# Foundations of XML Processing

Haruo Hosoya

Draft of April 2, 2007

# Contents

# Chapter 1

# Introduction

## 1.1 Documents, Schemas, and Schema Languages

XML is a data format for describing 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 whole data set thus obtained is often called *document*. The appearance of XML documents resembles to the well-known HTML used for the design of Web pages, but is different especially in the fact that the set of tag names that are usable in documents are not fixed *a priori*.

More precisely, the set of tag names is determined by *schemas* that can freely be defined by users. A schema is a description of constraints on the structure of documents and thus defines a "subset of XML." In this sense, XML is often said to be a "format for data formats." This genericity is one of the strengths of XML. By defining schemas, each application can define its own data format, yet many applications can share generic software tools for manipulating XML documents. As a result, we have seen an unprecedented speed and range in the adoption of XML; as an evidence, an enormous number of schemas have been defined and actually used in practice, e.g., XHTML (the XML version of HTML), SOAP (an RPC message format), SVG (a vector graphics format), and MathML (a format for mathematical formulas).

One would then naturally ask: what constraints can schemas describe? The answer is that this is defined by a *schema language*. However, there is not a single schema language but are many, each having different constraint mechanisms and thus different expressivenesses. (To give only few examples, DTD (W3C), XML Schema (W3C), and RELAX NG (OASYS/ISO) are actively used by various applications.) This can confuse application programmers since they would need to decide which schema language to choose, and it can trouble tool implementators since they would need to deal with multiple schema languages. This complicated situation can also, however, be seen as a natural consequence of XML's strength, since XML is so generic that many applications want to use

it even though they have vastly different software requirements.

## 1.2 Brief History

The predecessor of XML was SGML (Standard Generalized Markup Language). It was officially defined in 1986 (ISO), but had actually been used for a long time from 60s. However, the range of users was rather limited mainly because of its complexity. Nonetheless, several important data formats like HTML and DocBook (which are now reformed as XML formats) were actually invented for SGML in this period.

XML appeared as a drastic simplification of SGML, dropping a number of features that had made the use of SGML difficult. XML was standardized in W3C (World-Wide Web Consortium) in 1998. XML's simplicity, together with the high demand for standard, non-proprietary data formats, gained a wide, rapid popularity among major commercial software vendors and open-source software communities.

At first, XML adopted DTD (Document Type Definition) as a standard schema language, which was a direct adaptation of the same schema language DTD for SGML. This decision was from the consideration for the compatibility with SGML, for which a number of software tools were already available and exploiting these was urgently needed. However, the lack of certain expressiveness that was critical for some important applications had already been pointed out by then.

Motivated for a new, highly expressive schema language, the standardization activity for XML Schema started around 1998. However, it turned out to be an extremely difficult task. One notable difficulty comes from the attempt to mix two completely different notions, regular expressions and object-orientation. Regular expressions are more traditional and had been used in DTD for a long time. The use of regular expressions is quite natural since the basis of XML (or SGML) documents is the text. On the other hand, the demand for object-orientation 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 exchanging them among applications on the Internet. These two concepts, one based on automata theory and the other based on a hierarchical model, are after all impossible to integrate smoothly and the result yielded from four years' efforts was inevitably a complex giant specification (2001). Nonetheless, a number of software programmers nowadays try to cope with the complexity.

In parallel to the standardization of XML Schema, there were several other efforts for expressive schema languages. Among others, RELAX (REgular LAnguage description for XML) aimed at yielding a simple and clean schema language in the same tradition as DTD, but based on more expressive regular tree languages rather than simpler regular string languages. The design was so simple that the standardization went very rapidly (JIS/ISO, 2000) and became a candidate for a quick substitution for DTD. Later, a refinement was done

and publicized as RELAX NG (OASYS/ISO, 2001). Although these schema languages are not yet widely used compared to DTD or XML Schema (mainly due to the less influentiality of the standard organizations), the number of their users indicates a gradual increase.

## 1.3 This Book's Topics

This book is an introduction to the fundamental concepts in XML processing, which have been developed and cultivated in academic communities. A particular focus is brought on research results related to schemas for XML. Let us outline below the covered topics.

**Schemas and tree automata** As soon as the notion of schemas is introduced, we should discuss what description mechanisms should be provided by schemas and how these can be checked algorithmically. These two questions are interrelated since, usually, the more expressive schemas are, the more difficult the algorithmics becomes. Therefore, if we want to avoid a high time complexity of an algorithm, then we should restrict the expressiveness. However, if there is a way to deal with such complexity, then the design choice of allowing the full expressiveness becomes sensible.

In Chapter 2, we give a basic definition of *schema model* and, using this, we compare several schema languages, namely, DTD, XML Schema, and RELAX NG. Then, in Chapter 3, we introduce a notion of *tree automata*, which are a finite acceptor model for trees that has rich mathematical properties. Using tree automata, we can not only efficiently check that a given document satisfies a given schema, but can also solve other various problems mentioned below that are strongly related to schemas. Note that schemas specify constraints on documents and tree automata solve the corresponding algorithmics. In general, whenever there is a specification language, there is a corresponding automata formalism. The organization of this book loosely reflects such couplings.

Later in Chapter 7, a series of efficient algorithms are presented for dealing with core problems related to tree automata (and marking tree automata mentioned below). All these algorithms are designed in a single paradigm called *on-the-fly* technique, which explores only a part of a given state space of an automaton that is needed to obtain the final result, and is extremely valuable for constructing a practically usable system.

As more advanced schema techniques, we introduce, in Chapter 8, *intersection types* and their corresponding *alternating tree automata*. The latter is not only useful for algorithmics but also helps clear formalization of certain theoretical analyses. In Chapter 12, we discuss *ambiguity* properties of schemas and automata, which can be used for detecting typical mistakes made by users. In Chapter 14, we turn our attention to a schema mechanism for describing *unordered* document structure. Unorderedness is rather tricky since it does not fit well in the framework of tree automata.

**Subtree extraction** Once we know how to constrain documents, we are next interested in how to process them, where the most important is to extract information out of an input XML tree. Chapter 4 introduces a notion of *patterns*, which are a straightforward extension of the schema model that allows "variable binders" to be associated with conditions on subtrees. Corresponding to patterns are *marking tree automata* defined in Chapter 5, which not only accept a tree but also put marks on some of its nodes. We will discuss in these chapters various design choices in matching policies and representation techniques into marking automata.

As different approaches to subtree extraction, we also present *path expressions* and *logics*. Path expressions, given in Chapter 11, are a specification for navigation to reach a target node and are widely used as the most popular notation called *XPath* (standardized by W3C). In the same chapter, we describe the corresponding automata framework called *tree-walking automata* (which also navigate in a given tree by finite states) and compare their expressivenesses with normal tree automata. Chapter 13 gives a logical approach to subtree extraction, where we explain how a *first-order (predicate) logic* and its extension called *monadic second-order logic (MSO)* can be useful for the concise specification of subtree extraction. Then, the same chapter details a relationship between the latter MSO logic and tree automata—more precisely, MSO formulae can be converted to equivalent marking tree automata and vice versa—and, thereby, makes a tight link to the other approaches.

**Tree transformation and typechecking** Subtree extraction is about analysis and decomposition of an input tree, while tree transformation combines it with production of an output tree. In this book, we do not, however, go into details of a complex tree transformation language, but rather present several tiny transformation languages with different expressivenesses. This is because the primary purpose here is actually to introduce *typechecking*, the highlight of this book.

Typechecking is adopted by many popular programming languages for statically analyzing a given program and guaranteeing certain kinds of type-related errors never to occur. Since schemas for XML are just like types in such programming languages, we can imagine performing a similar analysis on an XML transformation. However, actual techniques are vastly different from conventional ones since schemas are based on regular expressions, which are not standard in usual programming languages. Thus, completely new techniques have been invented that are based on tree automata and set operations on them.

There are two separate approaches to XML typechecking, namely, *approximate typechecking* and *exact typechecking*. In approximate typechecking, we can treat a *general* transformation language, i.e., as expressive as Turing-machines, but may have a false-negative answer from the typechecker, i.e., may reject some correct programs. In exact typechecking, on the other hand, we can signal an error if and only a given program is incorrect, but can treat only a *restricted* transformation language. This trade-off is a direct consequence of the fact that

the behavior of a Turing machine cannot be predicted precisely. In Chapter 6, we define a small but general transformation language $\mu$XDuce and describe an approximate typechecking algorithm for this language. Later in Chapter 9, we introduce a family of *tree transducers*, which are finite-state machine models for tree transformations. These models are restricted enough to perform exact typechecking and Chapter 10 describes one such algorithm. The presented typechecking algorithm treats only the simplest model called *top-down tree transducers* but already conveys important ideas in exact typechecking such as *backward inference*.

# Part I

# Basic Topics

# Chapter 2

# Schemas

A community sharing and exchanging some kind of data naturally needs a description of constraints on the structure of those data for proper communications. Such structural constraints can formally be defined by means of *schemas* and exploited for mechanical checks. A *schema language* is a "meta framework" that defines what kind of schemas can be written.

In this chapter, we will study schema models that can accommodate most of major schema languages. Though the expressivenesses of these schema languages are different in various ways, our schema models can serve as, so to speak, the least common denominator of these schemas and, in particular, some important differences can be formalized as simple restrictions defining classes of schemas. We will see three such classes each roughly corresponding to DTD, XML Schema, and RELAX NG.

## 2.1 Data Model

The whole structure of an XML document, if we ignore minor features, is a tree. Each node is associated with a label (a.k.a. tag) and often called *element* in XML jargon. The ordering among the children nodes is significant. The leaves of the tree must be text strings. More detailed constrains are not defined in this level, e.g., what label or how many children each node can have.

Let's see an example of XML document representing a "family tree," which will repeatedly appear throughout this book.

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

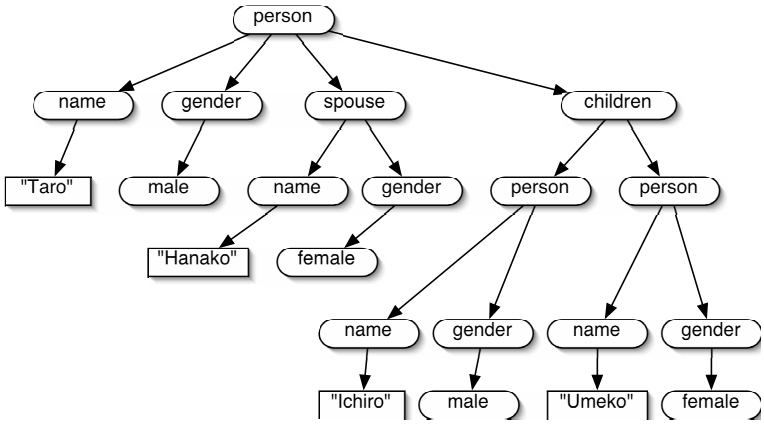



Figure 2.1: A family tree

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

Each pair of an open tag `<tag>` and its corresponding end tag `</tag>` forms a tree node. Different nodes may have different numbers of children and, in particular, some nodes have no children (namely, `male`s and `female`s). The tree structure can be depicted more visually as in Figure 2.1.

For later formal studies, let us define the data model more precisely. We first assume a finite set $\Sigma$ of *labels* ranged over by $a$, $b$, and so on. We will fix these sets throughout this book. Then, we mutually define *values* by the following grammar.

$$\text{value} \quad v \quad ::= \quad a_1[v_1], \ldots, a_1[t_n] \quad (n \geq 0)$$

The form $a[v]$ represents a node where $v$ is called *content*. A value is a sequence rather than a set, and therefore there is ordering among these trees. We write () for the value in the case $n = 0$ ("empty sequence") and write a label $a[]$ when the content is (). Text strings, though not directly handled in the data model, can easily be encoded by representing each character $c$ of a string as a single label $c[]$ with no content (e.g., `"foo"` is represented by $f[], o[], o[]$). There are also other kinds of discrepancies between our data model and real XML documents. We comment below on some of them that are worth mentioning.

**Attributes** Each element can be associated with a label-string mapping, e.g.:

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

Despite their apparent simplicity, treating attributes is rather tricky since there is no ordering among the labels and the same label cannot be declared in the same node, which makes attributes behave quite differently from elements. We will specially consider attributes in Chapter 14.1.

**Namespaces** Each label of element (and attribute) is actually a pair of a *name space* and a *local name*. This mechanism is provided for discriminating labels used in different applications and purely a lexical matter. Throughout this book, we ignore this feature (or regard each pair as a label).

**Single root** A whole XML document is actually a tree, that is, it has only one root. In the formalization, however, we always consider a sequence of trees for considering an arbitrary fragment of the document. We will soon see that this way of formalizing is more convenient.

## 2.2 Schema Model

Let us first look at an example. The following schema describes a constraint that we would naturally want to impose on family trees as in the last section.

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

The content of each label is described by the regular expression notation. A `person` contains a sequence of a `name` (with a string content) and a `gender` followed by a `spouse` and a `children`, where the `name` and the `gender` are mandatory while the `spouse` and the `children` are optional. The content of a `gender` is a `male` or a `female` that, in turn, contains the empty sequence. A `spouse` contains a `name` and a `gender`, and a `children` contains one or more `person`s.

To formalize, we first assume a set of *type names*, ranged over by $X$. A *schema* is a pair $(E, X)$ of a type definition and a "start" type name, where a *type definition* is a mapping from type names to types, written as follows

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

and *types* are regular expressions over label types and can be defined by the following grammar.

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

The type $T$ in a label type $a[T]$ is often called *content model* in XML terminology. Other common regular expression notations can be defined by short-hands:

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

For encoding the `String` type used in examples, let $\{c_1, \ldots, c_n\}$ be the characters that can constitute strings and abbreviate:

$$\texttt{String} \equiv (c_1 \,|\, \ldots \,|\, c_n)^*$$

For a schema

$$(\{X_1 = T_1; \ldots; X_n = T_n\}, X)$$

$X$ must be one of the declared type names $X_1, \ldots, X_n$. Also, each $T_i$ can freely contain any of these declared type name; thus type names are useful for both describing arbitrarily nested structures and defining type abbreviation as in the example. However, there is a restriction: any recursive use of a type name must be inside a label. As examples of this restriction, the following are disallowed

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

whereas the following is allowed.

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

This requirement is for ensuring the "regularity" of schemas, that is, correspondence with finite tree automata described later. Indeed, under no restriction, arbitrary context-free grammars can be written, which is too powerful since many useful operations on schemas like containment check (Section 3.4) become undecidable. Also, checking if a given grammar is regular or not is also undecidable and therefore we need to adopt a simple syntactic restriction that ensures regularity.

The semantics of schemas can be described in terms of the *conformance* relation $E \vdash v \in T$, read "under type definitions $E$, value $v$ conforms to type

$T$." The relation is defined by the following set of inference rules.

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

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

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

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

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

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

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

To check that a value conforms to a schema $(E, X)$, we have only to test the relation $E \vdash v \in E(X)$.

## 2.3 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 $a[T_2]$ of the same label occurring in $E$, the contents are syntactically identical: $T_1 = T_2$.

**Single** For any label types $a[T_1]$ and $a[T_2]$ of the same label occurring *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**, we mean by "in parallel" that the label types both occur in the top-level or both occur in the content of the same labeled type. For example, consider the schema $(E, X)$ where

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

Then, $a[T_1]$, $a[T_2]$, and $a[T_3]$ are in parallel each other, whereas $a[X]$ is not in parallel to any other.

These classes directly represent the expressivenesses of real schema languages. In DTD, schemas define a mapping from labels to content types and therefore obviously satisfy 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, however. It is 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. As a concrete counterexample, consider the following schema.

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

This schema is similar to the one in Section 2.2 except that it additionally allows `pet` elements. This schema 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, in **regular**, content types 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 2.2 an extra constraint where each person's spouse must have the opposite gender. This constraint is expressed by first separating male `person`s (represented by `MPerson`) and female `person`s (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. On the other hand, the schema *is* regular, of course. Note that the content type of a `person`'s `gender` depends on the `spouse`'s gender type (the grand parent's great grand child). Figure 2.2 depicts this "far dependency."

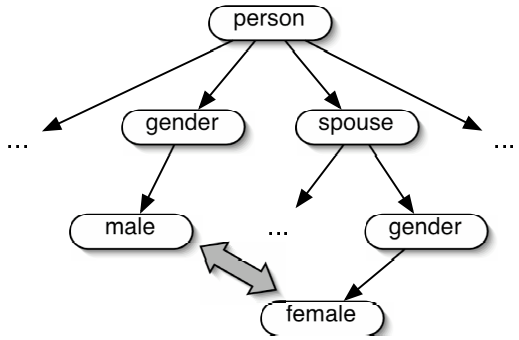



Figure 2.2: Far dependencies expressible by regular schemas

## 2.4 Bibliographic Notes

It is a classical topic to use tree grammars for describing structural constraints on unranked trees. An old formalization can be found in [12], where a sequence of unranked trees is called *hedge* and a grammar for hedges are called *hedge grammar*. A hedge is also called *forest* in [75] and *ordered forest* in [34]. The formalization and terminology in this book are taken from [51].

The specifications of various standards are available mostly on the Web: [10] for XML and DTD, [33] for XML-Schema, and [22] for RELAX NG ([72] for its predecessor RELAX). The comparison of their expressivenesses in this chapter is due to Murata, Lee, and Mali [74].

Basic materials on regular string languages and language classes beyond can be found in the classical book by Hopcroft and Ullman [44].

# Chapter 3

# Tree Automata

It is often that, for a specification framework, there is a corresponding automata formalism, with which properties on specifications can clearly be understood and efficient algorithms for relevant problems can easily be constructed. For schemas, the corresponding framework is tree automata. Tree automata are a finite-state machine model for accepting trees and are much like string automata. Indeed, they have quite similar properties such as closure properties under basic set operations. However, they also have their own special properties. For example, there are two forms of deterministic automata and one of them has a strictly weaker expressiveness than nondeterministic ones.

In this chapter, we will see the definition of tree automata, their correspondence with schemas, and their basic properties.

## 3.1 Definitions

The formalism that we study here is tree automata that accept binary trees. Since XML documents are unranked trees, where each node can have an arbitrary number of children, one would naturally wonder why considering only binary trees is enough. But, for now, we postpone this question to Section 3.2. There are other styles of tree automata such as those accepting n-ary trees, where the arity of each node (the number of its children) depends on its label, and those directly accepting unranked trees (a.k.a. hedge automata). We will not cover these in this book and refer interested readers to the bibliographic notes (Section 3.5).

We define *(binary) trees* by the following grammar (where $a$ ranges over labels from the set $\Sigma$ given in Chapter 2).

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

The height of a tree $t$, written $\mathbf{ht}(t)$, is defined by $\mathbf{ht}(\#) = 1$ and $\mathbf{ht}(a(t_1, t_2)) = 1 + \max(\mathbf{ht}(t_1), \mathbf{ht}(t_2))$. In the sequel, we will often point to an intermediate

node in a given tree. For this, first define *positions* $\pi$ by sequences from $\{1,2\}^*$, and define as below the function $\mathrm{subtree}_t$ for extracting a subtree locating at a position $\pi$ in a tree $t$.

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

Then, $\mathrm{nodes}(t)$ is the set of positions for which $\mathrm{subtree}_t$ is defined, that is, $\mathrm{dom}(\mathrm{subtree}_t)$. We also define $\mathrm{label}_t$ by

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

and $\mathrm{leaves}(t)$ by $\{\pi \in \mathrm{nodes}(t) \mid \mathrm{label}_t(\pi) = \#\}$. Further, we define the self-or-descendent relation $\pi_1 \leq \pi_2$ when $\pi_2 = \pi\pi_1$ for some $\pi$ and the (strict) descendent relation $\pi_1 < \pi_2$ by $\pi_1 \leq \pi_2$ and $\pi_1 \neq \pi_2$. Lastly, we define the document order relation $\preceq$ by the lexicographic order on $\{1,2\}^*$.

A *(nondeterministic) tree automaton* $A$ is a quadruple $(Q, I, F, \Delta)$ 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* ($F \subseteq Q$),
- $\Delta$ is a set of *transition rules* of the form

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

  where $q, q_1, q_2 \in Q$.

In a transition of the above form, 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 their runs. Given a tree automaton $A = (Q, I, F, \Delta)$ and a tree $t$, a mapping $r$ from $\mathrm{nodes}(t)$ to $Q$ is a *run* of $A$ on $t$ if

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

A run $r$ is *successful* if $r(\pi) \in F$ for each $\pi \in \mathrm{leaves}(t)$. We say that an automaton $A$ accepts a tree $t$ when there is a successful run of $A$ on $t$. Further, we define the language $L(A)$ of $A$ to be $\{t \mid A \text{ accepts } t\}$. A language accepted by some nondeterministic tree automata is called *regular tree language*; let **ND** be the class of regular tree languages. 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 $q$ is assigned to the current node and assign $q_1$ and $q_2$ to the subnodes $t_1$ and $t_2$, respectively. Finally, the run is successful when each leaf is assigned a final state. Since, in this definition,

a run (whether successful or not) goes from the root to the leaves, we call such a run *top-down run*.

Conversely, we can also consider bottom-up runs. Given a tree $t$, a mapping $r$ from nodes($t$) to $Q$ is a *bottom-up run* of $A$ on $t$ if

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

A bottom-up run $r$ is *successful* if $r(\epsilon) \in I$. Bottom-up runs can be understood as similarly to top-down ones. First, we first assign each leaf some final state. Then, for each intermediate node $a(t_1, t_2)$, we pick up a transition rule $q \to a(q_1, q_2)$ such that $t_1$ and $t_2$ are assigned $q_1$ and $q_2$, respectively, and assign $q$ to the current node. Finally, such a run is successful when the root is assigned an initial state. Obviously, the definitions of successful runs are identical for top-down and bottom-up ones.

Bottom-up runs are convenient in particular when we consider an "intermediate success" for a subtree of the whole tree. That is, if a bottom-up run $r$ of an automaton $A$ on a tree $t$, which could be a subtree of a bigger tree, maps the root to a state $q$, then we say that $A$ *accepts* $t$ *at* $q$, or simply, when it is clear which automaton we talk about, state $q$ *accepts* $t$.

**3.1.1 Example:** Let $A_{3.1.1}$ be $(\{q_0, q_1\}, \{q_0\}, \{q_1\}, \Delta)$ where

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

Then, $A_{3.1.1}$ accepts the set of trees whose every leaf is at the depth of an even number. Indeed, in any top-down run, all the nodes at odd-number depths are assigned $q_0$ and those at even-number depths are assigned $q_1$. Then, to make this run successful, all the leaves need to be at even-number depths since $q_1$ is the only final state.

## 3.2 Relationship to the Schema Model

First of all, we need to resolve the discrepancy in the data models, binary vs unranked trees. This can easily be done with by using the well-known *binarization* technique, much like Lisp's way of forming list structures by `cons` and `nil`. The following function bin formalizes translation from unranked trees to binary trees.

$$\begin{array}{lcl} \text{bin}(a[v_1], v_2) & = & a(\text{bin}(v_1),\ \text{bin}(v_2)) \\ \text{bin}(()) & = & a(\#,\ \#) \end{array}$$

That is, a value of the form $a[v_1], v_2$ in the unranked tree representation, i.e., a sequence that has the first element with label $a$, corresponds to the binary tree $a(t_1, t_2)$ whose left child $t_1$ corresponds to the content $v_1$ of the first element and whose right child $t_2$ corresponds to the remainder sequence $v_2$. The empty sequence () in the unranked tree representation corresponds to the leaf #. For

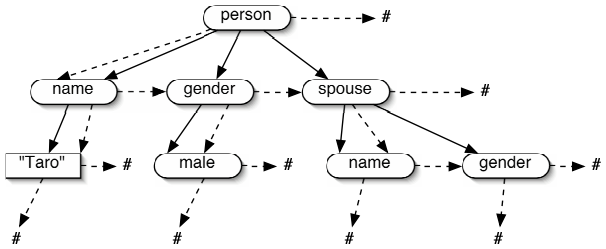



Figure 3.1: Binarization

example, Figure 3.1 shows an unranked tree (whose edges are solid lines) and its corresponding binary tree (whose edges are dotted lines).

The correspondence between schemas and tree automata is analogous. The following procedure translates a given schema to a tree automaton accepting the set of trees that conform to the schema (modulo binarization).

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

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

   That is, 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.

2. For each $i$, regard $E(X_i)$ as a regular expression whose each symbol has the form "$a[X]$" and construct a string automaton $A_i = (Q_i, I_i, F_i, \delta_i)$ from this regular expression. For this step, any construction algorithm from regular expressions to string automata can be used, e.g., the most well-known McNaughton-Yamada construction [44].

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

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

   That is, we start with $A_1$'s initial states since the start type name is $X_1$. Each transition rule $q_1 \to a(q_0, q_2)$ is constructed from a transition

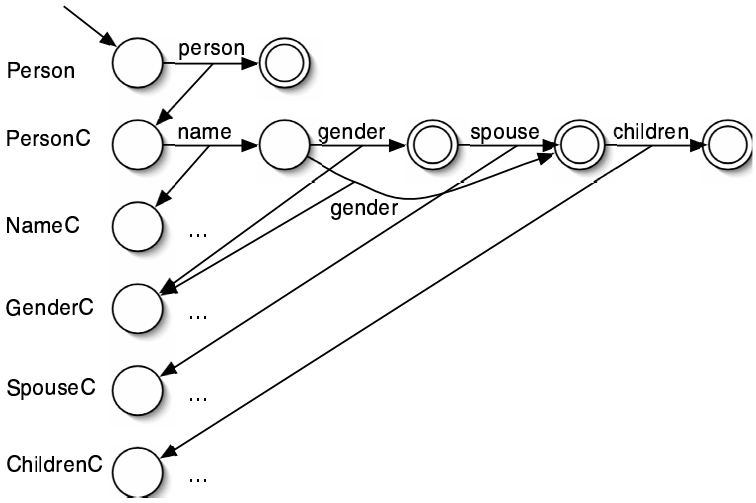
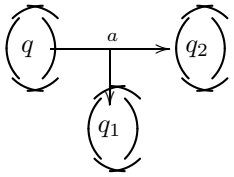



Figure 3.2: Tree automaton corresponding to the familty tree schema

rule $q_1 \xrightarrow{a[X_j]} q_2$ in some $A_i$ where $q_0$ is one of the initial states of the automaton $A_j$ corresponding to the content type $X_j$.

Note that, analogously to the tree binarization above, each transition rule has the first and second destination states for examining the first child and the next sibling, respectively. Thus, in drawing a diagram of an automaton, we use the following notation for a transition rule $q \to a(q_1, q_2)$.




As an example of translation from schemas to tree automata, consider the first schema shown in Section 2.2. First, this schema can be canonicalized as follows.

```
Person    = person[PersonC]
PersonC   = name[NameC], gender[GenderC],
              spouse[SpouseC], children[ChildrenC]?
```

We here omit the definitions of `NameC`, `GenderC`, `SpouseC`, and `ChildrenC`. We then construct the tree automaton for this schema as depicted in Figure 3.2.

**3.2.1 Exercise:** We have above seen a translation from schemas to tree automata. Construct the other direction assuming a conversion method from string automata back to regular expressions (e.g., [44]).

## 3.3 Determinism

Just like for string automata, considering determinism is important for tree automata. For example, proof of closure under complementation critically relies on a determinization procedure. However, one notable difference from string automata is that there are two natural definitions of determinism—top-down and bottom-up—and that bottom-up deterministic tree automata are as expressive as nondeterministic ones whereas top-down deterministic ones are *strictly* weaker.

### 3.3.1 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 any pair $q \to a(q_1, q_2)$ and $q \to a(q_1', q_2')$ of transitions in $\Delta$, we have $q_1 = q_1'$ and $q_2 = q_2'$.

Intuitively, a top-down deterministic automaton has at most one top-down run for any input tree. For example, the tree automaton in Example 3.1.1 is top-down deterministic. Let **TD** be the class of languages accepted by top-down deterministic tree automata.

**3.3.1 Theorem: TD $\subsetneq$ ND.**

Proof: Since **TD** $\subseteq$ **ND** is clear, it suffices to give an example of language that is in **ND** but not in **TD**. Let $A$ be $(Q, I, F, \Delta)$ where

$$\begin{array}{rcl} Q & = & \{q_0, q_1, q_2, q_3\} \\ I & = & \{q_0\} \\ F & = & \{q_3\} \\ \Delta & = & \left\{ \begin{array}{l} q_0 \to a(q_1, q_2) \\ q_0 \to a(q_2, q_1) \\ q_1 \to b(q_3, q_3) \\ q_2 \to c(q_3, q_3) \end{array} \right\}. \end{array}$$

Then,

$$L(A) = \left\{ \begin{array}{ccc} a & & a \\ \diagup\diagdown & , & \diagup\diagdown \\ b \quad c & & c \quad b \\ \diagup\diagdown \quad \diagup\diagdown & & \diagup\diagdown \quad \diagup\diagdown \\ \# \ \# \ \# \ \# & & \# \ \# \ \# \ \# \end{array} \right\}.$$

Suppose that a top-down deterministic tree automaton $A' = (Q', I', F', \Delta')$ accepting this set. Then, $I'$ must be a singleton set $\{q_0'\}$. Also, since the set contains a tree with the root labeled $a$, there is $q_0' \to a(q_1', q_2') \in \Delta'$ and no other

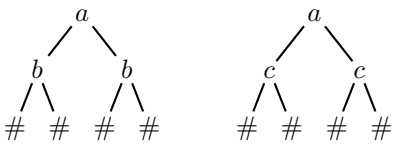
transition from $q'_0$ and $a$. Since $A'$ accepts the trees in $L(A)$ at $q'_0$, there must be the following transitions

$$q'_1 \to b(q_\#, q_\#)$$
$$q'_2 \to c(q_\#, q_\#)$$

in $\Delta$ and, similarly,

$$q'_1 \to c(q_\#, q_\#)$$
$$q'_2 \to b(q_\#, q_\#)$$

in $\Delta$ with $q_\# \in F'$. This implies that $A'$ must accept the following trees in addition to $L(A)$.




Hence, a contradiction is derived. □

Although the expressiveness is lower, top-down deterministic tree automata have algorithmically a desirable property. In particular, linear-time acceptance check can be done by a very simple algorithm (Section 7.1.1). For this reason, some major schema languages adopt restrictions on schemas for ensuring that they can easily be converted to top-down deterministic tree automata. For example, XML Schema has two restrictions, the singleness of grammar (as described in Section 2.3) and one-unambiguity; similarly, DTD takes the localness of grammar and one-unambiguity. (RELAX NG does not have either restriction.) The one-unambiguity restriction is for ensuring that a deterministic string automaton can easily be constructed from each content model and will be discussed in Chapter 12.

**3.3.2 Exercise:** Convince yourself that the construction algorithm in Section 3.2 yields a top-down deterministic tree automaton from a schema in the class **single** if the second step always produces a deterministic string automaton.

### 3.3.2 Bottom-up Determinism

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

- the set $F$ of final states is singleton, and
- for any pair $q \to a(q_1, q_2)$ and $q' \to a(q_1, q_2)$ of transitions in $\Delta$, we have $q = q'$.

Again, intuitively, a bottom-up deterministic automaton has at most one bottom-up run for any input tree. For example, the tree automaton in Example 3.1.1 is also bottom-up deterministic. Let **BU** be the class of languages accepted by bottom-up deterministic tree automata.

**3.3.3 Theorem: BU = ND.**

PROOF: Since **BU** $\subseteq$ **ND** is clear, the result follows by showing that, given a nondeterministic tree automaton $A = (Q, I, F, \Delta)$, there is a bottom-up deterministic one $A' = (Q', I', F', \Delta')$ accepting the same language. The proof proceeds by the classical "subset construction." Let

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

Then, in order to prove $L(A) = L(A')$, it suffices to show that, for any tree $t$ and any state $s \in Q'$,

$$A' \text{ accepts } t \text{ at } s \text{ if and only if } s = \{q \in Q \mid A \text{ accepts } t \text{ at } q\}.$$

In other words, each state $s = \{q_1, \ldots, q_n\}$ represents the set of trees that are accepted by all of $q_1, \ldots, q_n$ *and* are not accepted by any other state. The proof can be done by induction on the height of $t$. Details are left to Exercise 3.3.4. □

**3.3.4 Exercise:** Finish the proof of Theorem 3.3.3.

## 3.4 Basic Set Operations

In this section, we first study how to compute basic set operations—union, intersection, and complementation, and emptiness test—and see their applications.

### 3.4.1 Union, Intersection, and Complementation

Standard closure properties under union, intersection, and complementation can easily be proved in a similar way to the case of string automata. That is, we can take union by merging given two tree automata, intersection by taking the product of the automata, and complementation by using determinization used in the proof of Theorem 3.3.3.

**3.4.1 Proposition [Closure under Union]:** Given tree automata $A_1$ and $A_2$, a tree automaton $B$ accepting $L(A_1) \cup L(A_2)$ can effectively be computed.

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

**3.4.2 Proposition [Closure under Intersection]:** Given tree automata $A_1$ and $A_2$, a tree automaton $B$ accepting $L(A_1) \cap L(A_2)$ can effectively be computed.

PROOF: Let $A_i = (Q_i, I_i, F_i, \Delta_i)$ for $i = 1, 2$. Then, define the automaton $B = (Q_1 \cup Q_2, I_1 \times I_2, F_1 \times F_2, \Delta')$ where

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

To show $L(B) = L(A_1) \cap L(A_2)$, it suffices to prove that, for any tree $t$ and any states $q_1 \in Q_1$, $q_2 \in Q_2$,

$$A_1 \text{ accepts } t \text{ at } q_1 \text{ and } A_2 \text{ accepts } t \text{ at } q_2 \text{ iff } B \text{ accepts } t \text{ at } (q_1, q_2).$$

The proof can be done by induction on the height of $t$. □

In order to show closure under complementation, we need to introduce completion. A tree automaton $(Q, I, F, \Delta)$ is *complete* when, for any $a \in \Sigma$ and any $q_1, q_2 \in Q$, there is $q \in Q$ such that $q \to a(q_1, q_2) \in \Delta$. Clearly, any tree automaton can be made complete by adding a "sink" state and transitions for assigning the sink state to the trees that had no state to assign before completion. Concretely, the completion of $(Q, I, F, \Delta)$ is

$$(Q \cup \{q_s\}, I, F, \Delta \cup \{q_s \to a(q_1, q_2) \mid a \in \Sigma,\ q_1, q_2 \in Q \cup \{q_s\},\ q_s \to a(q_1, q_2) \notin \Delta\}).$$

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.

