RUN, XTATIC, RUN: EFFICIENT IMPLEMENTATION OF AN OBJECT-ORIENTED LANGUAGE WITH REGULAR PATTERN MATCHING

Michael Y. Levin

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2005

Benjamin C. Pierce
Supervisor of Dissertation

Rajeev Alur
Graduate Group Chairperson

ABSTRACT

RUN, XTATIC, RUN: EFFICIENT IMPLEMENTATION OF AN OBJECT-ORIENTED LANGUAGE WITH REGULAR PATTERN MATCHING

Michael Y. Levin

Benjamin C. Pierce

Schema languages such as DTD, XML Schema, and Relax NG have been steadily growing in importance in the XML community. A schema language provides a mechanism for defining the *type* of XML documents; i.e., the set of constraints that specify the structure of XML documents that are acceptable as data for a certain programming task.

A number of recent language designs—many of them descended from the XDUCE language of Hosoya, Pierce, and Vouillon—have showed how such schemas can be used *statically* for type-checking XML processing code and *dynamically* for evaluation of XML structures. The technical foundation of such languages is the notion of *regular types*, a mild generalization of nondeterministic top-down tree automata, which correspond to a core of most popular schema notations, and the notion of *regular patterns*—regular types decorated with variable binders—a powerful and convenient primitive for dynamic inspection of XML values.

This dissertation is concerned with one of XDUCE's descendants, XTATIC. The goal of the XTATIC project is to bring the regular type and regular pattern technologies to a wide audience by integrating them with a mainstream object-oriented language. My research focuses on an efficient implementation of XTATIC including a compiler that generates fast and compact target programs and a run-time system that is designed to support efficient manipulation of XML fragments. Many techniques described here are applicable not only to XTATIC but also to other XDUCE derivatives such as CDUCE and Cω.

Contents

Chapter 1

Introduction

This dissertation describes implementation strategies for the new object-oriented language XTATIC. XTATIC is a combination of the general purpose object-oriented language C♯ with features for type-safe processing of XML data. The thesis defended here is that it is feasible to implement XTATIC efficiently. In this introduction, we describe XTATIC, enumerate the major points of its implementation, and provide a road map for the rest of the dissertation.

1.1 Background

Schema languages such as DTD [72], XML Schema [73], and Relax NG [7] have been steadily growing in importance in the XML community. A schema language provides a mechanism for defining a *type* of XML documents; i.e., the set of constraints that specify the structure of XML documents that are acceptable as data for a certain programming task. Until recently, schema languages have been used largely for dynamic verification of XML documents or as a specification language to communicate—informally—the prescribed format of XML data without any goal of automated enforcement.

Some years ago, Hosoya, Pierce, and Vouillon [34, 37, 35, 36, 30] designed the XML transformation language XDUCE, which showed how such schemas can be used to establish and ensure *static* properties of programs. The language centers around the notion of *regular types*, which classify sets of XML documents or, equivalently, sets of ordered sequences of unranked node-labeled trees annotated with unordered sets of attributes. Like the type systems of other statically typed languages, the type system of XDUCE ensures that the invariants specified in a program hold at run-time. Unlike the types of other languages, however, regular types can be used specifically to describe sets of values that correspond to sets of documents that can be defined in one of the mentioned schema

languages.

In addition to being used for specification of statically-checked constraints, regular types can form a basis for exploring values dynamically. XDuce employs *regular patterns*—regular types decorated with variable binders—in conjunction with the algebraic pattern matching construct `match` popularized by ML and Haskell to provide a powerful and convenient primitive for dynamic inspection of XML values.

XDuce made a big impact as one of the first statically typed XML processing languages and resulted in several descendants. One such descendant is the Xtatic language whose goal is to bring the regular type and regular pattern technologies to a wide audience by integrating them with a mainstream object-oriented language.

This dissertation focuses on an efficient implementation of Xtatic including a compiler that generates fast and compact low-level programs and a run-time system that is designed to support efficient manipulation of XML fragments.

1.2 Contributions

Functional languages such as ML and Haskell share many features with Xtatic. In particular, the pattern matching constructs found in these languages resemble those in Xtatic, and ideas of immutable values and lazy implementation of certain operations on them are also natural in Xtatic. Consequently, many techniques used in implementation of compilers and run-time systems of functional languages can be reused in an implementation of Xtatic.

A lot of research has been concerned with efficient implementation of object-oriented languages, answering questions of how to speed up invocation of methods and how to optimize the memory layout of objects. Our implementation takes full advantage of this research by providing a source-to-source compiler from Xtatic into C♯. As a result, we do not have to worry about these issues since they are handled by the C♯ compiler.

Despite these points, however, implementing Xtatic involves a substantial amount of original research. The primary distinguishing aspect of Xtatic is the relative complexity of its pattern matching mechanism in comparison to pattern matching mechanisms of other languages. Unlike algebraic patterns of functional languages, Xtatic's regular patterns obey various semantic equivalence rules that make syntax directed pattern matching and pattern compilation difficult. Recursive patterns and patterns with Kleene operators `*` and `+` require nuanced compilation approaches that carefully balance performance goals with code size considerations. Regular patterns with variable binding can be ambiguous and, therefore, difficult to implement in a deterministic

and easy-to-understand way.

Furthermore, the run-time system must meet several requirements including support for 1) fast and memory-efficient pattern matching operations on both element sequences and textual data and 2) seamless and safe integration of data in foreign formats such as Document Object Model (DOM).

In response to all of the above challenges, this dissertation proposes the following contributions.

- The key issue in implementing XTATIC is how to compile regular pattern matching efficiently and compactly. An XTATIC compiler must address not only the familiar problems of pattern optimization for ML-style pattern matching, but also some new ones, arising principally from the use of recursion in patterns. To talk about these issues and to present compilation algorithms rigorously, we introduce *matching automata*—a formalism for modeling low-level pattern matching constructs at a level of abstraction that allows us to elide the specifics of a particular target language. We use the framework of matching automata to define a compilation algorithm for pattern matching with regular patterns without variable binders. This contribution is described in Chapter 4.

- To enhance the performance of our pattern compiler, we extend the matching automaton framework with a semantic optimization technique in which we use the *schema* of the value flowing into a pattern matching expression to generate efficient target code. We present a practical, but not always optimal, type-based pattern compilation algorithm, define a formal optimality criterion of "no useless tests", and show that the problem of generating optimal pattern matching code is decidable for finite (non-recursive) patterns. This work is described in Chapter 6.

- We next proceed to give a formal treatment of compiling pattern matching that involves patterns with variable binders. A principal difficulty in this undertaking involves treatment of *ambiguous* patterns. In the presence of variable binding, it becomes possible to match a value against a pattern in multiple ways each producing a different variable binding outcome. We define an easy-to-understand disambiguation semantics that specifies a particular way of pattern matching and design a compilation algorithm that generates a deterministic matching automaton performing variable binding according to this semantics. This contribution is covered in Chapter 5.

- The design of the run-time system also presents a set of novel challenges. Chapter 7 describes how XTATIC supports fast and compact operations on element sequences and textual data and facilitates seamless and safe integration with data in foreign formats such as Document Object Model (DOM).

We establish the thesis of this dissertation by 1) showing that the basic matching automaton framework and the compilers based on it work reasonably well for regular patterns without binders, 2) enhancing the prototype framework with type-based optimization, support for regular patterns with binders, and efficient run-time representation, and 3) demonstrating—in Section 7.4—that the obtained full implementation of XTATIC is quite successful when compared with other XML processing systems.

There is more work to be done! The next step in the evolution of XTATIC concerns graduating from being a research project and moving toward being accepted in a wider community of programmers. For this to become reality, XTATIC must become a *practical* general-purpose programming language: it must be well-integrated with existing XML standards, in particular, its type system must be better aligned with XML Schema; it must support more expressive and high-level pattern matching constructs. XTATIC's implementation must become more robust as well: it must provide stream-based processing to support very large documents and indexing mechanisms to improve pattern matching performance; the existing algorithms must be fine tuned, especially to reduce the size of the target program. Chapter 8 presents the conclusions of this dissertation and discusses some future directions in more detail.

1.3 Writing Credits

The overview of XTATIC presented in Chapter 3 is based on the work leading to the current design of the language conducted by Gapeyev and Pierce [23]. Chapter 7 is derived from a paper written in collaboration with Gapeyev, Pierce, and Schmitt [20] based on my implementation of the XTATIC compiler. Chapters 4, 6, and 5 are based on papers [50, 52, 51] written in collaboration with my adviser Benjamin C. Pierce.

Chapter 2

Related Work

This chapter considers two categories of related work. First we describe the landscape of XML transformation languages and briefly address their implementation practices. We then give an overview of other research projects that are related in some ways to the implementation techniques or formalisms developed in this dissertation. Chapters 4, 6, 5, 7 provide more detailed discussion of related work.

2.1 Related Languages

XDuce [35, 36, 37] was the first language featuring XML trees as built-in values, a type system based on regular types for statically type-checking computations involving XML, and a powerful form of pattern matching based on regular patterns.

XDuce's regular types were developed to mirror most core features of popular XML schema languages such as DTD [72], XML Schema [73], and Relax NG [7]. Like the schema languages, XDuce has concatenation, union, repetition and option type constructors and top-level mutually recursive type declaration. At the core of XDuce type checking is a *subtyping* algorithm that tests whether one type is subsumed by another. XDuce defines a *semantic* subtyping relation—type `T` is a subtype of type `S` if values belonging to `T` are a subset of values belonging `S`. One of the achievements of the XDuce project was to show how such set-based subtyping can be implemented efficiently.

In addition to being used for specification of statically-checked constraints, regular types can also provide a mechanism for exploring values dynamically. A value `v` can be *matched* against a type `T` yielding true if and only if `v` conforms to `T`. XDuce defines a more advanced matching construct *patterns* as types extended with variable binders. Patterns can serve not only as boolean

predicates on values but also as a tool for value deconstruction.

XDuce gave rise to several descendants including Xtatic. The focus of the Xtatic project is on integrating XDuce's regular types and regular patterns with a modern object-oriented language. Different aspects of the Xtatic design and implementation are described in a series of papers—the first outlines the design choices we have made to facilitate smooth integration while keeping Xtatic easy to understand for the programmer [22]; the second presents the core language design, integrating the object and tree data models and establishing basic soundness results [23]; the third proposes a technique for compiling regular patterns based on *matching automata* [50], and the fourth describes the run-time system of Xtatic [21].

Another XDuce descendant that is close to Xtatic in several respects is the CDuce language of Benzaken, Castagna, and Frisch [19, 4]. Like Xtatic, CDuce is based on XDuce-style regular types and emphasizes a declarative style of recursive tree transformation based on algebraic pattern matching. In other respects, the focus in CDuce is quite different: its type system includes several features (such as intersection and function types) not present in Xtatic, it is not object-oriented, and it is not integrated with an existing language. Xtatic, by contrast, has taken a more conservative approach in its type system, instead emphasizing smooth compatibility with an existing mainstream object-oriented language. Two significant differences are the object-oriented flavor of our representations and our approach to various interoperability issues such as cross-language calls and compatibility with legacy XML representations.

Another close cousin of Xtatic is Meijer, Schulte, and Bierman's Cω language (previously called Xen) [54, 55], an extension of C$^\sharp$ that smoothly integrates support for objects, relations, and XML. Some aspects of the Cω language design are much more ambitious than Xtatic: in particular, the extensions to its type system (`sequence` and `choice` type constructors) are more tightly intertwined with the core object model—indeed, XML itself is simply a syntax for serialized object instances. In other respects, Cω is more conservative than Xtatic: for example, its `choice` constructor is not a true least upper bound, and the subtype relation is defined by a conventional, semantically incomplete, collection of inference rules, while Xtatic's is given by a more straightforward (and, for the implementation, more demanding) "subtype = subset" construction.

Xact [47, 6] extends Java with XML processing, proposing another somewhat different programming idiom: the creation of XML values is done using XML *templates*, which are immutable first-class structures representing XML with named *gaps* that may be filled to obtain ordinary XML trees. Xact also features a static type system guaranteeing that, at a given point in the program, a template statically satisfies a given DTD. Xact's implementation, developed independently and in parallel with Xtatic but driven by similar needs (supporting efficient sharing, etc.) and targeting

a similar (object-oriented) run-time environment, has strong similarities to ours; in particular, lazy data structures are used to support efficient gap plugging.

XJ [29] is another extension of JAVA for native XML processing that emphasizes fidelity to the XML Schema and XPATH standards, for instance by only allowing subtyping by name (as opposed to the structural subtyping of the languages mentioned above). XJ is also one of the few XML processing languages that allow imperative modification of XML data. This feature, however, significantly weakens the safety guarantees offered by static typing: the updated tree must be re-validated dynamically, raising an exception if its new type fails to match static expectations. In keeping with its emphasis on standards and its imperative nature, XJ uses DOM for its run-time representation of XML data.

XOBE [46] is a source to source compiler for an extension of JAVA. From a language design point of view, it is very similar to XTATIC, allowing seamless integration of XML with JAVA, taking a declarative style of tree processing, and providing a rich type system and subtyping relation based on regular expression types. The run-time representation, like XJ, relies on DOM.

SCALA is a general-purpose experimental web services language that compiles into JAVA byte-code and therefore may be seen as an extension of JAVA since SCALA programs may still easily interact with JAVA code. SCALA is currently being extended with XML support [11].

Work also continues on XDUCE itself, including fully typed treatment of attributes *à la* RELAXNG [33] and regular expression filters [31]. These developments are highly relevant to future work on the XTATIC language design.

A recent survey paper by Møller and Schwartzbach [56] offers an excellent overview of recent work on static typechecking for XML transformation languages, with detailed comparisons between a number of representative languages, including XDUCE and XACT.

XQUERY [74]—arguably the current gold standard in stand-alone XML processing languages—and XSLT [68] are special-purpose XML processing languages specified by W3C that have strong industrial support, including a variety of implementations and wide user base.

2.2 Other Related Work

Compilation of datatype-based pattern matching has been researched extensively in the field of functional programming [3, 2, 12, 62]. XTATIC's pattern matching constructs inherit many characteristics of datatype-based pattern matching constructs, and, therefore, we can reuse functional languages compilation algorithms in an implementation of XTATIC. Section 4.8 provides an overview of different datatype-based compilation approaches and discusses how they relate to our algorithm.

Logic programming languages also employ compilation and run-time representation approaches whose goals are similar to those of the approaches used in XTATIC. Among them are algorithms and data structures for fast sequence concatenation [64, 53] and pattern match optimization techniques focused on reducing the amount of backtracking and promoting sharing of common tests performed by different branches of conditional expressions [48, 10]. We discuss logic programming techniques in sections 4.8 and 7.5.

Procedure inlining [40, 1, 69] is another relevant research area. XTATIC's pattern compiler is similar to a procedure inliner in that it examines a cyclic structure—a recursive pattern—and segregates the nodes of that structure that can be compiled inline from the nodes that must be compiled into stand alone code fragments. See more on this in Section 4.8.

A great deal of research has been conducted in the area of tree automata and regular tree languages [8, 5, 58, 59]. This field studies properties of regular tree and forest languages and algorithms for various decision problems such as membership testing, minimization, and determinization. A key distinction of our matching automata framework is that its focus is on modeling low-level target language code, and, as a result, existing tree automata techniques may not be directly applicable XTATIC's implementation. Section 4.8 gives an overview of tree automata and regular tree languages literature and discusses its relationship to our work.

Two categories of research projects are related to XTATIC's type-based optimization. The first encompasses type-based optimization algorithms for XTATIC-like languages and is represented by Alain Frisch's work on CDUCE [16, 17]. Unlike our matching-automaton-based algorithms, his optimization algorithms are based on a more abstract notion of tree automata that is not suitable for direct code generation. The second category includes research on minimizing XPATH queries. Some approaches in this area develop algorithms that generate optimal queries for subsets of XPATH [71, 13]; others give heuristic, not fully optimal, algorithms that work for full XPATH [26, 14, 27, 28]. Section 6.6 addresses these research projects in more detail.

Section 7.5 outlines research relevant to the development of XTATIC's run-time system. It discusses the run-time systems of several related languages [4, 6, 44] and efficient algorithms for various operations on sequence data structures [57, 63, 45, 41, 42, 43, 60, 38, 70, 66, 64, 53].

Chapter 3

The XTATIC Language

This chapter describes XTATIC and illustrates it by some sample programs. Gapeyev and Pierce [23] give a more detailed specification of the language.

3.1 Language Overview

XTATIC is a lightweight extension of C♯ offering native support for statically typed XML processing. XML trees are built-in values in XTATIC, and static analysis of the trees created and manipulated by programs is part of the ordinary job of the typechecker. "Tree grep" pattern matching is used to investigate and transform XML trees.

XTATIC inherits its key features from XDUCE [35, 36, 37]. These features include XML trees as built-in values, a type system based on *regular types* (closely related to popular schema languages such as DTD and XML-Schema) for static typechecking of computations involving XML, and a powerful form of pattern matching called *regular patterns*.

The integration of XML trees with the object-oriented data model of C♯ happens on two levels. First, the XML type hierarchy is grafted into the C♯ class hierarchy by making all regular types be subtypes of a special class `Seq`. This allows XML trees to be passed to generic C♯ library facilities such as collection classes, stored in fields of objects, etc. Conversely, the roles of labels in element sequences and their types are played by objects and classes; ordinary XML sequences are represented using objects from a special `Tag` class as labels.

3.1.1 Values

XTATIC values consist of native C♯ values and (potentially empty) sequences of elements each

containing a C♯ value used as the tag and a nested child value:

$$v ::= h \mid () \mid (h_1)[v_1] \ldots (h_n)[v_n]$$

where h and h_i range over native C♯ values that can be either objects or values of primitive types such as integers and characters.

For instance, we can write `(1)[]` for the sequence of one element whose label is the integer `1` and whose contents is the empty sequence. (We omit parentheses when the empty sequence is nested within a parent element.) Assuming that $\texttt{Tag}_{\texttt{author}}$ is a subclass of `Tag` representing the XML tag `author`, the XTATIC value $\texttt{(newTag}_{\texttt{author}}\texttt{())[]/>}$ corresponds to an XML sequence containing one `author`-labeled element with an empty contents. For such sequences, XTATIC provides a lighter notation in which the parenthesis in the label are dropped and the `new` expression is replaced by the corresponding XML tag. For example, the above value can be written as `author[]`.

Textual data is encoded by sequences of character-labeled elements. For instance `‘abc‘` is a short-hand for the value `('a')[], ('b')[], ('c')[]`.

Consider the following document fragment—a sequence of two entries from an address book—given here side-by side in XML and XTATIC concrete syntax.

```
<person>                                  person[
  <name>Haruo Hosoya</name>                 name[‘Haruo Hosoya‘]
  <email>hahasoya</email>                   email[‘hahasoya‘]
</person>                                 ]
<person>                                  person[
  <name>Jerome Vouillon</name>              name[‘Jerome Vouillon‘]
  <tel>123</tel>                            tel[‘123‘]
</person>                                 ]
```

The structure of the XTATIC document mirrors the structure of the XML document, the only difference being a more compact notation for elements and backquotes, which distinguish XML textual data from arbitrary XTATIC expressions yielding XML elements.

3.1.2 Types

XTATIC has two kinds of types: native C♯ types such as classes and primitive types for classifying C♯ values and regular types for describing XML data. This section concentrates on regular types and omits a thorough discussion of native C♯ types. More details about formalizing object-oriented aspects of XTATIC-like languages can be found in the description of Featherweight Java [39] and the definition of XTATIC [23].

XTATIC types are described by the following grammar:

$$T \quad ::= \quad H \mid () \mid (H)[T] \mid T_1, T_2 \mid T_1 | T_2 \mid T* \mid Any \mid X$$

These denote native C♯ types, the type of the empty sequence, a labeled element type, sequential composition, union, repetition, wild-card, and a type variable. Type variables are introduced by top-level mutually recursive declarations of the form *def* $X = T$.

One possible type for the telephone book value shown in Section 3.1.1 is a list of persons, each containing a name, an optional phone number, and a list of emails:

```
person[name[pcdata], tel[pcdata]?, email[pcdata]*]
```

where "`?`" marks optional components—it is an abbreviation for a union with the empty sequence—and "`pcdata`" describes sequences of characters.

In the presence of the following type definitions, our address book could be given the type `APers*`:

```
def Name = name[pcdata]
def Tel = tel[pcdata]
def Email = email[pcdata]
def TPers = person[Name,Tel]
def APers = person[Name,Tel?,Email*]
```

Subtyping in XTATIC inherits XDUCE's "semantic" definition of subtyping for regular types. In semantic subtyping, type $\texttt{T}_1$ is a subtype of type $\texttt{T}_2$ if the set of values classified by $\texttt{T}_1$ is a subset of the set of values classified by $\texttt{T}_2$. For example, every value of type `TPers` can also be described by the type `APers`, so we have `TPers <: APers`.

XDUCE's simple semantic definition of subtyping extends naturally to XTATIC's object-labeled trees and classes. The subclass relation on labels is lifted to the subtype relation on regular types: `(A)[]` is a subtype of `(B)[]` if `A` is a subclass of `B`.

The combined data model and type system, dubbed *regular object types*, have been formalized in [23]. Algorithms for checking subtyping and inferring types for variables bound in patterns can be adapted straightforwardly from those of XDUCE [37, 35].

3.1.3 Patterns

Types and subtyping are also the foundation of *regular pattern matching*, which generalizes both the `switch` statement of C♯ and the algebraic pattern matching popularized by functional languages

such as ML. A *regular pattern* is just a regular type decorated with variable binders. A value `v` can be matched against a pattern `p`, binding variables occurring in `p` to the corresponding parts of `v`, if `v` belongs to the language denoted by the regular type obtained from `p` by stripping variable binders. For matching against multiple patterns, XTATIC provides a `match` construct that is similar to the `switch` statement of C♯ and the `match` expression of functional languages such as ML. For example, the following method extracts a sequence of type `TPers` from a sequence of type `APers`, removing persons that do not have a phone number and eliding emails.

```
fun addrbook(APers* ps) : TPers* =
  TPers* res = ();
  bool cont = true;
  while cont
    match ps with
      person[name[Any] n, tel[Any] t, Any], Any rest →
        res = res, person[n,t];
        ps = rest;
      person[Any], Any rest →
        ps = rest;
      () →
        cont = false;
  return res;
```

Regular patterns are described by the following grammar in which P and Q range over all patterns and host language patterns respectively:

$$\begin{aligned} P &::= Q \mid () \mid (H)[P] \mid P_1, P_2 \mid P_1 | P_2 \mid T* \mid Any \mid P\ x \\ Q &::= H \mid Q\ x \end{aligned}$$

Compared with the facilities available in pure C♯ (such as the raw DOM API), regular pattern matching allows much cleaner and more readable implementations of many tree investigation and transformation algorithms. However, compared with other native XML processing languages, XTATIC's pattern matching primitives are still fairly low-level: for example, no special syntax is provided for collecting *all* sub-trees matching a given pattern, or for iterating over sequences. We are currently investigating how best to add more powerful pattern matching; for now, our implementation efforts are concentrated on achieving good performance for low-level XML processing code.

Chapter 4

Foundations of Pattern Compilation

This chapter describes the fundamental issues arising in compilation of XTATIC programs. The focus is on compilation of pattern matching, since this is the main distinguishing aspect of XTATIC compared to both traditional object-oriented and traditional functional languages. We use a limited subset of XTATIC in this chapter—not only are most constructs unrelated to pattern matching omitted, but patterns themselves are restricted to those without term binders and attributes.

Compilation of datatype-based patterns has been addressed extensively in the literature (see Chapter 2). The main issue in translating these patterns is how to minimize the number of tests performed during pattern matching while keeping the size of the output code small. XTATIC regular patterns, being more expressive than datatype-based patterns, raise the same issues and add some new ones—in particular, the handling of recursion. Whereas the number of tests required to determine whether a given input value matches an datatype-based pattern is bounded by a function of the size of the *pattern*, matching against a recursively defined pattern may involve a number of tests depending on the size of the *value*. (For example, the recursive pattern `X` defined by `X = () | a[X],X` describes trees of arbitrary depth whose nodes are all labeled by `a`; checking that a given tree matches `X` involves exploring all of its nodes.) Since XML documents may be large, the designer of a regular pattern compiler must be particularly sensitive to the performance of the generated code with respect to the size of input values.

Algorithms for high-quality pattern compilation in this domain are somewhat complex, and we

have found it useful to spend significant effort on developing careful proofs of correctness. To simplify these proofs—as well as the presentation of the algorithms themselves—we separate compilation conceptually into four phases. The first phase converts source program patterns into a simpler, less ambiguous form corresponding to states of a non-deterministic, top-down tree automaton. The structure of tree automata, however, is both too rigid and too high-level to suggest a direct way of generating equivalent low-level code; we need a more flexible automata model to bridge the gap between the source and the target of the compiler. This leads us to introduce *matching automata*, which extend tree automata with variable binding, subroutines, and integer indices tracking which of a set of patterns match an input value. The second compilation phase converts collections of tree automaton states into equivalent matching automata. The remaining phases generate code: the third phase produces intermediate language procedures from matching automata obtained in the second phase; the fourth phase—not covered in this dissertation—converts intermediate language code into target code in pure C♯.

To motivate the developments of this chapter, we explore a series of examples and identify several issues that greatly influence the quality of low-level pattern-matching code. Based on these observations, we propose two compilation algorithms—one generating backtracking code, and the other a non-backtracking variant. To present these algorithms, we introduce matching automata—a model of intermediate language programs, allowing us to elide the specifics of the intermediate language and reason about properties of the compilation algorithms at a more abstract level.

The remainder of the chapter is organized as follows. Section 4.1 introduces a subset of XTATIC called XTATICLITE. Section 4.2 describes the intermediate language XIL that will be used as a back-end for both the backtracking and the non-backtracking compilers. Section 4.3 previews the two compilation approaches. Section 4.4 reviews the definition of binary top-down tree automata, introduces matching automata, defines two particular forms of matching automata, called "simple backtracking" and "simple non-backtracking", and analyzes examples highlighting a number of important issues arising in regular pattern compilation. Section 4.5 explains how to build matching automata in backtracking and non-backtracking forms. Section 4.7 sketches our implementation and discusses some performance experiments.

4.1 Source Language

Since this chapter addresses compilation of pattern matching, we consider a subset of XTATIC that is sufficient to describe the concepts presented here. The difference between XTATIC and the subset XTATICLITE can be summarized by the following restrictions:

v	::=	`()`	empty sequence
		`l[`v_1`]`...`l[`v_k`]`	non-empty sequence
p	::=	`()`	empty sequence pattern
		`l[p]`	single element sequence pattern
		p_1`,`p_2	concatenation pattern
		p_1`|`p_2	union pattern
		`X`	top-level definition variable pattern
t	::=	`x`	term variable
		`n`	integer
		`()`	empty sequence
		`l[t]`	single element sequence
		t_1`,`t_2	sequence concatenation
		`A(t)`	function call
		`match x with` $\overline{p \rightarrow t}$	pattern matching
d	::=	`fun A(x) = t`	function declaration
		`def X = p`	pattern declaration

Figure 4.1: XtaticLite syntax

- Object-oriented features are omitted—a program is a collection of mutually recursive top-level function definitions instead of a collection of classes
- XML values are tagged by uninterpreted labels as opposed to C♯ objects
- Types are omitted
- Variable binding is omitted from patterns

4.1.1 Definition: Sequence values, patterns, terms, and top-level definitions of XtaticLite are described by the grammar shown in Figure 4.1 (where `l,x,n,X`, and `A` range over labels, term variable names, integers, pattern variable names, and function names respectively.)

XtaticLite has two kinds of values: sequence values and integers, ranged over by `v` and `n` respectively. Integer values are only introduced for presentational convenience to indicate different outcomes of pattern matching in sample programs. Sequence values, on the other hand, can be used as pattern matching arguments. A sequence value is a sequence of elements, where each element has the form `l[v]`, with `l` a label and `v` a sequence of child elements. The empty sequence is written `()`, but as before we omit the parentheses if the empty sequence is delimited by a label, writing just `l[]` instead of `l[()]`. Element sequences can be used to represent attribute-less XML documents. For instance, the XML element `<a><b/><c/></a>` can be encoded by the value `a[b[],c[]]`.

XtaticLite supports the following kinds of patterns: the empty sequence pattern, a labeled element pattern, sequential composition and union of two patterns, and a pattern variable. Pattern variables are introduced by top-level mutually recursive declarations of the form `def X = p`. Top-level declarations induce a function `def` that maps variables to the associated patterns: `def(X)` = `p` if and only if the program contains the declaration `def X = p`.

$$\texttt{()} \in \texttt{()} \qquad \text{(LXP-Emp)}$$

$$\frac{\texttt{v} \in \texttt{p}}{\texttt{l[v]} \in \texttt{l[p]}} \qquad \text{(LXP-Elem)}$$

$$\frac{\texttt{v} \in \texttt{def(X)}}{\texttt{v} \in \texttt{X}} \qquad \text{(LXP-Def)}$$

$$\frac{\texttt{v} = \texttt{v}_1, \texttt{v}_2 \qquad \texttt{v}_1 \in \texttt{p}_1 \qquad \texttt{v}_2 \in \texttt{p}_2}{\texttt{v} \in \texttt{p}_1\texttt{,}\texttt{p}_2} \qquad \text{(LXP-Cat)}$$

$$\frac{\texttt{v}_1 \in \texttt{p}_1}{\texttt{v} \in \texttt{p}_1\texttt{|}\texttt{p}_2} \qquad \text{(LXP-UniL)}$$

$$\frac{\texttt{v}_2 \in \texttt{p}_2}{\texttt{v} \in \texttt{p}_1\texttt{|}\texttt{p}_2} \qquad \text{(LXP-UniR)}$$

Figure 4.2: XtaticLite pattern-matching semantics

Figure 4.2 defines the semantics of patterns by a binary relation $\texttt{v} \in \texttt{p}$ over sequence values and patterns. The empty sequence matches the empty pattern; an element matches an element pattern if the labels of the value and the pattern are the same, and if the children of the element match the sub-pattern; a value matches a pattern variable if it matches the corresponding pattern; a non-empty sequence matches a concatenation pattern if it can be split into two parts both matching the corresponding sub-patterns; a value matches a union pattern if it matches one of the alternatives.

The term language of XTATICLITE includes variable references, value building constructs, function calls, and pattern matching expressions. Pattern matching expressions have the form `match t with` $\mathtt{p}_1 \rightarrow \mathtt{t}_1$ `|` $\ldots$ `|` $\mathtt{p}_n \rightarrow \mathtt{t}_n$ `else` $\mathtt{t}_0$ where the default clause is optional. To evaluate a `match` expression, XTATICLITE computes `t` and matches the result against patterns $\mathtt{p}_1 \ldots \mathtt{p}_n$ evaluating the right hand side of the first clause containing a matching pattern or $\mathtt{t}_0$ if no patterns matched. If the default clause is omitted, the `match` expression is said to be *exhaustive*. In such cases, we can assume that any input value matches at least one of the patterns. (This is ensured statically by the source language's type checker.)

4.1.2 Example: The following program defines two mutually recursive patterns, `X` and `Y` (matching respectively `a[]`; `a[a[a[]],a[]]`; etc. and `(); a[a[]],a[]`; etc.) and a function that checks whether its argument matches `X` or `Y` and returns 1 or 2, respectively, or 0 if the argument matches neither pattern.

```
def X = a[Y],Y
def Y = () | a[X],X
```

```
fun F(x) =
 match x with
  | X → 1
  | Y → 2
  else → 0
```

In the remainder of this chapter, we discusses compilation of pattern matching expressions only. Furthermore, we only consider a restricted form of `match` expressions—`match t with` $\mathtt{p}_1 \rightarrow$ `1 |` $\ldots$ `|` $\mathtt{p}_n \rightarrow$ `n else 0`—where each right hand side is an integer identifying the position of the corresponding clause in the list of clauses. This restriction together with the ones described in the beginning of this section helps us carry out a complete formal development of the algorithms presented in this chapter.

4.2 Intermediate Language

This section describes the intermediate language XIL that is sufficient to represent XTATICLITE programs at a low level.

4.2.1 Definition: Patterns, exit-free terms, terms with exits, and top-level definitions of XIL are described by the grammar shown in Figure 4.3 (where `l,x,n,A` and `j` range over XDuce labels, variable names, natural numbers, function names, and exception labels respectively.)

p	::=	`()`	empty sequence
		`l[`$\mathtt{x}_1$`],`$\mathtt{x}_2$	non-empty sequence
t	::=	`x`	term variable
		`n`	integer
		`()`	empty sequence
		`l[t]`	single element sequence
		$\mathtt{t}_1$`,`$\mathtt{t}_2$	sequence concatenation
		`[`$\mathtt{t}_1, \ldots, \mathtt{t}_k$`]`	tuple constructor
		π_n`(t)`	tuple projection
		`A(t)`	function call
		`and(`$\overline{\mathtt{t}}$`)`	conjunction
		`if` $\mathtt{t}_1$ `then` $\mathtt{t}_2$ `else` $\mathtt{t}_3$	conditional
e	::=	`t`	exit-free term
		`case x of` $\overline{\mathtt{p} \rightarrow \mathtt{e}}$ `else` $\mathtt{e}_0$	pattern matching
		`exit j(`$\overline{\mathtt{t}}$`)`	raising a lexical exception
d	::=	`fun A(x) =` $\mathtt{e}_1$ `with` $\overline{\mathtt{j}(\overline{\mathtt{x}}) \rightarrow \mathtt{e}}$	function declaration

Figure 4.3: XIL syntax

Four features distinguish XIL from XStaticLite: booleans, tuples, simpler pattern matching constructs, and static exceptions. Let us briefly discuss these features and motivation behind them.

It is not possible to compile some XStaticLite pattern matching expressions—particularly those involving recursive patterns—into single XIL fragments. To implement such expressions, our compiler generates auxiliary functions each performing some subtask of the overall task. To indicate whether a given subtask succeeds, the corresponding auxiliary function returns a boolean value. Instead of introducing a separate boolean data type, we found it convenient to encode booleans by integers—`0` and `1` representing `false` and `true` respectively. Boolean values can only be used in tuple constructors and `if` and `and` expressions.

Sometimes it is useful to employ auxiliary functions that return several bits of information. To implement multi-value return, we use tuples of booleans—the third kind of values in addition to sequence and integer values of XStaticLite. Tuple values can only be used in projection operations.

Pattern matching expressions of XIL have the same structure as `match` expressions of XStaticLite, but only two kinds of rudimentary patterns are available: one matches the empty sequence; the other matches a non-empty sequence extracting the contents of the first element, and the sequence of the remaining elements. Like `match` expressions, `case` expressions can have a fall-through case, and, if it is absent, the `case` expression's patterns can be assumed to be exhaustive.

Similarly to the internal language of the OCaml compiler [12], XIL uses static exceptions to provide unconditional parameterized forward jumps—also referred to as exits—within a function body. Static exceptions are introduced in conjunction with a function declaration. A function consists of a body and a collection of exception handlers. An exception handler consists of a unique label, a collection of formal parameters, and a body. An exception is raised by an `exit` statement specifying the label and the actual parameters. The body of a function can raise any exception associated with that function. An exception handler can raise only those exceptions that appear after itself in the declaration. Consider the following program. It takes a pair of boolean values and returns 1 if the first component of the pair is `true`. If the first component is `false`, it returns 2 if the second component is `true` or 3 otherwise. Notice that $\mathtt{j}_2$ can be raised either in the body of the function or in the body of the $\mathtt{j}_1$ handler; $\mathtt{j}_1$ can only be raised in the body of the function.

```
fun A(x) =
  if π1(x) then 1
  else exit j1(π2(x))
  with
    j1(y) →
      if y then 2
      else exit j2()
    j2() → 3
```

To simplify reasoning about static exceptions, we employ a two-level syntax that distinguishes two kinds of terms—those that can contain an exit statement as a subterm and those that cannot. Furthermore, we syntactically restrict exit statements to occur only in tail positions of other terms.

Static exceptions allow us to encode join points. The next example illustrates that this capability is critical for avoiding code duplication when two or more code fragments share some pattern-matching steps.

