar does not depend on the order of the individual steps, modulo renaming of nonterminals and modulo removal of useless productions and symbols:

- The order of individual RED-U steps does not matter.

- The order of individual RED-Z steps does not matter (see Lemma 3.3.36).

- As for interleaving RED-Z steps with a RED-U step, a RED-U step may lead to duplication of existing Z-nodes, as remarked in the discussion of step RED-U. Yet Lemma 3.3.36 shows that the order of processing Z-nodes is irrelevant, even for identical subtrees rooted at Z-nodes. This thus includes such subtrees resulting from duplication.

- Regarding interleaving RED-U steps with a RED-Z step, it is clear that a RED-Z step does not affect the unit productions in a grammar.

The following diagram thus is a commuting diagram, modulo renaming of nonterminals and removal of useless productions and symbols.
3.3.3.2 Effects on tree automaton constructions

The individual transformation steps may have an effect on the constructions of a tree acceptor based on an RTG such as the ones that will be discussed in Chapter 5. As we will see there, acceptor states correspond to (sets of) subtrees of RHSs, and the use of the transformation steps may therefore lead to acceptors with fewer or more states than those obtained from an RTG without using the transformation steps.

3.4 Finite tree automata

In regular string language theory, finite automata are a central notion. There, non-deterministic and deterministic automata are known to be equivalent in acceptance power. The direction in which strings are processed by automata is irrelevant, although it is often assumed to be left-to-right.

Different kinds of finite tree automata, the generalization to trees of finite (string) automata, can also be defined. Here, the direction in which trees are processed is relevant, and so-called deterministic root-to-frontier automata are known to be strictly less powerful than the other types that are all equally powerful.

We define various specific tree automata types appearing in the literature. We define acceptance of a tree by a finite tree automaton, leading to a definition of the language accepted by a tree automaton. Example tree automaton will be given as part of the discussion of acceptance, while examples of all the different kinds of tree automata will occur throughout Chapters 5 and 6. Using the definition of a tree automaton’s language, we show the relative acceptance power of the various tree automata types and show that the deterministic root-to-frontier automata are indeed less powerful.

In the literature, many different but similar definitions of finite tree automata occur. The following definition makes it explicit that a ranked alphabet is involved. In [CDG+07], the ranking function is not made explicit, whereas [Don70] does not explicitly include the alphabet and [Din87] does not include the ranked alphabet as
part of the tuple defining a tree automaton. Thatcher and Wright [TW65, TW68] use
an algebraic approach to finite tree automata and have quite a different definition.

**Convention 3.4.1.** As we only consider finite tree automata in this dissertation,
we usually refer to them merely as tree automata for brevity. □

**Definition 3.4.2** (Tree Automaton).
A tree automaton (TA) $M$ is a 5-tuple $(Q, \Sigma, r, R, Q_{ra})$ such that

- $Q$ is a finite set, the *state set*,
- $(\Sigma, r)$ is a ranked alphabet,
- $R = \{ \text{Set } a : a \in \Sigma : R_a \} \cup \{ R_e \}$ is the set of transition relations, where $R_a \subseteq Q \times Q^n$ for all $a \in \Sigma$ and $R_e \subseteq Q \times Q$ (the *epsilon transition relation*), and
- $Q_{ra} \subseteq Q$, the *root accepting states*.

**Notation 3.4.3.** Note that $\varepsilon$ as used here is different from the use of $\varepsilon$ to indicate
the root node of a tree. The new use of $\varepsilon$ will be easy to distinguish from the former,
as it will only be used in $R_e$ and in terms such as $\varepsilon$-transition and $\varepsilon$-free. □

**Remark 3.4.4.** Note that for $a \in \Sigma_0$, $R_a \subseteq Q \times Q^0 = Q \times 1$. Some definitions of
tree automata (including the one in [Din87]) have $R_a \subseteq Q \times Q$ for symbols of rank 0,
making it into a special case and using an extra set of states to assign ‘below leaf
nodes’ of a tree. In Section 3.4.1 we will see the effect our choice has on tree state
assignments and tree acceptance.

Furthermore, note that for $a \in \Sigma_1$, $R_a \subseteq Q \times Q^1 = Q \times Q$. We will denote a transition
between a state $q$ and (a tuple consisting of) a single state $q_1$ interchangeably as
$(q, q_1)$ and as $(q, (q_1))$. □

**Definition 3.4.5** ($\varepsilon$-Nondeterministic Root-to-Frontier Tree Automaton). An $\varepsilon$-
nondeterministic root-to-frontier tree automaton ($\varepsilon$NRFTA) $M = (Q, \Sigma, r, R, Q_{ra})$ is
a TA where $R_a \in Q \rightarrow \mathcal{P}(Q^n)$ for all $a \in \Sigma$ such that $\overrightarrow{q} \in R_a(p) \equiv pR_a \overrightarrow{q}$, and
$R_e \in Q \rightarrow \mathcal{P}(Q)$ such that $q \in R_e(p) \equiv pR_e q$. □

**Definition 3.4.6** ($\varepsilon$-Nondeterministic Frontier-to-Root Tree Automaton). An $\varepsilon$-
nondeterministic frontier-to-root tree automaton ($\varepsilon$NFRTA) $M = (Q, \Sigma, r, R, Q_{ra})$ is
a TA where $R_a \in Q^n \rightarrow \mathcal{P}(Q)$ for all $a \in \Sigma$ such that $p \in R_a(q) \equiv pR_a q$, and
$R_e \in Q \rightarrow \mathcal{P}(Q)$ such that $p \in R_e(q) \equiv pR_e q$. □

Note that considering the relations $R_a$ and $R_e$ in any of these two ways is not a
restriction, and therefore the classes of $\varepsilon$NRFTA, $\varepsilon$NFRTA and TA are equivalent.

By restricting the above automata to the cases where $R_e = \emptyset$, we obtain the $\varepsilon$-free
TA and the ($\varepsilon$-free) NRFTA and NFRTA. We omit the definitions of the latter two for
brevity.
3.4 Finite tree automata

**Definition 3.4.7 (ε-free Tree Automaton).** An **ε-free tree automaton** (ε-free TA) \( M = (Q, \Sigma, r, R, Q_r) \) is a TA where \( R_\varepsilon = \emptyset \).

By restricting the relations \( R_a \) of the NRFTA and NFRTA to be functions yielding a single state tuple or a single state respectively, we obtain the deterministic tree automata:

**Definition 3.4.8 (Deterministic Root-to-Frontier Tree Automaton).** A **deterministic root-to-frontier tree automaton** (DRFTA) \( M = (Q, \Sigma, r, R, Q_r) \) is an NRFTA where \( R_a \in Q \rightarrow Q^n \) for all \( a \in \Sigma \) i.e. the \( R_a \) are functions yielding a single state tuple for every state—and \( Q_r = \{ q_r \} \) i.e. there is a unique root accepting state.

**Definition 3.4.9 (Deterministic Frontier-to-Root Tree Automaton).** A **deterministic frontier-to-root tree automaton** (DFRTA) \( M = (Q, \Sigma, r, R, Q_r) \) is an NFRTA where \( R_a \in Q^n \rightarrow Q \) for all \( a \in \Sigma \) i.e. the \( R_a \) are functions yielding a single state for every state tuple.

**Remark 3.4.10.** Note that in a DFRTA, for all \( a \in \Sigma_0 \), \( R_a = \{ (((), q) \} \) for some \( q \in Q \), i.e. every leaf node labeled by \( a \) is associated with the same state.

Finite string automata are often represented visually by a state diagram. States are represented by a circle, with double circles for final states, and transitions by an edge connecting two states and labeled by the transition’s symbol or \( \varepsilon \).

For finite tree automata, we extend the state diagram notation from finite string automata to finite tree automata. Each state is represented by a circle, with double circles indicating root accepting states. A transition relating a state \( q_0 \) and states \( q_1, \ldots, q_n \) on a symbol \( a \) or an \( \varepsilon \) transition between a state \( q_0 \) and a state \( q_1 \) is represented by

1. an edge connecting state \( q_0 \) to a small unlabeled circle, labeled by \( a \) or \( \varepsilon \) respectively, and
2. \( n \) edges connecting this circle to states \( q_i \), labeled by \( i \) (for \( 1 \leq i \leq n \)) or a single edge connecting this circle to state \( q_1 \) respectively.

In case of (root-to-frontier or frontier-to-root) directed tree automata, edges are directed. Note that double borders indicate root accepting states, which may be counterintuitive for root-to-frontier tree automata. Figure 3.4.1 shows examples of undirected transitions, one on \( a \) of rank \( n \geq 0 \), a specific case for a symbol \( a \) of rank 0, and two different ones on \( \varepsilon \). Figure 3.4.3 shows an example of an undirected tree automaton (i.e. a TA). (This TA was constructed from the running example RTG of Example 3.3.28 using one of the constructions for tree acceptors that we will encounter in Chapter 5. It has a single root accepting state, but in general a TA that is not a DRFTA may have more than one, i.e. \( |Q_r| > 1 \) may hold.) The usefulness of such tree automaton state diagrams becomes apparent further on in this dissertation.
where they are used to visualize (small) tree automata. The notation is limited to
small automata, since the number of edges is too large for larger automata.

**Remark 3.4.11.** The above tree automaton state diagram notation is a rather
obvious extension of the one for string automata. It was created after initial literature
research surprisingly turned up no visualizations for tree automata. Afterwards, we
discovered that Reisig uses a similar representation in [Rei79].

### 3.4.1 Acceptance and language

We define acceptance of a tree by a tree automaton based on tree state assignments.
A *tree state assignment* is an assignment of a state to each tree node that respects
the transition relations of the tree automaton. For any TA $M$ and tree $t$, we will de-
note the set of such state assignments respecting the transition relations by $St_M(t)$.
A tree $t$ then is accepted by $M$ if and only if there is a tree state assignment in this
set $St_M(t)$ that assigns a root accepting state to the tree’s root. Our general de-
inition applies to undirected, root-to-frontier as well as frontier-to-root tree automata.
Before we give it, we first give an example of state assignments.

**Example 3.4.12** (Tree state assignment). For the example TA depicted in Fig-
ure 3.4.3, some trees and state assignments to them are depicted in Figure 3.4.2.
For each node in each of the trees, the state assigned to it by the state assignment
is positioned next to the node.

For tree $c$, a first state assignment is depicted on the left. It respects the transition
relations of the TA, since transition $(q_0, ())$ is part of $R_c$ as can be seen in the
depiction of the automaton.

The second state assignment also respects the relations, since $(q_6, ()) \in R_c$ (as we
just saw) and $(q_s, q_6) \in R_c$, and hence $(q_s, ()) \in R_c^* \circ R_c$ using state $q_6$. Since
this assignment assigns a root accepting state, $q_s$, to the tree’s root, it even is an
*accepting* state assignment, i.e. tree $c$ is accepted by the TA.

The third example is not a correct state assignment: the assignment does not respect
the transition relations, as $(q_s, ()) \notin R_c^* \circ R_d$. 

![Figure 3.4.1 Example TA transitions. Note that the two figure parts on the
lower right depict different transitions.](image-url)
The fourth example shows a larger state assignment respecting the transition relations: \((q_6, (\varepsilon)) \in R_c\) as we saw, \((q_5, (q_6)) \in R_b\), \((q_1, (\varepsilon)) \in R_d \circ R_d\) (using state \(q_2\)), and \((q_5, (q_5, q_1)) \in R_c \circ R_c\) (using state \(q_4\)). This is also an accepting state assignment, since \(q_a\) is assigned to the tree’s root. Note that \(q_4\) could be assigned to the root instead, in which case it would also be a correct state assignment yet not an accepting one.

Another accepting state assignment for tree \(a(b(c), d)\) exists, depicted on the right of the figure. It is correct since \((q_1, (\varepsilon)) \in R_d^2 \circ R_c\) (using states \(q_a\) and \(q_6\)), \((q_2, (\varepsilon)) \in R_d\), \((q_1, (q_1)) \in R_c \circ R_b\) (using \(q_0\)), and \((q_4, (q_1, q_2)) \in R_c \circ R_a\) (using state \(q_5\)). Note that \(q_4\) could be assigned to the root instead, making it a non-accepting (yet correct) state assignment.

For tree \(a(b(c), d)\) the two state assignments depicted are the only two accepting ones. A total of six correct state assignments exists, i.e. \(|\text{St}_M(a(b(c), d))| = 6\): the two accepting ones, the two indicated ones assigning state \(q_4\) and \(q_3\) respectively to the root node, and two additional ones with \(q_1\) assigned to the root node of each of the state assignments depicted in the figure (since \((q_1, q_a) \in R_c\).
We will now formally define function $St_M$, followed by the definition of an auxiliary function $RSt_M$ and of acceptance function $acc_M$ indicating whether a certain tree is accepted by $M$.

**Definition 3.4.13.** For a TA $M = (Q, \Sigma, r, R, Q_{ra})$, define $3$ $St_M \in Tr(\Sigma, r) \rightarrow \mathcal{P}(N_r^* \rightarrow Q)$ for $t \in Tr(\Sigma, r)$ and $f \in D_t \rightarrow Q$ by

$$f \in St_M(t) \iff \left\{ \forall n : n \in D_t : (f(n), (f(n \cdot 1), \ldots, f(n \cdot r(t(n)))) \in R_n^* \circ R_{t(n)} \right\}.$$ 

For a given tree $t$, set $St_M(t) \in \mathcal{P}(D_t \rightarrow Q)$ is a set of functions $f$ with signature $D_t \rightarrow Q$ and representing a correct state assignment (by means of a mapping from $t$’s domain to state set $Q$).

Note that the definition requires that for every leaf node a transition between the state assigned to this leaf node and the empty tuple () on the node’s symbol exists, as was apparent from Example 3.4.12.

**Remark 3.4.14.** For tree automata definitions that define $R_a$ to be a subset of $Q \times Q$ for $a \in \Sigma_0$, leaf accepting states assigned ‘below’ leaf nodes exist, and an element or subset of $Q$ called the leaf accepting state(s) is defined to determine whether an assignment of a particular state ‘below’ a leaf node labeled $a$ leads to leaf acceptance at that leaf or not. In our tree automata definitions with $R_a \subseteq Q \times I$ for such an $a$, this is determined by whether the state assigned to the node labeled $a$ has a transition on $a$ to the empty tuple () or not.

**Definition 3.4.15.** For a TA $M = (Q, \Sigma, r, R, Q_{ra})$, define $RSt_M \in Tr(\Sigma, r) \rightarrow \mathcal{P}(Q)$ as

$$RSt_M(t) = St_M(t)(\varepsilon).$$

Here we use $St_M(t)$ as if it had signature $D_t \rightarrow \mathcal{P}(Q)$, i.e. for $n \in D_t$, we use $St_M(t)(n)$ to denote the set of states assigned to $n$ in any of the state assignments to $t$.

Informally, $RSt_M(t)$ gives the set of states which may appear at the root of $t$ in a tree state assignment respecting the transition relations $R$.

**Definition 3.4.16 (Tree Acceptance).** For $M = (Q, \Sigma, r, R, Q_{ra})$, define $acc_M \in Tr(\Sigma, r) \rightarrow \mathbb{B}$ as:

$$acc_M(t) \equiv RSt_M(t) \cap Q_{ra} \neq \emptyset$$

**Definition 3.4.17 (Tree Automaton Language).** The language of a TA $M$ is defined as

$$L_{TA}(M) = \langle \text{Set } t : t \in Tr(\Sigma, r) \land acc_M(t) \land t \rangle.$$

---

3Note that $E^*$ in the function signature is as in Definition 3.1.1 i.e. it is the set of sequences of edge labels.
We often use $St$, $RSt$ or $acc$, i.e. without a subscript, if the automaton involved is clear from the context.

**Remark 3.4.18.** Remark 3.1.13 mentioned the relation between strings over an alphabet $\Sigma$ and trees over the extension of $\Sigma$ with a ranking function (assigning a rank of 1 to each of its symbols) and a symbol $\#$ of rank 0. Here we sketch the similarity between a string acceptor for a string $w = w_1 w_2 \ldots w_n$ and a tree acceptor for tree $w_1(w_2(\ldots(w_n(\#)\ldots)))$. Using the definitions of acceptance and tree state assignment, such a string tree is accepted if and only if there is a state assignment by a TA $M = (Q, \Sigma, r, R, Q_{ra})$ that is as depicted on the left of Figure 3.4.4 and such that

- $q_s \in Q_{ra}$,
- $(q_s, (q_1)) \in R_{w_1}$,
- $(q_{i-1}, (q_i)) \in R_{w_i}$ for all $i : 2 \leq i \leq n$, and
- $(q_n, (\)) \in R_{\#}$.

![Figure 3.4.4 State assignment for tree $w_1(w_2(\ldots(w_n(\#)\ldots)))$ and string $w = w_1 w_2 \ldots w_n$.](image)

Considering a string acceptor $M' = (Q', \Sigma, \delta, I, F)$ (i.e. with state set $Q'$, transition function $\delta \in Q' \times \Sigma \rightarrow \mathcal{P}(Q')$, and initial and final states $I$ and $F$, respectively) and extended transition function $\delta^* \in Q' \times \Sigma^* \rightarrow \mathcal{P}(Q')$ (defined in the usual way), string $w$ is accepted if and only if $\delta^*(q'_s, w) \cap F \neq \emptyset$ for some $q'_s \in I$. We can take an alternative view however, with transition relations $\delta_\sigma \in Q \times Q$ for every $\sigma \in \Sigma$, and considering a string to be accepted if there is a state assignment to positions before and after the string and in between string symbols, as depicted on the right of the figure, and such that
\begin{itemize}
\item $q'_r \in I$,
\item $(q'_s, q'_1) \in \delta_{w_1}$,
\item $(q'_{i-1}, q'_i) \in \delta_{w_i}$ for all $i : 2 \leq i \leq n$, and
\item $q'_n \in F$.
\end{itemize}

Comparing the two state assignments, the similarities are apparent.

Since Definition 3.4.13 does not provide an algorithm to compute either $St_M(t)$ or $RSt_M(t)$, the above definitions do not give us an algorithm to compute $acc_M(t)$. In Chapter 5, we will consider imperative algorithms to determine acceptance by either a frontier-to-root or a root-to-frontier computation.

Frontier-to-root algorithms will be based on a recursive version of $RSt_M(t)$, using root tree state assignments of direct subtrees of a given tree to define the root tree state assignments for the root of the whole tree. We present the following lemma without proof, as this would require a lot of (straightforward) formula manipulation but would not give much additional insight.

\textbf{Lemma 3.4.19.} Given a TA $M = (Q, \Sigma, r, R, Q_{ra})$, we have for every $a(t_1, \ldots, t_n) \in Tr(\Sigma, r)$ (using tuple projection operator $\pi$ of Section 2.2):

$$RSt_M(a(t_1, \ldots, t_n)) = \langle \text{Set} q, \overline{q} : (q, \overline{q}) \in R^*_r \circ R_a \land \langle \forall i : 1 \leq i \leq n : \pi_i(\overline{q}) \in RSt_M(t_i) \rangle : q \rangle$$

For use with root-to-frontier tree automata, we define a root-to-frontier acceptance function $Accept$, relate this function to $RSt$, and express $acc$ in terms of $Accept$. Function $Accept$ will be used in the root-to-frontier acceptance algorithms in Chapter 5.

\textbf{Definition 3.4.20} (Root-to-frontier Acceptance Function). For any $\varepsilon$NRFTA $M = (Q, \Sigma, r, R, Q_{ra})$, function $Accept \in Tr(\Sigma, r) \times Q \rightarrow \mathbb{B}$ is defined by

$$Accept(a(t_1, \ldots, t_n), q) \equiv \langle \exists \overline{q} : \overline{q} \in R_a(R^*_r(q)) \land \forall i : 1 \leq i \leq n : Accept(t_i, \pi_i(\overline{q})) \rangle$$

for every $a(t_1, \ldots, t_n) \in Tr(\Sigma, r)$ and $q \in Q$. Note that $Accept(a, q) \equiv () \in R_a(R^*_r(q))$ for $a \in \Sigma_0$ and $q \in Q$.

\textbf{Lemma 3.4.21.} Let $M = (Q, \Sigma, r, R, Q_{ra})$ be an $\varepsilon$NRFTA. For every $q \in Q$ and $t \in Tr(\Sigma, r)$,

$$q \in RSt(t) \equiv Accept(t, q).$$

\textbf{Proof.} By structural induction.

Case $t = a \in \Sigma_0$: 
\[ \text{Accept}(a, q) \]
\[ \equiv \{ \text{definition Accept} \} \]
\[ () \in R_a(R_\varepsilon^a(q)) \]
\[ \equiv \{ \text{Lemma 3.4.19} \} \]
\[ q \in RSt(a) \]

Case \( t = a(t_1, \ldots, t_n), a \in \Sigma \setminus \Sigma_0: \)
\[ \text{Accept}(a(t_1, \ldots, t_n), q) \]
\[ \equiv \{ \text{definition Accept, Induction Hypothesis} \} \]
\[ \langle \exists \overline{q} : \overline{q} \in R_a(R_\varepsilon^a(q)) : \langle \forall i : 1 \leq i \leq n : \pi_i(\overline{q}) \in RSt(t_i) \rangle \rangle \]
\[ \equiv \{ \text{Lemma 3.4.19, set calculus} \} \]
\[ q \in RSt(a(t_1, \ldots, t_n)) \]

\[ \square \]

**Corollary 3.4.22.** Let \( M = (Q, \Sigma, r, R, Q_{ra}) \) be an \( \varepsilon \)NRFA. For every \( t \in Tr(\Sigma, r), \)
\[ \text{acc}(t) \equiv \langle \exists q : q \in Q_{ra} : \text{Accept}(t, q) \rangle. \]
\[ \square \]

### 3.4.2 Removing unreachable states

Tree automata, like string automata, may have *unreachable states*: states that are not reachable from any root accepting state or which cannot occur at the root of a state assignment for any (sub-)tree. For *string* automata, unreachable states are precisely the *useless* states, i.e. states that do not occur in any accepting computation. For tree automata, unreachable states form a proper subset of the useless states, i.e. reachable states may still be useless, as we will show by example.

**Definition 3.4.23** (Useful, useless state). Let \( M = (Q, \Sigma, r, R, Q_{ra}) \) be any TA. A state \( q \in Q \) is *useful* if and only if it is used in an accepting computation of \( M \) on some \( t \in Tr(\Sigma, r) \). A state is *useless* if and only if it is not useful.

**Definition 3.4.24.** Given a TA \( M = (Q, \Sigma, r, R, Q_{ra}) \), set \( R\text{Reach}_M \subseteq Q \) is the smallest set satisfying (using tuple to set flattening operator \( \Pi \) of Section 2.2)
\[ R\text{Reach}_M = Q_{ra} \]
\[ \cup \]
\[ \langle \bigcup a, q, \overline{s} : (q, \overline{s}) \in R_a^* \circ R_a : \Pi(\overline{s}) \rangle \]
\[ \wedge q \in R\text{Reach}_M \]
\[ \cup \]
\[ \langle \text{Set} q, s : (q, s) \in R_a^* \wedge q \in R\text{Reach}_M : s \rangle. \]

A state \( q \in Q \) is root accepting state-reachable if and only if \( q \in R\text{Reach}_M \).
States that occur at the root of a correct state assignment for some (sub-)tree—called leaf acceptance-reachable states—could be defined as well. When $M$ is clear from the context, we use $\textit{RAReach}$ instead of $\textit{RAReach}_M$.

**Example 3.4.25.** In the TA of Figure 3.4.5, states $q_1$ and $q_2$ are root accepting state-reachable by transition $(q_s, (q_1, q_2)) \in R_a$. State $q_2$ is also leaf acceptance-reachable, as $q_2 \in R\text{St}(d)$ using $(q_2, (\_)) \in R_d$. State $q_1$ and $q_2$ are not, as there is no tree $t$ such that $q_1 \in R\text{St}(t)$ or $q_2 \in R\text{St}(t)$. As a result, they are both useless. State $q_2$ is useless as well however, since no accepting state assignment exists involving it. (In fact, the language of the TA depicted is the empty language.)

![Diagram of Example 3.4.5](image)

**Figure 3.4.5** Example TA with reachable yet useless state $q_2$.

Useless states arise for example in tree acceptor constructions presented in Chapter 5. As only root accepting state-unreachable states occur in these constructions, we only discuss the removal of such useless states. (Leaf acceptance-unreachable states could easily be removed similarly; the removal of other useless states—such as state $q_2$ in Example 3.4.25—is more complicated.)

Any TA which has states and transitions that are not root accepting state-reachable may be transformed into an equivalent TA without such states and transitions. We show how to compute root accepting state-reachable states of a TA, and how this computation can be used for such a transformation.

**Property 3.4.26.** The successive approximations of $\textit{RAReach}$ can be computed by computing the sequences $\textit{RAReach}_n$ for increasing $n$, starting with $n = 0$ and ending as soon as $\textit{RAReach}_{n+1} = \textit{RAReach}_n$, where $\textit{RAReach}_n$ is defined by:

\[
\begin{align*}
\textit{RAReach}_0 &= Q_{ra} \\
\textit{RAReach}_{n+1} &= \textit{RAReach}_n \\
&\cup \\
&\left\{ \bigcup a, q, \vec{s} : (q, \vec{s}) \in R^*_a \circ R_a : \Pi(\vec{s}) \right\} \\
&\cup \\
&\langle \text{Set} q, s : (q, s) \in R^*_e \land q \in \textit{RAReach}_n : s \rangle
\end{align*}
\]
Note that the above uses the standard approach for computing least fixed points by approximation from below, using monotonicity and finiteness of the sequence $R\text{AReach}_n$. 

**Transformation 3.4.27** (rem-$\text{UNR}_{ra}$, removing root accepting state-unreachable states and transitions). Let $M = (Q, \Sigma, r, R, Q_{ra})$ be a TA. Then

$$M_1 = (R\text{AReach}, \Sigma, r, R \cap (R\text{AReach} \times (\Sigma \cup \{\varepsilon\}) \times R\text{AReach}^*), Q_{ra})$$

is an equivalent automaton which has no root accepting state-unreachable states and transitions. 

The automaton resulting from Transformation 3.4.27 is equivalent to the original one as the only states and transitions removed are root accepting state-unreachable ones, and these do not influence the language accepted.

The transformation will reappear when considering a specific tree acceptor construction in Section 5.6.3.

### 3.4.3 Relating the tree automata types

From finite string automata theory, it is well known that nondeterministic and deterministic automata are equivalent, and that the direction in which such automata process their input—either from left to right or from right to left—does not influence the class of languages that can be recognized.

It turns out that this is not the case for finite tree automata: the class of languages accepted by automata of type TA, $\varepsilon$NRFTA, NRFTA, $\varepsilon$NFRTA, NFRTA or DFRTA are all the same, whereas the class of languages accepted by deterministic root-to-frontier tree automata (DRFTAs) is a strict subset thereof. We show this by giving an example of a language not recognizable by a DRFTA but for which an NRFTA can be constructed. We then show a relation between certain NRFTAs and the DRFTAs obtained from them based on the subset construction for NRFTAs.

To show the equivalence of the other classes, we present transformations from automata with epsilon transitions to automata without such transitions and define a subset construction to transform NFRTAs into DFRTAs. The equivalence between $\varepsilon$NRFTAs, $\varepsilon$NFRTAs and TAs follows from the fact that $\varepsilon$NRFTAs and $\varepsilon$NFRTAs are obtained from equivalent TAs and vice versa by merely viewing their transition relations as functions and vice versa.

The following overview diagram (based on [Din87, p. 58]) shows the relationships between the language classes accepted by the various tree automata types:
All of these results have long been known and more detail can be found in the literature, e.g. in [Eng75b, GS84, GS97, CDG+07].

Using the following transformation, an equivalent ε-free TA can be constructed for every TA.

**Transformation 3.4.28 (rem-ε, transforming a TA to an ε-free TA).** Let \( M = (Q, \Sigma, r, R, Q_{ra}) \) be a TA. Then \( M' = (Q, \Sigma, r, R', Q_{ra}) \) where

- \( R'_e = \emptyset \) and
- \( R'_a = R_e^* \circ R_a \) (for all \( a \in \Sigma \)).

\[ \]

As an example of this transformation, Figure 3.4.6 shows the removal of a single ε-transition from a tree automaton.

**Lemma 3.4.29.** Given \( M \) and \( M' \) as in Transformation 3.4.28 rem-ε, \( \mathcal{L}_{TA}(M) = \mathcal{L}_{TA}(M') \).

**Proof idea.** The only influence of the transformation on Definition 3.4.17 for \( \mathcal{L}_{TA} \) is that in the definition of St (indirectly, using the definitions of \( ace \) and \( RSi \)), which is not affected by the transformation, as \( R'_e^* \circ R'_a = R_e^* \circ R_a \).

For every εNRFTA, an equivalent NRFTA can be constructed and for every εNRFTA, an equivalent NRFTA can be constructed. This can be seen by looking at the transition relations of the above construction in either of the two directions. We explicitly give these two transformations here as well.

**Transformation 3.4.30 (εNRFTA to NRFTA).** Let \( M = (Q, \Sigma, r, R, Q_{ra}) \) be an εNRFTA. Then NRFTA \( M' = (Q, \Sigma, r, R', Q_{ra}) \) where
3.4 Finite tree automata

![Finite tree automata diagram]

**Figure 3.4.6** Removal of the single ε-transition \((q_{\varepsilon}, q_1)\) by transformation \(\text{REM-ε}\). The transitions involved have been highlighted for emphasis. Note that state \(q_1\) becomes a root accepting unreachable state.

- \(R'_{\varepsilon} = \emptyset\) and
- \(R'_a(q) = R_a(R^*_\varepsilon(q))\) (for all \(a \in \Sigma, q \in Q\)).

\[\Box\]

**Transformation 3.4.31 (εNFRTA to NFRTA).** Let \(M = (Q, \Sigma, r, R, Q_{ra})\) be an εNFRTA. Then NFRTA \(M' = (Q, \Sigma, r, R', Q_{ra})\) where

- \(R'_{\varepsilon} = \emptyset\) and
- \(R'_a(\overline{q}) = R^*_a(R_{\varepsilon}(\overline{q}))\) (for all \(a \in \Sigma, \overline{q} \in Q^n\)).

\[\Box\]

**Lemma 3.4.32.** There are NFRTAs for which no DRFTA accepting the same language can be constructed.

*Proof.* Let \(L = \{a(c, d), a(d, c)\}\). We try to construct a DRFTA accepting \(L\). Let \(q'_{ra}\) be its root accepting state. There must be exactly one pair of states \((q'_1, q'_2)\) such that \(R_a(q'_{ra}) = (q'_1, q'_2)\). To recognize both trees, \(R_c(q'_1) = R_d(q'_2) = R_d(q'_1) = R_c(q'_2) = ()\)
must hold, but this means that trees \( a(c, c) \) and \( a(d, d) \) are accepted as well (see Figure 3.4.7). A DRFTA accepting \( L \) therefore cannot exist.

An NRFTA accepting the language \( L \) can be constructed, with \( Q = \{ q_r, q_1, q_2 \} \), \( Q_{ra} = \{ q_{ra} \} \), \( R_a(q_{ra}) = \{ (q_1, q_2), (q_2, q_1) \} \), \( R_c(q_1) = \{ () \} \) and \( R_a(q_2) = \{ () \} \). This NRFTA is depicted in Figure 3.4.8.

We define a subset construction for NRFTAs. Due to the preceding lemma, for some NRFTAs this construction results in a DRFTA accepting a language different from that accepted by the NRFTA. We show that for an NRFTA without root accepting state reachable useless states, the language accepted by the DRFTA is the path closure (see Definition 3.1.43) of the language accepted by the NRFTA. The construction will play a role in certain stringpath based algorithms discussed in Part II. In Section 6.7 we will show how such a DRFTA can be used in these stringpath based algorithms.

**Construction 3.4.33** (\( \text{SUBSET}_{\text{NR}} \), subset construction). Let NRFTA \( M = (Q, \Sigma, r, R, Q_{ra}) \) be given, then \( M' = (Q', \Sigma, r, R', Q'_{ra}) \) is a DRFTA where

- \( Q' = \mathcal{P}(Q) \),
3.4 Finite tree automata

- $R'_a \in Q' \to (Q')^n$, defined for all $a \in \Sigma \setminus \Sigma_0, q' \in Q'$ by
  - $R'_a(q') = q'$ where
    - $\forall i \leq i \leq n : \pi_i(q') = (\bigcup q : q \in q' : \pi_i(R_a(q)))$
  - and by
    - $R'_a(q') = (\bigcup q : q \in q' : R_a(q))$ for $a \in \Sigma_0, q' \in Q'$, and

- $Q'_{ra} = \{ Q_{ra} \}$.

\[ \Box \]

**Example 3.4.34.** For the NRFTA as in Lemma 3.4.32, the Application of subset$_{RF}$ results in the DRFTA depicted in Figure 3.4.9.

![Figure 3.4.9 DRFTA obtained by subset construction from NRFTA for L](image)

The following lemma shows that for an NRFTA without root accepting reachable useless states, a tree is in the obtained DRFTA’s language if and only if each of its stringpaths is a stringpath of some tree in the language of the underlying NRFTA.

**Lemma 3.4.35.** Let $M$ and $M'$ be as in Construction 3.4.33 subset$_{RF}$, with $M$ having no root accepting reachable states that are useless, and let $L = L_{TA}(M)$ and $L' = L_{TA}(M')$ respectively, then, for every $t \in Tr(\Sigma, r),$

$$ t \in L' \equiv SPaths(t) \subseteq \bigcup u : u \in L : SPaths(u) $$

i.e. $L' = \text{PathCl}(L)$ (as in Definition 3.1.43).

**Proof.** We prove the equivalence for every $t \in Tr(\Sigma, r)$.

$\Leftarrow$: Let $SPaths(t) \subseteq \bigcup u : u \in L : SPaths(u)$, then for every stringpath $w \in SPaths(t)$ there is a tree $u \in L$ such that $w \in SPaths(u)$. Consider such a tree $u$.

Let $n = w \downarrow N_+$ i.e. the end node of stringpath $w$. (Note that $n \in D_t, n \in D_u$, that $u(n) = t(n) = w|1$, and that $u(m) = t(m)$ for every $m \in \text{pref}(n)$.)

Since $u \in L$, there are states $q_\varepsilon, \ldots, q_n$ of NRFTA $M$ such that $q_\varepsilon, \ldots, q_n$ get assigned to nodes $\varepsilon, \ldots, n$ (i.e. the nodes along stringpath $w$) and

- $q_\varepsilon \in Q_{ra}$,
- for every consecutive $m, m \cdot i$ along $w, q_{m \cdot i} \in \pi_i(R_u(m)(q_{m}))$, and
- $(\cdot) \in R_u(n)(q_n)$.
Since $q_e \in Q_{ra}$ and, by definition of $\text{SUBSET}_{\text{nf}}$, $Q'_{ra} = \{Q_{ra}\}$, there is a state $q'_e = Q_{ra}$ with $q_e \in q'_e$ and $q'_e \in Q'_{ra}$. Let $m \cdot i \in \text{pref}(n)$ and let $q'_m \in Q'$ be such that $q_m \in q'_m$. Recall from above that $q_{m \cdot i} \in \pi_i(R_{u(m)}(q_m))$ and that $u(m) = t(m)$, then, by definition of $\text{SUBSET}_{\text{nf}}$, $q_{m \cdot i} \in \pi_i(R'_t(m)(q'_m))$. Define $q'_{m \cdot i}$ to be $\pi_i(R'_t(m)(q'_m))$, then $q_{m \cdot i} \in q'_{m \cdot i}$.

Let $q'_n \in Q'$ such that $q_n \in q'_n$. Recall from above that $(\cdot) \in R_{u(n)}(q_n)$ and that $u(n) = t(n)$, then, by definition of $\text{SUBSET}_{\text{nf}}$, $(\cdot) = R'_t(n)(q'_n)$. Hence there exist $q'_e, \ldots, q'_n$ assigned to nodes $\varepsilon, \ldots, n$ along stringpath $w$ such that

- $q'_e \in Q'_{ra}$ (hence $q'_e = Q_{ra}$),
- for every consecutive $m, m \cdot i$ along $w$, $q'_{m \cdot i} = \pi_i(R'_t(m)(q'_m))$, and
- $(\cdot) = R'_t(n)(q'_n)$.

This assignment of states from $Q'$ to nodes along $w$ in tree $t$ is unique. For stringpaths with a common prefix, the states of $M'$ assigned to nodes along this prefix are the same for all such stringpaths. Since the above unique assignment exists for every stringpath $w$ of $t$, there exists a unique assignment of states from $Q'$ to $t$ and this assignment is an accepting one, i.e. $t \in L'$ holds. Hence we have proven $t \in L' \Leftarrow S\text{Paths}(t) \subseteq \bigcup u : u \in L : S\text{Paths}(u) \}$. 

$\Rightarrow$: Let $t \in L'$. We need to prove $S\text{Paths}(t) \subseteq \bigcup u : u \in L : S\text{Paths}(u) \}$. Let $w \in S\text{Paths}(t)$ and let $n$ be the end node of stringpath $w$.

Since $t \in L'$, there are states $q'_e, \ldots, q'_n$ of $\text{DRFTA} M'$ that get assigned to nodes $\varepsilon, \ldots, n$ of stringpath $w$ in the computation of $M'$ for $t$ and

- $q'_e = Q_{ra}$ (since $q'_e \in Q'_{ra}$),
- for every consecutive $m, m \cdot i$ along $w$, $q'_{m \cdot i} = \pi_i(R'_t(m)(q'_m))$, and
- $(\cdot) = R'_t(n)(q'_n)$.

We will now construct a tree $u \in L$ such that $w \in S\text{Paths}(u)$. Using the definition of $\text{SUBSET}_{\text{nf}}$, there is a state $q_n \in Q$ of $\text{NRFTA} M$ such that $q_n \in q'_n$ and $(\cdot) \in R_{u(n)}(q_n)$. (Note that $u(n) = t(n)$.) Let $m \cdot i \in \text{pref}(n)$, and let $q'_{m \cdot i} \in Q$ be such that $q_{m \cdot i} \in q'_{m \cdot i}$; then using the definition of $\text{SUBSET}_{\text{nf}}$, there is a $q_m \in Q$ such that $q_m \in q'_m$ and $q_{m \cdot i} \in \pi_i(R_{u(m)}(q_m))$. (Note that $u(m) = t(m)$.) Hence there exist $q_n \in Q$, $q_n \in q'_e, \ldots, q'_n$ assignable to nodes $\varepsilon, \ldots, n$ along stringpath $w$ such that

- $q_e \in q'_e$, i.e. $q_e \in Q_{ra}$,
- for every consecutive $m, m \cdot i$ along $w$, $q_{m \cdot i} \in \pi_i(R_{u(m)}(q_m))$, and
- $(\cdot) \in R_{u(n)}(q_n)$. 


Such a state assignment also assigns states to the sibling nodes of nodes $\varepsilon, \ldots, n$ along $w$. Let $q$ be the state assigned to such a sibling node. Clearly, $q$ is root accepting reachable, hence—using that the NFRTA has no root accepting reachable useless states—it is useful and there is a tree $s \in L$ such that $q$ is assigned to the root of some subtree $s/l$ (with $l \in D_s$) of $s$. We choose this subtree $s/l$ as the tree rooted at the sibling node. Doing this for every such sibling node, we obtain a tree $u$. This tree obviously has $w$ as one of its stringpaths and belongs to $L$.

This reasoning holds for every $w \in SPaths(t)$ and hence concludes our proof of $t \in L' \Rightarrow \text{SPaths}(t) \subseteq \bigcup u : u \in L : \text{SPaths}(u)$. 

**Construction 3.4.36 (subsetFr, subset construction).** Let NFRTA $M = (Q, \Sigma, r, R, Q_{ra})$ be given, then $M' = (Q', \Sigma, r, R', Q'_{ra})$ is a DFRTA where

- $Q' = \mathcal{P}(Q)$,
- $R'_a \in (Q')^n \rightarrow Q'$, defined for all $a \in \Sigma \setminus \Sigma_0$ by $R'_a(q') = \left\{ \bigcup \overline{q} : \forall i : 1 \leq i \leq n : \pi_i(q) \in \pi_i(\overline{q}) \right\}$ and by $R'_a(()) = R_a(())$ for $a \in \Sigma_0$, and
- $Q'_{ra} = \left\{ \text{Set } q' : q' \in Q' \land q' \cap Q_{ra} \neq \emptyset : q' \right\}$.

The following lemma shows that this construction is correct, i.e. for every NFRTA, a DFRTA accepting the same language can be constructed.

**Lemma 3.4.37.** Given $M$ and $M'$ as in Construction 3.4.36 subsetFr, $L_{TA}(M) = L_{TA}(M')$.

**Proof.** We first show that $RSt_M(t) = \{ RSt_M(t) \}$ for any tree $t$.

Case $t = a \in \Sigma_0$:

- $RSt_M(t) = \{ \text{Lemma 3.4.19, } r(a) = 0, M' \text{ is a DFRTA } \}$
- $\{ R'_a(()) \}$
- $\{ \text{definition } R'_a \}$
- $\{ R_a(()) \}$
- $\{ \text{Lemma 3.4.19, } r(a) = 0, M \text{ is an } (\varepsilon\text{-free) NFRTA } \}$
- $\{ RSt_M(t) \}$

Case $t = a(t_1, \ldots, t_n), a \in \Sigma \setminus \Sigma_0$:

- $RSt_M(t)$
\[ \text{Lemma 3.4.19, } M' \text{ is a DFRTA } \]

\[ \langle \text{Set } q', \overline{q'} : R'_a(\overline{q'}) = q' \]
\[ \land \langle \forall i : 1 \leq i \leq n : \pi_i(\overline{q'}) \in RSt_{M'}(t_i) \rangle \}

\[ \text{Induction Hypothesis, set calculus } \]

\[ \langle \text{Set } q', \overline{q'} : R'_a(\overline{q'}) = q' \land \langle \forall i : 1 \leq i \leq n : \pi_i(\overline{q'}) = RSt_M(t_i) \rangle \}

\[ \text{set calculus } \]

\[ \{ R'_a(RSt_M(t_1), \ldots, RSt_M(t_n)) \} \]

\[ \text{set calculus } \]

\[ \{ \langle \bigcup \overline{q} : \langle \forall i : 1 \leq i \leq n : \pi_i(\overline{q}) \in RSt_M(t_i) \rangle : R_a(\overline{q}) \rangle \} \]

\[ \text{set calculus } \]

\[ \{ \langle \text{Set } q, \overline{q} : q \in R_a(\overline{q}) \land \langle \forall i : 1 \leq i \leq n : \pi_i(\overline{q}) \in RSt_M(t_i) \rangle : q \rangle \} \]

\[ \text{Lemma 3.4.19, } M \text{ is an (ε-free) NFRTA } \]

\[ \{ RSt_M(t) \} \]

\[ \text{RSt}_M(t) \]

\[ \text{RSt}_M(t) \cap Q_{ra} \neq \emptyset \]

\[ \text{use of proof given above } \]

\[ \{ \text{RSt}_M(t) \} \cap Q_{ra} \neq \emptyset \]

\[ \text{definition of } Q_{ra}, \text{ set calculus } \]

\[ \text{RSt}_M(t) \in \langle \text{Set } q' : q' \in Q' \land q' \cap Q_{ra} \neq \emptyset : q' \rangle \]

\[ \text{set calculus, } \text{RSt}_M(t) \subseteq Q, Q' = \mathcal{P}(Q) \]

\[ \text{RSt}_M(t) \cap Q_{ra} \neq \emptyset \]

\[ \text{definition } \text{acc}_M \]

\[ \text{acc}_M(t) \]

We now show that \( L(M) = L(M') \) by showing that \( \text{acc}_{M'}(t) = \text{acc}_M(t) \).
3.5 Equivalence, closure and decision problems

One of the most important results from regular string language theory is the equivalence of different characterizations of regular string languages: the formalisms regular tree grammars, regular expressions, and finite automata are all equivalent in generating and accepting power respectively. Other important results include closure properties of the regular languages—most importantly, they are closed under set operations—and decision problems on regular languages—e.g. decidability of emptiness and membership.

In Section 3.2, we defined regular tree languages. We then discussed generating and accepting formalisms for regular tree languages, without saying anything about the class of languages generated or accepted. The equivalences between regular tree languages and the languages generated or accepted by the various formalisms, their closure properties, and the decision problems of these languages are similar to those for regular string languages. They are extensively treated in the literature [TW68, Don70, Eng75b, Din87, GS97, CDG+07]. As the proofs and details involved are not very important for this dissertation, we omit them and merely mention some important properties and results. More detail can be found in the references cited above, with [Eng75b, GS97, CDG+07] providing the most complete treatments.

As in the string case, the class of languages recognizable by finite tree automata, $L_{Ta}$, is closed under set union, intersection, negation and difference, as well as under concatenation and language closure.

The classes $L_{Ta}$ and $L_{RTG}$ are the same, as in the regular string language case. In particular, for every RTG an equivalent TA can be constructed. Such constructions will be discussed in Chapter 5. A construction to obtain an equivalent RTG given a TA (actually, given an equivalent DFRTA) is given in e.g. [Din87, Stelling 5.2.1] and [CDG+07, Section 2.1.2].

The usual decision problems such as language membership, emptiness and finiteness, and inclusion of one language in another are all decidable for the tree case as well, as shown in e.g. [CDG+07, Section 1.7]. For these and less common decision problems, more details, decision algorithms and complexity discussions for the decidable problems can be found there and in [Eng75b, GS97].

3.6 Relations with string language theory

We mention some important results on the relations between regular tree language theory and string language theory. As with the theoretical results on equivalence, closure and decision problems in the preceding section, we refer the reader interested in more details on the subject to e.g. [Eng75b, GS97, CDG+07].

A number of relations between regular tree language theory and string language theory exist. First of all, regular tree language theory is of course a generalization of
regular string language theory, and any string can indeed be seen as a non-branching
tree. Many generalizations of parts of the latter theory are present in the various
definitions etc. in the current chapter or were pointed out by remarks in it (see for
example Remarks 3.1.13, 3.3.2, and 3.4.18).

Trees can also be seen as terms, i.e. in preorder form, as described in Section 3.1.1.
Since these terms can be generated by a context-free string grammar, each regular
tree language is also a context-free string language. This gives another relation
between tree and string language theory, yet one that will not play an important
role in this dissertation.

More interesting relations exist between regular tree languages and derivation trees
of context-free string languages, and between context-free string languages and the
yields of regular tree languages:

- The set of derivation trees of a context-free string grammar is a regular tree lan-
guage. (Yet there are regular tree languages that are not the set of derivation
trees of any context-free string grammar. For example, consider the single tree
regular tree language \(\{a(b(b(c), b(c)))\}\), which can be generated by an RTG with
a single production \(S \rightarrow a(b(b(c), b(c)))\). Seeing this tree as the derivation tree
of a context-free string grammar requires that \(a\) and \(b\) are nonterminals, and
that \(a \rightarrow bb, b \rightarrow b\) and \(b \rightarrow c\) are productions of this context-free string gram-
mar. In that case however, other derivation trees exist, including \(a(b(c), b(c))\)
and \(a(b(c), b(b(c)))\).

- Each context-free string language is the yield of a regular tree language, and
for every regular tree language, its yield is a context-free string language.
(However, the yield of some language being a context-free string language
does not imply that the former language is a regular tree language. As an
example, consider the context-free string language \(\{cc\}\), which is the yield
of a regular tree language \(\{a(c, c)\}\), but also of a non-regular tree language
\(\langle\text{Set } n : 1 \leq n : a(b^n(c), b^n(c))\rangle\).

Such results can be used—and in the literature are used—to give alternative proofs
of some results in context-free string language theory based on results in regular tree
language theory. We will not pay particular attention to them in this dissertation.

Finally, an important relation between regular tree language theory and string lan-
guage theory arises from the representation of a tree by its set of stringpaths, as
in Section 3.1. This representation and the subsequent use of string matching tech-
niques play an important role in particular branches of the taxonomies in this dis-
sertation. They will be discussed in detail in Section 6.7 for tree pattern matching
algorithms, while their use in tree acceptance algorithms will be discussed in Sec-
tion 5.8.
Part II

Taxonomies
Chapter 4

Taxonomy construction

In this short chapter, we discuss what taxonomies and taxonomy construction are in our context. Taxonomy construction forms the domain modeling step of the TAMBASCO domain modeling and domain engineering method, which is discussed in Chapter 7. After defining taxonomies and their construction in our context, we compare taxonomies to feature models, give a brief example of a taxonomy of keyword pattern matching algorithms, and discuss the advantages of taxonomy construction. In the next two chapters, the results of taxonomy construction for the tree acceptance and tree pattern matching problems will be discussed in detail.

4.1 Taxonomies and taxonomy construction

The Merriam Webster’s Collegiate Dictionary defines a taxonomy as:

[An] orderly classification of plants and animals according to their presumed natural relationships. [Mis93, p. 1208]

Although this definition is somewhat biology oriented, we can create a classification according to essential details of algorithms or data structures from a certain field as well. We focus on algorithm taxonomies here, but the approach can equally well be used for data structure taxonomies.

The main goal of constructing a taxonomy is to improve our understanding of the various algorithms in a certain problem field or domain, their correctness, and their mutual relations, i.e. their commonalities and variations. As input to the taxonomy construction step, we have the algorithms found during the literature survey step, rephrased in a common presentation style and notation to make comparison easier.

The starting point or root algorithm of the taxonomy is a high-level algorithm whose correctness is easily shown. A specification is associated with this and each of the
other algorithms, consisting of e.g. a pre- and postcondition and invariants\(^1\).

By adding details, refinements or variations of algorithms can be obtained, leading to algorithms from the literature or to new ones. Looking at taxonomy construction as a top-down process, the addition of a detail results in a refinement or variation of an algorithm solving the same or a closely related problem as the original algorithm does. Such an algorithm may appear in the literature or may be new. Every detail has associated correctness arguments, so that each algorithm’s correctness follows from the correctness of the starting point and details added.

The details which separate the various algorithms are essentially of two types:

- **Problem details** involve minor changes to pre- and postconditions, restricting algorithm input or output
- **Algorithm details** are used to specify variance in algorithmic structure and data representations used, either internally to an algorithm or influencing the representation of the input and output data as well. Note that they do not influence the meaning of the input or output; such changes are indicated by problem details. Such details may be further divided into algorithmic structure details and representation details, but we do not do so explicitly.\(^2\)

As an effect of the addition of such details, performance details for algorithms may change as well. Such details are about the variance in running time and memory usage among the algorithms. The theoretical running time and memory usage using ‘big-O’ notation may be explicitly mentioned with an algorithm, while information on practical performance will be obtained during the later benchmarking step of TABASCO. (For users of a toolkit resulting from the TABASCO process, performance details, followed by problem details, will be more important than algorithm details; a user may not care about the latter, as long as the algorithmic components provided perform well for his or her particular problem. The effect is that a mapping—by means of a DSL—from user requirements to toolkit components implementing algorithms is needed, something we discuss later on when describing DSL design and implementation.)

To indicate a particular algorithm and form a path in the taxonomy graph, we use the sequence of details in order of introduction. In some cases, it may be possible to derive an algorithm in multiple ways through the application of some details in a different order, leading to different detail sequences for one and the same algorithm. Depicting an algorithm taxonomy as a graph, in which nodes refer to algorithms and branches refer to details, the graph thus is a directed acyclic one. As an example, Figure 4.1.1 shows the taxonomy graph for a taxonomy of keyword pattern matching algorithms, which will be discussed further on in this section.

---

\(^1\)For data structure taxonomies, representation invariants are used, and pre- and postconditions are absent.

\(^2\)The division occurs in e.g. [CWK+06, CWKB05, CWZ04b], but not in e.g. [Cle93, Wat95].
4.1 Taxonomies and taxonomy construction

Taxonomies may also be—and often (partially) are—developed bottom-up instead of top-down: initially one may start with a number of algorithms in the problem domain literature, and as one sees commonalities among different algorithms, one may find generalizations of such algorithms which allow one to combine them into a taxonomy with the new generalization as the root and the other algorithms as leaves. This process of finding generalizations may of course be repeated. Once a complete taxonomy for all the algorithms has been constructed, the taxonomy and its correctness arguments will be presented in a top-down fashion.

We refrain from giving specific guidelines or rules for constructing taxonomies here: the choice of details—including their granularity—and of how to structure a taxonomy depends on a taxonomist’s preferences and understanding of the algorithms in a domain. (To help in the construction process, a so-called concept lattice capturing all the commonalities between the algorithms may first be constructed. This may help in making implicit commonalities and differences explicit. The use of concept lattices is briefly discussed in [CWK+06, Section 5.2].)

Due to the dependence on the taxonomist, a taxonomy is a taxonomy of a problem field, but not the taxonomy of that field, and taxonomists may end up with different taxonomies for the same field, depending on their understanding of and preference for emphasizing certain details of algorithms. This makes an algorithm taxonomy

![Figure 4.1.1 A taxonomy of keyword pattern matching algorithms.](image)
different from a biological one, whose structure seems to be dictated by ancestry. The reader should note that our taxonomies deviate from the kind of taxonomy usually used in biology in another aspect: the different details added to a single algorithm node in our taxonomies are not necessarily different choices in a single dimension, and may be choices in different dimensions. An example is the root of the taxonomy in Figure 4.1.1, from which details P and OKW lead to algorithms. These details correspond to considering prefixes of the text in some order and to restricting the problem to the single keyword case.

4.2 A taxonomy of pattern matching algorithms

As mentioned, Figure 4.1.1 shows a taxonomy of keyword pattern matching algorithms. The (exact) keyword pattern matching problem can be described as “the problem of finding all occurrences of keywords from a given set, as substrings in a given string” [Wat95]. This problem has been frequently studied in the past, and many different algorithms have been suggested for solving it. Watson and Zwaan [WZ96] and Watson [Wat95, Chapter 4] presented a taxonomy containing a set of well known solutions to the problem, including single-keyword Knuth-Morris-Pratt [KMP77] and Boyer-Moore algorithms [BM77], as well as the multiple-keyword Aho-Corasick [AC75] and Commentz-Walter algorithms [CW79a, CW79b]. The taxonomy was later extended by Clephas in [Cle03, Chapter 3] and is also discussed in [CWZ04a, CWZ04b]. The root node in the figure represents the high-level algorithm for solving this problem (which only specifies that the set of all matches is assigned to the output variable), while algorithms further down in the taxonomy present refinements, with the algorithms close to or at the taxonomy graph leaves representing concrete algorithms. For example, variants of the Aho-Corasick algorithm are found in the subtree labeled (P, +, E, SP, AC), while the Boyer-Moore algorithm and its variants are found by instantiating details MO, SL and MI in the subtree (OKW, OBM, INDICES).

To give an idea of what kind of details labeling branches occur in the taxonomy, we consider the path labeled (P, +, E, SP, AC). The branch labeled (P) means that the matches are found by considering prefixes of the text in some order, and detail (+) indicates that the prefixes are considered in increasing length order. Detail (E) indicates that matches are registered by their endpoint, while detail (SP) indicates that the set of suffixes of the currently attempted match that are prefixes of a keyword is maintained, in order to easily compute new matches. Finally, detail (AC) indicates that the longest suffix of the current attempted match that is still a prefix of a keyword is maintained (since the set of suffixes mentioned can be computed from this longest suffix).

As an example of algorithm factorization in the taxonomy, we now show how two

---

3To avoid confusion, some algorithms that are part of the taxonomy are not shown in the figure, since they are not relevant to the discussion here.
variants of the well-known Aho-Corasick algorithm as well as a multiple keyword
Knuth-Morris-Pratt algorithm are related. The example used is a revised version of
an example used in [CW05].

The example is meant to show how one may consider different but related algorithms
as refinements from a common algorithm skeleton. Our specific notation therefore is
less important, but the interested reader can find more information on the particular
details of this example, our algorithmic notation and the taxonomy as a whole in
the references cited at the beginning of the section.

Formally, the keyword pattern matching problem, given alphabet $V$ (a non-empty
finite set of symbols), input string $S \in V^*$, and finite non-empty pattern set $P =
\{p_0, p_1, \ldots, p_{|P|-1}\} \subseteq V^*$, is to establish postcondition $R_e$,

$$O_e = \langle \text{Set } u, v, r : ur = S \land v \in \text{suff}(u) \cap P : (v, r) \rangle,$$

that is to let $O_e$ be the set of tuples $(u, v)$ such that $u$ and $r$ are a splitting in two
parts of text $S$ (i.e. their concatenation equals $S$) and $v$ is a suffix of $u$ representing
a keyword occurrence. New matches are registered whenever $\text{suff}(u) \cap P \neq \emptyset$,
i.e. whenever the intersection of $u$’s suffixes and the pattern set $P$ is non-empty.

The essential idea of both the Aho-Corasick and Knuth-Morris-Pratt algorithms is
the introduction of an easily updateable state variable, named $q$ in the following
algorithm, that gives information about (partial) matches in $\text{suff}(u)$ and allows easy
computation of the set $\text{suff}(u) \cap P$. Adding the previously explained details ($P$),
(+), ($\emptyset$), (SP) and (AC) to the high-level algorithm at the top of the taxonomy graph
gives us the following algorithm skeleton:

Algorithm 4.2.1($P, +, \emptyset, \text{SP}, \text{AC}$)

\[
u, r := \varepsilon, S; q := \varepsilon;
O_e := \text{Output}(q) \times S;
\{ \text{invariant: } ur = S
\land O_e = \langle \text{Set } x, y, z : xz = S \land x \leq_p u \land y \in \text{suff}(x) \cap P : (y, z) \rangle
\land q = \langle \text{MAX} \leq_p w : w \in \text{suff}(u) \cap \text{pref}(P) : w \rangle
\}
d o r \neq \varepsilon \rightarrow
q := \langle \text{MAX} \leq_p w : w \in \text{suff}(q(r[1])) \cap \text{pref}(P) : w \rangle;
u, r := u(r[1]), r[1];
O_e := O_e \cup \text{Output}(q) \times \{r\}
do \{ R_e \}
\]

We will not go into the details of our notation and how this algorithm skeleton
is obtained. Note however that the first statement in the do-loop updates $q$ to
represent the longest suffix $w$ of the part $q(r\uparrow 1)$ of $S$ read, that is also a prefix of a keyword from $P$.

As can be seen in Figure 4.1.1, this algorithm leads to algorithms $(P_+, E, SP, AC, AC\text{-}OPT)$, $(P_+, E, SP, AC, LS, AC\text{-}FAIL)$ and $(P_+, E, SP, AC, LS, KMP\text{-}FAIL)$, representing the Optimal Aho-Corasick, Failure function Aho-Corasick and multiple keyword Knuth-Morris-Pratt algorithms respectively. These algorithms differ only in the way that the above statement is further refined in order to be implementable in an imperative programming language like C++ or Java.

The Optimal Aho-Corasick algorithm [AC75, section 6] uses a precomputed goto function to compute the new value of state $q$. Using this function, the update of $q$ is replaced by:

$$q := \gamma_f(q, r\uparrow 1);$$

For the Failure function Aho-Corasick algorithm, the update to $q$ becomes:

$$\text{do } \tau_{ef}(q, r\uparrow 1) = \perp \rightarrow q := f_{\gamma}(q) \text{ od;}$$
$$q := \tau_{ef}(q, r\uparrow 1);$$

For the multiple keyword version of the Knuth-Morris-Pratt algorithm [KMP77], the statement is replaced by:

$$\text{do } q \neq \perp, \text{ cand } q(r\uparrow 1) \notin \text{pref}(P) \rightarrow q := f_{\gamma}(q) \text{ od;}$$
$$q := q(r\uparrow 1)\max_{\leq \varepsilon};$$

Once again, the details of our notation are not important here. The reader should notice though, that we can thus consider the three algorithms as further refinements of a common algorithm skeleton.

### 4.3 Taxonomies and feature models

Since a taxonomy is constructed based on an understanding of the algorithms in a particular domain as appearing in the literature, their commonalities and variations, a taxonomy also represents an understanding of a problem domain: it is a kind of domain model (also see Section 7.1). In [CE00], such models are not constructed based on a formally derived taxonomy of the algorithms, but based on feature models. Feature models consist of feature diagrams—depicting the variable as well as the common features in the domain—as well as constraints, defaults and optimizations regarding the combination of such features—forming the configuration knowledge. In taxonomy-based domain models, the configuration knowledge often is not described explicitly, but is implicit in the taxonomy and correctness arguments. Figure 4.3.1 shows an example of a feature diagram for the domain of sorting algorithms [CF02].
4.4 Advantages of taxonomy construction

Depending on the problem domain under consideration, one might choose the taxonomy construction approach over the feature modeling one, or vice versa. For a field like that of sorting algorithms, in which the working of the various algorithms is not very complex and their relationships are already well known from literature, it might not be necessary or worthwhile to construct a taxonomy. A feature model with a description of the configuration knowledge might be a better solution in this case. On the other hand, feature models are not well-suited for the description of pre- and postconditions and correctness arguments belonging to algorithms or data structures. For domains in which the algorithms are more complex or their relationships to each other are not clear—as was the case for the keyword pattern matching domain before the construction of our taxonomy—, it is wise to use the taxonomy approach.