**3.4.3 Proposition [Closure under Complementation]:** Given tree automata $A$, a tree automaton $B$ accepting $\overline{L(A)}$ can effectively be computed.

PROOF: By Theorem 3.3.3, we can construct a bottom-up deterministic tree automaton $A_d$ accepting $L(A)$. Further, we can complete it and obtain $A_c = (Q_c, I_c, F_c, \Delta_c)$ also accepting $L(A)$. Then, define $B = (Q_c, Q_c \setminus I_c, F_c, \Delta_c)$. Since $B$ is also complete and bottom-up deterministic and has the same final states and transitions as $A_c$, we know that both $A_c$ and $B$ have the same, unique bottom-up run on any tree. Therefore, $A_c$ accepts $t$ iff $B$ does not accept $t$, for any tree $t$. □

Since we have already established equivalence of schemas and tree automata, the above closure properties can directly be transferred to schemas.

**3.4.4 Corollary:** Schemas are closed under union, intersection, and complementation.

### 3.4.2 Emptiness Test

Whether a tree automaton accepts some tree or not can be tested by a simple procedure. However, unlike string automata, it is not sufficient to perform a linear traversal to check whether initial states can reach final states. It is because each transition branches to two destination states and, in order for this transition to contribute to accepting some tree, both of the two destination states must accept some tree.

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

```
1: function IsEmpty(Q, I, F, Δ)
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
```

The variable $Q_{\mathbf{NE}}$ collects states that are known to be non-empty, i.e., accepting some tree. Initially, $Q_{\mathbf{NE}}$ is assigned the set of all final states since they definitely accept the single-node-tree #. Then, we look for a transition whose both destinations are non-empty states, and add its source state since we now know that this state accepts some tree. We repeat this until no state can be added. Finally, 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.

**3.4.5 Exercise:** Design concrete data structures for states and transitions for executing the above algorithm in linear time in the number of transitions.

### 3.4.3 Applications

In practice, the basic set operations studied above are quite important in maintaining and manipulating schemas. Let us briefly look at rather straightforward uses of these operations and, later, we will see their much heavier uses in XML typechecking (Chapter 6).

Union is useful when a document comes from multiple possible sources. For example, we can imagine a situation where a server program on the 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 $S_1 \cup S_2$.

Intersection is useful when requiring a document to conform to several constraints at the same time. The usefulness becomes even higher 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 $S_2$ like "document with height less than 6" or "document containing no element from a certain namespace." Then, an application program exploiting both constraints can naturally require input documents to conform to the intersection $S_1 \cap S_2$. We can further combine intersection and emptiness test for obtaining the disjointness test $S_1 \cap S_2 \stackrel{?}{=} \emptyset$. This operation is useful, for example, when checking if one schema contains some document that breaks a certain restriction represented by another schema.

Complementation is most often used in conjunction with intersection, i.e., the difference test $S_1 \setminus S_2$ $(= S_1 \cap \overline{S_2})$. Difference is useful when we want to exclude, from a "format" schema $S_1$, the documents that break the restriction expressed by a "constraint" schema $S_2$. Further, combining intersection, complementation, and emptiness test yields containment test $S_1 \stackrel{?}{\subseteq} S_2$ $(\Leftrightarrow S_1 \cap \overline{S_2} \stackrel{?}{=} \emptyset)$. One typical use of containment is, after modifying an existing schema $S_1$ and obtaining $S_2$, to check whether $S_2$ is "safely evolved" from $S_1$, that is, any document in the old format $S_1$ also conforms to the new format $S_2$. As we will see in Chapter 6, containment test is extremely important in typechecking. For example, when a value known to 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 $S_1 \subseteq S_2$. In other words, containment can be seen as a form "subtyping," often found in programming languages (e.g., Java). Despite the importance of the containment check, it is known that this operation necessarily takes an exponential time in the worst case.

**3.4.6 Theorem [[82]]:** The containment problem of tree automata is EXPTIME-complete.

In Chapter 7, we will see an "on-the-fly" algorithm that deals with this high complexity by targeting certain cases that often arise in practice.

### 3.4.4 Useless States Elimination

This is not a set operation, but it will later be useful several times and therefore we present it here. A state is useful in two cases, first when it is an empty state and, second when it is unreachable. Here, a state is *empty* when it accept no tree. Also, a state is *reachable* when it is reachable state from an initial state. More precisely, a state $q$ is 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 that, 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 $r$.

As below, we can remove both kinds of states by easy procedures.

**Removal of empty states** We construct, from a given tree automaton $A =$

$(Q, I, F, \Delta)$, the automaton $(Q', I', F', \Delta')$ where:

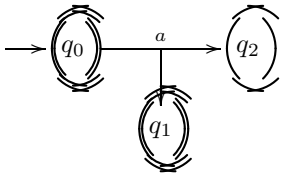
$$\begin{array}{lcl} Q' & = & \{q \in Q \mid \exists t.\ A \text{ accepts } t \text{ at } q\} \\ I' & = & I \cap Q' \\ F' & = & F \cap Q' \\ \Delta' & = & \{q \to a(q_1, q_2) \in \Delta \mid q, q_1, q_2 \in Q'\} \end{array}$$

We can recognize the empty states by using the set $Q_{\mathbf{NE}}$ non-empty states obtained from the emptiness test algorithm shown in Section 3.4.2.

**Removal of unreachable states** We construct, from a given tree automaton $A = (Q, I, F, \Delta)$, the automaton $(Q', I', F', \Delta')$ 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 that these two operations have to be done in this order and cannot be swapped. As a counterexample, consider the following automaton.




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

**3.4.7 Exercise:** Of course, we could further remove unreachable states to get rid of $q_1$. But it is sufficient to apply the two procedures each only once in the suggested order. Prove it.

## 3.5 Bibliographic Notes

A well-known, comprehensive reference to tree automata theory is Tree Automata Techniques and Applications available on-line [23]. This book manuscript also covers various theoretical results and related frameworks that are not in the scope of the present book.

An automata model that directly deals with unranked trees has been formalized in the name of *hedge automata* [12] and *forest automata* [75].

# Chapter 4

# Pattern Matching

So far, we have been interested mainly in how to *accept* documents, that is, checking whether a given document is valid or not, and, for this, we have introduced schemas and tree automata. The next question is how to *process* documents, in particular, how to extract subtrees from an input tree with a condition specified by the user. There are mainly three approaches to subtree extraction: pattern matching, path expressions, and logic. Each has its own way of specifying the extraction condition and its own advantages and disadvantages in terms of expressiveness and algorithmics. Since pattern matching can be presented along the same line as previous sections, we concentrate on it in this chapter. The other require introducing rather different formalisms and therefore we postpone it to Chapters 11 and 13.

## 4.1 From Schemas to Patterns

Schemas validate documents by constraints such as tag names and positional relationships between tree nodes. To these, pattern matching adds 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. As a simple example, we can write the following pattern.

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

Given an input value, this pattern works as follows.

- It first checks that the input value has the following type.

  ```
  person[Name, gender[male[]|female[]], Spouse?, Children?]
  ```

  This type is obtained by removing all variable binders from the pattern. If this check fails, the pattern matching itself fails.

- If it succeeds, then the pattern binds the variables `n` and `g` to the subparts of the input value that match `Name` and `(male[]|female[])`, respectively.

Although the semantics of patterns is fairly clear at a coarse level from the above informal description, it becomes more complicated 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 from which the design space of pattern matching becomes quite rich and this is the primary interest of this chapter.

## 4.2 Ambiguity

In the subsequent two sections, we will informally overview ambiguity and linearity. A formalization will follow after these.

Ambiguity refers to the property that, from some input, there are multiple ways of matching. Since the basis of patterns is regular expressions, they 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 `person`s, but can bind the variable `p` to a `person` in an arbitrary position.

What should be the semantics of ambiguous patterns? Various styles exist, as summarized below.

**Single match** The result is only one of the possible matches. The following sub-styles are common.

- **Prioritized match** The system has its built-in priority rules for uniquely determining which match to take.
- **Nondeterministic match** The system chooses an arbitrary match from all possibilities.
- **Unambiguous match** The pattern is statically required to have at most one match for any input.

**All matches** The result is the set of all possible matches.

Then, how should a language designer select one of these choices? The criteria is usually a requirement from the whole language design or a balance between the expressive power and the implementation effort. Let us review more details below.

### 4.2.1 All-matches Semantics

In general, the single-match semantics is preferred by a programming language since evaluation usually proceeds with a single binding of each variable. On the

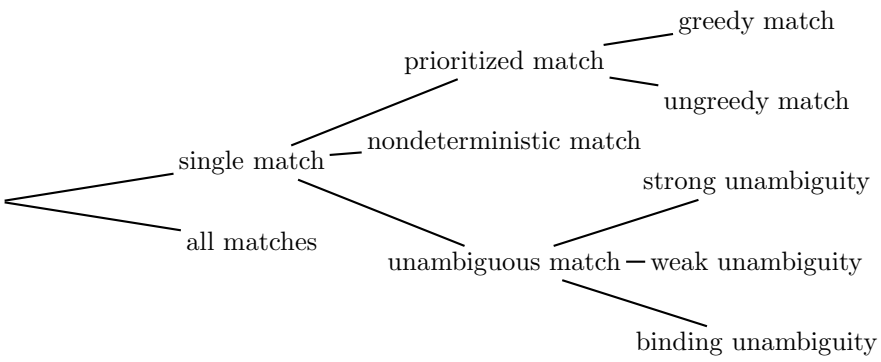



Figure 4.1: Matching policies

other hand, the all-matches semantics is more natural in a query language since it usually returns a *set* of results that satisfy a specified condition.

The most interesting kind of examples using the all-matches semantics involves retrieving data from deep places. Suppose that we want to extract, from a family tree shown in Chapter 2, all the `name` elements under `spouse`s. For this, we can first write the following recursive definitions

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

(we have not introduced "pattern definitions" yet, but it is surely writable since patterns are schemas with variable binders, as mentioned) and then match an input `person` value against `PersonX`. The first clause in the `PersonX` pattern uses the `SpouseX` pattern, where we extract the `name` stored in the `spouse` of the current `person`. The second clause uses the `ChildrenX` pattern, where we choose one of the `person`s in the `children` element for searching for a `name` in a deeper place.

### 4.2.2 Single-match Semantics

As mentioned, there are (at least) three styles of the single-match semantics, namely, the prioritized match, the nondeterministic match, and the unambiguous match.

The prioritized-match semantics can have various choices in the priority rules, though a typical design is a *greedy match*, which roughly works 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 $P*$, as many elements as possible are matched by the pattern $P$.

For example, in the pattern

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

the first `Person*` captures all the `person`s but the last and therefore the variable `p` is bound to the last `person`. We could also think of ungreedy match (the opposite of greedy match) or providing two separate operators for both greedy and ungreedy match.

While the prioritized match endows the user a full control in matching, the nondeterministic match deprives all such control from the user but instead gives a full freedom to the system. This means that there can be several implementations that behave differently and the user cannot complain for this difference. This design choice actually has a couple of advantages. First, the semantics becomes vastly simpler since we have only to specify "some match" whereas, in the prioritized match, we need a complicated description for the precise behavior. We will see how it is complicated in Section 4.4. Second, the implementation becomes much easier, in particular when we want to perform a static analysis on patterns. For example, we will see a certain type inference method for patterns in Section 6.3, where, if we take the prioritized match, we would also need to take priorities into account in the analysis in order to retain its precision.

One would prefer the unambiguous-match semantics, where the justification is that the ambiguity of a pattern is a sign of the user's mistake. For example, it is likely that the ambiguous pattern

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

shown above is actually an error in whichever semantics of single-match. Indeed, in the nondeterministic semantics, it wouldn't make sense to extract an arbitrary `person` from the sequence— the user should specify more precisely which `person` (the first one or the one satisfying a certain condition); in the greedy match, a more natural way of extracting the last `person` is to write

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

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

```
type 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 an ambiguous one, requiring disambiguation even in unnecessary situations can be a heavy burden for the user. For this reason, it can be more user-friendly to signal only a warning for ambiguity rather than an error.

## 4.3 Linearity

Linearity concerns the property that any match binds each variable exactly to one value. All the example patterns shown above are thus linear. The linearity constraint is in a sense natural when we consider adopting a pattern match facility as a part of a programming or query language, which usually expects a variable to be bound only to a single value. (To avoid confusion, note that a query language usually considers a set of bindings, but each binding maps each variable to a single value.)

If we do not impose a linearity restriction, then this gives a variable an ability to obtain a list of subtrees. This could be useful in particular with a single-match semantics. (The all-matches semantics already can collect a set of subtrees, a sensible language design would be to require linearity for avoiding complication.) To see this more clearly, let us try collecting subtrees with a single-match, linear pattern. First, since we do not have a built-in "collection" capability, let us accept the inconvenience that we need to combine pattern matching with an iteration facility equipped in the underlying language (e.g., recursive functions or `while` loops). Then, as an example, consider processing each `person` from a sequence of type `Person*`. For this, we can write the pattern

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

for extracting the first person and repeating the same pattern matching for the remainder sequence. We can extend this for a deep extraction. For example, consider again extracting `spouse`'s `name`s as in the last section. We cannot use the same pattern as before since, although we can obtain *one* `name`, we have no way to find *another* after this. The best we could do is to write a function where each call gets the `name` from the top `spouse` (if any) and recursively calls the same function to traverse each `person` in the `children`. Note that, for this, we only need shallow patterns that do not use recursive definitions. This suggests that, under the linearity constraint, recursively defined patterns are not useful in the single-match semantics. On the other hand, if we allow non-linear patterns, then we can write a recursive pattern that gathers all `spouse`s' `name`s:

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

```
ChildrenX = children[PersonX+]
```

Despite the potential usefulness of non-linear patterns, these are rather tricky to make use of since, as already mentioned, the underlying language usually expects only one value to be bound while such a pattern yields a binding of each variable to a list of values. However, there are a couple of proposals in the literature for a special language support to bridge this gap:

**Sequence-capturing semantics** A pattern actually returns a binding of each variable to the *concatenation* of the values that are matched with binder patterns. For example, the following collects all the `tel` elements from a list of `tel`s and `email`s.

```
((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 element. However, if we write the following pattern

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

to collect all the consecutive sequences of `entry`s, then the result of the match loses the information of where these sequences are separated. Nevertheless, this approach is handy and, in particular, implementation requires not much difficulty relative to linear patterns. (We will discuss more on implementation in Chapter 5)

**Iterator support** 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 $v_1$, ..., $v_n$, 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, we can write the following

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

to generate a sequence of `chunk` elements each containing an extracted sequence of consecutive `entry`s. This approach provides much more expressiveness than the previous approach, while a certain amount of extra efforts would be needed for implementation.

## 4.4 Formalization

Assume a set of pattern names, ranged over by $Y$, and a set $\mathcal{X}$ of variables, ranged over by $x$. A *pattern schema* is a pair $(F, Y)$ where $F$ is a *pattern definition*, a mapping from pattern names to patterns, of the form

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

and $Y$ is one of the pattern names $Y_1, \ldots, Y_n$. Patterns, ranged over by $P$, are defined by the following grammar.

| | | | |
|---|---|---|---|
| $P$ | ::= | $()$ | empty sequence |
| | | $a[P]$ | label pattern |
| | | $P \mid P$ | choice |
| | | $P, P$ | concatenation |
| | | $P^*$ | repetition |
| | | $Y$ | pattern name |
| | | $P$ as $x$ | variable binder |

As mentioned before, there are several styles for the semantics of patterns. In order to explain these in one framework, let us first define the *matching* relation $F \vdash v \in P \Rightarrow V;\ I$, read "under pattern definitions $F$, value $v$ matches pattern $P$ and yields binding $V$ and priority id $I$." Here, a binding $V$ is a sequence of pairs $(x \mapsto v)$ of variables and values, and a priority id $I$ is a sequence from $\{1, 2\}^*$. Here, bindings are allowed to contain multiple pairs for the same variable for treating non-linear patterns; priority ids will be used for defining the greedy or ungreedy semantics (details below). The matching relation is defined

by the following set of inference rules.

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

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

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

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

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

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

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

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

In the semantics other than the greedy- or ungreedy-match semantics, we simply ignore priority ids. For convenience, let us write $F \vdash v \in P \Rightarrow V$ when $F \vdash v \in P \Rightarrow V; I$ for some $I$.

In the nondeterministic semantics, to match a value against a pattern schema $(F, Y)$, we have only to find a binding $V$ satisfying the relation $F \vdash v \in F(Y) \Rightarrow V$. In the unambiguous semantics, patterns are guaranteed to have only one binding $V$ such that $F \vdash v \in F(Y) \Rightarrow V$, and therefore we take this binding. There are several definitions of ambiguity that ensure the uniqueness of binding. In Chapter 12, we will detail these definitions and corresponding checking algorithms. In the all-matches semantics, we collect the set of all bindings $V$ such that $F \vdash v \in F(Y) \Rightarrow V$.

In the greedy or the ungreedy match, we use priority ids. We first introduce the lexicographic ordering among priority ids for comparing them. In the greedy match, we take the smallest, that is, we find a binding $V$ such that there is the smallest priority id $I$ satisfying $F \vdash v \in F(Y) \Rightarrow V; I$. In the ungreedy match, we take the largest priority id instead. The treatment of priority ids in the above matching rules implement the priority policy informally explained in Section 4.2.2. 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 (P-Alt1) whereas it has the form $2, I$ for

$P_2$ (P-Alt2); therefore $P_1$ has a higher priority. For a concatenation pattern $P_1, P_2$, since the resulting priority id has the form $I_1, I_2$ (P-Cat), the first pattern is considered first. For a repetition pattern $P^*$, the resulting priority id has the form $1, I_1, \ldots, 1, I_n, 2$ (P-Rep) and therefore the priorities of earlier matchings become higher. In effect, $P^*$ uses $P$ as many times as necessary. As a special case where $P$ matches only a sequence of fixed length, $P$ behaves as a "longest-match."

**4.4.1 Exercise:** Find a pattern that does not behave as a longest-match even in the greedy match.

**4.4.2 Exercise:** Find a pattern that does not have the smallest priority id (i.e., for any priority id yielded by the pattern, there is a strictly smaller priority id yielded by the same pattern).

From a binding $V$ yielded by a match, the actual binding visible to the rest of the program depends on the treatment of linearity. Linearity is usually ensured by a syntactic linearity constraint consisting of the following.

- For a concatenation pattern $P_1, P_2$, no common variable appears on both sides.
- For an alternation pattern $P_1 \mid P_2$, the same set of variables appears on both sides.
- For a binder pattern ($P$ as $x$), there is no $x$ appearing in $P$
- For a repetition pattern $P^*$, there is no variable appearing in $P$.

If linearity is required on a pattern, then the yielded binding already assigns each variable to exactly one value. Note that, by the syntactic linearity condition,

- for $P_1 \mid P_2$, the domains of the bindings yielded by P-Alt1 and P-Alt2 are the same;
- for $P_1, P_2$, the domains of the yielded bindings $V_1$ and $V_2$ in P-Cat are disjoint;
- for $P$ as $x$, the domain of the binding $V$ in P-As does not contain $x$.
- for $P^*$, the domain of the binding $V$ in P-Rep is empty.

In case non-linear patterns are allowed, suppose that the bindings $(x \mapsto v_1)$, $\ldots$, $(x \mapsto v_n)$ of $x$ occur in $V$ in this order. If we use the sequence-capturing semantics, then we yield the combined binding $(x \mapsto (v_1, \ldots, v_n))$ where $v_1$ through $v_n$ are concatenated. If we use some iterator construct, we directly use $V$, though details would depend on a concrete language design.

## 4.5 Bibliographic Notes

The style of formalization of patterns here (originally called "regular expression patterns") appeared first in [50, 49] as a part of the design of the XDuce language, where the greedy-match semantics was adopted at that moment. Later, the nondeterministic and the unambiguous semantics have been investigated in [45]. A reformulation of patterns with the greedy and sequence-capturing semantics is found in the design of the ℂDuce language [39, 5]. It has been observed by [89] that the greedy-match does not necessarily coincide so-called "longest-match." As extensions of regular expression patterns with iterator supports, *regular expression filters* are introduced in [46] and ℂDuce's iterators are described in [5]. Also, a kind of non-linear patterns that bind a variable to a list of values are adopted in the design of a bidirectional XML transformation language biXid [57].

# Chapter 5

# Marking Tree Automata

In Chapter 4, we have learned pattern matching as an approach to specifying conditions for subtree extraction. The next question is how to execute such a subtree extraction. This chapter introduces marking tree automata, which work just like tree automata except that they specify how to put *marks* on each node. We will see several variations of marking tree automata and how each corresponds to pattern matching. However, efficient algorithms for executing such automata will be discussed later in Section 7.2. Beyond pattern matching, a variant of marking tree automata can actually be used for other subtree extraction specifications, and we will see such an example in the evaluation of the MSO (monadic second-order) logic in Chapter 13.

## 5.1 Definitions

Assume that the set $\mathcal{X}$ of variables (introduced in the last chapter) is finite. A *marking tree automaton* $A$ is a quadruple $(Q, I, F, \Delta)$ where $Q$ and $I$ are the same as in tree automata,

- $F$ is a set of pairs of the form $(q, X)$ where $q \in Q$ and $X \in 2^{\mathcal{X}}$, and
- $\Delta$ is a set of transition rules of the form

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

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

Note that each transition or final state is associated with a set of variables rather than a single variable. This is because we sometimes want to bind several variables to the same tree. Indeed, such situation occurs when representing, e.g., the pattern `((P as x) as y)`.

For defining the semantics, we introduce marking. Given a tree $t$, a marking $m$ is mapping from nodes($t$) to $2^{\mathcal{X}}$. In analogy to patterns, a sensible marking

is such that each variable is used exactly once. Formally, a marking $m$ is *linear* if, for each $x \in \mathcal{X}$, there is a unique node $\pi \in \text{nodes}(t)$ such that $x \in m(\pi)$.

A pair $(r, m)$ of a mapping from nodes$(t)$ to $Q$ and a marking is a *top-down marking run* of $A$ on $t$ if

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

A top-down marking run $(r, m)$ is *successful* when $(r(\pi), m(\pi)) \in F$ for each $\pi \in \text{leaves}(t)$. Bottom-up marking runs can also be defined similarly. Concretely, a pair $(r, m)$ is a *bottom-up marking run* of $A$ on $t$ if

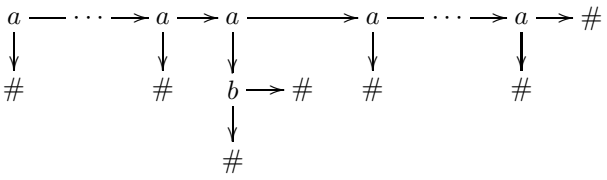
- $r(\pi) \to m(\pi) : a(r(1\pi), r(2\pi)) \in \Delta$ whenever $\text{label}_t(\pi) = a$, and
- $(r(\pi), m(\pi)) \in F$ for each $\pi \in \text{leaves}(t)$.

A bottom-up marking run $(r, m)$ is *successful* when $r$ is successful, i.e., $r(\epsilon) \in I$. Note that the definitions of successful top-down and bottom-up marking runs coincide. A marking automaton is *linear* if the marking $m$ is linear for any successful marking run $(r, m)$ of $A$ on any tree $t$. The *binding* for a marking $m$ on $t$ is a mapping from each $x$ to a node $\pi$ (or to a subtree $t'$ with $\text{subtree}_t(\pi) = t'$) where $x \in m(\pi)$.

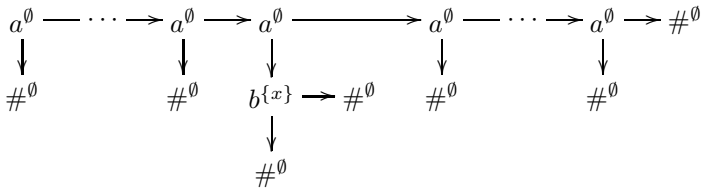
**5.1.1 Example:** Let $A_{5.1.1}$ be $(\{q_0, q_1, q_2, q_3\}, \{q_0\}, F, \Delta)$ where

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

This automaton matches a tree of the form



and binds $x$ to the node with label $b$. Indeed, the following marking on the above tree allows a successful run.



**5.1.2 Exercise:** Convince yourself that the automaton $A_{5.1.1}$ is linear.

Linearity of a marking tree automaton can actually be checked syntactically. We write $\mathrm{Var}(q)$ for the union of all the sets of variables associated with the states reachable from $q$.

**5.1.3 Lemma:** Let a marking tree automaton $(Q, I, F, \Delta)$ have all the useless states already removed (Section 3.4.4). The automaton is linear if and only if

- for each $q \to X : a(q_1, q_2) \in \Delta$,
  - $X$, $\mathrm{Var}(q_1)$, and $\mathrm{Var}(q_2)$ are disjoint each other and
  - $X \cup \mathrm{Var}(q_1) \cup \mathrm{Var}(q_2) = \mathrm{Var}(q)$, and
- for each $(q, X) \in F$, we have $X = \mathrm{Var}(q)$.

Note that the last line in the above lemma implies that a linear marking tree automaton has at most one pair $(q, X)$ in $F$ for each $q$.

**5.1.4 Exercise:** Prove Lemma 5.1.3.

## 5.2 Construction

Let us turn our attention to how to construct a marking tree automaton from a pattern. First of all, note that a marking automaton by itself can only mark a variable on a *node* in the binary tree representation. In the unranked tree representation, such a node corresponds to a *tail sequence*, i.e., 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 the tree shown in Example 5.1.1 and binds `x` to the sequence from the beginning up to the `a` element that contains `b`.

In order to handle a variable that captures non-tail sequences, a standard trick is to introduce two variables $\mathtt{x}_b$ ("beginning variable") and $\mathtt{x}_e$ ("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 range, while $\mathtt{x}_e$ captures the tail-sequence starting right after the end of `x`'s range. For example, the above pattern can be transformed to:

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

Let $v_b$ and $v_e$ be the values to which $\mathtt{x}_b$ and $\mathtt{x}_e$ are bound, respectively. Then, since $v_e$ is always a suffix of $v_b$, calculating $v$ such that $v_b = v, v_e$ gives us the intermediate sequence that we originally wanted. Let us call *tail variable* a variable binder that captures only tail sequences.

Given a pattern with only tail variables, we can construct a marking tree automaton in three steps similarly to the construction from schemas to tree automata given in Section 3.2 except that the second step is adjusted for handling variables.

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

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

2. For each $Y_i$, regarding $F(Y_i)$ as a regular expression where each symbol has the form "$a[Y]$" and variable binders may occur as subexpressions, construct a "marking string automaton" $(Q_i, I_i, F_i, \delta_i)$ with $\epsilon$-transitions. Here, the string automaton associates a set of variables on each transition and each final state, just as marking tree automata. The basis of the construction algorithm is McNaughton-Yamada's [44], which builds an automaton for each subexpression in a bottom-up way. Except for variable binders, we use same the building rules as the basis algorithm where each transition and final state is associated with an empty set of variable. For a binder pattern ($P_c$ as $x$), we modify the automaton $(Q_{P_c}, I_{P_c}, F_{P_c}, \delta_{P_c})$ built for $P_c$ as follows. We add a newly created state $q_1$ to the automaton. For each state $q$ reachable from any state in $I_{P_c}$ by $\epsilon$-path,

   - whenever there is a transition $q \xrightarrow{X:a[Y]} q'$ in $\delta_{P_c}$, we add $q_1 \xrightarrow{(X\cup\{x\}):a[Y]} q'$, and
   - whenever $(q, X)$ is in $F_{P_c}$, we add $(q_1, X \cup \{x\})$.

   Then, we let the set of the initial states be $\{q_1\}$. Finally, we apply the standard $\epsilon$-elimination.

3. Merge all the resulting marking string automata into a marking tree automaton $(Q, I, F, \Delta)$ such that

$$\begin{array}{rcl} Q & = & \bigcup_{i=1}^n Q_i \\ I & = & I_1 \\ F & = & \bigcup_{i=1}^n F_i \\ \Delta & = & \bigcup_{i=1}^n \{q_1 \to X : a(q_0, q_2) \mid q_1 \xrightarrow{X:a[Y_j]} q_2 \in \delta_i,\ q_0 \in I_j\}. \end{array}$$

**5.2.1 Exercise:** Apply the above construction algorithm to the pattern:

`(a[]*,a[b[]], (a[]* as x`$_e$`)) as x`$_b$

**5.2.2 Exercise:** Define a form of marking automata in a greedy-match semantics and give construction of such automata from greedy-match patterns.

## 5.3 Variations

So far, we introduced a formalism of marking tree automata that are most commonly found in the literature. However, other variations are possible and sometimes more convenient.

### 5.3.1 Marking on States

In this variation, marks are associated with states rather than transitions. This formalism slightly simplifies the syntax and semantics while complicates construction.

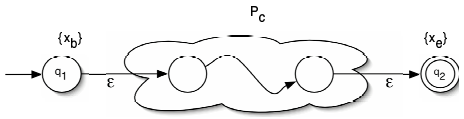
A *state-marking tree automaton* $A$ is a 5-tuple $(Q, I, F, \Delta, \Xi)$ where $Q$, $I$, $F$, and $\Delta$ are the same as in tree automata and $\Xi$ is a mapping from $Q$ to $2^{\mathcal{X}}$. Given a tree $t$, a pair $(r, m)$ of a mapping from nodes($t$) to $Q$ and a marking is a *(top-down) marking run* of $A$ on $t$ if

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

A marking run $(r, m)$ is *successful* if the run $r$ is successful, i.e., $r(\pi) \in F$ for each $\pi \in \text{leaves}(t)$. A state-marking tree automaton is linear if the marking $m$ is linear for any successful top-down marking run $(r, m)$ of $A$ on any tree $t$. Bottom-up marking runs can be defined similarly.

Construction of a state-marking tree automaton from a pattern is similar to the one presented in the last section except that a rather involved form of $\epsilon$-elimination is needed. Let a canonicalized pattern schema $(F, Y_1)$ given. At this point, patterns here may have non-tail variable binders. Then, for each $Y_i$, we construct from $F(Y_i)$ a string automaton $A_i = (Q_i, I_i, F_i, \delta_i)$ where each state has the form $(q, X)$ with $q$ being a "unique id" and $X$ being a variable set. This step has two sub-steps.

1. Just as before, convert $F(Y_i)$ to a string automaton with $\epsilon$-transitions using the usual automaton construction algorithm. Except for variable binders, we use same the building rules as the basis algorithm where each newly created state has the form $(q, \emptyset)$ where $q$ is a fresh id. For a variable binder ($P_c$ as $x$), we build the following automaton




   where we add, to the automaton built for $P_c$, a new initial state $(q_1, \{x_b\})$ and a new final state $(q_2, \{x_e\})$.

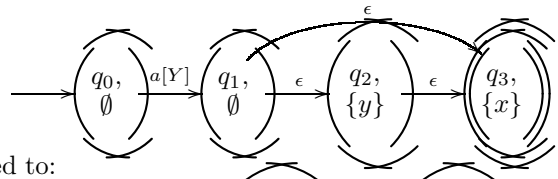
2. Eliminate $\epsilon$-transitions. We use an algorithm that slightly modifies the standard one for reflecting the fact that different sets of variables may occur depending on which $\epsilon$-path the automaton may take. First, define the $\epsilon$-closure of $(q_1, X_1)$ as the set of pairs of a state $q_n$ reachable from $(q_1, X_1)$ with $\epsilon$-transitions and the union of variable sets appearing along the path from $q_1$ to $q_n$:

$$\epsilon\text{-closure}((q_1, X_1)) = \{(q_n, X_1 \cup \ldots \cup X_n) \mid (q_1, X_1) \xrightarrow{\epsilon} (q_2, X_2),\ \ldots,\ (q_{n-1}, X_{n-1}) \xrightarrow{\epsilon} (q_n, X_n) \in \delta\}$$

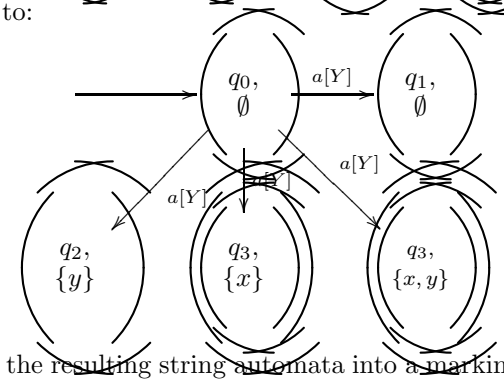
Then, for each string automaton $(Q_i, I_i, F_i, \delta_i)$ with $\epsilon$-transitions, we compute its $\epsilon$-elimination, that is, the $\epsilon$-free automaton $(Q'_i, I'_i, F'_i, \delta'_i)$ where

$$\begin{aligned}
Q'_i &= \bigcup\{\epsilon\text{-closure}(q, X) \mid (q, X) \in Q\} \\
I'_i &= \bigcup\{\epsilon\text{-closure}(q, X) \mid (q, X) \in I\} \\
F'_i &= \{(q, X) \in Q' \mid (q, X) \in F\} \\
\delta'_i &= \{(q, X) \xrightarrow{a[Y]} (q', X') \mid (q, X) \in Q'_i,\ (q, X) \xrightarrow{a[Y]} (q'', X'') \in \delta'_i, \\
&\qquad (q', X') \in \epsilon\text{-closure}((q'', X''))\}.
\end{aligned}$$

For example,




is transformed to:




Finally, we merge all the resulting string automata into a marking tree automaton $(Q, I, F, \Delta, \Xi)$ where

$$\begin{aligned}
Q &= \textstyle\bigcup_{i=1}^{n} Q_i \\
I &= I_1 \\
F &= \textstyle\bigcup_{i=1}^{n} F_i \\
\Delta &= \textstyle\bigcup_{i=1}^{n}\{\ (q_1, X_1) \to a((q_0, X_0), (q_2, X_2)) \mid \\
&\qquad (q_1, X_1) \xrightarrow{a[Y_j]} (q_2, X_2) \in \Delta_i,\ (q_0, X_0) \in I_j\} \\
\Xi((q, X)) &= X.
\end{aligned}$$

### 5.3.2 Sequence Marking

In the second variation, multiple nodes can be marked with the same variable if they are in the same sequence level. The intention is that the result is a mapping from 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 non-linear patterns in the sequence-capturing semantics (Section 4.3).