4.2.2 Example: Consider the XStaticLite program shown in Figure 4.4(a). It consists of a function that performs the following check on its argument `x`:

- if `x` is bound to `a[],b[]` or `b[],a[]`, then return 1;
- if `x` is bound to `a[],a[]` or `b[],b[]`, then return 2;
- if `x` contains a three-element sequence whose first two elements are either `a[]` or `b[]` and whose third element is `a[]`, then return 3.

```
fun A(x) =
 match x with
  | a[],b[] | b[],a[] → 1
  | a[],a[] | b[],b[] → 2
  | (a[]|b[]), (a[]|b[]), a[] → 3
```

(a)

```
fun A(x) =
 case x of
  | a[x1],x2 →
   case x2 of
    | b[x3],x4 → exit j1(x4)
    | a[x3],x4 → exit j2(x4)
  | b[x1],x2 →
   case x2 of
    | b[x3],x4 → exit j2(x4)
    | a[x3],x4 → exit j1(x4)
 with
  j1(x4) →
   case x4 of
    | () → 1
    else 3
  j2(x4) →
   case x4 of
    | () → 2
    else 3
```

(b)

Figure 4.4: An illustration of join points implemented by static exceptions: a XTATICLITE program (a) and an equivalent XIL programs (b)

Taking into account that the above `match` expression is exhaustive, and, hence, only the enumerated values can be passed as its parameters, the presented source program can be translated into the XIL program shown in Figure 4.4(b). This program starts by checking whether the first element is tagged by `a` or by `b`. In either case, it proceeds to perform a similar check on the second element. If the first and the second elements are tagged by different labels, the program must check whether the remainder of the sequence is empty; if so, the input value matches the pattern of the first `match` clause; otherwise, it matches the pattern of the third `match` clause. This check is encoded by static exception $\texttt{j}_1$. Static exception $\texttt{j}_2$ corresponds to the case when the first and second elements are tagged by the same label.

We can rewrite the displayed XIL program without static exceptions by inlining the `case` expressions appearing below the `with` keyword in place of the corresponding `exit` statements. This would lead to slight code duplication since there are two occurrences of exits to both $\texttt{j}_1$ and $\texttt{j}_2$. In general, avoiding join points by code duplication is impractical since it often leads to exponential code explosion.

We can also implement join points and jumps by functions and function calls respectively. For several reasons, however, we avoid doing so and reserve the machinery of top-level functions strictly

for handling recursive patterns. The benefits of using static exceptions instead of top-level functions are as follows:

- it simplifies correctness proofs for our intermediate code generation algorithm;
- it simplifies specification and implementation of various intermediate code optimization passes such as `exit` folding;
- it results in more aesthetically pleasing intermediate code since patterns are more likely to be translated inline into self-contained XIL fragments.

To define the semantics of static exceptions, we parameterize XIL's evaluation relation an mapping from exception labels to the corresponding exception handlers. An exit environment Δ is a partial function mapping an exception label to a triple $(\overline{\mathtt{x}},\mathtt{e},\Delta')$ containing a sequence of variables, a term, and an exit environment. A function declaration $\mathtt{fun\ A(x)\ =\ e}_1\ \mathtt{with\ j}_2(\overline{\mathtt{x}_2})\ \rightarrow\ \mathtt{e}_2\ \ldots\ \mathtt{j}_m(\overline{\mathtt{x}_m})\ \rightarrow\ \mathtt{e}_m$ induces a collection of exit environments $\Delta_1,\ldots,\Delta_m$ where $\Delta_i(\mathtt{j}_k) = (\overline{\mathtt{x}_k},\mathtt{e}_k,\Delta_k)$ for each $k \in \{i+1,\ldots,m\}$. Function declarations are represented by a mapping `fdef`; for example, the above declaration results in $\mathtt{fdef(A)} = (\mathtt{x},\mathtt{e}_1,\Delta_1)$.

4.2.3 Definition: Judgments for function application: $\mathtt{call\ A(v')} \longrightarrow \mathtt{v}$; and term evaluation: $E \vdash \mathtt{t} \longrightarrow \mathtt{v}$ and $E \bullet \Delta \vdash \mathtt{e} \longrightarrow \mathtt{v}$ are defined as the least fixed point of the inference rules in Figure 4.5. $E[\overline{\mathtt{v}}/\overline{\mathtt{x}}]$ denotes an environment mapping $\mathtt{x} \in \overline{\mathtt{x}}$ to the corresponding $\mathtt{v} \in \overline{\mathtt{v}}$ and agreeing with E on all other variables. $E\backslash\mathtt{y}$ denotes an environment which is undefined on `y`, but is otherwise equal to E.

4.3 Two Compilation Schemes

We intend to study in detail two compilation schemes, differing in their handling of recursive patterns. In the *backtracking* approach, every recursive pattern induces a target language helper function that returns true (1) if its input matches the pattern or false (0) if it does not. Figure 4.6(a) shows the result of compiling the source program of Example 4.1.2 using the backtracking approach. The two mutually recursive functions `X` and `Y` correspond to the source patterns with the same names. (For brevity, we omit `X`; it is similar to `Y`, except that it does not check for the empty sequence.) This program is backtracking because the tests of the "else if" branch of `F` involve traversing the values that are also processed during the tests of the "if" branch.

In the *non-backtracking* approach, helper functions correspond to sets of recursive patterns. Instead of returning booleans, they return tuples of booleans $[\mathtt{t}_1,\ldots,\mathtt{t}_n]$ indicating which of the

$$E \vdash \mathtt{n} \longrightarrow \mathtt{n} \qquad \text{(LXIL-Int)}$$

$$E \vdash \mathtt{x} \longrightarrow E(\mathtt{x}) \qquad \text{(LXIL-Var)}$$

$$\frac{\exists \mathtt{t} \in \overline{\mathtt{t}}.\ E \vdash \mathtt{t} \longrightarrow \mathtt{0}}{E \vdash \mathtt{and}(\overline{\mathtt{t}}) \longrightarrow \mathtt{0}} \qquad \text{(LXIL-And1)}$$

$$\frac{\forall \mathtt{t} \in \overline{\mathtt{t}}.\ E \vdash \mathtt{t} \longrightarrow \mathtt{1}}{E \vdash \mathtt{and}(\overline{\mathtt{t}}) \longrightarrow \mathtt{1}} \qquad \text{(LXIL-And2)}$$

$$\frac{E \vdash \mathtt{t}_1 \longrightarrow \mathtt{1} \qquad E \vdash \mathtt{t}_2 \longrightarrow \mathtt{v}_2}{E \vdash \mathtt{if\ t}_1\ \mathtt{then\ t}_2\ \mathtt{else\ t}_3 \longrightarrow \mathtt{v}_2} \qquad \text{(LXIL-If1)}$$

$$\frac{E \vdash \mathtt{t}_1 \longrightarrow \mathtt{0} \qquad E \vdash \mathtt{t}_3 \longrightarrow \mathtt{v}_3}{E \vdash \mathtt{if\ t}_1\ \mathtt{then\ t}_2\ \mathtt{else\ t}_3 \longrightarrow \mathtt{v}_3} \qquad \text{(LXIL-If2)}$$

$$\frac{E \vdash \mathtt{t} \longrightarrow \mathtt{v} \qquad \mathtt{call\ A(v)} \longrightarrow \mathtt{v}'}{E \vdash \mathtt{A(t)} \longrightarrow \mathtt{v}'} \qquad \text{(LXIL-App)}$$

$$\frac{E \vdash \mathtt{t} \longrightarrow \mathtt{v}}{E \bullet \Delta \vdash \mathtt{t} \longrightarrow \mathtt{v}} \qquad \text{(LXIL-SimpTerm)}$$

$$\frac{\begin{array}{c} E \vdash \overline{\mathtt{t}} \longrightarrow \overline{\mathtt{v}} \qquad \Delta(\mathtt{j}) = (\overline{\mathtt{x}}, \mathtt{e}', \Delta') \\ \emptyset[\overline{\mathtt{v}}/\overline{\mathtt{x}}] \bullet \Delta' \vdash \mathtt{e}' \longrightarrow \mathtt{v}' \end{array}}{E \bullet \Delta \vdash \mathtt{exit\ j}(\overline{\mathtt{t}}) \longrightarrow \mathtt{v}'} \qquad \text{(LXIL-Exit)}$$

$$\frac{\begin{array}{c} E(\mathtt{x}) = \mathtt{()} \qquad \mathtt{()} \rightarrow \mathtt{t}' \in \overline{\mathtt{p} \rightarrow \mathtt{t}} \\ E' = E \backslash \mathtt{x} \qquad E' \bullet \Delta \vdash \mathtt{t}' \longrightarrow \mathtt{v}' \end{array}}{E \bullet \Delta \vdash \mathtt{case\ x\ of\ } \overline{\mathtt{p} \rightarrow \mathtt{t}}\ \mathtt{else\ t}_0 \longrightarrow \mathtt{v}'} \qquad \text{(LXIL-Case1)}$$

$$\frac{\begin{array}{c} E(\mathtt{x}) = \mathtt{l[w_1], w_2} \qquad \mathtt{(l[u], v)} \rightarrow \mathtt{t}' \in \overline{\mathtt{p} \rightarrow \mathtt{t}} \\ E' = (E \backslash x)[\mathtt{w_1/u}, \mathtt{w_2/v}] \qquad E' \bullet \Delta \vdash \mathtt{t}' \longrightarrow \mathtt{v}' \end{array}}{E \bullet \Delta \vdash \mathtt{case\ x\ of\ } \overline{\mathtt{p} \rightarrow \mathtt{t}}\ \mathtt{else\ t}_0 \longrightarrow \mathtt{v}'} \qquad \text{(LXIL-Case2)}$$

$$\frac{\begin{array}{c} E(\mathtt{x}) = \mathtt{v} \qquad \forall \mathtt{p}' \rightarrow \mathtt{t}' \in \overline{\mathtt{p} \rightarrow \mathtt{t}}.\ E \vdash \mathtt{v} \notin \mathtt{p}' \\ E \bullet \Delta \vdash \mathtt{t}_0 \longrightarrow \mathtt{v}' \end{array}}{E \bullet \Delta \vdash \mathtt{case\ x\ of\ } \overline{\mathtt{p} \rightarrow \mathtt{t}}\ \mathtt{else\ t}_0 \longrightarrow \mathtt{0}} \qquad \text{(LXIL-Case3)}$$

$$\frac{\mathtt{fdef(A)} = \mathtt{(x, e', \Delta')} \qquad \emptyset[\mathtt{v/x}] \bullet \Delta' \vdash \mathtt{e}' \longrightarrow \mathtt{v}'}{\mathtt{call\ A(v)} \longrightarrow \mathtt{v}'} \qquad \text{(LXIL-Call)}$$

Figure 4.5: XIL Evaluation

```
fun F(x) =
 case x of
 | () →
   2
 | a[x],y →
   if Y(x) && Y(y) then 1
   else
   if X(x) && X(y) then 2
   else 0
 else 0

fun Y(x) =
 case x of
 | () → 1
 | a[x],y →
   if X(x) && X(y)
    then 1
   else 0
 else 0
```

(a)

```
fun F(x) = case x of
  | () → 2
  | a[x],y →
    let pr1 = XY(x) in
    let pr2 = XY(y) in
    if π2(pr1) && π2(pr2) then 1
    else
    if π1(pr1) && π1(pr2) then 2
    else 0
  else 0

fun XY(x) =
 case x of
  | () → [0,1]
  | a[x],y →
    let pr1 = XY(x) in
    let pr2 = XY(y) in
    [π2(pr1) && π2(pr2),
     π1(pr1) && π1(pr2)]
```

(b)

Figure 4.6: Backtracking (a) and non-backtracking (b) target programs

set of patterns match the function's input. Figure 4.6(b) contains the result of compiling the sample program in the non-backtracking approach. The helper function `XY` returns a pair whose first and second components correspond to patterns `X` and `Y` respectively.

The advantages of the backtracking approach are that operations on boolean values are more efficient than operations on tuples of boolean values and that the number of helper functions is guaranteed to be at most linear in the size of the patterns. The price of this is backtracking and suboptimal performance for some matching problems. Conversely, the non-backtracking approach generates programs that employ more complex operations and potentially exponentially many helper functions, but that are always guaranteed to run in time at worst linear in the size of the input.

We now proceed to a more formal development of these compilation schemes.

4.4 Matching Automata

In this section, we review standard top-down tree automata, introduce matching automata, define special backtracking and non-backtracking forms of matching automata, and present examples of

matching automata illustrating several important compilation issues.

4.4.1 Tree Automata

The semantics of source patterns described in the previous section does not directly lead to an efficient pattern matching algorithm. In particular, matching an input value $\mathtt{v}$ against a pattern of the form $\mathtt{p_1,p_2}$ involves splitting $\mathtt{v}$ at an arbitrary position, $\mathtt{v = v_1,v_2}$, and matching $\mathtt{v_1}$ and $\mathtt{v_2}$ against $\mathtt{p_1}$ and $\mathtt{p_2}$, respectively. If $\mathtt{v}$ is a long sequence, this kind of non-deterministic processing will be prohibitively expensive. For the same reason, it is difficult to compile source patterns into efficient pattern-matching code directly.

Matching against a pattern of the special form $\mathtt{l[p_1],p_2}$, on the other hand, can be executed deterministically by checking that the value's first element matches $\mathtt{l[p_1]}$ and its remaining elements match $\mathtt{p_2}$. Thus, converting source patterns into a form in which the first component of any concatenation is a labeled pattern will provide us with a better starting point for generating efficient pattern matching code. (In rare cases, this conversion may result in space blow-up; see discussion of complexity issues in Section 4.5.7.) This form of patterns can be described by states of a non-deterministic, top-down tree automaton.

4.4.1 Definition: A *non-deterministic top-down tree automaton* is a tuple $\mathtt{A = (S,T)}$, where $\mathtt{S}$ is a set of states and $\mathtt{T}$ is a set of transitions consisting of *empty* transitions of the form $\mathtt{s \to ()}$ and *label* transitions of the form $\mathtt{s \to l[s_1],s_2}$, where $\mathtt{s,s_1,s_2 \in S}$ and $\mathtt{l}$ is a label. The acceptance relation on values and states, denoted $\mathtt{v \in s}$, is defined by the following rules:

$$\frac{\mathtt{s \to () \in T}}{\mathtt{() \in s}}\ \text{(TA-Emp)} \qquad\qquad \frac{\mathtt{s \to l[s_1],s_2 \in T} \quad \mathtt{v_1 \in s_1} \quad \mathtt{v_2 \in s_2}}{\mathtt{l[v_1],v_2 \in s}}\ \text{(TA-Lab)}$$

The patterns of Example 4.1.2 shown in the previous section can be converted into a tree automaton with two states, $\mathtt{S} = \{\mathtt{s_1}, \mathtt{s_2}\}$ (corresponding to patterns $\mathtt{X}$ and $\mathtt{Y}$ respectively), two label transitions, $\mathtt{s_1 \to a[s_2],s_2}$ and $\mathtt{s_2 \to a[s_1],s_1}$, and an empty transition $\mathtt{s_2 \to ()}$. Let us derive $\mathtt{a[a[]],a[] \in s_2}$.

(1) $\mathtt{a[a[]],a[] \in s_2}$	by TA-Lab from 2 instances of (2)
(2) $\mathtt{a[] \in s_1}$	by TA-Lab from 2 instances of (3)
(3) $\mathtt{() \in s_2}$	by TA-Emp

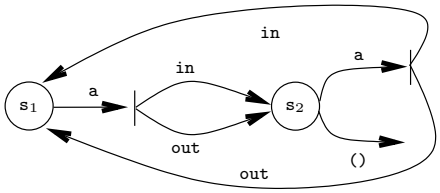


Figure 4.7: Sample Tree Automaton

Tree automata can be depicted by graphs whose nodes and edges represent states and transitions as follows. For any transition of the form $\mathtt{s}_1 \rightarrow \mathtt{l[s}_2\mathtt{],s}_3$, there is an edge labeled `l` from the node corresponding to $\mathtt{s}_1$ to a bar which has two outgoing edges: one, labeled `in`, leads to the node corresponding to $\mathtt{s}_2$, and the other, labeled `out`, leads to the node corresponding to $\mathtt{s}_3$. An empty transition of the form $\mathtt{s} \rightarrow \mathtt{()}$ is represented by a dangling edge labeled by `()`. Figure 4.7 shows the tree automaton discussed above.

From now on, we will assume that source program patterns have been converted into a tree automaton, and the subsequent algorithms will deal with the states of this automaton. Hosoya and Pierce [35] give a detailed description of the algorithm converting a collection of source patterns into states of a tree automaton. Essentially, it transforms patterns into a disjunctive normal form by applying associativity of concatenation and distributivity of concatenation with respect to union.

4.4.2 Matching Automata

Tree automata are a good first step, but several factors make them inappropriate for representing low-level pattern matching code. The first has to do with handling values of the form $\mathtt{l[v}_1\mathtt{],v}_2$. While a tree automaton processes $\mathtt{v}_1$ and $\mathtt{v}_2$ *independently*, a target program should be able to handle them sequentially and use information obtained during inspection of $\mathtt{v}_1$ to drive processing of $\mathtt{v}_2$. The second concerns the treatment of recursive patterns. While, in tree automata, circularities entailed by recursive patterns are implicit in the transition relation, in the target language they must be implemented by recursive procedures. Since there may be multiple ways of achieving this goal, tree automata transitions alone are not sufficient for modeling target language. The third issue arises from the fact that tree automata are designed to match against a single pattern, whereas, to implement `match` expressions, we need a mechanism for matching efficiently against *collections* of patterns.

For more effective processing of subtrees, we introduce transitions with variables. This new kind of transitions has a source state, a target language pattern that determines when the transition is

applicable and binds the subtrees of the current value, and a set of destination pairs that specify the continuation of the transition. For instance, a tree automaton transition $\mathtt{s}_1 \rightarrow \mathtt{a[s}_2\mathtt{],s}_3$ can be rewritten as a transition with variables $\mathtt{s}_1 : \mathtt{a[x],y} \rightarrow \{\mathtt{x} \in \mathtt{s}_2, \mathtt{y} \in \mathtt{s}_3\}$. When $\mathtt{s}_1$ receives a value $\mathtt{a[v}_1\mathtt{],v}_2$, the automaton binds $\mathtt{x}$ to $\mathtt{v}_1$ and $\mathtt{y}$ to $\mathtt{v}_2$ and transfers control to the destination states sending the contents of $\mathtt{x}$ to $\mathtt{s}_2$ and the contents of $\mathtt{y}$ to $\mathtt{s}_3$. Once we introduce variables, $\mathtt{v}_1$ and $\mathtt{v}_2$ do not have to be processed immediately; instead, the automaton can proceed examining one of them while keeping the other stored in a variable for future reference. The following example demonstrates that such flexibility can be advantageous.

4.4.2 Example: It can be shown that there does not exist a deterministic top-down tree automaton that implements the pattern `a[],b[] | a[b[]]`. With the help of transitions with variables, however, it is possible to recognize it deterministically. Consider an automaton `(S,T)` where $\mathtt{S} = \{\mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3, \mathtt{s}_2\}$, and `T` contains the following transitions:

$$\mathtt{s}_1 : \mathtt{a[x],y} \rightarrow \{\mathtt{x} \in \mathtt{s}_2\}$$
$$\mathtt{s}_2 : \mathtt{b[w],z} \rightarrow \{\mathtt{w} \in \mathtt{s}_4, \mathtt{z} \in \mathtt{s}_4, \mathtt{y} \in \mathtt{s}_4\}$$
$$\mathtt{s}_2 : \mathtt{()} \rightarrow \{\mathtt{y} \in \mathtt{s}_3\}$$
$$\mathtt{s}_3 : \mathtt{b[x],y} \rightarrow \{\mathtt{x} \in \mathtt{s}_4, \mathtt{y} \in \mathtt{s}_4\}$$
$$\mathtt{s}_4 : \mathtt{()} \rightarrow \emptyset$$

The transition originating in $\mathtt{s}_1$ saves subtrees $\mathtt{v}_1$ and $\mathtt{v}_2$ of the input value $\mathtt{a[v}_1\mathtt{],v}_2$ in `x` and `y` and sends the contents of `x` to $\mathtt{s}_2$. In $\mathtt{s}_2$, the automaton examines the shape of `x` and, depending on the result, processes the contents of `y`: if `x` contains a `b`-labeled element, `y` is sent to $\mathtt{s}_4$; if `x` contains the empty sequence, `y` is sent to $\mathtt{s}_3$. Observe that the above automaton is deterministic since no state is a source of multiple transitions with the same label.

In addition to simple transitions with variables discussed in the above example, we introduce subroutine transitions to make automata look more like target language code with respect to handling recursive patterns. A subroutine transition has the form $\mathtt{s} : \mathtt{A} \rightarrow \{\mathtt{y}_1 \in \mathtt{s}_1, \ldots, \mathtt{y}_k \in \mathtt{s}_k\}$ where `A` is the name of a subroutine automaton. When `s` receives a value `v`, the subroutine automaton `A` is invoked, and, if it accepts `v`, the destination pairs are evaluated as in simple transitions.

To support matching against multiple patterns, we introduce index sets and index mapping relations. The idea is for an automaton not to simply accept or reject its input, but also to output an integer index in case of acceptance. For instance, an automaton for a matching problem based on patterns $\mathtt{p}_1$, ..., $\mathtt{p}_k$ would output an index $\mathtt{i} \in \{1, \ldots, \mathtt{k}\}$ iff its input matches $\mathtt{p}_i$. To accommodate computing with indices, we enrich simple transitions with index sets and subroutine transitions with

index mapping relations. Thus, a simple transition has the form $\mathtt{q} : \mathtt{p} \xrightarrow{\mathtt{I}} \{\mathtt{y}_1 \in \mathtt{q}_1, \ldots, \mathtt{y}_n \in \mathtt{q}_n\}$ where $\mathtt{I}$ is a set integer indices, and a subroutine transition has the form $\mathtt{q} : \mathtt{A} \xrightarrow{\sigma} \{\mathtt{y}_1 \in \mathtt{q}_1, \ldots, \mathtt{y}_n \in \mathtt{q}_n\}$ where σ is a binary relation on indices. The index set in a simple transition indicates which of the original patterns can still match the input value when the transition is taken. The index mapping relation in a subroutine transition serves a similar function: the pattern $\mathtt{p}_k$ can still be matched when the transition is taken as long as the subroutine automaton accepts the current value returning $\mathtt{j}$ and $(\mathtt{j} \mapsto \mathtt{k}) \in \sigma$.

The following definition summarizes all of the above concerns. We write $E[\mathtt{v}_1/\mathtt{x}, \mathtt{v}_2/\mathtt{y}]$ to denote an environment mapping $\mathtt{x}$ to $\mathtt{v}_1$ and $\mathtt{y}$ to $\mathtt{v}_2$ and agreeing with E on all other variables and $E\backslash\mathtt{y}$ to denote an environment which is undefined on $\mathtt{y}$ and otherwise equal to E.

4.4.3 Definition: A *matching automaton* is a tuple $\mathtt{(Q,q}_s\mathtt{,R)}$, where $\mathtt{Q}$ is a set of states, $\mathtt{q}_s$ is a start state, and $\mathtt{R}$ is a set of transitions. There are two kinds of transitions: *simple* and *subroutine*. They have the following structure:

$$\mathtt{q} : \mathtt{p} \xrightarrow{\mathtt{I}} \{\mathtt{y}_1 \in \mathtt{q}_1, \ldots, \mathtt{y}_m \in \mathtt{q}_m\} \qquad \text{(simple)}$$
$$\mathtt{q} : \mathtt{A} \xrightarrow{\sigma} \{\mathtt{y}_1 \in \mathtt{q}_1, \ldots, \mathtt{y}_m \in \mathtt{q}_m\} \qquad \text{(subroutine)}$$

Both types of transitions have a *source state* $\mathtt{q}$ and a set of *destination pairs* $\{\mathtt{y}_1 \in \mathtt{q}_1 \ldots \mathtt{y}_m \in \mathtt{q}_m\}$. A destination pair consists of a *destination variable* $\mathtt{y}_i$ and a *destination state* $\mathtt{q}_i$. A simple transition contains a target language pattern $\mathtt{p}$—which can be of the form $\mathtt{()}$ or $\mathtt{l[x],z}$—and a set of integer indices $\mathtt{I}$. A subroutine transition contains a subroutine automaton name $\mathtt{A}$ and a relation σ mapping indices to indices.

Let $\mathcal{M}$ be a mapping of automaton names to matching automata and let $\mathtt{A} = \mathtt{(Q,q}_s\mathtt{,R)}$ be a matching automaton. The acceptance relation $E \vdash \mathtt{v} \in \mathtt{q} \Rightarrow \mathtt{k}$ is defined on environments, values, states, and indices by the following rules.

$$\frac{\begin{array}{c}\mathtt{q} : \mathtt{()} \xrightarrow{\mathtt{I}} \{\mathtt{y}_1 \in \mathtt{q}_1, \ldots, \mathtt{y}_m \in \mathtt{q}_m\} \in \mathtt{R} \\ \mathtt{k} \in \mathtt{I} \\ \forall i \in \{1, \ldots, m\}.\ E\backslash\mathtt{y}_i \vdash E(\mathtt{y}_i) \in \mathtt{q}_i \Rightarrow \mathtt{k}\end{array}}{E \vdash \mathtt{()} \in \mathtt{q} \Rightarrow \mathtt{k}} \qquad \text{(MA-Emp)}$$

$$\frac{\begin{array}{c}\mathtt{q} : \mathtt{l[x],z} \xrightarrow{\mathtt{I}} \{\mathtt{y}_1 \in \mathtt{q}_1, \ldots, \mathtt{y}_m \in \mathtt{q}_m\} \in \mathtt{R} \\ \mathtt{k} \in \mathtt{I} \qquad E' = E[\mathtt{v}_1/\mathtt{x}, \mathtt{v}_2/\mathtt{z}] \\ \forall i \in \{1, \ldots, m\}.\ E'\backslash\mathtt{y}_i \vdash E'(\mathtt{y}_i) \in \mathtt{q}_i \Rightarrow \mathtt{k}\end{array}}{E \vdash \mathtt{l[v_1],v_2} \in \mathtt{q} \Rightarrow \mathtt{k}} \qquad \text{(MA-Lab)}$$

$$\frac{\begin{array}{c}\mathtt{q} : \mathtt{B} \xrightarrow{\sigma} \{\mathtt{y}_1 \in \mathtt{q}_1, \ldots, \mathtt{y}_m \in \mathtt{q}_m\} \in \mathtt{R} \\ E \vdash \mathtt{v} \in \mathcal{M}(\mathtt{B}) \Rightarrow \mathtt{j} \qquad (\mathtt{j},\mathtt{k}) \in \sigma \\ \forall i \in \{1, \ldots, m\}.\ E \backslash \mathtt{y}_i \vdash E(\mathtt{y}_i) \in \mathtt{q}_i \Rightarrow \mathtt{k}\end{array}}{E \vdash \mathtt{v} \in \mathtt{q} \Rightarrow \mathtt{k}} \qquad \text{(MA-Sub)}$$

A value $\mathtt{v}$ is accepted by the automaton $\mathtt{A}$ with an index $\mathtt{k}$ in an environment E, written $E \vdash \mathtt{v} \in \mathtt{A} \Rightarrow \mathtt{k}$, if it is accepted by the automaton's start state: $E \vdash \mathtt{v} \in \mathtt{q}_s \Rightarrow \mathtt{k}$.

The rule MA-Emp says that the empty sequence `()` is accepted by a state $\mathtt{q}$ in an environment E returning an index $\mathtt{k}$ if there is a transition of the form $\mathtt{q} : () \rightarrow \ldots$ and for each destination pair $\mathtt{y}_i \in \mathtt{q}_i$, the value $E(\mathtt{y}_i)$ is accepted by $\mathtt{q}_i$ returning $\mathtt{k}$ in an environment obtained from E by removing $\mathtt{y}$'s binding. MA-Lab describes how a state can accept a value $\mathtt{l}[\mathtt{v}_1],\mathtt{v}_2$ using a transition of the form $\mathtt{q} : \mathtt{l}[\mathtt{x}],\mathtt{z} \rightarrow \ldots$. It is similar to MA-Emp except that the environments used for checking the destination pairs are extended with bindings of $\mathtt{x}$ to $\mathtt{v}_1$ and $\mathtt{z}$ to $\mathtt{v}_2$. MA-Sub deals with subroutine transitions. A value is accepted by a state $\mathtt{q}$ in an environment E producing an index $\mathtt{k}$ if there is a transition of the form $\mathtt{q} : \mathtt{B} \rightarrow \ldots$, the automaton $\mathcal{M}(\mathtt{B})$ accepts $\mathtt{v}$ in E producing an index $\mathtt{j}$ such that $(\mathtt{j} \mapsto \mathtt{k})$ is in the transition's index mapping relation, and the destination pairs are checked as in MA-Emp.

The index mapping relations in subroutine transitions serve two purposes. First, they allow us to reduce the number of subroutine automata since we can avoid building isomorphic automata that only differ in their indices. Second, and more importantly, they are essential for creating matching automata that represent non-backtracking target programs.

Let us consider a matching automaton which implements Example 4.1.2. This automaton, let us call it `XY`, consists of states $\mathtt{q}_1$ (the start state) and $\mathtt{q}_2$ and transitions

$$\begin{array}{ll}\mathtt{q}_1 : () \xrightarrow{\mathtt{I}_1} \emptyset & \text{where } \mathtt{I}_1 = \{2\}, \\ \mathtt{q}_1 : \mathtt{a}[\mathtt{x}],\mathtt{y} \xrightarrow{\mathtt{I}_2} \{\mathtt{x} \in \mathtt{q}_2\} & \text{where } \mathtt{I}_2 = \{1, 2\}, \text{ and} \\ \mathtt{q}_2 : \mathtt{IC} \xrightarrow{\sigma_1} \emptyset & \text{where } \sigma_1 = \{1 \mapsto 1, 2 \mapsto 2\}\end{array}$$

The subroutine automaton `IC` contains states $\mathtt{q}_3$ (its start state) and $\mathtt{q}_4$ and subroutine transitions

$$\begin{array}{ll}\mathtt{q}_3 : \mathtt{XY} \xrightarrow{\sigma_2} \{\mathtt{y} \in \mathtt{q}_4\} & \text{and} \\ \mathtt{q}_4 : \mathtt{XY} \xrightarrow{\sigma_2} \emptyset & \text{where } \sigma_2 = \{1 \mapsto 2, 2 \mapsto 1\}\end{array}$$

Diagrams of matching automata are similar to those of tree automata except that they must account for the additional annotations on transitions. Edges must be annotated with index sets in case of simple transitions and index mapping relations in case of subroutine transitions. The

parts of edges connecting a bar to a destination state are labeled by the corresponding destination variable instead of the keywords "in" and "out" used in tree automata figures. Figure 4.8(b) shows the automata discussed in the above example.

The goal of `XY` is to output 1 if its input matches `X` or 2 if its input matches `Y`. Let us derive $\emptyset \vdash$ `a[a[]],a[]` $\in$ `XY` $\Rightarrow 2$.

(1) $\emptyset \vdash$ `a[a[]],a[]` $\in$ `XY` $\Rightarrow 2$	by Definition 4.4.3 from (2)
(2) $\emptyset \vdash$ `a[a[]],a[]` $\in q_1 \Rightarrow 2$	by MA-Lab from (3)
(3) $E_1 \vdash$ `a[]` $\in q_2 \Rightarrow 2$	by MA-Sub from (4) $E_1 = \emptyset$[`a[]`/`y`]
(4) $E_1 \vdash$ `a[]` $\in$ `IC` $\Rightarrow 2$	by Definition 4.4.3 from (5)
(5) $E_1 \vdash$ `a[]` $\in q_3 \Rightarrow 2$	by MA-Sub from (6,7)
(6) $E_1 \vdash$ `a[]` $\in$ `XY` $\Rightarrow 1$	derived similarly to (1)
(7) $\emptyset \vdash$ `a[]` $\in q_4 \Rightarrow 2$	by MA-Sub from (8)
(8) $\emptyset \vdash$ `a[]` $\in$ `XY` $\Rightarrow 1$	derived similarly to (1)

4.4.3 Special Forms of Matching Automata

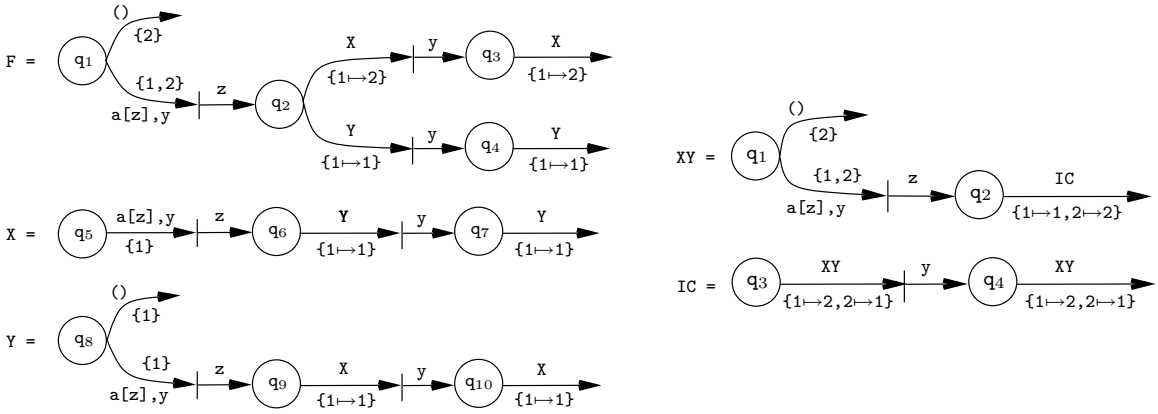


Figure 4.8: Matching Automata in Simple Backtracking (a) and Non-Backtracking (b) Forms

The definition of matching automata is very flexible. For instance, tree automata can be viewed as a special case of matching automata that implement single clause `match` expressions. As a result of this flexibility, not every matching automaton can be easily mapped to a target language program. The purpose of the forthcoming development is to identify matching automata for which this mapping is straightforward. Specifically, we will define two sets of restrictions ensuring that

matching automata correspond directly to either backtracking or non-backtracking target language programs introduced in the previous section. We say that automata satisfying the former set of restrictions are in *simple backtracking form*, and automata satisfying the latter set of restrictions are in *simple non-backtracking form*.

As we introduce various restrictions, we will show how parts of matching automata correspond to target language expressions. We will use the matching automata of Figure 4.8 and the corresponding programs of Figure 4.6 to illustrate this connection.

We begin by discussing restrictions that are pertinent to both backtracking and non-backtracking styles.

We say that a matching automaton is *sequential* if each of its transitions has at most one destination pair. The automata of Figure 4.8 are sequential, for example, but the automaton shown in Figure 4.9(b) below is not: its non-final transitions have two destination pairs.

We say that a matching automaton is *disjoint* if for any state of the automaton, simple transitions originating in this state are non-overlapping. Automata of Figure 4.8 are disjoint; the automaton of Figure 4.9(b) is not since $\mathtt{q}_1$ is a source of two `a`-transitions.

A state of a sequential and disjoint automaton can be converted to a target language `case` expression, each outgoing transition corresponding to a `case` branch. For example, observe how states $\mathtt{q}_1$ and $\mathtt{q}_8$ of Figure 4.8(a) and $\mathtt{q}_1$ of Figure 4.8(b) correspond to the `case` expressions appearing in Figure 4.6.

The remaining restrictions are related to subroutine transitions.

We say that an automaton has *separated* transitions if whenever a state has an outgoing subroutine transition, all other transitions originating in this state are subroutine transitions as well. States of such automata can be partitioned into *subroutine* and *simple* states; each serving as a source of only the corresponding kind of transitions. Moreover, we say that an automaton has *staged* transitions if the destination states of its subroutine transitions are subroutine states. Consider, for instance, the automaton `Y` shown in Figure 4.8(a). It has a simple state $\mathtt{q}_8$ and subroutine states $\mathtt{q}_9$ and $\mathtt{q}_{10}$. The automaton is staged since the destination state of the first subroutine transition is a subroutine state $\mathtt{q}_{10}$ and the second subroutine transition is final and does not have a destination state.