4.4 Advantages of taxonomy construction

A number of deficiencies that are meant to be solved by creating a domain model in the form of a formal taxonomy of algorithms were mentioned in Section 1.2. Here, we discuss the advantages of taxonomy construction, including ones that are related to solving the deficiencies.

- It brings order to a domain and makes it more accessible, gathering the various algorithms in a common presentation. The taxonomy can serve as a structured survey of the problem domain and as a teaching aid.
• It makes it easier to compare algorithms. Given some algorithms in the taxonomy, we can determine what they have in common and how they differ by comparing the paths from the taxonomy root to the algorithms. The comparison is also made easier by the absence of unnecessary details and the uniform presentation style and language used for the algorithms.

• It gives a clear and correct presentation of the algorithms in a problem domain, providing correctness arguments for each of the algorithms. Such arguments are often absent in the literature.

• Furthermore, it is well suited for discovering new algorithms. The process of taxonomy construction may reveal gaps in the taxonomy, which may suggest new algorithms. In developing these, the use of formal methods in the taxonomy construction process may prove useful.

• Lastly, it gives a formal specification of the algorithms. For each algorithm, we have a pre- and postcondition and invariants, as well as a specification of the interface it provides. Furthermore, related algorithms in the taxonomy will have related specifications. The specification forms an important part of the configuration knowledge. Capturing this knowledge is useful for mapping user requirements to toolkit components, as discussed in Section 7.5.

Apart from these advantages, the availability of the taxonomy aids in the construction of a toolkit for the domain under consideration, as we will see in Chapters 7 and 8.
Chapter 5

Tree acceptance

In Chapter 1, the tree acceptance problem was introduced as one of the two important and related problems which we consider in this dissertation and which underly many applications of regular tree language theory. In this chapter we consider various algorithms solving this problem.

The presentation of the various algorithms solving the problem takes the form of a taxonomy. This taxonomy is derived by stepwise refinement, according to the taxonomy construction step of the TABASCO process, as described in Chapter 7. Many of the algorithm details that are introduced in the current chapter will occur again in the discussion of a similar taxonomy of tree pattern matching algorithms in Chapter 6.

5.1 Taxonomy overview

Figure 5.1.1 shows the graph structure of the taxonomy of tree acceptance algorithms. As explained in Chapter 7, vertices correspond to (possibly abstract) algorithms while edges are labeled by details. The details and algorithms/vertices in the taxonomy will be introduced further on this chapter. A list of the details together with a brief description of each is also given at the end of the current section.

Each algorithm or vertex in the taxonomy graph is uniquely identified by the sequence of labels along the path from the graph’s root to the particular vertex, as mentioned in Chapter 7. Furthermore, an algorithm or page number is included for each, referring to the algorithm or to the page on which it is discussed. The taxonomy graph together with the table of details thus also serve as an alternative table of contents to this chapter.

The solutions to the tree acceptance problem presented as part of the taxonomy can
be divided into three main categories, corresponding to the three main branches of the taxonomy:

- **Algorithms using tree automata as acceptors.** These correspond to the taxonomy part starting with detail (T-ACCEPTOR). Instead of already detailing the complete structure of such automata, merely a few properties of such automata are assumed initially.

Detail (T-ACCEPTOR) and the first algorithm in this category, using an undirected nondeterministic tree automaton, are discussed in Section 5.3. By introducing more and more details and applying these to the algorithm, we obtain algorithms using automata with less nondeterminism that are directed either root-to-frontier or frontier-to-root—discussed in Sections 5.4 and 5.5.

The construction of the tree automata is considered separately in Section 5.6. By choosing which state set to use, whether \( \varepsilon \)-transitions are included, whether the automata should be undirected, root-to-frontier or frontier-to-root directed, and—for directed automata—whether the automata should be deterministic or not, a number of constructions are obtained. Many of these appear in the literature.

(In Section 5.7, attention will be given to practical aspects of constructions resulting in DFRTAS, as these can be constructed for any RTG and can be used efficiently.)

- **Algorithms based on the notion of match sets.** These algorithms correspond to the part of the taxonomy—depicted in Figure 5.1.1—that starts with the detail (MATCH-SET). Using this detail, the subset of all so-called Items is computed from which the subject tree can be derived. The problem can then be solved by determining whether start symbol \( S \) is in this match set.
The computation of such match sets is based on a recursive definition of match sets. This part of the taxonomy, starting with the detail sequence (MATCH-SET, REC), is discussed in Section 5.7. For efficiency reasons, match sets can be tabulated, so match set values do not need to be recomputed for every (sub)tree, regardless of whether some subtrees occur more than once.

We show that the recursive computation of match set values corresponds to the computation of one kind of frontier-to-root tree automata whose construction will be treated in Section 5.6, thus linking this category with the first category of algorithms. Whether explicitly based on tree automata or not, many solutions in the literature are based on tabulation of match sets. After establishing this link, imperative versions of the match set tabulation/DFRTA construction are treated. To reduce the size of the transition tables of such DFRTAs, the concept of filtering is treated. Some experimental results on the construction time and size of the unfiltered and filtered automata are discussed in Section 8.6 of the toolkit chapter.

- **Algorithms based on stringpath matching.** These algorithms occur in the taxonomy part starting with detail (S-PATH), briefly introduced in Section 5.3, and considered further in Section 5.8. They are based on decomposing trees into stringpaths and using stringpath matching techniques.

  In fact, we can and will describe the algorithms as extensions of those for stringpath-based tree pattern matching that are discussed in Section 6.7.

Other ways of solving the tree acceptance problem exist. For example, one could enumerate the trees generated by the RTG in order of increasing height, until all trees of height equal to t’s height have been generated or t has been generated. We do not consider such solutions here as they seem rather inefficient.

**Remark 5.1.1.** We will not explicitly consider time and space complexity of the algorithms in the current chapter. They are considered briefly in Chapter 6: near the end of the taxonomy overview at the beginning of that chapter for the tree pattern matching algorithms and in Section 6.6 for the match set tabulation/DFRTA construction algorithms. As is apparent when comparing the two taxonomies, acceptance and pattern matching algorithms with similar algorithm details in the two taxonomies are quite similar, and so are the tabulation/construction algorithms described in the two taxonomy chapters. As a result, time and space complexity of the algorithms are similar as well, and a complexity analysis for the algorithms in the current chapter would yield similar results to those discussed in Chapter 6.

Each detail labeling the branches of the taxonomy together with a brief description is listed in the following table:
<table>
<thead>
<tr>
<th>T-ACCEPTOR</th>
<th>Use a tree automaton accepting the language of an RTG.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF</td>
<td>Consider transition relations of a tree automaton in a root-to-frontier direction.</td>
</tr>
<tr>
<td>FR</td>
<td>Consider transition relations of a tree automaton in a frontier-to-root direction.</td>
</tr>
<tr>
<td>DET</td>
<td>Use a deterministic version of an automaton.</td>
</tr>
<tr>
<td>MATCH-SET</td>
<td>Use an item set and a match set function indicating the set of items from which a tree is derivable.</td>
</tr>
<tr>
<td>REC</td>
<td>Recursively compute match set values.</td>
</tr>
<tr>
<td>FILTER</td>
<td>Use a filtering function in the computation of match set function values.</td>
</tr>
<tr>
<td>TABULATE</td>
<td>Use a tabulated version of the match set function, in which a bijection is used to identify match sets by integers.</td>
</tr>
<tr>
<td>S-PATH</td>
<td>Decompose production right hand sides into stringpaths and determine right hand sides and nonterminals deriving the subject tree based on matching stringpaths.</td>
</tr>
<tr>
<td>SP-MATCHER</td>
<td>Use an automaton as a pattern matcher for a set of stringpaths in a root-to-frontier subject tree traversal.</td>
</tr>
<tr>
<td>ACA-SPM</td>
<td>Use an (optimal) Aho-Corasick automaton and define transition and output functions in terms of that automaton.</td>
</tr>
<tr>
<td>DRFTA-SPM</td>
<td>Use a DRFTA as a stringpath matcher and define transition and output functions in terms of that automaton.</td>
</tr>
</tbody>
</table>

The details used for describing the construction algorithms for tree acceptors will be described in Section 5.6, while the various filtering functions for detail FILTER will be described in Section 5.7.

### 5.2 Running example

We use the following RTG as a starting point for our running example in this chapter. It is the same grammar used as a running example in [Din87].

**Example 5.2.1** (Starting point for running example). Let \( G = (N, \Sigma, r, \text{Prods}, S) \) where

- \( N = \{S, B\} \),
- \( \Sigma = \{a, b, c, d\} \),
- \( r = \{(S, 0), (B, 0), (a, 2), (b, 1), (c, 0), (d, 0)\} \),
- \( \text{Prods} = \{(1) \quad S \rightarrow \begin{array}{c} a \end{array}, \quad (2) \quad S \rightarrow \begin{array}{c} a \end{array}, \quad (3) \quad S \rightarrow c, \quad \begin{array}{c} b \\ c \end{array} \begin{array}{c} B \\ d \end{array} \begin{array}{c} B \end{array} \} \)
\begin{equation}
(4) \quad B \to b, \quad (5) \quad B \to S, \quad (6) \quad B \to d, \quad B
\end{equation}

Note that we have numbered the production rules for ease of reference. Grammar \(G\) has characteristic values \(z_+\) and \(u_+\) (see Section 3.3.1).

As mentioned in Chapter 3, we assume RTGs to have no useless productions and nonterminals. In addition, before applying any tree acceptor construction, we add a new start symbol \(S'\) and production \(S' \to S\) to an RTG. We do this to simplify definitions later on: it ensures that \(S\) is part of the set of subtrees of right hand sides, which may not be the case in the absence of \(S' \to S\). We use \(N'\) for \(N \cup \{S'\}\), \(r'\) for \(r \cup \{(S', 0)\}\), and \(P'\) for \(Prd \cup \{S' \to S\}\). We call the resulting RTG augmented, similar to what is done for comparably extended context-free string grammars.

**Example 5.2.2** (Running example). Let \(G' = (N', \Sigma, r', Prd, S')\) be the augmented version of the RTG of Example 5.2.1. We assume production \(S' \to S\) to be numbered (0), consistent with the numbering in the original RTG.

**Remark 5.2.3.** RTGs without chain rules (i.e. with characteristic value \(u_+\), see Section 3.3.1) become augmented RTGs with a single chain rule. The only chain rule is the newly added rule \(S' \to S\). We can ignore this chain rule when determining derivations, since we are not interested in derivability from this new symbol \(S'\). We will therefore still refer to such RTGs as \(u_+\) RTGs.

### 5.3 The problem and three naive solutions

In this section we define the tree acceptance problem more formally and present a number of abstract solutions whose correctness is easily established. None of them are very practical, but they provide the top part of the taxonomy, from which different parts containing more concrete and practical algorithms can be derived. The part of the taxonomy discussed in the current section is depicted using solid lines in Figure 5.3.1.

In Chapter 1, the tree acceptance problem was informally stated as follows:

Given a regular tree grammar and an input tree, determine whether the input tree can be generated by the regular tree grammar, i.e. is an element of the language denoted by the regular tree grammar.

We define the problem more formally as follows:

**Definition 5.3.1** (Tree (Grammar) Acceptance (TGA)) \(^1\). The TGA problem for an

\(^1\)We use Tree (Grammar) Acceptance and its abbreviation to TGA since TA is already used as an abbreviation for Tree Automaton.
RTG $G = (N, \Sigma, r, Prods, S)$ is

Given an input tree $t \in Tr(\Sigma, r)$, determine whether $t \in \mathcal{L}(G)$.

In the literature, this problem is sometimes called the membership problem for the language of $G$. Based on this problem definition, we can state a first solution in algorithmic form:

**Algorithm 5.3.2()**

\[
\begin{array}{l}
\text{// const } G = (N',\Sigma,r',Prods',S') : \text{augmented RTG;}
\text{ const } t : Tr(\Sigma, r);
\text{ var } b : \mathbb{B}
\text{ | } b := t \in \mathcal{L}(G)
\text{ | } \{ b \equiv t \in \mathcal{L}(G) \}
\end{array}
\]

Since this algorithm forms the root of the taxonomy, it is identified by the empty sequence of details.

Algorithm (i) is trivially correct yet highly abstract. One way to obtain more concrete solutions is by introducing a tree automaton as an acceptor, similar to the use of a string automaton as an acceptor for the regular (string) language acceptance problem.

**Detail 5.3.3 (T-ACCEPTOR)**. Use a tree automaton accepting the language of an RTG.
Algorithm 5.3.4 (T-ACCEPTOR)

\[
\begin{align*}
\langle \text{const } G &= (N', \Sigma, r', \text{Prods}', S') : \text{augmented RTG}; \\
& t : \text{Tr}(\Sigma, r); \\
\text{var } b : \mathbb{B} \\
\langle \text{let } M &= (Q, \Sigma, r, R, Q_0) \text{ be a TA such that } \mathcal{L}(M) = \mathcal{L}(G); \\
& b : = t \in \mathcal{L}(M) \\
\langle b \equiv t \in \mathcal{L}(G) \rangle \\
\rangle
\end{align*}
\]

Algorithm 5.3.4 does not specify how to determine whether \( t \in \mathcal{L}(M) \). Based on Definition 3.4.17 for the language of a tree automaton, this can be determined by considering all state assignments to \( t \) that respect the transition relations \( R \), in some order, and considering the states assigned to root node \( \varepsilon \) of \( t \) in such assignments. This does not seem particularly efficient, and we will therefore consider algorithms using acceptors that are directed root-to-frontier or frontier-to-root in Sections 5.4 and 5.5.

The algorithm also does not specify how the acceptor \( M \) is to be constructed. In Section 5.6 constructions for (undirected as well as directed) tree acceptors satisfying the requirement \( \mathcal{L}(M) = \mathcal{L}(G) \) will be detailed.

A second way to obtain more concrete algorithms from Algorithm () is by introducing the concept of a match set, indicating the set of Items from which a tree is derivable. For now, we assume this set to be the set of subtrees of production RHSes. (Note that this set includes \( S \), since we use augmented RTGs.)

**Detail 5.3.5 (MATCH-SET).** Use an item set and a match set function indicating the set of items from which a tree is derivable. \( \square \)

Algorithm 5.3.6 (MATCH-SET)

\[
\begin{align*}
\langle \text{const } G &= (N', \Sigma, r', \text{Prods}', S') : \text{augmented RTG}; \\
& t : \text{Tr}(\Sigma, r); \\
\text{var } b : \mathbb{B} \\
\langle \text{let } Items &= \text{Subtrees}(\text{RHS}(\text{Prods}')); \\
\text{let } MS &\in \text{Tr}(\Sigma, r) \rightarrow \mathcal{P}(\text{Items}) \text{ such that } \\
& MS(t) = \left\{ \text{Set } p : p \in \text{Items} \land p \Rightarrow t : p \right\}; \\
& b : = S \in MS(t) \\
\langle b \equiv t \in \mathcal{L}(G) \rangle \\
\rangle
\end{align*}
\]
Algorithm 5.3.6 does not specify how to compute $MS(t)$. The property stated for $MS$ is expressed in terms of $\Rightarrow$, a relation which can be defined recursively in terms of subtrees of a tree. It therefore seems natural to define $MS$ recursively as well. This approach to computing match sets and algorithms based on the approach will be discussed in Section 5.7.

**Remark 5.3.7.** In the remainder of this chapter, we will encounter functions like $MS$ that take a tree as a parameter. When considering pseudo-code implementing such functions however, we will consider the tree to be accessible as a global value and use a node of the tree as a parameter. This is done for two reasons:

- In practical applications, tree representations usually are based on objects or records representing nodes and pointers linking them together. It is then more efficient to pass a reference or pointer to a node than to pass (a copy of) a subtree;
- Passing a node and having the entire tree available as a global value instead of passing a subtree provides context information, which will turn out to be useful for algorithms based on stringpath matching.

For function $MS$, this comes down to using a signature $D_t \rightarrow \mathcal{P}(Items)$ (recall that $D_t$ is the domain of tree $t$) instead of $Tr(\Sigma, r) \rightarrow \mathcal{P}(Items)$.

As a third solution strategy, each production right hand side can be decomposed into a set of stringpaths. Keyword pattern matching techniques can then be used to match occurrences of stringpaths of the right hand sides in a subject tree. A right hand side then matches at a node if and only if all its stringpaths match at such a node (with occurrences of nonterminals matching at a node if and only if such a nonterminal may derive the subtree at this node).

This approach was originated by Hoffmann & O’Donnell [HO82b], who use Aho-Corasick automata to recognize the stringpaths. As the approach extends a similar stringpath-based tree pattern matching approach, we base our discussion on the latter. In Section 5.8, the reader is therefor first referred to Section 6.7 on stringpath-based tree pattern matching, before the extension to tree acceptance is discussed briefly.

### 5.4 Using root-to-frontier tree acceptors

Figure 5.4.1 highlights the part of the taxonomy discussed here. Literature references for algorithms using root-to-frontier tree acceptors will be omitted here and instead be given in Section 5.6.5 when considering constructions for such acceptors.

The following detail corresponds to the use of $\varepsilon$NRFTAS (as in Definition 3.4.5):
5.4 Using root-to-frontier tree acceptors

Figure 5.4.1 Tree acceptance taxonomy, with solid lines and circles emphasizing taxonomy parts described in Section 5.4.

**Detail 5.4.1 (rf)**. Consider transition relations of a tree automaton in a root-to-frontier direction.

Based on detail (rf), we can use the root-to-frontier acceptance function $\text{Accept} \in Tr(\Sigma, r) \times Q \rightarrow \mathbb{B}$ of Definition 3.4.20 to compute whether a tree is in the language of an acceptor.

Without loss of generality, we assume an $\varepsilon$NRFTA to have a single root accepting state, i.e., we assume $Q_{\text{ra}} = \{q_s\}$ for some state $q_s$.

We now obtain the following algorithm, in which $t$ is a global value and function $\text{Traverse}$ is such that $\text{Traverse}(n, q) = \text{Accept}(t/n, q)$ holds. (Note that in function $\text{Traverse}$, $n$ represents the rank of the symbol $a$ being considered, as per Convention 3.1.9. Furthermore, parameter $n$ has type $D$ instead of $D_t$, as the function is defined independent of a particular tree $t$. The precondition requires that $n$ is part of $D_t$, the tree domain of $t$.)

**Algorithm 5.4.2 (t-acceptor, rf)**

```plaintext
const G = (N', \Sigma, r', Prods', S') : augmented RTG;
t : Tr(\Sigma, r);
var b : \mathbb{B}
let M = (Q, \Sigma, r, R, \{q_s\}) be an \varepsilon\text{NRFTA such that } \mathcal{L}(M) = \mathcal{L}(G);

b := Traverse(\varepsilon, q_s)
{ b \equiv t \in \mathcal{L}(G) }

func Traverse(\bot n : D, \bot q : \mathbb{B} =
{ Pre: n \in D_t }
||
| let a = t(n);
| Traverse := false;
```
if $n > 0 \rightarrow$
\[
\begin{array}{l}
\text{for } (q_1, \ldots, q_n) : (q_1, \ldots, q_n) \in R_a(R^*_a(q)) \rightarrow \\
\quad \{ \langle \forall i : 1 \leq i \leq n : n \cdot i \in D_t \rangle \}
\end{array}
\]

$Traverse :=$
\[
Traverse \lor (Traverse(n \cdot 1, q_1) \land \cdots \land Traverse(n \cdot n, q_n))
\]

$\text{rof}$
\[
| n = 0 \rightarrow \\
\quad \text{as } () \in R_a(R^*_a(q)) \rightarrow Traverse := true \text{ sa}
\]

$\triangledown \{ \text{Post: } Traverse \equiv \text{Accept}(t/n, q) \}$

Note that the assignment to $Traverse$ in function $Traverse$ uses a conjunction of booleans resulting from recursive calls to function $Traverse$, which is used in a disjunction determined by the for-loop surrounding the assignment. The recursive calls of $Traverse$ could thus be stopped as soon as one of the values for the conjunction is determined to be false, while iteration of the loop can be stopped as soon as one of the values for the disjunction is determined to be true.

To further improve the efficiency of function $Traverse$, memoization could be used, storing function values to prevent recomputation.

### 5.4.1 Using more specific root-to-frontier tree acceptors

Algorithm (T-ACCEPTOR, RF) assumes the use of an $\varepsilon$NRFTA. In case an $(\varepsilon$-free) NRFTA is used instead, the occurrences of $R_a(R^*_a(q))$ in function $Traverse$ reduce to $R_a(q)$, but the algorithm does not change otherwise.

In case a DRFTA is used, the version of $Traverse$ used can be further simplified.

**Detail 5.4.3 (DET).** Use a deterministic version of an automaton.

We give the resulting Algorithm (T-ACCEPTOR, RF, DET):

**Algorithm 5.4.4 (T-ACCEPTOR, RF, DET)**

\[
\begin{array}{l}
| \text{const } G = (N', \Sigma, r', \text{Prods}', \Sigma') : \text{augmented RTG;} \\
\quad t : Tr(\Sigma, r); \\
\quad \var b : \mathbb{B} \\
| \text{let } M = (Q, \Sigma, r, R, \{ q_s \}) \text{ be a DRFTA such that } \mathcal{L}(M) = \mathcal{L}(G); \\
\quad b := Traverse(\varepsilon, q_s) \\
\quad \{ b \equiv t \in \mathcal{L}(G) \}
\end{array}
\]
5.5 Using frontier-to-root tree acceptors

As was mentioned in Subsection 3.4.3, DRFTAs are less powerful than NRFTAs. As a result, a DRFTA $M$ as in this algorithm may not exist for a given RTG. The condition on the RTG for such a DRFTA to exist will be discussed with the automata constructions in Section 5.6.

5.5 Using frontier-to-root tree acceptors

Figure 5.5.1 Tree acceptance taxonomy, with solid lines and circles emphasizing taxonomy parts described in Section 5.5.

Figure 5.5.1 highlights the part of the taxonomy discussed here. Literature references for algorithms using frontier-to-root tree acceptors will be omitted here and instead be given in Section 5.6.6 when considering constructions for such acceptors.

The following detail corresponds to the use of $\varepsilon$NRFTAs (as in Definition 3.4.6):

**Detail 5.5.1 (FR).** Consider transition relations of a tree automaton in a frontier-to-root direction.

\[
\text{func } \text{Traverse} (\downarrow n : D, \downarrow q : Q) : B = \\
\{ \text{Pre: } n \in D_t \} \\
\| \\
\| \text{let } a = t(n); \\
\| \text{if } n > 0 \rightarrow \\
\| \text{let } (q_1, \ldots, q_n) = R_a(q); \\
\| \{ \forall i : 1 \leq i \leq n : n \cdot i \in D_t \} \\
\| \text{Traverse} := \text{Traverse}(n \cdot 1, q_1) \land \ldots \land \text{Traverse}(n \cdot n, q_n) \\
\| n = 0 \rightarrow \\
\| \text{Traverse} := (()) = R_a(q) \\
\| \} \\
\| \{ \text{Post: } \text{Traverse} \equiv \text{Accept}(t/n, q) \} \\
\|}
\]
Based on detail (FR), we can compute whether a tree is in the language of an acceptor based on using the recursive version of function \( RSt \) of Lemma 3.4.19.

We now obtain the following algorithm, in which function \( \text{Traverse} \) is such that \( \text{Traverse}(n) = RSt(t/n) \):

**Algorithm 5.5.2 (T-ACCEPTOR, FR)**

\[
\begin{aligned}
\text{|| const } G &= (N', \Sigma, r', \text{Prods}', S') : \text{augmented RTG;} \\
& \text{let } G' : = (N, \Sigma, r, \text{Prods}, S') \in \text{RTG} \\
\text{|| var } b : \mathbb{B} \\
\text{let } M &= (Q, \Sigma, r, R, Q_{ra}) \text{ be an} \in\text{FRTA such that } L(M) = L(G); \\
& b : = \text{Traverse}(\varepsilon) \cap Q_{ra} \neq \emptyset \\
& \text{let } a = t(n); \\
\text{func Traverse}(\downarrow n : D) : \mathcal{P}(Q) = \\
\text{Pre: } n \in D_i \\
\text{|| var } s_1, \ldots, s_n : \mathcal{P}(Q) \\
\text{let } a = t(n); \\
\text{if } n > 0 \rightarrow \\
\text{for } i : 1 \leq i \leq n \rightarrow \\
\quad \{ n \cdot i \in D_i \} \\
\quad s_i : = \text{Traverse}(n \cdot i) \\
\quad \{ s_i = RSt(t/(n \cdot i)) \} \\
\text{rof:} \\
\quad \{ (\forall i : 1 \leq i \leq n : s_i = RSt(t/(n \cdot i))) \} \\
\text{Traverse} : = \emptyset; \\
\text{for } (q_1, \ldots, q_n) : q_1 \in s_1, \ldots, q_n \in s_n \rightarrow \\
\quad \text{Traverse} : = \text{Traverse} \cup R^*_\varepsilon(R_a(q_1, \ldots, q_n)) \\
\text{rof} \\
& n = 0 \rightarrow \\
\text{Traverse} : = R^*_\varepsilon(R_a()) \\
\text{fi} \\
\text{||{ Post: Traverse = RSt(t/n) } }
\end{aligned}
\]

5.5.1 Using more specific frontier-to-root tree acceptors

Algorithm (T-ACCEPTOR, FR) assumes the use of an \( \in\text{FRTA} \). In case an (\( \varepsilon \)-free) \( \text{FRTA} \) is used instead, the occurrences of \( R^*_\varepsilon(R_a(q_1, \ldots, q_n)) \) and of \( R^*_\varepsilon(R_a()) \) in function \( \text{Traverse} \) reduce to \( R_a(q_1, \ldots, q_n) \) and to \( R_a() \), but the algorithm does not change otherwise.
In case a DFRTA is used, the version of Traverse used can be further simplified.

**Detail 5.5.3 (DET).** Use a deterministic version of an automaton.

We give the resulting Algorithm (T-ACCEPTOR, FR, DET):

**Algorithm 5.5.4 (T-ACCEPTOR, FR, DET)**

\[
\begin{align*}
|| \textbf{const} & \quad G = (N', \Sigma, r', \text{Prods}', S') : \text{ augmented RTG}; \\
& \quad t : \text{Tr} (\Sigma, r); \\
& \quad \textbf{var} \quad b : \mathbb{E} \\
| \textbf{let} \quad M = (Q, \Sigma, r, R, Q_{ra}) \text{ be a DFRTA such that } \mathcal{L}(M) = \mathcal{L}(G); \\
& \quad b := \text{Traverse}(\varepsilon) \in Q_{ra} \\
& \quad \{ \quad b \equiv t \in \mathcal{L}(G) \} \\
\textbf{func} \quad \text{Traverse}(\downarrow n : D) : Q = \\
& \quad \{ \quad \text{Pre: } n \in D_t \} \\
& \quad || \textbf{var} \quad q_1, \ldots, q_n : Q \\
& \quad | \textbf{let} \quad a = t(n); \\
& \quad \text{if } n > 0 \rightarrow \\
& \quad \quad \{ \quad \langle \forall i : 1 \leq i \leq n : n \cdot i \in D_t \rangle \} \\
& \quad \quad \text{Traverse} := R_a (\text{Traverse}(n \cdot 1), \ldots, \text{Traverse}(n \cdot n)) \\
& \quad \quad \| \quad n = 0 \rightarrow \\
& \quad \quad \text{Traverse} := R_a () \\
& \quad \textbf{fi} \\
& \quad ||\{ \quad \text{Post: } \{ \text{Traverse} \} = RSt (t/n) \} \\
\end{align*}
\]

### 5.6 Constructing tree acceptors

In this section, we consider various constructions resulting in tree acceptors, i.e., we consider constructions for various kinds of tree automata \( M \) such that \( \mathcal{L}(M) = \mathcal{L}(G) \), given an augmented RTG \( G = (N', \Sigma, r', \text{Prods}', S') \).

Algorithm (T-ACCEPTOR) and algorithms derived from it all use such acceptors. Depending on the algorithm, such a tree automaton may need to be undirected or directed either root-to-frontier or frontier-to-root, and directed automata may need to be nondeterministic or deterministic.

The constructions we present differ in a number of aspects:

- Which of three particular state sets introduced—labeled ALL-SUB, PROPER-S, and PROPER-N—is used,
• whether \( \varepsilon \)-\textit{transitions} are present or not—the latter being indicated by label \text{REM-}\varepsilon,

• whether the resulting automata are \textit{undirected}, \textit{root-to-frontier directed} (indicated by label \text{RF}), \textit{frontier-to-root directed} (indicated by \text{FR}) or \textit{undirected}, and

• in the case of constructions for \( \varepsilon \)-\textit{free} directed automata, whether the resulting automata are \textit{deterministic} or not—indicated by label \text{SUBSET}_{\text{RF}} or \text{SUBSET}_{\text{FR}} depending on the direction.

By combining all possible choices for these aspects, a total of twenty four different constructions can be obtained. Roughly half of these are treated in this section, seeming most interesting, usually because they (form the basis for constructions that) occur in the literature. The other ones are left implicit but are easily inferred.

Constructions are identified by the sequence of labels indicated, preceded by label \text{TGA-\textsc{ta}} to indicate that they are constructions for \textsc{tas} solving the \text{TGA} problem. For example, constructions include Construction \((\text{TGA-\textsc{ta}:\text{all-sub}})\) and \((\text{TGA-\textsc{ta:proper-n:REM-}\varepsilon:\text{FR:SUBSET}_{\text{FR}}})\). The table below lists each of the discussed constructions together with references to the section in which they are discussed.

<table>
<thead>
<tr>
<th>Construction</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{TGA-\textsc{ta:all-sub}}))</td>
<td>Section 5.6.1: Construction 5.6.2</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:all-sub:rf}}))</td>
<td>Section 5.6.5.1</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:all-sub:fr}}))</td>
<td>Section 5.6.6.1</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:all-sub:rem-}\varepsilon}))</td>
<td>Section 5.6.2: Construction 5.6.10</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:all-sub:rem-}\varepsilon:rf}))</td>
<td>Section 5.6.5.2</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:all-sub:rem-}\varepsilon:fr}))</td>
<td>Section 5.6.6.2</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:all-sub:rem-}\varepsilon:fr:subset}_{\text{RF}}))</td>
<td>Section 5.6.6.4</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:proper-s:rem-}\varepsilon}))</td>
<td>Section 5.6.3: Construction 5.6.15</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:proper-n:rem-}\varepsilon}))</td>
<td>Section 5.6.4: Construction 5.6.19</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:proper-n:rem-}\varepsilon:rf}))</td>
<td>Section 5.6.5.3</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:proper-n:rem-}\varepsilon:fr}))</td>
<td>Section 5.6.6.3</td>
</tr>
<tr>
<td>((\text{TGA-\textsc{ta:proper-n:rem-}\varepsilon:fr:subset}_{\text{fr}}))</td>
<td>Section 5.6.6.4</td>
</tr>
</tbody>
</table>

We start with a basic construction for undirected \textsc{tas} in Section 5.6.1. States of the resulting \textsc{ta} correspond to subtrees of production right hand sides, i.e. the \textit{Items} of Algorithm 5.3.6 (\textsc{match-set}). By applying \( \varepsilon \)-removal function \text{rem-}\varepsilon in Section 5.6.2 and/or restricting the state/item set in Sections 5.6.3 and 5.6.4, various related \textsc{ta} constructions are obtained.

Following the undirected \textsc{ta} constructions, we consider ones for root-to-frontier directed automata in Section 5.6.5. By considering the automata from the undirected constructions as root-to-frontier automata and/or applying the root-to-frontier subset construction we obtain various such constructions. For the correctness of those constructions resulting in \textsc{drftas}, we give a condition the input \textsc{rtg} has to satisfy.
We consider constructions for frontier-to-root directed automata in Section 5.6.6. By considering automata as frontier-to-root automata and/or applying the frontier-to-root subset construction we again obtain various constructions.

Recall that a tree automaton is defined by a 5-tuple \((Q, \Sigma, r, R, Q_{ra})\). Since the RTG \(G\) already defines ranked alphabet \((N' \cup \Sigma, r')\) and hence \((\Sigma, r)\), the automaton constructions only specify \(Q, R\) and \(Q_{ra}\) to completely define a tree automaton.

The constructions are all presented in a similar way, starting with some introductory text and the definition of any new details involved in the construction. The construction is then defined formally by specifying \(Q, R\) and \(Q_{ra}\), followed by an example of its application, a discussion of its correctness and finally a comparison to related constructions and reference to occurrences of the construction in the literature. Depending on the particular construction, one or more of these may be omitted, e.g. when they are obvious or irrelevant.

### 5.6.1 A first construction for undirected tree automata

The first construction we consider is based on the use of all subtrees of right hand sides as states. Its transition relations encode the relations between (tuples of) such states, based on the relation between a tree, say \(p\), and its direct subtrees, say \(p_1, \ldots, p_n\), on the symbol at the root of this tree \(p\), and the relation between a production left hand side, say \(A\), and right hand side, say \(\alpha\).

The intuition behind this and other constructions is that the state(s) assigned to the root of some tree \(t\) in a tree state assignment based on a tree automaton correspond(s) to the subtree(s) of right hand sides from which \(t\) can be derived. After presenting the construction and giving an example, we prove this in Lemma 5.6.5.

**Detail 5.6.1 (ALL-SUB).** Use all subtrees of production RHSes as states in a construction.

**Construction 5.6.2 (TGA-TA:ALL-SUB).**

**Input** (Augmented) RTG \(G = (N', \Sigma, r', \text{Prods}', S')\) of type \((\mathbb{Z}_+ \cup \mathbb{Z}_-)\)

**Output** TA \(M = (Q, \Sigma, r, R, Q_{ra})\) where

\[
Q = \text{Items} = \text{Subtrees}(\text{RHS(Prods')})
\]

\[
R_a = \left\{ p : p = a(p_1, \ldots, p_n) \land p, p_1, \ldots, p_n \in \text{Items} \right\}
\]

( for all \(a \in \Sigma\) )

\[
R_e = \left\{ \left. \text{Set } A, \alpha : A, \alpha \in \text{Items} \land A \rightarrow \alpha \in \text{Prods} \right| (A, \alpha) \right\}
\]

\[
Q_{ra} = \{ S \}
\]

**Note** that in this construction, \(R_e\) corresponds to Prods (and not Prods') considered as a relation (and thus excludes a transition \((S', S))\).
**Figure 5.6.1** TA resulting from Construction 5.6.2 (TGA-TA:ALL-SUB).

**Example 5.6.3.** Applying Construction 5.6.2 (TGA-TA:ALL-SUB) to the augmented RTG of Example 5.2.2 results in the TA depicted in Figure 5.6.1. Note that the diagram is undirected, i.e. a state may be depicted to the right of a state tuple to which it is connected by a transition. As an example, consider state $q_5$ and (the tuple consisting of) single state $q_6$, connected by $b$: this depicts a transition $(q_5, (q_6)) \in R_b$, not a transition $(q_6, (q_5)) \in R_b$ (also see Figure 3.4.1).

The correspondence between state labels used and subtrees they represent is:

\[
\begin{align*}
q_s &= S & q_0 &= b & q_1 &= B & q_2 &= d \\
q_3 &= a & q_4 &= a & q_5 &= b & q_6 &= c \\
& & B & d & c & B \\
& & b & c & B \\
\end{align*}
\]

The transition relations are as follows:

\[
\begin{align*}
R_a &= \{ (q_3, (q_1, q_2)), (q_4, (q_5, q_1)) \} & R_c &= \{ (q_6, ()) \} & R_e &= \{ (q_s, q_3), (q_s, q_4), (q_s, q_5), (q_1, q_1), (q_1, q_2) \} \\
R_b &= \{ (q_5, (q_6)), (q_0, (q_1)) \} & R_d &= \{ (q_2, ()) \} & &
\end{align*}
\]

We compute the possible state assignments for trees $a(b(c), d)$ and $a(d, c)$. We use a semicolon to separate on the one hand those states $q$ reached on a symbol transition for a (tuple of) child node state(s) and on the other hand those states reached from such states $q$ by one or more $\varepsilon$-transitions i.e. states reached by $\varepsilon$-closure. For example, $RS_t(c) = \{ q_6; q_s, q_1 \}$ since there is a transition from the empty tuple () to $q_6$ on $c$, while $q_s$ is related to $q_6$ by an $\varepsilon$-transition and $q_1$ in turn is related to $q_s$ by
an $\varepsilon$-transition.

$$
\begin{align*}
RSt(c) &= \{q_6; q_s, q_1\} \\
RSt(d) &= \{q_2; q_1\} \\
RSt(b(c)) &= \{q_5, q_6; q_1\} \\
RSt(a(b(c), d)) &= \{q_3, q_4; q_s, q_1\} \\
RSt(a(d, c)) &= \emptyset
\end{align*}
$$

As $q_s \in RSt(a(b(c), d))$ yet $q_s \notin RSt(a(d, c))$, only the first tree is accepted. \qed

**Lemma 5.6.4.** For every $\alpha, \beta \in Q$ we have

$$(\alpha, \beta) \in R^*_\varepsilon \equiv \alpha \xrightarrow{\varepsilon} \beta.$$ 

**Proof.** To prove this property, we prove a stronger one, namely that for every $k \geq 0$ and $\alpha, \beta \in Q$

$$(\alpha, \beta) \in R^k_\varepsilon \equiv \alpha \xrightarrow{k \varepsilon} \beta.$$ 

We prove this property by induction on $k$. For $k = 0$, this is trivial.

For $k > 0$ we derive:

$$\begin{align*}
\alpha R^k_\varepsilon \beta &
\equiv \{ k > 0 \} \\
& \langle \exists \gamma : \gamma \in Q : \alpha R_\varepsilon \gamma \land \gamma R^{k-1}_\varepsilon \beta \rangle \\
& \equiv \{ \text{Induction Hypothesis} \} \\
& \langle \exists \gamma : \gamma \in Q : \alpha R_\varepsilon \gamma \land \gamma R^{k-1}_\varepsilon \beta \rangle \\
& \equiv \{ \Rightarrow: \text{definition of } R_\varepsilon; \leftrightarrow: \alpha \in Q, \text{ Prods } \subseteq \text{ Prods}', \text{ definition } Q, R_\varepsilon \} \\
& \langle \exists \gamma : \alpha \rightarrow \gamma \in \text{ Prods} : \gamma R^{k-1}_\varepsilon \beta \rangle \\
& \equiv \{ k > 0 \} \\
& \alpha \xrightarrow{k \varepsilon} \beta
\end{align*}$$

\qed

**Lemma 5.6.5.** For every $t \in Tr(\Sigma, r)$, $\alpha \in Q$ we have $\alpha \in RSt_M(t) \equiv \alpha \xrightarrow{\varepsilon} t$.

**Proof.** We prove this property by induction on the depth of $t$.

Case $t = a \in \Sigma_0$:

$$
\begin{align*}
\alpha \in RSt_M(t) &
\equiv \{ \text{Lemma 3.4.19} \} \\
(\alpha, ()) \in R^*_\varepsilon \circ R_a
\end{align*}
$$
\[\exists q : q \in Q : \forall \alpha, q \in R^*_a \land (q, \alpha) \in R_a\]

\[\exists q : q \in Q : (\alpha, q) \in R^*_a \land q = a\]

\[\exists q : q \in Q : \alpha \not\Rightarrow q \land q = a\]

\[\alpha \not\Rightarrow a\]

Case \(t = a(t_1, \ldots, t_n), a \in \Sigma \setminus \Sigma_0:\)

\[\alpha \in RSt_M(t)\]

\[\exists q, \bar{q} \in Q^n : (\alpha, \bar{q}) \in R^*_a \land (q, \bar{q}) \in R_a \land \forall i : 1 \leq i \leq n : \pi_i(q) \in RSt_M(t_i)\]

\[\exists q, \bar{q} \in Q^n : (\alpha, q) \in R^*_a \land (a, q) \in R_a \land \forall i : 1 \leq i \leq n : \pi_i(q) \in RSt_M(t_i)\]

\[\exists q, \bar{q} : q = a(\pi_1(q), \ldots, \pi_n(q)) \land \forall i : 1 \leq i \leq n : \pi_i(q) \Rightarrow t_i\]

\[\exists q : q \in Q : \alpha \not\Rightarrow q \land q(\varepsilon) = a \land q \Rightarrow a(t_1, \ldots, t_n)\]

\[\alpha \not\Rightarrow a(t_1, \ldots, t_n)\]
Using Lemma 5.6.5 and the definitions of the language of a TA respectively an RTG we obtain the following result.

**Corollary 5.6.6.** \( \mathcal{L}(M) = \mathcal{L}(G). \)

The tree automata resulting from Construction (TGA-TA:ALL-SUB) may thus be used with Algorithm 5.3.4 (T-ACCEPTOR).

**Literature reference 5.6.7.** Construction (TGA-TA:ALL-SUB) does not occur in the literature, but forms the basis for other constructions discussed further on, many of which do appear in the literature.

### 5.6.1.1 Relation to match set computation

Match set function \( MS \) introduced in Algorithm 5.3.6 (MATCH-SET) and the computation of a TA resulting from Construction (TGA-TA:ALL-SUB) can be related quite easily:

**Lemma 5.6.8.** Let \( MS \) be as in Algorithm 5.3.6 (MATCH-SET) and let \( M \) be constructed according to Construction (TGA-TA:ALL-SUB), both using the same augmented RTG \( G \) as input, then for every \( t \in Tr(\Sigma, r) \), \( MS(t) = RSt_M(t) \).

**Proof.** We derive

\[
   MS(t) = \begin{cases} \text{ definition of } MS \text{ in Algorithm (MATCH-SET) } & \\ \langle \text{Set } p : p \in Items \land p \xRightarrow{t} p \rangle & \\ \text{ \{ Items } = Q \text{ in construction; Lemma 5.6.5, } RSt_M(t) \subseteq Q \text{ \}} & \\ RSt_M(t) & \\ \end{cases}
\]

As removing \( \varepsilon \)-transitions or directing transitions in an automaton does not influence set \( Q \) or \( RSt_M(t) \), this lemma also holds for constructions obtained from Construction (TGA-TA:ALL-SUB) by applying \( \varepsilon \)-removal or considering the automata to be directed. Such constructions are discussed further on.

### 5.6.2 A construction without \( \varepsilon \)-transitions

Construction (TGA-TA:ALL-SUB) results in automata with \( \varepsilon \)-transitions. These transitions are used to relate nonterminals appearing as left hand sides and corresponding right hand sides. For implementation reasons, it may be advantageous to do away with such transitions. Most of the nondeterministic tree acceptor constructions appearing in the literature indeed do so.
The removal of \(\varepsilon\)-transitions from a tree automaton is a straightforward generalization of the string automaton case. Any \(\varepsilon\)-transitions can be removed from a TA by applying transformation \(\text{REM}-\varepsilon\), which was discussed in Section 3.4.3.

In practice, the transformation is applied directly as part of a construction. This simultaneously replaces the various relations \(R_a\) and relation \(R_\varepsilon\) by \(R_\varepsilon^* \circ R_a\) and \(\emptyset\) respectively.

**Detail 5.6.9 (REM-\(\varepsilon\)).** Apply \(\text{REM}-\varepsilon\) in a tree automaton construction.

With some additional calculations to rewrite the definitions, we obtain the following construction resulting in \(\varepsilon\)-free TAs.

### Construction 5.6.10 (TGA-TA:ALL-SUB:REM-\(\varepsilon\)).

**Input** (Augmented) RTG \(G = (N', \Sigma, r', \text{Prods}', S')\) of type \((\mathbb{Z}_+ \cup \mathbb{N})\)

**Output** \(\varepsilon\)-free TA \(M = (Q, \Sigma, r, R, Q_{ra})\) where

\[
Q = \text{Items} = \text{Subtrees(RHS(Prods'))} \\
R_a = \left\{ \text{Set } p, p_1, \ldots, p_n : p = a(p_1, \ldots, p_n) \land p, p_1, \ldots, p_n \in \text{Items} \right\} \\
\cup \left\{ \text{Set } A, B, p, p_1, \ldots, p_n : p = a(p_1, \ldots, p_n) \land p, p_1, \ldots, p_n \in \text{Items} \land A, B \in N \land B \Rightarrow \ A \Rightarrow p \right\} \\
\text{for all } a \in \Sigma \\
R_\varepsilon = \emptyset \\
Q_{ra} = \{ S \}
\]

**Example 5.6.11.** Applying Construction 5.6.10 (TGA-TA:ALL-SUB:REM-\(\varepsilon\)) to the augmented RTG of Example 5.2.2 results in the \(\varepsilon\)-free TA depicted in Figure 5.6.2. The correspondence between state labels and subtrees is the same as in Example 5.6.3, while the transition relations are as follows:

\[
R_a = \{ (q_3, (q_1, q_2)), (q_4, (q_5, q_1)), (q_5, (q_1, q_2)), (q_5, (q_5, q_1)) \} \\
R_\varepsilon = \{ (q_6, ()), (q_7, ()), (q_1, ()) \} \\
R_d = \{ (q_2, ()), (q_1, ()) \} \\
R_b = \{ (q_5, (q_6)), (q_0, (q_1)), (q_1, (q_1)) \}
\]
5.6 Constructing tree acceptors

The correctness of the construction is obvious, as transformation REM-$\varepsilon$ does not change the language recognized by a TA.

**Literature reference 5.6.12.** Construction (TGA-TA:ALL-SUB:REM-$\varepsilon$) is similar to the $\varepsilon$-free TA construction of Ferdinand, Seidl & Wilhelm [FSW94, Section 6]. That construction defines the state set as $N \cup \text{Subtrees}(\text{RHS}(\text{Prods}))$. When the original RTG has no useless nonterminals, this merely ensures that start symbol $S$ is in the state set. In our construction, this is ensured by augmenting the original RTG by production $S' \rightarrow S$. The automata resulting from their construction on an unmodified RTG and resulting from our construction on an augmented RTG are isomorphic.

### 5.6.3 A construction without unreachable states

The example tree automaton in Figure 5.6.2 shows that $\varepsilon$-free TAS resulting from Construction 5.6.10 (TGA-TA:ALL-SUB:REM-$\varepsilon$) may have states that are root accepting state-unreachable (i.e. not in $\text{RReach}$, see Section 3.4.2). In the example, $q_0$, $q_1$ and $q_4$ are such states. Such states are useless for determining tree acceptance, i.e. for solving the TGA problem using a tree automaton.

These root accepting state-unreachable states can be removed by applying transformation REM-UNR$_{\varepsilon}$ to a TA. This transformation was already discussed in Section 3.4.2. Similar to what was done for REM-$\varepsilon$, one could apply this transformation as part of a construction directly.

We can characterize the set of root accepting state-unreachable states more directly however: as the following example clarifies, such states correspond to full RHSes that do not occur as proper subtrees of any RHS and that are different from $S$. (Due to the definition of $Q_{\varepsilon}a$ in the construction as $\{S\}$, state $S$ is always root accepting state-reachable, regardless of its occurrence as a proper subtree.)
Figure 5.6.3 TA with full RHS becoming root accepting state-unreachable when applying REM-ε.

Example 5.6.13. Consider the highlighted part of the TA of Figure 5.6.3 (which corresponds to the TA of Figure 5.6.1). The ε-transition between $q_s : S$ and $q_3 : a(B, d)$ corresponds to the production $S \rightarrow a(B, d)$. When applying REM-ε to the TA, the transition is replaced by one on a between $q_s : S$ and the state tuple $(q_1 : B, q_2 : d)$ corresponding to the direct subtrees of RHS $a(B, d)$. Since $q_3 : a(B, d)$ is not part of any state tuple related to some state by a transition on an alphabet symbol (i.e. does not occur as a proper subtree of a RHS), and since the ε-transition to be removed is the only such transition leading to the state, there are no transitions leading to the state and it therefore certainly is not root accepting-state reachable (i.e. reachable from state $q_s : S$ in the example).

Detail 5.6.14 (PROPER-S). Use the original RTG’s start symbol as well as all proper subtrees of RHSes as states in a construction.

This detail allows us to define Construction (TGA-TA:PROPER-S:REM-ε) as below, which differs from the earlier construction only in the state set and the restriction of the transitions to this new state set. Note that a construction with detail PROPER-S yet without detail REM-ε would not yield a usable acceptor: in the example, the resulting acceptor would not have states $q_0$, $q_3$ and $q_4$, yet no symbol transitions relating root accepting state $q_s$ to tuples of states would exist either.


Input (Augmented) RTG $G = (N', \Sigma, r', \text{Prods}', S')$ of type $(Z_+, \Pi_+)$

Output ε-free TA $M = (Q, \Sigma, r, R, Q_{ra})$ where
5.6 Constructing tree acceptors

\[
Q = \{S\} \cup \text{ProperSubtrees}(\text{RHS}(\text{Prods}'))
\]

\[
R_a = \left\{ \begin{align*}
\text{Set } p, p_1, \ldots, p_n : p &= a(p_1, \ldots, p_n) : (p, (p_1, \ldots, p_n)) \\
&\quad \land p, p_1, \ldots, p_n \in Q
\end{align*} \right\}
\]

\[
\cup
\left\{ \begin{align*}
\text{Set } A, B, p, p_1, \ldots, p_n : p &= a(p_1, \ldots, p_n) : (B, (p_1, \ldots, p_n)) \\
&\quad \land p_1, \ldots, p_n \in Q \\
&\quad \land A \in N \land B \in N \cap Q \\
&\quad \land B \Rightarrow A \Rightarrow p
\end{align*} \right\}
\]

\[
R_\varepsilon = \emptyset
\]

\[
Q_{ra} = \{S\}
\]

Example 5.6.16. Applying Construction 5.6.15 (TGA-TA:PROPER-S:REM-\varepsilon) to the augmented RTG of Example 5.2.2 results in the \varepsilon-free TA in Figure 5.6.4. The correspondence between state labels used and subtrees they represent is:

\[
\begin{align*}
q_s &= S \\
q_0 &= B \\
q_1 &= d \\
q_2 &= b \\
q_3 &= c
\end{align*}
\]

As automata resulting from this construction are isomorphic to those resulting from applying transformation REM-UNRx to automata resulting from Construction 5.6.10 (TGA-TA:ALL-SUB:REM-\varepsilon), and as accepting computations only involve states that are reachable from the root accepting state, it is clear that automata resulting from the construction can be used to solve the TGA problem.

\[\text{Figure 5.6.4 } \varepsilon\text{-free TA resulting from Construction 5.6.15 (TGA-TA:PROPER-S:REM-\varepsilon)}\]
Literature reference 5.6.17. The construction given above does not appear in the literature.

5.6.4 A construction including all nonterminals as states

In general, elements of $N \setminus \{S\}$ may be among the root accepting state-unreachable and hence excluded states in the preceding construction. Some constructions in the literature result in automata in which proper subtrees of right hand sides and all elements of $N$ are states.

Such automata can be used to determine derivability from any nonterminal versus only from an RTG’s start symbol, which is important in code selection, where there may not be a single unique start symbol.

Detail 5.6.18 (proper-N). Use the original RTG’s nonterminals as well as all proper subtrees of RHSes as states in a construction.

This detail allows us to define Construction (TGA-TA:proper-N:REM-ε) as follows:

**Construction 5.6.19 (TGA-TA:proper-N:REM-ε).**

**Input** (Augmented) RTG $G = (N', \Sigma, r', Prods', S')$ of type $(Z_+U_+)$

**Output** TA $M = (Q, \Sigma, r, R, Q_{ra})$ where

$\begin{align*}
Q &= N \cup \text{ProperSubtrees(RHS}(Prods')) \\
R_a &= \left\langle \text{Set } p, p_1, \ldots, p_n : p = a(p_1, \ldots, p_n) \quad \text{if } p, (p_1, \ldots, p_n) \in Q \right\rangle \\
&\quad \cup \\
&\left\langle \text{Set } A, B, p, p_1, \ldots, p_n : p = a(p_1, \ldots, p_n) \quad \text{if } p, (p_1, \ldots, p_n) \in Q \right\rangle \\
&\quad \text{if } p_1, \ldots, p_n \in Q \\
&\quad \text{if } A \in N \land B \in N \\
&\quad \text{if } B \Rightarrow A \Rightarrow p \\
R_\varepsilon &= \emptyset \\
Q_{ra} &= \{S\}
\end{align*}$

Note that the definition of this construction and that of Construction 5.6.10 (TGA-TA:all-sub:REM-ε) are indeed almost the same: this construction merely has a different state set and, relatedly, $p$ in the second set quantification for $R_a$ is not necessarily a state (since it may correspond to a full RHS that is not a proper subtree of any RHS).

**Example 5.6.20.** Applying Construction 5.6.19 (TGA-TA:proper-N:REM-ε) to the augmented RTG of Example 5.2.2 results in the same ε-free TA as in Example 5.6.16, depicted in Figure 5.6.4. The constructions result in the same automata for this
example because nonterminal $B$—the only element of $N \setminus \{ S \}$—is already included in set $\text{ProperSubtrees}(\text{RHS}(\text{Prods}'))$. When considering an extension of the augmented RTG with a new nonterminal $C$ and productions $B \rightarrow C$ and $C \rightarrow S$, a state corresponding to $C$ (and transitions for this state) would be constructed by Construction (TGA-TA:PROPER-N:REM-$\varepsilon$) but not by Construction 5.6.15 (TGA-TA:PROPER-S:REM-$\varepsilon$).

**Literature reference 5.6.21.** Construction 5.6.19 (TGA-TA:PROPER-N:REM-$\varepsilon$) for an augmented RTG corresponds to the improved $\varepsilon$-free TA construction presented in [FSW94, Section 6] for the corresponding original RTG. The construction also appears in [WM95, Section 11.7].

### 5.6.5 Constructions for root-to-frontier tree acceptors

In Section 5.4 we introduced algorithm detail RF, which is the consideration of transition relations in a root-to-frontier direction. We consider the constructions resulting from combining detail RF with the preceding constructions. To name the root-to-frontier-directed constructions, we append detail RF to the detail sequence of the respective original construction (although the order of this detail compared to the other details is not relevant).

Since the detail merely changes the way in which transitions are viewed, we do not explicitly present the formal definitions of the resulting constructions, and the examples and depictions of tree automata given here are merely directed versions of earlier examples.

The most interesting part of this section then is the discussion of literature references for the resulting constructions, as well as the discussion of deterministic constructions in Section 5.6.5.4.

#### 5.6.5.1 A first construction

**Example 5.6.22.** Applying Construction (TGA-TA:ALL-SUB:RF) to the augmented RTG of Example 5.2.2 results in the $\varepsilon$NRFTA in Figure 5.6.5, a directed version of the TA in Figure 5.6.1. The correspondence between state labels and trees they represent is as in Example 5.6.3.

**Literature reference 5.6.23.** Construction (TGA-TA:ALL-SUB:RF) for an augmented RTG corresponds to the $\varepsilon$NRFTA construction van Dinther gives in [Din87, Section 5.4.1] for the underlying unmodified RTG, with two apparent differences:

- A tree automaton in her construction uses signature $Q \rightarrow Q$ for symbols of rank 0 instead of $Q \rightarrow Q^0 = Q \rightarrow \mathbb{1}$ as we do. The rightmost $Q$ in the former signature is restricted to a unique additional frontier state and corresponds to the use of the empty tuple in our tree automaton definition.
Her construction explicitly adds all elements of \( N \) to the state set. Since we add a production \( S' \rightarrow S \) and assume all nonterminals to be useful, the definition of the state set in our construction already includes them.

\[\square\]

5.6.5.2 A construction without \( \varepsilon \)-transitions

Example 5.6.24. The result of applying Construction (TGA-TA:ALL-SUB:REM-\( \varepsilon \)-RF) to the augmented RTG of Example 5.2.2 is the NRFTA in Figure 5.6.6, a directed version of the TA in Figure 5.6.2. Note that the construction may result in an acceptor with root accepting state-unreachable states, as is the case with states \( q_0 \), \( q_3 \) and \( q_4 \) in this example.

\[\square\]
5.6 Constructing tree acceptors

**Literature reference 5.6.25.** Construction \(\text{TGA-TA:ALL-SUB:REM-} \varepsilon: \text{RF}\) does not appear in the literature. We presented an example nevertheless, as the construction forms the basis for other constructions that do.

\(\text{(Construction \(\text{TGA-TA:PROPER-S:REM-}\varepsilon: \text{RF}\) does not appear in the literature and is omitted.)}\)

### 5.6.5.3 A construction including all nonterminals as states

**Example 5.6.26.** Applying Construction\(\text{TGA-TA:PROPER-N:REM-} \varepsilon: \text{RF}\) to Example 5.2.2’s augmented RTG results in the NRFTA in Figure 5.6.7, a directed version of the TA in Figure 5.6.4. Note that state identifiers are as in Example 5.6.16 and differ from those in Example 5.6.24.

---

**Figure 5.6.7 NRFTA resulting from Construction \(\text{TGA-TA:PROPER-N:REM-} \varepsilon: \text{RF}\)**

---

**Literature reference 5.6.27.** We now consider construction \(\text{TGA-TA:PROPER-N:REM-} \varepsilon: \text{RF}\) and the effect on it of using an (augmented) \(\text{Z-U-} \) RTG:

- The state set corresponds to \(N\): Since productions are necessarily of the form \(A \rightarrow a(A_1, \ldots, A_n)\), set \(\text{ProperSubtrees}(\text{RHS}(\text{Prods}) )\) is a subset of \(N\).

- The transition function relates each nonterminal left hand side to the set of tuples of nonterminals occurring as proper subtrees of a corresponding right hand side.

The state set and transition relation set then correspond to those in the NRFTA construction for unmodified type \(\text{Z-U-}\) RTGs given in [CDG+07, Section 2.1.2], except that that construction explicitly defines the state set to be \(N\). For general RTGs the construction does not appear in the literature.
5.6.5.4 Deterministic constructions

As indicated in Section 3.4.3, deterministic root-to-frontier tree automata are less powerful than εNRFTAs and NRFTAs, and the subset construction on an NRFTA leads to an equivalent DRFTA if and only if the NRFTA’s language is path closed (see Definition 3.1.43).

A condition on an RTG may also be given that ensures that a DRFTA for the same language can be constructed. This condition appears in the work of Viragh on root-to-frontier tree automata [Vir81] for type (2,1,1,) RTGs, but is easily augmented to RTGs in general. Van Dinther [Din87, Section 6.3.4] essentially discusses the same condition and notes that it is similar to the LL(1) condition for string grammars.

Definition 5.6.28 (Deterministic RTG). An RTG $G = (N, \Sigma, r, Prods, S)$ is called deterministic if and only if for every $A \in N$ and $a \in \Sigma$, there is at most one production $B \to a(\alpha_1, \ldots, \alpha_n)$ with $B \in N$ such that $A \xrightarrow{r} B$ and $\alpha_1, \ldots, \alpha_n \in Tr(N \cup \Sigma, r)$.

Example 5.6.29. The RTG of Example 5.2.2 is not deterministic, due to the presence of the two rules (1) $S \to a(B, d)$, (2) $S \to a(b(c), B)$. An RTG from which one of these rules is removed would be deterministic.

Lemma 5.6.30. Given a deterministic RTG $G$, applying subset to any of the previously discussed constructions for NRFTAs results in constructions yielding DRFTAs accepting $L(G)$.

Proof idea. Due to the condition given for deterministic RTGs, it is clear that for every state, the constructions will yield at most one (composition of ε-transitions and symbol) transition per alphabet symbol to some state tuple.

For deterministic RTGs, constructions for DRFTAs can thus be obtained by applying detail subset to the underlying constructions (TGA-TA:ALL-SUB:REM-ε:RF), (TGA-TA:PROPER-S:REM-ε:RF) and (TGA-TA:PROPER-S:REM-ε:RF).

Literature reference 5.6.31. None of the constructions for DRFTAs were explicitly found in the literature. Van Dinther does discuss a condition on an RTG corresponding to the RTG being deterministic [Din87, Section 6.3.4]. She uses this to obtain a construction for a deterministic root-to-frontier parser as an extension and modification of an acceptor resulting from construction (TGA-TA:ALL-SUB:RF) discussed in Section 5.6.5.1.