A *sequence-marking tree automaton* $A$ has the same syntax as a marking tree automaton and uses the same notion of marking runs. 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 $\pi_{i+1} = 2 \cdots 2\pi_i$ 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

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

where $\pi_1, \ldots, \pi_n$ are those in the definition of sequence-linearity. Note that, in the case $n = 0$ (that is, 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.

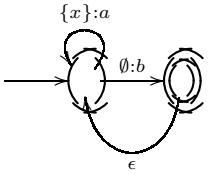
Construction of a sequence-marking tree automaton is quite simple. We only change the second step from the procedure in Section 5.2 as follows.

> In the construction of a string automaton, for each binder pattern $(P_c \text{ as } x)$, we modify the automaton built for $P_c$ by adding $x$ to the variable set of every label transition. We can apply the standard $\epsilon$-elimination on the result of the whole construction.

Note that this construction already allows us to encode non-linear patterns with the sequence-capturing semantics, though the result can possibly be non-sequence-linear. However, the construction yields a sequence-linear automaton from a non-linear pattern where each variable captures only elements that appear in the same sequence. For example, the following non-linear pattern for collecting all `a`s in a given sequence

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

can be represented by the following marking automaton (eliding the content states for brevity).




**5.3.1 Exercise:** Design a form of marking automata that can encode non-linear patterns that bind each variable to a list of values (without concatenating them).

## 5.4 Bibliographic Notes

It is quite a common idea to give marks on states of automata for identifying target nodes, e.g., [71, 61, 77] for extracting a set of nodes and [6, 81, 35, 52] for a set of tuples of nodes. A different presentation of encoding regular expression patterns by marking tree automata can be found in [49]. There are also researches on efficient compilation from marking tree automata to lower-level code [62, 63].

# Chapter 6

# Typechecking

Typechecking refers to static analysis on programs to detect possible dynamic type errors and is already standard in the area of programming languages. Since schemas in XML are analogous to data types in programming languages, we would naturally like to use this technique for verifying that XML processing programs properly manipulate and produce documents according to given schemas. However, a concrete analysis algorithm must entirely be renewed since schemas have a mathematical structure completely different from conventional data types (lists, records, object-oriented classes, and so on). In this chapter, we first overview various typechecking techniques arising from research communities and then see, as a case study, a core of the XDuce type system in detail.

## 6.1 Overview

Since typechecking aims at rejecting possibly incorrect programs, it is a language feature clearly visible to the user. Therefore we must construct a typechecker, considering 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 on what kind of type error to detect. That is, what type error may be raised from a program in the target language? The most typical kind of error is the final result being not conforming to the specified "output type." However, some languages also check intermediate results against types. For example, the XDuce language has a pattern match facility where, on failure, a match error occurs, which can be seen as a form of conformance test on an intermediate result.

The second consideration is on the precision of typechecking. From theory of computation, it is well known, that there is no algorithm that can exactly predict the behavior of a program written in a general language, that is, one equivalent to a Turing machine. From this fact, there are two approaches to obtaining a usable typechecker: (1) exact typechecking for a restricted language,

and (2) approximate typechecking (for either a general or a restricted language).

### 6.1.1 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 is, the more difficult the typechecking algorithm becomes. There is a stream of theoretical research for discovering the limit, in the name of "XML typechecking problem." Typechecking here is formulated as, given a program $P$, an input type $\tau_I$, and an output type $\tau_O$, decide whether

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

where $P(\tau_I)$ is the set of results from applying $P$ to inputs of type $\tau_I$. As types, regular tree languages or their subclasses are usually used.

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

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

It is well known, however, that, once the target language has a certain level of expressiveness, forward inference does not work since the computed output type $\tau'_O$ 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. The backward inference, instead, works as follows.

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

Some researchers have found that, in certain target languages even where the forward inference does not work, the computed input type $\tau'_I$ becomes regular and thus typechecking can be decided. We will come back to the backward inference and see a concrete algorithm in Chapter 10.

### 6.1.2 Approximate typechecking

While exact typechecking is theoretically interesting, a more realistic approach is to conservatively estimate run-time type errors. That is, programs that went through typechecking is guaranteed to be correct, whereas those that are actually innocent can possibly be refuted. However, the user has the right to ask why her program got rejected and what program would have been accepted. Here, the clearer the explanation is, the more user-friendly the language is. Thus, the main challenge in this approach is to obtain a typechecker whose error-detection

ability is high yet whose specification—called *type system*—is well understandable. Note that, in exact typechecking, no issue on specification arises since "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, based on which we calculate the types of subexpressions in a bottom-up way. This way of constructing 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 those type annotations is desirable. In Section 6.2, we will see an instructive example in this approach—μXDuce type system. This type system uses quite conventional and simple typing rules for the basic part, but makes a non-trivial effort for the inference of type annotations on variables in patterns, where computed types are guaranteed to be exact in a similar sense to the exact typechecking described in Section 6.1.1.

Since type annotations are usually tedious for the user to write, it would be nice if the type system infers 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 easy to understand. One approach is, rather than to make an absolute guarantee, to show empirical evidences that "false negative" seldom happen. Though theoreticians may not be satisfied, practitioners will find it valuable. Another approach is to find a reasonable "abstraction" of an original program and perform a exact typecheck. This has not yet been pursued extensively, but is potentially a promising approach. Chapter 10 gives relevant discussions.

## 6.2 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 by itself. However, in this section, we study only its tiny subset μXDuce as an illustrative example of how to build a simple yet powerful type system using operations on schemas. We also see an algorithm for type inference on patterns, which is also a good example showing how to construct an exact inference based on tree automata techniques.

### 6.2.1 Syntax and Semantics

We reuse the definitions of values ($v$), types ($T$), type definitions ($E$), and the conformance relation ($E \vdash T \in T$) from Section 2.2. Also, we incorporate the definitions of patterns ($P$), pattern definitions ($F$), bindings ($V$), and the matching relation ($F \vdash v \in P \Rightarrow V$) from Section 4.4. Note here that we take the nondeterministic semantics (of course, with an additional static check, this can also be the unambiguous semantics).

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 |
| | | $l[e]$ | labeling |
| | | $()$ | empty sequence |
| | | $e, e$ | concatenation |
| | | **match** $e$ **with** $P_1 \to e_1 \mid \ldots \mid P_n \to e_n$ | pattern match |

That is, we have, in addition to usual variable references and function calls, value constructors (i.e., labeling, empty sequence, 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 multi-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 $e_{main}$ for starting 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 4.2.) We assume that all patterns appearing in a given program are linear. From here on, let us fix a program $(E, G, e_{main})$.

The operational semantics of $\mu$XDuce is described by the evaluation relation $V \vdash e \Downarrow v$, read "under binding $V$, expression $e$ evaluates to value $v$." The relation is defined by the following set of rules.

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

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

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

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

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

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

Most of the rules are straightforward except for EE-MATCH. A pattern match tries the input value against the patterns $P_1$ through $P_n$ from top to bottom. Each clause matches the input with the pattern $P_i$ in the nondeterministic semantics (Section 4.2.2) and evaluates the corresponding body expression $e_i$ if the match succeeds. The semantics of the whole program is given by a value $v$ such that $\emptyset \vdash e_{main} \Downarrow v$ (there can be multiple such values due to nondeterministic pattern matching).

### 6.2.2 Typing Rules

The type system is described by using the following two important relations

- the subtyping relation $S \leq T$, and
- the typing relation $\Gamma \vdash e : T$

where $\Gamma$ is a type environment, i.e., a mapping from variables to types.

The subtyping relation $S \leq T$, read "$S$ is subtype of $T$," is for reinterpreting a value of type $S$ as having type $T$ and thus making functions able to accept multiple types of values. This feature often shows up in various programming languages. 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

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

directly formulating our intuition. 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 already discussed in Section 3.4.3. Though solving this problem takes an exponential time in the worst case, empirically efficient algorithms exist (Section 7.3).

The typing relation $\Gamma \vdash e : T$, read "under $\Gamma$, expression $e$ has type $T$," is in contrast defined by induction on the structure of expressions and works roughly as follows. Using the type of the parameter variable declared at 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 an automatic type inference for pattern variables.

Let us see 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, where the structure of each computed type parallels with that of the expression.

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

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

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

For a function call, we check, as usual, subtyping between the argument type and the declared type.

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

For a pattern match, we perform three non-trivial operations as explained below. Suppose that a pattern match

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

is given and $e$ has type $R$.

**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

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

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

**Irredundancy** This checks that each pattern is matched by some value that belongs to $R$ but does not matched by any of the preceding patterns. Here, we take the top-to-bottom matching policy into account. This can be done by testing

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

for each $i = 1, \ldots, n$. In this, we make use of the intersection operation discussed in Section 3.4.3.

**Type inference for pattern variables** This computes the best type for each variable with respect to the input type $R$. To specify more precisely, let us consider inference on the $i$-th case and write

$$R_i = R \setminus (\textbf{tyof}(P_1) | \ldots | \textbf{tyof}(P_{i-1})),$$

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

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

Let us write $R_i \vdash P \Rightarrow \Gamma$ when the above condition is satisfied. A concrete algorithm for obtaining such $\Gamma$ from $R_i$ and $P$ will be shown in Section 6.3. The "if" direction means that any value to which $x$ may be bound is predicted in $\Gamma(x)$ and therefore is necessary to construct a sound type system. The "only if" direction, on the other hand, means that $x$ may be bound to any predicted value in $\Gamma(x)$. This property makes the inference precise and thus provides the best flexibility to the user, avoiding false negative as much as possible.

Summarizing the above discussion, the typing rule for pattern matches can be formulated as follows.

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

Here, the type of the whole **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 all the function definitions in $G$ are well-typed and $e_{main}$ is well-typed under the empty type environment. 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 check, as usual, subtyping between the type $S$ of the body expression and the declared result type $T_2$.

Now, recall that our goal is to prove that a well-typed program never raises a run-time type error. This property 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

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

However, since we need to talk about bindings and type environments in the proof, we generalize the above statement: given a well-typed program, if an expression $e$ has type $T$ under a type environment $\Gamma$ and evaluates to a value $v$ under a binding $V$ conforming to $\Gamma$, then $v$ has type $T$. Here, a binding $V$ conforms to a type environment $\Gamma$, written $\Gamma \vdash V$, if $\mathrm{dom}(\Gamma) = \mathrm{dom}(V)$ and $V(x) \in \Gamma(x)$ for each $x \in \mathrm{dom}(\Gamma)$.

**6.2.1 Theorem [Type Preservation]:** Suppose that $\vdash G$ and $\Gamma \vdash e : T$ with $\Gamma \vdash V$. Then, $V \vdash e \Downarrow v$ implies $E \vdash v : T$.

**6.2.2 Exercise:** Prove Theorem 6.2.1 by induction on the derivation of the evaluation relation $V \vdash e \Downarrow v$.

To prove the second part, we need to slightly expand the formalization since we need to talk about intermediate states where a match error may occur. A usual way is first to extend the evaluation relation so that it returns an error when a match error is raised somewhere and then to prove that a well-typed program never returns an error. Note that it is not sufficient to say "not $V \vdash e \Downarrow v$" to mean that a match error occurs since it could also mean that $e$ goes into an infinite loop yielding no result.

**6.2.3 Exercise:** Redefine the evaluation relation such that an expression may return an error and then prove that a well-type expression never returns it.

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

> Under $\vdash G$, if $V \vdash e \Downarrow v$ and $E \vdash v \in T$ 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 can give to $e$ is

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

which is larger than the first one. In general, our typing rule for concatenation is not exact since it typechecks each operand independently. However, non-trivial efforts are needed for improve this imprecision. Chapter 10 will treat a relevant issue.

**6.2.4 Exercise:** Find examples of imprecisions caused by our rules for (1) subtyping and (2) pattern matches.

**6.2.5 Exercise:** Verify that the following program typechecks.

```
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` are defined somewhere. Assume also that the type `Any` denotes the set of all values.

Convince yourself in particular that the exhaustiveness and subtyping checks pass and the type inference on the pattern match gives appropriate types.

**6.2.6 Exercise:** Using each answer to Exercise 6.2.4, write a program that never raises an error but cannot be typechecked

## 6.3 Type Inference for Patterns

In the last section, we have seen the type system of $\mu$XDuce. Based on its definition, it is almost clear how to construct a typechecking algorithm for this language. The only remaining is type inference for patterns, for which we have only given the specification. This section shows a concrete algorithm that embodies this.

Since the algorithm works with tree automata, let us first rephrase the specification as follows: given a tree automaton $A$ ("input type") and a marking automaton $B$ ("pattern") containing variables $X = \{x_1, \ldots, x_n\}$, obtain a mapping $\Gamma$ from $X$ 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 $V(x_i) = u$ for some $t$ and $V$.

In fact, this specification and the algorithm shown in the next section that implements it do not work for a pattern with non-tail variables. We will see how to solve this problem in Section 6.3.2. Also, we assume that the marking automaton is linear (which is guaranteed if the automaton is converted from a linear pattern).

### 6.3.1 Algorithm

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

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

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

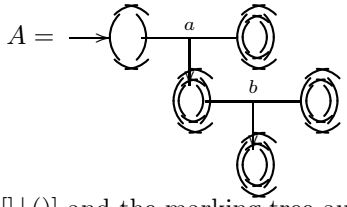
2. Eliminate useless states from $C$ by using the two steps (removal of empty states and removal of unreachable states) shown in Section 3.4.4. Let $D$ be the resulting automaton.

3. Return $\Gamma$ such that $\Gamma(x_j) = (Q_j, I_j, F_j, \Delta_j)$ where

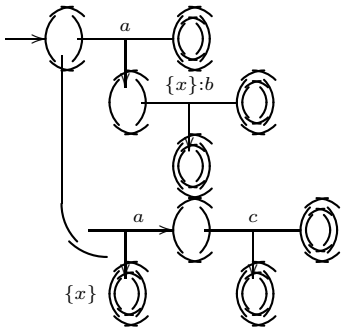
$$\begin{array}{lcl} Q_j & = & \{q_s\} \cup Q_D \\ I_j & = & \{q_s\} \cup \{q \mid (q, X) \in F_D,\ x_j \in X\} \\ F_j & = & \{q \mid (q, X) \in F_D\} \\ \Delta_j & = & \{q_s \to a(q_1, q_2) \mid q \to X : a(q_1, q_2) \in \Delta_D,\ x_j \in X\} \\ & & \cup \{q \to a(q_1, q_2) \mid q \to X : a(q_1, q_2) \in \Delta_D\} \end{array}$$

($q_s$ is a fresh state).

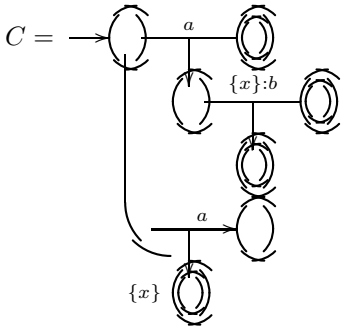
Let us illustrate the algorithm by using the following example. Consider the tree automaton

$A =$ 


representing the type $a[b[] \mid ()]$ and the marking tree automaton

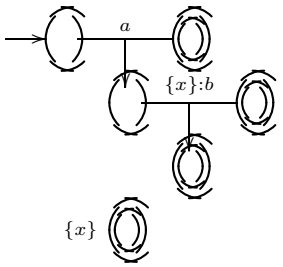
$B =$ 


representing the pattern $a[b[] \text{ as } x] \mid a[() \text{ as } x], c[]$. The first step is to take the product of $A$ and $B$, resulting in the following.

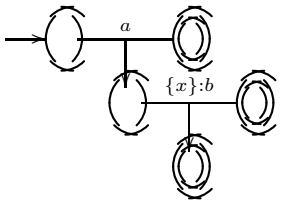
$C =$ 


The intuition behind this product construction is to obtain a marking automaton that behaves exactly the same as $B$ except that the accepted trees are restricted

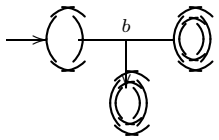
to those of $A$. In the second step, we first remove the empty states




and then the unreachable states.

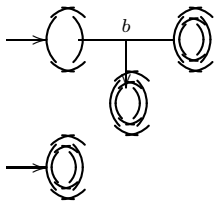
$D =$ 


In the final step, we add a new state $q_s$ and copy each transition with $x$ so that it emanates from $q_s$; we retain other transitions and final states. The result is the tree automaton

$\Gamma(x) =$ 


which represents the type $a[b[]]$. (We show only reachable states here.)

What if we skipped this step at all? Then, the result is that the type inference becomes inexact. In the above example, the final result would contain an additional initial state as in

$\Gamma(x) =$ 


which is incorrect since $x$ is never bound to the empty sequence () as a result of matching any tree from $A$ against $B$. In general, useless states carry 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 ambiguity checks in Chapter 12.

**6.3.1 Exercise:** Prove that the algorithm implements the specification. This can be done in the following steps.

1. Show that $C$ matches $t$ yielding binding $V$ if and only if $A$ accepts $t$ and $B$ matches $t$ yielding binding $V$.

2. Show, for any $q \in Q_D$, that $D$ matches $t$ at $q$ yielding binding $V$ if and only if $C$ matches $t$ at $q$ yielding binding $V$.

3. Show that $\Gamma(x_i)$ accepts $u$ if and only if $D$ matches $t$ yielding binding $V$ with $V(x_i) = u$.

### 6.3.2 Non-tail Variables

The algorithm shown above in Section 6.3.1 does not work once non-tail variables are considered. For example, see the following type and pattern

$$\begin{array}{rcl} T & = & S, S? \\ P & = & S \text{ as } x, S? \end{array}$$

for some type $S$. From the specification of type inference, the type we would like for $x$ is obviously $S$. Now, what would we get from our algorithm shown before? First, we need to transform $P$ to a pattern $P'$ with only the tail variables $x_b$ and $x_e$.

$$P' = (S, (S? \text{ as } x_b)) \text{ as } x_e$$

Then, our inference algorithm would yield

$$\begin{array}{rcl} \Gamma(x_b) & = & S? \\ \Gamma(x_e) & = & S, S?. \end{array}$$

But it is not immediately clear how to obtain the desired type $S$ from these two types. Naively, it seems that we want to compute a type in which each inhabitant $t$ is obtained by taking some tree $t_b$ from $(S, S?)$ and some tree $t_e$ from $S?$ and then cutting off the suffix $t_e$ from $t_b$. But the type we get by this calculation is

$$(S, S) \mid S \mid ()$$

which is bigger than we want.

Fortunately, by slightly modifying our inference algorithm, we can deal with non-tail variables. We keep the first two steps of our previous algorithm and change the third step as follows.

3. Return $\Gamma$ such that $\Gamma(x_j) = (Q_j, I_j, F_j, \Delta_j)$ where

$$\begin{array}{rcl} Q_j & = & \{q_s\} \cup Q_E \\ I_j & = & \{q_s\} \cup \{q \mid (q, X) \in F_E,\ x_{jb} \in X\} \\ F_j & = & \{q \mid q \to X : a(q_1, q_2) \in \Delta_E,\ x_{je} \in X\} \\ \Delta_j & = & \{q_s \to a(q_1, q_2) \mid q \to X : a(q_1, q_2) \in \Delta_E,\ x_{jb} \in X\} \\ & & \cup \{q \to a(q_1, q_2) \mid q \to X : a(q_1, q_2) \in \Delta_E,\ x_{je} \notin X\} \end{array}$$

   with a new state $q_s$.

That is, the resulting automaton $\Gamma(x_j)$ starts from states yielding transitions with $x_{jb}$ and ends with states yielding transitions with $x_{je}$; transitions are just like in $E$ except that we stop going further from states with $x_{je}$.

**6.3.2 Exercise:** Make an example for this algorithm and convince yourself that it works.

### 6.3.3 Matching Policies and Linearity

A type inference algorithm for patterns is sensitive to which matching policy is taken and whether linearity is imposed. The algorithm given is designed for (marking automata converted from) the nondeterministic semantics with the linearity requirement. It also works with the all-matches semantics (with linearity) since the specification tells 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. Obviously, the algorithm works also with the unambiguous semantics since there is only one match.

Treating the greedy match is trickier since each subpattern has "less chance" to match a value than the nondeterministic match since the priority rules may force the value to be matched with some other subpattern. Thus, the presented algorithm needs to be modified so that each state of the inferred automaton represents the values that can be matched by a pattern but *not* matched by other patterns with higher priorities.

**6.3.3 Exercise:** Construct a type inference algorithm for the greedy match. Hint: use an answer to Exercise 5.2.2; change the first "product construction" step in the above-presented inference algorithm so that each state has the form $(q, r, \{r_1, \ldots, r_n\})$ where $q$ represents a "type" of value, $r$ a "pattern," and $r_1, \ldots, r_n$ "patterns with high priorities".

Treating non-linear patterns is also possible by using a form of marking automata that properly encode such patterns. However, for sequence-capturing patterns, the sequence-linear requirement is essential since otherwise types that we would like to infer can go beyond regular tree languages.

**6.3.4 Exercise:** Construct a type inference algorithm for sequence-linear sequence-marking automata.

**6.3.5 Exercise:** Find a non-sequence-linear sequence-marking automaton where the set of values that a variable can be bound to goes beyond regularity.

**6.3.6 Exercise:** Construct a type inference algorithm for marking automata that encode non-linear patterns that bind each variable to a list of values. Use an answer to Exercise 5.3.1.

## 6.4 Bibliographic Notes

The XDuce language was a pioneer for approximate but realistic typechecking for XML [50]. An algorithm of type inference for patterns has been presented in [49] based on the greedy-match semantics and then in [45] based on the nondeterministic match. The design of XDuce has further been extended in ℂDuce [39, 5], where the main addition was higher-order functions. Integration of an XML processing language with a popular programming language has been

pursued in the projects of XHaskell [64] (with Haskell), Xtatic [41] (with C#), XJ [42] (with Java), and OCamlDuce [38] (with O'Caml).

As mentioned in the end of Section 6.1.2, there is an approach of approximate typechecking requiring no type annotations. In particular, JWig [19] and its descendent XAct [59] perform flow analysis on a Java program with an extension for XML processing constructs; XSLT Validator [69] does a similar analysis on XSLT style sheets.

For exact typechecking, we will review a detailed bibliography in Chapter 10.

# Part II

# Advanced Topics

# Chapter 7

# On-the-fly Algorithms

This chapter aims at showing efficient algorithms for some 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 that are yielded by matching 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 the subset relation). For these problems, we present several "on-the-fly" algorithms, where we explore only a part of the whole state space that needs to see in order 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 does this from the final states. In general, the bottom-up approach tends to have a better worst-case complexity, whereas the top-down often gives a higher efficiency in practical cases. We will also consider a further improvement by combining both ideas.

## 7.1 Membership Algorithms

In this section, we will see three algorithms for testing whether a given tree is accepted by a given tree automaton. The first, top-down algorithm is the one that can be obtained most naively from the semantics of tree automata but takes an exponential time in the size of the input tree in the worst case. The second, bottom-up algorithm, on the other hand, works in linear time. In this algorithm, we construct the bottom-up deterministic tree automaton on the fly where we generate only the states that are needed for deciding the acceptance of the input tree. The third, bottom-up algorithm with top-down preprocessing improves the second one by considering only the states that appear to be potentially useful from the top-to-bottom, left-to-right traversal. In practice, this improvement substantially decreases the number of states to carry around.

### 7.1.1 Top-down Algorithm

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

```
 1: function Accept(t, (Q, I, F, Δ))
 2:     for all q ∈ I do
 3:         if AcceptAt(t, q) then return true
 4:     end for
 5:     return false
 6:
 7:     function AcceptAt(t, q)
 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 AcceptAt(t1, q1) and AcceptAt(t2, q2) then return true
14:             end for
15:             return false
16:         end switch
17:     end function
18: end function
```

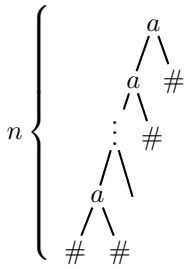
The internal function AcceptAt takes a node $t$ and a state $q$ and answers whether $q$ accepts $t$. By using this function, the main function Accept examines whether one of the initial states accepts the root. The body of the AcceptAt function is a simple rephrasing of the semantics of tree automata. That is, for a leaf #, we check that the given state $q$ is final; for an intermediate node $a(t_1, t_2)$, we try each transition of the form $q \to a(q_1, q_2)$, making a recursive call to AcceptAt with each child node $t_1$ or $t_2$ and the corresponding destination state $q_1$ or $q_2$.

Note that, when either of the two conditions in line 13 fails, we retry the same child node with another transition. Therefore, in an unfortunate case, we end up with 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.

**7.1.1 Example:** Consider the tree automaton $A_{7.1.1} = (\{q_1, q_2\},\ \{q_1\},\ \{q_1\},\ \Delta)$ where $\Delta$ consists of:

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

Then, this automaton accepts trees of the form




for any $n$. Suppose that the algorithm tries the transitions in $\Delta$ in the order written above. Then, the first transition always fails since we then have no choice other than reaching the left-most leaf with the non-final state $q_2$. Therefore we will always need to backtrack and try the second transition. Since we repeat this trial and error at each node, the whole acceptance check takes $O(2^n)$.

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 in fact be converted to a top-down deterministic tree automaton (see Section 3.3.1), and therefore, in this case, the above most naive algorithm suffices.

### 7.1.2 Bottom-up Algorithm

The next two algorithms are linear-time since they traverse each node exactly once. Of course, the most straightforward approach for linear-time acceptance check would be to determinize the given automaton. However, determinization is an exponential procedure (in the size of the automaton) and often preferred to be avoided.

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 that is assigned to this node. The following pseudo-code gives the algorithm.

```
1: function ACCEPT(t, (Q, I, F, Δ))
2:     if ACCEPTING(t)∩I ≠ ∅ then return true else return false
3:
4:     function ACCEPTING(t)
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 internal ACCEPTING function takes a node $t$ and returns the set of the states (of the nondeterministic tree automaton) that accept this node, which is the state of the bottom-up deterministic automaton that accepts the node. By using this function, the main ACCEPT function validates the input tree by seeing that the set resulting from ACCEPTING for the root contains an initial state. In the body of ACCEPTING, if the given tree is a leaf, then we return the set of final states since these are exactly the states that accept the leaf. For an intermediate node, we first collect the sets $s_1$ and $s_2$ of states that accept the children nodes $t_1$ and $t_2$ (respectively) by recursive calls to ACCEPTING, and then calculate the set $s$ of states that accept the present node $t$, which can be obtained by collecting the states that can transit to $q_1$ and $q_2$ with label $a$, where $q_1$ is in $s_1$ and $q_2$ is in $s_2$. Note that exactly the same calculation appears in the determinization procedure (each transition constructed in determinization has the form $s \to a(s_1, s_2)$).

**7.1.2 Exercise:** Consider again the automaton $A_{7.1.1}$ in Example 7.1.1 and the input tree there. Confirm that the bottom-up algorithm can validate the membership in a linear time.

Since this algorithm visits each node only once, we can avoid a potential catastrophic behavior as in the top-down algorithm. However, we in return 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 sorted lists, then each takes $O(|Q|^3)$ in the worst case (the factor $|Q|^2$ for looping on $s_1$ and $s_2$ and the factor $|Q|$ for inserting an element to a sorted list). Although the worst case would rarely happen, it would still be better to keep low the number of states to be manipulated at each tree node.

**7.1.3 Exercise:** Let $A_{7.1.3} = (\{q_1, q_2, q_3\},\ \{q_1\},\ \{q_1, q_3\},\ \Delta)$ where $\Delta$ consists of:

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

($A_{7.1.3}$ contains two additional transitions compared to $A_{7.1.2}$.) Apply the bottom-up algorithm to $A_{7.1.3}$ with the same input tree. Observe that $q_3$ is always contained in the set of states returned by each recursive call but is discarded in the end.

### 7.1.3 Bottom-up Algorithm with Top-down Preprocessing

The third algorithm—called bottom-up with top-down preprocessing—can be regarded as a combination of the two algorithms described above. Thus, the following views on the first two are quite useful for understanding the third.

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

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

```
1: function Accept(t, (Q, I, F, Δ))
2:     if Accepting(t, I) ≠ ∅ then return true else return false
3:
4:     function AcceptingAmong(t, r)
5:         switch t do
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 = AcceptingAmong(t2, {q2 | q → a(q1, q2) ∈ Δ, q ∈
    r, q1 ∈ s1})
11:            let s = {q | q → a(q1, q2) ∈ Δ, q ∈ r, q1 ∈ s1, q2 ∈ s2}
12:            return s
13:        end switch
14:    end function
15: end function
```

The internal function AcceptingAmong takes a tree node $t$ and a set $r$ of states and returns the set of states that accept this node *and* are in $r$. Thus, the main function Accept can check the whole acceptance by passing the root and the set of initial states to AcceptingAmong and examining if the resulting set is non-empty. The body of AcceptingAmong has a similar structure to that of the Accepting function of the bottom-up algorithm. If the given tree node $t$ is a leaf #, then we return the set $F$ of final states restricted to the passed set $r$. If $t$ is an intermediate node $a(t_1, t_2)$, then we first call AcceptingAmong recursively for the first child $t_1$. Here, we also pass the set of possible states for $t_1$, which is the set of all first destinations $q_1$ of transitions $q \to a(q_1, q_2)$ with $q \in r$. (We ignore $q_2$ at this moment.) Next, we call AcceptingAmong for the second child $t_2$. The set of possible states for $t_2$ is computed similarly except

that we can now use the result $s_1$ from the first call, which gives the possible states for $q_1$; thus we obtain the set of all second destinations $q_2$ of transitions of the form $q \to a(q_1, q_2)$ with $q \in r$ *and* $q_1 \in s_1$. Finally, we calculate the result of the original call in a similar way to ACCEPTING in the last section except that the source state $q$ can be constrained by $r$.

This algorithm does not improve the worst-case complexity of the bottom-up algorithm. However, experience tells that the size of the set of states passed around is usually small—typically 1 or 2—and therefore the improvement in practical cases is considerable.

**7.1.4 Exercise:** Apply the bottom-up algorithm with top-down preprocessing to the automaton $A_{7.1.3}$ in Exercise 7.1.3 with the same input tree. Confirm that the set of states returned by each recursive call never contains $q_3$.

## 7.2 Marking Algorithms

Let us next consider executing marking tree automata, that is, computing the set of all possible bindings for a given input tree. Like in the last section, we first see a naive, potentially exponential-time top-down algorithm and then a cleverer, linear-time bottom-up algorithm. More precisely, the bottom-up algorithm here works in time $O(|t| + |\Gamma|)$ where $|t|$ and $|\Gamma|$ are the input and the output sizes, respectively. The idea behind the linear time in the input size is similar to the case of membership. However, a simplistic adaptation of the bottom-up membership algorithm to the marking problem incurs the cost $O(|t|^k)$ where $k$ is the number of variables. This is unavoidable in case the output set itself already has size $O(|t|^k)$. However, the simplistic algorithm may have such cost even when the output is actually not so large. The presented algorithm improves this by a technique called *partially lazy set operations*.

In this section, we assume that a given marking tree automaton $(Q, I, F, \Delta)$ has no useless state and is linear. Recall by Lemma 5.1.3 that the following holds in this case:

- for each $q \to X : a(q_1, q_2) \in \Delta$,
  - $X$, $\mathrm{Var}(q_1)$, and $\mathrm{Var}(q_2)$ are pairwise disjoint and
  - $X \cup \mathrm{Var}(q_1) \cup \mathrm{Var}(q_2) = \mathrm{Var}(q)$, and
- for each $(q, X) \in F$, we have $X = \mathrm{Var}(q)$.

We introduce some notations for convenience. The form $\{X \mapsto t\}$ stands for the mapping $\{x \mapsto t \mid x \in X\}$. For a set of mappings, define its *domain* as the union of the domains of all the mappings. The "product" on sets $\Gamma_1$, $\Gamma_2$ of mappings with disjoint domains is defined such that

$$\Gamma_1 \times \Gamma_2 = \{\gamma_1 \cup \gamma_2 \mid \gamma_1 \in \Gamma_1,\ \gamma_2 \in \Gamma_2\}.$$

### 7.2.1 Top-down Algorithm

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

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

The internal function MATCHAT takes a tree node $t$ and a state $q$, and returns a set of all bindings that can be yielded by matching $t$ against $q$. (Note that, in case the matching fails, the function returns the empty set.) Thus, the main function MATCH returns the set of bindings from matching the input tree $t$ against any initial state.

The body of MATCHAT works as follows. When $t$ is a leaf, we return the set of bindings $\{X \mapsto t\}$ for each variable set $X$ associated with $q$ according to the set $F$. Linearity implies that $F$ can actually associate at most one variable set for each state. Therefore we return here either an empty or a singleton set. When $t$ is an intermediate node $a(t_1, t_2)$, we collect and union together the sets of bindings obtained for all the transitions $q \to X : a(q_1, q_2)$ emanating from $q$. Here, each set of bindings is computed by recursively calling MATCHAT with each child $t_1$ or $t_2$ and the corresponding destination state $q_1$ or $q_2$, respectively, and then combining, by product, the results with the binding $\{X \mapsto t\}$ for the current node. Linearity ensures that the domains of sets of mappings that we combine here are pairwise disjoint.

Note that an obvious optimization is possible (analogously to the top-down membership algorithm): when computing the clause

$$\{\{X \mapsto t\}\} \times \textsc{MatchAt}(t_2, q_2) \times \textsc{MatchAt}(t_2, q_2),$$

if the first call to MATCHAT returns $\emptyset$, then we can skip this second call since the whole clause will return $\emptyset$ whatever the second call would return. Nevertheless, by exactly the same argument as in the top-down membership algorithm, the worst-case time complexity is exponential to the size of the input.