The next concept is specific to the backtracking form of matching automata.

We say that a matching automaton is *boolean* if, for any of its simple transitions $\mathtt{q}_1 : \mathtt{p} \xrightarrow{\mathtt{I}} \ldots$ and for any of its subroutine transitions $\mathtt{q}_2 : \mathtt{A} \xrightarrow{\sigma} \ldots$, it is the case that $\mathtt{I} = \mathit{range}(\sigma) = \{1\}$. A subroutine transition $\mathtt{q} : \mathtt{A} \xrightarrow{\sigma} \ldots$ is *boolean* if $\mathit{dom}(\sigma) = \{1\}$. Since boolean matching automata involve a single index, their function, like tree automata, is either to accept or to reject the input

value. Thus, in the target language boolean automata can be represented by boolean functions.

The backtracking compilation approach employs two kinds of matching automata: a *matcher* implements a source `match` expression; a boolean *acceptor* implements a particular pattern and returns 1 if it matches the input value. Only the latter kind of automata are used as subroutines in the backtracking method.

The automata shown in Figure 4.8(a) are in simple backtracking form. In particular, `F` is a matcher and `X` and `Y` are boolean acceptors. Subroutine states q_2, q_6, and q_9 correspond to the `if` expressions in the target program. Each of these states is the start of one or more "call tails"—sequences of subroutine transitions sharing the same index mapping relation σ. A call tail corresponds to an `if` branch: the conjunction of the subroutine calls constitutes the test, and the index occurring in $range(\sigma)$ is returned if the test succeeds. The following definition summarizes the above restrictions.

4.4.4 Definition: A collection of matching automata is said to be in *simple backtracking form* (SBF) if it can be partitioned into matchers and acceptors, all of which are sequential and disjoint automata with separated and staged transitions. Furthermore, only acceptors may serve as subroutines, and acceptors and subroutine transitions must be boolean.

In the non-backtracking scheme, subroutine states must have at most one outgoing subroutine transition. (This is what ensures that there is no backtracking!) To satisfy this condition, we remove the restriction requiring subroutines to be boolean (so there is no longer a distinction between acceptors and matchers) and introduce an additional kind of matching automata called *index converters* whose purpose is to make a sequence of subroutine calls and convert the returned indices. The automata shown in Figure 4.8(b) are in simple non-backtracking form; `XY` is a matcher, and `IC` is an index converter.

Note that in this example, the index converter is unnecessary; we can inline the subroutine call invoking the converter by substituting q_3 for q_2 without changing the meaning of `XY`. Such inlining, however, is not always possible. Example 4.4.8 below will involve an essential index converter that cannot be eliminated. The following definition formalizes the simple non-backtracking form.

4.4.5 Definition: A collection of matching automata is said to be in a *simple non-backtracking form* (SNBF) if it can be partitioned into a collection of matchers and index converters, all of which are sequential and disjoint automata with separated and staged transitions. Furthermore, converters may only call matchers and vice versa; any subroutine state may be the source of exactly one transition; subroutine transitions in matchers must be final; and index converters may only contain subroutine states. We also require that subroutine transitions in matchers and index

converters be *compatible* with each other: if $\mathtt{q} : \mathtt{IC} \xrightarrow{\sigma} \emptyset$ is a subroutine transition in a matcher and $\mathtt{q}' : \mathtt{A} \xrightarrow{\rho} \ldots$ is a subroutine transition in the index converter `IC`, then $dom(\sigma) = range(\rho)$.

In the target language, matchers are implemented by functions that return tuples of booleans. The elements of the tuple correspond to the different indices output by the matcher. The matcher outputs an index iff the function returns a tuple with the corresponding element set to true. For example, matcher `XY` outputs 1 or 2; thus, the corresponding target function returns a pair. A subroutine state in a matcher gives rise to a `let` expression whose components are generated from the matcher's subroutine transition as well as the subroutine transitions of the index converter.

4.4.4 Examples

While designing our regular pattern compiler, we found that several factors play a major role in the quality of the output code. Sometimes we were surprised by a dramatic effect of some seemingly innocuous change to the compiler on either the performance or size of the generated code. The following series of examples is an extract of what we believe are the most important lessons learned from our experiments.

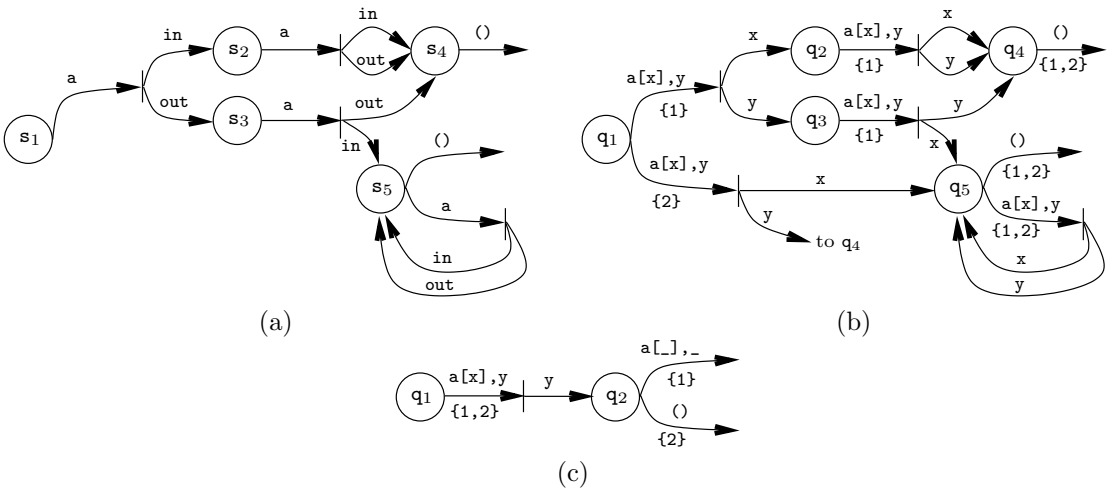


Figure 4.9: Example 4.4.6; correspondence between a tree automaton (a) and a matching automaton (b); an equivalent exhaustive matching automaton (c)

4.4.6 Example: Our first example illustrates two points. First, we show how we can create a matching automaton by a simple modification of the tree automaton corresponding to the matching problem's patterns. We then show how the obtained matching automaton can be converted into a substantially more efficient matching automaton. Consider the following source program fragment:

```
def A = () | a[A],A
```

```
match x with
  | a[a[]],a[A] → 1
  | a[A] → 2
```

A tree automaton built from the patterns of this expression is shown in Figure 4.9(a). Its states $\mathtt{s}_1$ and $\mathtt{s}_3$ correspond to the first and second patterns respectively. A corresponding matching automaton can be constructed directly from the states and transitions of the tree automaton. First, we must account for the difference in the structure of transitions by modifying tree automaton transitions of the form $\mathtt{s}_1 \rightarrow \mathtt{a[s}_2\mathtt{],s}_3$ and $\mathtt{s}_1 \rightarrow \mathtt{()}$ into transitions with variables $\mathtt{s}_1 : \mathtt{a[x],y} \rightarrow \{\mathtt{x} \in \mathtt{s}_2, \mathtt{y} \in \mathtt{s}_3\}$ and $\mathtt{s}_1 : \mathtt{()} \rightarrow \emptyset$ respectively. We also must create a start state that combines the transitions originating in $\mathtt{s}_1$ and $\mathtt{s}_3$, the states corresponding to the patterns of the `match` expression. Finally, we must annotate the transitions with appropriate index sets. The result of this transformation is the matching automaton shown in Figure 4.9(b). It succeeds, outputting 1 or 2, if its input matches the first or second pattern of the `match` expression, respectively; if the input matches neither, the automaton fails.

This matching automaton can be improved. Observe that the `match` expression of this example is exhaustive (it has no `else` branch), so we may assume that the matching automaton will never receive an input value that does not match either source pattern. For values that match one of the patterns, it is sufficient to count the number of top-level elements: if there are two, then the input value matches the first pattern; if one, then the second pattern. This is implemented by the matching automaton shown in Figure 4.9(c). In $\mathtt{q}_1$, it receives a value of the form $\mathtt{a[v}_1\mathtt{],v}_2$ and stores $\mathtt{v}_1$ in `x` and $\mathtt{v}_2$ in `y`; then, in $\mathtt{q}_2$ it investigates the contents of `y`, and, if it is of the form $\mathtt{a[v}_3\mathtt{],v}_4$, returns 1, or, if it is of the form `()`, returns 2. Note, that investigating the contents of `x` before the contents of `y` would not immediately reveal the answer since learning that $\mathtt{v}_1$ is of the form $\mathtt{a[v}_3\mathtt{],v}_4$ does not tell us whether the input value matches the first or the second source pattern.

This new matching automaton is in SBF—in particular, it is disjoint and sequential. In general, making matching automata disjoint and sequential yields two benefits. The obvious benefit is that a disjoint matching automaton involves less backtracking and hence is more efficient than its non-disjoint equivalent. The indirect, but, as our experience and this example indicate, significant benefit is that making automata disjoint and sequential can lead us to a compact solution that traverses only the parts of the input value necessary to determine the result.

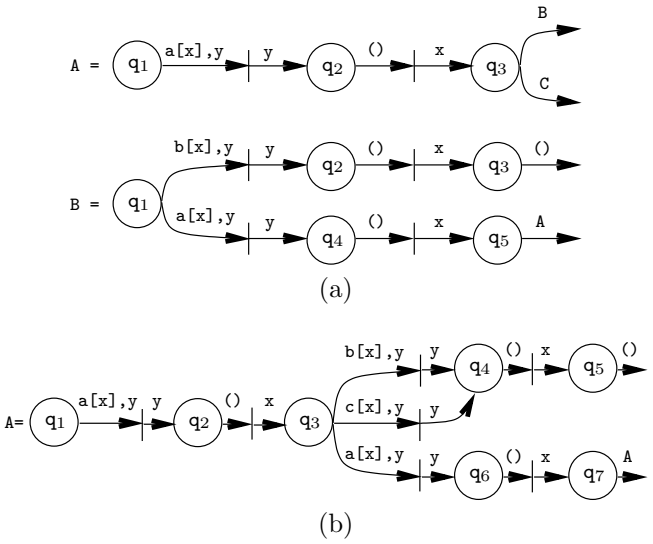


Figure 4.10: Example 4.4.7; exponential (a), and linear subroutine automata (b)

We now move to a discussion of issues related to subroutine automata and subroutine transitions—the issues that influenced most of the essential design choices during the development of our compiler.

4.4.7 Example: In this example, we discuss a program whose most obvious implementation has exponential running time. Consider the following fragment:

```
def A = a[B] | a[C]          match x with
def B = b[] | a[A]            | A → 1
def C = c[] | a[A]            else 0
```

Let us try to convert this `match` expression into a matching automaton in SBF. Our first instinct is to associate each of the three recursive patterns with a separate subroutine automaton; Figure 4.10(a) shows the corresponding solution. (We omit `C` since it is similar to `B`. Also, since the `match` expression of this example consists of a single clause, all index sets and index mapping relations in transitions are vacuously $\{1\}$ and $\{1 \mapsto 1\}$ respectively; so, we omit them from the figure.)

Observe that `A`, if executed sequentially, will take exponentially many steps to reject a value of the form `a[a[ ... [d[]] ... ]]`. The source of this inefficiency lies in the fact that there are two subroutine transitions originating in q_3, and the automaton will backtrack, repeatedly trying one of the transitions, failing, and trying the other transition.

We can obtain a linear matching automaton by observing that it is not necessary to associate a subroutine with each pattern defined recursively in this example. Figure 4.10(b) displays a solution in which only `A` has a corresponding matching automaton. Instead of having subroutine calls to `B` and `C`, this automaton incorporates their states and transitions directly. The new automaton is non-backtracking since it does not have a state with more than one outgoing subroutine transition.

This example shows the benefit of minimizing the number of subroutine automata. Later, however, we will see that this strategy should not be applied naively because it can lead to a huge size explosion.

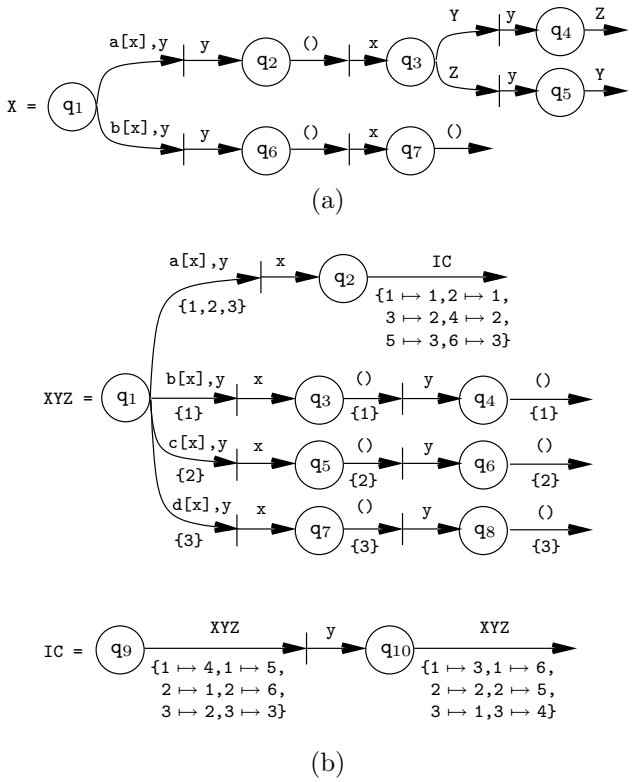


Figure 4.11: Example 4.4.8; exponential boolean (a) and linear non-boolean automata (b)

4.4.8 Example: This example shows that, for some matching problems, using boolean subroutines is not enough. Consider the following program:

```
def X = b[] | a[Y],Z | a[Z],Y          match x with
def Y = c[] | a[Z],X | a[X],Z           | X → 1
def Z = d[] | a[X],Y | a[Y],X           else 0
```

Figure 4.11(a) contains a boolean matching automaton for this `match` expression. This automaton, like the initial matching automaton built for Example 4.10, exhibits exponential running time. Unlike that example, however, it is not clear how to transform the exponential automaton into a more efficient boolean automaton. In such cases, we can fall back to using more general subroutine automata of SNBF. The automaton `XYZ` shown in Figure 4.11 implements this matching problem—it outputs 1, 2, or 3 if the input matches `X`, `Y`, or `Z` respectively. `XYZ`, like any automaton in SNBF, is linear.

As we have mentioned before, in this example, it is not possible to achieve the desired behavior by circumventing the index converter and making the two `XYZ` calls from q_2. Such an automaton would not distinguish values matching `a[Y],Z` and `a[Z],Y`—they should be accepted and 1 should be returned—from values matching `a[Y],Y` and `a[Z],Z`—they should be rejected.

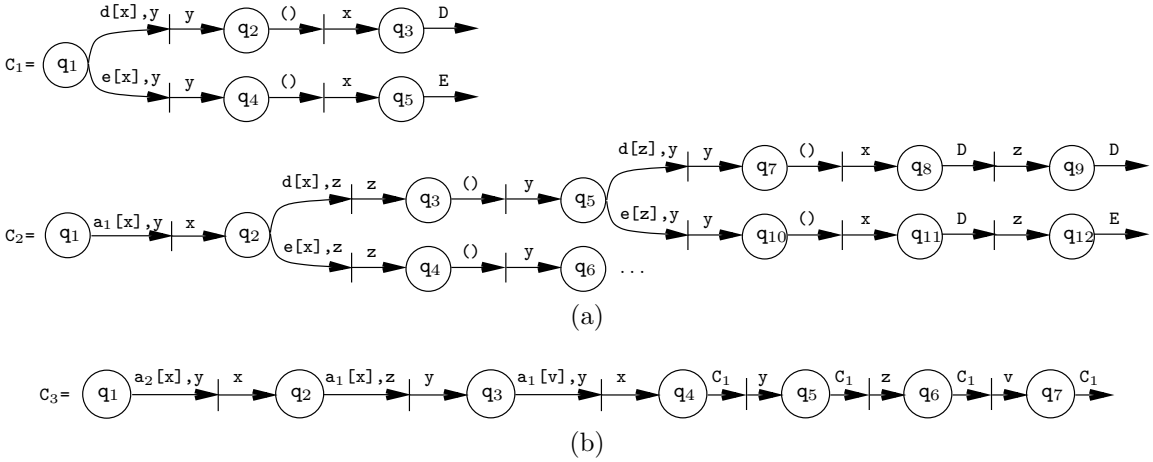


Figure 4.12: Example 4.4.9; equivalent matching automata with a large (a) and a small (b) number of states

4.4.9 Example: We conclude with an example showing a potential drastic explosion of the size of matching automata. Consider the following program:

```
def D = d[D?]
def E = e[E?]
def A1 = a1[C1]          match x with
def A2 = a2[C2]           | C3 → 1
def C1 = D | E            else 0
def C2 = A1,C1
def C3 = A2,C2
```

Only patterns D and E are recursive, and, so, it is reasonable to associate subroutine matching automata with these and only these patterns. Let us try to build matching automata for C_1, C_2, and C_3 in succession. Figure 4.12(a) shows a matching automaton for C_1; it has two final states with subroutine transitions. The same figure displays a significantly larger matching automaton for C_2. (The omitted part of C_2 indicated by ... is similar to the part above it: q_6 is isomorphic to q_5.) Automaton C_2 has four final states. If we build a matching automaton for C_3 following the same pattern, it will have sixteen final states and will not fit on the page. The size grows double-exponentially in the size of the source pattern!

To avoid the above size explosion, it is sufficient to associate a subroutine automaton with C_1 as well as with D and E. Figure 4.12(b) shows a compact matching automaton for C_3 that takes advantage of C_1's subroutine automaton.

Armed with various insights from these examples, we now proceed to a description of our compilation algorithms.

4.5 Compilation

Section 4.4.1 discussed how patterns of a source program can be converted into states of a top-down nondeterministic tree automaton A = (S,T). This section describes the *compilation* algorithm that builds matching automata in simple backtracking or non-backtracking forms for matching problems specified in terms of elements of S. In particular, we will show how, given an ordered sequence of tree automaton states $s_1 \ldots s_n$, we can build a matching automaton, in either of the two special forms, that on an input value v outputs k iff $v \in s_k$.

We start by giving a top-level overview of the SBF compilation algorithm. We then discuss in more detail several key techniques employed in the algorithm and state several of its properties. We conclude the section with an outline of the SNBF compilation algorithm and summarizing complexity of the presented algorithms.

4.5.1 Outline of SBF compilation

The compilation algorithm will manipulate a data structure that is a generalization of sequences of tree automaton states. In this data structure, tree automaton states are arranged into a matrix whose rows and columns are associated with integer indices and variables respectively. More formally:

4.5.1 Definition: Let $\mathtt{A} = \mathtt{(S,T)}$ be a tree automaton. A *configuration* over $\mathtt{S}$ consists of a tuple of distinct variables $\mathtt{<x}_1,\ldots,\mathtt{x}_n\mathtt{>}$ and a set of tuples $\{(\mathtt{s}_{11}, \ldots, \mathtt{s}_{1n}, \mathtt{j}_1), \ldots, (\mathtt{s}_{m1}, \ldots, \mathtt{s}_{mn}, \mathtt{j}_m)\}$ each associating a collection of states from $\mathtt{S}$ to an integer index. A configuration can be depicted as follows:

$$\mathtt{C} = \begin{array}{|ccc|c|} \hline \mathtt{x}_1 & \ldots & \mathtt{x}_n & \\ \hline \mathtt{s}_{11} & \ldots & \mathtt{s}_{1n} & \mathtt{j}_1 \\ & \ldots & & \\ \mathtt{s}_{m1} & \ldots & \mathtt{s}_{mn} & \mathtt{j}_m \\ \hline \end{array}$$

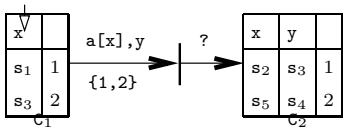
We say that an environment E *satisfies* $\mathtt{C}$ yielding an index $\mathtt{j}_r$, written $E \models \mathtt{C} \Rightarrow \mathtt{j}_r$, if $E(\mathtt{x}_i) \in \mathtt{s}_{ri}$ for all $i \in \{1, \ldots, n\}$. A configuration $\mathtt{C}$ is *satisfiable* by an environment E if there exists an index $\mathtt{k}$ such that $E \models \mathtt{C} \Rightarrow \mathtt{k}$.

The core of the SBF algorithm is a recursive function `sbf` that takes a configuration and produces a matching automaton. Any configuration encountered by `sbf` represents a state of the resulting matching automaton. The algorithm uses two methods of *expanding* configurations to generate transitions of the matching automaton. Expansion results in residual configurations that are used as parameters to recursive calls of `sbf`. The algorithm terminates when the current configuration has no columns.

The following sections provide more details. First, we describe two expansion techniques: one for generating simple transitions, and the other for generating subroutine transitions. Then, we show how `sbf` determines which of the two expansion techniques should be applied to a given configuration and, in a related development, address the size explosion concern raised in Example 4.4.9. We then describe several techniques for optimizing configurations that lead to smaller and more efficient matching automata. This is followed by a formal definition of the algorithm and the proof of its correctness.

4.5.2 Two Configuration Expansion Techniques

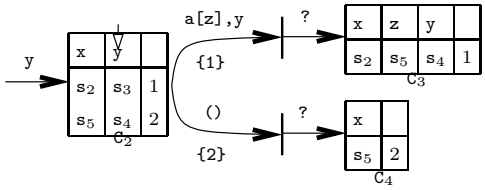
4.5.2 Example: Let us illustrate the expansion method producing simple transitions. We refer to this kind of expansion as expansion *by label*. Consider the following diagram that illustrates the first step in the construction of the matching automaton shown in Figure 4.9(c).



Here, $\mathtt{C}_1$ is the initial configuration that corresponds to the source state of the matching automaton. It includes the tree automaton states $\mathtt{s}_1$ and $\mathtt{s}_3$ that represent the source patterns of the underlying `match` expression (see Figure 4.9(a)). To expand a configuration by label, we must select one of its columns and a label appearing in some transition whose start state is an element of the selected column. There is only one column in $\mathtt{C}_1$; so, it is selected for expansion (as indicated by the vertical arrow.) Both of the states in the selected column are sources of a single `a`-labeled transition. Therefore, there is only one way to expand $\mathtt{C}_1$ by label, and the result of this expansion is the residual configuration $\mathtt{C}_2$.

$\mathtt{C}_2$ is obtained from $\mathtt{C}_1$ as follows. First we remove the selected column and the rows whose state in the selected column does not have an `a`-labeled transition. (No rows are removed in our case.) Then we generate two variables that do not appear in the obtained configuration and add two new columns consisting of these variables and the successor states of the `a`-labeled transitions. In particular, the contents of the two new columns in the first row of $\mathtt{C}_2$ arises from the transition $\mathtt{s}_1 \rightarrow \mathtt{a[s_2],s_3}$, and the contents of the second row arises from the transition $\mathtt{s}_3 \rightarrow \mathtt{a[s_5],s_4}$.

$\mathtt{C}_1$ and $\mathtt{C}_2$ correspond to the states $\mathtt{q}_1$ and $\mathtt{q}_2$ of the matching automaton shown in Figure 4.9(c). The pattern part of the transition between $\mathtt{C}_1$ and $\mathtt{C}_2$ consists of the label `a` and the variables used in the expansion. The index set consists of the indices appearing in the destination configuration. The destination variable is unknown at this stage; it will be determined by the column that is selected for expansion of $\mathtt{C}_2$. The following figure shows how $\mathtt{C}_2$ is expanded.



Let us base expansion of $\mathtt{C}_2$ on the second column (hence `y` becomes the destination variable of the transition generated in the previous paragraph.) There are two distinct labels occurring in transitions whose source states are in the second column: `()` and `a`. Hence, we must expand $\mathtt{C}_2$ twice: once with respect to each label. Expanding $\mathtt{C}_2$ by `a` is done similarly to how we expanded

$\mathtt{C}_1$. Expanding by `()` is even simpler: it follows the same steps as expanding by `a` but does not involve generating variable names or introducing new columns.

The following definition summarizes expansion by label. Function `expand_by_label` generates a residual configuration given the current configuration, a column, and a transition label. The definition considers two cases: one for some binary label `l`; the other for the empty sequence label `()`. In this definition and in the subsequent propositions, we will use `newvars(C,c)` to indicate a pair of variables that do not appear in `C` after its column `c` has been removed.

$$\texttt{expand\_by\_label}\left(\left|\begin{array}{ccc|c} \mathtt{x}_1 & \ldots & \mathtt{x}_n & \\ \hline \mathtt{s}_{11} & \ldots & \mathtt{s}_{1n} & \mathtt{j}_1 \\ & \ldots & & \\ \mathtt{s}_{m1} & \ldots & \mathtt{s}_{mn} & \mathtt{j}_m \end{array}\right|, \mathtt{c}, \mathtt{l}\right) =$$

u	v	$\mathtt{x}_1$	...	$\mathtt{x}_{c-1}$	$\mathtt{x}_{c+1}$	...	$\mathtt{x}_n$	
$\mathtt{t'}_{11}$	$\mathtt{t''}_{11}$	$\mathtt{s}_{11}$	...	$\mathtt{s}_{1(c-1)}$	$\mathtt{s}_{1(c+1)}$	...	$\mathtt{s}_{1n}$	$\mathtt{j}_1$
				...				
$\mathtt{t'}_{1k_1}$	$\mathtt{t''}_{1k_1}$	$\mathtt{s}_{11}$	...	$\mathtt{s}_{1(c-1)}$	$\mathtt{s}_{1(c+1)}$	...	$\mathtt{s}_{1n}$	$\mathtt{j}_1$
				$\vdots$				
$\mathtt{t'}_{m1}$	$\mathtt{t''}_{m1}$	$\mathtt{s}_{m1}$	...	$\mathtt{s}_{m(c-1)}$	$\mathtt{s}_{m(c+1)}$	...	$\mathtt{s}_{mn}$	$\mathtt{j}_m$
				...				
$\mathtt{t'}_{mk_m}$	$\mathtt{t''}_{mk_m}$	$\mathtt{s}_{m1}$	...	$\mathtt{s}_{m(c-1)}$	$\mathtt{s}_{m(c+1)}$	...	$\mathtt{s}_{mn}$	$\mathtt{j}_m$

where $\{(\mathtt{t'}_{i1}, \mathtt{t''}_{i1}), \ldots, (\mathtt{t'}_{ik_i}, \mathtt{t''}_{ik_i})\} = \{(\mathtt{t'}, \mathtt{t''}) \mid \mathtt{s}_{ic} \rightarrow \mathtt{l[t'],t''} \in \mathbf{T}\}$ for $i \in \{1, \ldots, m\}$ and `(u,v)` = `newvars(C,c)` for the input configuration `C`

$$\texttt{expand\_by\_label}\left(\left|\begin{array}{ccc|c} \mathtt{x}_1 & \ldots & \mathtt{x}_n & \\ \hline \mathtt{s}_{11} & \ldots & \mathtt{s}_{1n} & \mathtt{j}_1 \\ & \ldots & & \\ \mathtt{s}_{m1} & \ldots & \mathtt{s}_{mn} & \mathtt{j}_m \end{array}\right|, \mathtt{c}, \mathtt{()}\right) = \left|\begin{array}{ccccccc|c} \mathtt{x}_1 & \ldots & \mathtt{x}_{c-1} & & \mathtt{x}_{c+1} & \ldots & \mathtt{x}_n & \\ \hline \mathtt{s}_{k_1 1} & \ldots & \mathtt{s}_{k_1(c-1)} & & \mathtt{s}_{k_1(c+1)} & \ldots & \mathtt{s}_{k_1 n} & \mathtt{j}_{k_1} \\ & & & \ldots & & & & \\ \mathtt{s}_{k_i 1} & \ldots & \mathtt{s}_{k_i(c-1)} & & \mathtt{s}_{k_i(c+1)} & \ldots & \mathtt{s}_{k_i n} & \mathtt{j}_{k_i} \end{array}\right|$$

where $\{k_1, \ldots, k_i\} = \{k \mid \mathtt{s}_{kc} \rightarrow \mathtt{()} \in \mathbf{T}\}$

The following propositions indicate that expansion by label preserves the meaning of the input configuration. We use `var(C,c)` to denote the variable appearing in column `c` of configuration `C`.

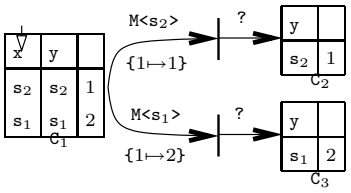
4.5.3 Proposition: If $\mathtt{C}_1 = \mathtt{expand\_by\_label(C,c,l)}$ and $\mathtt{y} = \mathtt{var(C,c)}$ and $\mathtt{(u,v)} = \mathtt{newvars\ (C,c)}$ and $E(\mathtt{y}) = \mathtt{l[w_1],w_2}$ and $E_1 = (E\backslash\mathtt{y})[\mathtt{w}_1/\mathtt{u}, \mathtt{w}_2/\mathtt{v}]$, then $E \models \mathtt{C} \Rightarrow \mathtt{k}$ iff $E_1 \models \mathtt{C}_1 \Rightarrow \mathtt{k}$.

Proof: Follows trivially from the definition of configuration by label, definition of tree automaton acceptance 4.4.1, and definition of configuration acceptance 4.5.1. □

4.5.4 Proposition: If $\mathtt{C}_1 = \mathtt{expand\_by\_label(C,c,())}$ and $\mathtt{y} = \mathtt{var(C,c)}$ and $E(\mathtt{y}) = \mathtt{()}$ and $E_1 = E\backslash\mathtt{y}$, then $E \models \mathtt{C} \Rightarrow \mathtt{k}$ iff $E_1 \models \mathtt{C}_1 \Rightarrow \mathtt{k}$.

Proof: Follows trivially from the definition of configuration by label, definition of tree automaton acceptance 4.4.1, and definition of configuration acceptance 4.5.1. □

4.5.5 Example: The second kind of expansion produces subroutine transitions and is called expansion *by state*. Consider the following configurations corresponding to states $\mathtt{q}_2$, $\mathtt{q}_3$, and $\mathtt{q}_4$ of Figure 4.8(a).



Just as in expansion by label, we must select a column whose states will serve as the basis for expansion. The first column of $\mathtt{C}_1$ was selected in this example. There are two distinct states in the first column, and, so, $\mathtt{C}_1$ is expanded two times: by $\mathtt{s}_2$ resulting in $\mathtt{C}_2$, and by $\mathtt{s}_1$ resulting in $\mathtt{C}_3$. $\mathtt{C}_2$ is obtained from $\mathtt{C}_1$ by removing the selected column and the rows whose state in the selected column is not equal to $\mathtt{s}_2$. $\mathtt{C}_3$ is produced similarly. In the generated transitions, $\mathtt{M<s_1>}$ and $\mathtt{M<s_2>}$ denote subroutine matching automata corresponding to states $\mathtt{s}_1$ and $\mathtt{s}_2$. The compilation algorithm constructs these automata using initial single-state configurations ($\mathtt{<x>}$,$\{(\mathtt{s}_1, 1)\}$) and ($\mathtt{<x>}$,$\{(\mathtt{s}_2, 1)\}$) respectively.

Expansion by state is formalized by the following function that generates a residual configuration given the current configuration, a column, and a tree automaton state:

$$\texttt{expand\_by\_state}\left(\left|\begin{array}{ccc|c} \mathtt{x}_1 & \ldots & \mathtt{x}_n & \\ \hline \mathtt{s}_{11} & \ldots & \mathtt{s}_{1n} & \mathtt{j}_1 \\ & \ldots & & \\ \mathtt{s}_{m1} & \ldots & \mathtt{s}_{mn} & \mathtt{j}_m \end{array}\right|, \mathtt{c}, \mathtt{s}\right) = \left|\begin{array}{ccccccc|c} \mathtt{x}_1 & \ldots & \mathtt{x}_{c-1} & & \mathtt{x}_{c+1} & \ldots & \mathtt{x}_n & \\ \hline \mathtt{s}_{k_1 1} & \ldots & \mathtt{s}_{k_1(c-1)} & & \mathtt{s}_{k_1(c+1)} & \ldots & \mathtt{s}_{k_1 n} & \mathtt{j}_{k_1} \\ & & & \ldots & & & & \\ \mathtt{s}_{k_i 1} & \ldots & \mathtt{s}_{k_i(c-1)} & & \mathtt{s}_{k_i(c+1)} & \ldots & \mathtt{s}_{k_i n} & \mathtt{j}_{k_i} \end{array}\right|$$

where $\{k_1, \ldots, k_i\} = \{k \mid \mathtt{s}_{kc} = \mathtt{s}\}$.

4.5.6 Proposition: If $\mathtt{y} = \mathtt{var(C,c)}$ and $E_1 = E \backslash \mathtt{y}$ and $L = \mathtt{column\_states(C,c)}$, then $E \models \mathtt{C} \Rightarrow \mathtt{k}$ iff $\exists \mathtt{s} \in L.\ E_1 \models \mathtt{expand\_by\_state(C,c,s)} \Rightarrow \mathtt{k}$ and $E(\mathtt{y}) \in \mathtt{s}$.

Proof: Follows trivially from the definition of configuration by state and definition of configuration acceptance 4.5.1. □

4.5.3 Loop Breakers

To help us determine whether a configuration should be expanded by label (as in Example 4.5.2) or by state (as in Example 4.5.5), we introduce the following concept.

4.5.7 Definition: Let $\mathtt{A} = \mathtt{(S,T)}$ be a tree automaton. We say that $\mathtt{Rec} \subseteq \mathtt{S}$ is a *set of loop breakers* for $\mathtt{A}$ if removing the transitions originating in $\mathtt{Rec}$ ensures that the remaining transition relation is acyclic.

The initial configuration that corresponds to the start state of the matching automaton is always expanded by label. A non-initial configuration is expanded by state if all of its columns contain a loop breaker. If a non-initial configuration contains columns that have no loop breakers, one of such columns is selected for expansion and the configuration is expanded by label. This strategy ensures that the compilation algorithm terminates. The goal of the initial configuration rule is to prevent generating a non-terminating matching automaton that calls itself (or some other automaton) recursively without making any progress.

The above definition specifies a necessary condition for a set of loop breakers, but does not tell us how to compute it. Let us consider several alternatives. The first one is the set of all states of a tree automaton; it is the *maximal* set of loop breakers. If the compilation algorithm uses the maximal set of loop breakers, the size of the generated matching automaton is guaranteed to be at worst linear in the size of the input tree automaton. The disadvantage of this approach is

that it generates too many subroutine transitions resulting in more backtracking and extra cost of subroutine invocation.

At the other extreme are *minimal* sets of loop breakers. They lead to more efficient matching automata by minimizing the number of subroutine transitions as shown in Example 4.4.7. Example 4.4.9 demonstrates, however, that minimal sets of loop breakers can result in a matching automaton with a double exponential number of states.

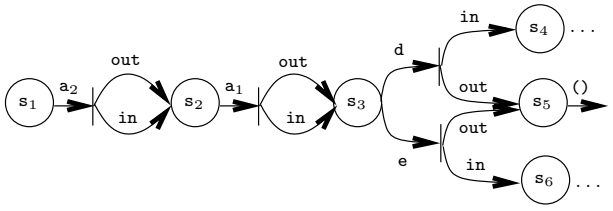


Figure 4.13: Tree Automaton for Example 4.4.9 that may lead to a space explosion

To understand what causes the double exponential blowup, consider Figure 4.13 that shows the tree automaton associated with Example 4.4.9. Observe, that the transitions originating in $\mathtt{s}_1$ and $\mathtt{s}_2$ duplicate the destination state and that $\mathtt{s}_3$ has two outgoing transitions. Both these factors contribute to the blowup.