5.6.6 Constructions for frontier-to-root tree acceptors

In Section 5.5 we introduced algorithm detail FR, which is the consideration of transition relations in a frontier-to-root direction. We consider the constructions resulting from combining detail FR with the preceding constructions. Since the detail merely changes the way in which transitions are viewed, we do not explicitly
present the formal definitions of the resulting construction, and the examples and depictions of tree automata given here are merely frontier-to-root directed versions of earlier examples.

As in Section 5.6.5 then, the most interesting part of this section is the discussion of literature references for the resulting constructions, as well as the discussion of deterministic constructions in Section 5.6.6.4.

5.6.6.1 A first construction

**Example 5.6.32.** The result of applying Construction (TGA-TA:ALL-SUB:FR ) to the augmented RTG of Example 5.2.2 is the εNFRTA in Figure 5.6.8. Its undirected and root-to-frontier directed version appear in Figures 5.6.1 and 5.6.5. The correspondence between state labels and trees they represent is as in Example 5.6.3. ♣

**Literature reference 5.6.33.** Construction (TGA-TA:ALL-SUB:FR) for an augmented RTG corresponds to the εNFRTA construction in [Din87, Section 5.3.1] for the underlying unmodified RTG, except for two apparent differences:

- The tree automata in her construction use signature \( \Sigma \rightarrow \Sigma \) for symbols of rank 0 instead of \( \sigma \rightarrow \Sigma = \mathbb{1} \rightarrow \Sigma \) as we do. The leftmost \( \Sigma \) in the former signature is restricted to a unique additional frontier state and corresponds to the use of the empty tuple in our tree automata definitions.

- Her construction explicitly adds all elements of \( N \) to the state set. Since we add a production \( S' \rightarrow S \) and assume all nonterminals to be useful, the definition of the state set in our construction already includes them.

♣
5.6.6.2 A construction without \( \varepsilon \)-transitions

**Example 5.6.34.** The result of applying Construction (TGA-TA:ALL-SUB:REM-\( \varepsilon \):FR) to the augmented RTG of Example 5.2.2 is the NFRTA depicted in Figure 5.6.9. Its undirected and root-to-frontier directed version appear in Figures 5.6.2 and 5.6.6. The correspondence between the state labels used and the trees they represent is the same as in Example 5.6.11. Note that the construction may result in a tree acceptor in which the root accepting state is no longer reachable from some states, as is the case with states \( q_0 \), \( q_3 \) and \( q_4 \) in this example.

**Literature reference 5.6.35.** Construction (TGA-TA:ALL-SUB:REM-\( \varepsilon \):FR) does not appear in the literature by itself, but is included as it forms the basis for a number of related constructions we discuss further on.

(Construction (TGA-TA:PROPER-S:REM-\( \varepsilon \):FR) does not appear in the literature and is omitted.)

5.6.6.3 A construction including all nonterminals as states

**Example 5.6.36.** Applying Construction (TGA-TA:PROPER-N:REM-\( \varepsilon \):FR) to Example 5.2.2’s augmented RTG results in the NFRTA in Figure 5.6.10. Its undirected and root-to-frontier directed version appear in Figures 5.6.4 and 5.6.7. Note that state identifiers are as in Example 5.6.16 and differ from those in Example 5.6.34.

**Literature reference 5.6.37.** We consider the effect on Construction (TGA-TA:PROPER-N:REM-\( \varepsilon \):FR) of using a \( (Z \_ U \_) \) RTG (augmented by production \( S' \rightarrow S \)):
Figure 5.6.10 NFRTA resulting from Construction (TGA-TA:PROPER-N:REM-ε:FR)

- The state set corresponds to $N$: Since productions in a $(z_u)_-$ RTG are necessarily of the form $A \rightarrow a(A_1, \ldots, A_n)$, set $\text{ProperSubtrees}(\text{RHS}(\text{Prods}'))$ is a subset of $N$.

- The transition function is simplified, relating tuples of nonterminals occurring as proper subtrees of a right hand side to the corresponding left hand side.

The state set and transition relation set then correspond to those in the NFRTA construction for unmodified type $(z_u)_-$ RTGs first described by Brainerd in [Bra67, Lemma 4.5] and [Bra69, Lemma 4.4] and also appearing in [Din87, Section 5.1.1], with two apparent differences:

- As was noted for Construction (TGA-TA:ALL-SUB:FR), transitions for symbols of rank 0 are in $Q \rightarrow Q$ in van Dinther’s definition of tree automata, instead of in $Q^0 \rightarrow Q = 1 \rightarrow Q$ as in ours; in Brainerd’s, they are in $Q$ (which is isomorphic with $1 \rightarrow Q$).

- Their constructions explicitly define the state set to be $N$, while our construction defines it to be $N \cup \text{ProperSubtrees}(\text{RHS}(\text{Prods}'))$. For type $(z_u)_-$ RTGs, the two sets are the same due to the first item of the preceding bullet list. For other RTG types the sets are not the same in general. Their constructions thus restrict input grammars to be of type $(z_u)_-$.

\[\square\]

5.6.6.4 Deterministic constructions

We can obtain constructions resulting in DFRTAs—colloquially called deterministic constructions—by applying subset construction $\text{SUBSET}_FR$ to NFRTA constructions
like the ones presented above. Subset construction \( \text{subset}_{\text{FR}} \) was defined in Subsection 3.4.3 and may in general result in a size explosion, due to the resulting DFRTA having \( 2^{|Q|} \) states where \( |Q| \) is the number of states of the underlying NFRTA. Often however many of the subsets forming DFRTA states are not reachable, and a reachability-based construction is used in practice. An imperative algorithm implementing this construction will be discussed in Section 5.7.

Construction \( \text{ Constr } (\text{TGA-TA:ALL-SUB:REM-} \varepsilon \text{-FR:subset}_{\text{FR}}) \) is the first DFRTA construction we consider.

**Example 5.6.38.** Applying Construction \( \text{ Constr } (\text{TGA-TA:ALL-SUB:REM-} \varepsilon \text{-FR:subset}_{\text{FR}}) \) to the augmented RTG of Example 5.2.2 results in the DFRTA depicted in Figure 5.6.11 (in which only reachable subsets are depicted). The tabulation of the DFRTA will be presented in Section 5.7 as part of Example 5.7.23.

![Figure 5.6.11 DFRTA resulting from Construction (TGA-TA:ALL-SUB:REM-\( \varepsilon \)-FR:subsetFR). Note that the root accepting states (indicated by a double border) are precisely the states containing root accepting state \( q_s \) : \( S \) of the underlying NFRTA.](image)

**Literature reference 5.6.39.** The DFRTA construction given by Hemerik & Katoen in [HK89, Section 4] corresponds to Construction \( \text{ Constr } (\text{TGA-TA:ALL-SUB:REM-} \varepsilon \text{-FR:subset}_{\text{FR}}) \). Their version constructs the reachable subsets only, including an explicit sink state corresponding to \( \varnothing \). Taking this into account, Example 5.6.38 corresponds to [HK89, Example 4.19]. Their presentation mostly disregards the
automata-centered view taken here, focusing on the use of match sets and tabulation, which we discuss in Section 5.7.

As a result of Lemma 5.6.8 and using the definition of $\text{SUBSET}_\text{FR}$, $RSt_M(t) = \{MS(t)\}$ for $M$ a DFRTA resulting from Construction \((\text{TGA-TA}:ALL-SUB:REM-\varepsilon:FR:SUBSET_\text{FR})\). A recursive definition of function $MS$ will be introduced in Section 5.7, and the relation between Algorithm (MATCH-SET, REC) based on that definition and Algorithm (T-ACCEPTOR, FR, DET) using such a DFRTA for a frontier-to-root computation of $RSt_M$ will be made clear.

**Remark 5.6.40.** The presentation in [HK89] contains a small error. In our notation, their paper defines $\text{Items}$ as $\text{Subtrees}(\text{RHS}(\text{Prods}))$ for the *unmodified* input RTG. As a result, the start symbol may be excluded, yet their further definitions and discourse make it clear that it is intended to be included. Either changing the definition of $\text{Items}$ to the union of the start symbol and $\text{Subtrees}(\text{RHS}(\text{Prods}))$ or using an augmented RTG and defining $\text{Items}$ to be $\text{Subtrees}(\text{RHS}(\text{Prods}'))$ as we do in this dissertation solves the problem.

\[\text{Figure 5.6.12} \text{ DFRTA resulting from Construction } (\text{TGA-TA}:\text{PROPER-N:REM-\varepsilon:FR:SUBSET}_\text{FR})\]

(Construction \((\text{TGA-TA}:\text{PROPER-S:REM-\varepsilon:FR:SUBSET}_\text{FR})\) does not appear in the literature and is omitted.)

The third deterministic construction is \((\text{TGA-TA}:\text{PROPER-N:REM-\varepsilon:FR:SUBSET}_\text{FR})\), obtained by applying $\text{SUBSET}_\text{FR}$ to Construction \((\text{TGA-TA}:\text{PROPER-N:REM-\varepsilon:FR})\).
Example 5.6.41. For the augmented RTG of Example 5.2.2, construction (TGA-TA:PROPER-N:REM-ε:FR:SUBSETFR) results in the DFRTA in Figure 5.6.12. Note that underlying NFRTA state identifiers used in the state subsets are as in Example 5.6.16 and differ from those in Example 5.6.38.

Literature reference 5.6.42. The DFRTA construction given by Ferdinand, Seidl & Wilhelm in [FSW94, Section 6] for an RTG corresponds to construction (TGA-TA:PROPER-N:REM-ε:FR:SUBSETFR) for a corresponding augmented RTG. Their construction combines the underlying NFRTA construction with a reachability-based subset construction. (In Section 5 of their paper, they mention both this reachability-based subset construction and the full subset construction.) The constructions are also presented in [WM95, Sections 11.6 & 11.7].

5.7 Recursive match set computation

In this section, we consider algorithms based on a recursive definition of match set function $MS$. The taxonomy part discussed is depicted using solid lines in Figure 5.7.1.

Throughout this section, we assume an augmented RTG $G = (N', \Sigma, r', Prods', S')$ to exist, which we usually omit from definitions and specifications for brevity.

Algorithm 5.3.6 (MATCH-SET ) in Section 5.3 did not provide a definition of the match set function $MS$ that was directly computable but merely specified that, given $Items = Subtrees(RHS(Prods'))$, the function needed to satisfy

$$MS(t) = \langle \text{Set } p : p \in Items \land p \Rightarrow t : p \rangle.$$  

Since the latter depends on $\Rightarrow$, which can be defined recursively, this leads naturally to a recursive definition for $MS$. This definition depends on two auxiliary functions.
Specification 5.7.1 (Composition of item set elements with alphabet symbol). For every \( a \in \Sigma \), function \( \text{Comp}_a \in \mathcal{P}(\text{Items})^n \rightarrow \mathcal{P}(\text{Items}) \) is specified for every \( U \in \mathcal{P}(\text{Items})^n \) by

\[
\text{Comp}_a(U) = \left\{ \text{Set} \ p_1, \ldots, p_n : p_1 \in U_1, \ldots, p_n \in U_n : a(p_1, \ldots, p_n) \right\} \cap \text{Items}.
\]

Note that for \( a \in \Sigma_0 \), this gives \( \text{Comp}_a \in 1 \rightarrow \mathcal{P}(\text{Items}) \) and \( \text{Comp}_a(()) = \{a\} \cap \text{Items} \). Identifying \( 1 \rightarrow \mathcal{P}(\text{Items}) \) with \( \mathcal{P}(\text{Items}) \), we will use \( \text{Comp}_a \) for \( \text{Comp}_a(()) \).

Specification 5.7.2. Closure function \( \text{Cl} \in \mathcal{P}(\text{Items}) \rightarrow \mathcal{P}(\text{Items}) \) is specified for every \( U \in \mathcal{P}(\text{Items}) \) by

\[
\text{Cl}(U) = U \cup \left\{ \text{Set} \ A, B, \alpha : A, B \in N \land B \Rightarrow A \Rightarrow \alpha \land \alpha \in U : B \right\}.
\]

Note that this closure is based on using the inverse of production rules, i.e. using them from right hand side to left hand side.

Definition 5.7.3. Function \( MS \in \text{Tr}(\Sigma, r) \rightarrow \mathcal{P}(\text{Items}) \) is defined for every \( t = a(t/1, \ldots, t/n) \in \text{Tr}(\Sigma, r) \) by

\[
MS(t) = \text{Cl}(\text{Comp}_a(MS(t/1), \ldots, MS(t/n))).
\]

Note that for \( a \in \Sigma_0 \), this reduces to \( MS(t) = \text{Cl}(\text{Comp}_a(()) = \text{Cl}(\text{Comp}_a) \).

Note that function \( \text{Cl} \) is based on \( \Rightarrow \) for the RTG used i.e. on Prods. Compare this to the \( \varepsilon \)-transitions of the TA in Construction 5.6.2 (TGA-TA:ALL-SUB), and its FR-directed version in Construction (TGA-TA:ALL-SUB:FR) in Section 5.6.6, which represent elements of Prods. Similarly, functions \( \text{Comp}_a \) bear resemblance to the \( a \)-transitions in \( R_a \) of these tree automata.

Example 5.7.4. For tree \( a(b(c), d) \) and the augmented RTG of Example 5.2.2—i.e. with productions

\[
\{ \begin{array}{ll}
(0) & S' \rightarrow S, \\
(1) & S \rightarrow a, \\
(2) & S \rightarrow b, \\
(3) & S \rightarrow c, \\
(4) & B \rightarrow b, \\
(5) & B \rightarrow S, \\
(6) & B \rightarrow d
\end{array} \}
\]


— the match set values are as follows, where elements added due to the closure function are separated from other elements by a semicolon:

\[
\begin{align*}
MS(c) &= \{ c; S, B \} \\
MS(d) &= \{ d; B \} \\
MS(b(c)) &= \{ b(c), b(B); B \} \\
MS(a(b(c), d)) &= \{ a(b(c), B), a(B, d); S, B \}
\end{align*}
\]

\[\square\]

**Lemma 5.7.5.** For every \( t \in Tr(\Sigma, r) \)

\[
MS(t) = \langle \text{Set}\ p : p \in \text{Items} \land p \Rightarrow t : p \rangle
\]

**Proof.** We prove the lemma using structural induction on \( t \).

Case \( t = a \in \Sigma_0 \):

\[
\begin{align*}
\langle \text{Set}\ p : p \in \text{Items} \land p \Rightarrow a : p \rangle \\
= & \langle \{ p \notin N \equiv p(\varepsilon) = a \text{ as productions are non-contractive, definition of } \Rightarrow \} \rangle \\
\cup & \langle \text{Set}\ p : p \in \text{Items} \land p \in N \land p \Rightarrow a : p \rangle \\
= & \langle \{ N \subseteq \text{Items}, \text{property of } \Rightarrow, \text{set calculus, dummy renaming } \} \rangle \\
\langle \{ a \} \cap \text{Items} \rangle \cup & \langle \text{Set}\ B : B \in N \land B \Rightarrow a : B \rangle \\
= & \langle \{ \text{specification } \text{Comp}_a, B \neq a \} \rangle \\
\text{Comp}_a \cup & \langle \text{Set}\ A, B : A, B \in \text{Items} \cap N \land a \Rightarrow A \Rightarrow a : B \rangle \\
= & \langle \{ A \Rightarrow a \text{ implies } a \in \text{Items}, \text{hence } \text{Comp}_a = \{ a \} \} \rangle \\
\text{Comp}_a \cup & \langle \text{Set}\ A, B : A, B \in \text{Items} \cap N \land a \in \text{Comp}_a \land B \Rightarrow a \Rightarrow a : B \rangle \\
= & \langle \{ \text{specification } \text{Cl} \} \rangle \\
\text{Cl(Comp}_a) \\
\end{align*}
\]

Case \( t = a(t_1, \ldots, t_n), a \in \Sigma \setminus \Sigma_0 \):

\[
\begin{align*}
\langle \text{Set}\ p : p \in \text{Items} \land p \Rightarrow a(t_1, \ldots, t_n) : p \rangle \\
= & \langle \{ p \notin N \equiv p(\varepsilon) = a \text{ as productions are non-contractive, definition of } \Rightarrow \} \rangle \\
\langle \text{Set}\ p : p \in \text{Items} \land p(\varepsilon) = a \land p \Rightarrow a(t_1, \ldots, t_n) : p \rangle
\end{align*}
\]
5.7 Recursive match set computation

∪ \{ \text{Set } p : p ∈ \text{Items} \land p ∈ N \land p \Rightarrow a(t₁, \ldots, tₙ) : p \}

= \{ \text{rewrite left and right quantification, see (⋆) and (⋆⋆) } \}

∪ \{ \text{Set } p : p ∈ \text{Items} \land p(ε) = a \land p/₁, \ldots, p/n ∈ MS(t₁), \ldots, MS(tₙ) : p \}

∪ \{ \text{Set } A, B, p : A, B ∈ \text{Items} \cap N \land B \Rightarrow A ⇒ p : p \land p ∈ \text{Items} \land p(ε) = a \land p/₁, \ldots, p/n ∈ MS(t₁), \ldots, MS(tₙ) : p \}

= \{ \text{specification } Compₐ, Cl \}

Cl(Compₐ(MS(t₁), \ldots, MS(tₙ)))

Ad (⋆):

p ∈ \text{Items} \land p(ε) = a \land p \Rightarrow a(t₁, \ldots, tₙ)

≡ \{ \text{property of } ⇒ \}

p ∈ \text{Items} \land p(ε) = a \land p/₁, \ldots, p/n ⇒ t₁, \ldots, tₙ

≡ \{ \text{Items is subtree-closed, Induction Hypothesis } \}

p ∈ \text{Items} \land p(ε) = a \land p/₁, \ldots, p/n ∈ MS(t₁), \ldots, MS(tₙ)

Ad (⋆⋆):

\{ \text{Set } p : p ∈ \text{Items} \land p ∈ N \land p \Rightarrow a(t₁, \ldots, tₙ) : p \}

= \{ \text{dummy renaming } p/B, p \neq a(t₁, \ldots, tₙ) \}

\{ \text{Set } A, B, α : A, B ∈ \text{Items} \cap N \land B ⇒ A ⇒ α : B \land α ⇒ a(t₁, \ldots, tₙ) \land α(ε) = a \}

= \{ A ⇒ α \text{ implies } α ∈ \text{Items } \}

\{ \text{Set } A, B, α : A, B ∈ \text{Items} \cap N \land B ⇒ A ⇒ α : B \land α ⇒ a(t₁, \ldots, tₙ) \land α ∈ \text{Items} \land α(ε) = a \}

= \{ (⋆) \}

\{ \text{Set } A, B, α : A, B ∈ \text{Items} \cap N \land B ⇒ A ⇒ α : B \land α ∈ \text{Items} \land α(ε) = a \land α/₁, \ldots, α/n ∈ MS(t₁), \ldots, MS(tₙ) \}

Lemma 5.7.5 allows us to use the recursive definition of MS with Algorithm (match-set). As mentioned at the end of Section 5.3, we use a signature D → P(Items)
instead of $Tr(\Sigma, r) \rightarrow P(\text{Items})$, i.e. a function $MS_{\text{node}}$ taking a reference to a node as a parameter instead of a subtree.

**Detail 5.7.6** (rec). Recursively compute match set values.

This leads to the following algorithm, in which $t$ is a global variable and $MS_{\text{node}}$ is such that $MS_{\text{node}}(n) = MS(t/n)$:

**Algorithm 5.7.7** (**MATCH-SET, REC**)

\[
\begin{array}{l}
\text{const } G = (N', \Sigma, r', \text{Prods}', S') : \text{augmented RTG};
\text{let } b : \mathbb{B} = \text{true};
\text{let } C_l \text{ be as in Specification 5.7.1 ;}
\text{let } Cl \text{ be as in Specification 5.7.2 ;}
\text{let } Comp_a \text{ be as in Specification 5.7.1 ;}
\text{let } a = t(n);
\text{let } b := S \in MS_{\text{node}}(\varepsilon)
\end{array}
\]

\[
\begin{array}{l}
\text{func } MS_{\text{node}}(n : D) : P(\text{Items}) =
\text{let } a = t(n);
\text{let } b = t \in L(G) \}
\text{func } MS_{\text{node}}(\varepsilon) = MS(\varepsilon) \]

\[
\begin{array}{l}
\text{for } i : 1 \leq i \leq n \rightarrow
\text{let } s_i := MS_{\text{node}}(n \cdot i)
\text{let } s_i := MS(t/(n \cdot i)) \}
\text{let } a = t(n);
\text{let } b := S \in MS_{\text{node}}(\varepsilon)
\end{array}
\]

\[
\begin{array}{l}
\text{for } i : 1 \leq i \leq n \rightarrow
\text{let } s_i := MS(t/(n \cdot i)) \}
\text{let } a = t(n);
\text{let } b := S \in MS_{\text{node}}(\varepsilon)
\end{array}
\]

\[
\begin{array}{l}
\text{for } i : 1 \leq i \leq n \rightarrow
\text{let } s_i := MS(t/(n \cdot i)) \}
\text{let } a = t(n);
\text{let } b := S \in MS_{\text{node}}(\varepsilon)
\end{array}
\]

\[
\begin{array}{l}
\text{for } i : 1 \leq i \leq n \rightarrow
\text{let } s_i := MS(t/(n \cdot i)) \}
\text{let } a = t(n);
\text{let } b := S \in MS_{\text{node}}(\varepsilon)
\end{array}
\]

In Section 5.6.1.1 we showed that for tree automata as in Construction 5.6.2 (**TGA-RA-ALL-SUB**) and (**TGA-RA-ALL-SUB:FR**) or their $\varepsilon$-free versions, $RS(t) = MS(t)$ holds. In other words, the direct computation of match set values using implementations of functions $Cl$ and $Comp_a$ corresponds to the simulation of such nondeterministic tree automata.

**Remark 5.7.8.** Such $\varepsilon$NFTA or NFTA simulation is similar to the simulation of nondeterministic finite string automata (NFAs), as described in e.g. [ALSU06, Section 3.7] as one of three alternatives to implement string acceptors.
In Section 5.7.1, we compare $MS$ computation using Algorithm 5.7.7 (match-set, rec) to Algorithm 5.5.4 (T-acceptor, FR, DET) with the use of a DFRTA resulting from Construction (TGA-TA:ALL-SUB:REM-$\varepsilon$:FR:SUBSET$_{pr}$) obtained by applying REM-$\varepsilon$ and SUBSET$_{pr}$ to Construction (TGA-TA:ALL-SUB:FR), showing that they are essentially the same.

In Section 5.7.2 we consider the computation of functions $Cl$ and $Comp_a$ and in particular the precomputation of the nonterminal closure part of $Cl$, similar to the use of a precomputed $\varepsilon$-closure in the case of string NFA simulation.

The tabulation of DFRTAs will be considered in detail in Section 5.7.3, while the use of filtering to reduce table space is considered in Sections 5.7.4 and 5.7.5.

**Remark 5.7.9.** Similar to the construction and tabulation of DFRTAs we mention above, Aho, Lam, Sethi, and Ullman mention construction and tabulation of the deterministic automata (DFAs) obtained from NFAs as a second implementation option [ALSU06, Section 3.7]. In addition, they mention a third, hybrid solution, using an NFA with on-the-fly DFA state construction and caching. A similar solution using an ε-NFRTA or NFRTA with on-the-fly DFRTA state construction and caching is possible but not investigated further here.

### 5.7.1 Relation to DFRTA computation

Recall from Section 5.6.6.4 that $RST_M(t) = \{MS(t)\}$ for $M$ a DFRTA resulting from Construction (TGA-TA:ALL-SUB:REM-$\varepsilon$:FR:SUBSET$_{pr}$). We repeat Algorithm 5.5.4 (T-acceptor, FR, DET) as Algorithm 5.7.10 and we show that, used with such a DFRTA, it is essentially the same as Algorithm 5.7.7 (match-set, rec).

**Algorithm 5.7.10** (T-acceptor, FR, DET)

\[
\begin{align*}
\| \text{const } G &= (N', \Sigma, r', \text{Prods}', S') : \text{ augmented RTG}; \\
\text{t} &= Tt(\Sigma, r); \\
\text{var } b : \mathbb{B} \\
| \text{let } M &= (Q, \Sigma, r, R, Q_{ra}) \text{ be a DFRTA such that } L(M) = L(G); \\
\text{b} &: = \text{Traverse}(\varepsilon) \in Q_{ra} \\
\{ \text{ b } &= t \in L(G) \} \\
\text{func Traverse}(\downarrow n : D) : Q = \\
\{ \text{ Pre: } n \in D_t \} \\
| \text{var } q_1, \ldots, q_n : Q \\
| \text{let } a &= t(n); \\
\text{if } n > 0 \rightarrow \\
\text{for } i : 1 \leq i \leq n \rightarrow \\
\{ n \cdot i \in D_t \}
\end{align*}
\]
\( q_i := Traverse(n \cdot i) \)
\[ \{ \{ q_i \} = RSt(t/(n \cdot i)) \} \]

\textbf{rof:}
\[ \{ \forall i : 1 \leq i \leq n : \{ q_i \} = RSt(t/(n \cdot i)) \} \]
\( \text{Traverse} := R_\alpha(q_1, \ldots, q_n) \)
\( n = 0 \rightarrow \text{Traverse} := R_\alpha() \)
\}
\{ Post: \{ \text{Traverse} \} = RSt(t/n) \}

We apply the following transformations to this algorithm:

- The use of \( \mathcal{P}(\text{Items}) \) for \( Q \), since \( Q = \mathcal{P}(\text{Items}) \) in the DFSRTA resulting from Construction (TGA-TA:ALL-SUB:REM-\( \varepsilon \):FR:SUBSET\( \alpha \) ) (using the pure subset construction).

- The use of equality \( RSt_M(t) = \{ MS(t) \} \) and subsequent renaming of variables \( q_1, \ldots, q_n \) to \( s_1, \ldots, s_n \) and of \( \text{Traverse} \) to \( MS_{\text{node}} \), to replace \( \{ q_i \} = RSt(t/(n \cdot i)) \) and \( \{ \text{Traverse} \} = RSt(t/n) \) by \( s_i = MS(t/(n \cdot i)) \) and \( \text{Traverse} = MS(t/n) \), respectively.

- The subsequent replacement of \( MS_{\text{node}}(\varepsilon) \in Q_{ra} \), by \( S \in MS_{\text{node}}(\varepsilon) \), is allowed since set \( Q_{ra} \) of the DFSRTA contains precisely those elements of \( \mathcal{P}(\text{Items}) \) that contain \( S \).

This gives the following algorithm:

\[ \text{const } G = (N', \Sigma, r', \text{Prods}', S') : \text{ augmented RTG}; \]
\[ t : Tr(\Sigma, r); \]
\[ \text{var } b : \mathbb{B} \]
\[ \text{let } M = (\mathcal{P}(\text{Items}), \Sigma, r, R, Q_{ra}) \text{ be a DFSRTA such that } \mathcal{L}(M) = \mathcal{L}(G); \]
\[ b := S \in MS_{\text{node}}(\varepsilon) \]
\[ b \equiv t \in \mathcal{L}(G) \}
\]
\[ \text{func } MS_{\text{node}}(\downarrow n : D) : \mathcal{P}(\text{Items}) = \]
\[ \{ \text{Pre: } n \in D_b \} \]
\[ \text{var } s_1, \ldots, s_n : \mathcal{P}(\text{Items}) \]
\[ \text{let } a = t(n); \]
\[ \text{if } n > 0 \rightarrow \]
\[ \text{for } i : 1 \leq i \leq n \rightarrow \]
\[ \{ n \cdot i \in D_t \} \]
\[ s_i := MS_{\text{node}}(n \cdot i) \]
\{ s_i = MS(t/(n \cdot i)) \}

\textbf{proof:}
\{ \langle \forall i : 1 \leq i \leq n : s_i = MS(t/(n \cdot i)) \rangle \}
\quad n = 0 \rightarrow
\quad MS_{\text{node}} := R_a()
\quad MS_{\text{node}} := R_a(s_1, \ldots, s_n)
\quad \}
\quad \langle \text{Post: } MS_{\text{node}} = MS(t/n) \rangle
\quad \}

From the \(\forall\)-quantification and the postcondition of function \(MS_{\text{node}}\) in the above algorithm, and from Definition 5.7.3 of \(MS\) it follows that \(Cl \circ \text{Comp}_a = R_a\) for every \(a \in \Sigma\), i.e. the transition function \(R_a\) of the DFRTA according to Construction (TGA-TA:ALL-SUB:REM:-FR:SUBSET_FN) corresponds to the function composition \(Cl \circ \text{Comp}_a\) yielding the match set value for a tree based on the match set values for its direct subtrees.

\subsection*{5.7.2 Computing auxiliary function values}

For computing match set values in function \(MS_{\text{node}}\), Algorithm (MATCH-SET, REC) assumes the existence of functions \(\text{Comp}_a\) and \(Cl\) satisfying Specifications 5.7.1 and 5.7.2. An implementation of functions \(\text{Comp}_a\) is straightforward based on their specification and is therefore omitted.

For function \(Cl\), an implementation involves computation of the \textit{nonterminal closure} (also known as \textit{chain rule closure}) of a nonterminal, i.e. the set of nonterminals from which the given nonterminal can be derived, corresponding to the \(B \Rightarrow A\) part of closure function \(Cl\)'s specification.\(^2\) We introduce a function \(NCl\) corresponding to the nonterminal closure:

\textbf{Specification 5.7.11 (NCl).} Function \(NCl \in N \rightarrow \mathcal{P}(N)\) is specified for all \(A \in N\) by

\[ NCl(A) = \left\langle \textbf{Set} \; B : B \in N \land B \Rightarrow A : B \right\rangle. \]

\(\square\)

Using this specification, we can refine the specification of \(Cl\) as follows:

\textbf{Specification 5.7.12.} Closure function \(Cl \in \mathcal{P}(\text{Items}) \rightarrow \mathcal{P}(\text{Items})\) is specified for every \(U \in \mathcal{P}(\text{Items})\) by

\[ Cl(U) = U \cup \left\langle \bigcup A, \alpha : A \in N \land \alpha \Rightarrow A : \alpha \land \alpha \in U : NCl(\alpha) \right\rangle. \]

\(\square\)

\(^2\)Note that both closures are based on using the inverse of production rules, i.e. using them from right hand side to left hand side.
An implementation of function $CI$ is straightforward now, assuming $NCI$ can be computed. Computing $NCI(A)$ for some nonterminal $A$ can be done as needed, but for efficiency reasons the function can be tabulated. Since it is likely that such function values will be used more than once, we consider the latter option in detail.

The nonterminal closure function may be tabulated using a variant of Warshall’s algorithm for computing the transitive closure on $\Rightarrow$ (for all nonterminals), to which the computation of the reflexive closure is added (Warshall’s algorithm and this variant were discussed in Section 2.4):

**Algorithm 5.7.13** (Tabulation of $NCI$)

\[
\begin{align*}
|| & \textbf{const} \ G = (N', \Sigma, \iota', \text{Prods}', S') : \text{RTG}; \\
| & \textbf{var} \ NCI : N \to \mathcal{P}(N) \\
| & \{ \text{Compute one-step relation} \} \\
| & \textbf{for} \ A : A \in N \to \\
| & \quad \text{NCI}(A) := \{ \text{Set } B : B \in N \land B \Rightarrow A : B \} \\
| & \textbf{rof:} \\
| & \{ \text{Compute transitive closure} \} \\
| & \textbf{for} \ B : B \in N \to \\
| & \quad \textbf{for} \ A : A \in N \to \\
| & \quad \quad \text{as } B \in \text{NCI}(A) \to \\
| & \quad \quad \quad \text{NCI}(A) := \text{NCI}(A) \cup \text{NCI}(B) \\
| & \quad \textbf{sa} \\
| & \textbf{rof} \\
| & \{ \text{Compute reflexive closure} \} \\
| & \textbf{for} \ A : A \in N \to \\
| & \quad \text{NCI}(A) := \text{NCI}(A) \cup \{ A \} \\
| & \textbf{rof} \\
||
\]

**Example 5.7.14.** For the augmented RTG of Example 5.7.4 (and Example 5.2.2), the only unit production is $B \to S$, hence $\text{NCI}(S) = \{ S, B \}$ and $\text{NCI}(B) = \{ B \}$. \hfill \square

### 5.7.3 Using tabulated match set values

Function $\text{MS}_{\text{node}}$ in Algorithm 5.7.7 (\textsc{match-set}, \textsc{rec}) computes the match set value for a subtree regardless of whether this value was already computed before.

Since set $\text{Items}$ is finite, so is $\mathcal{P}(\text{Items})$, i.e. the number of potential match sets is finite. Function $\text{MS}$ can thus be tabulated to improve computation efficiency by not having to recompute match sets on every application of the function.
We observe that $MS(a(t_1, \ldots, t_n))$ is of the form $Cl(Comp_a(MS(t_1), \ldots, MS(t_n)))$ (for all $a \in \Sigma$). Tabulating the $Cl \circ Comp_a$’s leads to an $n$-dimensional table $T_a$ for every $a \in \Sigma$ of rank $n$.

Note that due to the correspondence between $Cl \circ Comp_a$ in the match set function computation and $R_a$ in the computation of a certain DFRTA as noted in Section 5.7.1, the tabulation of the match set function corresponds to the tabulation of the transition function of such a DFRTA.

Observe that tabulation is necessary for the reachable part of $\mathcal{P}(\text{Items})$ only, i.e. the smallest subset of $\mathcal{P}(\text{Items})$ reachable under the set of $Cl \circ Comp_a$. This reachable part is likely to be relatively small, as any match set containing trees of the form $a(t_1, \ldots, t_n)$ and $b(u_1, \ldots, u_m)$ with $a \neq b$ obviously is not reachable.

For storage and computation time efficiency, we want to index match set tables by integers. As $\mathcal{P}(\text{Items})$ and its reachable part are finite, we can introduce a bijection between (the reachable part of) $\mathcal{P}(\text{Items})$ and a consecutive initial part of $\mathbb{N}$.

**Specification 5.7.15** (Tabulation of match sets identified by integers). Let $Z$ be the part of $\mathcal{P}(\text{Items})$ reachable under the set of $Cl \circ Comp_a$, let $m \in [0 \ldots |Z|] \rightarrow Z$ be a bijection and let tables $T_a \in [0 \ldots |Z|]^n \rightarrow [0 \ldots |Z|]$ for every $a \in \Sigma$ be such that for every $q \in [0 \ldots |Z|]^n$

$$m(T_a(qz)) = Cl(Comp_a(m(qz_1), \ldots, m(qz_n)))$$

**Detail 5.7.16** (tabulate). Use a tabulated version of the match set function, in which a bijection is used to identify match sets by integers.

**Remark 5.7.17.** Instead of directly applying this detail completely to obtain the algorithm given below, one could first introduce a version using a tabulation $T'_a \in (\mathcal{P}(\text{Items}))^n \rightarrow \mathcal{P}(\text{Items})$ with $T'_a(s) = Cl(Comp_a(s_1, \ldots, s_n))$, i.e. directly tabulating the match set function. This would result in an algorithm ‘somewhere in between’ Algorithm 5.7.7 and Algorithm 5.7.18 given below. It can be obtained from the former by replacing $Cl(Comp_a(\ldots))$ in that algorithm by $T'_a(\ldots)$.

Using detail (tabulate) yields the following algorithm:

**Algorithm 5.7.18** (match-set, rec, tabulate)

```latex
\textbf{Algorithm 5.7.18} (match-set, rec, tabulate)

\begin{verbatim}
| const G = (N', \Sigma, r', Prods', S') : augmented RTG;
| t : Tr(\Sigma, r);
| var b : B
| let m \in [0 \ldots |Z|) \rightarrow Z be as in Specification 5.7.15;
| let T_a \in [0 \ldots |Z|]^n \rightarrow [0 \ldots |Z|](\forall a \in \Sigma \setminus \Sigma_0) be as in Specification 5.7.15;
| let T_a \in [0 \ldots |Z|](\forall a \in \Sigma_0) be as in Specification 5.7.15;
\end{verbatim}
```
Chapter 5 Tree acceptance

\[ b := S \in m(MS'_{node}(\varepsilon)) \]
\[ \{ b = t \in \mathcal{L}(G) \} \]

func \( MS'_{node}(\downarrow n : D) : [0 \ldots |Z|) = \)
\[ \{ \text{Pre: } n \in D \} \]
\[ \| \text{var } qz_1, \ldots, qz_n : [0 \ldots |Z|) \]
\[ \| \text{let } a = t(n); \]
\[ \| \text{if } n > 0 \rightarrow \]
\[ \| \text{for } i : 1 \leq i \leq n \rightarrow \]
\[ \| \{ n \cdot i \in D_i \} \]
\[ \| qz_i := MS'_{node}(n \cdot i) \]
\[ \| qz_i = m^{-1}(MS(t/(n \cdot i))) \} \]
\[ \| \text{rof}; \]
\[ \| \{ \langle i : 1 \leq i \leq n : qz_i = m^{-1}(MS(t/(n \cdot i))) \} \}
\[ \| MS'_{node} := T_a(qz_1, \ldots, qz_n) \}
\[ \| n = 0 \rightarrow \]
\[ \| MS'_{node} := T_a \}
\[ \| \{ \text{Post: } MS'_{node} = m^{-1}(MS(t/n)) \} \}
\[ \| \}

An example of the algorithm’s application will be given in Example 5.7.23. Note that for function \( MS'_{node} \) above and function \( MS_{node} \) as in Algorithm 5.7.7 (match-set, \textsc{rec}) we have \( MS'_{node} = m \circ MS'_{node} \).

5.7.3.1 Reachability-based tabulation

As mentioned, the values of \( MS \) need to be tabulated for the reachable part of \( \mathcal{P}(\text{Items}) \) only. We therefore combine an algorithm for computing the reachable part of a set from an initial set—discussed in Section 2.5—with the tabulation. We first present an algorithm in which match sets are directly used as table indices. The invariants and variant function of the underlying reachability algorithm were presented in Section 2.5. Note in particular that in the \textsf{do}-loop of the algorithm, \( |W| + |G| \) (or equivalently \( |Reach| - |Z| \), with \( Reach \) the reachable subset of \( \mathcal{P}(\text{Items}) \)) decreases and serves as variant function.

Algorithm 5.7.19(Reachability-based tabulation of \( Cl \circ \text{Comp}_a \))

\[ \| \text{var } Z, G, W : \mathcal{P}(\mathcal{P}(\text{Items})); \]
\[ \| v, w : \mathcal{P}(\text{Items}); \]
\[ \| T_a : (\mathcal{P}(\text{Items}))^n \rightarrow \mathcal{P}(\text{Items}) \text{ for all } a \in \Sigma \setminus \Sigma_0; \]
\[ \| T_a : \mathcal{P}(\text{Items}) \text{ for all } a \in \Sigma_0; \]
let Comp₀ be as in Specification 5.7.1;
let Cl be as in Specification 5.7.2;

\[ \begin{align*}
Z, G, W & := \emptyset, \emptyset, \mathcal{P}(\text{Items}); \\
\text{for } a : a \in \Sigma_0 & \rightarrow \\
\ w & := \text{Cl}(\text{Comp}_a()); \\
\ \{ \text{(*)} \} \\
W, G & := W \setminus \{ w \}, G \cup \{ w \}; \\
\hat{T}_a & := w
\end{align*} \]

\text{rof:}
\text{do } G \neq \emptyset \rightarrow
\text{let } v \in G;
\text{for } a : a \in \Sigma \setminus \Sigma_0 \rightarrow
\text{for } \overrightarrow{U} : \overrightarrow{U} \in (Z \cup \{ v \})^n \setminus Z^n \rightarrow
\ w & := \text{Cl}(\text{Comp}_a(\overrightarrow{U})); \\
\ \{ \text{(**)} \} \\
\text{as } w \notin W & \rightarrow \\
W, G & := W \setminus \{ w \}, G \cup \{ w \}
\text{sa:}
\hat{T}_a(\overrightarrow{U}) & := w
\text{rof}
\text{rof:}
G, Z & := G \setminus \{ v \}, Z \cup \{ v \}
\text{od}
\{ \hat{T}_a(\overrightarrow{U}) \text{ has been computed for every } \overrightarrow{U} \in Z^n = \text{Reach}^n \}
\]

We do not give an explicit example of the application of this algorithm, as Example 5.7.23 gives an extensive example for the improved algorithm introduced below, which uses integers to identify match sets. (Examples of the tables \( \hat{T}_a \) constructed can thus be obtained by considering the tables \( T_a \) in that example and replacing the integer indices and values in those tables by the corresponding match sets, as indicated by the first table in the example.)

**Remark 5.7.20.** Note that the assignment to \( W \) (and \( G \)) at (\( \ast \)) assumes that \( \text{Cl}(\text{Comp}_a) \neq \text{Cl}(\text{Comp}_b) \) for \( a, b \in \Sigma_0 \) with \( a \neq b \). This is true for \( \text{Items} = \text{Subtrees}(\text{RHS}(\text{Prods}')) \) (assuming as usual that no useless symbols exist), but may not be true for \( \text{Items}' = N \cup \text{ProperSubtrees}(\text{RHS}(\text{Prods}')) \) (or for item set \{ \( S \)\} \cup \text{ProperSubtrees}(\text{RHS}(\text{Prods}'))). When using either of the latter item sets instead of the former, the assignment to \( W \) (and \( G \)) needs to be guarded as at (\( \ast\ast \)).

\[ \square \]

Note that to compute the nonterminal closure as part of function \( \text{Cl} \), this algorithm may compute values of function \( \text{NCI} \) as needed, use a tabulated version of \( \text{NCI} \), or...
tabulate the function on-the-fly. The latter makes more sense, since the number of applications of CI in the algorithm is expected to be relatively large.

Using the identification of match sets by integers introduced in Specification 5.7.15, sets \( Z \) (of ‘black states/match sets’) and \( G \) (of ‘grey states/match sets’) can be represented by \( m(i) \) with \( 0 \leq i < p \) and \( p \leq i < q \) respectively. Set \( W \) is no longer represented explicitly in the following version of the algorithm. (Note that \(|\text{Reach}| - p \) decreases and serves as variant function.)

**Algorithm 5.7.21** (Reachability-based tabulation of \( Cl \circ \text{Comp}_a \) using integers to identify match sets)

\[
\| \text{var } m : [0 \ldots |Z|] \rightarrow \mathcal{P}(\text{Items}); \\
p, q, k : \mathbb{N}; \\
T_a : [0 \ldots |Z|]^n \rightarrow [0 \ldots |Z|] \text{ for all } a \in \Sigma \setminus \Sigma_0; \\
T_a : [0 \ldots |Z|] \text{ for all } a \in \Sigma_0; \\
| \text{let } \text{Comp}_a \text{ be as in Specification 5.7.1; } \\
\text{let } Cl \text{ be as in Specification 5.7.2; } \\
p, q := 0, 0; \\
\text{for } a : a \in \Sigma_0 \rightarrow \\
m(q) := Cl(\text{Comp}_a()); \\
\{ (\star) \} \\
T_a := q; \\
q := q + 1; \\
\text{rof; } \\
\text{do } p \neq q \rightarrow \\
\text{for } a : a \in \Sigma \setminus \Sigma_0 \rightarrow \\
\text{for } \overrightarrow{pv} : \overrightarrow{pv} \in \{0, \ldots, p\}^n \setminus \{0, \ldots, p - 1\}^n \rightarrow \\
m(q) := Cl(\text{Comp}_a(m(pv_1), \ldots, m(pv_n))); \\
\{ (\star\star) \} \\
k := 0; \\
\text{do } m(k) \neq m(q) \rightarrow k := k + 1 \text{ od; } \\
\text{as } k = q \rightarrow q := q + 1 \text{ sa; } \\
T_a(\overrightarrow{pv}) := k \\
\text{rof} \\
\text{rof; } \\
p := p + 1 \\
\text{od} \\
\{ p = |Z| \} \\
\|

**Remark 5.7.22.** Note that similar to Remark 5.7.20 for Algorithm 5.7.19, the part of the for-loop body at (\( \star \)) needs to be replaced by that of the for-loop at (\( \star\star \)) in case Items' instead of Items is used as the item set. \( \square \)
Example 5.7.23. For the augmented RTG of Example 5.7.4 (and Example 5.2.2), the algorithm generates the following tables, assuming that the nondeterministic choice in the for-loops of the above algorithm is resolved by considering alphabet symbols and tuples of states in lexicographical order.

To give an idea of how the algorithm operates, we consider part of its application to the aforementioned grammar. In the first for-loop, states/match sets are computed for symbols $c$ and $d$ of rank 0, using $C_l, \text{Comp}_c$ and $\text{Comp}_d$, resulting in grey states 0 and 1 and the (zero-dimensional) tables $T_c = 0$ and $T_d = 1$.

$$
\begin{array}{|c|}
\hline
q & m(q) \\
\hline
0 & \{c; B, S\} \\
1 & \{d; B\} \\
2 & \emptyset \\
3 & \{b(c), b(B); B\} \\
4 & \{a(B, d); B, S\} \\
5 & \{b(B); B\} \\
6 & \{a(b(c), B); B, S\} \\
7 & \{a(B, d), a(b(c), B); B, S\} \\
\hline
\end{array}
$$

$$
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
T_a & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & 2 & 4 & 2 & 2 & 2 & 2 & 2 & 2 \\
1 & 2 & 4 & 2 & 2 & 2 & 2 & 2 & 2 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
3 & 6 & 7 & 2 & 6 & 6 & 6 & 6 & 6 \\
4 & 2 & 4 & 2 & 2 & 2 & 2 & 2 & 2 \\
5 & 2 & 4 & 2 & 2 & 2 & 2 & 2 & 2 \\
6 & 2 & 4 & 2 & 2 & 2 & 2 & 2 & 2 \\
7 & 2 & 4 & 2 & 2 & 2 & 2 & 2 & 2 \\
\hline
T_b & 0 & 3 & 1 & 5 & 2 & 2 & 3 & 5 \\
\hline
\end{array}
$$

$T_c = 0 \quad T_d = 1$

Exiting this loop, $p, q = 0, 2$ holds i.e. two grey states/match sets exist, hence the do-loop body is entered to compute transitions using symbols of rank $> 0$. For symbol $a \in \Sigma_2$, tuple set $\{0, \ldots, p\}^2 \setminus \{0, \ldots, p-1\}^2$ contains just the tuple $(0, 0)$, hence $m(q) = m(2) = Cl(\text{Comp}_a(m(0), m(0))) = \emptyset$ is computed, added as a new grey state/match set with an increment of $q$ to 3, and $T_a(0, 0)$ is set to 2. For symbol $b \in \Sigma_1$, tuple set $\{0, \ldots, p\}^1 \setminus \{0, \ldots, p-1\}^1$ just contains single-element tuple $(0, 0)$, hence $m(q) = m(3) = Cl(\text{Comp}_b(m(0))) = Cl(\{b(c), b(B)\}) = \{b(c), b(B); B\}$ is computed, added as a new grey state/match set with an increment of $q$ to 4, and $T_b(0)$ is set to 3. Next, $p$ is incremented to 1 making state 0 black, and since $p \neq q$ there exists another grey state and the loop body is executed again. The algorithm constructs additional match sets and table entries in a similar way until eventually no more new match sets are reached and $p = q$ holds i.e. no more grey states exist.

This example is essentially the same as [HK89, Example 4.9]. As mentioned in Literature Reference 5.6.39, the tabulated match set values/DFRTA states correspond
to the DFRTA of Example 5.6.38 depicted in Figure 5.6.11 (apart from the empty
match set, which is not depicted there).

For example, that DFRTA has a state \{q_1, q_2\}, say \(q'_1\), with \{(q'_1, ())\} = R_d i.e. \(q'_1\) will
be the state assigned to a leaf labeled \(d\). According to Example 5.6.3, \(q_1\) corresponds
to \(B\) and \(q_2\) to \(d\), i.e. the state corresponds to a match set containing \(d\) and \(B\). Here,
state 1 corresponds to the same match set, and \(T_d = 1\) i.e. this match set will be
computed for a leaf labeled \(d\).

The use of the tables in Algorithm 5.7.18 (MATCH-SET, REC, TABULATE) results in
the following values of \(MS'_{node}\) for tree \(a(b(c), d)\):

\[
MS'_{node}(1) = T_c = 3
\]
\[
MS'_{node}(1) = T_b(0) = 3
\]
\[
MS'_{node}(2) = T_d = 1
\]
\[
MS'_{node}(\varepsilon) = T_a(3, 1) = 7
\]

Since \(S \in m(7)\), the tree is accepted. For tree \(a(d, c)\) the values are:

\[
MS'_{node}(1) = T_d = 1
\]
\[
MS'_{node}(2) = T_c = 0
\]
\[
MS'_{node}(\varepsilon) = T_a(1, 0) = 2
\]

Since \(S \not\in m(2)\), the tree is not accepted.  

The tabulation algorithm only constructs the reachable match sets. In the worst
case, there may still be \(|\mathcal{P}(\text{Items})|\) such sets—i.e. \(|Q| = |Z| = |\mathcal{P}(\text{Items})|\) may
hold—but for practical cases, this exponential number of states of the DFRTA com-
pared to the underlying NFRTA does not occur. This is apparent in the example
above, where \(|\mathcal{P}(\text{Items})| = 2^8 = 256\) yet \(|Q| = |Z| = 8\) i.e. only 8 out of 256 match
sets occur. Experiments with RTGs for instruction selection for actual processor
families show similar results, as shown in Section 8.6.1. Even so, the resulting transi-
tion tables may be rather large in terms of number of entries and memory usage,
particularly when symbols of rank \(> 1\) occur: for the small example grammar above,
the table for symbol \(a\) of rank 2 already has \(8 \times 8 = 64\) entries, and the tables for
the four symbols have 74 entries combined.

To reduce the number of entries/ transitions and the memory usage of the tables, a
number of techniques can be applied:

- Minimization of the underlying DFRTAs. The disadvantage is that the known
  algorithms (see e.g. [Bra68, Che81, CDF07]) all seem to minimize an existing
  DFRTA instead of minimizing a DFRTA as it is constructed. For this reason we
do not discuss minimization here. (Related work on minimization of NFRTAs
  (albeit based on bisimulation equivalence instead of language equivalence) can
  be found in [AKH06].)

- Reducing the number of items, corresponding to the use of detail PROPER-
  S or PROPER-N instead of ALL-SUB in the DRFTA constructions described in
Section 5.6. Using the match set view on DFRTAs, this corresponds to adapting function \( Cl \) in Specification 5.7.2 or Specification 5.7.12 to intersect the original function values with either \( \{ S \} \cup ProperSubtrees(RHS(Prods')) \) or \( N \cup ProperSubtrees(RHS(Prods')) \). Since this function is used in Definition 5.7.3 for \( MS \), this change may reduce the number of different match set values and thus the size of \( Z \) considerably, as shown below (in the table, \( Items' \) is used for \( N \cup ProperSubtrees(RHS(Prods')) \)). As already indicated in Section 5.6, this approach is used e.g. by Ferdinand, Seidl, and Wilhelm [FSW94]. As the results in Section 8.6.1 show, the DFRTA construction time may also be reduced substantially.

The disadvantage of this approach is that extra work has to be done when using the tables in a tree parsing algorithm, as production rules usable at a node have to be determined based on complete RHSs matching, which may involve rules whose complete RHS is removed from match sets due to the intersection mentioned above. This increases the running time of such algorithms [Str07a, Section 4.4.2].

- **Filtering**, i.e. using specific properties of match sets to remove certain items from match sets when using them to compute the match set for a parent node based on those computed for its child nodes. This approach will be discussed in Section 5.7.4 and turns out to improve memory usage by an order of magnitude for three realistic instruction selection grammars, when not combined with the item reduction approach discussed above. When combining the two approaches, there may be very little improvement or even an increase in memory usage compared to only using item set reduction. Some experimental results on this will be discussed in Section 8.6.1. Ferdinand et al. [FSW94] also use this technique.

- **Standard table/matrix compression techniques** that do not use any properties of the DFRTA (such as state/item set structure). As the above techniques already lead to reasonable memory usage, such techniques will not be considered.

As indicated above, the filtering and item set reduction techniques are also used in [FSW94]. There, Ferdinand et al. additionally use a different representation of the \( n \)-dimensional transition tables. They represent transitions using decision trees, following the work of Kron [Kro75] (who uses this representation for tree pattern matching) and of Börstler, Möncke, and Wilhelm [BMW91]. The latter additionally applies decision tree compression and other techniques independent of specific DFRTA properties. This representation is not considered here (and was not implemented in the toolkit).

### 5.7.4 Filtering match sets

To reduce storage space for match set tabulation, we consider filtering. We first present the idea informally and in the context of directly computing match set values,
before giving a definition of various filter functions and proving their correctness. The use of filtering with tabulation of match set values is considered afterwards.

Recall that for a tree \( t = a(t_1, \ldots, t_n) \) with \( a \in \Sigma \setminus \Sigma_0 \), \( MS(a(t_1, \ldots, t_n)) \) is of the form \( Cl(Comp_a(MS(t_1), \ldots, MS(t_n))) \) and is computed using the values of \( MS(t_i) \) for \( 1 \leq i \leq n \). In other words, the match set for a tree is computed using the match sets for its subtrees. There may however be items in the \( MS(t_i) \) that cannot contribute to the inclusion of any item in \( MS(t) \). For the purpose of computing \( MS(t) \), such items may thus be safely filtered out of the \( MS(t_i) \). This filtering is done by means of filter functions or filters. As we will see, such filters may depend on both \( a \) and \( i \). We therefore represent them by \( Filt_{a,i} \). The computation of \( MS(a(t_1, \ldots, t_n)) \) then takes the form \( Cl(Comp_a(Filt_{a,1}(MS(t_1)), \ldots, Filt_{a,n}(MS(t_n)))) \). The filtered match sets \( Filt_{a,i}(MS(t_i)) \) are called \((a, i)\)-representer sets.

The objective of using filters is a reduction of the domain of the functions \( Comp_a \), leading to a reduction in computation time or a reduction in table space when using tabulations of all the \( Cl \circ Comp_a \)'s. Of course, there is a trade off between this saving in computation time/table space and the increase in memory usage/lookup time due to the use of (tabulations of) the filter functions \( Filt_{a,i} \).

We mention a number of item categories that can be filtered out, including those that are filtered out by filters appearing in the literature:

1. Items corresponding to trees that do not occur as a proper subtree. Such a filter depends on neither \( a \) nor \( i \). Such a filter first appears in work by Prescott K. Turner, who used it to filter what he calls extended states into so-called reduced states in his paper on tree parsing [Tur86].

2. Items corresponding to trees that do not occur as a child of a node labeled \( a \). Such a filter depends on \( a \) but not on \( i \).

3. Items corresponding to trees that do not occur as the \( i \)th child of a node. Such a filter depends on \( i \) but not on \( a \).

4. Items corresponding to trees that do not occur as the \( i \)th child of a node labeled \( a \). Such a filter depends on both \( a \) and \( i \). Such a filter was first used by David Chase [Cha87]. It is also discussed in [HK89] and [BDB90].

These categories are neither mutually exclusive—compare the fourth to the second or third category—nor complete: for example, items could be filtered based on the value of match sets for one or more sibling nodes as well. (This category does not seem to occur in the literature.)
Example 5.7.24. Let $G = (\{A\}, \Sigma, r, P, A)$ be an RTG with

$$P = \{ A \to b, \quad A \to a, \quad A \to a, \quad A \to b, \quad A \to c \}.$$ 

Note that for this RTG, $MS(b(b(b(c)))) = \{ b(b(b(c))), b(b(b(A))), b(b(A)), b(A), A \}$ and $MS(c) = \{ c, A \}$.

1. Filtering the match sets according to category 1 yields $\{ b(b(b(A))), b(b(A)), b(A), A \}$ and $\{ c, A \}$.
2. Filtering according to category 2 with symbol $a$ yields $\{ b(b(b(A))), b(b(A)), A \}$ and $\{ c, A \}$.
3. Filtering according to category 3 with position 1 yields $\{ b(b(b(A))), b(b(A)), b(A), A \}$ and $\{ c, A \}$; with position 2 it yields $\{ b(b(A)), A \}$ and $\{ A \}$.
4. Filtering according to category 4 yields $\{ b(b(b(A))) \}$ and $\{ c \}$ with symbol $a$ and position 1; with symbol $a$ and position 2 it yields $\{ b(b(A)), A \}$ and $\{ A \}$.

Remark 5.7.25. P.K. Turner [Tur86] at Prime Computer used the first kind of filter in a tree parser (forming an extension of a tree acceptor). This work resulted in patent applications for a “Retargetable code generator using up/down parsing”. In the USA, a patent application (578801) was filed on February 10, 1984, followed by applications in Europe (19850300855), Canada and Japan (60189036) on February 8, 1985. The applications in the US, Europe and Japan were apparently rejected at some point, with the European one being rejected in 1993. In Canada however, it was granted as patent CA 1226956, issued September 15, 1987. Until the patent’s expiration on February 8, 2005, the use of tree parsing with the simple filtering technique used by Turner was thus patented in Canada.

Remark 5.7.26. The use of filtering has some parallels in string automata construction and LR parsing. Watson discusses two different filters on dotted regular expressions in [Wat95, Section 6.4.1] as part of his taxonomy of string automata construction algorithms.

We define four filter functions, corresponding to the four categories mentioned. In these functions, we use $RPred.u.j.U$ as an abbreviation for $u \in \text{Items} \land 1 \leq j \leq r(u(c)) \land u/j \in U$. For a given $u, j$ and for $U \in P(\text{Items})$, this predicate indicates whether there is an item $u$ such that tree $u/j$ is a direct subtree of this item and tree $u/j$ occurs in set $U$. 

\[ 5.7 \text{ Recursive match set computation} \]
Definition 5.7.27. Subtree filter function $T\text{Fil} \in \mathcal{P} \langle \text{Items} \rangle \rightarrow \mathcal{P} \langle \text{Items} \rangle$ is defined for every $U \in \mathcal{P} \langle \text{Items} \rangle$ by

$$T\text{Fil}(U) = \langle \text{Set} u, j : R\text{Pred}.u.j.U : u/j \rangle.$$ 

Filter $T\text{Fil}$ is named after P.K. Turner, in whose paper on tree parsing it first appears [Tur86].

Definition 5.7.28. For every $a \in \Sigma \setminus \Sigma_0$, symbol filter function $S\text{Fil}_a \in \mathcal{P} \langle \text{Items} \rangle \rightarrow \mathcal{P} \langle \text{Items} \rangle$ is defined for every $U \in \mathcal{P} \langle \text{Items} \rangle$ by

$$S\text{Fil}_a(U) = \langle \text{Set} u, j : R\text{Pred}.u.j.U \wedge u(\varepsilon) = a : u/j \rangle.$$ 

Filter $S\text{Fil}$ does not occur in the literature.

Definition 5.7.29. For all $i, 1 \leq i \leq r_{\text{max}}$, function $I\text{Fil}_i \in \mathcal{P} \langle \text{Items} \rangle \rightarrow \mathcal{P} \langle \text{Items} \rangle$, called the index filter, is defined for every $U \in \mathcal{P} \langle \text{Items} \rangle$ by

$$I\text{Fil}_i(U) = \langle \text{Set} u, j : R\text{Pred}.u.j.U \wedge j = i : u/j \rangle.$$ 

Filter $I\text{Fil}$ does not occur in the literature.

Definition 5.7.30. For all $a \in \Sigma \setminus \Sigma_0$ and $i$ s.t. $1 \leq i \leq n$, symbol and index filter function $C\text{Fil}_{a,i} \in \mathcal{P} \langle \text{Items} \rangle \rightarrow \mathcal{P} \langle \text{Items} \rangle$ is defined for every $U \in \mathcal{P} \langle \text{Items} \rangle$ by

$$C\text{Fil}_{a,i}(U) = \langle \text{Set} u, j : R\text{Pred}.u.j.U \wedge u(\varepsilon) = a \wedge j = i : u/j \rangle.$$ 

This filter is named $C\text{Fil}$ after Chase, who first described it [Cha87].

Note that $C\text{Fil}_{a,i}(U) \subseteq S\text{Fil}_a(U) \cap I\text{Fil}_i(U)$ for every $a, i$, yet not necessarily vice versa. This is straightforward to understand: a tree appearing somewhere under symbol $a$ and appearing somewhere as the $i$th child of a node does not necessarily mean that it appears somewhere as the $i$th child of a node labeled $a$, as shown in the following example.

Example 5.7.31. In Example 5.7.24, tree $b(b(A))$ occurs under a node labeled $a$, and occurs as first subtree of some node, but does not occur as first subtree of a node labeled $a$. Thus, for a set $U$ containing this tree, the tree would occur in both $S\text{Fil}_a(U)$ and $I\text{Fil}_1(U)$ but not in $C\text{Fil}_{a,1}(U)$.

