FOUNDATIONS OF

XML Processing

The Tree-Automata Approach

HARUO HOSOYA

CAMBRIDGE

CAMBRIDGE www.cambridge.org/9780521196130

This page intentionally left blank

FOUNDATIONS OF XML PROCESSING

The Tree-Automata Approach

This is the first book to provide a solid theoretical account of the foundation of the popular data format XML.

Part I establishes basic concepts, starting with schemas, tree automata, and pattern matching, and concluding with static typechecking for XML as a highlight of the book. In Part II, the author turns his attention to more advanced topics, including efficient "on-the-fly" tree automata algorithms, path- and logic-based queries, tree transformation, and exact typechecking. Many examples of code fragments are given, and exercises are provided to enhance understanding. Thus the book should be very useful both for students and for XML researchers.

HARUO HOSOYA is a Lecturer in the Department of Computer Science at the University of Tokyo.

FOUNDATIONS OF XML PROCESSING

The Tree-Automata Approach

HARUO HOSOYA
University of Tokyo

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore,
São Paulo, Delhi, Dubai, Tokyo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521196130

First published in print format 2010

ISBN-13 978-0-511-90402-8 eBook (Adobe Reader)

ISBN-13 978-0-521-19613-0 Hardback

To the memory of my beloved mother

Contents

Preface

Computer science, like other mathematical fields, cannot live without a tight relationship with reality. However, such a relationship is, frankly, not very common. This is probably why people so enthusiastically welcome a true meeting of theory and practice. In that sense, the coming together of XML and tree automata theory was a beautiful marriage. Thus I have written this book in the earnest hope that the news of this marriage will be spread and celebrated all over the world!

The book is a summary of my ten years' work. It could not have been realized without my collaborators, Peter Buneman, Giuseppe Castagna, Alain Frisch, Vladimir Gapeyev, Kazuhiro Inaba, Shinya Kawanaka, Hiromasa Kido, Michael Y. Levin, Sebastian Maneth, Makoto Murata, Benjamin C. Pierce, Tadahiro Suda, Jérôme Vouillon, Takeshi Yashiro, and Philip Wadler. In particular, I thank Kazuhiro Inaba and Sebastian Maneth, who made uncountable comments on the draft and thus contributed to a huge improvement of the book. Lastly, I thank my dear wife, Ayako, who gave me the warmest and unceasing encouragement to finish the book.

1

Introduction

1.1 Documents, schemas, and schema languages

The data format known as extensible mark-up language (XML) describes tree structures based on mark-up texts. The tree structures are formed by inserting, between text fragments, open and end tags that are balanced, like parentheses. A data set thus obtained is often called a *document*. On the surface, XML resembles hypertext mark-up language (HTML), the most popular display format for the Web. The essential difference, however, is that in XML the structure permitted to documents, including the set of tag names and their usage conventions, is not fixed *a priori*.

More precisely, XML allows users to define their own *schemas*; a schema determines the permitted structure of a document. In this sense, it is often said that a schema defines a "subset of XML" and thus XML is a "format for data formats." With the support of schemas each individual application can define its own data format, while virtually all applications can share generic software tools for manipulating XML documents. This genericity is a prominent strength of XML in comparison with other existing formats. Indeed, XML has been adopted with unprecedented speed and range: an enormous number of XML schemas have been defined and used in practice. To raise a few examples, extensible HTML (XHTML) is the XML version of HTML, simple object access protocol (SOAP) is an XML message format for remote procedure calls, scalable vector graphics (SVG) is a vector graphics format in XML, and MathML is an XML format for mathematical formulas.

One naturally asks: what constraints can schemas describe? The answer is: such constraints are defined by a *schema language*. In other words, there is not one schema language but many, each having different constraint mechanisms and thus different expressiveness. To give only a few of the most important, document type definition (DTD), XML Schema, as defined by the World Wide Web Consortium

(W3C), and regular language description for XML (RELAX NG), defined by the Organization for the Advancement of Structured Information Standards (OASIS) and the International Standards Organization (ISO), are actively used in various applications. This lack of a single schema language certainly confuses application programmers, since they need to decide which schema language to choose, and it also troubles tool implementers since they need to support multiple schema languages. However, this situation can also be seen as a natural consequence of XML's strength; such genericity and usability had never been provided by any previous format and thus it has attracted a huge number of developers, though their requirements have turned out to be vastly different.

1.2 Brief history

The predecessor of XML was the standard generalized markup language (SGML). It was officially defined in 1986 (by ISO) but had been used unofficially since the 1960s. The range of users had been rather limited, however, mainly because of the complexity of its specification. Nonetheless it was in this period that several important data formats such as HTML and DocBook (which are now revised as XML formats) were invented for SGML.

In 1998 XML was standardized by W3C. The creators of XML made a drastic simplification of SGML, dropping a number of features that had made the use of SGML difficult. The tremendous success of XML was due to its simplicity, together with its timely fit to the high demand for a standard, non-proprietary, data format.

At first DTD was adopted as a standard schema language for XML. This was a direct adaptation of the version of DTD used for SGML, made in view of the compatibility of XML and SGML. A number of software tools for SGML were already available, and therefore exploiting these was highly desirable at that time. However, the lack of certain kinds of expressiveness that were critical for some applications had by then already been recognized.

Motivated by a new, highly expressive, schema language, the standardization activity for XML Schema started in around 1998. However, it turned out to be an extremely difficult task. One of the biggest difficulties was that the committee was attempting to mix two completely different notions: regular expressions and object orientation. Regular expressions, the most popular notation for string patterns, was the more traditional notion and had been used in DTD since the era of SGML. The reason for using this notion was "internal" since XML (and SGML) documents are based on *texts*, using regular expressions is a very natural way to represent constraints on such ordered data. However, the demand for object orientation was "external." It arose from the coincident popularity of the Java programming language. Java had a rich library support for network programming, and therefore

developers naturally wanted an object serialization format for the exchange of this data between applications on the Internet. These two concepts, one based on automata theory and the other based on a hierarchical model, cannot be integrated in a smooth manner. The result after four years' efforts was inevitably a complex gigantic specification (finalized in 2001). Nonetheless, a number of software programmers nowadays attempt to cope with this complexity.

In parallel with XML Schema, there were several other efforts towards the standardization of expressive schema languages. Among others, the aim with RELAX was to yield a simple and clean schema language in the same tradition as DTD but based on the more expressive regular tree languages rather than the more conventional regular string languages. The design was so simple that the standardization went very rapidly (it was released by ISO in 2000) and became a candidate for a quick substitute for DTD. Later, a refinement was made and released as RELAX NG (by OASIS and ISO in 2001). Although these schema languages are not yet widely used in comparison with DTD or XML Schema, the number of users exhibits a gradual increase.

Meanwhile, what has happened in the academic community? After the emergence of XML, at first most academicians were skeptical since XML appeared to be merely a trivial tree structure, which they thought they completely understood. Soon, however, researchers began to notice that this area was full of treasures. In particular, the notion of schemas, which looked like the ordinary *types* found in traditional programming languages, in fact had a very different mathematical structure. This meant that, in order to redo what had been done for conventional types, such as *static typechecking*, a completely new development was needed. Since then a huge amount of research has been undertaken. Scientists now agree that the most central concept relevant to XML's schemas is *tree automata*. The aim of this book is to introduce the theory and practice arising from this direction of research on the foundations of XML processing, that is, the tree-automata approach.

1.3 Overview of the book

Here we summarize the topics covered in this book and explain how they are organized into parts and chapters.

1.3.1 Topics

Schemas and tree automata

With regard to schemas, we first need to discuss the constraint mechanisms that should be provided by a schema and how these can be checked algorithmically.

In Chapter 3 we define an idealized schema language called the *schema model* and, using this, we compare three important schema languages, namely, DTD, XML Schema, and RELAX NG. Then, in Chapter 4, we introduce *tree automata*, which provide a finite acceptor model for trees that has an exact correspondence with schemas. Using the rich mathematical properties of tree automata, we can not only efficiently check the validity of a given document with respect to a given schema but also solve other important problems related to schemas, such as static typechecking.

Chapters 3 and 4 exemplify a pattern of organization used repeatedly in this book. That is, for each topic, we usually first define a specification language that is helpful for the user and then give a corresponding automata formalism that is useful for the implementer. In these two chapters, we introduce schema, primarily for document constraints, and then tree automata for validation and other algorithms. Specification and algorithmics are usually interrelated, since as schemas become more expressive, the algorithmics becomes more difficult. The chapter organization is intended to help the reader to appreciate such trade-offs.

The last paragraph might suggest that if efficiency is important then we should restrict expressiveness. However, if there is a way to overcome inefficiency then a design choice that allows for full expressiveness becomes sensible. In Chapter 8 we present one such case. This chapter gives a series of efficient algorithms for important problems related to tree automata, where all the algorithms are designed in a single paradigm, the *on-the-fly* technique. This technique has been extremely successful in achieving efficiency without compromising expressiveness. The core idea lies in the observation that we can often obtain a final result by exploring only a small part of the entire state space of an automaton.

We will also cover several other advanced techniques related to schemas. In Chapter 9 we will extend schemas with *intersection types* and their corresponding *alternating tree automata*. The latter are not only useful for algorithmics but also enable a clear formalization of certain theoretical analyses presented in later chapters. In Chapter 14 we discuss the *ambiguity* properties of schemas and automata. Such notions are often valuable in practical systems designed to discover typical mistakes made by users. In Chapter 15 we turn our attention to schema mechanisms for the description of *unordered* document structures. This chapter is unique in the sense that all the prior chapters treat ordered data. Unorderedness arises in certain important parts of XML documents, and in certain important kinds of application, and therefore cannot be ignored. However, treating it is technically rather tricky since it does not fit well into the framework of tree automata. Chapter 15 gives an overview of some recent attempts to solve the problem.

Subtree extraction

Once we know how to constrain documents, our next interest is in how to process them, when the most important aim is to extract information from an input XML tree. In Chapter 5 we introduce the notion of *patterns*, which are a straightforward extension of the schema model that allows *variable bindings* to be associated with subtrees. Corresponding to patterns are *marking tree automata*, introduced in Chapter 6. These automata not only accept a tree but also put marks on some of its internal nodes. In these chapters we will discuss various design choices involved in pattern matching and marking tree automata.

As alternatives to patterns, we present *path expressions* and *logic-based approaches*. Path expressions, discussed in Chapter 12, are a specification for the navigation required to reach a target node and are widely used in the notation called *XPath* (standardized by the World Wide Web Consortium, W3C). In the same chapter we define a framework for the corresponding automata, called *tree-walking automata*, which use finite states to navigate in a given tree. We will compare their expressiveness with that of normal tree automata. Chapter 13 gives a logic-based approach to subtree extraction; here we introduce *first-order (predicate) logic* and its extension, *monadic second-order (MSO) logic*, and explain how these can be useful for the concise specification of subtree extraction. Then, in the same chapter we detail the relationship between MSO logic and tree automata, thus linking this chapter to the rest of the book.

Tree transformation and typechecking

In subtree extraction we consider the analysis and decomposition of an input tree; tree transformation combines this with the production of an output tree. In this book we will not go into the details of complex tree transformation languages but, rather, will present several very small transformation languages with different expressivenesses. The reason is that our purpose is to introduce the *typechecking* technique, the book's highlight.

Typechecking is adopted by many popular programming languages in order to statically analyze a given program and to guarantee that certain kinds of error never occur. Since schemas for XML are just like types in such programming languages, we might imagine that there could be a similar analysis for XML transformations. However, vastly different techniques are needed for this since schemas are based on regular expressions (see Section 2.1), which are not standard in the usual programming languages. This book presents a particularly successful approach based on tree automata.

In XML typechecking there are two very different methodologies, namely, *exact typechecking* and *approximate typechecking*. Exact typechecking is the ideal

static analyzer; it signals an error if and only if a given program is incorrect. However, such a typechecker cannot deal with a *general* transformation language, that is, one as expressive as Turing machines. This is a direct consequence of the established notion that the behavior of a Turing machine cannot be predicted precisely. Therefore an exact typechecker is necessarily targeted to a *restricted* transformation language. Approximate typechecking, however, has no such limitation since it can give a false-negative answer, that is, it may reject some correct programs. The question is how can we design a "type system" that yields a reasonable set of accepted programs while ensuring tractability? In Chapter 7, we define a small but general transformation language called μXDuce – a subset of XDuce, the first practical XML processing language with static typechecking – and describe an approximate typechecking algorithm for this language. Later in Chapter 10, we introduce a family of *tree transducers*, which are finite-state machine models for tree transformations. These models are sufficiently restricted to perform exact typechecking, and Chapter 11 describes one such algorithm. This algorithm treats only the simplest case, that of *top-down tree transducers*, but even so it includes important ideas in exact typechecking such as *backward inference*.

1.3.2 Organization

Immediately following the introduction is a chapter giving the mathematical preliminaries including regular expressions and string automata. The remaining chapters are divided into two parts. Part I focuses on basic topics that need to be understood by all who are interested in the tree-automata approach to XML processing. This part starts with basic concepts, namely, schemas (Chapter 3), tree automata (Chapter 4), patterns (Chapter 5), and marking automata (Chapter 6). After this, we introduce the XML processing language, μXDuce, and a typechecking algorithm for it (Chapter 7), integrating the preceding basics.

Part II features more advanced topics and collects various notions that are often used in frontier research articles in this area. Therefore this part would be suitable reading for researchers who need a quick introduction to the background or starting points for new contributions. The topics presented are on-the-fly algorithms (Chapter 8), intersection types and alternating tree automata (Chapter 9), tree transducers (Chapter 10), exact typechecking (Chapter 11), path expressions and tree-walking automata (Chapter 12), logic-based queries (Chapter 13), ambiguity (Chapter 14), and unorderedness (Chapter 15). Figure 1.1 depicts the logical dependencies between the chapters in the two parts.

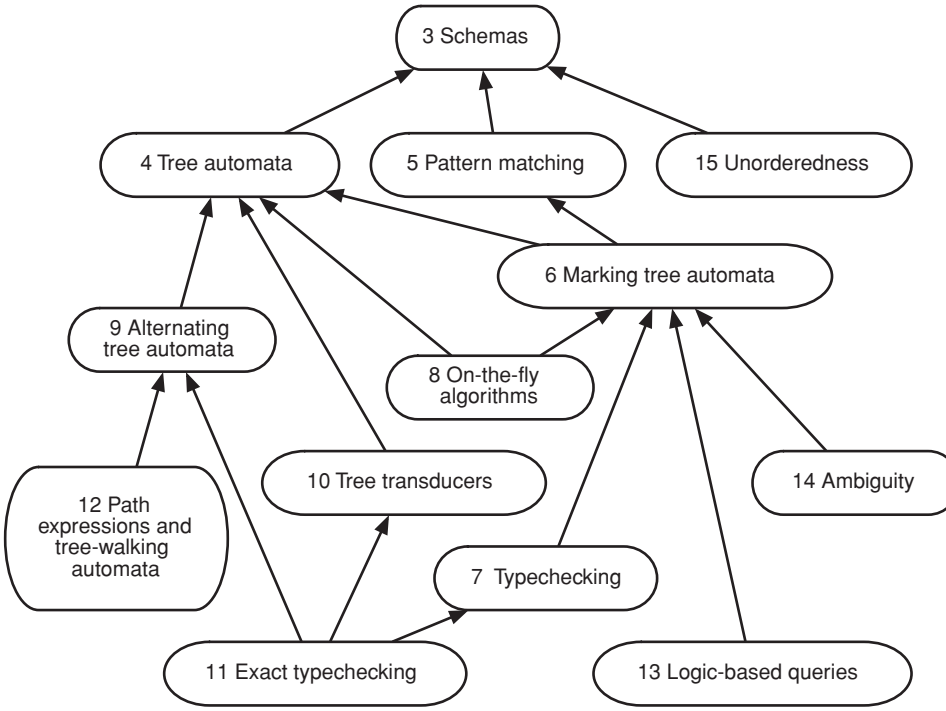


Figure 1.1 Chapter dependencies.

In order to assist understanding, exercises are given in most chapters. Each exercise is marked ⋆ (easy), ⋆⋆ (intermediate), or ⋆⋆⋆ (hard). The reader is encouraged to do all the exercises. Solutions to many of them are given in an appendix at the end of the book.

In an example or an exercise, we sometimes associate its number with concrete instances defined there, such as trees or automata. For example, in Example 4.1.1 we define a tree $t_{4.1.1}$. Such instances may be used several times in later examples, and the reader can use the number to turn back and refer to the original definitions.

Understanding this book requires an adequate background in elementary computer science, including basic set theory, algorithms, and data structures, complexity theory, and formal language theory. In particular, formal language theory is the basis of all the theories described in this book and this is why the summary in Chapter 2 is provided. Readers not familiar with this area are encouraged to study an introductory text first. For this, *Introduction to Automata Theory, Languages, and Computation* (Hopcroft and Ullman, 1979), is recommended. In addition, some background in programming language theory and in XML technologies would help, though these topics are not prerequisites; excellent references are *Types and*

Programming Languages (Pierce, 2002) and *An Introduction to XML and Web Technologies* (Møller and Schwartzbach, 2006). Also, readers who wish to refer to the actual specifications of various standards should look at Bray *et al.* (2000) for XML and DTD, Fallside (2001) for XML Schema, Clark and Murata (2001) for RELAX NG (Murata (2001b) for its predecessor RELAX), and Sperberg-McQueen and Burnard (1994) for a good introduction to SGML.

2

Preliminaries

This chapter introduces some notions used throughout the book, including the basic theory of regular expressions and finite string automata as well as the notational conventions of grammars and inference rules. Readers already familiar with them can skip this chapter.

2.1 Regular expressions

In this book, we often consider sequences of various kinds. For a given set $\mathcal{S}$, we write a *sequence* of elements from $\mathcal{S}$ by simply listing them (sometimes separated with commas); in particular, we write the sequence of length 0 by ϵ and call it the *empty sequence*. The *concatenation* of two sequences s and t is written st (or s, t when commas are used as separators); when a sequence s can be written as tu, we say that t and u are a *prefix* and a *suffix* of s, respectively. The length of a sequence s is written as $|s|$.

The set of all sequences of elements from $\mathcal{S}$ is denoted $\mathcal{S}^*$. For example, $\{a, b\}^*$ contains the sequences $\epsilon, a, aa, ab, abab, \ldots$ When a strict total order $<$ is defined on $\mathcal{S}$, the *lexicographic order* $\preceq$ on $\mathcal{S}^*$ can be defined: $s \preceq t$ if either $t = ss'$ for some s' (i.e., s is a prefix of t) or else $s = uas'$ and $t = ubt'$ for some sequences u, s', t' and some elements a, b with $a < b$ (i.e., s and t have a common prefix u, immediately after which t has a strictly larger element). For example, for sequences in $\{1, 2\}^*$ (with the order $1 < 2$) we have $11 \preceq 112$ and $112 \preceq 121$. Note that $\preceq$ is reflexive since we always have $s = ss'$ with $s' = \epsilon$ (i.e., s is a trivial prefix of s itself).

Let us assume a finite set Σ of *labels* $a, b, \ldots$ We call a sequence from Σ^* a *string*. The set of *regular expressions* over Σ, ranged over by r, is defined by the

following grammar:

$$\begin{array}{rcl} r & ::= & \epsilon \\ & & a \\ & & r_1 r_2 \\ & & r_1 \mid r_2 \\ & & r^* \end{array}$$

That is, ϵ and any element a of Σ are regular expressions; when r_1 and r_2 are regular expressions, so are $r_1 r_2$ and $r_1 \mid r_2$; when r is a regular expression, so is r^*. For example, we have regular expressions over $\Sigma = \{a, b\}$ such as $(a \mid \epsilon)b$ and $(ab)^*$.

Since this is the first time we are using the grammar notation, let us explain the general convention. A *grammar* of the form

$$\begin{array}{rcl} A & ::= & f_1[A_1, \ldots, A_n] \\ & & \vdots \\ & & f_k[A_1, \ldots, A_n] \end{array}$$

(the form $f[A_1, \ldots, A_n]$ is an expression containing metavariables $A_1, \ldots, A_n$ as subexpressions) inductively defines a set $\mathcal{S}$, ranged over by the metavariable A, such that, if $A_1, \ldots, A_n$ are all in the set $\mathcal{S}$ then $f_i[A_1, \ldots, A_n]$ is also in the set $\mathcal{S}$ for each $i = 1, \ldots, k$. Also, we sometimes use a mutually defined grammar in the form

$$\begin{array}{rcl} A & ::= & f_1[A_1, \ldots, A_n, B_1, \ldots, B_m] \\ & & \vdots \\ & & f_k[A_1, \ldots, A_n, B_1, \ldots, B_m] \\ \\ B & ::= & g_1[A_1, \ldots, A_n, B_1, \ldots, B_m] \\ & & \vdots \\ & & g_l[A_1, \ldots, A_n, B_1, \ldots, B_m] \end{array}$$

which inductively defines two sets $\mathcal{S}$ and $\mathcal{T}$, each ranged over by A and B, such that if $A_1, \ldots, A_n$ are all in the set $\mathcal{S}$ and $B_1, \ldots, B_m$ are all in the set $\mathcal{T}$ then each $f_i[A_1, \ldots, A_n, B_1, \ldots, B_m]$ is also in the set $\mathcal{S}$ and each $g_j[A_1, \ldots, A_n, B_1, \ldots, B_m]$ is also in the set $\mathcal{T}$. The notation can be generalized to an arbitrary number of sets.

Returning to regular expressions over Σ, their *semantics* is usually defined by interpreting a regular expression as a set of strings from Σ^*. Formally, the

language $\mathcal{L}(r)$ of a regular expression r is defined by

$$\begin{array}{lcl}
\mathcal{L}(\epsilon) & = & \{\epsilon\} \\
\mathcal{L}(a) & = & \{a\} \\
\mathcal{L}(r_1 r_2) & = & \{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1),\ s_2 \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1 \mid r_2) & = & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r^*) & = & \{s_1 \cdots s_n \mid s_1, \ldots, s_n \in \mathcal{L}(r),\ n \geq 0\}.
\end{array}$$

For example, $\mathcal{L}((a \mid \epsilon)b) = \{ab, b\}$ and $\mathcal{L}((ab)^*) = \{\epsilon, ab, abab, ababab, \ldots\}$. Alternatively, we can give the semantics of regular expressions by defining a relation between a string s and a regular expression r such that s belongs to r. Formally, let us define a *conformance* relation, written as $s\ in\ r$, by the following set of inference rules:

$$\frac{}{\epsilon\ in\ \epsilon}\ \textsc{Eps}$$

$$\frac{}{a\ in\ a}\ \textsc{Sym}$$

$$\frac{s\ in\ r_1}{s\ in\ r_1 \mid r_2}\ \textsc{Alt1}$$

$$\frac{s\ in\ r_2}{s\ in\ r_1 \mid r_2}\ \textsc{Alt2}$$

$$\frac{s_1\ in\ r_1 \qquad s_2\ in\ r_2}{s_1 s_2\ in\ r_1 r_2}\ \textsc{Cat}$$

$$\frac{s_i\ in\ r \quad \forall i = 1, \ldots, n \qquad n \geq 0}{s_1 \cdots s_n\ in\ r^*}\ \textsc{Rep}$$

That is, the rule Eps declares that the relation $\epsilon\ in\ \epsilon$ holds unconditionally (note that the first ϵ is a string and the second is a regular expression); similarly, the rule Sym declares that the relation $a\ in\ a$ holds unconditionally for all $a \in \Sigma$. The rule Alt1 declares that, for any string s and any regular expressions r_1 and r_2, the relation $s\ in\ r_1 \mid r_2$ holds if the relation $s\ in\ r_1$ holds; the rule Alt2 is similar. The rule Cat declares that, for any strings s_1 and s_2 and any regular expressions r_1 and r_2, the relation $s_1 s_2\ in\ r_1 r_2$ holds if the two relations $s_1\ in\ r_1$ and $s_2\ in\ r_2$ both hold. Finally, the rule Rep declares that, for any strings $s_1, \ldots, s_n$ and any regular expression r, where $n \geq 0$, the relation $s_1 \cdots s_n\ in\ r^*$ holds if all the relations $s_1\ in\ r$, $s_2\ in\ r$, ..., and $s_n\ in\ r$ hold; in particular, $\epsilon\ in\ r^*$ holds unconditionally ($n = 0$).

Since inference rules are used above for the first time, let us describe the general convention. An *inference rule* has the following form:

$$\frac{P_1(A_1, \ldots, A_n) \qquad \cdots \qquad P_k(A_1, \ldots, A_n)}{P(A_1, \ldots, A_n)}\ \textsc{Rule-Name}$$

(the form $P(A_1, \ldots, A_n)$ is a predicate with metavariables $A_1, \ldots, A_n$ as arguments), in which the predicates above the horizontal line are called *premises* and the one below the line is called the *conclusion*. Such a rule declares that, for any instantiations of the metavariables (as elements of the represented set), if all the premises hold then the conclusion holds.

After defining a certain relation by a set of inference rules, we often want to check whether the relation holds for particular instances. For this, we usually prove the relation by applying one of the rules; in this rule the relations in the premises are proved by some other rules, and so on. As a result, we have a proof in a tree structure where each node is some instantiation of a rule; we call this a *derivation*. For example, the following is a derivation of the relation $bab\ in\ ((a \mid \epsilon)b)^*$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon\ in\ \epsilon}\ \textsc{Eps}}{\epsilon\ in\ a \mid \epsilon}\ \textsc{Alt2} \qquad \dfrac{}{b\ in\ b}\ \textsc{Sym}}{b\ in\ (a \mid \epsilon)b}\ \textsc{Cat} \qquad \dfrac{\dfrac{\dfrac{}{a\ in\ a}\ \textsc{Sym}}{a\ in\ a \mid \epsilon}\ \textsc{Alt1} \qquad \dfrac{}{b\ in\ b}\ \textsc{Sym}}{ab\ in\ (a \mid \epsilon)b}\ \textsc{Cat}}{bab\ in\ ((a \mid \epsilon)b)^*}\ \textsc{Rep}$$

2.2 String automata

A *string automaton* on Σ is a quadruple $A = (Q, I, F, \delta)$, where

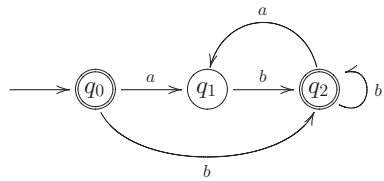
- Q is a finite set of *states*,
- I is a set of *initial states* ($I \subseteq Q$),
- F is a set of *final states* ($F \subseteq Q$), and
- δ is a set of *transition rules* of the form $q \xrightarrow{a} q'$ where $q, q' \in Q$ and $a \in \Sigma$.

The semantics of the string automaton is defined in terms of runs. That is, a *run* on a string $s = a_1 \cdots a_{n-1}$ is a sequence $q_1 \cdots q_n$ of states from Q such that $q_1 \in I$ and $q_i \xrightarrow{a_i} q_{i+1} \in \delta$ for each $i = 1, \ldots, n-1$. Such a run *accepts* s when $q_n \in F$. In this case, we say that the automaton accepts the string. Intuitively, the automaton starts with an initial state and successively consumes a label from the string while following a transition having that label; the automaton accepts the string if it arrives at a final state in the end. The set of strings accepted by a string automaton A is called the *language* of A and written as $\mathcal{L}(A)$. The class of languages each accepted by some string automaton is called the class of *regular string languages*.

For example, consider the string automaton $(\{q_0, q_1, q_2\}, \{q_0\}, \{q_0, q_2\}, \delta)$, where δ contains the following transitions:

$$q_0 \xrightarrow{a} q_1 \qquad q_1 \xrightarrow{b} q_2 \qquad q_0 \xrightarrow{b} q_2 \qquad q_2 \xrightarrow{a} q_1 \qquad q_2 \xrightarrow{b} q_2$$

This automaton can be represented visually as follows:



Here, a non-final state is shown by a single circle, a final state by a double circle, an initial state by a circle with an incoming arrow with no origin, and a transition by a labeled arrow connecting two circles. The above automaton accepts the string bab, for example, since there is a run $q_0q_2q_1q_2$ accepting it. More generally, this automaton accepts any string composed of a's and b's in which each a is followed by a b. Indeed, we will show later that the accepted language can be represented by the regular expression $((a \mid \epsilon)b)^*$.

We can extend the last definition of string automata so that they can *silently* (i.e., without observable effect) transit among states. A string automaton with ϵ-*transitions* is a quadruple $A = (Q, I, F, \delta)$ where Q, I, and F are similar to a normal string automaton and δ is a set of transition rules of the form $q \xrightarrow{l} q'$, where $q, q' \in Q$ and $l \in \Sigma \cup \{\epsilon\}$. The semantics can be defined exactly in the same way except that an ϵ-transition does not consume a label from the input string.

It is known that, for any string automaton with ϵ-transitions, there is a string automaton without ϵ-transitions that accepts the same language. To show this, we first define the following function for a given string automaton $A = (Q, I, F, \delta)$ with ϵ-transitions:

$$\epsilon\text{-}\mathbf{closure}(q_1) = \{q_k \mid q_1 \xrightarrow{\epsilon} q_2, \ldots, q_{k-1} \xrightarrow{\epsilon} q_k \in \delta,\ k \geq 1\}$$

That is, ϵ-**closure**(q_1) contains all the states q_k that can be reached from q_1 by following zero or more ϵ-transitions. Using it we can construct a string automaton $A' = (Q, I, F', \delta')$ without ϵ-transitions where

$$\begin{aligned} F' &= \{q \mid q \in Q,\ q' \in \epsilon\text{-}\mathbf{closure}(q),\ q' \in F\} \\ \delta' &= \{q \xrightarrow{a} q' \mid q \in Q,\ q'' \in \epsilon\text{-}\mathbf{closure}(q),\ q'' \xrightarrow{a} q' \in \delta\}. \end{aligned}$$

That is, whenever a final state q' is reachable from a state q by zero or more ϵ-transitions, we regard the state q as final. Also, whenever a state q' is reachable from a state q by zero or more ϵ-transitions followed by a single a-transition,

we add an a-transition from q to q'. Finally, we remove all ϵ-transitions. The constructed string automaton can be shown to accept the same language as the original automaton; the proof is omitted here.

It is also known that for any regular expression there is a string automaton accepting the same language. We present here a well-known construction called the *McNaughton–Yamada construction*, since it will be used several times in the rest of the book. In this construction we define a function $\mathcal{C}(r)$ that returns an automaton with ϵ-transitions for a given regular expression r. The function is defined inductively, on the structure of r, by the set of inference rules given below. First, for simple regular expressions ϵ and a, we produce trivial automata:

$$\frac{q : \text{new}}{\mathcal{C}(\epsilon) = (\{q\}, \{q\}, \{q\}, \emptyset)}$$

$$\frac{q_1, q_2 : \text{new}}{\mathcal{C}(a) = (\{q_1, q_2\}, \{q_1\}, \{q_2\}, \{q_1 \xrightarrow{a} q_2\})}$$

Here, q, q_1, and q_2 are newly introduced states for these automata. For a choice $r_1 \mid r_2$, we use the following rule:

$$\frac{\mathcal{C}(r_1) = (Q_1, \{q_1\}, \{q_1'\}, \delta_1) \qquad \mathcal{C}(r_2) = (Q_2, \{q_2\}, \{q_2'\}, \delta_2) \qquad q, q' : \text{new}}{\mathcal{C}(r_1 \mid r_2) = (Q_1 \cup Q_2 \cup \{q, q'\}, \{q\}, \{q'\}, \delta_1 \cup \delta_2 \cup \{q \xrightarrow{\epsilon} q_1, q \xrightarrow{\epsilon} q_2, q_1' \xrightarrow{\epsilon} q', q_2' \xrightarrow{\epsilon} q'\})}$$

That is, we first construct automata for the subexpressions r_1 and r_2; next we add a new initial state q that silently transits to the old initial states q_1 and q_2; and then add a new final state q' to which the old final states q_1' and q_2' transit silently. The following rule is for a concatenation $r_1 r_2$:

$$\frac{\mathcal{C}(r_1) = (Q_1, \{q_1\}, \{q_1'\}, \delta_1) \qquad \mathcal{C}(r_2) = (Q_2, \{q_2\}, \{q_2'\}, \delta_2)}{\mathcal{C}(r_1 r_2) = (Q_1 \cup Q_2, \{q_1\}, \{q_2'\}, \delta_1 \cup \delta_2 \cup \{q_1' \xrightarrow{\epsilon} q_2\})}$$

That is, we first construct automata for the subexpressions r_1 and r_2: we adopt the initial state q_1 of r_1's automaton and the final state q_2' of r_2's automaton, with an additional ϵ-transition from the final state q_1' of r_1's automaton to the initial state q_2 of r_2's automaton. Finally, we use the following rule for a repetition ${r_1}^*$:

$$\frac{\mathcal{C}(r_1) = (Q_1, \{q_1\}, \{q_1'\}, \delta_1) \qquad q, q' : \text{new}}{\mathcal{C}({r_1}^*) = (Q_1 \cup \{q, q'\}, \{q\}, \{q'\}, \delta_1 \cup \{q \xrightarrow{\epsilon} q_1, q_1' \xrightarrow{\epsilon} q', q_1' \xrightarrow{\epsilon} q_1, q \xrightarrow{\epsilon} q'\})}$$

That is, we first construct an automaton for the subexpression r_1; next we add a new initial state q that silently transits to the old initial state q_1 and a new final state q' to which the old final state q_1' transits silently; we further add a "looping" transition going from the old final state q_1' to the old initial state q_1 and a "skipping" transition

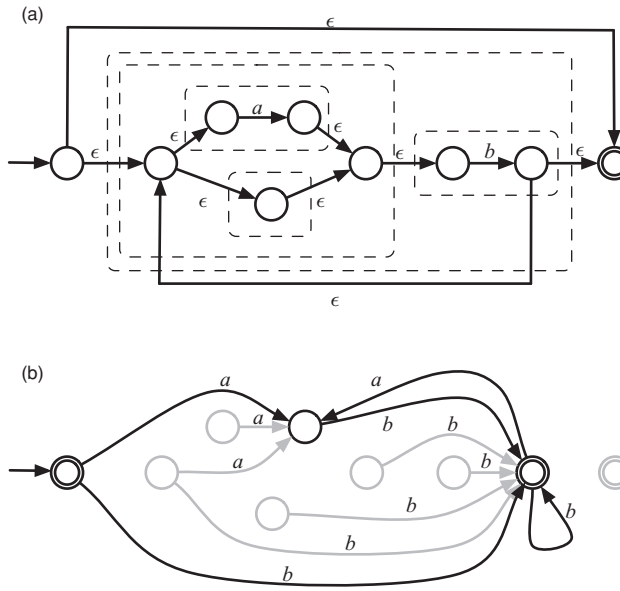


Figure 2.1 (a) McNaughton–Yamada construction from $((a \mid \epsilon)b)^*$. Each broken-lines box denotes a sub-automaton corresponding to a subexpression. (b) The string automaton after eliminating ϵ-transitions from the automaton in (a). The lighter-grey states and transitions are not reachable from the initial state and hence can be removed.

going from the new initial state q to the new final state q'. Note that from any regular expression the construction always produces an automaton with a single initial state and a single final state. Though omitted here, the reverse construction of a regular expression from a string automaton is possible.

As an example, Figure 2.1(a) shows the McNaughton–Yamada string automaton constructed from the regular expression $((a \mid \epsilon)b)^*$. By eliminating the ϵ-transitions we obtain the automaton in Figure 2.1(b). Further eliminating states unreachable from the initial state, we obtain the same string automaton as that given earlier in this section.

A string automaton $A = (Q, I, F, \delta)$ is said to be *deterministic* when I is singleton and, for each state $q \in Q$ and label $a \in \Sigma$, there is at most one transition $q \xrightarrow{a} q' \in \delta$. It is known that, for any string automaton, there is a deterministic string automaton accepting the same language. Indeed, we can construct, from a given string automaton $A = (Q, I, F, \delta)$, the deterministic string automaton $A' = (Q', I', F', \delta')$, where

$$\begin{aligned} Q' &= 2^Q \\ I' &= \{I\} \\ F' &= \{s \in Q' \mid s \cap F \neq \emptyset\} \\ \delta' &= \{s \xrightarrow{a} s' \mid s \in Q',\ s' = \{q' \mid q \xrightarrow{a} q' \in \delta,\ q \in s\}\}. \end{aligned}$$

(Here, we write 2^S for the power set of a set S.) In the construction, each new state is a set of old states. The unique new initial state is the set of all the old initial states; from any new state a new transition goes, with an attached label, to another new state. This new state is the set of all old states to which an old transition goes from one of the old states in the source with the same label; a new state that contains one of the old final states is a new final state. It is not difficult to show that the constructed automaton indeed accepts the same language, though the proof is omitted here. The above construction is called *subset construction* since we take the set of all subsets as the new set of states.

Regular string languages are known to be closed under basic set operations, namely, union, intersection, and complementation. In other words, for string automata A_1 and A_2 we can construct string automata whose languages are $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$, $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, and $\overline{\mathcal{L}(A_1)}$. Let $A_1 = (Q_1, I_1, F_1, \delta_1)$ and $A_2 = (Q_2, I_2, F_2, \delta_2)$. First, we can construct the automaton $A_\cup = (Q_1 \cup Q_2, I_1 \cup I_2, F_1 \cup F_2, \delta_1 \cup \delta_2)$. Since the resulting automaton is essentially a combination of two parallel automata, it clearly satisfies $\mathcal{L}(A_\cup) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. Next, we can construct the automaton $A_\cap = (Q_1 \times Q_2, I_1 \times I_2, F_1 \times F_2, \delta')$, where

$$\delta' = \{(q_1, q_2) \xrightarrow{a} (q_1', q_2') \mid q_1 \xrightarrow{a} q_1' \in \delta_1,\ q_2 \xrightarrow{a} q_2' \in \delta_2\}.$$

That is, we start with a pair of initial states; from any pair of states we transit with a label to another pair of states to which we can go simultaneously with the same label; we accept a string when we reach a pair of final states. It is easy to show that $\mathcal{L}(A_\cap) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. This construction is called *product construction*. Then, we can build the deterministic automaton $A_\neg = (Q', I', Q' \setminus F', \delta'')$, where $A' = (Q', I', F', \delta'')$ is the deterministic automaton constructed from A_1 by subset construction. For the resulting automaton we have that $\mathcal{L}(A_\neg) = \overline{\mathcal{L}(A_1)}$. Indeed, for any string, the deterministic automata A' and $A_\neg$ have the same unique run $q_1 \cdots q_n$ on the string, but if the state q_n is final in one automaton then it is not in the other, and vice versa.

We will conclude this chapter by stating that checking the emptiness of the language of a given string automaton is easy, since we only have to see whether a final state is reachable from an initial state, which can be done by simple graph traversal.

Part I

Basic topics

3

Schemas

Schemas are a central notion in XML for imposing structural constraints on data. A *schema language* is a "meta framework" that defines what kind of schemas can be written. There are several schema languages, providing different kinds of constraints, among which DTD, XML Schema, and RELAX NG are the most important. This chapter introduces a generic framework of schemas, called a *schema model*, that captures the essence of all these major schema languages. We will compare their expressivenesses using this model.

3.1 Data model

The whole structure of an XML document, if we ignore minor features, is a tree. Each node is associated with a label (also known as a tag) and is often called an *element* in XML jargon. The ordering among the children nodes is significant. The leaves of the tree must be text strings. That is all that data model requires *a priori*: before introducing schemas, no other constraints are imposed such as which label or how many children each node can have.

The following is an example of an XML document representing a "family tree," which will be used repeatedly throughout this book:

```
<person>
  <name>Taro</name>
  <gender><male></male></gender>
  <spouse>
    <name>Hanako</name>
    <gender><female></female></gender>
  </spouse>
  <children>
    <person>
```

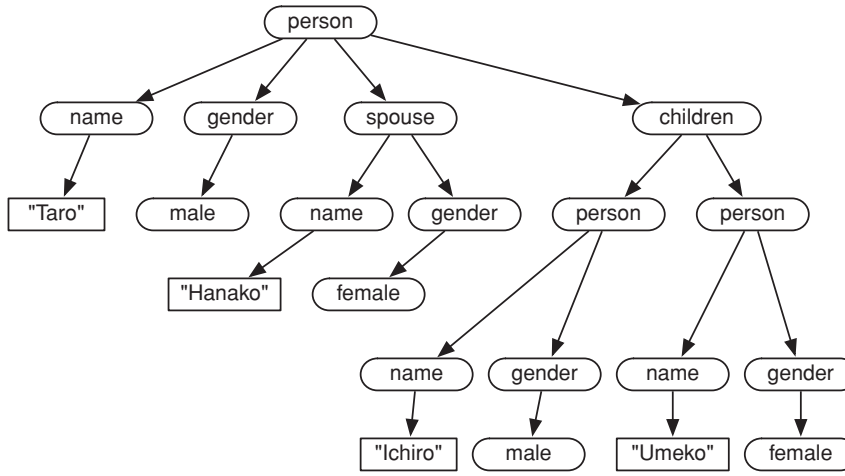


Figure 3.1 A family tree: the rounded boxes are elements and the rectangular boxes are text strings.

```
      <name>Ichiro</name>
      <gender><male></male></gender>
    </person>
    <person>
      <name>Umeko</name>
      <gender><female></female></gender>
    </person>
  </children>
</person>
```

As in the above example, an open tag such as `<person>` and its corresponding end tag, `</person>`, must appear in a nested way so that each pair of tags forms a node and the whole document represents a tree. Different nodes can have different numbers of children and, in particular, some nodes may have no child.[1] In the example, the root `person` has four children, while `male` and `female` have none. At this point we would naturally want to clarify the "rule" on the usage of those tags; however, this is the job of schema, and we will talk about it shortly. The above example document can be visualized as in Figure 3.1.

Let us now formalize the data model more precisely. We first assume a finite set Σ of *labels* ranged over by a, b, and so on. We will fix this set throughout this book, unless otherwise stated. Then, we define *values* and *nodes* mutually by the

[1] Some readers might be confused here, since usually computer science textbooks define leaves as nodes with no children. Here, we just take XML elements as nodes and text strings as leaves, which does not necessarily coincide with the usual definition. Any confusion should go away after formal definitions have been given.

following grammar:

$$\begin{array}{lcll} \text{value} & v & ::= & d_1, \ldots, d_n \quad (n \geq 0) \\ \text{node} & d & ::= & a[v] \end{array}$$

That is, a value is a sequence of nodes which can have any length including zero. It is important to emphasize that a value is *not* a set but a sequence: the nodes are ordered. Each node has the form $a[v]$, where the inner value v is often called the *content*. For readability, we write () for the empty sequence, that is, the value in the case $n = 0$. We also write $a[]$ for a node whose content is ().

Text strings, which are not directly handled in our data model, can easily be encoded by representing each character c of a string as $c[]$, a single node with no content. For instance, the string `"foo"` can be represented by $f[], o[], o[]$. In examples, however, we will continue to use the notation `"..."`, which should be read following the above convention.

Our family tree document shown earlier can be reformatted as follows in our formal notation:

```
person[
  name["Taro"],
  gender[male[]],
  spouse[
    name["Hanako"],
    gender[female[]]],
  children[
    person[
      name["Ichiro"],
      gender[male[]]],
    person[
      name["Umeko"],
      gender[female[]]]]]
```

There are many other features in real XML documents that are not supported by our data model. Those worth mentioning are as follows.

Attributes In XML, each element can be associated with a label–string mapping comprising *attributes*. For example, the following element,

```
<person age="35" nationality="japanese">
  ⋮
</person>
```

has the attributes `age` and `nationality`. Despite their apparent simplicity, treating attributes is rather tricky, because there is *no* ordering among attributes and the same

label cannot be declared more than once in the same element, which means that attributes behave quite differently from elements. We will give special consideration to attributes in Section 15.1.

Name spaces In XML, the label of each element is actually a pair consisting of a *name space* and a *local name*. This mechanism is provided in order to discriminate labels used in different applications, and is purely a lexical matter. We will ignore this feature throughout the book.

Single root In XML, a whole document is a tree and therefore has only one root. In our formalization, however, we most often treat a sequence of trees as representing a fragment of a document, and therefore a whole document is just a special case. We will soon see that this way of formalization is more convenient.

3.2 Schema model

Next, we introduce the schema model to describe the structure of data, in which the most important notion is that of *types*. A type is a formula that describes values by using the regular expression notation. Let us explain this step by step, using our family tree example. First, note that the example includes the following trivial fragments:

```
name["Taro"]            male[]
```

These can be given the following types, respectively.

```
name[String]            male[]
```

That is, a labeled node is given a label type that has the same label as the node itself. The label type has a "content" type, which describes the node's content. In the above, the content type of `name` is `String` while the content type of `male` is the empty sequence. The notations for values and types look alike and sometimes are exactly the same. So, it is important to distinguish between them from the context.

A type can be given not only to a node but also to a value, that is, a sequence of nodes. For example, for the two nodes above, the sequence

```
name["Taro"], male[]
```

can be given the type

```
name[String], male[]
```

by using the *concatenation* (expressed by a comma) of the two types. Further, types can be nested. For example, the family tree includes the sequence

```
person[
  name["Ichiro"],
  gender[male[]]],
person[
  name["Umeko"],
  gender[female[]]]
```

which can be given the following type:

```
person[
  name[String],
  gender[male[]]],
person[
  name[String],
  gender[female[]]]
```

At this moment, we would naturally think that an arbitrary number of occurrences of `person` should be allowed for such sequence. Also, the gender of each occurrence of `person` can be either `male` or `female`. Both possibilities can be specified by regular expression notations. Namely, the star * indicates zero or more *repetitions* of a type and the vertical bar | describes a *choice* of two types. Thus, we can generalize the above type as follows:

```
person[
  name[String],
  gender[male[] | female[]]]*
```

Since the type now looks rather crowded, it becomes more readable if we give names to some inner types. Therefore we introduce *type names*. For example, let us define the following type names, on the left-hand sides of the equals signs:

```
Person   = person[Name, Gender]
Name     = name[String]
Gender   = gender[Male | Female]
Male     = male[]
Female   = female[]
```

By using these, the last `person` type can be rewritten as just `Person*`.

Let us turn our attention to the content of `person`. As in our family tree document, we should allow a `spouse` node after a `gender` node. However, this should be optional since `person` may not have one. Such optionality can be described by the ? symbol. So, we redefine the `Person` type name, introducing a new type name for `spouse` nodes:

```
Person   = person[Name, Gender, Spouse?]
Spouse   = spouse[Name, Gender]
```

Likewise, we should also allow a `children` node in the tail of a `person` content. The `children` node should again be optional. However, what should be the content? It is certainly an arbitrary number of `person`. But if a `children` node is present then it does not make sense to have the empty sequence in the content; the `children` node itself should be omitted in that case. Therefore the most reasonable content type is one or more repetitions rather than zero or more. This can be expressed by a plus sign (+), another form of repetition. Summarizing all these, we obtain the following definitions of type names:

```
Person   = person[Name, Gender, Spouse?, Children?]
Name     = name[String]
Gender   = gender[Male | Female]
Male     = male[]
Female   = female[]
Spouse   = spouse[Name, Gender]
Children = children[Person+]
```

Note that the `Children` type name refers *recursively* to the `Person` type name. Thus, type names are useful not only for abbreviation but also for the description of an arbitrarily nested structure. Finally, we should specify a "starting point" in the type definitions in order to describe the whole document. A set of type definitions combined with such a start type name is called a *schema*. In our example, the start type name is `Person`; let us call the schema (E, `Person`), where E denotes the set of type definitions given above, the *family tree schema*.

Now we give a formalization of types and schemas. We first assume a set of *type names*, ranged over by X. Then *types* are defined by the following grammar:

$$
\begin{array}{rcll}
T & ::= & () & \text{empty sequence} \\
 & & a[T] & \text{label type} \\
 & & T \mid T & \text{choice} \\
 & & T, T & \text{concatenation} \\
 & & T^* & \text{repetition} \\
 & & X & \text{type name}
\end{array}
$$

Types can be seen as regular expressions over label types and type names. The inner type T in a label type $a[T]$ is often called the *content type* or *content model*. A *type definition* is a mapping from type names to types, written as follows:

$$E \quad ::= \quad \{X_1 = T_1; \ldots; X_n = T_n\}$$

A *schema* is a pair (E, X), where E is a type definition and X is a "start" type name. To avoid a nonsensical reference to a type name we require that the start type name X be one of the declared type names $X_1, \ldots, X_n$ and that each T_i contains only those declared type names. Regular expression notations not included in the above grammar can be defined as shorthands:

$$\begin{array}{lcl} T? & \equiv & T \mid () \\ T^+ & \equiv & T, T^* \end{array}$$

For encoding the `String` type, let $\{c_1, \ldots, c_n\}$ be the set of characters that can be used in strings and then abbreviate as follows:

$$\texttt{String} \quad \equiv \quad (c_1 \mid \cdots \mid c_n)^*$$

Regarding type names, there is an additional restriction called *well-formedness*. First, a *chain of references* is defined as a sequence of occurrences of type names $X_1, \ldots, X_n$ in a type definition E where each occurrence of the type name X_i lies in the type $E(X_{i-1})$, for each $i = 2, \ldots, n$. Then, the well-formedness restriction stipulates that if there is a "recursive" chain $X_1, \ldots, X_n$ with $X_1 = X_n$ then some type name X_i $(i = 2, \ldots, n)$ occurs in the content of a label type appearing in $E(X_{i-1})$. For example, on the one hand the following is disallowed:

$$\{X = a[], Y, b[]\ ;\ \ Y = X\}$$

In the recursive chain of references X, Y, X, neither Y or X occurs inside a label type. Note that Y appears *between* labels, which does not count as being inside a label. On the other hand,

$$\{X = a[], c[Y], b[];\ \ Y = X\}$$

is allowed since here Y occurs inside a label type. The purpose of this requirement is to ensure that all schemas correspond to the finite tree automata described in the next chapter. Indeed, under no such restriction, arbitrary context-free grammars can be written; however, this would be too powerful, since many operations on schemas, such as a containment check (Section 4.4), on which our later development heavily depends, become undecidable. Moreover, checking whether a given grammar is regular is also known to be an undecidable operation. Therefore we need to adopt a simple syntactic restriction that ensures well-formedness.

The meaning or *semantics* of schemas is given by specifying the values that conform to particular types. We formalize this in terms of the *conformance* relation $E \vdash v \in T$. This should informally be read as "under type definition E, value v conforms to type T." The conformance relation as in the previous line is defined by the set of inference rules given below. Let us introduce them one by

one. First, the most trivial is the following rule:

$$\frac{}{E \vdash () \in ()}\ \text{T-Eps}$$

This postulates that the empty sequence conforms to the empty sequence type, with no assumption. The next simplest rule is the following:

$$\frac{E \vdash v \in T}{E \vdash a[v] \in a[T]}\ \text{T-Elm}$$

This declares that if a value v conforms to a type T then a labeled value $a[v]$ conforms to a label type $a[T]$. The following two rules are for choices:

$$\frac{E \vdash v \in T_1}{E \vdash v \in T_1 \mid T_2}\ \text{T-Alt1}$$

$$\frac{E \vdash v \in T_2}{E \vdash v \in T_1 \mid T_2}\ \text{T-Alt2}$$

In the first rule, a value conforming to type T_1 also conforms to a type $T_1 \mid T_2$ having an additional choice T_2; the second rule is similar. The rule for concatenation is as follows:

$$\frac{E \vdash v_1 \in T_1 \qquad E \vdash v_2 \in T_2}{E \vdash v_1, v_2 \in T_1, T_2}\ \text{T-Cat}$$

That is, if values v_1 and v_2 each conform to types T_1 and T_2 then the concatenated value v_1, v_2 conforms to the concatenated type T_1, T_2. The following rule for repetition is somewhat similar:

$$\frac{E \vdash v_i \in T \quad \forall i = 1, \ldots, n \qquad n \geq 0}{E \vdash v_1, \ldots, v_n \in T^*}\ \text{T-Rep}$$

That is, if all the values $v_1, \ldots, v_n$ conform to type T then the concatenation of all the values conforms to the repetition type T^*. Note that this rule includes the case $n = 0$, that is, the empty sequence value conforms to T^* with no assumption. The final rule is for type names:

$$\frac{E \vdash v \in E(X)}{E \vdash v \in X}\ \text{T-Name}$$

That is, if a type name is defined as $X = T$ then a value conforming to T also conforms to X.

With the definition of the conformance relation, we can now describe the meaning of a schema: given a schema (E, X), we say that a value v conforms to the schema when the relation $E \vdash v \in E(X)$ holds.

Exercise 3.2.1 (⋆) Prove that our family tree document conforms to the family tree schema. ■

Exercise 3.2.2 (⋆) Write down a schema for your favorite data. Pick up an example of such data and prove that it conforms to your schema. ■

3.3 Classes of schemas

We are now ready to define classes of schemas and compare their expressivenesses. We consider here three classes, **local**, **single**, and **regular** and show that the expressiveness strictly increases in this order.

Given a schema (E, X), we define these three classes by the following conditions.

Local For any label types $a[T_1]$ and $a[T_2]$ with the same label a that occur in E, the contents are syntactically identical: $T_1 = T_2$.

Single For any label types $a[T_1]$ and $a[T_2]$ with the same label a that occur *in parallel* in E, the contents are syntactically identical: $T_1 = T_2$.

Regular No restriction is imposed.

In the definition of the class **single**, by "in parallel" we mean that the two label types may describe values in the same sequence. For example, consider the schema (E, X) where

$$E = \left\{ \begin{array}{l} X = a[T_1], (a[T_2] \mid Y) \\ Y = a[T_3], b[a[X]] \end{array} \right\}.$$

Then, $a[T_1]$, $a[T_2]$, and $a[T_3]$ are in parallel to each other since these may describe label nodes in the same sequence. However, $a[X]$ is not in parallel to any other label type.

These classes directly represent the expressivenesses of real schema languages. In DTD a schema is written in a way that defines a mapping from labels to content types and therefore obviously satisfies the requirement of the class **local**. XML Schema has a restriction called "element consistency" that expresses exactly the same requirement as the class **single**. RELAX NG has no restriction and therefore corresponds to the class **regular**.

Let us next compare their expressivenesses. First, the class **single** is obviously more expressive than the class **local**. This inclusion is strict, roughly because content types in **single** schemas can depend on the labels of any ancestor nodes

whereas in **local** schemas they can depend only on the parent's label. As a concrete counterexample, consider the following schema:

```
Person   = person[FullName, Gender, Spouse?,
                                    Children?, Pet*]
FullName = name[first[String], last[String]]
Pet      = pet[kind[String], PetName]
PetName  = name[String]
⋮
```

This schema is similar to the one in Section 3.2 except that in addition it allows `pet` elements. It is *not* local since two label types for `name` contain different content types. However, this schema *is* single since these label types do not occur in parallel. In other words, the content types of these label types depend on their grandparents, `person` or `pet`.

The class **regular** is also obviously more expressive than the class **single**. Again, this inclusion is strict. The reason is that content types in **regular** can depend not only on their ancestors but also on other "relatives", such as siblings of ancestors. For example, consider the following schema:

```
Person   = MPerson | FPerson
MPerson  = person[Name, gender[Male], FSpouse?,
                                      Children?]
FPerson  = person[Name, gender[Female], MSpouse?,
                                        Children?]
Male     = male[]
Female   = female[]
FSpouse  = spouse[Name, gender[Female]]
MSpouse  = spouse[Name, gender[Male]]
Children = children[Person+]
```

This adds to the schema in Section 3.2 an extra constraint whereby each person's spouse must have the opposite gender. This constraint is expressed by first separating the male occurrences of `person` (represented by `MPerson`) and the female occurrences of `person` (represented by `FPerson`) and then requiring each male `person` to have a female `spouse` and vice versa. This schema is *not* single since two label types for `person` appear in parallel yet have different content types. However, the schema *is* regular, of course. Note that the content type of a `person`'s `gender` depends on the `spouse`'s gender type, that is, the grandparent's "great grandchild" node. Figure 3.2 depicts this "far dependency."

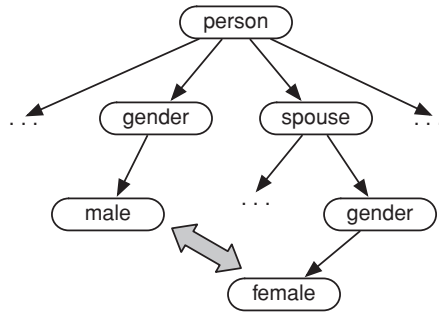


Figure 3.2 Far dependencies expressible by regular schemas. The `person` node's `gender` depends on the `spouse` node's `gender`.

Exercise 3.3.1 (⋆) What is the minimal class for your schema in Exercise 3.2.2? If it is **local** or **single**, then try to complicate your schema so that it fits only into an upper class. If it is **single** or **regular**, then try to simplify your schema so that it also fits into a lower class. ■

3.4 Bibliographic notes

Trees whose nodes can have an arbitrary number of children are called *unranked trees*. Classically, tree grammars have been used for the description of structural constraints on unranked trees. A formalization can be found in Brüggemann-Klein *et al.* (2001), where a sequence of unranked trees is called a *hedge* and a grammar for hedges is called a *hedge grammar*. A hedge is called a *forest* in Neumann and Seidl (1998) and an *ordered forest* in Fankhauser *et al.* (2001). The formalization and terminology in this book are taken from Hosoya *et al.* (2004). The comparison of schema languages made in this chapter is based on Murata *et al.* (2001). The actual specifications of the schema languages treated here are much more involved. Interested readers can find them in Bray *et al.* (2000) for XML and DTD, Fallside (2001) for XML Schema, and Clark and Murata (2001) for RELAX NG. Properties of regular string languages, context-free string languages, and more can be found in Hopcroft and Ullman (1979).

4

Tree automata

Tree automata provide a finite-state acceptor model for trees that naturally corresponds to schemas. In this chapter, we first give a definition of tree automata and explain how schemas can be converted to tree automata. Then, we introduce two different definitions of determinism for tree automata and compare them. Finally we demonstrate standard properties, such as closure properties, the decidability of emptiness and containment, and so on. All these notions will be of key importance in later chapters.

4.1 Definitions

In this book, we consider only tree automata accepting binary trees. This may sound rather restrictive, since in fact each node in XML documents can have an arbitrary number of children. However, it is actually enough. We will come back to this point soon, in Section 4.2.

4.1.1 Trees

We define *(binary) trees* by the following grammar, where a ranges over labels from the set Σ given in Section 3.1:[1]

$$\begin{array}{lcll} t & ::= & \# & \text{leaf} \\ & & a(t,t) & \text{intermediate node} \end{array}$$

[1] The nodes and leaves introduced in this chapter are for binary trees and therefore different from those for unranked trees given in Chapter 3. Also, we will often use the terms *node* and *subtree* interchangeably unless the distinction becomes important.

Let us define a few standard notions related to trees. First, the height of a tree t, written $\mathbf{ht}(t)$, is defined by

$$\begin{array}{lcl} \mathbf{ht}(\#) & = & 1 \\ \mathbf{ht}(a(t_1, t_2)) & = & 1 + \max(\mathbf{ht}(t_1), \mathbf{ht}(t_2)). \end{array}$$

In what follows, we will often consider a particular node in a given tree. For this, a convenient notion is the *position* π, defined as a sequence from $\{1, 2\}^*$; informally, the position locates a node by traversing a tree from the root, repeatedly following the left child for each prefix 1 and the right child for each prefix 2. Formally, the subtree located at a position π in a tree t, written $\mathbf{subtree}_t(\pi)$, is defined by

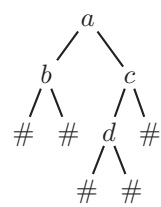
$$\begin{array}{lcl} \mathbf{subtree}_t(\epsilon) & = & t \\ \mathbf{subtree}_{a(t_1,t_2)}(i\pi) & = & \mathbf{subtree}_{t_i}(\pi) \quad (i = 1, 2). \end{array}$$

Then the set of nodes of a tree t is defined as $\mathbf{nodes}(t) = \mathbf{dom}(\mathbf{subtree}_t)$, that is, the set of positions on which the function $\mathbf{subtree}_t$ is defined (we write $\mathbf{dom}(f)$ for the domain of a function f). Also, the label of a node is defined by

$$\mathbf{label}_t(\pi) = \begin{cases} a & (\mathbf{subtree}_t(\pi) = a(t_1, t_2)) \\ \# & (\mathbf{subtree}_t(\pi) = \#). \end{cases}$$

The set of leaves (end nodes) is defined by $\mathbf{leaves}(t) = \{\pi \in \mathbf{nodes}(t) \mid \mathbf{label}_t(\pi) = \#\}$.

Example 4.1.1 Let $t_{4.1.1}$ be the tree $a(b(\#, \#), c(d(\#, \#), \#))$, which can be depicted as follows:



The height is $\mathbf{ht}(t_{4.1.1}) = 4$. The position 21 denotes the node labeled d. Indeed,

$$\begin{array}{lcl} \mathbf{subtree}_{t_{4.1.1}}(21) & = & \mathbf{subtree}_{c(d(\#,\#),\#)}(1) \\ & = & \mathbf{subtree}_{d(\#,\#)}(\epsilon) \\ & = & d(\#, \#). \end{array}$$

The label of this node is

$$\mathbf{label}_{t_{4.1.1}}(21) = d.$$

The set of nodes is

$$\mathbf{nodes}(t_{4.1.1}) = \{\epsilon, 1, 11, 12, 2, 21, 211, 212, 22\}$$

and the set of leaves is

$$\mathbf{leaves}(t_{4.1.1}) = \{11, 12, 211, 212, 22\}. \quad \blacksquare$$

By using the notion of positions, we can define several relations between nodes at far positions. First, a node at position π_1 has a *descendant* at position π_2, written $\pi_1 < \pi_2$, when $\pi_2 = \pi_1\pi$ for some $\pi \neq \epsilon$. We write $\pi_1 \leq \pi_2$ when $\pi_1 < \pi_2$ or $\pi_1 = \pi_2$. The relation $\leq$ is called the *self-or-descendant relation*. Another relation that we occasionally use is the *document order relation* $\preceq$. Intuitively, this relation represents the ordering of nodes occurring in a depth-first left-to-right traversal of the tree, or, in another view, the ordering of nodes appearing in the XML document format. Despite its non-triviality, the definition of this relation is surprisingly simple: the lexicographic order on $\{1, 2\}^*$.