### 7.2.2 Bottom-up Algorithm

We can construct a bottom-up marking algorithm using the same idea in the bottom-up membership algorithm, where rather than trying to match each node with a *single* state, we have collected the set of all states that match each node, thus achieving the goal by only a single scan of the whole tree. In marking, we additionally associate a set of bindings to each collected state. The following shows the pseudo-code for the algorithm. Here, regard the operators $\uplus$ and $\otimes$ as the same as $\cup$ and $\times$ for the moment.

```
1: function MATCH(t, (Q, I, F, Δ))
2:     let m = MATCHING(t)
3:     return ⨄_{q∈I} m(q)
4:
5:     function MATCHING(t)
6:         switch t do
7:           case #:
8:             return {q ↦ {{X ↦ t} | (q, X) ∈ F} | q ∈ Q}
9:           case a(t1, t2):
10:            let m1 = MATCHING(t1)
11:            let m2 = MATCHING(t2)
12:            return {q ↦ ⨄_{(q→X:a(q1,q2))∈Δ} {{X ↦ t}} ⊗m1(q1) ⊗m2(q2) | q ∈ Q}
13:        end switch
14:    end function
15: end function
```

The internal function MATCHING takes a tree node $t$ and returns a mapping from each state $q$ to a set of bindings that can be yielded by matching $t$ against $q$ (where a non-matching state maps to an empty set). Thus, the main function MATCH first calculates such a mapping for the root node and then collects all bindings given by the resulting mapping at the initial states. The body of MATCHING is remarkably similar to the function MATCHAT in the top-down algorithm. The only difference is essentially "currification" (in the $\lambda$-calculus terminology), that is, MATCHAT takes both a node $t$ and a state $q$ as arguments, whereas MATCHING takes only a node $t$ but returns a mapping from states. Though, in MATCHAT, the case of an intermediate node $a(t_1, t_2)$ moves out the recursive calls to MATCHAT from the construction of the mapping since these calls do not depend on $q$—in this way, we can make the algorithm perform only a linear scan.

However, a linear scan does not necessarily mean a linear time in the input size. In fact, it can take $O(|t|^k)$ in the worst case where $|t|$ is the size of the input tree $t$ and $k$ is the number of all variables. For example, consider the

marking automaton $(\{q_0, q_1, q_2\}, \{q_0\}, F, \Delta)$ where $\Delta$ consists of

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

and

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

That is, the state $q_3$ accepts any tree with label $a$ and yields no binding. Based on this, the state $q_1$ accepts any tree but binds $x$ to any node in the tree and the state $q_2$ is similar except that it binds $y$. Thus, the state $q_0$ binds $x$ to any node in the left subtree and $y$ to any node in the right subtree. Hence, for an input tree $t$ of the form $a(t_1, t_1)$, the size of the output, i.e., the cardinality of the result set of bindings, is $|t_1|^2 \approx O(|t|^2)$. This example can easily be generalized to $k$ variables.

Unfortunately, this non-linear complexity is intrinsic to the problem itself since, in the worst case, the output size is already $O(|t|^k)$ and therefore just enumerating it necessarily takes so. However, there are cases where the output size is not so big but a naive implementation of the above algorithm takes non-linear time. Consider the automaton $(\{q_A, q_0, q_1, q_2\}, \{q_A\}, F, \Delta')$ where $\Delta'$ contains the following in addition to $\Delta$ in the previous automaton.

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

What happens for an input tree $t$ where all nodes have label $a$? The output set contains only $\{x \mapsto t,\ y \mapsto t\}$ since the second transition above yields this mapping while the first one does not match when the root has label $a$. However, in the bottom-up algorithm, we compute exactly the same set up to the state $q_0$ as in the previous automaton, but discard it entirely at the root node since there is no transition that leads to $q_0$ from $q_A$ via label $a$. Consider another example: $(\{q_A, q_B, q_0, q_1, q_2\}, \{q_A\}, F, \Delta'')$ where $\Delta''$ contains the following in addition to $\Delta$ in the first automaton.

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

Again, what happens for an input tree $t$ with all nodes labeled $a$? The output set contains only $\{x \mapsto t,\ y \mapsto t\}$ for a different reason. This time, there *is* a

transition that leads to $q_0$ from $q_A$ via $a$. But the right subtree of the root does match the state $q_B$ (which has only a transition with $b$) and therefore returns the empty set; the product of some set and the empty set is also empty. Thus, the computation up to the state $q_0$ is, again, completely discarded.

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

- computation at an unreachable state and
- computation that will be combined to an unmatched state.

Fortunately, these are the only waste and both can be eliminated by using partially lazy set operation, explained next. This technique achieves linear-time complexity in the size of the output set. The idea behind is as follows.

- We delay the computations of $\uplus$ and $\otimes$ up to the root node where we perform only the computations that are relevant to the initial states. This eliminate wasted computations at unreachable states.
- We eagerly compute, however, the $\uplus$ and $\otimes$ operations when one of the arguments is the empty set. This eliminates, in particular, wasted computations that are combined with the empty set by $\otimes$.

Concretely, we use the following data structures to symbolically represent set operations. We define possibly empty sets $s_\emptyset$ and non-empty sets $s$ by the following grammar.

$$\begin{aligned} s_\emptyset &::= \emptyset \mid s \\ s &::= \mathrm{union}(s) \mid \mathrm{prod}(s) \mid \{\{X \mapsto t\}\} \end{aligned}$$

Then, in the bottom-up algorithm, we interpret the operations $\uplus$ and $\otimes$ so that they construct symbolic set operations for possibly empty sets.

$$\begin{aligned} \emptyset \uplus s_\emptyset &= s_\emptyset \\ s_\emptyset \uplus \emptyset &= s_\emptyset \\ s_1 \uplus s_2 &= \mathrm{union}(s_1, s_2) \end{aligned}$$

$$\begin{aligned} \emptyset \otimes s_\emptyset &= \emptyset \\ s_\emptyset \otimes \emptyset &= \emptyset \\ s_1 \otimes s_2 &= \mathrm{prod}(s_1, s_2) \end{aligned}$$

Here, we carry out the actual computation if one of the arguments is empty but otherwise delay it. Finally, we force the delayed set operations returned by the algorithm by using the following function eval:

$$\begin{aligned} \mathrm{eval}(\emptyset) &= \emptyset \\ \mathrm{eval}(\mathrm{union}(s_1, s_2)) &= \mathrm{eval}(s_1) \cup \mathrm{eval}(s_2) \\ \mathrm{eval}(\mathrm{prod}(s_1, s_2)) &= \mathrm{eval}(s_1) \times \mathrm{eval}(s_2) \\ \mathrm{eval}(\{\{X \mapsto t\}\}) &= \{\{X \mapsto t\}\} \end{aligned}$$

**7.2.1 Lemma:** Let a marking tree automaton fixed. With partially lazy set operations, the function MATCHING, given a tree $t$, returns a mapping $m$ from states to symbolic sets in $O(|t|)$ time such that computing $\Gamma = \text{eval}(m(q))$ takes in $O(|\Gamma|)$ time for each $q$.

PROOF: By induction on the structure of $t$. □

**7.2.2 Exercise:** Finish the proof of Lemma 7.2.1.

**7.2.3 Corollary:** For a fixed marking tree automaton, the bottom-up algorithm with partially lazy set operations takes in $O(|t| + |\Gamma|)$ time where $|t|$ and $|\Gamma|$ are the input and the output sizes, respectively.

**7.2.4 Exercise:** Combine with the present algorithm the idea used in the bottom-up membership algorithm with top-down preprocessing.

## 7.3 Containment

Let us now turn our attention to the problem of checking whether the languages of given two tree automata are in the containment relation. This operation is known to be EXPTIME-complete despite their importance in XML processing (Sections 3.4.3 and 6.2). We give two on-the-fly algorithms for the containment problem, first a bottom-up one and then a top-down one.

### 7.3.1 Bottom-up Algorithm

It is certainly possible to check the containment between two tree automata $A$ and $B$ by combining basic set operations presented in Section 3.4. That is, we first take the complement of $B$, then take its intersection with $A$, and finally check the emptiness of the resulting automaton. Disadvantages of this approach are that each intermediate step generates a whole big tree automaton and that the state space of the automaton after the second step may not entirely be explored by the emptiness test in the final step, thus the rest states becoming waste.

The bottom-up algorithm eliminates these disadvantages by combining all the three steps. Let two automata $A = (Q, I, F, \Delta)$ and $B = (Q', I', F', \Delta')$ given as inputs. For readability, let us first combine the first two steps, which yields the following automaton $(Q'', I'', F'', \Delta'')$ representing the difference $\mathcal{L}(A) \cap \overline{\mathcal{L}(B)}$:

$$
\begin{aligned}
Q'' &= Q \times 2^{Q'} \\
I'' &= \{(q, p) \mid q \in I,\ p \cap I' = \emptyset\} \\
F'' &= \{(q, F') \mid q \in F\} \\
\Delta'' &= \{(q, p) \to a((q_1, p_1), (q_2, p_2)) \mid \ p_1, p_2 \subseteq Q',\ q \to a(q_1, q_2) \in \Delta, \\
&\qquad\qquad\qquad\qquad\quad p = \{q' \mid q' \to a(q'_1, q'_2) \in \Delta',\ q'_1 \in p_1,\ q'_2 \in p_2\}\}
\end{aligned}
$$

Combining this with the emptiness check given in Section 3.4.2, we obtain the following algorithm.

```
1: function IsSubset((Q, I, F, Δ), (Q', I', F', Δ'))
2:     Q_⊑ ← {(q, F') | q ∈ F}
3:     repeat
4:         for all p1, p2 ⊆ Q', q → a(q1, q2) ∈ Δ s.t. (q1, p1), (q2, p2) ∈ Q_⊑ do
5:             let p = {q' | q' → a(q'1, q'2) ∈ Δ', q'1 ∈ p1, q'2 ∈ p2}
6:             Q_⊑ ← Q_⊑ ∪ {(q, p)}
7:         end for
8:     until Q_⊑ does not change
9:     return Q_⊑ ∩ {(q, p) | q ∈ I, p ∩ I' = ∅} = ∅
10: end function
```

### 7.3.2 Top-down Algorithm

Next, we design a top-down algorithm that explores the state space from the initial states. This algorithm can be viewed as the combination of computation of a difference automaton and emptiness test just like in the bottom-up algorithm. However, we need to take different ways of taking the difference and testing the emptiness since the emptiness algorithm and the difference computation that the last section's algorithm is based on are both intrinsically bottom-up and therefore combining them naturally yields a bottom-up algorithm, not a top-down one. In particular, it is rather difficult to explore the difference automaton in a top-down way since the initial states can be many from the first place and, moreover, the definition of $\Delta''$ above 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 $p = \{q' \mid q' \to a(q'_1, q'_2) \in \Delta',\ q'_1 \in p_1,\ q'_2 \in p_2\}$? We would end up looping over all sets of states for $p_1$ and $p_2$.

Thus, we use a difference automaton that is easier to explore from the initial states with a top-down emptiness check algorithm.

**Difference automaton** The idea here is to compute a difference automaton without going through determinization. Let two automata $A = (Q, I, F, \Delta)$ and $B = (Q', I', F', \Delta')$ given. We aim at computing an automaton with the state space $Q \times 2^{Q'}$ where each state $(q, p)$ denotes the set of trees accepted by the state $q$ in $A$ but by no state from $p$ in $B$. Based on this intention, the initial states should each have the form $(q, I')$ for $q \in I$ since we want the result automaton to accept the set of trees accepted by an initial state of $A$ but by no initial state of $B$. Also, the final states should each have the form $(q, p)$ with $q \in F$ and $p \cap F' = \emptyset$ since, in order for a leaf to be accepted by this state, it must be accepted by a final state of $A$ but by no final state of $B$.

Now, what transitions should we have from each state $(p, q)$? For this, let us consider a necessary and sufficient condition for a tree $a(t', t'')$ to be accepted by $(p, q)$. By our intended meaning of $(p, q)$, this holds if and only if

**(A)** there is $q \to a(q', q'') \in \Delta$ such that $t'$ and $t''$ are each accepted by $q'$ and $q''$, and

**(B1)** there is no $q \to a(q', q'') \in \Delta'$ such that $t'$ and $t''$ are each accepted by $q'$ and $q''$.

The condition (B1) is equivalent to:

**(B2)** for all $q \to a(q', q'') \in \Delta'$, either $t'$ is not accepted by $q'$ or $t''$ is not accepted by $q''$.

Let us write

$$\Delta'(p, a) = \{(q', q'') \mid q \in p,\ q \to a(q', q'') \in \Delta\}.$$

Then, the condition (B2) can be transformed to:

**(B3)** for all $i = 1, \ldots, n$, either $t'$ is not accepted by $q'_i$ or $t''$ is not accepted by $q''_i$

where $\Delta'(p, a) = \{(q'_1, q''_1), \ldots, (q'_n, q''_n)\}$, which is the same as

**(B4)** for some $J \subseteq \{1, \ldots, n\}$, we have that $t'$ is accepted by no state from $\{q'_i \mid i \in J\}$ and $t''$ is accepted by no state from $\{q''_i \mid i \in \overline{J}\}$.

(The notation $\overline{J}$ stands for $\{1, \ldots, n\} \setminus J$.) This exchange between conjunction and disjunction (i.e., "for all" becomes "for some" and "or" becomes "and") can intuitively be understood by seeing the following table.

| | 1 | 2 | 3 | 4 | 5 | 6 | |
|---|---|---|---|---|---|---|---|
| $q'_i$ | × | | × | × | | | $J$ |
| $q''_i$ | | × | | | × | × | $\overline{J}$ |

That is, the column-wise reading corresponds to the condition (B3) and the row-wise reading (B4). Now, combining (A) and (B4) yields the condition that

> for some $(q \to a(q', q'')) \in \Delta$ and some $J \subseteq \{1, \ldots, n\}$, the subtree $t'$ is accepted by $q'$ but by no state from $\{q'_i \mid i \in J\}$ and the subtree $t''$ is accepted by $q''$ but by no state from $\{q''_i \mid i \in \overline{J}\}$.

By recalling our intended meaning of each state in the result automaton, we see that the transitions that we want from the state $(q, p)$ has the form

$$(q, p) \to a((q', \{q'_i \mid i \in J\}), (q'', \{q''_i \mid i \in \overline{J}\}))$$

for each $q \to a(q', q'') \in \Delta$ and $J \subseteq \{1, \ldots, n\}$.

In summary, from the given automata, we compute the difference automaton $C = (Q'', I'', F'', \Delta'')$ such that:

$$\begin{array}{lcl}
Q'' & = & Q \times 2^{Q'} \\
I'' & = & \{(q, I') \mid q \in I\} \\
F'' & = & \{(q, p) \mid q \in F,\ p \cap F' = \emptyset\} \\
\Delta'' & = & \{(q, p) \to a((q', \{q'_i \mid i \in J\}), (q'', \{q''_i \mid i \in \overline{J}\})) \mid q \to a(q', q'') \in \Delta, \\
 & & \qquad \Delta'(p, a) = \{(q'_1, q''_1), \ldots, (q'_n, q''_n)\},\ J \subseteq \{1, \ldots, n\}\ \}
\end{array}$$

**7.3.1 Lemma:** The automaton $C$ accepts a tree $t$ iff $A$ accepts $t$ but $B$ does not.

PROOF: By induction on the structure of $t$. □

**7.3.2 Exercise:** Finish the proof of Lemma 7.3.1. Use the intuitive explanation given above.

**Top-down emptiness** How can we check emptiness 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 IsEmptyAt(q)
2:     if q ∈ F then return false
3:     for all q → a(q1, q2) ∈ Δ do
4:         if not IsEmptyAt(q1) and not IsEmptyAt(q2) then return false
5:     end for
6:     return true
7: end function
```

Unfortunately, this function may not terminate when the automaton has a loop.

A standard solution is to stop when we see a state for the second time. Concretely, we maintain a set of already encountered states, which we assume to be empty; we call the set *assumption set*, written $Q_{\text{asm}}$. In the algorithm, before examining a state $q$, we check whether $q$ is already in the assumption set $Q_{\text{asm}}$ and, if so, we immediate return true. Otherwise, we first assume $q$ to be empty (putting it to $Q_{\text{asm}}$) and proceed to checking with its finalness and transitions. Note, however, that, in case the first recursive call (IsEmptyAt($q_1$)) fails, we need to revert the assumption set as it used to be before the call since the assumptions that were made during this call may be incorrect. Thus, we obtain the following pseudo-code for the top-down emptiness checking algorithm.

```
1: function IsEmptyTop(Q, I, F, Δ)
2:     Q_asm ← ∅
3:     for all q ∈ I do
4:         if not IsEmptyAt(q) then return false
5:     end for
6:     return true
7:
8:     function IsEmptyAt(q)
9:         if q ∈ Q_asm then return true
10:        Q_asm ← Q_asm ∪ {q}
11:        if q ∈ F then return false
12:        for all q → a(q1, q2) ∈ Δ do
```

```
13:             let Q' = Q_asm
14:             if not IsEmptyAt(q_1) then
15:                 Q_asm ← Q'
16:                 if not IsEmptyAt(q_2) then return false
17:             end if
18:         end for
19:         return true
20:     end function
21: end function
```

Since the algorithm is a little subtle, we give a formal correctness proof. First, it is easy to see termination.

**7.3.3 Lemma:** The top-down emptiness test algorithm terminates for any input.

PROOF: At each call to the internal function IsEmptyAt, we add a state $q$ to $Q_{\text{asm}}$ whenever $q \notin Q_{\text{asm}}$ and never remove an element until it returns. Since $Q_{\text{asm}}$ never gets elements other than those in $Q$ and since $Q$ is finite, the function terminates with call depth less than $|Q|$. □

Then, we prove that the algorithm detects exactly whether a given automaton is empty or not.

**7.3.4 Lemma:** The top-down emptiness test algorithm returns true for an automaton if it accepts no tree and returns false otherwise.

PROOF: From Lemma 7.3.3, the result follows by proving that

- if IsEmptyAt($q$) with $Q_{\text{asm}}$ returns true with ${Q_{\text{asm}}}'$, then the emptiness of all states in $Q_{\text{asm}}$ implies the emptiness of all states in ${Q_{\text{asm}}}' \cup \{q\}$, and
- if IsEmptyAt($q$) with $Q_{\text{asm}}$ returns false, then $q$ is not empty.

The proof can be done by induction on the call depth. □

**7.3.5 Exercise:** Finish the proof of Lemma 7.3.4.

**7.3.6 Exercise:** Note that the proof sketch tells that when the IsEmptyAt function returns false for a state, this state is definitely non-empty regardless to the assumption set. This suggests that our algorithm can be modified so that it maintains a global "false set" that holds all the detected non-empty states. Do it.

**Top-down containment** Combining the difference automaton construction and the top-down emptiness check presented above, we obtain our top-down containment algorithm.

```
1: function IsSubsetTop((Q, I, F, Δ), (Q', I', F', Δ'))
2:     Q_asm ← ∅
```

```
3:     for all q ∈ I do
4:         if not IsSubsetAt(q, I′) then return false
5:     end for
6:     return true
7:
8:     function IsSubsetAt(q, p)
9:         if (q, p) ∈ Q_asm then return true
10:        Q_asm ← Q_asm ∪ {(q, p)}
11:        if q ∈ F and p ∩ F′ = ∅ then return false
12:        for all q → a(q′, q″) ∈ Δ do
13:            let {(q′_1, q″_1), ..., (q′_n, q″_n)} = Δ′(p, a)
14:            for all J ⊆ {1, ..., n} do
15:                let Q′ = Q_asm
16:                if not IsSubsetAt(q_1, {q′_i | i ∈ J}) then
17:                    Q_asm ← Q′
18:                    if not IsSubsetAt(q_2, {q″_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 made a rather intricate construction of our top-down containment algorithm. Then, what is an advantage of it? One answer is that it enables a powerful *state sharing* technique.

Before showing the technique, let us motivate it by the following example. Suppose 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 `person` label is not shown. Observe, however, that, whatever the content is, the containment should hold. Thus, we would wish that our containment algorithm decides this without looking at `person`'s content—this would indeed be a big advantage when the content type is a large expression.

Here is how our top-down algorithm can achieve this by a slight modification. First of all, in practical situations, it is often possible to made two input automata $(Q, I, F, \Delta)$ and $(Q', I', F', \Delta')$ share their states such that $Q = Q'$, $F = F'$, and $\Delta = \Delta'$ (but possibly $I \neq I'$). In particular, this can easily be done when the automata come from schemas as in the last paragraph: in the example, we can share the states that correspond to the content of `person` label.

When the top-down algorithm receives two automata that share their states, an argument $(q, p)$ passed to the internal function IsSubsetAt 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 any state from $p$.

**7.3.7 Exercise:** Indeed, in the example comparing `Person?` and `Person*`, we will encounter the pair $(q, \{q\})$ where $q$ represents the content of `person`. Confirm this.

Notice that the bottom-up algorithm cannot use the same technique since this explores the state space from final states, which means that, by the time when we encounter a pair $(q, p)$ with $q \in p$, we have already seen all states below and therefore it is too late to take any action.

## 7.4 Bibliographic Notes

Variants of the three algorithms for the membership checking are described in [74], which presents algorithms directly dealing with schemas instead of tree automata.

A number of efforts have been made for finding a fast marking algorithm. Early work by Neven has given a linear-time, two-pass algorithm for the unary case (marking with a single variable) based on boolean attribute grammars [76]. Then, Flum, Frick, and Grohe have found a linear-time (both in the sizes of the input and the output), three-pass algorithm for the general case [35]. The bottom-up, single-pass algorithm shown in this chapter is a refinement of the last algorithm with partially lazy set operations; this was given in [52] with a slightly different presentation. Meanwhile, several other algorithms have also been found that either have higher complexity or have linear-time complexity for more restricted cases [61, 6, 81], though each of these has made orthogonal efforts in either implementation or theory. The cited papers above 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 is in [51]. Several improvements for its backtracking behavior have also been proposed: [84] based on containment dependencies and [37] based on a local constraint solver. A completely different, bottom-up algorithm based on binary decision diagrams is described in [88].

# Chapter 8

# Alternating Tree Automata

In Chapter 3, we have established that nondeterministic tree automata are closed under intersection and that this closure property can be useful in certain applications. However, one clumsiness there is that when we want an intersection of given automata, we need to *calculate* another automaton by means of product construction. In this chapter, we introduce *alternating tree automata*, a formalism that directly incorporates intersection into tree automata. While ordinary tree automata support disjunction via nondeterminism, alternating tree automata additionally support conjunction. This framework does not solve any difficult problem in the complexity-theoretic sense. However, thanks to explicit intersection operations, this makes it much shorter to represent some automata and thus much easier to understand them. This is extremely useful in particular when we need to construct an automaton that satisfies a certain complicated condition and we will see such examples in Chapters 10 and 11.

## 8.1 Definitions

An *alternating tree automaton* $\mathcal{A}$ is a quadruple $(Q, I, F, \Phi)$ 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 $\Phi$ is a function that maps each pair $(q, a)$ of a state and a label to a formula, where formulas are defined by the following grammar.

$$\phi \quad ::= \quad \downarrow_i q \mid \phi \vee \phi \mid \phi \wedge \phi \mid \top \mid \bot$$

(with $i = 1, 2$). In particular, note that a formula with no occurrences of $\downarrow_i q$ evaluates naturally to a Boolean. Given an alternating tree automaton $\mathcal{A} = (Q, I, F, \Phi)$, we define acceptance of a tree by a state: $\mathcal{A}$ accepts a leaf $\#$ by a state $q$ when $q \in F$; also, $\mathcal{A}$ accepts an intermediate node $a(t_1, t_2)$ by a state $q$ when $(t_1, t_2) \vdash \Phi(q, a)$ holds, where the judgment $(t_1, t_2) \vdash \phi$ is 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 $\mathcal{A}$ accepts $t_i$ by $q$.

Then, $\mathcal{A}$ accepts a tree $t$ if $\mathcal{A}$ accepts $t$ by some initial state $q_0 \in I$. We define the language $L(\mathcal{A})$ of $\mathcal{A}$ by the set of trees accepted by $\mathcal{A}$; a tree language accepted by some alternating tree automaton is called *alternating tree language*; let **ATL** be the class of alternating tree languages.

**8.1.1 Exercise:** Let $\mathcal{A}_{8.1.1} = (\{q_0, q_1, q_2, q_3\}, \{q_0\}, \{q_1, q_2, q_3\}, \Phi)$ where

$$\begin{array}{rcl} \Phi(q_0, b) & = & \downarrow_1 q_1 \wedge \downarrow_1 q_2 \\ \Phi(q_1, a) & = & \downarrow_1 q_1 \wedge \downarrow_2 q_3 \\ \Phi(q_2, a) & = & \downarrow_1 q_3 \wedge \downarrow_2 q_2 \end{array}$$

and the other cases are set to $\bot$. Find all trees accepted by this automaton.

Since we have added built-in conjunction in the automata framework, it would also be sensible to add "intersection types" in the schema syntax. Let us write $T_1 \& T_2$ to denote the intersection of the sets denoted by $T_1$ and $T_2$. Converting a type containing intersections to an alternating tree automaton is entirely straightforward since, basically, the only additional work is to translate each intersection type to a conjunction. However, a care is needed in case an intersection type is concatenated to some other type, e.g., $(T_1 \& T_2), T_3$. In such a case, we might want to first combine sub-automata corresponding to $T_1$ and $T_2$ by conjunction and then somehow connect the result to another sub-automaton corresponding to $T_3$. However, this does not work since the sub-automata for $T_1$ and $T_2$ 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 rewrite the expression $(T_1 \& T_2), T_3$ to $((T_1, T_3) \& (T_2, T_3))$, thus avoiding such issue of synchronization. Unfortunately, this changes the meaning. For example, the following

```
(a[] & (a[],b[])), (b[]?)
```

denotes the empty set since the intersection of `a[]` and `a[],b[]` is empty, whereas the above-suggested rewriting yields the following

```
((a[],b[]?) & (a[],b[],b[]?))
```

which is not empty: it contains `a[],b[]`. For these reasons, an intersection type that is concatenated to another type needs to be expanded to an automaton without using conjunction (i.e., by using product construction).

## 8.2 Relationship with Tree Automata

Clearly, alternating tree languages include regular tree languages since alternating tree automata have both disjunction and conjunction whereas ordinary tree automata have only disjunction. To see the converse, we need to convert an alternating tree automaton to a regular tree automaton. There are 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 one by first using the former algorithm and then applying the determinization procedure (Section 3.3.2). However, this approach takes a double exponential time. The latter algorithm, on the other hand, directly determinizes a given alternating tree automaton and takes only exponential time.

**8.2.1 Algorithm [ATL-to-ND]:** Given an alternating tree automaton $\mathcal{A} = (Q, I, F, \Phi)$, we construct a nondeterministic tree automaton $\mathcal{M} = (S, I', F', \Delta)$ by a subset construction:

$$\begin{array}{lcl} S & = & 2^Q \\ I' & = & \{\{q_0\} \mid q_0 \in I\} \\ F' & = & \{s \mid s \subseteq F\} \\ \Delta & = & \{s \to a(s_1, s_2) \mid q \in s,\ (s_1, s_2) \in \mathrm{DNF}(\Phi(q, a))\} \end{array}$$

Intuitively, each state $\{q_1, \ldots, q_n\}$ in the resulting automaton $\mathcal{M}$ denotes the intersection of all the states $q_1, \ldots, q_n$ in the original automaton $\mathcal{A}$. In the above, $\mathrm{DNF}(\phi)$ computes $\phi$'s disjunctive normal form by pushing intersections under unions and regrouping atoms of the form $\downarrow_i q$ for a fixed $i$; the result is formatted as a set of pairs of state sets:

$$\begin{array}{lcl} \mathrm{DNF}(\top) & = & \{(\emptyset, \emptyset)\} \\ \mathrm{DNF}(\bot) & = & \emptyset \\ \mathrm{DNF}(\phi_1 \wedge \phi_2) & = & \{(s_1 \cup s_1', s_2 \cup s_2') \mid (s_1, s_2) \in \mathrm{DNF}(\phi_1),\ (s_1', s_2') \in \mathrm{DNF}(\phi_2)\} \\ \mathrm{DNF}(\phi_1 \vee \phi_2) & = & \mathrm{DNF}(\phi_1) \cup \mathrm{DNF}(\phi_2) \\ \mathrm{DNF}(\downarrow_1 q) & = & \{(\{q\}, \emptyset)\} \\ \mathrm{DNF}(\downarrow_2 q) & = & \{(\emptyset, \{q\})\} \end{array}$$

For example, if this function yields $\{(\{s_1, s_2\}, \{s_3\}), (\{s_4\}, \emptyset)\}$ then this denotes $(\downarrow_1 s_1 \wedge \downarrow_1 s_2 \wedge \downarrow_2 s_3) \vee \downarrow_1 s_4$.

**8.2.2 Theorem:** $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{A})$

PROOF: To prove the result, it suffices to show the following for any tree $t$ and any state $s$ from $\mathcal{M}$.

> $\mathcal{M}$ accepts $t$ at $s$ if and only if $\mathcal{A}$ accepts $t$ at all $q \in s$.

The proof can be done by induction on the height of the tree $t$. □

**8.2.3 Exercise:** Finish the proof of Theorem 8.2.2.

**8.2.4 Exercise:** Construct a nondeterministic tree automaton from the alternating tree automaton $\mathcal{A}_{8.1.1}$ in Exercise 8.1.1 by the conversion algorithm **ATL-to-ND**. Remove all states unreachable from the initial states.

**8.2.5 Corollary: ATL = ND.**

PROOF: Clearly, **ND** $\subseteq$ **ATL**. Indeed, a nondeterministic tree automaton $\mathcal{M} = (S, I, F, \Delta)$ can be seen as an equivalent alternating tree automaton with the same set of states, the same set of initial states, and the same set of final states by defining the function $\Phi$ as $\Phi(s, a) = \bigvee_{(s \to a(s_1,s_2)) \in \Delta} \bigwedge_{i=1,..,n} \downarrow_i s_i$. The converse **ATL** $\subseteq$ **ND** is already proved by Theorem 8.2.2. $\square$

**8.2.6 Algorithm [ATL-to-BU]:** Given an alternating tree automaton $\mathcal{A} = (Q, I, F, \Phi)$, we construct a bottom-up tree deterministic automaton $\mathcal{M}' = (S, I', F', \Delta)$ by another subset construction:

$$\begin{array}{lcl} S & = & 2^Q \\ I' & = & \{s \subseteq S \mid s \cap I \neq \emptyset\} \\ F' & = & \{F\} \\ \Delta & = & \{s \leftarrow a(s_1, s_2) \mid s = \{q \in Q \mid (s_1, s_2) \vdash \Phi(q, a)\}\} \end{array}$$

The intuition behind is the same as in determinization for nondeterministic tree automata (Section 3.3.2), that is, each state $s = \{q_1, \ldots, q_n\}$ denotes the set of trees that are accepted by all of $q_1, \ldots, q_n$ *and* are not accepted by any other state. In the above, 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 $\phi$ holds by interpreting each $\downarrow_i q$ as "$q$ is a member of the set $s_i$." Clearly, the whole procedure takes at most exponential time and the resulting automaton $\mathcal{M}'$ is bottom-up deterministic.

A critical observation for understanding this procedure is the following relationship between the judgments $(t_1, t_2) \vdash \phi$ and $(s_1, s_2) \vdash \phi$. That is, when we set each $s_i$ as the set of states that accept $t_i$, the judgment $(t_1, t_2) \vdash \phi$ holds if and only if $(s_1, s_2) \vdash \phi$ holds.

**8.2.7 Lemma:** $(t_1, t_2) \vdash \phi$ iff $(s_1, s_2) \vdash \phi$ where $s_i = \{q \in Q \mid t_i \text{ is accepted at } q\}$ for $i = 1, 2$.

PROOF: By induction on the structure of $\phi$. $\square$

**8.2.8 Theorem:** $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{M}')$.

PROOF: To prove the result, it suffices to show the following for any tree $t$ and any state $s$ from $\mathcal{M}$.

$$\mathcal{M} \text{ accepts } t \text{ at } s \text{ if and only if } s = \{q \in Q \mid \mathcal{A} \text{ accepts } t \text{ at } q\}.$$

The proof can be done by induction on the height of the tree $t$. Note that we have seen exactly the same statement in the proof of Theorem 3.3.3. □

**8.2.9 Exercise:** Finish the proof of Theorem 8.2.8. (Note that the inductive case uses Lemma 8.2.7.)

Now, one may wonder why we should care about the first algorithm (**ATL-to-ND**) since the second algorithm has the same complexity yet yields a bottom-up deterministic automaton. One answer is that the first one often yields a smaller automaton than the second one, for purposes that do not need determinism such as emptiness test, it is enough. We can see an instance from the following exercise.

**8.2.10 Exercise:** Construct a deterministic tree automaton from the alternating tree automaton $\mathcal{A}_{8.1.1}$ by using the conversion algorithm **ATL-to-BU**. Remove all states unreachable from the final states. Then, compare the resulting automaton with the one from Exercise 8.2.4.

## 8.3 Basic Set Operations

Since union and intersection are already built in, we only consider here membership test, emptiness test, and complementation. These can certainly be achieved by going through ordinary tree automata. However, algorithms would be more useful and possibly more efficient if they directly manipulate alternating tree automata.

### 8.3.1 Membership

We can obtain a "top-down algorithm" in the same style as in Section 7.1.1 directly from the semantics defined in Section 8.1. However, this has a potential blow-up by traversing the same node many times. This is caused not only by disjunction but also by conjunction. That is, to test whether a node is accepted by a disjunction $A \vee B$, we need to examine whether it is accepted by $A$ and, if it fails, then by $B$, just as we did in Section 7.1.1; in addition, for a conjunction $A \wedge B$, we need to test whether a node is accepted by $A$ and, if it succeeds, then by $B$.

We can also construct a linear-time "bottom-up algorithm" in a similar way to Section 7.1.2. In this, we create some states of a bottom-up deterministic tree automaton that are needed for deciding whether a given tree is accepted. The following shows pseudo-code for the algorithm.

**8.3.1 Algorithm [ATL-mem-bottom-up]:**

```
1: function ACCEPT(t, (Q, I, F, Φ))
2:     if ACCEPTING(t)∩I ≠ ∅ then return true else return false
3:
4:     function ACCEPTING(t)
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
```

That is, ACCEPTING$(t)$ returns the set of states that accept $t$. Thus, for a leaf $\#$, we return the set of final states; for a node $a(t_1, t_2)$, we first compute $s_1$ and $s_2$ as the set of states that accept $t_1$ and $t_2$, respectively, and then collect the set of states $s$ accepting $a(t_1, t_2)$ from $s_1$ and $s_2$, which can be obtained by finding all $s$ whose $\Phi(s, a)$ holds under $(s_1, s_2)$ (Lemma 8.2.7). Note also that line 11 uses the same computation done in the algorithm **ATL-to-BU**.