To show that the filters do not influence the computation of a match set, we prove the following lemma.
Lemma 5.7.32. For every \( a(t_1, \ldots, t_n) \in Tr(\Sigma, r) \),

\[ Comp_a(MS(t_1), \ldots, MS(t_n)) = Comp_a(CFilta_1(MS(t_1)), \ldots, CFilta_n(MS(t_n))). \]

Proof. We derive

\[
Comp_a(MS(t_1), \ldots, MS(t_n)) \\
\{ \text{Specification 5.7.1} \} \\
\big( \text{Set } p_1, \ldots, p_n : p_1 \in MS(t_1), \ldots, p_n \in MS(t_n) : a(p_1, \ldots, p_n) \big) \cap \text{Items} \\
\{ \exists : CFilta_i(U) \subseteq U \text{ for every } a, i; \subseteq : \text{see } (\star) \} \\
\big( \text{Set } p_1, \ldots, p_n : p_1 \in CFilta_1(MS(t_1)), \ldots, p_n \in CFilta_n(MS(t_n)) \big) \cap \text{Items} \\
\{ \text{set calculus, Specification 5.7.1} \} \\
Comp_a(CFilta_1(MS(t_1)), \ldots, CFilta_n(MS(t_n))).
\]

Ad (\( \star \)): For every \( i, 1 \leq i \leq n \), we derive

\[
p_i \in MS(t_i) \land a(p_1, \ldots, p_n) \in \text{Items} \\
\Rightarrow \\
\{ 1 \leq i \leq n, a(p_1, \ldots, p_n)(\varepsilon) = a, a(p_1, \ldots, p_n)/i = p_i \} \\
a(p_1, \ldots, p_n) \in \text{Items} \land 1 \leq i \leq n \land a(p_1, \ldots, p_n)/i = p_i \in MS(t_i) \\
\land a(p_1, \ldots, p_n)(\varepsilon) = a \\
\Rightarrow \\
\{ \text{definition } CFilta_i \} \\
p_i \in CFilta_i(MS(t_i)) \land a(p_1, \ldots, p_n) \in \text{Items}
\]

Since for every \( a \in \Sigma \setminus \Sigma_0 \) and \( i \) such that \( 1 \leq i \leq n \), \( CFilta_i(U) \subseteq SFilta(U) \), \( CFilta_i(U) \subseteq IFilta_i(U) \) and \( CFilta_i(U) \subseteq TFil(U) \) and for every \( U \subseteq \text{Items} \) the value of each of the functions \( CFilta_i, SFilta, IFil \) and \( TFil \) is a subset of \( U \), and since \( Comp_a \) is monotonic in each of its arguments, we have

Corollary 5.7.33. For every \( a(t_1, \ldots, t_n) \in Tr(\Sigma, r) \),

\[
Comp_a(TFil(MS(t_1)), \ldots, TFil(MS(t_n))) \\
= Comp_a(SFilta(MS(t_1)), \ldots, SFilta(MS(t_n))) \\
= Comp_a(IFilta_1(MS(t_1)), \ldots, IFilta_n(MS(t_n))) \\
= Comp_a(CFilta_1(MS(t_1)), \ldots, CFilta_n(MS(t_n))) \\
= Comp_a(MS(t_1), \ldots, MS(t_n)).
\]
Detail 5.7.34 (FILTER). Use a filtering function in the computation of match set function values.

Using any of the four filter functions for $Filt$, the following algorithm skeleton provides four filter-based algorithmic solutions to the tree acceptance problem. The use of a particular filter function will be denoted by using $\text{FILTER} = T_{Filt}, S_{Filt}, I_{Filt}$ or $C_{Filt}$. Note that using the identity function for $Filt$ yields Algorithm (MATCH-SET, REC) presented before.

**Algorithm 5.7.35 (MATCH-SET, REC, FILTER)**

\[
\begin{align*}
\text{const } G &= (N', \Sigma, r', \text{Prod}s', S') : \text{augmented RTG;} \\
t &: \text{Tr}(\Sigma, r); \\
\text{var } b &: \mathbb{E} \\
| \text{let } Comp_a &: \text{as in Specification 5.7.1} ; \text{ let } Cl &: \text{as in Specification 5.7.2} ; \\
&& b &= S \in MS_{node}(\varepsilon) \\
&& \{ b \equiv t \in \mathcal{L}(G) \} \\
\text{fun } MS_{node}(\downarrow n : D) : \mathcal{P}(\text{Items}) = \\
\{ \text{ Pre: } n \in D_i \} \\
&& | \text{ var } s_1, \ldots, s_n &: \mathcal{P}(\text{Items}) \\
&& | \text{ let } a &= t(n); \\
&& | \text{ if } n > 0 \rightarrow \\
&& \text{ for } i : 1 \leq i \leq n \rightarrow \\
&& \text{ \{ } n \cdot i \in D_i \text{ \} } \\
&& s_i &= MS_{node}(n \cdot i) \\
&& \{ s_i &= MS(t/(n \cdot i)) \} \\
\text{rof; } \\
& \{ \forall i : 1 \leq i \leq n : s_i = MS(t/(n \cdot i)) \} \\
& MS_{node} := Cl(Comp_a(Filt_{a,1}(s_1), \ldots, Filt_{a,n}(s_n))) \\
&& n &= 0 \rightarrow \\
& MS_{node} := Cl(Comp_a) \\
& | \{ \text{ Post: } MS_{node} = MS(t/n) \} \\
\end{align*}
\]

5.7.5 Using tabulated match set values with filtering

As before, we want to tabulate the reachable part of the $Cl \circ Comp_a$'s and use a bijection, leading to tables $T_a$ indexed by integers. In this case, the integers
do not correspond (using bijection \( m \)) to match sets, but (using bijections \( Rep_{a,i} \)) correspond to representer sets i.e. filtered match sets.

**Specification 5.7.36 (Tabulation of match sets identified by integers).** Let \( Z \) be the part of \( P(\text{Items}) \) reachable under the set of \( Cl \circ Comp_a \) and (for all \( a \in \Sigma, 1 \leq i \leq n \)) let \( Z_{a,i} = \text{Filt}_{a,i}(Z) \).

Let \( m \in [0 \ldots |Z|] \rightarrow Z \) and (for all \( a \in \Sigma, 1 \leq i \leq n \)) \( Rep_{a,i} \in [0 \ldots |Z_{a,i}|] \rightarrow Z_{a,i} \)

be bijections.\(^3\)

Let tables \( T_a \in [0 \ldots |Z_{a,1}|] \times \ldots \times [0 \ldots |Z_{a,n}|] \rightarrow [0 \ldots |Z|] \) for every \( a \in \Sigma \) and let tables \( \phi_{a,i} \in [0 \ldots |Z|] \rightarrow [0 \ldots |Z_{a,i}|] \) for all \( a \in \Sigma, 1 \leq i \leq n \) be such that

\[
\phi_{a,i} = Rep_{a,i}^{-1} \circ \text{Filt}_{a,i} \circ m
\]

(stated otherwise, \( Rep_{a,i} = \text{Filt}_{a,i} \circ m \circ \phi_{a,i}^{-1} \)) this determines \( Rep_{a,i} \) up to permutation) and, for every \( \bar{q}z \in [0 \ldots |Z|] \),

\[
m(T_a(\phi_{a,1}(qz_1), \ldots, \phi_{a,n}(qz_n))) = Cl(\text{Comp}_a(\text{Filt}_{a,1}(m(qz_1)), \ldots, \text{Filt}_{a,n}(m(qz_n)))).
\]

\( \square \)

In this specification,

- the \( \phi_{a,i} \) map integers corresponding to match sets to integers corresponding to their filtered versions i.e. \( a, i \)-representer sets, while
- the \( Rep_{a,i} \) form a bijection between the latter integers and the corresponding filtered match sets/\( a, i \)-representer sets.

The above yields the following algorithm:

**Algorithm 5.7.37 (\textsc{match-set}, \textsc{rec}, \textsc{filter}, \textsc{tabulate})**

\[
\| \text{const} \ G = (N', \Sigma, r', \text{Prods}', \ S') : \text{augmented RTG}; \\
\quad t : \text{Tr}(\Sigma, r); \\
\text{var} \ b : \mathbb{B} \\
\mid \text{let} \ m \in [0 \ldots |Z|] \rightarrow Z, \\
\quad Rep_{a,i} \in [0 \ldots |Z_{a,i}|] \rightarrow Z_{a,i}(\forall a \in \Sigma, 1 \leq i \leq n), \\
\quad \phi_{a,i} \in [0 \ldots |Z|] \rightarrow [0 \ldots |Z_{a,i}|](\forall a \in \Sigma, 1 \leq i \leq n), \\
\quad T_a \in [0 \ldots |Z_{a,1}|] \times \ldots \times [0 \ldots |Z_{a,n}|] \rightarrow [0 \ldots |Z|](\forall a \in \Sigma \setminus \{a_0\}) \text{ and} \\
\quad T_a \in [0 \ldots |Z|](\forall a \in \{a_0\}) \text{ be as in Specification 5.7.36} ;
\]

\(^3\)Rep since the \( Rep_{a,i} \) are the bijections for the \( a, i \)-representer sets.
\[ b := S \in m(MS'_{node}(\varepsilon)) \]
\[ \{ b \equiv t \in \mathcal{L}(G) \} \]

\begin{verbatim}
func MS'_{node}(\downarrow n : D) : [0 \ldots |Z|) =
{ Pre: n \in D_t }
\[ \{ \text{ var } qz_1, \ldots, qz_n : [0 \ldots |Z|) \} \]
\[ \{ \text{ let } a = t(n); \}
\[ \{ \text{ if } n > 0 \rightarrow \}
\[ \{ \text{ for } i : 1 \leq i \leq n \rightarrow \}
\[ \{ \text{ n \cdot i } \in D_t \} \]
\[ \{ qz_i := MS'_{node}(n \cdot i) \}
\[ \{ qz_i = m^{-1}(MS(t/(n \cdot i))) \} \}
\] \[ \{ \text{ end for } \}
\[ \{ \text{ end if } \}
\] \[ \{ \text{ Post: } MS'_{node} = m^{-1}(MS(t/n)) \} \]
\end{verbatim}

### 5.7.5.1 Reachability-based tabulation with filtering

Recall the earlier tabulation algorithm for \( Cl \circ Comp_o \)'s that used integers to represent match sets, i.e. Algorithm 5.7.21, discussed in Section 5.7.3.1. This allowed sets \( Z \) (of ‘black states/match sets’) and \( G \) (of ‘grey states/match sets’) for the reachability-based computation to be represented by \( m(i) \) for \( 0 \leq i < p \) and \( p \leq i < q \) respectively.

Here, sets \( Z_{a,j} \) and \( G_{a,j} \) for reachability as \( j \)th child of \( a \) are similarly represented by \( Rep_{a,j}(i) \) for \( 0 \leq i < p_{a,j} \) and \( p_{a,j} \leq i < q_{a,j} \) respectively. Sets \( Z_{a,j} \) and \( G_{a,j} \) will be called ‘\( a,j \)-black’ and ‘\( a,j \)-grey states/representer sets’. As before, the algorithm is based on the presentation in [HK89].

We first present the invariants for the algorithm:

- \( 0 \leq p \leq q, \)
- for all \( a \in \Sigma \setminus \Sigma_0 \) and \( j \) such that \( 1 \leq j \leq n, \)
  \[ 0 \leq p_{a,j} \leq q_{a,j}, \]
- for all \( a \in \Sigma \setminus \Sigma_0, \)
  \[ new_a = \{ \exists j : 1 \leq j \leq n : p_{a,j} < q_{a,j} \}, \]
5.7 Recursive match set computation

i.e. for every symbol $a$ of rank $> 0$, $new_a$ indicates there is a $j$ such that an
‘$a$, $j$-grey state/representer set’ exists,

- for all $a \in \Sigma \setminus \Sigma_0$, $i$ such that $0 \leq i < q$, and $j$ such that $1 \leq j \leq n$,

$$0 \leq \phi_{a,j}(i) < q_{a,j} \land \phi_{a,j}(i) = Rep_{a,j}^{-1}(Filt_{a,j}(m(i))),$$

i.e. for every symbol $a$ of rank $> 0$ and every $i$ that is a ‘grey or black match
set/state’, its $a$, $j$-representatives are ‘$a$, $j$-grey’ or ‘$a$, $j$-black’ and the tabulated
values of $m, Rep_{a,j}$ and $\phi_{a,j}$ are correct, and

- for all $a \in \Sigma$ and $\vec{i}$ such that $\langle \forall j : 1 \leq j \leq n : 0 \leq i_j < p \rangle$,

$$\langle \forall j : 1 \leq j \leq n : 0 \leq \phi_{a,j}(i_j) < p_{a,j} \rangle$$
$$\land 0 \leq T_a(\phi_{a,1}(i_1), \ldots, \phi_{a,n}(i_n)) < q$$
$$\land m(T_a(\phi_{a,1}(i_1), \ldots, \phi_{a,n}(i_n)))$$
$$= Cl(Comp_a(Filt_{a,1}(m(i_1)), \ldots, Filt_{a,n}(m(i_n))))$$

i.e. for every symbol $a$ and every tuple $\vec{i}$ of ‘black states/match sets’, the
$a$, $j$-representatives are ‘$a$, $j$-black’, the transition on $a$ from the tuple of $a$, $j$-
representatives leads to a ‘grey’ or ‘black’ state/match set, and for tuple $\vec{i}$ the
tabulated values of $m, T_a$ and $\phi_{a,j}$ are correct. (For $a \in \Sigma_0$, the tuples have
length 0 and the predicate becomes $0 \leq T_a < q \land m(T_a) = Cl(Comp_a).$

**Algorithm 5.7.38** (Reachability-based tabulation of $Cl \circ Comp_a$ and $Filt_{a,i}$ using
integers to identify match sets)

\[
\begin{align*}
| & \text{var } m : [0 \ldots |Z|] \rightarrow \mathcal{P}(\text{Items}); \\
& p, q, k : \mathbb{N}; \\
& T_a : [0 \ldots |Z_{a,1}|] \times \ldots \times [0 \ldots |Z_{a,n}|] \rightarrow [0 \ldots |Z|] \text{ for all } a \in \Sigma \setminus \Sigma_0; \\
& T_a : [0 \ldots |Z|] \text{ for all } a \in \Sigma_0; \\
& \phi_{a,j} : [0 \ldots |Z|] \rightarrow [0 \ldots |Z_{a,j}|] \text{ for all } a \in \Sigma \setminus \Sigma_0, 1 \leq j \leq n; \\
& Rep_{a,j} : [0 \ldots |Z_{a,j}|] \rightarrow Z_{a,j} \text{ for all } a \in \Sigma \setminus \Sigma_0, 1 \leq j \leq n; \\
& p_{a,j}, q_{a,j} : \mathbb{N} \text{ for all } a \in \Sigma \setminus \Sigma_0; \\
& new_a : \mathbb{B} \text{ for all } a \in \Sigma \setminus \Sigma_0; \\
& \text{let } Comp_a \text{ be as in Specification 5.7.1 ; }
\end{align*}
\]

let $Cl$ be as in Specification 5.7.2 ;

proc ComputeRepresenterSets($p : \mathbb{N}$) =

\[
\begin{align*}
& \{ (IV) \} \\
& \text{for } a : a \in \Sigma \setminus \Sigma_0 \rightarrow \\
& \quad \text{for } j : 1 \leq j \leq n \rightarrow \\
& \quad \quad Rep_{a,j}(q_{a,j}) := Filt_{a,j}(m(p)); \\
& \quad k := 0;
\end{align*}
\]
\begin{align*}
\textbf{do } & \text{Rep}_{a,j}(k) \neq \text{Rep}_{a,j}(q_{a,j}) \rightarrow k := k + 1 \textbf{ od;} \\
& \text{as } k = q_{a,j} \rightarrow q_{a,j}, \text{new}_a := q_{a,j} + 1, \text{true sa;} \\
& \phi_{a,j}(p) := k \\
\text{rof} \\
\text{rof} \\
\| \\
\{ (I) \} \\
\text{for } a : a \in \Sigma \setminus \Sigma_0 \rightarrow \\
\text{for } j : 1 \leq j \leq n \rightarrow \\
& p_{a,j}, q_{a,j} := 0, 0 \\
\text{rof; } \\
& \text{new}_a := \text{false} \\
\text{rof; } \\
& p, q, := 0, 0; \\
\{ (II) \} \\
\text{for } a : a \in \Sigma_0 \rightarrow \\
& m(q) := \text{Cl}(\text{Comp}_a()); \\
\{ (\ast) \} \\
& T_a := q; \\
& q := q + 1 \\
\text{rof} \\
\{ (III) \} \\
\text{do } p \neq q \rightarrow \\
& \text{ComputeRepresenterSets}(p); \\
& p := p + 1 \\
\text{od; } \\
\{ (V) \} \\
\text{do } (\exists a : a \in \Sigma \setminus \Sigma_0 : \text{new}_a) \rightarrow \\
\text{for } a : a \in \Sigma \setminus \Sigma_0 \land \text{new}_a \rightarrow \\
\text{for } p_v : p_v \in [0, \ldots, q_{a,1}) \times \ldots \times [0, \ldots, q_{a,n}) \rightarrow \\
& \text{Cl}(\text{Comp}_a(\text{Rep}_{a,1}(p_{v_1}), \ldots, \text{Rep}_{a,n}(p_{v_n}))); \\
\{ (\ast\ast) \} \\
& k := 0; \\
\text{do } m(k) \neq m(q) \rightarrow k := k + 1 \textbf{ od;} \\
& \text{as } k = q \rightarrow q := q + 1 \text{ sa;} \\
& T_a(p_v) := k \\
\text{rof; } \\
\{ (VI) \} \\
\text{for } j : 1 \leq j \leq n \rightarrow p_{a,j} := q_{a,j} \text{ rof; } \\
& \text{new}_a := \text{false} \\
\text{rof; }
\end{align*}
\[
\begin{array}{l}
do p \neq q \rightarrow \\
\quad \text{ComputeRepresenterSets}(p); \\
\quad p := p + 1 \\
\od \\
\od
\end{array}
\]

Note that this algorithm can be simplified when filters not depending on \(a\) or \(i\) are used, i.e. \(T\text{Filt}_a\), \(S\text{Filt}_a\) or \(I\text{Filt}_a\).

**Remark 5.7.39.** Note that similar to Remark 5.7.22 for Algorithm 5.7.21, the part of the for-loop body at (\(\ast\)) needs to be replaced by that of the for-loop at (\(\ast\ast\)) in case \(\text{Items}'\) instead of \(\text{Items}\) is used as the item set.

To indicate how the algorithm operates, we consider part of its application to the augmented RTG of Example 5.7.4, using Chase’s filter \(C\text{Filt}_{a,i}\). The reader can compare this operational description to that for the algorithm for reachability-based tabulation without filtering, given in Example 5.7.23, and to the complete application of the algorithm using Chase’s filter, treated in Example 5.7.43.

First, reachable states based on symbols of rank 0 are computed and tabulated. At (I), \(p_{a,1}, p_{a,2}, q_{a,1}, q_{a,2}, p_{b,1}\) and \(q_{b,1}\) are all set to 0 (i.e. all \(a,i\)-representer sets/states are white), \(new_a, new_b\) are set to false, and \(p, q\) to 0 (i.e. all match sets/states are white). In the for-loop at (II), states/match sets are computed for symbols \(c\) and \(d\), using \(C_{l, \text{Comp}}\) and \(C_{p, \text{Comp}}\), resulting in new grey states 0 and 1 and the (zero-dimensional) tables \(T_c = 0\) and \(T_d = 1\).

Exiting this loop, \(p, q = 0, 2\) holds (i.e. two grey states exist), hence the do-loop at (III) is entered to compute grey \(a,i\)-representer sets/states for grey match sets/states, using function \(\text{ComputeRepresenterSets}\). For \(p = 0\) i.e. grey state 0, at (IV), for symbol \(a\) of rank 2, the inner for-loop is executed for child indices \(j = 1, 2\). For \(j = 1\), \(\text{Rep}_{a,1}(q_{a,1}) = \text{Rep}_{a,1}(0)\) is set to \(C\text{Filt}_{a,1}(m(p)) = C\text{Filt}_{a,1}(m(0)) = C\text{Filt}_{a,1}\{c; B, S\} = \{B\}\), which is added as a new ‘\(a,1\)-grey’ value with an increment of \(q_{a,1}\), setting \(new_a\) to true and setting \(\phi_{a,1}(0)\) to k i.e. to 0. For \(j = 2\), \(\text{Rep}_{a,2}(q_{a,2}) = \text{Rep}_{a,2}(0)\) is set to \(C\text{Filt}_{a,2}(m(p)) = C\text{Filt}_{a,2}(m(0)) = C\text{Filt}_{a,2}\{c; B, S\} = \{B\}\), which is added as a new ‘\(a,2\)-grey’ value with an increment of \(q_{a,2}\), again setting \(new_a\) to true and setting \(\phi_{a,2}(0)\) to k i.e. to 0.

For symbol \(b\) of rank 1, the inner for-loop is executed for \(j = 1\). Here, \(\text{Rep}_{b,1}(q_{b,1}) = \text{Rep}_{b,1}(0)\) is set to \(C\text{Filt}_{b,1}(m(p)) = C\text{Filt}_{b,1}(m(0)) = C\text{Filt}_{b,1}\{c; B, S\} = \{c; B\}\), which is added as a new ‘\(b,1\)-grey’ value with an increment of \(q_{b,1}\), setting \(new_b\) to true and setting \(\phi_{b,1}(0)\) to k i.e. to 0.

After incrementing \(p\) to 1 i.e. making state 0 black (since all its grey \(a,i\)-representer sets have been computed), \(p \neq q\) still holds at (III), and function \(\text{ComputeRepresenterSets}\) is called for \(p = 1\) i.e. to compute grey \(a,i\)-representer states.
for grey state 1. This call leads to tabulation of \(Rep_{a,1}(q_{a,1}) = Rep_{a,1}(1) = \emptyset\), of \(Rep_{a,2}(q_{a,2}) = Rep_{a,2}(1) = \{d; B\}\), \(\phi_{a,1}(1) = 0, \phi_{a,2}(1) = 1, Rep_{b,1}(1) = \{B\}\), and of \(\phi_{b,1}(1) = 1\), with each of \(q_{a,1}, q_{a,2}\) and \(q_{b,1}\) being increased to 2.

After the two calls to ComputeRepresenterSets and increments of \(p, p = q\) holds i.e. no grey states/match sets remain and the loop at (III) is exited. As \(new_a\) and \(new_b\) are true (i.e. grey a, i-representer states exist), the do-loop at (V) is entered to compute states reachable using these symbols of rank > 0. For \(a, [0, \ldots, q_{a,1}] \times [0, \ldots, q_{a,2}]) \times [0, \ldots, p_{a,1}] \times [0, \ldots, p_{a,2}] = \{0, 1\} \times \{0, 1\}\) i.e. there are four two-tuples (of grey or black \(a, i\)-representer states such that at least one element is a grey \(a, i\)-representer state) to consider. For \((0, 0)\), \(m(q) = m(2) = Cl(Compa(Rep_{a,1}(0), Rep_{a,2}(0))) = \emptyset\) is computed, added as a new grey state/match set with an increment of \(q\) to 3, and \(T_a(0, 0)\) is set to 2. For the other three tuples, the computation is similar, giving additional new grey match set \(m(3) = \{a(B, d); B, S\}\) for \(T_a(0, 1)\). At (VI) the algorithm makes the grey \(a, i\)-representer states black and adapts \(new_a\) correspondingly. For \(b\), a similar computation is performed.

Additional grey \(a, i\)-representer sets for new grey states, and additional match sets and table entries are constructed similarly, until no more new match sets are reached and \(\forall a : \phi_{a} : a \in \Sigma_{\Sigma_0} : \neg new_a\) holds i.e. no more grey representer states exist.

We will now give examples of the complete tables for the example grammar for each of the four filters, including \(CFilt_{a,i}\). In these examples, we again assume that the nondeterministic choice in the for-loops of the above algorithm is resolved by considering alphabet symbols and tuples of states in lexicographical order. Using the above algorithm, this results in a reversed construction order of states 3 and 4 in these four examples compared to the unfiltered construction in Example 5.7.23.

**Example 5.7.40.** For the augmented RTG of Example 5.7.4 (and Example 5.2.2), the algorithm generates the following tables when applied with filter \(TFilt\):

<table>
<thead>
<tr>
<th>(q)</th>
<th>(m(q))</th>
<th>(\phi(q))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{c; B, S}</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>{d; B}</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(\emptyset)</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>{a(B, d); B, S}</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>{b(c), b(B); B}</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>{b(B); B}</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>{a(b(c), B); B, S}</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>{a(B, d), a(b(c), B); B, S}</td>
<td>3</td>
</tr>
</tbody>
</table>

\(T_a\) | 0 | 1 | 2 | 3 | 4 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

\(T_b\) | 0 | 1 |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

\(T_c = 0\) \hspace{1cm} \(T_d = 1\)
The use of these tables in Algorithm 5.7.37 (match-set, rec, filter, tabulate) results in the following values of $MS'_{node}$ for tree $a(b(c), d)$:

\[
\begin{align*}
MS'_{node}(1 \cdot 1) &= T_c = 0 \\
MS'_{node}(1) &= T_b(\phi(0)) = T_b(0) = 4 \\
MS'_{node}(2) &= T_d = 1 \\
MS'_{node}(\epsilon) &= T_a(\phi(4), \phi(1)) = T_a(4, 1) = 7
\end{align*}
\]

Since $S \in m(7)$, the tree is accepted. For tree $a(d, c)$ the values are:

\[
\begin{align*}
MS'_{node}(1) &= T_d = 1 \\
MS'_{node}(2) &= T_c = 0 \\
MS'_{node}(\epsilon) &= T_a(\phi(1), \phi(0)) = T_a(1, 0) = 2
\end{align*}
\]

Since $S \notin m(2)$, the tree is not accepted.

**Example 5.7.41.** For the augmented RTG of Example 5.7.4 (and Example 5.2.2), the algorithm generates the following tables when applied with filter $SFilt_a$:

<table>
<thead>
<tr>
<th>$q$</th>
<th>$m(q)$</th>
<th>$\phi_a(q)$</th>
<th>$\phi_b(q)$</th>
<th>$Rep_{q}$</th>
<th>$T_{a}$</th>
<th>$T_{b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{c; B, S}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>{d; B}</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>{B}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{a(B, d); B, S}</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>{d; B}</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{b(c), b(B); B}</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>{b(B); B}</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>{b(c); B}</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>{a(b(c)), B; B, S}</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>{a(B, d), a(b(c), B); B, S}</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The use of these tables in Algorithm 5.7.37 (match-set, rec, filter, tabulate) results in the following values of $MS'_{node}$ for tree $a(b(c), d)$:

\[
\begin{align*}
MS'_{node}(1 \cdot 1) &= T_c = 0 \\
MS'_{node}(1) &= T_b(\phi(0)) = T_b(0) = 4 \\
MS'_{node}(2) &= T_d = 1 \\
MS'_{node}(\epsilon) &= T_a(\phi(4), \phi(1)) = T_a(3, 1) = 7
\end{align*}
\]

Since $S \in m(7)$, the tree is accepted. For tree $a(d, c)$ the values are:

\[
\begin{align*}
MS'_{node}(1) &= T_d = 1 \\
MS'_{node}(2) &= T_c = 0 \\
MS'_{node}(\epsilon) &= T_a(\phi(1), \phi(0)) = T_a(1, 0) = 2
\end{align*}
\]

Since $S \notin m(2)$, the tree is not accepted.

\[\square\]
Example 5.7.42. For the augmented rtg of Example 5.7.4 (and Example 5.2.2), the algorithm generates the following tables when applied with filter $IFilt_t$:

<table>
<thead>
<tr>
<th>$q$</th>
<th>$m(q)$</th>
<th>$\phi_1(q)$</th>
<th>$\phi_2(q)$</th>
<th>$Rep_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${c; B, S}$</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>${d; B}$</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>${a(B, d); B, S}$</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>${b(c), b(B); B}$</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>${b(B); B}$</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>${a(b(c), B); B, S}$</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>${a(B, d), a(b(c), B); B, S}$</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$Rep_2$</th>
<th>$T_a$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>$T_b$</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${B}$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>${d; B}$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

$T_c = 0 \quad T_d = 1$

The use of these tables in Algorithm 5.7.37 (MATCH-SET, REC, FILTER, TABULATE) results in the following values of $MS_{node}'$ for tree $a(b(c), d)$:

\[
MS_{node}'(1 \cdot 1) = T_c = 0 \\
MS_{node}'(1) = T_b(\phi_1(0)) = T_b(0) = 4 \\
MS_{node}'(2) = T_d = 1 \\
MS_{node}'(\varepsilon) = T_a(\phi_1(4), \phi_2(1)) = T_a(3, 1) = 7
\]

Since $S \in m(7)$, the tree is accepted. For tree $a(d, c)$ the values are:

\[
MS_{node}'(1) = T_d = 1 \\
MS_{node}'(2) = T_c = 0 \\
MS_{node}'(\varepsilon) = T_a(\phi_1(1), \phi_2(0)) = T_a(1, 0) = 2
\]

Since $S \not\in m(2)$, the tree is not accepted.

Example 5.7.43. For the augmented rtg of Example 5.7.4 (and Example 5.2.2), the algorithm generates the following tables when applied with filter $CFilt_{a,i}$:

<table>
<thead>
<tr>
<th>$q$</th>
<th>$m(q)$</th>
<th>$\phi_{a,2}(q)$</th>
<th>$\phi_{a,1}(q)$</th>
<th>$\phi_{b,1}(q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${c; B, S}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>${d; B}$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>${a(B, d); B, S}$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>${b(c), b(B); B}$</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>${b(B); B}$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>${a(b(c), B); B, S}$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>${a(B, d), a(b(c), B); B, S}$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
5.8 Stringpath-based match set computation

<table>
<thead>
<tr>
<th>(Rep_{a,1})</th>
<th>(Rep_{a,2})</th>
<th>(T_a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 {B}</td>
<td>0 {B}</td>
<td>0 1 2</td>
</tr>
<tr>
<td>1 (\emptyset)</td>
<td>1 {d; B}</td>
<td>1 2 2 2</td>
</tr>
<tr>
<td>2 {b(c); B}</td>
<td>2 (\emptyset)</td>
<td>2 6 7 2</td>
</tr>
</tbody>
</table>

\(T_c = 0\) \(T_d = 1\)

This example is essentially the same as [HK89, Example 5.12]. The use of these tables in Algorithm 5.7.37 (MATCH-SET, REC, FILTER, TABULATE) results in the following values of \(MS'_{\text{node}}\) for tree \(a(b,c), d)\):

\[
\begin{align*}
MS'_{\text{node}}(1 \cdot 1) &= T_c = 0 \\
MS'_{\text{node}}(1) &= T_b(\phi_{b,1}(0)) = T_b(0) = 4 \\
MS'_{\text{node}}(2) &= T_d = 1 \\
MS'_{\text{node}}(\varepsilon) &= T_a(\phi_{a,1}(4), \phi_{a,2}(1)) = T_a(2, 1) = 7
\end{align*}
\]

Since \(S \in m(7)\), the tree is accepted. For tree \(a(d, c)\) the values are:

\[
\begin{align*}
MS'_{\text{node}}(1) &= T_d = 1 \\
MS'_{\text{node}}(2) &= T_c = 0 \\
MS'_{\text{node}}(\varepsilon) &= T_a(\phi_{a,1}(1), \phi_{a,2}(0)) = T_a(0, 0) = 2
\end{align*}
\]

Since \(S \not\in m(2)\), the tree is not accepted.

As the four examples indicate, the use of filtering reduces the number of entries in the main \(T_a\) tables, at the expense of additional filter tables \(\phi\) and representer tables \(Rep\). Chase’s filter will usually result in a larger total number of entries over all filter and representer tables than the other filters (e.g., one filter table for each symbol and index combination, versus one for each symbol for the symbol filter, one for each index for the index filter, and merely one for Turner’s filter). Experimental results on the use of the different filters with the small RTG used in the above examples as well as with larger RTGs are discussed in Section 8.6.1.

5.8 Stringpath-based match set computation

The part of the taxonomy discussed is depicted using solid lines in Figure 5.8.1.

As indicated at the beginning of the chapter, the stringpath-based tree acceptance algorithms can and will be described as extensions of the stringpath-based tree pattern matching algorithms discussed in Section 6.7. As the discussion below assumes
familiarity with the aforementioned section, we advise the reader to first read that section before continuing below.

Recall Algorithm 6.7.9 in Section 6.7. Using transition function $\gamma$ and output function $\textit{Output}$ (defined in terms of the transition and output function of a certain automaton) in a root-to-frontier subject tree traversal and determining the value of the output function at each node $n$ for symbols $\nu$ and $t(n)$, we obtained the algorithm, in which:

- All pattern stringpath matches are detected at their endpoints, using an automaton for detecting pattern stringpaths.

- Such matches are then registered at their beginpoint $m$, i.e. added to $\textit{OSP}(m)$, as in procedure $\textit{Register}$.

- In the same procedure, complete pattern matches resulting from the addition of such a stringpath match to $\textit{OSP}(m)$ are also registered, i.e. added to $O(m)$.

The extension of that algorithm to tree acceptance results in Algorithm 5.8.1 below. The ideas behind it are as follows:

- Instead of $P$ being the pattern set, $P$ is defined to equal $\textit{RHS}($Prods$)$. In other words, instead of detecting and registering occurrences of pattern stringpaths ending in the symbol at the end node or $\nu$, the extension detects and registers occurrences of production RHS stringpaths ending in the symbol at the end node or—at (11) in the algorithm—in a nonterminal deriving the tree rooted at this end node.

- The definition of $P$ equal to $\textit{RHS}($Prods$)$ ensures that the automaton used is a matcher for $\textit{SPaths}(\textit{RHS}($Prods$))$, and changes the signature of $\textit{OSP}$ to $D_s \rightarrow \mathcal{P}(\textit{SPaths}(\textit{RHS}($Prods$))))$.

- Consequently, variable $O$ has signature $D_s \rightarrow \mathcal{P}(\textit{RHS}($Prods$))$ and is used to register complete production RHSs instead of complete patterns.
• Additionally, nonterminals deriving a subtree $t/n$ are computed—at (I) in the algorithm—by determining the nonterminal closure of the complete production R/Hs registered in $O(n)$. These nonterminals are registered in $ON(n)$ with $ON \in D_t \rightarrow \mathcal{P}(N)$ being an additional variable.

• For a node $m$, assuming that for each node $n = m \cdot l$ in the subtree below $m$ the nonterminals $A$ such that $A \Rightarrow t/(m \cdot l)$ have been correctly computed, any stringpaths starting at node $m$ and ending at $n$ with an occurrence of such an $A$ are detected at node $n$ and registered at node $m$ at (II) in the algorithm.

Thus, all stringpath matches starting at $m$ and hence all complete R/Hs and nonterminals deriving $t/m$ are registered correctly at $m$. Clearly, the assumption on nodes $m \cdot l$ holds for leaf nodes $m$. Using induction on the height of $t/m$, the reasoning thus holds for every $m \in D_t$.

• Acceptance can thus be determined by checking whether $S \in ON(\varepsilon)$.

**Algorithm 5.8.1 (s-path, sp-matcher, det)**

```plaintext
|| const $G = (N', \Sigma, r', Prods', S')$ : augmented RTG;
$P = RHS(Prods)$;
$t : Tr(\Sigma, r)$;
var $b : \mathbb{B}$;
$ON : D_t \rightarrow \mathcal{P}(N)$;
$O : D_t \rightarrow \mathcal{P}(P)$;
$OSP : D_t \rightarrow \mathcal{P}(SPaths(P))$
| let $Q, q_0, \gamma, \gamma^*$, and $Output$ be as in Specification 6.7.7 ;

for $n : n \in D_t \rightarrow ON(n), O(n), OSP(n) := \emptyset, \emptyset, \emptyset$ rof;
Traverse($\varepsilon, q_0$);
\{
  \forall n : n \in D_t : ON(n) = \left< \text{Set } A : A \in N \land A \Rightarrow t/n : A \right> \}
\{ $b := S \in ON(\varepsilon)$
\{ $b = t \in L(G)$ \}

proc Traverse($\downarrow n : D_t \downarrow q : Q$) =
\{ Pre: $n \in D_t \land q = \gamma^*(q_0, RPath(t, n)[1])$ \}
|| var $q_{next} : Q$
| let $a = t(n)$;
if $n > 0$ →
  for $i : 1 \leq i \leq n$ →
  $q_{next} := \gamma(q, a \cdot i)$;
  \{ $n \cdot i \in D_t \land q_{next} = \gamma^*(q_0, RPath(t, n \cdot i)[1])$ \}
  Traverse($n \cdot i, q_{next}$)
  \{ Traverse.Post($n \cdot i$),($q_{next}$) \}
rof;
```
\[
\begin{align*}
| & \quad n = 0 \rightarrow \\
& \quad \text{Register}(n, q, a) \\
& \text{fi;} \\
& \{ (I) \} \\
& \text{ON}(n) := \text{Cl}(\langle \text{Set } A, \alpha : A \in N \land \alpha \in O(n) \land A \rightarrow \alpha \in \text{Prods} : A \rangle); \\
& \{ (II) \} \\
& \text{for } A : A \in \text{ON}(n) \rightarrow \text{Register}(n, q, A) \text{ rof} \\
\end{align*}
\]

\[
\begin{align*}
& \text{proc } \text{Register}(\downarrow n : D, \downarrow q : Q, \downarrow a : \Sigma') = \\
& \{ \text{Pre: } n \in D_i \} \\
& \| \text{var } sp : SPaths(P); \\
& \quad m : D \\
& \| \text{for } sp : sp \in \text{Output}(q, a) \rightarrow \\
& \quad m := n \mid \text{SPDepth}(sp); \\
& \quad OSP(m) := OSP(m) \cup \{ sp \}; \\
& \quad \text{for } p : p \in P \land sp \in SPaths(p) \rightarrow \\
& \quad \text{as } SPaths(p) \subseteq OSP(m) \rightarrow O(m) := O(m) \cup \{ p \} \text{ sa} \\
& \quad \text{rof} \\
& \| \{ \text{Post: } \text{Register}.Post.n.q.a \} \\
\end{align*}
\]

The postcondition \text{Register}.Post.n.q.a is as for Algorithm 6.7.9, while postcondition \text{Traverse}.Post.n.q is obtained by additionally adding a conjunct

\[
\text{ON}(n \cdot l) = \left< \text{Set } A : A \in N \land A \Rightarrow t/(n \cdot l) : A \right>
\]

to the first universal quantification of the postcondition of the same name for Algorithm 6.7.9. This additional conjunct indicates that for every node \( l \in D_i/n \) the correct nonterminals—i.e. those deriving \( t/(n \cdot l) \)—have been registered.

Like the corresponding Algorithm 6.7.9, this algorithm can be used with either an Aho-Corasick string automaton for a set of stringpaths, or with a stringpath \text{DRFTA} for the same set. The construction of the automata does not differ from the constructions discussed in Section 6.7, apart from using a different definition of \( P \).

**Remark 5.8.2.** Note that the resulting stringpath \text{DRFTA} will therefore have transitions on terminals and nonterminals—i.e. on symbols from \( \Sigma \cup N \)—compared to transitions on terminals and \( \nu \) for the tree pattern matching case. This is in contrast to earlier TA constructions in the current chapter, which only have transitions on terminal symbols.
Due to time constraints, we do not further consider the above algorithm or the automata used in it.

**Literature reference 5.8.3.** A version of the above algorithm using an Aho-Corasick string automaton occurs in work by Aho, Ganapathi, and Tjiang [AG85, AGT89] as well as by Weisgerber and Wilhelm [WW89].

The presentation in these publications is quite informal and complicated by optimizations. Van de Meerakker [Mee88] gives a stepwise account of how to obtain the algorithms, starting from the use of Aho-Corasick automata for string matching, extending this sequentially to string matching in trees, matching of stringpaths with $\nu$ symbols (discussed in Section 6.7), adding match registration at stringpath beginpoints (ibid) and then extending further to tree acceptance and parsing.

Also see Literature reference 6.7.14 for information on literature relating to the underlying stringpath-based tree pattern matching algorithm using an Aho-Corasick string automaton.

### 5.9 Conclusions

A number of conclusions can be drawn about the tree acceptance taxonomy:

- As with earlier algorithm taxonomies, the taxonomy presents related algorithms (in this case tree acceptance algorithms) in a common framework, highlighting their commonalities and differences, and factoring out common parts, leading to common paths in the taxonomy graph. The taxonomy graph also serves as a high-level table of contents to the algorithms in this chapter.

- In this taxonomy, further factoring and separation of concerns was achieved by separating the construction of tree automata (Section 5.6) from that of the basic acceptance algorithms in which they are used (Sections 5.3 through 5.5).

- For the automata constructions treated in Section 5.6, various basic details were introduced, indicating the state set used, the presence or absence of $\varepsilon$-transitions, the direction of the automata, and whether the automata are deterministic. These details allow for easy identification and comparison of constructions from the literature, by describing the constructions using compositions of the details. Furthermore, the states contain information relating them to grammar elements, making it easier to reason about the constructions’ correctness and to relate different constructions to one another.

- Somewhat remarkably, most of the acceptor constructions found in the literature were first described in the late 1980s or in the 1990s, despite the availability of the underlying theory since the late 1960s. Theoretical researchers turned their attention to generalizing and extending regular tree language theory soon after the development of the latter theory in the 1960s (see the remark...
at the beginning of Chapter 3) and apparently did not have a lot of interest in (practical) algorithms related to the theory. Brainerd’s tree acceptor construction [Bra67, Bra69] is the exception, being from the late 1960s.

- Many automata constructions in the literature—including those presented by Brainerd [Bra67, Bra69], van Dinther [Din87, Section 5.1.1], and Conon et al. [CDG+07]—are restricted to deal with \((z_{\cdots z})\) grammars, i.e. grammars having productions of the form \(A \to a(A_1, \ldots, A_n)\) (and as a special case \(A \to a\)) only. This results in simple definitions of the constructions—as states correspond to nonterminals instead of trees and no \(\varepsilon\)-transitions occur—which makes it easier to reason about them from a theoretical point of view. However, it requires the application of grammar normalization transformations (as discussed in Section 3.3.3) for grammars that are not of that type. We presented more general definitions of the constructions, which when used on \((z_{\cdots z})\) grammars reduce to the simplified definitions given in the literature.

- In describing the automata constructions, we first described the undirected version of the various constructions, before considering their directed variants. This allowed a focus on the fundamental aspects of the constructions. In the literature, the constructions are given for either root-to-frontier automata or for frontier-to-root ones, with the exception of some given by Ferdinand et al. [FSW94] that are described independent of a direction.

- For the frontier-to-root acceptance algorithms, both the \((DFRTA)\) automata computation view and the match set computation view were considered. In the literature, usually only one view is considered, e.g. Hemerik and Katoen consider the match set view [HK89] while Ferdinand et al. [FSW94] consider the automata view. We considered both (in Section 5.6.6.4 and 5.7 respectively) and explicitly described their relation (in Section 5.7.1).

- Similarly to what was done in separating tree automata constructions from the discussion of the acceptance algorithms using them, in Section 5.7 the details of the match set precomputation and tabulation (in Sections 5.7.2 and 5.7.3) were separated from the algorithms using match set computation. The same was done for the use of filtering techniques on the one hand (Section 5.7.4) and filter precomputation and tabulation on the other hand (Section 5.7.5). This helped to clarify these techniques separately from their application during tabulation.

- As for filtering of \(DFRTA\) transition tables/match set tables, only one filter function (originally described by Chase) was well-known from the literature [Cha87, HK89, FSW94]. We considered three additional filter functions that are simplifications of Chase’s filter function:

  - One of the filters can be found in a paper by Turner [Tur86], but due to his terminology and description it was not clear that a filter function was in fact used. As a result, this filter was apparently overlooked by
others and no references to it appear elsewhere in the literature. This is the more remarkable as patent applications for the method’s application in a tree parsing algorithm (based on an extension of a tree acceptance algorithm) were filed and a patent was awarded based on one of these.

- In addition, we gave two new filters, which are straightforward simplifications of Chase’s filter (or more advanced versions of Turner’s filter) yet could not be found in the literature. The new filters never reduce the number of entries in the main symbol tables more than Chase’s filter (and never less than Turner’s filter), and Chase and others using his results apparently focused on reduction of this number. The use of Chase’s filter will in most cases result in more entries in the filter and representer tables however. As we will see in Section 8.6, the memory usage of the combined tables may be substantially less when using the new filters instead of Chase’s filter.
Chapter 6

Tree pattern matching

The tree pattern matching problem was mentioned in Chapter 1 as the second of three problems related to regular tree language theory. As we did for the tree acceptance problem in the preceding chapter, we present a taxonomy including various algorithms solving the tree pattern matching problem. Many of the algorithm details introduced in the tree acceptance chapter occur again in this chapter. Since they have been extensively treated in the previous chapter, we keep their discussion brief in the current chapter, suggesting that the reader refer to the preceding chapter for more detail.

6.1 Taxonomy overview

Figure 6.1.1 shows the taxonomy graph for tree pattern matching algorithms, forming an alternative table of contents to the chapter. As before, algorithms are represented by nodes, and details label the edges. Both will be introduced in the course of this chapter, but a list of the details with a brief description of each is included at the end of this section. Their names already imply a great deal of similarity with details of the tree acceptance taxonomy of Chapter 5. The sequence of labels along the path from the graph root to a particular vertex again identifies each algorithm, and algorithm numbers refer to the algorithm’s discussion in the current chapter.

The tree pattern matching algorithms in the taxonomy can be divided into three main categories, corresponding to those in the tree acceptance taxonomy:

- **Algorithms using tree automata as pattern matchers.** These form the taxonomy part starting with detail (T-MATCHER), which is similar in structure to the corresponding part of the tree acceptance taxonomy. As we did in that taxonomy, we initially assume only a few properties to hold for such automata,
without giving their complete structure.

Detail (T-MATCHER) and the first algorithm in this category are discussed in Section 6.2, while algorithms using root-to-frontier and frontier-to-root directed automata are discussed in Sections 6.3 and 6.4.

The construction of tree automata serving as pattern matchers is considered in Section 6.5. Similar to the case of tree acceptors in the previous chapter, various constructions are obtained. A difference with the tree acceptor constructions is that there are substantially fewer constructions here, and that each of them results in automata that are $\varepsilon$-free and have an output function of some sort, indicating pattern matches corresponding to a state.

- **Algorithms based on the notion of match sets.** These algorithms occur in the taxonomy part starting with detail (MATCH-SET), introduced in Section 6.2, and treated in Section 6.6, and share the computation of all so-called Items matching a subject tree. The problem can then be solved by determining those patterns occurring in this match set.

As for tree acceptance, match set computation is based on a recursive definition of match sets. The structure and details in this part of the taxonomy are mostly the same as those used for the corresponding tree acceptance taxonomy part, albeit for tree pattern matching instead of acceptance. Thus, tabulation and filtering occur again in this part, although their discussion will be less detailed than in the preceding chapter.

- **Algorithms based on stringpath matching.** These algorithms occur in the taxonomy part starting with detail (S-PATH), introduced in Section 6.2, and treated in Section 6.7, and are based on decomposing trees into stringpaths. Stringpath matching can then be used to find all stringpath matches, and pattern matches can be determined based on these.
The focus will be on algorithms using Aho-Corasick automata and deterministic root-to-frontier tree automata for stringpath matching in a root-to-frontier direction.

Some of this work appeared earlier in [CHZ06, CHZ05] with Hemerik and Zwaan as co-authors, although the presentation here is more general and more extensive. Furthermore, we show the algorithms to be related by factoring out their commonalities into a common ancestor algorithm, instead of merely comparing the algorithms.

**Remark 6.1.1.** The similarity between tree acceptance and tree pattern matching and the resulting similarity between the taxonomies comes as no surprise, given the similarity between string acceptance and string pattern matching. For string processing, a matcher for a string or string language denoted by some expression $L$ may be used. Such a matcher more or less corresponds\(^1\) to an acceptor for $(\Sigma^* \cdot L)$ i.e. any string over an alphabet $\Sigma$ followed by an occurrence of an element of $L$. Alternatively, it corresponds to an acceptor for $L$ that is started from its start state at every position of an input string. For trees, a similar correspondence between pattern matchers and acceptors exists. The similarities to the string case will be emphasized throughout the chapter.

The details labeling the taxonomy branches are listed in the following table, each with a brief description:

<table>
<thead>
<tr>
<th>T-MATCHER</th>
<th>Use a tree automaton as a pattern matcher for a set of patterns.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF</td>
<td>Consider transition relations of a tree automaton in a root-to-frontier direction.</td>
</tr>
<tr>
<td>FR</td>
<td>Consider transition relations of a tree automaton in a frontier-to-root direction.</td>
</tr>
<tr>
<td>RA-LOOPS</td>
<td>Use a root-to-frontier directed tree automaton with a single root accepting state with ‘self loops’.</td>
</tr>
<tr>
<td>DET</td>
<td>Use a deterministic version of an automaton.</td>
</tr>
<tr>
<td>MATCH-SET</td>
<td>Use an item set and a match set function indicating the items matching a tree.</td>
</tr>
<tr>
<td>REC</td>
<td>Recursively compute match set values.</td>
</tr>
<tr>
<td>FILTER</td>
<td>Use a filtering function in the computation of match set function values.</td>
</tr>
</tbody>
</table>

\(^1\)More or less\(^2\), since the matcher additionally has an output function indicating the tree patterns matching for a state.
| TABULATE  | Use a tabulated version of the match set function, in which a bijection is used to identify match sets by integers. |
| S-PATH    | Decompose patterns into stringpaths and determine matching patterns based on their matching stringpaths. |
| SP-MATCHER| Use an automaton as a pattern matcher for a set of stringpaths in a root-to-frontier subject tree traversal. |
| ACA-SPM   | Use an (optimal) Aho-Corasick automaton and define transition and output functions in terms of that automaton. |
| DRFTA-SPM | Use a DRFTA as a stringpath matcher and define transition and output functions in terms of that automaton. |

The details used for describing the construction algorithms for tree pattern matchers will be described in Section 6.5, while the various filtering functions for detail filter will be described in Section 6.6.

Throughout this chapter, we will use the following pattern set as a running example. Recall from Definition 3.1.26 that ranked alphabet $(\Sigma', r')$ is obtained by extending $(\Sigma, r)$ by variable $\nu$ with $r'(\nu) = 0$.

**Example 6.1.2.** We define pattern set $P = \{p_1, p_2, p_3\} \subseteq T^r(\Sigma', r')$ with

\[
p_1 = a \quad \downarrow_{\nu}, \quad p_2 = a \quad \downarrow\nu \quad d \quad \downarrow_{\nu}, \quad p_3 = b \quad \downarrow\nu \quad d.
\]

**Remark 6.1.3.** We do not discuss the theoretical (worst case) time and space complexity of the algorithms in this taxonomy in detail, but focus on classifying and comparing algorithms based on algorithm details and practical performance. Some practical performance details are discussed in Chapter 8. We briefly consider the complexity of the more concrete algorithms near or at the leaves of the taxonomy graph.

- Algorithm (T-MATCHER, RF) basically tries to match every pattern at every node, and therefore has a time complexity of $\mathcal{O}(|t| \times \text{PatSize})$, where PatSize is the sum of pattern sizes over the patterns in $P$.

- Algorithm (T-MATCHER, FR) and descendants are essentially linear in the size of the subject tree, i.e. are $\mathcal{O}(|t|)$ for a tree $t$: for every node, a state is computed just once, requiring a single transition table lookup.

- Algorithm (MATCH-SET, REC, TABULATE) and Algorithm (MATCH-SET, REC, FILTER, TABULATE) are clearly linear as well, given the relationship between match sets and states that is shown in Section 6.6.1.
• Algorithm (s-path, sp-matcher, det, aca-spm) has a worst case matching
time of O(|t| × PatSize). Its initial presentation in [HO82b] includes a more
detailed analysis of its complexity, as does the work in [Bau96, Kos89]. Algo-
rithm (s-path, sp-matcher, det, drfta-spm) has the same time complexity.
(Preprocessing for the automata used in the two algorithms is different how-
ever, as shown in Section 6.7, and its time complexity is different for the two
cases, as evidenced by the practical results in Section 8.6.2).

The time and space needed to preprocess pattern sets and construct (deterministic)
root-to-frontier tree automata / tabulate match sets is discussed briefly towards
the end of Section 6.6.1 (for the basic tabulation) and 6.6.3 (for tabulation with
filtering).

More detailed analyses of time and space complexity of tree pattern matching algo-
rithms and of the associated preprocessing algorithms can be found in e.g. [CPT92,
Bau96].

\[\]

6.1.1 Related work

Other solution techniques for tree pattern matching exist. For example, the subject
tree can be encoded as a string [Kos89, RR92, DGM94], allowing the use of string
pattern matching techniques, or the problem can be transformed into an instance of
the subset matching problem (a generalization of string pattern matching in which
pattern and string positions correspond to a set of characters instead of a single
character) [CH97, CHI99]. These algorithms improve the worst-case time bounds,
but do so by using more elaborate algorithms, constructing larger auxiliary data
structures or preprocessing the subject tree. It is unclear whether the resulting
algorithms are more efficient in practice (and experiments in [Bau96] show that the
algorithms in [RR92, DGM94] are not). Algorithms preprocessing the subject tree
also appear in [LW96].

It is also possible to generalize other techniques from string pattern matching, such
as the use of shift functions in combination with automata as in the Boyer-Moore
and Commentz-Walter string pattern matching algorithms and similar ones [BM77,
CW79a, CW79b, CWZ04b, WZ96, Wat95]. In [Wat97], Watson sketches such a
generalization to trees, in which trees are traversed root-to-frontier while string path
matching takes place in the opposite direction. In the string case, the algorithms
use jumps from one position in a string to a position further to the right in a string.
Here, its direct generalization would require jumps from one node to a possibly
distant descendant. Watson’s algorithm therefore has to use a somewhat different
approach to circumvent this problem. No more detailed work on this or similar
algorithms and their efficiency has appeared, but it is unlikely that such approaches
can be employed efficiently.

Tree pattern matching and tree automata also play a role in XML processing. XML
trees are generally unranked and may be unordered trees and are thus different from
the kind of trees discussed here. As a result, the techniques used are different as well. In particular, regular (string) expressions may be used to describe the allowed sequences of child nodes of a node and related string matching techniques may be used in algorithms for processing such trees. A brief introduction to the use of formal language theory in XML processing can be found in e.g. [Nev02b], while a more extensive survey on the use of automata in particular appears as [Sch07].

The tree pattern matching algorithms discussed in [HO82b], which are discussed as part of the taxonomy, were intended for use in term rewriting by Hoffmann & O’Donnell. In later work, O’Donnell [O’D85] indicates that match set algorithms based on tabulation (as in the second branch of our taxonomy) were not used by them eventually, and mentions both size of the resulting tables and generation time as being problematic. It seems that based on their results and the fact that both subject tree (as a result of rewrite steps) and pattern set (as a result of rewrite rule set changes) change rather frequently, such algorithms were discarded for use in term rewriting systems by others. Term rewriting system implementations based on innermost rewriting often use pattern matching and application of rewrite rules as the subject tree to be rewritten is constructed in memory. Either a specific kind of pattern matching automaton [Wal91, Chapter 3] or direct root-to-frontier tree pattern matching is used [BHK02].

When using tree pattern matching algorithms described in the taxonomy in term rewriting system implementations, two issues come up. The first is the issue of updating match registrations after a single rewrite rule. Some work dealing with this issue—for various rewriting strategies, and both frontier-to-root as well as root-to-frontier tree pattern matching—appears in [Wit88]. The second issue is the fact that tree patterns are nonlinear in such applications: there is no single variable symbol $\nu$, but different variables may occur in a pattern, and a pattern matches additionally requires different occurrences of the same variable to correspond to equivalent subject tree parts. The latter can be checked after matching when using one of the algorithms in the taxonomy, but will not be discussed further here. (See e.g. [Bau96] for more information.)

### 6.2 The problem and some naive solutions

We first define the tree pattern matching problem more formally and then present a number of abstract solutions whose correctness is easily established. They are not very practical, but provide the taxonomy’s top part, from which parts with more concrete and practical algorithms can be derived. The taxonomy part discussed here is depicted using solid lines in Figure 6.2.1. Chapter 1 included the following informal statement of the tree pattern matching problem:

Given a finite, non-empty set of trees (the pattern set) and an input tree, find the set of all occurrences of the patterns in the input tree.
Here, we define the problem as follows (using Match from Definition 3.1.27):

**Definition 6.2.1** (Tree Pattern Matching (TPM)). The TPM problem for a pattern set \( P \subseteq Tr(\Sigma', r') \) is:

Given an input tree \( t \), determine

\[
\langle \text{Set } p, n : p \in P \land n \in Dt \land \text{Match}(p, t, n) : (p, n) \rangle.
\]

The problem could of course be restricted, for example to a single pattern, or to determine the set of matches for a single node (particularly the root node) only. (In applications of the latter case, preprocessing of the subject tree may make sense. References to work using this solution technique were given in the preceding section.) We do not explicitly treat such restrictions.

Note that in the problem definition above, a single set representing all matches is to be computed. We can rephrase the problem to determine a set of matches per pattern, or per node. The former may be useful in term rewriting systems. The latter will be used in this chapter, as it is used in many existing algorithms for tree pattern matching.

**Definition 6.2.2** (Tree Pattern Matching per pattern (TPMP)). The TPMP problem for a pattern set \( P \subseteq Tr(\Sigma', r') \) is:

Given an input tree \( t \), determine for every \( p \in P \)

\[
\langle \text{Set } n : n \in Dt \land \text{Match}(p, t, n) : n \rangle.
\]

**Definition 6.2.3** (Tree Pattern Matching per node (TPMN)). The TPNM problem for a pattern set \( P \subseteq Tr(\Sigma', r') \) is:

Given an input tree \( t \), determine for every \( n \in Dt \)

\[
\langle \text{Set } p : p \in P \land \text{Match}(p, t, n) : p \rangle.
\]
As an abbreviation, we define a function yielding the set of patterns matching a tree:

**Definition 6.2.4** (Matching Patterns). Given a pattern set \( P \subseteq Tr(\Sigma', r') \), function \( MP \in Tr(\Sigma, r) \rightarrow \mathcal{P}(P) \) is defined for every \( u \in Tr(\Sigma, r) \) by

\[
MP(u) = \langle \text{Set } p : p \in P \land \text{Match}(p, u, \varepsilon) : p \rangle.
\]

Using this definition as an abbreviation, the TPMn problem thus is to determine \( MP(t/n) \) for every node \( n \) of input tree \( t \).\(^2\) To store the computed sets of matching patterns, we assume a data structure \( O \in D_t \rightarrow \mathcal{P}(P) \) that will be used to store the set of matching patterns for each node in \( D_t \). (In an implementation of trees based on objects or records linked by pointers or references, the sets could be stored using an extra field in the objects or records.)

Based on Definition 6.2.3 for the TPMn problem and Definition 6.2.4 for function \( MP \), we can state a first algorithmic solution of TPMn, in which the value of \( O \) is computed for every node:

**Algorithm 6.2.5**

\[
\begin{align*}
\text{const } & P : \mathcal{P}(Tr(\Sigma', r')); \\
t : Tr(\Sigma, r); \\
\text{var } & O : D_t \rightarrow \mathcal{P}(P) \\
\text{for } & n : n \in D_t \rightarrow \\
& O(n) := MP(t/n) \\
\text{end for} \\
\{ \forall n : n \in D_t : O(n) = MP(t/n) \} \\
\end{align*}
\]

Since Algorithm 6.2.5 forms the root of the taxonomy, it is identified by the empty sequence of details. As in the tree acceptance taxonomy, this initial algorithm is trivially correct yet highly abstract.

One way to obtain more concrete solutions is by introducing a tree automaton as a pattern matcher, similar to the use of a string automaton as a pattern matcher for the keyword pattern matching problem.

**Detail 6.2.6** (T-MATCHER). Use a tree automaton as a pattern matcher for a set of patterns.

As we will see in Section 6.5, the automata constructions for tree pattern matching yield automata without \( \varepsilon \)-transitions, and we obtain the following algorithm when

\(^2\)This is similar to determining the set of string matches starting or ending at each position of the input in the case of string matching.
applying detail T-matcher. Note that this algorithm uses function $RSt$ of Definition 3.4.15, which yields the set of states which may appear at the root of a given tree in a tree state assignment respecting the transition relations $R$.