Example 4.1.2 Consider again the tree $t_{4.1.1}$. The nodes labeled c (position 2) and d (position 21) are in the descendant relation $2 < 21$. Also, the node labeled b (position 1) and the one labeled d are in the document order relation $1 \preceq 21$. $\blacksquare$

4.1.2 Tree automata

A *(nondeterministic) tree automaton* A is a quadruple (Q, I, F, Δ), where

- Q is a finite set of *states*,
- I is a set of *initial states* ($I \subseteq Q$),
- F is a set of *final states* ($F \subseteq Q$), and
- Δ is a set of *transition rules* of the form

$$q \to a(q_1, q_2)$$

  where $q, q_1, q_2 \in Q$ and $a \in \Sigma$.

In transition rules of the form $q \to a(q_1, q_2)$ we often call q the "source state" and q_1 and q_2 the "destination states."

The semantics of tree automata is described in terms of runs. Given a tree automaton $A = (Q, I, F, \Delta)$ and a tree t, a *run* r of A on t is a mapping from $\mathbf{nodes}(t)$ to Q such that

- $r(\epsilon) \in I$, and
- $r(\pi) \to a(r(\pi 1), r(\pi 2)) \in \Delta$ whenever $\mathbf{label}_t(\pi) = a$ for $\pi \in \mathbf{nodes}(t)$.

A run r is *successful* if $r(\pi) \in F$ for each $\pi \in \mathbf{leaves}(t)$. The intuition behind a run is as follows. First, we assign an initial state to the root. At each intermediate node $a(t_1, t_2)$ we pick up a transition rule $q \to a(q_1, q_2)$ such that the source state

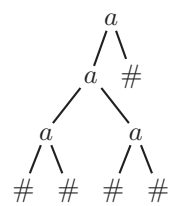
q is assigned to the current node and then assign the destination states q_1 and q_2 to the subnodes t_1 and t_2. We repeat this procedure for the whole tree. Finally, the run is successful when each leaf is assigned a final state. Since in this intuition a run (whether successful or not) "goes" from the root to the leaves, we call such a run a *top-down run*.

We say that a tree automaton A *accepts* a tree t when there is a successful run of A on t. Further, we define the language $\mathcal{L}(A)$ of A to be the set of trees accepted by A. A language accepted by some nondeterministic tree automaton is called a *regular tree language*; let **ND** be the class of regular tree languages.

Example 4.1.3 Let $A_{4.1.3}$ be the tree automaton $(\{q_0, q_1\}, \{q_0\}, \{q_1\}, \Delta)$, where

$$\Delta = \begin{Bmatrix} q_0 \to a(q_1, q_1), \\ q_1 \to a(q_0, q_0) \end{Bmatrix}.$$

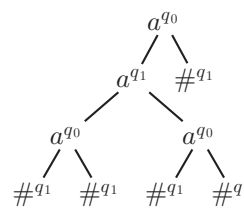
Consider the following tree, $t_{4.1.3}$:



This tree is accepted by the tree automaton $A_{4.1.3}$. Indeed, we can find the following top-down run:

$$r_{4.1.3} = \begin{Bmatrix} \epsilon \mapsto q_0, & & & \\ 1 \mapsto q_1, & 2 \mapsto q_1, & & \\ 11 \mapsto q_0, & 12 \mapsto q_0, & & \\ 111 \mapsto q_1, & 112 \mapsto q_1, & 121 \mapsto q_1, & 122 \mapsto q_1 \end{Bmatrix}$$

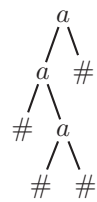
This may be easier to see in an overlay on the tree, in which superscripts give the state at a node.



This run is successful since all the leaves, 2, 111, 112, 121, and 122, are assigned the final state q_1. In general, the tree automaton $A_{4.1.3}$ accepts the set of trees in which every leaf is at an even depth. Indeed, in any top-down run, all the nodes at odd

depths are assigned state q_0 and those at even depths are assigned the state q_1. Therefore, in order for this run to be successful, all the leaves need to be at even depths since q_1 is the only final state. ■

Exercise 4.1.4 (⋆) Show that the tree automaton $A_{4.1.3}$ does not accept the following tree, $t_{4.1.4}$:



■

Analogously to top-down runs, we can also consider bottom-up runs. Given a tree automaton $A = (Q, I, F, \Delta)$ and a tree t, a *bottom-up run* r of A on t is a mapping from $\mathbf{nodes}(t)$ to Q such that

- $r(\pi) \in F$ for each $\pi \in \mathbf{leaves}(t)$, and
- $r(\pi) \to a(r(\pi 1), r(\pi 2)) \in \Delta$ whenever $\mathbf{label}_t(\pi) = a$ for $\pi \in \mathbf{nodes}(t)$.

A bottom-up run r is *successful* if $r(\epsilon) \in I$. Intuitively, we first assign some final state to each leaf. Then, for each intermediate node $a(t_1, t_2)$, we choose a transition rule $q \to a(q_1, q_2)$ such that t_1 and t_2 are assigned the states q_1 and q_2, respectively, and then we assign q to the current node. We repeat this for the whole tree. Finally, such a run is successful when the root is assigned an initial state. Note that the definitions of successful runs are identical for top-down and bottom-up runs.

Example 4.1.5 Consider again the tree automaton $A_{4.1.3}$. The top-down run $r_{4.1.3}$ can also be seen as a bottom-up run and it is successful since it maps the root to the initial state q_0. ■

Exercise 4.1.6 (⋆) Is there a bottom-up run on the tree $t_{4.1.4}$? ■

Later, when we consider various properties of tree automata, it is often convenient to define an "intermediate state" of acceptance. Formally, given a tree t (which can be a subtree of a whole tree), a tree automaton $A = (Q, I, F, \Delta)$ *accepts* t *in state* $q \in Q$ when a bottom-up run r of A on t maps the root of t to the state q. When it is obvious from the context which automaton we are talking about, we also say that *state* q *accepts* t. Clearly, a state q accepts a tree t if and only if

- $t = \#$ and $q \in F$, or
- t has the form $a(t_1, t_2)$ and there is a transition $q \to a(q_1, q_2) \in \Delta$ such that q_1 accepts t_1 and q_2 accepts t_2.

Example 4.1.7 The tree automaton $A_{4.1.3}$ accepts

#	in state q_1,
$a(\#, \#)$	in state q_0,
$a(a(\#, \#), a(\#, \#))$	in state q_1, and
$a(a(a(\#, \#), a(\#, \#)), \#)$	in state q_0.

The last tree equals $t_{4.1.3}$. ■

4.1.3 Tree automata with ϵ-transitions

As in the case of string automata, it is useful to introduce tree automata that can silently transit between states while staying on the same node. Formally, a tree automaton A with ϵ-transitions is a quadruple (Q, I, F, Δ), where Q, I, and F are similar to a normal tree automaton and Δ may contain transition rules of the form

$$q \xrightarrow{\epsilon} q'$$

where $q, q' \in Q$, in addition to labeled transition rules. The semantics is given by extending that of tree automata. For this, we first define

$$\epsilon\text{-}\mathbf{closure}(q_1) = \{q_k \mid q_1 \xrightarrow{\epsilon} q_2, \ldots, q_{k-1} \xrightarrow{\epsilon} q_k \in \Delta,\ k \geq 1\}.$$

That is, ϵ-**closure**(q_1) contains all the states that can be reached from q_1 by following zero or more ϵ-transitions. Then, given a tree t, a (top-down) run r of A on t is a mapping from **nodes**(t) to Q such that

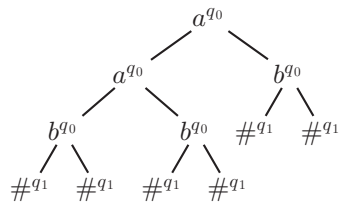
- $r(\epsilon) \in I$, and
- there exists $q \in \epsilon$-**closure**$(r(\pi))$, with $q \to a(r(\pi 1), r(\pi 2)) \in \Delta$, whenever $\mathbf{label}_t(\pi) = a$ for $\pi \in \mathbf{nodes}(t)$.

A run r is *successful* if, for each $\pi \in \mathbf{leaves}(t)$, there exists $q \in \epsilon$-**closure**$(r(\pi))$ with $q \in F$. We can also consider bottom-up runs in an analogous way.

Example 4.1.8 Consider the tree automaton $A_{4.1.8} = (\{q_0, q_1\}, \{q_0\}, \{q_1\}, \Delta)$ with ϵ-transitions where

$$\Delta = \left\{ \begin{array}{l} q_0 \to a(q_0, q_0), \\ q_0 \xrightarrow{\epsilon} q_1, \\ q_1 \to b(q_1, q_1) \end{array} \right\}.$$

This automaton accepts a tree with labels a and b, where any b-labeled node has no a-labeled node in its subtree. For example, the following tree is accepted, with the run indicated:



Note that the run follows the ϵ-transition from q_0 to q_1 at each node with label b. ■

As in the case of string automata, the ϵ-transitions of a tree automaton can be eliminated without changing the accepting set. Indeed, for a tree automaton $A = (Q, I, F, \Delta)$ with ϵ-transitions, its ϵ-elimination is the tree automaton $A' = (Q, I, F', \Delta')$, where

$$\begin{aligned} F' &= \{q \mid q \in Q,\ q' \in \epsilon\text{-}\mathbf{closure}(q),\ q' \in F\} \\ \Delta' &= \{q \to a(q_1, q_2) \mid q \in Q,\ q' \in \epsilon\text{-}\mathbf{closure}(q),\ q' \to a(q_1, q_2) \in \Delta\}. \end{aligned}$$

Lemma 4.1.9 A tree automaton A with ϵ-transitions and its ϵ-elimination A' accept the same language.

Proof: We can easily show that, by the definition of ϵ-elimination, a successful run r of A on a tree t is also a successful run of A' on t and vice versa, from which the result follows immediately. □

Exercise 4.1.10 (⋆) Obtain the ϵ-elimination of the tree automaton $A_{4.1.8}$. ■

4.2 Relationship with the schema model

First we need to resolve the discrepancy in the data models for binary and unranked trees. This can easily be done using the well-known *binarization* technique, in much the same way that the Lisp programming language forms list structures by using `cons` and `nil` constructors. The function **bin** formalizes a translation from unranked trees to binary trees:

$$\begin{aligned} \mathbf{bin}(a[v_1], v_2) &= a(\mathbf{bin}(v_1),\ \mathbf{bin}(v_2)) \\ \mathbf{bin}(()) &= \# \end{aligned}$$

The first rule states that a value of the form $a[v_1], v_2$ in the unranked tree representation, that is, a sequence whose first element has the label a, corresponds to the binary tree $a(t_1, t_2)$ whose left child t_1 corresponds to the first element's content v_1 and whose right child t_2 corresponds to the remainder sequence v_2. The empty

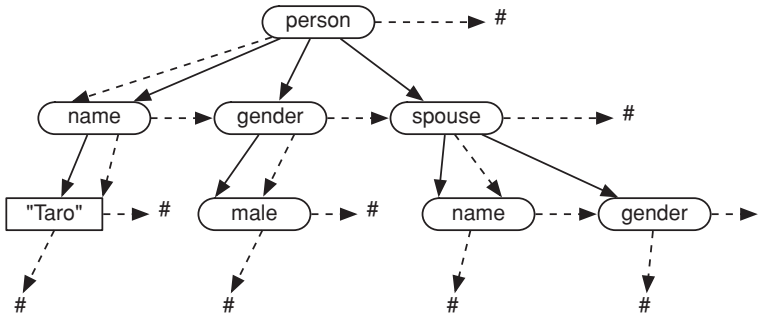


Figure 4.1 Binarization. An unranked tree is shown overlaid with its corresponding binary tree. The edges of the unranked tree are shown by solid lines and those of the binary tree by broken lines.

sequence () in the unranked tree representation corresponds to the leaf #. Figure 4.1 shows an example of an unranked tree overlaid with its corresponding binary tree. Both representations have the same labeled nodes but different edges.

Analogously, schemas and tree automata correspond to each other through binarization. The following procedure constructs a tree automaton from a given schema in which the set of accepted trees consists of the binarizations of the unranked trees that conform to the schema.

1. In order to simplify the procedure, we first convert the given schema to a canonical form. That is, a schema (E, X_1) where $E = \{X_1 = T_1, \ldots, X_n = T_n\}$ is *canonical* if each type T_i is a canonical type. Here, *canonical types* T_c are defined by the following grammar:

$$\begin{array}{rcl} T_c & ::= & () \\ & & a[X] \\ & & T_c \mid T_c \\ & & T_c, T_c \\ & & T_c{}^* \end{array}$$

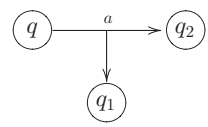
   That is, in canonical types, the content of a label is always a type name and, conversely, a type name can appear only there. It is clear that an arbitrary schema can be canonicalized. Here, it is critical that the schema is well formed, as defined in Section 3.2, otherwise canonicalization may fail, e.g., we could have $(\{X = X\}, X)$. (In fact, well-formedness is important only in the context of canonicalization.)
2. For each i, regard the type T_i as a regular expression over the set of forms $a[X]$ and construct a string automaton $A_i = (Q_i, I_i, F_i, \delta_i)$ from this regular expression, so that each automaton has a disjoint set of states. Note that each transition in the automaton has the form $q_1 \xrightarrow{a[X]} q_2$. For this step we can use any construction algorithm from regular expressions to string automata, such as McNaughton–Yamada construction given in Section 2.2.

3. Merge all the string automata $A_1, \ldots, A_n$ to form the tree automaton $A = (Q, I, F, \Delta)$, where

$$\begin{array}{lcl} Q & = & \bigcup_{i=1}^{n} Q_i \\ I & = & I_1 \\ F & = & \bigcup_{i=1}^{n} F_i \\ \Delta & = & \bigcup_{i=1}^{n} \left\{ q_1 \to a(q_0, q_2) \,\middle|\, q_1 \xrightarrow{a[X_j]} q_2 \in \delta_i,\ q_0 \in I_j \right\}. \end{array}$$

   That is, we start with one of the initial states I_1 of A_1, since the start type name is X_1. Each transition rule $q_1 \to a(q_0, q_2)$ of the tree automaton A is constructed from a transition rule $q_1 \xrightarrow{a[X_j]} q_2$ in some string automaton A_i, where q_0 is one of the initial states I_j of the string automaton A_j corresponding to the content type X_j.

Note that, analogously to the binarization of trees, the first and the second destination states of each transition rule correspond to the first child's content and to the remainder sequence, respectively. To emphasize this, we will draw a diagram of a tree automaton using the following notation for a transition rule $q \to a(q_1, q_2)$:



As an example of translation from a schema to a tree automaton, consider the family-tree schema given in Section 3.2. First, this schema can be canonicalized as follows:

```
Person     = person[PersonC]
PersonC    = name[String], gender[GenderC],
                    spouse[SpouseC]?, children[ChildrenC]?
GenderC    = male[Empty] | female[Empty]
SpouseC    = name[String], gender[GenderC]
ChildrenC  = person[PersonC]+
String     = (c1[Empty]|...| cn[Empty])*
Empty      = ()
```

where `c1`, ..., `cn` denote possible characters in a string. We can then construct, for this canonical schema, the tree automaton depicted in Figure 4.2.

Exercise 4.2.1 (⋆) Construct tree automata from the two extended schemas given in Section 3.3. ■

Exercise 4.2.2 (⋆) Do the same for your schemas in Exercises 3.2.2 and 3.3.1. ■

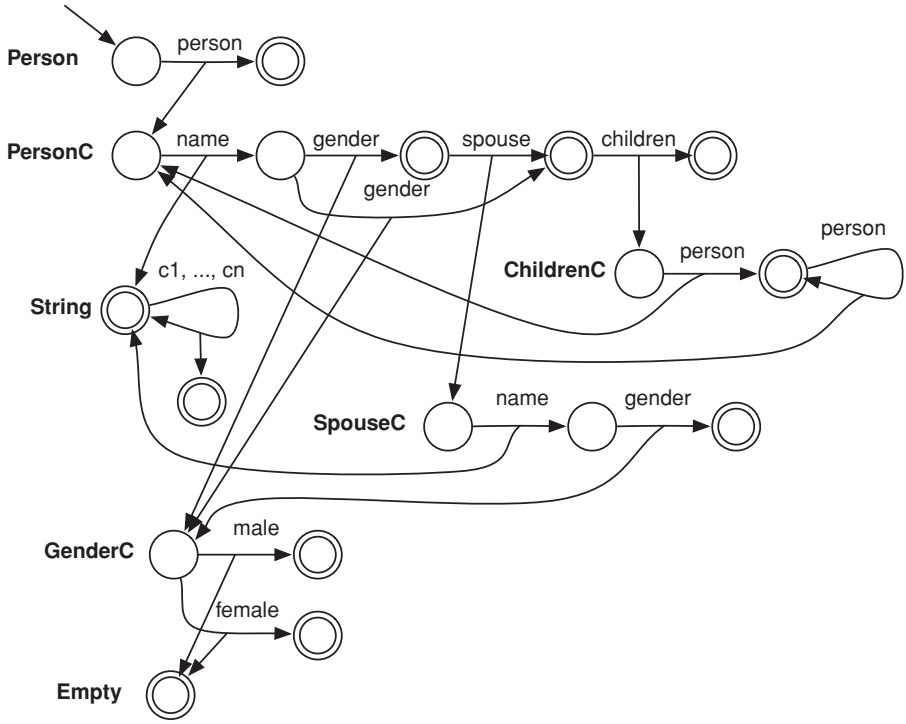


Figure 4.2 Tree automaton corresponding to the family tree schema given in Section 3.2.

Exercise 4.2.3 (⋆⋆) Above we discussed the translation from schemas to tree automata. Construct an algorithm for a translation in the other direction, assuming the existence of a conversion method from string automata back to regular expressions (see the bibliographic notes at the end of the chapter). ■

4.3 Determinism

In the case of tree automata, just as for string automata, it is important to consider determinism. For example, the proof of closure under complementation given in Section 4.4.1 below critically relies on a procedure that converts a tree automaton to a deterministic automaton. However, unlike string automata, there are two natural definitions of determinism: top-down and bottom-up. We will see below that bottom-up deterministic tree automata are as expressive as nondeterministic tree automata, whereas top-down deterministic tree automata are *strictly* weaker.

4.3.1 Top-down determinism

A tree automaton $A = (Q, I, F, \Delta)$ is *top-down deterministic* when

- the set I of initial states is singleton, and
- for each $q \in Q$ and $a \in \Sigma$ there is at most one transition rule of the form $q \to a(q_1, q_2)$ in Δ.

Intuitively, in a top-down deterministic tree automaton, a top-down run can easily be found on any tree by deterministically assigning a state to each node in a top-down manner. This clearly means that a top-down deterministic tree automaton allows at most one top-down run on any tree. The tree automaton $A_{4.1.3}$ is top-down deterministic. However, a tree automaton is not necessarily top-down deterministic even if it allows at most one top-down run on any tree.

Example 4.3.1 Let $A_{4.3.1}$ be (Q, I, F, Δ), where

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3\} \\ I &= \{q_0\} \\ F &= \{q_3\} \\ \Delta &= \begin{Bmatrix} q_0 \to a(q_1, q_2) \\ q_0 \to a(q_2, q_1) \\ q_1 \to b(q_3, q_3) \\ q_2 \to c(q_3, q_3) \end{Bmatrix}. \end{aligned}$$

The accepted language is

$$\mathcal{L}(A_{4.3.1}) = \left\{ \begin{array}{cccccccc} & & a & & & & a & \\ & b & & c & , & c & & b \\ \# & \# & \# & \# & & \# & \# & \# & \# \end{array} \right\}.$$

For either tree, only one top-down run is possible. Nevertheless, the tree automaton is not top-down deterministic. ■

Let **TD** be the class of languages accepted by top-down deterministic tree automata. We can show that this class is strictly smaller than the class of regular tree languages.

Theorem 4.3.2 $\mathbf{TD} \subsetneq \mathbf{ND}$.

Proof: Since $\mathbf{TD} \subseteq \mathbf{ND}$ is clear, it suffices to raise an example of a language that is in **ND** but not in **TD**. Indeed, the automaton $A_{4.3.1}$ gives such an example. To show this, first suppose that the accepted language $\mathcal{L}(A_{4.3.1})$ is accepted by a top-down deterministic tree automaton $A' = (Q', I', F', \Delta')$. Then by definition

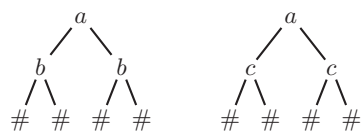
I' must be a singleton set $\{q'_0\}$. Moreover, since the set $\mathcal{L}(A_{4.3.1})$ contains a tree with the root labeled a, there must be a transition $q'_0 \to a(q'_1, q'_2) \in \Delta'$ for some $q'_1, q'_2 \in Q'$, and no other transition from q'_0 with label a. Since A' accepts both the trees in $\mathcal{L}(A_{4.3.1})$ in state q'_0, there must also be transitions

$$q'_1 \to b(q'_{11}, q'_{12})$$
$$q'_2 \to c(q'_{21}, q'_{22})$$

and

$$q'_1 \to c(q'_{13}, q'_{14})$$
$$q'_2 \to b(q'_{23}, q'_{24})$$

in Δ, where all q'_{ij} are in F'. This implies that A' must accept the following trees in addition to $\mathcal{L}(A_{4.3.1})$:



Thus, a contradiction is derived. □

Although the expressiveness is lower than that of nondeterministic tree automata, top-down deterministic tree automata have algorithmically more desirable properties. In particular, acceptance can be checked in linear time using an extremely simple algorithm (Section 8.1.1). For this reason, some major schema languages adopt restrictions which ensure that schemas can easily be converted to top-down deterministic tree automata. For example, XML Schema has two restrictions, singleness of grammar (Section 3.3) and one-unambiguity (see the end of Section 14.1.2); similarly, DTD has locality of grammar (Section 3.3) and one-unambiguity. The one-unambiguity restriction ensures that a deterministic string automaton can easily be constructed from each content model; see Chapter 14 for more discussion.

Exercise 4.3.3 (⋆⋆) Show that the construction algorithm in Section 4.2 yields a top-down deterministic tree automaton from a **single** schema if the second step always produces a deterministic string automaton. ■

4.3.2 Bottom-up determinism

The definition of bottom-up deterministic tree automata is symmetric to that of top-down deterministic tree automata. A tree automaton $A = (Q, I, F, \Delta)$ is *bottom-up*

deterministic when

- the set F of final states is a singleton, and
- for each $q_1, q_2 \in Q$ and $a \in \Sigma$, there is at most one transition rule of the form $q \to a(q_1, q_2)$ in Δ.

Again, intuitively, in a bottom-up deterministic tree automaton we can always find a bottom-up run by assigning a uniquely possible state to each node in a bottom-up manner. This implies that at most one bottom-up run is possible on any tree. For example, the tree automaton $A_{4.1.3}$ is also bottom-up deterministic. Let **BU** be the class of languages accepted by bottom-up deterministic tree automata.

Exercise 4.3.4 (⋆) Find a tree automaton that allows at most one bottom-up run on any tree but is not bottom-up deterministic. ■

Unlike the top-down case, the class of languages accepted by bottom-up deterministic tree automata coincides with the class of regular tree languages. To prove this property we use the classical "subset construction" technique, which is used also for determinizing (i.e., making deterministic) string automata (Section 2.2). That is, given a nondeterministic tree automaton $A = (Q, I, F, \Delta)$, the *subset tree automaton* of A is the tree automaton $A' = (Q', I', F', \Delta')$ such that

$$\begin{array}{lcl} Q' & = & 2^Q \\ I' & = & \{s \in Q' \mid s \cap I \neq \emptyset\} \\ F' & = & \{F\} \\ \Delta' & = & \big\{s \to a(s_1, s_2) \mid s_1, s_2 \in Q', \\ & & \qquad s = \{q \mid q \to a(q_1, q_2) \in \Delta,\ q_1 \in s_1,\ q_2 \in s_2\}\big\}. \end{array}$$

That is, we take every subset of the old states Q as a new state. A new initial state is such a subset that includes an old initial state. A new final state is uniquely the set of all old final states. A new transition $s \to a(s_1, s_2)$ is formed for every pair of subsets s_1, s_2 such that s is the set of all old states q that have a transition $q \to a(q_1, q_2)$ going to a pair of old states q_1, q_2 in the sets s_1, s_2, respectively. Clearly, the subset tree automaton is bottom-up deterministic.

We say that a tree automaton (Q, I, F, Δ) is *complete* when, for any $a \in \Sigma$ and any $q_1, q_2 \in Q$, there exists $q \in Q$ such that $q \to a(q_1, q_2) \in \Delta$. The subset tree automaton constructed above is complete, and for a complete tree automaton there is a bottom-up run on *any* tree. In particular, for a complete bottom-up deterministic tree automaton there is a *unique* bottom-up run on any tree.

Theorem 4.3.5 If A' is the subset tree automaton of A then $\mathcal{L}(A) = \mathcal{L}(A')$.

Proof: Let $A = (Q, I, F, \Delta)$ and $A' = (Q', I', F', \Delta')$. We first show that $\mathcal{L}(A) \subseteq \mathcal{L}(A')$. Suppose that r is a successful run of A on a tree t. Also, since A' is bottom-up deterministic and complete, there is a unique bottom-up run r' on t. Then we can show that $r(\pi) \in r'(\pi)$ for all $\pi \in \mathbf{nodes}(t)$ by induction on the height of $\mathbf{subtree}_t(\pi)$, as follows.

- For $\pi \in \mathbf{leaves}(t)$ we have $r(\pi) \in F = r'(\pi)$.
- For $\mathbf{label}_t(\pi) = a$, we have $r(\pi) \to a(r(\pi 1), r(\pi 2)) \in \Delta$. Also, since $r'(\pi) \to a(r'(\pi 1), r'(\pi 2)) \in \Delta'$, we have $r'(\pi) = \{q \mid q \to a(q_1, q_2) \in \Delta,\ q_1 \in r'(\pi 1),\ q_2 \in r'(\pi 2)\}$. By the induction hypothesis, $r(\pi 1) \in r'(\pi 1)$ and $r(\pi 2) \in r'(\pi 2)$. Therefore $r(\pi) \in r'(\pi)$.

Since $r(\epsilon) \in I$ and $r(\epsilon) \in r'(\epsilon)$, we conclude that $r'(\epsilon) \in I'$, that is, r' is successful.

We next show that $\mathcal{L}(A') \subseteq \mathcal{L}(A)$. Suppose that r' is a successful run of A' on a tree t. Then we can construct a top-down run r on A on t such that $r(\pi) \in r'(\pi)$ for all $\pi \in \mathbf{nodes}(t)$, from the root to the leaves, as follows.

- For the root, since $r'(\epsilon) \cap I \neq \emptyset$, choose $q \in r'(\epsilon) \cap I$ and let $r(\epsilon) = q$.
- For $\mathbf{label}_t(\pi) = a$, since $r'(\pi) \to a(r'(\pi 1), r'(\pi 2)) \in \Delta'$ and $r(\pi) \in r'(\pi)$, there exist q_1 and q_2 such that $r(\pi) \to a(q_1, q_2) \in \Delta$ with $q_1 \in r'(\pi 1)$ and $q_2 \in r'(\pi 2)$; let $r(\pi 1) = q_1$ and $r(\pi 2) = q_2$.

For each $\pi \in \mathbf{leaves}(t)$, since $r(\pi) \in r'(\pi) = F$ we have $r(\pi) \in F$. Therefore r is successful. □

Corollary 4.3.6 $\mathbf{BU} = \mathbf{ND}$.

Proof: That $\mathbf{BU} \subseteq \mathbf{ND}$ is clear. The converse follows from Theorem 4.3.5, since, for any nondeterministic tree automaton, its subset tree automaton is bottom-up deterministic and accepts the same language. □

4.4 Basic set operations

In this section we describe how to compute basic set operations on tree automata, namely, union, intersection, complementation, and emptiness test, together with their applications. The membership test, perhaps the most basic set operation, will not be considered here but instead will be studied thoroughly in the context of efficient algorithms in Chapter 8.

4.4.1 Union, intersection, and complementation

Like regular string languages, regular tree languages have standard closure properties. That is, given two regular tree languages, their union, and intersection are

regular tree languages. Also, the complementation of a regular tree language is a regular tree language. The proof techniques are similar to those used for regular string languages (Section 2.2).

First, closure under union is extremely easy.

Proposition 4.4.1 (Closure under union) Given tree automata A_1 and A_2, a tree automaton B accepting $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$ can be computed effectively (i.e., by some algorithm).

Proof: Let $A_i = (Q_i, I_i, F_i, \Delta_i)$ for $i = 1, 2$. Then, the tree automaton $B = (Q_1 \cup Q_2, I_1 \cup I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$ clearly satisfies $\mathcal{L}(B) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. □

Next, we consider closure under intersection. The proof is based on the product-construction technique. Let A_i be tree automata $(Q_i, I_i, F_i, \Delta_i)$ for $i = 1, 2$. Then define the *product tree automaton* $B = (Q_1 \times Q_2, I_1 \times I_2, F_1 \times F_2, \Delta')$, where

$$\Delta' = \{(q_1, q_2) \to a((r_1, r_2), (s_1, s_2)) \mid q_1 \to a(r_1, s_1) \in \Delta_1,\ q_2 \to a(r_2, s_2) \in \Delta_2\}.$$

That is, every pair of old states becomes a new state, every pair of old initial states a new initial state, and every pair of old final states a new final state. A new transition is formed for every pair of old transitions with the same label by grouping the components together. It is worth noting that if both A_1 and A_2 are top-down deterministic then so is their product. Also, if both A_1 and A_2 are bottom-up deterministic, then so is their product.

Proposition 4.4.2 (Closure under intersection) Given tree automata A_1 and A_2, a tree automaton B accepting $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ can be computed effectively.

Proof: Let B be the product tree automaton of A_1 and A_2. Then, it suffices to prove that $\mathcal{L}(B) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. To show that $\mathcal{L}(A_1) \cap \mathcal{L}(A_2) \subseteq \mathcal{L}(B)$, suppose that we have successful runs r_1 of A_1 and r_2 of A_2 both on a tree t. Define r such that $r(\pi) = (r_1(\pi), r_2(\pi))$ for each $\pi \in \mathbf{nodes}(t)$. Then, by the definition of B, we have that r is a successful run of A on t. The converse, $\mathcal{L}(B) \subseteq \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, can be shown similarly. □

Finally, we prove closure under complementation.

Proposition 4.4.3 (Closure under complementation) Given a tree automaton A, a tree automaton B accepting $\overline{\mathcal{L}(A)}$ can be computed effectively.

Proof: As shown in Section 4.3.2, the subset tree automaton $A' = (Q', I', F', \Delta')$ of A is complete and bottom-up deterministic and, by Theorem 4.3.5 satisfies $\mathcal{L}(A') = \mathcal{L}(A)$. Let $B = (Q', Q' \setminus I', F', \Delta')$. Since B is also complete and bottom-up deterministic and has the same final states and transitions as A', both A' and B have the same, unique, bottom-up run on any tree. Since the initial states of A' are

the complement of those of B, we conclude that, for any tree t, A' accepts t iff B does not accept t. □

Since in Section 4.2 we have already established an equivalence between schemas and tree automata, the above closure properties can be transferred directly to schemas.

Corollary 4.4.4 Schemas are closed under union, intersection, and complementation.

4.4.2 Emptiness test

It is easy to test whether a tree automaton accepts some tree or not. However, unlike the case of string automata, it is not sufficient to perform a linear traversal to check whether a final state is reachable from an initial state: each transition branches to two destination states and, in order for a transition to contribute to the acceptance of some tree, the two destination states must each accept some tree.

The algorithm presented here examines states in a bottom-up way, from final states to initial states. The following shows the pseudo-code for the algorithm.

```
1: function IsEmpty(Q, I, F, Δ)                    ▷ returning a boolean
2:     Q_NE ← F
3:     repeat
4:         for all q → a(q1, q2) ∈ Δ s.t. q1, q2 ∈ Q_NE do
5:             Q_NE ← Q_NE ∪ {q}
6:         end for
7:     until Q_NE does not change
8:     return Q_NE ∩ I = ∅
9: end function
```

In the loop the variable $Q_{\mathbf{NE}}$ collects states that have been shown to be non-empty and therefore accept some tree. Initially, $Q_{\mathbf{NE}}$ is assigned the set of all final states since they definitely accept the tree # with a single leaf. Then, we look for a transition both of whose destinations are non-empty states and add its source state since now we know that this state accepts some tree. We repeat this until no state can be added. Finally, then, the obtained $Q_{\mathbf{NE}}$ contains the set of *all* non-empty states. Thus we conclude that the given automaton is empty if and only if $Q_{\mathbf{NE}}$ contains no initial state. Note that termination is guaranteed: the loop continues only if each repetition augments $Q_{\mathbf{NE}}$, and this procedure eventually breaks down

because of its finite upper bound. Clearly, the algorithm runs in polynomial time to the number of the states.

4.4.3 Applications

How are the basic set operations given above important in practice? In this section, we look briefly at some straightforward uses of these operations in maintaining and manipulating schemas. Later, in Chapter 7, we will see much heavier uses in XML typechecking.

The union operation is useful when a document comes from multiple possible sources. For example, we can imagine a situation where a server program on a network receives request packets from two types of client programs and these packets conform to different schemas S_1 and S_2. In this case, the server can assume that each received packet from whichever source conforms to the union $S_1 \cup S_2$.

The intersection operation is useful when a document is required to conform to several constraints at the same time. Its usefulness becomes even greater when these constraints are defined independently. For example, we can imagine combining a schema S_1 representing a standard format, such as XHTML or MathML, and another schema representing an orthogonal constraint S_2 such as "a document with height less than 6" or "a document containing no element from a certain name space." Then, an application program exploiting both constraints can require input documents to conform to the intersection $S_1 \cap S_2$ in a natural way. We can further combine intersection with an emptiness test to obtain the disjointness test $S_1 \cap S_2 \stackrel{?}{=} \emptyset$. This operation is useful, for example, when a check is required of whether a schema contains some document that breaks a certain restriction represented by another schema.

Complementation is most often used in conjunction with intersection, that is, the difference operation $S_1 \setminus S_2$ $(= S_1 \cap \overline{S_2})$. Difference is useful when we want to exclude, from a "format" schema S_1, documents that break the restriction expressed by a "constraint" schema S_2. Further, combining intersection, complementation, and the emptiness test yields the containment test $S_1 \stackrel{?}{\subseteq} S_2$ $(\Leftrightarrow S_1 \cap \overline{S_2} \stackrel{?}{=} \emptyset)$. One typical use of containment is, after an existing schema S_1 has been modified to S_2, to check whether S_2 has been "safely evolved" from S_1, that is, whether every document in the old format S_1 also conforms to the new format S_2. As we will see in Chapter 7, the containment test is extremely important in typechecking. For example, when a value known to have type S_1 is given to a function expecting a value of type S_2, we can check the safety of this function application by examining whether $S_1 \subseteq S_2$. In other words, containment can be seen as a form of "subtyping"

and is often found in popular programming languages (e.g., Java). Despite the importance of the containment check, it is known that in the worst case this operation necessarily takes exponential time.

Theorem 4.4.5 (Seidl 1990) The containment problem of tree automata is EXPTIME-complete.

In Chapter 8 we will see an "on-the-fly" algorithm that deals with this high complexity by targeting certain practical cases.

4.4.4 Useless-state elimination

Before closing this chapter, we present a simple operation on a tree automaton that eliminates "useless" states. This operation is rather technical (and is not even a set operation), but it will often be used in later chapters.

First, a state is *useful* when there is a tree on which some successful run assigns the state to some node; a state that is not useful is *useless*. There are in fact two cases in which useless states arise. The first occurs when the state is empty, that is, it accepts no tree. The second occurs when the state is not *reachable* from an initial state; a state q is said to be reachable from another state q' when there is a sequence $q_1, \ldots, q_n$ of states such that $q_1 = q'$ and $q_n = q$ and, for each $i = 1, \ldots, n-1$, there is a transition of either the form $q_i \to a(q_{i+1}, r)$ or the form $q_i \to a(r, q_{i+1})$ for some a and r.

Given a tree automaton $A = (Q, I, F, \Delta)$, useless states can be eliminated by the following two steps.

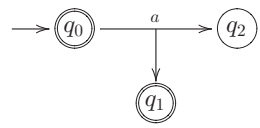
(1) **Removal of empty states** Perform the emptiness test algorithm given in Section 4.4.2 and, using the internal variable $Q_{\mathbf{NE}}$ obtained at the end of this algorithm, construct the automaton $A' = (Q', I', F', \Delta')$ where

$$\begin{array}{lcl} Q' & = & Q_{\mathbf{NE}} \\ I' & = & I \cap Q' \\ F' & = & F \cap Q' \\ \Delta' & = & \{q \to a(q_1, q_2) \in \Delta \mid q, q_1, q_2 \in Q'\}. \end{array}$$

(2) **Removal of unreachable states** From the automaton A' construct the automaton $A'' = (Q'', I'', F'', \Delta'')$, where

$$\begin{array}{lcl} Q'' & = & \{q \in Q' \mid \exists q_0 \in I'.\ q \text{ is reachable from } q_0\} \\ I'' & = & I' \cap Q'' \\ F'' & = & F' \cap Q'' \\ \Delta'' & = & \{q \to a(q_1, q_2) \in \Delta' \mid q, q_1, q_2 \in Q''\}. \end{array}$$

It is important to do these two steps in the order given. If we swap them, useless states may not be completely eliminated. For example, consider the following automaton.



The state q_2 is empty and therefore, after eliminating this state (and the connected transition), we can remove the unreachable state q_1. But if we first perform the removal of unreachable states (which cannot eliminate any state at this moment) then the removal of empty states will eliminate q_2 but still leave the useless state q_1.

Exercise 4.4.6 (⋆⋆) Show that the above procedure eliminates all useless states. ■

4.5 Bibliographic notes

A well-known, comprehensive, reference to tree automata theory is *Tree Automata Techniques and Applications*, available on-line (Comon *et al.*, 2007). This also covers various theoretical results and related frameworks that go beyond the scope of the present book. An automata model that deals directly with unranked trees has been formalized under the names *hedge automata* (Brüggemann-Klein *et al.*, 2001) and *forest automata* (Neumann and Seidl, 1998). The tree automata construction given in Section 4.2 depends on a conversion algorithm from regular expressions to string automata. A useful taxonomy on this topic is Watson (1993). A backward conversion algorithm from string automata to regular expressions, assumed in Exercise 4.2.3, can be found in Hopcroft and Ullman (1979).

5

Pattern matching

So far, we have focused on how to validate data using schemas and tree automata. Our next interest is how to process them and, in particular, how to extract subtrees from an input tree, with a condition specified by the user. In this chapter, we introduce the notion of *patterns* as a straightforward adaptation of schemas. We address the issues of ambiguity and linearity intrinsic in patterns, and describe several kinds of semantics that treat these issues in different ways. A formal definition is given at the end of the chapter.

5.1 From schemas to patterns

Schemas validate documents by constraints such as tag names and positional relationships between tree nodes. Patterns simply add to these constraints the functionality of subtree extraction.

Syntactically, patterns have exactly the same form as schemas except that *variable binders* of the form "`... as x`" can be inserted in subexpressions. To give an example, let us assume that "family tree" type definitions are given as in Section 3.2:

```
Person   = person[Name, Gender, Spouse?, Children?]
Name     = name[String]
Gender   = gender[Male | Female]
Male     = male[]
Female   = female[]
Spouse   = spouse[Name, Gender]
Children = children[Person+]
```

Then, we can write the following pattern

```
person[Name as n,
       gender[(Male|Female) as g],
       Spouse?, Children?]
```

where binders for the variables `n` and `g` are inserted around the expressions `Name` and `(Male|Female)`, respectively. Given an input value, this pattern works as follows.

1. It first checks that the input value has the following type:

   ```
   person[Name, gender[(Male|Female)], Spouse?, Children?]
   ```

   The type is obtained by removing all variable binders from the pattern. If this check fails, then the pattern matching itself fails.
2. If the above check succeeds then the pattern binds the variables `n` and `g` to the subparts of the input value that match `Name` and `(Male|Female)`, respectively.

From the above informal description, the semantics of patterns may look rather obvious. However, it is actually not so simple at a more detailed level. In particular, it is not immediately clear what to do when a pattern can yield several possible matches ("ambiguity") or when it can bind a single variable to more than one value ("non-linearity"). Indeed, these are the main sources of complication in the design space of pattern matching, which is the primary interest of this chapter.

5.2 Ambiguity

Ambiguity refers to the property that, from some input, there are multiple ways of matching. Since the basis of patterns is regular expressions, patterns can be ambiguous. For example, the following is ambiguous:

```
children[Person*, (Person as p), Person*]
```

Indeed, this pattern matches any `children` containing one or more `persons`, but it can bind the variable `p` to a `person` at an arbitrary position.

Then, what should be the semantics of such ambiguous patterns? Figure 5.1 gives a summary of various styles that can be considered. The major categories are

- the *all-matches* semantics, returning the set of all possible matches, and
- the *single-match* semantics, returning only one of the possible matches.

Which style should be taken depends on how patterns are to be used. When they are used as a part of a programming language, single-match semantics is preferred since evaluation usually proceeds with a single binding of each variable. However, all-matches semantics is more natural in a query language since it usually returns a *set* of results that satisfy a specified condition. Let us review each style in more detail.

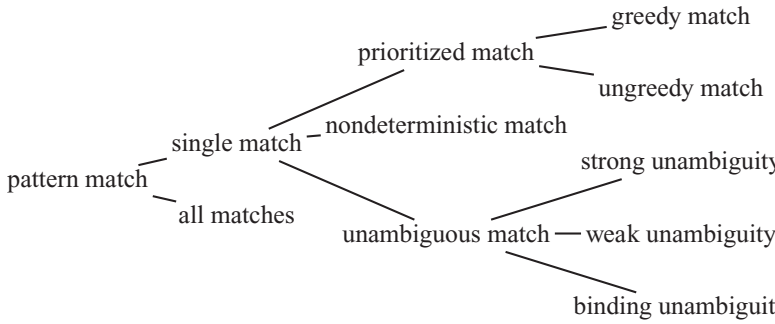


Figure 5.1 Categorized matching policies.

5.2.1 All-matches semantics

As already mentioned, all-matches semantics returns the set of all possible bindings. For example, consider the following pattern:

```
children[Person*,
         person[Name as n,
                gender[(Male|Female) as g],
                Spouse?, Children?],
         Person*]
```

Following the informal description of semantics in Section 5.1, we can see that this pattern matches a value of the type `children[Person+]` (which is equivalent to the type obtained after removing variables from the pattern above) and returns the set of all bindings each mapping `n` to the name and `g` to the gender of a `person`. For example, for the input data

```
children[
  person[
    name["Ichiro"], gender[male[]]],
  person[
    name["Sachiko"], gender[female[]],
    spouse[
      name["Hideki"], gender[male[]]]],
  person[
    name["Umeko"], gender[female[]]]]
```

the pattern gives back the following set:

$$\left\{\begin{array}{ll}\{\texttt{n} \mapsto \texttt{name["Ichiro"]}, & \texttt{g} \mapsto \texttt{male[]}\},\\ \{\texttt{n} \mapsto \texttt{name["Sachiko"]}, & \texttt{g} \mapsto \texttt{female[]}\},\\ \{\texttt{n} \mapsto \texttt{name["Umeko"]}, & \texttt{g} \mapsto \texttt{female[]}\}\end{array}\right\}$$

The most interesting kind of example using the all-matches semantics involves retrieving data from deep places. For example, consider the following entire family tree document:

```
person[
  name["Taro"], gender[male[]],
  spouse[
    name["Hanako"], gender[female[]]],
  children[
    person[
      name["Ichiro"], gender[male[]]],
    person[
      name["Sachiko"], gender[female[]],
      spouse[
        name["Hideki"], gender[male[]]]],
    person[
      name["Umeko"], gender[female[]]]]]
```

How can we retrieve, from such data, all combinations of the `name` of a `person` that has a `spouse` and the name of that `spouse`? For this, we may first write the following recursive definitions:

```
PersonX   = person[Name as n, Gender, SpouseX, Children?
                 | Name, Gender, Spouse?, ChildrenX]
SpouseX   = spouse[Name as s, Gender]
ChildrenX = children[Person*, PersonX, Person*]
```

(we have not introduced "pattern definitions" yet, but nothing prevents us from writing these down since we have said that patterns are schemas with variable binders). Then, what remains is to match the input value against the pattern `PersonX`.

To understand how this pattern works, first look more closely at the definition of `PersonX`, which has two clauses in the content of `person`. The first clause binds `n` to the `name` element of `person` and, through `SpouseX`, binds `s` to the `name` element of `spouse`. Note that this clause can be matched only when the `spouse` is present. The second clause does not bind `n` or `s` immediately, but instead chooses, in `ChildrenX`, one of the `person` elements in the `children` element, from which other possible matches can be looked for recursively with `PersonX`.

Exercise 5.2.1 (⋆) What is the result of the last pattern matching? ■

5.2.2 Single-match semantics

As mentioned, single-match semantics returns one of the possible matches. Unlike all-matches semantics, we need to specify which match is returned, and this gives rise to various styles of matching policy. Below we introduce three styles, namely, prioritized match, nondeterministic match, and unambiguous match.

In the *prioritized-match* style, a built-in set of priority rules is used for uniquely determining which match to take. This semantics can have further variety in the choice of priority rules, though a typical design is *greedy match*, which works roughly as follows.

- For a choice pattern $P_1 \mid P_2$, the earlier matching pattern is taken.
- For a concatenation pattern P_1, P_2, the priority of the earlier pattern is considered first.
- For a repetition pattern $P*$, the pattern P is used as many times as possible to match the whole sequence.

For example, in the pattern

```
children[Person*, (Person as p), Person*]
```

the first `Person*` captures all the occurrences of `person` but the last and therefore the variable `p` is bound to the last occurrence of `person`. We could also think of ungreedy match (the opposite of greedy match) or of providing two separate operators for greedy and ungreedy matches, though we will not pursue these possibilities here.

In the *nondeterministic-match* style, an arbitrary match is chosen from all the possibilities. While prioritized match endows the user with full control in matching, nondeterministic match removes this from the user but instead gives full freedom to the system. This means that there can be several implementations that behave differently, about which the user should not complain! The advantages of this design choice are twofold. First, the semantics becomes vastly simpler since we only have to say "some match" whereas in prioritized match we need a complicated description for the precise behavior, as will be seen in Section 5.4. Second, in the nondeterministic-match style, implementation becomes much easier, in particular when we want to perform a static analysis on patterns. For example, in Section 7.3 we will see a certain type inference method for patterns, where, if we used prioritized match, we would also need to take priorities into account to retain the precision of the analysis.

In the *unambiguous-match* style the pattern is statically required to have at most one match for any input. The unambiguous-match semantics might be preferable

when one considers that an ambiguity is a sign of the user's mistake. For example, it is likely that the ambiguous pattern

```
children[Person*, (Person as p), Person*]
```

is actually an error in either of the two single-match semantics given already. Indeed, in the nondeterministic semantics it would not make sense to extract an arbitrary `person` from the sequence – the user should specify more precisely which `person` (e.g., the first or the one satisfying a certain condition). In the greedy match, a more natural way of extracting the last `person` is to write the following:

```
children[Person*, (Person as p)]
```

However, it occasionally happens that writing ambiguous patterns is reasonable. One typical case is when the application programmer knows, from a certain implicit assumption, that there is only one possible match. For example, suppose that we have a simple book database of the type `Book*`, where `Book` is defined as follows.

```
Book = book[key[String], title[String], author[String]]
```

Let us further make the (implicit) assumption that there is only one `book` entry with the same `key` field value. Then the following ambiguous pattern can uniquely extract a book entry with a specified key from this database:

```
Book*,
book[key["Hosoya2001"],
     title[String as t],
     author[String as a]],
Book*
```

Since writing an unambiguous pattern is sometimes much more complicated than writing an ambiguous one, requiring disambiguation even in unnecessary situations can be a heavy burden for the user. Therefore it can be more user-friendly to signal only a warning, rather than an error, when an ambiguous pattern is encountered.

Several definitions of ambiguity ensure the uniqueness of binding, some of which are shown in Figure 5.1. In Chapter 14, we will detail these definitions and corresponding checking algorithms.

5.3 Linearity

Linearity is the property of a pattern whereby, in any match, each variable is bound to exactly one value. All the examples of patterns shown so far are linear. The linearity constraint is quite natural when we adopt a pattern-match facility as a part

of a programming or query language, in which usually we expect a variable to be bound only to a single value. (Note that a query language normally involves a set of bindings, but *each* binding maps a variable to a single value.)

If we do not impose a linearity restriction, this endows a variable with the capability of capturing a list of subtrees. This may not be useful in the all-matches semantics since it already has a similar functionality and allowing both would be too confusing. However, non-linearity can be useful in a single-match semantics. For example, to extract all `spouses`' `names` from a family tree document, we can use the following recursive pattern.

```
PersonX   = person[Name, Gender, SpouseX, ChildrenX]
SpouseX   = spouse[Name as n, Gender]
ChildrenX = children[PersonX+]
```

Note that, unlike the all-matches patterns shown in Section 5.2.1, the above pattern has only a single match but may capture many `name` nodes for the variable `n`.

The above example shows that recursive patterns in a single-match semantics are useful with nonlinear patterns. It is worth noting that this is not quite the case with linear patterns, however. To see this, let us consider how we can collect subtrees with a single-match linear pattern. First, since we do not have a built-in "collection" capability, let us accept the inconvenience of combining pattern matching with an iteration facility equipped in an underlying language (e.g., with recursive functions or `while` loops). Then, for example, in order to process each `person` from a sequence of type `Person*`, we can use the following pattern,

```
(Person as p), (Person* as r)
```

to extract the first `person` and the remainder sequence and then continue iterating over the latter sequence. Let us extend this example for a deep extraction, obtaining all `spouse names` from a family tree document. We cannot use a pattern like the recursive one in Section 5.2.1, because, although we can obtain *one* `spouse`'s `name`, we have no way to find another after this. The best we could do would be to write a function where each call extracts a `name` from the top `spouse` node (if any), using a pattern such as

```
person[Name, Gender,
       spouse[Name as n, Gender],
       children[Person+ as ps]]
```

and the same function recursively calls itself to traverse each `person` in `children` nodes. Note that to do this we need only shallow patterns that involve no recursive pattern definitions. Thus, under the linearity constraint and the single-match semantics, recursively defined patterns have very little chance of being useful.

Despite the potential usefulness of nonlinear patterns, they are rather tricky to make use of, since, as already mentioned, an underlying language is usually designed in such a way that a variable is bound to only one value, while a nonlinear pattern binds a variable to a list of values. However, there have been a couple of proposals in the literature to bridge this gap.

Sequence-capturing semantics In this semantics a pattern actually returns a binding of each variable to the *concatenation* of the values matched by the variable. For example, the following binds the variable `t` to the sequence of all the `tel` elements in a given sequence of `tel` and `email` elements:

```
((tel[String] as t) | email[String])*
```

Note that this feature is useful when we know how each matched value is separated in the resulting concatenation. In the above case it is clear, since each matched value is a single `tel` element. However, if we write the following pattern,

```
((entry[String]* as s), separator[])*
```

to collect all the consecutive sequences of `entry` elements between `separator` elements, then the result of the match loses the information of where these sequences are separated. Nevertheless, this approach is handy and, in particular, its implementation does not involve any additional difficulties in comparison with linear patterns. (We will discuss more on the implementation in Chapter 6.)

Iterator support In this proposal, a variable bound to a list of values must be used with a built-in iterator. For example, one can think of a language construct `iter` x `with` e (where e is an expression in the language) that works as follows. When x is bound to a list consisting of values $v_1, \ldots, v_n$, then the result of the `iter` expression is the concatenation of e evaluated under the binding of x to v_i, for each $i = 1, \ldots, n$. For example, the following,

```
((entry[String]* as s), separator[])*
   ->  iter s with chunk[s]
```

generates a sequence of `chunk` elements each containing an extracted sequence of consecutive `entry` elements. This approach provides much more expressiveness than the previous approach, although an extra effort would be needed for the implementation. We will not pursue this possibility in this book.

5.4 Formalization

Let us begin by defining the syntax for pattern matching. Assume a set of *pattern names*, ranged over by Y and a set $\mathcal{X}$ of variables, ranged over by x. Then, a *pattern*

P is defined by the following grammar:

$$
\begin{array}{rcll}
P & ::= & () & \text{empty sequence} \\
 & & a[P] & \text{label pattern} \\
 & & P \mid P & \text{choice pattern} \\
 & & P, P & \text{concatenation pattern} \\
 & & P^* & \text{repetition pattern} \\
 & & Y & \text{pattern name} \\
 & & P \text{ as } x & \text{variable binder}
\end{array}
$$

Note that the only additional ingredient compared with the grammar for types is variable binders.

A *pattern schema* is a pair (F, Y), where F is a *pattern definition*, or mapping from pattern names to patterns, of the form

$$\{Y_1 = P_1; \ldots; Y_n = P_n\}.$$

As in the case of types, we require that the start pattern name Y be one of the declared pattern names $Y_1, \ldots, Y_n$, and that each P_i contains only those declared pattern names. We also require the well-formedness of pattern definitions in a similar way to that for type definitions. In examples we often implicitly incorporate, as part of a pattern definition, a type definition that has been introduced already.

As previously discussed, there are several styles in the matching policies of patterns. We first consider a simple semantics that can account for all the described matching policies except for the greedy match. To treat the greedy match we give a more involved semantics afterwards.

5.4.1 Simple semantics

The semantics is given by the *matching* relation, written $F \vdash v \in P \Rightarrow V$. This relation should be read informally as "under pattern definition F, value v matches pattern P and yields binding V." Here, a *binding* V is a sequence of pairs of variables and values, each of the form $(x \mapsto v)$. Such bindings are allowed to contain multiple pairs for the same variable in order that nonlinear patterns can be treated. If each variable appears only once then a binding is said to be *linear*, in which case it can be regarded as a mapping. The matching relation is defined by

the following set of inference rules:

$$\frac{}{F \vdash () \in () \Rightarrow \epsilon}\ \text{P-Eps}$$

$$\frac{F \vdash v \in P \Rightarrow V}{F \vdash a[v] \in a[P] \Rightarrow V}\ \text{P-Elm}$$

$$\frac{F \vdash v \in P_1 \Rightarrow V}{F \vdash v \in P_1 \mid P_2 \Rightarrow V}\ \text{P-Alt1}$$

$$\frac{F \vdash v \in P_2 \Rightarrow V}{F \vdash v \in P_1 \mid P_2 \Rightarrow V}\ \text{P-Alt2}$$

$$\frac{F \vdash v_1 \in P_1 \Rightarrow V_1 \qquad F \vdash v_2 \in P_2 \Rightarrow V_2}{F \vdash v_1, v_2 \in P_1, P_2 \Rightarrow V_1, V_2}\ \text{P-Cat}$$

$$\frac{F \vdash v_i \in P \Rightarrow V_i \qquad \forall i = 1, \ldots, n \qquad n \geq 0}{F \vdash v_1, \ldots, v_n \in P^* \Rightarrow V_1, \ldots, V_n}\ \text{P-Rep}$$

$$\frac{F \vdash v \in F(X) \Rightarrow V}{F \vdash v \in X \Rightarrow V}\ \text{P-Name}$$

$$\frac{F \vdash v \in P \Rightarrow V}{F \vdash v \in P \text{ as } x \Rightarrow (x \mapsto v), V}\ \text{P-As}$$

These rules are very similar to those for the conformance relation given in Section 3.2, except for the treatment of the binding. In the rule P-Eps, the empty sequence value () unconditionally matches the empty sequence pattern () with the empty binding. In the rule P-Elm, a labeled value $a[v]$ matches a labeled pattern $a[P]$ with a binding V when the inner value v matches the inner pattern P with the same binding V. The rules P-Alt1, P-Alt2, and P-Name can be read similarly. In the rule P-Cat, the resulting binding is the concatenation of the bindings yielded by the two submatches. The rule P-Rep can be read similarly. The only rule that substantiates a binding is P-As, where the binding yielded from the sub-match is prepended with the binding of the target variable x to the current value v. With the matching relation in hand, the matching of a value v against a pattern schema (F, Y) is defined by $F \vdash v \in F(Y) \Rightarrow V$.

Several matching policies can be treated by the above-defined matching relation. In the nondeterministic semantics we return an arbitrary binding V satisfying the relation $F \vdash v \in P \Rightarrow V$. In the unambiguous semantics, a pattern is guaranteed to have only one binding V such that $F \vdash v \in P \Rightarrow V$, and therefore we adopt

this binding. In the all-matches semantics, we collect the set of all bindings V such that $F \vdash v \in P \Rightarrow V$.

As already mentioned, linearity may be imposed on a pattern; then any yielded binding is linear. Let $\mathbf{Var}(P)$ be the set of all variables occurring in P and those arising after any chain of references has been followed. The linearity restriction consists of the following syntactic constraints.

- For a concatenation pattern P_1, P_2, no common variable occurs in P_1 and in P_2, that is, $\mathbf{Var}(P_1) \cap \mathbf{Var}(P_2) = \emptyset$.
- For a choice pattern $P_1 \mid P_2$, the same set of variables occurs in P_1 and in P_2, that is, $\mathbf{Var}(P_1) = \mathbf{Var}(P_2)$.
- For a binder pattern (P as x), no x occurs in P, that is, $x \notin \mathbf{Var}(P)$.
- For a repetition pattern P^*, no variable occurs in P, that is, $\mathbf{Var}(P) = \emptyset$.

By this syntactic linearity condition we can ensure that, in any binding yielded by a pattern, each variable occurring in the pattern is mapped to exactly one value.

5.4.2 Greedy-match semantics

To treat the greedy match, we first define *priority ids* I as sequences from $\{1, 2\}^*$. We will use priority ids to tag different matches and give an ordering among them. The semantics is now given in terms of the *prioritized matching* relation $F \vdash v \in P \Rightarrow V;\ I$. This should informally be read as "under pattern definition F, value v matches pattern P and yields binding V and priority id I." This relation is defined by the following set of inference rules:

$$\frac{}{F \vdash () \in () \Rightarrow ();\ \epsilon}\ \text{PP-Eps}$$

$$\frac{F \vdash v \in P \Rightarrow V;\ I}{F \vdash a[v] \in a[P] \Rightarrow V;\ I}\ \text{PP-Elm}$$

$$\frac{F \vdash v \in P_1 \Rightarrow V;\ I}{F \vdash v \in P_1 \mid P_2 \Rightarrow V;\ 1, I}\ \text{PP-Alt1}$$

$$\frac{F \vdash v \in P_2 \Rightarrow V;\ I}{F \vdash v \in P_1 \mid P_2 \Rightarrow V;\ 2, I}\ \text{PP-Alt2}$$

$$\frac{F \vdash v_1 \in P_1 \Rightarrow V_1;\ I_1 \qquad F \vdash v_2 \in P_2 \Rightarrow V_2;\ I_2}{F \vdash v_1, v_2 \in P_1, P_2 \Rightarrow V_1, V_2;\ I_1, I_2}\ \text{PP-Cat}$$

$$\frac{F \vdash v_i \in P \Rightarrow V_i;\ I_i \qquad 0 \le i \le n}{F \vdash v_1, \ldots, v_n \in P^* \Rightarrow V_1, \ldots, V_n;\ 1, I_1, \ldots, 1, I_n, 2}\ \text{PP-Rep}$$

$$\frac{F \vdash v \in F(X) \Rightarrow V;\ I}{F \vdash v \in X \Rightarrow V;\ I}\ \text{PP-Name}$$

$$\frac{F \vdash v \in P \Rightarrow V;\ I}{F \vdash v \in P \text{ as } x \Rightarrow (x \mapsto v), V;\ I}\ \text{PP-As}$$

The rules are identical to those in Section 5.4.1 except that each match is given a unique priority id. When we are deciding which match to take, we compare such priority ids by their lexicographic order. That is, given a value v, the greedy match returns the binding V yielded by the matching $F \vdash v \in P \Rightarrow V; I$ with the smallest priority id I in the lexicographic order. Note that the ungreedy match, which behaves in exactly the opposite way to the greedy match, can be defined similarly except that the largest priority id is taken instead.

The treatment of priority ids in the above matching rules implements the priority policy explained informally in Section 5.2.2. That is, for a choice pattern $P_1 \mid P_2$, the resulting priority id has the form $1, I$ when P_1 is matched (PP-Alt1) whereas it has the form $2, I$ when P_2 is matched (PP-Alt2). Note that here the earlier pattern P_1 has a higher priority. For a concatenation pattern P_1, P_2, the resulting priority id has the form I_1, I_2 (PP-Cat); the priority of the earlier pattern is considered first. For a repetition pattern P^*, the resulting priority id has the form $1, I_1, \ldots, 1, I_n, 2$ (PP-Rep); again, the priorities of earlier matches are considered first. In effect P^* uses P as many times as possible to match the whole sequence. In the special case in which P matches only a sequence of fixed length, P behaves as a "longest match."