In view of this, we consider two other approaches to computing the set of loop breakers: a *multiple predecessor* set of loop breakers is the union of a minimal set and the set of states with multiple incoming transitions (such as $\mathtt{s}_2$ and $\mathtt{s}_3$); a *multiple successor* set of loop breakers is the union of a minimal set and the set of states with multiple outgoing transitions (such as $\mathtt{s}_3$). Using multiple successor or multiple predecessor loop breakers ensures that the size of the generated automaton is at worst exponential or polynomial respectively. For the tree automaton of Figure 4.13, for example, the multiple successor set of loop breakers $\{\mathtt{s}_3, \mathtt{s}_4, \mathtt{s}_6\}$ leads to the matching automaton shown in Figure 4.12(b).

Our current compiler implementation uses the same loop breaker set throughout compilation. Tests have shown that using either multiple predecessor or multiple successor sets of loop breakers results in satisfactory target programs that are almost as fast as those generated with the minimal set of loop breakers and almost as small as those generated with the maximal set of loop breakers. In the future, we would like to consider an adaptive strategy that starts with the minimal set of loop breakers and switches to a more size conscious set of loop breakers if the generated program exceeds a certain size threshold.

4.5.4 Optimizing Configurations

Depending on whether the input `match` expression is exhaustive or not, the `sbf` operates in either the exhaustive or non-exhaustive mode. In the exhaustive mode, `sbf` can take advantage of the fact that only values that match one of the alternatives can be given as input to the generated matching automaton. The first exhaustiveness optimization simplifies configurations all of whose rows contain the same index into zero-column configurations. Consider, for instance, $\mathtt{C}_3$ and $\mathtt{C}_4$ of Example 4.5.2. For any satisfying environments, these configurations are equivalent to the zero-column configurations (<>,{1}) and (<>,{2}) respectively. Using this optimization, we can finalize the matching automaton generated in Example 4.5.2 by making the transitions originating in $\mathtt{C}_2$ final (c.f. the matching automaton of Figure 4.9(c)).

Another exhaustive mode optimization involves eliminating columns containing the same state. Again we refer to configurations $\mathtt{C}_3$ and $\mathtt{C}_4$ of Example 4.5.2. These configurations are subject to the described optimization since they have only one row. Removing the columns will result in the same zero-column configurations that we obtained by applying the optimization described in the previous paragraph.

```
fun sbf_simple(C,A,Rec,exh) =
 let c = simple_col(C)
 let L = labels(C,c)
 for l ∈ L do
  let C_l = expand_by_label(C,c,l)
  let I_l = indices(C_l)
  if width(C_l) = 0 then
   let M_l = ({C},C,0)
   let r_l = C : () --I_l--> ∅
  else
   let (M_l,y_l) = sbf(C_l,A,Rec,exh)
   if l = () then
    let r_l = C : () --I_l--> {y_l ∈ sstate(M_l)}
   else
    let (u,v) = newvars(C,c)
    let r_l = C : l[u],v --I_l--> {y_l ∈ sstate(M_l)}
 return (⋃_{l∈L} C --r_l--> M_l, var(C,c))
```

```
fun sbf_sub(C,A,Rec,exh) =
 let S = column_states(C,1)
 for s ∈ S do
  let C_s = expand_by_state(C,1,s)
  let σ_s = {(1,j) | j ∈ indices(C_s)}
  if width(C_s) = 0 then
   let M_s = ({C},C,0)
   let r_s = C : M<s> --σ_s--> ∅
  else
   let (M_s,y_s) = sbf_sub(C_s,A,Rec,exh)
   let r_s = C : M<s> --σ_s--> {y_s ∈ sstate(M_s)}
 return (⋃_{s∈S} C --r_s--> M_s, var(C,1))

fun sbf(C,A,Rec,exh) =
 let C = optimize(C,exh)
 if ∃c. Rec ∩ column_states(C,c) = 0 then
  return sbf_simple(C,A,Rec,exh)
 else return sbf_sub(C,A,Rec,exh)
```

Figure 4.14: Functions of the SBF compilation algorithm

4.5.5 Formalization of the SBF Algorithm

The SBF algorithm employs three functions `sbf_simple`, `sbf_sub`, and `sbf` displayed in Figure 4.14. All three functions take an input configuration, a description of the tree automaton on which the configuration is based, a set of loop breakers, and a boolean exhaustiveness flag and generate a matching automaton whose semantics is equivalent to the input configuration and the variable inspected by the start state of the matching automaton.

Functions `sbf_simple` and `sbf_sub` generate a matching automaton whose start state is a simple state and a subroutine state respectively; `sbf` determines whether a simple state can be generated from a given configuration and depending on this invokes either `sbf_simple` or `sbf_sub`.

The following helper functions are used: `simple_col` takes a configuration and returns a column that has no loop breakers; `labels` takes a configuration and a column and returns all the labels that mark tree automaton transitions originating in the tree automaton states of the column; `indices` returns the indices of a given configuration; `width` denotes the number of columns in a given configuration; `sstate` returns the start state of a given matching automaton; `var` takes a configuration and a column and returns the variable corresponding to the column, and `column_states` returns the set of states appearing in a given column.

Function `sbf_simple` goes through the following steps. It starts by finding a column that does not have loop breakers. It then expands the current configuration by label using the label that occur in the transitions originating in the states of the selected column. It converts the obtained residual configurations to corresponding matching automata by making recursive calls to `sbf`. After this, `sbf_simple` generates matching automaton transitions from the current state to the start states of these matching automata. In the last line, `sbf_simple` puts everything together by combinging the transitions generated in the previous step, the matching automata generated by the recursive calls, and the current state into a resulting matching automaton denoted by $\bigcup_{\mathtt{l}\in\mathtt{L}} \mathtt{C} \xrightarrow{\mathtt{r}_l} \mathtt{M}_l$.

Function `sbf_sub` works similarly except that uses expansion by state instead of expansion by label. `M<s>` denotes a subroutine matching automaton built from an initial configuration containing one tree automaton state `s`.

At the top level, the compilation algorithm proceeds as follows. It first creates a sequence of states corresponding to each `match` expression of the source program. Let us denote this collection of sequences $\overline{\mathtt{s}_1}$, ..., $\overline{\mathtt{s}_j}$. Let $\mathtt{f}_1 \ldots \mathtt{f}_j$ be boolean flags indicating whether the corresponding `match` expressions are exhaustive or not. Additionally, to ensure that there is a subroutine automaton for any generated subroutine transition, the algorithm also creates a singleton sequence for every state of the input tree automaton. Let us denote the collection of these singleton sequences $\overline{\mathtt{s}_{j+1}}$, ..., $\overline{\mathtt{s}_p}$.

The algorithm then constructs an initial configuration for both types of sequences and invokes `sbf_simple` on the obtained configurations thus producing the resulting collection of matching automata.

To formalize these steps, let us introduce a function that builds an initial configuration from a sequence of states: `config(<`$\mathtt{s}_1,\ldots,\mathtt{s}_m$`>)` $= (\mathtt{<x>}, \{(\mathtt{s}_1, 1), \ldots, (\mathtt{s}_m, m)\})$. Let `A` be the input tree automaton, and let `Rec` be a set of loop breakers for `A`. Then, the top level of SBF can be described by the following equations.

$$
\begin{array}{lcl}
(\mathtt{M}\overline{\mathtt{s}_1}\mathtt{,\_}) & = & \mathtt{sbf\_simple(config(}\overline{\mathtt{s}_1}\mathtt{),A,Rec,f}_1\mathtt{)} \\
\ldots & & \\
(\mathtt{M}\overline{\mathtt{s}_j}\mathtt{,\_}) & = & \mathtt{sbf\_simple(config(}\overline{\mathtt{s}_j}\mathtt{),A,Rec,f}_j\mathtt{)} \\
(\mathtt{M}\overline{\mathtt{s}_{j+1}}\mathtt{,\_}) & = & \mathtt{sbf\_simple(config(}\overline{\mathtt{s}_{j+1}}\mathtt{),A,Rec,false)} \\
\ldots & & \\
(\mathtt{M}\overline{\mathtt{s}_p}\mathtt{,\_}) & = & \mathtt{sbf\_simple(config(}\overline{\mathtt{s}_p}\mathtt{),A,Rec,false)}
\end{array}
$$

Note that acceptors must be generated in the non-exhaustive mode since their goal is to check whether or not an *arbitrary* input value matches the underlying tree automaton state.

Also, observe that the use of `sbf_simple` in the above equations ensures that the initial configurations are expanded by label. As we have mentioned above, this is necessary to avoid generating non-terminating matching automata.

We now describe the important properties of the SBF algorithm. The first lemma states that the algorithm terminates.

4.5.8 Lemma: Let `A = (S,T)` be a tree automaton and let `Rec` $\subseteq$ `S` be a loop breaker set. Then `sbf_simple`, `sbf_sub`, and `sbf` terminate on arguments `C`, `A`, `Rec`, and `exh` for any configuration `C` over `S` and any boolean value `exh`.

Proof: Let the *height* of a tree automaton state `s` $\in$ `S` be calculated as follows. If `s` $\in$ `Rec` or `s` is a final state, then the height of `s` is 1. Otherwise, it is $1 + max(\{h_1 + h_2 \mid \exists \mathtt{l}.\ \mathtt{s} \rightarrow \mathtt{l[s}_1\mathtt{],s}_2 \in \mathtt{R}$ and h_i is the height of $\mathtt{s}_i\})$. Let the height of a column in a configuration be the maximum height of the column's states, and let the height of a configuration be the sum of the heights of its columns. Since `Rec` is a set of loop breakers for `S`, height is well defined for all states in `S`—see Definition 4.5.7—and consequently for all configurations over `S`. From the definitions of `expand_by_label` and `expand_by_state`, we can see that the height of the residual configuration is smaller than the height of the original configuration. Therefore, the height of the current configuration can be used as a termination measure that decreases as the algorithm makes recursive calls in the bodies of `sbf_simple` and `sbf_sub`. □

The next lemma establishes that the matching automaton generated by `sbf` is indeed in simple backtracking form.

4.5.9 Lemma: Let `A` = `(S,T)` be a tree automaton; let `Rec` $\subseteq$ `S` be a loop breaker set, and let `(M,y)` be the result of applying `sbf_simple`, `sbf_sub`, or `sbf` to the arguments `C`, `A`, `Rec`, and `exh` where `C` is a configuration over `S` and `exh` is a boolean value. Then `M` is in SBF.

Proof: The fact that `M` is sequential and disjoint and its transitions are separated and staged is evident from observing the algorithm. We can also show that transitions of `M` only involve `indices(C)`; therefore, all subroutine automata are boolean since they are obtained from singleton configurations. □

To prove that the SBF algorithm is correct, we introduce a collection of inference rules defining a syntactic acceptance relation between environments, configurations and integers and prove that it is equivalent to the semantic satisfaction relation introduced above. The inference rules will have a structure similar to the structure of the algorithm. This will lead to a straight forward statements and proofs of the correctness property that will relate the algorithm and the inference rules.

Figure 4.15 defines three mutually recursive judgments that correspond to the three procedures of the forthcoming algorithm. The judgment $E \vdash_{sim} \texttt{C} \Rightarrow i$ is based on `expand_by_label` and normally is applicable when the underlying configuration contains a non-recursive column. (The one exception to this is in the rule SubState where this judgment is invoked regardless of the configuration's shape.) The judgment $E \vdash_{sub} \texttt{C} \Rightarrow i$ is based on `expand_by_state` and is applied to configurations all of whose columns contain recursive states. The judgment $E \vdash \texttt{C} \Rightarrow i$ combines the other judgments by invoking one or the other depending on the shape of the configuration.

The rule SimLab is applicable when the variable of the selected column is bound to a non-empty sequence. The rule's conclusion is satisfied if the assertion involving the residual configuration, computed by `expand_by_label`, is satisfied in the environment in which the selected variable is replaced by the pair of variables generated by `newvars`. Notice that `expand_by_label` also uses `newvars` thus ensuring that the variables added to the environment are the same as the variables added to the residual configuration.

If the selected variable is bound to the empty sequence `()`, the rule SimEmp applies. It is similar to the previous rule except that it does not add new bindings to the environment and, correspondingly, its residual configuration contains one fewer column (instead of one more) than the configuration in the conclusion.

The rule SubZero is used for configurations with no columns. Such configurations are satisfied in any environment by any of their indices. The conclusion of the other subroutine rule SubState

$$\frac{\texttt{c} = \texttt{simple\_col(C)} \qquad \texttt{y} = \texttt{var(C,c)} \qquad \texttt{(u,v)} = \texttt{newvars(C,c)} \qquad E(\texttt{y}) = \texttt{l[w}_1\texttt{],w}_2 \qquad (E\backslash\texttt{y})[\texttt{w}_1/\texttt{u}, \texttt{w}_2/\texttt{v}] \vdash \texttt{expand\_by\_label(C,c,l)} \Rightarrow \texttt{k}}{E \vdash_{sim} \texttt{C} \Rightarrow \texttt{k}} \quad \text{(SimLab)}$$

$$\frac{\texttt{c} = \texttt{simple\_col(C)} \qquad \texttt{y} = \texttt{var(C,c)} \qquad E(\texttt{y}) = \texttt{()} \qquad E\backslash\texttt{y} \vdash \texttt{expand\_by\_label(C,c,())} \Rightarrow \texttt{k}}{E \vdash_{sim} \texttt{C} \Rightarrow \texttt{k}} \quad \text{(SimEmp)}$$

$$\frac{\texttt{width(C)} = 0 \qquad \texttt{k} \in \texttt{indices(C)}}{E \vdash_{sub} \texttt{C} \Rightarrow \texttt{k}} \quad \text{(SubZero)}$$

$$\frac{\texttt{s} \in \texttt{column\_states(C,1)} \qquad \texttt{y} = \texttt{var(C,1)} \qquad \emptyset[E(\texttt{y})/\texttt{x}] \vdash_{sim} \texttt{config(<s>)} \Rightarrow 1 \qquad E\backslash\texttt{y} \vdash_{sub} \texttt{expand\_by\_state(C,1,s)} \Rightarrow \texttt{k}}{E \vdash_{sub} \texttt{C} \Rightarrow \texttt{k}} \quad \text{(SubState)}$$

$$\frac{\exists \texttt{c} \in \{1, \ldots, \texttt{width(C)}\}.\ \texttt{Rec} \cap \texttt{column\_states(C,c)} = \emptyset \qquad E \vdash_{sim} \texttt{C} \Rightarrow \texttt{k}}{E \vdash \texttt{C} \Rightarrow \texttt{k}} \quad \text{(Sim)}$$

$$\frac{\forall \texttt{c} \in \{1, \ldots, \texttt{width(C)}\}.\ \texttt{Rec} \cap \texttt{column\_states(C,c)} \neq \emptyset \qquad E \vdash_{sub} \texttt{C} \Rightarrow \texttt{k}}{E \vdash \texttt{C} \Rightarrow \texttt{k}} \quad \text{(Sub)}$$

Figure 4.15: Syntactic Satisfaction

is satisfied if, given an arbitrary state from the configuration's first column `s`, the two premises are satisfied. The first assertion contains the environment (without the binding for `y`, the variable of the first column) and the residual configuration generated after expanding by `s`. The other assertion checks whether `s` accepts the value `v` to which the selected variable `y` is bound. This assertion is composed of a new environment that binds `x` to `v` and a new configuration generated by `config`. Note that the new environment is created with a single binding for the variable `x` which is also the variable used by `config` in the initial configuration. Also, observe that the environment used in the conclusion of SubState binds the same value as the environment created for the second premise. Therefore, to avoid infinite derivation branches consisting of instances of SubState, we force this premise to use the $\vdash_{sim}$ judgment regardless of whether the created configuration is recursive or not. This ensures progress since the $\vdash_{sim}$ rules always reduce the size of values in the environment.

The combined judgment $\vdash$ is used in SimLab and is defined by the rules Sim and Sub which check whether the given configuration contains a column with non-recursive states or not and invoke the $\vdash_{sim}$ or $\vdash_{sub}$ judgments respectively.

The following lemma shows that the introduced syntactic relations are sound with respect to the semantic satisfaction relation.

4.5.10 Lemma: If $E \vdash_{sim} \mathtt{C} \Rightarrow \mathtt{k}$ or $E \vdash_{sub} \mathtt{C} \Rightarrow \mathtt{k}$ or $E \vdash \mathtt{C} \Rightarrow \mathtt{k}$, then $E \models \mathtt{C} \Rightarrow \mathtt{k}$.

Proof: The proof proceeds by simultaneous induction on derivations of the syntactic judgments.

Case SimLab**:**

From the premises of the rule, we have the following assertions: `c = simple_col(C)` and `y = var(C,c)` and `u,v = newvars(C,c)` and $E(\mathtt{y}) = \mathtt{l[w_1],w_2}$ and $E_1 = (E \backslash \mathtt{y})[\mathtt{w_1/u}, \mathtt{w_2/v}]$ and $\mathtt{C_1} = \mathtt{expand\_by\_label(C,c,l)}$ and $E_1 \vdash \mathtt{C_1} \Rightarrow \mathtt{k}$. By the induction hypothesis, they imply $E_1 \models \mathtt{C_1} \Rightarrow \mathtt{k}$. It follows by Proposition 4.5.3 that $E \models \mathtt{C} \Rightarrow \mathtt{k}$.

Case SimEmp**:**

Similar to SimLab

Case SubZero**:**

From the premises of the rule, we have $\mathtt{width(C)} = 0$ and $\mathtt{k} \in \mathtt{indices(C)}$. The result $E \models \mathtt{C} \Rightarrow \mathtt{k}$ follows by the definition of semantic satisfaction.

Case SubState**:**

From the premises of the rule, we have $\mathtt{s} \in \mathtt{column\_states(C,1)}$ and $\mathtt{y} = \mathtt{var(C,1)}$ and $E_0 = \emptyset[E(\mathtt{y})/\mathtt{x}]$ and $E_1 = E \backslash \mathtt{y}$ and $\mathtt{C_1} = \mathtt{expand\_by\_state(C,1,s)}$ and $E_0 \vdash_{sim} \mathtt{config(<s>)} \Rightarrow 1$ and $E_1 \vdash_{sub} \mathtt{C_1} \Rightarrow \mathtt{k}$. Applying the induction hypothesis twice, we obtain $E_0 \models \mathtt{config(<s>)} \Rightarrow 1$ and $E_1 \models \mathtt{C_1} \Rightarrow \mathtt{k}$. From the definition of configuration acceptance, it then follows that $E(y) \in \mathtt{s}$ which implies $E \models \mathtt{C} \Rightarrow \mathtt{k}$ by Proposition 4.5.6.

Case Sim**:**

From the premises of the rule, we have $E \vdash_{sim} \mathtt{C} \Rightarrow \mathtt{k}$. By the induction hypothesis, $E \models \mathtt{C} \Rightarrow \mathtt{k}$.

Case Sub**:**

Similar to Sim. □

Now, we will prove that the syntactic satisfaction relations are complete with respect to the semantic satisfaction relation.

4.5.11 Lemma: If $E \models \mathtt{C} \Rightarrow \mathtt{k}$ then $E \vdash_{sub} \mathtt{C} \Rightarrow \mathtt{k}$ and $E \vdash \mathtt{C} \Rightarrow \mathtt{k}$. Furthermore, if $\mathtt{width(C)} \neq 0$, then $E \vdash_{sim} \mathtt{C} \Rightarrow \mathtt{k}$ also.

Proof: The proof proceeds by induction on the size of values in E. We first prove the statement for the $\vdash_{sim}$ relation, then use it to prove the statement for the $\vdash_{sub}$ relation, and finally, use both of these results to prove the statement for the combined relation $\vdash$.

Case: Assume that `width(C)` $\neq 0$ and let `c` = `simple_col(C)`; let `y` = `var(C,c)`; let E(`y`) = `l[`$\mathtt{w}_1$`],`$\mathtt{w}_2$; let `(u,v)` = `newvars(C,c)`; let $E_1 = (E\backslash \mathtt{y})[\mathtt{w}_1/\mathtt{u}, \mathtt{w}_2/\mathtt{v}]$, and let $\mathtt{C}_1$ = `expand_by_label (C,c,l)`.

By Proposition 4.5.3, we have $E_1 \models \mathtt{C}_1 \Rightarrow \mathtt{k}$. Since E_1 was obtained by replacing a binding with `l[`$\mathtt{w}_1$`],`$\mathtt{w}_2$ by two bindings with strictly smaller values $\mathtt{w}_1$ and $\mathtt{w}_2$, we can apply the induction hypothesis which results in $E_1 \vdash \mathtt{C}_1 \Rightarrow \mathtt{k}$. Now, by SimLab we have $E \vdash_{sim} \mathtt{C} \Rightarrow \mathtt{k}$.

Case: Conversely, assume that `width(C)` $\neq 0$ and `c` = `simple_col(C)` and `y` = `var(C,c)` and E(`y`) = `()` and $E_1 = E\backslash \mathtt{y}$ and $\mathtt{C}_1$ = `expand_by_label(C,c,())`.

By Proposition 4.5.4, we have $E_1 \models \mathtt{C}_1 \Rightarrow \mathtt{k}$. Since we obtained E_1 by eliminating a binding from E, we can apply the induction hypothesis which results in $E_1 \vdash \mathtt{C}_1 \Rightarrow \mathtt{k}$. Now, by SimEmp we have $E \vdash_{sim} \mathtt{C} \Rightarrow \mathtt{k}$.

Case: Assume `width(C)` $= 0$.

By the definition of semantic acceptance it must be the case that `k` $\in$ `indices(C)`. Then, by SubZero, $E_1 \vdash_{sub} \mathtt{C}_1 \Rightarrow \mathtt{k}$.

Case: Conversely, assume that `width(C)` $\neq 0$ and `y` = `var(C,1)` and $E_1 = E\backslash \mathtt{y}$.

By Proposition 4.5.6, there exists a state `s` $\in$ `column_states(C,1)` such that $E_1 \models$ `expand_by_state (C,1,s)` $\Rightarrow$ `k` and E(`y`) $\in$ `s`. The latter and the definition of configuration acceptance imply that $E_0 \models$ `config(<s>)` $\Rightarrow 1$ where $E_0 = 0[E(\mathtt{y})/\mathtt{x}]$. By the induction hypothesis, since E_1 is smaller than E, we have $E_1 \vdash_{sub}$ `expand_by_state(C,1,s)` $\Rightarrow$ `k`. From the result for the $\vdash_{sim}$ relation proved in the first two cases, we have $E_0 \vdash_{sim}$ `config(<s>)` $\Rightarrow 1$. Therefore, by SubState, $E \vdash_{sub} \mathtt{C} \Rightarrow \mathtt{k}$.

Case: Assume $\exists \mathtt{c} \in \{1, \ldots,$ `width(C)`$\}$. `Rec` $\cap$ `column_states(C,c)` $= \emptyset$.

From the result for the $\vdash_{sim}$ relation proved in the first two cases, we have $E \vdash_{sim} \mathtt{C} \Rightarrow 1$. Hence by Sim, $E \vdash \mathtt{C} \Rightarrow 1$.

Case: Conversely, assume $\forall \mathtt{c} \in \{1, \ldots,$ `width(C)`$\}$. `Rec` $\cap$ `column_states(C,c)` $\neq \emptyset$.

From the result for the $\vdash_{sub}$ relation proved in the third and fourth cases, we have $E \vdash_{sub} \mathtt{C} \Rightarrow 1$. Hence by Sub, $E \vdash \mathtt{C} \Rightarrow 1$. □

Observe the importance of using $\vdash_{sim}$ instead of $\vdash$ in the second premise of SubState. This allows our inductive argument to go through by proving the statement for $\vdash_{sim}$ first and using it in the proof of the statement for $\vdash_{sub}$. This arrangement is what ensures termination of the syntactic acceptance relation and consequently termination of the automaton generated by the algorithm of the following subsection.

We now show that the algorithm coincides with the syntactic satisfaction relation. First, we prove a technical proposition that establishes that the automata built by recursive invocations of `sbf` and `sbf_sub` are disjoint as specified by the following definition.

4.5.12 Definition: Automata $\mathtt{A} = (\mathtt{Q}_1, \mathtt{q}_1, \mathtt{R}_1)$ and $\mathtt{B} = (\mathtt{Q}_2, \mathtt{q}_2, \mathtt{R}_2)$ are **disjoint** if for any state $\mathtt{q}$ and transition $\mathtt{r}$ such that $\mathtt{q} \in \mathtt{Q}_1$ and $\mathtt{q} \in \mathtt{Q}_2$ and $\mathtt{r}$ originates in $\mathtt{q}$, it is the case that $\mathtt{r} \in \mathtt{R}_1$ iff $\mathtt{r} \in \mathtt{R}_2$.

4.5.13 Proposition: Given disjoint automata $\mathtt{A}_1 = (\mathtt{Q}_1, \mathtt{q}_1, \mathtt{R}_1)$ and $\mathtt{A}_2 = (\mathtt{Q}_2, \mathtt{q}_2, \mathtt{R}_2)$, let $\mathtt{B} = (\{\mathtt{q}_0\} \cup \mathtt{Q}_1 \cup \mathtt{Q}_2, \mathtt{q}_0, \mathtt{R}_0 \cup \mathtt{R}_1 \cup \mathtt{R}_2)$ where $\mathtt{q}_0 \notin \mathtt{Q}_1 \cup \mathtt{Q}_2$ and $\mathtt{R}_0$ contains only transitions originating in $\mathtt{q}_0$. For any environment E, value $\mathtt{v}$, index $\mathtt{k}$, and $i \in \{1, 2\}$, if $\mathtt{q} \in \mathtt{Q}_i$ and $E \vdash \mathtt{v} \in \mathtt{q(B)} \Rightarrow \mathtt{k}$, then $E \vdash \mathtt{v} \in \mathtt{q}(\mathtt{A}_i) \Rightarrow \mathtt{k}$.

Proof: Straight forward induction on the acceptance derivation for $\mathtt{B}$. □

The following lemma states the algorithm's correctness property by associating the functions of Figure 4.14 with the syntactic relations of the previous subsection.

4.5.14 Lemma: Let $\mathtt{A} = (\mathtt{S}, \mathtt{T})$ be a tree automaton; let $\mathtt{Rec} \subseteq \mathtt{S}$ be a loop breaker set; let E be an environment, and let $\mathtt{C}$ be a configuration over $\mathtt{S}$ with at least one column. Then, if $\mathtt{exh} = \mathtt{true}$ and $\mathtt{C}$ is satisfiable by E or if $\mathtt{exh} = \mathtt{false}$, we have

- if $(\mathtt{M}, \mathtt{y}) = \mathtt{sbf\_simple(C,A,Rec,exh)}$, then $E \vdash_{sim} \mathtt{C} \Rightarrow \mathtt{k}$ iff $E \backslash \mathtt{y} \vdash E(\mathtt{y}) \in \mathtt{M} \Rightarrow \mathtt{k}$
- if $(\mathtt{M}, \mathtt{y}) = \mathtt{sbf\_sub(C,A,Rec,exh)}$, then $E \vdash_{sub} \mathtt{C} \Rightarrow \mathtt{k}$ iff $E \backslash \mathtt{y} \vdash E(\mathtt{y}) \in \mathtt{M} \Rightarrow \mathtt{k}$
- if $(\mathtt{M}, \mathtt{y}) = \mathtt{sbf\_simple(C,A,Rec,exh)}$, then $E \vdash \mathtt{C} \Rightarrow \mathtt{k}$ iff $E \backslash \mathtt{y} \vdash E(\mathtt{y}) \in \mathtt{M} \Rightarrow \mathtt{k}$

Proof: We prove the forward direction of the lemma by induction on the size of the derivations of the left hand side judgments.

Case: Assume $(\mathtt{M}, \mathtt{y}) = \mathtt{sbf\_simple(C,A,Rec,exh)}$ and $E \vdash_{sim} \mathtt{C} \Rightarrow \mathtt{k}$.

Either the premises of SimLab or the premises of SimEmp must hold.

Subcase: Consider the former

Let $\mathtt{c} = \mathtt{simple\_col(C)}$, let $\mathtt{y} = \mathtt{var(C,c)}$, let $(\mathtt{u}, \mathtt{v}) = \mathtt{newvars(C,c)}$, let $E(\mathtt{y}) = \mathtt{l[w1], w2}$, let $E_l = (E \backslash \mathtt{y})[\mathtt{w}_1/\mathtt{u}, \mathtt{w}_2/\mathtt{v}]$, and let $\mathtt{C}_l = \mathtt{expand\_by\_label(C,c,l)}$. Then, $E_l \vdash \mathtt{C}_l \Rightarrow \mathtt{k}$. Let $\mathtt{I}_l = \mathtt{indices}(\mathtt{C}_l)$. By Proposition 4.5.3, we have $E \models \mathtt{C} \Rightarrow \mathtt{k}$. For this to take place, we must have $\mathtt{l} \in$

`labels(C,c)`. It cannot be the case that `width(C`$_l$`)` $= 0$ since `l` is a binary label and by definition of `expand_by_label`, `width(C`$_l$`)` $>$ `width(C)` ≥ 0. Under these conditions, `sbf_simple` executes code `(M`$_l$`,y`$_l$`)` = `sbf(C`$_l$`,A,Rec,exh)`, creates a transition `r`$_l$ $=$ `C : l[u],v` $\xrightarrow{\texttt{I}_l}$ $\{$`y`$_l \in$ `sstate(M`$_l$`)`$\}$, and ensures that `M` includes `C` as the start state, `r`$_l$ among its transitions, and all the states and transitions of `M`$_l$. By the induction hypothesis, $E_l \vdash$ `C`$_l \Rightarrow$ `k` implies $E_l\backslash$`y`$_l \vdash E_l($`y`$_l) \in$ `M`$_l \Rightarrow$ `k`, and, hence, it must be the case that $E_l\backslash$`y`$_l \vdash E_l($`y`$_l) \in$ `sstate(M`$_l$`)` $\Rightarrow$ `k`. By MA-Lab, we can conclude $E\backslash$`y` $\vdash E($`y`$) \in$ `C` $\Rightarrow$ `k`, and, so, $E\backslash$`y` $\vdash E($`y`$) \in$ `M` $\Rightarrow$ `k`.

Subcase: Otherwise, SimEmp is applicable

Let `c` = `simple_col(C)`, let `y` = `var(C,c)`, let `C`$_l$ = `expand_by_label(C,c,())`, and let $E($`y`$)$ = `()`, let $E_l = E\backslash$`y`. Then, $E_l \vdash$ `C`$_l \Rightarrow$ `k`. Let `I`$_l$ = `indices(C`$_l$`)`. By Proposition 4.5.3, we have $E \models$ `C` $\Rightarrow$ `k`. For this to take place, we must have `()` $\in$ `labels(C,c)`. Suppose that `width(C`$_l$`) = 0`. Then, $E_l \vdash$ `C`$_l \Rightarrow$ `k` implies $E_l \vdash_{sub}$ `C`$_l \Rightarrow$ `k` by inversion of $\vdash$ and, by inversion of $\vdash_{sub}$, we have `k` $\in$ `I`$_l$. Under these conditions, `M` must include the start state `C` and a transition `r`$_l$ $=$ `C : ()` $\xrightarrow{\texttt{I}_l}$ $\emptyset$. By MA-Emp, we can conclude $E\backslash$`y` $\vdash E($`y`$) \in$ `C` $\Rightarrow$ `k`, and, so, $E\backslash$`y` $\vdash E($`y`$) \in$ `M` $\Rightarrow$ `k`. Otherwise, `width(C`$_l$`)` $\neq 0$. In this case, `sbf_simple` generates `(M`$_l$`,y`$_l$`)` = `sbf(C`$_l$`,A,Rec,exh)`, a transition `r`$_l$ $=$ `C : ()` $\xrightarrow{\texttt{I}_l}$ $\{$`y`$_l \in$ `sstate(M`$_l$`)`$\}$, and `M` must contain `C` as the start state, `r`$_l$ among its transitions, and all the states and transitions of `M`$_l$. By the induction hypothesis, $E_l \vdash$ `C`$_l \Rightarrow$ `k` implies $E_l\backslash$`y`$_l \vdash$ $E_l($`y`$_l) \in$ `M`$_l \Rightarrow$ `k`, and, hence, it must be the case that $E_l\backslash$`y`$_l \vdash E_l($`y`$_l) \in$ `sstate(M`$_l$`)` $\Rightarrow$ `k`. By MA-Lab, we can conclude $E\backslash$`y` $\vdash E($`y`$) \in$ `C` $\Rightarrow$ `k`, and, so, $E\backslash$`y` $\vdash E($`y`$) \in$ `M` $\Rightarrow$ `k`.

Case: Assume `(M,y)` = `sbf_sub(C,A,Rec,exh)` and $E \vdash_{sub}$ `C` $\Rightarrow$ `k`.

The $\vdash_{sub}$ assertion could only be derived by SubState. Let `s` $\in$ `column_states(C,1)`, let `y` $=$ `var(C,1)`, let $E_0 = \emptyset[E($`y`$/$`x`$]$, let $E_s = E\backslash$`y`, and let `C`$_s$ $=$ `expand_by_state(C,1,s)`. We must have $E_0 \vdash_{sim}$ `config(<s>)` $\Rightarrow 1$ and $E_s \vdash_{sub}$ `C`$_s \Rightarrow$ `k`. Let `I`$_s$ = `indices(C`$_s$`)`. Since `(M<s>,z)` = `sbf_simple(config<s>,A,Rec,false)` where `z` = `x` (since `x` is the variable of the only column in `config<s>`), by the induction hypothesis, $E_0 \vdash_{sim}$ `config<s>` $\Rightarrow 1$ implies $\emptyset \vdash E($`y`$) \in$ `M<s>` $\Rightarrow 1$.

Subcase: Suppose that `width(C`$_s$`)` $= 0$.

By inversion of $\vdash_{sub}$, we have `k` $\in$ `I`$_s$. Under these conditions, `M` must include the start state `C` and a transition `r`$_l$ $=$ `C : M<s>` $\xrightarrow{\sigma_s}$ $\emptyset$ where σ_s $=$ `{(1,j) | j` $\in$ `I`$_s$`}`. By MA-Sub, we can conclude $E\backslash$`y` $\vdash E($`y`$) \in$ `C` $\Rightarrow$ `k`, and, so, $E\backslash$`y` $\vdash E($`y`$) \in$ `M` $\Rightarrow$ `k`.

Subcase: Conversely suppose that `width(C`$_s$`)` $\neq 0$.

Then `M` must include `C` as the start state, a transition `r`$_s$ $=$ `C : M<s>` $\xrightarrow{\sigma_s}$ $\{$`y`$_s \in$ `sstate(M`$_s$`)`$\}$ and all the states and transitions of `M`$_s$ where `(M`$_s$`,y`$_s$`)` = `sbf_sub(C`$_s$`,A,Rec,exh)`. By the induction hypothesis, $E_s\backslash$`y`$_s \vdash E_s($`y`$_s) \in$ `M`$_s \Rightarrow$ `k`. Then, it must be the case that, $E_s\backslash$`y`$_s \vdash E_s($`y`$_s) \in$

$\texttt{sstate(M}_s\texttt{)} \Rightarrow \texttt{k}$. By MA-Sub, the assertions $\emptyset \vdash E(\texttt{y}) \in \texttt{M<s>} \Rightarrow 1$ and $\texttt{r}_s \in \texttt{transitions(M)}$ and $E_s\backslash \texttt{y}_s \vdash E_s(\texttt{y}_s) \in \texttt{sstate(M}_s\texttt{)} \Rightarrow \texttt{k}$ and $\texttt{(1,k)} \in \sigma_s$ imply $E\backslash \texttt{y} \vdash E(\texttt{y}) \in \texttt{C} \Rightarrow \texttt{k}$, which implies $E\backslash \texttt{y} \vdash E(\texttt{y}) \in \texttt{M} \Rightarrow \texttt{k}$.

Case: Assume $\texttt{(M,y)} = \texttt{sbf(C,A,Rec,exh)}$ and $E \vdash \texttt{C} \Rightarrow \texttt{k}$.

The $\vdash$ assertion could be derived by Sim or Sub. In either case, the result follows by the inductive assumption.

We prove the backward direction by induction on the size of the derivations of the right hand side judgments.

Case: Assume that $\texttt{(M,y)} = \texttt{sbf\_simple(C,A,Rec,exh)}$ and $E\backslash \texttt{y} \vdash E(\texttt{y}) \in \texttt{M} \Rightarrow \texttt{k}$.