**Algorithm 6.2.7 (T-matcher)**

\[
\begin{align*}
| & \text{const } P : \mathcal{P}(Tr(\Sigma', r')); \\
    & t : Tr(\Sigma, r); \\
    & \text{var } O : D_t \rightarrow \mathcal{P}(P) \\
| & \text{let } M = (Q, \Sigma, r, R, Q_{ra}) \text{ be an } \varepsilon \text{-free TA and} \\
    & \text{Output } \in Q \rightarrow \mathcal{P}(P) \text{ be such that} \\
    & \text{Output}(RSt(u)) = MP(u) \text{ for every } u \in Tr(\Sigma, r); \\
    & \text{for } n : n \in D_t \rightarrow \\
    & O(n) := \text{Output}(RSt(t/n)) \\
| & \text{for } n : n \in D_t \rightarrow \\
    & \{ \langle \forall n : O(n) = MP(t/n) \rangle \} \\
| \end{align*}
\]

Algorithm 6.2.7 does not yet specify how to determine $\text{Output}(RSt(t/n))$. Based on Definition 3.4.15 for $RSt$, this can be determined by computing all tree state assignments to $t/n$, in some order, then considering the states assigned to node $n$ in such assignments, and applying $\text{Output}$ to those states. This does not seem particularly efficient. Similar to what we did for algorithms using tree acceptors in the preceding chapter, we will therefore consider algorithms using pattern matchers that are directed root-to-frontier or frontier-to-root in Sections 6.3 and 6.4. Root-to-frontier ones turn out to be even more difficult to use efficiently than root-to-frontier tree acceptors are. The construction of (undirected as well as directed) tree pattern matchers for use in the above and derived algorithms is treated in Section 6.5.

Similar to the introduction of match sets to obtain more concrete tree acceptance algorithms, we may use match sets to obtain more concrete tree pattern matching algorithms. Such match sets indicate the set of Items matching a tree, where set Items consists of the subtrees of the patterns. By taking the intersection of a match set with the pattern set, we obtain the set of matching patterns.

**Detail 6.2.8 (match-set).** Use an item set and a match set function indicating the items matching a tree.

**Algorithm 6.2.9 (match-set)**

\[
\begin{align*}
| & \text{const } P : \mathcal{P}(Tr(\Sigma', r')); \\
    & t : Tr(\Sigma, r); \\
    & \text{var } O : D_t \rightarrow \mathcal{P}(P)
\end{align*}
\]
let $Items = \text{Subtrees}(P)$;
let $MS \in \text{Tr}(\Sigma, r) \rightarrow \mathcal{P}(Items)$ such that
$$MS(u) = \langle \text{Set } p : p \in Items \land \text{Match}(p, u, \varepsilon) : p \rangle$$
for every $u \in \text{Tr}(\Sigma, r)$;
for $n : n \in D_t \to$
$$O(n) := MS(t/n) \cap P$$
rof
$$\{ \langle \forall n : n \in D_t : O(n) = MP(t/n) \rangle \}$$

This algorithm does not specify how to compute $MS(t/n)$. As for tree acceptance,
$MS$ can be defined recursively, based on a recursive definition of predicate $\text{Match}$,
as will be discussed in Section 6.6.

As in Chapter 5, from here on we consider the subject tree to be a globally accessible
value and use a (reference to a) node of the tree as a parameter to functions like
$MS$, instead of using the subtree pointed to by such a reference. That is, we use a
signature $D_t \rightarrow \mathcal{P}(Items)$ instead of $\text{Tr}(\Sigma, r) \rightarrow \mathcal{P}(Items)$ for function $MS$. This is
again done for efficiency and for providing context information.

As a third solution strategy, each pattern can be decomposed into its stringpaths and
keyword pattern matching techniques can be used. Based on the stringpaths whose
matches start at a node, the patterns matching at the node can be determined. The
basic idea behind such algorithms is that a pattern matches at a subject tree node
if and only if all its stringpaths match starting at this node, based on the relation
between $\text{Match}$ and $\text{SPMatch}$ given by Lemma 3.1.33.

**Detail 6.2.10 (s-path).** Decompose patterns into stringpaths and determine
matching patterns based on their matching stringpaths.

Using this detail and introducing a variable $\text{OSP}$ to store stringpath matches at their
beginpoints, we obtain the following algorithm:

**Algorithm 6.2.11 (s-path)**

$$\langle \text{const } P : \mathcal{P}(\text{Tr}(\Sigma', r')); t : \text{Tr}(\Sigma, r); \rangle$$
$$\text{var } O : D_t \rightarrow \mathcal{P}(P);$$
$$\text{OSP} : D_t \rightarrow \mathcal{P}(\text{SPaths}(P))$$
for $n : n \in D_t \rightarrow$
$$O(n) := \langle \text{Set } p : p \in P \land s \in \text{SPaths}(p) \land \text{SPMatch}(s, t, n) : s \rangle;$$
$$\text{OSP}(n) := \langle \text{Set } p : p \in P \land \text{SPaths}(p) \subseteq \text{OSP}(n) : p \rangle$$
rof
$$\{ \langle \forall n : n \in D_t : O(n) = MP(t/n) \rangle \}$$
The algorithm does not specify how to compute the sets $OSP(n)$ of matching string-paths. In Section 6.7 on page 187 we will consider the use and construction of different kinds of automata to determine stringpath matches in algorithms derived from Algorithm 6.2.11 (S-PATH) above.

6.3 Using root-to-frontier pattern matchers

Figure 6.3.1 Tree pattern matching taxonomy, with solid lines and circles emphasizing taxonomy parts described in Section 6.3.

Figure 6.3.1 highlights the part of the taxonomy discussed here. As neither the algorithms for root-to-frontier tree pattern matching nor constructions for the tree automata used in them appear in the literature, the discussion will be brief.

As in Chapter 5, detail RF may be used to obtain an algorithm using a root-to-frontier directed tree automaton.

**Detail 6.3.1 (RF).** Consider transition relations of a tree automaton in a root-to-frontier direction.

Since $q \in RSt(t) \equiv Accept(t,q)$ according to Lemma 3.4.21, we can use the root-to-frontier acceptance function $Accept \in Tr(\Sigma, r) \times Q \rightarrow \mathbb{B}$ of Definition 3.4.20.

Applying this in a naive way and not making any additional assumptions about the NRFTA used—apart from the restriction on the underlying undirected TA and output function as in the let-clause in Algorithm 6.2.7 (T-MATCHER)—yields an Algorithm (T-MATCHER, RF).

The algorithm—not explicitly presented here—recursively computes $Accept(t/n,q)$ for every $n \in D_t$ and for every $q \in Q_r$, and adds $Output(q)$ to $O(n)$ if and only if $Accept(t/n,q) \equiv true$. Such an algorithm seems very inefficient, possibly recomputing the value of $Accept$ for a given subtree and state numerous times.

**Remark 6.3.2.** In fact, the NRFTAs constructed in Section 6.5 for use with Algorithm (T-MATCHER, RF) allow the restriction of $q \in Q$ to $q \in Q_r$, i.e. from
all states to the root accepting ones, as precisely these correspond to full patterns in $P$ (i.e. $M$ and $Output$ are such that $Output(RSt(u) \cap Q_{ra}) = MP(u)$ for every $u \in Tr(\Sigma, r)$).

In fact, the structure of the automata is such that the algorithm (with the restriction mentioned above) corresponds to trying to match (in a root-to-frontier direction) every pattern at every node.\footnote{This makes it similar to using a string acceptor as a keyword pattern matcher by starting it from its start state at every position of an input string (as mentioned in Remark 6.1.1).}

In string pattern matching, a more efficient pattern matcher may be obtained by adding a 'self loop' (i.e. a transition leading from a state to itself) to the start state of the string automaton and then determinizing the resulting automaton by applying a subset construction. A similar addition could be considered here.

We do not present such a construction here, as $NRFTAs$ or $\varepsilon NRFTAs$ seem hard to use efficiently by themselves, while the corresponding $DRFTAs$ are not considered for two reasons:

- Such $DRFTAs$ will have an output function yielding the complete pattern set for its single root accepting state and thus cannot easily be used as a pattern matcher for a set of patterns.

- As mentioned in Section 3.4.3, deterministic root-to-frontier tree automata are less powerful than $NRFTAs$, and the subset construction on an $NRFTA$ leads to an equivalent $DRFTA$ if and only if the $NRFTA$'s language is path closed (see Definition 3.1.43), which may not be the case for a pattern set. (Strictly speaking, equivalence additionally requires the $NRFTA$ to have no root accepting reachable yet useless states. The $NRFTAs$ resulting from Constructions (TPM-TA:ALL-SUB:RF) satisfy that property.)

In Section 6.7 however, a different kind of $DRFTA$ will be used as a stringpath matcher, solving the tree pattern matching problem in a different way.

### 6.4 Using frontier-to-root pattern matchers

Figure 6.4.1 highlights the part of the taxonomy discussed here. Literature references for algorithms using frontier-to-root tree pattern matchers will be omitted here and instead be given in Section 6.5.3 when considering constructions for such pattern matchers.

As in Chapter 5 we can use detail FR to get an algorithm using an FR-directed TA.

**Detail 6.4.1 (FR).** Consider transition relations of a tree automaton in a frontier-to-root direction.\qed
Using this detail, we can use the recursive version of function $RSt$ of Lemma 3.4.19. We obtain the following algorithm, in which $\text{Traverse}$ is such that $\text{Traverse}(n) = RSt(t/n)$ and stores values in $O$ as a side effect. As $\text{Traverse}$ is based on the recursive version of $RSt$, the algorithm can use $\text{Traverse}(\varepsilon)$ as its main line, instead of the for-loop iterating over $D_t$ used in Algorithm 6.2.7 (T-Matcher).

In this and following algorithms, we will use $OPred.1$ as an abbreviation for the predicate

$$\langle \forall m : t \cdot m \in D_t : O(t \cdot m) = MP(t/(t \cdot m)) \rangle,$$

i.e. indicating that matching patterns have been correctly computed and stored for the subtree of $t$ rooted at node $i$.

**Algorithm 6.4.2** (T-Matcher, FR)

\[
\begin{align*}
\text{const} & \quad P : \mathcal{P}(\text{Tr}(\Sigma', r')); \\
& \quad t : \text{Tr}(\Sigma, r); \\
\text{var} & \quad O : D_t \rightarrow \mathcal{P}(P); \\
& \quad qs : \mathcal{P}(Q) \\
| & \quad \text{let } M = (Q, \Sigma, r, R, Q_{ra}) \text{ be an NFRTA and} \\
& \quad \text{Output} \in Q \rightarrow \mathcal{P}(P) \text{ be such that} \\
& \quad \text{Output}(RSt(u)) = MP(u) \text{ for every } u \in \text{Tr}(\Sigma, r); \\
\end{align*}
\]

\[\text{Traverse}(\varepsilon, qs)\]

\[
\{ \langle \forall n : n \in D_t : O(n) = MP(t/n) \rangle \}
\]

\[
\text{proc } \text{Traverse}(\downarrow n : D, \uparrow s : \mathcal{P}(Q)) = \\
\{ \text{Pre: } n \in D_t \}
\]

\[
\begin{align*}
| & \quad \text{var } s_1, \ldots, s_n : \mathcal{P}(Q) \\
| & \quad \text{let } a = t(n); \\
& \quad \text{if } n > 0 \rightarrow \\
& \quad \quad \text{for } i : 1 \leq i \leq n \rightarrow \\
& \quad \quad \quad \{ n \cdot i \in D_t \}
\end{align*}
\]
\[ \text{Traverse}(n \cdot i, s_i) \]
\[ \{ \ s_i = RSt(t/(n \cdot i)) \land OPred.(n \cdot i) \ \} \]
\[ \text{rof:} \]
\[ s := \emptyset; \]
\[ \text{for} \ (q_1, \ldots, q_n) : q_1 \in s_1, \ldots, q_n \in s_n \rightarrow \]
\[ s := s \cup R_a(q_1, \ldots, q_n) \]
\[ \text{rof} \]
\[ | \ n = 0 \rightarrow \]
\[ s := R_a() \]
\[ \text{fi:} \]
\[ \{ \ s = RSt(t/n) \ \} \]
\[ O(n) := Output(s) \]
\[ \} \]
\[ \{ \ \text{Post:} \ s = RSt(t/n) \land OPred. \} \]
\]

### 6.4.1 Using deterministic frontier-to-root tree pattern matchers

Algorithm (T-MATCHER, FR) assumes the use of an NFRTA. In case a DFRTA is used, procedure Traverse can be simplified.

**Detail 6.4.3 (DET).** Use a deterministic version of an automaton. \(\square\)

We give the resulting Algorithm (T-MATCHER, FR, DET):

**Algorithm 6.4.4 (T-MATCHER, FR, DET)**

\[ \| \ \text{const} \ P : \mathcal{P}(Tr(\Sigma', r')); \]
\[ t : \mathcal{P}(\Sigma, r); \]
\[ \text{var} \ O : D_t \rightarrow \mathcal{P}(P); \]
\[ q : Q \]
\[ | \ \text{let} \ M = (Q, \Sigma, r, R, Q_a) \text{ be a DRFTA and} \]
\[ \text{Output} \in Q \rightarrow \mathcal{P}(P) \text{ be such that} \]
\[ \text{Output}(RSt(u)) = MP(u) \text{ for every } u \in Tr(\Sigma, r); \]

\[ \text{Traverse}(\varepsilon, q) \]
\[ \{ \ \forall n : n \in D_t : O(n) = MP(t/n) \} \]

\[ \text{proc} \ Traverse(\downarrow n : D, \uparrow q : Q) = \]
\[ \{ \ \text{Pre:} \ n \in D_t \ \} \]
\[ \| \ \text{var} \ q_1, \ldots, q_n : Q \]
\[ | \ \text{let} \ a = t(n); \]
\[ | \ \text{if} \ n > 0 \rightarrow \]
for \( i : 1 \leq i \leq n \rightarrow \)
\[
\{ \ n \cdot i \in D_i \} \]
\[
\text{Traverse}(n \cdot i, q_i) \]
\[
\{ \ \{q_i\} = \text{RSt}(t/(n \cdot i)) \land \text{OPred.}(n \cdot i) \} \]
\[
\text{ref;} \]
\[
q := R_n(q_1, \ldots, q_n) \]
\[
\| \ n = 0 \rightarrow \]
\[
q := R_n() \]
\[
\text{fi;} \]
\[
\{ \ \{q\} = \text{RSt}(t/n) \land \forall i : 1 \leq i \leq n : \text{OPred.}(n \cdot i) \} \]
\[
O(n) := \text{Output}(q) \]
\[
\| \}
\[
\{ \ \text{Post: } \{q\} = \text{RSt}(t/n) \land \text{OPred.n} \} \]
\[
\|
\]

### 6.5 Constructing tree pattern matchers

As in the previous chapter, various constructions resulting in tree pattern matchers and corresponding output functions can be defined. We consider constructions for various kinds of pairs of tree automata \( M \) and output functions \( Output \) such that \( Output(\text{RSt}(u)) = \text{MP}(u) \) for every tree \( u \), given a pattern set \( P \).

As we will see however, the constructions we present here are based on a single state set—corresponding to all the subtrees of the patterns in \( P \)—and transformation \( \text{REM-}\varepsilon \) does not play a role. The constructions thus are closely related, as they differ only in the following aspects:

- Whether the automata are undirected, root-to-frontier directed—indicated by label \( \text{RF} \)—or frontier-to-root directed—indicated by label \( \text{FR} \),

- in the case of constructions for directed automata, whether the automata are deterministic or not—indicated by label \( \text{SUBSET}_{\text{RF}} \) or \( \text{SUBSET}_{\text{FR}} \) depending on the direction.

By combining the possible choices for these aspects, we can obtain just five different constructions, only four of which seems interesting and will therefore be considered here. As before, constructions are identified by the sequence of labels indicated, preceded this time by a label \( \text{TPM-TA} \) to indicate they are for \( \text{TAS} \) solving the TPMn problem. The table below lists the discussed constructions together with references to the section in which they are discussed.
We first consider the basic construction for undirected TAs in Section 6.5.1. Its root-to-frontier directed version is briefly discussed in Section 6.5.2. The nondeterministic frontier-to-root version is presented in Section 6.5.3, while its deterministic counterpart is considered in Section 6.5.3.1.

Recall that a tree automaton is defined by a 5-tuple \((Q, \Sigma, r, R, Q_{\text{ra}})\). Since the pattern set \(P\) already (implicitly) defines the ranked alphabet \((\Sigma', r')\) and hence \((\Sigma, r)\), the tree pattern matcher constructions only specify \(Q, R\) and \(Q_{\text{ra}}\) to completely define a tree automaton and \textit{Output} to define the corresponding output function.

The construction presentation is similar to that in the preceding chapter, starting with some introductory text and the definition of any new details involved. The construction is then defined formally by specifying \(Q, R, Q_{\text{ra}}\) and \textit{Output}, followed by an example of its application, a discussion of its correctness and finally a comparison to related constructions and reference to occurrences of the construction in the literature. Since the later constructions are variations of the first one, one or more of the parts mentioned may be omitted, e.g. when obvious or irrelevant.

### 6.5.1 A construction for undirected tree automata

The basic construction we consider uses subtrees of patterns as states. Its transition relations encode the relations between (tuples of) such states, based on the relation between a tree and its direct subtrees. The intuition behind the construction is that the state(s) assigned to the root of a tree \(t\) in a tree state assignment based on a tree automaton correspond to the subtree(s) of patterns that match the tree. We present the construction and give an example before proving this.

**Detail 6.5.1 (ALL-SUB).** Use all subtrees of patterns in a pattern set as states in a construction.

**Construction 6.5.2 (TPM-TA:ALL-SUB).**

\textbf{Input} Pattern set \(P \subseteq \text{Tr}(\Sigma', r')\)

\textbf{Output} TA \(M = (Q, \Sigma, r, R, Q_{\text{ra}})\) where
Figure 6.5.1 TA resulting from Construction 6.5.2 (TPM-TA:ALL-SUB). Note the loop for state $q_2$, which depicts a transition $(q_2, (q_2)^n) \in R_a$ for every $a \in \Sigma$.

\[
Q = \text{Items} = \text{Subtrees}(P)
\]
\[
R_a = \left\{ \begin{array}{l}
\text{Set} \ p, p_1, \ldots, p_n : p = a(p_1, \ldots, p_n) \\
\wedge \ p, p_1, \ldots, p_n \in \text{Items}
\end{array} \right\}
\]
\[
\cup \left\{ \begin{array}{l}
\emptyset & \text{if } \nu \notin \text{Items} \\
\{ (\nu, \nu^n) \} & \text{if } \nu \in \text{Items}
\end{array} \right\}
\]
\[
R_\epsilon = \emptyset \\
Q_{ra} = P
\]

and function $Output : Q \rightarrow \mathcal{P}(P)$ defined by

\[
Output(q) = \{ q \} \text{ if } q \in Q_{ra}
\]
\[
Output(q) = \emptyset \text{ if } q \notin Q_{ra}
\]

Note that in the definition of $R_a$ the right part of the set union ensures that a transition relating state $\nu$ to a tuple of states $\nu$ exists for each alphabet symbol $a$ if and only if $\nu$ occurs in some pattern of $P$. This ensures that $\nu \in RSt_M(t)$ for every tree $t$, corresponding to $\nu$ matching every tree.

Example 6.5.3. Applying Construction 6.5.2 (TPM-TA:ALL-SUB) to pattern set $P = \{ p_1, p_2, p_3 \}$ of Example 6.1.2 results in the TA depicted in Figure 6.5.1.

The correspondence between state labels used and subtrees they represent is:

\[
q_0 = \begin{array}{c}
a \\
b \\
p_1 \\
\end{array} \quad (= q_2 = \nu) \quad q_4 = \begin{array}{c}
a \\
b \\
p_2 \\
d \\
\end{array} \quad (= q_6 = b \ (= p_3))
\]
\[
q_1 = \begin{array}{c}
b \\
c \\
\end{array} \quad q_3 = c \quad q_5 = d
\]
Note that \( Q_{ra} = \{ q_0, q_1, q_6 \} \), hence Output is defined for \( q_0, q_4 \), and \( q_6 \) by \( q_0(= p_1) \), \( q_4(= p_2) \) respectively \( q_6(= p_3) \), and as \( \emptyset \) for other states.

The transition relations are such that \((q_2, (q_2)^n) \in R_a\) for every \( a \in \Sigma \), and the only other transitions in the \( R_a \) are as indicated below.

\[
R_a \supseteq \{ (q_0, (q_1, q_2)), \quad (q_4, (q_2, q_5)) \} \\
R_b \supseteq \{ (q_1, (q_3)), \quad (q_6, (q_5)) \} \\
R_c \supseteq \{ (q_3, ()) \} \\
R_d \supseteq \{ (q_5, ()) \}
\]

We compute the possible state assignments for a subject tree \( a(b(c), a(c, c)) \):

\[
RSt(a) = \{ q_3, q_2 \} \\
RSt(b(c)) = \{ q_1, q_2 \} \\
RSt(a(c, c)) = \{ q_2 \} \\
RSt(a(b(c), a(c, c))) = \{ q_0, q_2 \}
\]

As function Output for states \( q_0, q_2 \) has values \( \{ q_0 \} \) and \( \emptyset \) respectively, and \( q_0 \) corresponds to pattern \( p_1 \), this pattern matches at the root of the subject tree.

For a subject tree \( a(b(c), d) \), additionally to the above \( RSt \) values we compute:

\[
RSt(d) = \{ q_5, q_2 \} \\
RSt(a(b(c), d)) = \{ q_0, q_4, q_2 \}
\]

As Output for states \( q_0, q_4, q_2 \) equals \( \{ q_0 \}, \{ q_4 \} \) and \( \emptyset \) respectively, patterns \( p_1 \) and \( p_2 \) both match at the root of this tree.

\[\square\]

**Lemma 6.5.4.** For every \( t \in Tr(\Sigma, r) \), \( \alpha \in Q \) we have

\[\alpha \in RSt_M(t) \equiv \text{Match}(\alpha, t, \varepsilon).\]

**Proof.** We prove this property for \( \alpha = \nu \) and \( \alpha \neq \nu \) separately.

Case \( \alpha = \nu \):

\[
\alpha \in RSt_M(t) \\
\equiv \{ \text{Lemma 3.4.19, } R_c = \emptyset \} \\
(\alpha, ()) \in R_a \\
\equiv \{ \text{definition of } R_a, \ \alpha = \nu, \ \nu \in Q = \text{Items} \} \\
\text{true} \\
\equiv \{ \alpha = \nu, \ \varepsilon \in D_t, \ \text{Definition 3.1.29} \} \\
\text{Match}(\alpha, t, \varepsilon)
\]

For \( \alpha \neq \nu \), we prove it by induction on the depth of \( t \).

Case \( t = a \in \Sigma_0 \):
\[
game \in RSt_M(t)
\equiv \begin{cases} 
\text{Lemma 3.4.19, } R_e = \emptyset \end{cases}
\]
\[
\langle \alpha, () \rangle \in R_a
\equiv \begin{cases} 
definition of R_a, \ \alpha \neq \nu, \ \alpha \in Q = \text{Items} \end{cases}
\]
\[
\alpha = a = t(e) \land a \in \text{Items}
\equiv \begin{cases} 
\Rightarrow: \ t = a, \ e \in D_t : \Leftarrow: \alpha \in \text{Items} \end{cases}
\]
\[
\varepsilon \in D_t \land t(e) = a = \alpha
\equiv \begin{cases} 
\alpha \neq \nu, \ e \in D_t, \text{Definition 3.1.29} \end{cases}
\]
\[
\text{Match}(\alpha, t, \varepsilon)
\]

Case \( t = a(t_1, \ldots, t_n), a \in \Sigma \setminus \Sigma_0 \): \[
\alpha \in RSt_M(t)
\equiv \begin{cases} 
\text{Lemma 3.4.19, } R_e = \emptyset \end{cases}
\]
\[
\langle \exists \overline{q} : \overline{q} \in Q^n : \langle \alpha, \overline{q} \rangle \in R_a \land \langle \forall i : 1 \leq i \leq n : \pi_i(\overline{q}) \in RSt_M(t_i) \rangle \rangle
\equiv \begin{cases} 
definition of R_a, \ \alpha \neq \nu, \ \alpha \in Q = \text{Items, Items is subtree closed} \end{cases}
\]
\[
\langle \exists \overline{q} : \overline{q} \in Q^n : \alpha = a(\pi_1(\overline{q}), \ldots, \pi_n(\overline{q})) \\
\text{and } \langle \forall i : 1 \leq i \leq n : \pi_i(\overline{q}) \in RSt_M(t_i) \rangle \rangle
\equiv \begin{cases} 
\text{Induction Hypothesis, } t_i(\varepsilon) = t(i) \end{cases}
\]
\[
\langle \exists \overline{q} : \overline{q} \in Q^n : \alpha = a(\pi_1(\overline{q}), \ldots, \pi_n(\overline{q})) \\
\text{and } \langle \forall i : 1 \leq i \leq n : \text{Match}(\pi_i(\overline{q}), t, i) \rangle \rangle
\equiv \begin{cases} 
\varepsilon \in D_t, \ t(\varepsilon) = a, \text{Definition 3.1.29} \end{cases}
\]
\[
\text{Match}(\alpha, t, \varepsilon)
\]

\( \square \)

Using the definition of \( M \) and \( \text{Output} \) in Construction (TPM-TA:ALL-SUB) and of \( MP \) in Definition 6.2.4 we obtain the following result.

**Corollary 6.5.5.** For every \( t \in Tr(\Sigma, r) \)

\[
\text{Output}(RSt(t)) = MP(t).
\]

\( \square \)

The tree automata and output functions resulting from the construction may thus be used with Algorithm 6.2.7 (T-MATCHER) to solve the TPMt problem.
Literature reference 6.5.6. Construction (TPM-TA:ALL-SUB) corresponds to the undirected TA construction of Ferdinand, Seidl & Wilhelm [FSW94, Section 5], although their definition does not use an explicit output function.

6.5.1.1 Relation to match set computation

Match set function MS introduced in Algorithm 6.2.9 (MATCH-SET) and the computation of a TA resulting from Construction (TPM-TA:ALL-SUB) can be related quite easily, analogous to the case for tree acceptors in the preceding chapter:

Lemma 6.5.7. Let MS be as in Algorithm (MATCH-SET) and let M be constructed according to Construction (TPM-TA:ALL-SUB), both using the same pattern set P as input, then for every $t \in Tr(\Sigma, r)$, $MS(t) = RSt_M(t)$.

Proof. We derive

$$MS(t)$$

$$\begin{aligned}
&= \{ \text{ definition of } MS \text{ in Algorithm (MATCH-SET) } \}
&= \langle \text{Set } p : p \in \text{Items} \land \text{Match}(p, t, \varepsilon) : p \rangle
&= \{ \text{ Items } = Q \text{ in construction; Lemma 6.5.4, } RSt_M(t) \subseteq Q \}
&= RSt_M(t)
\end{aligned}$$

As directing transitions in an automaton does not influence set Q or $RSt_M(t)$, this lemma also holds for constructions obtained from Construction (TPM-TA:ALL-SUB) by considering the automata to be directed. Such constructions are discussed further on.

6.5.2 Constructions for root-to-frontier tree pattern matchers

By applying detail RF, we obtain a version of Construction 6.5.2 (TPM-TA:ALL-SUB) in which transition relations are directed root-to-frontier. We omit the resulting Construction (TPM-TA:ALL-SUB:RF), as it merely differs from the first one in the way in which the transition relations are viewed.

Example 6.5.8. Applying Construction (TPM-TA:ALL-SUB:RF) to the pattern set of Example 6.1.2 results in the (\varepsilon-free) NRFTA in Figure 6.5.2, a directed version of the TA in Figure 6.5.1. The correspondence between state labels and trees they represent is as in Example 6.5.3.

In Section 6.7 we will consider a different DRFTA construction that can be used to solve the tree pattern matching problem by using it as a stringpath matcher.
6.5 Constructing tree pattern matchers

![Figure 6.5.2 (ε-free) NRFTA resulting from Construction (TPM-TA:ALL-SUB:-RF)](image)

6.5.3 Constructions for frontier-to-root tree pattern matchers

A frontier-to-root directed version of Construction 6.5.2 (TPM-TA:ALL-SUB) can be obtained by applying detail FR. The following example shows the FR-directed version of the undirected and RF-directed examples seen earlier.

**Example 6.5.9.** Applying Construction (TPM-TA:ALL-SUB:FR) to the pattern set of Example 6.1.2 results in the (ε-free) NRFTA in Figure 6.5.3, an FR-version of the TA in Figure 6.5.1 and NRFTA of Figure 6.5.2. The correspondence between state labels and trees they represent is as in Example 6.5.3.

![Figure 6.5.3 (ε-free) NRFTA resulting from Construction (TPM-TA:ALL-SUB:-FR)](image)

**Literature reference 6.5.10.** Although Ferdinand, Seidl & Wilhelm state that their definition of tree pattern matching is undirected, their construction in [FSW94, Section 5] can easily be viewed as (and in fact is essentially used as) one for NRFTAs. See Literature reference 6.5.6 as well.
6.5.3.1 Deterministic construction

By applying subset construction $\text{SUBSET}_{\text{FR}}$ to the NFRTA definition of Construction TPM-TA:ALL-SUB:FR, and defining an appropriate output function based on the output function defined by the construction, we can obtain constructions resulting in DFRTAs usable as pattern matchers. As with the use of $\text{SUBSET}_{\text{FR}}$ in tree acceptor constructions, either all $2^{|Q|}$ subsets—with $Q$ the state set of the underlying NFRTA—or just the reachable subsets can be constructed. The reachability-based algorithm, very similar to the one for DFRTAs in the context of tree acceptance, will be discussed in Section 6.6.

**Definition 6.5.11.** Let $M' = (Q', \Sigma, r, R', Q'_{r_{a}})$ be a DFRTA obtained for a pattern set $P$ using the application of detail $\text{SUBSET}_{\text{FR}}$ to the NFRTA definition in Construction (TPM-TA:ALL-SUB:FR). Its corresponding output function $\text{Output}' \in Q' \rightarrow \mathcal{P}(P)$ is defined for every $q' \in Q'_{r_{a}}$ by

$$\text{Output}'(q') = \left\{ q : q \in q' \land q \in P : \text{Output}(q) \right\}.$$

**Example 6.5.12.** Applying Construction (TPM-TA:ALL-SUB:FR:SUBSET$_{\text{FR}}$) to the pattern set of Example 6.1.2 (and only constructing reachable subsets of underlying NFRTA states/pattern subtrees) results in the DFRTA in Figure 6.5.4. Transitions that are not shown lead to state $\{q_2\}$. Underlying NFRTA state identifiers used in the state subsets are as in earlier examples.

**Literature reference 6.5.13.** The construction already appears in the work of Hoffmann & O’Donnell [HO82b, Sections 3–4], although they phrase their discussion in terms of recursive match set computation, considered in Section 6.6.

The construction appears in its essential form in the earlier work of Kron [Kro75, Chapter 4]. Instead of directly using DFRTAs, he uses so-called orthogonal tree automata (OTA). Such an OTA exists for every symbol $a$. It forms an encoding of the (non-trivial) values of $R_a$ based on a decision tree or trie whose branch sequences are labeled by state tuples of the DFRTA for such transition function values. Given the DFRTA state tuple $q_1, \ldots, q_n$, the resulting state is computed using the OTA for symbol $a$, by taking sequential transitions on $q_1, \ldots, q_n$ from the initial state of this OTA. Given the last OTA state reached, a function ‘final’ determines the corresponding DFRTA state $R_a(q_1, \ldots, q_n)$. The actual OTAs constructed are compacted versions of the tries and form deterministic, directed acyclic finite string automata over alphabet $Q$, the state set of the DFRTA.

Construction (TPM-TA:ALL-SUB:FR:SUBSET$_{\text{FR}}$) constructs DFRTAs isomorphic to the ones constructed by using the DFRTA construction described by Ferdinand, Seidl & Wilhelm [FSW94, Section 5]. It corresponds to their initial TA construction followed by application of their ‘Subset Construction I’ (yielding a DFRTA with all subsets of
states of the underlying TA) or ‘Subset Construction II’ (yielding one with just the reachable subsets).

6.6 Recursive match set computation

In this section, we consider algorithms based on a recursive definition of function $MS$. The taxonomy part discussed is depicted using solid lines in Figure 6.6.1.

Algorithm (match-set) in Section 6.2 did not provide a definition of the match set function $MS$ that was directly computable but merely specified that, given $Items = Subtrees(P)$, the function needed to satisfy, for every $u \in Tr(\Sigma, r)$,

$$MS(u) = \{ p : p \in Items \land Match(p, u, \varepsilon) : p \}.$$ 

Since $Match$ can be defined recursively, this leads naturally to a recursive definition for $MS$, which may then be used to compute match set values during a recursive traversal of a subject tree.

As was the case for the tree acceptance match set function in Section 5.7, the recursive definition depends on two auxiliary functions: one to compose item set elements
with an alphabet symbol, and one to compute the closure. The only difference
between the two cases is in the latter function: there, it adds nonterminal left hand
sides reachable from full right hand sides in a match set, while here it merely adds
the variable $\nu$ (assuming $\nu \in \text{Items}$).

**Specification 6.6.1** (Composition of item set elements with alphabet symbol).
For every $a \in \Sigma$, function $\text{Comp}_a \in \mathcal{P}(\text{Items})^n \to \mathcal{P}(\text{Items})$ is specified for every
$U \in \mathcal{P}(\text{Items})^n$ by

$$\text{Comp}_a(U) = \langle \text{Set} p_1, \ldots, p_n : p_1 \in U_1, \ldots, p_n \in U_n : a(p_1, \ldots, p_n) \rangle \cap \text{Items}. \quad \square$$

Note that this specification is exactly the same as Specification 5.7.1. As we did
there, we will often use $\text{Comp}_a$ instead of $\text{Comp}_a(())$ for $a \in \Sigma_0$.

**Specification 6.6.2.** Closure function $\text{Cl} \in \mathcal{P}(\text{Items}) \to \mathcal{P}(\text{Items})$ is specified for
every $U \in \mathcal{P}(\text{Items})$ by

$$\text{Cl}(U) = U \cup (\{\nu\} \cap \text{Items}). \quad \square$$

Note that $\{\nu\} \cap \text{Items}$ equals $\{\nu\}$ if and only if at least one pattern in $P$ has a leaf
labeled $\nu$, i.e. taking the closure amounts to adding the pattern subtree $\nu$, based on
$\nu$ matching every tree.

Given the auxiliary functions, function $MS$ can be defined exactly as in Section 5.7:

**Definition 6.6.3.** Function $MS \in \text{Tr}(\Sigma, r) \to \mathcal{P}(\text{Items})$ is defined for every $t = a(t_1, \ldots, t_n) \in \text{Tr}(\Sigma, r)$ by

$$MS(t) = \text{Cl}(\text{Comp}_a(MS(t_1), \ldots, MS(t_n))). \quad \square$$
Example 6.6.4. For tree $a(b(c), d)$ and the pattern set $P = \{p_1, p_2, p_3\}$ over $(\Sigma', r')$ of Example 6.1.2—i.e. with

$$
p_1 = \begin{array}{c}
\begin{array}{c}
\begin{array}{c}
a \downarrow \nu \\
\begin{array}{c}
b \\
c
\end{array}
\end{array}
\end{array}
\end{array}, \quad
p_2 = \begin{array}{c}
\begin{array}{c}
\begin{array}{c}
a \downarrow \nu \\
\begin{array}{c}
\nu \\
d
\end{array}
\end{array}
\end{array}
\end{array}, \quad
p_3 = \begin{array}{c}
\begin{array}{c}
\begin{array}{c}
b \\
d
\end{array}
\end{array}
\end{array}

\end{array}
$$

—the match set values are as follows, where $\nu$ as added by the closure function is separated from other elements by a semicolon:

- $MS(c) = \{c; \nu\}$
- $MS(d) = \{d; \nu\}$
- $MS(b(c)) = \{b(c); \nu\}$
- $MS(a(b(c), d)) = \{a(b(c), \nu), a(\nu, d); \nu\}$

Lemma 6.6.5. For every $t \in Tr(\Sigma, r)$

$$
MS(t) = \langle \text{Set } p : p \in \text{Items} \land Match(p, t, \varepsilon) : p \rangle.
$$

Proof idea. The lemma can be proven using structural induction on $t$. The proof is very similar to that of the corresponding Lemma 5.7.5 for tree acceptance.

This lemma allows the use of the recursive definition of $MS$ with Algorithm 6.2.9 (match-set). As before, we use a function $MS_{\text{node}}$ with signature $D \to \mathcal{P}(\text{items})$ i.e. taking a reference to a node as a parameter, such that $MS_{\text{node}}(n) = MS(t/n)$.

Detail 6.6.6 (rec). Recursively compute match set values.

Algorithm 6.6.7 (match-set, rec)

```plaintext
|| const $P : \mathcal{P}(Tr(\Sigma', r'))$;
\quad $t : Tr(\Sigma, r)$;
\var O : D_t \to \mathcal{P}(P);
\quad $s : \mathcal{P}(\text{Items})$
\let Comp_a be as in Specification 6.6.1 ;
\let Cl be as in Specification 6.6.2 ;

$MS_{\text{node}}(\varepsilon, s)$
\{ $\langle \forall n : n \in D_t : O(n) = MP(t/n) \rangle$ \}

proc $MS_{\text{node}}(\downarrow n : D_t, \uparrow s : \mathcal{P}(\text{Items})) =$
\{ $\quad \text{Pre: } n \in D_t$ \}
\\| $\var s_1, \ldots, s_n : \mathcal{P}(\text{items})$
\let a = t(n);
```
if \( n > 0 \) →
for \( i : 1 \leq i \leq n \) →
\{ \( n \cdot i \in D_t \) \}
\[ MS_{node}(n \cdot i, s_i) \]
\{ \( s_i = MS(t/(n \cdot i)) \land OPred.(n \cdot i) \) \}
\[ \langle \forall i : 1 \leq i \leq n : s_i = MS(t/(n \cdot i)) \land OPred.(n \cdot i) \rangle \]
\( s := Cl(Comp_a(s_1, \ldots, s_n)) \)
\[ n = 0 \rightarrow \]
\( s := Cl(Comp_a) \)
fi;
\( O(n) := s \cap P \)
\[ \langle Post: s = MS(t/n) \land OPred.n \rangle \]
As with Algorithm 5.7.7 (match-set, rec) in the preceding chapter, the above algorithm assumes the existence of functions \( Comp_a \) and \( Cl \), here satisfying Specifications 6.6.1 and 6.6.2. Implementation of these functions is straightforward and therefore omitted here.

Similar to what we saw in Chapter 5, we have the following correspondences:

- The direct computation of match set values using functions \( Cl \) and \( Comp_a \) corresponds to the simulation of the NFRTA of Construction (TPM-TA:ALL-SUB:FR) using Algorithm 6.4.2 (T-MATCHER, FR).
- As a result, the compositions \( Cl \circ Comp_a \), thus correspond to the transition functions \( R_a \) of the derived DFRTA construction (TPM-TA:ALL-SUB:FR:SUBSET\( R_a \)).

As we already considered these correspondences in detail for the tree acceptance case in the preceding chapter (see the end of Sections 5.7 as well as Section 5.7.1), and the case of tree pattern matching is very similar, we do not consider them in detail in the current chapter.

### 6.6.1 Using tabulated match set values

In this section, the match set function (i.e. the transition function of a DFRTA) is tabulated to reuse computed values and decrease running time, at the expense of pre-computation time and table storage. Using such tabulation, an algorithm traversing a subject tree from frontier to root only needs to perform a \( (n \text{-dimensional}) \) table lookup per node.

As in Section 5.7.3, such tabulation of \( Cl \circ Comp_a \)'s leads to \( n \text{-dimensional tables} \ T_a \) for each \( a \in \Sigma \) of rank \( n \).
6.6 Recursive match set computation

To further reduce storage space and computation time, the tabulation is done for reachable match set values only, i.e. for the reachable part of \( \mathcal{P}(\text{Items}) \). This reachable part is likely to be relatively small, as any match set containing trees of the form \( a(t_1, \ldots, t_n) \) and \( b(u_1, \ldots, u_m) \) with \( a \neq b \) obviously is not reachable. In addition to tabulation itself, a bijection between match set values and integers is used to allow tables to be indexed by and contain integers.

The following specification then equals Specification 5.7.15, except that \( \mathcal{C}l \) in it of course refers to the \( \mathcal{C}l \) for tree pattern matching as in Specification 6.6.2.

**Specification 6.6.8** (Tabulation of match sets identified by integers). Let \( Z \) be the part of \( \mathcal{P}(\text{Items}) \) reachable under the set of \( \mathcal{C}l \circ \text{Comp}_a \), let \( m \in [0 \ldots |Z|] \) \( \rightarrow \) \( Z \) be a bijection and let tables \( T_a \in [0 \ldots |Z|]^n \rightarrow [0 \ldots |Z|] \) for every \( a \in \Sigma \) be such that for every \( qz \in [0 \ldots |Z|]^n \)

\[
m(T_a(qz)) = \mathcal{C}l(\text{Comp}_a(m(qz_1), \ldots, m(qz_n))).
\]

\( \Box \)

**Detail 6.6.9** (Tabulate). Use a tabulated version of the match set function, in which a bijection is used to identify match sets by integers.

\( \Box \)

**Algorithm 6.6.10** (\textsc{match-set, rec, tabulate})

\[
\begin{align*}
| \text{const} & \ P : \mathcal{P}(\text{Tr}(\Sigma', r')); \\
| t : \text{Tr}(\Sigma, r); \\
| \text{var} & \ O : D_t \rightarrow \mathcal{P}(P); \\
| qz : [0 \ldots |Z|] & | \\
| \text{let} & \ m \in [0 \ldots |Z|] \rightarrow Z \text{ as in Specification 6.6.8} ; \\
| T_a & \in [0 \ldots |Z|]^n \rightarrow [0 \ldots |Z|] (\forall a \in \Sigma \setminus \Sigma_0) \text{ be as in Specification 6.6.8} ; \\
| T_a & \in [0 \ldots |Z|]^n (\forall a \in \Sigma_0) \text{ be as in Specification 6.6.8} ; \\
\end{align*}
\]

\( MS'_{\text{node}}(\varepsilon, s) \)
\[
\{ \langle \forall n : n \in D_t : O(n) = MP(t/n) \rangle \}
\]

\[
\begin{align*}
\text{func} & \ MS'_{\text{node}}(\downarrow n : D_t, \uparrow qz : [0 \ldots |Z|]) = \\
| \text{Pre:} & \ n \in D_t \} \\
| | \text{var} & \ qz_1, \ldots, qz_n : [0 \ldots |Z|] \\
| \text{let} & \ a = t(n); \\
| \text{if} & \ n > 0 \rightarrow \\
| \text{for} & \ i : 1 \leq i \leq n \rightarrow \\
| | \{ \ n \cdot i \in D_t \} \\
| | \ MS'_{\text{node}}(n \cdot i, qz_i) \\
| | \{ qz_i = m^{-1}(MS(t/(n \cdot i))) \wedge OPred.(n \cdot i) \} \\
| \text{rof:} & \\
| | \{ \langle \forall i : 1 \leq i \leq n : qz_i = m^{-1}(MS(t/(n \cdot i))) \wedge OPred.(n \cdot i) \rangle \}
\end{align*}
\]
\[ q_z := T_a(q_{z1}, \ldots, q_{zn}) \]
\[ n = 0 \quad \rightarrow \quad q_z := T_a \]
\[ \text{fi} ; \]
\[ O(n) := m(q_z) \cap P \]

Note that for function \( MS'_{node} \) above and function \( MS_{node} \) as in Algorithm 6.6.7 (MATCH-SET, REC) we have \( MS_{node} = m \circ MS'_{node} \).

To tabulate reachable values of \( MS \), reachability-based Algorithms 5.7.19 and 5.7.21 of Section 5.7.3.1 can again be used, albeit with function \( Cl \) as in Specification 6.6.2 instead of as in Specification 5.7.2. Since the latter is the only difference, we merely give an example here. (The example is similar to Example 5.7.23. That example includes an operational description of the algorithm’s application, which is omitted here.)

**Example 6.6.11.** For the pattern set of Example 6.1.2, the algorithm generates the following tables, assuming that the nondeterministic choice in the for-loops of the above algorithm is resolved by considering alphabet symbols and tuples of states in lexicographical order.

<table>
<thead>
<tr>
<th>( q )</th>
<th>( m(q) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{c; \nu}</td>
</tr>
<tr>
<td>1</td>
<td>{d; \nu}</td>
</tr>
<tr>
<td>2</td>
<td>{\nu}</td>
</tr>
<tr>
<td>3</td>
<td>{b(c); \nu}</td>
</tr>
<tr>
<td>4</td>
<td>{a(\nu, d); \nu}</td>
</tr>
<tr>
<td>5</td>
<td>{b(d); \nu}</td>
</tr>
<tr>
<td>6</td>
<td>{a(b(c), \nu); \nu}</td>
</tr>
<tr>
<td>7</td>
<td>{a(b(c), \nu), a(\nu, d); \nu}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( T_a )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( T_b )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

\( T_c = 0 \quad T_d = 1 \)

The tabulated match set values/DFRTA states correspond (modulo some renaming)
6.6 Recursive match set computation 181
to those of the dfrta of Example 6.5.12 depicted in Figure 6.5.4 (note that not all transitions leading to state \(q_2\) (state 2 here) are depicted in the figure).

For example, that dfrta has a state \(\{q_2, q_3\}\), say \(q_1\), with \(\{(q_1, ())\} = R_d\) i.e. \(q_1\) will be the state assigned to a leaf labeled \(d\). According to Example 5.6.3, \(q_2\) corresponds to \(\nu\) and \(q_3\) to \(d\), i.e. the state corresponds to a match set containing \(d\) and \(\nu\). Here, state 1 corresponds to the same match set, and \(T_d = 1\) i.e. this match set will be computed for a leaf labeled \(d\).

The use of the tables in Algorithm 6.6.10 (match-set, rec, tabulate) results in the following values of \(MS'_{\text{node}}\) and \(O\) for tree \(a(b(c), d)\):

\[
MS'_{\text{node}}(1 \cdot 1) = T_e = 0; \quad O(1 \cdot 1) = m(0) \cap P = \emptyset \\
MS'_{\text{node}}(1) = T_b(0) = 3; \quad O(1) = m(3) \cap P = \emptyset \\
MS'_{\text{node}}(2) = T_d = 1; \quad O(2) = m(1) \cap P = \emptyset \\
MS'_{\text{node}}(\varepsilon) = T_a(3, 1) = 7; \quad O(\varepsilon) = m(7) \cap P = \{p_1, p_2\}
\]

Hence the only matches are those of \(p_1\) and \(p_2\) occurring at the root of the tree. For tree \(a(d, c)\) the values are:

\[
MS'_{\text{node}}(1) = T_d = 1; \quad O(1) = m(1) \cap P = \emptyset \\
MS'_{\text{node}}(2) = T_e = 0; \quad O(2) = m(0) \cap P = \emptyset \\
MS'_{\text{node}}(\varepsilon) = T_a(1, 0) = 2; \quad O(\varepsilon) = m(2) \cap P = \emptyset
\]

Hence no matches occur in this tree.

As shown by Hoffmann & O’Donnell [HO82b], tabulation takes \(O(|Z|^{(r_{\text{max}}+1)} \times |\Sigma| \times \text{PatSize})\) time and the tables use \(O(|Z|^{r_{\text{max}} \times |\Sigma|})\) space, where \(\text{PatSize}\) is the sum of pattern sizes over the patterns in \(P\).

As with tree acceptance, the tabulation algorithm only constructs reachable match sets, but in the worst case there may still be \(|\mathcal{P}(\text{Items})|\) such sets, i.e. \(Z = \mathcal{P}(\text{Items})\) may hold. In the experimental results shown in Section 8.6.2, just as in the experimental results for tree acceptance, the worst case does not occur. Hoffmann & O’Donnell [HO82b] also indicate that the worst case does not seem to occur in practice.

Since tabulation time and space may nevertheless be quite large, we mention a number of—not necessarily mutually exclusive—techniques to reduce number of entries or memory usage of the tables, some of which were already mentioned in Section 5.7.3.1.

- Minimization of the underlying dfrtas. A brief discussion with references can be found in the aforementioned section.
- Reducing the number of items. This was mentioned (and applied in experiments) in that section, but does not seem possible for the case of tree pattern matching, except by reducing the size of pattern set \(P\).
• Restricting the pattern set. Hoffmann & O’Donnell [HO82b] consider pattern sets in which no two patterns are independent (two patterns are independent if there is a tree for which one matches, another tree for which the other matches, and a third for which both match). They call such a pattern set a simple pattern forest and present a match set tabulation algorithm with a better worst case construction time and table space for it [HO82b, Section 6]: \( O(PatSize^2 \times r_{max} + SGHeight \times |\Sigma| \times PatSize^{r_{max}}) \) (where \( SGHeight \) is the height of the subsumption tree constructed during preprocessing, bounded from above by \( PatSize \) construction time and \( O(PatSize^{r_{max}} \times |\Sigma|) \) table space. They claim the restriction is not a problem for the application of tree pattern matching to the kind of term rewriting they had in mind.

In [CPT92, Section 5] an improved algorithm with better bounds for the case of simple pattern forests than Hoffmann & O’Donnell’s algorithm is considered. Since the techniques discussed below put no restriction on the pattern set, we do not consider tabulation for simple pattern forests in more detail.

• Filtering, i.e. using specific properties of match sets to remove certain items from them before computing the parent node match set based on the match sets for child nodes, in effect reducing the domain of the match set function. This approach was discussed in Section 5.7.4 and turned out to greatly improve memory usage there. It is not surprising that the same approach is also used for tree pattern matching, as we discuss in Section 6.6.2. This technique is used in the context of tree pattern matching in the work of Chase [Cha87]—where it originates—but also in that of Cai, Paige and Tarjan [CPT92]. (The improved algorithm for simple pattern forests in [CPT92, Section 5] that we mentioned above also applies filtering.) We discuss filtering for the case of tree pattern matching in the next section.

• Standard table/matrix compression techniques, or (in the case of representation by OTAs as in Kron’s work) standard DAG/acyclic DFA compression techniques. These are treated in e.g. [BMW91, FH91] and—for the case of OTAs—in [FSW94].

6.6.2 Filtering match sets

Filtering of match sets, as considered for tree acceptance in Section 5.7.4, may also be applied in the case of match set computation for tree pattern matching.

By itself, i.e. without tabulation of match set functions, the technique does not seem useful. The combination of filtering with match set tabulation can be useful and is discussed in Section 6.6.3.

We refer the reader to Section 5.7.4 for a discussion of and definitions related to filtering, as the concept of filtering and the definitions of the four filters are exactly the same for the two cases. Here, we merely repeat the corresponding detail and
present the resulting algorithm skeleton for filter-based tree pattern matching algorithms. As before, any of the filter functions can be used for $\text{Filt}$ in the algorithm, while using the identity function yields Algorithm 6.6.7 ($\text{MATCH-SET, REC}$).

**Detail 6.6.12** (FILTER). Use a filtering function in the computation of match set function values.

**Algorithm 6.6.13** (MATCH-SET, REC, FILTER)

\[
\begin{align*}
&\text{const } P : \mathcal{P}(T(S, r)) ; \\
&\quad t : T(S, r) ; \\
&\quad \text{var } O : D_t \to \mathcal{P}(P) \\
&\quad \text{let } \text{Comp}_a \text{ be as in Specification 6.6.1} ; \\
&\quad \text{let } \text{Cl} \text{ be as in Specification 6.6.2} ; \\
&\quad MS_{\text{node}}(\varepsilon) \\
&\quad \{ (\forall n : n \in D_t : O(n) = \text{MP}(t/n) \} \\
\end{align*}
\]

\[
\begin{align*}
&\text{func } MS_{\text{node}}(\langle n : D \rangle) : \mathcal{P}(\text{Items}) = \\
&\quad \{ \text{Pre: } n \in D_t \} \\
&\quad \{ \text{var } s_1, \ldots, s_n : \mathcal{P}(\text{Items}) \\
&\quad \text{let } a = t(n) ; \\
&\quad \text{if } n > 0 \to \\
&\quad \text{for } i : 1 \leq i \leq n \to \\
&\quad \quad \{ n \cdot i \in D_t \} \\
&\quad \quad s_i := MS_{\text{node}}(n \cdot i) \\
&\quad \quad \{ s_i = MS(t/(n \cdot i)) \land \text{Opred}(n \cdot i) \} \\
&\quad \quad \text{ref:} \\
&\quad \quad \{ \langle i : 1 \leq i \leq n : s_i = MS(t/(n \cdot i)) \land \text{Opred}(n \cdot i) \} \\
&\quad \quad MS_{\text{node}} := \text{Cl}(\text{Comp}_a(Filt_{a,1}(s_1), \ldots, Filt_{a,n}(s_n))) \\
&\quad \quad \text{if:} \\
&\quad \quad O(n) := MS_{\text{node}} \cap P \\
&\quad \quad \{ \text{Post: } MS_{\text{node}} = MS(t/n) \land \text{Opred.n} \} \\
\end{align*}
\]

**6.6.3 Using tabulated match set values with filtering**

In this section, we combine tabulation of match sets with filter functions, leading to algorithms using tabulated match set and filter function values.

When using tabulated match set values without filtering, the resulting tables $T_a$ may become very large, even when constructed for reachable match sets only. By using
filtering, the match set function’s domain is reduced, leading to a reduction in size of the match set tables \( T_a \).

We assume the filter functions to be precomputed and tabulated as well. Provided the extra precomputation cost is not too high, the combination is useful compared to tabulation without filtering when the size of the filter tables does not offset the reduction in size of match set tables and the cost of the additional filter table lookups is not too large. As with match set tabulation without filtering, only the reachable match set values will be tabulated.

**Specification 6.6.14** (Tabulation of filters and filtered match sets, identified by integers\(^4\)). Let \( Z \) be the part of \( \mathcal{P}(\text{Items}) \) reachable under the set of \( Cl \circ \text{Comp}_a \) and (for all \( a \in \Sigma, 1 \leq i \leq n \)) let \( Z_{a,i} = \text{Filt}_{a,i}(Z) \).

Let \( m \in [0 \ldots |Z|] \rightarrow Z \) and (for all \( a \in \Sigma, 1 \leq i \leq n \)) \( \text{Rep}_{a,i} \in [0 \ldots |Z_{a,i}|] \rightarrow Z_{a,i} \) be bijections.\(^5\)

Let tables \( T_a \in [0 \ldots |Z_{a,1}|] \times \ldots \times [0 \ldots |Z_{a,n}|] \rightarrow [0 \ldots |Z|] \) for every \( a \in \Sigma \) and let tables \( \phi_{a,i} \in [0 \ldots |Z|] \rightarrow [0 \ldots |Z_{a,i}|] \) for all \( a \in \Sigma, 1 \leq i \leq n \) be such that

\[
\phi_{a,i} = \text{Rep}_{a,i}^{-1} \circ \text{Filt}_{a,i} \circ m
\]

(stated otherwise, \( \text{Rep}_{a,i} = \text{Filt}_{a,i} \circ m \circ \phi_{a,i}^{-1} \); this determines \( \text{Rep}_{a,i} \) up to permutation) and, for every \( \overrightarrow{qz} \in [0 \ldots |Z|]^n \),

\[
m(T_a(\phi_{a,1}(qz_1), \ldots, \phi_{a,n}(qz_n))) = Cl(\text{Comp}_a(\text{Filt}_{a,1}(m(qz_1)), \ldots, \text{Filt}_{a,n}(m(qz_n)))).
\]

In this specification,

- the \( \phi_{a,i} \) map integers corresponding to match sets to integers corresponding to their filtered versions i.e. \( a, i \)-representer sets, while
- the \( \text{Rep}_{a,i} \) form a bijection between the latter integers and the corresponding filtered match sets/a, i-representer sets.

The above yields the following algorithm:

**Algorithm 6.6.15** (MATCH-SET, REC, FILTER, TABULATE)

\[
\| \text{const } P : \mathcal{P}(\text{Tr}(\Sigma', r')); \quad t : \text{Tr}(\Sigma, r); \quad \text{var } O : D_t \rightarrow \mathcal{P}(P)
\]

\(^4\)The specification equals Specification 5.7.36 except for using \( Cl \) as in Specification 6.6.2.

\(^5\)\( \text{Rep} \) since the \( \text{Rep}_{a,i} \) are the bijections for the \( a, i \)-representer sets.
let \( m \in [0\ldots[Z]) \rightarrow Z \),
\[
\begin{align*}
&\forall a \in \Sigma, 1 \leq i \leq n, \\
&\forall a \in \Sigma, 1 \leq i \leq n, \\
&T_a \in [0\ldots[Z] \times \ldots \times [0\ldots[Z] \rightarrow [0\ldots[Z]) \forall a \in \Sigma \setminus \Sigma_0 \text{ and} \\
&T_a \in [0\ldots[Z] ) (\forall a \in \Sigma_0) \text{ be as in Specification 6.6.14};
\end{align*}
\]

\[
MS'_{\text{node}}(\varepsilon) \begin{cases} \\
\forall n : n \in D_t : O(n) = MP(t/n) \} \\
\end{cases}
\]

\[
\text{func } MS'_{\text{node}}(\downarrow n : D) : [0\ldots[Z]) = \\
\begin{cases} \\
\text{Pre: } n \in D_t \\
|\text{ var } qz_1, \ldots, qz_n : [0\ldots[Z]) \\
|\text{ let } a = t(n); \\
|\text{ if } n > 0 \rightarrow \\
|\text{ for } i : 1 \leq i \leq n \rightarrow \\
|\quad \{ n \cdot i \in D_t \} \\
|\quad qz_i := MS'_{\text{node}}(\downarrow n \cdot i) \\
|\quad \{ qz_i = m^{-1}(MS(t/(n \cdot i))) \wedge \text{OPred}(n \cdot i) \} \\
|\text{ ref: } \\
|\quad \{ \forall i : 1 \leq i \leq n : qz_i = m^{-1}(MS(t/(n \cdot i))) \wedge \text{OPred}(n \cdot i) \} \\
|\quad MS'_{\text{node}} := T_a(\phi_{a,1}(qz_1), \ldots, \phi_{a,n}(qz_n)) \\
|\text{ if } n = 0 \rightarrow \\
|\quad MS'_{\text{node}} := T_a \\
|\\text{ Post: } MS'_{\text{node}} = m^{-1}(MS(t/n)) \wedge \text{OPred}.n \} \\
|\end{cases}
\]

To tabulate filter function and match set values for the reachable part of \( \mathcal{P}(\text{Items}) \),
Algorithm 5.7.38 in Section 5.7.5.1 can be used, albeit with function \( \text{Cl} \) as in Specification 6.6.2. We refer the reader to Section 5.7.5.1 for a discussion of the algorithm and give examples of its application to just one of the four filters here.

**Example 6.6.16.** For the pattern set of Example 6.1.2, the algorithm generates the following tables when applied with filter \( \text{CFilter}_{a,i} \):

<table>
<thead>
<tr>
<th>( q )</th>
<th>( m(q) )</th>
<th>( \phi_{a,1}(q) )</th>
<th>( \phi_{a,2}(q) )</th>
<th>( \phi_{6,1}(q) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{ e; \nu }</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>{ d; \nu }</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>{ \nu }</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>{ a(\nu, d); \nu }</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>{ b(\nu); \nu }</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>{ b(d); \nu }</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>{ a(b(\nu), \nu); \nu }</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>{ a(b(\nu), a(\nu, d); \nu }</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
The use of the tables in Algorithm 6.6.15 (match-set, rec, filter, tabulate) results in the following values of $MS'_{node}$ and $O$ for tree $a(b(c), d)$:

$MS'_{node}(1 \cdot 1) = T_c = 0;\ O(1 \cdot 1) = m(0) \cap P = \emptyset$
$MS'_{node}(1) = T_b(\phi_{b,1}(0)) = T_b(0) = 4;\ O(1) = m(4) \cap P = \emptyset$
$MS'_{node}(2) = T_d = 1;\ O(2) = m(1) \cap P = \emptyset$
$MS_{node}(\varepsilon) = T_a(\phi_{a,1}(4), \phi_{a,2}(1)) = T_a(1, 1) = 7;\ O(\varepsilon) = m(7) \cap P = \{p_1, p_2\}$

Hence the only matches are those of $p_1$ and $p_2$ occurring at the root of the tree. For tree $a(d, c)$ the values are:

$MS'_{node}(1) = T_d = 1;\ O(1) = m(1) \cap P = \emptyset$
$MS'_{node}(2) = T_c = 0;\ O(2) = m(0) \cap P = \emptyset$
$MS'_{node}(\varepsilon) = T_a(\phi_{a,1}(1), \phi_{a,2}(0)) = T_a(0, 0) = 2;\ O(\varepsilon) = m(2) \cap P = \emptyset$

Hence no matches occur in this tree.