An example should help the reader to understand how priorities are treated. Consider matching the value

```
name["Ichiro"],name["Sachiko"],name["Umeko"]
```

against the following pattern:

```
Name*, (Name as n), Name*
```

There are three ways of matching the value against the pattern. In the match that binds `n` to the first `name` element, the subpatterns yield the following matching

relations for some I_1, I_2, and I_3:

$$\frac{}{F \vdash \epsilon \in \texttt{Name*} \Rightarrow \epsilon;\ 2}\ \text{PP-Rep}$$

$$\frac{F \vdash \texttt{name[...]} \in \texttt{Name} \Rightarrow \epsilon;\ I_1}{F \vdash \texttt{name[...]} \in (\texttt{Name as n}) \Rightarrow \{\texttt{n} \mapsto \texttt{"Ichiro"}\};\ I_1}\ \text{PP-As}$$

$$\frac{F \vdash \texttt{name[...]} \in \texttt{Name} \Rightarrow \epsilon;\ I_2 \qquad F \vdash \texttt{name[...]} \in \texttt{Name} \Rightarrow \epsilon;\ I_3}{F \vdash \texttt{name[...]}, \texttt{name[...]} \in \texttt{Name*} \Rightarrow \epsilon;\ 1, I_2, 1, I_3, 2}\ \text{PP-Rep}$$

Concatenating all the resulting priority ids, we obtain

$$2, I_1, 1, I_2, 1, I_3, 2$$

for the whole match. Similarly, the matches that bind `n` to the middle and the last `name` elements give

$$1, I_1, 2, I_2, 1, I_3, 2$$

and

$$1, I_1, 1, I_2, 2, I_3, 2$$

respectively. Comparing these three priority ids by lexicographic order, we conclude that the result of the greedy match is the last one.

Exercise 5.4.1 (⋆⋆) Find a pattern in which a repetition pattern does not behave as a longest match even in the greedy-match semantics. ■

Exercise 5.4.2 (⋆⋆) Find a pattern that does not have the smallest priority id, that is, for any priority id yielded by the pattern there is a strictly smaller priority id yielded by the same pattern. ■

5.5 Bibliographic notes

The style of formalization of patterns here (originally called "regular expression patterns") appeared first in Hosoya and Pierce (2002, 2003) as a part of the design of the XDuce language, for which the greedy-match semantics was adopted at that point. Later, the nondeterministic and the unambiguous semantics were investigated (Hosoya, 2003). A reformulation of patterns using the greedy and sequence-capturing semantics is found in the design of the ℂDuce language (Frisch *et al.*, 2008; Benzaken *et al.*, 2003). It was observed by Vansummeren (2006) that the

greedy match does not necessarily coincide with the so-called "longest match." As an extension of regular expression patterns with iterator support, *regular expression filters* were introduced in Hosoya (2006) and ℂDuce's iterators in Benzaken *et al.* (2003). Also, a kind of nonlinear pattern that binds a variable to a list of values was adopted in the design of the bidirectional XML transformation language biXid (Kawanaka and Hosoya, 2006).

6

Marking tree automata

Marking tree automata arise naturally as an automata formalism corresponding to patterns. The concept can be defined as a simple extension of tree automata in which not only are trees accepted but also marks are put on each node. In this chapter, we give two definitions of marking tree automata and show how patterns can be translated into these automata. Efficient algorithms for executing marking tree automata will be discussed in detail in Section 8.2.

6.1 Definitions

A *marking tree automaton* A is a quintuple $(Q, I, F, \Delta, \mathcal{X})$, where Q and I are the same as in the case of tree automata:

- $\mathcal{X}$ is a finite set of *variables*,
- F is a set of pairs of the form $(q, \mathbf{x})$ where $q \in Q$ and $\mathbf{x} \in 2^{\mathcal{X}}$, and
- Δ is a set of transition rules of the form

$$q \to \mathbf{x} : a(q_1, q_2)$$

where $q, q_1, q_2 \in Q$ and $\mathbf{x} \in 2^{\mathcal{X}}$.

The difference from normal tree automata is that each transition or final state is associated with a set of variables. Intuitively, whenever the automaton encounters a transition or a final state, it binds each variable in the set to the current node. In XML terms, a variable on a transition captures a non-empty sequence such as $a[v_1]$, v_2, while a variable on a final state captures an empty sequence (). Note that a set of variables, rather than a single variable, is associated with a transition or final state. The reason is that sometimes we want to bind several variables to the same tree, as in `((P as x) as y)`.

Let us define the semantics of a marking tree automaton $A = (Q, I, F, \Delta, \mathcal{X})$. First, given a tree t, a mapping m from **nodes**(t) to $2^{\mathcal{X}}$ is called a *marking*. Then a

top-down marking run of A on t is a pair (r, m), of a mapping r from $\mathbf{nodes}(t)$ to Q and a marking m, such that

- $r(\epsilon) \in I$, and
- $r(\pi) \to m(\pi) : a(r(\pi 1), r(\pi 2)) \in \Delta$ whenever $\mathbf{label}_t(\pi) = a$ for $\pi \in \mathbf{nodes}(t)$.

A top-down marking run (r, m) is *successful* when $(r(\pi), m(\pi)) \in F$ for each $\pi \in \mathbf{leaves}(t)$. The difference from runs of tree automata is that the marking of a marking run is constrained by the variable set associated with each transition rule or final state. Similarly, a *bottom-up marking run* of A on t is a pair (r, m) such that

- $(r(\pi), m(\pi)) \in F$ for each $\pi \in \mathbf{leaves}(t)$, and
- $r(\pi) \to m(\pi) : a(r(\pi 1), r(\pi 2)) \in \Delta$ whenever $\mathbf{label}_t(\pi) = a$ for $\pi \in \mathbf{nodes}(t)$.

A bottom-up marking run (r, m) is *successful* when $r(\epsilon) \in I$. Again, the definitions of successful top-down and bottom-up marking runs coincide. A marking tree automaton A *matches* a tree t and yields a binding V when there is a successful marking run (r, m) on t and V is the set of all pairs (x, u) such that $x \in m(\pi)$ and $u = \mathbf{subtree}_t(\pi)$.

Example 6.1.1 Let $A_{6.1.1}$ be the marking tree automaton

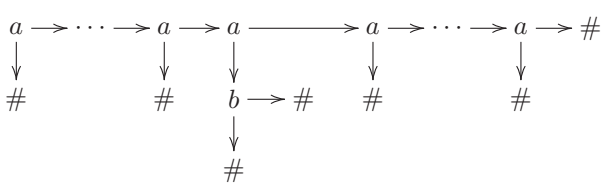
$$(\{q_0, q_1, q_2, q_3\}, \{q_0\}, F, \Delta, \{x\})$$

where

$$\Delta = \left\{ \begin{array}{l} q_0 \to \emptyset : a(q_1, q_0) \\ q_0 \to \emptyset : a(q_2, q_3) \\ q_2 \to \{x\} : b(q_1, q_1) \\ q_3 \to \emptyset : a(q_1, q_3) \end{array} \right\}$$

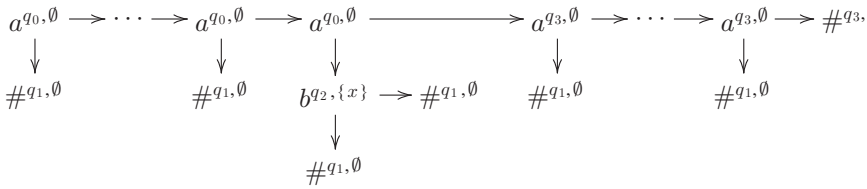
$$F = \{(q_1, \emptyset), (q_3, \emptyset)\}.$$

This automaton matches a tree of the form



and yields the binding $(x, b(\#, \#))$. (We use a notation in which a down arrow points to a left child and a right arrow points to a right child.) Indeed, there is a successful run on the above tree as illustrated in the following, where each node is

associated with a state and a variable set:



■

Analogously to the case of patterns, we sometimes consider markings that use each variable exactly once. Formally, a marking m is *linear* if, for each $x \in \mathcal{X}$, there is a unique node $\pi \in \mathbf{nodes}(t)$ such that $x \in m(\pi)$. Analogously, a binding is linear if each variable appears at most once. A marking tree automaton A is *linear* if, for any tree t and for any successful marking run (r, m) of A on t, the marking m is linear. A binding yielded from a linear marking tree automaton is always linear.

The linearity of a marking tree automaton can be checked syntactically. Let $\mathbf{Var}(q)$ be the union of all the sets of variables associated with the states reachable from q (where reachability is defined as in Section 4.4.4). Then, a syntactic linearity check verifies the following conditions on a given marking tree automaton $(Q, I, F, \Delta, \mathcal{X})$.

(L1) For each $q \in I$, we have $\mathbf{Var}(q) = \mathcal{X}$.
(L2) For each $(q, \mathbf{x}) \in F$, we have $\mathbf{Var}(q) = \mathbf{x}$.
(L3) For each $q \to \mathbf{x} : a(q_1, q_2) \in \Delta$,
  (a) $\mathbf{x}$, $\mathbf{Var}(q_1)$, and $\mathbf{Var}(q_2)$ are pairwise disjoint, and
  (b) $\mathbf{x} \cup \mathbf{Var}(q_1) \cup \mathbf{Var}(q_2) = \mathbf{Var}(q)$.

Intuitively, condition (L1) ensures that from whichever initial state we start all variables will be bound; condition (L2) ensures that any final state is associated with a unique variable set; condition (L3) ensures that, in any transition rule, each variable from $\mathbf{Var}(q)$ is bound uniquely to the current node, some node in the left subtree, or some node in the right subtree.

Lemma 6.1.2 A marking tree automaton is linear if it satisfies all the conditions of the syntactic linearity check.

Exercise 6.1.3 (⋆⋆) Prove Lemma 6.1.2. ■

Exercise 6.1.4 (⋆) Show that the automaton $A_{6.1.1}$ is linear. ■

Lemma 6.1.2 proves the soundness of the syntactic linearity check. The check is also complete, provided that all useless states have been eliminated from the marking tree automaton, as in Section 4.4.4. However, we will not pursue this direction.

As in the case of tree automata, it is useful to allow ϵ-transitions. Formally, a marking tree automaton A with ϵ-transitions is $(Q, I, F, \Delta, \mathcal{X})$, defined similarly to a normal marking tree automaton except that Δ may also contain transition rules of the form

$$q \xrightarrow{\mathbf{x}:\epsilon} q'$$

where $q, q' \in Q$ and $\mathbf{x} \in 2^{\mathcal{X}}$. To give the semantics, we first define

$$\begin{aligned}&\epsilon\text{-}\mathbf{closure}(q_1)\\ &\quad = \{(q_k, \mathbf{x}_1 \cup \cdots \cup \mathbf{x}_{k-1}) \mid q_1 \xrightarrow{\mathbf{x}_1:\epsilon} q_2, \ldots, q_{k-1} \xrightarrow{\mathbf{x}_{k-1}:\epsilon} q_k \in \delta,\ k \geq 1\}.\end{aligned}$$

That is, ϵ-**closure**(q_1) contains pairs consisting of a state reachable by ϵ-transitions and the set of all variables associated with these ϵ-transitions. Then, given a tree t, a (top-down) marking run r of A on t is a pair (r, m), of a mapping r from **nodes**(t) to Q and a marking m, such that

- $r(\epsilon) \in I$, and
- there is a pair $(q, \mathbf{x}) \in \epsilon$-**closure**$(r(\pi))$ such that $q \to \mathbf{x}' : a(r(\pi 1), r(\pi 2)) \in \Delta$ and $m(\pi) = \mathbf{x} \cup \mathbf{x}'$ whenever $\mathbf{label}_t(\pi) = a$ for $\pi \in \mathbf{nodes}(t)$.

A marking run r is *successful* if, for each $\pi \in \mathbf{leaves}(t)$, there is a pair $(q, \mathbf{x}) \in \epsilon$-**closure**$(r(\pi))$ such that $(q, \mathbf{x}') \in F$ and $m(\pi) = \mathbf{x} \cup \mathbf{x}'$. Bottom-up marking runs can be considered analogously.

We can eliminate ϵ-transitions in marking tree automata in the same way as for normal tree automata. Thus, for a marking tree automaton $A = (Q, I, F, \Delta, \mathcal{X})$ with ϵ-transitions, its ϵ-elimination is the marking tree automaton $A' = (Q, I, F', \Delta', \mathcal{X})$, where

$$\begin{aligned}F' &= \{(q', \mathbf{x} \cup \mathbf{x}') \mid q' \in Q,\ (q, \mathbf{x}) \in \epsilon\text{-}\mathbf{closure}(q'),\ (q, \mathbf{x}') \in F\}\\ \Delta' &= \{q' \to (\mathbf{x} \cup \mathbf{x}') : a(q_1, q_2) \mid q' \in Q,\ (q, \mathbf{x}) \in \epsilon\text{-}\mathbf{closure}(q'),\\ &\qquad\qquad q \to \mathbf{x}' : a(q_1, q_2) \in \Delta\}.\end{aligned}$$

Lemma 6.1.5 A marking tree automaton A with ϵ-transitions and its ϵ-elimination A' matches the same set of trees with the same markings.

Proof: The result follows since we can show easily that, by the definition of ϵ-elimination, a successful marking run (r, m) of A on a tree t is also a successful marking run of A' on t and vice versa. □

6.2 Construction

Let us turn our attention to how to construct a marking tree automaton from a pattern. Although the main idea is the same as in the construction of tree automata

from schemas, given in Section 4.2, an additional treatment of variables is needed here.

The first complication arises from "non-tail sequences." First, recall that a marking automaton marks variables on a *node* in the binary tree representation. In the unranked tree representation, such a node corresponds to a *tail sequence*, that is, a sequence that ends at the tail. However, a variable in a pattern can capture an arbitrary intermediate sequence, possibly a non-tail one. For example, the pattern

```
(a[]*,a[b[]]) as x, a[]*
```

matches a value whose binary tree representation is a tree of the form shown in Example 6.1.1 and this binds `x` to the sequence from the beginning up to the `a` element that contains `b`. A *tail variable* is a variable that can be bound only to tail sequences; any other variable is a *non-tail variable*.

In order to handle non-tail variables, a standard trick is to introduce two tail variables, $\mathtt{x}_b$ (the "beginning variable") and $\mathtt{x}_e$ (the "ending variable"), for each variable `x` appearing in the given pattern. The variable $\mathtt{x}_b$ captures the tail sequence starting from the beginning of `x`'s scope (i.e., the sequence captured by `x`), while $\mathtt{x}_e$ captures the tail sequence starting straight after the end of `x`'s scope. For example, the above pattern might be transformed to the following:

$$\texttt{(a[]*,a[b[]], (a[]* as } \mathtt{x}_e\texttt{)) as } \mathtt{x}_b$$

If v_b and v_e are the values to which $\mathtt{x}_b$ and $\mathtt{x}_e$ are bound, respectively, then calculating v such that $v_b = v,\ v_e$ gives us the intermediate sequence that we originally wanted (note that v_e is always a suffix of v_b). Indeed, the last pattern matches, for example, the value

```
a[],a[],a[b[]],a[],a[]
```

and yields the bindings

$$\mathtt{x}_b \mapsto \texttt{a[],a[],a[b[]],a[],a[]} \qquad \mathtt{x}_e \mapsto \texttt{a[],a[]}$$

from which we obtain the following result:

$$x \mapsto \texttt{a[],a[],a[b[]]}$$

Let us present a construction algorithm for marking tree automata. This comprises three steps similar to those in the schema-to-tree-automaton translation given in Section 4.2 except for the additional treatment of variables in the second and third steps. The introduction of the beginning and ending variables, as illustrated above, is done during construction, however.

1. Canonicalize the given pattern and pattern schema. Here, a canonical pattern schema has the form (F, Y_1), where F maps pattern names $\{Y_1, \ldots, Y_n\}$ to canonical patterns and canonical patterns P_c are defined by the following:

$$\begin{array}{rcl} P_c & ::= & () \\ & & a[Y] \\ & & P_c \mid P_c \\ & & P_c, P_c \\ & & P_c{*} \\ & & P_c \text{ as } x \end{array}$$

2. For each i, regarding $F(Y_i)$ as a regular expression over the set of forms "$a[Y]$" in which variable binders may occur as subexpressions, construct a "marking string automaton" $(Q_i, I_i, F_i, \delta_i)$; here δ_i may contain ϵ-transitions, each associated with a set of variables, of the form $q \xrightarrow{\mathbf{x}:\epsilon} q'$. The basis of the construction algorithm is that of McNaughton and Yamada, given in Section 2.2. Except for variable binders, we use the same building rules as in the basic algorithm, where each ϵ-transition is additionally associated with an empty set of variables. For a binder pattern $P'_c = (P_c \text{ as } x)$, we first build an automaton $(Q, \{q_1\}, \{q_2\}, \delta)$ for P_c and then construct the automaton $(Q \cup \{q'_1, q'_2\}, \{q'_1\}, \{q'_2\}, \delta')$ where

$$\delta' = \delta \cup \left\{ q'_1 \xrightarrow{\{x_b\}:\epsilon} q_1,\ q_2 \xrightarrow{\{x_e\}:\epsilon} q'_2 \right\}$$

   with new states q_1, q_2.
3. Merge all the resulting marking string automata into the marking tree automaton $(Q, I, F, \Delta, \mathbf{Var}(Y_1))$ with ϵ-transitions, where

$$\begin{array}{rcl} Q & = & \bigcup_{i=1}^{n} Q_i \\ I & = & I_1 \\ F & = & \bigcup_{i=1}^{n} F_i \\ \Delta & = & \bigcup_{i=1}^{n} \left\{ q_1 \to \emptyset : a(q_0, q_2) \mid q_1 \xrightarrow{a[Y_j]} q_2 \in \delta_i,\ q_0 \in I_j \right\} \\ & & \quad \cup \left\{ q_1 \xrightarrow{\mathbf{x}:\epsilon} q_2 \in \delta_i \right\} \end{array}$$

   and perform ϵ-elimination.

Exercise 6.2.1 (⋆) By using the above algorithm, construct a marking tree automaton from the following pattern:

```
(a[]*,a[b[]]) as x, a[]*
```

■

Exercise 6.2.2 (⋆⋆⋆) If we want a form of marking tree automata in a greedy-match semantics, what changes do we need in the syntax, the semantics, and the construction? ■

6.3 Sequence-marking tree automata

We have introduced a formalism for marking tree automata that is commonly found in the literature. In this section, we describe a variation in which multiple nodes can be marked with the same variable if they are in the same sequence. The intention is that the resulting binding maps each variable to the *concatenation* of the nodes marked with the variable. One advantage of this formalism is that it is easy to encode a form of nonlinear patterns in the sequence-capturing semantics (Section 5.3).

A *sequence-marking tree automaton A* has the same syntax as a normal marking tree automaton except that final states are not associated with variable sets. Also, we use the same notion of marking runs except that we do not check the marks on leaves. A marking m is *sequence-linear* if, for any x, we can find $\{\pi_1, \ldots, \pi_n\} = \{\pi \mid x \in m(\pi)\}$ such that $\pi_{i+1} = \pi_i 2 \cdots 2$ for each $i = 1, \ldots, n-1$. Intuitively, the nodes $\pi_1, \ldots, \pi_n$ appear in the same sequence. Then the *binding* for a sequence-linear marking m on t is a mapping from each variable x to the tree

$$a_1(t_1, a_2(t_2, \ldots, a_n(t_n, \#) \cdots))$$

where the nodes $\pi_1, \ldots, \pi_n$ are those in the definition of sequence-linearity and $t_i = \mathbf{subtree}_t(\pi_i 1)$ for $i = 1, \ldots, n$. Note that, in the case $n = 0$ (that is, when no mark of x is present anywhere), x is bound simply to #. A sequence-marking tree automaton is *linear* if the marking m is sequence-linear for any successful marking run (r, m) on any tree.

The construction of a sequence-marking tree automaton is simpler than the construction given in Section 6.2. We need the following changes.

- In the second step, we construct a marking string automaton where each label transition (but not ϵ-transition) is associated with a set of variables. The construction is then similar except that, for a binder pattern (P_c as x), we modify the automaton built for P_c by adding x to the variable set of every label transition.
- In the third step, when performing ϵ-elimination on the resulting marking string automata we need not consider variable sets on ϵ-transitions.

Exercise 6.3.1 (⋆) By using the above algorithm, construct a sequence-marking tree automaton from the following pattern:

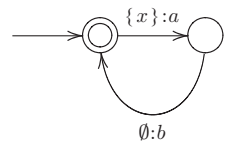
```
(a[]*,a[b[]]) as x, a[]*
```

■

Note that this construction already allows us to encode nonlinear patterns with the sequence-capturing semantics (Section 5.3). For example, the following nonlinear pattern

```
((a[] as x), b[])*
```

which collects all occurrences of `a` in a given sequence, can be represented by the following marking automaton (eliding the "content" states for brevity):



It is also guaranteed that a constructed automaton is sequence-linear if, in the input nonlinear pattern, each variable captures only elements that appear in the same sequence.

6.4 Bibliographic notes

It is a common practice to extend automata to have a capability to put marks on nodes. Such automata are typically used for extracting a set of nodes (Murata, 2001; Koch, 2003; Neven and Schwentick, 2002; Hosoya and Pierce, 2002) or a set of tuples of nodes (Berlea and Seidl, 2002; Planque *et al.*, 2005; Flum *et al.*, 2002; Inaba and Hosoya, 2007). The present chapter has used a style of marking automata in which variable sets are associated with each transition or final state. However, another style in which variable sets are associated with each state is also common (Murata, 2001; Koch, 2003; Neven and Schwentick, 2002). The construction of marking tree automata based on partial derivatives of regular expressions (Antimirov, 1996) can be found in Hosoya and Pierce (2002). There has also been research on efficient compilations from marking tree automata to lower-level code (Levin, 2003; Levin and Pierce, 2005).

7

Typechecking

Typechecking is a static analysis on programs that enables the detection of potential dynamic type errors. Since it is already a standard technique in programming languages and since XML's schemas are analogous to data types in programming languages, we might expect existing typechecking techniques to be usable also for the verification of XML processing programs. However, a concrete analysis algorithm must be renewed entirely since schemas have a mathematical structure completely different from conventional data types. In this chapter, we first overview various approaches to XML typechecking and then consider, as a case study, a core of the XDuce type system in detail.

7.1 Compact taxonomy

Since the aim of typechecking is the rejection of possibly incorrect programs, it is a language feature clearly visible to the user. Therefore, in constructing a typechecker we must take into account not only the internal analysis algorithm but also the external specification, which further influences the design of the whole language.

The first design consideration is the kind of type error that should be detected. That is, what sort of type error should be raised from a program in the target language? The most typical kind of error occurs when the final result does not conform to the specified "output type." However, some languages also check intermediate results against types. For example, the XDuce language has a pattern match facility, according to which a match error occurs when a form of conformance test on an intermediate result fails (Section 5.1).

The second design consideration is the precision of typechecking. From the theory of computation it is well known that no algorithm can exactly predict the behavior of a program written in a general language, that is, in a language equivalent to Turing machines. Thus there are two ways of creating a usable

typechecker: (1) use exact typechecking for a restricted language; or (2) use approximate typechecking for either a general or a restricted language.

7.1.1 Exact typechecking

In this approach, the principal question is how much we need to restrict the language in order to make typechecking exact yet decidable. In general, the more expressive the language, the more difficult the typechecking algorithm becomes. Thus there has been a stream of theoretical research to discover the optimum; it is known as the "XML typechecking problem." Here, typechecking is formulated as follows: given a program P, an input type τ_I, and an output type τ_O, decide whether

$$P(\tau_I) \subseteq \tau_O$$

where $P(\tau_I)$ is the set of results from applying P to inputs of type τ_I. As types, regular tree languages or their subclasses (represented as some kind of schema or automaton) are commonly used.

There are two major approaches to exact typechecking, *forward inference* and *backward inference*. Forward inference solves the typechecking problem directly. That is,

1. compute $\tau'_O = \{P(t) \mid t \in \tau_I\}$ from P and τ_I, and
2. test whether $\tau'_O \subseteq \tau_O$.

it is well known, however, that forward inference does not work once the target language has a certain level of expressiveness since the computed output type τ'_O does not fit in the class of regular tree languages; it can even go beyond so-called context-free tree languages, in which case the test $\tau'_O \subseteq \tau_O$ becomes undecidable.

Backward inference works in the opposite way:

1. compute $\tau'_I = \{t \mid P(t) \in \tau_O\}$ from P and τ_O, that is, the set of inputs from which P yields outputs of type τ_O, and
2. test whether $\tau_I \subseteq \tau'_I$.

Researchers have found that, for certain target languages where forward inference does not work, the computed input type τ'_I becomes regular and thus typechecking can be decided. In Chapter 11 we will describe details of the backward inference approach.

7.1.2 Approximate typechecking

While exact typechecking is theoretically interesting, a more realistic approach is to estimate run-time type errors conservatively. In this approach, programs

that have gone through typechecking successfully are guaranteed to be correct; however, some that are actually innocent might have been refuted. The user has the right to ask why a program got rejected and what changes would be needed for it to be accepted. Here, the clearer the explanation, the more user-friendly is the language. Thus the main challenge in this approach is to develop a typechecker whose error-detecting ability is reasonably high yet whose specification – called the *type system* – is readily understandable. Note that in exact typechecking no such issue arises, since "there must be no run-time type error" *is* the specification.

A typical conservative type system uses *typing rules* to assign a type to each subexpression of the program. In this, we usually exploit *type annotations* on variable declarations given by the user, on the basis of which we calculate the types of subexpressions in a bottom-up way. This way of designing a type system makes both the specification and the algorithmics quite simple. However, the amount of type annotations that the user needs to supply tends to be large. Therefore some form of automatic inference of such type annotations is desirable. In Section 7.2 we will see an instructive example of this approach, the μXDuce type system. This type system uses quite conventional and simple typing rules as a basis but adopts a non-trivial mechanism in the inference of type annotations on variables in patterns, in which the computed types are guaranteed to be exact in a similar sense to the exact typechecking introduced in Section 7.1.1.

Since type annotations are usually tedious for the user to write, it is desirable for the type system to infer them completely. Since inferred types cannot be exact in a general-purpose language, they must be approximate. Then the challenge is, again, to obtain a specification that is easy to understand. One approach involves giving empirical evidence that "false negatives" seldom happen, rather than to giving an absolute guarantee of this; theoreticians may not be satisfied but practitioners will find it valuable. Another approach involves finding a reasonable "abstraction" of an original program and performing exact typechecking; although this has not yet been pursued extensively, it has possiblities.

The bibliographic notes list concrete proposals for approximate typechecking found in the literature.

7.2 Case study: μXDuce type system

XDuce is a functional programming language specialized to XML processing. The full language contains a number of features, each of which is of interest in itself. However, in this section we present only a tiny subset, μXDuce, to illustrate how we can build a simple yet powerful type system for XML processing.

7.2.1 Syntax and semantics

We will borrow the definitions of values (v), types (T), type definitions (E), and the conformance relation ($E \vdash v \in T$) from Section 3.2 and incorporate the definitions of patterns (P), pattern definitions (F), bindings (V), and the matching relation in the nondeterministic semantics ($F \vdash v \in P \Rightarrow V$) from Section 5.4.

Assume a set of *function names*, ranged over by f. *Expressions* e are defined by the following grammar:

e	::=	x	variable
		$f(e)$	function call
		$()$	empty sequence
		$a[e]$	labeling
		e, e	concatenation
		match e **with** $P_1 \to e_1 \mid \cdots \mid P_n \to e_n$	pattern match

That is, in addition to the usual variable references and function calls, we have value constructors (i.e., empty sequence, labeling, and concatenation), and value deconstructors (i.e., pattern match). A *function definition* has the following form:

$$\textbf{fun}\ f(x : T_1) : T_2 = e$$

For brevity, we treat only one-argument functions here; the extension to multiple arguments is routine. Note that both the argument type and the result type must be specified explicitly. Then, a *program* is a triple (E, G, e_{main}) of a set E of type definitions, a set G of function definitions, and an expression e_{main} from which to start evaluation. We regard type definitions also as pattern definitions. (Note that this means that no variable binders are allowed in defined patterns; in practice, this causes little problem. See Section 5.2.) We assume that all patterns appearing in a given program are linear. From here on, let us fix on a particular program (E, G, e_{main}).

The operational semantics of μXDuce is described by the evaluation relation $V \vdash e \Downarrow v$, which should be read as "under binding V, expression e evaluates to value v." The relation is defined by the set of rules given below. First, the rules for variables and an empty sequence are straightforward:

$$\frac{}{V \vdash x \Downarrow V(x)}\ \text{EE-Var}$$

$$\frac{}{V \vdash () \Downarrow ()}\ \text{EE-Emp}$$

The rules for labeling and concatenation are also simple:

$$\frac{V \vdash e \Downarrow v}{V \vdash a[e] \Downarrow a[v]}\ \text{EE-Lab}$$

$$\frac{V \vdash e_1 \Downarrow v_1 \qquad V \vdash e_2 \Downarrow v_2}{V \vdash e_1, e_2 \Downarrow v_1, v_2}\ \text{EE-Cat}$$

In EE-Lab, under a binding V a labeling expression $a[e]$ evaluates to a label value $a[v]$ when the inner expression e evaluates to the value v. The rule EE-Cat can be read similarly. The rule for function calls is the following:

$$\frac{V \vdash e_1 \Downarrow v \qquad \textbf{fun}\ f(x : T_1) : T_2 = e_2 \in G \qquad x \mapsto v \vdash e_2 \Downarrow w}{V \vdash f(e_1) \Downarrow w}\ \text{EE-App}$$

That is, when the argument e_1 evaluates to a value v under the binding V, the function call $f(e_1)$ evaluates to a value w if the body expression e_2 also evaluates to w under the binding of the parameter x to the argument value v. The most interesting rule is for pattern matches:

$$\frac{\begin{array}{c} V \vdash e \Downarrow v \\ v \notin P_1 \quad \cdots \quad v \notin P_{i-1} \qquad v \in P_i \Rightarrow W \qquad V, W \vdash e_i \Downarrow w \end{array}}{V \vdash \textbf{match}\ e\ \textbf{with}\ P_1 \to e_1 \mid \cdots \mid P_n \to e_n \Downarrow w}\ \text{EE-Match}$$

That is, a pattern match tries to match the input value v against the patterns P_1 through P_n, from left to right. When the matching fails up to the $(i-1)$th pattern but succeeds for the ith pattern, we obtain the new binding W from the last matching and then evaluate the corresponding body expression e_i under the combined binding V, W; this gives the result of the whole pattern match expression. (Here, $v \in P \Rightarrow W$ is the matching relation in the nondeterministic semantics defined in Section 5.2.2, where the fixed type definition E is omitted from the full form $E \vdash v \in P \Rightarrow W$. Also, $v \notin P$ means "match failure," that is, that there is no W such that $v \in P \Rightarrow W$. We use the same convention in what follows.) Finally, the semantics of the whole program is given by a value v such that $\emptyset \vdash e_{\text{main}} \Downarrow v$. Note that there can be multiple such values owing to our use of nondeterministic pattern matching.

Example 7.2.1 [⋆] Consider the following program:

```
type Person = person[(Name,Tel?,Email*)]
type Result = person[(Name,Tel,Email*)]
```

```
fun filterTelbook (ps : Person*) : Result* =
  match ps with
    person[Name,Email*], (Any as rest)  ->
            filterTelbook(rest)
  | person[Any] as hd, (Any as rest)  ->
            hd, filterTelbook(rest)
  | ()  ->  ()
```

Here, assume that the type names `Name`, `Tel`, and `Email` have been defined somewhere. Also assume that the type `Any` has been defined to denote the set of all values. Answer the following questions. (Ignore the typechecking issue here.)

1. Find the result of applying the function `filterTelbook` to the following value:

```
person[name["Taro"], tel["123-456"], email["taro@xml"]],
person[name["Ichiro"], email["ichiro@tree"], email["w@xyz"]],
person[name["Hanako"], tel["987-654"]]
```

2. The function is applied to the following value:

```
person[name["Taro"], tel["123-456"], email["taro@xml"]],
ortdon[name["Ichiro"], email["ichiro@tree"], email["w@xyz"]],
person[name["Hanako"], tel["987-654"]]
```

   Where and why does it fail?
3. State what happens if the function is now applied to the following value:

```
person[name["Taro"], tel["123-456"], email["taro@xml"]],
person[name["Ichiro"], email["ichiro@tree"], email["w@xyz"]],
person[name["Hanako"], tel["987-654"], tel["987-654"]]
```

■

7.2.2 Typing rules

The type system of μXDuce can be described by two important relations:

- the subtyping relation $S \leq T$, read as "S is subtype of T;"
- the typing relation $\Gamma \vdash e : T$, read as "under type environment Γ, expression e has type T."

Here, a *type environment* Γ is a mapping from variables to types.

The subtyping relation $S \leq T$ allows us to reinterpret a value of type S as type T. This plays an important role in making the type system flexible. For example, by subtyping, functions can accept different types of values. This feature often appears in programming languages, even though each language uses different definitions. In object-oriented languages, the subtyping relation is determined by the class

hierarchy specified by the user. In some functional languages, the subtyping relation is defined by induction on the structure of types, using inference rules. In our setting we define the subtyping relation in terms of the semantics of types:

$$S \leq T \text{ if and only if } v \in S \text{ implies } v \in T \text{ for all } v.$$

This directly formulates our intuition. (Recall that here the relation $v \in S$ omits the fixed type environment E.) In conventional definitions the "only if" direction usually holds (derived as a property), whereas the "if" direction is not expected since decidability is far from clear in the presence of objects or higher-order functions. The situation is different for us: we have only types denoting regular tree languages and therefore our subtyping problem is equivalent to the containment problem for tree automata, as discussed in Section 4.4.3. Though solving this problem takes exponential time in the worst case (Theorem 4.4.5), empirically efficient algorithms exist (see Section 8.3 below).

The typing relation $\Gamma \vdash e : T$ assigns a type to each expression, and thus serves as the main component of the type system. This relation is defined by induction on the structure of expressions and works roughly as follows. Using the type of the parameter variable declared in the function header, we compute the type of each expression from the types of its subexpressions. The typing rules are mostly straightforward except for pattern matches, where we perform several non-trivial operations including an automatic type inference for patterns.

Let us look at the typing rules one by one. The rule for variables is obvious:

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{TE-VAR}$$

The rules for value constructors are also straightforward. The structure of each computed type parallels that of the following expression:

$$\frac{}{\Gamma \vdash () : ()}\ \text{TE-EMP}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash l[e] : l[T]}\ \text{TE-LAB}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1, e_2 : T_1, T_2}\ \text{TE-CAT}$$

The rule for function calls is as follows:

$$\frac{\textbf{fun}\ f(x : T_1) : T_2 = e_2 \in G \qquad \Gamma \vdash e_1 : U \qquad U \leq T_1}{\Gamma \vdash f(e_1) : T_2}\ \text{TE-APP}$$

Note that we check the subtyping of the argument type against the declared parameter type. That is, a function can receive any value as long as it can be reinterpreted as the parameter type. The last typing rule is for pattern matches but, since it is complicated, let us first give an informal explanation. Suppose that, for a pattern match

$$\textbf{match } e \textbf{ with } P_1 \to e_1 \mid \cdots \mid P_n \to e_n$$

we have already calculated type R for the expression e. Then, we perform the following operations.

Exhaustiveness This checks that any value from R is matched by (at least) one of the patterns $P_1, \ldots, P_n$. This can be formulated as

$$R \leq \textbf{tyof}(P_1) \mid \cdots \mid \textbf{tyof}(P_n)$$

where $\textbf{tyof}(P)$ stands for the type obtained after all variable binders have been eleminated from P (i.e., every occurrence of $(P' \text{ as } x)$ has been replaced by P').

Irredundancy This checks that each pattern is matched by some value that belongs to R but does not match any preceding pattern; here, we take the first-match policy into account. The irredundancy check can be done by testing whether

$$R \cap \textbf{tyof}(P_i) \not\leq \textbf{tyof}(P_1) \mid \cdots \mid \textbf{tyof}(P_{i-1})$$

holds for each $i = 1, \ldots, n$. Note that in this we make use of the intersection operation introduced in Section 4.4.3.

Type inference for patterns This computes the "best" type for each variable with respect to the input type R. To specify this more precisely, let us consider inference on the ith case and write

$$R_i = R \setminus (\textbf{tyof}(P_1) \mid \cdots \mid \textbf{tyof}(P_{i-1})).$$

That is, R_i represents the set of values from R that are not matched by preceding patterns. Then, what we infer is a type environment Γ_i satisfying, for each variable x and each value v,

> $v \in \Gamma_i(x)$ if and only if there exists a value u such that $u \in R_i$ and $u \in P \Rightarrow V$ for some V with $V(x) = v$.

Let us write $R_i \vdash P_i \Rightarrow \Gamma_i$ when the above condition is satisfied; a concrete algorithm for obtaining such Γ_i from R_i and P_i will be given in Section 7.3. The "if" direction in the above statement means that any value to which x may be bound is predicted in $\Gamma_i(x)$ and therefore is necessary to construct a sound type system. The "only if" direction, however, means that x may be bound to any predicted value in $\Gamma_i(x)$. This property makes the inference precise and thus provides the best flexibility to the user, avoiding false negatives as much as possible.

Summarizing the above explanation, the typing rule for pattern matches can be written down as follows:

$$\frac{\begin{array}{c}\Gamma \vdash e : R \qquad R \leq \mathbf{tyof}(P_1) \mid \cdots \mid \mathbf{tyof}(P_n) \\ \forall i.\ \left(\begin{array}{l} R \cap \mathbf{tyof}(P_i) \not\leq \mathbf{tyof}(P_1) \mid \cdots \mid \mathbf{tyof}(P_{i-1}) \\ R \setminus (\mathbf{tyof}(P_1) \mid \cdots \mid \mathbf{tyof}(P_{i-1})) \vdash P_i \longrightarrow \Gamma_i \\ \Gamma, \Gamma_i \vdash e_i : T_i \end{array}\right)\end{array}}{\Gamma \vdash \mathbf{match}\ e\ \mathbf{with}\ P_1 \to e_1 \mid \cdots \mid P_n \to e_n : T_1 \mid \cdots \mid T_n}\ \text{TE-MATCH}$$

Here, the type of the complete **match** expression is simply the union of the types $T_1, \ldots, T_n$ of the body expressions.

Having defined the subtyping and typing relations, we can now define the well-typedness of programs. We say that a program (E, G, e_{main}) is well-typed when the function definitions in G are all well-typed and e_{main} is well-typed under the empty type environment, where the well-typedness of a function definition, written $\vdash \mathbf{fun}\ f(x : T_1) : T_2 = e$, is defined by the following rule:

$$\frac{x : T_1 \vdash e : S \qquad S \leq T_2}{\vdash \mathbf{fun}\ f(x : T_1) : T_2 = e}\ \text{TF}$$

Here, we are checking the subtyping between the type S of the body expression and the declared result type T_2; a function can return any value as long as it can be reinterpreted as the result type.

Exercise 7.2.2 (⋆) Continue with the example in Exercise 7.2.1:

```
type Person = person[(Name,Tel?,Email*)]
type Result = person[(Name,Tel,Email*)]

fun filterTelbook (ps : Person*) : Result* =
  match ps with
    person[Name,Email*], (Any as rest)  ->
            filterTelbook(rest)
  | person[Any] as hd, (Any as rest)  ->
            hd, filterTelbook(rest)
  | ()  ->  ()
```

Answer the following questions to verify that the above program typechecks and to convince yourself that even a slight change to the program would cause a problem in typechecking.

1. Perform an exhaustiveness check. Confirm that such a check would fail if the second or the third clause were missing.
2. Perform an irredundancy check. Confirm that such a check would fail if the first and the second clauses were swapped.
3. Perform a type inference on the pattern match. Confirm that it would give different types if the first clause were missing.
4. By using the inferred types, finish the remaining typechecking. Confirm that this would fail if we used the types appearing in the pattern (i.e., `person[Any]` for `hd` and `Any` for `rest`) instead of the inferred types. Confirm further that typechecking would fail if the first clause were missing and thus the inferred types were different. ■

7.2.3 Correctness

Having defined the type system, it remains to establish its correctness, that is, the property that a well-typed program never raises a run-time type error. This section gives a brief sketch, referring to the literature for the full proof.

Correctness consists of two parts: (1) if a well-typed program returns a final result then the value is conformant; and (2) a well-typed program never raises a match error during evaluation.

The first part can precisely be stated as follows:

> for a well-typed program (E, G, e_{main}) with $\emptyset \vdash e_{\text{main}} : T$, if $\emptyset \vdash e_{\text{main}} \Downarrow v$ then $v \in T$.

However, in order for an inductive proof to work we need to generalize the above statement so that bindings and type environments are also considered. To do this, first write $\Gamma \vdash V$ ("a binding V conforms to a type environment Γ") when $\mathbf{dom}(\Gamma) = \mathbf{dom}(V)$ and $V(x) \in \Gamma(x)$ for each $x \in \mathbf{dom}(\Gamma)$. Then, the following can be proved.

Theorem 7.2.3 (Type preservation (Hosoya and Pierce, 2003)) Suppose that $\vdash G$ and $\Gamma \vdash e : T$ with $\Gamma \vdash V$. Then $V \vdash e \Downarrow v$ implies that $v \in T$.

Proof sketch: The proof proceeds by induction on the derivation of $V \vdash e \Downarrow v$. □

To prove the part (2) ("no match error during evaluation") of the definition of correctness, we need to expand the formalization of evaluation slightly, since we need to consider an intermediate state where a match error occurs. Note that it is not sufficient to say that "not $V \vdash e \Downarrow v$," since this would mean either that a match error occurs, or that e goes into an infinite loop yielding no result, which is not an error that we are considering here. Thus, we define the erroneous evaluation

relation $V \vdash e \Downarrow \bot$ by the following set of rules:

$$\frac{V \vdash e \Downarrow \bot}{V \vdash a[e] \Downarrow \bot}\ \text{EN-Lab}$$

$$\frac{\begin{array}{c} V \vdash e_1 \Downarrow \bot \\ \textit{or} \\ V \vdash e_2 \Downarrow \bot \end{array}}{V \vdash e_1, e_2 \Downarrow \bot}\ \text{EN-Cat}$$

$$\frac{\begin{array}{c} V \vdash e_1 \Downarrow \bot \\ \textit{or} \\ V \vdash e_1 \Downarrow v \qquad \textbf{fun}\ f(x : T_1) : T_2 = e_2 \in G \qquad x \mapsto v \vdash e_2 \Downarrow \bot \end{array}}{V \vdash f(e_1) \Downarrow \bot}\ \text{EN-App}$$

$$\frac{\begin{array}{c} V \vdash e \Downarrow \bot \\ \textit{or} \\ V \vdash e \Downarrow v \qquad E \vdash v \not\in P_1 \quad \cdots \quad E \vdash v \not\in P_n \\ \textit{or} \\ V \vdash e \Downarrow v \qquad E \vdash v \not\in P_1 \quad \cdots \quad E \vdash v \not\in P_{i-1} \\ E \vdash v \in P_i \Rightarrow W \quad V, W \vdash e_i \Downarrow \bot \end{array}}{V \vdash \textbf{match}\ e\ \textbf{with}\ P_1 \to e_1 \mid \cdots \mid P_n \to e_n \Downarrow \bot}\ \text{EN-Match}$$

Almost all these rules tell us that an expression yields an error when one of the subexpressions does. The only exception is the second case in the rule EN-Match when the input value v is matched by none of the patterns, constituting a match error. With the relation defined above we can prove that a well-typed program never returns an error.

Theorem 7.2.4 (Progress (Hosoya and Pierce, 2003)) Suppose that $\vdash G$. If $\Gamma \vdash e : T$ with $\Gamma \vdash V$ then $V \vdash e \Downarrow \bot$ is never derived.

Proof sketch: We in fact need to prove the contraposition

if $V \vdash e \Downarrow \bot$ then there are no T and Γ such that $\Gamma \vdash e : T$ with $\Gamma \vdash V$.

The proof proceeds by induction on the derivation of $V \vdash e \Downarrow \bot$. □

It is important to note that the above theorems do not guarantee precision. Indeed, it is incorrect to claim that an expression always evaluating to a value of type T can be given type T. That is, the following does not hold:

under $\vdash G$, if $V \vdash e \Downarrow v$ and $v \in T$ for all V with $\Gamma \vdash V$ then $\Gamma \vdash e : T$.

As a counterexample, consider the expression $e = (x, x)$ and the type environment $\Gamma = x \mapsto (a[] \mid b[])$. Then, under a binding V satisfying $\Gamma \vdash V$, the result value of the expression always conforms to

$$(a[], a[]) \mid (b[], b[]).$$

However, according to the typing rules the type we should give to e is

$$(a[] \mid b[]), (a[] \mid b[])$$

which is larger than the above type. In general, our typing rule for concatenation is not exact since it typechecks each operand independently. Improving such imprecision needs non-trivial efforts; Chapter 11 treats a relevant issue.

Exercise 7.2.5 (⋆⋆) In the type system we have described so far, there are in fact many other sources of imprecision. For instance, (1) a function does not allow multiple combinations of argument and result types even if they work and (2) the inference does not reflect the dependency among values that are bound in different variables. Give concrete examples corresponding to these sources of imprecision. ■

7.3 Type inference for patterns

The type system of μXDuce described in the previous section uses type inference for patterns in order to give the best types to pattern variables. This section presents an algorithm for this type inference.

Since the algorithm works with tree automata, let us first rephrase the specification as follows. Given a tree automaton A (the "input type") and a marking tree automaton B (the "pattern") containing variables $\mathbf{x}_B = \{x_1, \ldots, x_n\}$, obtain a mapping Γ from $\mathbf{x}_B$ to tree automata such that

> $\Gamma(x_i)$ accepts u if and only if there exists t such that A accepts t and B matches t with some binding V, where $V(x_i) = u$.

We will assume that the marking automaton is linear (which is guaranteed if the automaton is converted from a linear pattern).

7.3.1 Algorithm

For simplicity, let us first consider an algorithm that works for patterns that do not contain non-tail variables. We will consider patterns that do contain non-tail variables in Section 7.3.2.

Let $A = (Q_A, I_A, F_A, \Delta_A)$ and $B = (Q_B, I_B, F_B, \Delta_B)$. Then the following algorithm produces a mapping Γ satisfying our specification.

1. Take the product C of A and B, that is, $(Q_C, I_C, F_C, \Delta_C)$ where

$$\begin{array}{lcl} Q_C & = & Q_A \times Q_B \\ I_C & = & I_A \times I_B \\ F_C & = & \{((q,r), \mathbf{x}) \mid q \in F_A,\ (r, \mathbf{x}) \in F_B\} \\ \Delta_C & = & \{(q,r) \to \mathbf{x} : a((q_1,r_1),(q_2,r_2)) \mid q \to a(q_1,q_2) \in \Delta_A, \\ & & \qquad\qquad\qquad\qquad\quad r \to \mathbf{x} : a(r_1,r_2) \in \Delta_B\}. \end{array}$$

2. Eliminate all useless states from C by means of the two steps (the removal of empty states and the removal of unreachable states) given in Section 4.4.4. Let $D = (Q_D, I_D, F_D, \Delta_D)$ be the resulting automaton.
3. Return Γ such that $\Gamma(x_j)$ is the tree automaton $(Q_j, I_j, F_j, \Delta_j)$ with ϵ-transitions, where

$$\begin{array}{lcl} Q_j & = & Q_D \times \{0,1\} \\ I_j & = & \{\langle q_0, 0\rangle \mid q_0 \in I_D\} \\ F_j & = & \{\langle q,k\rangle \mid (q,\mathbf{x}) \in F_D,\ k \in \{0,1\}\} \\ \Delta_j & = & \{\langle q,0\rangle \to a(\langle q_1,1\rangle, \langle q_2,1\rangle) \mid q \to \mathbf{x} : a(q_1,q_2) \in \Delta_D,\ x_j \in \mathbf{x}\} \\ & & \cup\, \{\langle q,0\rangle \xrightarrow{\epsilon} \langle q_1,0\rangle \mid q \to \mathbf{x} : a(q_1,q_2) \in \Delta_D,\ x_j \notin \mathbf{x},\ x_j \in \mathbf{Var}(q_1)\} \\ & & \cup\, \{\langle q,0\rangle \xrightarrow{\epsilon} \langle q_2,0\rangle \mid q \to \mathbf{x} : a(q_1,q_2) \in \Delta_D,\ x_j \notin \mathbf{x},\ x_j \notin \mathbf{Var}(q_1)\} \\ & & \cup\, \{\langle q,1\rangle \to a(\langle q_1,1\rangle, \langle q_2,1\rangle) \mid q \to \mathbf{x} : a(q_1,q_2) \in \Delta_D\} \end{array}$$

   and perform ϵ-elimination.

Since the last step is somewhat complex, an explanation is in order. First we duplicate the whole state space of the automaton D to 0-space and 1-space, where the states are coupled with 0 and 1, respectively. The 0-space captures labels that are matched by variables, while 1-space captures the whole tree below those matched labels. Thus we start with initial states in 0-space. In 0-space, we retain the label transitions that are associated with the target variable x_j but change both destinations into 1-space (dropping the variable set). For a label transition not associated with x_j, we replace it with an ϵ-transition whose destination state is either the first original destination q_1 or the second q_2, depending on whether the target variable x_j appears in q_1 or not. In 1-space, we simply retain all transitions (again dropping the variable sets).

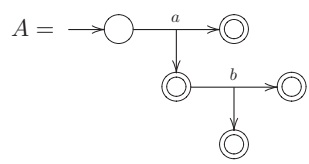
To illustrate the algorithm, we will perform type inference with the input type

```
a[b[]|()]
```

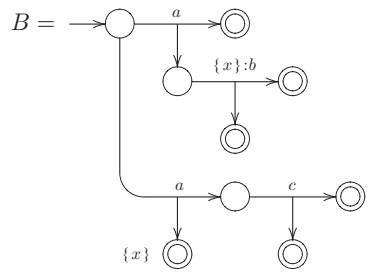
and the following pattern:

```
a[b[] as x] | a[() as x],c[]
```

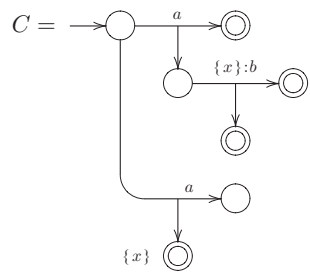
The input type is translated to the tree automaton



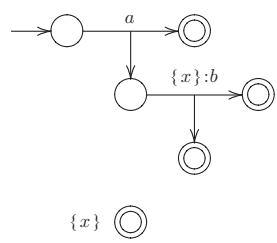
and the pattern is translated to the following marking tree automaton:



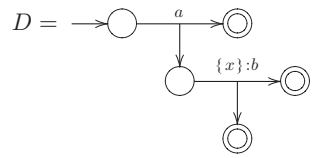
The first step is to take the product of A and B, resulting in the following automaton (in which we have elided states unreachable from the initial state):



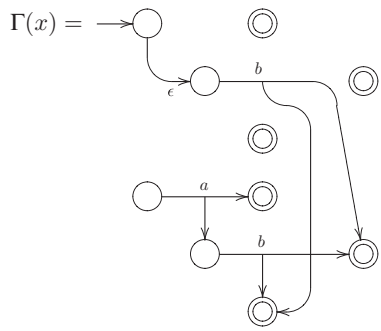
The intuition behind this product construction is that we should obtain a marking automaton that behaves exactly the same as B except that the accepted trees are restricted to those of A. In the second step we first remove the empty state



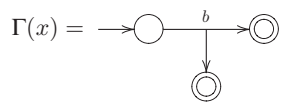
and then remove the unreachable state:



In the final step we duplicate D's state space to 0-space, which captures matched labels, and to 1-space, which captures the subtree rooted at those labels. This yields the tree automaton

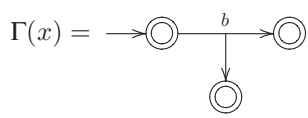


(the upper half represents 0-space and the lower half 1-space), whose ϵ-elimination is just



(showing only reachable states). The result represents the type `b[]`.

What if we skip the second step? The answer is that then the result of the type inference can be inexact. In the above example, if we directly computed $\Gamma(x)$ from the automaton C then the final result would contain an additional initial state, as in



which is incorrect since x is never bound to the empty sequence () by the matching of any tree from A against B. In general, useless states encode effects that actually never happen and therefore eliminating them is essential. The same technique is used in other analyses and we will see such examples in the ambiguity checks in Chapter 14.

We now give a formal proof that the type inference algorithm really implements the specification. The lemmas that follow show the correctness of each step.

Lemma 7.3.1 The automaton C matches t, yielding a binding V, if and only if A accepts t and B matches t, yielding the binding V.

Proof: To show the "if" direction, suppose that r is a successful run of A on t and (r', m') is a successful marking run of B on t. Let $r''(\pi) = (r(\pi), r'(\pi))$ for $\pi \in \mathbf{nodes}(t)$. Then we can easily show that (r'', m') is a successful marking run of C on t by induction on the length of π. The "only if" direction can be shown similarly. □

Lemma 7.3.2 For any $q \in Q_D$, D matches t in q, yielding a binding V, if and only if C matches t in q, yielding the binding V.

Proof: The "only if" direction is clear. To show the "if" direction, suppose that (r, m) is a successful marking run of C on t, where $r(\epsilon) = q$. Since $q \in I_C$ and all states used in r are reachable from q, the second step retains those states and the transitions relating them. Therefore (r, m) is also a successful marking run of D on t, where $r(\epsilon) = q$. □

Lemma 7.3.3 The tree automaton $\Gamma(x_j)$ (before ϵ-eliminaton) accepts u if and only if D matches some t, yielding some binding V with $V(x_j) = u$.

Proof: We first show the "if" direction. Suppose that (r, m) is a successful marking run of D on t, where $x_j \in m(\pi)$. We aim at finding a successful run r' of $\Gamma(x_j)$ on $\mathbf{subtree}_t(\pi)$. From the assumption, there is a sequence of states

$$(r(\epsilon) =)\ q_1, \quad \ldots, \quad q_k\ (= r(\pi))$$

such that, for each $i = 1, \ldots, k-1$,

$$q_i \to \mathbf{x}_i : a_i(q_{i+1}, q') \in \Delta_D \qquad \text{or} \qquad q_i \to \mathbf{x}_i : a_i(q', q_{i+1}) \in \Delta_D$$

for some $a_i \in \Sigma$, $q' \in Q_D$, and $\mathbf{x}_i \in 2^{\mathcal{X}}$ with $x_j \notin \mathbf{x}_i$. Also, since $x_j \in \mathbf{Var}(q_k)$, linearity ensures $x_j \in \mathbf{Var}(q_{i+1})$ for each $i = 1, \ldots, k-1$. Therefore Δ_j has ϵ-transition $\langle q_i, 0\rangle \xrightarrow{\epsilon} \langle q_{i+1}, 0\rangle$ for each $i = 1, \ldots, k-1$, that is, $\langle q_k, 0\rangle \in \epsilon\text{-}\mathbf{closure}(\langle q_1, 0\rangle)$. Now, define r' such that

$$\begin{array}{lcll} r'(\epsilon) & = & \langle q_1, 0\rangle & \\ r'(\pi') & = & \langle r(\pi\pi'), 1\rangle & \text{for } \pi' \neq \epsilon. \end{array}$$

If $\pi \in \mathbf{leaves}(t)$ then the result immediately follows, since $\langle q_k, 0\rangle \in F_j$. If $\mathbf{label}_t(\pi) = a$ then, from $x_j \in m(\pi)$, we have

$$\langle r(\pi), 0\rangle \to a(\langle r(\pi 1), 1\rangle, \langle r(\pi 2), 1\rangle) \in \Delta_j.$$

Further, by the definition of Δ_j we also have

$$\langle r(\pi\pi'), 1\rangle \to a(\langle r(\pi\pi'1), 1\rangle, \langle r(\pi\pi'2), 1\rangle) \in \Delta_j$$

for $\pi' \neq \epsilon$. The result follows since $\langle r(\pi\pi'), 1\rangle \in F_j$ whenever $\pi\pi' \in \mathbf{leaves}(t)$.

We next show the "only if" direction. Suppose that r' is a successful run of $\Gamma(x_j)$ on u. We aim at finding a tree t and a position π together with a successful marking run (r, m) of D on t such that $x_j \in m(\pi)$ and $u = \mathbf{subtree}_t(\pi)$. Note first that $r'(\epsilon) = \langle q_1, 0\rangle \in I_j$. If $\mathbf{label}_t(\epsilon) = \#$ then there exists $\langle q_k, 0\rangle \in F_j$ such that $\langle q_k, 0\rangle \in \epsilon\text{-}\mathbf{closure}(\langle q_1, 0\rangle)$. If $\mathbf{label}_t(\epsilon) = a$ then there exists $\langle q_k, 0\rangle \to a(\langle r'(1), 1\rangle, \langle r'(2), 1\rangle) \in \Delta_j$ such that $\langle q_k, 0\rangle \in \epsilon\text{-}\mathbf{closure}(\langle q_1, 0\rangle)$. In either case, by the definition of Δ_j we have, for each $i = 1, \ldots, k-1$,

$$(1)\quad q_i \to \mathbf{x}_i : a_i(q_{i+1}, q_i') \in \Delta_D \qquad \text{or} \qquad (2)\quad q_i \to \mathbf{x}_i : a_i(q_i', q_{i+1}) \in \Delta_D$$

for some $a_i \in \Sigma$, $q_i' \in Q_D$, and $\mathbf{x}_i \in 2^{\mathcal{X}}$ with $x_j \notin \mathbf{x}_i$. Since each state $q_i' \in Q_D$ is non-empty, there is a tree t_i on which a run r_i is successful with $r_i(\epsilon) = q_i'$. Then, letting $\pi_1 = \epsilon$, define t_i, π_i, r, and m, for each $i = 1, \ldots, k-1$, such that in case (1)

$$t_i = a_i(t_{i+1}, t_i') \qquad \pi_{i+1} = \pi_i 1 \qquad r(\pi_i) = q_i \qquad m(\pi_i) = \mathbf{x}_i,$$
$$r(\pi_i 2\pi') = r_i(\pi') \qquad \text{for } \pi' \in \mathbf{nodes}(t_i')$$

and, in case (2),

$$t_i = a_i(t_i', t_{i+1}) \qquad \pi_{i+1} = \pi_i 2 \qquad r(\pi_i) = q_i \qquad m(\pi_i) = \mathbf{x}_i,$$
$$r(\pi_i 1\pi') = r_i(\pi') \qquad \text{for } \pi' \in \mathbf{nodes}(t_i').$$

Here, we define $t_k = u$ and $r(\pi_k) = q_k$ and, if $\mathbf{label}_t(\epsilon) = \#$,

$$m(\pi_k) = \mathbf{x} \qquad \text{s.t.} \quad (q_k, \mathbf{x}) \in F_D$$

(in which case $x_j \in \mathbf{x}$ by linearity); otherwise,

$$m(\pi_k) = \mathbf{x} \qquad \text{s.t.} \quad q_k \to \mathbf{x} : a(r'(1), r'(2)) \in \Delta_D \text{ with } x_j \in \mathbf{x}$$
$$r(\pi_k\pi') = q \qquad m(\pi_k\pi') = \emptyset \qquad \text{s.t.} \quad r'(\pi') = \langle q, 1\rangle \text{ for } \pi' \neq \epsilon.$$

Hence we can easily show that (r, m) as defined is a successful marking run of D on t_1 such that $x_j \in m(\pi_k)$ and $u = \mathbf{subtree}_t(\pi)$. □

Corollary 7.3.4 The type inference algorithm satisfies the specification.

7.3.2 Non-tail variables

In Section 6.2 we introduced marking tree automata with beginning and ending variables in order to treat non-tail variables. The type inference algorithm in the last subsection also works for such automata if a slight modification is made. Specifically, we keep the first two steps of the algorithm in Section 7.3.1 but change the third step as follows.

3. Return Γ such that $\Gamma(x_j)$ is the tree automaton $(Q_j, I_j, F_j, \Delta_j)$ with ϵ-transitions where

$$
\begin{aligned}
Q_j &= Q_d \times \{0, 0', 1\} \\
I_j &= \{\langle q_0, 0\rangle \mid q_0 \in I_D\} \\
F_j &= \{\langle q, k\rangle \mid (q, \mathbf{x}) \in Q_D,\ k \in \{0, 0', 1\}\} \\
\Delta_j &= \{\langle q, 0\rangle \to a(\langle q_1, 1\rangle, \langle q_2, 0'\rangle) \mid q \to \mathbf{x} : a(q_1, q_2) \in \Delta_D,\ x_{jb} \in \mathbf{x}\} && \text{(a)} \\
&\quad \cup \{\langle q, 0\rangle \xrightarrow{\epsilon} \langle q_1, 0\rangle \mid q \to \mathbf{x} : a(q_1, q_2) \in \Delta_D,\ x_{jb} \notin \mathbf{x},\ x_{jb} \in \mathbf{Var}(q_1)\} \\
&\quad \cup \{\langle q, 0\rangle \xrightarrow{\epsilon} \langle q_2, 0\rangle \mid q \to \mathbf{x} : a(q_1, q_2) \in \Delta_D,\ x_{jb} \notin \mathbf{x},\ x_{jb} \notin \mathbf{Var}(q_1)\} \\
&\quad \cup \{\langle q, 0\rangle \xrightarrow{\epsilon} \langle q_2, 0\rangle \mid q \to \mathbf{x} : a(q_1, q_2) \in \Delta_D,\ x_{je}, x_{jb} \in \mathbf{x}\} && \text{(b)} \\
&\quad \cup \{\langle q, 0'\rangle \xrightarrow{\epsilon} \langle q_2, 0\rangle \mid q \to \mathbf{x} : a(q_1, q_2) \in \Delta_D,\ x_{je} \in \mathbf{x}\} && \text{(c)} \\
&\quad \cup \{\langle q, 0'\rangle \to a(\langle q_1, 1\rangle, \langle q_2, 0'\rangle) \mid q \to \mathbf{x} : a(q_1, q_2) \in \Delta_D,\ x_{je} \notin \mathbf{x}\} && \text{(d)} \\
&\quad \cup \{\langle q, 1\rangle \to a(\langle q_1, 1\rangle, \langle q_2, 1\rangle) \mid q \to \mathbf{x} : a(q_1, q_2) \in \Delta_D\}
\end{aligned}
$$

   and perform ϵ-elimination.