**8.3.2 Exercise:** Construct a "bottom-up membership algorithm with top-down preprocessing" for alternating tree automata in the same way as Section 7.1.3.

### 8.3.2 Complementation

For nondeterministic tree automata, complementation requires an exponential-time computation (Theorem 3.4.3). It is not the case, however, for alternating tree automata: complementation is linear time thanks to the fact that they have both disjunction and conjunction. This implies that, if we need to frequently perform complementation, alternating tree automata are the right representation.

**8.3.3 Algorithm [ATL-compl]:** Let an alternating tree automaton $\mathcal{A} = (Q, I, F, \Phi)$ be given. Without loss of generality, we can assume that $I$ is a singleton set $\{q_0\}$ (otherwise, we can combine all initial states, taking the union of all their transitions). Then, we construct the alternating tree automaton $\mathcal{A}' = (Q, I, Q \setminus F, \Phi')$ such that $\Phi'(q, a) = \mathrm{flip}(\Phi(q, a))$ where $\mathrm{flip}(\phi)$ is defined as follows.

$$
\begin{array}{lcl}
\mathrm{flip}(\top) & = & \bot \\
\mathrm{flip}(\bot) & = & \top \\
\mathrm{flip}(\phi_1 \wedge \phi_2) & = & \mathrm{flip}(\phi_1) \vee \mathrm{flip}(\phi_2) \\
\mathrm{flip}(\phi_1 \vee \phi_2) & = & \mathrm{flip}(\phi_1) \wedge \mathrm{flip}(\phi_2) \\
\mathrm{flip}(\downarrow_i q) & = & \downarrow_i q
\end{array}
$$

That is, we modify the given alternating automaton so that each state $q$ becomes to denote the complement of the original meaning. For this, we negate each formula in the transition by flipping between $\wedge$ and $\vee$, between $\top$ and $\bot$, and between final and non-final states.

**8.3.4 Theorem:** Complementing an alternating tree automaton can be done in linear time.

PROOF: To show the result, it suffices to prove that $\mathcal{A}$ accepts a tree $t$ at $q$ if and only $\mathcal{A}'$ does not accept $t$ at $q$, for any $t$ and $q$. The proof can be done by induction on the height of $t$. □

**8.3.5 Exercise:** Extend alternating tree automata for directly supporting complementation operations (thus complementation becomes a constant-time operation). Modify the algorithms **ATL-to-ND**, **ATL-to-BU**, and **ATL-mem** so as to work on such automata.

### 8.3.3 Emptiness

First, it is easy to prove that checking emptiness of an alternating tree automaton takes an 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. Nevertheless, experience tells that a practically efficient algorithm can be constructed by dealing with alternating automata.

**8.3.6 Theorem:** 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 (Theorem 8.2.2 and then checking emptiness of the resulting automaton can be done in polynomial time (Section 3.4.2). The problem is EXPTIME-hard since the containment problem for tree automaton, which is EXPTIME-complete (Theorem 3.4.6), can be reduced by polynomial time to the present problem, where the reduction uses linear-time complementation (Section 8.3.2). □

**8.3.7 Exercise:** Based on the conversion algorithm **ATL-to-ND**, derive a top-down algorithm for checking emptiness in a similar fashion to Section 7.3.2.

**8.3.8 Exercise:** Based on the determinization algorithm **ATL-to-BU**, derive a bottom-up algorithm for checking emptiness in a similar fashion to Section 7.3.1. Discuss advantages and disadvantages between the top-down (Exercise 8.3.7) and the bottom-up algorithms.

## 8.4 Bibliographic Notes

Alternating tree automata have first been introduced and investigated in [83]. This notion has been exploited in the area of XML typechecking. For example, the implementation of ℂDuce [39, 5] extensively uses alternating tree automata (with negation operation) as an internal representation of types for XML. A top-down algorithm for emptiness check can be found in [37].

# Chapter 9

# Tree Transducers

Tree transducers are finite-machine models for tree transformation. In Chapter 3, we have introduced tree automata for accepting trees; tree transducers add the capability to produce trees. On the other hand, in Chapter 6, we have described the simple tree transformation language $\mu$XDuce; this language is in fact quite powerful—as expressive as Turing machines; tree transducers are more restricted because states that they can work with are only finite.

An important implication from the restriction is that tree transducers can only inspect the input tree and can never look at a part of the output tree that they are producing. This might seem quite restrictive since no intermediate data structure can be used (other than their finite states themselves). Nevertheless, this restriction is not too unrealistic. Indeed, the XSLT language—the currently most popular language for XML transformation—has this property. In addition, thanks to this restriction, we have nice properties that otherwise hardly hold, among which the most important is the exact typechecking property, that is, a typechecking algorithm exists such that it signals *if and only if* the given program raises an error for some input; recall that $\mu$XDuce has only the "if" direction. We will go into details in this topic in Chapter 10.

Tree transducers have a quite long history where the most basic ones date back to early 70's. By now, so many kinds have been defined and investigated that it is hopeless to cover all of them. In this book, we will consider a few simplest ones, namely, top-down tree transducers and some of their extensions, and see their basic properties.

## 9.1 Top-down Tree Transducers

Top-down tree transducers express a form of transformation that traverses a given tree from the root to the leaves where, at each node, we produce a fragment of the output tree determined by the label of the current node and the current state. Since a state can then be seen as a rule from a label to a tree fragment, we call a state *procedure* from now on.

When considering the tree transducer family, we often consider nondeterminism just like in automata. When a transducer is nondeterministic, it may have a choice of multiple rules at a single procedure and thus may produce multiple results. There are several reasons why we consider nondeterminism. For one thing, as we will see, this actually changes the expressiveness. For another thing, nondeterminism can be used as a means of "abstracting" more complex computation so as to make it easier to perform static analysis on it. For example, if we consider a higher-level language that has an if-then-else expression whose boolean condition may be too complicated to analyze. In such a case, we can abstract such a conditional by regarding that both branches can nondeterministically happen. Exact typechecking on such an abstract program can be seen as a form of approximate typechecking that is different from what has been presented in Chapter 6 and can even be better in some cases. We will see more details in Chapter 10.

Formally, a *top-down tree transducer* $\mathcal{T}$ is a triple $(P, P_0, \Pi)$ where $P$ is a finite set of procedures, $P_0 \subseteq P$ is a set of initial procedures, and $\Pi$ is a set of (transformation) rules each having either of the following forms

$$p(a(x_1, x_2)) \to e \qquad \text{(node rule)}$$
$$p(\#) \to e \qquad \text{(leaf rule)}$$

where $p \in P$. 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 resulted from evaluating any of $p$'s rules. In the evaluation, we first match and deconstruct the input tree with the head of the rule and give the subordinate values to the rule, that is, we pass the pair of $t$'s children to a node rule and a dummy (written $\_$) to a leaf rule. We jointly write $\rho$ to mean either a pair of trees or a dummy.

$$\begin{array}{lcl} [\![p]\!](a(t_1, t_2)) & = & \bigcup_{(p(a(x_1,x_2))\to e)\in\Pi} [\![e]\!](t_1, t_2) \\ [\![p]\!](\#) & = & \bigcup_{(p(\#)\to e)\in\Pi} [\![e]\!]\_ \end{array}$$

Then, an expression $e$ takes a pair of trees or dummy and returns the set of trees resulted from evaluating $e$:

$$\begin{array}{lcl} [\![a(e_1, e_2)]\!]\rho & = & \{a(u_1, u_2) \mid u_i \in [\![e_i]\!](\rho) \text{ for } i = 1, 2\} \\ [\![\#]\!]\rho & = & \{\#\} \\ [\![p(x_h)]\!](t_1, t_2) & = & [\![p]\!](t_h) \end{array}$$

A constructor expression $a(e_1, e_2)$ evaluates each subexpression $e_i$ and reconstructs a tree node with the 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 $h$-th subtree. (Recall that a procedure call can appear in a node rule to which a pair of trees is always passed.) 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 $p$. For such a transducer, $\mathcal{T}(t)$ has at most one element for any $t$. A transducer $\mathcal{T}$ is *total* when it has at least one rule $p(a(x_1, x_2)) \to e$ for each $p$ and $a$.

**9.1.1 Example:** Let $\mathcal{T}_{9.1.1} = (\{p_0, p_1\}, \{p_0\}, \Pi)$ where $\Pi$ consists of:

$$\begin{array}{ll} p_0(a(x_1,x_2)) & \to a(p_1(x_1), p_0(x_2)) \\ p_0(b(x_1,x_2)) & \to c(p_0(x_1), p_0(x_2)) \\ p_0(\#) & \to \# \\ p_1(a(x_1,x_2)) & \to a(p_1(x_1), p_0(x_2)) \\ p_1(b(x_1,x_2)) & \to p_0(x_2) \\ p_1(\#) & \to \# \end{array}$$

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 $c$. Indeed, each procedure works as follows. 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 it and call $p_1$ for the left child and $p_0$ for the right child. For a $b$ node, on the other hand, $p_0$ renames its label with $c$, while $p_1$ only calls $p_0$ with its right child (thus removing the current $b$ node and its entire left subtree).

Determinism indeed changes the expressive power. The following example shows that, for a nondeterministic transducer, even when it cannot decide exactly which tree fragment to produce only from the current node and state, it can produce several possibilities for the moment and later discard some of them after looking at a descendant node.

**9.1.2 Example:** Let $\mathcal{T}_{9.1.2} = (\{p_0, p_1, p_2, p_3\}, \{p_0\}, \Pi)$ where $\Pi$ consists of:

$$\begin{array}{ll} p_0(a(x_1,x_2)) & \to a(p_1(x_1), p_3(x_2)) \\ p_0(a(x_1,x_2)) & \to b(p_2(x_1), p_3(x_2)) \\ p_1(c(x_1,x_2)) & \to c(p_3(x_1), p_3(x_2)) \\ p_2(d(x_1,x_2)) & \to d(p_3(x_1), p_3(x_2)) \\ p_3(\#) & \to \# \end{array}$$

This translates $a(c(\#,\#),\#)$ to itself and $a(d(\#,\#),\#)$ to $b(d(\#,\#),\#)$. This translation cannot be expressed by a deterministic top-down transducer since we need to decide whether to change the top $a$ label to $b$ after looking at the left child.

## 9.2 Height Property

A natural question that would arise here is what transformation a top-down tree transducer *cannot* express. As an example, consider the transformation that deletes each $b$ label from the input in such a way that the right-most leaf of the left tree is replaced with the right tree. This kind of transfomation often happens in XML processing: for instance, we may want to delete a `div` tag from a value like

```
div[p[...], img[...]], p[...]
```

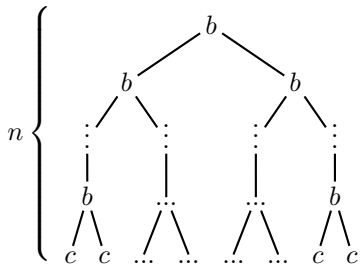
while retaining the content of the `div`:

```
p[...], img[...], p[...]
```

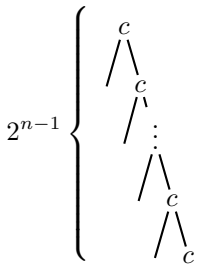
Now, the question is: is this transformation expressible by a top-down tree transducer? The answer is "no." But how can we prove this?

A powerful technique for characterizing tree transformations is a *height property*, which tells how much taller an output tree can be with respect to an input tree for a given transformation. As we will see below, we can bound the height increase for any top-down transducer (or of some other kind). By using this, we can show that a particular transformation is not expressible by a top-down (or some other kind of) tree transducer or even for comparing different formalisms of tree transducers.

For example, the above delete-$b$ transformation can exponentially increase the height of the tree. To see this, take the following full binary tree of height $n$ (leaves # are omitted).




Then, from this tree, the delete-$b$ transformation will produce a right-biased tree of height $2^{n-1}$.




However, we can prove that any output tree produced by a top-down tree transducer has only a linear height increase from the input tree.

**9.2.1 Theorem:** Let $\mathcal{T}$ be a top-down tree transducer and $t$ be a tree. Then, there exists $c > 0$ such that $\mathbf{ht}(u) \leq c \cdot \mathbf{ht}(t)$ for any $u \in \mathcal{T}(t)$.

PROOF: Define $c$ as 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 concludes that the delete-$b$ transformation is not expressible by a top-down tranducer. The next question is: what form of tree transducer can express this? This will be answered in the next section.

## 9.3 Macro Tree Transducers

In this section, we introduce an extension of top-down tree transducers with parameters. Since, historically, this extension has been invented for studying "macro expansion" in programming languages, it is called *macro tree transducers*. Like in a top-down one, each procedure of a macro tree transducer takes an implicit parameter that points to a node in the input tree. However, it can also take extra parameters that point to not nodes in the input but tree fragments that will potentially become a 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 $(P, P_0, \Pi)$ where $P$ and $P_0$ are the same as before and $\Pi$ is a set of rules each either of the following forms:

$$p^{(k)}(a(x_1, x_2), y_1, \ldots, y_k) \to e \qquad \text{(node rule)}$$
$$p^{(k)}(\#, y_1, \ldots, y_k) \to e \qquad \text{(leaf rule)}$$

Each $y_i$ is called *(accumulating) parameter*. We will abbreviate the tuples $(y_1, \ldots, y_k)$ to $\vec{y}$. Note that each procedure is now associated with its arity, i.e., the number of parameters; we write $p^{(k)}$ to denote a procedure $p$ with arity $k$. 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 of 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. The semantics of the macro tree transducer is again defined by a 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 resulted from evaluating any of $p$'s rules.

$$[\![p^{(k)}]\!](a(t_1, t_2), \vec{w}) = \bigcup_{(p^{(k)}(a(x_1,x_2),\vec{y})\to e)\in\Pi} [\![e]\!]((t_1, t_2), \vec{w})$$
$$[\![p^{(k)}]\!](\#, \vec{w}) = \bigcup_{(p^{(k)}(\#,\vec{y})\to e)\in\Pi} [\![e]\!](\_, \vec{w})$$

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 resulted from the evaluation:

$$\begin{array}{lcl}
[\![a(e_1, \ldots, e_m)]\!](\rho, \vec{w}) & = & \{a(u_1, u_2) \mid u_i \in [\![e_i]\!](\rho, \vec{w}) \text{ for } i = 1, 2\} \\
[\![\#]\!](\rho, \vec{w}) & = & \{\#\} \\
[\![p^{(l)}(x_h, e_1, \ldots, e_l)]\!]((t_1, t_2), \vec{w}) & = & \\
\multicolumn{3}{r}{\{[\![p^{(l)}]\!](t_h, (w'_1, \ldots, w'_l)) \mid w'_j \in [\![e_j]\!]((t_1, t_2), \vec{w}) \text{ for } j = 1, \ldots, l\}} \\
[\![y_j]\!](\rho, \vec{w}) & = & \{w_j\}
\end{array}$$

The difference from top-down transducers is that a procedure call $p(x_h, e_1, \ldots, e_l)$ passes the results of $e_1, \ldots, e_l$ as parameters when evaluating the procedure $p$ with the $h$-th subtree $t_h$. Also, a rule for a variable expression $y_j$ is added, where we simply return 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 $\mathcal{T}(t) = \bigcup_{p_0 \in P_0} [\![p_0]\!](t)$. (Recall that an initial procedure takes no parameter since it has arity zero.) Deterministic and total transducers can be defined similarly.

**9.3.1 Example:** Let us express the delete-$b$ transformation used in Section 9.2 by a macro tree transducer. Define $\Sigma = \{a, b\}$ and $\mathcal{T}_{9.3.1} = (\{p_0, p_1\}, \{p_0\}, \Pi)$ where $\Pi$ consists of:

$$\begin{array}{ll}
p_0(a(x_1, x_2)) & \to a(p_1(x_1, \#), p_1(x_2, \#)) \\
p_0(b(x_1, x_2)) & \to p_1(x_1, p_1(x_2, \#)) \\
p_0(\#) & \to \# \\
p_1(a(x_1, x_2), y) & \to a(p_1(x_1, \#), p_1(x_2, y)) \\
p_1(b(x_1, x_2), y) & \to p_1(x_1, p_1(x_2, y)) \\
p_1(\#, y) & \to y
\end{array}$$

That is, the procedure $p_1$ takes an accumulating parameter $y$ that will be appended after the "current sequence." 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 pass $y$ for the right child since we are still in the same sequence, while we pass # for the left child since we go into a new sequence. When we encouter a $b$ label, we do not copy the $b$ but make two recursive calls, where we pass $y$ for the right child while, for the left child, we pass 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 an initial parameter #. However, we cannot simply write

$$p_0(x) \to p_1(x, \#)$$

since a macro tree transducer, as it is defined, needs to go down by one label for each procedure call. (We could extend tranducers with direct procedure calls, which we will mention later on.) Therefore we instead define the procedure $p_0$ to be exactly the same as $p_1$ except that it does not take an accumulating parameter but uses # whenever $p_1$ would use $y$.

**9.3.2 Example:** Let us write a more serious macro tree transducer that, given an XHTML document, puts the list of all `img` elements to the end of the `body` element: $\mathcal{T}_{9.3.2} = (\{\mathsf{main}, \mathsf{getimgs}, \mathsf{putimgs}, \mathsf{putimgs2}, \mathsf{copy}\}, \{\mathsf{main}\}, \Pi)$ where $\Pi$ consists of the followings rules.

$$
\begin{array}{ll}
\mathsf{main}(\texttt{html}(x_1, x_2)) & \to \texttt{html}(\mathsf{putimgs}(x_1, \mathsf{getimgs}(x_1, \#)), \mathsf{copy}(x_2)) \\
\\
\mathsf{getimgs}(\texttt{img}(x_1, x_2), y) & \to \texttt{img}(\mathsf{copy}(x_1), \mathsf{getimgs}(x_2, y)) \\
\mathsf{getimgs}(*(x_1, x_2), y) & \to \mathsf{getimgs}(x_1, \mathsf{getimgs}(x_2, y)) \\
\mathsf{getimgs}(\#, y) & \to y \\
\\
\mathsf{putimgs}(\texttt{body}(x_1, x_2), y) & \to \texttt{body}(\mathsf{putimgs2}(x_1, y), \mathsf{copy}(x_2)) \\
\mathsf{putimgs}(*(x_1, x_2), y) & \to *(\mathsf{putimgs}(x_1, y), \mathsf{putimgs}(x_2, y)) \\
\mathsf{putimgs}(\#, y) & \to \# \\
\\
\mathsf{putimgs2}(*(x_1, x_2), y) & \to *(\mathsf{copy}(x_1), \mathsf{putimgs2}(x_2, y)) \\
\mathsf{putimgs2}(\#, y) & \to y \\
\\
\mathsf{copy}(*(x_1, x_2)) & \to *(\mathsf{copy}(x_1), \mathsf{copy}(x_2)) \\
\mathsf{copy}(\#) & \to \#
\end{array}
$$

Here, each rule with a `*` pattern should be read as the set of rules where the `*` label and all `*` 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 puts it to the content of the `body` element by using putimgs. The putimgs procedure, when it finds the `body` element, uses putimgs2 to append the `img` list in the end of the content sequence. The auxiliary copy procedure simply copies the whole subtree.

In Example 9.3.1, we have seen a macro tree transducer that expresses the delete-$b$ transformation, which can have an exponential increase in the tree height (Section 9.2). In fact, we can assert that this is the maximum increase for this transducer since, in general, any macro tree transducer can grow the tree height at most exponentially, as stated by the following theorem.

**9.3.3 Theorem:** 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 rules. Then, the result follows by proving the following.

(A) $u \in [\![p^{(k)}]\!](t, \vec{w})$ implies $\mathbf{ht}(u) \leq c^{\mathbf{ht}(t)} + \max(0, \mathbf{ht}(w_1), \ldots, \mathbf{ht}(w_k))$.

The proof proceeds by induction on $t$. The above statement (A) immediately follows if the following statement (B) holds.

(B) $u \in [\![e]\!](\rho, \vec{w})$ implies $\mathbf{ht}(u) \leq \mathbf{ht}(e) c^{\mathbf{ht}(\rho)} + \max(0, \mathbf{ht}(w_1), \ldots, \mathbf{ht}(w_k))$.

Here, $\mathbf{ht}(\rho)$ is defined as 0 when $\rho = \_$ and as $\max(\mathbf{ht}(t_1), \mathbf{ht}(t_2))$ when $\rho = (t_1, t_2)$. The condition (B) can in turn be proved by induction on the structure of $e$. Let $d = \max(0, \mathbf{ht}(w_1), \ldots, \mathbf{ht}(w_k))$.

- When $e = \#$ or $e = y_j$, the condition (B) trivially holds.
- When $e = a(e_1, e_2)$, 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 below.

$$\begin{array}{rcll} \mathbf{ht}(u) & = & 1 + \max(\mathbf{ht}(u_1), \mathbf{ht}(u_2)) & \\ & \leq & 1 + \max(\mathbf{ht}(e_1)c^{\mathbf{ht}(\rho)} + d, \mathbf{ht}(e_2)c^{\mathbf{ht}(\rho)} + d) & \text{by I.H. for (B)} \\ & \leq & (1 + \max(\mathbf{ht}(e_1), \mathbf{ht}(e_2)))c^{\mathbf{ht}(\rho)} + d & \\ & = & \mathbf{ht}(e)c^{\mathbf{ht}(\rho)} + d & \end{array}$$

- When $e = p(x_h, e_1, \ldots, e_l)$, note that $\rho$ has the form $(t_1, t_2)$ and $u \in [\![p]\!](t_h, \vec{w'})$ where $w'_j \in [\![e_j]\!](\rho, \vec{w})$ for $j = 1, \ldots, l$. Thus, we can derive (B) as below.

$$\begin{array}{rcll} \mathbf{ht}(u) & \leq & c^{\mathbf{ht}(t_h)} + \max(0, \mathbf{ht}(w'_1), \ldots, \mathbf{ht}(w'_l)) & \text{by I.H. for (A)} \\ & \leq & c^{\mathbf{ht}(t_h)} + \max(0, \mathbf{ht}(e_1)c^{\mathbf{ht}(\rho)} + d, \ldots, \mathbf{ht}(e_l)c^{\mathbf{ht}(\rho)} + d) & \\ & & & \text{by I.H. for (B)} \\ & \leq & (1 + \max(0, \mathbf{ht}(e_1), \ldots, \mathbf{ht}(e_l)))c^{\mathbf{ht}(\rho)} + d & \\ & = & \mathbf{ht}(e)c^{\mathbf{ht}(\rho)} + d & \end{array}$$

□

**9.3.4 Exercise:** Consider the following transformation. Given an XHTML tree, whenever there is a `div` element, e.g.,

```
div[h3[...], a[...], ...]
```

we collect the list of all `img` elements appearing in the whole subtree of the `div` and prepend the list to the `div`'s content:

```
div[img[...], img[...], img[...], h3[...], a[...], ...]
```

Using Theorem 9.3.3, prove that this transformation is not expressible by a macro tree transducer. (Note that, in XHTML, `div` elements can be nested.)

## 9.4 Bibliographic Notes

Various variations of top-down and macro tree transducers have been studied [26, 27, 31]. 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). Another extension is to allow *regular look ahead*. In this, we assume an automaton for the input tree and associate each rule with a state like:

$$p(a(x_1, x_2), y_1, \ldots, y_k)@q \to e$$

Each rule is fired when the input can be accepted by the associated state. It is known that both extensions do not increase the expressive power of top-down nor macro tree transducers. 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 the transformation for nondeterministic transducers. *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 [80]. Their expressive power actually changes by the addition of the concatenation operator; in particular, macro forest transducers have a double-exponential height property. Finally (but not lastly), macro tree transducers extended with higher-order functions are called *high-level tree transducers* [32].

In general, when there is an acceptor model, we can think of its corresponding transducer model. In this sense, top-down tree transducers correspond to top-down tree automata. Similarly, we can consider *bottom-up tree transducers* corresponding to bottom-up tree automata [26]. In bottom-up tree transducers, we process each node of the input tree from the leaves to the root where, at each node, we use the states assigned to the child nodes and the current node label for deciding how to trasform the node and which state to transit. A difference in expressiveness between top-down and bottom-up transducers is that bottom-up ones 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. However, the special ability of bottom-up transducers seems rather useless in practice and this is perhaps why they are relatively less studied.

Chapter 11 will present tree-walking automata, which can move not only down but also up in the input tree. A transducer model closely related to this is *$k$-pebble tree transducers* [68], which allow moving up and down $k$ pointers to nodes in the input tree and are thereby capable of representing various realistic XML transformations. Chapter 13 will present the MSO logic for describing constraints among tree nodes. *MSO-definable tree transducers* lift this ability to relate tree nodes in the input and in the output [24].

Theoretical properties of these transducer models have actively been investigated, such as expressive powers, exact typechecking, and composability (e.g., can the transformation of a top-down transducer composed with another always be realized by a single top-down transducer?). Besides exact typechecking (which will be covered in Chapter 10), further details are out of scope in this book; interested readers should consult the literature, e.g., [31, 29]

# Chapter 10

# Exact Typechecking

In Chapter 6, we have previewed the exact typechecking approach to the static verification of XML transformations. In this approach, rather than considering a general, Turing-complete language on which only an approximate analysis can be made, we take a restricted language for which a precise analysis is decidable. Various XML transformation languages and typechecking algorithms for them have been investigated. Generally, the more expressive the language is, the more complicated the typechecking algorithm becomes. In this chapter, we study exact typechecking for one of the simplest tree transformation formalisms as our target language, namely, top-down tree transducers introduced in Chapter 9. Although this formalism is so simple that many interesting transformations are not expressible, we can explain many technical ideas behind exact typechecking that are applicable to other formalisms. In particular, we will see in detail where forward inference fails and backward inference is needed, how nondeterminism influences on the typechecking algorithm, how alternating tree automata are useful for building the algorithm, and so on.

## 10.1 Motivation

One may wonder why exact typechecking would be important from the first place. Let us give an instructive example, which is an identity function written in the $\mu$XDuce language presented in Chapter 6. This is such a trivial example, but still 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 did not define the type `T`. Indeed, the function should typecheck whatever `T` is defined to be (as long as it 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 $\mu$XDuce typechecker or not actually depends on how `T` is defined. For example, if we define

```
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 thus the body can be validated. Indeed, in the second case, pattern type inference (Section 6.3) 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` and similarly for the other cases. However, if we instead define