As exemplified above, the use of filtering may reduce the size of the tables $T_a$ at the expense of additional filter and representor tables $\phi$ and $Rep$. Detailed experimental results on the reduction are discussed in Section 8.6.2.

**Remark 6.6.17.** Cai, Paige and Tarjan [CPT92] present an improvement on the time bounds of the above tabulation algorithm by Chase. According to them, Chase’s algorithm has a preprocessing time bound of $O(|Z|r_{max} \times |\Sigma| \times r_{max})$ time and the tables use $O(|Z|r_{max} \times r_{max})$ space.

Cai, Paige and Tarjan use tree-shaped data structures instead of multi-dimensional arrays to encode the filter function and match set function values. They claim that their resulting algorithm improves on the worst case time bound of Chase’s and that it is roughly 10 times faster in a preliminary experiment using the algorithm as part of a transformation system. They do not give any more detail on the experiment performed and no later work by them on the subject seems to have appeared.
6.7 Stringpath-based match set computation

In this section, we discuss algorithms based on stringpath matching. The part of the taxonomy containing such algorithms is depicted using solid lines in Figure 6.7.1. The basic idea behind Algorithm 6.2.11 (s-PATH) in Section 6.2 was that a tree corresponds to a set of stringpaths and that tree pattern matching can be expressed and solved in terms of tree pattern stringpath matching. The algorithm did not yet specify how to compute the matching pattern stringpaths.

Before considering how to compute the matching stringpaths, we introduce an abbreviation. It is similar to function $MP$ indicating the set of patterns matching a given tree, but instead indicates the set of stringpaths matching (starting at the root of) a given tree.

**Definition 6.7.1** (Matching Stringpaths). Given a pattern set $P \subseteq Tr(\Sigma', r')$, function $MSP \in Tr(\Sigma, r) \rightarrow \mathcal{P}(SPaths(P))$ is defined for every $u \in Tr(\Sigma, r)$ by

$$MSP(u) = \langle \text{Set } sp : sp \in SPaths(P) \land SMatch(sp, u, \varepsilon) : sp \rangle.$$

One way to compute the matching stringpaths would be to traverse the subject tree root-to-frontier and try matching every pattern stringpath at every node. This seems rather inefficient, and we therefore do not consider it any further.

The main ideas behind the approach taken in this section are:

1. As in Algorithm 6.2.11 (s-PATH), a tree pattern is represented by its set of stringpaths.
2. An automaton for matching such stringpaths is used during a root-to-frontier traversal of the subject tree $t$. 

![Figure 6.7.1 Tree pattern matching taxonomy, with solid lines and circles emphasizing taxonomy parts described in Section 6.7.](image-url)
• During this traversal, automaton transitions are taken. The transitions taken are based on the rootpath leading to a node, i.e. on the alternating sequence of node symbols and branch numbers leading from the subject tree’s root to the node.
• Such a traversal allows detection of stringpath matches in such rootpaths at their endpoints (using the output function associated with the automaton). This is similar to the use of e.g. Aho-Corasick automata for keyword pattern matching, which detect keyword matches in an input string at their endpoint.
• Given the stringpath match’s endpoint and the stringpath, stringpath match registration is done at its beginpoint, say m. Such stringpath matches are registered in \( OSP(m) \), using a newly introduced data structure \( OSP \in D_t \to \mathcal{P}(SPaths(P)) \).
• While traversing and registering stringpath matches, a complete pattern match is registered at a node \( m \) whenever all its stringpaths have been registered at a node. Such pattern matches are registered in \( O(m) \), using data structure \( O \in D_t \to \mathcal{P}(P) \) as before.

Assuming \( OSP \) and \( O \) at each node to be initialized to \( \emptyset \), at each node the value of \( OSP \) and \( O \) after the traversal equals the value of \( MSP \) and \( MP \) at the same node respectively.

Before giving the details of the algorithm, we give an intuitive informal example. This example uses an Aho-Corasick automaton for stringpath matching. The construction and use of such an automaton will be considered in detail in Section 6.7.1.

![String automaton used for stringpath match detection.](image)

**Figure 6.7.2** String automaton used for stringpath match detection.

**Example 6.7.2.** The string automaton in Figure 6.7.2 is a matcher for the string-paths of pattern set \( P \) of Example 6.1.2. (Back transitions to state \( q_1 \) on \( a \), to state \( q_{10} \) on \( b \) and to state \( q_0 \) on other symbols are not shown, to increase readability.) For \( t = a(b(c), a(b(c), a(c, c))) \), Figure 6.7.3 shows the automaton state associated...
with each node \( n \) and the matches detected at their endpoints based on a state and symbol \( t(n) \) or \( \nu \), using a root-to-frontier traversal of the tree.

Figure 6.7.4 shows the same tree, but with each stringpath match registered at its beginpoint, i.e. with the values of \( OSP(t/n) \) at each node \( n \), which equal \( MSP(t/n) \) after the entire subject tree has been traversed.

Figure 6.7.5 again shows the tree, but now with each complete pattern match registered at its beginpoint, i.e. with the values of \( O(t/n) \) at each node \( n \), which equal \( MP(t/n) \) after the entire subject tree has been traversed.

We start our formal presentation of the algorithm using the above approach by introducing a new detail:

**Detail 6.7.3 (SP-MATCHER).** Use an automaton as a pattern matcher for a set of stringpaths in a root-to-frontier subject tree traversal.
We do not specify the automaton of the above detail in full yet, but assume that we have a state set $Q$, a (partial) function $\gamma \in Q \times (\Sigma \cdot N_{\leq r}) \rightarrow Q$ and an output function $Output \in Q \times \Sigma' \rightarrow \mathcal{P}(\text{SPaths}(P))$. Note that transition function $\gamma$ assumes transitions on a sequence of two symbols: an alphabet symbol followed by a branch number. The transition function of the Aho-Corasick automaton—such as the one in Example 6.7.2—has transitions on a single alphabet symbol or branch number. Transition function $\gamma$ will therefore need to be related to the transition function $\delta$ of an Aho-Corasick string automaton, but for now it does not matter how they are related.

Remark 6.7.4. Note that we assume $\gamma$ to yield a single state, not a state set. This corresponds to assuming the underlying automaton to be deterministic, i.e. assuming the use of detail DET in addition to detail SP-MATCHER. A nondeterministic automaton and a function yielding a set of states could be used instead, yielding a slightly different algorithm.

Before giving a specification of the additional properties to be satisfied by $\gamma$ and $Output$, we define the following auxiliary function.

**Definition 6.7.5.** Let $Q$ be a state set and $\gamma \in Q \times (\Sigma \cdot N_{\leq r}) \rightarrow Q$ be a (partial) function, then (partial) function $\gamma^* \in Q \times (\Sigma \cdot N_{\leq r})^* \rightarrow Q$ is defined for all $q \in Q$ by

$$\gamma^*(q, \varepsilon) = q$$

and

$$\gamma^*(q, a \cdot i \cdot w) = \gamma^*(\gamma(q, a \cdot i), w)$$

for all $a \in \Sigma', i \in N_{\leq r}, w \in (\Sigma \cdot N_{\leq r})^*$ such that $\gamma(q, a \cdot i)$ is defined.

**Notation 6.7.6.** In the remainder of this section, we will use $\text{SPDepth}(sp)$ as an abbreviation for $|(sp \downarrow N_{\leq r})|$, for every $sp \in \text{SPaths}(P)$. Note that $\text{SPDepth}(sp) = |sp| \div 2$ and indicates the number of branch identifiers occurring in $sp$. 

![Diagram](image_url)
Furthermore, we will use \( SPMatchAtEnd(sp, t, n) \) as an abbreviation for
\[
\{ \exists m : m = n | SPathEnd(sp) : SPMatchAtEnd(sp, t, m) \},
\]
i.e. indicating whether a stringpath matches in a tree ending at node \( n \).

**Specification 6.7.7.** Given a state set \( Q \) with initial state \( q_0 \), (partial) function \( \gamma \in Q \times (\Sigma \times N_{\leq r}) \rightarrow Q \) and output function \( Output \in Q \times \Sigma' \rightarrow 2P(SPaths(P)) \) are such that (using \( \gamma^* \) as in Definition 6.7.5, \( RPath \) as in Definition 3.1.19, and the right take and right drop operators \( | \) and \( \lhd \) as in Definition 2.3.1)
\[
q = \gamma^*(q_0, RPath(t, n)|1)
\]
implies that for every \( a \in \Sigma' \)
\[
Output(q, a) = \left\{ sp, m : sp \in SPaths(P) \land sp|1 = a : sp \right\} \land SPMatchAtEnd(sp, t, n)
\]

The specification states that given the state reached by reading node symbols and branch numbers leading from \( t \)'s root to a given node \( n \), the output function value for this state and a symbol yields the pattern stringpaths for which a match occurs ending at node \( n \) and ending with the symbol. Clearly, this symbol must be either the symbol at the node, i.e. \( t(n) \), or the variable symbol \( \nu \), and the function value will thus always be \( \emptyset \) for other symbols.

**Example 6.7.8.** Consider the annotated tree \( t \) of Figure 6.7.3 and assume that \( \gamma \) is defined in terms of the Aho-Corasick's transition function \( \delta \) such that \( q_2 = \gamma^*(q_0, a2a1) = \gamma^*(q_0, a2a1b|1) = \gamma^*(q_0, RPath(t, n)|1) \) with \( n = 2 \cdot 1 \). Specification 6.7.7 then states that

- \( Output(q_2, \nu) = \{ a1\nu \} \) (since stringpath \( a1\nu \) is the only stringpath ending in \( \nu \) and matching ending at node \( 2 \cdot 1 \), i.e. the stringpath occurs at the end of string \( a2a1\nu \) read from the root of \( t \) to node \( n \) followed by \( \nu \) ),
- \( Output(q_2, b) = \emptyset \) (since no stringpath ending in \( b \) and matching ending at node \( 2 \cdot 1 \) exists, i.e. no stringpath occurs at the end of string \( a2a1 \) read from the root of \( t \) to node \( n \) followed by that node's symbol \( b \) ),
- \( Output(q_2, a) = \emptyset \) for all other \( a \in \Sigma' \).

Thus, using transition function \( \gamma \) and output function \( Output \) in a root-to-frontier subject tree traversal and determining the value of the output function at each node \( n \) for symbols \( \nu \) and \( t(n) \), we obtain the following algorithm, in which:
• All pattern stringpath matches are detected at their endpoints.
• Such matches are then registered at their beginpoint \( m \), i.e. added to \( OSP(m) \), as in procedure Register.
• In the same procedure, complete pattern matches resulting from the addition of such a stringpath match to \( OSP(m) \) are also registered, i.e. added to \( O(m) \).

**Algorithm 6.7.9 (S-PATH, SP-MATCHER, DET)**

```plaintext
|| const \( P : \mathcal{P}(\text{Tr}(\Sigma', r')) \);
\( t : \text{Tr}(\Sigma, r) \);
var \( O : D_t \rightarrow \mathcal{P}(P) \);
\( OSP : D_t \rightarrow \mathcal{P}(\text{SPaths}(P)) \)
| let \( Q, q_0, \gamma, \gamma^* \), and Output be as in Specification 6.7.7 ;

for \( n : n \in D_t \rightarrow \text{O}(n) \), \( OSP(n) := \emptyset, \emptyset \) rof;
Traverse(\( \varepsilon, q_0 \))
\{ \langle \gamma n : n \in D_t : \text{O}(n) = \text{MP}(t/n) \rangle \} \}

proc Traverse(\( \downarrow n : D, \downarrow q : Q \)) =
\{ Pre: \( n \in D_t \land q = \gamma^*(q_0, \text{RPath}(t, n)|1) \) \}
|| var \( q_{\text{next}} : Q \)
| let \( a = t(n) \);
if \( n > 0 \) →
for \( i : 1 \leq i \leq n \) →
\( q_{\text{next}} := \gamma(q, a \cdot i) \);
\{ \( n \cdot i \in D_t \land q_{\text{next}} = \gamma^*(q_0, \text{RPath}(t, n \cdot i)|1) \) \}
\( \text{Traverse}(n \cdot i, q_{\text{next}}) \)
\{ \( \text{Traverse.Post}(n \cdot i).(q_{\text{next}}) \) \}
\rof;
\| n = 0 →
\( \text{Register}(n, q, a) \)
fi;
\( \text{Register}(n, q, \nu) \)
||
\{ Post: \( \text{Traverse.Post.n.q} \) \}

proc Register(\( \downarrow n : D, \downarrow q : Q, \downarrow a : \Sigma' \)) =
\{ Pre: \( n \in D_t \) \}
|| var \( sp : \text{SPaths}(P) \);
\( m : D \)
| for \( sp : sp \in \text{Output}(q, a) \) →
\( m := n | \text{SPDepth}(sp) \);
\( OSP(m) := OSP(m) \cup \{ sp \} \);
```
for $p : p \in P \land sp \in SPaths(p) \rightarrow$

$$\text{as } SPaths(p) \subseteq OSP(m) \rightarrow O(m) := O(m) \cup \{p\}$$

\r

\r

\r

\r

\r

In the algorithm, we used $Traverse.Post.n.q$ and $Register.Post.n.q.a$ as abbreviations for the postconditions of the two procedures. Postcondition $Traverse.Post.n.q$ essentially states that every stringpath match in $t$ whose endpoint is in the subtree $t/n$ has been registered at its $beginpoint$. Formally, it equals (using $\text{pref}_#$ for the proper prefixes of a string, as introduced following Definition 2.3.5)

$$\langle \forall l : l \in D_{(t/n)} : OSP(n \cdot l) = MSP(t/(n \cdot l)) \land O(n \cdot l) = MP(t/(n \cdot l)) \rangle$$

$$\land \langle \forall l : l \in \text{pref}_#(n) : OSP(l) \supseteq \langle \text{Set } sp : sp \in MSP(t/l) \land n \in \text{pref}(l \cdot (sp \downarrow N_{\leq r})) : sp \rangle \land O(l) \supseteq \langle \text{Set } p : p \in P \land SPaths(p) \subseteq OSP(l) : p \rangle \rangle.$$  

A number of notes can be made to explain this postcondition:

- The first conjunct states that for nodes $l$ in subtree $t/n$, $OSP$ and $O$ already have their final value, i.e. equal $MSP$ and $MP$.

- The second conjunct indicates that for nodes $l$ on the path to subtree $t/n$, at least the stringpath matches starting at $l$ and leading through or ending at node $n$ have been registered at their $beginpoint$, and so have patterns for which all stringpaths are registered at such a $beginpoint$. (Stringpath $sp$ matching via node $n$ is expressed by $n \in \text{pref}(l \cdot (sp \downarrow N_{\leq r}))$, noting that $l \in \text{pref}_#(n)$.)

- The values of $OSP$ and $O$ are unchanged otherwise, as the procedure does not remove matches of stringpaths or patterns from these variables.

- Finally, note that for $n = \varepsilon$ and $q = q_0$, the second conjunct of the postcondition above reduces to true and the first conjunct implies the postcondition of the entire algorithm, $\langle \forall n : n \in D_t : O(n) = MP(t/n) \rangle$.

Postcondition $Register.Post.n.q.a$ equals

$$\langle \forall sp, m : m = n | SPathDepth(sp) : \langle \forall p : SPaths(p) \subseteq OSP(m) : p \in O(m) \rangle \rangle.$$  

$$\land sp \in Output(q, a) \land sp \in SPaths(p) \land p \in P \land sp \in OSP(m)$$
In words, it states that every stringpath that is part of the output for the given state \( q \) and output symbol \( a \)—i.e. every stringpath match ending at node \( n \) with a final symbol \( a \) (which therefore must be either \( t(n) \) or \( v \))—has been registered at its beginpoint \( m \), and the value of \( O(m) \) has been correctly updated accordingly.

Different automata and different output and \( \gamma \) functions can be used in this algorithm skeleton. As we already saw in Example 6.7.2, string automata can be used, but a certain kind of tree automata can be used as well. We will consider both in the course of this section. In particular, we will consider:

- The use of deterministic string automata. The construction and use of two such automata will be considered, namely that of the original optimal Aho-Corasick automata (Section 6.7.1) and that of a slight modification which we call \textit{Aho-Corasick stringpath automata} (Section 6.7.1.1).

- The use of deterministic root-to-frontier tree automata. The stepwise construction and subsequent use of such automata based on states corresponding to dotted patterns will be considered in Section 6.7.2. The high-level construction and hence the automata turn out to be highly similar to the Aho-Corasick stringpath automata.

\textbf{Literature reference 6.7.10.} The usability of both tree automata as well as string automata was already discussed in [CHZ05, CHZ06]. Although the resulting algorithms (and the automata used) were shown to be similar in [CHZ05], the two approaches were not shown as instances of a common algorithm skeleton, and the discussion was restricted to the single pattern case of the \textit{TPM} \( n \) problem (i.e. Tree Pattern Matching with match registration per subject tree node). More details can be found in Literature references 6.7.14 and 6.7.29.

\section*{6.7.1 Using Aho-Corasick automata}

The first kind of automaton for use with Algorithm 6.7.9 (\textsc{s-path}, \textsc{sp-matcher}, \textsc{det}) we consider is an Aho-Corasick automaton (\textsc{aca}) for the pattern stringpaths.

Many algorithms and automata exist for finding all occurrences of keywords from a set of strings \( L \) over alphabet \( \Sigma \) in a subject string \( S \) (see e.g. [CWZ04b, WZ96, Wat95]). In particular, any deterministic finite automaton accepting \( \Sigma^* \cdot L \)—i.e. accepting any sequence of symbols followed by a string from language \( L \)—may be used to detect occurrences of elements of \( L \) in \( S \) at their endpoints, and the Aho-Corasick automaton is such an automaton.

We restrict ourselves to the optimal\(^6\) \textsc{aca}, as it is used in algorithms occurring in the literature. Additionally, as we will see in Section 6.7.2, that automaton is similar in structure to a kind of \textsc{drfTA} that is also usable for stringpath matching based on Algorithm (\textsc{s-path}, \textsc{sp-matcher}, \textsc{det}).

\footnote{\textsc{aca} is a version of the \textsc{aca} without failure transitions [Wat95, AC75].}
6.7 Stringpath-based match set computation

Construction algorithms for (optimal) ACA s can be found in e.g. [AC75, Wat95].

The optimal ACA for keyword set \( L \) is a 5-tuple \( M_{ACA} = (Q, \Sigma, \delta, q_0, output_{ACA}) \) in which \( Q \) is the state set, \( \Sigma \) the alphabet, \( \delta \in Q \times \Sigma \rightarrow Q \) the (partial) transition function, \( q_0 \) the start state, and \( output_{ACA} \in Q \rightarrow \mathcal{P}(L) \) the output function. Given the state reached by the automaton by processing an input string up to a given position, the output function determines the set of keyword occurrences ending at this position.

Hoffmann & O’Donnell [HO82b] were the first to suggest the use of an (optimal) ACA, recognizing the stringpaths of tree patterns, in a TPMn algorithm. For this use, the automaton is constructed from keyword set \( SPaths(P) \) over alphabet \( \Sigma' \cup \mathbb{N}_{\leq r} \) to accept \((\Sigma' \cup \mathbb{N}_{\leq r})^* \cdot SPaths(P)\). This automaton is then used to match against the subject tree’s stringpaths—i.e. \( SPaths(t) \)—by using the automaton in a root-to-frontier traversal to detect occurrences of elements of \( SPaths(P) \) at their endpoints and register them at their beginpoints, as in Algorithm 6.7.9 (S-PATH, SP-MATCHER, DET).

On a high level, the optimal ACA’s construction can be described as follows:

1. Construct a trie [CR03] for the keyword set. Such a trie is a deterministic string automaton whose state set corresponds to \( \text{pref}(L) \), i.e. each state corresponds to a prefix of the keyword set \( L \), and whose transitions are such that the trie accepts every keyword from the set. In our case, the state set corresponds to \( \text{pref}(SPaths(P)) \), i.e. each state corresponds to a prefix of the set of pattern stringpaths and the trie accepts every pattern stringpath.

2. For every state, define its output value equal to the (possibly empty) set of matching keywords.

3. Add a ‘self-loop’ transition on every alphabet symbol—including \( \nu \)—to the start state.

4. Determine the automaton and output function.

**Example 6.7.11 (ACA for keyword set \( SPaths(P) \)).** Figure 6.7.7, ignoring the states \( q_0^a \) and \( q_0^b \), shows the result of the first step of the construction, i.e. shows the trie. Applying the preceding construction to keyword set \( \{a1b1c, a2\nu, a1\nu, a2d, b1d\} \) (i.e. to \( SPaths(P) \)) leads to the ACA seen earlier in Figure 6.7.2. The only back transition shown is that from \( q_4 \) to \( q_{12} \) on \( d \). Back transitions on \( a \) (to state \( q_{1} \)), on \( b \) (to state \( q_{10} \)) and back transitions to state \( q_0 \) (on \( c, d, 1, 2 \)) are not shown, to increase readability.

The output function values are defined as \( output_{ACA}(q_5) = \{a1b1c\}, output_{ACA}(q_{11}) = \{a2\nu\}, output_{ACA}(q_8) = \{a1\nu\}, output_{ACA}(q_9) = \{a2d\}, output_{ACA}(q_{12}) = \{b1d\} \), and \( output_{ACA}(q) = \emptyset \) for all other states \( q \).

The input states \( q_0 \) through \( q_{12} \) are shown in Figure 6.7.7.

To use an optimal ACA with Algorithm 6.7.9 (S-PATH, SP-MATCHER, DET) we have to define \( \gamma \) and \( Output \) in terms of \( \delta \) and \( output_{ACA} \) such that Specification 6.7.7 is
satisfied. To this purpose, we define $\gamma$ and $Output$ (for $q \in Q$, $a \in \Sigma$, $1 \leq i \leq n$) by

$$
\gamma(q, a \cdot i) = \delta(\delta(q, a), i) \quad (\text{i.e. consecutive ACA transitions on } a \text{ and } i) \quad \text{and}
$$

$$
Output(q, a) = output_{ACA}(\delta(q, a)).
$$

**Detail 6.7.12 (ACA-SPM).** Use an (optimal) Aho-Corasick automaton and define transition and output functions in terms of that automaton.\[\]

The resulting Algorithm (S-PATH, SP-MATCHER, DET, ACA-SPM) can be obtained by substituting in Algorithm 6.7.9 (S-PATH, SP-MATCHER, DET) the definitions of $\gamma$ and $Output$ as given above. Thus, the state used for the recursive call of Traverse on a node $n \cdot i$ is determined from the state used for node $n$ by taking an ACA transition on symbol $t(n)$ followed by one on $i$, while the output for $a = \nu$ and $a = t(n)$ is determined by making a transition on the symbol $a$ and determining the ACA’s output value for the resulting state.

It can be shown that with $\gamma$ defined as above, and with $\delta^*$ the extended transition function for the ACA (defined based on transition function $\delta$ in the usual way), the state $q$ used in a call of Traverse for node $n$ is such that $q = \delta^*(q_0, RPath(t, n)|1)$. As a result,

$$
Output(q, t(n)) = output_{ACA}(\delta^*(q_0, RPath(t, n)))
$$

and

$$
Output(q, \nu) = output_{ACA}(\delta^*(q_0, (RPath(t, n)|1) \cdot \nu)),
$$

which yield the set of pattern stringpaths for which a match occurs ending at node $n$ and ending with symbol $t(n)$ and $\nu$ respectively. These are precisely the values of $Output$ required by Specification 6.7.7.

**Example 6.7.13.** For subject tree $t = a(b(c), a(b(c), a(c, c)))$, Figure 6.7.6 shows the states associated with every node and matches detected by Algorithm (S-PATH, SP-MATCHER, DET, ACA-SPM) using the Aho-Corasick automaton of Example 6.7.11 depicted in Figure 6.7.2. Note that even though the ACA used is deterministic, two states may be associated with a tree node $n$: the states corresponding to $\delta(q, t(n))$ and to $\delta(q, \nu)$. To avoid cluttering the figure, the latter states are only shown when indicating a stringpath match (ending in symbol $\nu$). States corresponding to stringpath matches are underlined. The detection of matches as indicated in the figure and subsequent registration leads to the values of OSP and $O$ as in Figures 6.7.4 and 6.7.5.\[\]

**Literature reference 6.7.14.** Root-to-frontier tree pattern matching algorithms based on deterministic string automata frequently occur in the literature, often as the basis for tree acceptance and parsing algorithms. The underlying tree pattern matching algorithm occurs in the early work by Hoffmann & O’Donnell [HO82b], although their presentation of the recursive algorithm uses an explicit stack. They also sketch how the algorithm could be modified to register stringpath and full
pattern matches based on bitvectors associated with nodes and bitstring operations using these bitvectors.

The presentation of tree algorithms based on deterministic string automata in the literature is often quite informal and complicated by optimizations. Van de Meerakker [Mee88] gave a stepwise account of how to obtain the algorithms, starting with the use of Aho-Corasick automata for string matching, extending this sequentially to matching of stringpaths in trees, matching of stringpaths with \( \nu \) symbols, matching with match registration at stringpath beginpoints, and then extending it further to tree acceptance (which we discuss in Section 5.8) and parsing.

In [Mee88], the recursive calls in procedure \textit{Traverse} are made \textit{after} the registration of matches ending with \( t(n) \) and before that of matches ending with \( \nu \), and a local variable is introduced to store and reuse the value of \( \delta(q, t(n)) \). The order of the calls does not affect correctness.

\[\begin{align*}
\text{Figure 6.7.6} & \quad \text{ACA state assignment and stringpath matches} \\
& \text{\hspace{1cm}} \begin{array}{c}
\text{[a1v]} \rightarrow q_2, q_3 \\
\text{[a1b1c]} \rightarrow q_4 \\
\text{[a1b1c]} \rightarrow q_5 \\
\text{[a1b1c]} \rightarrow q_6 \\
\end{array}
\end{align*}\]

\[\begin{align*}
\text{\hspace{1cm}} \begin{array}{c}
q_0 \\
q_1 \\
q_2 \\
q_3 \\
q_4 \\
q_5 \\
q_6 \\
q_7 \\
q_8 \\
q_9 \\
q_{10} \\
q_{12} \\
q_{13} \\
q_{14} \\
q_{15} \\
q_{16} \\
q_{17} \\
q_{18} \\
q_{19} \\
q_{20} \\
\end{array}
\end{align*}\]

6.7.1.1 Using Aho-Corasick stringpath automata

Considering the ACA depicted in Figure 6.7.2, we can make an important observation: as all of the algorithms assume the subject tree to be a well-formed tree, a transition on a symbol \( a \in \Sigma' \) can only be followed by a transition on a symbol \( i \) such that \( 1 \leq i \leq n \). Thus, in the example, transitions on symbols from \( \Sigma' \) originating from states \( q_1, q_3, q_5, q_7, q_8, q_9, q_{10} \) and \( q_{12} \) will never be used, and transitions from state \( q_0 \) on \( c, d \) and \( \nu \) (symbols of rank 0) back to state \( q_0 \) will never be followed by any other transitions.

Looking at the high-level description of the Aho-Corasick automaton construction in Section 6.7.1, it is not hard to see that step (3) causes these superfluous transitions: it adds a ‘self-loop’ transition on every alphabet symbol, i.e. on every symbol in \( \Sigma' \cup \mathbb{N}_{\leq r} \). For an Aho-Corasick stringpath automaton (ACSPA), we may replace step (3) by
(3') For each symbol of rank $> 0$, add a new state, a transition from the start state to this state on the symbol, and, for $i$ between 1 and the rank of the symbol, a transition on $i$ from this state back to the start state.

Example 6.7.15 (ACSPA for keyword set $SPaths(P)$). The trie with loops constructed by steps (1), (2) and (3’) is depicted in Figure 6.7.7.

Output function values are defined as $output_{trie}(q'_{8}) = \{a1b1c\}$, $output_{trie}(q'_{7}) = \{a2\nu\}$, $output_{trie}(q'_{8}) = \{a1\nu\}$, $output_{trie}(q'_{9}) = \{a2d\}$, $output_{trie}(q'_{12}) = \{b1d\}$, and $output_{trie}(q') = \emptyset$ for all other states $q'$.

![Figure 6.7.7 Trie with stringpath loops for pattern set $P$](image)

Applying step (4) of the high-level construction to this automaton leads to the ACSPA depicted in Figure 6.7.8 (in which transitions that are not shown lead to $q_0$). The output function values are defined as $output_{ACSA}(q_0) = \{a1b1c\}$, $output_{ACSA}(q_7) = \{a2\nu\}$, $output_{ACSA}(q_8) = \{a1\nu\}$, $output_{ACSA}(q_9) = \{a2d\}$, $output_{ACSA}(q_{12}) = \{b1d\}$, and $\emptyset$ otherwise. 

![Figure 6.7.8 ACSPA for pattern set $P$](image)
In Section 6.7.2, we will see that this modification makes the resulting ACSPA very similar to the stringpath DRFTA constructed there for use with a variant of Algorithm 6.7.9 (S-PATH, SP-MATCHER, DET).

### 6.7.2 Using deterministic root-to-frontier tree automata

The second kind of automaton for use with Algorithm 6.7.9 (S-PATH, SP-MATCHER, DET) we consider is a deterministic root-to-frontier tree automaton. As discussed in Sections 3.4.3 and 6.3, the use of DRFTAs in tree pattern matching algorithms is problematic, since they are in general less powerful than their nondeterministic counterparts (NRFTAs).

On a high level, the construction considered here works as follows:

1. Construct an NRFTA recognizing the pattern trees.

2. For every state with an outgoing transition on a symbol (including $\nu$) of rank 0—indicating a stringpath match—define the output of the state and symbol to be this stringpath; for other combinations of state and symbol, define the output to be empty.

3. Add a ‘self-loop’ transition to the root accepting state, for every symbol of rank $> 0$.

4. Determinize the automaton and adapt the output function accordingly.

The construction clearly resembles the ACSPA construction of Section 6.7.1. We discuss the construction in detail and then show that the results can be used for root-to-frontier tree pattern matching based on stringpath matching in a variant of Algorithm 6.7.9 (S-PATH, SP-MATCHER, DET).

Steps (1)–(3) result in a TPNm NRFTA while step (4) is applied to obtain a DRFTA. We discuss each of the construction steps in some detail and then show how $\gamma$ and \textit{Output} may be defined in terms of the resulting DRFTA. We then show how this DRFTA can be used for stringpath based tree pattern matching, even though it cannot be used for tree pattern matching directly.

For step (1), given a pattern set, we can construct an NRFTA $M$ accepting the pattern set using the following construction.

**Construction 6.7.16** \(\text{(spm-ta:all-dotted:rf)}\).

**Input** Pattern set $P \subseteq \text{Tr}(\Sigma', r')$

**Output** NRFTA $M = (Q, \Sigma', r', R, Q_{\text{ra}})$ where
\[ Q = DottedTr(P) \]
\[
R_a = \left\{ \text{Set } p, m : p(m) = a \wedge p \in P \wedge m \in D_p : ((p, m), ((p, m \cdot 1), \ldots, (p, m \cdot n))) \right\}
\]
\[
(\text{for all } a \in \Sigma')
\]
\[ R_e = \emptyset \]
\[ Q_{ra} = \langle \text{Set } p : p \in P : (p, \varepsilon) \rangle \]

Note that this construction results in a deterministic root-to-frontier tree automaton in case \(|P| = 1\).

The construction can be compared to the constructions in Section 6.5, as well as to the trie construction forming step (1) of the ACA construction in Section 6.7.1.

- It is based on dotted patterns as states, whereas earlier tree automata constructions are based on subtrees of patterns as states. As we will see, having dotted patterns as states is necessary here since every state needs to have a unique path leading to it in order for a usable output function to be definable. Furthermore, it increases similarity to the trie construction.

- It has transitions on alphabet \(\Sigma',\) i.e. \(\Sigma\) extended with \(\nu\). Earlier tree automata constructions have transitions on alphabet \(\Sigma\) and loops on every symbol from \(\Sigma\) at nodes corresponding to the variable symbol \(\nu\). The difference can be explained by the fact that earlier automata were used for matching instances of patterns directly, whereas the construction above is an acceptor for patterns and will form the basis for an automaton based on stringpath matching, including matching of stringpaths ending in \(\nu\) occurrences, not instances of such stringpaths.

- It has multiple root accepting states such that the automaton consists of one disjoint part for each pattern, whereas the trie construction has a single starting state from which stringpaths of all the patterns start. The construction could be modified to already determinize the original NRFTA resulting from step (1), replacing the root accepting states by a single root accepting state and identifying states reachable on the same transitions. As the automata from the construction will be determinized in the end anyway (as step (4) of the high-level procedure given above), we do not consider this modification here. (The NRFTA resulting from step (1) is comparable to constructing a separate trie for each pattern’s set of stringpaths.)

**Example 6.7.17.** Applying Construction (SPM-TA:ALL-DOTTED:RF) to pattern set \(P\) of Example 6.1.2 leads to the (\(\varepsilon\)-free) NRFTA in Figure 6.7.9.

The automaton resulting from Construction 6.7.16 (SPM-TA:ALL-DOTTED:RF) may be used as an acceptor in a root-to-frontier traversal of and state assignment to a
subject tree \( t \). The following theorem relates assignments of states to nodes (in a root-to-frontier traversal) on the one hand and rootpath matches on the other hand.

**Theorem 6.7.18.** Given subject \( t \in Tr(\Sigma, r) \), a pattern set \( P \subseteq Tr(\Sigma', r') \), pattern \( p \in P \), nodes \( n \in D_t \) and \( m \in D_p \), and the NRFTA for \( P \) as in Construction 6.7.16,

\[
(p, m) \in St(t)(n) \\
\land \ (t(n) = p(m) \lor \nu = p(m)) \Rightarrow RPMatch(RPath(p, m), t, n(\rvert m\lvert)).
\]

*Proof idea.* We may prove this theorem by structural induction on \( m \). A brief non-derivational proof for the single-pattern case can be found in [CHZ05, CHZ06].

Recall from Section 3.1.1.1 that stringpaths are rootpaths ending in symbols of rank 0. For cases where \( r'(p(m)) = 0 \), the consequent in the above theorem therefore is equivalent to \( SPMatch(RPath(p, m), t, n(\rvert m\lvert)) \).

This brings us to step (2) of the high-level construction. Based on the observation above, we define an output function to return such a stringpath match: Given a state assigned to a subject tree node, as well as a symbol (the symbol at the node or \( \nu \)), the function will return the corresponding stringpath match, if any.

**Definition 6.7.19.** For tree automata as in Construction 6.7.16, partial function \( output_{NRFTA} \in Q \times \Sigma' \rightarrow SPaths(P) \) is defined for \( (p, m) \in Q \) and \( p(m) \in \Sigma' \) such that \( r'(p(m)) = 0 \) by

\[
output_{NRFTA}((p, m), p(m)) = RPath(p, m).
\]

**Example 6.7.20.** For the pattern set of Example 6.1.2 and the automaton constructed for it using Construction 6.7.16—as in Example 6.7.17—the output function...
values are
\[
\begin{align*}
output_{\text{nrfta}}((p_1, 1 \cdot 1), c) &= a_1b_1c, \\
output_{\text{nrfta}}((p_1, 2), \nu) &= a_2\nu, \\
output_{\text{nrfta}}((p_2, 1), \nu) &= a_1\nu, \\
output_{\text{nrfta}}((p_2, 2), d) &= a_2d, \text{ and} \\
output_{\text{nrfta}}((p_3, 1), d) &= bld.
\end{align*}
\]

We now come to step (3) of the construction. The inverse of the implication in Theorem 6.7.18 does not hold; the automaton is a tree acceptor for the patterns, and can be used to detect a pattern occurrence that starts at the subject tree root. As a result, detectable stringpath matches also start at the subject tree root. To enable pattern and stringpath matches starting at other input tree nodes to be detected, an extension similar to the addition of the ‘self-loop’ transitions of the aca is necessary, leading to the following detail and construction:

**Detail 6.7.21** (ra-loops). Use a root-to-frontier directed tree automaton with a single root accepting state with ‘self loops’.

**Construction 6.7.22** (spm-ta:all-dotted:rf:ra-loops).

**Input** Pattern set \( P \subseteq \text{Tr}(\Sigma', r') \)

**Output** \( \text{nrfta} \ M = (Q, \Sigma', r', R, Q_{ra}) \) where

\[
\begin{align*}
Q &= Dotted\text{Tr}(P) \\
R_a &= \left\langle \text{Set } p, m : p(m) = a : \left( (p, m), ((p, m \cdot 1), \ldots, (p, m \cdot n)) \right) \right\rangle \\
&\cup \left\{ \emptyset \right\} \quad \text{if } a \in \Sigma_0' \\
&\cup \left\{ \text{Set } p : p \in P : ((p, \varepsilon), ((p, \varepsilon)^n)) \right\} \quad \text{if } a \in \Sigma' \setminus \Sigma_0' \\
&\left( \text{for all } a \in \Sigma' \right) \\
R_e &= \emptyset \\
Q_{ra} &= \left\langle \text{Set } p : p \in P : (p, \varepsilon) \right\rangle \quad \square
\end{align*}
\]

The resulting \( \text{nrfta} \) accepts all trees ending in pattern occurrences (not instances).

**Example 6.7.23.** The \( \text{nrfta} \) with ‘self-loops’ constructed for pattern set \( P \) by Construction 6.7.22—corresponding to steps 1–3 of the high-level construction at the beginning of the section—is depicted in Figure 6.7.10. Output function values remain as in Example 6.7.20.

For automata resulting from Construction 6.7.22, we give the following theorem, in which the implication of Theorem 6.7.18 has been replaced by an equivalence:

**Theorem 6.7.24.** Given subject \( t \in \text{Tr}(\Sigma, r) \), a pattern set \( P \subseteq \text{Tr}(\Sigma', r') \), pattern \( p \in P \), nodes \( n \in D_t \) and \( m \in D_p \), and the \( \text{nrfta} \) for \( P \) as in Construction 6.7.22,

\[
(p, m) \in \text{St}(t)(n) \quad \land \quad (t(n) = p(m) \lor \nu = p(m)) \equiv R \text{PMatch}(R \text{Path}(p, m), t, n, |m|).
\]
Proof idea. As with Theorem 6.7.18, the proof is by structural induction on $m$. The proof in the $\Rightarrow$ direction is along the lines of the proof of the earlier theorem. As for that theorem, a brief non-derivation proof for the single-pattern case of this theorem can be found in [CHZ05, CHZ06].

Similarly to the determination of the trie with ‘self-loops’ to obtain a deterministic ACA, the NRFTA resulting from steps (1)–(3)—i.e. the NRFTA resulting from Construction 6.7.22 (SPM-TA:ALL-DOTTED:RF:RA-LOOPS)—can be determined, as per step (4) of the high-level construction.

The subset construction for NRFTAs, $\text{SUBSET}_{\text{RF}}$, was discussed in Section 3.4.3. It was shown there that the resulting DRFTA in general accepts a superset of the NRFTA’s language.

However, Lemma 3.4.35 stated that the set of trees recognized by the DRFTA is the set of trees of which every stringpath occurs as a stringpath in a tree from the NRFTA’s language, provided the NRFTA has no root accepting reachable states that are useless. The NRFTAs resulting from Construction 6.7.22 satisfy this restriction. As we aim at using the resulting DRFTA for stringpath matching, $\text{SUBSET}_{\text{RF}}$ can therefore be used in this case. We refer to the resulting construction as Construction (SPM-TA:ALL-DOTTED:RF:RA-LOOPS:SUBSET$_{\text{RF}}$).

Example 6.7.25. Applying the subset construction to the NRFTA of Example 6.7.23 leads to the stringpath DRFTA in Figure 6.7.11. Output function values for this
Figure 6.7.11 Stringpath DRFTA for $P$ resulting from Construction (SPM-TA: ALL-DOTTED:RF:RA-LOOPS:SUBSET$_{RF}$).

DRFTA are as follows:

\[
\begin{align*}
\text{output}_{\text{DRFTA}}(q_4, c) &= \{a1b1c\}, \\
\text{output}_{\text{DRFTA}}(q_4, d) &= \{b1d\}, \\
\text{output}_{\text{DRFTA}}(q_5, \nu) &= \{a2\nu\}, \\
\text{output}_{\text{DRFTA}}(q_6, \nu) &= \{a1\nu\}, \\
\text{output}_{\text{DRFTA}}(q_6, d) &= \{a2d\}, \text{ and} \\
\text{output}_{\text{DRFTA}}(q_{11}, d) &= \{b1d\}.
\end{align*}
\]

Note the structural similarity to the NRFTAs of Examples 6.7.17 and 6.7.23.

Using Theorem 6.7.24, NRFTAs resulting from Construction 6.7.22 not having any useless states, and using the subset construction, we obtain:

**Corollary 6.7.26.** Given subject $t \in Tr(\Sigma, r)$, a pattern set $P \subseteq Tr(\Sigma', r')$, pattern $p \in P$, nodes $n \in D_t$ and $m \in D_p$, and the DRFTA for $P$ as obtained using Construction (SPM-TA: ALL-DOTTED:RF:RA-LOOPS:SUBSET$_{RF}$), and letting $q$ be the unique state such that $\{q\} = St(t)(n)$,

\[
(p, m) \in q \land (t(n) = p(m) \lor \nu = p(m)) \equiv RPath(RPath(p, m), t, n, |m|).
\]

In other words, the state assigned to a node and the symbol at that node and $\nu$ together determine the set of rootpath matches and therefore—as indicated following Theorem 6.7.18—the set of stringpath matches ending at that node. As indicated before, the DRFTA and associated output function can thus be used in a root-to-frontier subject tree traversal to detect all stringpath matches.
To use a DRFTA as above with Algorithm 6.7.9 (s-PATH, SP-MATCHER, DET) we have to define $\gamma$ and $Output$ in terms of DRFTA transition functions $R_a$ and output function $output_{DRFTA}$ such that Specification 6.7.7 is satisfied. To this purpose, we define $\gamma$ and $Output$ (for $q \in Q$, $a \in \Sigma'$, $1 \leq i \leq n$) by

$$\gamma(q, a \cdot i) = \pi_i(R_a(q))$$
$$Output(q, a) = output_{DRFTA}(q, a),$$

using the tuple projection operator $\pi_i$ defined at the end of Section 2.2.

**Detail 6.7.27 (DRFTA-SPM).** Use a DRFTA as a stringpath matcher and define transition and output functions in terms of that automaton.

The resulting Algorithm (s-PATH, SP-MATCHER, DET, DRFTA-SPM) can be obtained by substituting in Algorithm 6.7.9 (s-PATH, SP-MATCHER, DET) the definitions of $\gamma$ and $Output$ as given above. Thus, the state used for the recursive call of $Traverse$ on a node $n \cdot i$ is determined from the state used for node $n$ by taking a DRFTA transition on symbol $t(n)$ and projecting to element $i$ of the resulting tuple, while the output for $a = \nu$ and $a = t(n)$ is determined by the DRFTA’s output value for the state used for node $n$ and the symbol $a$.

It can be shown that with $\gamma$ defined as above, the state $q$ used in a call of $Traverse$ for node $n$ is such that $q = \gamma^*(q_0, RPath(t, n)|1)$. As a result,

$$Output(q, t(n)) = output_{DRFTA}(\gamma^*(q_0, RPath(t, n)|1), t(n))$$

and

$$Output(q, \nu) = output_{DRFTA}(\gamma^*(q_0, RPath(t, n)|1), \nu),$$

which yield the set of pattern stringpaths for which a match occurs ending at node $n$ and ending with symbol $t(n)$ and $\nu$ respectively. These are precisely the values of $Output$ required by Specification 6.7.7.

**Figure 6.7.12** DRFTA state assignment and stringpath matches
Example 6.7.28. Figure 6.7.12 shows the states associated with every node and matches detected by Algorithm (S-PATH, SP-MATCHER, DET, DRFTA-SPM) using the stringpath DRFTA of Example 6.7.25 for subject tree $a(b(c), a(b(c), a(c, c)))$. Combinations of states and symbols corresponding to stringpath matches are underlined. Note that symbol $\nu$ is only explicitly depicted for nodes at which it occurs in a stringpath match. The detection of matches as indicated in the figure and subsequent registration leads to the same values of OSP and $O$ as in Figures 6.7.4 and 6.7.5.

Literature reference 6.7.29. Root-to-frontier tree pattern matching algorithms based on DRFTAs for a single tree pattern were described in [CHZ05] and its extended journal version [CHZ06]. Those two publications formed the basis for the current section, in which the material was revised, generalized to deal with pattern sets, extended w.r.t. discussion of the automata constructions, and further formalized. In those publications, no common ancestor Algorithm (S-PATH, SP-MATCHER, DET) was described for the DRFTA-based and Aho-Corasick-based algorithm.

In a book chapter by Li and Wood [LW96], DRFTAs with output are briefly discussed. Such automata are defined directly in terms of a generalized Aho-Corasick transition function, i.e. not in a way similar to the tree automata constructions used throughout this dissertation, and correctness arguments and a comparison to the (use of the) Aho-Corasick automata for stringpath matching of Hoffmann & O’Donnell [HO82b] are omitted, even though the latter authors’ paper is cited. Furthermore, the type of patterns used by Li and Wood is slightly different (not allowing leaf nodes to be labeled by symbols other than $\nu$).

6.7.2.1 Comparing the stringpath automata

In Sections 6.7.1.1 and 6.7.2 we discussed the construction of ACSPAs and the construction of DRFTAs for stringpath matching. The constructions’ four step high-level description and the examples given showed a large degree of similarity, and it was shown that by properly defining $\gamma$ and Output both could be used in variants of a common ancestor Algorithm (S-PATH, SP-MATCHER, DET). We further conjecture that—for the single pattern case, or with some changes to either the trie or the NRFTA construction in the respective step (1)—automata resulting from one kind of construction can be transformed into those of the other kind and vice versa. Intuitively, this becomes clear when comparing the visual representations of the two kinds of automata: by replacing ACSPA states with incoming transitions on symbols from $\Sigma$ and outgoing ones on symbols from $\mathbb{N}_{\leq r}$ by the small circles used in DRFTAs, the ACSPA and corresponding DRFTA become highly similar.

We briefly sketch our arguments supporting the conjecture for the single pattern case in a stepwise fashion, following the high-level constructions. This simplifies the discussion and gives insight in the relation between the intermediate automata.

Step (1) is the construction of the trie respectively NRFTA accepting the pattern
trees. Examples of such automata—albeit for the multiple pattern case—can be seen in Figure 6.7.7 (ignoring loops on start state \(q'_0\)) and Figure 6.7.9 respectively. In general, the automata parts corresponding to a tree node labeled by a symbol \(a\) are as shown in Figure 6.7.13. It is not hard to see that \(\delta(p,a) = q\) and 

\[
\langle \forall i : 1 \leq i \leq r'(a) : \delta(q,i) = q_i \rangle
\]

for the trie if and only if \((q_1, \ldots, q_{r'(a)}) \in R_a(p)\) for the NRFTA, and that states \(p, q_1, \ldots, q_n\) in the trie and NRFTA can be pairwise identified.

**Figure 6.7.13** Trie and NRFTA parts corresponding to a symbol \(a\).  

Step (2) concerns the definition of the automata’s output functions. For the trie of Figure 6.7.7 the functions’ values can be found in Example 6.7.15, while Example 6.7.20 gives them for the NRFTA of Figure 6.7.9. Recall that stringpath matches end in a symbol \(a \in \Sigma_0'\). To compare the output functions, we therefore only need to consider automata parts as in Figure 6.7.14, where states \(p\) in the respective automata are identified as above. It should be clear that \(\text{output}_{\text{trie}}(\delta(p,a)) = \{sp\}\) in the trie if and only if \(\text{output}_{\text{NRFTA}}(p,a) = sp\) in the NRFTA.

**Figure 6.7.14** Trie and NRFTA parts with output. For the trie, \(\text{output}_{\text{trie}}(q) = \{sp\}\), while for the NRFTA, \(\text{output}_{\text{NRFTA}}(p,a) = sp\), for \(sp\) the stringpath recognized.

Step (3’) in the ACSPA construction and (3) in the stringpath DRFTA construction construct the ‘self-loops’ on the trie’s and NRFTA’s start state. Examples of these loops can be seen in Figure 6.7.7 and Figure 6.7.10. In general, they take the shape depicted in Figure 6.7.15 and the identification as for step (1) applies (note that the NRFTA with loops may have multiple root accepting states, but that these may be merged into one, and are merged during the later determinization step).

Step (4) of the constructions determinizes the automata. Examples of the resulting automata can be found in Figures 6.7.8 and 6.7.11. They exhibit quite a similar structure, but apparently differ in one aspect. Firstly, the stringpath DRFTA shows five transitions on symbol \(a\) to the state tuple \((q_2, q_6)\). Following the arguments given for step (1), the ACSPA should therefore have five states with an incoming transition.
on $a$ and outgoing transitions on 1 to $q_2$ and on 2 to $q_6$. Such states would clearly be equivalent though, and could therefore be merged, doing away with the apparent difference. The same argument holds for the four transitions on symbol $b$ to the single-state tuple $(q_{11})$.

### 6.8 Conclusions

A number of conclusions can be drawn and remarks can be made about the taxonomy of algorithms presented in this chapter:

- As with earlier taxonomies—including the one in Chapter 5—the taxonomy presents related algorithms (in this case tree pattern matching algorithms) in a common framework, highlighting their commonalities and differences, and factoring out common parts, leading to common ancestors in the taxonomy graph. The taxonomy graph again serves as a high-level table of contents to the algorithms.

- In fact, this taxonomy and the one for tree acceptance in Chapter 5 are closely related, due to the close relationships between algorithms solving the respective problems, which in turn are due to the close relationship between the two problems. As a result, a number of remarks made about the preceding taxonomy hold for the current one more or less unchanged:
  
  - Further factoring and separation of concerns was achieved by separating the construction of tree automata (Section 6.5) from that of the pattern matching algorithms in which they are used (Sections 6.2 through 6.4).
  
  - For the constructions in Section 6.5, basic details similar to or the same as those used in Chapter 5 were used. These details again allow for easy identification and comparison of constructions, by describing them using compositions of such details. In the case of pattern matching, states in the constructions contain information relating them to pattern set elements or parts thereof, making it easier to reason about the constructions’ correctness and to relate them to one another.
6.8 Conclusions

- In describing the automata constructions, we again described the undirected version first and added details to arrive at other constructions. As in Chapter 5, this allowed a stepwise introduction of more complex constructions.

- For frontier-to-root acceptance algorithms, both the (DFRTA) automata computation view and the match set computation view were considered, in Section 6.5.3.1 and 6.6 respectively.

- In Section 6.6, the details of match set tabulation (in Section 6.6.1) were again separated from the algorithms using match set computation. The same was done for the use of filtering techniques on the one hand (Section 6.6.2) and filter precomputation and tabulation on the other hand (Section 6.6.3).

- As for filtering of DFRTA transition tables/match set tables, only Chase's filter function was known from the literature [Cha87, HK89, FSW94]. We additionally considered the same three additional filter functions we discussed in the preceding chapter, none of which seem to have appeared in the literature on tree pattern matching.

- We also presented algorithms based on stringpath matching. These are based on a root-to-frontier tree traversal and the use of either a deterministic string automaton (exemplified here by an Aho-Corasic automaton) or a certain kind of deterministic root-to-frontier tree automaton (DRFTA). The latter automaton's construction was discussed in detail and in a stepwise fashion, compared to the Aho-Corasic automaton's construction. For this comparison, a modification of the high-level Aho-Corasic construction was applied that did not appear in the earlier work by Hoffmann & O'Donnell [HO82b] or others [Mee88].

- In contrast to the work on tree acceptor constructions—which, with the exception of the work by Brainerd in the late 1960s, mostly appeared in the late 1980s and in the 1990s, see Chapter 5—tree pattern matcher ones were already considered earlier, in Kron's work [Kro75] and the highly influential work of Hoffmann & O'Donnell [HO82b]. The latter already contained many of the essential ideas and algorithms in branches (T-ACCEPTOR, FR), (MATCH-SET), and (S-PATH, SP-MATCHER) of the taxonomy. A number of publications appeared from the mid 1980s to the mid 1990s, improving on earlier work [Cha87, Mee88, CPT92, FSW94].

- Remarkably, despite the close relation between the two problems and many of their solutions, few publications deal with both tree pattern matching and tree acceptance. The early work by Kron and Hoffmann & O'Donnell does not discuss acceptance at all, while Brainerd's discussion of a tree acceptor construction in the late 1960s does not consider pattern matching either [Bra67, Bra69]. It is only in the more recent work of Ferdinand, Seidl & Wilhelm [FSW94] that the similarities are fully exploited for the frontier-to-root case, although they
are apparent and pointed out in earlier work [AG85, AGT89, WW89, Mee88, Cha87], particularly for stringpath-based algorithms.

- Finally, as we indicated in the introduction to the chapter, a lot of theoretically interesting algorithms for tree pattern matching exist that were not treated here. These algorithms for example restrict the problem to a single pattern or use elaborate encodings of the subject tree. For some of them, experiments in [Bau96] show their practical use to be limited. Regardless, formally describing them in the framework of the taxonomy and implementing and testing them could be considered to see whether this holds true for all of them.
Part III

Toolkits
Chapter 7

TABASCO

In this chapter, we discuss TABASCO, a method for domain modeling and domain engineering.

The contents of the chapter are partially based on joint work published earlier. Most of this work was done with Bruce Watson, with extensive feedback and suggestions by Derrick Kourie and Iwan Vosloo. Sergei Obiedkov and Andrew Boake provided further input. An outline of TABASCO—co-authored by Vosloo and Watson—appears as [CVW05]. An overview of the method based on a subset of an earlier version of this chapter appeared as [CWKB05]—with Watson, Kourie, and Boake as co-authors—and additionally co-authored by Obiedkov—as [CWXK06].

We first give brief overviews of domain modeling and domain engineering in general in Section 7.1 and of generative programming in Section 7.2. Although the domains used in these contexts are usually much larger and less restricted than the ones we consider, concepts and ideas from both are used in the TABASCO method, which is presented in Section 7.3.

TABASCO—for TAxonomy BAsed Software COstruction—is a domain modeling and domain engineering method which is aimed at algorithmic domains and in which generative programming techniques may be used. One of the most important steps of the method—taxonomy construction as a domain modeling approach—was discussed in Chapter 4. We discuss other important steps—toolkit and domain specific language design and implementation as a domain engineering approach—in detail in Sections 7.4 through 7.5, giving examples based on actual applications of the method. The evolution of taxonomies and software in the TABASCO context is briefly discussed in Section 7.6.

In Section 7.7, we discuss related work. This includes a discussion of similar domain modeling and domain engineering methods for algorithmic domains, of earlier work that influenced TABASCO, and of applications of the method.
Finally, some concluding remarks are given in Section 7.8.

Apart from the discussion and examples of TABASCO in the current chapter, this dissertation focuses on TABASCO’s application to the algorithmic problems of tree acceptance and tree pattern matching. This resulted in the domain models in the form of taxonomies described in Chapters 5 and 6, and in the domain implementation in the form of a toolkit described in Chapter 8.

7.1 Domain engineering

We consider domain engineering such as discussed by Czarnecki and Eisenecker [CE00] as well as in the Software Engineering Institute’s Model-Based Software Engineering approach [SoF97]. Most current software engineering methods focus on the development of single software systems, i.e. they are application engineering methods. As a result, such methods are not well-suited for the development of reusable software in the form of generic systems or software components. In contrast to this, domain engineering emphasizes software reuse, since it focuses on the development of reusable software, i.e. generic software from which different concrete systems can be instantiated or parts of which can be reused in different systems. According to [CE00],

> Domain Engineering is the activity of collecting, organizing and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means for reusing these assets when building new systems.

The idea behind this approach is that it is helpful to capture domain knowledge in the form of a domain model and reusable assets (including software components), and to have a means for actually reusing those, in order to reduce production cost (in terms of time and money) and increase quality.

Domain engineering is a three-step process consisting of domain analysis, domain design and domain implementation. The results—in the form of reusable assets and ways of reusing them—are used as part of traditional application engineering. Figure 7.1.1 depicts the domain engineering and application engineering processes and their interactions.

The domain analysis step consists of selecting and defining a specific domain (domain scoping), collecting all relevant information from this domain and building a model based on this information (domain modeling). Such a domain model is created based on an analysis of the common and variable properties of all systems in a domain. A complete domain model may consist of a domain definition (defining scope and contents of the domain), domain dictionary (defining important concepts within the domain), concept models (capturing concepts from the domain in some
modeling formalism and/or textual form) and feature models (describing features in the domain as well as the ways in which to combine them). Domain modeling forms an important part of TABASCO. We discussed domain modeling as part of the method in Chapter 4, which included a comparison of our taxonomy construction approach to the feature modeling approach used in [CE00].

The main purpose of domain design is the development of a system family architecture based on the domain model resulting from the preceding step. In addition, a production plan should be developed, describing how concrete systems can be built using the components that will implement the family architecture. This production process can be manual, supported by tools, or fully automatic (as in the generative programming approach, discussed in the next section). It often involves a so-called Domain Specific Language (DSL) which allows domain experts to specify their product requirements using domain concepts they are familiar with. An expression in such a DSL is then mapped to an actual software product. The architecture of a system family should be flexible, in the sense that not even the architectural skeleton is fixed, but can be adapted by using different components. The development of a systems family architecture based on a domain model—like almost any software architecture development—is still very much an activity based on experience and creativity: although there exist many methods and strategies for this task, none of them gives a precise set of rules according to which one can arrive at such an architecture more or less automatically. In Section 7.4, we will see what guidelines
and methods may be used to get from domain model to architecture as part of the TABASCO process, in which domain design is applied on a smaller scale.

Following the design of an architecture, the domain implementation takes place. This step involves the implementation of the components, DSLs and generators, as well as the production plan. As this step is dependent on the implementation techniques and languages chosen, we will not discuss it in detail. Some remarks on domain implementation as part of TABASCO will be made in Section 7.5, where DSL construction is considered in detail.

7.2 Generative programming

Generative Programming [CE00] is an approach to software development that seeks to automate the creation of products. For this, the domain engineering method that is used needs to satisfy certain properties.

The domain models created need to be generative: as part of the domain analysis, configuration knowledge needs to be captured. Such knowledge specifies the validity of e.g. feature combinations, defaults, construction rules, and optimizations.

In addition to the modeling of product families, the domain design step needs to produce a formal specification of the configuration knowledge, i.e. the mapping from user requirements (i.e. product specifications, usually in the form of a DSL expression) to implementation components. Having the configuration knowledge captured in a formal way allows for the automatic generation of software components based on user requirements.

Furthermore, implementation of the configuration knowledge by generators—the programs actually generating components or systems based on a user requirements specification—becomes an additional part of the domain implementation step.

7.3 TABASCO

TABASCO is a domain modeling and domain engineering method that is intended for algorithmic domains and in which generative programming techniques may be used. It has a strong emphasis on domain modeling and uses formal domain models in the form of taxonomies.

In the literature, domain engineering is often presented as applicable to business software domains, in which families of related large and complex software systems are developed. It seems to us that the application of domain engineering to such fields entails a high risk, related to the complexity of the systems involved as well as the involvement of third-party developers in the component development process. Such domains may therefore not be ready for the development of product families using a domain engineering method.
Applying a formal domain engineering approach on a smaller scale, that is to more restricted domains, seems useful. TABASCO restricts a domain to be a family of related algorithms or data structures\(^1\), all solving a single problem or one of a few similar problems, instead of a family of software systems, i.e., the method restricts domain complexity. It also assumes that no third-party involvement occurs, apart from the use of the resulting taxonomy, toolkit and DSL by third parties.

The domain to which the method is applied needs to satisfy a number of conditions:

**Richness** The domain must contain enough published algorithms solving a single or a few related problem(s).

**Maturity** A mature theory underlying the domain must be available to reason about the algorithmic problems and their solutions.

**Applicability** The algorithms from the domain must have broad (potential) applicability in practical software systems.

Unfortunately, a number of deficiencies may exist for the domain considered:

1. Inaccessibility of the theory and algorithms, which may be scattered over the literature, and for which few or no overview publications—particularly algorithm oriented ones—may exist.