The last construction becomes somewhat more complicated since both beginning and ending variables have been introduced. The state space is now replicated to three spaces, where the 1-space is similar to before but the 0-space now represents states before a beginning variable is encountered or after an ending variable is encountered, and the additional $0'$-space represents states between the beginning and ending variables. In the definition of Δ_j the differences from the last algorithm are clauses (a), (b), (c), and (d). In 0-space, (a) if we encounter a beginning variable then we move to $0'$-space as the second destination (and to 1-space as the first destination, as before); (b) if we encounter both beginning and ending variables at the same time then we simply skip this transition. In $0'$-space, (c) if we encounter an ending variable then we skip the transition and move back to 0-space; otherwise (d) we retain the transition (to 1-space as the first destination).

As an example, let us perform type inference with the input type

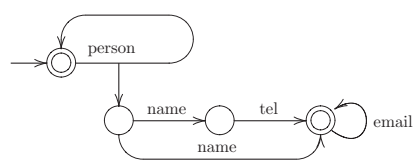
```
person[(Name, Tel?, Email*)]*
```

and the following pattern:

```
person[Name,Email*] as x, Any
```

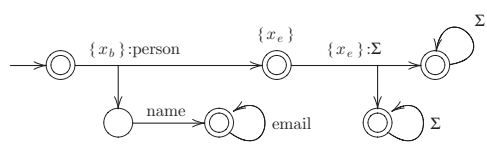
First, we translate the input type to the tree automaton

$A =$



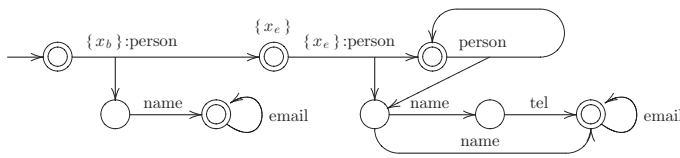
and the pattern to the following marking tree automaton:

$B =$



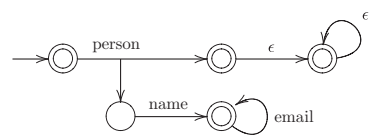
Here, a transition with the label Σ denotes a set of transitions that have the same destination and source states for all labels from Σ. Also, the first destinations of some transitions are omitted for brevity. Next, we take the product C of A and B:

$C =$



(we show only reachable states). After eliminating useless states (in fact there are none), we finally extract the part between x_b and x_e:

$\Gamma(x) =$



(again only reachable states are shown). We can obtain the final result by ϵ-elimination, but this step is omitted here.

Another way to treat non-tail variables is to use sequence-marking tree automata, introduced in Section 6.3. Type inference for this form of marking tree automata is in fact simpler than that just presented. Furthermore, the sequence-capturing semantics described in Section 5.3 can be handled automatically.

Exercise 7.3.5 (⋆⋆⋆) Give a type inference algorithm for sequence-linear sequence-marking automata. ■

7.3.3 Matching semantics

A type inference algorithm for patterns is sensitive to which matching policy is taken. The algorithms given thus far are designed for patterns in the nondeterministic semantics. Obviously, the same algorithms work also with the unambiguous semantics since there is only one match. They also work with the all-matches semantics since the specification tells us that the inference result $\Gamma(x)$ for a variable x contains a value u whenever *any* match yields a binding of x to this value.

However, treating the prioritized-match semantics is trickier since, even if a value could be matched with some part of the pattern, the priority rules may force the value to be matched with some other part. Therefore, in the prioritized-match semantics, each sub-pattern has less chance of being matched to a value than in the nondeterministic semantics. Accordingly, the presented inference algorithms need to be modified so that each state of the inferred automaton represents the set of values that can be matched by a certain sub-pattern but *not* matched by other sub-patterns with higher priorities.

7.4 Bibliographic notes

The XDuce language pioneered approximate and realistic typechecking for XML (Hosoya and Pierce, 2003). An algorithm of type inference for patterns based on the greedy-match semantics was presented in Hosoya and Pierce (2002) and one based on nondeterministic match in Hosoya (2003). The design of XDuce was extended further in ℂDuce (Frisch *et al.*, 2008; Benzaken *et al.*, 2003); the main addition was higher-order functions. Extensions with parametric polymorphism have also been proposed, in Vouillon (2006) and Hosoya *et al.* (2009). Integration of an XML processing language with a popular programming language has been pursued in the projects XHaskell (Sulzmann and Lu, 2007), Xtatic (Gapeyev *et al.*, 2006), XJ (Harren *et al.*, 2005), and OCamlDuce (Frisch, 2006). These extended Haskell, C#, Java, and O'Caml, respectively.

As mentioned at the end of Section 7.1.2, there is an approximate typechecking approach requiring no type annotations. In particular, JWig (Christensen *et al.*, 2002) and its descendant XAct (Kirkegaard and Møller, 2006) perform flow analysis on a Java program with an extension for XML processing constructs; XSLT Validator (Møller *et al.*, 2007) uses a similar analysis on XSLT style sheets. For exact typechecking, see Chapter 11.

Part II

Advanced topics

8

On-the-fly algorithms

This chapter presents efficient algorithms for several important problems related to XML processing, namely: (1) membership for tree automata (to test whether a given tree is accepted by a given tree automaton); (2) evaluation of marking tree automata (to collect the set of bindings yielded by the matching of a given tree against a given marking tree automaton); and (3) containment for tree automata (to test whether the languages of given two tree automata are in subset relation). For these problems we describe several "on-the-fly" algorithms, in which only a part of the whole state space is explored to obtain the final result. In such algorithms there are two basic approaches, top-down and bottom-up. The top-down approach explores the state space from the initial states whereas the bottom-up approach does this from the final states. In general, the bottom-up approach tends to have a lower complexity in the worst case whereas the top-down often gives a higher efficiency in practical cases. We will also consider a further improvement from combining these ideas.

8.1 Membership algorithms

In this section, we will consider three algorithms for testing membership. The first is a top-down algorithm, which can be obtained rather simply from the semantics of tree automata but takes time that is exponential in the size of the input tree, in the worst case. The second is a bottom-up algorithm, which constructs a bottom-up deterministic tree automaton on the fly, generating only the states needed for deciding the acceptance of the particular input tree. This algorithm works in linear time; however, it suffers from the overhead of carrying large sets of states. The third is a bottom-up algorithm with top-down filtering and overcomes the latter problem by taking into account only states that are potentially useful in top-down runs. In practice, this improvement substantially reduces the sizes of the state sets that must be carried, thus making the algorithm much more efficient.

8.1.1 Top-down algorithm

In the top-down algorithm, we recursively visit each node with a possible state and examine whether this state accepts the node. If this fails then we backtrack to the previous choice point and examine another state. The following gives pseudo-code for the algorithm.

```
1:  function Accept(t, (Q, I, F, Δ))                    ▷ returning a boolean
2:      for all q ∈ I do
3:          if AcceptIn(t, q) then return true
4:      end for
5:      return false
6:
7:      function AcceptIn(t, q)                         ▷ returning a boolean
8:          switch t do
9:            case #:
10:             if q ∈ F then return true else return false
11:           case a(t1, t2):
12:             for all q → a(q1, q2) ∈ Δ do
13:                 if AcceptIn(t1, q1) and AcceptIn(t2, q2) then return true
14:             end for
15:             return false
16:         end switch
17:     end function
18: end function
```

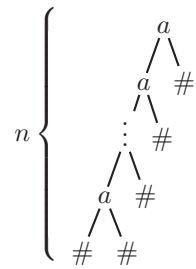
The function Accept takes a tree and a tree automaton and checks whether the automaton accepts a given tree t; for this it simply checks acceptance of the tree in one of the initial states, using the internal function AcceptIn. This internal function checks whether a given t is accepted in a given state q. The body of the function is a simple rephrasing of the semantics of tree automata. That is, for a leaf # the function checks whether q is final. For an intermediate node $a(t_1, t_2)$, each transition of the form $q \to a(q_1, q_2)$ is considered; recursive calls are made to AcceptIn to check whether the subtrees t_1 and t_2 are each accepted in the destination states q_1 and q_2.

Note that, when either of the two conditions in line 13 fails, the same subtree is retried with another transition. Therefore, in an unfortunate case we end up traversing the same subtree multiple times. In general, the worst-case time complexity of the top-down algorithm is exponential in the size of the input tree. (Note that a simple optimization is possible: if the first recursive call in line 13 returns "false" then the second can be skipped.)

Example 8.1.1 Consider the tree automaton $A_{8.1.1} = (\{q_1, q_2\}, \{q_1, q_2\}, \{q_2\}, \Delta)$, where Δ consists of

$$q_1 \to a(q_1, q_1)$$
$$q_1 \to a(q_1, q_2)$$
$$q_2 \to a(q_2, q_2).$$

This automaton accepts trees $t_{8.1.1}$ of the form



for any $n \geq 1$ since a successful run can assign q_2 to all nodes. Suppose that, in the algorithm, we try first q_1 as an initial state and then q_2. When we start in the state q_1 we arrive at each a-node still in the state q_1; from such a state two transitions are always possible but both fail since the leftmost leaf will never be assigned the final state q_2. Thus, if we write T_n for the number of visits to the nth a-node from the root then we have the recurrence equation $T_n = 2T_{n-1}$ with $T_1 = 1$, which can be solved since $T_n = 2^{n-1}$. Therefore this part of the search necessarily takes time $O(2^n)$. After this we try the second initial state q_2, and this search succeeds in linear time. ■

It should be noted, however, that if the given tree automaton is top-down deterministic then no backtrack arises and therefore the complexity becomes linear. Many schemas used in practice can be converted to a top-down deterministic tree automaton (see Section 4.3.1), and therefore in this case the above naive algorithm suffices.

8.1.2 Bottom-up algorithm

The next two algorithms are linear-time since they traverse each node exactly once. Of course, the most straightforward approach for a linear-time acceptance check would be to determinize the given automaton (Section 4.3.2). However, determinization involves subset construction, which takes time that is exponential in the size of the automaton, and therefore often one prefers to avoid this approach.

The bottom-up algorithm traverses the input tree from the leaves to the root, creating, at each node, the state of the bottom-up deterministic tree automaton assigned to this node. The following pseudo-code gives the algorithm.

```
1:  function ACCEPT(t, (Q, I, F, Δ))                    ▷ returning a boolean
2:      if ACCEPTING(t)∩I ≠ ∅ then return true else return false
3:
4:      function ACCEPTING(t)                           ▷ returning a set of states
5:          switch t do
6:            case #:
7:              return F
8:            case a(t1, t2):
9:              let s1 = ACCEPTING(t1)
10:             let s2 = ACCEPTING(t2)
11:             let s = {q | q → a(q1, q2) ∈ Δ, q1 ∈ s1, q2 ∈ s2}
12:             return s
13:         end switch
14:     end function
15: end function
```

The main ACCEPT function validates the input tree by first computing the set of states accepting the tree and then checking that this set includes an initial state. For computing the accepting states we use the internal function ACCEPTING. This function takes a tree t and returns the set of states, of the given nondeterministic tree automaton, that accept this tree. This set of states in fact describes the state of the (conceptually constructed) bottom-up deterministic automaton that accepts the tree. Thus, in the body of the function, if the given tree is a leaf then the set of final states is returned since these are all the states and the only states that accept the leaf. For an intermediate node we first collect the sets s_1 and s_2 of states that accept the subtrees t_1 and t_2, respectively, by recursive calls to ACCEPTING. Then, we calculate the set s of states that accept the current tree t, by collecting the states that can transit with label a to states q_1 and q_2 belonging to s_1 and s_2, respectively. Note that exactly the same calculation as in line 11 is used in the subset construction (Section 4.3.2).

Exercise 8.1.2 (⋆) Apply the bottom-up algorithm to the tree automaton $A_{8.1.1}$ and the trees $A_{8.1.1}$. ■

Since the above algorithm visits each node only once, we can avoid potentially catastrophic behavior, as in the top-down algorithm. However, in return we need to perform a rather complicated set manipulation in line 11. The complexity of this operation depends on the representation of sets but, if we use standard binary search trees, each operation takes time $O(|Q|^2 \log |Q|)$ in the worst case (the factor $|Q|^2$ relates to looping on s_1 and s_2 and the factor $\log |Q|$ to inserting an element into a binary search tree). Even if we do not consider the worst case, it would still be better to keep the size of the manipulated set small, and this will be addressed in the next section.

Exercise 8.1.3 (⋆) Consider the tree automaton $A_{8.1.3} = (\{q_1, q_2, q_3\}, \{q_1\}, \{q_1, q_3\}, \Delta)$, where Δ consists of

$$\begin{aligned} q_1 &\to a(q_2, q_1) \\ q_1 &\to a(q_1, q_1) \\ q_2 &\to a(q_2, q_1) \\ q_3 &\to a(q_3, q_1). \end{aligned}$$

Apply the given bottom-up algorithm to the automaton $A_{8.1.3}$ and the tree $A_{8.1.1}$. Observe that q_3 is always contained in the set of states returned by each recursive call but is discarded at the end. ■

8.1.3 Bottom-up algorithm with top-down filtering

This third algorithm can be regarded as a combination of the two algorithms described above. The following ways of looking at the first two algorithms are quite useful for understanding the third.

- In the top-down algorithm, we essentially generate top-down runs one after another and, during this generation, we check for the failure of a run at each leaf.
- In the bottom-up algorithm, we essentially generate the set of all bottom-up runs and, in the end, check that a successful one is contained in that set.

An important observation is that, among the bottom-up runs collected in the second algorithm, in practice there are not many that are top-down runs. Therefore, in the third algorithm below, we visit each node with the set of states that can be assigned to the node in a top-down run. We then use this set for filtering the set of states collected in a bottom-up manner. The following pseudo-code gives the algorithm.

```
1:  function Accept(t, (Q, I, F, Δ))                    ▷ returning a boolean
2:      if AcceptingAmong(t, I) ≠ ∅ then return true else return false
3:
4:      function AcceptingAmong(t, r)                   ▷ returning a set of states
5:          switch t do                                 ▷ argument r is also a set of states
6:            case #:
7:              return F ∩ r
8:            case a(t1, t2):
9:              let s1 = AcceptingAmong(t1, {q1 | q → a(q1, q2) ∈ Δ, q ∈ r})
10:             let s2 =
11:              AcceptingAmong(t2, {q2 | q → a(q1, q2) ∈ Δ, q ∈ r, q1 ∈ s1})
12:             let s = {q | q → a(q1, q2) ∈ Δ, q ∈ r, q1 ∈ s1, q2 ∈ s2}
13:             return s
14:         end switch
15:     end function
16: end function
```

The main function Accept checks the validity of the given tree by first collecting the set of initial states that accept the tree and then checking that this set is non-empty. For computing the set of accepting initial states, the internal function AcceptingAmong is used. This function takes a tree t and a set r of states and returns the set of states that accept this tree *and* are in the set r. The body has a similar structure to that of the Accepting function of the bottom-up algorithm. If the given tree t is a leaf # then the set F of final states restricted to the passed set r is returned. If t is an intermediate node $a(t_1, t_2)$ then AcceptingAmong is called recursively for the first subtree t_1 (line 9). Here the set of possible states for t_1 is given; this is the set of all first destinations q_1 of transitions $q \to a(q_1, q_2)$ emanating from states q in r. (We ignore q_2 at this moment.) Next, AcceptingAmong is called for the second subtree t_2 (line 11). The set of possible states for t_2 is computed similarly except that now the result s_1 from the first call can be used; this gives the possible states for q_1. Thus we obtain the set of all second destinations q_2 of transitions $q \to a(q_1, q_2)$ such that q belongs to r *and* q_1 belongs to s_1. Finally, the result of the original call is calculated in a similar way to Accepting in the bottom-up algorithm, except that it is restricted it to the given set r (line 12).

This algorithm does not improve the worst-case complexity of the bottom-up algorithm. However, experience tells us that the size of the set of states passed around is usually small – such a set contains typically one or two states – and therefore the improvement in practical cases is considerable.

Exercise 8.1.4 (⋆) Apply the bottom-up algorithm with top-down filtering to the tree automaton $A_{8.1.3}$ and the trees $t_{8.1.1}$. Confirm that the set of states returned by each recursive call never contains q_3. ■

Exercise 8.1.5 (⋆) Find a tiny optimization to the bottom-up algorithm with top-down filtering by which the algorithm can return quickly in case of failure. ■

8.2 Marking algorithms

Let us next consider the execution of marking tree automata, that is, the computation of all possible bindings for a given input tree. As in the last section, we first look at a naive, potentially exponential-time top-down, algorithm and then a cleverer, linear-time bottom-up, algorithm.

We will assume in this section that a given marking tree automaton $(Q, I, F, \Delta, \mathcal{X})$ has no useless state and passes the syntactic linearity check given in Section 6.1. In this setting, a binding can be regarded as a mapping from variables to trees. In addition we consider below a multiset, rather than a set, of bindings yielded from matching. That is, when there are multiple ways of matching an input tree, some resulting bindings can be identical if the automaton has ambiguity or the input tree includes structurally identical subtrees; we do not unify such bindings. The reasons are first that a duplication check is computationally expensive and second that often such bindings are actually not identical to each other when we match with real XML documents, which usually contain some minor data ignored here (attributes, comments, etc.).

For convenience, we now introduce some notations related to mappings. The form $\{\mathbf{x} \mapsto t\}$ stands for the mapping $\{x \mapsto t \mid x \in \mathbf{x}\}$. For a multiset of mappings, we define its *domain* as the union of the domains of all the mappings. The "product" of multisets W_1 and W_2 of mappings with disjoint domains is defined as

$$W_1 \times W_2 = \{w_1 \cup w_2 \mid w_1 \in W_1,\ w_2 \in W_2\}.$$

8.2.1 Top-down algorithm

The top-down marking algorithm works in exactly the same way as the top-down membership algorithm except that we additionally construct a multiset of bindings at each node. The following shows pseudo-code for this algorithm.

```
1:  function MATCH(t, (Q, I, F, Δ, X))                ▷ returning a multiset of bindings
2:      return ⋃_{q∈I} MATCHIN(t, q)
3:
4:      function MATCHIN(t, q)                        ▷ returning a multiset of bindings
5:          switch t do
6:            case #:
7:              return ⋃_{(q,x)∈F} {{x ↦ t}}
8:            case a(t1, t2):
9:              return ⋃_{(q→x:a(q1,q2))∈Δ} {{x ↦ t}} × MATCHIN(t1, q1) × MATCHIN(t2, q2)
10:         end switch
11:     end function
12: end function
```

The main function MATCH returns the multiset of all bindings that result from matching the input tree t against any initial state. For computing a multiset of binding for each initial state, we use the internal function MATCHIN, which takes a tree t and a state q and returns the multiset of all bindings that are yielded by matching t against q. (Note that if the matching fails then the function returns an empty multiset.) In the body of the function, when the tree t is a leaf, the multiset of bindings $\{\mathbf{x} \mapsto t\}$ is returned for each variable set $\mathbf{x}$ associated with the state q according to the set F. Since linearity (Section 6.1) actually ensures that F associates at most one variable set with each state, by (L2), either an empty or a singleton multiset is returned. When t is an intermediate node $a(t_1, t_2)$, we collect and combine the multisets of bindings obtained for all the transitions $q \to \mathbf{x} : a(q_1, q_2)$ emanating from q. Here, the multisets of bindings are first computed by recursively calling MATCHIN with the subtrees t_1 and t_2 and the corresponding destination states q_1 and q_2, respectively, and they are then combined, by product, together with the binding $\{\mathbf{x} \mapsto t\}$ for the current node. Linearity ensures that the domains of the multisets of bindings to be combined are pairwise disjoint, by (L3a).

By exactly the same argument as for the top-down membership algorithm, the worst-case time complexity is exponential in the size of the input. However, we can add a simple optimization analogous to the top-down membership algorithm: when computing the clause

$$\{\{\mathbf{x} \mapsto t\}\} \times \text{MATCHIN}(t_2, q_2) \times \text{MATCHIN}(t_2, q_2)$$

in line 9, if the first call to MATCHIN returns $\emptyset$ then we can skip the second call since the whole clause will return $\emptyset$ whatever the second call returns.

8.2.2 Bottom-up algorithm

We can construct an efficient bottom-up marking algorithm by adapting the idea used in the bottom-up membership algorithm. That is, rather than trying to match each node with a single state, we obtain the set of all states that match each node. This means that the algorithm requires only a single scan of the whole tree. However, with only this improvement the algorithm still incurs a cost $O(|t|^k)$ for the generation of a multiset of bindings, where $|t|$ is the size of the input tree and k is the number of variables. This is unavoidable when the output multiset itself already has size $O(|t|^k)$. However, the unsatisfactory aspect is that such a cost is present even when the output is not so large. We will present a more sophisticated marking algorithm that uses a technique called *partially lazy multiset operations* and thereby runs in $O(|t| + |W|)$ time, where $|W|$ is the size of the output multiset.

Let us start with a naive algorithm, given in the following pseudo-code.

1: **function** MATCH(t, $(Q, I, F, \Delta, \mathcal{X})$) ▷ returning a multiset of bindings
2: **let** $M = \text{MATCHING}(t)$
3: **return** $\bigcup_{q \in I} M(q)$
4:
5: **function** MATCHING(t)
6: ▷ returning a mapping from a state to a multiset of bindings
7: **switch** t **do**
8: **case** #:
9: **return** $\left\{ q \mapsto \bigcup_{(q,\mathbf{x}) \in F} \{\{\mathbf{x} \mapsto t\}\} \;\middle|\; q \in Q \right\}$
10: **case** $a(t_1, t_2)$:
11: **let** $M_1 = \text{MATCHING}(t_1)$
12: **let** $M_2 = \text{MATCHING}(t_2)$
13: **return** $\left\{ q \mapsto \bigcup_{(q \to \mathbf{x}:a(q_1,q_2)) \in \Delta} \{\{\mathbf{x} \mapsto t\}\} \times M_1(q_1) \times M_2(q_2) \;\middle|\; q \in Q \right\}$
14: **end switch**
15: **end function**
16: **end function**

The main function MATCH, as before, calculates the multiset of all bindings resulting from matching the input tree against any possible initial state. We obtain this multiset by first finding the mapping M from each state q to the multiset of bindings obtained from matching the input tree against q and then taking the union of the multisets of bindings for all initial states. To compute the mapping M we use the internal function MATCHING, which takes a tree t and returns the mapping from each state q to the multiset of bindings resulting from matching t against q (where, in particular, a non-matching state is mapped to an empty multiset). The structure of the body of MATCHING closely resembles the internal function MATCHIN in the top-down algorithm. However, there are a few differences that enable the algorithm to run using only a linear scan. First, MATCHIN, in the top-down case, takes both a node t and a state q as arguments whereas MATCHING, in the bottom-up case, takes only a node t but returns a mapping from states. (Readers familiar with λ-calculi will recognize it as a "currification.") Second, in MATCHING, the case of an intermediate node $a(t_1, t_2)$ moves the recursive calls to MATCHING out of the construction of the mapping since these calls do not depend on q.

As already mentioned, however, a linear scan does not necessarily run in time that is linear in the input size. Indeed, it can take time $O(|t| + |t|^k)$, where the term $|t|^k$ is incurred for generating the output multiset.

Example 8.2.1 Consider the marking tree automaton $A_{8.2.1} = (\{q_0, q_1, q_2, q_3\}, \{q_0\}, F, \Delta, \{x, y\})$, where the set Δ of transition rules consists of

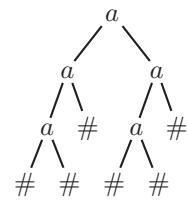
$$
\begin{array}{lll}
q_0 \to \emptyset : a(q_1, q_2) & & \\
q_1 \to \{x\} : a(q_3, q_3) & q_1 \to \emptyset : a(q_1, q_3) & q_1 \to \emptyset : a(q_3, q_1) \\
q_2 \to \{y\} : a(q_3, q_3) & q_2 \to \emptyset : a(q_2, q_3) & q_2 \to \emptyset : a(q_3, q_2) \\
q_3 \to \emptyset : a(q_3, q_3) & &
\end{array}
$$

and we have

$$F = \{(q_3, \emptyset)\}.$$

Let us explain each state's meaning. First, the state q_3 accepts any tree whose only label is a and yields no binding. Given this state, the state q_1 accepts any tree whose only label is a but binds x to some node in the tree. The state q_2 is similar to q_1 except that it binds y. Given the states q_1 and q_2, the state q_0 accepts a tree whose only label is a and binds x to some node in the left subtree and y to some node in the right subtree. Hence, for an input tree t of the form $a(t', t')$, the size of the output, that is, the number of possible bindings, is $|t'|^2 \approx O(|t|^2)$. This example can easily be generalized to k variables. ■

Exercise 8.2.2 (⋆) Run the bottom-up algorithm with the marking tree automaton $A_{8.2.1}$ and the following tree:

$t_{8.2.2} =$ 

Observe how a large multiset of bindings is generated. ■

Example 8.2.1 shows that in unfortunate cases the output size is already $O(|t|^k)$ and therefore just enumerating it necessarily takes time proportional to this. However, there are cases where the output size is not so large and yet the algorithm takes exponential time.

Example 8.2.3 Consider the marking tree automaton $A_{8.2.3} = (\{q_A, q_0, q_1, q_2, q_3\}, \{q_A\}, F, \Delta \cup \Delta', \{x, y\})$, where F and Δ are as in $A_{8.2.1}$ and Δ' consists of the following transitions:

$$q_A \to \emptyset : b(q_0, q_3)$$
$$q_A \to \{x, y\} : a(q_3, q_3)$$

Thus the state q_A accepts either

- a tree of the form $b(a(t_1, t_2), t_3)$, where t_1, t_2, t_3 contain only label a, with a binding of x to some node in t_1 and of y to some node in t_2, or
- a tree of the form $a(t_1, t_2)$, where t_1, t_2 contain only label a, with the binding of both x and y to the root.

How does the bottom-up algorithm work for the second type of input tree? We compute exactly the same huge multiset of bindings up to the node t_1 for the state q_0 as in Example 8.2.1, but then we discard this entire intermediate result at the root node since there is no transition from q_A to q_0 via label a. ■

Exercise 8.2.4 (⋆) Run the bottom-up algorithm with the marking tree automaton $A_{8.2.3}$ and the tree $a(t_{8.2.2}, \#)$. Observe how a large intermediate result is discarded at the end. ■

Example 8.2.5 Consider the marking tree automaton $A_{8.2.5} = (\{q_A, q_B, q_0, q_1, q_2, q_3\}, \{q_A\}, F, \Delta \cup \Delta'', \{x, y\})$, where F and Δ are as in $A_{8.2.1}$ and Δ'' consists

of the following:

$$q_A \to \emptyset : a(q_0, q_B)$$
$$q_A \to \{x, y\} : a(q_3, q_3)$$
$$q_B \to \emptyset : b(q_3, q_3)$$

This time, the first of the alternative trees accepted by state q_A in Example 8.2.3 is replaced by the following:

- a tree of the form $a(a(t_1, t_2), b(t_3, t_4))$, where t_1, t_2, t_3, t_4 contain only label a, with a binding of x to some node in t_1 and of y to some node in t_2

and q_B accepts a tree of the form $b(t_1, t_2)$ (where t_1, t_2 contain only label a) with an empty binding. Again, how does the bottom-up algorithm work for an input tree t of the form $a(t_1, t_2)$ with all nodes labeled a? As before, we compute a huge multiset of bindings up to the left child t_1 for the state q_0, but this time there *is* a transition that leads to q_0 from q_A via a. However, the right child t_2 of the root does not match the state q_B (since there is only a transition with b from q_B), and therefore we return the empty multiset. Then at the root we compute the product of the huge multiset and the empty multiset, which results in the empty multiset. Therefore the intermediate result from the computation up to the left child t_1 is, again, completely discarded, for a different reason this time. ■

Exercise 8.2.6 (⋆) Run the bottom-up algorithm with the marking tree automaton $A_{8.2.5}$ and the tree $a(t_{8.2.2}, \#)$. Observe again how a large intermediate result is discarded in the end. ■

In summary, we have seen two sources of wasted computation:

- computation for an unreachable state;
- computation that will be combined with the result from an unmatched state.

Fortunately, these are the only wastes and both can be eliminated by using partially lazy multiset operations, which are explained next.

Partially lazy multiset operations

This technique achieves for the bottom-up algorithm a linear-time complexity in the size of the output multiset. The idea is as follows.

- We delay the computations of the union $\cup$ and product $\times$ up to the root node, where we perform only the computations that are relevant to the initial states. This eliminates wasted computations for unreachable states.
- We eagerly compute, however, the union and product operations when one of the arguments is the empty multiset. This eliminates, in particular, wasted computations that will be combined with the empty multiset by product.

Concretely, we use the following data structures to represent multisets symbolically. First, we define *possibly empty multisets* $s_\emptyset$ and *non-empty multisets* s by the following grammar:

$$\begin{array}{lcl} s_\emptyset & ::= & \emptyset \mid s \\ s & ::= & \{\{\mathbf{x} \mapsto t\}\} \mid \mathbf{union}(s, s) \mid \mathbf{prod}(s, s) \end{array}$$

Then, in the bottom-up algorithm, we replace the $\cup$ and $\times$ operations with the operations $\uplus$ and $\otimes$, respectively, which symbolically construct possibly empty multisets:

$$\begin{array}{lcl} \emptyset \uplus s_\emptyset & = & s_\emptyset \\ s_\emptyset \uplus \emptyset & = & s_\emptyset \\ s_1 \uplus s_2 & = & \mathbf{union}(s_1, s_2) \\ \\ \emptyset \otimes s_\emptyset & = & \emptyset \\ s_\emptyset \otimes \emptyset & = & \emptyset \\ s_1 \otimes s_2 & = & \mathbf{prod}(s_1, s_2) \end{array}$$

Here, we carry out the actual computation if one of the arguments is empty but otherwise delay it. Finally, when the bottom-up algorithm returns a symbolically represented multiset for the whole tree, we force all the delayed multiset operations to enumerate the yielded bindings. For this, we use the function **eval**:

$$\begin{array}{lcl} \mathbf{eval}(\emptyset) & = & \emptyset \\ \mathbf{eval}(s) & = & \mathbf{eval}'(s, \emptyset) \end{array}$$

where

$$\begin{array}{lcl} \mathbf{eval}'(\{\{\mathbf{x} \mapsto t\}\}, V) & = & \{\{\mathbf{x} \mapsto t\}\} \cup V \\ \mathbf{eval}'(\mathbf{union}(s_1, s_2), V) & = & \mathbf{eval}'(s_1, \mathbf{eval}'(s_2, V)) \\ \mathbf{eval}'(\mathbf{prod}(s_1, s_2), V) & = & \mathbf{eval}'(s_1, \emptyset) \times \mathbf{eval}(s_2, \emptyset) \cup V. \end{array}$$

Here, the function $\mathbf{eval}'$ is defined in such a way that the enumeration can be performed in time that is linear in the size of the output multiset.

Now we will show the bottom-up marking algorithm using partially lazy multiset operations:

```
1: function MATCH(t, (Q, I, F, Δ, 𝒳))          ▷ returning a multiset of bindings
2:     let M = MATCHING(t)
3:     return eval(⨄_{q∈I} M(q))
4:
5:     function MATCHING(t)
6:                 ▷ returning a mapping from a state to a multiset of bindings
```

```
7:        switch t do
8:          case #:
9:            return { q ↦ ⊎_{(q,x)∈F} {{x ↦ t}} | q ∈ Q }
10:         case a(t1, t2):
11:           let M1 = MATCHING(t1)
12:           let M2 = MATCHING(t2)
13:           return { q ↦ ⊎_{(q→x:a(q1,q2))∈Δ} {{x ↦ t}} ⊗ M1(q1) ⊗ M2(q2) | q ∈ Q }
14:       end switch
15:     end function
16: end function
```

Note that the only differences from the last version are that $\cup$ and $\times$ are respectively replaced by $\uplus$ and $\otimes$ and that the **eval** function is applied in the very end.

Let us prove that the algorithm indeed takes only time $O(|t| + |W|)$. First, a crucial observation is that, by the linearity of the marking tree automaton, the domains of the multisets of bindings are always disjoint whenever a product is taken and therefore the depth of nestings of $\mathbf{prod}(\cdot, \cdot)$ in a symbolic multiset returned from MATCHING can be bounded by the number k of variables. Formally, we define

$$\begin{aligned} \mathbf{dp}(\{\{\mathbf{x} \mapsto t\}\}) &= 0 \\ \mathbf{dp}(\mathbf{union}(s_1, s_2)) &= \mathbf{max}(\mathbf{dp}(s_1), \mathbf{dp}(s_2)) \\ \mathbf{dp}(\mathbf{prod}(s_1, s_2)) &= \mathbf{max}(\mathbf{dp}(s_1), \mathbf{dp}(s_2)) + 1. \end{aligned}$$

Then, the following lemma can be proved.

Lemma 8.2.7 For a symbolic multiset $s_\emptyset$ with $\mathbf{dp}(s_\emptyset) = k$, we can compute $W = \mathbf{eval}(s_\emptyset)$ in time $O(3^k|W|)$. In particular, for a fixed k we can compute it in time $O(|W|)$.

Proof: Let us represent a (concrete) multiset by a list. It suffices to show that $W = \mathbf{eval}'(s, V)$ takes at most $3^{\mathbf{dp}(s)}|s|$ list insertions or extractions, where $|s|$ is defined by

$$\begin{aligned} |\{\{\mathbf{x} \mapsto t\}\}| &= 1 \\ |\mathbf{union}(s_1, s_2)| &= |s_1| + |s_2| \\ |\mathbf{prod}(s_1, s_2)| &= |s_1| \times |s_2|. \end{aligned}$$

Note that $|s| \geq 1$ for any s. We below consider only the number of list insertions, since the number of list extractions can be estimated similarly.

Case 1: $s = \{\{\mathbf{x} \mapsto t\}\}$
The result follows since $\mathbf{dp}(s) = 0$ and $|s| = 1$.

Case 2: $s = \mathbf{union}(s_1, s_2)$
By the induction hypothesis, the computation of $W_1 = \mathbf{eval}'(s_2, V)$ takes at most $3^{\mathbf{dp}(s_2)}|s_2|$ list insertions and that of $W = \mathbf{eval}'(s_1, W_1)$ takes at most $3^{\mathbf{dp}(s_1)}|s_1|$ list insertions. Therefore, in computing $W = \mathbf{eval}'(s, V)$ the number of list insertions is bounded by

$$\begin{aligned} 3^{d(s_2)}|s_2| + 3^{d(s_1)}|s_1| &\leq 3^{d(s)}|s_2| + 3^{d(s)}|s_1| \\ &\leq 3^{d(s)}(|s_2| + |s_1|) \\ &\leq 3^{d(s)}|s|. \end{aligned}$$

Case 3: $s = \mathbf{prod}(s_1, s_2)$
By the induction hypothesis, the computation of $W_1 = \mathbf{eval}'(s_1, \emptyset)$ takes at most $3^{\mathbf{dp}(s_1)}|s_1|$ list insertions and that of $W_2 = \mathbf{eval}'(s_2, \emptyset)$ takes at most $3^{\mathbf{dp}(s_2)}|s_2|$ list insertions. Furthermore, computing $W_1 \times W_2 \cup V$ takes $|W_1| \times |W_2| = |s_1| \times |s_2| = |s|$ list insertions. Therefore, in computing $W = \mathbf{eval}'(s, V)$ the number of list insertions is bounded by

$$\begin{aligned} 3^{d(s_1)}|s_1| + 3^{d(s_2)}|s_2| + |s_1| \times |s_2| &\leq (3^{d(s_1)} + 3^{d(s_2)} + 1)|s_1| \times |s_2| \\ &\leq (3 \times 3^{\mathbf{max}(d(s_1), d(s_2))})|s_1| \times |s_2| \\ &\leq 3^{d(s)}|s|. \end{aligned}$$

□

Theorem 8.2.8 For a fixed linear marking tree automaton, the bottom-up algorithm with partially lazy multiset operations runs in time $O(|t| + |W|)$.

Proof: We can easily show that, for a given tree t, MATCHING returns a mapping M from states to symbolic multisets in $O(|t|)$ time, by induction on the structure of t. The result follows from Lemma 8.2.7. □

Exercise 8.2.9 (⋆) Redo Exercises 8.2.4 and 8.2.6 using partially lazy multiset operations. Observe how wasted computation is avoided. ■

Exercise 8.2.10 (⋆⋆) In the bottom-up membership algorithm with top-down filtering, we additionally pass around a set of states that will restrict the states to be matched by the node. Modify the bottom-up marking algorithm so as to incorporate this idea. ■

8.3 Containment algorithms

Let us now turn our attention to the problem of checking whether the language of one tree automaton is included in that of another. This containment operation is known to be EXPTIME-complete (Theorem 4.4.5) and hence we cannot avoid an exponential blow-up in the worst case. However, we can attain reasonable efficiency in practical cases by using an on-the-fly algorithm. In this section, we first give a bottom-up algorithm which can easily be derived by a combination of the set operations given in Section 4.4. Then we present a top-down algorithm, which is much more involved but allows an important optimization called *state sharing*; this has a huge impact in practice.

8.3.1 Bottom-up algorithm

It is easy to see that containment between two tree automata A and B can be checked by successively performing the basic set operations presented in Section 4.4. That is, we first take the complement of B, then take its intersection with A, and finally check the emptiness of the resulting automaton. The disadvantages of this approach are that each intermediate step generates a large tree automaton and that the state space of the automaton after the second step may not be explored entirely by the emptiness test in the final step, thus wasting the unexplored states.

The bottom-up algorithm eliminates these disadvantages by combining all three steps. Let two automata $A = (Q_A, I_A, F_A, \Delta_A)$ and $B = (Q_B, I_B, F_B, \Delta_B)$ be given as inputs. To simplify explanation, let us first combine the first two steps, which yields the following automaton $(Q_C, I_C, F_C, \Delta_C)$ representing the difference $\mathcal{L}(A) \cap \overline{\mathcal{L}(B)}$:

$$
\begin{aligned}
Q_C &= Q_A \times 2^{Q_B} \\
I_C &= \{(q, p) \mid q \in I_A,\ p \cap I_B = \emptyset\} \\
F_C &= \{(q, F_B) \mid q \in F_A\} \\
\Delta_C &= \{(q, p) \to a((q_1, p_1), (q_2, p_2)) \mid p_1, p_2 \subseteq Q_B,\ q \to a(q_1, q_2) \in \Delta_A, \\
&\qquad\qquad p = \{q' \mid q' \to a(q'_1, q'_2) \in \Delta_B,\ q'_1 \in p_1,\ q'_2 \in p_2\}\}
\end{aligned}
$$

Combining this with the emptiness check given in Section 4.4.2, we obtain the following algorithm:

1: **function** IsSubset($(Q_A, I_A, F_A, \Delta_A), (Q_B, I_B, F_B, \Delta_B)$)
2: ▷ returning a boolean
3: $Q_{\text{bot}} \leftarrow \{(q, F_B) \mid q \in F_A\}$
4: **repeat**
5: **for all** $p_1, p_2 \subseteq Q_B,\ q \to a(q_1, q_2) \in \Delta_A$

6: s.t. $(q_1, p_1), (q_2, p_2) \in Q_{\mathrm{bot}}$ **do**
7: **let** $p = \{q' \mid q' \to a(q'_1, q'_2) \in \Delta_B,\ q'_1 \in p_1,\ q'_2 \in p_2\}$
8: $Q_{\mathrm{bot}} \leftarrow Q_{\mathrm{bot}} \cup \{(q, p)\}$
9: **end for**
10: **until** Q_{bot} does not change
11: **return** $\neg\exists (q, p) \in Q_{\mathrm{bot}}.\ q \in I_A \ \wedge\ p \cap I_B = \emptyset$ $\quad \triangleright$ i.e., $Q_{\mathrm{bot}} \cap I_C = \emptyset$
12: **end function**

This algorithm can be understood as follows. The variable Q_{bot} holds a set of pairs (q, p), where q is a state of A and p is a set of states of B. Note that, in the constructed subset tree automaton, each state accepts a tree if and only if the state is the set of all original states accepting t (cf. the proof of Theorem 4.3.5). Thus, in the algorithm, we hold as an invariant that for each pair (q, p) in Q_{bot} there is a tree t such that q is a state (of A) accepting t and p is the set of all states (of B) accepting t.

At the beginning we set Q_{bot} to the set of pairs (q, F_B) with $q \in F_A$, since q is a state accepting a leaf # and F_B is the set of all states accepting a leaf #. Then we pick up pairs (q_1, p_1) and (q_2, p_2) from Q_{bot} and a transition $q \to a(q_1, q_2)$ from Δ_A, and obtain p as in line 7. Since for (q_1, p_1) and (q_2, p_2) there are trees t_1 and t_2, respectively, satisfying the invariant, we can also find a tree for (q, p) satisfying the invariant, namely $a(t_1, t_2)$. Therefore the new pair (q, p) can be added to Q_{bot}. We repeat this until Q_{bot} does not change. Finally, if Q_{bot} has no pair (q, p) such that q is an initial state of A but p contains no initial state of B, that is, if no evidence is found that some tree is accepted by A but not by B then we judge that A's language is contained in B's.

8.3.2 Top-down algorithm

Next we present a top-down algorithm that explores the state space by starting from the initial states. This algorithm can be viewed as combining the construction of a "difference" automaton and an emptiness test, just as in the bottom-up algorithm. However, now we need to both take the difference and test the emptiness in different ways, since the methods used in the last subsection are intrinsically bottom-up and therefore combining them yields only a bottom-up algorithm, not a top-down one. In particular, it is difficult to explore the difference automaton given there in a top-down way. Indeed, it can have a huge number of initial states – how many p's are there such that $p \cap I_B = \emptyset$? Moreover, the definition of Δ_C in the previous subsection can yield a huge number of transitions from each state – how can we compute p_1 and p_2 from a given p such that $p = \{q' \mid q' \to a(q'_1, q'_2) \in$

$\Delta_B,\ q_1' \in p_1,\ q_2' \in p_2\}$, other than by iterating over all sets of states for p_1 and p_2?

Therefore we will use another construction, for a difference automaton whose state space is easier to explore from the initial states.

Difference automaton

The idea here is to compute a difference automaton without going through determinization. Let two automata $A = (Q_A, I_A, F_A, \Delta_A)$ and $B = (Q_B, I_B, F_B, \Delta_B)$ be given. We are aiming to compute an automaton with state space $Q_A \times 2^{Q_B}$ where each state (q, p) accepts the set of trees that are accepted in the state q of A but not in any state of B from p. Thus, we set each initial state to a pair (q, I_B) with $q \in I_A$, since we want the resulting automaton to accept the set of trees that are accepted in an initial state of A but in none of the initial states of B. Also, we set each final state to a pair (q, p) with $q \in F_A$ and $p \cap F_B = \emptyset$ since, in order for the state (q, p) to accept a leaf, q must be a final state of A and p must not contain a final state of B.

Now, what transitions should we have from each state (q, p)? For this, let us consider a necessary and sufficient condition for a tree $a(t', t'')$ to be accepted in (q, p). Considering the intended meaning of (q, p), this holds if and only if:

(A) there is a $q \to a(q', q'') \in \Delta_A$ such that t' and t'' are each accepted in q' and q''; and

(B1) there is no $r \to a(r', r'') \in \Delta_B$, where $r \in p$, such that t' and t'' are each accepted in r' and r''.

The condition (B1) is equivalent to:

(B2) for all $r \to a(r', r'') \in \Delta_B$, where $r \in p$, either t' is not accepted in r' or t'' is not accepted in r''.

For convenience, let us set

$$\Delta(p, a) = \{(r', r'') \mid r \in p,\ r \to a(r', r'') \in \Delta\}.$$

Then the condition (B2) can be rewritten as

(B3) for all $i = 1, \ldots, n$, either t' is not accepted in r_i' or t'' is not accepted in r_i'',

where $\Delta_B(p, a) = \{(r_1', r_1''), \ldots, (r_n', r_n'')\}$. This can be transformed further to

(B4) for some $J \subseteq \{1, \ldots, n\}$, we have that t' is accepted in no state from $\{r_i' \mid i \in J\}$ and t'' is accepted in no state from $\{r_i'' \mid i \in \overline{J}\}$.

(The notation $\overline{J}$ stands for $\{1, \ldots, n\} \setminus J$.) In the last transformation, we have used an exchange between conjunction and disjunction (that is, "for all" becomes "for

some" and "or" becomes "and"). This is illustrated by the following table:

	1	2	3	4	5	6	
r_i'	×		×	×			J
r_i''		×			×	×	$\overline{J}$

That is, in a column-wise reading either r_i' or r_i'' is checked (with a cross), as in the condition (B3), whereas in a row-wise reading a subset of $\{1, \ldots, 6\}$ is checked for r_i' and the complement is checked for r_i'', as in the condition (B4). Now, combining (A) and (B4) yields the condition that

> for some $q \to a(q', q'') \in \Delta_A$ and some $J \subseteq \{1, \ldots, n\}$, the subtree t' is accepted in q' but in no state from $\{r_i' \mid i \in J\}$ and the subtree t'' is accepted in q'' but in no state from $\{r_i'' \mid i \in \overline{J}\}$.

By recalling again the intended meaning of each state in the result automaton, we see that the transitions from the state (q, p) that we want have the form

$$(q, p) \to a((q', \{r_i' \mid i \in J\}), (q'', \{r_i'' \mid i \in \overline{J}\}))$$

for each $q \to a(q', q'') \in \Delta_A$ and $J \subseteq \{1, \ldots, n\}$.

In summary, from the given automata, we can compute the difference automaton $C = (Q_C, I_C, F_C, \Delta_C)$ such that

$$\begin{aligned}
Q_C &= Q_A \times 2^{Q_B} \\
I_C &= \{(q, I_B) \mid q \in I_A\} \\
F_C &= \{(q, p) \mid q \in F_A,\ p \cap F_B = \emptyset\} \\
\Delta_C &= \{(q, p) \to a((q', \{r_i' \mid i \in J\}), (q'', \{r_i'' \mid i \in \overline{J}\})) \mid q \to a(q', q'') \in \Delta_A, \\
&\qquad \Delta_B(p, a) = \{(r_1', r_1''), \ldots, (r_n', r_n'')\},\ J \subseteq \{1, \ldots, n\}\}.
\end{aligned}$$

We can state the following theorem. The proof proceeds by induction on the structure of an input tree t, following exactly the informal explanation given above; the details are omitted.

Theorem 8.3.1 For any tree t, the tree automaton C defined above accepts the tree t iff A accepts t but B does not.

Top-down emptiness

How can we check the emptiness of a tree automaton in a top-down way? The first observation is that, in order for a state q to be empty, a necessary and sufficient condition is that

- q is not final, and,
- for all transitions $q \to a(q_1, q_2)$, either q_1 or q_2 is empty.

Thus, we might write the following recursive "algorithm:"

```
1: function IsEmptyIn(q)                                   ▷ returning a boolean
2:     if q ∈ F then return false
3:     for all q → a(q1, q2) ∈ Δ do
4:         if not IsEmptyIn(q1) and not IsEmptyIn(q2) then return false
5:     end for
6:     return true
7: end function
```

Unfortunately, this function may not terminate if the automaton has a loop. A standard solution is to stop when we see a state for the second time. We then obtain the following top-down emptiness test algorithm:

```
1: function IsEmpty(Q, I, F, Δ)                            ▷ returning a boolean
2:     Q_asm ← ∅
3:     for all q ∈ I do
4:         if not IsEmptyIn(q) then return false
5:     end for
6:     return true
7:
8:     function IsEmptyIn(q)          ▷ returning a boolean, manipulating Q_asm
9:         if q ∈ Q_asm then return true
10:        if q ∈ F then return false
11:        Q_asm ← Q_asm ∪ {q}
12:        for all q → a(q1, q2) ∈ Δ do
13:            let Q′ = Q_asm
14:            if not IsEmptyIn(q1) then
15:                Q_asm ← Q′
16:                if not IsEmptyIn(q2) then return false
17:            end if
18:        end for
19:        return true
20:    end function
21: end function
```

Here, we maintain a set Q_{asm} of already encountered states, which we assume to be empty; we call the set the *assumption set*. In the main function IsEmpty, we start by assuming that there is no empty state in Q_{asm}. Then, if any initial state is

found to be non-empty according to the inner function IsEmptyIn, we return false; otherwise, we return true. In the function IsEmptyIn, we first check whether q is already in the assumption set Q_{asm} and, if so, we immediate return true. If q is final, we immediately return false. Thereafter, we add a new emptiness assumption for q. Then we proceed to examine each transition emanating from q and check the emptiness of each destination state. Note, however, that, if the first recursive call (IsEmptyIn(q_1)) fails, we need to revert to the assumption set as it used to be before the call ("backtracking") since the assumptions that were made during this call may be incorrect.

The algorithm is somewhat subtle. For example, why is it right to judge that a state is empty that has been assumed to be empty? One might suspect a vicious circle. Therefore let us give a formal correctness proof. First, it is easy to see termination.

Proposition 8.3.2 The top-down emptiness test algorithm terminates for any input.

Proof: At each call to the internal function IsEmptyIn, we add a state q to Q_{asm} whenever $q \notin Q_{\mathrm{asm}}$ and never remove an element until it returns. Since Q_{asm} never acquires elements other than those in Q and since Q is finite, the function terminates with call depth less than $|Q|$. □

Then we can show that the algorithm detects whether a given automaton is empty. We will prove separately that the algorithm returns true correctly and that it returns false correctly.

Lemma 8.3.3 The top-down emptiness test algorithm returns true if the given automaton accepts no tree.

Proof sketch: From Proposition 8.3.2, the result follows by proving the following, where a state is said k-empty if it accepts no tree whose height is k or less.

> Suppose that IsEmptyIn(q) with $Q_{\mathrm{asm}} = Q_1$ returns true with $Q_{\mathrm{asm}} = Q_2$. Then, for any k, if all states in Q_1 are k-empty then all states in Q_2 are k-empty.

The proof proceeds by induction on the call depth. □

Exercise 8.3.4 (⋆⋆⋆) Finish the proof of Lemma 8.3.3. ■

Lemma 8.3.5 The top-down emptiness test algorithm returns false if the given automaton accepts some tree.

Proof sketch: From Proposition 8.3.2, the result follows by proving that if IsEmptyIn(q) (with any Q_{asm}) returns false then q is not empty. Again the proof proceeds by induction on the call depth. □

Exercise 8.3.6 (⋆⋆) Finish the proof of Lemma 8.3.5. ■

Corollary 8.3.7 The top-down emptiness test algorithm returns true if and only if the given automaton accepts no tree.

The proof sketch of Lemma 8.3.5 tells us that when the IsEmptyIn function returns false for a state this state is definitely non-empty, regardless of the assumption set. This suggests an optimization that maintains a global "false set" holding all the detected non-empty states, so that the IsEmptyIn function can return false as soon as it encounters a state in this set.

Top-down containment

Combining the difference automaton construction and the top-down emptiness check presented above, we obtain the top-down containment algorithm as follows.

```
1:  function IsSubset((Q_A, I_A, F_A, Δ_A), (Q_B, I_B, F_B, Δ_B))
2:      Q_asm ← ∅                                          ▷ returning a boolean
3:      for all q ∈ I_A do
4:          if not IsSubsetOf(q, I_B) then return false
5:      end for
6:      return true
7:
8:      function IsSubsetOf(q, p)      ▷ returning a boolean, manipulating Q_asm
9:          if (q, p) ∈ Q_asm then return true
10:         if q ∈ F_A and p ∩ F_B = ∅ then return false
11:         Q_asm ← Q_asm ∪ {(q, p)}
12:         for all q → a(q', q'') ∈ Δ_A do
13:             let {(r'_1, r''_1), ..., (r'_n, r''_n)} = Δ_B(p, a)
14:             for all J ⊆ {1, ..., n} do
15:                 let Q' = Q_asm
16:                 if not IsSubsetOf(q', {r'_i | i ∈ J}) then
17:                     Q_asm ← Q'
18:                     if not IsSubsetOf(q'', {r''_i | i ∈ J̄}) then return false
19:                 end if
20:             end for
21:         end for
22:         return true
23:     end function
24: end function
```

State sharing

We have presented a rather intricate construction for a top-down containment algorithm, but what is its advantage relative to a bottom-up containment algorithm? One answer is that it enables a powerful *state sharing* technique.

Let us first give a motivation for this technique. Suppose that we want to use our containment algorithm for checking the following relation between types:

$$\texttt{Person?} \subseteq \texttt{Person*}$$

where `Person` itself is defined as follows:

```
Person = person[...]
```

Here, the content of the `person` label is not shown. Observe, however, that, whatever the content, the containment relation should hold. Thus we would wish that our containment algorithm can also decide this relation without looking at the content of `person` – this requirement can indeed have a big impact when the content type is a large expression.

Here is how we can incorporate this idea in our top-down algorithm by a slight modification. First of all, in practical situations it is often possible to make two input automata $(Q_A, I_A, F_A, \Delta_A)$ and $(Q_B, I_B, F_B, \Delta_B)$ share their states in such a way that $Q_A = Q_B$, $F_A = F_B$, and $\Delta_A = \Delta_B$ (but possibly $I_A \neq I_B$). In particular, this can easily be done when the automata come from schemas such as that in the last paragraph: in the example, we can share the states that correspond to the content of the `person` label. Then, when the top-down algorithm receives two automata that share their states, an argument (q, p) passed to the internal function IsSubsetOf may satisfy $q \in p$, in which case we can immediately return true since it is obvious that any tree accepted by q is accepted by some state from p.

Exercise 8.3.8 (⋆) Suppose that we represent the types `Person?` and `Person*` by tree automata that have a shared state q representing the content of `person`. Confirm that the top-down containment algorithm will encounter the pair $(q, \{q\})$ and thus quickly return the answer. ■

Notice that the bottom-up algorithm cannot use the same technique, since this explores the state space from the final states, and this means that, by the time we encounter a pair (q, p) with $q \in p$, we have already seen all the states below and therefore it is too late to take any action.

8.4 Bibliographic notes

Variants of the three algorithms for the membership checking are described in Murata *et al.* (2001), which presents algorithms dealing directly with schemas rather than tree automata.

A number of efforts have been made towards finding a fast marking algorithm. Early work gave a linear-time two-pass algorithm for the unary case (marking with a single variable), on the basis of boolean attribute grammars (Neven and den Bussche, 1998). Later, a linear-time (both in the sizes of the input and the output) three-pass algorithm was discovered for the general case (Flum *et al.*, 2002). A refinement of the last-mentioned algorithm with partially lazy set operations is the bottom-up single-pass algorithm presented in this chapter; this was given in Inaba and Hosoya (2007, 2009) with a slightly different presentation. Meanwhile, several other algorithms have been found that have either higher complexity or linear-time complexity for more restricted cases (Koch, 2003; Berlea and Seidl, 2002; Planque *et al.*, 2005), though in each of these papers orthogonal efforts have been made in either implementation or theory. These cited papers refer to the marking problem by the name "MSO querying," since marking tree automata are equivalent to the MSO (monadic second-order) logic (Chapter 13).

The first presentation of an on-the-fly top-down algorithm for checking tree automata containment was given in Hosoya *et al.* (2004). Several improvements on its backtracking behavior have also been proposed: in Suda and Hosoya (2005) these are based on containment dependencies and in Frisch (2004) they are based on a local constraint solver. A completely different bottom-up algorithm based on binary decision diagrams was proposed in Tozawa and Hagiya (2003).

9

Alternating tree automata

This chapter introduces an extension of tree automata called alternating tree automata. While the usual tree automata have nondeterminism, which can be seen as a "set union" operator, alternating tree automata further allow a "set intersection" operator. This extension does not change the expressiveness nor solve any difficult problem on tree automata in the complexity-theoretic sense. However, explicit intersection operations are extremely useful in representing complicated conditions, and we will see such examples in Chapters 11 and 12.

9.1 Definitions

An *alternating tree automaton* A is a quadruple (Q, I, F, Φ), where Q is a finite set of states, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and Φ is a function that maps each pair (q, a) of a state and a label to a transition formula. *Transition formulas* are defined by the following grammar:

$$\phi \quad ::= \quad \phi \vee \phi \mid \phi \wedge \phi \mid \top \mid \bot \mid \downarrow_i q$$

(with $i = 1, 2$). That is, transition formulas form a Boolean logic with no negation but with atomic propositions of the form $\downarrow_i q$, which intuitively means that the ith subtree is accepted in the state q.

The semantics of an alternating tree automaton $A = (Q, I, F, \Phi)$ is given by inductively defining the acceptance of a tree in a state:

- A accepts a leaf # in a state q when $q \in F$, and
- A accepts an intermediate node $a(t_1, t_2)$ in a state q when $(t_1, t_2) \vdash \Phi(q, a)$ holds.

Here the judgment $(t_1, t_2) \vdash \phi$ is in turn defined inductively as follows:

- $(t_1, t_2) \vdash \phi_1 \wedge \phi_2$ if $(t_1, t_2) \vdash \phi_1$ and $(t_1, t_2) \vdash \phi_2$;
- $(t_1, t_2) \vdash \phi_1 \vee \phi_2$ if $(t_1, t_2) \vdash \phi_1$ or $(t_1, t_2) \vdash \phi_2$;

- $(t_1, t_2) \vdash \top$;
- $(t_1, t_2) \vdash \downarrow_i q$ if A accepts t_i in q.

Then A accepts a tree t if A accepts t in some initial state $q_0 \in I$. We define the language $\mathcal{L}(A)$ of A by the set of trees accepted by A; a tree language accepted by some alternating tree automaton is called an *alternating tree language*; let **ATL** be the class of alternating tree languages.

Example 9.1.1 Let $\Sigma = \{a, b, c\}$ and let an alternating tree automaton $A_{9.1.1} = (\{q_0, q_1, q_2, q_3\}, \{q_0\}, \{q_3\}, \Phi)$, where Φ is defined by

$$\begin{aligned} \Phi(q_0, a) &= \downarrow_1 q_1 \wedge \downarrow_2 q_2 \vee \downarrow_1 q_2 \wedge \downarrow_2 q_1 \\ \Phi(q_1, b) &= \downarrow_1 q_3 \wedge \downarrow_2 q_3 \\ \Phi(q_2, c) &= \downarrow_1 q_3 \wedge \downarrow_2 q_3 \end{aligned}$$

the other cases being set to $\bot$. This represents exactly the same language as the tree automaton $A_{4.3.1}$. ■

Example 9.1.2 Let $\Sigma = \{a, b\}$ and let an alternating tree automaton $A_{9.1.2} = (\{q_0, q_1, q_2, q_3\}, \{q_0\}, \{q_1, q_2, q_3\}, \Phi)$, where Φ is defined by

$$\begin{aligned} \Phi(q_0, b) &= \downarrow_1 q_1 \wedge \downarrow_1 q_2 \\ \Phi(q_1, a) &= \downarrow_1 q_1 \wedge \downarrow_2 q_3 \\ \Phi(q_2, a) &= \downarrow_1 q_3 \wedge \downarrow_2 q_2 \end{aligned}$$

the other cases being set to $\bot$. To see which trees are accepted by this automaton, first note that the only initial state, q_0, accepts trees of the form $b(t_1, t_2)$, where t_1 must be accepted in both states q_1 and q_2 while t_2 is unconstrained. Then observe that both q_1 and q_2 accept trees only with the label a, but those accepted in q_1 are left-biased while those accepted in q_2 are right-biased. Therefore a tree that is accepted in both q_1 and q_2 must have the form # or $a(\#, \#)$. Thus, we conclude that $A_{9.1.2}$ accepts the set of trees of the form $b(\#, t)$ or $b(a(\#, \#), t)$ for any tree t. ■

The semantics of alternating tree automata can be defined equivalently in terms of alternating runs, which we will use more often in proofs. Given an alternating tree automaton $A = (Q, I, F, \Phi)$ and a tree t, a (top-down) *alternating run* r of A on t is a mapping from $\mathbf{nodes}(t)$ to 2^Q such that

- $r(\epsilon) = \{q\}$ with $q \in I$, and
- $(r(\pi 1), r(\pi 2)) \in \mathbf{DNF}(\bigwedge_{q \in r(\pi)} \Phi(q, a))$ whenever $\mathbf{label}_t(\pi) = a$ for $\pi \in \mathbf{nodes}(t)$.

Here, $\mathbf{DNF}(\phi)$ is ϕ's disjunctive normal form, obtained by pushing intersections under unions and regrouping atoms of the form $\downarrow_i q$ for each i; the result is

formatted as a set of pairs of state sets:

$$
\begin{array}{lcl}
\mathbf{DNF}(\top) & = & \{(\emptyset, \emptyset)\} \\
\mathbf{DNF}(\bot) & = & \emptyset \\
\mathbf{DNF}(\phi_1 \wedge \phi_2) & = & \{(s_1 \cup s_1', s_2 \cup s_2') \mid (s_1', s_2') \in \mathbf{DNF}(\phi_2)\}, (s_1, s_2) \in \mathbf{DNF}(\phi_1) \\
\mathbf{DNF}(\phi_1 \vee \phi_2) & = & \mathbf{DNF}(\phi_1) \cup \mathbf{DNF}(\phi_2) \\
\mathbf{DNF}(\downarrow_1 q) & = & \{(\{q\}, \emptyset)\} \\
\mathbf{DNF}(\downarrow_2 q) & = & \{(\emptyset, \{q\})\}
\end{array}
$$

For example, if **DNF** yields $\{(\{s_1, s_2\}, \{s_3\}), (\{s_4\}, \emptyset)\}$ then this denotes $(\downarrow_1 s_1 \wedge \downarrow_1 s_2 \wedge \downarrow_2 s_3) \vee \downarrow_1 s_4$. Then, an alternating run r is successful if $r(\pi) \subseteq F$ for each $\pi \in \mathbf{leaves}(t)$.We say that A accepts a tree t when there is a successful alternating run on t. Intuitively, an alternating run assigns to each node a set of states in which the node is accepted. It is easy to see that this semantics defines exactly the same notion as before.