Let $\texttt{c} = \texttt{simple\_col(C)}$. For the acceptance assertion to hold, it must be the case that $E\backslash \texttt{y} \vdash E(\texttt{y}) \in \texttt{C} \Rightarrow \texttt{k}$. Since C is a simple state, for this to hold, either premises of MA-Lab or MA-Emp must hold.

Subcase: Consider MA-Lab.

In this case, M must contain a transition of the form $\texttt{C} : \texttt{l[u],v} \xrightarrow{\texttt{I}} \texttt{S}$. There is only one place in sbf_simple where such a transition is created. It happens if $\texttt{l} \neq \texttt{()}$ for some $\texttt{l} \in \texttt{labels(C,c)}$ and $\texttt{width(C}_l\texttt{)} \neq 0$ where $\texttt{C}_l = \texttt{expand\_by\_label(C,c,l)}$. We have $\texttt{(u,v)} = \texttt{newvars(C,c)}$ and $\texttt{S} = \{\texttt{y}_l \in \texttt{sstate(M}_l\texttt{)}\}$ where $\texttt{(M}_l\texttt{,y}_l\texttt{)} = \texttt{sbf(C}_l\texttt{,A,Rec,exh)}$. Inverting MA-Lab, we have $E_l\backslash \texttt{y}_l \vdash E_l(\texttt{y}_l) \in \texttt{sstate(M}_l\texttt{)(M)} \Rightarrow \texttt{k}$ where $E_l = E\backslash \texttt{y}[\texttt{w}_1/\texttt{u}, \texttt{w}_2/\texttt{v}]$ and $E(y) = \texttt{l[w}_1\texttt{],w}_2$. Since the algorithm is deterministic, i.e. processing the same configuration several times, it generates the same states and transitions each time, the subautomata created by the recursive calls are mutually disjoint. Therefore, by 4.5.13, $E_l\backslash \texttt{y}_l \vdash E_l(\texttt{y}_l) \in \texttt{sstate(M}_l\texttt{)(M}_l\texttt{)} \Rightarrow \texttt{k}$, and, hence, $E_l\backslash \texttt{y}_l \vdash E_l(\texttt{y}_l) \in \texttt{M}_l \Rightarrow \texttt{k}$. By the induction hypothesis, $E_l \vdash \texttt{C}_l \Rightarrow \texttt{k}$. Now, by SimLab, $E \vdash_{sim} \texttt{C} \Rightarrow \texttt{k}$.

Subcase: Otherwise, MA-Emp must be applicable.

In this case, M must contain a transition of the form $\texttt{C} : \texttt{()} \xrightarrow{\texttt{I}} \texttt{S}$. Such a transition can be generated by sbf_simple only if $\texttt{l} = \texttt{()}$ for some $\texttt{l} \in \texttt{labels(C,c)}$. Let $\texttt{C}_l = \texttt{expand\_by\_label(C,c,())}$ and, let $\texttt{I}_l = \texttt{indices(C}_l\texttt{)}$. If $\texttt{width(C}_l\texttt{)} = 0$, then the above mentioned transition has the form $\texttt{C} : \texttt{()} \xrightarrow{\texttt{I}_l} \emptyset$. Inverting MA-Emp, we have $\texttt{k} \in \texttt{I}_l$. Then, by SubZero, $E_l \vdash_{sub} \texttt{C}_l \Rightarrow \texttt{k}$, and by Sub, $E_l \vdash \texttt{C}_l \Rightarrow \texttt{k}$. Now, by SimEmp, $E \vdash_{sim} \texttt{C} \Rightarrow \texttt{k}$. Conversely, if $\texttt{width(C}_l\texttt{)} \neq 0$, the argument follows the same steps as in the MA-Lab subcase.

Case: Assume that $\texttt{(M,y)} = \texttt{sbf\_sub(C,A,Rec,exh)}$ and $E\backslash \texttt{y} \vdash E(\texttt{y}) \in \texttt{M} \Rightarrow \texttt{k}$.

For the acceptance assertion to hold, it must be the case that $E\backslash \texttt{y} \vdash E(\texttt{y}) \in \texttt{C} \Rightarrow \texttt{k}$. Since C is a subroutine state, for the last assertion to hold, premises of MA-Sub must hold. Suppose the

accepting transition was generated when considering some $\mathtt{s} \in \mathtt{column\_states(C,1)}$. In part, this implies $\emptyset \vdash E(\mathtt{y}) \in \mathtt{M<s>} \Rightarrow 1$. Recall that $\mathtt{(M<s>,z)} = \mathtt{sbf\_simple(config<s>,A,Rec,exh)}$ where $\mathtt{z} = \mathtt{x}$ since $\mathtt{x}$ is the variable of the only column in $\mathtt{config<s>}$. By the induction hypothesis, we have $\emptyset[E(\mathtt{y})/\mathtt{x}] \vdash_{sim} \mathtt{config<s>} \Rightarrow 1$. Let $\mathtt{C}_s = \mathtt{expand\_by\_state(C,1,s)}$.

Subcase: Suppose $\mathtt{width(C}_s\mathtt{)} = 0$.

In this case, for MA-Sub to be satisfied, it must be the case that $\mathtt{k} \in \mathtt{indices(C}_s\mathtt{)}$. Then, by SubZero, $E\backslash\mathtt{y} \vdash_{sub} \mathtt{C}_s \Rightarrow \mathtt{k}$. Now, by SubState, $E \vdash_{sub} \mathtt{C} \Rightarrow \mathtt{k}$.

Subcase: Conversely, $\mathtt{width(C}_s\mathtt{)} \neq 0$.

Then, the accepting transition must be of the form $\mathtt{C} : \mathtt{M<s>} \xrightarrow{\sigma_s} \{\mathtt{y}_s \in \mathtt{sstate(M}_s\mathtt{)}\}$ where $\sigma_s = \mathtt{\{(1,j) \mid j \in indices(C}_s\mathtt{)\}}$ and $\mathtt{(M}_s\mathtt{,y}_s\mathtt{)} = \mathtt{sbf\_sub(C}_s\mathtt{,A,Rec,exh)}$. Inverting MA-Sub, we have $(E\backslash\mathtt{y})\backslash\mathtt{y}_s \vdash (E\backslash\mathtt{y})(\mathtt{y}_s) \in \mathtt{sstate(M}_s\mathtt{)(M)} \Rightarrow \mathtt{k}$. By Proposition 4.5.13, $(E\backslash\mathtt{y})\backslash\mathtt{y}_s \vdash (E\backslash\mathtt{y})(\mathtt{y}_s) \in \mathtt{sstate(M}_s\mathtt{)(M}_s\mathtt{)} \Rightarrow \mathtt{k}$, and, hence, $(E\backslash\mathtt{y})\backslash\mathtt{y}_s \vdash (E\backslash\mathtt{y})(\mathtt{y}_s) \in \mathtt{M}_s \Rightarrow \mathtt{k}$. By the induction hypothesis, $E\backslash\mathtt{y} \vdash_{sub} \mathtt{C}_s \Rightarrow \mathtt{k}$. So, by SubState, $E \vdash_{sub} \mathtt{C} \Rightarrow \mathtt{k}$.

Case: Assume that $\mathtt{(M,y)} = \mathtt{sbf(C,A,Rec,exh)}$ and $E\backslash\mathtt{y} \vdash E(\mathtt{y}) \in \mathtt{M} \Rightarrow \mathtt{k}$.

Either `sbf_simple` or `sbf_sub` must have been used to generate `(M,y)`. Follow the same reasoning steps as in the above two cases to conclude $E \vdash_{sim} \mathtt{C} \Rightarrow \mathtt{k}$ or $E \vdash_{sub} \mathtt{C} \Rightarrow \mathtt{k}$ respectively. The result $E \vdash \mathtt{C} \Rightarrow \mathtt{k}$ follows by either Sim or Sub. □

4.5.6 The SNBF Algorithm

There are two differences between the SNBF and SBF algorithms. The first concerns handling configurations that cannot be expanded by label. Instead of doing expansion by state as it was described for `sbf`, the SNBF algorithm generates a fresh index converter that makes a subroutine transition for every column of the current configuration. Subroutine automata used in these transitions are based on the states appearing in the corresponding column.

Since the SNBF algorithm employs subroutine automata that are based on collections of tree automaton states, rather than just one state, at the top level, the algorithm generates a subroutine for every subset of `S`. (Our implementation does not generate a subroutine automaton unless it encounters a call to it while constructing another automaton.)

4.5.7 Summary of Complexity Results

Let us consider the running time and the size of the generated matching automata for the compilation algorithms described in this section. Example 4.4.8 shows that backtracking matching automata generated by SBF can exhibit exponential running time in the size of the input value.

Non-backtracking matching automata generated by SNBF are at worst linear. We have not studied running time complexity of matching automata in relation to the size of the tree automaton given as input to SBF and SNBF. In our application domain of large XML documents and relatively small patterns, this question is less important than complexity in the size of the value.

Space complexity involves two components: the number of generated subroutine matching automata and the size of an individual matching automaton. The SBF algorithm generates at most a linear number of subroutine automata in the size of the input tree automaton. The SNBF algorithm can result in exponentially many subroutine automata.

The size of an individual matching automaton depends on the strategy for selecting loop breakers. The maximal set of loop breakers results in a matching automaton whose size is at worst linear in the size of the input tree automaton. As Example 4.4.9 shows, a minimal set of loop breakers can result in a double exponential matching automata. Finally, we can show that using multiple predecessor or multiple successor loop breaker sets ensures that the size of the generated automaton is no worse than polynomial and exponential respectively.

4.6 From Matching Automata to Intermediate Code

This section describes how a collection of matching automata in SBF or SNBF can be converted into a collection of XIL functions. We assume that the collection of matching automata is *call-closed*—that is if a matching automaton `A` from the collection has a subroutine transition to a matching automaton `B`, then `B` is also in the collection. We start by presenting the conversion algorithm for matching automata in SBF.

4.6.1 Converting SBF Matching Automata into XIL

Our goal is to convert every matching automaton to an equivalent integer-valued XIL function. To associate matching automata and XIL functions formally, we must account for the difference in the semantics of these two models. One mismatch arises from the fact that while the matching automaton acceptance relation is non-deterministic—one of multiple values may be output for a given input value—the evaluation semantics of XIL is deterministic—a function can return exactly one integer for a given input. We must also specify how rejection of a value is represented in the intermediate language.

Since matching automaton indices are intended to identify clauses of a `match` expression, we must take into consideration the first-match semantics of pattern matching when we specify how to disambiguate the non-determinism of the matching automaton acceptance relation. Thus if a

matching automaton accepts some value `v` with an output index `i`, we require the corresponding XIL function to terminate on `v` producing some integer `j` that is smaller than or equal to `i`. Of course, the function must also be sound with respect to the matching automaton; so, if the function result is `j` for an input value `v`, it must be the case that the matching automaton accepts `v` outputting `j` as well.

Rejection of a value in XIL can be signaled by returning a special integer that may not be the result of any accepting computation. Since matching automaton indices are necessarily positive, 0 can serve as such a rejection indicator. Our goal for the relationship between the two models therefore is as follows: the matching automaton rejects an input value `v` if and only if the corresponding XIL function terminates on `v` producing 0.

In the remainder of this section, we will show how to build XIL programs that satisfy to the above relationship. We will use the same identifiers to denote both the matching automaton and the XIL function obtained from it; however, since the matching automaton acceptance and XIL evaluation judgments are syntactically different, there will be no confusion in the formal statements.

Let `A` be some matching automaton in SBF. Since `A` is acyclic, it is possible to sort its states topologically obtaining $\mathtt{q}_1 \ldots \mathtt{q}_n$ where $\mathtt{q}_1$ is the start state. Figure 4.16(a) displays the structure of the corresponding XIL function. Each $\mathtt{e}_k$ is obtained from state $\mathtt{q}_k$, and there is a static exception handler associated with every non-start state. Introducing exception handlers allows us to avoid code duplication arising from states with multiple incoming transitions, but this approach leads to an unnecessarily large number of exception handlers and `exit` statements. We do not deal with this inefficiency until the following chapter where we will discuss an optimization phase that performs inlining of superfluous exceptions.

Figure 4.16(b) contains function `conv_ma` that takes matching automaton $\mathtt{A} = (\mathtt{Q},\mathtt{q}_s,\mathtt{R})$ and converts it into a XIL function. The first line of the algorithm uses the function `fresh_id` to create a fresh variable `x` that will be used as the formal parameter of the function and the first parameter of every exception handler. The second line performs the topological sort of the matching automaton states. The function `top_sort` takes the set of states, the transition relation, and the start state of a matching automaton and returns the states sorted topologically according to the transition relation with the start state in the first position. In the following lines, the algorithm uses `fresh_id` to generate an exception label for every non-start state of the matching automaton and creates a map `Exit` associating each state with one exception label. The algorithm then proceeds to generate XIL code for each state using function `conv_state` that will be described below. The next line computes the formal parameters of the exception handlers. We will describe the specifics of this process when we discuss how code for individual states is generated. Finally, `conv_ma` constructs

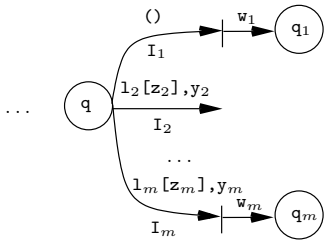
```
fun A(x) =
 e1
 with
  j2(...) → e2
  ...
  jn(...) → en
```

(a)

```
function conv_ma(A,Q,qs,R) =
 let x = fresh_id() in
 let <q1 ... qn> = top_sort(Q,qs,R) in
 let ji = fresh_id() for i ∈ {2...n} in
 let Exit = {qi ↦ ji | i ∈ {2...n}} in
 let ei = conv_state(qi,R,x,Exit) for i ∈ {1...n} in
 let Vi = sort(FV(qi)) for i ∈ {2...n} in
 fun A(x) = e1 with
 j2(x,V2) → e2
 ...
 jn(x,Vn) → en
end
```

(b)

Figure 4.16: Converting matching automata to XIL: a sketch of a function corresponding to a matching automaton (a); a conversion algorithm for a matching automaton $\mathtt{A} = (\mathtt{Q}, \mathtt{q_s}, \mathtt{R})$



(a)

```
j(x,V) →
 case x of
  | () → exit j1(w1,V1)
  | l2[z2],y2 → i
  | ...
  | lm[zm],ym → exit jm(wm,Vm)
  else 0
```

(b)

Figure 4.17: Example of simple state conversion

and returns the declaration of a XIL function corresponding to the input matching automaton.

In `conv_ma` and in other algorithms described below, we use the following quoting conventions for dealing with code fragments. Expressions delimited by a rectangle represent code fragments that should not be evaluated any further. Any identifier occurring inside a code fragment is a variable ranging over some type of code (pattern, expression, integer, exit label, integer, variable name); its contents should be spliced in place of the variable occurrence before the code fragment is returned.

Let us now discuss how to generate XIL code for individual states. Consider an example of a simple state and the corresponding code fragment shown in Figure 4.17. The latter is obtained from the former as follows. Overall, a simple state `q` corresponds to a `case` expression with a clause for each outgoing transition. Depending on whether `q` is a start state or not, the `case` expression resides outside of exception handlers (like $\mathtt{e}_1$ in Figure 4.16(a)) or as part of some exception handler

(like $\texttt{e}_2$ through $\texttt{e}_n$ in the same example.) Figure 4.17 presents an example of a non-start state; for every incoming transition with a destination variable `y`, the generated program will contain an `exit` statement of the form `exit j(y ...)`.

Since XIL patterns coincide with patterns annotating simple transitions in matching automata, the pattern of each `case` clause is taken directly from the corresponding transition. The right hand side of each clause is calculated from the destination part of the corresponding transition. Because of sequentiality, all transitions have at most one destination pair; so, we must consider two cases: transitions with one destination pair—such as the transitions annotated by `()` and $\texttt{l}_m\texttt{[z}_m\texttt{],y}_m$ in Figure 4.17(a)—and *terminal* transitions with no destination pairs—such as the transition annotated by $\texttt{l}_2\texttt{[z}_2\texttt{],y}_2$.

When the control of a matching automaton is in a simple state with a terminal outgoing transition whose pattern matches the current value, the matching automaton may succeed outputting any of the indices from the transition's index set. We mirror this behavior in XIL by returning the smallest index of the transition's index set. This is illustrated in Figure 4.17(b) by the second clause of the `case` expression where `i` is the integer constant equal to the smallest element of $\texttt{I}_2$.

When the current value matches the pattern of a non-terminal transition originating in the current state, the matching automaton will succeed if the transition's destination state accepts the value stored in the destination variable. This is simulated by an `exit` statement invoking the exception handler corresponding to the destination state with the destination variable passed as the first parameter. The first and the last clauses of the `case` expression shown in 4.17(b) are examples of code generated for non-terminal transitions; they correspond to the non-terminal transitions of the displayed matching automaton fragment.

It remains to discuss how we compute formal parameters of exception handlers and actual parameters of `exit` statements. The first parameter has a special significance. In `exit` statements, it is the destination variable of the corresponding transition as discussed in the previous paragraph. In exception handlers it is the identifier generated in the first line of `conv_ma`. The same identifier is used as the parameter of the `case` expression residing in the body of the exception handler. The remaining exception parameters are determined by the *free variables* of the state associated with the exception handler. Intuitively, a variable is free in a state `q` if it is the destination variable of some transition `t` reachable from `q`, but it does not appear in the pattern part of any transition between `q` and `t` inclusively. This is formalized by the following definition.

4.6.1 Definition: Let $\texttt{A} = \texttt{(Q,}\texttt{q}_s\texttt{,R)}$ be a sequential acyclic matching automaton and let `P` be

the set of all XIL patterns. Sets of pattern, free and bound variables are defined as follows.

$\mathtt{p} \in \mathtt{P}$	$\mathtt{Vars(p)}$	$=$	$\emptyset$	if $\mathtt{p} = \mathtt{()}$
	$\mathtt{Vars(p)}$	$=$	$\{\mathtt{y}\} \cup \{\mathtt{z}\}$	if $\mathtt{p} = \mathtt{l[y],z}$
$\mathtt{q} \in \mathtt{Q}$	$\mathtt{FV(q)}$	$=$	$\mathtt{FV}(\mathtt{r}_1) \cup \ldots \cup \mathtt{FV}(\mathtt{r}_n)$	where $\mathtt{r}_1 \ldots \mathtt{r}_n$ are transitions originating in $\mathtt{q}$
	$\mathtt{BV(q)}$	$=$	$\mathtt{BV}(\mathtt{r}_1) \cup \ldots \cup \mathtt{BV}(\mathtt{r}_n)$	where $\mathtt{r}_1 \ldots \mathtt{r}_n$ are transitions reachable from $\mathtt{q}$
$\mathtt{r} \in \mathtt{R}$	$\mathtt{BV(r)}$	$=$	$\mathtt{Vars(p)}$	if $\mathtt{r}$ is a simple transition of the form $\mathtt{q} : \mathtt{p} \xrightarrow{\mathtt{I}} \mathtt{S}$
	$\mathtt{BV(r)}$	$=$	$\emptyset$	if $\mathtt{r}$ is a subroutine transition
	$\mathtt{FV(r)}$	$=$	$(\{\mathtt{y}\} \cup \mathtt{FV(q)}) \setminus \mathtt{BV(r)}$	if the destination pair of $\mathtt{r}$ is of the form $\{\mathtt{y} \in \mathtt{q}'\}$
	$\mathtt{FV(r)}$	$=$	$\emptyset$	if $\mathtt{r}$ is a terminal transition,

The matching automaton is said to be **well-scoped** if $\mathtt{FV}(\mathtt{q}_s) = \emptyset$.

Figure 4.17 illustrates the relationship between free variables and exception parameters. Meta-variables $\mathtt{V}$ and $\mathtt{V}_1 \ldots \mathtt{V}_m$ appearing in `exit` statements denote lists of free variables of states $\mathtt{q}$ and $\mathtt{q}_1 \ldots \mathtt{q}_m$ respectively.

```
function conv_simple(q,R,x,Exit) =
 let {(p1,I1,S1)...(pm,Im,Sm)} =
   {(p,I,S) | q : p --I--> S ∈ R} in
 let fi = conv_trans(Ii,Si,Exit)
   for i ∈ {1...m} in
 case x as
  | p1 → f1
  ...
  | pm → fm
  else 0
end
```

(a)

```
function conv_trans(I,S,Exit) =
 if S = ∅ then
   let i = min(I) in
   i
 else
   assume S = {y ∈ q} in
   let V = sort(FV(q)) in
   let j = Exit(q) in
   exit j(y,V)
end
```

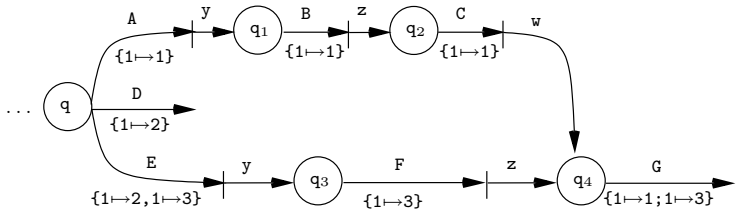
(b)

Figure 4.18: Simple state conversion algorithm: state conversion (a); transition conversion (b)

Figure 4.18 contains function `conv_simple` that generates a `case` expression for a simple state of a matching automaton `q`. In addition to the state, `conv_simple` takes the transition relation of the matching automaton, the name of the variable to be used as the argument of the `case` expression, and the mapping of states to exception labels `Exit`. The first line of `conv_simple` extracts the transitions originating in `q`. Each transition will result in a clause in the constructed `case` expression. The second line of `conv_simple` invokes function `conv_trans` to generate the right hand side of a clause from the index set and the set of destination pairs of the corresponding transition. A `case` expression is then built from the patterns appearing in the transitions and the right hand side expressions generated by `conv_trans`.

Function `conv_trans` constructs right hand sides of clauses as outlined above. For terminal transitions, it returns the smallest associated index; for non-terminal transitions, it constructs an `exit` statement raising the exception associated with the destination state. The sixth line of `conv_trans` constructs an alphabetically sorted list of variables that will be used as actual parameters of the `exit` statement. Similarly, the sixth line of `conv_ma` shown in Figure 4.16(b) computes formal parameters of exception handlers. Note how we slightly abuse notation by using the same lists of variables both as actual parameters in `exit` statements and as formal parameters in the declaration of exception handlers; while this is consistent in XIL, in a typed setting, we would have to generate different syntactic objects for formal and actual parameters based on the same list of variables.

The following lemma formally associates the XIL statement produced by `conv_simple` with the input matching automaton state.



(a)

```
j(x,y,z,w) →
 if and(A(x),B(y),C(z),G(w))
   then 1
 else if D(x)
   then 2
 else if and(E(x),F(y),G(z))
   then 3
 else 0
```

(b)

Figure 4.19: Example of subroutine state conversion

4.6.2 Lemma: Let $\mathtt{A} = \mathtt{(Q,q_s,R)}$ be a matching automaton in SBF, and let $\mathtt{q} \in \mathtt{Q}$ be a simple state. Let $\{\mathtt{q} : \mathtt{p}_k \xrightarrow{\mathtt{I}_k} \{\mathtt{y}_k \in \mathtt{q}_k\} \mid \mathtt{k} \in \{1 \ldots m\}\}$ be the set of all non-terminal transitions originating in $\mathtt{q}$, and let `Exit` be a mapping from states to exception labels that maps $\mathtt{q}_1 \ldots \mathtt{q}_m$ to distinct exception labels $\mathtt{j}_1 \ldots \mathtt{j}_m$ respectively. Let $\mathtt{V}_1 \ldots \mathtt{V}_m$ be alphabetically sorted list of free variables of states $\mathtt{q}_1 \ldots \mathtt{q}_m$ respectively. Let $\mathtt{x}$ be a variable distinct from those appearing in $\mathtt{V}_k$ for all $k \in \{1 \ldots m\}$. Let Δ be an exit environment such that $\Delta(\mathtt{j}_k) = \mathtt{((x,V}_k\mathtt{),e}_k\mathtt{,}\Delta_k\mathtt{)}$ for $k \in \{1 \ldots m\}$ where $\mathtt{e}_k$ and Δ_k satisfy the following assumptions for any sequence value $\mathtt{v}$ and $k \in \{1 \ldots m\}$:

- for any environment E, if $E[\mathtt{v}/\mathtt{x}] \bullet \Delta_k \vdash \mathtt{e}_k \longrightarrow \mathtt{i}$ for some $\mathtt{i} > \mathtt{0}$, then $E \vdash \mathtt{v} \in \mathtt{q}_k \Rightarrow \mathtt{i}$.
- for any environment E, if $E[\mathtt{v}/\mathtt{x}] \bullet \Delta_k \vdash \mathtt{e}_k \longrightarrow \mathtt{0}$, then $E \vdash \mathtt{v} \notin \mathtt{q}_k$;
- for any environment E such that $\mathtt{FV(q}_k\mathtt{)} \subseteq dom(E)$, if $E \vdash \mathtt{v} \in \mathtt{q}_k \Rightarrow \mathtt{i}$, then $E[\mathtt{v}/\mathtt{x}] \bullet \Delta_k \vdash \mathtt{e}_k \longrightarrow \mathtt{j}$ for some $\mathtt{j} \leq \mathtt{i}$.
- for any environment E such that $\mathtt{FV(q}_k\mathtt{)} \subseteq dom(E)$, if $E \vdash \mathtt{v} \notin \mathtt{q}_k$ and the automaton does not diverge on v, then $E[\mathtt{v}/\mathtt{x}] \bullet \Delta_k \vdash \mathtt{e}_k \longrightarrow \mathtt{0}$.

Then a similar collection of assertions hold for the input state q and the corresponding XIL expression e = conv_simple(q,R,x,Exit) for any sequence value v:

- for any environment E, if $E[\mathtt{v}/\mathtt{x}] \bullet \Delta \vdash \mathtt{e} \longrightarrow \mathtt{i}$ for some $\mathtt{i} > \mathtt{0}$, then $E \vdash \mathtt{v} \in \mathtt{q} \Rightarrow \mathtt{i}$.
- for any environment E, if $E[\mathtt{v}/\mathtt{x}] \bullet \Delta \vdash \mathtt{e} \longrightarrow \mathtt{0}$, then $E \vdash \mathtt{v} \notin \mathtt{q}$;
- for any environment E such that $\mathtt{FV(q)} \subseteq dom(E)$, if $E \vdash \mathtt{v} \in \mathtt{q} \Rightarrow \mathtt{i}$, then $E[\mathtt{v}/\mathtt{x}] \bullet \Delta \vdash \mathtt{e} \longrightarrow \mathtt{j}$ for some $\mathtt{j} \leq \mathtt{i}$.
- for any environment E such that $\mathtt{FV(q)} \subseteq dom(E)$, if $E \vdash \mathtt{v} \notin \mathtt{q}$ and the automaton does not diverge on v, then $E[\mathtt{v}/\mathtt{x}] \bullet \Delta \vdash \mathtt{e} \longrightarrow \mathtt{0}$.

We now move to a discussion of how to generate XIL from subroutine states. Figure 4.19 shows an example of a subroutine state and the code fragment obtained from it. First observe that q has free variables y, z, and w since they are used as destination variables in several transitions but are not defined anywhere else in the fragment. As explained above, these variables in addition to x are the formal parameters of the exception handler associated with q. When the matching automaton is in state q, it can succeed by following one of the following three scenarios:

- Subroutines A, B, C, and G succeed with output 1 given as input the contents of x, y, z, and w respectively. In this case, the displayed matching automaton succeeds with output 1 since the index mapping relations of all four subroutine transitions contain the pair $1 \mapsto 1$.
- Subroutine D succeeds with output 1 given as input the contents of x. In this case, the displayed matching automaton succeeds with output 2 because of the index mapping relation $\{1 \mapsto 2\}$.
- Subroutines E, F, and G succeed with output 1 given as input the contents of x, y, and z respectively. In this case, the displayed matching automaton succeeds with output 3 because

of the index mapping pair $1 \mapsto 3$. Notice that the index mapping pair $1 \mapsto 2$ present in the E transition does not play any role since the subsequent transitions originating in q_3 and q_4 cannot result in 2.

Each of these scenarios correspond to a *call tail* originating in q. A call tail is a sequence of states connected by subroutine transitions with the same index mapping pair. More formally, a call tail is defined as follows:

4.6.3 Definition: Let $A = (Q, q_s, R)$ be a matching automaton in SBF, and let $q \in Q$ be a subroutine state. We say that A contains a call tail originating in q and containing a call to B_0 with the current value followed by calls $B_1(x_1), \ldots, B_m(x_m)$, and yielding an index k—written $q{:}B_0(*), B_1(x_1), \ldots, B_m(x_m) \Rightarrow k \in R$—if

$$q : B_0 \xrightarrow{\sigma_0} \{x_1 \in q_1\} \in R \quad \text{and}$$
$$q_1 : B_1 \xrightarrow{\sigma_1} \{x_2 \in q_2\} \in R \quad \text{and}$$
$$\ldots$$
$$q_m : B_m \xrightarrow{\sigma_m} \emptyset \in R,$$

where $1 \mapsto k \in \sigma_i$ for all $i \in \{0 \ldots m\}$.

```
function conv_sub(q,R,x) =
  let S = {(i,B,CC) | q : B(*),CC ⇒ i ∈ R} in
  let <(i1,B1,CC1)...(im,Bm,CCm)> =
    sort_tails(S) in
  if and(B1(x),CC1) then i1
  ...
  else if and(Bm(x),CCm) then im
  else 0
end
```

(a)

```
function conv_state(q,R,x,Exit) =
  if is_simple(q) then
    conv_simple(q,R,x,Exit)
  else
    conv_sub(q,R,x)
end
```

(b)

Figure 4.20: Subroutine state conversion algorithm (a); general state conversion algorithm (b)

A subroutine state corresponds to a sequence of XIL if/then branches—each arising from a call tail. The test of a branch is a conjunction of the subroutine calls that make up the corresponding call tail; if the calls succeed, the then part of the branch returns the index associated with the call tail. The order of the branches is determined by first-match considerations—the smaller the index output by a call tail is, the earlier this call tail has to be tested by the program.

Figure 4.20(a) shows how these ideas are implemented in function conv_sub. Its first line extracts the call tails originating in the input subroutine state and places them into S. We use CC

to range over sequences of subroutine/variable pairs such as `B(y),C(z),G(w)` in the example of Figure 4.19. The next line of `conv_sub` uses function `sort_tails` to sort the call tails in ascending index order. A sequence of `if/then` expressions is then created from the call tails and returned.

The following lemma associates the XIL statement produced by `conv_sub` with the input matching automaton state.

4.6.4 Lemma: Let $\mathtt{B}_1 \ldots \mathtt{B}_m$ be a call-closed collection of matching automata in SBF. Let $\mathtt{B}_1 \ldots \mathtt{B}_m$ also denote a collection of corresponding XIL functions satisfying the following conditions for any sequence value `v` and any $k \in \{1 \ldots m\}$:

- if $\mathtt{call\ B}_k\mathtt{(v)} \longrightarrow \mathtt{0}$, then $\emptyset \vdash \mathtt{v} \notin \mathtt{B}_k$;
- if $\mathtt{call\ B}_k\mathtt{(v)} \longrightarrow \mathtt{i}$ for some $\mathtt{i} > \mathtt{0}$, then $\emptyset \vdash \mathtt{v} \in \mathtt{B}_k \Rightarrow \mathtt{i}$;
- if $\emptyset \vdash \mathtt{v} \in \mathtt{B}_k \Rightarrow \mathtt{i}$, then $\mathtt{call\ B}_k\mathtt{(v)} \longrightarrow \mathtt{j}$ for some $\mathtt{j} \leq \mathtt{i}$;
- if $\emptyset \vdash \mathtt{v} \notin \mathtt{B}_k$ and $\mathtt{B}_k$ does not diverge on `v`, then $\mathtt{call\ B}_k\mathtt{(v)} \longrightarrow \mathtt{0}$;

Let $\mathtt{A} = \mathtt{(Q,q}_s\mathtt{,R)}$ be a matching automaton from the above collection, and let $\mathtt{q} \in \mathtt{Q}$ be a subroutine state. Let `x` be a variable distinct from the variables appearing in `A`. The following collection of assertions hold for the input state `q` and the corresponding XIL expression $\mathtt{e} = \mathtt{conv\_sub(q,R,x)}$ for any sequence value `v` and for any environment E:

- if $E[\mathtt{v/x}] \vdash \mathtt{e} \longrightarrow \mathtt{i}$ for some $\mathtt{i} > \mathtt{0}$, then $E \vdash \mathtt{v} \in \mathtt{q} \Rightarrow \mathtt{i}$.
- if $E[\mathtt{v/x}] \vdash \mathtt{e} \longrightarrow \mathtt{0}$, then $E \vdash \mathtt{v} \notin \mathtt{q}$;
- if $E \vdash \mathtt{v} \in \mathtt{q} \Rightarrow \mathtt{i}$, then $E[\mathtt{v/x}] \vdash \mathtt{e} \longrightarrow \mathtt{j}$ for some $\mathtt{j} \leq \mathtt{i}$.
- if $E \vdash \mathtt{v} \notin \mathtt{q}$ and the automaton does not diverge on `v`, then $E[\mathtt{v/x}] \vdash \mathtt{e} \longrightarrow \mathtt{0}$.

Figure 4.20(b) shows function `conv_state` that generates XIL code for an arbitrary matching automaton state. Using predicate `is_simple`, it checks whether the input state is simple or subroutine and invokes `conv_simple` or `conv_sub` appropriately.

We conclude this section with a lemma stating the correspondence between a collection of matching automata and a collection of XIL functions generated from them using `conv_ma`.

4.6.5 Lemma: Let $\texttt{A}_1 = \texttt{(Q}_1\texttt{,q}_1\texttt{,R}_1\texttt{)} \ldots \texttt{A}_m = \texttt{(Q}_m\texttt{,q}_m\texttt{,R}_m\texttt{)}$ be a call-closed collection of matching automata in SBF, and let $\texttt{A}_1 \ldots \texttt{A}_m$ also denote a collection of XIL functions generated from these automata:

$$\begin{array}{lcl} \texttt{conv\_ma(A}_1\texttt{,Q}_1\texttt{,q}_1\texttt{,R}_1\texttt{)} & = & \texttt{fun A}_1\texttt{...} \\ & \ldots & \\ \texttt{conv\_ma(A}_m\texttt{,Q}_m\texttt{,q}_m\texttt{,R}_m\texttt{)} & = & \texttt{fun A}_m\texttt{...} \end{array}$$

The correspondence between the matching automata and the associated XIL functions is described by the following assertions that hold for any sequence value v and any $k \in \{1 \ldots m\}$:

- if $\texttt{call A}_k\texttt{(v)} \longrightarrow \texttt{0}$, then $\emptyset \vdash \texttt{v} \notin \texttt{A}_k$;
- if $\texttt{call A}_k\texttt{(v)} \longrightarrow \texttt{i}$ for some $\texttt{i} > \texttt{0}$, then $\emptyset \vdash \texttt{v} \in \texttt{A}_k \Rightarrow \texttt{i}$;
- if $\emptyset \vdash \texttt{v} \in \texttt{A}_k \Rightarrow \texttt{i}$, then $\texttt{call A}_k\texttt{(v)} \longrightarrow \texttt{j}$ for some $\texttt{j} \leq \texttt{i}$;
- if $\emptyset \vdash \texttt{v} \notin \texttt{A}_k$ and $\texttt{A}_k$ does not diverge on v, then $\texttt{call A}_k\texttt{(v)} \longrightarrow \texttt{0}$;

In SBF, we do not distinguish between matchers and acceptors when converting matching automata into XIL. An acceptor is just a special case of matchers that uses only two indices—0 and 1. In the above lemma, therefore, $\texttt{B}_i$ can refer to either an acceptor or a matcher. SNBF is different in a sense that matchers and index converters are treated differently—while the former are converted to XIL functions as in SBF, the latter do not correspond to stand alone functions, but, rather, are translated inline as part of subroutine state conversion.

rng2xt	max Rec	succ Rec	pred Rec
size of gen'd code	8,788	7,758	9,169
# of eval steps	955,714	580,813	455,270
format_html			
size of gen'd code	18,127	15,577	18,357
# of eval steps	8,484	9,269	7,384
format_bibtex			
size of gen'd code	24,729	41,518	52,856
# of eval steps	131,642	45,104	22,892

rng2xt	backtr.	non-backtr.
size of gen'd code	9,169	17,096
# of eval steps	455,270	542,753
format_html		
size of gen'd code	18,357	23,138
# of eval steps	7,384	10,234
format_bibtex		
size of gen'd code	52,856	34,207
# of eval steps	22,892	34,722

Figure 4.21: Comparison of various loop breaker sets (a), and SBF vs. SNBF approaches (b)

4.7 Experiments

This section describes our performance experiments. To evaluate the code generated by our compiler, we have implemented a simple XIL interpreter instrumented to report the size of the generated

program estimated by the number of abstract syntax tree nodes and its speed estimated by the number of evaluation steps such as function calls, variable lookups, and primitive applications.