2. Difficulty of comparing the algorithms due to differences in presentation style and level of formality.


4. Lack of a large and coherent collection of implementations of the algorithms.

5. Difficulty of choosing between different algorithms for practical applications.

TABASCO aims to solve the first three deficiencies by creating a domain model in the form of a formal taxonomy of algorithms, and the last two by creating a toolkit and DSL based on the domain model. The TABASCO process consists of a number of steps:

1. Selection of an algorithmic domain satisfying the conditions mentioned.

2. Literature survey, to gather the theory and create an overview of it, and to find existing algorithms solving the problem(s). During and after this literature search, one should keep an eye open for the commonalities and variations between the various algorithms.

\(^1\) Although TABASCO has been applied to families of related data structures, we focus on its application to families of related algorithms, since the applications of the method discussed in this dissertation are of that type.
3. Taxonomy construction, discussed in detail in Chapter 4.

4. Toolkit design, discussed in detail in Section 7.4.

5. DSL design, discussed in detail in Section 7.5.

6. Toolkit implementation. As mentioned in Section 7.1, this step depends on implementation techniques and languages chosen, and therefore is only briefly touched upon, in Sections 7.4 and 7.5.

7. Benchmarking, considered with DSLs in Section 7.5.

8. DSL implementation, discussed in detail in Section 7.5 as well.

Figure 7.3.1 depicts TABASCO as a domain engineering method. As mentioned, the concept of a domain in TABASCO is a quite restricted version of the domain concept in general domain engineering. As a result, the amount of custom design and development in building an application using a TABASCO toolkit and/or DSL will usually be substantial.
7.4 Toolkit design

When we discussed the domain design step in Section 7.1, we noted that the development of a system family architecture or design is not driven by rules but much more by experience and creativity. Domain design is not a straightforward task in general, but here we are helped by the restricted form of our domains and domain models: our domains consist of algorithms or data structures solving the same or a few closely related problems, and our domain model consists of a taxonomy which formally relates the algorithms or data structures to one another, grouping them according to their commonalities. The restriction of domains naturally results in restricted complexity and high levels of uniformity in the domain models. In turn, this makes the domain design step less complex.

We do not claim that by using the TABASCO method, the design of an algorithm or data structure toolkit directly follows from the domain model. However, although some choices still have to be made whose outcome may depend on the designer’s experience and/or creativity, the use of the TABASCO method does make the task of domain design more straightforward. The taxonomy makes the commonalities and variations of the algorithms explicit, thereby showing a grouping of the algorithms as well as the ways in which algorithms in such a group differ. As a result, the high-level design choices are guided by the taxonomy structure, and the choice of which language constructs to use to implement various design parts can be made based on standard design techniques, such as multi-paradigm design—described by Coplien [Cop00, Cop98]—and design patterns—such as those described in the famous ‘Gang of Four’ book [GHJV95].

To show how this works in practice, we will discuss the design of the SPARE Time toolkit as an example of domain design according to TABASCO. The example uses C++ as the toolkit implementation language, and the work by Coplien focuses on the use of C++. Depending on similarities or differences in algorithm structure, data structures and state, Coplien’s work suggests particular C++ constructs for implementation. In case another implementation language, is used, this mapping would have to be adapted to the particular constructs of such a language, but would likely be quite similar, assuming the language to have many of the same constructs as C++. For Java, for example, the mapping is quite similar, as will become apparent in Chapter 8; the toolkit for the tree algorithm taxonomies of Chapters 5 and 6 that is discussed there was implemented in that language.

According to [GHJV95, page 26],

A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. Toolkits don’t impose a particular design on your application; they just provide functionality that can help your application do its job. They are the object-oriented equivalent of subroutine libraries.
We will use the terms *class library*, *library* and *toolkit* interchangeably. Important aspects and design goals of an algorithm toolkit include the following:

- Toolkits should provide a client interface that is easy to understand. Clients should only have to do a minimum of reading to use the library.

- Toolkits should enable practical performance comparisons of the various algorithms included.

- Toolkits should enable reuse in user applications in different settings, i.e. with respect to memory usage, CPU usage and performance, and should thus provide a large collection of algorithms implementations. In other words, they should be general purpose.

- Classes in a toolkit should have a coherent design. They should be designed and coded in the same style, and have a clear relationship and logical class hierarchy.

- The efficiency of algorithm implementations in the toolkit should be comparable to that of hand-coded special-purpose routines—the toolkit must be applicable to production-quality software.

- The design and implementation should be clear and understandable, both to allow clients to easily modify or extend the toolkit and for educational purposes.

### 7.4.1 The design of SPARE Time

As an example, let us consider the design of SPARE Time, a pattern matching toolkit based on the taxonomy represented in Figure 4.1.1. This example is a revised and expanded version of one given in an earlier paper [CW05].

Looking at Figure 4.1.1, one can see taxonomy parts corresponding to the Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick and Comentz-Walter keyword pattern matching algorithms. The algorithms in the taxonomy solve two similar problems, namely multiple and single keyword pattern matching, but resulting in slightly different interfaces: the first will have a match function taking a set of keywords, the match function of the second will take a single keyword. The toolkit therefore contains an abstract class *PM* and two classes *PMSingle* and *PMMultiple* for single and multiple keyword pattern matchers, each with their own function *match*.

For single-keyword pattern matching, the toolkit includes three approaches: brute-force/naïve pattern matching, Knuth-Morris-Pratt pattern matching, and Boyer-Moore pattern matching. Since all these have related operations, but differ in algorithm as well as some data structure and state, the suggested C++ construct is the use of inheritance with virtual functions [Cop98, p. 151]. Thus, the *match* function in class *PMSingle* is a virtual one, and classes *PMBFSingle*, *PMKMP* and *PMBM*
inherit from that class. The use of inheritance with virtual functions in such a way corresponds to the Strategy design pattern [GHJV95, p. 315]. Similarly, for multiple-keyword pattern matching, the approaches are brute-force/naive, Aho-Corasick and Commentz-Walter style pattern matching, resulting in classes PMBFMulti, PMAC and PMCW inheriting from PMMultiple, in which the match function is virtual.

As we saw in Section 4.2, our taxonomy contains three variants of the Aho-Corasick algorithm: the Failure function Aho-Corasick algorithm [AC75, Wat95], the Goto function (Optimal) Aho-Corasick algorithm, and a multiple keyword generalization of the Knuth-Morris-Pratt algorithm [Wat95]. We showed that these algorithms share the same algorithm skeleton yet update state q differently, i.e. use different types of automata. Since the general algorithm structure and behavior are common to all three, Coplien [Cop98, p. 149] suggests the use of templates. Class PMAC therefore is a template class accepting a parameter indicating the particular type of automaton to use in the function match. Although not precisely corresponding to the description of the pattern, this part of the design can also be seen as an instance of the Template Method design pattern [GHJV95, p. 325].

Since there is only a single variant of the single keyword Knuth-Morris-Pratt algorithm in our taxonomy, the corresponding implementation is a straightforward one: class PMKMP is a class of PMSingle, implementing the match function.

Figure 4.1.1 also shows that many Boyer-Moore algorithm variants are part of the taxonomy. Algorithm details (MO) (Match Order), (SL) (Skip Loop) and (MI) (Match Info reuse) allow for three, four and three$^2$ choices respectively. Since the choice for each of the details MO, SL and MI can be made independent of the choice for the other two details, this results in a total of $3 \times 4 \times 3 = 36$ potential algorithm variants. An implementation with 36 child classes of class PMBM is certainly not a good solution. The commonality in algorithm structure but differences in fine structure once more suggest the use of templates. In SPARE Time, we have templatized the class by match order, skip loop and match info reuse. The match function uses the three template arguments to implement a particular shift function. Similarly to class PMAC, we can also see this as an example of the Template Method design pattern.

Finally, the taxonomy contains a large number of Commentz-Walter-like algorithm variants. These are shown in the part of Figure 4.1.1 from the edge labeled (SSD) downward. As was shown in [Cle03, Chapter 3], these differ not only in the particular shift functions used, but also in the type of automaton used (i.e. (RSA) (Reverse Suffix Automaton), (RFA) (Reverse Factor Automaton) or (RFO) (Reverse Factor Oracle), determined by the choice for detail (EGC) (Efficient Guard Computation), which allows for different forms of efficient guard computation (and the corresponding choice for detail (GS) (Guard Strengthening))). As before, algorithm structure and behavior are the same for all variants, but variants differ in the details. Class PMCW is therefore templatized by automaton and shift function.

$^2$Choices for the latter detail are not shown in the figure.
The design (depicted in Figure 7.4.1) uses merely two different match functions for the complete toolkit, resulting in a clean and easy to understand interface.

As an example of algorithm instantiation, class \texttt{PMCW<\textsc{CWShiftBMH,RFact:}} is an instantiation of the \texttt{PMCW} template with (the multiple keyword version of) the Boyer-Moore-Horspool shift function and a reverse factor automaton.

Once a toolkit design has been constructed based on the taxonomy, implementing the toolkit is straightforward, as the specific algorithms have all been derived in a common, abstract format as part of the taxonomy construction. We do not discuss the toolkit implementation step of the TABASCO process here, both for the aforementioned reason and because it is highly dependent on the implementation language chosen, C++ in the example given.

In addition to simplifying the design task, the taxonomy gives us structural invariants for the toolkit classes. These increase confidence in algorithm correctness and safety, and are therefore implemented by the classes: each has a class invariant member function which returns a Boolean to indicate an object’s conformance to this structural invariant. These functions have been useful in understanding the code as well as debugging it, and we expect them to be useful to future toolkit extenders as well.

More details on the design and implementation of \textsc{SPARE Time} (as well as its predecessor \textsc{SPARE Parts})—including the choice of C++ over Java, Eiffel and Delphi, the use of the C++ Standard Template Library, alternative alphabets, multi-threading support and the various shift functions and automata implemented—can be found in [WC04, Cle03, Wat95].

Since C++ was used as the implementation language for \textsc{SPARE Time}, our discussion was centered on design constructs available in C++. As indicated earlier, other implementation languages having similar design constructs could be used in a similar way. Additionally, the use of the \textit{Template Method} design pattern could be replaced by the use of different generative approaches: essentially, it is a way
of compile-time composing a number of small components (the template parameters) and an algorithm skeleton (the template method) into a single algorithmic component, and other generative ways to do this could be used.

### 7.4.2 Advantages of taxonomy-based toolkit design

Taxonomy-based toolkit design in our opinion has a number of advantages:

- The toolkit architecture is guided by the taxonomy structure, simplifying the design process.

- The uniformity of the taxonomy results in uniformity of the toolkit. This makes the toolkit (including its client interface, architecture, and implementation) easier to understand, debug, and extend. It also gives confidence in the accuracy of relative performance comparisons among the different algorithms. This accuracy is often not present in existing algorithm collections, due to the sometimes vastly differing styles used to implement and present the individual algorithms, as mentioned before.

- The factoring of algorithm commonalities in the taxonomy enables code sharing in the toolkit implementation.

- The correctness arguments associated with the algorithms in the taxonomy give confidence in correctness and safety of the toolkit, as pointed out in the previous section. Furthermore, the availability of the taxonomy presentation of the algorithms and their correctness arguments makes implementation painless and even fun.

- Related to the correctness arguments, the taxonomy in fact gives a complete formal specification of algorithms (including pre- and postcondition, invariant and possibly specification of theoretical running time and/or memory usage). As a result, each toolkit component has an explicit requirements specification as well. This helps in understanding such components, in creating a mapping from user requirements to toolkit components in the form of a DSL (see Section 7.5), and in tracking requirements (see Section 7.6).

- Finally, the toolkit design and implementation can serve as examples of relatively straightforward design and implementation techniques for toolkits for algorithmic domains.

Comparing these arguments to the list of toolkit design goals given previously, it is clear that the taxonomy-based design of a toolkit can help in attaining those goals. The advantages of taxonomy-based toolkit design may be partially offset by the amount of extra time required by the taxonomy construction, but that step has many additional advantages as well, as mentioned in Section 4.4.
In Chapter 8 a toolkit is described that covers many of the algorithms in the tree acceptance taxonomy of Chapter 5 and the tree pattern matching taxonomy of Chapter 6, as well as the underlying foundational algorithms and data structures. In that chapter, we will also consider how beneficial the availability of the taxonomies and formal definitions of foundational algorithms and data structures were for the design and implementation of the toolkit.

7.5 DSL design and implementation

The toolkit—whose design is obtained based on the taxonomy—is implemented as part of TABASCO’s domain implementation phase (see Figure 7.3.1). The resulting toolkit may then be used in application engineering. Since its structure is based on the taxonomy, algorithm details will play an important role in that structure. Users will usually be interested on the other hand in performance and problem details and may not know much about the algorithms or data structures in the domain. This difference makes it hard for the average user to effectively and easily use the toolkit components.

To overcome this problem a DSL (domain specific language) may be designed and implemented as part of TABASCO, as depicted in Figure 7.3.1. Such a DSL or ‘little language’ [Ben86, DK98] can be tailored towards the domain concepts that the average user is interested in or familiar with and allows him or her to specify requirements that a requested toolkit component should satisfy. This forces the user to specify his requirements as a DSL expression i.e. as a domain specific description, reducing the complexity of requirements management (compared to allowing the user to specify his or her requirements in a more general format).

A mapping is needed to get from user requirements in the form of a domain specific description, to the selection of one or more toolkit components satisfying these requirements. The mapping is part of a domain specific processor, which takes a domain specific description as input, uses the mapping to select one or more toolkit components, and yields the selected components as output to be used by the toolkit user.

The need for this mapping, and the user emphasis on performance, explains the presence of the benchmarking substep in the domain implementation phase of Figure 7.3.1, in addition to the toolkit and DSL implementation substeps. Note that the benchmarking step, which may also include profiling of toolkit code, can lead to changes in toolkit implementations, in effect forming a feedback loop within the domain implementation phase.

In the most basic form, the DSL will not be explicit, and the mapping will consist of some paragraphs of text specifying which toolkit components to use under which conditions. In this case the toolkit user essentially acts as the domain specific processor. For example, for the SPARE Time pattern matching toolkit such a text may
include sentences like ‘if performance independent of keyword set is required then use the Optimal Aho-Corasick algorithm implemented by PMAC(ACMachineOpt), else ...’ (based on [Wat95, p. 309]).

Alternatively, an explicit DSL may be designed which the user can use to specify the requirements he or she expects a component to satisfy. A domain specific description in the format of this DSL (i.e. a user requirements specification) can then be automatically processed to map to a certain toolkit component. This can be done by means of a domain specific processor, which may not only select a component from a toolkit but even be a generator i.e. generate new components based on the parts that are available as part of a toolkit.

As an example, we give a small DSL for specifying sorting algorithm components. This example is based on a sorting toolkit and DSL that were developed based on the sorting algorithms feature diagram we saw before [CF02]. The grammar of this DSL shows that a user may specify his requirements in terms of size of the array to be sorted, variance in values of such a list, running time performance and so on.

The DSL is specified in GenVoca grammar format, the format used in Czarnecki & Eisenecker’s book [CE00] as well. The GenVoca grammar has the following form:

```
Sorter sorter[ArraySize, UniqueRange, Randomness, 
    Stable, Auxiliaries, Order]
ArraySize   arraySmall | arrayLarge
UniqueRange rangeSmall | rangeLarge
Randomness  random | nearSorted | nearReversed | unknown
Stable      stable | nonStable
Auxiliaries useAuxiliaries | noAuxiliaries
Order       n_log_n | n_n
```

In this case, the DSL and mapping from user requirements (DSL expression) to toolkit components by means of a generator were implemented using C++ template metaprogramming techniques, but other techniques may be used instead, such as scripting languages and/or makefiles.

Using the C++ template metaprogramming approach, the parameters specified in the DSL expression by the user are compared to a decision table at compile time. The first row in the table that matches the parameters (and/or the default parameter values in case a user does not specify any) determines which sorter component is returned. Some parameters take precedence over others when selecting an algorithm component type to return.

For example, if the user requires an O(n log n) algorithm even for the worst case, the selection of a Mergesort or Heapsort is dictated, and due to the former having better performance according to the benchmarking results, Mergesort is chosen, unless auxiliary data structures may not be used, in which case Heapsort is the only remaining alternative. An example of how the DSL can be used to instantiate a sorter is the following:

```
#include "miloDSL.h"
```
typedef SORTER_GENERATOR<sorter<arraySmall<>, whatever, whatever,
    whatever, whatever, n_log_n<> > >RET aSorterType;

aSorterType mySorter
mySorter::sort( myArray, myArray+1000 );

The encoding of the mapping from DSL to toolkit components also includes informa-
tion on valid and preferred combinations of components, i.e. it includes the con-
figuration knowledge. Decision tables encoding the mapping from DSL expres-
sion to component are constructed based on benchmark results for the toolkit.

Taking it one step further, the domain specific processors—taking a domain specific
description and returning an appropriate toolkit component—may be generic, i.e. not
merely developed for a single DSL, but for DSLs in general. We have recently de-
veloped such a generic domain specific processor, in the form of an experimental tool
called FUEL [Ouw04].3 FUEL takes a description of a DSL—in the form of an XML
file containing its grammar and requirements-to-components mapping—and a do-
maint specific description/DSL expression, and generates an appropriate component
based on a template file. In this way, a new DSL can easily be created by supplying
different XML and template files to FUEL. When supplied with an XML file and
DSL expression (in the form of a sequence of parameters and corresponding values),
FUEL tries to generate a source file providing a component satisfying the DSL user’s
specification. FUEL is implemented in the Ruby scripting language [DTH04].

As an example, we consider the DSL for SPARE Time, called SPARE FUEL, that
was implemented using the FUEL tool. A part of the XML file specifying the DSL
is given below:

<fuel>
  ...
  <delimiter open="*" close="*"/>
</fuel>

<codefiles>
  <file src="sparefuel.src" target="sparefuel.h"/>
</codefiles>

<algorithms>
  <alg id="CW-[NFS, Factoracle]"
       classname="PMCW\CWShiftNFS, RFactoracle">"
    ...
  </alg>
  ...
</algorithms>

<rules>
  ...
  <rule name="alphabet" value="English" default="CWNFS">
    ...

3The FUEL DSL creation tool was developed by Jan Ouwens as part of his Master’s studies
internship [Ouw04].
<rule name="memory" value="high" default="CWNFS">
...
<rule name="length" value="3" default="ACOpt">
<rule name="setsize" value="1" default="CWNFS"/>
<rule name="setsize" value="2" default="CWNFS"/>
<rule name="setsize" value="3..20" default="ACOpt"/>
</rule>
...
</rule>
</rules>
</fuel>

The codefiles part specifies the name of the template file and of the output file to be generated. The algorithms part introduces identifiers for the various algorithms, which are represented by a name, class name and source file. The rules section, finally, codifies the mapping from user requirements to toolkit components. In the example, we see that for an English alphabet, with a high amount of memory available and a minimal keyword length of 3, by default, algorithm ACOpt is chosen. In case the size of the keyword set is specified to be 1 however, this default is overridden and algorithm CWNFS is selected instead.

The template file `sparefuel.src` referred to in the XML above looks like this:

/*
 * SPARE Time include file.
 * Generated by [COPYRIGHT.NAME*], version [COPYRIGHT.VERSION*].
 * Chosen algorithm: [ALGORITHM.NAME*]
 * Parameters: [CMDLINE.PARAMS*]
 */

#ifndef SPAREFUEL_H
#define SPAREFUEL_H

#include "[ALGORITHM.PARAM1*]"
typedef [ALGORITHM.CLASSNAME*] PMAlg;

#endif

Using the example XML and template files, the call

```
./fuel.rb sparefuel alphabet=English memory=high length=3 setsize=1
```

generates the following include file for the toolkit user.

/*
 * SPARE Time include file.
 * Generated by SPARE Fuel, version 0.1.
 * Chosen algorithms: C=W[NFS, Factoracle]
 * Parameters: alphabet=English, memory=high, length=3, setsize=1
 */
#ifndef SPAREFUEL_H
#define SPAREFUEL_H

#include "cws.hpp"
typedef PMC0<CWShiftNFS, RFactoracle> PMAlg;
#endif

The SPARE FUEL DSL example is based on SPARE TIME, a toolkit written in C++, but FUEL is language-independent in that it can deal with template files from languages like Pascal or Java as well.

The current version of FUEL is still experimental. Apart from its use for SPARE FUEL, it has only been applied to small example toolkits. One important restriction is that the order in which parameters appear on the command line should match the nesting order in which they appear in the XML file. This restriction could be removed in a future version of FUEL or a similar tool.

## 7.6 Evolution as part of TABASCO

Once a toolkit has been implemented—optionally, together with a corresponding DSL—and released for use, it will evolve. We will briefly discuss such toolkit and DSL evolution here. Evolution of a toolkit and DSL as well as of the underlying taxonomy may fall into either of two categories.

First, there is the category of toolkit- or use-driven evolution. Users may of course change their initial requirements based on what the toolkit actually offers, but it may also be that the toolkit does not offer any component satisfying his or her requirements, or that a user finds bugs in the toolkit. This may lead to feedback to the toolkit builder, who can then choose to improve or extend the toolkit by improving or adding algorithms. If an addition is to be made in the form of a new algorithmic or data structure component, this will then result in an addition to the taxonomy and a revision of the domain model in general and of the domain design.

It may also lead to feedback to the taxonomist, in cases where algorithms turn out to be incorrect regardless of their implementation. This type of evolution is depicted by the feedback arrow from the Application Engineering box to the TABASCO box in Figure 7.3.1.

Alternatively, the evolution may be taxonomy-driven. The taxonomist may discover new algorithms or data structures or improve his understanding of the commonalities and variations between existing taxonomy algorithms or data structures. This will then lead to changes in the taxonomy, and the changes may propagate to the domain design and domain implementation steps. In the end, they will result in a modified toolkit and DSL. The taxonomy changes discussed may consist of restructuring of taxonomy parts. Such restructuring may involve reordering details (to arrive at
certain algorithms in other ways), splitting up details (to refine the derivation steps, possibly allowing other algorithms to be derived or placed in a part of the taxonomy) or adding completely new details (to add new algorithms to the taxonomy). Since each algorithm has an associated specification, such taxonomy changes may influence the specification as well. This in turn influences the mapping from user requirements to toolkit components that is part of the DSL implementation.

7.7 Related work

The type of taxonomy development and algorithm derivation used in TABASCO has been used in the past by Jonkers for garbage collection algorithms [Jon83] and by Marcelis for attribute evaluation algorithms [Mar90]. Earlier taxonomies—for sorting algorithms—were constructed by Darlington [Dar78] and Broy [Bro83].

In [FGNZ00], a general and systematic approach to the construction of robust toolkits is described. In this approach, larger class hierarchies may be generated from formal specifications. Our approach has so far only been applied to construct relatively small toolkits, since the domains have been restricted in scope to algorithms or data structures solving one or a few closely related problems. The construction of a taxonomy—and thus of a formal specification of the problem and all its solutions—has a number of benefits by itself and is a fundamental part of TABASCO. In contrast, the approach discussed in [FGNZ00] already assumes a formal specification to exist to begin with.

In [Cza98], [CE00, Section 5.6], the Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL) is presented. DEMRAL is similar to TABASCO in its restriction to algorithmic domains and assumption of little or no third-party involvement during domain engineering. However, the method does not construct as formal and precise a domain model as an algorithm taxonomy and lacks many of the advantages of doing so. In particular, comparing algorithms—possibly with an educational goal—and finding new algorithms becomes harder. Furthermore, the resulting model is relatively informal, hence not as good for reasoning about algorithm correctness.

The TABASCO approach was previously described by Watson [Wat95] and in a paper by Cleophas and Watson [CW05]. In both publications, it had not yet matured into the domain modeling and domain engineering method presented here. As mentioned at the beginning of this chapter, an outline of TABASCO—co-authored by Vosloo and Watson—was presented to Dutch industry practitioners and researchers at the JACQUARD2005 event [CVW05]. Based on a subset of an earlier version of this chapter, an overview of the method appears in [CWKB05], co-authored by Watson, Kourie, and Boake. A journal version of that paper, in which Obiedkov added a discussion of concept lattices as an approach for taxonomy construction and was added as a co-author, appeared as [CWK*06].
The TABASCO approach has been (completely or partially) used for a number of algorithmic or data structure domains, including the examples mentioned earlier in this chapter. In particular, a lot of work was done or is being done as part of FASTAR. FASTAR (Finite Automata Systems—Theoretical and Applied Research) involves finite automata-related research at the Technische Universiteit Eindhoven and the University of Pretoria.\footnote{More information on FASTAR is available at \url{http://www.fastar.org}.} TABASCO work in FASTAR includes:

- **FIRE LITE**, a toolkit of finite automata construction and minimization algorithms based on taxonomies of such algorithms [Wat95].
- A taxonomy of *Minimal Acyclic Deterministic Finite Automata* (MADFA) construction algorithms [Wat01].
- A taxonomy [CWZ04b, CWZ04a, Cle03, Wat95] and toolkit (SPARE TIME) [CW05, WC04] of keyword pattern matching algorithms, used in many of the examples in this chapter.
- A taxonomy of approximate keyword pattern matching algorithms [Bos05].
- A taxonomy of algorithms for 2-dimensional pattern matching [Rij05].
- The taxonomies and toolkit of regular tree language algorithms described in the remaining chapters of this dissertation.

A somewhat different kind of taxonomy of *implementation strategies* for *Deterministic Finite Automata* (DFAs) is presented in [KKW06].

The TABASCO approach has been used for domains completely unrelated to finite automata as well. The results include:

- A taxonomy of Lempel-Ziv compression algorithms [Kou03].
- A taxonomy [BSWK04, BS02] and toolkit [Koo] of graph representations.

### 7.8 Final remarks

In this chapter, we presented TABASCO, our TAxonomy-BAsed Software COstruction domain modeling & engineering method. We gave a brief overview of domain engineering and generative programming and described TABASCO, explaining and motivating its steps in detail. In particular, the advantages of taxonomy construction were highlighted in Section 4.4 of Chapter 4, and those of taxonomy-based toolkit design were highlighted in Section 7.4.2.

In describing our method, we gave some examples from two domains to which TABASCO has been applied: keyword pattern matching and sorting algorithms. In
the preceding two chapters, we applied the domain modeling steps to the problems of tree acceptance and tree pattern matching, resulting in a taxonomy for each of them. Its domain engineering steps were applied afterwards, resulting in a toolkit containing algorithms from the taxonomies and underlying basic data structures and algorithms. This toolkit is considered in Chapter 8.

As future work, we see the application of TABASCO to other algorithmic or data structure fields. In particular, such work should lead to more explicit and more general guidelines on how to get from taxonomy or feature diagram to domain specific toolkit, and which other design patterns may play a role in this.

Finally, the experimental generic domain specific processor FUEL should be applied to domains other than the small examples and keyword pattern matching domain to which it has been applied so far. This research should show whether it is general enough to be used in future applications of TABASCO or should be extended.
Chapter 8

Forest FIRE

In this chapter, we consider FOREST FIRE, a toolkit of algorithms and automata constructions based on the two taxonomies discussed in Chapters 5 and 6. The toolkit’s taxonomy-based design, subsequent implementation, and benchmarking are applications of steps of the TABASCO domain engineering method considered in Chapter 7. We discuss the high-level design, which is based on analysis of the commonalities and variabilities apparent in the taxonomies. The underlying basic data structures and algorithms from Chapters 2 and 3 are considered briefly. We then compare some toolkit algorithms to the abstract algorithms serving as their counterparts, indicating the ease with which the former are obtained from the latter. Furthermore, we discuss the results of benchmarking experiments, and consider the experiences gained during the toolkit design, implementation, and benchmarking steps.

The FOREST FIRE toolkit and the accompanying graphical user interface FIRE WOOD were developed together with Roger Strolenberg. As part of his Master’s thesis research [Str07a] and during a temporary appointment afterward, he implemented the toolkit and performed the experiments. The design and data representation choices as well as the choice of experiments to be performed were made together with the author of this dissertation.

8.1 Introduction

In this chapter, we discuss the design of the FOREST FIRE toolkit, the experiments performed with the toolkit, and the experiences gained and conclusions reached as a result. Our discussion of the design takes place in the context of the TABASCO domain engineering method and focuses on the high-level design of the toolkit. In particular, we intend to show how the uniformity of presentation and factorization of commonalities in the taxonomies helped to obtain a similarly uniform and factorized
design. Furthermore, we indicate how the presentation of the taxonomy algorithms helped in implementing them. The design goals of the toolkit were manifold. First of all, TABASCO’s general toolkit design goals—mentioned in Section 7.4—apply to the toolkit. That is, it should

- provide an easy to understand client interface,
- have a coherent design,
- be comprehensible for modification or extension purposes,
- be general purpose,
- have a high level of efficiency, and
- enable practical performance comparisons.

For this toolkit, the last goal and two additional ones are particularly important:

- Enabling practical performance comparisons, as mentioned above. The results of such comparisons make it easier to choose between different algorithms for practical applications.
- Enabling experimentation with the various algorithms to aid in understanding the algorithms.

A graphical user interface named FIRE Wood was therefore developed together with the toolkit. The GUI includes facilities for input, output, creation and manipulation of data structures from the toolkit, e.g. ranked alphabets, trees, tree sets, regular tree grammars, and tree automata. This allows the use of the toolkit for interactive experiments.

- Providing more experience with and evidence for taxonomy-based toolkit design.

As witnessed by the related work on taxonomies and toolkits mentioned in Section 7.7 of the TABASCO chapter, relatively few taxonomy-based toolkits exist, even though quite a few algorithm taxonomies have been developed. The application of taxonomy-based toolkit design to the taxonomies of Chapters 5 and 6 should give more evidence and guidelines for the usefulness of the method.

Note that the absolute efficiency of the algorithms in the toolkit was a less important goal for this particular toolkit. For practical applications in term rewriting or instruction selection, the algorithms could probably be tuned to exhibit better running time and memory usage.
The toolkit was implemented in the Java programming language, using Eclipse as a development environment. The GUI was developed using Java and the Standard Widget Toolkit (SWT), a multi-platform graphical widget toolkit. The toolkit and GUI were developed for and used on the Apple Mac OS X and Microsoft Windows XP platforms in particular.

The FOREST FIRE toolkit contains algorithms from the tree acceptance and tree pattern matching taxonomies of Chapters 5 and 6, as well as basic data structures and algorithms related to the concepts and transformations contained in Chapters 2 and 3.

The latter include representations of trees, alphabets, regular tree grammars, pattern sets, various kinds of tree automata, algorithms for determining certain properties of grammars and grammar productions, and grammar transformations. These are not part of the two algorithm taxonomies, but are required or useful as basic building blocks for the implementation of algorithms from the taxonomies.

The taxonomy algorithms that are part of the toolkit mainly occur near the leaf ends of the taxonomy graphs, as those algorithms seem the most concrete and useful; the more abstract algorithms higher up in the taxonomy graphs may not even be implementable (although they may lead to abstract base classes/interfaces).

For the tree acceptance taxonomy, algorithms implemented include those in the (T-ACCEPTOR, RF), (T-ACCEPTOR, FR), and (MATCH-SET, REC, TABULATE) taxonomy branches, as well as the automata constructions used in these branches. In total, the implemented algorithms cover five algorithm nodes and over twenty different automata constructions.

![Diagram of Tree acceptance taxonomy](image)

**Figure 8.1.1** Tree acceptance taxonomy

For the tree pattern matching taxonomy, algorithms implemented include those in the (T-MATCHER, RF), (T-MATCHER, FR), (MATCH-SET, REC, TABULATE), and (S-PATH, SP-MATCHER, DET) branches of that taxonomy, as well as the automata constructions and filters used in these branches. In total, the implemented algorithms cover seven algorithm nodes and about ten different automata constructions.
The rest of this chapter is structured as follows:

- In Section 8.2 we consider related work in brief.
- In Section 8.3 we discuss the high-level design of the toolkit, based on the algorithm taxonomies and the algorithms and automata constructions in them. Here, the focus is on the influence of the taxonomies on toolkit design. The design hierarchy of the automata classes for the various automata types is discussed here as well.
- Section 8.4 discusses the design of basic data structures and algorithms from Chapters 2 and 3, including trees, pattern sets, regular tree grammars, and analyses and transformations of such grammars.
- Section 8.5 compares some of the abstract algorithms and constructions from the taxonomies to their implementations as part of the toolkit.
- Section 8.6 briefly considers the experiments that were performed with the toolkit. The results of some of these experiments were already considered in Chapters 5 and 6.
- Section 8.7 concludes the chapter by discussing the experiences gained during the design, implementation and experimentation with the toolkit and GUI and drawing some conclusions from the results.

## 8.2 Related work

For related work on taxonomy-based toolkits in general, we refer to Chapter 7 on TABASCO, and in particular to the related work section of that chapter, i.e. to Section 7.7. Here, we merely discuss related work in the area of tree algorithms.

Compared to the descriptions of tree pattern matching and tree acceptance algorithms in the literature, implementations of such algorithms (and the extension to
tree parsing) are rather scarce and are just as scattered. No large collections of implementations of such algorithms seem to exist.

Applications in tools for instruction selection choose and implement a single tree parsing algorithm (usually based on an extension of a tree acceptance or tree pattern matching algorithm). Such applications include work by Hatcher and Christopher [HC86] and tools like BEG [ESL89], BURG and iBURG [FHP92a, FHP92b, Pro95], and TWIG [AGT89].

Timbuk [GT01] uses tree algorithms and tree automata for a completely different purpose, namely for reachability analysis on term rewriting systems. It uses an NFRTA to represent an approximation of the set of terms reachable by a term rewriting system and another NFRTA to represent terms for which to determine reachability. It implements operations such as intersection and emptiness to solve the reachability problem.

Finally, TREEBAG [Dre] is a system for generating, transforming and displaying trees. It contains various kinds of tree grammars and tree transducers and allows them to be composed to generate and transform trees.

8.3 High-level design

We consider the high-level design of the taxonomy algorithms, the automata constructions, and the automata used in them (as well as related concepts such as states, items, and item sets). We show how the commonalities and variabilities between various algorithms and constructions influence the design. It is these commonalities and variabilities that are made explicit in the taxonomies, which thus help in designing the toolkit.

8.3.1 Taxonomy algorithms

In the tree acceptance taxonomy, algorithms implemented as part of the toolkit fall into two categories: branch (T-ACCEPTOR, RF) with algorithms using RF-based nondeterministic TAs, and branch (T-ACCEPTOR, FR) with those using FR-based (nondeterministic or deterministic) TAs. In the tree pattern matching taxonomy, a third category is implemented, consisting of stringpath matching algorithms, forming branch (S-PATH, SP-MATCHER, DET).

All of the tree acceptance and tree pattern matching algorithms are implemented as Java classes. For each taxonomy, all of the implemented algorithms have the same public interface:

- a constructor setting the automaton to be used for acceptance and pattern matching respectively, and
• a method `accept` or `match` taking a tree parameter and running an acceptance or pattern matching algorithm on the tree

Coplien’s work, focusing on C++, suggests the use of abstract base classes and inheritance [Cop98, p. 151] as a design construct. In Java, the two interfaces are more naturally represented by Java interfaces, called `IAcceptor` and `IMatcher`, respectively.

The classes for tree acceptance algorithms share an operation for computing acceptance, yet they differ in algorithm, data structure and state. The suggested C++ design [Cop98, p. 151] would use inheritance with virtual functions; in Java, classes implementing the above interfaces can be used instead. Classes `DFRAcceptor`, `NFRAcceptor`, and `NRFAcceptor` therefore implement interface `IAcceptor`. Similarly, for the tree pattern matching taxonomy, classes `DFRMatcher`, `NFRMatcher`, and `SPMatcher` implement interface `IMatcher`.

The reader may note that the use of filtering in `DFRTA/match` set-based tree acceptance or pattern matching changed the algorithms in the taxonomy, i.e. led to Algorithms `{match-set, rec, filter, tabulate}` in both taxonomies, compared to Algorithms `{match-set, rec, tabulate}` for `DFRAs/match` sets without filtering. However, the change is limited to the state computation at a node based on the states for the child nodes: The latter algorithm uses

\[ MS'_{\text{node}} := T_a(qz_1, \ldots, qz_n), \]

while the former uses

\[ MS'_{\text{node}} := T_a(\phi_{a,1}(qz_1), \ldots, \phi_{a,n}(qz_n)). \]

This change can be and is encapsulated in method `nextState` of the automata classes used (discussed in Section 8.3.3). This corresponds to the use of the `Strategy` design pattern [Cop98, p. 250], [GHJV95, p. 315]. The use of filtering thus has no influence on the class hierarchy for the taxonomy algorithm classes.

The class hierarchy for tree pattern matching taxonomy algorithms is depicted in Figure 8.3.1. Note that each of the classes implementing interface `IMatcher` has a generic parameter for a descendant of interface `IItem`, since the type of automaton used in each class depends on the item type used.

The hierarchy for tree acceptance is omitted but similar. In Section 8.5.1 we compare the implementation of method `accept` in class `DFRAcceptor` to the underlying abstract acceptance algorithm (`TAcceptor`, `FR`, `DET`).

**Remark 8.3.1.** Note that an additional level in the class hierarchy could be introduced to group the nondeterministic acceptance algorithms and the nondeterministic pattern matching algorithms respectively, independent of traversal direction. We have not done so, since these would have no unique commonality except for the notion of nondeterminism.
8.3 High-level design

8.3.2 Automata constructions

Tree automata constructions appeared as mathematical definitions in Sections 5.6 and 6.5. Algorithmic, imperative versions of the DFRTA constructions appeared in Sections 5.7 and 6.6, where their imperative reachability-based construction was discussed (using a match set view on DFRTAs).

(For stringpath-based tree pattern matching, a separate tree automaton construction and two string automaton constructions were discussed in Section 6.7; their implementation as part of the toolkit is discussed in Section 8.3.2.6.)

Figure 8.3.2 Class hierarchy for generators implementing automata constructions

In the toolkit, the constructions are implemented as part of automata generator classes. For the DFRTA constructions, these implement the imperative reachability-based constructions of Sections 5.7 and 6.6; for the nondeterministic automata con-
structions, these implement imperative versions of the mathematical definitions of Sections 5.6 and 6.5.

As we saw in those four sections, the constructions differ in a number of aspects:

- the \((\text{non})\text{determinism}\) of the resulting automata,
- the presence of \(\varepsilon\)-transitions in the resulting automata,
- the direction of the resulting automata,
- the use of filtering, in the case of DFRTAs, and
- the item types and sets used to construct the states of the resulting automata.

### 8.3.2.1 Deterministic and nondeterministic automata

The construction of deterministic and nondeterministic tree automata is quite different, and the use of filtering is applied for DFRTAs only. The construction algorithms therefore would have the availability of an operation for generating a tree automaton as their main commonality; however, even the parameters supplied to these operations would be different (e.g. a boolean to indicate whether to create an automaton with or without \(\varepsilon\)-transitions for the case of nondeterministic tree automata, but not for the case of deterministic ones). Two separate tree automata generator classes were therefore created: \textit{DFRTAGenerator} and \textit{NTAGenerator}.

### 8.3.2.2 Tree acceptance and tree pattern matching

Both generators need to work for TA constructions for tree acceptance as well as for tree pattern matching. Since they share semantics/operations (they both generate tree automata), but the actual function signature and algorithms differ (one taking an RTG parameter, the other a pattern set one), we use function overloading, as suggested by Coplien’s work [Cop98, p. 150]. That is, both class \textit{DFRTAGenerator} and class \textit{NTAGenerator} have two methods \textit{generateAutomaton}, one with an RTG input parameter and one with a pattern set input parameter. (Internally, the functions re-use a lot of code for the two cases, as the two cases mainly differ in the definition of the closure function used, i.e. a chain rule closure versus the addition of the variable symbol \(\nu\)—see Section 6.6.)

### 8.3.2.3 Encapsulating direction

As for the \textit{direction} of the automata generated, the difference between RF and FR directed nondeterministic tree automata does not play a role in class \textit{NTAGenerator}: the \textit{generateAutomaton} methods are passed an empty automaton of a certain type, and use such an automaton’s \textit{addTransition} method to add transitions. This
can again be seen as an instance of the Strategy design pattern [Cop98, p. 249]; the direction of the stored transition is encapsulated in the directed automaton’s transition addition method. (The design of the various automata classes is considered in Section 8.3.3.)

8.3.2.4 Encapsulating filtering

The use of filtering changes the reachability-based DFRTA/match set tabulation algorithm (see Algorithms 5.7.21 and 5.7.38 in Sections 5.7.3.1 and 5.7.5.1 respectively). The changes are mainly in the computation and updating of filter tables. In fact, the imperative algorithm for unfiltered DFRTAs is a simplification of the one for filtered DFRTAs; in the toolkit, class DFRTAGenerator therefore contains a method generateAutomaton containing a general reachability-based algorithm. The algorithm parts that differ among the four filtered constructions and the unfiltered construction are encapsulated as methods of the respective DFRTA subtypes. An empty instance of such a subtype is passed to the generateAutomaton method. This again corresponds to the use of the Strategy design pattern.

8.3.2.5 Different item types and item sets

Both the DFRTAGenerator and the NTAGenerator also depend on the type of items used to construct tree automata states. The item types correspond to e.g. the subtrees and dotted trees used in constructions described in the taxonomies. The structure of the generators and the generateAutomaton methods does not depend on the item type used. The suggested C++ design would be to templatize both classes by item type [Cop98, p. 123]; in Java, we correspondingly use generics, giving the classes a type parameter for the item type. (The design of the various item type classes is considered in Section 8.3.3.) The set of items to be used in an automaton construction is passed as an additional parameter to the generateAutomaton methods, of type AbstractItemSet.

In total, 20 DFRTA constructions from the taxonomies are implemented by class DFRTAGenerator: 15 for tree acceptance, and 5 for tree pattern matching, corresponding to the unfiltered and four filtered DFRTAs, and—for tree acceptance—to three different item sets. Furthermore, class NTAGenerator implements eight constructions—εNRFTA, εNFRTA, NRFTA, and NFRTA, the latter two for the three different subtree item sets ALL-SUB, PROPER-N, and PROPER-S—for tree acceptance, and two—NRFTA and NFRTA—for tree pattern matching.

8.3.2.6 Constructing stringpath matching automata

Three stringpath-based tree pattern matching automata were discussed in Section 6.7: an Aho-Corasick string automaton and a variant of it, and a stringpath DFRTA. The actual construction of the Aho-Corasick automata and that of the
DRFTA are quite different: the former use Aho-Corasick’s direct construction instead of the conceptually simpler but slower four step procedure discussed in Section 6.7. The two Aho-Corasick automata kinds’ constructions do have many commonalities and mainly differ in small details of the algorithms, suggesting the use of the Template Method design pattern [Cop98, p. 249]. This leads to separate generator classes SPDRFTAGenerator, AbstractACGenerator, and descendants OriginalACGenerator and StringPathACGenerator of the latter. Class SPDRFTAGenerator has a method generateAutomaton taking a pattern collection as an input parameter. Class AbstractACGenerator has a similar method, whose implementation includes calls to private template methods that are implemented in its two subclasses.

### 8.3.3 Automata, states, and items

Each of the automata types that plays a role in the taxonomies is represented by a class. The inheritance hierarchy of the various automata classes is depicted in Figure 8.3.3.

![Class hierarchy for automata](image)

**Figure 8.3.3** Class hierarchy for automata. To simplify the diagram, class DFRTAFILTERED is shown instead of the four actual sibling classes for filtered DFRTAS.

As mentioned when we discussed the automata constructions, automata differ in direction and (non)determinism. This explains the AbstractNTA subpart of hierarchy, as well as the presence of the AbstractDFRTA class. Furthermore, the use of filtering explains the inheritance hierarchy rooted at class AbstractDFRTA.
8.3 High-level design

8.3.3.1 States

For the tree automata used in this dissertation, a state corresponds to either a single item or a set of items, depending on the (non)determinism of the automaton. For simplicity reasons, the toolkit uses a single class AutomatonState containing a set of items for states of both deterministic and nondeterministic automata. Most automata use subtrees as items in states, but the stringpath DRFTAs of Section 6.7 are based on dotted trees as items. States thus have a variability in item type. The suggested C++ design would add a template parameter to the AutomatonState class; as we have seen before, the corresponding Java design therefore uses a generic parameter for the item type, in this case, for a descendant of interface IItem. Automata classes have the state type as a variability, which would lead to a generic parameter corresponding to the state type used (which itself is dependent on a generic parameter for the item type, as mentioned above), i.e. a descendant of class AutomatonState<IItem>. Unfortunately, generics in Java are currently limited such that an explicit generic parameter for the item type alone is not enough; both a generic parameter for the item type—a class implementing IItem—and one for the state type based on this item type—a descendant of AutomatonState<IItem>—are needed, although a single concrete type needs to be substituted for both IItem instances.

8.3.3.2 Items and item set providers

As mentioned before, various types of items for use in states exist. These have related operations, but differ in data structure and algorithm. The design therefore uses an interface IItem—defining methods for comparing items and retrieving item structure—and inheritance with virtual functions, leading to classes Subtree et cetera implementing the interface (see Figure 8.3.4).

Furthermore, sets of items play a role in the toolkit. Such sets depend on the type of item contained in them, again leading to the use of generics with a generic parameter for the item type of an item set. Additionally, sets of items of different types (subtrees, dotted trees, ...) have different operations, e.g. for retrieving the item corresponding to a particular node in a tree. This again leads to the use of inheritance with virtual functions: classes such as SubtreeSet descend from the generic class AbstractItemSet<IItem>, as shown in Figure 8.3.4.

Similarly, item set provider classes exist, which create and store specific item sets based on a given RTG or pattern set. This leads to AbstractItemSetProvider<IItem>, another generic class. Specific item type instantiations of this generic class form the parent class for the different item set providers. Part of the class hierarchy is depicted in Figure 8.3.5 (in which item set providers for dotted rules based on RTGs have been omitted).
8.3.3.3 Stringpath automata

As indicated before, the constructions of Aho-Corasick automata and of DRFTAs for stringpath matching are quite different. This resulted in separate generators for the classes. Similarly, we also have separate classes for the automata, which have some commonalities (the use of stringpaths), but differ in e.g. data structure (transition tables). As shown in Figure 8.3.3, class AbstractSPA (inheriting from instance AbstractTA<DottedTree, AutomatonState<DottedTree>> of generic class AbstractTA) has two descendants, class AhoCorasickAutomaton and class SP-DRFTA. These automata types use a different kind of state from the state type used by other automata; their states store stringpaths related to the output function for stringpath automata. These states are objects of class StringPathAutomatonState, a descendant of AutomatonState<DottedTree>.
8.4 Basic data structures and algorithms

The FO\textsc{rest} FO\textsc{r}e\textsc{e}t \textsc{f}I\textsc{ire} toolkit contains a number of basic data structures and algorithms related to the concepts and transformations from Chapters 2 and 3: representations of trees, alphabets, regular tree grammars, and pattern sets clearly are required to implement the high-level design of Section 8.3. Certain basic algorithms described in those chapters, most of them dealing with regular tree grammars, are useful to implement as well.

We only mention some design aspects to give the reader an impression. More details can be found in \cite{Str07a, Section 3.1}.

Representations for the most basic concepts (trees, alphabets, pattern sets) were relatively easy to choose. (On the other hand, the choice of efficient representations for the transition tables of the various tree automata—whose high-level design and structure were discussed in the preceding section—took quite some time and effort. See \cite{Str07a, Section 3.1} for details on the representations chosen.)

Data structures such as symbols, ranked symbols, and alphabets are each represented using classes of similar names, with class \textsf{RankedSymbol} inheriting from class \textsf{Symbol}. Trees are represented using a class \textsf{Tree}, containing for example references to their alphabet, root node and sequence of frontier nodes, as well as methods to for example get and set these references, or obtain a tree's string representation. Nodes in turn are represented using a class \textsf{Node}. Apart from obvious references to parent and child nodes and to a symbol, instances of this class also contain a dictionary data structure to store annotations. The latter is used by tree pattern matchers to store patterns matching at a node.

Regular tree grammars are represented using class \textsf{RegularTreeGrammar}, which contains an \textsf{Alphabet} (in which each \textsf{Symbol} object indicates the symbol's type as terminal or nonterminal), a start symbol reference, and a set of production rules. The latter are represented using class \textsf{Production}, which contains a reference to a left hand side \textsf{Symbol} and right hand side \textsf{Tree}.

Additionally, classes to analyze RTGs and perform transformations on them are included in the toolkit: \textsf{RTGUsabilityAnalyzer}, \textsf{RTGUsabilityRemover}, \textsf{RTGStandardAnalyzer}, and \textsf{RTGStandardRemover}. The former two contain methods to compute and remove productions and symbols that are unreachable, unproductive, or useless. The latter two contain similar methods for chain rules and Z-nodes (see Sections 3.3.1–3.3.3). Furthermore, a class \textsf{Warshall} implements Warshall's algorithm (Section 2.4) for the transitive nonterminal closure of a given RTG. This class is used in the implementation of e.g. chain rule removal in class \textsf{RTGStandardRemover}, as we will show in Section 8.5.2.

Other classes exist to represent e.g. pattern sets and stringpaths, to generate trees, and to analyze trees and regular tree grammars.
8.5 Abstract algorithms versus implementations

In this section, we compare two abstract algorithms from the taxonomy and theory chapters of this dissertation to their implementations as part of Forest FIRE. The comparisons serve as examples of how close the abstract algorithms and their implementations are, and of how straightforward implementation was, given the taxonomy and definitions in the tree theory chapter.

8.5.1 Example 1: Algorithm (T-ACCEPTOR, FR, DET)

As a first example, we consider Algorithm 5.5.4 (T-ACCEPTOR, FR, DET), which we repeat here for convenience.

\[
\begin{align*}
\text{const } G &= (N', \Sigma, r', \text{Prods}', S') : \text{ augmented RTG}; \\
t &\in \text{ Tr}(\Sigma, r); \\
\text{var } b : \exists \\
\text{let } M = (Q, \Sigma, r, R, Q_{ra}) \text{ be a DFRTA such that } \mathcal{L}(M) = \mathcal{L}(G); \\
b &:= \text{ Traverse}(e) \in Q_{ra} \\
\text{func Traverse}(\downarrow n : D) : Q = \\
\text{var } q_1, \ldots, q_n : Q \\
\text{let } a = t(n); \\
\text{if } n > 0 \rightarrow \\
\text{ Traverse } := R_a(\text{ Traverse}(n \cdot 1), \ldots, \text{ Traverse}(n \cdot n)) \\
\text{else } \\
\text{ Traverse } := R_a() \\
\text{fi}
\end{align*}
\]

As indicated earlier, the algorithm is implemented by public method accept in class DFRAcceptor<Subtree>, which implements interface IAcceptor. The code for the generic class DFRAcceptor is shown below.

When comparing the mainline of the abstract algorithm and its function Traverse to the Java implementation in public method accept and private method traverse, the similarities are obvious. The code is slightly different, as the node symbol \(a\) needs to be looked up in the DFRTA's alphabet, and due to the introduction of local variables qList and q to store the vector of child node states retrieved and the return value computed.

The availability of the short, abstract algorithms and the preceding high-level design made the implementation of this and similar acceptors and matchers quite straightforward. The acceptors in the first version of the toolkit [Str07a] were implemented in less than an hour, while an extension to (two different) tree parsing algorithms—described in [Str07a, Section 4.4]—took under two hours [Str07b].
8.5 Abstract algorithms versus implementations

```java
public class DFRAcceptor<T extends AbstractItem> implements IAcceptor {
    protected AbstractDFRTA<T, AutomatonState<T>> m;
    public DFRAcceptor(AutomatonState<T> m) {
        m = m;
    }
    public boolean accept(Tree t) {
        AutomatonState<T> rootState = Traverse(t.getRoot());
        return m.rootAccepting().contains(rootState);
    }
    private AutomatonState<T> traverse(Node n) {
       RankedSymbol a = (RankedSymbol)m.getAlphabet().getSymbol(n.symbol().name());
        ArrayList<AutomatonState<T>> qList = new ArrayList<AutomatonState<T>>();
        for (int i=0; i < a.rank(); i++) {
            qList.add(traverse(n.children[i], get(i)));
        }
        AutomatonState<T> q = null;
        if (n.children().size() > 0) {
            q = m.nextState(qList, a);
        } else {
            q = m.nextState(qList, a);
        }
        return q;
    }
}
```

8.5.2 Example 2: Transformation step RED-U

As a second example, we consider Transformation step 3.3.37 RED-U, which we repeat here for convenience.

Transformation step (RED-U, removing a single unit production). Let $G = (N, \Sigma, r, Prods, S)$ be an RTG with characteristic value $u_+$. Then there must be a production $A \rightarrow B \in Prods$ with $A, B \in N$. Then $G' = (N, \Sigma, r, Prods', S)$ where

$$Prods' = Prods \setminus \{A \rightarrow B\} \cup \left\{ \text{Set } C, \gamma : B \Rightarrow C \land C \rightarrow \gamma \in Prods : A \rightarrow \gamma \land \gamma \notin N \right\}$$

is the resulting transformed grammar.

Informally, the unit production $A \rightarrow B$ is removed and a rule $A \rightarrow \gamma$ is added for each tree $\gamma$ which is not a single nonterminal and which can be derived in one step
from a nonterminal \( C \) which is reachable (in zero or more steps) from \( B \). Note that \( B \Rightarrow C \) in the set quantification can only involve unit productions, due to the restricted form of productions in an RTG.

A remark following this definition in Section 3.3.3 already indicated that Warshall’s algorithm (of Section 2.4)—implemented in class Warshall—can be used in the computation of the reflexive transitive closure in this transformation step. We give the code of method removeChainRule of class RTGStandardRemover, which implements the above transformation step using class Warshall.

In line 2, the chain rule \( A \rightarrow B \) to be removed is looked up in the grammar (as the object representing the chain rule input parameter may not be the same object representing the chain rule in the grammar). Lines 4–5 construct an instance of the Warshall class, and use it to compute the transitive nonterminal closure for \( B \). Line 6 adds \( B \) itself, to obtain the reflexive transitive closure.