Exercise 9.1.3 (⋆) Give successful alternating runs of $A_{9.1.2}$ on the following trees:

$$b(\#, a(\#, \#)) \qquad b(a(\#, \#), a(\#, \#))$$ ■

Since we have added built-in conjunction in the automata framework, it would be sensible to add "intersection types" to the schema syntax also. Let us write $T_1 \& T_2$ to denote the intersection of the sets denoted by T_1 and T_2. Then, converting a type containing intersections to an alternating tree automaton can be done trivially by translating each intersection type to a conjunction, except that care is needed when one intersection type is concatenated with another, such as in $(T_1 \& T_2), T_3$. We might want to first combine sub-automata corresponding to T_1 and T_2 by conjunction, and then somehow connect the combined automaton to another corresponding to T_3. However, there is no easy way to connect them together since the sub-automata for T_1 and T_2 would need to "synchronize" just before continuing to the sub-automaton for T_3 – there is no such mechanism in alternating tree automata. What if we naively rewrote the expression $(T_1 \& T_2), T_3$ as $((T_1, T_3) \& (T_2, T_3))$, thus avoiding the issue of synchronization? Unfortunately, this would change the meaning. For example,

$$(a[] \& (a[], b[])), (b[]?)$$

denotes the empty set since the intersection of $a[]$ and $a[], b[]$ is empty, whereas the rewriting suggested above yields

$$((a[], b[]?) \& (a[], b[], b[]?)$$

which is not empty: it contains $a[]$, $b[]$. Therefore, in such a situation an intersection type needs to be expanded to an automaton without conjunction (i.e., by using product construction).

9.2 Relationship with tree automata

Clearly, alternating tree languages include regular tree languages, since alternating tree automata have both disjunction and conjunction whereas normal tree automata have only disjunction. To see the converse, we need to convert an alternating tree automaton to a normal tree automaton. We present two algorithms that achieve this, one that yields a possibly nondeterministic tree automaton and one that yields a bottom-up deterministic tree automaton. It is certainly possible to obtain a bottom-up deterministic tree automaton by using the first algorithm followed by the determinization procedure (Section 4.3.2). However, this approach takes double exponential time. The second algorithm, however, determinizes a given alternating tree automaton directly and takes only exponential time.

9.2.1 Conversion to nondeterministic tree automata

Given an alternating tree automaton $A = (Q, I, F, \Phi)$, we construct a nondeterministic tree automaton $M = (S, I', F', \Delta)$ by a subset construction:

$$\begin{array}{lcl} S & = & 2^Q \\ I' & = & \{\{q\} \mid q \in I\} \\ F' & = & \{s \mid s \subseteq F\} \\ \Delta & = & \{s \to a(s_1, s_2) \mid (s_1, s_2) \in \mathbf{DNF}(\bigwedge_{q \in s} \Phi(q, a))\} \end{array}$$

Intuitively, each state $\{q_1, \ldots, q_n\}$ in the resulting automaton M denotes the intersection of all the states $q_1, \ldots, q_n$ in the original automaton A.

It is straightforward to prove that the constructed tree automaton accepts the same language.

Theorem 9.2.1 $\mathcal{L}(M) = \mathcal{L}(A)$.

Proof: We can easily show that if r is a successful run of M on a tree t then r is also a successful alternating run of A on t, and vice versa. The result thus follows. □

Exercise 9.2.2 (⋆) Construct a nondeterministic tree automaton from the alternating tree automaton $A_{9.1.2}$ by using the above conversion algorithm. Remove the states unreachable from the initial states. ■

Corollary 9.2.3 **ATL** = **ND**.

Proof: Since **ATL** $\subseteq$ **ND** has already been proved, by Theorem 9.2.1 we need only to show **ND** $\subseteq$ **ATL**, and this is easy since, from a nondeterministic tree automaton $M = (Q, I, F, \Delta)$, we can construct an equivalent alternating tree automaton $A = (Q, I, F, \Phi)$ where, for each $q \in Q$ and $a \in \Sigma$,

$$\Phi(q, a) = \bigvee_{(q \to a(q_1, q_2)) \in \Delta} (\downarrow_1 q_1 \wedge \downarrow_2 q_2).$$

□

9.2.2 Conversion to bottom-up deterministic tree automata

Given an alternating tree automaton $A = (Q, I, F, \Phi)$, we can arrive at a bottom-up deterministic tree automaton $M = (S, I', F', \Delta)$ by another subset construction:

$$\begin{array}{lcl} S & = & 2^Q \\ I' & = & \{s \in S \mid s \cap I \neq \emptyset\} \\ F' & = & \{F\} \\ \Delta & = & \{s \to a(s_1, s_2) \mid s_1, s_2 \in S,\ s = \{q \in Q \mid (s_1, s_2) \vdash \Phi(q, a)\}\} \end{array}$$

where the judgment $(s_1, s_2) \vdash \phi$ is defined inductively as follows:

- $(s_1, s_2) \vdash \phi_1 \wedge \phi_2$ if $(s_1, s_2) \vdash \phi_1$ and $(s_1, s_2) \vdash \phi_2$;
- $(s_1, s_2) \vdash \phi_1 \vee \phi_2$ if $(s_1, s_2) \vdash \phi_1$ or $(s_1, s_2) \vdash \phi_2$;
- $(s_1, s_2) \vdash \top$;
- $(s_1, s_2) \vdash \downarrow_i q$ if $q \in s_i$.

That is, $(s_1, s_2) \vdash \phi$ means that ϕ holds if each $\downarrow_i q$ is interpreted as "q is contained in s_i." The intuition behind the construction is the same as in determinization for nondeterministic tree automata (Section 4.3.2), that is, each state $s = \{q_1, \ldots, q_n\}$ of M denotes the set of trees such that $q_1, \ldots, q_n$ are all and the only states of A accepting those trees. Clearly, the whole procedure takes at most exponential time and the resulting automaton M is bottom-up deterministic and complete.

Let us prove that the constructed bottom-up deterministic tree automaton M accepts the same language as A, i.e., $\mathcal{L}(A) = \mathcal{L}(M)$. First, it is easy to show the following technical lemma.

Lemma 9.2.4 The relation $(s_1, s_2) \vdash \phi$ holds if and only if $(s_1', s_2') \in \mathbf{DNF}(\phi)$ for some $s_1' \subseteq s_1$ and $s_2' \subseteq s_2$.

Proof: This proceeds by straightforward induction on the structure of ϕ. □

Theorem 9.2.5 $\mathcal{L}(A) = \mathcal{L}(M)$.

Proof: To show that $\mathcal{L}(M) \subseteq \mathcal{L}(A)$, suppose that r is a successful run of M on a tree t. Then we can construct a top-down alternating run r' of A on t, from the root to the leaves, such that $r'(\pi) \subseteq r(\pi)$ for all $\pi \in \mathbf{nodes}(t)$, as follows.

- Since $r(\epsilon) \cap I \neq \emptyset$, choose $q \in r(\epsilon) \cap I$ and let $r'(\epsilon) = \{q\}$.
- For $\mathbf{label}_t(\pi) = a$, since $r(\pi) \to a(r(\pi 1), r(\pi 2)) \in \Delta$, we have $r(\pi) = \{q \in Q \mid (r(\pi 1), r(\pi 2)) \vdash \Phi(q, a)\}$. Since $r'(\pi) \subseteq r(\pi)$, we have $(r(\pi 1), r(\pi 2)) \vdash \bigwedge_{q \in r'(\pi)} \Phi(q, a)$. By Lemma 9.2.4 there exist $s_1 \subseteq r(\pi 1)$ and $s_2 \subseteq r(\pi 2)$ such that $(s_1, s_2) \in \mathbf{DNF}(\bigwedge_{q \in r'(\pi)} \Phi(q, a))$. Let $r'(\pi 1) = s_1$ and $r'(\pi 2) = s_2$.

For each $\pi \in \mathbf{leaves}(t)$, since $r'(\pi) \subseteq r(\pi) = F$ we have $r'(\pi) \in F'$. Therefore r' is successful.

To show that $\mathcal{L}(A) \subseteq \mathcal{L}(M)$, suppose that r is a successful alternating run of A on a tree t. Also, since M is bottom-up deterministic and complete, there is a unique bottom-up run r' of M on t. Then we can show that $r(\pi) \subseteq r'(\pi)$ for all $\pi \in \mathbf{nodes}(t)$ by induction on the height of $\mathbf{subtree}_t(\pi)$.

- For $\pi \in \mathbf{leaves}(t)$, we have $r(\pi) \subseteq F = r'(\pi)$.
- For $\mathbf{label}_t(\pi) = a$, we have $(r(\pi 1), r(\pi 2)) \in \mathbf{DNF}(\bigwedge_{q \in r(\pi)} \Phi(q, a))$. By the induction hypothesis, $r(\pi 1) \subseteq r'(\pi 1)$ and $r(\pi 2) \subseteq r'(\pi 2)$. By Lemma 9.2.4, $(r'(\pi 1), r'(\pi 2)) \vdash \bigwedge_{q \in r(\pi)} \Phi(q, a)$. Therefore, for all $q \in r(\pi)$, we have $(r'(\pi 1), r'(\pi 2)) \vdash \Phi(q, a)$. Further, since $r'(\pi) \to a(r'(\pi 1), r'(\pi 2)) \in \Delta$, we have $r'(\pi) = \{q \in Q \mid (r'(\pi 1), r'(\pi 2)) \vdash \Phi(q, a)\}$. Therefore $r(\pi) \subseteq r'(\pi)$.

Since $r(\epsilon) = \{q\}$ with $q \in I$ and $r(\epsilon) \subseteq r'(\epsilon)$ we conclude that $r'(\epsilon) \in I'$, that is, r' is successful. □

One may wonder why we should care about the conversion to nondeterministic tree automata (Section 9.2.1), since the conversion just presented has the same complexity yet yields a bottom-up deterministic tree automaton. One answer is that the first often yields a smaller automaton that is sufficient for purposes that do not need determinism, such as an emptiness test.

Exercise 9.2.6 (⋆) Construct a bottom-up deterministic tree automaton from the alternating tree automaton $A_{9.1.2}$ by using the conversion algorithm above. Remove all empty states. Then compare the result with that of Exercise 9.2.2. ■

9.3 Basic set operations

Since union and intersection are already built in, we consider here only the membership test, the emptiness test, and complementation. These can certainly be obtained by going through ordinary tree automata. However, it is sometimes more efficient to manipulate alternating tree automata directly, as will be discussed below.

9.3.1 Membership

We can obtain a top-down membership algorithm by combining the algorithm for conversion to a nondeterministic tree automaton (Section 9.2.1) with the top-down membership algorithm for a tree automaton (Section 8.1.1). Like the latter algorithm, that presented here has a potential blow-up since it traverses the same node many times. The following shows pseudo-code for the algorithm.

```
1:  function Accept(t, (Q, I, F, Φ))                    ▷ returning a boolean
2:      for all q ∈ I do
3:          if AcceptIn(t, {q}) then return true
4:      end for
5:      return false
6:
7:      function AcceptIn(t, s)                         ▷ returning a boolean
8:          switch t do
9:            case #:
10:             if s ⊆ F then return true else return false
11:           case a(t1, t2):
12:             for all (s1, s2) ∈ DNF(⋀_{q∈s} Φ(q, a)) do
13:                 if AcceptIn(t1, s1) and AcceptIn(t2, s2) then return true
14:             end for
15:             return false
16:         end switch
17:     end function
18: end function
```

We can also construct a linear-time bottom-up algorithm, by using a similar idea to that of the bottom-up membership algorithm (Section 8.1.2). As before, we create the states of a bottom-up deterministic tree automaton needed for deciding whether a given tree is accepted. The following shows the pseudo-code for the algorithm.[1]

[1] It might be possible to combine the top-down and bottom-up algorithms to construct a bottom-up algorithm with top-down filtering, as in Section 8.1.3, though the author is not certain about this.

```
1:  function ACCEPT(t, (Q, I, F, Φ))                    ▷ returning a boolean
2:      if ACCEPTING(t)∩I ≠ ∅ then return true else return false
3:
4:      function ACCEPTING(t)                           ▷ returning a set of states
5:          switch t do
6:            case #:
7:              return F
8:            case a(t1, t2):
9:              let s1 = ACCEPTING(t1)
10:             let s2 = ACCEPTING(t2)
11:             let s = {q | (s1, s2) ⊢ Φ(q, a)}
12:             return s
13:         end switch
14:     end function
15: end function
```

This algorithm works exactly in the same way as the bottom-up algorithm for normal tree automata (Section 8.1.2) except that, in line 11, the set s for the current node is obtained by collecting the states q such that $\Phi(q, a)$ holds with the sets s_1 and s_2 computed for the child nodes. Note that the same computation occurs in the conversion algorithm to a bottom-up deterministic tree automaton (Section 9.2.2).

9.3.2 Complementation

For nondeterministic tree automata, complementation requires an exponential-time computation (see Section 4.3.2 and Proposition 4.4.3). It is not the case, however, for alternating tree automata: complementation runs in linear time thanks to the fact that such automata have both disjunction and conjunction. This implies that if we should need to perform complementation frequently, alternating tree automata would be the right representation to use.

Let an alternating tree automaton $A = (Q, I, F, \Phi)$ be given. Without loss of generality, we can assume that I is a singleton set $\{q_0\}$ (otherwise, we could combine all initial states, taking the union of all their transitions). Then we construct the alternating tree automaton $A' = (Q, I, Q \setminus F, \Phi')$ such that $\Phi'(q, a) = \mathbf{flip}(\Phi(q, a))$, where $\mathbf{flip}(\phi)$ is defined inductively as follows:

$$
\begin{array}{lcl}
\mathbf{flip}(\top) & = & \bot \\
\mathbf{flip}(\bot) & = & \top \\
\mathbf{flip}(\phi_1 \wedge \phi_2) & = & \mathbf{flip}(\phi_1) \vee \mathbf{flip}(\phi_2) \\
\mathbf{flip}(\phi_1 \vee \phi_2) & = & \mathbf{flip}(\phi_1) \wedge \mathbf{flip}(\phi_2) \\
\mathbf{flip}(\downarrow_i q) & = & \downarrow_i q
\end{array}
$$

That is, we modify the given alternating automaton so that each state q now represents the complement of the original meaning. For this, we negate each formula in the transition by flipping $\wedge$ and $\vee$, $\top$ and $\bot$, and final and non-final states. Clearly, the algorithm is linear-time.

Let us prove the correctness of the algorithm. First, the following technical lemma can easily be shown.

Lemma 9.3.1 For $s_1, s_2 \subseteq Q$, if $(s_1, s_2) \vdash \phi$ then $(Q \setminus s_1, Q \setminus s_2) \nvdash \mathbf{flip}(\phi)$, and vice versa.

Proof: By straightforward induction on the structure of ϕ. □

Theorem 9.3.2 $\mathcal{L}(A) = \overline{\mathcal{L}(A')}$.

Proof: First convert both A and A' to equivalent bottom-up deterministic tree automata M and M', respectively, by the construction in Section 9.2.2. Then we can write $M = (2^Q, I', \{F\}, \Delta)$ and $M' = (2^Q, I', \{Q \setminus F\}, \Delta')$, where $I' = \{s \subseteq Q \mid s \cap I \neq \emptyset\}$ and

$$\begin{aligned} \Delta &= \{s \to a(s_1, s_2) \mid s_1, s_2 \subseteq Q,\ s = \{q \in Q \mid (s_1, s_2) \vdash \Phi(q, a)\}\} \\ \Delta' &= \{s \to a(s_1, s_2) \mid s_1, s_2 \subseteq Q,\ s = \{q \in Q \mid (s_1, s_2) \vdash \mathbf{flip}(\Phi(q, a))\}\}. \end{aligned}$$

By Lemma 9.3.1,

$$\begin{aligned} \Delta' &= \{s \to a(s_1, s_2) \mid s_1, s_2 \subseteq Q,\ s = \{q \in Q \mid (Q \setminus s_1, Q \setminus s_2) \nvdash \Phi(q, a)\}\} \\ &= \{s \to a(s_1, s_2) \mid s_1, s_2 \subseteq Q,\ s = Q \setminus \{q \in Q \mid (Q \setminus s_1, Q \setminus s_2) \vdash \Phi(q, a)\}\}. \end{aligned}$$

Since both M and M' are bottom-up deterministic and complete, there are unique bottom-up runs r of M and r' of M' on any tree t. We can easily show that $r(\pi) = Q \setminus r'(\pi)$ for all $\pi \in \mathbf{nodes}(t)$, by induction on the height of $\mathbf{subtree}_t(\pi)$. If r is successful, that is, $r(\epsilon) \cap I \neq \emptyset$, then since $I = \{q_0\}$, as assumed at the beginning, $r'(\epsilon) \cap I = (Q \setminus r(\epsilon)) \cap I = \emptyset$ and therefore r' is not successful. Likewise, if r' is successful then r is not. □

9.3.3 Emptiness test

First, it is easy to prove that checking the emptiness of an alternating tree automaton takes exponential time at worst. This implies that switching from ordinary tree automata to alternating ones does not solve a difficult problem but simply moves it to a different place. Nevertheless, experience tells us that in practice it is often more efficient to deal with alternating tree automata directly.

Theorem 9.3.3 The emptiness problem for alternating tree automata is EXPTIME-complete.

Proof: The problem is in EXPTIME since an alternating tree automaton can be converted to an ordinary tree automaton in exponential time (Section 9.2.1); then, checking the emptiness of the resulting automaton can be achieved in polynomial time (Section 4.4.2). The problem is EXPTIME-hard since the containment problem for tree automaton, which is EXPTIME-complete (Theorem 4.4.5), can be reduced in polynomial time to the present problem, since the reduction uses linear-time complementation (Section 9.3.2). □

Exercise 9.3.4 (⋆⋆) Derive a bottom-up emptiness test algorithm for alternating tree automata by combining the algorithm for conversion to deterministic bottom-up tree automata (Section 9.2.2) and the bottom-up emptiness test algorithm for tree automata (Section 4.4.2). ■

Exercise 9.3.5 (⋆⋆) Derive a top-down emptiness test algorithm for alternating tree automata by combining the algorithm for conversion to nondeterministic tree automata (Section 9.2.1) and the top-down emptiness check for tree automata (Section 8.3.2). Discuss the advantages and disadvantages of the bottom-up algorithm (Exercise 9.3.4) and the top-down algorithm. ■

9.4 Bibliographic notes

Alternating tree automata were first introduced and investigated in Slutzki (1985). The notion has since been exploited in XML typechecking. For example, the implementation of ℂDuce (Frisch *et al.*, 2008; Benzaken *et al.*, 2003) makes considerable use of alternating tree automata (extended by explicit complementation operations) as an internal representation of types for XML. A top-down algorithm for the emptiness check can be found in Frisch (2004).

10

Tree transducers

Tree transducers form a class of finite-state models that not only accept trees but also transform them. While in Chapter 7 we described a simple yet very powerful tree transformation language, μXDuce, tree transducers are more restricted and work with only a finite number of states. An important aspect of the restriction is that tree transducers can never use intermediate data structure. However, this restriction is not too unrealistic, as it is one of the design principles of the XSLT language – currently the most popular language for XML transformation.[1] In addition, this restriction leads to several nice properties that otherwise would hardly hold; among the most important is the exact typechecking property to be detailed in Chapter 11. The present chapter introduces two standard tree transducer frameworks, namely, top-down tree transducers and macro tree transducers, and shows their basic properties.

10.1 Top-down tree transducers

Top-down tree transducers involve a form of transformation that traverses a given tree from the root to the leaves while at each node producing a fragment of the output tree determined by the label of the current node and the current state. Since a state of a tree transducer can then be seen as a rule from a label to a tree fragment, we call a state a *procedure* from now on.

When considering the tree transducer family, we often take a nondeterministic approach, as in the case of automata. When a transducer is nondeterministic a single procedure may have a choice of multiple rules for the same label and thus may produce multiple results. There are several reasons why we consider nondeterminism. First, as we will see, this actually changes the expressiveness. Second, nondeterminism can be used as a means of "abstracting" a more complex

[1] In fact the most recent version of XSLT has dropped this property, though the "spirit" of the property is still there.

computation on which it is easier to perform static analysis. We will come back to this second point in Chapter 11.

Formally, a *top-down tree transducer* $\mathcal{T}$ is a triple (P, P_0, Π), where P is a finite set of procedures (states), $P_0 \subseteq P$ is a set of initial procedures, and Π is a finite set of (transformation) rules each having either of the following forms:

$$\begin{array}{lcll} p(a(x_1, x_2)) & \to & e & \text{(node rule)} \\ p(\#) & \to & e & \text{(leaf rule)} \end{array}$$

where $p \in P$ and $a \in \Sigma$. Here, the expressions e are defined by

$$e \quad ::= \quad a(e_1, e_2) \mid \# \mid p(x_h)$$

where the form $p(x_h)$ with $h = 1, 2$ can appear only in node rules. The semantics of the top-down tree transducer is defined by the denotation function $[\![\cdot]\!]$ given below. First, a procedure p takes a tree t and returns the set of trees resulting from evaluating any of p's rules. In the evaluation of a rule we first match and deconstruct the input tree with the head of the rule and then evaluate the body expression, with the pair of t's children for a node rule or a dummy (written $\_$) for a leaf rule:

$$\begin{array}{lcl} [\![p]\!](a(t_1, t_2)) & = & \bigcup_{(p(a(x_1,x_2))\to e)\in\Pi} [\![e]\!](t_1, t_2) \\ [\![p]\!](\#) & = & \bigcup_{(p(\#)\to e)\in\Pi} [\![e]\!]\_ \end{array}$$

Below, when we write p we mean either a pair of trees or a dummy. Then an expression e is evaluated as follows:

$$\begin{array}{lcl} [\![a(e_1, e_2)]\!]\rho & = & \{a(u_1, u_2) \mid u_i \in [\![e_i]\!]\rho \quad \text{for} \quad i = 1, 2\} \\ [\![\#]\!]\rho & = & \{\#\} \\ [\![p(x_h)]\!](t_1, t_2) & = & [\![p]\!](t_h) \end{array}$$

That is, a constructor expression $a(e_1, e_2)$ evaluates each subexpression e_i and reconstructs a tree node with label a and the results of these subexpressions. A leaf expression # evaluates to itself. A procedure call $p(x_h)$ evaluates the procedure p with the hth subtree. (Recall that a procedure call can appear in a node rule, to which a pair of trees is always given.) The whole semantics of the transducer with respect to a given input tree t is defined by the evaluation of any of the initial procedures: $\mathcal{T}(t) = \bigcup_{p_0 \in P_0} [\![p_0]\!](t)$.

A transducer $\mathcal{T}$ is *deterministic* when it has at most one rule $p(a(x_1, x_2)) \to e$ for each procedure p and label a and at most one rule $p(\#) \to e$ for each procedure p. A transducer $\mathcal{T}$ is *total* when it has at least one rule $p(a(x_1, x_2)) \to e$ for each p, a and at least one rule $p(\#) \to e$ for each p. For a deterministic (total) transducer, $\mathcal{T}(t)$ has at most (resp. at least) one element for any t.

Example 10.1.1 Let $\mathcal{T}_{10.1.1} = (\{p_0, p_1\}, \{p_0\}, \Pi)$, where Π consists of the following:

$$\begin{array}{lcl} p_0(a(x_1, x_2)) & \to & a(p_1(x_1), p_0(x_2)) \\ p_0(b(x_1, x_2)) & \to & c(p_0(x_1), p_0(x_2)) \\ p_0(\#) & \to & \# \\ p_1(a(x_1, x_2)) & \to & a(p_1(x_1), p_0(x_2)) \\ p_1(b(x_1, x_2)) & \to & p_0(x_2) \\ p_1(\#) & \to & \# \end{array}$$

This transducer replaces every b node with its right subtree if the node appears as the left child of an a node; otherwise, the b node is renamed c. The procedure p_1 is called when the current node appears as the left child of an a node; p_0 is called in any other context. Then, for an a node, both p_0 and p_1 retain the node label and call p_1 for the left child and p_0 for the right child. For a b node, however, p_0 renames its label as c, while p_1 only calls p_0 for its right child (thus removing the current b label and its entire left subtree). ■

Exercise 10.1.2 (⋆) Transform the following trees by $\mathcal{T}_{10.1.1}$:

$$a(b(\#, b(\#, \#)), \#) \qquad a(b(b(\#, \#), \#), \#)$$

■

Determinism indeed changes the expressive power. The following example shows that, in the nondeterministic case, even when a transducer cannot decide exactly which tree fragment to produce given only the current node and state, it can still produce several possibilities for the moment and later discard some of them after looking at a descendant node.

Example 10.1.3 Let $\mathcal{T}_{10.1.3} = (\{p_0, p_1, p_2, p_3\}, \{p_0\}, \Pi)$, where Π consists of the following:

$$\begin{array}{lcl} p_0(a(x_1, x_2)) & \to & a(p_1(x_1), p_3(x_2)) \\ p_0(a(x_1, x_2)) & \to & b(p_2(x_1), p_3(x_2)) \\ p_1(c(x_1, x_2)) & \to & c(p_3(x_1), p_3(x_2)) \\ p_2(d(x_1, x_2)) & \to & d(p_3(x_1), p_3(x_2)) \\ p_3(\#) & \to & \# \end{array}$$

This translates $a(c(\#, \#), \#)$ to itself and $a(d(\#, \#), \#)$ to $b(d(\#, \#), \#)$. This translation cannot be expressed by a deterministic top-down transducer since a decision about whether to change the top a label to a b label can only be made after looking at the left child. ■

Exercise 10.1.4 (⋆) Transform the tree $a(a(\#, \#), \#)$ by $\mathcal{T}_{10.1.3}$. ■

10.2 Height property

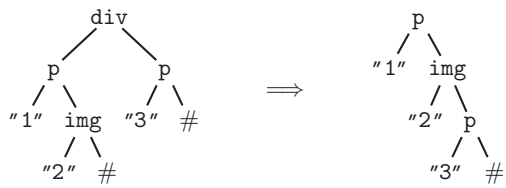
A natural question that arises next is which transformations *cannot* be expressed by a top-down tree transducer. As an example, consider a transformation that deletes each *b* label from the input in such a way that in a binary tree format the right tree is moved to the rightmost leaf of the left tree. This kind of transformation often happens in XML processing: for instance, we may want to delete a `div` tag from a value such as

```
div[p["1"], img["2"]], p["3"]
```

while retaining the content of the `div`:

```
p["1"], img["2"], p["3"]
```

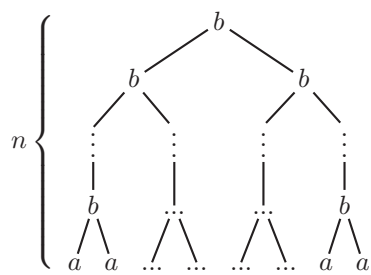
In the binary tree format, this transformation can be drawn as follows:



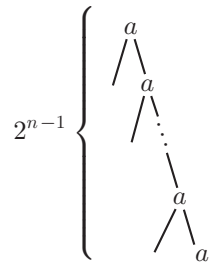
Now, the question is: "Is this transformation expressible by a top-down tree transducer?" The answer is "No." But how can we prove this?

A powerful technique for characterizing tree transformations is a *height property*, which tells us how tall an output tree of a given transformation can be with respect to the input tree. Below, we will show the upper bound of the height increase for top-down tree transducers. By using this upper bound, we can show that a particular transformation is not expressible by a top-down tree transducer, and thus we can compare different formalisms for tree transducers.

For example, the above delete-*b* transformation can increase the height of the tree exponentially. To see this, take the following full binary tree of height *n* (the leaves # are omitted):



Then, from this tree, the delete-b transformation will produce a right-biased tree of height 2^{n-1}:



However, as will be seen below, we can prove that any output tree produced by a top-down tree transducer has only a linear height increase from the input tree.

Theorem 10.2.1 Let $\mathcal{T}$ be a top-down tree transducer and t be a tree. Then there exists $c > 0$ such that $\mathbf{ht}(u) \leq c\,\mathbf{ht}(t)$ for any $u \in \mathcal{T}(t)$. (The function **ht** was defined in Section 4.1.1.)

Proof: Let c be the maximum height of the expressions appearing in all rules. Then the result can be proved by induction on the height of t. □

This result implies that the delete-b transformation is not expressible by a top-down transducer. The next question is: "What form of tree transducer can express such a transformation?" This will be answered in the next section.

10.3 Macro tree transducers

In this section we introduce an extension of top-down tree transducers with parameters. Since, historically, these extended transducers were invented for studying "macro expansion" in programming languages, they are called *macro tree transducers*. As for an unextended top-down tree transducer, each procedure of a macro tree transducer takes an implicit parameter that refers to a node in the input tree. However, it can also take extra parameters that refer not to nodes in the input but to tree fragments that will potentially become part of the output tree. We will see below that this additional functionality provides a substantial increase in expressiveness.

Formally, a *macro tree transducer* $\mathcal{T}$ is a triple (P, P_0, Π), where P and P_0 are the same as for the top-down tree transducer in Section 10.1 and Π is a finite set of

rules each having either of the following forms:

$$\begin{array}{lcll} p^{(k)}(a(x_1, x_2), y_1, \ldots, y_k) & \to & e & \text{(node rule)} \\ p^{(k)}(\#, y_1, \ldots, y_k) & \to & e & \text{(leaf rule)} \end{array}$$

Here, each y_i is called an *(accumulating) parameter*. We will abbreviate a tuple $(y_1, \ldots, y_k)$ of parameters as $\vec{y}$. Note that each procedure is now associated with its "arity", that is, its number of parameters; we write $p^{(k)}$ to denote a k-arity procedure p. The expressions e are defined by the following grammar:

$$e \quad ::= \quad a(e_1, e_2) \mid \# \mid p^{(l)}(x_h, e_1, \ldots, e_l) \mid y_j$$

where only y_j with $1 \leq j \leq k$ can appear in a rule for a k-arity procedure and $p(x_h, \ldots)$ with $h = 1, 2$ can appear only in a node rule. We assume that each initial procedure has arity zero. As in the case of an unextended top-down tree transducer, the semantics of the macro tree transducer is defined by the denotation function $[\![\cdot]\!]$. First, a procedure $p^{(k)}$ takes a current tree as well as a k-tuple of parameters $\vec{w}$ and returns the set of trees resulting from evaluating any of p's rules:

$$\begin{array}{lcl} [\![p^{(k)}]\!](a(t_1, t_2), \vec{w}) & = & \bigcup_{(p^{(k)}(a(x_1,x_2),\vec{y})\to e)\in\Pi} [\![e]\!]((t_1, t_2), \vec{w}) \\ [\![p^{(k)}]\!](\#, \vec{w}) & = & \bigcup_{(p^{(k)}(\#,\vec{y})\to e)\in\Pi} [\![e]\!](\_, \vec{w}) \end{array}$$

Then, an expression e takes a pair of trees or a dummy as well as parameters $\vec{w} = (w_1, \ldots, w_k)$ and returns the set of trees resulting from the evaluation:

$$\begin{array}{lcl} [\![a(e_1, \ldots, e_m)]\!](\rho, \vec{w}) & = & \{a(u_1, u_2) \mid u_i \in [\![e_i]\!](\rho, \vec{w}) \text{ for } i = 1, 2\} \\ [\![\#]\!](\rho, \vec{w}) & = & \{\#\} \\ [\![p^{(l)}(x_h, e_1, \ldots, e_l)]\!]((t_1, t_2), \vec{w}) & = & \bigcup\{[\![p^{(l)}]\!](t_h, (w'_1, \ldots, w'_l)) \\ & & \quad \mid w'_j \in [\![e_j]\!]((t_1, t_2), \vec{w}) \text{ for } j = 1, \ldots, l\} \\ [\![y_j]\!](\rho, \vec{w}) & = & \{w_j\} \end{array}$$

The difference from the case of top-down transducers is that a procedure call $p^{(l)}(x_h, e_1, \ldots, e_l)$ gives the results of $e_1, \ldots, e_l$ as parameters when evaluating the procedure $p^{(l)}$ with the hth subtree t_h. Also, a rule for a variable expression y_j is added; this rule simply returns the corresponding parameter's value w_j. The whole semantics of the macro tree transducer with respect to a given input tree t is defined by $\mathcal{T}(t) = \bigcup_{p_0^{(0)} \in P_0} [\![p_0^{(0)}]\!](t)$ (with no parameter since the initial procedures have arity zero). Determinism and totality can be defined in a similar way to before.

Example 10.3.1 Let us express the delete-b transformation discussed in Section 10.2 by a macro tree transducer. Let $\mathcal{T}_{10.3.1} = (\{p_0, p_1\}, \{p_0\}, \Pi)$, where Π

consists of the following rules (omitting the arity superscripts):

$$\begin{array}{lcl} p_0(a(x_1, x_2)) & \to & a(p_1(x_1, \#), p_1(x_2, \#)) \\ p_0(b(x_1, x_2)) & \to & p_1(x_1, p_1(x_2, \#)) \\ p_0(\#) & \to & \# \\ p_1(a(x_1, x_2), y) & \to & a(p_1(x_1, \#), p_1(x_2, y)) \\ p_1(b(x_1, x_2), y) & \to & p_1(x_1, p_1(x_2, y)) \\ p_1(\#, y) & \to & y \end{array}$$

That is, the procedure p_1 takes an accumulating parameter y that will be appended after the current tree, interpreted as a sequence in XML format. Thus, when we encounter a leaf, we emit y. When we encounter an a label, we first copy the a and then make a recursive call to p_1 for each child node, where we give y for the right child since we are still in the same sequence while we give # for the left child since now we are going into a new sequence. When we encounter a b label, we do not copy the b but make two recursive calls, where we give y for the right child while, for the left child, we give the result from the right child; in this way, we can append the result for the left child to the result for the right child.

Now we would like to start up the transducer with initial parameter #. However, we cannot simply write

$$p_0(x) \to p_1(x, \#)$$

since a macro tree transducer, as defined, needs to reduce one label for each procedure call. (Although, we could extend transducers with direct procedure calls; see the bibliographic notes.) Therefore instead we define the procedure p_0 to be exactly the same as p_1 except that it does not take an accumulating parameter but uses # wherever p_1 would use y. ■

Exercise 10.3.2 (⋆) Transform the following tree by $\mathcal{T}_{10.3.1}$:

$$b(a(\#, b(a(a(\#, \#), \#), \#)), a(\#, \#))$$

■

Example 10.3.3 Let us write a more realistic macro tree transducer that, given an XHTML document, puts the list of all `img` elements at the end of the `body` element. Assume that Σ is the set of all XHTML labels. Let $\mathcal{T}_{10.3.3} = (\{\mathsf{main}, \mathsf{getimgs}, \mathsf{putimgs}, \mathsf{putimgs2}, \mathsf{copy}\}, \{\mathsf{main}\}, \Pi)$, where Π consists

of the followings rules (again omitting arities):

$$
\begin{array}{lcl}
\mathsf{main}(\mathtt{html}(x_1, x_2)) & \to & \mathtt{html}(\mathsf{putimgs}(x_1, \mathsf{getimgs}(x_1, \#)), \mathsf{copy}(x_2)) \\
\\
\mathsf{getimgs}(\mathtt{img}(x_1, x_2), y) & \to & \mathtt{img}(\mathsf{copy}(x_1), \mathsf{getimgs}(x_2, y)) \\
\mathsf{getimgs}(*(x_1, x_2), y) & \to & \mathsf{getimgs}(x_1, \mathsf{getimgs}(x_2, y)) \\
\mathsf{getimgs}(\#, y) & \to & y \\
\\
\mathsf{putimgs}(\mathtt{body}(x_1, x_2), y) & \to & \mathtt{body}(\mathsf{putimgs2}(x_1, y), \mathsf{copy}(x_2)) \\
\mathsf{putimgs}(*(x_1, x_2), y) & \to & *(\mathsf{putimgs}(x_1, y), \mathsf{putimgs}(x_2, y)) \\
\mathsf{putimgs}(\#, y) & \to & \# \\
\\
\mathsf{putimgs2}(*(x_1, x_2), y) & \to & *(\mathsf{copy}(x_1), \mathsf{putimgs2}(x_2, y)) \\
\mathsf{putimgs2}(\#, y) & \to & y \\
\\
\mathsf{copy}(*(x_1, x_2)) & \to & *(\mathsf{copy}(x_1), \mathsf{copy}(x_2)) \\
\mathsf{copy}(\#) & \to & \#
\end{array}
$$

Here, each rule with a star ($*$) pattern should be read as a set of rules in which the star label in the pattern and all star constructors in the body are replaced by each label not matched by the preceding rules of the same procedure.

The initial main procedure first collects the list of all `img` elements from the given tree by using getimgs, and then passes it in the putimgs procedure. The putimgs procedure, when it finds the `body` element, uses putimgs2 to append the `img` list at the end of the content sequence. The auxiliary copy procedure simply copies the whole subtree. ■

In Example 10.3.1 we saw a macro tree transducer that expresses the delete-b transformation, which can have an exponential increase in the tree height as discussed in Section 10.2. More generally, we can assert that for any macro tree transducer the tree height grows at most exponentially.

Theorem 10.3.4 Let $\mathcal{T}$ be a macro tree transducer and t be a tree. Then there exists $c > 0$ such that $\mathbf{ht}(u) \leq c^{\mathbf{ht}(t)}$ for any $u \in \mathcal{T}(t)$.

Proof: Define c as the maximum height of the expressions appearing in all the rules. Then, the result follows by proving the following:

(A) $u \in [\![p^{(k)}]\!](t, \vec{w})$ implies that $\mathbf{ht}(u) \leq c^{\mathbf{ht}(t)} + \max(0, \mathbf{ht}(w_1), \ldots, \mathbf{ht}(w_k))$.

The proof proceeds by induction on t. Statement (A) immediately follows if the following statement holds:

(B) $u \in [\![e]\!](\rho, \vec{w})$ implies that $\mathbf{ht}(u) \leq \mathbf{ht}(e)c^{\mathbf{ht}(\rho)} + \max(0, \mathbf{ht}(w_1), \ldots, \mathbf{ht}(w_k))$.

Here $\mathbf{ht}(\rho)$ is defined as 0 when $\rho = \_$ and as $\mathbf{max}(\mathbf{ht}(t_1), \mathbf{ht}(t_2))$ when $\rho = (t_1, t_2)$. The condition (B) can in turn be proved by induction on the structure of e. Let $d = \mathbf{max}(0, \mathbf{ht}(w_1), \ldots, \mathbf{ht}(w_k))$.

Case 1: $e = \#$ or $e = y_j$
The condition (B) trivially holds.

Case 2: $e = a(e_1, e_2)$
Note that $u = a(u_1, u_2)$ where $u_i \in [\![e_i]\!](\rho, \vec{w})$ for $i = 1, 2$. Thus we can derive (B) as follows (I. H. means "induction hypothesis"):

$$\begin{aligned}
\mathbf{ht}(u) &= 1 + \mathbf{max}(\mathbf{ht}(u_1), \mathbf{ht}(u_2)) \\
&\leq 1 + \mathbf{max}(\mathbf{ht}(e_1)c^{\mathbf{ht}(\rho)} + d, \mathbf{ht}(e_2)c^{\mathbf{ht}(\rho)} + d) && \text{by I.H. for (B)} \\
&\leq (1 + \mathbf{max}(\mathbf{ht}(e_1), \mathbf{ht}(e_2)))c^{\mathbf{ht}(\rho)} + d \\
&= \mathbf{ht}(e)c^{\mathbf{ht}(\rho)} + d
\end{aligned}$$

Case 3: $e = p^{(l)}(x_h, e_1, \ldots, e_l)$
Note that ρ has the form (t_1, t_2) and $u \in [\![p^{(l)}]\!](t_h, \vec{w'})$, where $w'_j \in [\![e_j]\!](\rho, \vec{w})$ for $j = 1, \ldots, l$. Thus we can derive (B) as follows:

$$\begin{aligned}
\mathbf{ht}(u) &\leq c^{\mathbf{ht}(t_h)} + \max(0, \mathbf{ht}(w'_1), \ldots, \mathbf{ht}(w'_l)) && \text{by I.H. for (A)} \\
&\leq c^{\mathbf{ht}(t_h)} + \max(0, \mathbf{ht}(e_1)c^{\mathbf{ht}(\rho)} + d, \ldots, \mathbf{ht}(e_l)c^{\mathbf{ht}(\rho)} + d) \\
& && \text{by I.H. for (B)} \\
&\leq (1 + \max(0, \mathbf{ht}(e_1), \ldots, \mathbf{ht}(e_l)))c^{\mathbf{ht}(\rho)} + d \\
&= \mathbf{ht}(e)c^{\mathbf{ht}(\rho)} + d
\end{aligned}$$

□

Exercise 10.3.5 (⋆⋆⋆) Consider the following transformation. Given an XHTML tree, whenever there is a `div` element, for example,

```
div[h3[...], a[...], ... "1" ...], ... "2" ...
```

collect the list of all `img` elements appearing in the whole content of the `div`, remove the `div` itself, and add the list of `img`s to the content of the `div`:

```
img[...], img[...], img[...], h3[...], a[...],
... "1" ..., ... "2" ...
```

Using Theorem 10.3.4 prove that this transformation (working in the binary tree representation) is not expressible by a macro tree transducer. Hint: `div` elements can be nested in XHTML. ■

10.4 Bibliographic notes

A number of variations of top-down and macro tree transducers have been studied (Engelfriet, 1975, 1977; Engelfriet and Vogler, 1985). One of the simplest

extensions is to allow a *stay* rule of the form

$$p(x, y_1, \ldots, y_k) \to e$$

in which we can pass the current node itself to another procedure (without going down the tree). Another extension is to allow *regular look ahead*. In this, we assume a separate tree automaton and associate each rule with a state in the form:

$$p(a(x_1, x_2), y_1, \ldots, y_k)@q \to e$$

Each rule is fired when the input tree is accepted by the associated state. It is known that the expressive power of top-down tree transducers is increased by both extensions; that of macro tree transducers, however, increases by stay rules but not by regular look ahead. Macro tree transducers presented in this chapter take the *call-by-value* semantics, where a procedure call evaluates its (accumulating) parameters before executing the procedure itself. We can also think of the *call-by-name* semantics, where parameters are evaluated when their values are used. These evaluation strategies change the actual behavior of macro tree transducers when the transducers contain nondeterministic or partial (non-total) procedures. *Forest transducers* extend top-down tree transducers with a built-in concatenation operator, thus directly supporting XML-like unranked trees. Analogously, *macro forest transducers* are an extension of macro tree transducers with concatenation (Perst and Seidl, 2004). They actually change the expressive power by the addition of the concatenation operator; in particular, macro forest transducers have a double-exponential height property. Finally, macro tree transducers extended with of higher-order functions are called *high-level tree transducers* (Engelfriet and Vogler, 1988).

In general, where there is an acceptor model we can think of a corresponding transducer model. In this sense, top-down tree transducers correspond to top-down tree automata. Similarly, we can consider *bottom-up tree transducers* as corresponding to bottom-up tree automata (Engelfriet, 1975). In bottom-up tree transducers each node of the input tree is processed from the leaves to the root, and, at each node, the states assigned to the child nodes and the current node label are used to decide how to transform the node and which state to transit. A difference in expressiveness between top-down and bottom-up transducers is that bottom-up transducers can "process a node and then copy the result n times" but cannot "copy a node n times and then process each," whereas the opposite is the case for top-down transducers. The special abilities of bottom-up transducers seem not particularly useful in practice, however, and this is perhaps why they are relatively poorly studied.

In Chapter 12 we present tree-walking automata, which can move not only down but also up the input tree. A transducer model closely related to such automata is

k-pebble tree transducers (Milo *et al.*, 2003), which allows k pointers to nodes to be moved up and down in the input tree and are thereby capable of representing various realistic XML transformations. In Chapter 13 we explain how monadic second-order (MSO) logic is used to describe constraints among tree nodes. *MSO-definable tree transducers* lift this ability in order to relate tree nodes in the input and output (Courcelle, 1994).

Theoretical properties of these transducer models have been actively investigated. These include their expressive powers, their capability for exact typechecking, and their composability (e.g., can the composition of two top-down transducers be realized by a single top-down transducer?). Apart from exact typechecking, which will be covered in Chapter 11, further details are beyond the scope of this book; interested readers should consult the literature, e.g., Engelfriet and Vogler (1985) and Engelfriet and Maneth (2003).

11

Exact typechecking

Exact typechecking is an approach in which, rather than a general, Turing-complete language on which only an approximate analysis is possible, a restricted language is taken for which a precise analysis is decidable. In this chapter, we study exact typechecking for one of the simplest tree transformation languages, namely, the top-down tree transducers introduced in Chapter 10. Although this transformation language is so simple that many interesting transformations are not expressible, using it we can explain many technical ideas behind exact typechecking that are applicable to other formalisms.

11.1 Motivation

One may wonder why exact typechecking is important in the first place. Let us give an instructive example: an identity function written in the μXDuce language presented in Chapter 7. This is a seemingly trivial example yet it cannot be validated by the approximate typechecking given there. Let us first assume that there are only three labels, `a`, `b`, and `c` in the input. Then an identity function can be written as follows:

```
fun id(x : T) : T =
  match x with
    ()       ->  ()
  | a[y],z   ->  a[id(y)],id(z)
  | b[y],z   ->  b[id(y)],id(z)
  | c[y],z   ->  c[id(y)],id(z)
```

Here, we have not defined the type `T`. Indeed, the function should typecheck whatever `T` is defined to be (as long as `T` uses only `a`, `b`, and `c` labels), since the function translates any tree in `T` to itself and therefore the result also has type `T`.

However, whether the function goes through the μXDuce typechecker does depend on how `T` is defined. For example, if we set

```
type T = (a[T] | b[T] | c[T])*
```

then the function typechecks since, in each case of the `match` expression, both variables `y` and `z` are given the type `T` and thereby the body can be validated. For example, in the second case, pattern type inference (Section 7.3) gives both `y` and `z` the type `T = (a[T]|b[T]|c[T])*`, since it describes exactly the set of values coming inside or after `a`; therefore the body has type `a[T],T`, which is valid. However, if we instead set

```
type T = a[T]*,b[T]*,c[T]*
```

then typechecking fails. To see why, let us focus on this third case. First, pattern type inference gives the variable `z` the type `(b[T]*,c[T]*)`, since this describes what can follow after a `b` label; therefore the typechecker asserts that the argument type is valid at the function call `id(z)`. However, the typechecker gives `T` as the return type of the function call since at the outset this was declared to be the case (and similarly for the other call `id(y)`) and, as a result, the body of the present case is given the type `b[T],T` (= `b[T],a[T]*,b[T]*,c[T]*`), which is not a subtype of `T` (= `a[T]*,b[T]*,c[T]*`) since a `b` label can come before an `a` label.

Why does this false negative occur? The reason is that the μXDuce typechecker does not consider the context dependency of the function's type. That is, the function produces, from an input of type `a[T]*,b[T]*,c[T]*`, an output of type `a[T]*,b[T]*,c[T]*`, but from an input of type `b[T]*,c[T]*` it produces, an output of type `b[T]*,c[T]*`, and so on. If the typechecker used the fact that the output type can be more specific, depending on the input type, it would be able to validate the above function. However, it monolithically gives each expression a single type independently from the context and thus fails to typecheck the identity function.

In this sense the exact typechecking algorithm presented in this chapter can be seen as a context-sensitive analysis. However, as we will see, a naive attempt to extend the μXDuce-like typechecker in a context-sensitive way fails to achieve exactness. That is, the μXDuce typechecker infers an "output type" of a function body from its declared input type; such a *forward inference* cannot give a type precisely describing the set of output values since in general this set can go beyond regular tree languages. Instead, we need to consider an output type as a context and infer an input type from it; this approach is called *backward inference*.

Before going into the technicalities, it is worth remarking on the usability of the exact typechecking approach. One may argue that this approach can be too limited since, once a transformation goes slightly beyond the target language, the approach

immediately becomes inapplicable. On the contrary, however, this is exactly where nondeterminism becomes useful. That is, even when a given transformation is not exactly expressible, an "approximation" could be, if some complicated computations are abstracted as "anything may happen" by means of nondeterminism. For example, if a given transformation has an if-then-else expression whose conditional is not expressible by a transducer, we can replace it with a nondeterministic choice of the two branches. This approach gives rise to another kind of approximate typechecking that can be more precise than a more naive μXDuce-like method yet has a clear specification: "first abstraction and then exact typechecking." This is an important consideration when explaining to the user the reason for a raised error.

11.2 Forward type inference: limitation

Given a transformation $\mathcal{T}$, an input type τ_I, and an output type τ_O, in the forward inference approach the image $\tau'_O = \mathcal{T}(\tau_I)$ of τ_I resulting from the transformation $\mathcal{T}$ (we write $\mathcal{T}(\tau_I) = \bigcup_{t \in \tau_I} \mathcal{T}(t)$) is first compared and then the inclusion $\tau'_O \subseteq \tau_O$ is checked.

To see the limitation of this approach, let us first consider the following top-down tree transducer $\mathcal{T}$ with an initial procedure p_0:

$$\begin{array}{lcl} p_0(a(x_1, x_2)) & \to & a(p_1(x_1), p_1(x_1)) \\ p_1(a(x_1, x_2)) & \to & a(p_1(x_1), p_1(x_2)) \\ p_1(\#) & \to & \# \end{array}$$

The procedure p_1 simply copies the whole input tree and thus the procedure p_0 duplicates the left subtree, that is, translates any tree of the form $a(t_1, t_2)$ to $a(t_1, t_1)$. So, for the input type $\tau_I = \{a(t, \#) \mid t \text{ is any tree}\}$, we can easily see that the image is $\mathcal{T}(\tau_I) = \{a(t, t) \mid t \text{ is any tree}\}$, which is well known to go beyond regular tree languages.

One may argue that the last set is in fact a so-called context-free tree language and therefore can be checked against a given output type if the output type is regular, since this check is known to be decidable. However, we can think of a more complicated transformation whose image goes beyond even context-free tree languages. For example, consider the following macro tree transducer $\mathcal{T}'$ with an initial procedure p_0:

$$\begin{array}{lcl} p_0(a(x_1, x_2)) & \to & p_1(x_2, b(p_1(x_2, b(p_1(x_2, \#))))) \\ p_1(a(x_1, x_2), y) & \to & a(p_1(x_1, \#), p_1(x_2, y)) \\ p_1(\#, y) & \to & y \end{array}$$

First, the procedure p_1 concatenates the input tree with the tree given as a parameter, that is, it replaces the rightmost leaf of the input with the parameter tree. Then, the procedure p_0 creates three copies of the input and concatenates them together, using b labels as separators. So, in the unranked tree notation, for the input type

$$\tau'_I = \{\overbrace{a[]\cdots a[]}^{n+1} \mid n \geq 0\}$$

the image is

$$\mathcal{T}'(\tau'_I) = \{\overbrace{a[]\cdots a[]}^{n}\, b[]\, \overbrace{a[]\cdots a[]}^{n}\, b[]\, \overbrace{a[]\cdots a[]}^{n} \mid n \geq 0\}.$$

This set is not a context-free but a context-sensitive tree language, for which an inclusion check with a regular tree language is known to be undecidable.

Forward type inference is easy in the special case of top-down tree transducers, however. For example, when a top-down tree transducer is *linear*, that is, the right-hand side of each rule uses each variable at most once, then the set of output values is actually regular.

Exercise 11.2.1 (⋆⋆⋆) Construct a forward type inference algorithm for linear top-down tree transducers. ■

11.3 Backward type inference

The backward inference approach, in principle, first computes a type τ'_I representing $\{t \mid \mathcal{T}(t) \subseteq \tau_O\}$ and then checks the inclusion $\tau_I \subseteq \tau'_I$. However, to compute such a type as τ'_I directly is often rather complicated. Instead, we show below a method that computes $\overline{\tau'_I}$ (the complement of τ'_I) and tests whether $\overline{\tau'_I} \cap \tau_I = \emptyset$. The idea is that the set $\overline{\tau'_I}$ equals the preimage $\mathcal{T}^{-1}(\overline{\tau_O})$ of $\overline{\tau_O}$, that is, the set $\{t \mid \exists t' \in \overline{\tau_O}.\, t' \in \mathcal{T}(t)\}$, which is normally easier to compute. (Although we will see later that, in the deterministic case, the direct approach is equally easy.)

Let $\mathcal{T}$ be a top-down tree transducer (P, P_0, Π) and M be a tree automaton (Q, I, F, Δ) representing the type $\overline{\tau_O}$. Then, we construct an alternating tree automaton $A = (R, R_0, R_F, \Phi)$, representing the preimage $\overline{\tau'_I}$, in the following way:

$$\begin{array}{lcl} R & = & P \times Q \\ R_0 & = & P_0 \times I \\ R_F & = & \{\langle p, q\rangle \mid \vdash \bigvee_{(p(\#)\to e)\in\Pi} \mathbf{Inf}(e, q)\} \\ \Phi(\langle p, q\rangle, a) & = & \bigvee_{(p(a(x_1,x_2))\to e)\in\Pi} \mathbf{Inf}(e, q) \end{array}$$

The function **Inf** is defined inductively as follows:

$$\begin{aligned}
\mathbf{Inf}(\#, q) &= \begin{cases} \top & (q \in F) \\ \bot & (q \notin F) \end{cases} \\
\mathbf{Inf}(a(e_1, e_2), q) &= \bigvee_{(q \to a(q_1,q_2)) \in \Delta} \mathbf{Inf}(e_1, q_1) \wedge \mathbf{Inf}(e_2, q_2) \\
\mathbf{Inf}(p(x_h), q) &= \downarrow_h \langle p, q \rangle
\end{aligned}$$

For a leaf rule, since it contains no procedure call, a formula returned by **Inf** never contains the form $\downarrow_i \langle p, q \rangle$ and therefore is naturally evaluated to a Boolean; we write $\vdash \phi$ (in the definition of R_F) when a formula ϕ is evaluated to true in this sense.

Intuitively, each state $\langle p, q \rangle$ represents the set of input trees that can be translated by the procedure p to an output tree in the state q. Thus, the initial states for the inferred automaton can be obtained by collecting all pairs $\langle p_0, q_0 \rangle$ of an initial procedure p_0 and an initial state q_0. The function **Inf** takes an expression e and an "output type" q and returns a formula. If e appears in a leaf rule then the formula represents whether or not a leaf node can be translated to a tree in the state q. By using this we collect, as A's final states, $\langle p, q \rangle$ such that any such formula for p's leaf rule evaluates to true. If the expression e appears in a node rule then the formula returned by **Inf** represents the set of pairs of trees that can be translated to a tree in the state q. By using this we collect, as $\Phi(\langle p, q \rangle, a)$, the union of such formulas for all the a-node rules.

Each case for the function **Inf** can be explained as follows.

- In order for a leaf expression # to produce a tree conforming to the state q, a sufficient and necessary condition is that q is final. That is, if the state q is final then, whatever the input, the expression # will produce a tree that conforms to q (namely, a leaf tree #). Conversely, if q is not final then the output will not conform to q whatever for any input.
- In order for a label expression $a(e_1, e_2)$ to produce a tree conforming to the state q, a sufficient and necessary condition is that there exists a transition $q \to a(q_1, q_2)$ (with the same label) such that each e_i produces a tree that conforms to the corresponding state q_i. This condition is equivalent to saying that both **Inf** with e_1 and q_1 and **Inf** with e_2 and q_2 hold for some transition $q \to a(q_1, q_2)$.
- In order for a procedure call $p(x_h)$ to produce a tree conforming to the state q, a sufficient and necessary condition is that the procedure p produces such a tree from the hth subtree. This condition can be rephrased as follows: the hth subtree is in the set of trees from which the procedure p translates to a tree in the state q. The last set is represented exactly by the state $\langle p, q \rangle$.

Now, let us prove that the constructed alternating tree automaton indeed represents the preimage. In the proof, we will write $[\![q]\!]$ to denote the set of trees accepted

in a state q of an alternating or normal tree automaton. Clearly, the following holds for a normal tree automaton (Q, I, F, Δ):

$$[\![q]\!] = \{\# \mid q \in F\} \cup \{a(t_1, t_2) \mid q \to a(q_1, q_2) \in \Delta,\ t_1 \in [\![q_1]\!],\ t_2 \in [\![q_2]\!]\}$$

and, for an alternating tree automation (Q, I, F, Φ), it holds mutually with the denotation $[\![\phi]\!]$ of a formula ϕ:

$$\begin{array}{lcl}
[\![q]\!] & = & \{\# \mid q \in F\} \cup \{a(t_1, t_2) \mid (t_1, t_2) \in [\![\Phi(q, a)]\!]\} \\
[\![\phi_1 \vee \phi_2]\!] & = & [\![\phi_1]\!] \cup [\![\phi_2]\!] \\
[\![\phi_1 \wedge \phi_2]\!] & = & [\![\phi_1]\!] \cap [\![\phi_2]\!] \\
[\![\top]\!] & = & \{(t_1, t_2) \mid t_1, t_2 \text{ are any trees}\} \\
[\![\bot]\!] & = & \emptyset \\
[\![\downarrow_1 q]\!] & = & \{(t_1, t_2) \mid t_1 \in [\![q]\!],\ t_2 \text{ is any tree}\} \\
[\![\downarrow_2 q]\!] & = & \{(t_1, t_2) \mid t_1 \text{ is any tree},\ t_2 \in [\![q]\!]\}
\end{array}$$

Theorem 11.3.1 $\mathcal{L}(A) = \mathcal{T}^{-1}(\mathcal{L}(M))$.

Proof: To show the result, it suffices to prove that, for any t,

(A) $t \in [\![\langle p, q\rangle]\!]$ iff $[\![p]\!](t) \cap [\![q]\!] \neq \emptyset$.

The proof proceeds by induction on the height of t. Statement (A) follows by demonstration of the following two statements:

(B1) $(t_1, t_2) \in [\![\mathbf{Inf}(e, q)]\!]$ iff $[\![e]\!](t_1, t_2) \cap [\![q]\!] \neq \emptyset$ for any t_1, t_2

for an expression e appearing in a node rule and

(B2) $\vdash \mathbf{Inf}(e, q)$ iff $[\![e]\!]_- \cap [\![q]\!] \neq \emptyset$

for an expression e appearing in a leaf rule. We show only (B1) since (B2) is similar. The proof proceeds by induction on the structure of e.

Case 1: $e = \#$
We have

$$\begin{array}{lcl}
(t_1, t_2) \in [\![\mathbf{Inf}(e, q)]\!] & \iff & q \in F \\
& \iff & [\![e]\!](t_1, t_2) \cap [\![q]\!] \neq \emptyset.
\end{array}$$

Case 2: $e = a(e_1, e_2)$
We have

$$
\begin{array}{lll}
& (t_1, t_2) \in [\![\mathbf{Inf}(e, q)]\!] & \\
\Longleftrightarrow & \exists (q \to a(q_1, q_2)) \in \Delta\,.\, (t_1, t_2) \in [\![\mathbf{Inf}(e_1, q_1)]\!] \ \wedge\ (t_1, t_2) \in [\![\mathbf{Inf}(e_2, q_2)]\!] & \\
\Longleftrightarrow & \exists (q \to a(q_1, q_2)) \in \Delta\,.\, [\![e_1]\!](t_1, t_2) \cap [\![q_1]\!] \neq \emptyset \ \wedge\ [\![e_2]\!](t_1, t_2) \cap [\![q_2]\!] \neq \emptyset & \\
& & \text{by I.H. for (B1)} \\
\Longleftrightarrow & [\![e]\!](t_1, t_2) \cap [\![q]\!] \neq \emptyset. &
\end{array}
$$

Case 3: $e = p(x_h)$
We have

$$
\begin{array}{lll}
(t_1, t_2) \in [\![\mathbf{Inf}(e, q)]\!] & \Longleftrightarrow & t_h \in [\![\langle p, q \rangle]\!] \\
& \Longleftrightarrow & [\![p]\!](t_h) \cap [\![q]\!] \neq \emptyset \qquad \text{by I.H. for (A)} \\
& \Longleftrightarrow & [\![e]\!](t_1, t_2) \cap [\![q]\!] \neq \emptyset. \qquad \square
\end{array}
$$

Example 11.3.2 Consider the top-down tree transducer $\mathcal{T}_{11.3.2} = (\{p_0\}, \{p_0\}, \Pi)$, where Π consists of the following:

$$
\begin{array}{lcl}
p_0(a(x_1, x_2)) & \to & a(p_0(x_1), p_0(x_2)) \\
p_0(b(x_1, x_2)) & \to & a(p_0(x_1), p_0(x_2)) \\
p_0(b(x_1, x_2)) & \to & b(p_0(x_1), p_0(x_2)) \\
p_0(\#) & \to & \#
\end{array}
$$

This transducer produces a tree with a and b labels from an input tree with a and b labels. However, from a tree with all a labels it produces only a tree with all a labels. Let us check this.