We analyze three test programs. The first one, rng2xt, is a 500 line program (2,200 AST nodes, 63% of which are in patterns) that converts a Relax NG schema into a collection of XDuce regular types. It is run on a 900 line XML document. The second, format_html, is a 3,000 line program (7,400 AST nodes, 92% of which are in patterns) that traverses an html page, finds all of its headings and makes a table of contents with references to them. The third, format_bibtex, is a 1,200 line program (4,400 AST nodes, 55% of which are in patterns) that reads a bibtex file and converts it into an html page displaying the file's contents.

The first experiment compares different methods of selecting sets of loop breakers. In the table shown in Figure 4.21(a), max `Rec`, succ `Rec`, and pred `Rec` denote the maximal, multiple successor, and multiple predecessor sets of loop breakers respectively.

For the first two programs, all loop breaker selection strategies result in programs of roughly the same size. For format_bibtex, using the maximal set of loop breakers produces a substantially smaller program as discussed in Section 4.5.3. Maximal sets of loop breakers generally lead to slower programs. The fastest programs, for our tests, were generated using multiple predecessor loop breaker sets. Apparently, this strategy introduces the least number of subroutine functions, and, hence, incurs the least amount of penalty arising from function calls. Selecting minimal sets of loop breakers simply did not work for format_html and format_bibtex resulting in dramatic code size explosion.

The table shown in Figure 4.21(b)compares the backtracking and non-backtracking compilation algorithms. In both cases, we use multiple predecessor loop breaker sets.

The backtracking approach results in faster programs for all the test cases. It seems that the cost of backtracking occurring in the programs generated by this method is far outweighed by the cost of operations on tuples of boolean values employed in the programs generated by the non-backtracking approach.

4.8 Related Work

The XDuce programming language [36, 35], provided the starting point for our project and is the source language of our compilers. As mentioned in Section 4.4.1, our compiler uses XDuce's algorithm for converting source patterns into states of a binary top-down tree automaton. Hosoya and Pierce [35] provide a detailed account of this algorithm.

Compilation of datatype-based pattern matching has been researched extensively in the past.

Papers in this field usually distinguish two general approaches. Baudinet and MacQueen [3] describe a method based on decision trees (used in the SML/NJ compiler) that is geared toward producing efficient non-backtracking code but might suffer from an occasional blowup of the output code size. Conversely, Augustsson [2] introduces a backtracking approach that restrains the size of the output code at the expense of its efficiency. Le Fessant and Maranget [12] combine the advantages of both approaches by introducing several optimizations (used in the OCaml compiler) to the backtracking technique that result in more efficient but still compact output code. (On a slightly different note, Sestoft [62] shows how instrumentation and partial evaluation can help derive a reasonably efficient pattern match compiler from a simple pattern matching algorithm. The resulting compiler, however, is less advanced than the compilers mentioned above.)

A similar trade-off between performance and space considerations is present in the compilation algorithms of this paper. By varying loop breaker sets used in the algorithm, we can either minimize the code size but introduce more subroutine function calls and hence more backtracking, or, conversely, minimize the number of subroutine calls at the risk of generating very large code. Since our work focuses on recursive patterns and the many issues that arise as a result of dealing with recursive patterns, our algorithms can be viewed as an extension of datatype based pattern compilation.

Compilers for logic programming languages employ optimizations focused on avoiding backtracking and sharing common tests performed by different branches of conditional expressions. Like datatype pattern matching optimizations, these techniques ([48] and [10]) do not handle recursive patterns and, thus, cannot be used directly in our compiler.

The topic of procedure inlining optimization ([40], [1], [69]) is relevant to our work. Our compilation algorithm is similar to procedure inliners in that it examines a potentially cyclic structure—a tree automaton—and determines which nodes of that structure can be implemented inline and which nodes must correspond to procedures. Peyton Jones and Marlow [40] introduce the notion of loop breakers, show how selecting different loop breakers can have a significant effect on the quality of the generated code, and describe a heuristic for locating loop breakers.

One might consider an alternative approach to compiling regular patterns. First, generate a target program using a simple code generation method that associates a procedure with every tree automaton state and does not perform any of the optimizations described in this paper. Then, hand the obtained program to an existing procedure inliner, such as the one described by Peyton Jones and Marlow [40], and let it do its job. The problem with this approach is that by the time we generate the first version of target code, all the tree automaton-related information is lost and cannot be taken advantage of by the optimizer. Hence, our approach of doing code generation and

optimization in the same stage is advantageous.

A great deal of research has been conducted in the area of regular tree languages. This field studies properties of regular tree and forest languages. Two kinds of trees are considered: ranked in which any label has an arity, and the number of child subtrees in a node is determined by the arity of the node's label ([8]); and unranked in which any node can have arbitrary number of children regardless of the label ([5], [58].)

One of the problems investigated in this area, the membership problem, is relevant to our research. Its goal is to check whether a tree belongs to a particular regular tree language specified by a regular tree grammar. A standard solution described in the literature involves converting the grammar into a non-deterministic bottom-up tree automaton (NTA), building an equivalent bottom-up deterministic tree automaton (DTA), and matching the input value against the obtained DTA. The first and third components of the above process can be accomplished in linear time in the size of the input. The process of determinization, however, can result in a DTA whose size is exponential in the size of the original NTA. Seidl and Neumann [59] introduce pushdown forest automata—bottom-up automata with a top-down twist—that exhibit better determinization characteristics.

Bottom-up automata do not give us a natural framework for modeling target language code. It is unclear how to "read-off" a target language program from a bottom-up automaton in such a way that this program can be further optimized by inlining and other low-level transformations. For this reason, we base our compilation algorithms on the top-down approach, and, hence, bottom-up techniques cannot be applied for our purposes directly.

Furthermore, our matching automata are not meant—at least at this time—to compete with the automata theoretic approaches on the terms that are of interest to that community. In our framework, we are interested in and can express transformations that may give us constant factor improvements in the performance of the generated program for common source programs rather than asymptotic complexity improvements for certain rare cases. This interest is reflected in the design of matching automata that features subroutine transitions to help us examine code inlining approaches; indices to help us implement pattern matching with multiple patterns and experiment with exhaustiveness optimization; and variables in transitions for more flexibility in scheduling evaluation of subtrees of the input tree.

It can be shown [8] that word automata minimization algorithms can be generalized to tree automata. Minimization of tree automata can be employed in conjunction with the compilation algorithms discussed in this paper. Recall that the first stage of our compiler converts source patterns into states of a tree automaton. Currently, the algorithm used in this stage [35] does not

produce a minimal tree automaton. In the future, we would like to experiment with employing existing minimization algorithms at this stage to give us a better starting point for SBF and SNBF.

Another question concerns minimizing matching automata produced by SBF and SNBF. This problem is hard. Because of subroutine transitions, minimizing matching automata is not unlike trying to find an optimal solution to function inlining, and we are not aware of a function inliner that boasts optimality; they all are based on heuristics. Similarly, our compilation algorithms do not claim to produce minimal matching automata, but rather matching automata that are good enough in practice.

Chapter 5

Unambiguous Regular Pattern Matching with Variables

Extending regular pattern matching with variable binding is not a trivial matter. Not only does it require untangling intricate interactions between variable binding on one side and recursion and other complex aspects of pattern compilation on the other, but it also raises the question of principled treatment of ambiguous patterns. The goal of this chapter is to provide a formal foundation of variable binding and pattern disambiguation in the context of matching automata.

A pattern is ambiguous if there exists a value that can be matched against the pattern in more than one way. Consider XtaticLite pattern `p = a[]?,a[]?` where `p?` stands for `p|()`. There are two ways of matching `p` against `a[]`: in the first one, the left subpattern of `p` matches the whole value and the right subpattern of `p` matches the empty sequence; in the second, the left subpattern of `p` matches the empty sequence and the right subpattern of `p` matches the whole value.

Ambiguity can also arise in patterns with repetition operators. For instance, consider pattern `p = a[]*,a[]*`. Similarly to the previous example, there are two ways of matching this pattern against `a[]`.

There are several known ways of specifying a deterministic semantics for regular pattern matching. The POSIX disambiguation semantics [65] gives preference to the subpatterns that come earlier in the left to right traversal of the abstract syntax tree and attempts to match the subpattern with the highest priority to a portion of input that starts as early as possible and extends as long as possible. For both patterns above, the POSIX disambiguation semantics prescribes the whole value to be matched to the left subpattern.

The first/longest match disambiguation semantics [65] has a special provision for union and

repetition patterns. It specifies that a union pattern must match its left (first) alternative if it is at all possible and only match the right alternative of the union if the left one does not match. A repetition pattern must be matched against a portion of the input value that starts as early as possible and extends as long as possible. Like the POSIX semantics, the first/longest semantics prescribes the whole value to be matched to the left subpattern for the two examples above. The two disambiguation policies differ if `p?` is encoded by `()|p`. In this case, `a[]?,a[]?` matches `a[]` as before if the POSIX semantics is used because the left subpattern still tries to consume as much as possible. Using the first/longest semantics, however, results in matching the first subpattern to the empty sequence since the `()` alternative of the union has the higher priority.

The XDuce disambiguation semantics [30] is an approximation of the first/longest semantics in which `p*` is encoded by `X` where `X = X,p | ()` and the first match semantics is used for union patterns. In many cases, this technique simulates the longest match semantics, but in some cases, the two disambiguation strategies produce different results [65].

These issues of disambiguation, however, are of little significance to XtaticLite pattern matching as it was introduced in Chapter 4. Because pattern matching against a single pattern is essentially a true/false test, the end result of matching—whether a `match` clause is selected or not—is independent of a particular way in which the input value is matched against the pattern. This changes as soon as patterns are extended with variable binding.

Because of this direct correlation between variable binding and ambiguity, we propose a variable-centric disambiguation semantics that selects a particular matching outcome based on how it affects variable bindings rather than on how it affects the way in which subpatterns are associated with subvalues. More specifically, we distinguish two kinds of variables: leftmost and rightmost. The former prefer to be bound as early as possible in the input value; the latter as late as possible. Consider this extension of one of the above patterns with a binder: `p = a[]?, a[]? x`. This pattern matches the same inputs as the original, but it also binds `x` to the subvalue that is matched against the right subpattern of `p`. If `x` is leftmost, then matching `a[]` against `p` results in matching the empty sequence to the left subpattern and `a[]` to the right one therefore binding `x` to `a[]`. Conversely, if `x` is rightmost, the first subpattern is matched to `a[]` and the right subpattern is matched to the empty sequence.

Using the proposed variable-centric disambiguation approach, we can simulate longest and shortest match policies by decorating appropriate subpatterns with either leftmost or rightmost variables.

In the presence of multiple variables, the proposed approach does not provide full disambiguation. Consider the pattern `p = a[], a[() x] y | a[() x] y, a[]`. When matched against `a[],a[]`, it produces two potential answers: in the first, `x` is bound to the contents of the first

element and `y` is bound to the first element; in the second, `x` is bound to the contents of the second element and `y` is bound to the second element. If `x` is leftmost and `y` is rightmost, neither outcome is better than the other unless the variables are prioritized.

In this example, a natural prioritization order can be inferred from the structure of the patter: `y` appears "earlier" than `x` in the abstract syntax tree of the pattern. In fact, `y` is always bound to a subvalue that occurs before than the subvalue associated with `x` in the left to right scan of the input value. So, in this example, it is natural to disambiguate based on `y` first, and only then based on `x` if further ambiguity remains.

In general a variable ordering cannot be inferred from the syntactic structure of the pattern. Consider the pattern `a[] x, b[] y | b[] y, a[] x` in which neither of the two variables appear before the other. This is a common type of pattern that may, for example, occur as a result of desugaring a pattern construct describing all possible interleaving of two-element `a` and `b`-labeled sequences. We would like to be able to unambiguously match against such patterns without having to specify an arbitrary variable ordering.

Even though the structure of the above pattern does not suggest a natural variable ordering, any given input value imposes a particular order in which the variables are bound. For instance, if the above pattern is matched against `a[],b[]`, variable `x` is bound before variable `y`. Conversely, if the pattern is matched against `b[],a[]`, variable `y` is bound first. Thus, our disambiguation semantics infers a different set of variable priorities for each input value.

For some patterns, even a given input does not lead to a natural variable prioritization order. An example of such a pattern is `a[] x, b[] y | a[] y, b[] x` in which `x` and `y` are completely symmetric. We say that such patterns do not have *ordered binders*; they are unfit for deterministic pattern matching and must be rejected by the compiler.

The contribution of this chapter are twofold: it gives a precise formalization of the disambiguation approach sketched above, and it defines a compilation algorithm that generates efficient single-pass matching automata implementing the proposed policy.

The rest of the chapter is as follows. Section 5.1 defines values, subvalue locations, and variable environments. Section 5.2 describes regular patterns with binders and formalizes our disambiguation policy by defining a deterministic pattern matching relation. Section 5.3 introduces tree automata with binding and defines both non-deterministic and deterministic acceptance relations for them. Section 5.4 describes deterministic matching automata with binding. Section 5.5 presents the compilation algorithm and establishes its correctness properties. Section 5.6 covers related work.

5.1 Values

This section introduces values, subvalue locations, and variable environments.

5.1.1 Definition: A *value* is either the empty sequence () or a non-empty sequence of elements, $a_1[v_1] \ldots a_k[v_k]$, each consisting of a label and a nested child value.

A non-empty sequence value can be viewed as a labeled binary tree whose root label and left and right subtrees correspond to the label of the first element, the child value of the first element, and the rest of the sequence respectively. Any subvalue then can be addressed by a binary sequence. For instance, in the value `a[c[]],b[]`, the locations 0 and 1 denote the subvalues `c[]` and `b[]` respectively.

5.1.2 Definition: A *location* is a sequence over $\{0, 1\}$; each 0 and 1 indicate the left and right subtree of the current node respectively. The empty sequence location is denoted ϵ. Given a value v and a location π within it, the corresponding subvalue is denoted v^π and is determined according to the following rules:

$$\begin{aligned} v^\epsilon &= v \\ (a[v_0], v_1)^{0\pi} &= v_0^\pi \\ (a[v_0], v_1)^{1\pi} &= v_1^\pi \end{aligned}$$

The lexicographic *smaller than* relation on locations is defined according to the following rules:

$$\begin{aligned} \epsilon &< \pi && \text{if } \pi \neq \epsilon \\ 0\pi_1 &< 1\pi_2 \\ 0\pi_1 &< 0\pi_2 && \text{if } \pi_1 < \pi_2 \\ 1\pi_1 &< 1\pi_2 && \text{if } \pi_1 < \pi_2 \end{aligned}$$

We will use $\leq$ to denote the reflexive closure of $<$.

Later, we will introduce patterns with binders and explain how values are matched against such patterns. To help define the semantics of pattern matching, we introduce variable environments which will be used as results of matching values against patterns with multiple binders.

5.1.3 Definition: A *variable environment* is a mapping from variable names to locations. The *concatenation* of a location π and an environment Σ, denoted $\pi \bullet \Sigma$, is an environment Σ_1 such that $dom(\Sigma_1) = dom(\Sigma)$ and for any $x \in dom(\Sigma)$ and $\Sigma(x) = \pi'$, we have $\Sigma_1(x) = \pi\pi'$.

When a value v is matched against a pattern p, the result is a variable environment that associates the binders occurring in p with the locations of the corresponding subvalues in v. Matching against an ambiguous pattern may result in multiple environments, and a deterministic pattern matching semantics must chose of them as the designated answer.

Comparing variable environments is based on the lexicographic ordering of locations introduced above. For some matching problems, it may be preferred that a variable be bound as early as possible in the input value; for other problems, the preference may be to bind a variable as late as possible. A set of such preferences is called a binding policy.

5.1.4 Definition: A *binding policy* for a collection of variables V is a mapping from V to $\{\mathit{left}, \mathit{right}\}$.

Given a binding policy and an ordering of variables, two variable environments can be compared as follows.

5.1.5 Definition: Let Σ_1 and Σ_2 be environments containing the same variables: $\mathit{dom}(\Sigma_1) = \mathit{dom}(\Sigma_2)$. Let $V = x_1 \ldots x_n$ be a sequence of all the variables in $\mathit{dom}(\Sigma_1)$ listed in some particular order, and let L be a binding policy for $\mathit{dom}(\Sigma_1)$. Then Σ_1 is *smaller than* Σ_2 with respect to V and L, written $\Sigma_1 <_V^L \Sigma_2$, if there exists $i \in \{1 \ldots n\}$ such that $L(x_i) = \mathit{left}$ and $\Sigma_1(x_i) < \Sigma_2(x_i)$ or $L(x_i) = \mathit{right}$ and $\Sigma_2(x_i) < \Sigma_1(x_i)$ and, furthermore, for all $j \in \{1 \ldots i-1\}$, it is the case that $\Sigma_1(x_j) = \Sigma_2(x_j)$.

For example, let Σ_1 be a map from `x` to 11 and from `y` to 0, and let Σ_2 be a map from `x` to 1 and from `y` to 01. Let V denote the variable sequence `x`, `y`. Then, $\Sigma_1 <_V^L \Sigma_2$ for any L that maps `x` to *right*, and $\Sigma_2 <_V^L \Sigma_1$ for any L that maps `x` to *left*.

5.2 Patterns with Variables

This section discusses patterns with binders and their ambiguous and deterministic semantics. We start by extending the notion of regular patterns introduced in Section 4.1 with binders.

5.2.1 Definition: *Regular patterns with binders* are described by the following grammar:

$$p \quad ::= \quad () \mid a[p] \mid p_1, p_2 \mid p_1 | p_2 \mid p\ x \mid X$$

These denote the empty sequence pattern, a labeled element pattern, sequential composition and union of two patterns, pattern with a binder, and a pattern variable. Pattern variables are introduced by top-level mutually recursive declarations of the form $\mathit{def}\ X = p$. Top-level declarations

induce a function *def* that maps variables to the associated patterns; e.g. the above declaration implies $def(X) = p$. Variable with binders are restricted to tail positions; $vars(p)$ denotes the set of all binders in p.

We require that binders appear in tail positions. For instance, the pattern `a[c[]] x, b[] y` does not meet this condition since `x` is not in a tail position. On the other hand, `a[c[] x], b[] y` is legal since `x` is bound to the whole sequence that is the sub-value of the first element. Patterns with binders can be nested: `(a[], b[] y) x`, for example, matches two-element `a`-`b`-sequences binding `x` to the whole sequence and `y` to its one-element tail.

The pattern matching relation in the presence of binders must not only indicate whether a particular value matches a particular pattern, but also return computed bindings for successful matches. We accomplish this by extending the pattern matching relation of Section 4.1 with variable environments.

5.2.2 Definition: A value v *matches* a pattern p yielding an environment Σ, written $v \in p \Rightarrow \Sigma$, if one of the following rules apply:

$$() \in () \Rightarrow \emptyset \qquad \text{(P-Emp)}$$

$$\frac{v \in p \Rightarrow \Sigma}{a[v] \in a[p] \Rightarrow 0 \bullet \Sigma} \qquad \text{(P-Elem)}$$

$$\frac{v_1 \in p_1 \Rightarrow \Sigma_1 \qquad |v_1| = k \qquad v_2 \in p_2 \Rightarrow \Sigma_2}{v_1, v_2 \in p_1, p_2 \Rightarrow \Sigma_1 \cup (1^k \bullet \Sigma_2)} \qquad \text{(P-Cat)}$$

$$\frac{v_1 \in p_1 \Rightarrow \Sigma}{v \in p_1 | p_2 \Rightarrow \Sigma} \qquad \text{(P-UniL)}$$

$$\frac{v_2 \in p_2 \Rightarrow \Sigma}{v \in p_1 | p_2 \Rightarrow \Sigma} \qquad \text{(P-UniR)}$$

$$\frac{v \in p \Rightarrow \Sigma}{v \in p\ x \Rightarrow \{x \mapsto \epsilon\} \cup \Sigma} \qquad \text{(P-Bind)}$$

$$\frac{v \in def(X) \Rightarrow \Sigma}{v \in X \Rightarrow \Sigma} \qquad \text{(P-Def)}$$

Let v_1 be the value `a[],b[]`, and let p_1 be the pattern `(a[], b[] y) x | (a[], b[] x) y`. The pattern matching rules derive the following assertions: $v_1 \in p_1 \Rightarrow \Sigma_1$ and $v_1 \in p_1 \Rightarrow \Sigma_2$ where

$$\Sigma_1 = \{x \mapsto \epsilon, y \mapsto 1\}$$
$$\Sigma_2 = \{x \mapsto 1, y \mapsto \epsilon\}$$

To compute a unique answer, we must chose the "better" environment. If we specify a binding policy and a variable ordering, we can do that by employing the comparison operator $<$ defined in the previous section. For example, let L be the binding policy $\{x \mapsto \mathit{left}, y \mapsto \mathit{right}\}$, and let V be the variable ordering x,y. Then, $\Sigma_1 <_V^L \Sigma_2$ and hence Σ_1 is the selected answer for the above pattern match. More generally:

5.2.3 Definition: Let v be a value, p a pattern, V an ordered sequence of *vars*(p), and L a binding policy for *vars*(p). Then v *unambiguously matches* p with respect to V and L yielding an environment Σ, written $v \in_V^L p \Rightarrow \Sigma$, if $v \in p \Rightarrow \Sigma$, and, for any Σ' such that $v \in p \Rightarrow \Sigma'$, it is the case that $\Sigma <_V^L \Sigma'$.

How to select a variable ordering in a non-arbitrary way is not always apparent, since binders are often independent of each other and there are no natural precedence relation on them. Consider, for example, the pattern p_2 = `(a[], b[] y) x | (b[], a[] x) y`. Variables x and y are symmetric, and so there is no reason for one to have a higher priority than the other. Such patterns commonly arise from desugaring interleaving operations, and our goal is to handle them in a principal and convenient way.

Even though there is no variable ordering that makes sense for p_2 in general, for any given value that matches p_2, there *is* a natural variable ordering. For instance, when p_2 is matched with v_1, the beginning of the x binding precedes the beginning of the y binding (since x is bound to the whole value, while y is bound to its subsequence starting from the second element.) Conversely, when p_2 is matched with the value `b[],a[]`, the y binding precedes the x binding. Patterns like p_2 are said to have ordered bindings, and for such patterns it is possible to define deterministic pattern matching without fixing variable ordering in advance.

5.2.4 Definition: Two environments Σ_1 and Σ_2 with the same domain are said to be *compatible* if, for any distinct $x, y \in \mathit{dom}(\Sigma_1)$, we have simultaneously $\Sigma_1(x) \leq \Sigma_1(y)$ and $\Sigma_2(x) \leq \Sigma_2(y)$ or $\Sigma_1(y) \leq \Sigma_1(x)$ and $\Sigma_2(y) \leq \Sigma_2(x)$.

A pattern p has *ordered binders* if, for any value v, whenever $v \in p \Rightarrow \Sigma$ and $v \in p \Rightarrow \Sigma'$, the environments Σ and Σ' are compatible. Let v be a value and p a pattern with ordered binders. Then, the *induced order* $V = x_1 \ldots x_n$ on *vars*(p) is an order satisfying the condition that for

any environment Σ such that $v \in p \Rightarrow \Sigma$ and for any i, j such that $1 \leq i < j \leq n$, we have $\Sigma(x_i) \leq \Sigma(x_j)$.

Let v be a value, p a pattern with ordered binders, and L a binding policy for $vars(p)$. Then v *unambiguously matches* p with respect to L yielding an environment Σ, written $v \in^L p \Rightarrow \Sigma$, if $v \in^L_V p \Rightarrow \Sigma$ where V is the order over $vars(p)$ induced by v.

Returning to our examples, we can see that p_1 does not have ordered bindings since the two variable environments produced when p_1 is matched with v_1 are incompatible. On the other hand, p_2 has ordered binders and we have $v_1 \in^L p_2 \Rightarrow \Sigma_1$.

5.3 Tree Automata with Binders

This section extends tree automata (Section 4.4.1) with variable binding support. We define the syntax of tree automata with binders; introduce a non-deterministic matching/acceptance relation, and show how if can be disambiguated.

5.3.1 Definition: A *tree automaton with binders* is a tuple $A = (S, T, B)$, where S is a set of states, T is a set of transitions, and B is a mapping from transitions to sets of variables. There are two types of transitions: *empty* transitions of the form $s \rightarrow ()$ and *label* transitions of the form $s \rightarrow a[s_1], s_2$, where $s, s_1, s_2 \in S$ and a is a label.

Recall pattern $p_1 =$ `(a[], b[] y) x | (a[], b[] x) y` from the previous section. It gives rise to the tree automaton $A_1 = (S_1, T_1, B_1)$ where:

$$
\begin{aligned}
S_1 &= \{s_1, s_2, s_3, s_4\} \\
T_1 &= \{t_1 = s_1 \rightarrow a[s_4], s_2, \\
&\quad\ t_2 = s_1 \rightarrow a[s_4], s_3, \\
&\quad\ t_3 = s_2 \rightarrow a[s_4], s_4, \\
&\quad\ t_4 = s_3 \rightarrow a[s_4], s_4, \\
&\quad\ t_5 = s_4 \rightarrow ()\} \\
B_1 &= \{t_1 \mapsto \{x\}, \\
&\quad\ t_2 \mapsto \{y\}, \\
&\quad\ t_3 \mapsto \{y\}, \\
&\quad\ t_4 \mapsto \{x\}, \\
&\quad\ t_5 \mapsto \emptyset\}
\end{aligned}
$$

Let us see how this automaton simulates matching of an input value against p_1. It starts in s_1 by checking the label of the first element in the input sequence. If it is a, the automaton can non-deterministically proceed by taking either transition t_1 or transition t_2. In both cases, the contents of the first element is sent to s_4 where it is verified to be the empty sequence. If t_1 is taken, variable x is bound to the location of the whole input sequence and the tail of the sequence starting from the second element is sent to s_2. If the automaton chooses t_2, the location of the whole input sequence is assigned to y and the rest of the sequence is sent to s_3. In states s_2 and s_3, the label of the first element is checked to be b, the contents of the first element is tested for emptiness, and the rest of the sequence is sent to s_4 to be tested for emptiness. Variables y and x are bound respectively to the location of the suffix of the input value starting from the second element.

Here is a formalization of the acceptance relation. We say that a value v is *accepted* by a state s yielding a variable environment Σ, written $v \in s \Rightarrow \Sigma$, if this assertion is in the least fixed point of the following rules:

$$\frac{t = s \to () \in T \qquad \Sigma = \bigcup_{x \in B(t)} (x \mapsto \epsilon)}{() \in s \Rightarrow \Sigma} \tag{TA-Emp}$$

$$\frac{\begin{array}{l} t = s \to a[s_1], s_2 \in T \qquad v_1 \in s_1 \Rightarrow \Sigma_1 \\ v_2 \in s_2 \Rightarrow \Sigma_2 \qquad \Sigma = \bigcup_{x \in B(t)} (x \mapsto \epsilon) \end{array}}{a[v_1], v_2 \in s \Rightarrow \Sigma \cup (0 \bullet \Sigma_1) \cup (1 \bullet \Sigma_2)} \tag{TA-Lab}$$

Using A_1 with value v_1 and variable environments Σ_1 and Σ_2 defined in the previous section, we have $v_1 \in s_1 \Rightarrow \Sigma_1$ and $v_1 \in s_1 \Rightarrow \Sigma_2$.

The following definition associates a tree automaton state s with the variables that are bound during a run of the automaton starting from the point when it enters s until the end of the run when the whole input value is accepted.

5.3.2 Definition: Given a tree automaton $A = (S, T, B)$ and a state $s \in S$, function $vars(s)$ denotes all the variables that may be bound as a result of matching a value against s. It is defined as the least fixed point of the following equations:

$$\begin{array}{lcl} vars(s) & = & \bigcup \{vars(t) \mid t \in T \text{ and } s \text{ is the source state of } t\} \\ vars(t = s \to ()) & = & B(t) \\ vars(t = s \to a[s_1], s_2) & = & B(t) \cup vars(s_1) \cup vars(s_2) \end{array}$$

Now, like we did in the previous section, we can define deterministic acceptance with respect to a variable ordering and a binding policy.

5.3.3 Definition: Let v be a value, p a state, V an ordered sequence of $vars(s)$, and L a binding policy for $vars(s)$. Then v is *unambiguously accepted* by s with respect to V and L yielding an environment Σ, written $v \in_V^L s \Rightarrow \Sigma$, if $v \in s \Rightarrow \Sigma$, and, for any Σ' such that $v \in s \Rightarrow \Sigma'$, it is the case that $\Sigma <_V^L \Sigma'$.

Again mirroring the developments of the previous section, we define the notion of ordered binders for tree automaton states and introduce unambiguous acceptance with respect to a binding policy.

5.3.4 Definition: A state s has *ordered binders* if, for any value v, whenever $v \in s \Rightarrow \Sigma$ and $v \in s \Rightarrow \Sigma'$, the environments Σ and Σ' are compatible. Let v be a value and s a state with ordered binders. Then, the *induced order* $V = x_1 \ldots x_n$ on $vars(s)$ is an order satisfying the condition that for any environment Σ such that $v \in s \Rightarrow \Sigma$ and for any i, j such that $1 \leq i < j \leq n$, we have $\Sigma(x_i) \leq \Sigma(x_j)$.

Let v be a value, s a state with ordered binders, and L a binding policy for $vars(s)$. Then v is *unambiguously accepted* by s with respect to L yielding an environment Σ, written $v \in^L s \Rightarrow \Sigma$, if $v \in_V^L s \Rightarrow \Sigma$ where V is the order over $vars(s)$ induced by v.

In the rest of the chapter we assume that both source patterns and tree automata that we are dealing with have ordered binders.

5.4 Matching Automata with Binders

This section extends matching automata with binding operations. Unlike tree automata, we only consider deterministic matching automata with binders building on the SNBF model described in Section 4.5.

Other than binding operations, there are two aspects that distinguish matching automata of this section from the matching automata of Chapter 4. The first one is that we only consider matching automata in simple non-backtracking form omitting a more general account. The second is that we have a different form of subroutine transitions: instead of allowing just one subroutine call per transition and having non-final subroutine states that may have other subroutine states as descendants, we require subroutine states to be final and allow subroutine transition to employ multiple subroutine calls each associated with its own result conversion relation. This approach

```
def p1 = a[]*, a[]? x
def p2 = (a[],a[])*, a[]? x
def p3 = a[]*, a[]? y
def p4 = (a[],b[])*, a[]? y

match u with
  | a[p1],p3 | a[p2],p4 → 1
  | a[p1],p4 → 2
```

(a)

$$
\begin{array}{lcl}
A & = & (S, q_1, T, X, L) \\
S & = & \{q_1, q_2\} \\
T & = & \{q_1(u) : a[v], w \xrightarrow{I} q_2, \\
 & & \quad q_2 \xrightarrow{\sigma} \{A_1(v@0) \mapsto \sigma_1, A_2(w@1) \mapsto \sigma_2\} \\
X & = & \{q_1 \mapsto u\} \\
L & = & \{x \mapsto \mathit{left}, y \mapsto \mathit{left}\} \\
I & = & \{1, 2\} \\
\sigma & = & \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 2\} \\
\sigma_1 & = & \{1 \mapsto 1, 1 \mapsto 3, 2 \mapsto 2\} \\
\sigma_2 & = & \{1 \mapsto 1, 2 \mapsto 2, 2 \mapsto 3\}
\end{array}
$$

(b)

Figure 5.1: A `match` expression (a) and the corresponding matching automaton with binders (b)

allows us to avoid index converters (Section 4.4.3) and gives us a more convenient framework for dealing with binding.

5.4.1 Definition: A *deterministic matching automaton with binders* is a tuple (S, q, T, X, L), where S is a set of states, $q \in S$ is a start state, T is a set of transitions, X is a mapping from states to registers, and L is a binding policy.

There are three kinds of transitions: *final*, *simple* and *subroutine*. They have the following structure:

$$
\begin{array}{ll}
q \to B & \text{(final)} \\
q(r) : p \xrightarrow{I} q_2 & \text{(simple)} \\
q \xrightarrow{\sigma} \{A_1(r_1@\pi_1) \mapsto \sigma_1 \ldots A_k(r_k@\pi_k) \mapsto \sigma_k\} & \text{(subroutine)}
\end{array}
$$

A final transition contains a *source state* q, a register r associated with q via the mapping X, and a partial mapping from results to variable environments B. A simple transition contains a source state q, the associated register r, a target language pattern p—which can be of the form () or $a[r_1], r_2$—a set of integer results I, and a destination state q_2. A subroutine transition contains a source state q, a binary relation on results σ, and a set of subroutine calls each containing a subroutine matching automaton's name, a register paired with the location of the register's contents in the original input value, and a binary relation on results.

To illustrate matching automata with binders, let us consider a somewhat contrived example shown in Figure 5.1. It contains a XtaticLite fragment defining patterns $\mathtt{p}_1$, $\mathtt{p}_2$, $\mathtt{p}_3$, and $\mathtt{p}_4$ and a

`match` expression based on them. The patterns of both clauses have ordered binders—x is always bound before y in the document order—therefore, unambiguous pattern matching as introduced in Definition 5.2.4 is possible.

Let $\mathtt{v}_1 = \mathtt{a[]}$, and let $\mathtt{v}_2 = \mathtt{a[],a[]}$. Evaluating the `match` expression with the value $\mathtt{a[v}_1\mathtt{],v}_2$ results in a successful match for the patterns of both clauses. The pattern of the first clause yields the following potential answers:

$$\Sigma_1 = \{x \mapsto 0, y \mapsto 11\}$$
$$\Sigma_2 = \{x \mapsto 0, y \mapsto 111\}$$
$$\Sigma_3 = \{x \mapsto 01, y \mapsto 11\}$$
$$\Sigma_4 = \{x \mapsto 01, y \mapsto 111\}$$

Let L be the binding policy in which both x and y are mapped to *left*. According to L, the disambiguated answer is Σ_1. The pattern of the second clause yields two potential answers Σ_2 and Σ_4; so, according to L, the disambiguated answer is Σ_2.

The matching automaton shown in Figure 5.1(b) is an implementation of the displayed `match` expression. It starts in the initial state q_1 where it inspects the input value stored in register u. It verifies that the first element is a-labeled and decomposes the input by storing the contents of the first element in v and the suffix of the sequence starting from the second element in w. The matching automaton then transitions into q_2 where subroutine matching automata A_1 and A_2 are invoked on v and w respectively. Matching automaton A_1 checks its input against p_1 and p_2 and, depending on which of them matches, returns results 1 and 2 together with the variable environments obtained as a result of matching against the successful pattern. Matching automaton A_2 performs similar operations for patterns p_3 and p_4. When calls to the subroutine matching automata return, we combine the results using result relations σ_1, σ_2, and σ to obtain the final result. Here is a formalization of matching automata acceptance rules.