```
type T = a[T]*,b[T]*,c[T]*
```

then typechecking fails. To see why, let us focus on the third case. First, pattern type inference gives the variable `z` `(b[T]*,c[T]*)` since this describes what can follow after a `b` label; therefore typechecker asserts that the argument type is valid at the function call `id(z)`. However, typechecker gives `T` as the return type of the function call since it is declared so (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 happen? It is because the $\mu$XDuce typechecker does not consider 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]*`, 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 $\mu$XDuce-like typechecker in a context-sensitive way fails to achieve exactness. That is, the $\mu$XDuce 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 this set can in general 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 technicality, a remark is in order on the usability of the exact typechecking approach. One may argue that this approach can be too limited since, once a transformation slightly goes beyond the target language, this method immediately becomes unapplicable. On the contrary, this is exactly

where nondeterminism becomes useful. That is, even if a given transformation is not exactly expressible, its "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 $\mu$XDuce-like method, yet has a clear specification—"first abstraction and then exact typechecking"—which is important for explaining to the user the reason for a raised error.

## 10.2 Where Forward Type Inference Fails

Given a transformation $\mathcal{T}$, an input type $\tau_I$, and an output type $\tau_O$, the forward inference approach first computes the image $\tau'_O = \mathcal{T}(\tau_I)$ of $\tau_I$ by the transformation $\mathcal{T}$ (we write $\mathcal{T}(\tau_I) = \bigcup_{t \in \tau_I} \mathcal{T}(t)$) and then checks the inclusion between $\tau'_O$ and $\tau_O$.

To see the limitation of this approach, let us consider the following top-down transducer $\mathcal{T}$ with the initial procedure $p_0$:

$$\begin{aligned} p_0(a(x_1, x_2)) &\to a(p_1(x_1), p_1(x_1)) \\ p_1(a(x_1, x_2)) &\to a(p_1(x_1), p_1(x_2)) \\ p_1(\#) &\to \# \end{aligned}$$

The procedure $p_1$ simply copies the whole given tree and thus the procedure $p_0$ duplicates the left subtree, i.e., translates any tree of the form $a(t_1, t_2)$ to $a(t_1, t_1)$. So, for the input type $T = \{a(t, \#) \mid t \text{ is any tree}\}$, we can easily see that the image is $\mathcal{T}(T) = \{a(t, t) \mid t \text{ is any tree}\}$, which is well known to go beyond regular tree languages. (This set is in fact within so-called context-free tree languages and therefore could be checked against a given output type since inclusion between a context-free tree language and a regular language is decidable, cf. [27, 30, 28]). However, it is easy to further extend the above example so as to obtain an image $\{a(t, a(t, t)) \mid t \text{ is any tree}\}$, which is even beyond context-free tree languages.)

In a special case, however, forward type inference works for top-down tree transducers. For example, when a transducer is *linear*, that is, each right hand side of its rules uses at most once for each variable, then the set of output values actually fits in regular tree language.

**10.2.1 Exercise:** Construct an algorithm of forward type inference for linear top-down tree transducers.

## 10.3 Backward Type Inference

The back inference approach does the opposite direction. In principle, this approach first computes a type $\tau'_I$ representing $\{t \mid \mathcal{T}(t) \subseteq \tau_O\}$ and then checks

the inclusion $\tau_I \subseteq \tau'_I$. However, it is rather complicated to directly compute such type $\tau'_I$. Instead, we below show a method that computes $\overline{\tau'_I}$ (the complement of $\tau'_I$), which equals to the preimage $\mathcal{T}^{-1}(\overline{\tau_O})$ of $\overline{\tau_O}$, that is, the set $\{t \mid \exists t' \in \overline{\tau_O}.\ t' \in \mathcal{T}(t)\}$. By using this, we can complete typechecking by testing $\overline{\tau'_I} \cap \tau_I = \emptyset$. (However, we also see later that, for deterministic transducer, the direct approach is equally easy.)

Let $\mathcal{T}$ be a top-down transducer $(P, P_0, \Pi)$ and $\mathcal{M}$ be a (nondeterministic) tree automaton $(Q, I, F, \Delta)$ (which represents the type $\overline{\tau_O}$ above). We construct an alternating automaton $\mathcal{A} = (R, R_0, R_F, \Phi)$ (which represents the preimage $\overline{\tau'_I}$) in the following way.

$$\begin{array}{lcl} R & = & \{\langle p, q\rangle \mid p \in P,\ q \in Q\} \\ R_0 & = & \{\langle p_0, q_0\rangle \mid p_0 \in P_0,\ q \in I\} \\ R_F & = & \{\langle p, q\rangle \mid \ \vdash \bigvee_{(p(\#)\to e)\in\Pi} \mathrm{Inf}(e, q)\} \\ \Phi(\langle p, q\rangle, a) & = & \displaystyle\bigvee_{(p(a(x_1,x_2))\to e)\in\Pi} \mathrm{Inf}(e, q) \end{array}$$

Here, the function Inf is defined inductively as follows.

$$\begin{array}{lcl} \mathrm{Inf}(a(e_1, e_2), q) & = & \displaystyle\bigvee_{(q\to a(q_1,q_2))\in\Delta} \mathrm{Inf}(e_1, q_1) \wedge \mathrm{Inf}(e_2, q_2) \\ \mathrm{Inf}(p(x_h), q) & = & \downarrow_h \langle p, q\rangle \\ \mathrm{Inf}(\#, q) & = & \begin{cases} \top & (q \in F) \\ \bot & (q \notin F) \end{cases} \end{array}$$

Informally, 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 the expression $e$ appears in a node rule, then the formula represents the set of pairs of trees that can be translated to a tree in the state $q$. 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 $q$. Recall that no procedure call appears in a leaf rule, which means that a formula $\phi$ returned by Inf never contains the form $\downarrow_i \langle p, q\rangle$ and therefore is naturally evaluated to a Boolean; we write $\vdash \phi$ when $\phi$ is evaluated to true. By using this, we collect, as $\mathcal{A}$'s final states, $\langle p, q\rangle$ such that a leaf rule of $p$ translates a leaf node to a tree in the state $q$.

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, whatever the input is, 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 the input is.

- 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 can be obtained by recursively calling Inf with $e_1$ and $q_1$ and with $e_2$ and $q_2$ for each transition $q \to a(q_1, q_2)$ and combine all the results by a union.

- 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 $h$-th subtree. This condition can be rephrased that the $h$-th subtree is in the set of trees from which the procedure $p$ translates to a tree in the state $q$. This set can directly be obtained from the state $\langle p, q\rangle$.

**10.3.1 Theorem:** $\mathcal{L}(\mathcal{A}) = \mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}))$.

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 $t$. The statement (A) follows by showing the following two:

(B1) $(t_1, t_2) \in [\![\mathrm{Inf}(e, q)]\!]$ 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) $\vdash \mathrm{Inf}(e, q)$ 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 is done by induction on the structure of $e$.

- When $e = \#$, we have:

$$\begin{aligned}(t_1, t_2) \in [\![\mathrm{Inf}(e, q)]\!] &\iff q \in F \\ &\iff [\![e]\!](t_1, t_2) \cap [\![q]\!] \neq \emptyset.\end{aligned}$$

- When $e = a(e_1, e_2)$, we have:

$$\begin{aligned}&(t_1, t_2) \in [\![\mathrm{Inf}(e, q)]\!] \\ &\iff \exists(q \to a(q_1, q_2) \in \Delta).\ (t_1, t_2) \in [\![\mathrm{Inf}(e_1, q_1)]\!] \ \wedge\ (t_1, t_2) \in [\![\mathrm{Inf}(e_2, q_2)]\!] \\ &\iff \exists(q \to a(q_1, q_2) \in \Delta).\ [\![e_1]\!](t_1, t_2) \cap [\![q_1]\!] \neq \emptyset \ \wedge\ [\![e_2]\!](t_1, t_2) \cap [\![q_2]\!] \neq \emptyset \\ &\qquad\qquad \text{(by I.H. for (B1))} \\ &\iff [\![e]\!](t_1, t_2) \cap [\![q]\!] \neq \emptyset\end{aligned}$$

- When $e = p(x_h)$, we have:

$$\begin{aligned}&(t_1, t_2) \in [\![\mathrm{Inf}(e, q)]\!] \\ &\iff t_h \in [\![\langle p, q\rangle]\!] \\ &\iff [\![p]\!](t_h) \cap [\![q]\!] \neq \emptyset \qquad \text{(by I.H. for (A))} \\ &\iff [\![e]\!](t_1, t_2) \cap [\![q]\!] \neq \emptyset\end{aligned}$$

□

**10.3.2 Example:** Consider the following top-down tree transducer: $(\{p_0\}, \{p_0\}, \Pi)$ where $\Pi$ consists of:

$$\begin{aligned} p_0(a(x_1, x_2)) &\to a(p_0(x_1), p_0(x_2)) \\ p_0(b(x_1, x_2)) &\to a(p_0(x_1), p_0(x_2)) \\ p_0(b(x_1, x_2)) &\to b(p_0(x_1), p_0(x_2)) \\ p_0(\#) &\to \# \end{aligned}$$

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 the latter.

Both the input and the output types are the automaton $(\{q\}, \{q\}, \{q\}, \{q \to a(q, q)\})$ representing the set of trees with all $a$ labels. What we do first is to take the complement of the output automaton. We obtain, for example, the automaton $(\{q_0, q_1\}, \{q_0\}, \{q_1\}, \Delta)$ where $\Delta$ consists of the following.

$$\begin{aligned} q_0 &\to a(q_0, q_1) \\ q_0 &\to a(q_1, q_0) \\ q_0 &\to b(q_1, q_1) \\ q_1 &\to a(q_1, q_1) \\ q_1 &\to b(q_1, q_1) \end{aligned}$$

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 automaton w.r.t. the transducer: $(R, R_0, R_F, \Phi)$ where

$$\begin{aligned} R &= \{\langle p_0, q_0\rangle, \langle p_0, q_1\rangle\} \\ R_0 &= \{\langle p_0, q_0\rangle\} \\ R_F &= \{\langle p_0, q_1\rangle\} \end{aligned}$$

and $\Phi$ is defined by the following.

$$\begin{aligned} \Phi(\langle p_0, q_0\rangle, a) &= \downarrow_1 \langle p_0, q_0\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \ \vee \ \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_0\rangle \\ \Phi(\langle p_0, q_0\rangle, b) &= \downarrow_1 \langle p_0, q_0\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \ \vee \ \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_0\rangle \\ &\quad \vee \ \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \\ \Phi(\langle p_0, q_1\rangle, a) &= \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \\ \Phi(\langle p_0, q_1\rangle, b) &= \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \ \vee \ \downarrow_1 \langle p_0, q_1\rangle \wedge \downarrow_2 \langle p_0, q_1\rangle \end{aligned}$$

This alternating tree automaton can be translated to the following nondeterministic tree automaton by using the construction **ATL-to-ND** (Section 8.2):

$(\{r_0, r_1\}, \{r_0\}, \{r_1\}, \Delta')$ (let $r_0 = \{\langle p_0, q_0\rangle\}$ and $r_1 = \{\langle p_0, q_1\rangle\}$) where $\Delta'$ consists of the following.

$$\begin{aligned} r_0 &\to a(r_0, r_1) \\ r_0 &\to a(r_1, r_0) \\ r_0 &\to b(r_0, r_1) \\ r_0 &\to b(r_1, r_0) \\ r_0 &\to b(r_1, r_1) \\ r_1 &\to a(r_1, r_1) \\ r_1 &\to b(r_1, r_1) \end{aligned}$$

This automaton is disjoint with the input automaton (accepting 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 $\Delta''$ consists of

$$\begin{aligned} (q, r_0) &\to a((q, r_0), (q, r_1)) \\ (q, r_0) &\to a((q, r_1), (q, r_0)) \\ (q, r_1) &\to a((q, r_1), (q, r_1)) \end{aligned}$$

which accepts no tree (note that $(q, r_1)$ accepts some tree but $(q, r_0)$ does not).

Notice that, when the transformation $\mathcal{T}$ yields at most result from any input (in particular when it is deterministic), the set $\tau_I' = \{t \mid \mathcal{T}(t) \subseteq \tau_O\}$ coincides the preimage of the output type $\tau_O$, 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 above, we can construct an automaton representing $\tau_I'$ from another representing $\tau_O$. Then, typechecking can be done by testing $\tau_I \subseteq \tau_I'$. Note that the worst-case complexity does not change: exponential time in both cases.

**10.3.3 Exercise:** Confirm by using Example 10.3.2 that the above method does not work for a transducer that may yield multiple results from an input.

**10.3.4 Example:** Let us typecheck the example used in Section 10.2. The input type is an automaton representing $T = \{a(t, \#) \mid t$ is any tree with all $a$ labels$\}$ and the output type is the automaton $(\{q\}, \{q\}, \{q\}, \{q \to a(q, q)\})$ accepting any tree with all $a$ labels. From the tree transducer in that section and the output automaton, we construct the alternating tree automaton $(R, R_0, R_F, \Phi)$ where

$$\begin{aligned} R &= \{\langle p_0, q\rangle, \langle p_1, q\rangle\} \\ R_0 &= \{\langle p_0, q\rangle\} \\ R_F &= \{\langle p_1, q\rangle\} \end{aligned}$$

and $\Phi$ is defined by:

$$\Phi(\langle p_0, q\rangle, a) \rightarrow \downarrow_1 \langle p_1, q\rangle \wedge \downarrow_1 \langle p_1, q\rangle$$
$$\Phi(\langle p_1, q\rangle, a) \rightarrow \downarrow_1 \langle p_1, q\rangle \wedge \downarrow_2 \langle p_1, q\rangle$$

This alternating tree automaton can be translated to $\mathcal{M} = (Q, I, F, \Delta)$ where

$$Q = \{\emptyset, \{\langle p_0, q\rangle\}, \{\langle p_1, q\rangle\}\}$$
$$I = \{\{\langle p_0, q\rangle\}\}$$
$$F = \{\emptyset, \{\langle p_1, q\rangle\}\}$$

and $\Delta$ consists of:

$$\emptyset \rightarrow a(\emptyset, \emptyset)$$
$$\{\langle p_0, q\rangle\} \rightarrow a(\{\langle p_1, q\rangle\}, \emptyset)$$
$$\{\langle p_1, q\rangle\} \rightarrow a(\{\langle p_1, q\rangle\}, \{\langle p_1, q\rangle\})$$

This automaton in fact represents the set of all non-leaf trees only with $a$ labels. We can thus easily see that $T \subseteq \mathcal{L}(\mathcal{M})$.

**10.3.5 Exercise:** By using Theorem 10.3.1, prove that a macro tree transducer cannot express the transformation that returns `true` if the give tree has identical subtrees and `false` otherwise.

**10.3.6 Exercise:** We can extend the above typechecking algorithm for macro tree transducers by taking a tuple $\langle p, q, q_1, \ldots, q_k\rangle$ as a state representing the set of inputs from which the procedure $p$ translates to a result that conforms to the state $q$, assuming that $k$ parameters conform to $q_1, \ldots, q_k$ respectively. Observe that the tree automaton for the output type must be bottom-up deterministic.

## 10.4 Bibliographic Notes

Exact typechecking techniques have been investigated for a long time on various tree transducer models. For example, very early work has already treated top-down and bottom-up transducers [27]; later work has dealt with macro tree transducers [31] and macro forest transducers [80]. In the relation to XML, exact typechecking started with forward-inference-based techniques, treating relatively simple transformation languages [70, 67, 79]. A backward inference technique has first been used for XML typechecking by Milo, Suciu, and Vianu in their seminal work on $k$-pebble tree transducers [68]. Then, Tozawa has proposed exact typechecking for a subset of XSLT based on alternating tree automata [86]; this chapter adopts his style of formalization. Maneth, Perst, Berlea, and Seidl [65] have shown a typechecking method for a non-trivial tree transformation language *TL* by decomposition to *macro tree transducers*, whose typechecking is already known. As macro tree transducers appear to be a

promising model that is simple yet expressive, some efforts have been made towards practically usable typechecking [66, 40]. A typechecking technique treating equality on leaf values can be found in [3] and one treating high-level tree transducers in [87].

# Chapter 11

# Path Expressions and Tree-Walking Automata

We have already seen pattern matching as an approach to specifying subtree extraction (Chapter 4). In this chapter, we learn an alternative approach called path expressions. While patterns use structural constraints to point to target subtrees, path expressions use "navigation," which specifies a sequence of movements on the tree and checks on the traversed nodes. In both frameworks, the user writes requirements to the local properties (i.e., the names) of nodes as well as the positional relationship among them. A critical difference lies what they specify for the nodes that the user does *not* mention. Patterns regard such nodes as *absence*, whereas paths *don't care* their presence. Since, in practice, most nodes are irrelevant in subtree extraction, paths are often more useful. However, as we will see soon, path expressions are theoretically less expressive (they cannot express all regular tree languages) unless enough extension is done.

In this chapter, we first review path expressions used in actual XML processing and then see their refinement called *caterpillar expressions*. After this, we study their corresponding automata formalism called *tree-walking automata* and compare their expressiveness with tree automata.

## 11.1 Path Expressions

Although path expressions can be found in a variety of styles, the basic idea is to match a node such that, from the "current" node to the matched node, there is a "path" conforming to the given path expression. When there are multiple matching nodes, we usually take all of them.

The main differences among styles lies in what paths are allowed. A path is, roughly, a sequence consisting of either movements or tests on nodes. In each movement, we specify which direction to go, which we call *axis*. For example, some styles allow only going to predecessor nodes—called *forward axis*—while other allow also going to ancestor nodes—called *backward axis*. There are also

other axes for following and preceding siblings, and so on. Tests on nodes can be local or global. A local one includes the node's label, whether the node is root, and so forth. A typical global one is a path expression itself—whether a specified node exists or not 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 expressions differ in what path languages to allow. Simple styles allow only a single path to be specified, whereas others allow regular languages of paths.

### 11.1.1 XPath

XPath is a framework of path expressions standardized in W3C and nowadays widely used in various languages and systems. Let us see below some examples in XPath, where we take the family tree in Figure 2.1 as the input and some `person` node as the current node.

| | |
|---|---|
| `children/person` | a `person` subnode of a `children` subnode (of the current node) |
| `.//person/name` | a `name` subnode of a `person` predecessor |
| `../../spouse` | a `spouse` subnode of the grandparent node |

Here, a label like `children` denotes the test whether the current node has that label. The axes for moving to a child, an ancestor, and a parent are written by `/`, `//`, and `...` "Not moving" (or staying) is also an axis and written `.`. Slightly verbose notations are used for going to siblings:

| | |
|---|---|
| `following-sibling::person` | a `person` right sibling |
| `preceding-sibling::person` | a `person` left sibling |

Here are some uses of path expressions as node tests, written in square brackets.

`children/person[gender/male]`
: a `children` node's `person` subnode that has a `gender` subnode with a `male` subnode

`children/person[gender/male and spouse]`
: similar to the previous except a `spouse` subnode to additionally be required in the `person`

Note that the last example above contains an `and` in the test; other logical connectives `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)*` | The nodes on the path from the current node to the matched node have labels `children` and `person` alternatingly. |

### 11.1.2 Caterpillar Expressions

This section presents a formalism of regular expression paths called caterpillar expressions. (The name stems from the analogy that this insect can crawl on a tree in any direction.) It was first proposed by Brüggemann-Klein and Wood as an alternative to XPath and has a direct relationship to tree-walking automata, thus forming a basis for theoretical analysis on expressiveness and algorithmics of path expressions. While caterpillar expressions were proposed originally for unranked trees, we present a variant that walks on binary trees for the sake of smooth transition to the automata framework.

Let $\Sigma_\#$ be $\Sigma \cup \{\#\}$, ranged over by $a$. A *caterpillar expression* $e$ is a regular expression where each symbol is a *caterpillar atom* $c$ 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 *caterpillar path*.

Let a (binary) tree $t$ given. A sequence $\pi_1 \dots \pi_n$ of nodes each from nodes($t$) belongs to a caterpillar path $c_1 \dots c_n$ if, for $i = 1, \dots, n-1$,

- 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 $\text{label}_t(\pi_i) = a$,
- 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 $\pi$,
- if $c_i = \text{is2}$, then $\pi_{i+1} = \pi_i = \pi 2$ for some $\pi$,

Since a node is a leaf if and only if its label is #, we already have the test whether the current node is a leaf or a parent.[1] Note also that, since each node of a binary tree either has two subnodes or is a leaf, the same test is enough for checking that either child exists. A sequence of nodes belongs to a caterpillar expression $e$ if the sequence belongs to a caterpillar path generated by $e$. A node $\pi$ matches a caterpillar expression $e$ if there is a sequence $\pi_1 \dots \pi_n$ of nodes belonging to $e$ where $\pi_1 = \epsilon$ and $\pi_n = \pi$.

[1] In the original proposal, since a leaf can have an arbitrary label, it explicitly supports the leaf-or-parent test.

Some of XPath examples seen in the last section can be expressed by caterpillar expressions. However, since the previous examples were on unranked trees, we need to adjust the up and the down axes. Let us define

$$\begin{aligned} \text{Xup} &= (\text{is2 up})^* \text{is1 up} \\ \text{Xdown} &= \Sigma\ 1\ (\Sigma\ 2)^*. \end{aligned}$$

($\Sigma$ in these expressions stands for $a_1 \mid \ldots \mid a_n$ where $\Sigma = \{a_1, \ldots, a_n\}$.) That is, 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 if the node is a parent, and then repeatedly goes down taking the second child if the node is a parent. By using these two expressions, we can represent previous XPath expressions in the following way.

$$\begin{aligned} \texttt{../../spouse} &\Rightarrow \text{Xup Xup Xdown } \texttt{spouse} \\ \texttt{(children/person)*} &\Rightarrow (\text{Xdown } \texttt{children}\ \text{Xdown } \texttt{person})^* \\ \texttt{following-sibling::person} &\Rightarrow (\Sigma\ 2)^+\ \texttt{person} \end{aligned}$$

Note that there is no obvious way of encoding conjunctive path expressions. Indeed, a theoretical result tells that it is fundamentally impossible (Section 11.2.3).

We conclude this section by showing an amusing example that starts from the root, traverses *every* node, and returns to the root.

$$(1^*\ \#\ (\text{is2 up})^*\ \text{is1 up}\ 2)^*\ (\text{is2 up})^*$$

That is, if the current node is a parent, then we first go all the way down by taking the first child until the leaf. From there, we go left-up repeatedly (this can happen only when we didn't go down in the last step) and then go right-up once. We iterate these until we reach the right-most leaf and finally go straight back to the root.

## 11.2 Tree-Walking Automata

Tree-walking automata (TWA) are a finite-state machine model directly corresponding to caterpillar expressions. They move up and down in a binary tree, changing states based on the kind of the current node and the current state.

### 11.2.1 Definitions

Let a set $K$ of node kinds be $\Sigma_\# \times \{\text{root}, 1, 2\}$ and ranged over by $k$. A *tree-walking automaton* (TWA) is a quadruple $A = (Q, I, F, \delta)$ where $Q$ and $I$ are as usual,

- $F$ is a set of pairs from $Q \times K$, and[2]

[2] A usual definition of tree-walking automata does not have node tests in final states. However, these are needed for encoding caterpillar expressions since they may perform node tests in the end.

- $\delta$ 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 $d \in \{\mathrm{up}, 1, 2\}$.

For a binary tree $t$, we define the kind of a node $\pi$ in it as follows:

$$\kappa_t(\pi) \quad = \quad (\mathrm{label}_t(\pi), h) \qquad \text{where } h = \begin{cases} \text{root} & \text{if } \pi = \epsilon \\ 1 & \text{if } \pi = \pi'1 \text{ for some } \pi' \\ 2 & \text{if } \pi = \pi'2 \text{ for some } \pi' \end{cases}$$

A sequence $(\pi_1, q_1) \ldots (\pi_n, q_n)$ is a *run* of a tree walking automaton $(Q, I, F, \delta)$ on a binary tree $t$ if $\pi_i \in \mathrm{nodes}(t)$ for each $i = 1, \ldots, n$ and

- $q_i \xrightarrow{\kappa_t(\pi_i),\mathrm{up}} q_{i+1} \in \delta$ 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 $\pi_1$ in state $q_1$ and finishes at node $\pi_n$ in state $q_n$ where each step makes a movement according to a transition rule that matches the node kind. A run $(\pi_1, q_1) \ldots (\pi_n, q_n)$ is *successful* when $\pi_1 = \epsilon$, $q_1 \in I$, and $(q_n, \kappa_t(\pi_n)) \in F$. In this case, the node $\pi_n$ is *matched* by the automaton. When $\pi_n$ is the root, we say that the whole tree $t$ is *accepted* by the automaton. We define the language $L(A)$ of a TWA $A$ to be $\{t \mid A \text{ accepts } t\}$. Let **TWA** be the class of languages accepted by tree-walking automata.

It is quite clear that caterpillar expressions and tree-walking automata are equivalent notions. The only part that needs a 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 done in a caterpillar expression.

**11.2.1 Exercise:** Write down conversions between caterpillar expressions and tree walking automata.

### 11.2.2 Expressiveness

Since we have been considering path expressions as an alternative to patterns, a question will naturally arise on relationship between the expressivenesses of tree-walking automata and tree automata. The bottom line is that the former are strictly less expressive. This can be proved by showing (1) **TWA** $\subseteq$ **ND** and (2) a counterexample against **ND** $\subseteq$ **TWA**. The first part is easier and covered in this section. The second part, on the other hand, is much more difficult and the details cannot be presented in this book; in fact, it was a long-standing

problem that had not been proved for several decades until 2004 by Bojańczyk and Colcombat.

To show **TWA** $\subseteq$ **ND**, it suffices to show a construction of a nondeterministic tree automaton from a tree-walking automaton. It is possible to directly do so, but we instead show a construction of an alternating tree automaton defined in Chapter 8, from which we already know how to construct a normal tree automaton.

Before our general discussion, let us see first an example for an intuitive understanding. Figure 11.1 depicts a tree and a series of movements by a tree-walking automaton with the indicated state transitions. That is, the automaton starts from the root $\pi_0$ in $q_0$ and moves to the left child $\pi_1$ in $q_0'$. Then, it walks around the nodes below $\pi_1$ with some state transitions, after which it goes back to $\pi_1$ in $q_1'$ and then to the root $\pi_0$ in $q_1$. Then, the automaton moves to the right child, walks around the nodes below it, and returns to the root in $q_2$. Finally, it moves again to the left child $\pi_1$ in $q_2'$ and then immediately goes back to the root in $q_3$.

To analyze complex state transitions of tree-walking automata, a useful concept is *tour*. A tour from a node $\pi$ is a run starting from $\pi$, walking around the nodes below $\pi$, and ends at $\pi$. More formally, a tour from $\pi$ is a run of the form

$$(\pi_1, q_1) \dots (\pi_n, q_n)$$

where $\pi_1 = \pi_n = \pi$ and, for each $i = 2, \dots, n-1$, we have $\pi < \pi_i$, i.e., $\pi_i$ is a strict descendant of $\pi$. (Note that $n$ must be an odd number from the definition of runs.) In our example, the tours from the root $\pi_0$ are

$$\begin{aligned}
&(\pi_0, q_0)(\pi_1, q_0') \dots (\pi_1, q_1')(\pi_0, q_1) \\
&(\pi_0, q_1) \dots (\pi_0, q_2) \\
&(\pi_0, q_2) \dots (\pi_0, q_3)
\end{aligned}$$

and those from the left child $\pi_1$ are

$$\begin{aligned}
&(\pi_1, q_0') \dots (\pi_1, q_1') \\
&(\pi_1, q_2').
\end{aligned}$$

Note that, in the above, the set of tours from the root forms the entire run on the tree. That is, the first tour starts in an initial state and ends in a state from which the next tour starts, and so on; then, the last tour ends in a final state. Of course, it is not the case for the tours from non-root nodes. It is also worth mentioning that a tour can be a singleton run, i.e., a run of length 1, as in the second tour from $\pi_1$ above. It happens when the automaton does not walk around the subtree at all. In particular, a tour from a leaf node is always a singleton run.

Now, we construct an alternating tree automaton from this tree-walking automaton in the following way. We generate each of its states as a pair of states from the tree-walking automaton. In our example, one of the generated

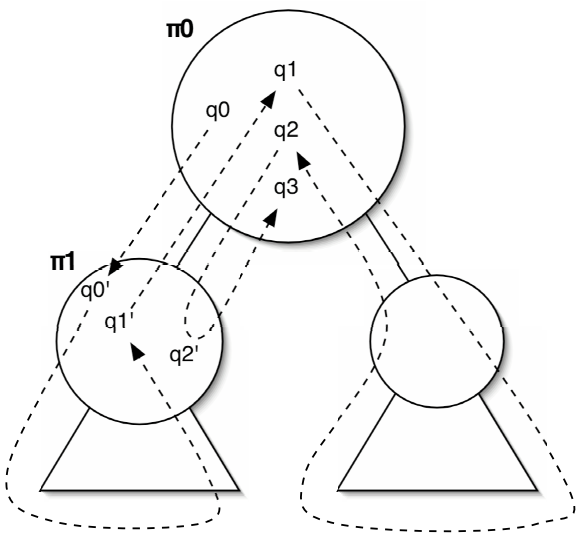



Figure 11.1: TWA transitions

states is the pair $\langle q_0, q_1 \rangle$. This intuitively means that it accepts a node $\pi$ such that there is a tour from $\pi$ starting in $q_0$ and ending in $q_1$:

$$(\pi, q_0) \ldots (\pi, q_1)$$

We also generate other states including

$$\langle q_1, q_2 \rangle \quad \langle q_2, q_3 \rangle \quad \langle q_0', q_1' \rangle \quad \langle q_2', q_2' \rangle$$

whose intuitive meanings would similarly be understood. Next, we need to form transitions among such states. The key observation is that, when there is a tour, it must be either

1. a run that makes first a left-down move, then a sub-tour from the left child, and finally an up move to the original node,

2. a run that makes first a right-down move, then a sub-tour from the left child, and finally an up move to the original node, or

3. a singleton run.

In our example, the states $\langle q_0, q_1 \rangle$ and $\langle q_2, q_3 \rangle$ fall into the first kind, the state $\langle q_1, q_2 \rangle$ into the second, and the state $\langle q_2', q_2' \rangle$ into the third (the state $\langle q_0', q_1' \rangle$ is unclear from the figure). Note then that, for the first kind of tour, if the current state of the alternating automaton has a transition that corresponds to the (first) left-down move and the (last) up move, then the sub-tour in the

middle will be taken care of by another state. Thus, if the TWA in the figure has the transitions

$$q_0 \xrightarrow{k,1} q_0' \qquad q_1' \xrightarrow{k',\mathrm{up}} q_1$$

(for some node kinds $k$, $k'$), then we add the formula

$$\downarrow_1 \langle q_0', q_1' \rangle$$

to the transition function of the alternating automaton from the state $\langle q_0, q_1 \rangle$ in order to constrain the left child. We can similarly form transitions corresponding to the second kind of tour, e.g., if the TWA has

$$q_1 \xrightarrow{k,2} q_1'' \qquad q_2'' \xrightarrow{k',\mathrm{up}} q_2$$

then we add the formula

$$\downarrow_2 \langle q_1'', q_2'' \rangle$$

to the transition from $\langle q_1, q_2 \rangle$. The third kind of tour imposes no constraint on the node and therefore we need no corresponding transition.

A slight complication is that, compared to TWA, transitions of an alternating automaton can perform a limited form of node tests and therefore we need a little care in order not to lose information during the construction. For example, from the set of pairs of TWA transitions like

$$q_0 \xrightarrow{k,1} q_0' \quad q_1' \xrightarrow{k',\mathrm{up}} q_1$$

how can we preserve, in an alternating 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 automaton cannot examine.) A trick to solve this problem is to augment each state of the alternating automaton with a node kind. The details will be shown as a formal proof.

**11.2.2 Theorem: TWA $\subseteq$ ND.**

PROOF: Give a TWA $A = (Q, I, F, \delta)$, construct $(R, R_I, R_F, \Phi)$ where:

$$R = K \times Q \times Q$$

$$R_I = \left\{ \begin{array}{l} \langle k, q_1, q_2 \rangle \wedge \langle k, q_2, q_3 \rangle \wedge \ldots \wedge \langle k, q_{n-1}, q_n \rangle \mid \\ \quad k \in \Sigma_\# \times \{\mathrm{root}\},\ q_1 \in I,\ q_2, \ldots, q_{n-1} \in Q,\ \langle q_n, k \rangle \in F \end{array} \right\}$$

$$R_F = \{\langle k, q, q \rangle \mid k \in \{\#\} \times \{\mathrm{root}, 1, 2\}\}$$

$$\Phi(\langle k, q, q' \rangle, a) = \left\{ \begin{array}{ll} \bigvee\{\downarrow_1 \langle k_1, q_1, q_1' \rangle \mid q \xrightarrow{k,1} q_1,\ q_1' \xrightarrow{k_1,\mathrm{up}} q' \in \delta\} & \\ \vee \bigvee\{\downarrow_2 \langle k_2, q_2, q_2' \rangle) \mid q \xrightarrow{k,2} q_2,\ q_2' \xrightarrow{k_2,\mathrm{up}} q' \in \delta\} & \\ \qquad\qquad \text{if } k \in \{a\} \times \{\mathrm{root}, 1, 2\} & \\ \bot & \text{otherwise} \end{array} \right.$$

In fact, the above automaton has a syntax error in the initial states: conjunction of states is not allowed there. Therefore we instead construct the alternating tree automaton $B = (R \cup R', R_I \cup R', R \cup R'_F, \Phi \cup \Phi')$ where:

$$\begin{array}{lcl} R' & = & \{\langle k, q_1, \ldots, q_n\rangle \mid k \in \Sigma_\# \times \{\text{root}\}, \\ & & \quad q_1 \in I,\ q_2, \ldots, q_{n-1} \in Q,\ \langle q_n, k\rangle \in F\} \\ R'_F & = & \{\langle k, q_1, \ldots, q_n\rangle \mid \langle k, q_1, q_2\rangle, \ldots, \langle k, q_{n-1}, q_n\rangle \in R_F\} \\ \Phi'(\langle k, q_1, \ldots, q_n\rangle, a) & = & \begin{cases} \Phi(\langle k, q_1, q_2\rangle, a) \wedge \ldots \wedge \Phi(\langle k, q_{n-1}, q_n\rangle, a) \\ \qquad\qquad \text{if } k \in \{a\} \times \{\text{root}, 1, 2\} \\ \bot \qquad\qquad \text{otherwise} \end{cases} \end{array}$$

To prove that $t \in L(A) \Leftrightarrow t \in L(B)$, it suffices to show both that

> there is a tour $(\pi, q) \ldots (\pi, q')$ in $A$ on $t$ if and only if $B$ accepts $\text{subtree}_t(\pi)$ at $\langle \kappa_t(\pi), q, q'\rangle$.

and that

> there are tours
> $$\begin{array}{l} (\pi, q_1) \ldots (\pi, q_2) \\ \ldots \\ (\pi, q_{n-1}) \ldots (\pi, q_n) \end{array}$$
> in $A$ on $t$ if and only if $B$ accepts $\text{subtree}_t(\pi)$ at
> $$\langle \kappa_t(\pi), q_1, \ldots, q_n\rangle.$$

The proof can be done by induction on the structure of $\pi$. □

**11.2.3 Exercise:** Complete the proof of Theorem 11.2.2

As mentioned, the converse does not hold and therefore the inclusion is strict. The proof is very difficult.

**11.2.4 Theorem [Bojańczyk and Colcombat, 2004]: TWA $\subsetneq$ ND.**

### 11.2.3 Variations

#### Deterministic Tree-Walking Automata

A TWA $(Q, I, F, \delta)$ 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 only a single way of running 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 by Bojańczyk and Colcombat that deterministic TWA are strictly less expressive than nondeterministic ones. The proof of this property is also complex.

**11.2.5 Theorem [Bojańczyk and Colcombat, 2004]: DTWA $\subsetneq$ TWA.**

### Alternating Tree-Walking Automata

In Section 11.1.1, we have seen examples of using path expressions themselves as node tests and discussed that this cannot be expressed by caterpillar expressions or, equivalently, by TWA. A direct way of allowing such node tests is to extend TWA with 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 (ATWA) is a quadruple $(Q, I, F, \delta)$ where $Q$, $I$, and $F$ are the same as in TWA and $\delta$ 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 $d \in \{\mathrm{up}, 1, 2\}$. Given an ATWA, a tree where each node is labeled a pair of the form $(\pi, q)$ is a *run* if, for each intermediate node that is labeled $(\pi, q)$ and whose children are labeled $(\pi_1, q_1), \ldots, (\pi_n, q_n)$, there is a set of transitions

$$\begin{array}{l} q \xrightarrow{\kappa_t(\pi), d_1} q_1 \\ \ldots \\ q \xrightarrow{\kappa_t(\pi), d_n} q_n \end{array}$$

where each directions $d_i$ matches the positional relationship between $\pi$ and $\pi_i$. A run is *successful* if the root is labeled $(\epsilon, q)$ with $q \in I$ and every leaf is labeled $(\epsilon, q)$ and has $k$ with $(q, k) \in F$. An ATWA *accepts* a tree when there is a successful run and we write **ATWA** be the class of languages accepted by ATWA.

It is rather easy to show that ATWA have at least the expressive power of FTA since each FTA transition $q \to a(q_1, q_2)$ can (roughly) be considered to be the conjunction of two ATWA transitions $q \xrightarrow{k,1} q_2$ and $q \xrightarrow{k,2} q_2$ for an appropriate $k$. Note that, for this encoding, ATWA do not need the ability to move up.

**11.2.6 Theorem: ND $\subseteq$ ATWA**.

It is also known that the converse also holds. In other words, the backward axis does not increase the expressiveness of ATWA. The proof is omitted here and can be found in, e.g., [83].

**11.2.7 Theorem: ATWA $\subseteq$ ND**.

## 11.3 Bibliographic Notes

The specification of XPath is available in [21]. Regular expression paths have been proposed by database researchers [1, 16, 25]. Caterpillar expressions are proposed by Brüggemann-Klein and Wood [14]. Their paper presents *caterpillar automata*, but, in fact, this notion is identical to tree-walking automata. An

extension of path expressions has been proposed that has the same expressive power as regular expression patterns with one variable [71].

Tree-walking automata were first introduced by Aho and Ullman [2]. In the initial formulation, automata do not have access to node kinds. It is known that such automata are incapable of systematically traversing all the nodes of a tree as in Section 11.1.2. For such automata, strict weakness relative to tree automata was shown by Kamimura and Slutzki [55]. With the addition of the access to node kinds, the same property was proved by Bojańczyk and Colcombet [8]. The same authors also proved that tree-walking automata cannot be determinized [7]. Equivalence between tree-walking automata and their alternating variation is shown by Slutzki [83].

As a separate line of work, extensive investigations have been made on expressiveness and complexity properties of various fragments of XPath. A comprehensive survey can be found in [4].

# Chapter 12

# Ambiguity

As introduced in Chapter 2, ambiguity refers to the property that regular expressions or patterns have multiple possibilities of matching. Ambiguity can make the behavior of the program harder to understand and can actually be a programming error. Therefore it is sometimes useful to report such an ambiguity to the user.

When it comes to ask what exactly we mean by ambiguity, there is no single consensus. In Chapter, we review three different definitions. Two, called strong and weak ambiguities, concern how a regular expression matches an input. The third, called binding-ambiguity, concerns how a pattern yields bindings from an input. We will study how these notions are related each other and how these can be checked algorithmically.

Caveat: In this chapter, we concentrate on regular expressions and patterns on *strings* rather trees for highlighting the essence of ambiguity. Extending for the case of trees is routine and left for exercises.

## 12.1 Ambiguities for Regular Expressions

In this section, we study what strong and weak ambiguities are and how these can be decided by using a checking algorithm for ambiguity for automata.

### 12.1.1 Definitions

First of all, ambiguity arises when a regular expression has several occurrences of the same symbol. Therefore we need to be able to distinguish between these occurrences. Let $\tilde{\Sigma}$ be the set of *elaborated symbols*, written by $a^{(i)}$, where $a \in \Sigma$ and $i$ is an integer. We use $\tilde{s}$ to range over elaborated strings from $\tilde{\Sigma}^*$ and $s$ over strings from $\Sigma^*$. 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 $s$ is the *unelaboration* of $\tilde{s}$; we write unelab($\tilde{s}$) for such $s$. Let $r$ range over regular expressions over $\tilde{\Sigma}$

where every occurrence of symbol is given a unique integer, e.g., $a^{(1)^*}b^{(2)}a^{(3)^*}$. Throughout this section, we only consider such regular expressions.

We define strong ambiguity in terms of the *derivation* relation $s\ in\ r$ given by the following set of rules (which is much like the rules for the conformance relation in Section 2.2).

$$\frac{}{\epsilon\ in\ \epsilon}\ \text{T-Eps}$$

$$\frac{}{a\ in\ a^{(i)}}\ \text{T-Sym}$$

$$\frac{s\ in\ r_1}{s\ in\ r_1 \,|\, r_2}\ \text{T-Alt1}$$

$$\frac{s\ in\ r_2}{s\ in\ r_1 \,|\, r_2}\ \text{T-Alt2}$$

$$\frac{s_1\ in\ r_1 \qquad s_2\ in\ r_2}{s_1 s_2\ in\ r_1 r_2}\ \text{T-Cat}$$

$$\frac{s_i\ in\ r \qquad 1 \le i \le n}{s_1 \ldots s_n\ in\ r^*}\ \text{T-Rep}$$

Then, a regular expression $r$ is *strongly unambiguous* if, for any string $s$, there is at most one derivation of $s\ in\ r$.

**12.1.1 Example:** The regular expression $r_{12.1.1} = (a^{(1)^*})^*$ is strongly ambiguous since there are at least two derivations of $aa\ in\ r_{12.1.1}$.

$$\frac{\dfrac{\dfrac{a\ in\ a^{(1)}}{a\ in\ a^{(1)^*}} \quad \dfrac{a\ in\ a^{(1)}}{a\ in\ a^{(1)^*}}}{aa\ in\ (a^{(1)^*})^*} \quad \dfrac{\dfrac{a\ in\ a^{(1)} \quad a\ in\ a^{(1)}}{aa\ in\ a^{(1)^*}} \quad \epsilon\ in\ a^{(1)^*}}{aa\ in\ (a^{(1)^*})^*}$$

**12.1.2 Example:** The regular expression $r_{12.1.2} = a^{(1)} \,|\, a^{(2)}$ is also strongly ambiguous since there are at least two derivations of $a\ in\ r_{12.1.2}$.

$$\frac{a\ in\ a^{(1)}}{a\ in\ a^{(1)} \,|\, a^{(2)}} \quad \frac{a\ in\ a^{(2)}}{a\ in\ a^{(1)} \,|\, a^{(2)}}$$

Note that elaboration of the regular expression makes these two derivations distinct.

Let the language $L(r) \subseteq \tilde{\Sigma}^*$ of a regular expression $r$ be defined in the standard way; we also say that $r$ *generates* an element of $L(r)$. Note that $s\ in\ r$ if and only if there is a string $\tilde{s} \in L(r)$ such that $s = \text{unelab}(\tilde{s})$. Then, $r$ is *weakly unambiguous* if $r$ generates at most one elaboration of $s$ for any string $s$.

**12.1.3 Example:** The regular expression $r_{12.1.2}$ in Example 12.1.2 is weakly unambiguous since it generates only strings of the form $a^{(1)} \dots a^{(1)}$. The regular expression $r_{12.1.1}$ in Example 12.1.1 is weakly ambiguous since it generates $a^{(1)}$ and $a^{(2)}$.

**12.1.4 Example:** The regular expression $r_{12.1.4} = (a^{(1)} \mid a^{(2)}b^{(3)})(b^{(4)} \mid \epsilon)$ is strongly ambiguous since there are two derivations for $ab\ in\ r_{12.1.4}$:

$$\dfrac{\dfrac{a\ in\ a^{(1)}}{ab\ in\ (a^{(1)} \mid a^{(2)}b^{(3)})} \quad \dfrac{b\ in\ b^{(4)}}{b\ in\ (b^{(4)} \mid \epsilon)}}{ab\ in\ (a^{(1)} \mid a^{(2)}b^{(3)})(b^{(4)} \mid \epsilon)} \qquad \dfrac{\dfrac{\dfrac{a\ in\ a^{(2)} \quad b\ in\ b^{(3)}}{ab\ in\ a^{(2)}b^{(3)}}}{ab\ in\ (a^{(1)} \mid a^{(2)}b^{(3)})} \quad \dfrac{\epsilon\ in\ \epsilon}{\epsilon\ in\ (b^{(4)} \mid \epsilon)}}{ab\ in\ (a^{(1)} \mid a^{(2)}b^{(3)})(b^{(4)} \mid \epsilon)}$$

The regular expression is also weakly ambiguous since it generates $a^{(1)}b^{(4)}$ and $a^{(2)}b^{(3)}$.

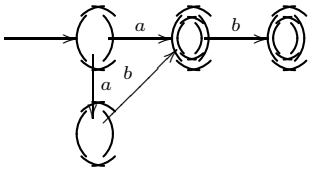
As in Example 12.1.4, for a derivation of $s\ in\ r$, the concatenation (from left to right) of the elaborated symbols appearing in the leaves coincides an $s$'s elaboration generated by $r$. Therefore, if $r$ generates two different elaborations of $s$, then the derivations corresponding to these must be different. This observation leads to the following proposition.

**12.1.5 Proposition:** If a regular expression $r$ is strongly unambiguous, then it is also weakly unambiguous.

The converse does not hold as Example 12.1.1 is a counterexample.

Given a (string) automaton $A = (Q, I, F, \delta)$, a *path* is a sequence $q_1, \dots, q_n$ of states from $Q$ such that $q_i \xrightarrow{a_i} q_{i+1} \in \delta$ for each $i = 1, \dots, n-1$. Such a path *accepts* $s$ when $s = a_1 \dots a_{n-1}$. Then, a automaton is *unambiguous* if, for any string $s \in \Sigma^*$, there is at most one path accepting $s$.

**12.1.6 Example:** The following automaton



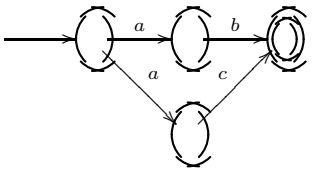

is ambiguous since there are two paths accepting $ab$.

The following proposition follows obviously.

**12.1.7 Proposition:** If a automaton is deterministic, then it is also unambiguous.

The converse does not hold as the following shows a counterexample.

**12.1.8 Example:** The automaton




is nondeterministic yet unambiguous.

### 12.1.2 Glushkov Automata and Star Normal Form

The next question that we are interested in is the relationship among strong ambiguity, weak ambiguity, and ambiguity for automata. The key concepts connecting these are Glushkov automata and star normal form.

For an (elaborated) regular expression $r$, we define the following.

$$\begin{array}{lcl} \mathbf{pos}(r) & = & \{a^{(i)} \mid \tilde{s}a^{(i)}\tilde{s}' \in L(r)\} \\ \mathbf{first}(r) & = & \{a^{(i)} \mid a^{(i)}\tilde{s} \in L(r)\} \\ \mathbf{last}(r) & = & \{a^{(i)} \mid \tilde{s}a^{(i)} \in L(r)\} \\ \mathbf{follow}(r, a^{(i)}) & = & \{b^{(j)} \mid \tilde{s}a^{(i)}b^{(j)}\tilde{s}' \in L(r)\} \end{array}$$

Intuitively, these sets contain the elaborated symbols 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.

**12.1.9 Exercise:** Give an algorithm for computing these sets.

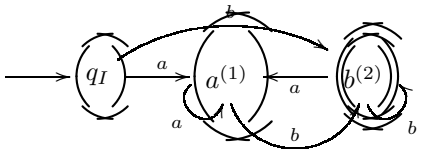
The *Glushkov automaton* $M_r$ of a regular expression $r$ is $(Q \cup \{q_I\}, \{q_I\}, F, \delta)$ where

$$\begin{array}{lcl} Q & = & \mathbf{pos}(r) \\ q_I & \notin & Q \\ F & = & \begin{cases} \mathbf{last}(r) \cup \{q_I\} & \epsilon \in L(r) \\ \mathbf{last}(r) & \epsilon \notin L(r) \end{cases} \\ \delta & = & \{q_I \xrightarrow{b} b^{(j)} \mid b^{(j)} \in \mathbf{first}(r)\} \\ & \cup & \{a^{(i)} \xrightarrow{b} b^{(j)} \mid a^{(i)} \in \mathbf{pos}(e),\ b^{(j)} \in \mathbf{follow}(r, a^{(i)})\}. \end{array}$$

That is, the automaton $M_r$ has $r$'s elaborated symbols as states in addition to a fresh state $q_I$ used as an initial state. If the automaton reads a symbol $b$ in the state $q_I$, then it takes one of $r$'s first elaborated symbol $b_j$ as the state to transit. If $M_r$ reads $b$ in a state $a^{(i)}$, then it takes, as the next state, one of $r$'s elaborated symbol $b^{(j)}$ that follows after $a^{(i)}$. $M_r$ halts in a last elaborated symbol $b_j$ or in $q_I$ in case where $e$ allows the empty sequence.

**12.1.10 Example:** Let $r_{12.1.10} = (a^{(1)^*}b^{(2)^*})^*$. Then, $\mathbf{pos}(r_{12.1.10}) = \mathbf{first}(r_{12.1.10}) = \mathbf{last}(r_{12.1.10}) = \mathbf{follow}(r_{12.1.10}, a^{(1)}) = \mathbf{follow}(r_{12.1.10}, b^{(2)}) = \{a^{(1)}, b^{(2)}\}$.

The automaton $M_{r_{12.1.10}}$ 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, the regular expression is weakly unambiguous iff its Glushkov automaton is unambiguous. This property can easily be obtained from the fact that there is one-to-one correspondence between a word in the regular expression and an accepting path in the automaton.

**12.1.11 Lemma:** ${a_1}^{(i_1)} \dots {a_n}^{(i_n)} \in L(r)$ iff there is a path $q_I {a_1}^{(i_1)} \dots {a_n}^{(i_n)}$ from an initial state to a final state.

**12.1.12 Exercise:** Prove Lemma 12.1.11.

**12.1.13 Corollary:** $M_r$ accepts $s$ iff $s$ *in* $r$. Moreover, $r$ is weakly unambiguous iff $M_r$ is unambiguous.

**12.1.14 Example:** The regular expression $r_{12.1.10}$ in Example 12.1.10 is weakly unambiguous and $M_{r_{12.1.10}}$ is also unambiguous.

**12.1.15 Exercise:** Construct the Glushkov automaton of the regular expression $r_{12.1.4}$ in Example 12.1.4 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 the *star normal form* if, for every $r$'s subexpression of the form $d^*$, the following condition (SNF condition) holds.

$$\mathbf{follow}(d, \mathbf{last}(d)) \cap \mathbf{first}(d) = \emptyset$$

Here, we generalize the definition of **follow** so that $\mathbf{follow}(r, S) = \cup\{\mathbf{follow}(r, a^{(i)}) \mid a^{(i)} \in S\}$ for a set $S$ of elaborated symbols. 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 Kleene star makes it ambiguous.

**12.1.16 Example:** The regular expression $(a^{(1)}b^{(2)^*})^*$ is in the star normal form whereas $(a^{(1)^*}b^{(2)^*})^*$ is not.

A regular expression $r$ is in the *epsilon normal form* if, for every $r$'s subexpression $d$, there is at most one derivation of $\epsilon$ *in* $d$ (the $\epsilon$-NF condition).

**12.1.17 Example:** The regular expression $a^{(1)^*} \mid b^{(2)^*}$ is not in the epsilon normal form.

Having these two definitions of normal forms, we can describe the following connection between strong and weak ambiguities.

**12.1.18 Theorem [Brüggemann-Klein, 1996]:** A regular expression $r$ is strongly unambiguous if and only if $r$ is weakly unambiguous, in the star normal form, and in the 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:** $r = r_1 r_2$

Suppose that $r$ is strongly ambiguous. Then, there are two derivations of $s \; in \; r$ for a string $s$. But, 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 $s_2 \neq \epsilon$ with

$$\begin{array}{ll} s_1 s_2 \; in \; r_1 & s_3 \; in \; r_2 \\ s_1 \; in \; r_1 & s_2 s_3 \; in \; r_2. \end{array}$$

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

$$\begin{array}{ll} \tilde{s}_1 \tilde{s}_2 \in L(r_1) & \tilde{s}_3 \in L(r_2) \\ \tilde{s}'_1 \in L(r_1) & \tilde{s}'_2 \tilde{s}'_3 \in L(r_2). \end{array}$$

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 different elaborations $\tilde{s}_1 \tilde{s}_2 \tilde{s}_3$ and $\tilde{s}'_1 \tilde{s}'_2 \tilde{s}'_3$ of $s$ are in $r$, contradicting the assumption that $r$ is weakly unambiguous.

**Case:** $r = r_1 \mid r_2$

Similarly to the last case, suppose that $r$ is strongly ambiguous. Then, there are two derivations of $s \; in \; r$ for a string $s$. But, 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

$$\tilde{s} \in L(r_1) \quad \tilde{s}' \in L(r_2).$$

Since $r$ is in the 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 two different elaborations of $s$ are in $r$, contradicting the assumption that $r$ is weakly unambiguous.

**Case:** $r = {r_1}^*$

Similarly to the above cases, suppose that $r$ is strongly ambiguous. Then, there are two derivations of $s\ in\ r$ for a string $s$. But, 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 $s_1, \ldots, s_n, s'_1, \ldots, s'_m$ such that $s = s_1 \ldots s_n = s'_1 \ldots s'_m$ 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 $s'_i$ is $\epsilon$ since, otherwise, $r_1$ generates $\epsilon$ and therefore obviously $r$ becomes not in the epsilon normal form. Then, there are elaborations $\tilde{s}_i$ of $s_i$ and $\tilde{s}'_i$ of $s'_i$ such that

$$\begin{array}{ll} \tilde{s}_i \in L(r_1) & i = 1, \ldots, n \\ \tilde{s}'_i \in L(r_1) & i = 1, \ldots, m \end{array}$$

From $(s_1, \ldots, s_n) \neq (s'_1, \ldots, s'_m)$, we have that $\tilde{s}_1 = \tilde{s}'_1, \ldots, \tilde{s}_{k-1} = \tilde{s}'_{k-1}$ 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 symbol of $\tilde{s}'_k$ and $f$ be the first 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 12.1.5, we only need to show that, if $r$ is strongly unambiguous, then $r$ is both in the star normal form and in the 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 $\epsilon$-NF condition. To reach the contradition that $r$ is strongly ambiguous, it suffices to show that $d$ is strongly ambiguous. In the case that $d$ breaks the $\epsilon$-NF condition, $d$ is trivially strongly ambiguous by definition. In the case that $d = {d'}^*$ 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 strings $sa^{(i)}$ and $b^{(j)}s'$ in $d'$ for some $s, s'$. Since these imply that the Glushov automaton $M_{d'}$ contains a path accepting $sa^{(i)}$ and one accepting $b^{(j)}s'$ as well as a transition from $a^{(i)}$ to $b^{(j)}$, it also contains a path accepting $sa^{(i)}b^{(j)}s'$, that is, $d'$ also generates $sa^{(i)}b^{(j)}s'$. Therefore, there are at least two derivations of $sa^{(i)}b^{(j)}s'\ in\ {d'}^*$:

$$\dfrac{\dfrac{\ldots \qquad \ldots}{sa^{(i)}\ in\ d' \quad b^{(j)}s'\ in\ d'}}{sa^{(i)}b^{(j)}s'\ in\ {d'}^*} \qquad \dfrac{\dfrac{\ldots}{sa^{(i)}b^{(j)}s'\ in\ d'}}{sa^{(i)}b^{(j)}s'\ in\ {d'}^*}$$

Hence, $d$ is strongly ambiguous. □

The star normal form and the epsilon normal form are rather easy to check. Thus, by Theorem 12.1.18, we can reduce the check for strong ambiguity to that for weak ambiguity.

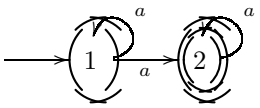
**12.1.19 Exercise:** Find algorithms to check the star normal form and the epsilon normal form.

### 12.1.3 Ambiguity Checking for Automata

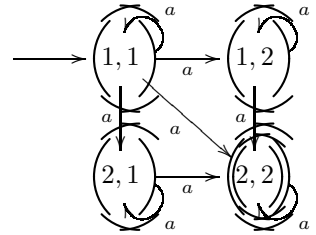
Given an automaton $A = (Q, I, F, \delta)$, the following checks whether $A$ is ambiguous or not.

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 $C$ after eliminating the useless states from $B$. (See Section 6.3.1 for useless state elimination.)
3. Answer "unambiguous" iff every state of $C$ has the form $(q, q)$.

**12.1.20 Example:** From the automaton

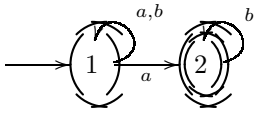
$A_{12.1.20} =$ 


the first step yields:

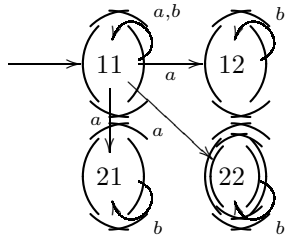
$B_{12.1.20} =$ 


(Note that the diagram of a self-product is always symmetric.) The second step returns the same automaton $C_{12.1.20} = B_{12.1.20}$. Thus, $A_{12.1.20}$ is ambiguous since $C_{12.1.20}$ has the states $(1, 2)$ and $(2, 1)$.

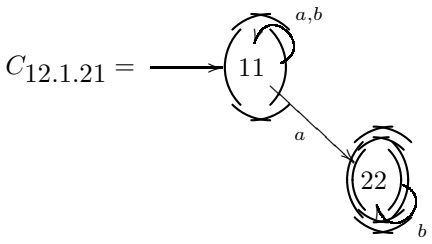
**12.1.21 Example:** From the automaton

$A_{12.1.21} =$ 


the first step yields:

$B_{12.1.21} =$ 

Then, the second step eliminates the states $(1,2)$ and $(2,1)$, resulting in:




Thus, $A_{12.1.21}$ is unambiguous since $C_{12.1.21}$ has only states of the form $(q,q)$.

**12.1.22 Theorem:** The ambiguity checking algorithm returns "unambiguous" for an automaton $A$ if and only $A$ is unambiguous.

PROOF: Suppose that the automaton $A$ is ambiguous. Then, there are two paths from initial states to final states

$$q_1, q_2, \ldots, q_n$$
$$r_1, r_2, \ldots, r_n$$

both accepting a string $s$ where $q_i \neq r_i$ for some $1 \leq i \leq n$. Therefore the automaton $B$ also has a path

$$(q_1, r_1), (q_2, r_2), \ldots, (q_n, r_n)$$

from an initial state to a final state accepting $s$. Since $(q_i, r_i)$ is in this path, this state remains in the automaton $C$. Thus, the algorithm returns "ambiguous."

Conversely, suppose the algorithm returns "ambiguous." Then, there is a state $(q,r)$ in $C$ with $q \neq r$. Since this states remain in $C$, there is, in $C$, a path from an initial state to a final state passing through $(q,r)$. Let the path accept a string $s$. Then, $A$ has two different path from initial states to final states accepting $s$ where one passes through $q$ and another through $r$. Therefore $A$ is ambiguous. □

## 12.2 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 alright when 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 pattern

$$((a^*)^* \text{ as } x).$$

(In this section, we consider patterns as defined in Chapter 4 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 of programming, we may want to incorporate a part $T$ of an externally provided schema as a subexpression of pattern:

$$(T \text{ as } x)$$

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 define that a pattern is *binding-unambiguous* if it yields at most one binding for any input. However, there is one trickiness in this definition. Should we regard a pattern like

$$((a \text{ as } x), a) \mid (a, (a \text{ as } x))$$

as ambiguous or not? This pattern matches only the string $aa$ and does yield only one binding $x \mapsto a$. However, the $a$ in the binding comes from either the first symbol or the second in the input. If we take the "structural view" that these two symbols 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 only the physical view since it is algorithmically simpler and tricky cases like the above example rarely occur in practice.

### 12.2.1 Definitions

Let us formalize "physical" binding-ambiguity, first for patterns and then for marking automata. For patterns, we need to define a semantics of patterns that takes positions in input and output strings. Thus, we define the matching relation $\tilde{s} \in P \Rightarrow \tilde{V}$ where $\tilde{s}$ is an elaborated string, $P$ is a pattern (with no contents in labels), and $\tilde{V}$ is a mapping from variables to elaborated strings. We assume that each symbol in elaborated strings is given a unique integer. The matching relation is defined by the same set of rules as in Section 4.4 (without pattern definitions $F$ and priority ids $I$) except that rule P-Elm is replaced by the following.

$$\frac{}{a^{(i)} \in a \Rightarrow ()}$$

Then, a pattern $P$ is *binding-unambiguous* if $\tilde{s} \in P \Rightarrow \tilde{V}$ and $\tilde{s} \in P \Rightarrow \tilde{V}'$ imply $\tilde{V} = \tilde{V}'$ for any $\tilde{s}$.

For marking automata on strings (whose definition is identical to those on trees except that each transition has only one destination state), we define ambiguity in terms of markings yielded from them. A marking 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 for elaborated strings has the same effect as marking substrings with variables, we can easily prove the following.

**12.2.1 Proposition:** Let $P$ be a pattern and $A$ be a marking automaton constructed from $P$. Then, $P$ is binding-unambiguous iff $A$ is marking-unambiguous.

Note that the proposition does not specify how the marking automaton is constructed. Indeed, this property holds regardless to which automata construction is used.

### 12.2.2 Algorithm

Given a marking automaton $A = (Q, I, F, \delta, \Xi)$, the following checks whether $A$ is marking-ambiguous or not.

1. Calculate $B = (Q \times Q, I \times I, F \times F, \delta', \Xi')$ where $\delta'$ is the same as in Section 12.1.3 and $\Xi'$ is as follows:

$$\Xi'((q, r)) = (\Xi(q), \Xi(r))$$

2. Obtain the automaton $C$ after eliminating the useless states from $B$.

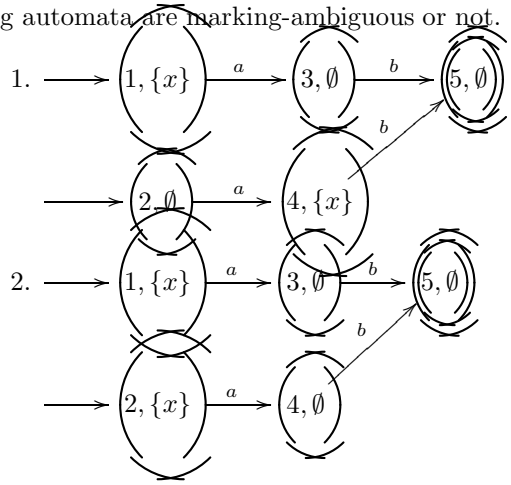
3. Answer "marking-unambiguous" iff $\Xi'((q, r))$ has the form $(X, X)$ for every state $(q, r)$ of $C$.

The algorithm works quite similarly to the ambiguity checking algorithm shown in Section 12.1.3 except for the last step, where we check how the automaton marks each node with variables rather than how it assigns each node with states. More precisely, the last step ensures that, for any two paths accepting the same string, the same set of variables is assigned to each position. Thus, the following expected property can be proved similarly to Theorem 12.1.22

**12.2.2 Theorem:** The marking-ambiguity checking algorithm returns "marking-unambiguous" for an automaton $A$ if and only $A$ is marking-unambiguous.

**12.2.3 Exercise:** Prove Theorem 12.2.2.

**12.2.4 Exercise:** Apply the above algorithm for checking whether the following automata are marking-ambiguous or not.



## 12.3 Bibliographic Notes

Ambiguity for string regular expressions and automata has been a classical question [9, 11]. In [11], Brüggemann-Klein introduced the terminology of strong and weak ambiguities with their relationship based on Glushkov automata and star normal form. 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 is a folklore. For tree grammars, there are algorithms for weak ambiguity [56] and for strong ambiguity [45]. A study on one-unambiguous regular expressions can be found in [13]. Marking-ambiguity was first introduced by Hosoya, Frisch, and Castagna as a way to ensure the uniqueness of solutions to a certain type inference problem arising in parametric polymorphism for XML [47]. Although the purpose was different from the present chapter, the definition was the same.

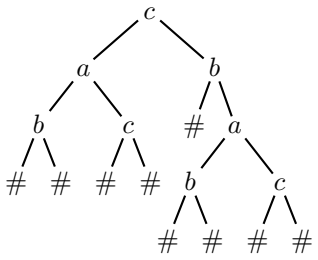
# Chapter 13

# Logic-based Queries

In this chapter, we revisit the question of how we express a subtree extraction from an XML document. For answering the same question, we have already presented two formalisms, pattern matching in Chapter 4 and path expressions in Chapter 11. Here, we are going to introduce the third one, namely, the first-order (predicate) logic and its extension the monadic second-order logic (MSO). As we will see soon, one of the advantages of using logic-based queries is that it allows a direct way of expressing retrieval conditions. That is, we do not need to specify the structure of a whole tree (like patterns) or do not need to navigate around (like paths); we simply need to specify logical relationships among relevant nodes. Further, MSO has the additional advantage that it can express any regular queries, that is, it is at least as powerful as marking tree automata (and thus as pattern matching), defined in Chapter 5. On the other hand, MSO can express not more than regular queries; indeed, any MSO formula can be translated to a marking tree automaton. This means, however, that we can use efficient execution algorithms that we have already learned in Chapter 7.

## 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, models are XML documents; for simplicity, let us again take binary trees here. Then, we can say that a given (binary) tree satisfies a certain logic formula. For example, the following 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 $a$; ditto for $b(y)$ and $c(z)$. Also, $\text{leftchild}(x, y)$ means that $y$ is the left child of $x$; ditto for $\text{rightchild}(x, y)$. So the whole formula can be read "for any node with $a$ label, its left and right children exist with label $b$ and $c$ respectively." Further, a formula that contains free variables is interpreted with a tree *and* an assignment of variables to its nodes. For example, the above tree with the following variable assignment

$$\{y \mapsto 11,\ z \mapsto 12\}$$

($y$ and $z$ are mapped to the left-most $b$ 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 $x$ that has $y$ as the left child and $z$ as the right child"). Note that the following assignment

$$\{y \mapsto 221,\ z \mapsto 222\}$$

($y$ and $z$ are mapped 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. Thus, we can take a logic formula as a *query* on a tree that yields a set of variable assignments.

Formally, the syntax of (first-order) formulas $\psi$ is defined by the following grammar, where $x$ ranges over (first-order) variables.

| | | | |
|---|---|---|---|
| $\psi$ | $::=$ | $R_u(x)$ | unary relation |
| | | $R_b(x_1, x_2)$ | 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 unary relation $R_u$, we have each label $a$ from $\Sigma$ as well as root and leaf. Their formal meanings are: given a node $\pi$ in a tree $t$,

$$\begin{aligned} a(\pi) &\overset{\text{def}}{\iff} \text{label}_t(\pi) = a \\ \text{root}(\pi) &\overset{\text{def}}{\iff} \pi = \epsilon \\ \text{leaf}(\pi) &\overset{\text{def}}{\iff} \text{label}_t(\pi) = \#. \end{aligned}$$

As a binary relation, we have leftchild and rightchild as already introduced above as well as the equality $=$, the self-or-descendent relation $\leq$, and the document

order $\preceq$. Their formal meanings are: given nodes $\pi_1$, $\pi_2$ in a tree $t$,

$$\begin{aligned}
\text{leftchild}(\pi_1, \pi_2) &\stackrel{\text{def}}{\iff} \pi_2 = 1\pi_1 \\
\text{rightchild}(\pi_1, \pi_2) &\stackrel{\text{def}}{\iff} \pi_2 = 2\pi_1 \\
\pi_1 \leq \pi_2 &\stackrel{\text{def}}{\iff} \exists \pi.\ \pi_2 = \pi\pi_1 \\
\pi_1 \preceq \pi_2 &\stackrel{\text{def}}{\iff} \pi_2 \text{ is equal to or larger than } \pi_1 \text{ by lexicographic order on } \{1,2\}^*.
\end{aligned}$$

(The equality $=$ means itself.) A formula $\psi$ is interpreted in terms of a binary tree $t$ and a function $\gamma$ that maps each free variable of $\psi$ to a node in $t$. The semantics is described by the judgment of the form $t, \gamma \vdash \psi$, read "under tree $t$ and assignment $\gamma$, formula $\psi$ is satisfied" and defined inductively by the following rules.

$$\begin{aligned}
t, \gamma \vdash R_u(x) &\stackrel{\text{def}}{\iff} R_u(\gamma(x)) \\
t, \gamma \vdash R_b(x_1, x_2) &\stackrel{\text{def}}{\iff} R_u(\gamma(x_1), \gamma(x_2)) \\
t, \gamma \vdash \psi_1 \wedge \psi_2 &\stackrel{\text{def}}{\iff} t, \gamma \vdash \psi_1 \text{ and } t, \gamma \vdash \psi_2 \\
t, \gamma \vdash \psi_1 \vee \psi_2 &\stackrel{\text{def}}{\iff} t, \gamma \vdash \psi_1 \text{ or } t, \gamma \vdash \psi_2 \\
t, \gamma \vdash \psi_1 \Rightarrow \psi_2 &\stackrel{\text{def}}{\iff} t, \gamma \vdash \psi_1 \text{ implies } t, \gamma \vdash \psi_2 \\
t, \gamma \vdash \neg\psi_1 &\stackrel{\text{def}}{\iff} \text{not } t, \gamma \vdash \psi_1 \\
t, \gamma \vdash \forall x.\ \psi &\stackrel{\text{def}}{\iff} \text{for all } \pi \in \text{nodes}(t),\ \ t, \gamma \cup \{x \mapsto \pi\} \vdash \psi \\
t, \gamma \vdash \exists x.\ \psi &\stackrel{\text{def}}{\iff} \text{for some } \pi \in \text{nodes}(t),\ \ t, \gamma \cup \{x \mapsto \pi\} \vdash \psi
\end{aligned}$$

In Chapters 4 and 11, we have introduced other formalisms for subtree extraction, namely, pattern matching and path expressions. What is an advantage of logic-based queries over these?

The most important is its ability to *directly* express retrieval conditions. In patterns, we need to specify the structure of the whole tree to match even though there are usually much more nodes irrelevant to the query than relevant ones. Here, irrelevant nodes mean that we *don't care* whether these nodes exist or not. On the other hand, logical framework has a built-in "don't-care" semantics in the sense that it allows us even not to mention such nodes at all. Moreover, when we want to extract a node from deeper places in the tree, pattern matching require us to write a recursive definition (Section 4.2.1) whereas logic allows us to directly jump at 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 can also be defined by using the `~` label and recursion). In logic, we only need to write:

$$a(x)$$

Unlike patterns, we do not need to mention any nodes other than the node we are interested in; we do not need to form a recursion to reach a node that may be located at an arbitrarily deep position. Path expressions may seem to be better than patterns in this sense since we only need to write the following (in the caterpillar expression notation) for expressing the same:

$$(1|2)^*a$$

That is, 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 a *navigation* from the root, which is not necessary in logic. What concerns here is not the visual succinctness but the number of concepts that are involved in expressing the same condition.

Let us see 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 belonging `h2` nodes, by which we can form a list of sections each with a list of subsections). To express this query in logic, it would look like the following.

$$h1(x) \wedge h2(y) \wedge x \preceq y \wedge \forall z.\ (h1(z) \wedge x \preceq z \Rightarrow y \preceq z)$$

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, since XHTML headings may not appear in a flat list but can be intervened by other tags, 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 and the second `h2`s, but not with the last `h2` since the second `h1` appears inbetween.

How can we express this example by a pattern? It is rather difficult mainly because we need the document order; indeed, if we simplify the example so as to consider only a flat list of heading tags, then we can write the following.