Let us represent both the input and the output types by the tree automaton $(\{q\}, \{q\}, \{q\}, \{q \to a(q, q)\})$ accepting the set of trees with all a labels. First, we take the complement of the output tree automaton. Assume that $\Sigma = \{a, b\}$. Then, we obtain the tree automaton $(\{q_0, q_1\}, \{q_0\}, \{q_1\}, \Delta)$, where Δ consists of the following:

$$
\begin{array}{lcl}
q_0 & \to & a(q_0, q_1) \\
q_0 & \to & a(q_1, q_0) \\
q_0 & \to & b(q_1, q_1) \\
q_1 & \to & a(q_1, q_1) \\
q_1 & \to & b(q_1, q_1)
\end{array}
$$

Intuitively, q_0 accepts a tree with at least one b node and q_1 accepts any tree with a and b labels. Then,we compute an alternating tree automaton representing the

preimage of this complemented output tree automaton with respect to the transducer $\mathcal{T}_{11.3.2}$, namely, (R, R_0, R_F, Φ) where

$$\begin{array}{lcl} R & = & \{\langle p_0, q_0\rangle, \langle p_0, q_1\rangle\} \\ R_0 & = & \{\langle p_0, q_0\rangle\} \\ R_F & = & \{\langle p_0, q_1\rangle\} \end{array}$$

and Φ is defined by the following:

$$\begin{array}{lcl} \Phi(\langle p_0, q_0\rangle, a) & = & \downarrow_1 \langle p_0, q_0\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \ \vee \ \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_0\rangle \\ \Phi(\langle p_0, q_0\rangle, b) & = & \downarrow_1 \langle p_0, q_0\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \ \vee \ \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_0\rangle \\ & & \vee \ \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \\ \Phi(\langle p_0, q_1\rangle, a) & = & \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \\ \Phi(\langle p_0, q_1\rangle, b) & = & \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \ \vee \ \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \end{array}$$

Using the conversion algorithm given in Section 9.2.1, this alternating tree automaton can be translated to the nondeterministic tree automaton $(\{r_0, r_1\}, \{r_0\}, \{r_1\}, \Delta')$, where $r_0 = \{\langle p_0, q_0\rangle\}$, $r_1 = \{\langle p_0, q_1\rangle\}$, and Δ' consists of the following:

$$\begin{array}{lcl} r_0 & \to & a(r_0, r_1) \\ r_0 & \to & a(r_1, r_0) \\ r_0 & \to & b(r_0, r_1) \\ r_0 & \to & b(r_1, r_0) \\ r_0 & \to & b(r_1, r_1) \\ r_1 & \to & a(r_1, r_1) \\ r_1 & \to & b(r_1, r_1) \end{array}$$

This automaton is disjoint with the input automaton since it accepts no tree with all a labels. Indeed, taking the product of the input automaton and the inferred automaton we obtain $(\{(q, r_0), (q, r_1)\}, \{(q, r_0)\}, \{(q, r_1)\}, \Delta'')$, where Δ'' consists of

$$\begin{array}{lcl} (q, r_0) & \to & a((q, r_0), (q, r_1)) \\ (q, r_0) & \to & a((q, r_1), (q, r_0)) \\ (q, r_1) & \to & a((q, r_1), (q, r_1)) \end{array}$$

which accepts no tree (note that (q, r_1) accepts at least one tree but (q, r_0) does not). ■

Notice that, when a transformation $\mathcal{T}$ yields at most one result from any input (in particular when it is a deterministic top-down tree transducer), the set $\tau_I' = \{t \mid \mathcal{T}(t) \subseteq \tau_O\}$ coincides with the preimage of the output type τ_O, that is, $\mathcal{T}^{-1}(\tau_O) = \{t \mid \exists t' \in \tau_O.\, t' \in \mathcal{T}(t)\}$. Therefore, by using exactly the same inference algorithm as above, we can construct an automaton representing τ_I' from another representing τ_O. Then, typechecking can be done by testing the relation $\tau_I \subseteq \tau_I'$.

Proposition 11.3.3 The typechecking problem for top-down tree transducers is EXPTIME-complete both in the deterministic and the nondeterministic cases.

Proof: The inference algorithm for the nondeterministic case runs in exponential time since computing the complement $\overline{\tau_O}$ takes exponential time, computing the alternating tree automaton for the preimage $\overline{\tau'_I}$ takes polynomial time, and the test $\overline{\tau'_I} \cap \tau_I = \emptyset$ takes polynomial time. The problem is EXPTIME-hard even in the deterministic case. Indeed, by Theorem 4.4.5, the inclusion check for tree automata is EXPTIME-complete, and this check trivially reduces to typechecking by an identity tree transformation, which can be expressed by a deterministic top-down tree transducer. □

Exercise 11.3.4 (⋆⋆) Using the transducer $\mathcal{T}_{11.3.2}$, confirm that the above inference method for the deterministic case does not work in the nondeterministic case. ■

Example 11.3.5 Let us typecheck the top-down tree transducer in Section 11.2, which was difficult using forward type inference. The input type is a tree automaton representing $\tau_I = \{a(t, \#) \mid t \text{ is any tree with all } a \text{ labels}\}$ and the output type is the tree automaton $(\{q\}, \{q\}, \{q\}, \{q \to a(q, q)\})$ accepting any tree with all a labels. From the tree transducer and the output automaton we construct the alternating tree automaton (R, R_0, R_F, Φ), where

$$\begin{array}{lcl} R & = & \{\langle p_0, q\rangle, \langle p_1, q\rangle\} \\ R_0 & = & \{\langle p_0, q\rangle\} \\ R_F & = & \{\langle p_1, q\rangle\} \end{array}$$

and Φ is defined by the following:

$$\begin{array}{lcl} \Phi(\langle p_0, q\rangle, a) & = & \downarrow_1 \langle p_1, q\rangle \wedge \downarrow_1 \langle p_1, q\rangle \\ \Phi(\langle p_1, q\rangle, a) & = & \downarrow_1 \langle p_1, q\rangle \wedge \downarrow_2 \langle p_1, q\rangle \end{array}$$

This alternating tree automaton can be translated to $M = (Q, I, F, \Delta)$, where

$$\begin{array}{lcl} Q & = & \{\emptyset, \{\langle p_0, q\rangle\}, \{\langle p_1, q\rangle\}\} \\ I & = & \{\{\langle p_0, q\rangle\}\} \\ F & = & \{\emptyset, \{\langle p_1, q\rangle\}\} \end{array}$$

and the transitions Δ consist of the following:

$$\begin{array}{lcl} \emptyset & \to & a(\emptyset, \emptyset) \\ \{\langle p_0, q\rangle\} & \to & a(\{\langle p_1, q\rangle\}, \emptyset) \\ \{\langle p_1, q\rangle\} & \to & a(\{\langle p_1, q\rangle\}, \{\langle p_1, q\rangle\}) \end{array}$$

This automaton in fact represents the set of all non-leaf trees with only a labels. We can thus easily see that $\tau_I \subseteq \mathcal{L}(M)$. ■

Exercise 11.3.6 (⋆⋆) By using Theorem 11.3.1, prove that a top-down tree transducer cannot express the transformation that returns `true` if the given tree has identical subtrees and `false` otherwise. ■

11.4 Bibliographic notes

Exact typechecking techniques have been investigated for a long time on various tree transducer models. For example, even very early work treated top-down and bottom-up tree transducers (Engelfriet, 1977); later work has dealt with macro tree transducers (Engelfriet and Vogler, 1985) and macro forest transducers (Perst and Seidl, 2004). With regard to XML, exact typechecking started with forward-inference-based techniques, which were used to treat relatively simple transformation languages (Murata, 1997; Milo and Suciu, 1999; Papakonstantinou and Vianu, 2000). A backward inference technique was then employed in XML typechecking, where k-pebble tree transducers were used as an XML transformation model (Milo *et al.*, 2003). Exact typechecking was proposed for a subset of XSLT based on alternating tree automata (Tozawa, 2001); in this chapter his style of formalization has been adopted. Also, a typechecking method was described for a non-trivial tree transformation language *TL* by decomposition to macro tree transducers (Maneth *et al.*, 2005). As macro tree transducers appear to offer a promising model that is simple yet expressive, some efforts have been made towards practically usable typechecking (Maneth *et al.*, 2007; Frisch and Hosoya, 2007). A typechecking technique treating equality tests on leaf values can be found in Alon *et al.* (2003) and one treating high-level tree transducers in Tozawa (2006).

12

Path expressions and tree-walking automata

While in Chapter 5 we defined patterns as an approach to subtree extraction, in this chapter we introduce an alternative approach called path expressions. While patterns use structural constraints for matching subtrees, path expressions use "navigation," which specifies a sequence of movements on the tree and makes checks on the visited nodes. In practice, path expressions tend to be more concise since they do not require conditions to be specified for the nodes not on the path. In theory, however, path expressions are strictly less expressive (i.e., they cannot express all regular tree languages) unless sufficient extensions are made. In this chapter we first review path expressions used in actual XML processing and then consider a refinement called caterpillar expressions. After this, we study the corresponding automata formalism, called tree-walking automata, and compare their expressiveness with tree automata.

12.1 Path expressions

Although there are various styles of path expression, the basic idea is to find a node to which there is a "path" from the "current" node obeying the given path expression. When multiple such nodes are found, usually all are returned.

The main differences between the styles lie in which paths are allowed. A path is, roughly, a sequence consisting of either movements between or tests on nodes. In each movement we specify in which direction to go; we call this *axis*. For example, some styles allow only movement to descendant nodes – called *forward axis* movement – while others allow also movement to ancestor nodes – called *backward axis* movement. There are also axes for moving to the following or preceding siblings, and so on.

Tests on nodes can be local or global. Locally, there are tests for whether a node has a given label, whether it is the root, and so forth. A typical global test

is itself a path expression – for example, a test for whether a specified node exists with respect to the present node. Path expressions containing themselves as node tests are often called *conjunctive path expressions*. In addition to these, various styles of path expression differ in which path languages they allow. Simple styles allow only a single path to be specified whereas others allow regular languages of paths.

12.1.1 XPath

XPath is a framework of path expressions standardized by W3C and nowadays widely adopted in various systems. Let us see below some examples in XPath, where we take the family tree in Figure 3.1 as the input and some `person` node as the current node:

`children/person`	Find a `person` subnode of a `children` subnode (of the current node)
`.//person/name`	Find a `name` subnode of a `person` descendant
`../../spouse`	Find a `spouse` subnode of the grandparent node

Here, a label such as `children` denotes a test for whether the current node has that label. The axes for moving to a child, an ancestor, and a parent are written by a slash (`/`), a double slash (`//`), and a double dot (`..`), respectively. "Not moving" (or staying) is also an axis and is indicated by a single dot (`.`). Slightly verbose notation is used for going to siblings:

`following-sibling::person`	Find a right sibling with a `person` label
`preceding-sibling::person`	Find a left sibling with a `person` label

Here are some uses of path expressions, written inside square brackets, as node tests:

`children/person[gender/male]`
: Find a `children` subnode's `person` subnode that has a `gender` subnode with a `male` subnode

`children/person[gender/male and spouse]`
: similar to the previous except that a `spouse` node is required as an additional subnode of `person`

Note that the last example above contains an `and` in the test; other logical connectives such as `or` and `not` can also be used.

Regular expression paths An XPath expression only specifies a single path. However, it is easy to extend it so as to specify a set of paths, by means of regular

expressions. Such path expressions are usually called *regular expression paths*. An example is:

`(children/person)*`	Find a node to which there is a path from the current node in which the labels `children` and `person` appear alternately

12.1.2 Caterpillar expressions

This section presents a formalism of regular expression paths called caterpillar expressions. It was first proposed as an alternative to XPath and has a direct relationship to tree-walking automata, thus serving as a basis for theoretical analysis on the expressiveness and algorithmics of path expressions. While caterpillar expressions were proposed originally for unranked trees, for the sake of smooth transition to the automata framework we present a variant that walks on binary trees.

Let $\Sigma_\#$ be $\Sigma \cup \{\#\}$, ranged over by $a^\#$. A *caterpillar expression* e is a regular expression over *caterpillar atoms* c defined as follows:

c	::=	up	move up
		1	move to the first child
		2	move to the second child
		$a^\#$	"is its label $a^\#$?"
		isRoot	"is it the root?"
		is1	"is it a first child?"
		is2	"is it a second child?"

The first three are movements while the rest are node tests. We call a sequence of caterpillar atoms a *caterpillar path*.

Let a (binary) tree t be given. A sequence $\pi_1 \cdots \pi_n$ of nodes each from $\mathbf{nodes}(t)$ belongs to a caterpillar path $c_1 \cdots c_n$ when the following hold for all $i = 1, \ldots, n-1$:

- if $c_i = \text{up}$ then $\pi_{i+1} j = \pi_i$ for some $j = 1, 2$;
- if $c_i = 1$ then $\pi_{i+1} = \pi_i 1$;
- if $c_i = 2$ then $\pi_{i+1} = \pi_i 2$;
- if $c_i = a^\#$ then $\pi_{i+1} = \pi_i$ and $\mathbf{label}_t(\pi_i) = a^\#$;
- if $c_i = \text{isRoot}$ then $\pi_{i+1} = \pi_i = \epsilon$;
- if $c_i = \text{is1}$ then $\pi_{i+1} = \pi_i = \pi 1$ for some π;
- if $c_i = \text{is2}$ then $\pi_{i+1} = \pi_i = \pi 2$ for some π.

Note that, since a node is a leaf if and only if its label is #, we already have a test of whether the current node is a leaf.[1] A sequence of nodes belongs to a caterpillar expression e if the sequence belongs to a caterpillar path generated by e. A node π matches a caterpillar expression e if there is a sequence $\pi_1 \cdots \pi_n$ of nodes belonging to e, where $\pi_1 = \epsilon$ and $\pi_n = \pi$.

Some XPath examples seen in the last subsection can be written using caterpillar expressions. However, since the previous examples were for unranked trees, we need to adjust the up and the down axes as well as the sibling axes. Let us define the following:

$$\begin{array}{lcl} \text{Xup} & \equiv & (\text{is2 up})^* \text{ is1 up} \\ \text{Xdown} & \equiv & 1\ 2^* \\ \text{Xright} & \equiv & 2^+ \\ \text{Xleft} & \equiv & (\text{is2 up})^+ \end{array}$$

Intuitively, Xup keeps going up, in a binary tree, as long as the visited node is a second child and then goes up once more if the node is a first child. The expression Xdown reverses these movements: it first goes down once to the first child and then repeatedly goes down visiting each second child. Note that Xup goes to the unique parent node in the unranked tree while Xdown can go to any child nondeterministically; similarly for Xleft and Xright. Using these expressions we can represent some previous XPath expressions in the following way:

`../../spouse`	$\Rightarrow$	Xup Xup Xdown `spouse`
`(children/person)`*	$\Rightarrow$	(Xdown `children` Xdown `person`)*
`following-sibling::person`	$\Rightarrow$	Xright `person`
`preceding-sibling::person`	$\Rightarrow$	Xleft `person`

Note that there is no obvious way of encoding conjunctive path expressions. Indeed, a theoretical result tells that this is fundamentally impossible (Section 12.2.3).

We conclude this section with an amusing example that starts from the root, traverses *all* the nodes, and then returns to the root:

$$(1^* \ \#\ (\text{is2 up})^* \text{ is1 up } 2)^* \ \#\ (\text{is2 up})^*$$

That is, we first go all the way down by following each first child until we reach a leaf. From there, we go left-up repeatedly (this actually happens only when we did not go down in the last step) and then go right-up once and right-down once. We iterate these until we reach the rightmost leaf and finally go straight back to the root.

[1] In the original proposal, since a leaf can have an arbitrary label, this explicitly supports the leaf test.

Exercise 12.1.1 (⋆) Give the caterpillar path generated by the above caterpillar expression on the following tree, starting from and ending at the root:

$$a(a(\#, \#), a(\#, \#))$$ ■

12.2 Tree-walking automata

Tree-walking automata provide a finite-state model directly corresponding to caterpillar expressions. They move up and down on a binary tree, changing states on the basis of the current node kind (see below) and the current state.

12.2.1 Definitions

Let a set K of node kinds be $\Sigma_\# \times \{\text{isRoot}, \text{is1}, \text{is2}\}$ and let it be ranged over by k. A *tree-walking automaton* is a quadruple $A = (Q, I, F, \delta)$, where Q and I are defined as before,

- F is a set of pairs from $Q \times K$, and
- δ is a set of transition rules of the form

$$q_1 \xrightarrow{k,d} q_2$$

where $q_1, q_2 \in Q$, $k \in K$, and $d \in \{\text{up}, 1, 2\}$.

For a binary tree t, we define the *kind* $\kappa_t(\pi)$ of a node π by $(\mathbf{label}_t(\pi), h(\pi))$, where

$$h(\pi) = \begin{cases} \text{isRoot} & \text{if } \pi = \epsilon \\ \text{is1} & \text{if } \pi = \pi'1 \text{ for some } \pi' \\ \text{is2} & \text{if } \pi = \pi'2 \text{ for some } \pi'. \end{cases}$$

The semantics of a tree-walking automaton (Q, I, F, δ) is given as follows. A sequence $(\pi_1, q_1) \cdots (\pi_n, q_n)$ is a *run* of A on a binary tree t if $\pi_i \in \mathbf{nodes}(t)$ for each $i = 1, \ldots, n$ and

- $q_i \xrightarrow{\kappa_t(\pi_i),\text{up}} q_{i+1} \in \delta$ with $\pi_{i+1} j = \pi_i$ for some $j = 1, 2$,
- $q_i \xrightarrow{\kappa_t(\pi_i),1} q_{i+1} \in \delta$ with $\pi_{i+1} = \pi_i 1$, or
- $q_i \xrightarrow{\kappa_t(\pi_i),2} q_{i+1} \in \delta$ with $\pi_{i+1} = \pi_i 2$.

for each $i = 1, \ldots, n-1$. That is, the automaton starts from node π_1 in state q_1 and finishes at node π_n in state q_n; each step makes a movement according to a transition rule that matches the node kind. A run $(\pi_1, q_1) \cdots (\pi_n, q_n)$ is *successful* when $\pi_1 = \epsilon$, $q_1 \in I$, and $(q_n, \kappa_t(\pi_n)) \in F$. In this case the node π_n is *matched* by the automaton. When π_n is the root we say that the whole tree t is *accepted* by the automaton. We define the language $\mathcal{L}(A)$ of A to be the set of trees accepted by A. Let **TWA** be the class of languages accepted by tree-walking automata.

It should be intuitively clear that caterpillar expressions and tree-walking automata are equivalent notions. The only part that needs care is that a test of node kind done on a transition or at a final state in a tree-walking automaton corresponds to a consecutive sequence of tests on the same node in a caterpillar expression.

Exercise 12.2.1 (⋆⋆) Write down conversions between caterpillar expressions and tree-walking automata. ■

Exercise 12.2.2 (⋆) Convert the "traverse-all" caterpillar expression given in Section 12.1.2 to a tree-walking automaton. Then give a successful run on the tree $a(a(\#, \#), a(\#, \#))$. ■

12.2.2 Expressiveness

Since we have been considering path expressions as an alternative to patterns, the question naturally arises of the relationship between the expressivenesses of tree-walking automata and tree automata. In short, the former are strictly less expressive. This can be proved by (1) showing that **TWA** $\subseteq$ **ND** and (2) finding a counterexample for **ND** $\subseteq$ **TWA**. The first part is easier and covered in this section. The second part, however, is much more difficult and the details cannot be presented in this book; in fact, it was a longstanding problem for several decades and was only proved recently.

To prove that **TWA** $\subseteq$ **ND**, it suffices to give a construction of a tree automaton from a tree-walking automaton. It is possible to do so directly, but instead we give a construction of an alternating tree automaton, as defined in Chapter 9; we already know how to construct a normal tree automaton from an alternating tree automation.

Before starting a general discussion, let us first look at an example to gain an intuitive understanding. Figure 12.1 depicts a tree and a series of movements on it by a tree-walking automaton, with the state transitions indicated. That is, the automaton starts from the root π_0 in the initial state q_0 and moves to the left child π_1 in state q_1. Then it walks around the nodes below π_1 with some state transitions, after which it goes back to π_1 in state q_2 and then to the root π_0 in state q_3. Then, the automaton moves to the right child π_2, walks around the nodes below it through states q_4 and q_5, and returns to the root π_0 in state q_6. Finally, it moves again to the left child π_1 in state q_7 and then immediately goes back to the root π_0 in the final state state q_8.

To analyze such complex state transitions of tree-walking automata, we introduce a notion of a *tour*. A tour of a tree-walking automaton from a node π is a run starting from π, walking around the nodes at or below π, and ending at π.

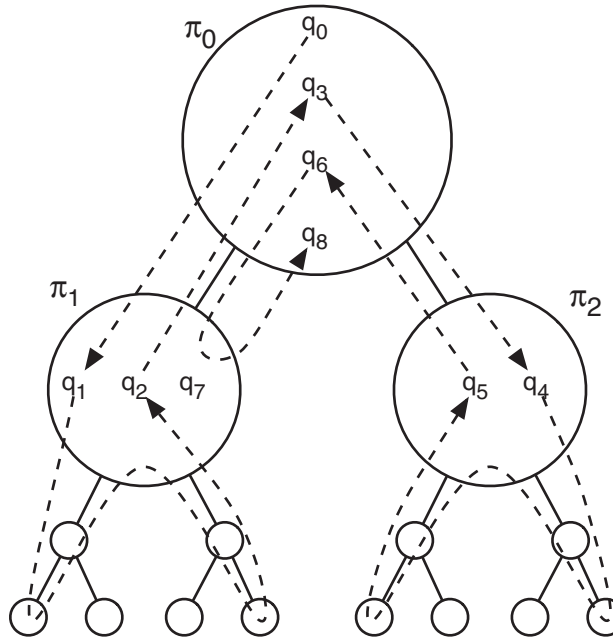


Figure 12.1 A tour made by a tree-walking automaton.

More formally, a tour from π is a run of the form

$$(\pi_1, q_1) \cdots (\pi_n, q_n)$$

where $\pi_1 = \pi_n = \pi$ and, for each $i = 2, \ldots, n-1$, we have $\pi \leq \pi_i$, that is, π_i is a self or descendant of π. (Note that n must be an odd number from the definition of a run.) In our example, from the root π_0 we have the tour

$$(\pi_0, q_0)(\pi_1, q_1) \cdots (\pi_1, q_2)(\pi_0, q_3)(\pi_2, q_4) \cdots (\pi_2, q_5)(\pi_0, q_6)(\pi_1, q_7)(\pi_0, q_8)$$

and the tours from the left child π_1

$$(\pi_1, q_1) \cdots (\pi_1, q_2) \qquad \text{and} \qquad (\pi_1, q_7).$$

Note that in the above the tour from the root forms the entire run on the tree since it starts in an initial state and ends in a final state. Also, a tour can be a singleton run, that is, a run of length 1, like the second tour from π_1 in the above; we call such a tour *trivial*. Such a tour happens when the automaton does not walk around the subtrees at all. In particular, a tour from a leaf node is always trivial.

Now we will construct an alternating tree automaton from this tree-walking automaton, in the following way. We generate each state of the alternating tree automaton as a pair of states of the tree-walking automaton. In our example, one of the generated states is the pair $\langle q_0, q_8 \rangle$. This state intuitively means that, from

each node π accepted by this state, there is a tour starting in q_0 and ending in q_8:

$$(\pi, q_0) \cdots (\pi, q_8)$$

We can also generate other states, including $\langle q_1, q_2 \rangle$ and $\langle q_7, q_7 \rangle$, whose intuitive meanings can be understood similarly. Next we need to form transitions between such states. The key observation is that a tour consists of several *simple* tours. Here, a simple tour is a tour $(\pi_1, q_1) \cdots (\pi_n, q_n)$ such that $\pi_i \neq \pi_1$ for any $i = 2, \ldots, n-1$. More specifically, a simple tour is in one of two groups: it is either

1. a run that makes first a move to the left child, then a subtour from there, and finally a move up to the original node, or
2. a run that makes first a move to the right child, then a subtour from there, and finally a move up to the original node.

In our example, the entire tour consists of the simple tours

$$(\pi_0, q_0)(\pi_1, q_1) \cdots (\pi_1, q_2)(\pi_0, q_3)$$

$$(\pi_0, q_3)(\pi_2, q_4) \cdots (\pi_2, q_5)(\pi_0, q_6)$$

$$(\pi_0, q_6)(\pi_1, q_7)(\pi_0, q_8)$$

where the top and the bottom simple tours fall into the first group and the middle simple tour falls into the second group. Thus, in order to represent the simple tours in group 1 we generate a transition formula that constrains the left subtree with the "destination" state corresponding to the subtour in the middle of the simple tour. In our example, for the simple tour from q_0 to q_3, since the tree-walking automaton should have transitions

$$q_0 \xrightarrow{k,1} q_1 \qquad q_2 \xrightarrow{k',\mathrm{up}} q_3$$

for some node kinds k, k' (where k' has the form $(a^{\#}, \mathrm{is1})$), we generate the formula

$$\downarrow_1 \langle q_1, q_2 \rangle.$$

Likewise, in order to represent simple tours in group 2, we generate a similar formula (but with $\downarrow_2$) constraining the right subtree. For a general tour, since it is composed of several simple tours we create a conjunction of all formulas representing these simple tours. In our example, we generate

$$\downarrow_1 \langle q_1, q_2 \rangle \wedge \downarrow_2 \langle q_4, q_5 \rangle \wedge \downarrow_1 \langle q_7, q_7 \rangle.$$

In general, since there can be many combinations of such simple tours, we combine formulas for them by disjunction. Care is needed here, however, since a tour could visit the same node an arbitrary number of times by looping among the same states, and therefore the number of possible combinations could be infinite. Fortunately,

we can ensure that such loops can safely be skipped and thereby only finite combinations of simple tours are possible. Formally, a tour $(\pi_1, q_1) \cdots (\pi_n, q_n)$ is *reduced* if $(\pi_i, q_i) \neq (\pi_j, q_j)$ for $i \neq j$. Then, the following can be shown.

Lemma 12.2.3 If a tree-walking automaton A has a tour $(\pi, q_1) \cdots (\pi, q_n)$ on a tree t then it has a reduced tour $(\pi, q'_1) \cdots (\pi, q'_m)$ on t such that $q_1 = q'_1$ and $q_n = q'_m$.

Proof: We can easily see that, when a given tour contains two equal pairs (π_i, q_i) and (π_j, q_j), with $i < j$, the sequence after removing the part $(\pi_i, q_i) \cdots (\pi_{j-1}, q_{j-1})$ from the original tour is also a tour preserving the beginning and the ending states. Therefore, by performing this operation repeatedly we can obtain a reduced tour. □

A final complication is that, compared with tree-walking automata, transitions of an alternating tree automaton can perform a limited form of node test and therefore we need a little care in order not to lose information during the construction. For example, from the set comprising pairs of tree-walking-automaton transitions such as

$$q_0 \xrightarrow{k,1} q_1 \qquad q_2 \xrightarrow{k',\mathrm{up}} q_3$$

how can we preserve, in an alternating tree automaton, the information that the kind of the source node must match k and that of the left destination node must match k'? (Note that the source node kind contains not only the label but also the root-or-not flag, which an alternating tree automaton cannot examine.) We will solve this problem by using a common technique in which each state of the alternating tree automaton is augmented with a node kind.

Theorem 12.2.4 **TWA** $\subseteq$ **ND**.

Proof: Given a tree-walking automaton $A = (Q, I, F, \delta)$, construct an alternating tree automaton (R, R_I, R_F, Φ), where

$$\begin{aligned}
R &= K \times Q \times Q \\
R_I &= \{\langle k, q, q' \rangle \mid k \in \Sigma_\# \times \{\mathrm{isRoot}\},\ q \in I,\ (q', k) \in F\} \\
R_F &= \{\langle k, q, q \rangle \mid k \in \{\#\} \times \{\mathrm{isRoot}, \mathrm{is1}, \mathrm{is2}\},\ q \in Q\} \\
\Phi(\langle k, q, q' \rangle, a) &= \begin{cases} \displaystyle\bigvee_{\substack{q_1, \ldots, q_n \in \mathbf{red}(Q^*) \\ \text{s.t. } q = q_1,\, q' = q_n}} \Phi^s(\langle k, q_1, q_2 \rangle) \wedge \cdots \wedge \Phi^s(\langle k, q_{n-1}, q_n \rangle) \\ \qquad\qquad \text{if } k \in \{a\} \times \{\mathrm{isRoot}, \mathrm{is1}, \mathrm{is2}\} \\ \qquad\qquad \text{otherwise} \end{cases}
\end{aligned}$$

where

$$\Phi^s(\langle k, q, q'\rangle) = \bigvee \left\{ \downarrow_1 \langle k_1, q_1, q_1'\rangle \,\middle|\, q \xrightarrow{k,1} q_1,\ q_1' \xrightarrow{k_1,\mathrm{up}} q' \in \delta,\ k_1 \in \Sigma_\# \times \{\mathrm{is1}\} \right\}$$
$$\vee \bigvee \left\{ \downarrow_2 \langle k_2, q_2, q_2'\rangle) \,\middle|\, q \xrightarrow{k,2} q_2,\ q_2' \xrightarrow{k_2,\mathrm{up}} q' \in \delta,\ k_2 \in \Sigma_\# \times \{\mathrm{is2}\} \right\}.$$

Here, we define $\mathbf{red}(L) = \{q_1, \ldots, q_n \in L \mid q_i \neq q_j,\ \forall i \neq j\}$.

Let us first show that $\mathcal{L}(A) \subseteq \mathcal{L}(B)$. To prove this, it suffices to show that

> if there is a tour $(\pi, q)\cdots(\pi, q')$ of A on t then B accepts $\mathbf{subtree}_t(\pi)$ in $\langle \kappa_t(\pi), q, q'\rangle$.

By Lemma 12.2.3 we can assume, without loss of generality, that the tour in the assumption is reduced. The proof proceeds by induction on the height of $\mathbf{subtree}_t(\pi)$.

Case 1: $\pi \in \mathbf{leaves}(t)$
Since the tour in the assumption must be trivial, $q = q'$. The result follows since $\langle \kappa_t(\pi), q, q\rangle \in R_F$.

Case 2: $\mathbf{label}_t(\pi) = a$
The tour in the assumption must have the form

$$(\pi, q_1)\cdots(\pi, q_2)\cdots(\pi, q_{n-1})\cdots(\pi, q_n)$$

for some $q_1, \ldots, q_n \in Q$, with $q_1 = q$, $q_n = q'$, $n \geq 1$, such that each $(\pi, q_i)\cdots(\pi, q_{i+1})$ is a simple tour $(i = 1, \ldots, n-1)$. For each of these simple tours there are two possibilities.

- The ith simple tour has the form $(\pi, q_i)(\pi 1, q_i')\cdots(\pi 1, q_i'')(\pi, q_{i+1})$, for some $q_i', q_i'' \in Q$, where the part $(\pi 1, q_i')\cdots(\pi 1, q_i'')$ is a tour. In this case δ contains transitions $q_i \xrightarrow{\kappa_t(\pi),1} q_i'$ and $q_i'' \xrightarrow{\kappa_t(\pi 1),\mathrm{up}} q_{i+1}$. Also, by the induction hypothesis, B accepts $\mathbf{subtree}_t(\pi 1)$ in state $\langle \kappa_t(\pi 1), q_i', q_i''\rangle$. From these we have

$$\mathbf{subtree}_t(\pi 1), \mathbf{subtree}_t(\pi 2) \vdash \Phi^s(\langle \kappa_t(\pi), q_i, q_{i+1}\rangle).$$

- The ith simple tour has the form $(\pi, q_i)(\pi 2, q_i')\cdots(\pi 2, q_i'')(\pi, q_{i+1})$, for some $q_i', q_i'' \in Q$, where the part $(\pi 2, q_i')\cdots(\pi 2, q_i'')$ is a tour. By a similar argument to that above, we obtain

$$\mathbf{subtree}_t(\pi 1), \mathbf{subtree}_t(\pi 2) \vdash \Phi^s(\langle \kappa_t(\pi), q_i, q_{i+1}\rangle).$$

Since the above holds for all the simple tours, noting that $q_1, \ldots, q_n \in \mathbf{red}(Q^*)$ from the reducedness of the given tour we conclude that

$$\mathbf{subtree}_t(\pi 1), \mathbf{subtree}_t(\pi 2) \vdash \Phi(\langle \kappa_t(\pi), q_i, q_{i+1}\rangle, a)$$

from which the result follows.

Let us next show that $\mathcal{L}(B) \subseteq \mathcal{L}(A)$. To prove this, it suffices to show that

> if B accepts $\mathbf{subtree}_t(\pi)$ in $\langle k, q, q' \rangle$ with $k = (a^{\#}, h(\pi))$ then there is a tour $(\pi, q) \cdots (\pi, q')$ of A on t with $a^{\#} = \mathbf{label}_t(\pi)$.

Again the proof proceeds by induction on the height of $\mathbf{subtree}_t(\pi)$.

Case 1: $\pi \in \mathbf{leaves}(t)$
Since $\langle k, q, q' \rangle \in R_F$ and $k = (a^{\#}, h(\pi))$ from the assumption, $a^{\#} = \# = \mathbf{label}_t(\pi)$ and $q = q'$. Therefore the trivial tour (π, q) exists.

Case 2: $\mathbf{label}_t(\pi) = a$
By the assumption,

$$\mathbf{subtree}_t(\pi 1), \mathbf{subtree}_t(\pi 2) \vdash \Phi(\langle k, q, q' \rangle, a)$$

where $k = (a^{\#}, h(\pi))$. First, we have $a^{\#} = a = \mathbf{label}_t(\pi)$ since otherwise $\Phi(\langle k, q, q' \rangle, a) = \bot$ and the above relation would not hold. Then, for some $q_1, \ldots, q_n \in \mathbf{red}(Q^*)$ with $q = q_1$ and $q' = q_n$,

$$\mathbf{subtree}_t(\pi 1), \mathbf{subtree}_t(\pi 2) \vdash \Phi^s(\langle k, q_i, q_{i+1} \rangle)$$

for each $i = 1, \ldots, n-1$. Further, for each $i = 1, \ldots, n-1$ we have two cases.

- $\mathbf{subtree}_t(\pi 1), \mathbf{subtree}_t(\pi 2) \vdash\downarrow_1 \langle k_i, q_i', q_i'' \rangle$, where δ contains the transitions $q_i \xrightarrow{k,1} q_i'$ and $q_i'' \xrightarrow{k_i,\mathrm{up}} q_{i+1}$ with $k_i \in \Sigma_{\#} \times \{\mathrm{is1}\}$. Therefore, noting that $h(\pi 1) = \mathrm{is1}$, we have that B accepts $\mathbf{subtree}_t(\pi 1)$ in state $\langle k_i, q_i', q_i'' \rangle$, where $k_i = ({a^{\#}}_i, h(\pi 1))$ for some ${a^{\#}}_i \in \Sigma_{\#}$. By the induction hypothesis there is a tour $(\pi 1, q_i') \cdots (\pi 1, q_i'')$ with ${a^{\#}}_i = \mathbf{label}_t(\pi 1)$. Hence, there is a simple tour $(\pi, q_i)(\pi 1, q_i') \cdots (\pi 1, q_i'')(\pi, q_{i+1})$.
- $\mathbf{subtree}_t(\pi 1), \mathbf{subtree}_t(\pi 2) \vdash\downarrow_2 \langle k_i, q_i', q_i'' \rangle$ where δ contains the transitions $q_i \xrightarrow{k,2} q_i'$ and $q_i'' \xrightarrow{k_i,\mathrm{up}} q_{i+1}$ with $k_i \in \Sigma_{\#} \times \{\mathrm{is2}\}$. By a similar argument to that above, we obtain a simple tour $(\pi, q_i)(\pi 2, q_i') \cdots (\pi 2, q_i'')(\pi, q_{i+1})$.

Since the above hold for all $i = 1, \ldots, n-1$, we have the tour

$$(\pi, q_1) \cdots (\pi, q_2) \cdots (\pi, q_{n-1}) \cdots (\pi, q_n)$$

as desired. □

As mentioned earlier, the converse does not hold and therefore the inclusion is strict.[2]

Theorem 12.2.5 (Bojańczyk and Colcombet, 2008) **TWA** $\subsetneq$ **ND**.

[2] Unlike ours, the usual definition of tree-walking automata does not allow node tests in final states. The theorem actually applies to this definition. However, the result should not change for our definition since their counterexample is not in **TWA** not because of this reason.

12.2.3 Variations

Deterministic tree-walking automata

A tree-walking automaton (Q, I, F, δ) is *deterministic* if

- I is a singleton and
- whenever $q_1 \xrightarrow{k,d} q_2$, $q_1 \xrightarrow{k,d'} q_2' \in \delta$, we have $d = d'$ and $q_2 = q_2'$.

Intuitively, for any tree there is at most a single run from the root with an initial state. We write **DTWA** for the class of languages accepted by deterministic tree-walking automata. It has also been proved that deterministic tree-walking automata are strictly less expressive than nondeterministic ones.[3]

Theorem 12.2.6 (Bojańczyk and Colcombet, 2006) $\mathbf{DTWA} \subsetneq \mathbf{TWA}$.

Alternating tree-walking automata

In Section 12.1.1 we introduced path expressions containing themselves as node tests and argued that these cannot in general be expressed by caterpillar expressions or, equivalently, by tree-walking automata. A direct way of allowing such node tests is to extend tree-walking automata by the use of conjunctions. Intuitively, the automaton walks not along a single-threaded path but along a multi-threaded "path" that forks at some point from which all the branches must succeed.

Formally, an alternating tree-walking automaton is a quadruple (Q, I, F, δ), where Q, I, and F are the same as for normal tree-walking automata and δ is a set of *sets* of transition rules of the form

$$q_1 \xrightarrow{k,d} q_2$$

where $q_1, q_2 \in Q$, $k \in K$, and $d \in \{\text{up}, 1, 2\}$. Given an alternating tree-walking automaton A and a tree t, a *run* of A on t is another tree, in which each node is labeled by a pair of the form (π, q), where $\pi \in \mathbf{nodes}(t)$ and $q \in Q$, in the following way: for each intermediate node that is labeled (π, q) and whose children are labeled $(\pi_1, q_1), \ldots, (\pi_n, q_n)$ there is a set of transitions

$$\begin{array}{c} q \xrightarrow{\kappa_t(\pi), d_1} q_1 \\ \vdots \\ q \xrightarrow{\kappa_t(\pi), d_n} q_n \end{array}$$

in δ such that each direction d_i matches the positional relationship between π and π_i. A run is *successful* if the root is labeled (ϵ, q) for some $q \in I$ and every leaf

[3] This theorem is also for tree-walking automata without node tests in final states. The result should also hold with our definition since their counterexample is not in **DTWA** for a different reason.

is labeled (ϵ, q) for some $(q, k) \in F$, with $\kappa_t(\epsilon) = k$. An alternating tree-walking automaton *accepts* a tree when there is a successful run, and we write **ATWA** for the class of languages accepted by alternating tree-walking automata.

It is rather easy to show that alternating tree-walking automata have at least the expressive power of normal tree automata, since each transition $q \to a(q_1, q_2)$ of a tree automaton can (roughly) be considered to be the conjunction of two alternating tree-walking automata transitions $q \xrightarrow{k,1} q_2$ and $q \xrightarrow{k,2} q_2$, for an appropriate k. Note that, for this encoding, alternating tree-walking automata do not need the ability to move up the tree.

Theorem 12.2.7 $\mathbf{ND} \subseteq \mathbf{ATWA}$.

It is known that the converse of this theorem also holds. In other words, the backward axis does not increase the expressiveness of alternating tree automata.

Theorem 12.2.8 (Slutzki, 1985) $\mathbf{ATWA} \subseteq \mathbf{ND}$.

12.3 Bibliographic notes

The specification of XPath is available in Clark and DeRose (1999). Regular expression paths have been proposed by database researchers (Abiteboul *et al.,* 1997; Buneman *et al.,* 2000; Deutsch *et al.,* 1998;). Caterpillar expressions was proposed in (Brüggemann-Klein and Wood, 2000). This paper presented *caterpillar automata* but, in fact, this notion is identical to that of tree-walking automata. An extension of path expressions was proposed that has the same expressive power as regular expression patterns with one variable (Murata, 2001).

Tree-walking automata were first introduced in Aho and Ullman (1971). In this initial formulation, automata do not have access to node kinds. This makes the expressiveness quite weak and, in particular, such automation are incapable of systematically traversing all the nodes of a tree (cf. Section 12.1.2). Indeed, for such automata, strict weakness relative to tree automata was shown in Kamimura and Slutzki (1981). The expressiveness substantially increases with the addition of access to node kinds. Nevertheless, the strict weakness property was proved in Bojańczyk and Colcombet (2008). The same authors had also proved that tree-walking automata cannot be determinized (Bojańczyk and Colcombet, 2006). The equivalence of tree-walking automata and their alternating variation was shown in Slutzki (1985).

As a separate line of work, extensive investigations have been made on the expressiveness and complexity properties of various fragments of XPath. A comprehensive survey can be found in Benedikt and Koch (2008).

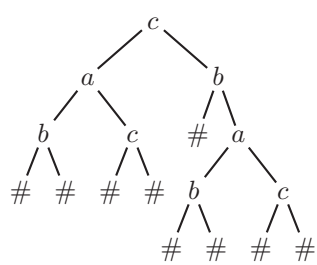
13

Logic-based queries

In this chapter we present the third approach to subtree extraction, namely, first-order (predicate) logic, and its extension, monadic second-order logic (MSO). An advantage of using logic-based queries is that this allows a direct way of expressing retrieval conditions. That is, we do not need to specify the structure of a whole tree (as for patterns in Chapter 5) nor to navigate around the tree (as for path expressions in Chapter 12); we simply need to specify the logical relationships among relevant nodes. Furthermore MSO can express any regular queries, that is, it is at least as powerful as marking tree automata (and thus patterns). However, MSO can express no more than regular queries; indeed, any MSO formula can be translated to a marking tree automaton (Chapter 6). It does mean, however, that we can use efficient on-the-fly algorithms (Chapter 8).

13.1 First-order logic

When we consider a logical framework, we usually think of *models* on which logic formulas are interpreted. In our setting, the models are XML documents; for simplicity, we consider here the binary tree format. Then we can talk about whether a given (binary) tree satisfies a given logic formula. For example, the tree



satisfies the formula

$$\forall x.\ (a(x) \Rightarrow \exists y.\exists z.\ b(y) \wedge c(z) \wedge \text{leftchild}(x, y) \wedge \text{rightchild}(x, z)).$$

Here, the predicate $a(x)$ means that the node x has label a, and similarly for $b(y)$ and $c(z)$. Also, leftchild(x, y) means that y is the left child of x; ditto for rightchild(x, y). So the whole formula can be read as "for any node with label a, its left and right children exist with labels b and c, respectively." Further, a formula that contains free variables is interpreted with respect to a tree *and* an assignment of the variables to nodes. For example, the above tree with the variable assignment

$$\{y \mapsto 11,\ z \mapsto 12\}$$

which maps y and z to the leftmost b node and its right-sibling c node, respectively, satisfies the formula

$$b(y) \wedge c(z) \wedge \exists x.\ \text{leftchild}(x, y) \wedge \text{rightchild}(x, z)$$

("node y has label b, node z has label c, and there is a node x such that y is its left child and z its right child"). Note that the assignment

$$\{y \mapsto 221,\ z \mapsto 222\}$$

which maps y and z to the bottom-most b node and its right-sibling c node, respectively, also satisfies the same formula. In general, for a given tree and a given formula, there are multiple variable assignments satisfying the formula. Thus we can regard a logic formula as a *query* on a tree that yields a set of variable assignments.

Formally, the syntax of (first-order) formulas ψ is defined by the following grammar, where x ranges over the (first-order) variables:

ψ	$::=$	$R_u(x)$	primitive unary relation
		$R_b(x_1, x_2)$	primitive binary relation
		$\psi_1 \wedge \psi_2$	conjunction
		$\psi_1 \vee \psi_2$	disjunction
		$\psi_1 \Rightarrow \psi_2$	implication
		$\neg\psi$	negation
		$\forall x.\ \psi$	universal quantification
		$\exists x.\ \psi$	existential quantification

As a primitive unary relation R_u, we have each label a from Σ as well as "root" and "leaf". Their formal meanings are as follows. Given a node π of a tree t,

$$\begin{aligned} a(\pi) &\overset{\text{def}}{\iff} \mathbf{label}_t(\pi) = a \\ \text{root}(\pi) &\overset{\text{def}}{\iff} \pi = \epsilon \\ \text{leaf}(\pi) &\overset{\text{def}}{\iff} \mathbf{label}_t(\pi) = \#. \end{aligned}$$

As a primitive binary relation we have leftchild and rightchild, as already seen, whose meanings are as follows. Given nodes π_1, π_2 of a tree t,

$$\begin{aligned} \text{leftchild}(\pi_1, \pi_2) &\stackrel{\text{def}}{\iff} \pi_2 = \pi_1 1 \\ \text{rightchild}(\pi_1, \pi_2) &\stackrel{\text{def}}{\iff} \pi_2 = \pi_1 2. \end{aligned}$$

We also have other binary relations: the equality $=$, the self-or-descendant relation $\leq$, and the document order $\preceq$ (the latter two are defined in Section 4.4.1). Then, a formula ψ is interpreted in terms of a binary tree t and a function γ that maps each free variable of ψ to a node of t. The semantics is described by a satisfaction judgment of the form $t, \gamma \vdash \psi$, read as "under tree t and assignment γ, formula ψ is satisfied" and defined inductively by the rules

$$\begin{aligned} t, \gamma \vdash R_u(x) &\stackrel{\text{def}}{\iff} R_u(\gamma(x)) \\ t, \gamma \vdash R_b(x_1, x_2) &\stackrel{\text{def}}{\iff} R_b(\gamma(x_1), \gamma(x_2)) \\ t, \gamma \vdash \psi_1 \wedge \psi_2 &\stackrel{\text{def}}{\iff} t, \gamma \vdash \psi_1 \text{ and } t, \gamma \vdash \psi_2 \\ t, \gamma \vdash \psi_1 \vee \psi_2 &\stackrel{\text{def}}{\iff} t, \gamma \vdash \psi_1 \text{ or } t, \gamma \vdash \psi_2 \\ t, \gamma \vdash \psi_1 \Rightarrow \psi_2 &\stackrel{\text{def}}{\iff} t, \gamma \vdash \psi_1 \text{ implies } t, \gamma \vdash \psi_2 \\ t, \gamma \vdash \neg\psi_1 &\stackrel{\text{def}}{\iff} \text{not } t, \gamma \vdash \psi_1 \\ t, \gamma \vdash \forall x.\, \psi &\stackrel{\text{def}}{\iff} \text{for all } \pi \in \mathbf{nodes}(t),\ \ t, \gamma \cup \{x \mapsto \pi\} \vdash \psi \\ t, \gamma \vdash \exists x.\, \psi &\stackrel{\text{def}}{\iff} \text{for some } \pi \in \mathbf{nodes}(t),\ \ t, \gamma \cup \{x \mapsto \pi\} \vdash \psi. \end{aligned}$$

In Chapters 5 and 12 we introduced other formalisms for subtree extraction, namely, patterns and path expressions. What advantages do logic-based queries have over these?

The most important advantage is the ability of logic-based queries to express retrieval conditions *directly*. With patterns, we need to specify the structure of the whole tree even though there are usually many more nodes that are irrelevant to the query than relevant ones. Here, irrelevant nodes mean that we *do not care* whether these nodes exist. Logic, however, has a built-in "don't-care" semantics in the sense that it allows us not to mention such nodes at all. Moreover, when we want to extract a node from deeper places in the tree, patterns require us to write a recursive definition (Section 5.2.1) whereas logic allows us to jump directly to the subject node. For example, consider extracting all nodes x with label a. In patterns, we need to write the following recursively defined pattern:

```
X = (Any, (a[Any] as x), Any)
  | (Any, ~[X], Any)
```

That is, the a node that we want to extract is located either somewhere at the top level or at a deeper level through some label. Here, we assume that patterns are extended with the "wildcard" label `~` that matches any label and the "wildcard"

type `Any` that matches any value (of course, the `Any` type itself can be defined in terms of the ~ label and recursion). In logic, we only need to write the following:

$$a(x)$$

Unlike the case of patterns, here we do not need to mention any nodes other than the node in which we are interested; we do not need to form a recursion to reach a node that may be located at an arbitrarily deep position. In this sense, path expressions are better than patterns since we only need to write the following (in the caterpillar expression notation) for expressing the same thing:

$$(1|2)^*a$$

The meaning of this expression is that we collect nodes with label a that are reachable from the root by repeatedly following either the left or the right child. However, we still need *navigation* from the root, which is not necessary in logic. What concerns us here is not visual succinctness but the number of concepts that are involved in expressing the same condition.

Let us consider next a more substantial example. Suppose that we want to build, from an XHTML document, a table of contents that reflects the implicit structure among the heading tags, `h1`, `h2`, `h3`, etc. For this, we need to collect a set of pairs of `h1` node x and `h2` node y such that y "belongs" to x (this set of pairs can be viewed as a mapping from an `h1` node to a set of `h2` nodes belonging to the `h1` node, from which we can form a list of sections, each having a list of subsections). This query can be expressed in logic by the following formula:

$$h1(x) \wedge h2(y) \wedge x \preceq y \wedge \forall z.\, (h1(z) \wedge x \prec z \Rightarrow y \preceq z)$$

where $x \prec z$ stands for $x \preceq z \wedge \neg(x = z)$. That is, the `h2` node y must come after the `h1` node x and, for any `h1` node z that comes after the node x, the node y must come before z. Note that XHTML headings may not appear in a flat list but can be intervened by other tags, and this is the reason why we need to use the document order $\preceq$. As an example, in the following fragment of document,

```
<body>
  <h1> ... </h1>
  <h2> ... </h2>
  <p> <h2> ... </h2> </p>
  <div> <h1> ... </h1> </div>
  <h2> ... </h2>
</body>
```

we associate the first `h1` with the first `h2` and the second `h2` but not with the last `h2`, since the second `h1` appears in between.

How can we express this example by a pattern? First, if we simplify the example so that only a flat list of heading tags may appear, then we can easily express this simplified condition as follows:

```
Any, (h1[Any] as x), (^h1)[Any]*, (h2[Any] as y), Any
```

(where we introduce the notation `^h1` to indicate that any label should be matched except `h1`). In the general case, however, it becomes extremely difficult mainly because there is no easy way to express document order with patterns. This comparison of logic and patterns might be unfair, though, since logic provides the document order as a primitive. Even so, it is difficult to imagine how patterns could be extended to include document order, and therefore the advantage of logic seems clear. Relevantly, we will see in the next section that document order itself can easily be expressed by a slightly extended logical framework, namely, MSO.

What about expressing the above example as a path expression? We could easily extend path expressions by a "document order axis." However, even with this, it would still not be possible to use a path expression without some form of universal quantification. Essentially, the path expressions introduced in Chapter 12, in particular those without conjunction (i.e., those for which node tests do not contain path expressions), can combine only existential conditions on nodes.

13.2 Monadic second-order logic

Monadic second-order (MSO) logic adds to first-order logic the ability to quantify over sets of nodes. Here, "second-order" by itself means quantification over relations and "monadic" means that relations to be quantified can take only one argument and are therefore sets. We present MSO as interpreted under finite binary trees, which is sometimes called weak second-order logic with two successors or WS2S ("weak" for finite models and "two successors" for binary trees), but we prefer to call it MSO here.

We extend the syntax of first-order logic as follows, where X ranges over second-order variables:

$$
\begin{array}{lll}
\psi & ::= & \vdots \\
 & & x \in X \quad \text{variable unary relation} \\
 & & \forall X.\, \psi \quad \text{second-order universal quantification} \\
 & & \exists X.\, \psi \quad \text{second-order existential quantification}
\end{array}
$$

Accordingly, the semantics is extended so that a formula is interpreted under a second-order variable assignment Γ (a mapping from second-order variables to sets of nodes) in addition to a tree and a first-order assignment. We thus extend the

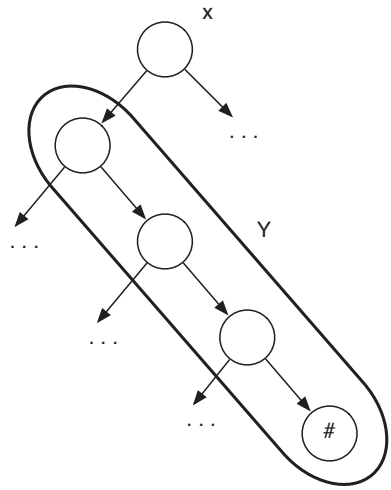


Figure 13.1 The relation xchildren(x, Y).

satisfaction judgment as $t, \gamma, \Gamma \vdash \psi$ and define it by the rules

$$\begin{aligned}
t, \gamma, \Gamma \vdash x \in X &\overset{\text{def}}{\iff} \gamma(x) \in \Gamma(X) \\
t, \gamma, \Gamma \vdash \forall X.\ \psi &\overset{\text{def}}{\iff} \text{for all } \Pi \subseteq \mathbf{nodes}(t),\ \ t, \gamma, \Gamma \cup \{X \mapsto \Pi\} \vdash \psi \\
t, \gamma, \Gamma \vdash \exists X.\ \psi &\overset{\text{def}}{\iff} \text{for some } \Pi \subseteq \mathbf{nodes}(t),\ \ t, \gamma, \Gamma \cup \{X \mapsto \Pi\} \vdash \psi
\end{aligned}$$

plus the first-order rules already given, with the addition of Γ in the judgment.

Let us see how powerful second-order quantification is. As a start, let us define the relation xchild(x, y) to mean that y is a child of x in the unranked tree (XML) view. For this, we first define the auxiliary relation xchildren(x, Y) to mean that Y is the set of children of x:

$$\text{xchildren}(x, Y) \equiv \forall y.\ y \in Y \Leftrightarrow (\text{leftchild}(x, y) \vee \exists z.\ (z \in Y \wedge \text{rightchild}(z, y)))$$

where $\psi_1 \Leftrightarrow \psi_2$ stands for $\psi_1 \Rightarrow \psi_2 \wedge \psi_2 \Rightarrow \psi_1$. This relation says that each element in Y is either the left child of x or the right child of some other element in Y and vice versa. This is depicted in Figure 13.1. Using the definition of xchildren, we can easily define xchild, as follows:

$$\text{xchild}(x, y) \equiv \exists Y.\ (\text{xchildren}(x, Y) \wedge y \in Y)$$

As an example of a use of xchild, the following collects all b nodes y appearing as a child of an a node (in the XML view):

$$\exists x.\ \text{xchild}(x, y) \wedge a(x) \wedge b(y)$$

Note here that relation definitions are not part of our logical framework but simply an informal macro notation. For example, the use of xchild(x, y) above should be

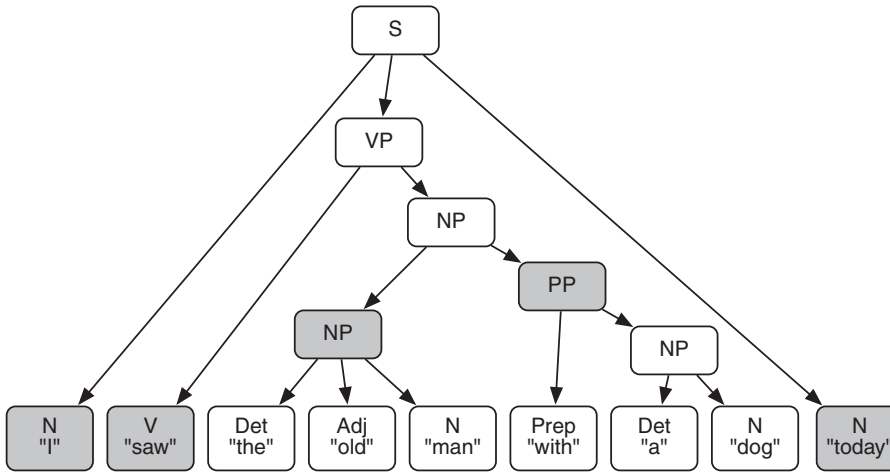


Figure 13.2 A proper analysis (consisting of the shaded nodes) on a parse tree, whose nodes are labeled by grammatical categories: S, sentence; VP, verb phrase; etc.

expanded by the body of xchild, in which the use of xchildren(x, Y) should further be expanded by the body of xchildren. This repeated expansion works since we never define "recursive" relations.

Exercise 13.2.1 (⋆) In the definition of xchildren, if we replace the symbol $\Leftrightarrow$ with the symbol $\Leftarrow$, how does the meaning change? What happens if we replace $\Rightarrow$ by $\Leftarrow$? ■

Exercise 13.2.2 (⋆⋆) In the same fashion as for xchild, define xdescendant, standing for the self-or-descendant relation in the XML view. Also, define the self-or-descendant relation $\leq$ and the document order $\preceq$, which were the primitives in the first-order logic in the previous section. ■

Let us conclude this section with a more substantial example from computational linguistics. In this area we often need to query a parse tree of a sentence in a natural language. Figure 13.2 shows an example of a parse tree. In such a tree, we sometimes need to find all nodes that follow after a given node in a "proper analysis." A proper analysis originally referred to the set of nodes that appear at a certain moment during a parsing process of a given sentence, but for the present purpose it is enough to define it by a set of nodes such that each node not in the set is either an ancestor or a descendant of a node from this set and vice versa. Let us define the "follow in a proper analysis" relation in MSO. First, set

$$x//y \equiv \text{xdescendant}(x, y) \wedge \neg(x = y)$$

where xdescendant is defined in Exercise 13.2.2. Then, the notion of proper analysis can be expressed by

$$\text{proper\_analysis}(A) \equiv \forall x.\, \neg(x \in A) \Leftrightarrow \exists y.\, y \in A \wedge (x//y \vee y//x)$$

and the "follow" relation by

$$\text{follow}(x, y) \equiv \exists A.\, \text{proper\_analysis}(A) \wedge x \in A \wedge y \in A \wedge x \preceq y.$$

Observe that these *literally* translate the informal descriptions to logical formulas.

13.3 Regularity

As already mentioned at the beginning of the chapter, MSO expresses only regular queries and thus is equivalent to marking tree automata. This section aims at establishing this equivalence. We will show below a concrete construction of a marking tree automaton from an MSO formula. The opposite direction will be suggested as an exercise for the reader.

13.3.1 Canonicalization

In order to simplify the presentation, we first translate a given MSO formula into a canonical form. In this form we elide, as usual, logical combinators expressible by other combinators but, further, eliminate first-order variables. Thus, each first-order variable used in the original formula is replaced with a second-order variable confined to a singleton set, and each first-order quantification or primitive relation is modified in a way that preserves its meaning.

Formally, we define the syntax of canonical (MSO) formulas by the following:

$$\begin{aligned} \psi_c \quad ::= \quad & X \subseteq Y \mid R_u(X) \mid R_b(X, Y) \mid \mathbf{single}(X) \\ & \mid \neg\psi_c \mid \psi_c \wedge \psi'_c \mid \exists X.\, \psi_c \end{aligned}$$

Here we do not have disjunction ($\vee$), implication ($\Rightarrow$), or second-order universal quantification ($\forall X$); these will be represented, as usual, by combinations of conjunction, negation, and second-order existential quantification. Neither do we have set membership ($x \in Y$), primitive relations on first-order variables ($R_u(x)$ and $R_b(x, y)$), or first-order quantification ($\forall x$ and $\exists x$). For encoding these, we will first introduce a second-order variable X_x for each first-order variable x. In order to ensure that X_x refers to exactly one element, as in the original formula, we add the extra condition $\mathbf{single}(X_x)$, meaning that the set X_x is a singleton. Then, assuming that X_x refers to the same element as that to which x refers, we replace any given formula containing x by another containing X_x. Namely, set membership $x \in Y$ is replaced with set inclusion $X_x \subseteq Y$ and primitive relations $R_u(x)$ and $R_b(x, y)$ with

$R_u(X_x)$ and $R_b(X_x, X_y)$, respectively. Here, we extend the semantics of R_u and R_b to apply to second-order variables in the following way: assuming that $\Pi = \{\pi\}$, $\Pi_1 = \{\pi_1\}$, and $\Pi_2 = \{\pi_2\}$, we make the definitions

$$\begin{array}{lcl} R_u(\Pi) & \stackrel{\text{def}}{\Longleftrightarrow} & R_u(\pi) \\ R_b(\Pi_1, \Pi_2) & \stackrel{\text{def}}{\Longleftrightarrow} & R_b(\pi_1, \pi_2). \end{array}$$

Note that the semantics of R_u is undefined when Π is not a singleton; likewise that of R_b is undefined when either Π_1 or Π_2 is not a singleton. This fact will be crucial in the automata construction below. Also, we need only leftchild and rightchild as primitive binary relations, since the relations $\leq$ and $\preceq$ can be defined within MSO (Section 13.2), and the equality $x = y$ is translated to $X_x = X_y$, which can be rewritten as $X_x \subseteq X_y \wedge X_y \subseteq X_x$, using only inclusions.

Now the canonicalization $\mathbf{canon}(\psi)$ of a formula is defined as follows:

$$\begin{array}{lcl} \mathbf{canon}(x \in Y) & = & X_x \subseteq Y \\ \mathbf{canon}(R_u(x)) & = & R_u(X_x) \\ \mathbf{canon}(R_b(x, y)) & = & R_b(X_x, X_y) \\ \mathbf{canon}(\psi_1 \vee \psi_2) & = & \neg(\neg\mathbf{canon}(\psi_1) \wedge \neg\mathbf{canon}(\psi_2)) \\ \mathbf{canon}(\psi_1 \Rightarrow \psi_2) & = & \neg(\mathbf{canon}(\psi_1) \wedge \neg\mathbf{canon}(\psi_2)) \\ \mathbf{canon}(\psi_1 \wedge \psi_2) & = & \mathbf{canon}(\psi_1) \wedge \mathbf{canon}(\psi_2) \\ \mathbf{canon}(\exists X.\ \psi) & = & \exists X.\ \mathbf{canon}(\psi) \\ \mathbf{canon}(\forall X.\ \psi) & = & \neg\exists X.\ \neg\mathbf{canon}(\psi) \\ \mathbf{canon}(\exists x.\ \psi) & = & \exists X_x.\ \mathbf{single}(X_x) \wedge \mathbf{canon}(\psi) \\ \mathbf{canon}(\forall x.\ \psi) & = & \neg\exists X_x.\ \mathbf{single}(X_x) \wedge \neg\mathbf{canon}(\psi) \end{array}$$

Here, the translation of a first-order universal quantification can be understood as transforming $\forall x.\ \psi$ first to $\neg\exists x.\ \neg\psi$ and then to $\neg\exists X_x.\ \mathbf{single}(X_x) \wedge \neg\mathbf{canon}(\psi)$ (applying the translation of a first-order existential). Note that it is incorrect to swap the order, that is, to transform $\forall x.\ \psi$ first to $\forall X_x.\ \mathbf{single}(X_x) \wedge \mathbf{canon}(\psi)$ and then to $\neg\exists X_x.\ \neg\mathbf{single}(X_x) \vee \neg\mathbf{canon}(\psi)$, since the internal existential in the result would always hold because a non-singleton set X_x always exists.