The chain rule \( A \rightarrow B \) is removed in line 8, after which lines 10–18 compute the new productions to be added and store them in newRules: if a production is of the form \( C \rightarrow \gamma \) with \( B \Rightarrow C \) (line 12) and \( \gamma \notin N \) (line 13), a new production \( A \rightarrow \gamma \) is created (lines 14–15) and added to a set newRules (line 16). Finally, line 20 adds set newRules to the grammar’s production set. (Note that adding new productions to this set as they are computed would change the collection of productions over which the for-loop of lines 11–18 iterates during iteration, possibly leading to incorrect results.)

```
public static void removeChainRule(RegularTreeGrammar g, Production rule) {
    Production rAt0B = CloneRetrieval.findRuleInGrammar(g, rule);
    Warshall cl = new Warshall(g);
    ESSet<Symbol> cIB = cl.getClosureSymbols(rAt0B.rhs().getRoot().symbol(), true);
    cIB.add(rAt0B.rhs().getRoot().symbol());
    g.productions().remove(rAt0B);
    ArrayList<Production> newRules = new ArrayList<Production>();
    for (Production rCtoGamma: g.productions()) {
        if (cIB.contains(rCtoGamma.getLhs()))
            if (rCtoGamma.rhs().getRoot().symbol().symbolType() != SymbolType.NonTerminal) {
                Production rAt0Gamma = new Production( rAt0B.getLhs(), rCtoGamma.rhs().clone() );
                newRules.add(rAt0Gamma);
            }
    }
    for (Production rAt0Gamma: newRules) { g.productions().add(rAt0Gamma); }
}
8.6 Experiments

A number of experiments was performed with the FOREST FIRE toolkit and the accompanying FIRE WOOD graphical user interface. The experiments fall into three categories:

- Experiments with the RTG transformations for removal of chain rules and Z-nodes (non-root terminal nodes). These transformations were discussed in Section 3.3.3, while their implementation in the toolkit was briefly referred to in Section 8.4.

The experiments were performed with different RTGs (including the ones used in the experiments reported on in Chapter 5) and with different orderings of transformation steps. The results can be found in [Str07a, Section 4.2]. The experiments played an important role in getting insight into the effects and interactions of such transformations steps. They influenced the content of Section 3.3.3; in particular, the experiments disproved a ‘proof’ for an earlier version of Lemma 3.3.36.

- Experiments with the different automata constructions. In Chapters 5 and 6, various constructions for tree and string automata for use in tree acceptance and tree pattern matching algorithms were discussed.

Experiments with various constructions’ implementations were performed, using a number of different RTGs/pattern sets. Results of these experiments with DFRTA constructions and with stringpath automata constructions are reported and analyzed in Sections 8.6.1 and 8.6.2.

More detailed results (and results for (ε)NFRTA and (ε)NRFTA constructions) can be found in [Str07a, Appendix E].

- Experiments using the different automata constructed. Both automata for tree acceptance and for tree pattern matching were used, using the appropriate algorithms from the taxonomies. These experiments ran the algorithms for acceptance and pattern matching on different sets of subject trees. The results are discussed in Section 8.6.3, and are followed by some conclusions based on this category of experiments and the previous one.

8.6.1 Acceptance automata constructions

Experiments were performed with the small RTG of Example 5.7.4 used throughout Chapter 5, and with RTGs for instruction selection for actual processor families (Intel x86, Intel IA64, and SPARC instruction set architectures) taken from the Mono project (an open source implementation of the Microsoft.NET architecture). The results in Table 8.1 include the memory usage of the transition tables for the basic DFRTA construction.
Table 8.1 DFRTA constructions using item set Items

| RTG      | \(|\text{Items}\)| | \(|\text{Q}\)| | Entries in tables | Memory usage of tables | Construction time |
|----------|----------------|---------|-----------------|-------------------|------------------|
| Example 5.7.4 | 8 | 8 | 74 | 5.1 KiB | < 1 ms |
| Intel X86 | 532 | 557 | 24907955 | 144.3 MiB | 260459 ms |
| Intel IA64 | 441 | 438 | 14075158 | 81.5 MiB | 200014 ms |
| Sun SPARC | 491 | 487 | 18342396 | 106.1 MiB | 176338 ms |

Reducing the number of items by using \(\text{Items}' = N \cup \text{ProperSubtrees}(\text{RHS}(\text{Prods}'))\) instead of \(\text{Items} = \text{Subtrees}(\text{RHS}(\text{Prods}'))\) results in reduced memory use, as shown in Table 8.2. Comparing the construction times to those in Table 8.1 shows that the DFRTA construction time may also be reduced substantially. (Note however that the use of \(\text{Items}'\) has the important disadvantage of increasing the running time of tree parsing algorithms, as noted at the end of Section 5.7.3.1.)

Table 8.2 DFRTA constructions using item set Items'

| RTG      | \(|\text{Items}'\)| | \(|\text{Q}\)| | Entries in tables | Memory usage of tables | Construction time |
|----------|----------------|---------|-----------------|-------------------|------------------|
| Example 5.7.4 | 5 | 6 | 44 | 4.2 KiB | < 1 ms |
| Intel X86 | 63 | 65 | 348299 | 2.3 MiB | 1626 ms |
| Intel IA64 | 44 | 42 | 135562 | 1.0 MiB | 636 ms |
| Sun SPARC | 51 | 53 | 225066 | 1.5 MiB | 925 ms |

For filtering as described in Section 5.7.5.1, the examples there showed that the number of entries in the main tables for the small RTG of Example 5.7.4 was reduced most by combined symbol & index filtering (CFILT), followed by index filtering (IFILT), symbol filtering (SFILT), and subtree filtering (TFILT).

As Table 8.3 shows, the order of reduction of number of entries is the same for the Mono grammars, except for the order of IFILT and SFILT.

The tables’ memory usage for all the examples however increases from IFILT to TFILT, then SFILT, and finally CFILT. This can be explained by noting that the different filters may result in different numbers and sizes of filtering tables. (Note that TFILT always results in one filtering table. As there are hundreds of terminals of rank 1 or 2 (and none of greater rank) in the Mono grammars, IFILT will result in just two filtering tables, while SFILT will result in hundreds of filtering tables, and CFILT will result in even more as there are also symbols of rank > 1.)
### Table 8.3 DFRTA constructions including ones with filtering

<table>
<thead>
<tr>
<th>RTG</th>
<th>Item set</th>
<th>Filter</th>
<th>Entries in tables</th>
<th>Memory usage of tables</th>
<th>Construction time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 5.7.4</td>
<td>Items</td>
<td>None</td>
<td>74</td>
<td>5.1 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>Items′</td>
<td>None</td>
<td>44</td>
<td>4.2 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>TFLIT</td>
<td>32</td>
<td>6.3 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>SFILT</td>
<td>21</td>
<td>8.3 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>IFILT</td>
<td>18</td>
<td>7.2 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>CFILT</td>
<td>14</td>
<td>9.6 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td>Intel X86</td>
<td>Items</td>
<td>None</td>
<td>24907955</td>
<td>14.3 MiB</td>
<td>273009 ms</td>
</tr>
<tr>
<td></td>
<td>Items′</td>
<td>None</td>
<td>348299</td>
<td>2.3 MiB</td>
<td>1626 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>TFLIT</td>
<td>337821</td>
<td>2.5 MiB</td>
<td>7398 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>SFILT</td>
<td>2097</td>
<td>4.6 MiB</td>
<td>267 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>IFILT</td>
<td>160651</td>
<td>1.4 MiB</td>
<td>2883 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>CFILT</td>
<td>1207</td>
<td>10.1 MiB</td>
<td>288 ms</td>
</tr>
<tr>
<td>Intel IA64</td>
<td>Items</td>
<td>None</td>
<td>14075158</td>
<td>82.7 MiB</td>
<td>200014 ms</td>
</tr>
<tr>
<td></td>
<td>Items′</td>
<td>None</td>
<td>135562</td>
<td>1.0 MiB</td>
<td>636 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>TFLIT</td>
<td>129342</td>
<td>2.0 MiB</td>
<td>3900 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>SFILT</td>
<td>1351</td>
<td>4.5 MiB</td>
<td>168 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>IFILT</td>
<td>54706</td>
<td>1.5 MiB</td>
<td>1495 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>CFILT</td>
<td>915</td>
<td>8.7 MiB</td>
<td>216 ms</td>
</tr>
<tr>
<td>Sun SPARC</td>
<td>Items</td>
<td>None</td>
<td>18342396</td>
<td>106.1 MiB</td>
<td>176338 ms</td>
</tr>
<tr>
<td></td>
<td>Items′</td>
<td>None</td>
<td>225066</td>
<td>1.5 MiB</td>
<td>925 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>TFLIT</td>
<td>208720</td>
<td>1.6 MiB</td>
<td>4339 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>SFILT</td>
<td>1502</td>
<td>4.1 MiB</td>
<td>187 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>IFILT</td>
<td>97543</td>
<td>0.9 MiB</td>
<td>1593 ms</td>
</tr>
<tr>
<td></td>
<td>Items</td>
<td>CFILT</td>
<td>1001</td>
<td>8.9 MiB</td>
<td>245 ms</td>
</tr>
</tbody>
</table>

For the small RTG of Example 5.7.4, the additional memory usage due to filtering is larger than the reduction in memory usage for the main tables. For the three Mono grammars, memory usage for each of the four filters is substantially less than without filtering. The construction time is also reduced drastically.

The results for the Mono grammars show that index filtering (IFILT) results in the smallest memory usage, at a substantially (two orders of magnitude) reduced construction time compared to not using filtering. If construction time is crucial, SFILT (symbol filtering) should be used, taking the least construction time (another order of magnitude less) for these grammars, while still using relatively little memory.

It is remarkable that the results clearly show that CFILT performs worse than IFILT and SFILT in construction time and memory usage, but the latter two were not mentioned or considered previously in the literature (i.e. by Chase [Cha87], Hemerik
Figure 8.6.1 Relative memory usage for DFRTA/match set tables, Example 5.7.4

Figure 8.6.2 Relative memory usage for DFRTA/match set tables, Intel X86
and Katoen [HK89, or Ferdinand et al. [FSW94]).

The causes for the larger memory usage when using Chase’s filter compared to the symbol or index filter were explained at the end of Sections 5.7.5.1 and 5.9: although Chase’s filter may reduce the number of entries in the main symbol tables much more, it will likely have a larger number of entries in the filter and representor tables, as it results in a filter and representor table for each valid symbol-index combination instead of just one for every symbol or one for every index in case of symbol SFILT and IFILT respectively.)

![Relative memory usage for DFRTA/match set tables, Intel IA64](image)

**Figure 8.6.3** Relative memory usage for DFRTA/match set tables, Intel IA64

The relative memory usage of each of the constructions compared to that of the construction using *Items* without filtering is given in Figures 8.6.1 through 8.6.4.

### 8.6.2 Pattern matching automata constructions

For experiments with automata constructions for pattern matching, three pattern sets were used:

- The small pattern set of Example 6.1.2, with an alphabet size of four.
- A pattern set obtained from rewrite rules from a specification of a rewrite system used with the ASF+SDF Meta Environment [BHJ+01]. This set had
an alphabet size of 34.

- A pattern set obtained from the Mono rtg for the Intel X86 family, used in the preceding section. The set was obtained by taking the right hand sides of non-chain rule productions and replacing every nonterminal occurrence by a variable occurrence. The alphabet size was 267.

The results for the basic DFRTA construction are shown in Table 8.4. For the three

<table>
<thead>
<tr>
<th>Pattern set</th>
<th></th>
<th>Entries in tables</th>
<th>Memory usage of tables</th>
<th>Construction time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 6.1.2</td>
<td>7</td>
<td>8</td>
<td>4.0 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td>ASF+SDF</td>
<td>11</td>
<td>11</td>
<td>14.0 KiB</td>
<td>3 ms</td>
</tr>
<tr>
<td>Intel X86</td>
<td>486</td>
<td>529</td>
<td>131.0 MiB</td>
<td>1472739 ms</td>
</tr>
</tbody>
</table>

pattern sets used, Table 8.5 shows that combined symbol & index filtering (CFILT) reduces the number of entries of tables $T_a$ the most, followed by symbol filtering (SFILT), index filtering (IFILT), and lastly subtree filtering (TFILT).
The memory usage for the tables for the Example 6.1.2 and ASF+SDF pattern sets is least for Tfilt, then IFilt, SFilt, and finally CFilt. For the Intel X86 pattern set, IFilt results in smaller memory usage than Tfilt.

For the small Example 6.1.2, total memory use when using a filter is larger than when not using filtering, i.e. filtering is not useful. For the pattern set based on an ASF+SDF rewrite system, memory usage when using some filters is somewhat smaller than when not using filters, but construction time is reduced by filtering.

For the large Mono Intel X86 pattern set, memory usage and construction time for each of the four filters is substantially less than without filtering, just as for large RTGs in the tree acceptance case. Filter IFilt performs best on memory use, but filter SFilt should be used if construction time is crucial. As in the previous section, Chase’s filter [Cha87] CFilt performs worse than (some) other filters in both memory usage and construction time.

<table>
<thead>
<tr>
<th>Pattern set</th>
<th>Filter</th>
<th>Entries in tables</th>
<th>Memory usage of tables</th>
<th>Construction time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 6.1.2</td>
<td>None</td>
<td>74</td>
<td>4.0 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>Tfilt</td>
<td>22</td>
<td>4.5 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>SFilt</td>
<td>14</td>
<td>6.2 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>IFilt</td>
<td>14</td>
<td>5.3 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>CFilt</td>
<td>9</td>
<td>7.2 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td>ASF+SDF</td>
<td>None</td>
<td>874</td>
<td>14.0 KiB</td>
<td>3 ms</td>
</tr>
<tr>
<td></td>
<td>Tfilt</td>
<td>55</td>
<td>7.1 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>SFilt</td>
<td>40</td>
<td>12.1 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>IFilt</td>
<td>55</td>
<td>8.0 KiB</td>
<td>&lt; 1 ms</td>
</tr>
<tr>
<td></td>
<td>CFilt</td>
<td>38</td>
<td>18.6 KiB</td>
<td>1 ms</td>
</tr>
<tr>
<td>Intel X86</td>
<td>None</td>
<td>22470891</td>
<td>131.0 MiB</td>
<td>1472739 ms</td>
</tr>
<tr>
<td></td>
<td>Tfilt</td>
<td>250719</td>
<td>1.9 MiB</td>
<td>17678 ms</td>
</tr>
<tr>
<td></td>
<td>SFilt</td>
<td>1208</td>
<td>4.3 MiB</td>
<td>280 ms</td>
</tr>
<tr>
<td></td>
<td>IFilt</td>
<td>109151</td>
<td>1.0 MiB</td>
<td>7326 ms</td>
</tr>
<tr>
<td></td>
<td>CFilt</td>
<td>639</td>
<td>9.5 MiB</td>
<td>328 ms</td>
</tr>
</tbody>
</table>

The Aho-Corasick automaton (ACA), stringpath Aho-Corasick variation (ACSPA), and the stringpath DRFTA (SPDRFTA) were also implemented and benchmarked using the same pattern sets. The quite large construction time for the stringpath DRFTAs can be explained by the implementation used: for these automata, the conceptual stepwise construction algorithm described in this dissertation was used, while for the Aho-Corasick automata, the more efficient implementation given by
Aho & Corasick [AC75] was used instead of the conceptual stepwise construction algorithms described here. It seems likely that a more efficient construction for the stringpath DRFTA is also possible, but this remains to be investigated.

### 8.6.3 Acceptance and pattern matching algorithms

Table 8.7 shows the running time for acceptance algorithms using the various automata, constructed based on the Mono project RTG for the Intel X86. The percentages in the third column are relative to the results for the NRFTA with PROPER-S item set, and are omitted for the εNRFTA, whose running time is far higher than that of any other automaton used. The results are based on running the algorithms using the automata on a set of 500 subject trees of approximately 150 nodes each. Benchmarking with a set of 150 subject trees of approximately 500 nodes each was also performed. This yielded similar results, except for the cases using (ε)NRFTAs, which had longer running times. This is due to the algorithm using such automata: it traverses subject tree parts multiple times, which takes more time for deeper subject trees.

Table 8.8 shows the running times for pattern matching algorithms, using a pattern set based on the productions of the Mono project RTG used above. The percentages in the third column are relative to the results for the NRFTA. These results are based on running the algorithms using the automata on a set of 150 subject trees of approximately 500 nodes each, in which a total of 89781 matches occurred.

Here, benchmarking with 500 trees of approximately 150 nodes each was also performed. Similarly to the tree acceptance case, the differences in running time between the respective cases were negligible, except for the cases using (ε)NRFTAs.
8.6 Experiments

Table 8.7 Acceptance running times

<table>
<thead>
<tr>
<th>Automaton type and item set</th>
<th>Time (ms)</th>
<th>Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNRFTA - ALL-SUB</td>
<td>3944300.80</td>
<td></td>
</tr>
<tr>
<td>NRFTA - ALL-SUB</td>
<td>91765.62</td>
<td>99.85</td>
</tr>
<tr>
<td>NRFTA - PROPER-N</td>
<td>91896.34</td>
<td>100.00</td>
</tr>
<tr>
<td>NRFTA - PROPER-S</td>
<td>91899.09</td>
<td>100.00</td>
</tr>
<tr>
<td>εNRFTA - ALL-SUB</td>
<td>2136.80</td>
<td>2.33</td>
</tr>
<tr>
<td>NFRTA - ALL-SUB</td>
<td>1824.64</td>
<td>1.99</td>
</tr>
<tr>
<td>NFRTA - PROPER-N</td>
<td>902.54</td>
<td>0.98</td>
</tr>
<tr>
<td>NFRTA - PROPER-S</td>
<td>881.66</td>
<td>0.96</td>
</tr>
<tr>
<td>DFRTA - ALL-SUB</td>
<td>117.48</td>
<td>0.13</td>
</tr>
<tr>
<td>DFRTA - PROPER-N</td>
<td>109.26</td>
<td>0.12</td>
</tr>
<tr>
<td>DFRTA - PROPER-S</td>
<td>112.09</td>
<td>0.12</td>
</tr>
<tr>
<td>DFRTA with TFILT - ALL-SUB</td>
<td>115.66</td>
<td>0.13</td>
</tr>
<tr>
<td>DFRTA with TFILT - PROPER-N</td>
<td>112.45</td>
<td>0.12</td>
</tr>
<tr>
<td>DFRTA with TFILT - PROPER-S</td>
<td>112.64</td>
<td>0.12</td>
</tr>
<tr>
<td>DFRTA with IFILT - ALL-SUB</td>
<td>115.73</td>
<td>0.13</td>
</tr>
<tr>
<td>DFRTA with IFILT - PROPER-N</td>
<td>113.41</td>
<td>0.12</td>
</tr>
<tr>
<td>DFRTA with IFILT - PROPER-S</td>
<td>113.94</td>
<td>0.12</td>
</tr>
<tr>
<td>DFRTA with SFILT - ALL-SUB</td>
<td>124.21</td>
<td>0.14</td>
</tr>
<tr>
<td>DFRTA with SFILT - PROPER-N</td>
<td>115.63</td>
<td>0.13</td>
</tr>
<tr>
<td>DFRTA with SFILT - PROPER-S</td>
<td>115.92</td>
<td>0.13</td>
</tr>
<tr>
<td>DFRTA with CFILT - ALL-SUB</td>
<td>134.29</td>
<td>0.15</td>
</tr>
<tr>
<td>DFRTA with CFILT - PROPER-N</td>
<td>117.54</td>
<td>0.13</td>
</tr>
<tr>
<td>DFRTA with CFILT - PROPER-S</td>
<td>118.90</td>
<td>0.13</td>
</tr>
</tbody>
</table>

A number of conclusions can be drawn from the experimental results as well as the results of the automata construction experiments mentioned earlier:

- The use of filtering should be considered carefully, as the relative memory usage and running time of tabulation algorithms with various filters vary depending on the RTG or pattern set.

- Furthermore, the most well-known filter, CFILT (described in [Cha87]), performs worse than newly described and simpler filters. The simplification of the filter function leads to smaller memory usage for the combined filter tables, even though the number of entries in the main tables is reduced furthest by Chase’s filter.

- The use of filtering in DFRTAS hardly influences running times, despite the extra level of indirection caused by the use of the filter tables.

- (ε)NRFTAS should not be used, since the running time of algorithms using them
Table 8.8 Pattern matching running times

<table>
<thead>
<tr>
<th>Automaton type</th>
<th>Time (ms)</th>
<th>Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NRFTA</td>
<td>72220.22</td>
<td>100.00</td>
</tr>
<tr>
<td>NFRTA</td>
<td>3020.01</td>
<td>4.18</td>
</tr>
<tr>
<td>DFRTA</td>
<td>1782.74</td>
<td>2.47</td>
</tr>
<tr>
<td>DFRTA with TFILT</td>
<td>1819.67</td>
<td>2.52</td>
</tr>
<tr>
<td>DFRTA with IFILT</td>
<td>1817.05</td>
<td>2.52</td>
</tr>
<tr>
<td>DFRTA with SFILT</td>
<td>1825.35</td>
<td>2.53</td>
</tr>
<tr>
<td>DFRTA with CFILT</td>
<td>1857.12</td>
<td>2.57</td>
</tr>
<tr>
<td>ACA</td>
<td>682.05</td>
<td>0.94</td>
</tr>
<tr>
<td>ACSPA</td>
<td>684.59</td>
<td>0.95</td>
</tr>
<tr>
<td>SPDRFTA</td>
<td>969.22</td>
<td>1.34</td>
</tr>
</tbody>
</table>

is longer than that for algorithms using the corresponding (ε)NFRTAs, while their memory consumption and construction time (not shown here) are the same.

- If memory use or construction time is the primary concern, (ε)NFRTAs should be used. (Memory use and construction time for such automata were omitted here, but can be found in [Str07a, Appendix E]).

- For tree pattern matching, ACAS should be used if matching speed is the primary concern. If memory use should be smaller, but not quite as small as that of the NFRTAs, one of the DFRTAs with filtering can be used. (Compare the memory use of the ACAS and DFRTAs given in Sections 6.6.3 and 6.7.2.1.)

- For tree acceptance, the above recommendation may hold as well, although ACAS for tree acceptance should be implemented and benchmarked to confirm this.

Additional benchmarking—with different RTGs and pattern sets, and with different subject tree sets—could be performed to see if these results hold for other cases as well.

### 8.7 Experiences and conclusions

In this chapter, we considered some aspects related to the Forest FIRE toolkit. In particular, we considered the toolkit’s high-level design, showing how the commonalities and variabilities between various algorithms, constructions, and related concepts guided and simplified the design. It is these commonalities and variabilities that are made explicit by the taxonomy presentation of the various algorithms and
8.7 Experiences and conclusions

constructions, in Chapters 5 and 6, and by the definitions of underlying concepts
given in the tree theory chapter, Chapter 3.

Furthermore, we showed how the presentations of the algorithms, constructions and
underlying concepts helped to implement them once the high-level design of the
toolkit had been created. The abstract algorithms were translated into Java code
in a straightforward way, as evidenced by two typical examples we presented in the
current chapter.

The resulting toolkit and accompanying graphical user interface FIRE WOOD were
used in various experiments reported on in this dissertation and in [Str07a]. Such
experiments gave insight in the working of algorithms and constructions, as well as
results on running times and memory use of the tree acceptance and tree pattern
matching algorithms and constructions.

The current toolkit and GUI are separated into two packages. The FOREST FIRE
package contains 82 classes with 8778 lines of code, while FIRE WOOD consists of
56 classes with 7363 lines of code. FOREST FIRE is compiled into a Java archive
file (forestfire.jar). FIRE WOOD is compiled into a Java archive file as well (fire-
wood.jar), using the FOREST FIRE and Standard Widget Toolkit .jar files as in-
cludes.

The toolkit and GUI could be further extended and further benchmarked. Variations
of included algorithms could be added, e.g. bitvector implementations of the
algorithms based on stringpath matching and of those based on match set/DFRTA
computation, as suggested in [HO82b, AGT89].

The toolkit can be used independent of the GUI. In particular, it can be used in other
(command line or graphical) applications, as long as they use the tree representation
used in the toolkit. As that representation is a rather straightforward one, it should
not be hard to convert to it from other representations. The toolkit has already
been applied to tree parsing, by extending some of the tree acceptance algorithms,
which took less than two hours [Str07b, Str07a].

The toolkit and GUI could also be extended to implement term rewriting, and ex-
periments could be performed to see how the various tree pattern matching algorithms
perform in this context. For such an extension, the file format for input/output
would need to be extended to allow specification of rewrite rules—consisting of a
left hand side pattern and a right hand side pattern—and algorithms to rewrite
a tree—using a pattern matching algorithm and updating the matches after each
rewrite step—would need to be implemented. Experiments could then be performed
related to the use of tree pattern matching in term rewriting systems, and in com-
bination with different rewriting strategies. The implicit assumption in the field of
term rewriting systems seems that use of frontier-to-root match set computation and
tabulation cannot be used efficiently (regardless of the rewriting strategy applied).
Experiments should be performed to determine whether this assumption is (still)
valid.
Part IV

Epilogue
Chapter 9

Conclusions

In this chapter, we turn to the problem statement and research questions formulated in Chapter 1, and consider the answers and contributions provided by this dissertation. We also present a list of suggested future work to extend the research reported on in the dissertation.

9.1 Contributions

In Chapter 1, we introduced tree acceptance and tree pattern matching as two important algorithmic problems from the field of regular tree languages. Although many algorithms solving these problems had appeared in the literature since the 1960s, a number of deficiencies existed related to these solutions:

1. Inaccessibility of the theory and algorithms.
2. Difficulty of comparing the algorithms.
3. Lack of reference to the theory and lack of correctness arguments.
4. Lack of a large and coherent collection of implementations of the algorithms.
5. Difficulty of choosing between different algorithms.

To solve these deficiencies, three research questions were formulated:

RQ1 How are the algorithms solving these algorithmic problems—found in the literature or as variations of those found in the literature—related, i.e. what are their commonalities and differences?

RQ2 How can the algorithms be presented together and in a common style such that their relations become clear and their correctness becomes apparent?
RQ3 How can the taxonomies—with the formal description of the tree algorithms, constructions and basic data structures and algorithms involved—be used in the design and implementation of a collection of implementations?

The first two questions relate mainly to the first three deficiencies. To answer these questions and solve these deficiencies, the use of taxonomies to classify the algorithms was proposed, following positive experiences with taxonomies in other areas, particularly in the related area of regular string language theory [Wat95].

For the last two of the five deficiencies, and to answer the related third research question, the construction of a taxonomy-based toolkit according to the TAXonomy-BAseD Software COstruction method (TABASCO) was suggested, following similar positive experiences [Wat95, Cle03].

We now present the main conclusions and contributions of this dissertation, and in doing so we indicate how they answer the above research questions.

- The two taxonomies cover many algorithms and automata constructions for tree acceptance and tree pattern matching, which appeared in the literature in a time span of about forty years.

- As for earlier taxonomies in [Wat95], the construction of the two taxonomies required a lot of time and effort to study the original papers describing them and to distill the essential details from the published algorithms.

- Two essential and powerful techniques played an important role: Abstraction from irrelevant details, and the description of more complex algorithms by sequentially adding details to a more abstract algorithm in a stepwise fashion. Many of the algorithms from the literature could be and were described more clearly by using such a stepwise presentation and by disregarding irrelevant details. Furthermore, the stepwise presentation makes the correctness of the algorithms more apparent, as the correctness of the individual details is easier to establish than the correctness of a complete algorithm as presented in the literature.

- The uniform presentation as part of the taxonomies improved the accessibility of algorithms from the literature, and in particular also showed their relations, thereby answering question [RQ1] and partially answering [RQ2]. Furthermore, it lead to new and rediscovered algorithms:
  - Two new filters for use in algorithms based on recursive match set computation/the use of DFRTAs were described: an index filter, IFILT, and a symbol filter, SFILT. Although conceptually straightforward, these simple filters were not described or considered in previous literature. Our experiments showed that they are nonetheless often practically relevant.
  - One filter, the subtree filter TFILT, was more or less rediscovered, as it was not mentioned in any literature apart from Turner’s paper in which it was originally described [Tur86].
9.1 Contributions

- Stringpath-based DRFTA pattern matching algorithms were described. Although variations were briefly discussed in [LW96], the presentation in this dissertation is more extensive, providing a stepwise description of the construction of the used stringpath DRFTAs, as well as a comparison of the automata to Aho-Corasick string automata. In fact, we showed that algorithms using the different kinds of automata can be seen as variations of a common algorithm skeleton in the tree pattern matching taxonomy.

- The taxonomy presentation of the algorithms makes the consultation of the original papers no longer necessary. Furthermore, comparing different algorithms to one another in the taxonomy is much simpler than comparing the versions from papers written in different styles and using different formalisms and notations.

- The toolkit, implementing many of the algorithms and constructions from the two taxonomies, was designed based on the taxonomies. It was shown that the high-level structure of the toolkit was influenced by the taxonomies, which brought out the commonalities and variations among the algorithms. This made the high-level design easier to create, while the uniformity of the algorithms’ presentation in the taxonomies made the algorithms easy to implement as well. Together, this answers question [RQ3].

- Although the availability of the taxonomies made the high-level toolkit design easy to create, the choice of representations for the basic data structures and algorithms underlying the taxonomy algorithms was quite open and took some time and effort to decide on. In particular, the experimental evaluation and optimization of representations and implementations took quite some time. This is particularly evident when compared to the taxonomy-based toolkits of regular string language algorithms of [Wat95]. For those toolkits, many of the basic data structures were already part of implementation languages, or were more easily implemented. The latter is also due to the relative abundance of implementation experience with such algorithms in applications, compared to the case of regular tree language algorithms: the former are used in many text processing tools, scanning and parsing toolkits, etc., while the latter are mainly used in a few code generation tools.

- Experiments with the toolkit gave some interesting results, including the following:

  - Although the basic DFRTA constructions are indeed not practical for larger grammars/pattern sets, due to table space and construction time, the use of filtering is quite effective: filtering reduces both the table space and the construction time drastically.

  - Furthermore, some simple filters, which were new or not mentioned in much of the literature, turned out to often be more efficient in memory usage and construction time than more complicated ones. Chase’s symbol
and index filter [Cha87] in particular is more complicated, but does not yield substantially better results. It is therefore rather remarkable that his filter is the one most described and used in the literature.

- As expected, the experiments on running time of tree acceptance and tree pattern matching algorithms show that the use of nondeterministic tree automata, especially root-to-frontier directed ones, is generally not to be recommended.

- Despite the additional level of indirection in table lookups, the use of filtering hardly influences the running time of the tree acceptance or tree pattern matching algorithms. Combined with the reduced table space and construction running time for filtering, this makes the use of filtering quite worthwhile, as already indicated by earlier work [Cha87].

- Nevertheless, the stringpath-based tree pattern matching algorithms using Aho-Corasick automata turned out to be fastest among the algorithms tested. As the memory use and construction time of these automata are similar to those for filtered DFRTAs, the former algorithms seem worthwhile to use in many cases.

- The results from the research reported in this dissertation are both theoretical and practical, ranging from formal definitions and algorithm taxonomies to a toolkit and experimental results. A form of symbiosis occurred between the theoretical and the practical: the taxonomies were helpful in constructing the toolkit, and the experimentation with various algorithms part of the toolkit in turn lead to a better understanding of the theoretical definitions and algorithm descriptions.

9.2 Future work

As future work, we see a number of extensions of the research reported on in this dissertation:

- Adding algorithms to the taxonomies of tree acceptance and tree pattern matching algorithms, including
  
  - bitvector implementations of the algorithms based on stringpath matching and of those based on match set/DFRTA computation, based on the suggestions in [HO82b, AGT89];
  
  - considering combinations of frontier-to-root matching with preprocessing during the descent to the frontier.

- Constructing a taxonomy of tree parsing algorithms, using the tree acceptance taxonomy as a starting point.
• Further benchmarking of the toolkit
  – with different pattern sets, grammars, and subject tree collections, and
  – with different data representations, particularly for DFRTA transitions.

• Extending the research (taxonomy, toolkit and domain specific language construction) to the case of directed acyclic graphs, particularly given the use of directed acyclic graphs (DAGs) instead of trees in instruction selection applications of (tree/DAG) parsing.

• Extending the research to other domains, including for example 2D pattern matching, graph pattern matching, and other domains, such as those for which taxonomies have already been constructed (as in Section 7.7).
References


269


of Computer Science and Automation, Technical University of Ilmenau, Germany, October 1998.


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.


References


References


References


References


Index

/, see tree, subtree at node
D, see tree domain
N, see nonterminal
Σ, see alphabet
Σ_n, see alphabet, rank n symbols
N_{≤r}, 25
Π, see tuple, flattening
⇒, see derivation step
∪, see quantification
ο, see composition
∃, see quantification
∀, see quantification
\mathcal{L}_{RTG}, see regular tree grammar, language
\mathcal{L}_{TA}, see tree automaton, language
\mid, see left drop
\mid, see left take
π, see tuple, projection
\mathcal{P}, see powerset
\lor, see right drop
\downarrow, see string projection
\downarrow, see right take
r_{max}, see rank, maximum
\tau_a, see tree product
ε, see string, empty, see tree automaton, epsilon transition
n, see rank, unspecified symbol
\|, see tree, size
*, see string concatenation

abstract base class, 238
ACA, see Aho-Corasick automaton
Accept, see tree automaton, root-to-frontier acceptance function
ACSPA, see Aho-Corasick automaton, stringpath

Aho, A.V., 121, 149, 256
Aho-Corasick algorithm, 78–80, 220
Aho-Corasick automaton, 90, 149, 194, 196, 197, 200, 202, 203, 241, 244, 255, 258
deterministic, 194
optimal, 194
stringpath, 194, 197–199, 206, 207, 255
alphabet, 14
rank n symbols, 25
ranked, 24, 245
application engineering, 214
arity, see rank
ASF+SDF Meta-Environment, 253

Bal, H.E., 26
BEG, 237
benchmarking, 224, 226, 249–258
bijection, 125, 136, 179, 184
Boake, A., 213, 229
Boyer-Moore algorithm, 78, 220
Brainerd, W.S., 113
Broy, M., 229
BURG, 237
Börstler, J., 131

C++, 219, 222, 238, 241, 243
template metaprogramming, 225
Cai, J., 182, 186
CFilt_{a,i}, see match set, filtering, CFilt_{a,i}
chain rule, see unit production
chain rule closure, see closure, nonterminal
Chase, D., 132, 134, 141, 186, 251, 255, 257

281
Christopher, T.W., 237

Cl, see closure
closure, 117, 123, 125, 127, 176, 178–180
nonterminal, 15, 123, 124, 127, 245
tabulation, 124
reflexive, 49, 124
transitive, 15, 49, 124, 245
code generation, see instruction selection
Commentz-Walter algorithm, 78, 220
Comp_n, see item, composition with alphabet symbol
composition, 13
concatenation, 23, see also tree language, concatenation
configuration knowledge, 216, 226
construction, 8
deterministic, 113, 174
SPM-TA:ALL-DOTTED:RF, 199
SPM-TA:ALL-DOTTED:RF:RA-LOOPS, 202
SPM-TA:ALL-DOTTED:RF:RA-LOOPS:-SUBSET_RF, 203
TGA-TA:ALL-SUB, 97–101, 120
TGA-TA:ALL-SUB:FR, 111, 120
TGA-TA:ALL-SUB:REM-ε, 102
TGA-TA:ALL-SUB:REM-ε:FR, 112
TGA-TA:ALL-SUB:REM-ε:FR:-SUBSET_FR, 114, 121, 122
TGA-TA:ALL-SUB:REM-ε:RF, 108
TGA-TA:ALL-SUB:RF, 107
TGA-TA:PROPER-N:REM-ε, 106
TGA-TA:PROPER-N:REM-ε:RF, 109
TGA-TA:PROPER-S:REM-ε, 104
TPM-TA:ALL-SUB, 168–172, 178
TPM-TA:ALL-SUB:FR, 173
TPM-TA:ALL-SUB:FR:SUBSET_FR, 174, 178
TPM-TA:ALL-SUB:RF, 172
tree acceptor, 95
tree pattern matcher, 167
context-free string matcher, 167
context-free string grammar, 42, 72
context-free string language, 72
derivation tree, 72
derivation free tree grammar, 37
derivation free tree language, 23
Coplen, J.O., 219, 238, 240
Corasick, M.J., 256
Czarnecki, K., 214, 225

Darlington, J., 229
derivation, 38
derivation step, 38
detail, 7, 8, 76
ACA-SPM, 196
algorithm detail, 76
ALL-SUB, 95, 97, 130, 168
CFILT, 136, 250, 255, 257
DET, 92, 95, 121, 166, 190
DRFTA-SPM, 205
FILTER, 136, 183
FR, 93, 96, 110, 121, 164, 167, 173
IFILT, 136, 250, 255
MATCH-SET, 89, 116, 119, 121, 161, 175, 177
performance detail, 76
problem detail, 76
PROPER-N, 95, 106, 130
PROPER-S, 95, 104, 130
RA-LOOPS, 202
REC, 120, 121, 177
REM-ε, 96, 102
RF, 91, 96, 107, 163, 167, 172
S-PATH, 162
SFILT, 136, 250, 255
SP-MATCHER, 189
SUBSET_FR, see subset construction, frontier-to-root
SUBSET_RFR, see subset construction, root-to-frontier
T-ACCEPTOR, 88, 95, 101, 121
T-MATCHER, 160, 171
TABULATE, 125, 179
TFILT, 136, 250, 255
Index

TGA-ta, 96
Tpm-ta, 167
DFRTA, see tree automaton, deterministic frontier-to-root
Dinther, Y. van, 107, 110
domain, 6, 217, 218
deficiencies, 217
domain analysis, 214
domain design, 214, 215, 219
domain engineering, 214, 233
domain implementation, 214, 216, 224
domain model, 214, 217
domain modeling, 214
domain scoping, 214
domain specific description, 224–228
domain specific language, 6, 7, 215, 217, 218, 224–229
evolution, 228
domain specific language expression, see domain specific description
domain specific processor, 224–228
dotted tree, see tree, dotted
DRFTA, see tree automaton, deterministic root-to-frontier
DSL, see domain specific language

Eclipse, 235
dotted tree, see tree, dotted
Eisencker, U.W., 214, 225
εNFRTA, see tree automaton, ε-nondeeterministic frontier-to-root
εNRFTA, see tree automaton, ε-nondeeterministic root-to-frontier

FASTAR, 230
feature model, 80
Ferdinand, C., 103, 116, 131, 172–174, 253
Filtα,i, see match set, filtering, Filtα,i
filter, see match set, filtering
filtering, see match set, filtering
finite tree automaton, see tree automaton

FIRE wood graphical user interface, 234, 259
FOREST FIRE toolkit, 233–248, 259
FUEL tool, 226–228, 231
function overloading, 240

Ganapathi, M., 149
generative programming, 216
generic class, 238, 241, 243, 246
GenVoca, 225
grammar characteristic, 39
non-root nodes, 40
unit productions, 40
grammar transformation, see transformation
graphical user interface, 7, 234, 259
Grune, D., 26
GUI, see graphical user interface

Hatcher, P.J., 237
Hemerik, C., 114, 251
Hoffmann, C.M., 90, 174, 182, 195, 196, 206

iBURG, 237
IFilti, see match set, filtering, IFilti
inheritance, 220, 238, 243
instruction selection, 3, 4, 6, 130, 249
interface, 238, 243, 246
composition with alphabet symbol, 117, 123, 125, 176, 178
item set
reachability, 125, 179
item set provider, 243
Items', see item
Items, see item
Jacobs, C.J.H., 26
Java, 235
Jonkers, H.B.M., 229
Katoen, J.P., 114, 253
keyword pattern matcher, 160, 164
keyword pattern matching, 7, 78–79, 90, 160, 162, 194
Knuth-Morris-Pratt algorithm, 78–80, 220
Kourie, D.G., 213, 229
Kron, H.H., 131, 174, 182

Lam, M., 121
Langendoen, K.G., 26
language closure, 35
language exponentiation, 35
leaf, see node, leaf
leaf node, see node, leaf
left drop, 14–15
left take, 14–15
LHS, see left hand side
Li, M., 206
little language, see domain specific language
LR parsing, 133

Marcelis, A.J.J.M., 229
Match, see tree pattern, match
match set, 89, 90, 101, 116–123, 137, 161, 162, 172, 175–178, 184
direct computation, 120, 178
filtering, 131–145, 182–186, 238, 241, 242, 253, 257
CFilt_{a,i}, 134, 141, 142, 144, 185
Filt_{a,i}, 132, 183
Ifilt_{a,i}, 134, 144
reachability-based tabulation, 138
reachable, 136, 183
SFilt_{a,i}, 134, 143
tabulation, 136, 183
Tfilt, 134, 142
reachability-based tabulation, 124–131, 179–182
MAX, see quantification
Meerakker, H.J.A. van de, 149, 197
Mono project, 249
MP, see tree pattern matching
MS, see match set

MSP, see stringpath matching
Möncke, U., 131

NCI, see closure, nonterminal
NFRTA, see tree automaton, nondeterministic frontier-to-root
node, 24, 245
leaf, 24, 27, 54, 58
nonterminal, 37, 101, 106, 124
fresh, 45–48
superfluous, 46–48
nonterminal closure, see closure, nonterminal
NRFTA, see tree automaton, nondeterministic root-to-frontier

O’Donnell, M.J., 90, 174, 182, 195, 196, 206
Obiedkov, S., 213, 229
OTA, see tree automaton, orthogonal
Output, see output function
output function, 161, 164, 167, 190, 191, 195, 196, 205, 206
Aho-Corasick automaton, 198
deterministic frontier-to-root automaton, 174
deterministic root-to-frontier tree automaton, 199, 204, 205
nondeterministic root-to-frontier tree automaton, 201, 207
optimal Aho-Corasick automaton, 195
trie, 198, 207
output function, deterministic frontier-to-root automaton, 174
Ouwens, T.J., 226

Paige, R., 182, 186
PathCl, see tree language, path closure
pattern, see tree pattern
pattern set, see tree pattern set
pattern stringpath, 187–208
pattern tree, see tree pattern powerset, 12
Index

pref, see prefix
prefix, 15, 23, 195
Prime Computer, 133
problem domain, see domain
production, 4, 37, 97, 117, 131, 245
  left hand side, 38, 97, 101, 117
  right hand side, 38, 97, 101, 103,
  104, 106, 117, 131
production rule, see production
Productive, see productive symbol
productive, see symbol, productive
ProperSubtrees, see tree, proper sub-tree
quantification, 12
rank, 24, 245
  maximum, 25
  unspecified symbol, 25
ranked alphabet, see alphabet, ranked
ranking function, see rank
RAReachM, see tree automaton state,
root accepting state-reachable
reachability, 18–20
  tree automaton state, 18, 114, 174
RED-U, see transformation step, removing
single unit production
RED-Z, see transformation step, reduce
number of Z-nodes
RED-Z-NAIVE, see transformation step,
reduce number of Z-nodes
reflexive closure, see closure, reflexive
regular language, see string language,
regular
regular string grammar, see string gram-
mar, regular
regular string language, see string lan-
guage, regular
regular tree expression, 37
regular tree grammar, 4, 6, 37, 86–87,
  245
  ambiguous, 39
  augmented, 87, 116
  deterministic, 110
  equivalent, 39
  language, 39
regular tree language, 3, 4, 22, 36, 71
  augmented, 95
  closure properties, 71
  decision problems, 71
  equivalence of formalisms, 71
  relations with string languages, 71
  yield, 72
regular tree system, 36
Reisig, W., 56
REM-ε, see transformation, removing
ε-transitions
REM-UNRε, see transformation, remov-
ing root accepting state-unreach-
able states and transitions
REM-USELESS, see transformation, remov-
ing useless productions and
symbols
representer set, 132, 137, 184
RHS, see production, right hand side
right drop, 14–15
right take, 14–15
rootpath, 27, 201
RPath, see rootpath
RSt, see tree state assignment, root of
RTG, see regular tree grammar
rule, see production
Seidl, H., 103, 116, 131, 172–174, 253
Set, see quantification
Sethi, R., 121
SFilt, see match set, filtering, SFilt
simple pattern forest, see tree pattern
set, simple
sorting, 81, 225
  domain specific language, 225
toolkit, 225
SPARE FUEL domain specific language,
  226–228
SPARE Parts toolkit, 222
SPARE Time toolkit, 219, 222, 224–
  228
design, 220
SPaths, see stringpath
Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMatch</td>
<td>see stringpath, match</td>
</tr>
<tr>
<td>SReach</td>
<td>see start-reachable symbol</td>
</tr>
<tr>
<td>St</td>
<td>see tree state assignment</td>
</tr>
<tr>
<td>Standard Widget Toolkit</td>
<td>235, 259</td>
</tr>
<tr>
<td>start symbol</td>
<td>87, 245</td>
</tr>
<tr>
<td>STF</td>
<td>see Z-node, selection, shortest tree first</td>
</tr>
<tr>
<td>Strategy design pattern</td>
<td>221, 238, 241</td>
</tr>
<tr>
<td>string</td>
<td>14</td>
</tr>
<tr>
<td>empty</td>
<td>14</td>
</tr>
<tr>
<td>pattern matching, see keyword pattern matching</td>
<td></td>
</tr>
<tr>
<td>string acceptor</td>
<td>59, 88, 120, 164</td>
</tr>
<tr>
<td>string automaton</td>
<td>6, 88, 149, 160, 190, 194</td>
</tr>
<tr>
<td>construction</td>
<td>133</td>
</tr>
<tr>
<td>deterministic</td>
<td>121, 194, 196</td>
</tr>
<tr>
<td>on-the-fly construction</td>
<td>121</td>
</tr>
<tr>
<td>nondeterministic</td>
<td>120, 121</td>
</tr>
<tr>
<td>simulation</td>
<td>120</td>
</tr>
<tr>
<td>string concatenation</td>
<td>14</td>
</tr>
<tr>
<td>string grammar</td>
<td>14</td>
</tr>
<tr>
<td>LL(1) condition</td>
<td>110</td>
</tr>
<tr>
<td>regular</td>
<td>38</td>
</tr>
<tr>
<td>string language</td>
<td>14</td>
</tr>
<tr>
<td>regular</td>
<td>21</td>
</tr>
<tr>
<td>acceptance</td>
<td>88</td>
</tr>
<tr>
<td>string projection</td>
<td>15</td>
</tr>
<tr>
<td>string tree</td>
<td>59, 72</td>
</tr>
<tr>
<td>stringpath</td>
<td>27, 32, 67–69, 72, 90, 162, 187–208</td>
</tr>
<tr>
<td>stringpath match</td>
<td>27, 32, 67–69, 72, 90, 162, 187–208</td>
</tr>
<tr>
<td>beginpoint</td>
<td>146, 192, 193</td>
</tr>
<tr>
<td>detection</td>
<td>146, 192, 196, 206</td>
</tr>
<tr>
<td>endpoint</td>
<td>146, 192</td>
</tr>
<tr>
<td>registration</td>
<td>146, 192, 193, 196, 206</td>
</tr>
<tr>
<td>stringpath matching</td>
<td>32–33, 187–208, 241</td>
</tr>
<tr>
<td>Stroblenberg, R.G.W.,</td>
<td>233</td>
</tr>
<tr>
<td>subset construction</td>
<td></td>
</tr>
<tr>
<td>frontier-to-root</td>
<td>69–70, 96, 114, 167, 174</td>
</tr>
<tr>
<td>root-to-frontier</td>
<td>66–69, 96, 167, 203</td>
</tr>
<tr>
<td>SUBSETfn</td>
<td>see subset construction, frontier-to-root</td>
</tr>
<tr>
<td>SUBSETrp</td>
<td>see subset construction, root-to-frontier</td>
</tr>
<tr>
<td>substitution</td>
<td>see tree substitution</td>
</tr>
<tr>
<td>Subtrees</td>
<td>see tree, subtree</td>
</tr>
<tr>
<td>suffix</td>
<td>see suffix</td>
</tr>
<tr>
<td>suffix</td>
<td>15</td>
</tr>
<tr>
<td>symbol</td>
<td>14, 245</td>
</tr>
<tr>
<td>productive</td>
<td>41–44</td>
</tr>
<tr>
<td>start-reachable</td>
<td>41–44</td>
</tr>
<tr>
<td>TABASCO</td>
<td>5–9, 213–219, 233</td>
</tr>
<tr>
<td>process steps</td>
<td>217</td>
</tr>
<tr>
<td>Tarjan, R.</td>
<td>182, 186</td>
</tr>
<tr>
<td>taxonomy</td>
<td>5–8, 75–78, 216, 217, 223</td>
</tr>
<tr>
<td>construction advantages</td>
<td>81</td>
</tr>
<tr>
<td>evolution</td>
<td>228</td>
</tr>
<tr>
<td>tree acceptance</td>
<td>83, 224</td>
</tr>
<tr>
<td>tree pattern matching</td>
<td>153, 224</td>
</tr>
<tr>
<td>versus feature model</td>
<td>80</td>
</tr>
<tr>
<td>taxonomy graph</td>
<td>76</td>
</tr>
<tr>
<td>template class</td>
<td>221, 243</td>
</tr>
<tr>
<td>Template Method design pattern</td>
<td>221, 242</td>
</tr>
<tr>
<td>term</td>
<td></td>
</tr>
<tr>
<td>ordered, ranked</td>
<td>see tree, ordered, ranked</td>
</tr>
<tr>
<td>term rewriting</td>
<td>4, 237, 259</td>
</tr>
<tr>
<td>terminal</td>
<td>37</td>
</tr>
<tr>
<td>TFilter</td>
<td>see match set, filtering, TFilter</td>
</tr>
<tr>
<td>TGA</td>
<td>see tree acceptance</td>
</tr>
<tr>
<td>Timbuk</td>
<td>237</td>
</tr>
<tr>
<td>Tjiang, S.W.K.,</td>
<td>149</td>
</tr>
<tr>
<td>TL</td>
<td>see tree language</td>
</tr>
<tr>
<td>toolkit</td>
<td>5, 7, 217–220, 224, 226, 259</td>
</tr>
<tr>
<td>design goals</td>
<td>220, 223, 234</td>
</tr>
<tr>
<td>evolution</td>
<td>228</td>
</tr>
<tr>
<td>implementation</td>
<td>222</td>
</tr>
<tr>
<td>toolkit design</td>
<td></td>
</tr>
<tr>
<td>taxonomy-based</td>
<td>223, 234, 237–248</td>
</tr>
<tr>
<td>advantages</td>
<td>223</td>
</tr>
</tbody>
</table>
TPM, see tree pattern matching
TPMn, see tree pattern matching
TPMn, see tree pattern matching
transformation
  reducing number of Z-nodes, 245
  removing \( \varepsilon \)-transitions, 64–65, 102
  frontier-to-frontier tree automaton, 65
  root-to-frontier tree automaton, 64
  removing root accepting state-unreachable states and transitions, 63, 103
  removing useless productions and symbols, 43, 245
transformation step
  reduce number of Z-nodes, 45–47
  removing single unit production, 48, 245, 247
transitive closure, see closure, transitive
tree, 24, 245
  dotted, 26, 200
  ordered, 25
  ordered, ranked, 3, 4, 24, 25
  ordered, unranked, 4, 24
  proper subtree, 27, 104, 131, 132
  ranked, 25
  size, 24
  subtree, 132, 134
  subtree at node, 26
tree acceptance, 4, 9, 58, 87, 235, 256, 258
tree acceptor, see tree automaton
tree adjoining grammar, 26
tree automaton, 6, 53–70, 194, 239, 243
  acceptance, see also tree state assignment, 56–61
deterministic frontier-to-root, 55, 63, 95, 121, 125, 130, 166, 178, 181, 257
deterministic root-to-frontier, 55, 63, 65, 92, 110, 164, 194, 200
stringpath, 199, 203, 204, 206, 207, 241, 244, 255
\( \varepsilon \)-free, 54, 55
\( \varepsilon \)-nondeterministic frontier-to-root, 54, 63, 93, 120, 241, 258
\( \varepsilon \)-nondeterministic root-to-frontier, 54, 63, 90, 241, 257
\( \varepsilon \)-transition, 54, 96, 101, 117
\( \varepsilon \)-transition relation, 54
frontier-to-root, 93, 164
generator class, 239, 240, 244
language, 58
minimization, 130
nondeterministic frontier-to-root, 54, 63, 94, 120, 164, 241, 258
simulation, 120, 178
nondeterministic root-to-frontier, 54, 63, 65, 92, 163, 200, 206, 241, 257
  pattern matching, 199
  pattern recognition, 199, 202, 203
orthogonal, 174, 182
root accepting state, 54, 200
root-to-frontier, 90
  acceptance, 60, 91, 163
state
  root accepting state-reachable, 61–63, 67, 103, 106
  root accepting state-reachable, 203
  unreachable, 61
  useless, 61–63, 67, 203
state assignment, see tree state assignment
state diagram notation, 55–56
state set, 54, 95
subset construction, see subset construction
transition relation, 54
useful state, 61
visualization, 55
tree concatenation, 28–30
tree domain, 23–25
element, see node
ordered, 25

tree language, 33
can concatenation, 34–35
context-free, see context-free tree language
path closed, 36
path closure, 36, 67
substitution, 34–35
tree parsing, 131, 132, 237, 246, 259
tree pattern, 31–33, 156, 176
dotted, 200, 202
independent, 182
instance, 31
match, 31
tree pattern matcher, see tree automaton
tree pattern matching, 4, 6, 9, 31–33, 159, 160, 235, 256, 258, 259
tree pattern set, 4, 6, 156, 167, 182
simple, 182
tree product, see tree concatenation
tree state assignment, 56–61, 98, 101, 170, 172, 201
accepting, 56
root of, 58–61, 94, 101, 120, 121, 165, 172
tree substitution, 27–30, see also tree language, substitution
all occurrences of a single subtree, 28
concurrent substitution at multiple leaf symbols, see tree concatenation
occurrences of leaf symbol by different subtrees, 28–30
single subtree occurrence, 28
tree transducer, 23

TREEBAG, 237
trie, 195, 198, 200, 203, 206

TTF, see Z-node, selection, tallest tree first
tuple
flattening, 14

projection, 14

Turner, P.K., 132–134
TWIG, 237

U, see grammar characteristic, unit productions

Ullman, J., 121
unit production, 40, 44–53, 245
useful production, 42
useful symbol, 41
useless nonterminal, 87
useless production, 42–53, 87
useless symbol, 41–53

Viragh, J., 110
virtual function, 220, 243

Vosloo, I., 213, 229

Warshall’s algorithm, 15–17, 49, 124, 245

Watson, B.W., 78, 213, 229

Weisgerber, B., 149

Wilhelm, R., 103, 116, 131, 149, 172–174, 253

Wood, D., 206

z, see grammar characteristic, non-root nodes

Z-node, 40, 44–53
selection, 47
nondeterministic, 47
shortest tree first, 47
tallest tree first, 47

Zwaan, G., 78
Summary

Tree Algorithms
*Two Taxonomies and a Toolkit*

The subject of this dissertation is the construction of two taxonomies and a toolkit of algorithms solving two different but closely related problems from the domain of regular tree languages.

In our context, a taxonomy is a classification of algorithms according to their essential details. Its starting point is a high-level algorithm whose correctness is easily shown. By adding details, one obtains refinements or variations of that algorithm and of algorithms obtained in such a way. This may lead to algorithms from the literature or to new ones. Every detail has correctness arguments associated with it, so that each algorithm’s correctness follows from the correctness of the starting point and details added. The sequence of details from the starting point to an algorithm shows how the algorithm can be obtained from the starting point and can be used to characterize the particular algorithm. An algorithm taxonomy may be depicted as a directed acyclic graph, in which nodes refer to algorithms and branches refer to details.

We construct taxonomies for two algorithmic problems from the domain of regular tree languages on ordered, ranked trees. This domain has a rich theory, and parts of this theory have broad applicability in areas such as code generation in compilers—particularly for instruction selection or optimization—and term rewriting. Underlying the practical applications are a number of algorithmic problems, of which tree acceptance and tree pattern matching are two. Many algorithms solving these problems have been described and used in practice. Unfortunately, the domain suffers from a number of deficiencies:

1. Inaccessibility of the theory and algorithms, as they are scattered over the literature and few or no overview publications—particularly algorithm oriented ones—exist.

2. Difficulty of comparing the algorithms due to differences in presentation style

289
and level of formality.


4. Lack of a large and coherent collection of implementations of the algorithms.

5. Difficulty of choosing between different algorithms for practical applications.

The construction of taxonomies aims to solve the first three deficiencies. The construction process is part of TABASCO, a method for domain modeling and domain engineering for a particular kind of domain in computer science. As a first step, an overview of relevant parts of the theory is created. A literature survey is then performed to gather algorithms solving the algorithmic problem or problems. Based on the results, the algorithms are rephrased in a common presentation style and a model in the form of an algorithm taxonomy for the algorithmic problem or problems is constructed. The taxonomy makes the commonalities and variations between the algorithms as well as the correctness arguments for the algorithms explicit.

The problems of tree acceptance and tree pattern matching are related and the algorithms solving them involve many of the same algorithmic ingredients. The commonalities lead to similarities between the taxonomies constructed for each of the two problems. Furthermore, references to and discussions of the algorithms’ occurrences in the literature are included.

The last two of the five deficiencies mentioned above can be solved using TABASCO’s domain engineering steps, which include the creation of a toolkit of algorithm implementations and of a domain specific language to allow more effective use of such a toolkit. The design of a coherent toolkit is simplified by the availability of a taxonomy, which indicates the commonalities and differences between the various algorithms. The high-level design choices are guided by the taxonomy structure, and the choice of which language constructs to use to implement various design parts can be made based on standard design techniques.

We consider the domain of keyword pattern matching as a brief case study for all of the method’s domain engineering steps. For the domain of regular tree languages, we consider the design of a toolkit of tree acceptance and tree pattern matching algorithms based on the two taxonomies. The taxonomy-based toolkit design results in a coherent and easily extendable toolkit.
Samenvatting

Boomalgoritmen
Twee Taxonomieën en een Toolkit

Dit proefschrift behandelt de constructie van twee taxonomieën en een toolkit van algoritmen voor twee verschillende maar nauw verwante problemen uit het domein van reguliere boombalen.

In onze context is een taxonomie een classificatie van algoritmen op basis van hun essentiële details. Het startpunt is een abstract algoritme waarvan de correctheid eenvoudig aan te tonen is. Door details toe te voegen verkrijgt men verfijningen of variaties van dat algoritme en van aldus verkregen algoritmen. Dit kan leiden tot uit de literatuur bekende of tot nieuwe algoritmen. Met elk detail worden correctheidsovertuigingen geassocieerd, zodat de correctheid van elk algoritme volgt uit de correctheid van het startpunt en de toegevoegde details. De reeks van details vanaf het startpunt naar een algoritme geeft aan hoe dat algoritme uit het startpunt verkregen kan worden en kan gebruikt worden om het algoritme te karakteriseren. Een taxonomie van algoritmen kan afgebeeld worden als een gericht acyclische graaf, waarin knopen verwijzen naar algoritmen en takken naar details.

We construeren taxonomieën voor twee algoritmische problemen uit het domein van reguliere boombalen op geordende bomen met rang. Dit domein kent een rijke theorie en delen van deze theorie zijn breed toepasbaar in gebieden zoals codegeneratie in compilers—in het bijzonder voor instructie-selectie of optimalisatie—en termherschrijving.

Een aantal algoritmische problemen, waarvan boomacceptatie en boompatroonherkennings de belangrijkste zijn, liggen ten grondslag aan de praktische toepassingen. Veel algoritmen die deze problemen oplossen zijn beschreven en in de praktijk gebruikt. Helaas lijdt het domein onder een aantal tekortkomingen:

1. Theorie en algoritmen zijn ontvankelijk, omdat deze verspreid zijn over de literatuur en er weinig of geen—in het bijzonder op algoritmen gerichte—overzichtspublicaties zijn.
2. De algoritmen zijn moeilijk te vergelijken door verschillen in presentatiestijl en in mate van formaliteit.

3. Referenties aan de theorie en aan correctheidsargumenten ontbreken in publicaties over de praktische algoritmen.

4. Een grote en samenhangende verzameling van implementaties van de algoritmen ontbreekt.

5. Het is moeilijk te kiezen tussen verschillende algoritmen voor praktische toepassingen.

De constructie van taxonomieën tracht de eerste drie tekortkomingen op te lossen. Het constructieproces maakt deel uit van TABASCO, een methode voor domain modeling en domain engineering voor een bepaald soort domein binnen de informatica. Als eerste stap wordt een overzicht van relevante delen van de theorie samengesteld. Vervolgens wordt een literatuuronderzoek gedaan om algoritmen te verzamelen die het algemene probleem of de algemene problemen oplossen. Op basis van de resultaten worden de algoritmen opnieuw uitgedrukt in een gemeenschappelijke presentatiestijl en wordt een model in de vorm van een taxonomie van algoritmen geconstrueerd voor het algemene probleem of de algemene problemen. De taxonomie maakt de overeenkomsten en verschillen tussen de algoritmen alsmede de correctheidsargumenten voor de algoritmen expliciet.

De problemen van boomacceptatie en -patroonherkenning zijn gerelateerd en de algoritmen die ze oplossen bevatten veel gelijke ingrediënten. De gemeenschappelijkheden leiden tot overeenkomsten tussen de taxonomieën die voor elk van de twee problemen geconstrueerd werden. Verder worden er verwijzingen naar en discussies van voorkomens van de algoritmen in de literatuur opgenomen.

De laatste twee van bovengenoemde vijf tekortkomingen kunnen opgelost worden door gebruik te maken van TABASCO’s domain engineering-stappen. Deze omvatten het creëren van een toolkit van algoritme-implementaties en van een domein-specifieke taal om effectiever gebruik te kunnen maken van zo’n toolkit. Het ontwerpen van zo’n toolkit wordt makkelijker gemaakt door de beschikbare taxonomie, die de overeenkomsten en verschillen tussen de diverse algoritmen aangeeft. Ontwerpkeuzes op hoog niveau worden gestuurd door de structuur van de taxonomie en taalconstructies voor het implementeren van delen van het ontwerp kunnen gekozen worden op basis van standaard-ontwerptechneken.

We beschouwen het domein van tekstpatroonherkenning als een korte case study van al deze stappen van TABASCO. Voor het domein van reguliere boontalen beschouwen we het ontwerp van een toolkit van algoritmen voor boomacceptatie en -patroonherkenning op basis van de twee taxonomieën. Het ontwerp op basis van de taxonomieën leveren een coherente en eenvoudig uitbreidbare toolkit op.
Curriculum Vitae

I was born on 19 September 1979 in Roermond. From 1991 until 1997 I attended the Collegium Mariamum secondary school (VWO / Gymnasium-β) in Venlo.

I enrolled to study computer science at the Eindhoven University of Technology starting in September 1997, receiving a scholarship sponsored by KPN, Océ Technologies and the van Doorne family.

From February 2001 until February 2002, as part of my study program, I performed an (extended) internship at a company called ZNOW in Berkeley, California, USA, working on their search technology.


I then joined the Software Construction group as a PhD student. This group became the Software Engineering & Technology group in January 2006. From September 2003 until December 2007, I worked in these groups on research related to taxonomies and toolkits of tree algorithms, leading to this dissertation.

As of February 2008, I am working as a researcher (onderzoeker) in the Software Engineering & Technology group.
**Titles in the IPA Dissertation Series since 2002**

**M.C. van Wezel.** *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01

**V. Bos and J.J.T. Kleijn.** *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

**T. Kuipers.** *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

**S.P. Luttik.** *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

**R.J. Willemen.** *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05

**M.I.A. Stoelinga.** *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

**N. van Vugt.** *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07

**A. Fehnker.** *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

**R. van Stee.** *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09

**D. Tauritz.** *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10

**M.B. van der Zwaag.** *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

**J.I. den Hartog.** *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

**L. Moonen.** *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

**J.I. van Hemert.** *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining.* Faculty of Mathematics and Natural Sciences, UL. 2002-14

**S. Andova.** *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15

**Y.S. Usenko.** *Linearization in μCRL.* Faculty of Mathematics and Computer Science, TU/e. 2002-16

**J.J.D. Aerts.** *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01

**M. de Jonge.** *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty
of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02


S.M. Bohte. *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04


S.V. Nedea. *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06


H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08


M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components.* Faculty of Mathematics and Natural Sciences, UL. 2003-10


G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving.* Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems.* Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation.* Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05


L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary
Investigation of Fundamentals, Strategies, and Business Applications. Faculty of Technology Management, TU/e. 2004-08

N. Goga. Control and Selection Techniques for the Automated Testing of Reactive Systems. Faculty of Mathematics and Computer Science, TU/e. 2004-09


J. Pang. Formal Verification of Distributed Systems. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. Evolutionary Agent-Based Economics. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. Indoor Ultrasonic Position Estimation Using a Single Base Station. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. On Distributed Verification and Verified Distribution. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17


P.J.L. Cuijpers. Hybrid Process Algebra. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. Supervisory Machine Control by Predictive-Reactive Scheduling. Faculty of Mechanical Engineering, TU/e. 2004-21


R. Ruimerman. Modeling and Remodeling in Bone Tissue. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. Experiments in Rights Control - Expression and Enforcement. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. Design and Verification of Lock-free Parallel Algorithms. Faculty of Mathematics and Computing Sciences, RUG. 2005-04
H.M.A. van Beek. Specification and Analysis of Internet Applications. Faculty of Mathematics and Computer Science, TU/e. 2005-05


I. Kurtev. Adaptability of Model Transformations. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. Computational Aspects of Treewidth - Lower Bounds and Network Reliability. Faculty of Science, UU. 2005-09

O. Tveretina. Decision Procedures for Equality Logic with Uninterpreted Functions. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. Evolution of Finite Populations in Dynamic Environments. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. Data Mining using Genetic Programming: Classification and Symbolic Regression. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. Top Quality Type Error Messages. Faculty of Science, UU. 2005-13

G.F. Frehse. Compositional Verification of Hybrid Systems using Simulation Relations. Faculty of Science, Mathematics and Computer Science, RU. 2005-14


T. Gelsema. Effective Models for the Structure of pi-Calculus Processes with Replication. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. Composing Constraint Solvers. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18


M. Valero Espada. Modal Abstraction and Replication of Processes with Data. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. Stepping through Haskell. Faculty of Science, UU. 2005-21

Y.W. Law. Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense. Faculty of Electrical Engineering,
E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01


P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03


M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijsen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14


M.E. Warnier. *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18
L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19


J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03


I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10


R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12


C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML. Fac-
ulty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. Component-based Configuration, Integration and Delivery. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-15

B.S. Graaf. Model-Driven Evolution of Software Architectures. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijsen. Logical Calculi for Reasoning with Binding. Faculty of Mathematics and Computer Science, TU/e. 2007-17


W. Pieters. La Volonté Machinale: Understanding the Electronic Voting Controversy. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. Practical Automation Proofs in PVS. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. Renovation of Idiomatic Crosscutting Concerns in Embedded Systems. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03


N.C.W.M. Braspenning. Model-based Integration and Testing of High-tech Multi-disciplinary Systems. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates. Faculty of Science, UU. 2008-06


I.S.M. de Jong. Integration and Test Strategies for Complex Manufacturing Machines. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. Tracing Anonymity with Coalgebras. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Clephas. Tree Algorithms: Two Taxonomies and a Toolkit. Faculty of Mathematics and Computer Science, TU/e. 2008-10