5.4.2 Definition: Let E be a register environment—a mapping from registers to values. We write $E{\downarrow}r$ for an environment which agrees with E on r and is undefined on all other registers, and $E\backslash y$ for an environment which is undefined on y and otherwise equal to E. Let $\mathcal{M}$ be a mapping from names to subroutine matching automata. We say that a register environment E is *accepted* by a state q yielding a result j and a variable environment Σ, written $E \in q \Rightarrow j \otimes \Sigma$, if the concluding assertion is in the least fixed point defined by the following rules. To ensure determinism, we require that transitions be *disjoint*—each state can only be the source of either several non-overlapping simple transitions, a single final transition, or a single subroutine transition.

$$\frac{q \to B \in T \qquad k \in dom(B) \qquad \Sigma = B(k)}{E \in q \Rightarrow k \otimes \Sigma} \tag{MA-Fin}$$

$$\frac{\begin{array}{c} q(r) : () \xrightarrow{I} q_2 \in T \qquad E(r) = () \qquad k \in I \\ E' = E \backslash r \qquad E' \in q_2 \Rightarrow k \otimes \Sigma \end{array}}{E \in q \Rightarrow k \otimes \Sigma} \tag{MA-Emp}$$

$$\frac{\begin{array}{c} q(r) : a[r_1], r_2 \xrightarrow{I} q_2 \in T \qquad E(r) = a[v_1], v_2 \qquad k \in I \\ E' = (E \backslash r)[v_1/r_1, v_2/r_2] \qquad E' \in q_2 \Rightarrow k \otimes \Sigma \end{array}}{E \in q \Rightarrow k \otimes \Sigma} \tag{MA-Lab}$$

$$\frac{\begin{array}{c} q \xrightarrow{\sigma} \{A_1(r_1@\pi_1) \mapsto \sigma_1 \ldots A_m(r_m@\pi_m) \mapsto \sigma_m\} \in T \\ Envs = \{\Sigma' \mid \begin{array}{c} \forall\, i \in \{1 \ldots m\}.\ (E{\downarrow}r_i \in \mathcal{M}(A_i) \Rightarrow j_i \otimes \Sigma_i \text{ and } (j_i, h) \in \sigma_i) \\ \Sigma' = (\pi_1 \bullet \Sigma_1) \cup \ldots \cup (\pi_m \bullet \Sigma_m) \end{array} \text{ and } (h, k) \in \sigma\} \\ \forall \Sigma_1 \Sigma_2 \in Envs.\ \Sigma_1 \text{ is compatible with } \Sigma_2 \qquad \Sigma = \min^L(Envs) \end{array}}{E \in q \Rightarrow k \otimes \Sigma} \tag{MA-Sub}$$

Returning to our example, let us see how these rules derive the assertion $E \in q_1 \Rightarrow 1 \otimes \Sigma_1$ where Σ_1 is the variable environment mentioned above in the context of the XTATICLITE program, and E is the register environment mapping u to the value `a[`$\mathtt{v}_1$`],`$\mathtt{v}_2$ with $\mathtt{v}_1 =$ `a[]` and $\mathtt{v}_2 =$ `a[],a[]`. The above assertion follows by MA-Lab from $E_1 \in q_2 \Rightarrow 1 \otimes \Sigma_1$ where $E_1 = \{v \mapsto v_1, w \mapsto v_2\}$. This assertion is derived from two pairs of subroutine assertions. The first pair is $E_2 \in \mathcal{M}(A_1) \Rightarrow 1 \otimes \Delta_1$ and $E_3 \in \mathcal{M}(A_1) \Rightarrow 1 \otimes \Delta_2$ where:

$$E_1 = \{v \mapsto v_1\}$$
$$E_2 = \{w \mapsto v_2\}$$
$$\Delta_1 = \{x \mapsto \epsilon\}$$
$$\Delta_2 = \{y \mapsto 1\}$$

Combining Δ_1 and Δ_2 as specified by the rule MA-Sub, we obtain the variable environment Σ_1. The result mapping relations σ_1, σ_2, and σ serve as identities in this case mapping 1 to itself.

The second pair of subroutine assertions that can be used to derive the above assertion for q_2 contains $E_2 \in \mathcal{M}(A_1) \Rightarrow 2 \otimes \Delta_3$ and $E_3 \in \mathcal{M}(A_1) \Rightarrow 2 \otimes \Delta_4$ where E_1 and E_2 are as before and the resulting variable environments are as follows:

$$\Delta_3 = \{x \mapsto \epsilon\}$$
$$\Delta_4 = \{y \mapsto 11\}$$

Combining Δ_3 and Δ_4 produces the variable environment Σ_2. Notice how 2 returned by A_1 is mapped to itself by σ_1; similarly, 2 returned by A_2 is mapped to itself by σ_2, and, finally, the 2 obtained from these two mappings is converted to the overall answer 1 by the global result converter σ.

Now, since Σ_1 and Σ_2 are compatible, we can find the smaller of the two according to binding policy L. This produces Σ_1 as the variable environment used in the concluding assertion of MA-Sub.

5.5 Compilation Algorithm

This section describes an algorithm for generating matching automata with binders. It is an extension of the compilation algorithm presented in Chapter 4. Here, we give a staged presentation in which the algorithm maintain an incomplete version of the matching automaton with some states represented by configurations. It gradually expands these configurations replacing them with newly generated ordinary states and possibly more configurations. When this process terminates, the resulting matching automaton—composed entirely of ordinary states—is an implementation of the original matching problem.

First, we give a revised definition of configurations (Section 4.5) extended with variable binding.

5.5.1 Definition: A *configuration* over a tree automaton with binders comprises a tuple of distinct registers associated with locations $(r_1@\pi_1 \ldots r_n@\pi_n)$ and a set of tuples $\{(s_{11} \ldots s_{1n}, \Sigma_1, j_1)$ $\ldots$ $(s_{m1} \ldots s_{mn}, \Sigma_m, j_m)\}$, each associating a collection of TA's states to a variable environment and a result. We depict a configuration as follows:

$$C = \begin{array}{|lll|l|} \hline \mathtt{x}_1 & \ldots & \mathtt{x}_n & \\ \hline \mathtt{s}_{11} & \ldots & \mathtt{s}_{1n} & \mathtt{j}_1 \\ & \ldots & & \\ \mathtt{s}_{m1} & \ldots & \mathtt{s}_{mn} & \mathtt{j}_m \\ \hline \end{array}$$

Given such a configuration C, we define these auxiliary functions:

$$
\begin{array}{lcl}
\mathit{registers}(C) & = & \{r_1 \ldots r_n\} \\
\mathit{envs}(C) & = & \{\Sigma_1 \ldots \Sigma_m\} \\
\mathit{results}(C) & = & \{j_1 \ldots j_m\} \\
\mathit{vars}(C) & = & \bigcup_{i \in \{1 \ldots m\}} (\mathit{dom}(\Sigma_i) \cup \mathit{vars}(s_{i1}) \cup \ldots \cup \mathit{vars}(s_{in}))
\end{array}
$$

We say that a register environment E is accepted by C yielding a result j_r and a variable environment Σ, written $E \in C \Rightarrow j_r \otimes \Sigma$, if there exist variable environments $\Sigma_{r1} \ldots \Sigma_{rn}$ such that $E(r_i) \in s_{ri} \Rightarrow \Sigma_{ri}$ for all $i \in \{1 \ldots n\}$ and $\Sigma = \Sigma_r \cup (\pi_1 \bullet \Sigma_{r1}) \cup \ldots \cup (\pi_n \bullet \Sigma_{rn})$.

The acceptance relation introduced above is non-deterministic since it is defined in terms of the non-deterministic acceptance relation for tree automata with binders. Let us go through the familiar steps of arriving at a deterministic relation. First, we define acceptance with respect to a variable ordering and a binding policy.

5.5.2 Definition: Let C be configuration, E a register environment on C's registers, L a binding policy for $\mathit{vars}(C)$, and V an ordered sequence of $\mathit{vars}(C)$. Then E is *unambiguously accepted* by C with respect to L and V yielding a result j and a variable environment Σ, written $E \in^L_V C \Rightarrow j \otimes \Sigma$, if $E \in C \Rightarrow j \otimes \Sigma$ and, for any Σ' such that $E \in C \Rightarrow j \otimes \Sigma'$, it is the case that $\Sigma <^L_V \Sigma'$.

The next step is to identify a subset of configurations with ordered binders and define deterministic acceptance with respect to a binding policy.

5.5.3 Definition: A configuration of the above form has *ordered binders* if, for any result j and register environment E defined on $r_1 \ldots r_n$, whenever $E \in C \Rightarrow j \otimes \Sigma$ and $E \in C \Rightarrow j \otimes \Sigma'$, the environments Σ and Σ' are compatible. Let E be a register environment defined on $r_1 \ldots r_n$ and k a result. The *induced order* $V = x_1 \ldots x_n$ on $\mathit{vars}(C)$ is an order satisfying the condition that, for any environment Σ with $E \in C \Rightarrow k \otimes \Sigma$ and i, j with $1 \leq i < j \leq n$, we have $\Sigma(x_i) \leq \Sigma(x_j)$.

Let C be configuration with ordered binders, k a result, E a register environment on $\mathit{registers}(C)$, and L a binding policy for $\mathit{vars}(C)$. Then E is *unambiguously accepted* by C with respect to L yielding a result k and a variable environment Σ, written $E \in^L C \Rightarrow k \otimes \Sigma$, if $E \in^L_V C \Rightarrow k \otimes \Sigma$ where V is the order over $\mathit{vars}(C)$ induced by E and k.

We now introduce a version of matching automata in which configurations can appear as pseudo-states. We will use such matching automata during intermediate steps of the compilation algorithm as "incomplete" results. We will formalize the process of pseudo-state/configuration expansion that allows us to convert the current incomplete matching automaton into a slightly less incomplete matching automaton.

5.5.4 Definition: An *incomplete deterministic matching automaton with binders* over a tree automaton A is a tuple (S, K, q, T, X, L), where S is a set of states, q is a start state, T is a set of transitions, X is a mapping from states to registers, and L is a binding policy as in matching automata (Definition 5.4.1). Additionally, K is a set of pseudo-states represented by configurations with ordered binders over A. The start state s can be either an ordinary state $s \in S$ or a pseudo-state $s \in K$.

The form of transitions is similar to that of matching automaton transitions except that simple transitions can have pseudo-states as their destinations and subroutine transitions can refer to incomplete matching automata. Pseudo-states cannot appear as sources of transitions.

The semantics of transitions is defined as in the matching automaton case, except that, whenever a configuration C appears as the destination state of a simple transition, the judgment for unambiguous configuration acceptance, $E \in^L C \Rightarrow k \otimes \Sigma$, is used instead of the judgment for state acceptance, $E \in q \Rightarrow k \otimes \Sigma$. Here, for example, is the adaption of the MA-Emp rule for incomplete matching automata.

$$\frac{t = q(r) : () \xrightarrow{I} s \in T \qquad E(x) = () \qquad k \in I \qquad s \in S \qquad E' = E \backslash r \qquad E' \in s \Rightarrow k \otimes \Sigma}{E \in q \Rightarrow k \otimes \Sigma} \quad \text{(IMA-Emp1)}$$

$$\frac{t = q(r) : () \xrightarrow{I} s \in T \qquad E(x) = () \qquad k \in I \qquad s \in K \qquad E' = E \backslash r \qquad E' \in^L s \Rightarrow k \otimes \Sigma}{E \in q \Rightarrow k \otimes \Sigma} \quad \text{(IMA-Emp2)}$$

Recalling the example of the previous section, consider this incomplete matching automaton that may serve as the starting point for generating matching automaton A of Figure 5.1: $M_1 = (\emptyset, \{C_1\}, C_1, \emptyset, \emptyset, L)$ where $L = \{x \mapsto \mathit{left}, y \mapsto \mathit{left}\}$ and C is the following *initial* configuration:

$C_1 =$

$\mathtt{u}@\epsilon$		
$\mathtt{a[p_1],p_3 \mid a[p_2],p_4}$	$\emptyset$	1
$\mathtt{a[p_1],p_4}$	$\emptyset$	2

For brevity, we display configurations with source patterns instead of tree automaton states; this allows us to omit discussing the tree automaton corresponding to our example matching problem.

We now refine expansion by label and expansion by state originally defined in Chapter 4 to take into account binding information.

5.5.5 Definition: Let $A = (S, T, B)$ be a tree automaton and C a configuration over A consisting of register-location pairs $(r_1@\pi_1 \ldots r_n@\pi_n)$ and tuples $\{(s_{11} \ldots s_{1n}, \Sigma_1, j_1) \ldots (s_{m1} \ldots s_{mn}, \Sigma_m, j_m)\}$. Let c be a column in C identified by r_c and let p be a target language pattern. An *expansion* of C based on c and p, denoted $\mathit{expand}(C, c, p)$, is a configuration C' such that

1. if $p = ()$, then

$$C' = \begin{array}{|cccccc|c|c|} \hline r_1@\pi_1 & \ldots & r_{c-1}@\pi_{c-1} & r_{c+1}@\pi_{c+1} & \ldots & r_n@\pi_n & & \\ \hline s_{k_1 1} & \ldots & s_{k_1(c-1)} & s_{k_1(c+1)} & \ldots & s_{k_1 n} & \Sigma_{k_1} \cup \Sigma'_1 & j_{k_1} \\ & & & \ldots & & & & \\ s_{k_i 1} & \ldots & s_{k_i(c-1)} & s_{k_i(c+1)} & \ldots & s_{k_i n} & \Sigma_{k_i} \cup \Sigma'_i & j_{k_i} \\ \hline \end{array}$$

   where $\{(k_1, \Sigma'_1) \ldots (k_i, \Sigma'_i)\} = \{(k, B(t) \to \pi_c) \mid t = s_{kc} \to () \in T\}$ and $B(t) \to \pi$ denotes the variable environment $\{x \to \pi \mid x \in B(t)\}$,

2. if $p = a[z], y$ for some label a and registers $z, y \notin \mathit{registers}(C) \backslash \{r_c\}$, then

$$C' = \begin{array}{|cccccccc|c|c|} \hline z@0\pi_c & y@1\pi_c & r_1@\pi_1 & \ldots & r_{c-1}@\pi_{c-1} & r_{c+1}@\pi_{c+1} & \ldots & r_n@\pi_n & & \\ \hline s'_{11} & s''_{11} & s_{11} & \ldots & s_{1(c-1)} & s_{1(c+1)} & \ldots & s_{1n} & \Sigma_1 \cup \Sigma'_{11} & j_1 \\ & & & & \ldots & & & & & \\ s'_{1k_1} & s''_{1k_1} & s_{11} & \ldots & s_{1(c-1)} & s_{1(c+1)} & \ldots & s_{1n} & \Sigma_1 \cup \Sigma'_{1k_1} & j_1 \\ & & & & \vdots & & & & & \\ s'_{m1} & s''_{m1} & s_{m1} & \ldots & s_{m(c-1)} & s_{m(c+1)} & \ldots & s_{mn} & \Sigma_m \cup \Sigma'_{m1} & j_m \\ & & & & \ldots & & & & & \\ s'_{mk_m} & s''_{mk_m} & s_{m1} & \ldots & s_{m(c-1)} & s_{m(c+1)} & \ldots & s_{mn} & \Sigma_m \cup \Sigma'_{mk_m} & j_m \\ \hline \end{array}$$

   where $\{(s'_{i1}, s''_{i1}, \Sigma'_{i1}) \ldots (s'_{ik_i}, s''_{ik_i}, \Sigma'_{ik_i})\} = \{(s', s'', B(t) \to \pi_c) \mid t = s_{ic} \to a[s'], s'' \in T\}$ for $i \in \{1 \ldots m\}$.

Configuration C_1 above can be expanded on its only column using the target language pattern `a[v],w` yielding the following residual configuration:

$$C_2 = \begin{array}{|cc|c|c|} \hline \texttt{v@0} & \texttt{w@1} & & \\ \hline \texttt{p}_1 & \texttt{p}_3 & \emptyset & 1 \\ \texttt{p}_2 & \texttt{p}_4 & \emptyset & 1 \\ \texttt{p}_1 & \texttt{p}_4 & \emptyset & 2 \\ \hline \end{array}$$

Expanding a configuration by all the target language patterns that can be generated from the tree automaton states of the selected column gives us a procedure for turning a configuration into an ordinary matching automaton state.

5.5.6 Definition: Let $M = (S, K, q_0, T, X, L)$ be an incomplete matching automaton, and let $C \in K$ be a configuration with at least one column. Let c be a column of C corresponding to register r, and let z and y be some registers such that $z, y \notin \mathit{registers}(C) \backslash \{r\}$. Let $\{p_1 \ldots p_k\} = \{() \mid s \in c \text{ and } s \to () \in T\} \cup \{a[z], y \mid s \in c \text{ and } \exists\ s', s''.\ s \to a[s'], s'' \in T\}$. Let $C_i = \mathit{expand}(C, c, p_i)$ and $I_i = \mathit{results}(C_i)$ for each $i \in \{1 \ldots k\}$. Let q be a fresh matching automaton state such that $q \notin S$.

A *simple expansion* of M with respect to C and c is an incomplete matching automaton in which the pseudo-state C is replaced by q, the transitions with C as the destination state are replaced by the corresponding transitions with q as the destination state, and new transitions of the form $q(x) : p_i \xrightarrow{I_i} C_i$ are added for $i \in \{1 \ldots k\}$.

Applying simple expansion to M_1 with respect to C_1 yields this incomplete matching automaton: $M_2 = (S_2, K_2, q_1, T_2, X_2, L)$ where:

$$\begin{aligned} S_2 &= \{q_1\} \\ T_2 &= \{q_1(u) : a[v], w \xrightarrow{I} C_2\} \\ X_2 &= \{q_1 \mapsto u\} \\ I &= \{1, 2\} \end{aligned}$$

The following proposition states that simple expansion preserves the semantics of the original incomplete matching automaton.

5.5.7 Proposition: Let M be an incomplete matching automaton with a matching policy L, let C be one of M's pseudo-states, and let c be one of C's columns. Let M' be the simple expansion of M with respect to C and c, and let q be a concrete state in M' corresponding to pseudo-state C. Then, for any register environment E, result k, and variable environment Σ, we have $E \in^L C \Rightarrow k \otimes \Sigma$ iff $E \in q \Rightarrow k \otimes \Sigma$.

Proof: Follows directly from the definitions. Notice that expansion of a configuration with ordered binders via *expand* produces a residual configuration with ordered bindings. This guarantees that an incomplete matching automaton generated by simple expansion is well-defined since all of its new pseudo-states are configurations with ordered binders. □

The second kind of configuration expansion allows us to expand configurations all of whose columns contain loop breakers (Section 4.5.3.)

5.5.8 Definition: Let $M = (S, K, q_0, T, X, L)$ be an incomplete matching automaton, and let $C \in K$ be a configuration consisting of register-location pairs $(r_1@\pi_1 \ldots r_n@\pi_n)$ and tuples $\{(s_{11} \ldots s_{1n}, \Sigma_1, j_1) \ldots (s_{m1} \ldots s_{mn}, \Sigma_m, j_m)\}$. Let σ be a result mapping relation $\{1 \mapsto j_1 \ldots m \mapsto j_m\}$. Suppose that column i contains k distinct tree automaton states $s_1 \ldots s_k$, and let σ_i be a result mapping relation that associates a position in this list with the positions of the corresponding state in the original column. Let C_i be a configuration composed of registers-location pair $r_i@\epsilon$ and tuples $\{(s_1, \emptyset, 1) \ldots (s_k, \emptyset, k)\}$, and let A_i be the incomplete matching automaton $(\emptyset, \{C_i\}, C_i, \emptyset, \emptyset, L)$. Let q be a fresh matching automaton state such that $q \notin S$.

A *subroutine expansion* of M with respect to C is an incomplete matching automaton in which the pseudo-state C is replaced by q, the transitions with C as the destination state are replaced by the corresponding transitions with q as the destination state, and a new transition $q \xrightarrow{\sigma} \{A_1(r_1@\pi_1) \mapsto \sigma_1 \ldots A_k(r_k@\pi_k) \mapsto \sigma_k\}$ is added.

Incomplete matching automaton M_2, for example, can be expanded with respect to C_2 yielding matching automaton A displayed in Figure 5.1. In this expansion we assume that subroutine matching automata A_1 and A_2 are based on the tree automaton states corresponding to the sequences $\mathtt{p}_1\mathtt{,p}_2$ and $\mathtt{p}_3\mathtt{,p}_4$ respectively. Note how σ_1 and σ_2 are derived from these two sequences and the positions of the respective tree automaton states in C_2's columns.

The following proposition establishes that subroutine expansion preserves the semantics of the incomplete matching automaton.

5.5.9 Proposition: Let M be an incomplete matching automaton with a matching policy L, let C be one of M's pseudo-states, and let i be one of C's columns. Let M' be the subroutine expansion of M with respect to C, and let q be a concrete state in M' corresponding to pseudo-state C. Then, for any register environment E, result k, and variable environment Σ, we have $E \in^L C \Rightarrow k \otimes \Sigma$ iff $E \in q \Rightarrow k \otimes \Sigma$.

The third kind of configuration expansion is applicable to configurations with no columns.

5.5.10 Definition: Let $M = (S, K, q_0, T, X, L)$ be an incomplete matching automaton and $C \in K$ with no columns; i.e., consisting of the empty collection of register-location pairs and a set of tuples of the form $\{(\Sigma_1, j_1) \ldots (\Sigma_m, j_m)\}$. Let B be a mapping from results to variable environments $B(k) = min^L\{\Sigma_i \mid i \in \{1 \ldots m\} \text{ and } j_i = k\}$. Let q be a fresh matching automaton state such that $q \notin S$.

A *final expansion* of M with respect to C is an incomplete matching automaton in which the pseudo-state C is replaced by q, the transitions with C as the destination state are replaced by the corresponding transitions with q as the destination state, and a new transition $q \rightarrow B$ is added.

Like the other two kinds of expansion, final expansion preserves the semantics of the underlying incomplete matching automaton.

5.5.11 Proposition: Let M be an incomplete matching automaton with a matching policy L, and let C be one of M's pseudo-states with no columns. Let M' be the final expansion of M with respect to C, and let q be a concrete state in M' corresponding to pseudo-state C. Then, for any register environment E, result k, and variable environment Σ, we have $E \in^L C \Rightarrow k \otimes \Sigma$ iff $E \in q \Rightarrow k \otimes \Sigma$.

Proof: Follows directly from the definitions. □

At the top level, the compilation algorithm proceeds as described in Section 4.5. Given a collection of tree automaton states $s_1 \ldots s_m$ corresponding to the patterns of some `match` expression, the algorithm builds an initial configuration containing these states associated with results $1 \ldots m$ respectively. This configuration has ordered binders since the underlying tree automaton states have ordered binders and each column in a configuration is associated with a unique result.

The algorithm constructs an incomplete matching automaton consisting of the initial configuration as its start state and its only pseudo-state. The algorithm then repeats the following sequence of steps: it selects one of the remaining pseudo-states in the current incomplete matching automaton and applies one of the three expansion techniques depending on the structure of the selected pseudo-state. If it is a configuration with no columns, the algorithm uses final expansion. If this is the first expansion step and the selected configuration is initial, the algorithm uses simple expansion with respect to its first column. Otherwise, if the selected configuration has a column with no loopbreakers, the algorithm uses simple expansion with respect to the first such column. Otherwise, the algorithm applies subroutine expansion. The algorithm proceeds in this way until no more pseudo-states remain.

Like in Section 4.5, we can show that the above algorithm terminates on all inputs. Combining this termination property with properties 5.5.11, 5.5.7, and 5.5.9, we can conclude that the algorithm generates a deterministic matching automaton equivalent to the original matching problem and implementing the given matching policy.

5.6 Related Work

The starting point of our research was the treatment of ambiguous pattern matching in XDuce [35, 36, 37]. Different implementations of XDuce chose one of the two approaches with respect to disambiguation: one prohibiting ambiguous patterns altogether; the other using the first match semantics

in which repetition patterns are encoded in terms of recursive patterns and union. The former approach is not very practical since most patterns are naturally ambiguous and rewriting them into unambiguous patterns would result in bloated code. The disambiguation semantics used in the latter approach [30] is not declarative and is difficult to understand. Unlike the disambiguation semantics proposed in this chapter, XDuce's approach does not provide an easy way of predicting the outcome of pattern matching just from looking at the overall structure of the pattern.

XDuce's implementation is not concerned with the run-time efficiency of its pattern matcher. In particular, XDuce employs a backtracking interpreter that does not guarantee linear pattern matching. XDuce's type checker, however, does a good job of inferring precise types for pattern variables reflecting the first match disambiguation semantics in the type inference engine. Xtatic, on the other hand, does not support type inference.

CDuce [19, 4] also features the first match disambiguation semantics. In additional to simple variable binding operators as described here, CDuce supports more sophisticated binders that may appear inside repetition and comprehension patterns. As a result, CDuce's implementation binding is quite intricate.

In a related development [18], Alain Frisch formalizes CDuce's disambiguation approach in the framework of pattern matching with heterogeneous values rather than homogeneous element sequences. More specifically, Frisch shows how a flat sequence value can be matched—in linear time—against a pattern producing as a result a structured value reflecting the disambiguation choices made during pattern matching.

Probably the most closely related project is described by Ville Laurikari [49]. He presents a linear one-pass pattern matching algorithm that implements the POSIX disambiguation semantics for strings. The formalism described in this work is developed for string regular expression matching and subsequent addressing of the matched fragments. As such, it is not directly applicable to our problem of compiling `match` expressions.

Stijn Vansummeren gives a thorough overview [65] of unambiguous pattern matching. He formally defines the POSIX and first/longest match disambiguation semantics both for strings and for trees and presents sound and complete type inference algorithms for all four situations. His work focuses on type inference and is not concerned with efficient pattern matching algorithms.

Chapter 6

Type-Based Optimization

A significant challenge in compiling languages with regular patterns is understanding how to translate regular pattern matching expressions into a low-level target language efficiently and compactly. One powerful class of techniques that can help achieve this goal relies on using static type information to generate optimized pattern matching code. The work described here aims to integrate type-based optimization techniques with the high-performance, but type-insensitive, compilation methods described in Chapter 4.

A simple example shows the benefits of using type information during compilation. Figure 6.1 shows how a high-level source program (a) can be compiled into a low-level target program (b) without taking the input type into account. The first source pattern, `Any,a[]`, matches sequences composed of an arbitrary prefix matching `Any` followed by an `a`-tagged element with the empty contents matching `a[]`. In a low-level target language, this pattern can be implemented by a recursive function that walks the input sequence from the beginning to the end and checks the tag and the contents of the last element. This is precisely the behavior of the procedure in Figure 6.1(b). The second clause of the `case` expression, for example, uses the pattern `a[x],y` to check whether the first element in the input sequence is tagged by `a`; then, it employs two nested `case` expressions to ensure that both the contents of the first element and the rest of the sequences are empty. If either is non-empty, the same procedure is invoked recursively on the rest of the input sequence.

However, suppose we know that the input type to the `match` expression is `T = a[],(a[]|b[])`; i.e., only two-element sequences whose first element is tagged by `a` and has the empty contents, and whose second element is tagged by either `a` or `b` and also has the empty contents can be used as pattern matching input. The program shown is Figure 6.1(b) works correctly for this input type, but we can do much better. First, there is no longer any need for the recursive loop, since the input

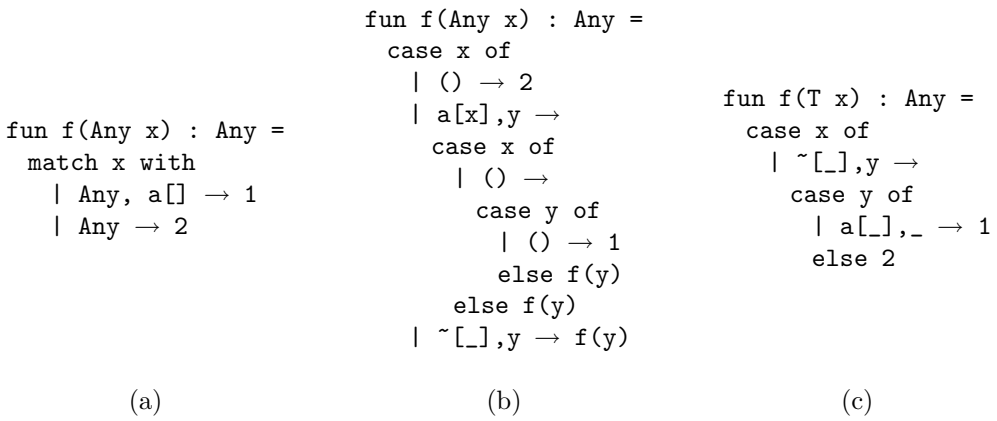


Figure 6.1: A source program (a); an equivalent target program (b); a target program for a restricted input type `T = a[],(a[]|b[])` (c)

sequence is known to contain exactly two elements, and we can simply skip the first element and examine the second. Furthermore, it is unnecessary to check whether the contents of the second element is empty, since this is prescribed by the input type. The optimal (in terms of both size and speed) target program corresponding to input type `T` is shown in Figure 6.1(c).

The contributions of this chapter are as follows:

- In Section 6.2, we present the efficient type-based compilation algorithm and some preliminary measurements that demonstrate the algorithm's effectiveness (compared with the type-insensitive compilation method).

- In Section 6.3, we introduce and justify an optimality criterion that lets us formally compare the efficiency of pattern matching code in target language programs. In Section 6.4, we demonstrate that optimal compilation is possible, in principle, for matching problems with non-recursive patterns, by presenting a refinement of the above algorithm that produces optimal target code for this case. (The refined algorithm is too inefficient for use in a real compiler; finding a lower bound on the complexity of optimal compilation is left as future work.) In Section 6.5, we generalize this algorithm to the case with recursive patterns.

Section 6.6 discusses related work, in particular the *non-uniform automata* [16] used in Frisch's implementation of CDuce [4].

6.1 Background

The purpose of this section is to recapitulate some of the essential definitions from previous chapters. It talks about values, regular patterns, tree automata, matching automata, and configurations.

6.1.1 Values

A *value* is either the empty sequence `()` or a non-empty sequence of elements, $\texttt{a}_1\texttt{[v}_1\texttt{]} \ldots \texttt{a}_k\texttt{[v}_k\texttt{]}$, each consisting of a label and a nested child value. Values represent fragments of XML documents. For example, the XML fragment `<person><name><john/> </name><age><two/></age></person>` is encoded by the value `person[name[john[]],age[two[]]]`.

In the rest of the chapter, it will be convenient to view values as binary trees. The empty sequence value `()` corresponds to the empty binary tree ϵ. A non-empty sequence value $\texttt{a[v}_1\texttt{],v}_2$ corresponds to the labeled binary tree $a(t_1, t_2)$ whose root label and left and right subtrees correspond to the label of the first element, the child value of the first element, and the rest of the sequence respectively.

We use environments mapping variables to values. We write $E[v_1/x, v_2/y]$ to denote an environment mapping x to v_1 and y to v_2 and agreeing with E on all other variables and $E\backslash y$ to denote an environment which is undefined on y and otherwise equal to E.

It is possible to determine the outcome of many matching problems without traversing the whole input value. The fewer the number of nodes that must be inspected to arrive at the result, the more efficient the corresponding matching automaton can be. To reason about such concerns more easily, we introduce extended values whose nodes are labeled by $+$ or $-$ to indicate whether they are traversed or skipped.

An *annotated value* can be of the form ϵ_* or $a_*(v_1, v_2)$ where v_1 and v_2 are annotated subvalues, l is an element label, and $* \in \{+, -\}$. We say that a value is annotated *consistently*, if for every node of the form $a_-(v_1, v_2)$, both v_1 and v_2 have all their nodes annotated by $-$. A value is *fully traversed* if all its nodes are annotated by $+$. The *erasure* of an annotated value v written $|v|$ is an ordinary value of the same structure with all the annotations eliminated.

Let v_1 and v_2 be consistently annotated values. We say that v_1 is *less traversed* than v_2, written $v_1 \leq v_2$, if $|v_1| = |v_2|$ and, for any node in v_1 labeled by $+$, the corresponding node in v_2 is also labeled by $+$. We say that v_1 is *strictly less traversed* than v_2, written $v_1 < v_2$, if $v_1 \leq v_2$ and $v_1 \neq v_2$.

An *annotated value environment* is a mapping from variable names to annotated values. An environment is *fully traversed* if its range contains only fully traversed values. The erasure operation

on annotated value environments $|E|$ producing an ordinary environment is defined pointwise. The $<$ and $\leq$ relations on annotated values are extended point-wise to annotated environments.

6.1.2 Regular Patterns

Regular patterns are described by the following grammar:

$$p \quad ::= \quad () \mid a[p] \mid p_1, p_2 \mid p_1|p_2 \mid p* \mid \mathit{Any} \mid X$$

These denote the empty sequence pattern, a labeled element pattern, sequential composition, union, repetition, wild-card, and a pattern variable. Pattern variables are introduced by top-level, mutually recursive declarations of the form $\mathit{def}\, X = p$. Top-level declarations induce a function def that maps variables to the associated patterns (e.g. the above declaration implies $\mathit{def}(X) = p$.)

The pattern membership relation $v \in p$ is described by the following rules: first, $() \in ()$; second, $a[v] \in a[p]$ if $v \in p$; third, $v \in p*$ if v can be decomposed into a concatenation of $v_1 \ldots v_n$ with each $v_i \in p$; fourth, $v \in p_1, p_2$ if v is the concatenation of two sequences v_1 and v_2 such that $v_1 \in p_1$ and $v_2 \in p_2$; fifth, $v \in \mathit{Any}$ for any v, and finally, $v \in p_1|p_2$ if $v \in p_1$ or $v \in p_2$.

6.1.3 Tree Automata

A *non-deterministic top-down tree automaton* is a tuple $A = (S, T)$, where S is a set of states and T is a set of transitions consisting of *empty* transitions of the form $s \rightarrow ()$ and *label* transitions of the form $s \rightarrow a[s_1], s_2$, where $s, s_1, s_2 \in S$ and a is a label. The acceptance relation on values and states, denoted $v \in s$, is defined by the following rules:

$$\frac{s \rightarrow () \in T}{() \in s}\ (\text{TA-Emp}) \qquad \frac{s \rightarrow a[s_1], s_2 \in T \qquad v_1 \in s_1 \qquad v_2 \in s_2}{a[v_1], v_2 \in s}\ (\text{TA-Lab})$$

From now on, we will assume that all the regular patterns in the source program have been converted to states of one global tree automaton, and we will use these states in place of the corresponding regular types and patterns.

6.1.4 Matching Automata

We now introduce an updated view of matching automata. Two aspects differentiate the following definition from the one appearing in Section 4.4. First, the formalism is adjusted for annotated values. Second, to simplify the presentation of this chapter's material, we associate variables with

states rather than transitions. While in Chapter 4 matching automaton transitions had destination pairs containing variables and destination states, here, transitions only have destination states, but each state is paired with its own variable. (A useful consequence of this change is that the judgment describing the semantics of states will have exactly the same form as the judgment describing the semantics of configurations.)

6.1.1 Definition: A *matching automaton* is a tuple (Q, q_s, V, R), where Q is a set of states, q_s is a start state, V is a mapping from states to variables, and R is a set of transitions. There are two kinds of transitions: *simple* and *subroutine*. They have the following structure:

$$\begin{aligned} &q(x) : p \xrightarrow{I} \{q_1 \dots q_m\} && \text{(simple)} \\ &q(x) : A \xrightarrow{\sigma} \{q_1 \dots q_m\} && \text{(subroutine)} \end{aligned}$$

Both types of transitions have a *source state* q—associated with some variable x via the mapping V—and a set of *destination states* $\{q_1 \dots q_m\}$. A simple transition contains a target language pattern p—which can be of the form $()$ or $a[x], z$—and a set of integer results I. A subroutine transition contains a subroutine automaton name A and a binary relation σ over results.