The formal correctness of canonicalization can be stated as follows: a given formula ψ holds under first- and second-order assignments γ and Γ if and only if the canonical formula $\mathbf{canon}(\psi)$ holds under the canonicalization of γ. The canonicalization of a first-order assignment is defined as follows:

$$\mathbf{canon}(\{x_1 \mapsto \pi_1, \ldots, x_n \mapsto \pi_n\}) = \{X_{x_1} \mapsto \{\pi_1\}, \ldots, X_{x_n} \mapsto \{\pi_n\}\}$$

Lemma 13.3.1 We have that $t, \gamma, \Gamma \vdash \psi$ iff $t, \emptyset, \Gamma \cup \mathbf{canon}(\gamma) \vdash \mathbf{canon}(\psi)$.

Proof: The proof proceeds by induction on the structure of ψ.

Case 1: $\psi = x \in Y$

Note that $\mathbf{canon}(\psi) = X_x \subseteq Y$. We can make the derivation

$$\begin{aligned} t, \gamma, \Gamma \vdash x \in Y \quad &\Leftrightarrow \quad \gamma(x) \in \Gamma(Y) \\ &\Leftrightarrow \quad \mathbf{canon}(\gamma)(X_x) = \{\gamma(x)\} \subseteq \Gamma(Y) \\ &\Leftrightarrow \quad t, \emptyset, \Gamma \cup \mathbf{canon}(\gamma) \vdash X_x \subseteq Y. \end{aligned}$$

The other base cases can shown similarly.

Case 2: $\psi = \forall x.\, \psi'$
Note that $\mathbf{canon}(\psi) = \neg\exists X_x.\, \mathbf{single}(X_x) \wedge \neg\mathbf{canon}(\psi')$. We can make the derivation

$$\begin{aligned} & t, \gamma, \Gamma \vdash \forall x.\psi' \\ & \quad \Leftrightarrow \quad \forall \pi \in \mathbf{nodes}(t).\ t, \gamma \cup \{x \mapsto \pi\}, \Gamma \vdash \psi' \\ & \quad \Leftrightarrow \quad \nexists \pi \in \mathbf{nodes}(t).\ t, \gamma \cup \{x \mapsto \pi\}, \Gamma \nvdash \psi' \\ & \quad \overset{\text{I.H.}}{\Leftrightarrow} \quad \nexists \pi \in \mathbf{nodes}(t).\ t, \emptyset, \Gamma \cup \mathbf{canon}(\gamma) \cup \{X_x \mapsto \{\pi\}\} \nvdash \mathbf{canon}(\psi') \\ & \quad \Leftrightarrow \quad \nexists \Pi \subseteq \mathbf{nodes}(t).\ \mathbf{single}(\Pi) \wedge \\ & \qquad\qquad t, \emptyset, \Gamma \cup \mathbf{canon}(\gamma) \cup \{X_x \mapsto \Pi\} \vdash \neg\mathbf{canon}(\psi') \\ & \quad \Leftrightarrow \quad \nexists \Pi \subseteq \mathbf{nodes}(t). \\ & \qquad\qquad t, \emptyset, \Gamma \cup \mathbf{canon}(\gamma) \cup \{X_x \mapsto \Pi\} \vdash \mathbf{single}(X_x) \wedge \neg\mathbf{canon}(\psi') \\ & \quad \Leftrightarrow \quad t, \emptyset, \Gamma \cup \mathbf{canon}(\gamma) \vdash \neg\exists X_x.\, \mathbf{single}(X_x) \wedge \neg\mathbf{canon}(\psi'). \end{aligned}$$

Here, the induction hypothesis is used to derive the third equivalence. The other inductive cases can be shown by straightforward use of the induction hypothesis. □

Note that Lemma 13.3.1 does not tell us whether $\mathbf{canon}(\psi)$ holds when variables that were originally first-order are inappropriately assigned to non-singleton sets. However, in queries with a canonical formula we actually want to collect only appropriate assignments, and therefore we need to ensure that the canonical formula holds only under such assignments. Thus, for a given top-level formula ψ_0, we add, to the canonical form of ψ_0, the singleton constraints on the converted first-order variables, that is,

$$\mathbf{canon}(\psi_0) \wedge \bigwedge_{x \in \mathbf{FV}_1(\psi_0)} \mathbf{single}(X_x)$$

where $\mathbf{FV}_1(\psi_0)$ is the set of first-order free variables appearing in ψ_0.

13.3.2 Automata construction

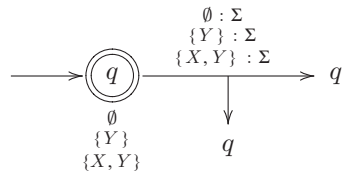
Having a canonical formula ψ_c in hand, what remains is to construct a marking tree automaton M such that

$t, \emptyset, \Gamma \vdash \psi_c$ holds if and only if M accepts t with marking Γ

where a second-order assignment (a mapping from variables to node sets) is regarded as a marking (a mapping from nodes to variable sets).

The construction is done by induction on the structure of the formula. First, let us see the base cases.

Case 1: $\psi_c = X \subseteq Y$
This formula is translated to the following:

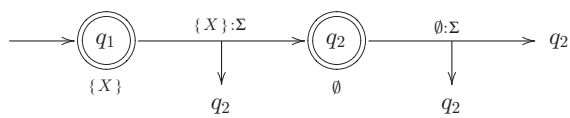


This depicts a marking tree automaton with a single state q, which is initial, and three kinds of transition, namely,

$$q \to \emptyset : a(q, q) \qquad q \to \{Y\} : a(q, q) \qquad q \to \{X, Y\} : a(q, q)$$

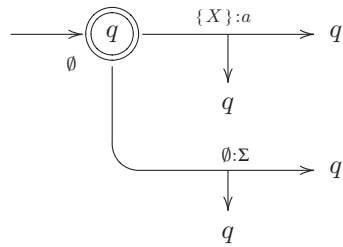
for every label $a \in \Sigma$. Here, the q within the double circle and the q's to which the arrows lead are the same state (we avoid arrows looping around for readability). The state q is final, with a variable set equal to $\emptyset$, $\{Y\}$, or $\{X, Y\}$; that is, the pairs $(q, \emptyset)$, $(q, \{Y\})$, and $(q, \{X, Y\})$ are in the F-set of the marking tree automaton. Thus the automaton accepts any tree and marks each node with X whenever it also marks the same node with Y. In other words, the set of X-marked nodes is a subset of the set of Y-marked nodes. The notational convention here will be used in the rest of the proof.

Case 2: $\psi_c = \mathrm{root}(X)$
This formula is translated to the following:



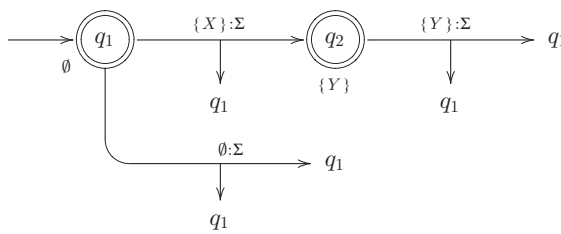
This automaton consists of two states, q_1 and q_2. The state q_2 accepts any tree with no marking. Thus, the state q_1 accepts any tree with the marking X on the root node (no matter whether it is a leaf or an intermediate node).

Case 3: $\psi_c = a(X)$
This formula is translated to the following:



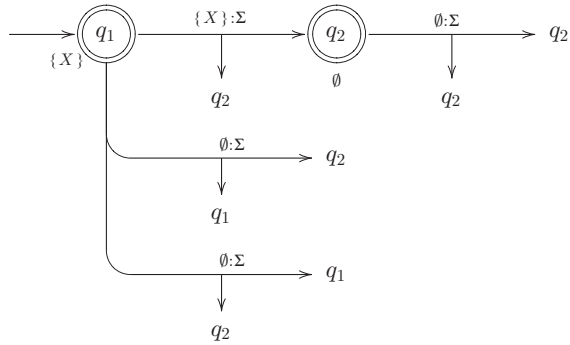
This automaton consists of a single state q with two transitions. The state would accept any tree with no marking by always following the second transition. However, it can also take the first transition, which accepts a node with label a and marks it with X. Therefore, if there is exactly one node marked with X then this node must be labeled a, as desired by the semantics of $a(X)$. If there is no node or more than one node marked with X then the tree may or may not be accepted, but it does not matter which (recall that the semantics of $a(X)$ is undefined in such a case). The formula leaf(X) can be translated similarly.

Case 4: rightchild(X, Y)
This formula is translated to the following:



This automaton consists of two states, q_1 and q_2. Again the state q_1 accepts any tree with no marking by always following the second transition. However, when we take the first transition we mark the current node with X and then, at the state q_2, the right child node with Y. Note also that the automaton implements the desired meaning when we use each of X and Y exactly once. In other cases the behavior of the automaton does not matter. A formula leftchild(X, Y) can be translated similarly.

Case 5: $\psi_c = \mathbf{single}(X)$
This formula is translated to the following:



Again the automaton consists of two states, q_1 and q_2. First, the state q_2 accepts any tree with no marking. Then, the state q_1 also accepts any tree but marks X on exactly one node. Indeed, if the current tree is a leaf then we mark it with X. If it is an intermediate node then we take any of three transitions. In the first transition we put the mark X on the current node and no mark on any node below. In the second, we put no mark on the right subtree but continue with the state q_1 for the left subtree, that is, we mark X exactly once somewhere in this subtree. The third transition is similar except that we mark X exactly once in the right subtree instead.

Next, let us see the inductive cases.

Case 6: $\psi_c = \neg\psi'_c$
To translate this formula we first compute a marking tree automaton from the subformula ψ'_c. Then we notice that a marking tree automaton can be regarded as an ordinary tree automaton each of whose labels is a pair consisting of a label from Σ and a set of variables (thus the automaton accepts a binary tree each of whose nodes is already marked with a set of variables). Now, we take the complement of this tree automaton. Note that such an automaton requires multiple leaf symbols in general (owing to the presence of variable sets), whereas our definition of tree automata allows only a single symbol (#). However, this generalization is entirely straightforward and all procedures for basic set operations can easily be adapted accordingly.

Case 7: $\psi_c = \psi'_c \wedge \psi''_c$
As in the previous case, we first compute marking tree automata from the subformulas ψ'_c and ψ''_c. However, we cannot simply take their intersection since the sets of variables used in the two marking automata can be different. To make the

necessary adjustments, we augment the set of transitions in one automaton so as to allow arbitrary extra variables to be used only in the other automaton. That is, suppose that the two marking automata, M_1 and M_2, allow sets $\mathcal{X}_1$ and $\mathcal{X}_2$ of variables, respectively. Then, for each transition

$$q \to \mathbf{x} : a(q_1, q_2)$$

in the automaton M_1 we add the transition

$$q \to (\mathbf{x} \cup \mathbf{x}') : a(q_1, q_2)$$

for any $\mathbf{x}'$ such that $\emptyset \subsetneq \mathbf{x}' \subseteq (\mathcal{X}_2 \setminus \mathcal{X}_1)$ (i.e., any variables that are only in $\mathcal{X}_2$). The other automaton M_2 is adjusted similarly. After this operation, regarding the adjusted marking automata as ordinary tree automata, just as in the previous case, we take their intersection.

Case 8: $\psi_c = \exists X.\, \psi'_c$
To translate this formula we first compute a marking tree automaton for ψ'_c as before and then simply remove the variable X from any transition. That is, each transition

$$q \to \mathbf{x} : a(q_1, q_2)$$

is replaced with

$$q \to (\mathbf{x} \setminus \{X\}) : a(q_1, q_2).$$

This operation can be understood by noticing that ψ'_c holds for a tree t under an assignment Γ if and only if ψ_c holds for the same tree t under the assignment Γ *minus* the mapping for the variable X.

Note that, if a top-level formula contains only first-order free variables then its canonical formula added with singleton constraints (as at the end of Section 13.3.1) is translated to a linear marking tree automaton, that is, one that marks each variable on exactly one node (Section 6.1). This fact enables us to use the efficient on-the-fly evaluation algorithm for marking tree automata presented in Section 8.2.

Exercise 13.3.2 (⋆⋆⋆) Conversely, a marking tree automaton can be translated to an MSO formula by directly expressing the semantics of marking tree automata in MSO. Give a concrete procedure. ■

13.4 Bibliographic notes

A theoretical connection of MSO with tree transformation was made in the framework of MSO-definable tree transducers (Courcelle, 1994). The first attempt to

bring the MSO logic to practical XML transformation was made in the work on the *MTran* language (Inaba and Hosoya, 2007); some examples in this chapter are taken from this. A compilation from MSO to marking tree automata was first given in Thatcher and Wright (1968). Although the lower bound of the time and space complexity required for such a compilation is hyper-exponential (i.e., a stack of exponentials whose height is proportional to the size of the formula) (Meyer, 1975), the MONA system (Henriksen *et al.*, 1995) provided an efficient implementation of MSO based on binary decision diagrams and various other techniques. The MSO compilation presented in this chapter was taken from the MONA manual (Klarlund and Møller, 2001), with a slight modification. Other workers have proposed XML processing based on other logical frameworks, such as Ambient Logic (Cardelli and Ghelli, 2004), Sheaves Logic (Zilio and Lugiez, 2003), or Context Logic (Calcagno *et al.*, 2005).

14

Ambiguity

Ambiguity refers to the property whereby regular expressions or patterns have multiple matching possibilities. As discussed in Chapter 5, ambiguity can make the behavior of a program harder to understand and can actually be the result of a programming error. Therefore it is often useful to report it to the user. However, when we come to ask exactly what we mean by ambiguity, there is no consensus. In this chapter, we review three different definitions, strong ambiguity, weak ambiguities and binding ambiguity, and discuss how these notions are related to each other and how they can be checked algorithmically.

Caveat: In this chapter, we concentrate on regular expressions and patterns on *strings* rather than on trees in order to highlight the essence of ambiguity. The extension for the case of trees is routine and can be found in the literature.

14.1 Ambiguities for regular expressions

In this section, we study strong and weak ambiguities and how they can be decided by using an ambiguity checking algorithm for automata.

14.1.1 Definitions

Ambiguity arises when a regular expression has several occurrences of the same label. Therefore we need to be able to distinguish between these occurrences. Let $\tilde{\Sigma}$ be the set of *elaborated labels*, written by $a^{(i)}$, where $a \in \Sigma$ and i is an integer. We use $\tilde{s}$ to range over the elaborated strings from $\tilde{\Sigma}^*$ and s to range over the strings from Σ^*. When s is the string obtained after removing all the integers from $\tilde{s}$ we say that $\tilde{s}$ is an *elaboration* of s and that s is the *unelaboration* of $\tilde{s}$; we write **unelab**$(\tilde{s})$ for such an s. Let r range over regular expressions on $\tilde{\Sigma}$, where every occurrence of a label is given a unique integer, as for example in $a^{(1)*}b^{(2)}a^{(3)*}$. Throughout this section we consider only such regular expressions.

We define strong ambiguity in terms of a *conformance* relation, written $s \ in \ r$. The relation is defined by the following set of rules, which are exactly the same as those in Section 2.1 except for RE-SYM:

$$\frac{}{\epsilon \ in \ \epsilon}\ \text{RE-EPS}$$

$$\frac{}{a \ in \ a^{(i)}}\ \text{RE-SYM}$$

$$\frac{s \ in \ r_1}{s \ in \ r_1 \mid r_2}\ \text{RE-ALT1}$$

$$\frac{s \ in \ r_2}{s \ in \ r_1 \mid r_2}\ \text{RE-ALT2}$$

$$\frac{s_1 \ in \ r_1 \qquad s_2 \ in \ r_2}{s_1 s_2 \ in \ r_1 r_2}\ \text{RE-CAT}$$

$$\frac{s_i \ in \ r \quad \forall i = 1, \ldots, n \qquad n \geq 0}{s_1 \cdots s_n \ in \ r^*}\ \text{RE-REP}$$

Then, a regular expression r is *strongly unambiguous* if, for any string s, there is at most one derivation of the relation $s \ in \ r$ (see the end of Section 2.1).

Example 14.1.1 The regular expression $r_{14.1.1} = (a^{(1)*})^*$ is strongly ambiguous since there are at least two derivations of the relation $aa \ in \ r_{14.1.1}$:

$$\frac{\dfrac{a \ in \ a^{(1)}}{a \ in \ a^{(1)*}} \qquad \dfrac{a \ in \ a^{(1)}}{a \ in \ a^{(1)*}}}{aa \ in \ (a^{(1)*})^*} \qquad \frac{\dfrac{a \ in \ a^{(1)} \qquad a \ in \ a^{(1)}}{aa \ in \ a^{(1)*}} \qquad \epsilon \ in \ a^{(1)*}}{aa \ in \ (a^{(1)*})^*}$$

■

Example 14.1.2 The regular expression $r_{14.1.2} = a^{(1)} \mid a^{(2)}$ is also strongly ambiguous since there are at least two derivations of the relation $a \ in \ r_{14.1.2}$:

$$\frac{a \ in \ a^{(1)}}{a \ in \ a^{(1)} \mid a^{(2)}} \qquad \frac{a \ in \ a^{(2)}}{a \ in \ a^{(1)} \mid a^{(2)}}$$

Note that the elaboration of the regular expression makes these two derivations distinct. ■

Let the language $\mathcal{L}(r) \subseteq \tilde{\Sigma}^*$ of a regular expression r be defined in the standard way; we also say that r *generates* an element of $\mathcal{L}(r)$. Note that $s \ in \ r$ holds if

and only if there is a string $\tilde{s} \in \mathcal{L}(r)$ such that $s = \mathbf{unelab}(\tilde{s})$. Then r is *weakly unambiguous* if r generates at most one elaboration of s for any string s.

Example 14.1.3 The regular expression $r_{14.1.1}$ is weakly unambiguous since it generates only strings of the form $a^{(1)} \cdots a^{(1)}$. The regular expression $r_{14.1.2}$ is weakly ambiguous since it generates $a^{(1)}$ and $a^{(2)}$. ■

Example 14.1.4 The regular expression $r_{14.1.4} = (a^{(1)} \mid a^{(2)}b^{(3)})(b^{(4)} \mid \epsilon)$ is strongly ambiguous since there are two derivations of the relation $ab \ in \ r_{14.1.4}$:

$$\dfrac{\dfrac{a \ in \ a^{(1)}}{ab \ in \ (a^{(1)} \mid a^{(2)}b^{(3)})} \qquad \dfrac{b \ in \ b^{(4)}}{b \ in \ (b^{(4)} \mid \epsilon)}}{ab \ in \ (a^{(1)} \mid a^{(2)}b^{(3)})(b^{(4)} \mid \epsilon)} \qquad \dfrac{\dfrac{\dfrac{a \ in \ a^{(2)} \quad b \ in \ b^{(3)}}{ab \ in \ a^{(2)}b^{(3)}}}{ab \ in \ (a^{(1)} \mid a^{(2)}b^{(3)})} \qquad \dfrac{\epsilon \ in \ \epsilon}{\epsilon \ in \ (b^{(4)} \mid \epsilon)}}{ab \ in \ (a^{(1)} \mid a^{(2)}b^{(3)})(b^{(4)} \mid \epsilon)}$$

The regular expression is also weakly ambiguous since it generates $a^{(1)}b^{(4)}$ and $a^{(2)}b^{(3)}$. ■

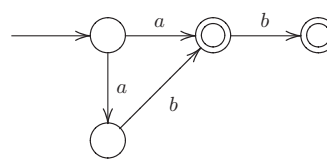
In Example 14.1.4, for any derivation of a relation $s \ in \ r$ the concatenation (from left to right) of all the elaborated labels appearing in the leaves of the derivation coincides with an elaboration of s in $\mathcal{L}(r)$. Therefore, if r generates two different elaborations of s then the derivations corresponding to these must be different. This observation leads to the following proposition.

Proposition 14.1.5 If a regular expression r is strongly unambiguous then it is weakly unambiguous.

The converse does not hold; $r_{14.1.1}$ is a counterexample.

Ambiguity can also be defined for (string) automata. Recall from Section 2.2 that, given an automaton $A = (Q, I, F, \delta)$, a *run* on a string $s = a_1 \cdots a_{n-1}$ is a sequence $q_1 \cdots q_n$ of states from Q such that $q_i \xrightarrow{a_i} q_{i+1} \in \delta$ for each $i = 1, \ldots, n-1$ with $q_1 \in I$; such a run *accepts* s when $q_n \in F$. Then an automaton is *unambiguous* if, for any string $s \in \Sigma^*$, there is at most one run accepting s.

Example 14.1.6 The automaton



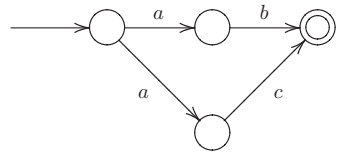
is ambiguous since there are two runs accepting ab. ■

The following proposition follows in an obvious way.

Proposition 14.1.7 If an automaton is deterministic then it is unambiguous.

The converse does not hold; the following shows a counterexample.

Example 14.1.8 The automaton



is nondeterministic yet unambiguous. ■

14.1.2 Glushkov automata and star normal form

Having defined three different notions of ambiguity (strong ambiguity, weak ambiguity, and ambiguity for automata), the next question is what the relationships are among them. The key concepts connecting them are Glushkov automata and the star normal form.

For an (elaborated) regular expression r, we define the following:

$$\begin{array}{lcl} \mathbf{pos}(r) & = & \{a^{(i)} \mid \tilde{s}a^{(i)}\tilde{s}' \in \mathcal{L}(r)\} \\ \mathbf{first}(r) & = & \{a^{(i)} \mid a^{(i)}\tilde{s} \in \mathcal{L}(r)\} \\ \mathbf{last}(r) & = & \{a^{(i)} \mid \tilde{s}a^{(i)} \in \mathcal{L}(r)\} \\ \mathbf{follow}(r, a^{(i)}) & = & \{b^{(j)} \mid \tilde{s}a^{(i)}b^{(j)}\tilde{s}' \in \mathcal{L}(r)\} \end{array}$$

Intuitively, these sets contain the elaborated labels appearing in r's words, those at the beginning, those at the end, and those just after $a^{(i)}$, respectively. We can easily compute these sets from the given regular expression.

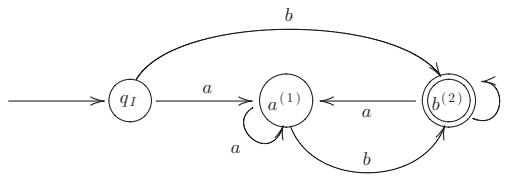
Exercise 14.1.9 (⋆⋆) Give algorithms that compute these sets by traversing the given regular expression. ■

The *Glushkov automaton* M_r of a regular expression r is $(Q \cup \{q_I\}, \{q_I\}, F, \delta)$ with $q_I \notin Q$, where

$$\begin{array}{lcl} Q & = & \mathbf{pos}(r) \\ F & = & \begin{cases} \mathbf{last}(r) \cup \{q_I\} & \epsilon \in \mathcal{L}(r) \\ \mathbf{last}(r) & \epsilon \notin \mathcal{L}(r) \end{cases} \\ \delta & = & \{q_I \xrightarrow{b} b^{(j)} \mid b^{(j)} \in \mathbf{first}(r)\} \\ & & \cup \{a^{(i)} \xrightarrow{b} b^{(j)} \mid a^{(i)} \in \mathbf{pos}(r),\ b^{(j)} \in \mathbf{follow}(r, a^{(i)})\}. \end{array}$$

That is, the automaton M_r has r's elaborated labels as states in addition to a fresh state q_I constituting as an initial state. If M_r reads a label b in the state q_I then it takes one of r's first elaborated labels, $b^{(j)}$, as the state to which to transit. If M_r reads b in a state $a^{(i)}$ then as the next state it takes an elaborated label $b^{(j)}$ that follows after $a^{(i)}$. The automaton M_r halts at a last elaborated label $b^{(j)}$ or at q_I if r allows the empty sequence.

Example 14.1.10 Let $r_{14.1.10} = (a^{(1)*}b^{(2)*})^*$. Then we have that $\mathbf{pos}(r_{14.1.10}) = \mathbf{first}(r_{14.1.10}) = \mathbf{last}(r_{14.1.10}) = \mathbf{follow}(r_{14.1.10}, a^{(1)}) = \mathbf{follow}(r_{14.1.10}, b^{(2)}) = \{a^{(1)}, b^{(2)}\}$. The automaton $M_{r_{14.1.10}}$ is as follows:



■

The above procedure constructs an automaton in a way that preserves not only the language of the regular expression but also its ambiguity. More precisely, a regular expression is weakly unambiguous if and only if its Glushkov automaton is unambiguous. This property can easily be obtained from the fact that there is one-to-one correspondence between an elaborated string in the regular expression and an accepting run in the automaton.

Lemma 14.1.11 We have $a_1{}^{(i_1)} \cdots a_n{}^{(i_n)} \in \mathcal{L}(r)$ iff M_r has an accepting run $q_I a_1{}^{(i_1)} \cdots a_n{}^{(i_n)}$.

Proof: For $n = 0$, we obtain

$$\epsilon \in \mathcal{L}(r) \quad \Leftrightarrow \quad q_I \in F \quad \Leftrightarrow \quad M_r \text{ has an accepting run } q_I.$$

For $n \geq 1$, we obtain

$$\begin{aligned}
&a_1{}^{(i_1)} \cdots a_n{}^{(i_n)} \in \mathcal{L}(r) \\
&\Leftrightarrow a_1{}^{(i_1)} \in \mathbf{first}(r),\ a_n{}^{(i_n)} \in \mathbf{last}(r), \\
&\qquad \text{and } \forall j = 1, \ldots, n-1.\ a_{j+1}{}^{(i_{j+1})} \in \mathbf{follow}(r, a_j{}^{(i_j)}) \\
&\Leftrightarrow q_I \xrightarrow{a_1} a_1{}^{(i_1)},\ a_n{}^{(i_n)} \in F, \text{ and } \forall j = 1, \ldots, n-1.\ a_j{}^{(i_j)} \xrightarrow{a_{j+1}} a_{j+1}{}^{(i_{j+1})} \in \delta \\
&\Leftrightarrow M_r \text{ has an accepting run } q_I a_1{}^{(i_1)} \cdots a_n{}^{(i_n)}.
\end{aligned}$$

□

Corollary 14.1.12 The automaton M_r accepts s iff s *in* r holds. Moreover, r is weakly unambiguous iff M_r is unambiguous.

Example 14.1.13 The regular expression $r_{14.1.10}$ is weakly unambiguous and $M_{r_{14.1.10}}$ is also unambiguous. ■

Exercise 14.1.14 (⋆) Construct the Glushkov automaton of the regular expression $r_{14.1.4}$ and show that it is ambiguous. ■

By the property shown above we can reduce the weak ambiguity of a regular expression to the ambiguity of an automaton. Next, we reduce strong ambiguity to weak ambiguity.

A regular expression r is in *star normal form* (*SNF*) if, for every subexpression of r of the form d^*, the following condition (the SNF condition) holds:

$$\mathbf{follow}(d, \mathbf{last}(d)) \cap \mathbf{first}(d) = \emptyset$$

Here, we have generalized the definition of **follow** so that now $\mathbf{follow}(r, S) = \bigcup\{\mathbf{follow}(r, a^{(i)}) \mid a^{(i)} \in S\}$ for a set S of elaborated labels. The intuition behind the star normal form is that, when a subexpression d^* breaks the SNF condition, d itself is already a repetition and therefore enclosing it by the Kleene star * makes it ambiguous.

Example 14.1.15 The regular expression $(a^{(1)}b^{(2)*})^*$ is in star normal form since $\mathbf{follow}(a^{(1)}b^{(2)*}, \mathbf{last}(a^{(1)}b^{(2)*})) = \{b^{(2)}\}$ and $\mathbf{first}(a^{(1)}b^{(2)*}) = \{a^{(1)}\}$ are disjoint. The regular expression $(a^{(1)*}b^{(2)*})^*$ is not in star normal form since $\mathbf{follow}(a^{(1)*}b^{(2)*}, \mathbf{last}(a^{(1)*}b^{(2)*})) = \{a^{(1)}, b^{(2)}\}$ and $\mathbf{first}(a^{(1)*}b^{(2)*}) = \{a^{(1)}, b^{(2)}\}$ are overlapping. ■

A regular expression r is in *epsilon normal form* (ϵ-*NF*) if, for every subexpression d of r, there is at most one derivation of the relation ϵ *in* d (the ϵ-NF condition).

Example 14.1.16 The regular expression $a^{(1)*} \mid b^{(2)*}$ is not in epsilon normal form. ■

Having these two definitions of normal forms, we can make the following connection between strong and weak ambiguities.

Theorem 14.1.17 (Brüggemann-Klein, 1993) A regular expression r is strongly unambiguous if and only if r is weakly unambiguous in star normal form and in epsilon normal form.

Proof: We first prove the "if" direction by induction on the structure of r. The cases $r = \epsilon$ and $r = a^{(i)}$ are trivial.

Case 1: $r = r_1 r_2$

Suppose that r is strongly ambiguous. Then there are two derivations of the relation $s \; in \; r$ for a string s. Since r_1 and r_2 are strongly unambiguous by the induction hypothesis, the ambiguity arises only in how we divide s. That is, there are s_1, s_2, s_3 such that $s = s_1 s_2 s_3$ and $s_2 \neq \epsilon$ and the following relations hold:

$$\begin{array}{ll} s_1 s_2 \; in \; r_1 & s_3 \; in \; r_2 \\ s_1 \; in \; r_1 & s_2 s_3 \; in \; r_2. \end{array}$$

Therefore there are elaborations $\tilde{s}_1$ and $\tilde{s}'_1$ of s_1, $\tilde{s}_2$ and $\tilde{s}'_2$ of s_2, and $\tilde{s}_3$ and $\tilde{s}'_3$ of s_3 such that

$$\begin{array}{ll} \tilde{s}_1 \tilde{s}_2 \in \mathcal{L}(r_1) & \tilde{s}_3 \in \mathcal{L}(r_2) \\ \tilde{s}'_1 \in \mathcal{L}(r_1) & \tilde{s}'_2 \tilde{s}'_3 \in \mathcal{L}(r_2). \end{array}$$

Since $\tilde{s}_2$ comes from r_1 and $\tilde{s}'_2$ from r_2, these must be elaborated differently: $\tilde{s}_2 \neq \tilde{s}'_2$. This implies that in r there are different elaborations $\tilde{s}_1 \tilde{s}_2 \tilde{s}_3$ and $\tilde{s}'_1 \tilde{s}'_2 \tilde{s}'_3$ of s, contradicting the assumption that r is weakly unambiguous.

Case 2: $r = r_1 \mid r_2$

As in the last case, suppose that r is strongly ambiguous. Then there are two derivations of $s \; in \; r$ for a string s. Since r_1 and r_2 are strongly unambiguous by the induction hypothesis, the ambiguity arises only in which choice to take. That is, we have both

$$s \; in \; r_1 \quad s \; in \; r_2.$$

Therefore there are elaborations $\tilde{s}$ and $\tilde{s}'$ of s such that

$$\tilde{s} \in \mathcal{L}(r_1) \quad \tilde{s}' \in \mathcal{L}(r_2).$$

Since r is in epsilon normal form, $s \neq \epsilon$ and therefore $\tilde{s} \neq \tilde{s}'$. Moreover, $\tilde{s}$ and $\tilde{s}'$ are distinct since these come from r_1 and r_2, respectively. This implies that in r there are two different elaborations of s, again contradicting the assumption that r is weakly unambiguous.

Case 3: $r = {r_1}^*$

As in the cases above, suppose that r is strongly ambiguous. Then there are two derivations of $s \; in \; r$ for a string s. Since r_1 is strongly unambiguous by the induction hypothesis, the ambiguity arises only in how we divide s. That is, there are $s_1, \ldots, s_n, s'_1, \ldots, s'_m$ such that $s = s_1 \cdots s_n = s'_1 \cdots s'_m$, where

$$\begin{array}{ll} s_i \; in \; r_1, & i = 1, \ldots, n \\ s'_i \; in \; r_1, & i = 1, \ldots, m \\ (s_1, \ldots, s_n) \neq (s'_1, \ldots, s'_m). & \end{array}$$

No s_i or s'_i equals ϵ since otherwise r_1 would generate ϵ and therefore obviously r would not be in epsilon normal form. Thus there are elaborations $\tilde{s}_i$ of s_i and $\tilde{s}'_i$ of s'_i such that

$$\begin{aligned} \tilde{s}_i &\in \mathcal{L}(r_1) \qquad i = 1, \ldots, n \\ \tilde{s}'_i &\in \mathcal{L}(r_1) \qquad i = 1, \ldots, m. \end{aligned}$$

From the fact that $(s_1, \ldots, s_n) \neq (s'_1, \ldots, s'_m)$, we have $\tilde{s}_1 = \tilde{s}'_1, \ldots, \tilde{s}_{k-1} = \tilde{s}'_{k-1}$ and $\tilde{s}_k \neq \tilde{s}'_k$ for some k. Without loss of generality, we can assume that $\tilde{s}_k = \tilde{s}'_k \tilde{s}''$ with $\tilde{s}'' \neq \epsilon$. Let l be the last elaborated label of $\tilde{s}'_k$ and f be the first elaborated label of $\tilde{s}''$. Noting that $l \in \mathbf{last}(r_1)$ and $f \in \mathbf{first}(r_1)$, we conclude that $\tilde{s}_k = \tilde{s}'_k \tilde{s}''$ implies $f \in \mathbf{follow}(r_1, l)$, that is, ${r_1}^*$ does not satisfy the SNF condition. This contradicts that r is in the star normal form.

We next prove the converse. By Proposition 14.1.5 we need only to show that if r is strongly unambiguous then r is both in star normal form and in epsilon normal form. Suppose that r is strongly unambiguous but not in one of these forms. Then there is a subexpression d that breaks either the SNF condition or the ϵ-NF condition. To reach the contradiction that r is strongly ambiguous, it suffices to show that d is strongly ambiguous. In the case where d breaks the ϵ-NF condition, d is trivially strongly ambiguous by definition. In the case where $d = {d'}^*$ breaks the SNF condition then

$$b^{(j)} \in \mathbf{follow}(d', a^{(i)})$$

for some $a^{(i)} \in \mathbf{last}(d')$ and $b^{(j)} \in \mathbf{first}(d')$. That is, there are elaborated strings $\tilde{s}a^{(i)}$ and $b^{(j)}\tilde{s}'$ in $\mathcal{L}(d')$ for some $\tilde{s}, \tilde{s}'$. These imply that the Glushkov automaton $M_{d'}$ has a run $q_I \tilde{s} a^{(i)}$ and another run $q_I b^{(j)} \tilde{s}'$, with q_I an initial state, as well as a transition from $a^{(i)}$ to $b^{(j)}$. Thus the automaton must also have a run $q_I \tilde{s} a^{(i)} b^{(j)} \tilde{s}'$, that is, $\tilde{s}a^{(i)}b^{(j)}\tilde{s}'$ is in $\mathcal{L}(d')$. From these, there are at least two derivations of the relation $sabs'$ *in* ${d'}^*$, where $\mathbf{unelab}(\tilde{s}) = s$ and $\mathbf{unelab}(\tilde{s}') = s'$:

$$\dfrac{\dfrac{\vdots}{sa \text{ in } d' \quad bs' \text{ in } d'}}{sabs' \text{ in } {d'}^*} \qquad \dfrac{\dfrac{\vdots}{sabs' \text{ in } d'}}{sabs' \text{ in } {d'}^*}$$

Hence d is strongly ambiguous. □

Star normal form and epsilon normal form are easy to check. Thus, by Theorem 14.1.17, we can reduce the check for strong ambiguity to that for weak ambiguity.

Regular expressions whose corresponding Glushkov automata are deterministic are called *one-unambiguous*. The DTD and XML Schema languages require

content models to be one-unambiguous so that they can easily be implemented efficiently.

14.1.3 Ambiguity checking for automata

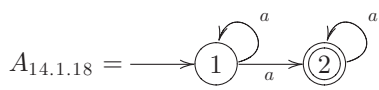
Given an automaton $A = (Q, I, F, \delta)$, the following procedure checks whether A is ambiguous.

1. Take the self-product B of A, that is, $B = A \times A = (Q \times Q, I \times I, F \times F, \delta')$, where

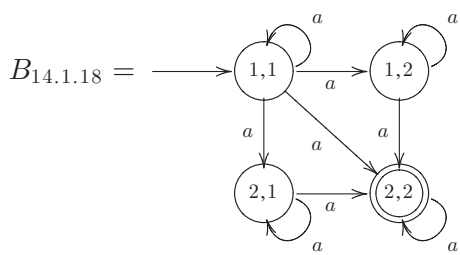
$$\delta' = \{(q, r) \to a((q_1, r_1), (q_2, r_2)) \mid q \to a(q_1, q_2),\ r \to a(r_1, r_2) \in \delta\}.$$

2. Obtain the automaton C that remains after elimination of the useless states from B. (See Section 7.3.1 for useless state elimination.)
3. The answer is "unambiguous" iff every state of C has the form (q, q).

Example 14.1.18 From the automaton

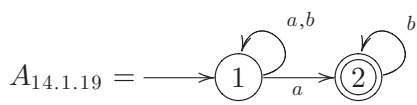


the first step yields the following:

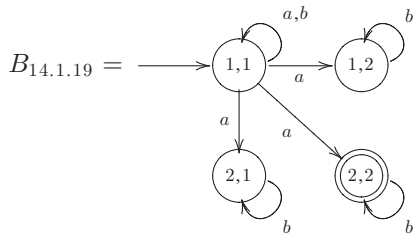


(Note that the diagram of a self-product is always symmetric.) The second step returns the same automaton: $C_{14.1.18} = B_{14.1.18}$. Thus, $A_{14.1.18}$ is ambiguous since $C_{14.1.18}$ has the states (1, 2) and (2, 1). ■

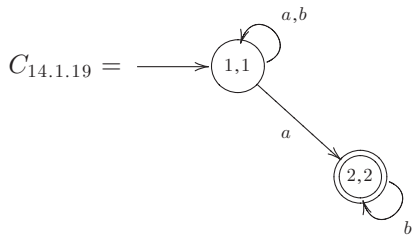
Example 14.1.19 From the automaton



the first step yields the following:



The second step eliminates the states (1, 2) and (2, 1), resulting in the following:



Thus, $A_{14.1.19}$ is unambiguous since $C_{14.1.19}$ has only states of the form (q, q). ■

Theorem 14.1.20 The ambiguity-checking algorithm returns "unambiguous" for an automaton A if and only if A is unambiguous.

Proof: Suppose that the automaton A is ambiguous. Then, there are two runs

$$q_1 q_2 \cdots q_n \qquad \text{and} \qquad r_1 r_2 \cdots r_n$$

both accepting a string s, where $q_i \neq r_i$ for some $1 \leq i \leq n$. Therefore some automaton B also has a run

$$(q_1, r_1)(q_2, r_2) \cdots (q_n, r_n)$$

accepting s. Since (q_i, r_i) is in this run, this state remains in the automaton C. Thus, the algorithm returns "ambiguous."

Conversely, suppose that the algorithm returns "ambiguous." Then there is a state (q, r) in C with $q \neq r$. Since this state remains in C, the state (q, r) is reachable from an initial state and a final state is reachable from that state. In other words, there is a run accepting a string s. Thus A has two different runs accepting s, one of which passes through q and the other through r. Therefore A is ambiguous. □

14.2 Ambiguity for patterns

So far, we have considered two definitions of ambiguity for plain regular expressions. However, these definitions can be too restrictive if we are interested in bindings of patterns. That is, it would be acceptable that a pattern (in the nondeterministic semantics) yields a unique binding for any input even if its underlying regular expression is weakly or strongly ambiguous. For example, consider the following pattern:

$$((a^*)^* \text{ as } x)$$

(In this section, we consider patterns as defined in Chapter 5, where labels do not have contents.) The regular expression after removing the binder is strongly ambiguous. However, obviously the pattern itself yields a unique binding for any input – x is always bound to the whole sequence. Permitting such a harmless ambiguity can be important in practice. For example, for convenience in programming we may want to incorporate a part T of an externally provided schema as a pattern subexpression, as follows:

$$(T \text{ as } x)$$

However, we may not have any control on the schema from the programmer's side. In such a case it would be unreasonable to reject this pattern on the ground that T is ambiguous.

So let us call a pattern *binding-unambiguous* when it yields at most one binding for any input. However, there is one tricky point in this definition. Should we regard a pattern such as

$$((a \text{ as } x), a) \mid (a, (a \text{ as } x))$$

as ambiguous? This pattern matches only the string aa and yields only one binding, $x \mapsto a$. However, the a in the binding can come from either the first label or the second in the input. If we take the "structural view" that these two labels are the same then the above pattern is unambiguous; if we take the "physical view" that these are distinct then the pattern is ambiguous. In this section we treat binding ambiguity only in the physical view since this is algorithmically simpler and tricky cases like the above rarely occur in practice.

14.2.1 Definitions

Let us formalize "physical" binding ambiguity, first for patterns and then for marking string automata. For patterns we need to define a semantics that takes positions in the input and output strings into account. That is, we define a matching

relation $\tilde{s} \in P \Rightarrow \tilde{V}$, where $\tilde{s}$ is an elaborated string, P is a pattern (with no contents in any label), and $\tilde{V}$ is a mapping from variables to elaborated strings. We assume that each label in elaborated strings is given a unique integer. The matching relation is defined by the same set of rules as in Section 5.4.1 (without involving pattern definitions) except that the rule P-ELM is replaced by the following:

$$\frac{}{a^{(i)} \in a \Rightarrow \epsilon}$$

Then, a pattern P is *binding-unambiguous* if $\tilde{s} \in P \Rightarrow \tilde{V}$ and $\tilde{s} \in P \Rightarrow \tilde{V}'$ imply that $\tilde{V} = \tilde{V}'$ for any $\tilde{s}$.

Example 14.2.1 The pattern $((a^*)^* \text{ as } x)$ is binding-unambiguous since it matches any elaborated string of the form $a^{(1)}a^{(2)} \cdots a^{(n)}$, from which the only possible binding is $\{x \mapsto a^{(1)}a^{(2)} \cdots a^{(n)}\}$. The pattern $((a \text{ as } x), a) \mid (a, (a \text{ as } x))$ is binding-ambiguous since it matches only the elaborated string $a^{(1)}a^{(2)}$, from which two bindings are possible: $\{x \mapsto a^{(1)}\}$ and $\{x \mapsto a^{(2)}\}$. ■

For marking string automata (whose definition is identical to those on marking tree automata except that each transition has only one destination state), we define ambiguity in terms of markings yielded from them. A marking string automaton A is *marking-unambiguous* if $m = m'$ whenever A yields successful marking runs (r, m) and (r', m') on a string s. Noting that obtaining a binding of variables to elaborated substrings has the same effect as marking substrings with variables, we can easily prove the following.

Proposition 14.2.2 Let P be a pattern and A be a marking string automaton constructed from P. Then, P is binding-unambiguous iff A is marking-unambiguous.

Note that the proposition does not specify how the marking string automaton is constructed. Indeed, this property holds regardless of which automata construction is used.

14.2.2 Algorithm for checking marking ambiguity

Given a marking string automaton $A = (Q, I, F, \delta, \mathcal{X})$, the following algorithm checks whether A is marking-ambiguous.

1. Calculate $B = (Q \times Q, I \times I, F', \delta', \mathcal{X} \times \mathcal{X})$ where F' and I' are as follows:

$$\begin{aligned} F' &= \{((q, r), \mathbf{x} \times \mathbf{x}') \mid (q, \mathbf{x}), (r, \mathbf{x}') \in F\} \\ \delta' &= \{(q_1, r_1) \xrightarrow{\mathbf{x}\times\mathbf{x}':a} (q_2, r_2) \mid q_1 \xrightarrow{\mathbf{x}:a} q_2,\ r_1 \xrightarrow{\mathbf{x}':a} r_2 \in \delta\} \end{aligned}$$

2. Obtain the automaton C after eliminating the useless states from B.

3. The answer is "marking-unambiguous" iff we have $\mathbf{x} = \mathbf{x}'$ for each $(q_1, r_1) \xrightarrow{\mathbf{x}\times\mathbf{x}':a} (q_2, r_2) \in \delta$ and for each $((q, r), \mathbf{x} \times \mathbf{x}') \in F$.

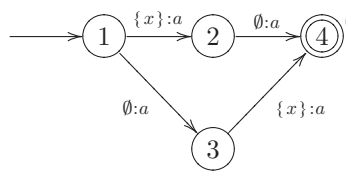
The algorithm works in a quite similar way to the ambiguity checking algorithm given in Section 14.1.3 except that it checks how the automaton marks each node with variables rather than how it assigns states to each node. More precisely, the last step above ensures that, for any two runs accepting the same string, the same set of variables is assigned to each position. Thus, as expected, the following property, which is similar to Theorem 14.1.20, can be proved.

Theorem 14.2.3 The marking-ambiguity checking algorithm returns "marking-unambiguous" for an automaton A iff A is marking-unambiguous.

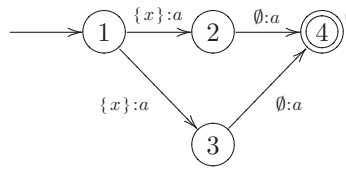
Exercise 14.2.4 (⋆⋆) Prove Theorem 14.2.3. ■

Exercise 14.2.5 (⋆) Apply the above algorithm to the following marking string automata to check whether each is marking-ambiguous.

(A)



(B)



■

14.3 Bibliographic notes

Ambiguity for string regular expressions and automata has been a classical question (Book *et al.*, 1971; Brüggemann-Klein, 1993). The terminology of strong and weak ambiguities with a relationship based on Glushkov automata and the star normal form was introduced in Brüggemann-Klein (1993). In the same paper, a checking procedure for strong ambiguity is given as a reduction to an ambiguity-checking algorithm for LR(0) grammars. The ambiguity-checking algorithm for automata presented in this chapter has its origin in folklore. For tree grammars, there are algorithms for weak ambiguity (Kawaguchi, 2001) and for strong ambiguity

(Hosoya, 2003). A study of one-unambiguous regular expressions can be found in Brüggemann-Klein and Wood (1998). Marking ambiguity was first introduced as a way to ensure the uniqueness of solutions to a certain type of inference problem arising in parametric polymorphism for XML (Hosoya *et al.*, 2009). Although our purpose was different from that in the present chapter, the definition was the same.

15

Unorderedness

The previous chapters have focused on the element structure of XML, where ordering is significant. However, in reality XML also requires unordered structures to be treated. These arise from two directions, namely, from XML attributes and from shuffle expressions. Attributes are auxiliary data associated with elements and are intrinsically unordered in the data model, whereas shuffle expressions introduce unorderedness into the ordered data model of elements. Unorderedness is a rather unexplored area since dealing with it is surprisingly tricky. In this chapter we review briefly some previous proposals for description mechanisms and relevant algorithmics.

15.1 Attributes

As mentioned in passing in Section 3.1, attributes are a label–string mapping associated with each element and used in almost all applications of XML. Recall the following example:

```
<person age="35" nationality="japanese">
 ⋮
</person>
```

Here, the `person` element is associated with two attributes, `age` and `nationality`. Attributes are different from elements, in that attributes have no ordering and cannot repeat the same label within the same element. Thus, the above example is considered to be identical to

```
<person nationality="japanese" age="35">
 ⋮
</person>
```

and the following,

```
<person nationality="japanese" nationality="french"
                                                age="35">
 ⋮
</person>
```

is an error.

What constraints do we want to impose on attributes and how can we describe them? First, it is clearly necessary to associate each attribute label with its value type. For this we can introduce a notation such as the following:

```
person[@nationality[String], @age[String], ...]
```

where label–type pairs are listed in front of the content type of `person`. (We may also want to specify attribute values more precisely, for which we could introduce types denoting some subsets of all strings, e.g., positive integers, but here we will not go further in this direction.) Next, it would be useful to express some more "structural" constraints, such as the following:

Optional attributes For example, we might want an attribute `nationality` to be present always but `age` possibly to be omitted.

Choice of attributes For example, we might want either an `age` or a `date` to be present. A more complicated example would require either just an `age` or all of a `year`, a `month`, and a `day` at the same time.

Open attributes An arbitrary number of arbitrary attributes from a given range of labels is allowed. For example, we might want to allow attributes other than `age` or `nationality`. In real applications, we often want to allow arbitrary attributes not in a certain name space (see Section 3.1 for name spaces).

How can we describe these constraints? For the first two, we are already able to express them thanks to choice types. For example, "a mandatory `nationality` and an optional `age`" can be described as

```
  person[@nationality[String], @age[String], ...]
| person[@nationality[String], ...]
```

and "either just an `age` or all of a `year`, a `month`, and a `day`" as:

```
  person[@age[String], ...]
| person[@year[String], @month[String], @day[String],
                                                ...]
```

At this stage we notice that repeating the element label `person` is rather awkward. Moreover, if both `nationality` and `age` are optional then we need to list four combinations; if there are k optional attributes then there are 2^k combinations. So, in

order to make our description more concise, we can introduce regular-expression-like notations for attributes. For example, the above two examples of types are equivalently written as

```
person[@nationality[String], @age[String]?, ...]
```

and

```
person[(@age[String] |
        @year[String], @month[String],
                       @day[String]), ...]
```

respectively. What about openness? Since we have no way to say "arbitrary number of" or "arbitrary label from," let us add special notations for these. First, we introduce expressions denoting sets of labels. Namely, we write `~` for all labels and `a` for a singleton set of label `a`; for label sets A_1 and A_2 we write $(A_1|A_2)$ as their union and $(A_1 \setminus A_2)$ as their difference. Then, we write $@A[T]^*$ for a type denoting an arbitrary number of arbitrary attributes from a label set A with value type T. Thus, the above example of openness can be expressed as follows:

```
person[@(~\age\nationality)[String]*, ...]
```

Note that the data model requires that the same attribute label is not repeated.

So far, we have introduced quite rich notation for constraining attributes. However, it occasionally happens that we want to mix constraints on attributes and those on elements. For example, the following are sometimes useful.

Attribute-dependent element types For example, we might want an element `person` to contain a sub-element `visa-id` if the `person` has an attribute `nationality` with a value other than `japanese`.

Attribute-element choices For example, we might want a `nationality` and an `age` to be given either both as attributes or both as elements.

In order to express attribute-dependent element types, we can simply use choice types, for example,

```
  person[@nationality["japanese"], ...]
| person[@nationality[^"japanese"],
             visa-id[String], ...]
```

(Here, assume that the type `^string` denotes the set of strings excluding `string`.) However, to express a choice that embraces either an attribute or an element, at this moment we have only a way that potentially incurs a blow-up. For example, if we want such a choice for both `nationality` and `age`, we need to enumerate four combinations:

```
  person[@nationality[String], @age[String], ...]
| person[@age[String], nationality[String], ...]
| person[@nationality[String], age[String], ...]
| person[nationality[String], age[String], ...]
```

A solution to this problem is *attribute-element constraints*, adopted in the schema language RELAX NG. In this, we can mix both attribute and element constraints in the same expression. For example, the above can be rewritten as follows:

```
person[(@nationality[String] | nationality[String]),
       (@age[String] | age[String]), ...]
```

Note that each expression constrains a *pair* comprising an attribute set and an element sequence.

To formalize attribute-element constraints, we first extend the syntax of types in Section 3.2 to include attribute types:

$$
\begin{array}{rcll}
T & ::= & \vdots & \\
 & & @a[T] & \text{attribute} \\
 & & @A[T]^* & \text{attribute repetition}
\end{array}
$$

We require that, in any concatenation type T_1, T_2, no common attribute label appears in the top levels of both T_1 and T_2 and that, in any repetition T^*, no attribute type appears in the top level of T except for the special attribute repetition form $@A[T]^*$. Then, the semantics is described by a conformance relation $E \vdash v \in T$ that is slightly modified from the one in Section 3.2. First, a value v is defined as a pair $\langle \alpha; \beta \rangle$ of

- a set α of attributes each of the form $@a[v]$, and
- a sequence β of elements each of the form $a[v]$.

Then, the rules T-Eps, T-Elm, T-Cat, and T-Rep are replaced by those below, while the others are retained:

$$\frac{}{E \vdash \langle \emptyset; () \rangle \in ()} \text{ TA-Eps}$$

$$\frac{E \vdash v \in T}{E \vdash \langle \emptyset; a[v] \rangle \in a[T]} \text{ TA-Elm}$$

$$\frac{E \vdash \langle \alpha_1; \beta_1 \rangle \in T_1 \qquad E \vdash \langle \alpha_2; \beta_2 \rangle \in T_2}{E \vdash \langle \alpha_1 \cup \alpha_2; \beta_1, \beta_2 \rangle \in T_1, T_2} \text{ TA-Cat}$$

$$\frac{E \vdash \langle \emptyset; \beta_i \rangle \in T \qquad \forall i = 1, \ldots, n \qquad n \geq 0}{E \vdash \langle \emptyset; \beta_1, \ldots, \beta_n \rangle \in T^*} \text{ TA-Rep}$$

Furthermore, the following new rules are added:

$$\frac{E \vdash v \in T}{E \vdash \langle \{@a[v]\}, () \rangle \in @a[T]} \text{ TA-Att}$$

$$\frac{E \vdash v_i \in T \qquad \forall i = 1, \ldots, n \qquad \{a_1, \ldots, a_n\} \subseteq A \qquad n \geq 0}{E \vdash \langle \{@a_1[v_1], \ldots, @a_n[v_n]\}, () \rangle \in @A[T]^*} \text{ TA-AttRep}$$

15.2 Shuffle expressions

Another kind of unorderedness comes from the requirement to regard some ordering among elements as insignificant. Typically, this kind of demand arises when one wants to define data-oriented documents, where a sequence is often taken as a set or a record.

If one wants to disregard all ordering among elements then it would be sensible to take an unordered data model in the first place and equip a completely different, non-regular-expression-based, schema language. However, the reality is more complicated. For one thing, sometimes parts of a document are required to be ordered but other parts of the same document unordered; for example, in a document representing a "stream of records," the topmost sequence would be ordered whereas each sequence in the next level would be unordered. For another thing, sometimes a single schema language is required for all purposes, rather than different schema languages for different purposes. *Shuffle expressions* have arisen as a compromise, by which any permutation of elements can be allowed for specified parts of documents. Note here that the data model is still ordered but an intended meaning can be unordered.

Let us first introduce some notation. First, we add shuffle types of the form $T_1 \&\& T_2$ to the syntax of types. The meaning of this expression is the set of all "interleaves" of values from T_1 and ones from T_2. More formally, we add the following rule to the semantics of types:

$$\frac{E \vdash v_1 \in T_1 \qquad E \vdash v_2 \in T_2 \qquad v \in \textbf{interleave}(v_1; v_2)}{E \vdash v \in T_1 \&\& T_2} \text{ T-Shu}$$

Here, $\textbf{interleave}(v_1; v_2)$ is the set of values v such that v contains v_1 as a (possibly non-consecutive) subsequence and taking away v_1 from v yields v_2. For example, $\textbf{interleave}((a[], b[]); (c[], d[]))$ contains all the following values:

$$a[], b[], c[], d[] \quad a[], c[], b[], d[] \quad a[], c[], d[], b[]$$
$$c[], a[], b[], d[] \quad c[], a[], d[], b[] \quad c[], d[], a[], b[]$$

Note that the operator && is commutative. The following is an example of a schema using shuffle types:

```
List    = list[Record*]
Record  = record[Name && Address? && Email*]
Name    = name[first[String], last[String]]
Address = address[String]
Email   = email[String]
```

In this, the top element, `list`, contains an ordered sequence of `record` elements, each of which contains an unordered list of a mandatory `name`, an optional `address`, and any number of `email`s, where the `name` element contains an ordered pair.

It is important to note that shuffle expressions by themselves do not increase the expressive power of the schema model but allow an exponentially more concise description. For example, the type `a[] && b[] && c[]` can be rewritten as the non-shuffle type

```
  a[],b[],c[],d[] | a[],c[],b[],d[] | a[],c[],d[],b[]
| c[],a[],b[],d[] | c[],a[],d[],b[] | c[],d[],a[],b[]
```

leading to a blow-up. The type `Name && Address? && Email*` appearing in the above schema example can also be rewritten as the following rather awkward type:

```
    Email*, Name, Email*, Address?, Email*
  | Email*, Address?, Email*, Name, Email*
```

In general shuffle expressions represent no more than regular languages since they can easily be converted to automata, as discussed in the next section.

We have seen that shuffle expressions are quite powerful. However, their full power would not necessarily be of use while allowing it would make algorithmics difficult. Therefore some schema languages adopt shuffle expressions with certain syntactic restrictions. The most typical is to require that, in a shuffle type $T_1 \&\& T_2$, the sets of labels appearing in the top levels of T_1 and T_2 are disjoint. For example, the `record` type in the above obeys this restriction, since the types `Name`, `Address?` and `Email*` have disjoint sets of top-level labels. However, a type like `(Name,Tel?) && (Name,Email*)` is disallowed. This restriction can make the algorithmics easier to some extent, since then a certain kind of nondeterminism can be avoided in shuffle automata (to be described below). The RELAX NG schema language adopts this restriction.

Another restriction that is often used in conjunction with the previous one is to require that, in a shuffle type $T_1 \&\& T_2$, each T_1 or T_2 denotes a sequence of

length at most 1. For example, a type like `(Name,Address?) && Email` is disallowed. Also, the above example of a `list` of `record`s is rejected since a component of a shuffle type is `Email*`, which can arbitrarily repeat the element `email`. Nevertheless, even with this restriction most common record-like data, which simply bundle together a set of fields of different names (that are possibly optional), can be represented. Moreover the algorithmics can be made easier since a sequence described by a shuffle type can be taken simply as a set (rather than a complicated interleaving of sequences). Note that, with a further restriction that disallows a choice between or the concatenation of a shuffle type and some other type, or the repetition of a shuffle type, we can clearly separate unordered and ordered sequences with no mixture; XML Schema adopts this restriction.

15.3 Algorithmic techniques

As we have seen, both attribute-element constraints and shuffle expressions can be converted to more elementary forms of type. Thereby, relevant algorithmic problems such as the membership test and basic set operations can be solved by using existing algorithms (possibly with a small modification). However, this easily causes a combinatorial explosion. Although some efforts have been made to deal with this difficulty, so far none has been completely satisfactory. Nevertheless, discussing some of them is worthwhile since at least they could provide good starting points.

Attribute-element automata

A straightforward way of representing an attribute-element constraint is to use a tree automaton with two kinds of transitions: *element transitions* (the usual kind) and *attribute transitions*. Such an automaton accepts a pair consisting of an element sequence and an attribute set: when the automaton follows an element transition it consumes an element from the head of the sequence; when it follows an attribute transition it consumes an attribute from the set.

It is known how to adapt the on-the-fly membership algorithms (Section 8.1) for attribute-element automata. However, it is rather tricky to adapt such an algorithm to perform other set operations such as intersection and complement, because the ordering among attribute transitions can be exchanged and, in general, all permutations have to be generated for performing set operations. This could be improved by a "divide and conquer" technique in the special cases that actually arise in practice. That is, if a given automaton can be divided as a concatenation of independent sub-automata, (i.e., whose sets of top-level labels are disjoint), then a certain kind of analysis can be carried out separately on these sub-automata. For example, suppose that two automata M and N are given and that each can be

divided as a concatenation of sub-automata, say M_1 and M_2 from M and N_1 and N_2 from N, such that both M_1 and N_1 have label set A_1 and both M_2 and N_2 have label set A_2. Then the intersection of M and N can be computed by taking that of M_1 and N_1, taking that of M_2 and N_2, and then reconcatenating the resulting automata. This technique can also be used for computing a complement.

Shuffle automata

A well-known automata-encoding of a shuffle expression $T_1 \&\& T_2$ equips two automata M_1 and M_2 corresponding to T_1 and T_2 and runs them in an interleaving way. That is, in each step of reading the input sequence, either M_1 or M_2 reads the head label; at the end of the sequence, both M_1 and M_2 must be in final states. (This implies that a single automaton can be constructed from the interleaving of two automata by a product construction. Compare this with the product construction used for computing the intersection in Section 4.4.1, where two automata run in parallel but simultaneously read the head label in each step.) Note that, by the "disjoint label sets" restriction on shuffle expressions, we can decide uniquely which of M_1 or M_2 reads each head label. With the "at most length 1" restriction, a shuffle automaton becomes trivial since it is now a list of normal automata each with at most one transition, which can be seen simply as a set of occurrence flags. Further, with the disallowance of choice, concatenation, or repetition of a shuffle, we can avoid a mixture of ordinary and shuffle automata; the behavior of such a mixture would be extremely complicated. Thus, while these restrictions provide an option that is attractive from the algorithmic point of view, the schema model quickly loses closure properties that are useful in various analyses including static typechecking.

15.4 Bibliographic notes

As already mentioned, RELAX NG allows both attribute-element constraints and shuffle expressions (Clark and Murata, 2001); XML Schema provides a restricted version of shuffles (Fallside, 2001). Although XML's document type definition (DTD) does not have shuffles, SGML's DTD (Sperberg-McQueen and Burnard, 1994) provides a weak notion of the shuffle operator `&`, where `(a,b)&(c,d)` means just the choice of `(a,b,c,d)` and `(c,d,a,b)` with no other interleaving. Reasoning for this operator is usually rather unclean.

The algorithmic aspects of attribute-element constraints have been studied in several papers. A membership (validation) algorithm based on attribute-element automata is given in Murata and Hosoya (2003); a similar algorithm was tested in the context of bidirectional XML transformation (Kido, 2007). Another algorithm based on the "partial derivatives" of regular expressions (Brzozowski, 1964) was

proposed in Clark (2002); this algorithm also treats shuffle expressions. Algorithms for basic set operations based on divide-and-conquer are studied in Hosoya and Murata (2006), though these algorithms operate directly on constraint expressions rather than automata for clarity of presentation.