```
Any, (h1[Any] as x), (^h1)[Any]*, (h2[Any] as y), Any
```

(Here, we introduce the notation `^h1` to match any label except `h1`.) Then, one might argue that logic wins because it has the document order as a primitive. However, as we will see in the next section, the document order itself can easily be expressed by slightly extending the logical framework, namely, to MSO.

What about expressing the above example by a path expression? First, we can easily extend paths by a "document order axis." However, even with this, it is still not possible since we need universal quantification. Path expressions, essentially, can combine only existential conditions on nodes.

## 13.2 Monadic Second-order Logic

Monadic second-order 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 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.

$$\begin{array}{rcll} \psi & ::= & \ldots & \\ & & x \in X & \text{variable unary relation} \\ & & \forall X.\ \psi & \text{second-order universal quantification} \\ & & \exists X.\ \psi & \text{second-order existential quantification} \end{array}$$

Accordingly, the semantics is extended so that a formula is interpreted under a second-order variable assignment $\Gamma$ (a mapping from second-order variables to sets of nodes) in addition to a tree and a first-order assignment. We thus extend the satisfaction judgment as $t, \gamma, \Gamma \vdash \psi$ and define it by the following rules

$$\begin{array}{lcl} t, \gamma, \Gamma \vdash x \in X & \stackrel{\text{def}}{\iff} & \gamma(x) \in \Gamma(X) \\ t, \gamma, \Gamma \vdash \forall X.\ \psi & \stackrel{\text{def}}{\iff} & \text{for all } \Pi \subseteq \text{nodes}(t),\quad t, \gamma, \Gamma \cup \{X \mapsto \Pi\} \vdash \psi \\ t, \gamma, \Gamma \vdash \exists X.\ \psi & \stackrel{\text{def}}{\iff} & \text{for some } \Pi \subseteq \text{nodes}(t),\quad t, \gamma, \Gamma \cup \{X \mapsto \Pi\} \vdash \psi \end{array}$$

plus exactly the same rules as first-order logic except that $\Gamma$ is additionally passed around.

Let us see how powerful second-order quantification is. To warm up, let us define the relation $\text{xchild}(x, y)$ to mean that $y$ is a child of $x$ in the unranked tree (XML) view. For this, we first define the auxiliary relation $\text{xchildren}(x, Y)$ to mean that $Y$ is the set of children of $x$:

$$\text{xchildren}(x, Y) \equiv \forall y.\ y \in Y \Leftrightarrow (\text{leftchild}(x, y) \vee \exists z.\ (z \in Y \wedge \text{rightchild}(z, y)))$$

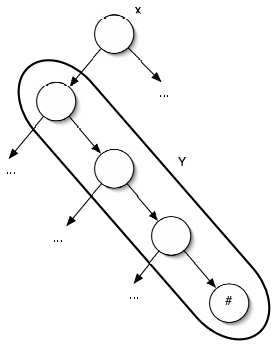



Figure 13.1: xchildren$(x, Y)$

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. By using xchildren, we can easily define xchild as follows.

$$\text{xchild}(x, y) \equiv \exists Y.\ (\text{xchildren}(x, Y) \wedge y \in Y)$$

We can use xchild for, e.g., collecting all $b$ nodes $y$ appearing as a child of an $a$ node (in XML view).

$$\exists x.\ \text{xchild}(x, y) \wedge a(x) \wedge b(y)$$

(Note here that "relation definitions" are not a part of our logical framework but simply an informal macro notation, e.g., the use of xchild$(x, y)$ above should be 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.)

**13.2.1 Exercise:** In the definition of xchildren, if we replace the $\Leftrightarrow$ with a $\Leftarrow$, then how does the meaning change? How about $\Rightarrow$ instead of $\Leftarrow$?

**13.2.2 Exercise:** In the same fashion as xchild, define xdescendent standing for the self-or-descendent relation in XML view. Also, define the self-or-descendent relation $\leq$ and the document order $\preceq$ that were primitives in 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 on 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 refers 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 where

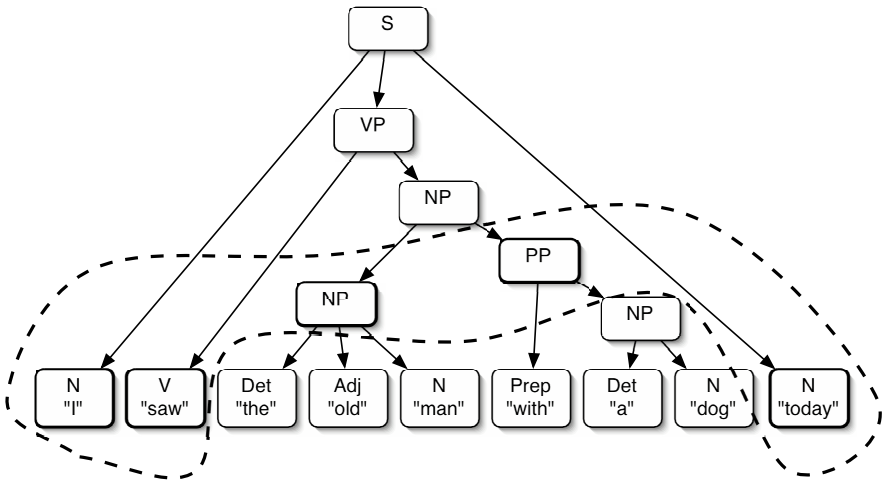



Figure 13.2: A proper analysis

each node not in the set is either an ancestor or a descendent of a node from this set and *vice versa*. Let us define such "follow in a proper analysis" relation in MSO. First, abbreviate:

$$x//y \quad \equiv \quad \text{xdescendents}(x, y) \wedge \neg(x = y)$$

Then, the notion of proper analysis can be expressed by

$$\text{proper\_analysis}(A) \equiv \forall x.\ \neg(x \in A) \Leftrightarrow \exists y.\ x \in A \wedge (x//y \vee y//x)$$

and the "follow" relation by:

$$\text{follow}(x, y) \equiv \exists A.\ \text{proper\_analysis}(A) \wedge y \in A \wedge y \in A \wedge x \preceq y$$

Observe that these *literally* translate the informal definitions given above by logical formulas.

## 13.3 Regularity

As already mentioned in the beginning, MSO expresses exactly regular queries, i.e., is equivalent to marking tree automata. This section aims at proving this equivalence. We will show below a concrete construction of a marking tree automaton from an MSO formula. However, the opposite direction will be left as an exercise for the reader.

### 13.3.1 Canonicalization

For simplifying 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. Each first-order variable used in the original formula is replaced with a second-order one and each first-order quantification or primitive relation is modified so as to preserve the meaning.

Formally, we define the syntax of canonical (MSO) formulas by the following.

$$\begin{array}{rcl} \psi_c & ::= & X \subseteq Y \mid R_u(X) \mid R_b(X,Y) \mid \mathbf{single}(X) \\ & & \mid \neg\psi_c \mid \psi_c \wedge \psi'_c \mid \exists X.\ \psi_c \end{array}$$

Here, we do not have disjunction ($\vee$), implication ($\Rightarrow$), and second-order universal quantification ($\forall X$), which will be represented, as usual, by combinations of conjunction, negation, and second-order existential quantification. We also do not have set membership ($x \in Y$), primitive relations on first-order variables ($R_u(x)$ and $R_b(x,y)$), and first-order quantification ($\forall x$ and $\exists x$). For encoding these, we first introduce a second-order variable $X_x$ for each first-order one $x$. 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 that $x$ refers to, we replace each use of $x$ by $X_x$. Namely, a set membership $x \in Y$ is replaced with a set inclusion $X_x \subseteq Y$ and primitive relations $R_u(x)$ and $R_b(x,y)$ with $R_u(X_x)$ and $R_b(X_x, X_y)$. Here, we extend the semantics of $R_u$ and $R_b$ to work with second-order variables in the following way: assuming $\Pi = \{\pi\}$, $\Pi_1 = \{\pi_1\}$, and $\Pi_2 = \{\pi_2\}$,

$$\begin{array}{rcl} R_u(\Pi) & \stackrel{\text{def}}{\Longleftrightarrow} & R_u(\pi) \\ R_b(\Pi_1, \Pi_2) & \stackrel{\text{def}}{\Longleftrightarrow} & R_b(\pi_1, \pi_2). \end{array}$$

Note that the semantics of $R_u$ and $R_b$ are undefined when $\Pi$, $\Pi_1$, or $\Pi_2$ is not a singleton. This fact will be crucial in automata construction below.

Now, the canonicalization $\mathbf{canon}(\psi)$ of a formula is defined as follows.

$$\begin{array}{rcl} \mathbf{canon}(x \in Y) & = & X_x \subseteq Y \\ \mathbf{canon}(R_u(x)) & = & R_u(X_x) \\ \mathbf{canon}(R_b(x,y)) & = & R_b(X_x, X_y) \\ \mathbf{canon}(\psi_1 \vee \psi_2) & = & \neg(\neg\mathbf{canon}(\psi_1) \wedge \neg\mathbf{canon}(\psi_2)) \\ \mathbf{canon}(\psi_1 \Rightarrow \psi_2) & = & \neg(\mathbf{canon}(\psi_1) \wedge \neg\mathbf{canon}(\psi_2)) \\ \mathbf{canon}(\psi_1 \wedge \psi_2) & = & \mathbf{canon}(\psi_1) \wedge \mathbf{canon}(\psi_2) \\ \mathbf{canon}(\exists X.\ \psi) & = & \exists X.\ \mathbf{canon}(\psi) \\ \mathbf{canon}(\forall X.\ \psi) & = & \neg\exists X.\ \neg\mathbf{canon}(\psi) \\ \mathbf{canon}(\exists x.\ \psi) & = & \exists X_x.\ \mathbf{single}(X_x) \wedge \mathbf{canon}(\psi) \\ \mathbf{canon}(\forall x.\ \psi) & = & \neg\exists X_x.\ \mathbf{single}(X_x) \wedge \neg\mathbf{canon}(\psi) \end{array}$$

Here, the translation of a first-order universal quantification can be understood that we first transform $\forall x.\ \psi$ to $\neg\exists x.\ \neg\psi$ and then to $\neg\exists X_x.\ \mathbf{single}(X_x) \wedge$

$\neg$**canon**($\psi$) (applying the translation of a first-order existential). Note that it is incorrect to swap the order, i.e., first transform $\forall x.\ \psi$ to $\forall X_x.$ **single**$(X_x) \wedge$ **canon**($\psi$) and then to $\neg\exists X_x.\ \neg$**single**$(X_x) \vee \neg$**canon**($\psi$), since the internal existential in the result would always hold as a non-singleton set $X_x$ always exists.

A formal correctness of canonicalization can be stated as a given formula $\psi$ holds under first- and second-order assignments $\gamma$ and $\Gamma$ if and only if the canonical formula **canon**($\psi$) holds under the canonicalization of $\gamma$. The canonicalization of a first-order assignment is defined as follows.

$$\textbf{canon}(\{x_1 \mapsto \pi_1, \ldots, x_n \mapsto \pi_n\}) = \{X_{x_1} \mapsto \{\pi_1\}, \ldots, X_{x_n} \mapsto \{\pi_n\}\}$$

**13.3.1 Lemma:** $t, \gamma, \Gamma \vdash \psi$ iff $t, \emptyset, \Gamma \cup$ **canon**$(\gamma) \vdash$ **canon**$(\psi)$.

PROOF: By induction on the structure of $\psi$. □

**13.3.2 Exercise:** Finish the proof of Lemma 13.3.1.

Note that the above lemma does not tell whether or not **canon**($\psi$) holds when the variables converted from first-order variables are assigned to non-singleton sets. In order to ensure that the canonical formula does not hold in such a case, we add the singleton constraints on the converted variables, i.e.,

$$\textbf{canon}(\psi_0) \wedge \bigwedge_{x \in \mathbf{FV}_1(\psi_0)} \textbf{single}(X_x)$$

where $\psi_0$ is the original formula given to the whole translation ($\mathbf{FV}_1(\psi_0)$ is the set of first-order free variables appearing in $\psi_0$).

### 13.3.2 Automata Construction

Having a canonical formula $\psi_c$ in hand, we remain to construct a marking tree automaton $\mathcal{M}$ such that

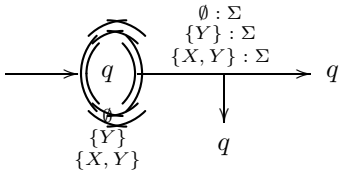
$t, \Gamma, \emptyset \vdash \psi_c$ holds if and only if $\mathcal{M}$ accepts $t$ with marking $\Gamma$.

(Note that a mapping from variables to node sets can be viewed as 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:** $\psi_c = X \subseteq Y$

This formula is translated to:



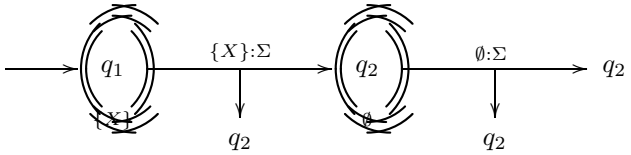
This depicts a marking tree automaton with a single state $q$ and three kinds of transitions, namely,

$$q \to \emptyset : a(q,q) \quad q \to \{Y\} : a(q,q) \quad q \to \{X,Y\} : a(q,q)$$

for every label $a \in \Sigma$. (Note that the  and the $q$'s that the arrows are led to are the same state; for a better readability, we avoid letting each arrow go around.) The state $q$ is initial and is final with a variable set either $\emptyset$, $\{Y\}$, or $\{X,Y\}$ (i.e., all $(q,\emptyset)$, $(q,\{Y\})$, 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 is a subset of the set of $Y$-marked nodes.

**Case:** $\psi_c = \text{root}(X)$

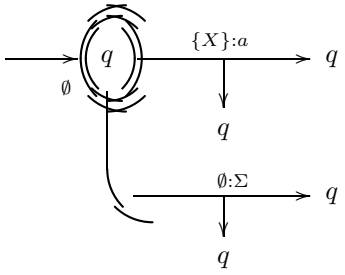
This formula is translated to:




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 marking $X$ on the root node (no matter whether it is a leaf or an intermediate node).

**Case:** $\psi_c = a(X)$

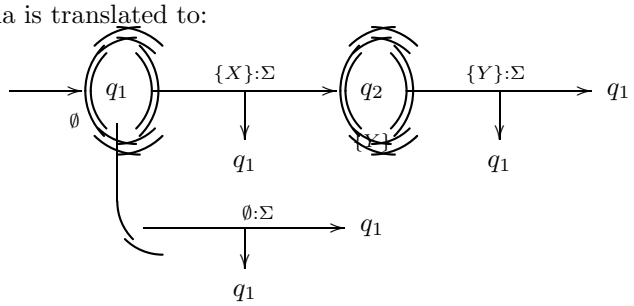
This formula is translated to:




This automaton consists of a single state $q$ with two transitions. The state accepts 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 the node must be labeled $a$, as desired by the semantics of $a(X)$. If there are no node or more than one node marked with $X$, then the tree may or may not be accepted, but this does not matter (recall that the semantics of $a(X)$ is undefined in such a case).

**Case:** rightchild$(X, Y)$

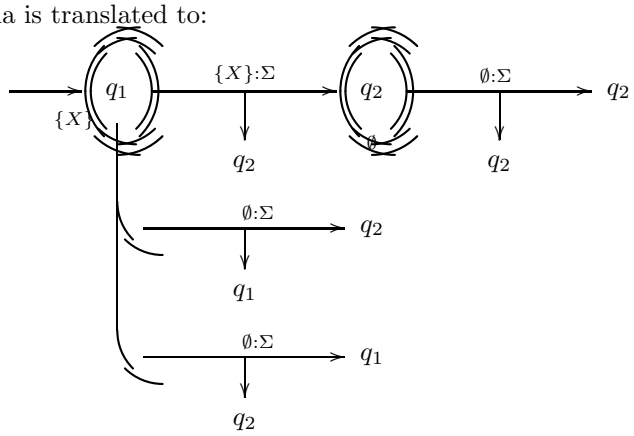
This formula is translated to:




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.

**Case:** $\psi_c = \mathbf{single}(X)$

This formula is translated to:




This 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 either 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, 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:** $\psi_c = \neg\psi'_c$

To translate this formula, we first compute a marking tree automaton from the subformula $\psi'_c$. Then, notice that a marking tree automaton can be regarded as an ordinary tree automaton whose each label is a pair of a label from $\Sigma$ and a set of variables (thus, the automaton accepts a binary tree whose each node is already marked with a set of variables). Then, we take the complement of this tree automaton. Note, however, that such an automaton requires multiple leaf symbols in general (due to 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:** $\psi_c = \psi'_c \wedge \psi''_c$

Similarly to the previous case, we first compute marking tree automata from the subformulas $\psi'_c$ and $\psi''_c$. However, we cannot simply take their intersection since the sets of variables used in the two marking automata can be different. To adjust them, we augment each transition in one automaton so as to allow arbitrary extra variables that are used only in the other automaton. That is, suppose that the two marking automata, called $\mathcal{M}_1$ and $\mathcal{M}_2$, allow the sets $\Xi_1$ and $\Xi_2$ of variables. Then, whenever there is a transition in the automaton $\mathcal{M}_1$

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

we add the following transition as well

$$q \to (\Xi \cup \Xi') : a(q_1, q_2)$$

for any $\Xi'$ such that $\emptyset \not\subseteq \Xi' \subseteq (\Xi_2 \setminus \Xi_1)$ (any variables that are only in $\Xi_2$). The other automaton is adjusted similarly. After this operation, regarding the adjusted marking automata as ordinary tree automata just as before, we take their intersection.

**Case:** $\psi_c = \exists X.\ \psi'_c$

To translate this formula, we first compute a marking tree automaton for $\psi'_c$ as before and then simply remove the variable $X$ from any transition, i.e., each transition

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

is replaced with the following.

$$q \to (\Xi \setminus \{X\}) : a(q_1, q_2)$$

This operation can be understood by noticing that $\psi'_c$ holds for a tree $t$ under an assignment $\Gamma$ if and only if $\psi_c$ holds for the same tree $t$ under the assignment $\Gamma$ *minus* the mapping for the variable $X$.

Note that, if the originally given formula contains only first-order variables, then the resulting marking tree automaton is linear, i.e., yields only a marking that puts each variable on exactly one node (Section 5.1). This fact enables us to use the efficient evaluation algorithm for marking tree automata presented in Section 7.2.

**13.3.3 Exercise:** A marking tree automaton can be translated to an MSO formula by directly expressing the semantics of marking tree automata in MSO. Show a concrete procedure.

## 13.4 Bibliographic Notes

The first attempt to bring the MSO logic to practical XML transformation has been made in the work on the *MTran* language [52], though a theoretical connection of MSO to tree transformation has already been made in the framework of MSO-definable tree transducers [24]. Compilation from MSO to marking tree automata has first been given in [85]. The MONA system [43] provides an efficient implementation of MSO based on binary decision diagrams and various other techniques. The MSO compilation presented in this chapter was taken from MONA manual [60] with a slight modification. Some other work proposes XML processing based on other kinds of logic, such as Ambient Logic [18], Sheaves Logic [90], and Context Logic [17].

# Chapter 14

# Unorderedness

In all the previous chapters, our discussions on the data model, schemas, automata, transformations, typechecking, and logic focus on the element structure of XML, where ordering is significant. However, reality of XML requires also unordered structure to be treated. This arises from two directions, namely, XML attributes and shuffle expressions. Attributes are auxiliary data that are associated to elements and are intrinsically unordered by data model, whereas shuffle expressions introduce unorderedness into the ordered data model of elements. Unorderedness is relatively less studied since dealing with it is surprisingly tricky. In this chapter, we briefly review previous proposals for description mechanisms and relevant algorithmics.

## 14.1 Attributes

As briefly mentioned in Section 2.1, attributes are a label-string mapping associated with each element and used in almost all applications of XML. Recall the following example there:

```
<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 in the same element. Thus, the above example is considered to be identical to:

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

Also, 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, the most obvious necessity is to associate each attribute label to its value type. For this, we can introduce a notation like in the following

```
person[@nationality[String], @age[String], ...]
```

where we list label-type pairs in front of the content type of `person`. (We would 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 we do not go further in this direction.) Then, we would like to express more "structural" constraints as follows:

- *optionality*. For example, we may want an attribute `nationality` to always be present but an `age` to be possibly omitted.
- *choice of labels*. For example, we may 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.
- *openness*, that is, allowance of an arbitrary number of arbitrary attributes from a given range of labels. For example, we may want to allow any attributes other than `age` or `nationality`. In real applications, we often want to allow arbitrary attributes not in a certain name space (name spaces are briefly introduced in Section 2.1).

How can we describe these constraints? For the first two, we are actually already able to express them thanks to union types. For example, "a mandatory `nationality` and an optional `age`" can be described as

```
  person[@nationality[String], @age[String], ...]
| person[@age[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 moment, we notice that repeating the element label `person` is rather awkward. Moreover, if both a `nationality` and an `age` are optional, then we need to list up four combinations; if there are $k$ optional attributes, then there are $2^k$ combinations. For more concise description, 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 them. First, we introduce expressions that each denote a set of labels. We write `~` for all labels and `a` for a singleton set of label `a`; for label sets $L_1$ and $L_2$, we write $(L_1|L_2)$ as their union and $(L_1 \setminus L_2)$ as their difference. Then, we write $@L[T]^*$ for a type denoting an arbitrary number of arbitrary attributes from a label set $L$ with value type $T$. Thus, the above example used in the openness can be written:

```
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 notations for constraining attributes. In practice, it is occasionally required to further mix constraints on both attributes and elements, for example, the following:

- *attribute-dependent element types*. For example, we may want an element `person` to contain a sub-element `visa-id` if and only if the `person` has an attribute `nationality` with a value other than `japanese`.
- *choice of attribute or element*. For example, we may want each of a `nationality` and an `age` to be given as either an attribute or an element.

For attribute-dependent element types, we can express them by using union types, e.g.,

```
  person[@foreigner["japanese"], ...]
| person[@foreigner[^"japanese"], visa-id[String], ...]
```

(Here, assume that the type `^string` denotes the set of strings except the `string`.) On the other hand, we do not have yet a way to write a choice of an attribute or an element without a potential blow-up. For example, if we want to grant such choices to both a `nationality` and an `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:

```
person[(@nationality[String] | nationality[String]),
       (@age[String] | age[String]), ...]
```

Note that each expression constrains a *pair* of an attribute set and an element sequence.

**14.1.1 Exercise:** Formalize the syntax and semantics of attribute-element constraints. Use the syntactic restriction where, when two types $T_1$ and $T_2$ are concatenated, these contain no common attribute label. In this semantics, in what cases is concatenation commutative?

## 14.2 Shuffle Expressions

Another kind of unorderedness comes from the desire to consider some ordering among elements to be 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 from the first place and equip a completely different, non-regular-expression-based schema language. However, reality is more complicated. For one thing, some people want some parts of a document to be ordered but other parts of the same document to be unordered. For example, in a document representing a "stream of records," the top-most sequence would be ordered and each sequence in the next-level would be unordered. For another thing, some people want a single schema language for all purposes, rather than different schema languages for different purposes. *Shuffle expressions* have arisen as a compromise, by which we can allow any permutation of elements for specified parts of document. Note here that the data model is still ordered, but an intended meaning can be unordered.

Let us first introduce notations. We add shuffle types of the form $T_1 \&\& T_2$ in the syntax of types. Its meaning is the set of all "interleaves" of values from $T_1$ and ones from $T_2$. More formally, we add the following rule in the semantics of types:

$$\frac{E \vdash v_1 \in T_1 \qquad E \vdash v_2 \in T_2 \qquad v \in \mathbf{interleave}(v_1, v_2)}{E \vdash v \in T_1 \&\& T_2}\ \text{T-Shu}$$

Here, $\mathbf{interleave}(v_1, v_2)$ is the set of values $v$ where $v$ contains $v_1$ as a (possibly non-consecutive) sub-sequence and taking away $v_1$ from $v$ yields $v_2$. For example, $\mathbf{interleave}((a[], b[]),\ (c[], d[]))$ contains all the following values:

$$\begin{array}{lll} a[], b[], c[], d[] & a[], c[], b[], d[] & a[], c[], d[], b[] \\ c[], a[], b[], d[] & c[], a[], d[], b[] & c[], d[], a[], b[] \end{array}$$

Note that the operator && is commutative. As a concrete example, we can express, by using shuffle types, a list of `record`s each of which has a mandatory `name`, an optional `address`, and any number of `email`s, in any order:

```
List    = list[Record*]
Record  = record[Name && Address? && Email*]
Name    = name[first[String], last[String]]
Address = address[String]
Email   = email[String]
```

In this example, the top `list` element contains an ordered sequence of `record`s, each of which contains an unordered list of fields, of which a `name` contains an ordered pair. It is important that shuffle expressions by themselves do not increase the expressive power of the schema model, but only allow an exponentially more concise description. For example, the type `a[] && b[] && c[]` can be rewritten to 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[]
```

with a blow-up.

**14.2.1 Exercise:** Rewrite the type `Name && Address? && Email*` to a non-shuffle type. Find a general procedure for eliminating shuffle expressions.

We have seen that shuffle expressions are quite powerful. However, their full power would not necessarily be useful, while allowing it might make algorithmics difficult. Therefore some schema languages adopt shuffle expressions with certain syntactic restrictions. The most typical one 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. On the other hand, a type like `(Name,Tel?) && (Name,Email*)` is an error. This restriction can make algorithmics easier to some extent since a certain kind of nondeterminism can be avoided in shuffle automata (described later). The RELAX NG schema language takes the restriction.

Another restriction that is often used in conjunction with the last 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 one. 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 `email`s. Nevertheless, this restriction can represent most usual record-like data that simply bundle a set of fields of different names (that are possibly optional). Moreover, algorithmics can be made easier since a sequence described by a shuffle type can be taken simply as a set (rather than a complicated interleave of sequences). Note that, with the further restriction that disallows concatenation of a shuffle type to some other type or repetition of a shuffle type, we can clearly separate unordered and ordered sequences with no mixture. XML Schema adopts this restriction.

## 14.3 Algorithmic techniques

As we have seen, both attribute-element constraints and shuffle expressions can be converted to more elementary forms of types. Thereby, relevant algorithmic problems, such as membership test and basic set operations, could be solved by using existing algorithms (possibly with a small modification). However, this easily causes a combinatorial blow-up. Although some efforts have been made to deal with this difficulty, none is completely satisfactory. Nevertheless, discussing some of them are worthwhile since they are at least good starting points.

**Attribute-element automata** A straightforward way to represent an attribute-element constraint is to use a tree automaton with two kinds of transitions: *element transitions* (which are usual ones) and *attribute transitions*. Such an automaton accepts a pair of an element sequence and an attribute set: when it follows an element transition, it consumes an element from the head of the sequence and, 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 7.1) for attribute-element automata. However, it is rather tricky to perform basic set operations such intersection and complement. It is 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 special cases that often 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 analysis can be done separately on these sub-automata. For example, suppose that two automata $A$ and $B$ are given and that each can be divided as a concatenation of sub-automata, say $A_1$ and $A_2$ from $A$ and $B_1$ and $B_2$ from $B$, such that both $A_1$ and $B_1$ have label set $L_1$ and both $A_2$ and $B_2$ have label set $L_2$. Then, intersection of $A$ and $B$ can be computed by taking that of the $A_1$ and $B_1$, taking that of $A_2$ and $B_2$, and then re-concatenating the resulting automata. This technique can also be applied to computing a complement.

**Shuffle automata** A well-known automata-encoding of a shuffle expression $T_1 \&\& T_2$ is to equip two automata $A_1$ and $A_2$ corresponding to $T_1$ and $T_2$ and run them in an interleaving way. That is, in each step of reading the input sequence, either $A_1$ or $A_2$ reads the head label; at the end of the sequence, both $A_1$ and $A_2$ must be in final states. (A similar idea is used in product construction, which also runs two automata in parallel but, in this case, both of two automata read the head label in each step.) Note that, by the "disjoint label sets" restriction on shuffle expressions, we can uniquely decide which of $A_1$ or $A_2$ reads each head label. With the "at most length one" restriction, a shuffle automaton becomes trivial—just a set of occurrence flags. Further, with the disallowance of concatenation or repetition of a shuffle, we can avoid

mixing ordinary and shuffle automata since the behavior of such mixture would extremely be complicated.

## 14.4 Bibliographic Notes

Whether attributes are important or not has been debated for a long time and has never settled [78]. Meanwhile, most applications extensively use attributes and the situation seems to be impossible to be reverted.

The specification of RELAX NG formalizes both attribute-element constraints and shuffle expressions [22]; XML schema provides its version of shuffles [33]. Although XML's DTD does not have shuffles, SGML's DTD [53] provides a weak notion of shuffle operator `&`, where `(a,b)&(c,d)` means just the union of `(a,b,c,d)` and `(c,d,a,b)` with no other interleaving. Reasoning of this operator is usually rather unclean.

Algorithmic sides of attribute-element constraints have been studied in several papers. A membership (validation) algorithm based on attribute-element automata is given in [73]; a similar algorithm has been tested in the context of bidirectional XML transformation [58]; another algorithm based on "derivatives" of regular expressions [15] can be found in [20] . Algorithms for basic set operations based on divide-and-conquer are in [48], though these algorithms directly operate on constraint expressions rather than automata for the purpose of presentation.

Shuffle automata have long been known and are formalized in, e.g., [54], though these have rarely been used in the context of XML. The above-mentioned derivative-based validation algorithm treats also shuffle expressions [20].

A completely different approach to dealing with unorderedness is to use Presburger's arithmetics, which can be used for describing constraints on the *counts* of element labels [90]. Such counting constraints have their own source of exponential blow-up, though an attempt has been made for a practical implementation [36].

# Bibliography


[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] A. V. Aho and J. D. Ullman. Translation on a context-free grammar. *Information and Control*, 19(5):439–475, 1971.

[3] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, 2001.

[4] M. Benedikt and C. Koch. XPath leashed. *ACM Computing Surveys*.

[5] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 51–63, 2003.

[6] A. Berlea and H. Seidl. Binary queries. In *Extreme Markup Languages*, 2002.

[7] M. Bojańczyk and T. Colcombet. Tree-walking automata cannot be determinized. In *ICALP*, pages 246–256, 2004.

[8] M. Bojańczyk and T. Colcombet. Tree-walking automata do not recognize all regular languages. In *STOC*, pages 234–243, 2005.

[9] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20:149–153, 1971.

[10] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML™), 2000. `http://www.w3.org/XML/`.

[11] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120:197–213, 1993.

[12] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[13] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 140(2):229–253, 1998.

[14] A. Brüggemann-Klein and D. Wood. Caterpillars: A context specification technique. *Markup Languages*, 2(1):81–106, 2000.

[15] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, Oct. 1964.

[16] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

[17] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–282, 2005.

[18] L. Cardelli and G. Ghelli. A query language for semistructured data based on the Ambient Logic. In *Proceedings of 10th European Symposium on Programming*, number 2028 in LNCS, pages 1–22, 2001.

[19] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Static analysis for dynamic xml. In *PLAN-X: Programming Language Technologies for XML*, 2002.

[20] J. Clark, 2002. `http://www.thaiopensource.com/relaxng/implement.html`.

[21] J. Clark and S. DeRose. XML path language (XPath). `http://www.w3.org/TR/xpath`, 1999.

[22] J. Clark and M. Murata. RELAX NG, 2001. `http://www.relaxng.org`.

[23] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Draft book; available electronically on `http://www.grappa.univ-lille3.fr/tata`, 1999.

[24] B. Courcelle. Monadic second-order definable graph transductions: A survey. *Theoretical Computer Science*, 126(1):53–75, 1994.

[25] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML, 1998. `http://www.w3.org/TR/NOTE-xml-ql`.

[26] J. Engelfriet. Bottom-up and top-down tree transformations - a comparison. *Mathematical Systems Theory*, 9(3):198–231, 1975.

[27] J. Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10:289–303, 1977.

[28] J. Engelfriet. Context-free graph grammars. In *Handbook of Formal Languages*, pages 125–213. Springer-Verlag, 1997.

[29] J. Engelfriet and S. Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Informatica*, 39(9):613–698, 2003.

[30] J. Engelfriet and E. M. Schmidt. Io and oi. ii. *Journal of Computer and System Sciences*, 16(1):67–99, 1978.

[31] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31(1):710–146, 1985.

[32] J. Engelfriet and H. Vogler. High level tree transducers and iterated pushdown tree transducers. *Acta Informatica*, 26(1/2):131–192, 1988.

[33] D. C. Fallside. XML Schema Part 0: Primer, W3C Recommendation, 2001. `http://www.w3.org/TR/xmlschema-0/`.

[34] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 Formal Semantics, 2001. `http://www.w3.org/TR/query-semantics/`.

[35] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *Journal of ACM*, 49(6):716–752, 2002.

[36] J. N. Foster, B. C. Pierce, and A. Schmitt. A logic your typechecker can count on: Unordered tree types in practice. In *ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X), Nice, France*, pages 80–90, Jan. 2007.

[37] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. PhD thesis, Universit Paris 7, 2004.

[38] A. Frisch. Ocaml + xduce. In *International Conference on Functional Programming (ICFP)*, pages 192–200, 2006.

[39] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science*, pages 137–146, 2002.

[40] A. Frisch and H. Hosoya. Towards practial typechecking for macro tree transducers. Technical report, INRIA, 2007.

[41] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.

[42] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: Facilitating XML processing in Java. In *14th International Conference on World Wide Web (WWW2005)*, pages 278–287. ACM Press, May 2005.

[43] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019, pages 89–110. Springer, 1995.

[44] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[45] H. Hosoya. Regular expression pattern matching — a simpler design. Technical Report 1397, RIMS, Kyoto University, 2003.

[46] H. Hosoya. Regular expression filters for XML. In *Programming Languages Technologies for XML (PLAN-X)*, pages 13–27, 2004.

[47] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 50–62, 2005.

[48] H. Hosoya and M. Murata. Boolean operations and inclusion test for attribute-element constraints. In *Eighth International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 201–212. Springer-Verlag, 2003.

[49] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(6):961–1004, 2002. Short version appeared in Proceedings of The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 67–80, 2001.

[50] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003. Short version appeared in Proceedings of Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science, pp. 226–244, Springer-Verlag.

[51] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2004. Short version appeared in Proceedings of the International Conference on Functional Programming (ICFP), pp.11-22, 2000.

[52] K. Inaba and H. Hosoya. XML transformation language based on monadic second-order logic. In *Programming Language Technologies for XML (PLAN-X)*, pages 49–60, 2007.

[53] International Organization for Standardization. *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, 1986.

[54] J. Jedrzejowicz. Structural properties of shuffle automata. *Grammars*, 2(1):35–51, 1999.

[55] T. Kamimura and G. Slutzki. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51, 1981.

[56] K. Kawaguchi. Ambiguity detection of RELAX grammars. `http://www.kohsuke.org/relaxng/ambiguity/AmbiguousGrammarDetection.pdf`, 2001.

[57] S. Kawanaka and H. Hosoya. bixid: A bidirectional transformation language for XML. In *International Conference on Functional Programming (ICFP)*, pages 201–214, 2006.

[58] H. Kido. Acceleration of biXid using attribute-element automata, 2007. Bachelor Thesis, The University of Tokyo.

[59] C. Kirkegaard and A. Møller. Xact - xml transformations in java. In *Programming Language Technologies for XML (PLAN-X)*, page 87, 2006.

[60] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA 1.4. `http://www.brics.dk/mona/`, 1995.

[61] C. Koch. Efficient processing of expressive node-selecting queries on xml data in secondary storage: A tree automata-based approach. In *VLDB*, pages 249–260, 2003.

[62] M. Y. Levin. Compiling regular patterns. In *International Conference on Functional Programming (ICFP)*, pages 65–77, 2003.

[63] M. Y. Levin and B. C. Pierce. Type-based optimization for regular patterns. In *Database Programming Languages (DBPL)*, Aug. 2005.

[64] K. Z. M. Lu and M. Sulzmann. XHaskell: Regular expression types for Haskell. Manuscript, 2004.

[65] S. Maneth, T. Perst, A. Berlea, and H. Seidl. XML type checking with macro tree transducers. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 283–294, 2005.

[66] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *International Conference on Database Theory (ICDT)*, pages 254–268, 2007.

[67] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proceedings of Symposium on Principles of Database Systems*, pages 215–226, Philadelphia, May 1999.

[68] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, May 2000.

[69] A. Møller, M. Ø. Olesen, and M. I. Schwartzbach. Static validation of XSL Transformations. Technical Report RS-05-32, BRICS, October 2005. Draft, accepted for TOPLAS.

[70] M. Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing '96*, volume 1293 of *Lecture Notes in Computer Science*, pages 153–169. Springer-Verlag, 1997.

[71] M. Murata. Extended path expressions for XML. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 126–137, 2001.

[72] M. Murata. RELAX (REgular LAnguage description for XML), 2001. `http://www.xml.gr.jp/relax/`.

[73] M. Murata and H. Hosoya. Validation algorithm for attribute-element constraints of RELAX NG. In *Extreme Markup Languages*, 2003.

[74] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.

[75] A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In *18th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *LNCS*, pages 134–145, 1998.

[76] F. Neven and J. V. den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 11–17, 1998.

[77] F. Neven and T. Schwentick. Query automata over finite trees. *Theoretical Computer Science*, 275(1-2):633–674, 2002.

[78] OASIS. SGML/XML elements versus attributes, 2002. `http://xml.coverpages.org/elementsAndAttrs.html`.

[79] Y. Papakonstantinou and V. Vianu. DTD Inference for Views of XML Data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 35–46, Dallas, Texas, May 2000.

[80] T. Perst and H. Seidl. Macro forest transducers. *Information Processing Letters*, 89(3):141–149, 2004.

[81] L. Planque, J. Niehren, J.-M. Talbot, and S. Tison. N-ary queries by tree automata. In *DBPL*, 2005.

[82] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990.

[83] G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41:305–318, 1985.

[84] T. Suda and H. Hosoya. Non-backtracking top-down algorithm for checking tree automata containment. In *Proceedings of Conference on Implementation and Applications of Automata (CIAA)*, pages 83–92, 2005.

[85] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. pages 2:57–82, 1968.

[86] A. Tozawa. Towards static type checking for XSLT. In *Proceedings of ACM Symposium on Document Engineering*, 2001.

[87] A. Tozawa. XML type checking using high-level tree transducer. In *Functional and Logic Programming (FLOPS)*, pages 81–96, 2006.

[88] A. Tozawa and M. Hagiya. XML schema containment checking based on semi-implicit techniques. In *8th International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 213–225. Springer-Verlag, 2003.

[89] S. Vansummeren. Type inference for unique pattern matching. *ACM Transactions on Programming Languages and Systems*, to appear.

[90] S. D. Zilio and D. Lugiez. Xml schema, tree logic and sheaves automata. In *International Conference on Rewriting Techniques and Applications*, volume 2706, pages 246–263. Springer-Verlag, 2003.