Shuffle automata have long been known and are formalized in, for example, Jedrzejowicz (1999), though they have rarely been used in the context of XML. Although shuffle expressions are rather difficult to deal with, some possible approaches have recently been proposed. A group of researchers has found a set of rather drastic syntactic restrictions on regular expressions containing shuffles, with which a large class of schemas used in practice is expressible while many important set operations such as membership and containment can be performed efficiently (Colazzo *et al.*, 2009; Ghelli *et al.*, 2008). In separate work it has been shown that a different set of syntactic restrictions allows the encoding of regular expressions by zero-suppressed binary decision diagrams (ZBDDs) and thereby set operations such as containment can be computed efficiently (Tozawa, 2004). Also, a completely different approach using Presburger's arithmetics has been proposed; this can be used for describing constraints on the counts of element labels (Zilio and Lugiez, 2003). Such counting constraints have their own source of exponential blow-up, though an attempt has been made at a practical implementation (Foster *et al.*, 2007).

Appendix
Solutions to selected exercises

3.2.1 For brevity, we assume that all the strings have been checked against `String`. Also, we implicitly use T-NAME below.

By T-ELM, all `name` nodes have type `Name`. Also, by T-EPS and T-ELM, `male[]` has type `Male` and `female[]` has type `Female`. Further, by T-ALT1, T-ALT2, and T-ELM:

$$\texttt{gender[male[]]} \in \texttt{Gender} \qquad \texttt{gender[female[]]} \in \texttt{Gender}.$$

Then, by T-CAT, all occurrences of the form `name[...],gender[...]` can be given type `Name,Gender`. Thus, by T-ELM, the part `spouse[...]` is given type `Spouse`. Noting that $T? \equiv T \mid ()$, T-EPS, T-ALT2, and T-CAT give:

$$\texttt{()} \in \texttt{Spouse?,Children?}$$

Hence, by T-CAT, each content of the last two `person` nodes is given type as follows:

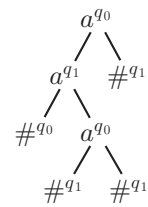
$$\begin{aligned}\texttt{name[...],gender[...]} &= \texttt{name[...],gender[...],()} \\ &\in \texttt{Name,Gender,Spouse?,Children?}\end{aligned}$$

Thus, by T-ELM, the two `person` nodes are given type `Person`. By T-REP, the sequence of these two `person` nodes has type `Person*` and therefore, by T-ELM, the `children` node is given type `Children`. Noting again that $T? \equiv T \mid ()$, T-ALT1 and T-CAT ensures that a tail part in the root `person` can be given type as follows:

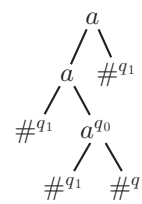
$$\texttt{spouse[...],children[...]} \in \texttt{Spouse?,Children?}$$

Finally, by T-CAT and T-ELM, the root `person` node is given type `Person`.

4.1.4 The following unique run assigns a non-final q_0 to the leftmost leaf (at position 111), which is not successful:

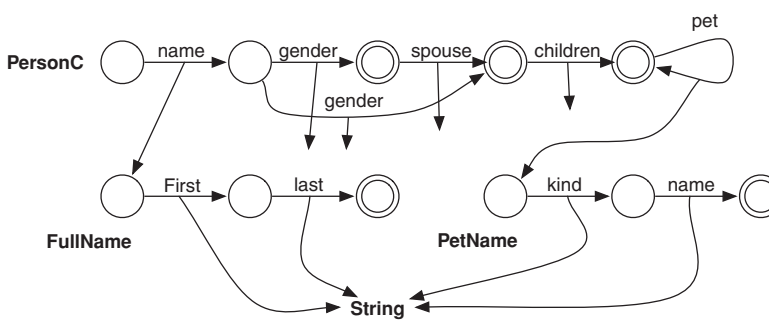


4.1.6 No. The following is an attempt to assign a state to each node in a bottom-up way, which fails for the node at position 11:

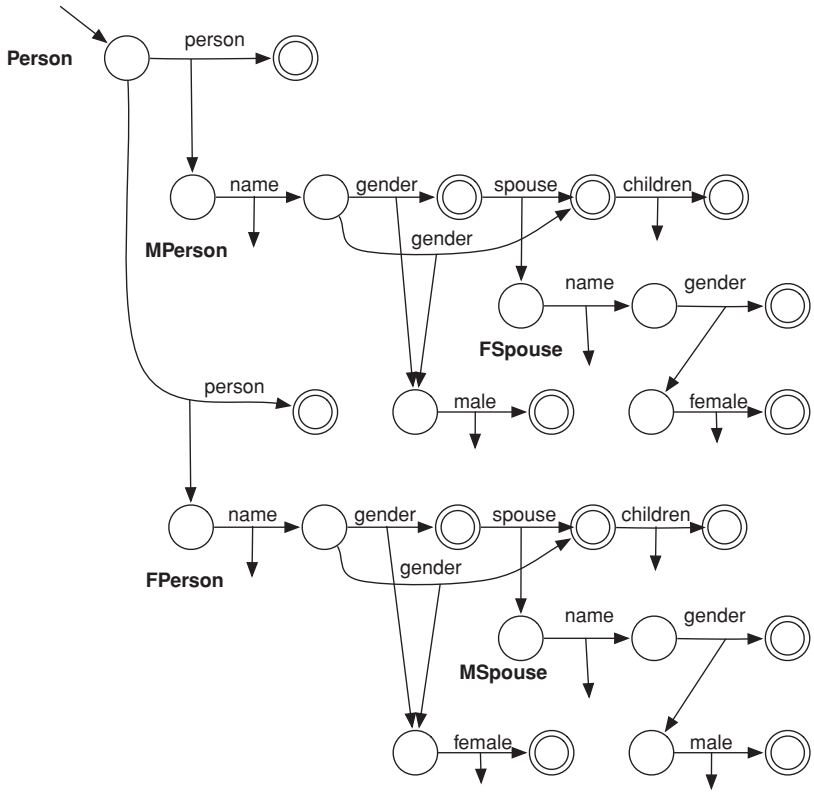


4.1.10 The ϵ-elimination is $(\{q_0, q_1\}, \{q_0\}, \{q_0, q_1\}, \{q_0 \to a(q_0, q_0),\ q_0 \to b(q_1, q_1),\ q_1 \to b(q_1, q_1)\})$.

4.2.1 For the first extended schema, replace, in Figure 4.2, the part for `PersonC` with the following:



Likewise, for the second, duplicate and modify the parts for `Person`, `PersonC`, and `Spouse` as follows.



4.2.3 Construct as follows a schema whose type names are the states of the given tree automaton. First, for each state q_0 retrieve all the states reachable by following the second destination state of each transition and form a string automaton A_{q_0} whose atoms have the form $a[q]$ by rewriting each transition $q_1 \to a(q, q_2)$ as $q_1 \xrightarrow{a[q]} q_2$. Then convert each string automaton A_{q_0} to a regular expression T_{q_0}. Finally, construct a schema where each type name q_0 is defined as the corresponding converted regular expression T_{q_0} and a start type name q_{start} is added and defined as the choice for all the initial states.

4.3.3 The second step yields a deterministic string automaton; this automaton has at most one transition $q_1 \xrightarrow{a[X_j]} q_2$ from each source q_1 and label $a[X_j]$. Moreover, since the **single** restriction ensures that X_j is uniquely determined from each symbol a in such a string automaton, it actually has at most one transition $q_1 \xrightarrow{a[X_j]} q_2$ from each source q_1 and label a. Finally, since each string automaton is deterministic and hence has a unique initial state, the tree automaton produced in the third step has one transition $q_1 \to a(q_0, q_2)$ from each source q_1 and label a.

4.3.4 For example, in the tree automaton

$$(\{q_0, q_1, q_2\}, \{q_0\}, \{q_1\}, \{q_0 \to a(q_1, q_2),\ q_1 \to a(q_1, q_2),\ q_0 \to a(q_1, q_1)\})$$

the first two transitions mean that it is not bottom-up deterministic but, since they are never used, the automaton allows at most one bottom-up run.

4.4.6 It suffices to show that no new empty state arises after step (2). Thus, suppose that a state q remains after step (2). Since it has also remained after step (1), there is a bottom-up run r of A' on a tree t such that $r(\epsilon) = q$. Since step (2) has not eliminated q this state is reachable from initial states, and therefore all states used in r are also reachable and hence remain after step (2). This means that r is also a bottom-up run of A'' on t such that $r(\epsilon) = q$, witnessing the non-emptiness of r.

5.2.1 The result is $\left\{ \begin{array}{ll} \{\texttt{n} \mapsto \texttt{name["Taro"]}, & \texttt{s} \mapsto \texttt{name["Hanako"]}\}, \\ \{\texttt{n} \mapsto \texttt{name["Sachiko"]}, & \texttt{s} \mapsto \texttt{name["Hideki"]}\} \end{array} \right\}$.

5.4.1 Take, for example, `(a[] | a[],b[])*, (b[] | ())`. Consider matching it with the value `a[],b[]`. In the longest match the repetition pattern captures the whole sequence `a[],b[]` and the remaining pattern `()`, in which case we get priority id 1222. However, if the repetition pattern captures only `a[]` and the remaining pattern captures `b[]` then we get 1121, which is smaller.

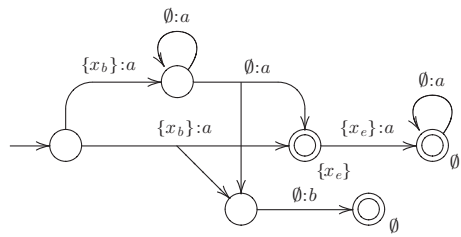
5.4.2 Take, for example, `(() | a[])*`. Consider matching it with the value `a[]`. When applying PP-Rep, if we take $n = 1$ in the premise then we get the priority id 122. If we divide the value as `a[] = (),a[]` and take $n = 2$ then we get a smaller id, 11122. If we divide the value as `a[] = (),(),a[]` with $n = 3$ then we get an even smaller id, 1111122, and so on.

6.1.3 Let us write $\mathbf{Var}(\pi) = \bigcup\{m(\pi\pi') \mid \pi\pi' \in \mathbf{nodes}(t)\}$. Suppose that (r, m) is a successful marking run on t. We need to show that, for each $\pi \in \mathbf{nodes}(t)$, we have (1) $\mathbf{Var}(\pi) = \mathbf{Var}(r(\pi))$ and, if $\mathbf{label}_t(\pi) = a$, then (2) $m(\pi)$, $\mathbf{Var}(\pi 1)$, and $\mathbf{Var}(\pi 2)$ are pairwise disjoint. Thereby, together with condition (L1), the linearity of m will follow. The proof is by induction on the height of $\mathbf{subtree}_t(\pi)$. If $\mathbf{label}_t(\pi) = \#$ then there exists $(r(\pi), \mathbf{x}) \in F$ with $m(\pi) = \mathbf{x}$; the result (1) follows since $\mathbf{Var}(\pi) = m(\pi) = \mathbf{x} = \mathbf{Var}(r(\pi))$ by condition (L2). If $\mathbf{label}_t(\pi) = a$ then there exists $r(\pi) \to m(\pi) : a(r(\pi 1), r(\pi 2)) \in \Delta$ with $m(\pi) = \mathbf{x}$; the result (1) follows since $\mathbf{Var}(\pi 1) = \mathbf{Var}(r(\pi 1))$ and $\mathbf{Var}(\pi 2) = \mathbf{Var}(r(\pi 2))$ by the induction hypothesis, and therefore $\mathbf{Var}(\pi) = m(\pi) \cup \mathbf{Var}(\pi 1) \cup \mathbf{Var}(\pi 2) =$

$\mathbf{x} \cup \mathbf{Var}(r(\pi 1)) \cup \mathbf{Var}(r(\pi 2))$ by condition (L3b). The result (2) follows from condition (L3a).

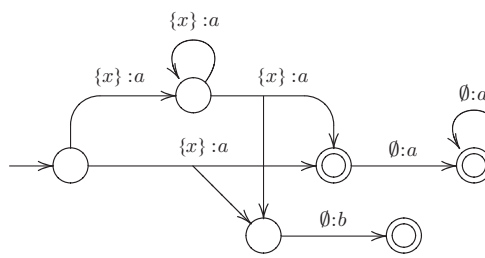
6.1.4 First, note that $\mathbf{Var}(q_0) = \mathbf{Var}(q_2) = \{x\}$ and $\mathbf{Var}(q_1) = \mathbf{Var}(q_3) = \emptyset$. Condition (L1) holds since the unique initial state q_0 satisfies $\mathbf{Var}(q_0) = \mathcal{X} = \{x\}$. Condition (L2) holds since $\mathbf{Var}(q_1) = \emptyset$ for $(q_1, \emptyset) \in F$ and $\mathbf{Var}(q_3) = \emptyset$ for $(q_3, \emptyset) \in F$. Condition (L3) holds since, for $q_0 \to \emptyset : a(q_1, q_0) \in \Delta$, (a) $\emptyset, \mathbf{Var}(q_1)$, and $\mathbf{Var}(q_0)$ are pairwise disjoint and (b) $\emptyset \cup \mathbf{Var}(q_1) \cup \mathbf{Var}(q_0) = \mathbf{Var}(q_0) = \{x\}$; similarly for the other transitions.

6.2.1 Such an automaton is given below (only reachable states are shown). Here any missing first destination is a final state with an empty variable set and no transition.



6.2.2 Here is a sketch of the answer. In the syntax, impose total orderings among the initial states, among the final states, and among the transitions. In the semantics, based on these orderings, define a total ordering among runs on a given tree and take the smallest successful run. In the construction, we first need to reject patterns that may not have the smallest priority id; these were considered in Exercise 5.4.2. Then, we perform the same construction except that for the case of a choice we give a higher priority to a transition to the first choice; for the case of a repetition we give a higher priority to a transition entering or continuing the inner loop than to one skipping or exiting it.

6.3.1 Such an automaton is given below (only reachable states are shown). As in Exercise 6.2.1, any missing first destination is a final state with an empty variable set and no transition.



7.2.1

1. The result is:

   ```
   person[name["Taro"], tel["123-456"], email["taro@xml"]],
   person[name["Hanako"], tel["987-654"]]
   ```

2. The program fails when pattern matching the sequence from the second element, since the misspelled head label does not match any pattern.
3. The program does not fail, but the result contains the element

   ```
   person[name["Hanako"], tel["987-654"], tel["987-654"]]
   ```

   which does not conform to the `Result` type.

7.2.2

1. The program passes the exhaustiveness check since the non-empty sequence is covered by the second clause and the empty sequence by the third. If the second clause were missing, the case where the head `person` contains a `tel` would not be covered. If the third clause were missing then the empty sequence would not be covered.
2. The program passes the irredundancy check since the second clause captures sequences where the head `person` contains a `tel`, and the third clause captures the empty sequence. If the first and second clauses were swapped, the former second clause would already capture the case where the head `person` contains no `tel` and therefore the former first clause would capture no value.
3. Inference gives the binding {`rest` ↦ `Person*`} for the first pattern, {`hd` ↦ `person[Name,Tel,Email*]`, `rest` ↦ `Person*`} for the second, and {} for the third. If the first clause were missing, inference would return the type `person[Name,Tel?,Email*]` for `hd` for the second clause.
4. Checking each recursive call gives a positive result since the argument `rest` has type `Person*`, which matches the declared type. The type for the pattern match is then calculated as

   ```
   Result* | person[Name,Tel,Email*],Result* | ()
   ```

   which is a subtype of `Result*`. Thus, typechecking succeeds. If we use the type `Any` for `rest` then typechecking fails at a recursive call, since `Any` is not a subtype of `Person*`. If we use the type `person[Any]` for `hd`, then the pattern match has type

   ```
   Result* | person[Any],Result* | ()
   ```

   which is not a subtype of `Result*` and therefore typechecking fails. If the first clause is missing and we use the type `person[Name,Tel?,Email*]` for `hd`

then the pattern match has type

```
person[Name,Tel?,Email*],Result* | ()
```

which is also not a subtype of `Result*`, and therefore typechecking fails.

7.2.5

1. The identity function

   ```
   fun f(x :···) :···= x
   ```

   would work for any value if the declared types are ignored.
2. For an input `x` of type

   ```
   a[b[]],b[] | a[c[]],c[]
   ```

   type inference on the pattern match

   ```
   match x with a[Any as x], Any as y  ->  ···
   ```

   will yield

   $$\{\texttt{x} \mapsto \texttt{b[]}\,|\,\texttt{c[]},\ \texttt{y} \mapsto \texttt{b[]}\,|\,\texttt{c[]}\}$$

   which loses the information that the values bound in `x` and `y` are both `b[]` or else both `c[]`.

7.3.5 Modify the algorithm in Section 7.3.1 so that in the first step the final states are $F_C = F_A \times F_B$, in the third step the final states are $F_j = F_D$, and the first clause of the transitions is as follows:

$$\Delta_j = \{\langle q, 0\rangle \to a(\langle q_1, 1\rangle, \langle q_2, 0\rangle) \mid q \to \mathbf{x} : a(q_1, q_2) \in \Delta_D,\ x_j \in \mathbf{x}\} \cup \cdots$$

8.1.2 As a result of the ACCEPTING function, each leaf returns $\{q_2\}$ and thereafter each intermediate node also returns $\{q_2\}$. Since the root thus returns $\{q_2\}$, which contains the initial state q_2, the final result from the main ACCEPT function is true.

8.1.3 As a result of the ACCEPTING function, each leaf returns $\{q_1, q_3\}$ and thereafter each intermediate node also returns $\{q_1, q_3\}$; thus, the returned set always contains q_3. Since the root thus returns $\{q_1, q_3\}$, which contains the initial state q_1, the final result from the main ACCEPT function is true; thus, q_3 is unused in the end.

8.1.4 The root receives $\{q_1\}$ as an argument passed to the ACCEPTINGAMONG function and thereafter each intermediate node and the leftmost leaf receive $\{q_1, q_2\}$. The leftmost leaf returns $\{q_1\}$ as the result of the ACCEPTINGAMONG function and then its right-sibling receives $\{q_1\}$ and returns $\{q_1\}$; on the basis of the two sets

returned from the leftmost leaf and its right-sibling, their parent returns $\{q_1\}$. Each intermediate node performs the same computation and returns $\{q_1\}$; thus each set returned from the ACCEPTING function never contains q_3. Since the root thus returns $\{q_1\}$, which is not empty, the final result from the main ACCEPT function is true.

8.1.5 Right after line 4, insert the following:

if $r = \emptyset$ **then** return $\emptyset$

8.2.2 For each leaf we get

$$\big\{q_0 \mapsto \emptyset,\ q_1 \mapsto \emptyset,\ q_2 \mapsto \emptyset,\ q_3 \mapsto \{\emptyset\}\big\}.$$

For each node $a(\#, \#)$ we get

$$\big\{q_0 \mapsto \emptyset,\ q_1 \mapsto \{\{x \mapsto a(\#, \#)\}\},\ q_2 \mapsto \{\{y \mapsto a(\#, \#)\}\},\ q_3 \mapsto \{\emptyset\}\big\}.$$

For each node $a(a(\#, \#), \#)$ we get

$$\left\{\begin{array}{l} q_0 \mapsto \emptyset, \\ q_1 \mapsto \{\{x \mapsto a(a(\#, \#), \#)\}, \{x \mapsto a(\#, \#)\}\}, \\ q_2 \mapsto \{\{y \mapsto a(a(\#, \#), \#)\}, \{y \mapsto a(\#, \#)\}\}, \\ q_3 \mapsto \{\emptyset\} \end{array}\right\}.$$

For the root we get

$$M_{8.2.2} = \left\{\begin{array}{l} q_0 \mapsto \left\{\begin{array}{l} \{x \mapsto a(a(\#, \#), \#),\ y \mapsto a(a(\#, \#), \#)\}, \\ \{x \mapsto a(\#, \#),\ y \mapsto a(a(\#, \#), \#)\}, \\ \{x \mapsto a(a(\#, \#), \#),\ y \mapsto a(\#, \#)\}, \\ \{x \mapsto a(\#, \#),\ y \mapsto a(\#, \#)\} \end{array}\right\} \\ q_1 \mapsto \cdots, \\ q_2 \mapsto \cdots, \\ q_3 \mapsto \{\emptyset\} \end{array}\right\}.$$

(where the parts indicated by $\cdots$ are omitted for brevity). Here, a large set of bindings is computed by enumerating all combinations of x's bindings and y's bindings. Finally, $M_{8.2.2}(q_0)$ is returned.

8.2.4 For the node $t_{8.2.2}$ we get last set $M_{8.2.2}$ of bindings. Then, for the root we get

$$M_{8.2.4} = \left\{\begin{array}{l} q_0 \mapsto \cdots, \\ q_1 \mapsto \cdots, \\ q_2 \mapsto \cdots, \\ q_3 \mapsto \{\emptyset\}, \\ q_A \mapsto \{\{x \mapsto a(t_{8.2.2}, \#),\ y \mapsto a(t_{8.2.2}, \#)\}\} \end{array}\right\}.$$

Finally, $M_{8.2.4}(q_A)$ is returned. Here, $M_{8.2.4}(q_A)$ is computed only from $M_{8.2.2}(q_3)$ since the transition $q_A \to \emptyset : b(q_0, q_3)$ does not match the node label. Therefore the large intermediate result $M_{8.2.2}(q_0)$ is unused in the end.

8.2.6 For each node other than the root, we get exactly the same set of bindings as in $A_{8.2.4}$ except that $q_B \mapsto \emptyset$ is added. For the root, although the transition $q_A \to \emptyset : a(q_0, q_B)$ matches the node label, this does not contribute to the final set of bindings because of the mapping $q_B \mapsto \emptyset$ for the right child, which has the effect of discarding the large intermediate result $M_{8.2.2}(q_0)$. Therefore, again we get the same set $M_{8.2.4}$ of bindings with the addition of $q_B \mapsto \emptyset$.

8.2.9 For either of the automata $A_{8.2.3}$ and $A_{8.2.5}$ we get the following for the node $t_{8.2.2}$:

$$M_{8.2.9} = \left\{ \begin{array}{l} q_0 \mapsto \left\{ \begin{array}{l} (\{x \mapsto a(a(\#,\#),\#)\} \uplus \{x \mapsto a(\#,\#)\}) \\ \otimes(\{y \mapsto a(a(\#,\#),\#)\} \uplus \{y \mapsto a(\#,\#)\}) \end{array} \right\} \\ q_1 \mapsto \cdots, \\ q_2 \mapsto \cdots, \\ q_3 \mapsto \{\emptyset\}, \\ q_B \mapsto \emptyset \end{array} \right\}$$

Here, $M_{8.2.9}(q_0)$ does not enumerate the set of all combinations of bindings but represents it symbolically. Then, with the same reasoning as in Exercises 8.2.4 or 8.2.6, we can discard the symbolic set $M_{8.2.9}(q_0)$ before expanding it. In particular, for $A_{8.2.5}$ this can be done by partially computing the symbolic set as follows:

$$\left(\begin{array}{l} (\{x \mapsto a(a(\#,\#),\#)\} \uplus \{x \mapsto a(\#,\#)\}) \\ \otimes(\{y \mapsto a(a(\#,\#),\#)\} \uplus \{y \mapsto a(\#,\#)\}) \end{array} \right) \otimes \emptyset = \emptyset$$

8.2.10 The modified bottom-up marking algorithm is as follows.

```
1:  function MATCH(t, (Q, I, F, Δ, X))        ▷ returning a multiset of bindings
2:      let M = MATCHINGAMONG(t, I)
3:      return eval( ⨄_{q∈I} M(q) )
4:
5:      function MATCHINGAMONG(t, r)
6:                ▷ returning a mapping from a state in r to a multiset of bindings
7:          switch t do
8:            case #:
9:              return { q ↦ ⨄_{(q,x)∈F} {{x ↦ t}} | q ∈ r }
10:           case a(t1, t2):
11:             let r1 = {q1 | q → x : a(q1, q2) ∈ Δ, q ∈ r}
```

12: **let** $M_1 = \textsc{MatchingAmong}(t_1, r_1)$
13: **let** $r_2 = \{q_2 \mid q \to \mathbf{x} : a(q_1, q_2) \in \Delta,\ q \in r,\ q_1 \in r_1\}$
14: **let** $M_2 = \textsc{MatchingAmong}(t_2, r_2)$
15: **return** $\left\{ q \mapsto \biguplus_{(q \to \mathbf{x}:a(q_1,q_2)) \in \Delta} \{\{\mathbf{x} \mapsto t\}\} \otimes M_1(q_1) \otimes M_2(q_2) \;\middle|\; \begin{array}{l} q \in r, \\ q_1 \in r_1, \\ q_2 \in r_2 \end{array} \right\}$
16: **end switch**
17: **end function**
18: **end function**

8.3.4 The statement to prove is: suppose that $\textsc{IsEmptyIn}(q)$ with $Q_{\text{asm}} = Q_1$ returns true with $Q_{\text{asm}} = Q_2$; then, for any k, if all states in Q_1 are k-empty then all states in Q_2 are k-empty. The proof proceeds by induction on the call depth. There are two cases.

- The function returns true in line 9. Then the result immediately follows since $Q_{\text{asm}} = Q_1 = Q_2$.
- The function returns true in line 19. Let the transitions in Δ with source state q be the following:

$$q \to a_1(q_1, q_1') \quad \cdots \quad q \to a_n(a_n, a_n').$$

Then

$$\begin{array}{l} \textsc{IsEmptyIn}(q_1) \text{ with } Q_{\text{asm}} = R_1 \text{ returns true with } Q_{\text{asm}} = R_1', \\ \textsc{IsEmptyIn}(q_1') \text{ with } Q_{\text{asm}} = R_1' \text{ returns true with } Q_{\text{asm}} = R_2, \\ \vdots \\ \textsc{IsEmptyIn}(q_n) \text{ with } Q_{\text{asm}} = R_n \text{ returns true with } Q_{\text{asm}} = R_n', \text{ and} \\ \textsc{IsEmptyIn}(q_n') \text{ with } Q_{\text{asm}} = R_n' \text{ returns true with } Q_{\text{asm}} = R_{n+1}, \end{array}$$

where $R_1 = Q_1 \cup \{q\}$ and $R_{n+1} = Q_2$. By the induction hypothesis, for all h,

$$\begin{array}{l} \text{if all states in } R_1 \text{ are } h\text{-empty then all states in } R_1' \text{ are } h\text{-empty,} \\ \text{if all states in } R_1' \text{ are } h\text{-empty then all states in } R_2 \text{ are } h\text{-empty,} \\ \vdots \\ \text{if all states in } R_n \text{ are } h\text{-empty then all states in } R_n' \text{ are } h\text{-empty, and} \\ \text{if all states in } R_n' \text{ are } h\text{-empty then all states in } R_{n+1} \text{ are } h\text{-empty.} \end{array}$$

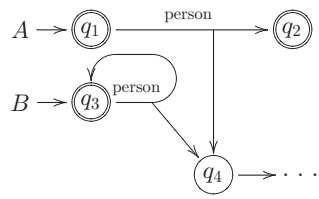
Altogether, if all states in $R_1 = Q_1 \cup \{q\}$ are h-empty then all states in $R_{n+1} = Q_2$ are h-empty. Therefore we only have to show that all states in $Q_1 \cup \{q\}$ are k-empty, from which the result follows. For this, note that $q_i \in R_i$ and $q_i' \in R_i'$ for all $i = 1, \ldots, n$. Then, from the above list of statements, we have that if all states in $Q_1 \cup \{q\}$ are h-empty then all q_i and q_i' ($i = 1, \ldots, n$) are h-empty, implying that q is $(h+1)$-empty. Since we know by assumption that all states in Q_1 are h-empty for all $0 \le h \le k$ and obviously

that q is 0-empty, it follows that q is h-empty for all $0 \leq h \leq k$, by induction on h. Hence all states in $Q_1 \cup \{q\}$ are k-empty.

8.3.6 We need to prove that if IsEmptyIn(q) returns false then q is not empty, by induction on the call graph. There are two cases.

- The function returns false in line 10. Then, the result immediately follows since $q \in F$.
- The function returns false in line 16. Then, there is a transition $q \to a(q_1, q_2) \in \Delta$ such that both IsEmptyIn(q_1) and IsEmptyIn(q_2) return false. By the induction hypothesis, both q_1 and q_2 are non-empty. Therefore q is non-empty.

8.3.8 The automata would look like the following:

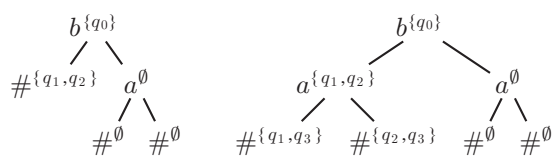


We start the containment algorithm with the pair $(q_1, \{q_3\})$. Then, since A has the unique transition $q_1 \to \text{person}(q_2, q_4)$ and B has the unique transition $q_3 \to \text{person}(q_3, q_4)$, four recursive calls with the following pairs are possible:

$$(q_2, \emptyset), \qquad (q_4, \{q_4\}), \qquad (q_2, \{q_3\}), \qquad (q_4, \emptyset).$$

In order for the current call to return true, either of the first two calls must return true and either of the last two must return true. Indeed, the call with $(q_2, \emptyset)$ returns false by the finality check while the call with $(q_4, \{q_4\})$ returns true, since the containment holds trivially regardless of states further away than q_4. The call with $(q_2, \{q_3\})$ returns true since q_2 has no transition; thus the call with $(q_4, \emptyset)$ is not needed and hence is skipped.

9.1.3 The successful alternating runs of $A_{9.1.2}$ are as follows:



9.2.2 The required nondeterministic tree automaton is (S, I', F', Δ), where

$$\begin{array}{lcl} S & = & \{\emptyset, \{q_0\}, \{q_1, q_2\}, \{q_1, q_3\}, \{q_2, q_3\}\} \\ I' & = & \{\{q_0\}\} \\ F' & = & \{\emptyset, \{q_1, q_2\}, \{q_1, q_3\}, \{q_2, q_3\}\} \\ \Delta & = & \left\{ \begin{array}{l} \{q_0\} \to b(\{q_1, q_2\}, \emptyset) \\ \{q_1, q_2\} \to a(\{q_1, q_3\}, \{q_2, q_3\}) \\ \emptyset \to a(\emptyset, \emptyset) \\ \emptyset \to b(\emptyset, \emptyset) \end{array} \right\}. \end{array}$$

9.2.6 The required bottom-up deterministic tree automaton is (S, I', F', Δ), where

$$\begin{array}{lcl} S & = & \{\{q_1, q_2, q_3\}, \{q_1, q_2\}, \{q_0\}, \{q_1\}, \{q_2\}, \emptyset\} \\ I' & = & \{\{q_0\}\} \\ F' & = & \{\{q_1, q_2, q_3\}\} \end{array}$$

and Δ contains transitions of the forms $s \to a(s_1, s_2)$ and $s \to b(s_1, s_2)$ for all combinations of $s_1 \in S$ and $s_2 \in S$ (here the details are omitted). This automaton has 72 transitions while that in Exercise 9.2.2 has only four.

9.3.4 The required bottom-up emptiness test algorithm is as follows:

```
1:  function IsEmpty(Q, I, F, Φ)                    ▷ returning a boolean
2:      S_NE ← F
3:      repeat
4:          for all s1, s2 ∈ S_NE do
5:              let s = {q ∈ Q | s1, s2 ⊢ Φ(q, a)}
6:              S_NE ← S_NE ∪ {s}
7:          end for
8:      until S_NE does not change
9:      return ∀s ∈ S_NE. s ∩ I = ∅
10: end function
```

9.3.5 The required top-down emptiness test algorithm is as follows:

```
1:  function IsEmpty(Q, I, F, Φ)                    ▷ returning a boolean
2:      S_asm ← ∅
3:      for all q ∈ I do
4:          if not IsEmptyIn({q}) then return false
5:      end for
```

```
    return true

    function IsEmptyIn(s)          ▷ returning a boolean, manipulating S_asm
        if s ∈ S_asm then return true
        if s ⊆ F then return false
        S_asm ← S_asm ∪ {s}
        for all (s1, s2) ∈ DNF(⋀_{q∈s} Φ(q, a)) do
            let S' = S_asm
            if not IsEmptyIn(s1) then
                S_asm ← S'
                if not IsEmptyIn(s2) then return false
            end if
        end for
        return true
    end function
end function
```

The bottom-up algorithm is advantageous when non-empty states are fewer than states reachable from initial states. The top-down algorithm is advantageous in the opposite situation. In addition, the top-down algorithm can use various optimizations that are analogous to those in Section 8.3.2. For example, at the beginning of IsEmptyIn we can immediately return false when $s = \emptyset$. Also, we can maintain a "false set" of states that are found to be definitely non-empty.

10.1.2 The sets of transformed trees are $\{a(c(\#, \#), \#)\}$ and $\{a(\#, \#)\}$.

10.1.4 The result is the empty set.

10.3.2 We obtain $\{a(\#, a(a(\#, \#), a(\#, \#)))\}$.

10.3.5 Let d_n $(n \geq 0)$ be (simplified) XHTML fragments in the unranked tree format defined by

$$d_n = \texttt{div}[d_{n-1}], d_{n-1} \qquad (n \geq 1)$$
$$d_0 = \texttt{img}[].$$

The transformation given in the exercise is expressed by the following function f:

$$f(\texttt{div}[d], d') = g(d),\ f(d),\ f(d')$$
$$f(\texttt{img}[]) = ()$$

where $g(d)$ is the sequence of all `img`s appearing in d. Let us write $\mathrm{len}(d)$ for the length of the sequence d; clearly, $\mathrm{len}(d)$ is smaller than or equal to the height of d in the binary tree format. Note that $\mathrm{len}(g(d_n)) = 2^n$. Therefore, if we write $L_n = \mathrm{len}(f(d_n))$ then

$$L_n = 2^n + 2L_{n-1}$$

for $n \geq 1$, and $L_0 = 0$. This recurrence equation can be solved to give

$$L_n = n2^{n-1}.$$

This cannot be bounded by c^n for any $c \geq 0$. Since the height of d_n in the binary tree format equals $n + 2$, we may conclude, by Theorem 10.3.4, that the transformation cannot be expressed by a macro tree transducer.

11.2.1 Let a linear top-down tree transducer (P, P_0, Π) and the "input" tree automaton (Q, I, F, Δ) be given. Let E, E' be the sets of subexpressions appearing in node rules and leaf rules, respectively, of Π. Construct the "output" tree automaton (Q', I', F', Δ'), where

$$\begin{aligned}
Q' &= \{\langle p, q\rangle \mid p \in P,\ q \in Q\} \\
&\quad \cup \{\langle e, q_1, q_2\rangle \mid e \in E,\ q_1, q_2 \in Q\} \\
&\quad \cup \{\langle e, _\rangle \mid e \in E'\} \\
I' &= \{\langle p, q\rangle \mid p \in P_0,\ q \in I\} \\
F' &= \{\langle \#, q_1, q_2\rangle \mid q_1, q_2 \in Q\} \\
&\quad \cup \{\langle \#, _\rangle\} \\
\Delta' &= \{\langle p, q\rangle \xrightarrow{\epsilon} \langle e, q_1, q_2\rangle \mid p(a(x_1, x_2)) \to e \in \Pi,\ q \to a(q_1, q_2) \in \Delta\} \\
&\quad \cup \{\langle p, q\rangle \xrightarrow{\epsilon} \langle e, _\rangle \mid p(\#) \to e \in \Pi,\ q \in F\} \\
&\quad \cup \{\langle a(e_1, e_2), q_1, q_2\rangle \to a(\langle e_1, q_1, q_2\rangle, \langle e_2, q_1, q_2\rangle) \mid e_1, e_2 \in E,\ q_1, q_2 \in Q\} \\
&\quad \cup \{\langle a(e_1, e_2), _\rangle \to a(\langle e_1, _\rangle, \langle e_2, _\rangle) \mid e_1, e_2 \in E'\} \\
&\quad \cup \{\langle p(x_h), q_1, q_2\rangle \xrightarrow{\epsilon} \langle p, q_h\rangle \mid q_1, q_2 \in Q\}.
\end{aligned}$$

Note that, in the third clause of Δ', linearity ensures that q_1 and q_2 are each used only once in the two branches. This fact is crucial in representing the output type precisely.

11.3.4 Let the input type τ_I be the set of trees with labels a and b, and the output type τ_O be the set of trees with only label a. Then, the transducer $\mathcal{T}_{11.3.2}$ should not typecheck since $\mathcal{T}_{11.3.2}(\tau_I)$ is the set of trees with a and b and therefore $\mathcal{T}_{11.3.2}(\tau_I) \not\subseteq \tau_O$. However, the "algorithm" would wrongly answer yes since $\mathcal{T}^{-1}_{11.3.2}(\tau_O) = \{t \mid \exists t' \in \tau_O . t' \in \mathcal{T}_{11.3.2}(t)\}$ is the set of trees with a and b, and therefore $\tau_I \subseteq \mathcal{T}^{-1}_{11.3.2}(\tau_O)$.

11.3.6 Suppose that the transformation can be expressed by a top-down tree transducer $\mathcal{T}$. Noting that the transformation returns at most one result for any input, since the set $\{\texttt{true}\}$ is regular Theorem 11.3.1 tells us that the preimage $\mathcal{T}^{-1}(\{\texttt{true}\})$ is regular. However, this contradicts the assumption that the preimage is the set of trees that have identical subtrees, which is not regular.

12.1.1 The required caterpillar path is: 1 1 # is1 up 2 # is2 up is1 up 2 1 # is1 up 2 # is2 up is2 up.

12.2.1 Let us consider string automata where the symbols are caterpillar atoms and call such automata caterpillar automata. Through the usual conversions between regular expressions and string automata, we can convert between caterpillar expressions and caterpillar automata.

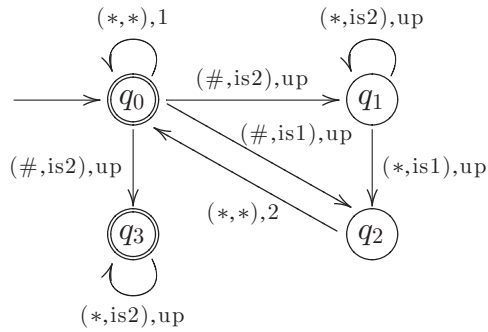
From a tree-walking automaton (Q, I, F, δ) we can construct a caterpillar automaton for which we use the same initial states and

- for each transition $q_1 \xrightarrow{(a^\#,h),d} q_2 \in \delta$, we generate three transitions $q_1 \xrightarrow{a^\#} q_1' \xrightarrow{h} q_1'' \xrightarrow{d} q_2$ with additional states q_1' and q_1'', and
- for each final state $(q, (a^\#, h)) \in F$, we generate two transitions $q \xrightarrow{a^\#} q' \xrightarrow{h} q''$ with additional states q' and q'', where q'' is a new final state.

From a caterpillar automaton (Q, I, F, δ) we can construct a tree-walking automaton for which we use the same initial states; furthermore,

- whenever there are transitions $q_1 \xrightarrow{c_1} \cdots \xrightarrow{c_{n-1}} q_n \xrightarrow{d} q \in \delta$ and $\{a^\#, h\} \supseteq \{c_1, \ldots, c_{n-1}\}$ we generate the transition $q_1 \xrightarrow{(a^\#,h),d} q$ and
- whenever there are transitions $q_1 \xrightarrow{c_1} \cdots \xrightarrow{c_{n-1}} q_n \in \delta$ with $q_n \in F$ and $\{a^\#, h\} \supseteq \{c_1, \ldots, c_{n-1}\}$ we generate the final state $(q, (a^\#, h))$.

12.2.2 The required tree-walking automaton is shown below, where a transition containing $*$ should be read as a set of transitions in which $*$ is replaced by each label or element from {isRoot, is1, is2}:



A successful run is:

$$(\epsilon, q_0)\ (1, q_0)\ (11, q_0)\ (1, q_2)\ (12, q_0)\ (1, q_1)\ (\epsilon, q_2)$$
$$(2, q_0)\ (21, q_0)\ (2, q_2)\ (22, q_0)\ (2, q_3)\ (\epsilon, q_3)$$

13.2.1 If $\Leftrightarrow$ is replaced by $\Leftarrow$ then Y becomes a set of nodes including all children of x but possibly more. If $\Leftrightarrow$ is replaced by $\Rightarrow$ then Y becomes the set of the first several children of x. That is, if the children are $\pi_1, \ldots, \pi_n$, in this order, then Y consists of $\pi_1, \ldots, \pi_k$ for some $k \leq n$.

13.2.2 Define

$$\text{xdescendant}(x, y) \equiv \exists Y.\ (\text{xdescendants}(x, Y) \wedge y \in Y)$$

where

$$\text{xdescendants}(x, Y) \equiv \forall y.\ y \in Y \Leftrightarrow (x = y \vee \exists z.\ (z \in Y \wedge \text{xchild}(x, y))).$$

Define

$$x \leq y \equiv \exists Y.\ (\text{descendants}(x, Y) \wedge y \in Y)$$

where

$$\text{descendants}(x, Y) \equiv$$
$$\forall y.\ y \in Y \Leftrightarrow (x = y \vee \exists z.\ (z \in Y \wedge (\text{leftchild}(z, y) \vee \text{rightchild}(z, y)))).$$

Define

$$x \preceq y \equiv x \leq y \vee \exists z, u, v.\ (\text{leftchild}(z, u) \wedge u \leq x \wedge \text{rightchild}(z, v) \wedge v \leq y).$$

13.3.2 Let a marking tree automaton $(Q, I, F, \Delta, \mathcal{X})$ be given, where $Q = \{q_1, \ldots, q_n\}$ and $\mathcal{X} = \{x_1, \ldots, x_k\}$. Then, construct the formula

$$
\begin{aligned}
&\exists\, q_1, \ldots, q_n, x_1, \ldots, x_k.\\
&\quad \forall\, u.\ \bigvee_{q \in Q} \left(u \in q \wedge \bigwedge_{q' \in Q \setminus \{q\}} u \notin q' \right)\\
&\quad \wedge \exists u.\ \mathrm{root}(u) \wedge \bigvee_{q \in I} u \in q\\
&\quad \wedge \forall\, u.\ \neg \mathrm{leaf}(u)\\
&\qquad \Rightarrow \exists\, v, w. \mathrm{leftchild}(u, v) \wedge \mathrm{rightchild}(u, w)\\
&\qquad \wedge \bigvee_{q \to X : a(q_1, q_2) \in \Delta} \left(u \in q \wedge a(u) \wedge v \in q_1 \wedge w \in q_2 \wedge \bigwedge_{x \in X} u \in x \right)\\
&\qquad \wedge \forall u.\ \mathrm{leaf}(u) \Rightarrow \bigvee_{(q, X) \in F} \left(u \in q \wedge \bigwedge_{x \in X} u \in x \right).
\end{aligned}
$$

Here, all the large $\bigwedge$'s and $\bigvee$'s are to be expanded in accordance with their "statically known" subscripts.

14.1.9 The set $\mathbf{pos}(r)$ can be computed simply by collecting all the elaborated symbols in r. The set $\mathbf{first}(r)$ can be computed as follows:

$$
\begin{aligned}
\mathbf{first}(r^*) &= \mathbf{first}(r)\\
\mathbf{first}(r_1 \mid r_2) &= \mathbf{first}(r_1) \cup \mathbf{first}(r_2)\\
\mathbf{first}(r_1, r_2) &= \begin{cases} \mathbf{first}(r_1) & \text{if } \epsilon \notin \mathcal{L}(r_1)\\ \mathbf{first}(r_1) \cup \mathbf{first}(r_2) & \text{if } \epsilon \in \mathcal{L}(r_1) \end{cases}\\
\mathbf{first}(a^{(i)}) &= \{a^{(i)}\}\\
\mathbf{first}(\epsilon) &= \emptyset
\end{aligned}
$$

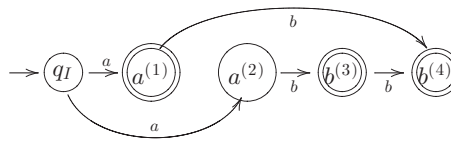
The set $\mathbf{last}(r)$ is computed similarly. The set $\mathbf{follow}(r, a^{(i)})$ (assuming that $a^{(i)} \in \mathbf{pos}(r)$) is be computed as follows:

$$
\begin{aligned}
\mathbf{follow}(r^*, a^{(i)}) &= \begin{cases} \mathbf{follow}(r, a^{(i)}) & \text{if } a^{(i)} \in \mathbf{pos}(r) \setminus \mathbf{last}(r)\\ \mathbf{follow}(r, a^{(i)}) \cup \mathbf{first}(r, a^{(i)}) & \text{if } a^{(i)} \in \mathbf{last}(r) \end{cases}\\
\mathbf{follow}((r_1 \mid r_2), a^{(i)}) &= \begin{cases} \mathbf{follow}(r_1, a^{(i)}) & \text{if } a^{(i)} \in \mathbf{pos}(r_1)\\ \mathbf{follow}(r_2, a^{(i)}) & \text{if } a^{(i)} \in \mathbf{pos}(r_2) \end{cases}
\end{aligned}
$$

$$\mathbf{follow}((r_1, r_2), a^{(i)}) = \begin{cases} \mathbf{follow}(r_2, a^{(i)}) & \text{if } a^{(i)} \in \mathbf{pos}(r_2) \\ \mathbf{follow}(r_1, a^{(i)}) & \text{if } a^{(i)} \in \mathbf{pos}(r_1) \setminus \mathbf{last}(r_1) \\ \mathbf{follow}(r_1, a^{(i)}) \cup \mathbf{first}(r_2) & \text{if } a^{(i)} \in \mathbf{last}(r_1) \end{cases}$$

$$\mathbf{follow}(a^{(i)}, a^{(i)}) = \emptyset$$

14.1.14 The Glushkov automaton is as follows:



Two runs, $q_I a^{(1)} b^{(4)}$ and $q_I a^{(2)} b^{(3)}$, each accept ab.

14.2.4 We can adjust the proof of Theorem 14.1.20 as follows. Suppose that the automaton A is binding-ambiguous. Then, there are two runs

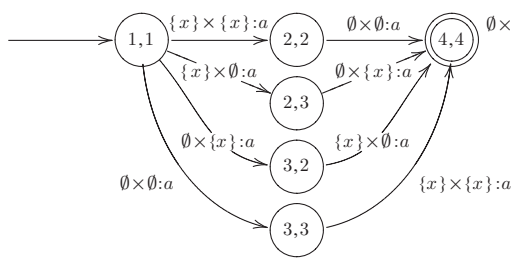
$$q_1 q_2 \cdots q_n \qquad \text{and} \qquad r_1 r_2 \cdots r_n$$

both accepting a string s, where $q_i \xrightarrow{\mathbf{x}:a} q_{i+1}$, $r_i \xrightarrow{\mathbf{x}':a} r_{i+1} \in \delta$ for some $1 \leq i \leq n$ with $\mathbf{x} \neq \mathbf{x}'$ or $(q_1, \mathbf{x}), (r_1, \mathbf{x}') \in F'$ for $n = 1$ with $\mathbf{x} \neq \mathbf{x}'$. Therefore the automaton B also has a run

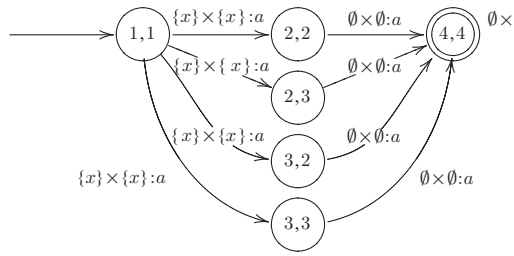
$$(q_1, r_1)(q_2, r_2) \cdots (q_n, r_n)$$

accepting s, where $(q_i, r_i) \xrightarrow{\mathbf{x} \times \mathbf{x}'} (q_{i+1}, r_{i+1}) \in \delta'$ or $((q_1, r_1), (\mathbf{x} \times \mathbf{x}')) \in F$ for $n = 1$. In the first case, since (q_i, r_i) and (q_{i+1}, r_{i+1}) are in the accepting run these states and the last transition remain in the automaton C. In the second case, since (q_1, r_1) is in the accepting run this state remains in C. Thus, in either case the algorithm returns "binding-ambiguous." The converse can be adjusted similarly.

14.2.5 For case (A), the second step of the algorithm yields the following:



Therefore the original marking automaton is binding-ambiguous. For case (B), the second step of the algorithm yields the following:



Therefore the original marking automaton is binding-unambiguous.

References

Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L. (1997). The Lorel query language for semistructured data. *Int. Journal on Digital Libraries*, 1(1): 68–88.

Aho, A. V. and Ullman, J. D. (1971). Translation on a context-free grammar. *Information and Control*, 19(5): 439–475.

Alon, N., Milo, T., Neven, F., Suciu, D., and Vianu, V. (2003). XML with data values: typechecking revisited. *Journal of Computer and System Sciences*, 66(4): 688–727.

Antimirov, V. (1996). Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155: 291–319.

Benedikt, M. and Koch, C. (2008). XPath leashed. *ACM Computing Surveys*, 41(1): 3: 1–54.

Benzaken, V., Castagna, G., and Frisch, A. (2003). CDuce: an XML-centric general-purpose language. *SIGPLAN Notices*, 38(9): 51–63.

Berlea, A. and Seidl, H. (2002). Binary queries. In *Extreme Markup Languages*. Electronic edition.

Bojańczyk, M. and Colcombet, T. (2006). Tree-walking automata cannot be determinized. *Theoretical Computer Science*, 350(2–3): 164–173.

Bojańczyk, M. and Colcombet, T. (2008). Tree-walking automata do not recognize all regular languages. *SIAM Journal on Computing*, 38(2): 658–701.

Book, R., Even, S., Greibach, S., and Ott, G. (1971). Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20: 149–153.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E. (2000). Extensible markup language (XML™). `http://www.w3.org/XML/`.

Brüggemann-Klein, A. (1993). Regular expressions into finite automata. *Theoretical Computer Science*, 120: 197–213.

Brüggemann-Klein, A. and Wood, D. (1998). One-unambiguous regular languages. *Information and Computation*, 140(2): 229–253.

Brüggemann-Klein, A. and Wood, D. (2000). Caterpillars: a context specification technique. *Markup Languages*, 2(1): 81–106.

Brüggemann-Klein, A., Murata, M., and Wood, D. (2001). Regular tree and regular hedge languages over unranked alphabets. Technical Report HKUST-TCSC-2001-0, Hongkong University of Science and Technology.

Brzozowski, J. A. (1964). Derivatives of regular expressions. *Journal of the ACM*, 11(4): 481–494.

Buneman, P., Fernandez, M. F., and Suciu, D. (2000). UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1): 76–110.

Calcagno, C., Gardner, P., and Zarfaty, U. (2005). Context logic and tree update. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pp. 271–282.

Cardelli, L. and Ghelli, G. (2004). TQL: a query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14(3): 285–327.

Christensen, A. S., Møller, A., and Schwartzbach, M. I. (2002). Static analysis for dynamic XML. In *Proc. Workshop on Programming Language Technologies for XML (PLAN-X)*.

Clark, J. (2002). An algorithm for RELAX NG validation. `http://www.thaiopensource.com/relaxng/implement.html`.

Clark, J. and DeRose, S. (1999). XML path language (XPath). `http://www.w3.org/TR/xpath`.

Clark, J. and Murata, M. (2001). RELAX NG. `http://www.relaxng.org`.

Colazzo, D., Ghelli, G., and Sartiani, C. (2009). Efficient inclusion for a class of XML types with interleaving and counting. *Information Systems*, 34(7): 643–656.

Comon, H., Dauchet, M., Gilleron, R., *et al.* (2007). Tree automata techniques and applications. Available on: `http://tata.gforge.inria.fr`. Release 12 October 2007.

Courcelle, B. (1994). Monadic second-order definable graph transductions: a survey. *Theoretical Computer Science*, 126(1): 53–75.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. (1998). XML-QL: a query language for XML. `http://www.w3.org/TR/NOTE-xml-ql`.

Engelfriet, J. (1975). Bottom-up and top-down tree transformations – a comparison. *Mathematical Systems Theory*, 9(3): 198–231.

Engelfriet, J. (1977). Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10: 289–303.

Engelfriet, J. and Maneth, S. (2003). A comparison of pebble tree transducers with macro tree transducers. *Acta Informatica*, 39(9): 613–698.

Engelfriet, J. and Vogler, H. (1985). Macro tree transducers. *Journal of Computer and System Sciences*, 31(1): 710–146.

Engelfriet, J. and Vogler, H. (1988). High level tree transducers and iterated pushdown tree transducers. *Acta Informatica*, 26(1/2): 131–192.

Fallside, D. C. (2001). XML Schema Part 0: Primer, W3C Recommendation. `http://www.w3.org/TR/xmlschema-0/`.

Fankhauser, P., Fernández, M., Malhotra, A., Rys, M., Siméon, J., and Wadler, P. (2001). XQuery 1.0 Formal Semantics. `http://www.w3.org/TR/query-semantics/`.

Flum, J., Frick, M., and Grohe, M. (2002). Query evaluation via tree-decompositions. *Journal of the ACM*, 49(6): 716–752.

Foster, J. N., Pierce, B. C., and Schmitt, A. (2007). A logic your typechecker can count on: unordered tree types in practice. In *Proc. Workshop on Programming Language Technologies for XML (PLAN-X)*, pp. 80–90.

Frisch, A. (2004). *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. Ph.D. thesis, Université Paris 7.

Frisch, A. (2006). OCaml + XDuce. In *Proc. Int. Conf. on Functional Programming (ICFP)*, pp. 192–200.

Frisch, A. and Hosoya, H. (2007). Towards practial typechecking for macro tree transducers. In *Proc. Int. Symp. on Database Programming Languages (DBPL)*, pp. 246–260.

Frisch, A., Castagna, G., and Benzaken, V. (2008). Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4).

Gapeyev, V., Garillot, F., and Pierce, B. C. (2006). Statically typed document transformation: an Xtatic Experience. In *Proc. Workshop on Programming Language Technologies for XML (PLAN-X)*, pp. 2–13.

Ghelli, G., Colazzo, D., and Sartiani, C. (2008). Linear time membership in a class of regular expressions with interleaving and counting. In *Proc. 17th ACM Conf. on Information and Knowledge Management*, pp. 389–398.

Harren, M., Raghavachari, M., Shmueli, O., *et al.* (2005). XJ: facilitating XML processing in Java. In *Proc. 14th Int. Conf. on World Wide Web (WWW2005)*, pp. 278–287.

Henriksen, J. G., Jensen, J. L., Jørgensen, M. E., *et al.* (1995). Mona: monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1019 of *Lecture Notes in Computer Science*, pp. 89–110.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Hosoya, H. (2003). Regular expression pattern matching – a simpler design. Technical Report 1397, RIMS, Kyoto University.

Hosoya, H. (2006). Regular expression filters for XML. *Journal of Functional Programming*, 16(6): 711–750.

Hosoya, H. and Murata, M. (2006). Boolean operations and inclusion test for attribute-element constraints. *Theoretical Computer Science*, 360(1–3): 327–351.

Hosoya, H. and Pierce, B. C. (2002). Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(6): 961–1004.

Hosoya, H. and Pierce, B. C. (2003). XDuce: a typed XML processing language. *ACM Transactions on Internet Technology*, 3(2): 117–148.

Hosoya, H., Vouillon, J., and Pierce, B. C. (2004). Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1): 46–90.

Hosoya, H., Frisch, A., and Castagna, G. (2009). Parametric polymorphism for XML. *ACM Transactions on Programming Languages and Systems*, 32(1): 2: 1–56.

Inaba, K. and Hosoya, H. (2007). XML transformation language based on monadic second-order logic. In *Proc. Workshop on Programming Language Technologies for XML (PLAN-X)*, pp. 49–60.

Inaba, K. and Hosoya, H. (2009). Compact representation for answer sets of n-ary regular queries. In *Proc. Conf. on Implementation and Application of Automata (CIAA)*, pp. 94–104.

Jedrzejowicz, J. (1999). Structural properties of shuffle automata. *Grammars*, 2(1): 35–51.

Kamimura, T. and Slutzki, G. (1981). Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1): 10–51.

Kawaguchi, K. (2001). Ambiguity detection of RELAX grammars. `http://www.kohsuke.org/relaxng/ambiguity/AmbiguousGrammarDetection.pdf`.

Kawanaka, S. and Hosoya, H. (2006). bixid: a bidirectional transformation language for XML. In *Proc. Int. Conf. on Functional Programming (ICFP)*, pp. 201–214.

Kido, H. (2007). Acceleration of biXid using attribute-element automata. Bachelor thesis, University of Tokyo.

Kirkegaard, C. and Møller, A. (2006). XACT – XML transformations in Java. In *Proc. Workshop on Programming Language Technologies for XML (PLAN-X)*, page 87.

Klarlund, N. and Møller, A. (2001). MONA version 1.4 user manual. `http://www.brics.dk/mona/`.

Koch, C. (2003). Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *Proc. Int. Conf. on Very Large Data Bases*, pp. 249–260.

Levin, M. Y. (2003). Compiling regular patterns. In *Proc. Int. Conf. on Functional Programming (ICFP)*, pp. 65–77.

Levin, M. Y. and Pierce, B. C. (2005). Type-based optimization for regular patterns. In *Proc. Int. Symp. on Database Programming Languages (DBPL)*, pp. 184– 198.

Maneth, S., Perst, T., Berlea, A., and Seidl, H. (2005). XML type checking with macro tree transducers. In *Proc. Symp. on Principles of Database Systems (PODS)*, pp. 283–294.

Maneth, S., Perst, T., and Seidl, H. (2007). Exact XML type checking in polynomial time. In *Proc. Int. Conf. on Database Theory (ICDT)*, pp. 254–268.

Meyer, A. R. (1975). Weak monadic second-order theory of successor is not elementary recursive. In *Logic Colloquium*, vol. 453 of *Lecture Notes in Mathematics*, pp. 132–154.

Milo, T. and Suciu, D. (1999). Type inference for queries on semistructured data. In *Proc. Symp. on Principles of Database Systems (PODS)*, pp. 215–226.

Milo, T., Suciu, D., and Vianu, V. (2003). Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66(1): 66–97.

Møller, A. and Schwartzbach, M. I. (2006). *An Introduction to XML and Web Technologies*. Addison-Wesley.

Møller, A., Olesen, M. Ø., and Schwartzbach, M. I. (2007). Static validation of XSL transformations. *ACM Transactions on Programming Languages and Systems*, 29(4): 21: 1–47.

Murata, M. (1997). Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing '96*, vol. 1293 of *Lecture Notes in Computer Science*, pp. 153–169.

Murata, M. (2001a). Extended path expressions for XML. In *Proceedings of Symp. on Principles of Database Systems (PODS)*, pp. 126–137.

Murata, M. (2001b). RELAX (REgular LAnguage description for XML). `http://www.xml.gr.jp/relax/`.

Murata, M. and Hosoya, H. (2003). Validation algorithm for attribute-element constraints of RELAX NG. In *Extreme Markup Languages*. Electronic edition.

Murata, M., Lee, D., and Mani, M. (2001). Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*. Electronic edition.

Neumann, A. and Seidl, H. (1998). Locating matches of tree patterns in forests. In *Proc. 18th Conf. on Foundations of Software Technology and Theoretical Computer Science*, vol. 1530 of *Lecture Notes in Computer Science*, pp. 134–145.

Neven, F. and den Bussche, J. V. (1998). Expressiveness of structured document query languages based on attribute grammars. In *Proc. Symp. on Principles of Database Systems (PODS)*, pp. 11–17.

Neven, F. and Schwentick, T. (2002). Query automata over finite trees. *Theoretical Computer Science*, 275(1–2): 633–674.

Papakonstantinou, Y. and Vianu, V. (2000). DTD inference for views of XML data. In *Proc. Symp. on Principles of Database Systems (PODS)*, pp. 35–46.

Perst, T. and Seidl, H. (2004). Macro forest transducers. *Information Processing Letters*, 89(3): 141–149.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

Planque, L., Niehren, J., Talbot, J.-M., and Tison, S. (2005). *N*-ary queries by tree automata. In *Proc. Int. Symp. on Database Programming Languages (DBPL)*, pp. 217–231.

Seidl, H. (1990). Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3): 424–437.

Slutzki, G. (1985). Alternating tree automata. *Theoretical Computer Science*, 41: 305–318.

Sperberg-McQueen, C. M. and Burnard, L. (1994). A gentle introduction to SGML. `http://www.isgmlug.org/sgmlhelp/g-index.htm`.

Suda, T. and Hosoya, H. (2005). Non-backtracking top-down algorithm for checking tree automata containment. In *Proc. Conf. on Implementation and Applications of Automata (CIAA)*, pp. 83–92.

Sulzmann, M. and Lu, K. Z. M. (2007). XHaskell – adding regular expression types to Haskell. In *Proc. Conf. on Implementation and Application of Functional Language*, vol. 5083 of *Lecture Notes in Computer Science*, pp. 75–92.

Thatcher, J. W. and Wright, J. B. (1968). Generalized finite automata with an application to a decision problem of second-order logic. *Theory of Computing Systems*, 2(1): 57–81.

Tozawa, A. (2001). Towards static type checking for XSLT. In *Proc. ACM Symp. on Document Engineering*, pp. 18–27.

Tozawa, A. (2004). Regular expression containment with `xs:all`-like operators. Unpublished.

Tozawa, A. (2006). XML type checking using high-level tree transducer. In *Functional and Logic Programming (FLOPS)*, pp. 81–96.

Tozawa, A. and Hagiya, M. (2003). XML schema containment checking based on semi-implicit techniques. In *Proc. 8th Int. Conf. on Implementation and Application of Automata*, vol. 2759 of *Lecture Notes in Computer Science*, pp. 213–225.

Vansummeren, S. (2006). Type inference for unique pattern matching. *ACM Transactions on Programming Languages and Systems*, 28(3): 389–428.

Vouillon, J. (2006). Polymorphism and XDuce-style patterns. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, pp. 49–60.

Watson, B. W. (1993). A taxonomy of finite automata construction algorithms. Computing Science report 93/43, University of Technology Eindhoven, Eindhoven, The Netherlands.

Zilio, S. D. and Lugiez, D. (2003). XML schema, tree logic and sheaves automata. In *Proc. Int. Conf. on Rewriting Techniques and Applications*, vol. 2706 of *Lecture Notes in Computer Science*, pp. 246–263.

Index