Let $\mathcal{M}$ be a mapping of names to matching automata, and let $A = (Q, q_s, V, R)$ be a matching automaton. The acceptance relation on annotated environments, states, and results, denoted $E \in q \Rightarrow k$, is defined by the following rules:

$$\frac{q(x) : () \xrightarrow{I} \{q_1 \dots q_m\} \in R \qquad E(x) = \epsilon_+ \qquad k \in I \qquad E' = E \backslash x \qquad \forall i \in \{1 \dots m\}.\ E' \in q_i \Rightarrow k}{E \in q \Rightarrow k} \qquad \text{(MA-Emp)}$$

$$\frac{q(x) : a[y], z \xrightarrow{I} \{q_1 \dots q_m\} \in R \qquad E(x) = a_+(v_1, v_2) \qquad k \in I \qquad E' = (E \backslash x)[v_1/y, v_2/z] \qquad \forall i \in \{1 \dots m\}.\ E' \in q_i \Rightarrow k}{E \in q \Rightarrow k} \qquad \text{(MA-Lab)}$$

$$\frac{q(x) : B \xrightarrow{\sigma} \{q_1 \dots q_m\} \in R \qquad E \in \mathcal{M}(B) \Rightarrow j \qquad (j, k) \in \sigma \qquad E \text{ is fully traversed} \qquad E' = E \backslash x \qquad \forall i \in \{1 \dots m\}.\ E' \in q_i \Rightarrow k}{E \in q \Rightarrow k} \qquad \text{(MA-Sub)}$$

An annotated environment E is accepted by matching automaton A with result k, written $E \in A \Rightarrow k$, if it is accepted by the automaton's start state: $E \in q_s \Rightarrow k$.

A concrete implementation of pattern matching deals with ordinary rather than annotated values. Initially, we intended annotations to be returned as a *result* of a matching automaton run indicating which parts of the unannotated input value were inspected. We discovered, however, that we can develop a simpler formalism with annotations as part of input values. So, when we say that one automaton accepts $a_+(\epsilon_+, \epsilon_+)$ and another $a_+(\epsilon_-, \epsilon_+)$, we mean that they both accept the same sequence `a[]` but the former performs more inspections than the latter.

The rule MA-Emp says that a state q accepts an environment E yielding a result `k` if: 1) q is the source state for a transition of the form $q(x) : () \xrightarrow{I} \{q_1 \ldots q_m\}$; 2) the variable x associated with q contains the empty sequence ϵ_+, and 3) each destination state accepts the environment obtained from E by removing x's binding yielding the same result k.

The rule MA-Lab describes how a state can accept an environment if the associated variable contains a non-empty sequence value. It is similar to MA-Emp except that the environments used with the destination states are extended with bindings of fresh variables y and z to the left and right subtrees of the input.

For an environment to be accepted by a state with the help of a subroutine transition, it has to be fully traversed. The intent of this requirement is that once a subroutine matching automaton is invoked, we do not attempt to track which nodes are touched by the subroutine automaton, and it is assumed that any part of any input value that has not been processed yet by the current matching automaton may be inspected. According to MA-Sub, an environment is accepted by a state q yielding a result `k` if: 1) there is a subroutine transition of the form $q(x) : B \xrightarrow{\sigma} \{q_1 \ldots q_m\}$, the subroutine matching automaton accepts E yielding a result j such that $(j \mapsto k)$ is in the transition's result mapping relation σ, and the destination pairs are checked as in MA-Emp.

The result mapping relations in subroutine transitions serve two purposes. First, they allow us to reduce the number of subroutine automata since we can avoid building isomorphic automata that only differ in their indices. Second, and more importantly, they are essential for creating matching automata that represent non-backtracking target programs.

Figure 6.2 shows a source program, an equivalent matching automaton, and a corresponding target program. Matching automaton states are depicted with their associated variables inside and their names above the circle. Observe the correspondence between the matching automaton and the target program: states correspond to `case` expressions and transitions to `case` clauses.

6.1.5 Configurations

During execution of a matching automaton, the current state is faced with an environment mapping variables to values. The following data structure will help us describe the types of values stored in

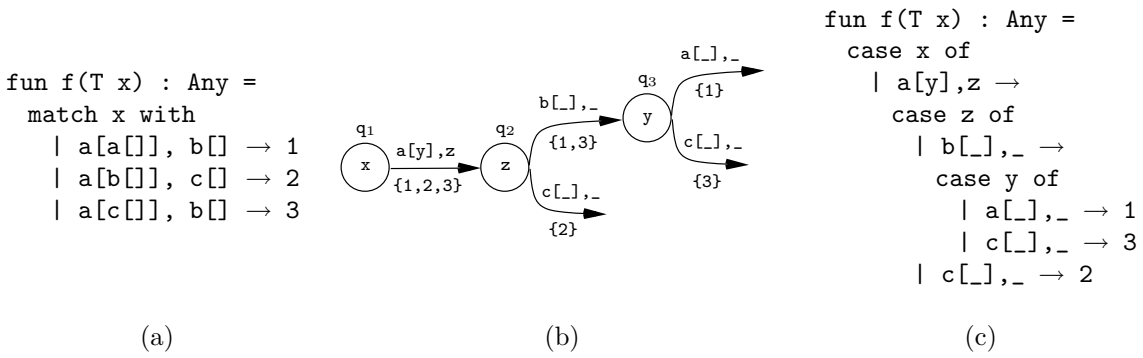


Figure 6.2: Matching automaton illustration: a source program (a); an equivalent matching automaton (b); an equivalent target program (c); input type `T = a[a[]],b[] | a[b[]],c[] | a[c[]],b[]`

the environment and will be used in the matching automaton generation algorithm.

6.1.2 Definition: A *configuration* over a tree automaton comprises a tuple of distinct variables $(x_1 \ldots x_n)$ and a set of tuples $\{(s_{11} \ldots s_{1n}, j_1) \ldots (s_{m1} \ldots s_{mn}, j_m)\}$, each associating a collection of the tree automaton's states to a result. We depict a configuration by a matrix as follows:

$$C = \begin{array}{|lll|l|} \hline \mathtt{x}_1 & \ldots & \mathtt{x}_n & \\ \hline \mathtt{s}_{11} & \ldots & \mathtt{s}_{1n} & \mathtt{j}_1 \\ & \ldots & & \\ \mathtt{s}_{m1} & \ldots & \mathtt{s}_{mn} & \mathtt{j}_m \\ \hline \end{array}$$

Two auxiliary functions are defined on configurations: $\mathit{vars}(C) = \{x_1 \ldots x_n\}$ and $\mathit{results}(C) = \{j_1 \ldots j_m\}$. We say that an ordinary environment E is accepted by C yielding a result j_r, written $E \in C \Rightarrow j_r$, if $E(x_i) \in s_{ri}$ for all $i \in \{1 \ldots n\}$. An environment E is accepted by a configuration C, written $E \in C$, if there exists a result j such that $E \in C \Rightarrow j$.

A configuration describes the work that remains to be done to determine the outcome of pattern matching in a `match` expression. The variables contain subtrees that have yet to be examined. The integer results correspond to the different clauses of the `match` expression.

The notions of boolean operations and subtyping for regular types and tree automaton states can be extend to configurations. Let C and C_0 be a configuration and an input configuration, respectively. For example, we say that $C' = C \cap C_0$ is a configuration such that, for any environment E, if $E \in C_0$ and $E \in C \Rightarrow j$, then $E \in C' \Rightarrow j$.

Unlike the acceptance relation for matching automaton states, the acceptance relation for configurations is defined with respect to ordinary rather than annotated environments. This is because

```
fun f(T1 x) : Any =
  match x with
    | Any, a[] → 1
    | Any → 2
```

(a)

```
fun f(T2 x) : Any =
  match x with
    | a[b[]],a[Any] → 1
    | a[Any] → 2
```

(b)

x	
a[],a[]	1
a[],(a[]|b[])	2

(c)

x	y	
()	a[]	1
()	a[]|b[]	2

(d)

y	
a[]	1
a[]|b[]	2

(e)

x	
a[b[]],a[Any]	1
a[Any]	2

(f)

x	y	
b[]	a[Any]	1
Any	()	2

(g)

```
fun f(T1 x) : Any =
  case x of
    | a[x],y →
      case y of
        | a[x],y → 1
        | b[x],y → 2
```

(h)

```
fun f(T2 x) : Any =
  case x of
    | a[x],y →
      case y of
        | a[x],y → 1
        | () → 2
```

(i)

Figure 6.3: Two source programs (a, b); configurations used in code generation (c - g); and the obtained target programs (h, i). Input types are: T_1 = `a[],(a[]|b[])` and T_2 = `(a[b[]],a[Any]) | a[Any]`

configuration acceptance is expressed in terms of tree automaton state acceptance, and tree automaton states, unlike matching automaton states, traverse input values fully.

6.2 XTATIC Pattern Compiler

This section presents an efficient type-based compilation algorithm that is used in the current XTATIC compiler. It outlines a general compilation strategy of starting with an initial configuration and gradually expanding it into a completed matching automaton. We describe an effective heuristic for selecting a column on which expansion is based.

The second part of this section summarizes the results of performance experiments for several XTATIC programs. We ran them in the current implementation of XTATIC, recording the size and running time of the generated target programs.

6.2.1 Heuristic Algorithm

The goal of the compiler is to construct a matching automaton that implements pattern matching in a given `match` expression.

It starts with an *initial configuration* containing the patterns of the `match` expression intersected with the input type. From this point on, the input type is not taken into account. Figures 6.3(c) and 6.3(f) are examples of initial configurations for the source programs shown in (a) and (b).

A configuration describes the work that must still be done before the outcome of pattern matching can be determined. The variables contain subtrees that have yet to be examined. Pattern matching will succeed with the result given at the end of some row if all of the row's patterns match the subtrees stored in the corresponding variables.

When faced with a configuration, the compiler has a choice of which subtree (i.e., which column) to examine next. We use the following heuristic. Let `P` be a set of patterns. A partition of `P` into a number of subsets is *disjoint* if for any two patterns $\mathtt{p}_1, \mathtt{p}_2 \in \mathtt{P}$, if $\mathtt{p}_1 \cap \mathtt{p}_2 \neq \emptyset$, then both $\mathtt{p}_1$ and $\mathtt{p}_2$ are in the same subset. We say that the *branching factor* of a column is the number of subsets in the largest disjoint partition of the column's patterns. The maximal branching factor heuristic then tells us to select the column with the largest branching factor. The motivation behind this heuristic is to arrive at single-result configurations as fast as possible. Such configurations need no further pattern matching, since the result has already been determined.

Figure 6.3 shows two examples. In the first one, the initial configuration (c) contains patterns matching non-empty `a`-labeled trees. From this configuration, the compiler generates the pattern `a[x],y` of the outer `case` and proceeds to the next configuration (d). The first column of this configuration is then eliminated by the optimization technique described in Section 4.5.4 The resulting configuration (e) is used to generated the clauses of the inner `case`. In the second example, we get to employ the maximal branching heuristic for configuration (g). The branching factor of its first column is one, since its patterns are overlapping; the branching factor of the second column, on the other hand, is two. Hence, the inner `case` in Figure 6.3(i) examines `y` rather than `x`.

Further optimizations are possible. The pattern variables that are not referenced can be replaced by `_`. The last `case` clause can be replaced by a default `else` clause if its pattern does not bind any variables and if there is not already a default clause. A label in a pattern can be replaced by the wild card `~` if the `case` has neither a default clause nor any other label pattern. Applying these simplifications to the first program (h), for example, results in the optimal program shown in Figure 6.3(a).

Since our heuristic selects the second column of configuration (e), the generated target program is able to skip the subtree stored in `x` completely. This demonstrates an advantage of our approach

over the strictly left-to-right type propagation approaches used in some versions of XDuce and CDuce.

For some examples the heuristic approach falls short of optimality (both informally and in the precise sense defined in Section 6.3). Consider this configuration:

x	y	z	
a[]	a[]	a[]|b[]	1
a[]|b[]	b[]	a[]|b[]	2
a[]|b[]	a[]|b[]	b[]	3

As we will see in Section 6.4, it is more beneficial to examine the `y` and `z` columns before examining the `x` column. The heuristic method defined above, however, considers all three columns equal, since they all have a branching factor of 1. So, the heuristic algorithm can potentially generate a suboptimal matching automaton.

Our experience shows that the maximal branching factor heuristic results in high-quality target code for most source programs, since intersection of the input type with `match` patterns and the simple configuration optimizations described above seem to account for most type-based optimization opportunities.

6.2.2 Experiments

To give a sense of the impact of type-based optimization, we compare the performance of three Xtatic programs with and without type-based optimization. The first, `addrbook`, is a small 60 line application that filters an address book and converts the result into a phone book format. The default input for this program is a 31Kb file containing 1,000 address records. We iterate the processing part of the program 10 times to obtain stable results. The second program, `cwn`, converts raw XML newsgroup data into a formatted HTML presentation. The source program contains 400 lines of code; the default input is a 7.7Kb file with seven newsgroup articles. This program is also iterated 10 times. The third program, `bibtex`, is a 700 line program that reads a bibtex file formatted as XML, filters and sorts its contents, and outputs the result as an HTML page. The default input for `bibtex` is a 560Kb file with approximately 1,500 bibtex entries. This processing step of this program is run only once.

Note that Xtatic's compiler is quite efficient even when its type-based optimization is turned off. It employs a variety of other optimizations that go a long way toward producing efficient code. In fact, a previous version of Xtatic's compiler that did not have type-based optimization compared favorably with several other XML processing languages [21].

addrbook		cwn		bibtex	
no tb	tb	no tb	tb	no tb	tb
710	569	19200	17600	35300	26800

(a)

	addrbook		cwn		bibtex	
	no tb	tb	no tb	tb	no tb	tb
n = 1	13	12	300	290	3100	900
n = 2	17	16	420	390	4000	1900
n = 3	23	21	660	510	4900	2900
n = 4	31	28	640	590	11500	4000
n = 5	39	35	770	690	27400	20300

(b)

Figure 6.4: Size (a) and speed in ms (b) of three source programs with and without type-based optimization; n is a size factor w.r.t. the default input size

Figure 6.4 displays our measurements. Table (a) lists sizes of the output programs in terms of the number of nodes in their ASTs. Table (b) contains running times of the programs for the default input as well as duplicated inputs whose sizes are factors 2, 3, 4, and 5 of the default input's size. Both size and running time measurements are listed for the case when the program was compiled without ("no tb") and with ("tb") type-based optimization as described above.

Overall, these examples illustrate a steady benefit of type-based optimization. It gives us a 10% to 25% improvement in the size of the target program and a similar—or in case of `bibtex` even more dramatic—improvement in the running time. Let us take a closer look at these examples individually.

The `addrbook` program demonstrates a modest improvement in size and speed when compiled with type-based optimization. Figure 6.5(a) shows a slightly simplified version of `addrbook`'s core fragment. The regular types on the left describe the program's input. Each address book entry contains a `person` element with a `Name`, an optional `Tel`, and a list of `Email`s. Function `mkTelbook` inspects each `Person` entry and checks whether it has a `Tel` subelement. Just using simple configuration optimizations, the XTATIC compiler generates a fairly efficient output code for this program(b). For instance, the produced program does not check the outer `person` label for each input record since from the `match` patterns alone it can be seen that no other label can be expected in this position. The only benefit of type-based analysis here is the ability to infer that the first child of a `person` element must be a `Name`, and, therefore, that there is no need to check for the presence of the `name` label (c). This is precisely what accounts for the better measurements when `addrbook` is compiled with type-based optimization.

In the case of `cwn`, type-based optimization matters less. The only difference of any significance occurs in a function that performs a character-for-character traversal of its input in order to locate a particular substring. Either a match is found in the beginning of the input or the first character

```
def Name = name[pcdata]
def Tel = tel[pcdata]
def Email = email[pcdata]
def Person = person[Name,Tel?,Email*]
```

```
fun mkTelbook (Person* ps) : Any =
  match ps with
    | person[name[Any],tel[Any],Any], Any
      → 1
    | person[Any], Any → 2
    | () → 3
```

(a)

```
fun mkTelbook (Person* ps) : Any =
  case ps of
    | () → 3
    | ~[x],y →
      case x of
        | name[z],w →
          case w of
            | tel[u], _ → 1
            else 2
        else 2
```

(b)

```
fun mkTelbook (Person* ps) : Any =
  case ps of
    | () → 3
    | ~[x],y →
      case x of
        | ~[z],w →
          case w of
            | tel[u],_ → 1
            else 2
```

(c)

Figure 6.5: A fragment from `addrbook` example: source program (a); corresponding target language code generated without (b) and with (c) type-based optimization

is skipped and the same process is repeated from the next character. Since the input type to this function is `pcdata`—a sequence of character-labeled elements without attributes—there is no need to check for the absence of attributes in every element.

The `bibtex` program gives us the most revealing example of the benefits of type-based optimization. Most of the improvement arises from function `do_xml` that examines the current entry in a bibtex document and determines its type. Figure 6.6(a) shows the regular types associated with this program. There are fourteen kinds of bibtex entries described by regular type `entry`. The structure of each kind of entry is described by the corresponding regular type (`article` e.g.) Figure 6.6(b) contains a skeleton of `do_xml`—a dispatch function that branches to different subtasks depending on the kind of the current entry.

Because of the default fall-through case in the `match` expression, a naive compilation strategy that does not take the input type into account results in a *huge* target program that meticulously checks whether the structure of the current element *completely* matches one of the bibtex entry types. In the case of an `article` element, for example, the target program checks whether its contents starts with an `author` element containing `pcdata` and followed by a `bib_title`, `journal`, and `year` elements, and then potentially followed by a `volume` element etc. Using the input type

```
def article =
  article[author, bib_title, journal,
        year, volume?, number?, pages?,
        month?, note?, fields]

def author = author[pcdata]
def bib_title = bib_title[pcdata]
```

```
def entry =
  article | book | booklet | conference |
  inbook | incollection | inproceedings |
  manual | masterthesis | misc |
  phdthesis | proceedings | techreport |
  unpublished
```

(a)

```
fun do_xml(entry x) : Any =
  match x with
    | article → 1
    | inproceedings → 2
    | unpublished → 3
    | incollection → 4
    | phdthesis → 5
    | techreport → 6
    | book | inbook | manual
    | masterthesis | proceedings → 7
    | Any → 8
```

(b)

```
fun do_xml(entry x) : Any =
  case x of
    | article[_],_ → 1
    | inproceedings[_],_ → 2
    | unpublished[_],_ → 3
    | incollection[_],_ → 4
    | phdthesis[_],_ → 5
    | techreport[_],_ → 6
    | book[_],_ → 7
    | inbook[_],_ → 7
    | manual[_],_ → 7
    | masterthesis[_],_ → 7
    | proceedings[_],_ → 7
    else 8
```

(c)

Figure 6.6: A fragment from `bibtex` example: source program types (a); source language processing function (b); optimal corresponding target program compiled with type-based optimization (c)

information, however, the compiler realizes that, since only valid `entry` elements can be given as arguments to `do_xml`, and since each entry type has a distinct outer label, checking that outer label is sufficient to determine the type of the entry. Figure 6.6(c) shows a compact and efficient target program that is the result of compiling `do_xml` with type-based optimization.

6.3 Optimality Criterion

We have stated in Section 6.2.1 that XTATIC's heuristic algorithm is not always "optimal". What exactly did we mean by that? This section addresses this by presenting a formal view of optimality. We start by observing that "full optimality"—i.e., running at least as fast as any other matching automaton on every input—is not possible. We then define an optimality criterion according to which a program is optimal if there does not exist an equivalent strictly more efficient program. We conclude this section by discussing several limitations of the proposed criterion.

```
fun f(T x) : Any =
  match x with
    | a[b[]], c[] → 1
    | a[b[]], d[] → 2
    | a[d[]], c[] → 3
```

(a)

```
fun f(T x) : Any =
  case x of
    | ~[y], z →
      case y of
        | b[_], _ →
          case z of
            | c[_], _ → 1
            | else 2
        else 3
```

(b)

```
fun f(T x) : Any =
  case x of
    | ~[y], z →
      case z of
        | c[_], _ →
          case y of
            | b[_], _ → 1
            | else 3
        else 2
```

(c)

Figure 6.7: Perfect optimality is unreachable: a source program (a) with input type `T = a[b[]],c[] | a[b[]],d[] | a[d[]],c[]`; a target program that is fast for the third case (b); a target program that is fast for the second case (c)

We now turn to a formal discussion of what it means for one target program (or matching automaton) to be better than another one.

Ideally, we would like to perform the minimal number of tests for any input value. Figure 6.7 demonstrates that this is not always possible. The source program shown in Figure 6.7(a) contains a `match` expression with three clauses. The clause patterns match sequences starting from `a`-labeled elements. To determine the outcome, the pattern matcher can first investigate the contents of the first element—as in Figure 6.7(b)—or else look at the rest of the sequence—as in Figure 6.7(c). In the former case, two tests are required to determine results 1 and 2, but only one test to determine result 3. In the latter case, it takes two tests to determine outcomes 1 and 3 and one to determine result 2. It is not possible for any target language pattern matcher to be as fast as the first program for the input matching the third clause and as fast as the second program for the input matching the second clause.

Consequently, we must settle for near-optimality and, for any pattern matching task, try to build a matcher that is not clearly bested by any other but may not be necessarily *the* best one. First, we recall the formalities of matching automata.

Comparing matching automata is only meaningful with respect to the type of the input values: matching automaton A may be more efficient than matching automaton B for some set of values but not more efficient—or even not equivalent—for a larger set of values. Intuitively, one matching automaton A is more efficient than another matching automaton B if it needs to traverse a smaller or equal part of an input value to arrive at the same result. This must be the case for any input—A cannot be more efficient than B if it requires more traversal even for one value.

6.3.1 Definition: Let $M_1 = (Q_1, q_1, V_1, R_1)$ and $M_2 = (Q_2, q_2, V_2, R_2)$ be matching automata,

```
fun f(T x) : Any =
  match x with
    | a[a[]], b[]|c[] → 1
    | a[b[]], c[] → 2
    | a[c[]], b[] → 3
```

(a)

```
fun f(T x) : Any =
  case x of
    | ~[y],z →
      case z of
        | b[_],_ →
          case y of
            | a[_],_ → 1
            | c[_],_ → 3
        | c[_],_ →
          case y of
            | a[_],_ → 1
            | b[_],_ → 2
```

(b)

```
fun f(T x) : Any =
  case x of
    | ~[y],_ →
      case y of
        | a[_],_ → 1
        | b[_],_ → 2
        | c[_],_ → 3
```

(c)

Figure 6.8: An illustration of optimality criterion: a source program (a) with input type `T = (a[a[]], b[]|c[]) | a[b[]],c[] | a[c[]],b[]`; a suboptimal target program (b); an optimal target program (c)

and let C be an input configuration. We say that $q \in Q_1$ is *more efficient* than $q' \in Q_2$ for C, written $C \vdash q \leq q'$, if, for any E such that $|E| \in C$ and $E \in q' \Rightarrow j$, there exists $E' \leq E$ such that $E' \in q \Rightarrow j$. We say that q is *strictly more efficient* than q' for C, written $C \vdash q < q'$ if $C \vdash q \leq q'$ and, for some E such that $|E| \in C$ and $E \in q \Rightarrow j$, it is not the case that $E \in q' \Rightarrow j$. We say that M_1 is more efficient than M_2 for C, written $C \vdash M_1 < M_2$, if $C \vdash q_1 < q_2$.

Consider the example in Figure 6.8. It shows a source program and two possible translations into the target language. The target program in Figure 6.8(b) is suboptimal. It tests the right subtree of the input value, and, regardless of the result, inspects the left subtree as well. The program in Figure 6.8(c) is better—it never inspects the right subtree. This program is more efficient than the suboptimal one since, for any annotated value accepted by the latter, the former accepts a less traversed value producing the same result. It is *strictly* more efficient since, for example, $a_+(b_-(\epsilon_-, \epsilon_-), c_+(\epsilon_-, \epsilon_-))$ is accepted by it but not by the suboptimal program.

Note that the proposed measure of optimality does not precisely reflect the amount of work performed by a matching automaton. Consider Figure 6.9, which shows a source program and two ways of compiling it to the target language. Target program (b) starts by inspecting the right subtree of the input; if it finds a `c` leaf, it can select the first `match` clause; otherwise, it checks whether the root of left subtree is labeled by `b`, and selects the first or the second clause depending on that. Target program (c) checks only the left subtree: if its root is `b`-labeled, it selects the first clause; otherwise the second. The latter program performs fewer than or the same number of node tests as the former for any input. It is *not*, however, any more efficient according to our definition

```
fun f(T x) : Any =
  match x with
    | a[b[Any],Any],
      a[Any] → 1
    | a[Any],a[b[d[]]]
      → 2
```

(a)

```
fun f(T x) : Any =
  case x of
    | ~[y],z →
      case z of
        | ~[w],_ →
          case w of
            | c[_],_ → 1
            else
              case y of
                | b[_],_ → 1
                else 2
```

(b)

```
fun f(T x) : Any =
  case x of
    | ~[y],_ →
      case y of
        | b[_],_ → 1
        else 2
```

(c)

Figure 6.9: Optimality criterion limitation: a source program (a); with input type `T = (a[b[]], a[b[c[]]]) | (a[Any], a[b[d[]]])`; an optimal target program (b); a *better* optimal target program (c)

since, for the values matching `a[Any], a[b[c[]]]`, program (b) completely skips the left subtree, while program (c) inspects its root node.

A more precise measure of optimality would involve counting the number of node tests performed by a matching automaton regardless of where in the input value they occur. According to such a measure, program (c) of Figure 6.9 would be more efficient than program (b). It is difficult, however, to reason about this kind of a measure. For instance, performing various boolean operations such as intersection and difference on regular patterns does not shed any light on how many node tests may be necessary to match a value against them. We leave investigation of this kind of optimality measures for future work.

6.4 Optimal Compilation for Finite Patterns

Is it possible to generate an optimal matching automaton for a given `match` expression? This section positively answers this questions for a particular class of matching problems—those involving finite (non-recursive) patterns. Building on the intuitions given in Section 6.2.1—where we informally presented XTATIC's not-always-optimal pattern compiler—we give a formal account of key aspects of the optimal algorithm.

In Section 6.4.1, we start by introducing an extended version of matching automata—*incomplete matching automata*—in which pairs of configurations can appear as pseudo-states. We then formalize the process of configuration *expansion*, briefly sketched in Section 6.2.1, and show how using expansion, we can convert a pseudo-state into an ordinary matching automaton state—thus going

from the current incomplete matching automaton to a slightly less incomplete matching automaton. Iterating the expansion step will eventually lead to a conventional matching automaton which implements the original pattern matching task.

Section 6.4.2 pays special attention to a particular way of selecting the expansion column for the current pair of configurations during each iteration of the algorithm. The proposed method is precisely what ensures optimality of the generated matching automata. Section 6.4.3 concludes by establishing the optimality property.

6.4.1 Incomplete Matching Automata

First, let us introduce an abridged form of configurations without the result column. We will refer to such configurations as *input configurations*. They can be used as a precise specification of the input type for multiple values. Note that an input configuration gives us more information than a simple type assignment. Consider the following configuration:

y	z
T_1	T_2
T_3	T_4

It specifies inputs in which y and z can have respectively either types T_1 and T_2 or types T_3 and T_4. Compare that with the type assignment $\{y \mapsto T_1|T_3, z \mapsto T_2|T_4\}$, which, in addition to the above two scenarios, allows y to be in T_1 while z is in T_4 and y in T_3 while z is in T_2.

Now, we are ready to extend matching automata with pseudo-states. Each pseudo-state is a pair of configurations—one ordinary describing the remaining tests necessary to determine the outcome of pattern matching; the other an input configuration describing the type of the input environment. The initial state can be either a conventional state or a pseudo-state configuration pair.

6.4.1 Definition: An *incomplete matching automaton* is a tuple (Q, K, i, V, R), where Q, V, and R are a set of states, a mapping from states to variables, and a set of transitions respectively just as in matching automata (Definition 6.1.1). Additionally, K is a set of pairs of configurations constituting pseudo-states, and i is the initial state which is either an ordinary state $i \in Q$ or a pseudo-state $i \in K$.

The form of transitions is similar to that of matching automaton transitions except that simple transitions can have pseudo-states in addition to states as their destinations. Pseudo-states cannot appear as sources of transitions.

The semantics of simple transitions is defined as in the matching automaton case except that whenever a configuration pair C, D appears in the destination set of a transition, the judgment for

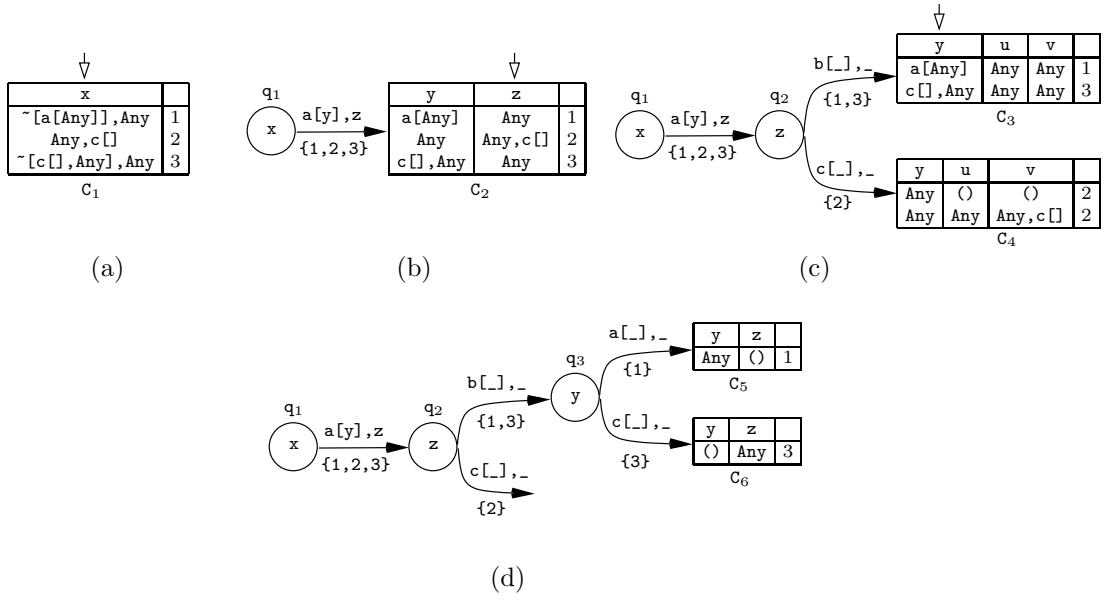


Figure 6.10: An illustration of a gradual expansion of an input configuration into a matching automaton

configurations $|E| \in C \Rightarrow k$ is used instead of the judgment for states $E \in q \Rightarrow k$:

$$\frac{q(x) : () \xrightarrow{I} \{q_1 \dots q_m, (C_1, D_1) \dots (C_j, D_j)\} \in R \qquad E(x) = \epsilon_+ \qquad k \in I \qquad E' = E \backslash x \qquad \forall i \in \{1 \dots m\}.\ E' \in q_i \Rightarrow k \qquad \forall i \in \{1 \dots j\}.\ |E'| \in C_i \Rightarrow k}{E \in q \Rightarrow k} \quad \text{(IMA-Emp)}$$

$$\frac{q(x) : a[y], z \xrightarrow{I} \{q_1 \dots q_m, (C_1, D_1) \dots (C_j, D_j)\} \in R \qquad E(x) = a_+(v_1, v_2) \qquad k \in I \qquad E' = (E \backslash x)[v_1/y, v_2/z] \qquad \forall i \in \{1 \dots m\}.\ E' \in q_i \Rightarrow k \qquad \forall i \in \{1 \dots j\}.\ |E'| \in C_i \Rightarrow k}{E \in q \Rightarrow k} \quad \text{(IMA-Lab)}$$

Figure 6.10 displays several incomplete matching automata in which we omit input configurations from pseudo-states for space reasons. The first example (a) shows an incomplete matching automaton with a single pseudo-state, which is also the initial state. The second (b) is an incomplete matching automaton with an ordinary initial state and a pseudo-state. The automaton contains one transition from the ordinary state to the pseudo-state. We will return to this example below to illustrate the matching automaton generation algorithm.

Now we have all the necessary tools to specify the skeleton of the generation algorithm. The goal is to construct a matching automaton that implements pattern matching in a particular `match` expression. We start with an incomplete matching automaton consisting of one pseudo-state: a

pair of an initial configuration C and an initial input configuration D. The former contains a single column composed of the tree automaton states corresponding to the patterns of the `match` expression. The latter contains a single column whose only row has the tree automaton state corresponding to the input type.

Consider the following `match` expression with the input type `T = a[a[]],b[] | a[b[]],c[] | a[c[]],b[]` and the corresponding pair of initial configurations:

```
fun f(T x) =
  match x with
    | ~[a[Any]],Any → 1
    | Any,c[] → 2
    | ~[c[],Any],Any → 3
```

x	
`~[a[Any]],Any`	1
`Any,c[]`	2
`~[c[],Any],Any`	3

x
`a[a[]],b[] | a[b[]],c[] | a[c[]],b[]`

The left configuration describes the pattern matching work that must still be done for the outcome to be determined. The input configuration specifies the type of values stored in the still-to-be-explored variables.

This pair of configurations can be turned into a matching automaton state by the process of *expansion*. For a pseudo-state to become a state, we must determine the variable associated with the new state and all the transitions originating in it. The former is selected among the variables of the configuration and specifies which subtree will be examined next. At this point, we will elide the details of how the selection is made, but we will return to this issue in Section 6.4.2. The selected variable is indicated by a vertical arrow in the picture. For now, let us concentrate on generating the transitions.

Having selected the column in the current pair of configurations, we can construct all possible target language patterns that match the input type of the selected variable, and, for each pattern, derive a pair of residual configurations that describe the remaining pattern matching work and the types of the still uninspected variables. The example above gives rise to one such target language pattern `a[y],z`, since all the sequences specified by the input type must be non-empty and must start with an `a`-labeled element. The corresponding residual configuration and input configuration

are as follows:

y	z	
a[Any]	Any	1
Any	Any,c[]	2
c[],Any	Any	3

y	z
a[]	b[]
b[]	c[]
c[]	b[]

This expansion step is illustrated in Figure 6.10(a,b). The obtained incomplete matching automaton (b) is equivalent to the initial incomplete matching automaton (a) and, therefore, correctly implements the original `match` expression.

Proceeding in the same way, we can pick out an unexpanded pseudo-state, expand it, and replace it with the obtained state, transitions, and residual configuration pairs. Eventually, this process will terminate when no more pseudo-states are left. The result will be a complete matching automaton.

Note that threading input configurations throughout the generation process allows us to track what types of values can flow into the currently generated matching automaton state; based on that, we will be able to construct optimal matching automata.

Before we present an outline of the algorithm, we give a formal definition of configuration expansion. The same can be applied for input configurations by dropping result columns.

6.4.2 Definition: Let $A = (S, T)$ be a tree automaton, and let C be a configuration over A consisting of variables $(x_1 \dots x_n)$ and tuples of tree automaton states $\{(s_{11} \dots s_{1n}, j_1) \dots (s_{m1} \dots s_{mn}, j_m)\}$. Let c be a column in C identified by x_c, and let p be a target language pattern. An *expansion* of C *based on* c *by* p, denoted *expand*(C, c, p), is a configuration C' such that: if $p =$ `()`,

$$C' = \begin{array}{|cccccc|c|} \hline x_1 & \dots & x_{c-1} & x_{c+1} & \dots & x_n & \\ \hline s_{k_1 1} & \dots & s_{k_1(c-1)} & s_{k_1(c+1)} & \dots & s_{k_1 n} & j_{k_1} \\ & & & \dots & & & \\ s_{k_i 1} & \dots & s_{k_i(c-1)} & s_{k_i(c+1)} & \dots & s_{k_i n} & j_{k_i} \\ \hline \end{array}$$

where $\{k_1 \dots k_i\} = \{k \mid s_{kc} \rightarrow \texttt{()} \in T\}$ or, if $p =$ `l[z],y` for some label l and variables $z, y \notin$ *vars*$(C) \backslash \{x_c\}$,

$$C' = \begin{array}{|llllllll|l|}
\hline
z & y & x_1 & \ldots & x_{c-1} & x_{c+1} & \ldots & x_n & \\
\hline
t'_{11} & t''_{11} & s_{11} & \ldots & s_{1(c-1)} & s_{1(c+1)} & \ldots & s_{1n} & j_1 \\
 & & & & \ldots & & & & \\
t'_{1k_1} & t''_{1k_1} & s_{11} & \ldots & s_{1(c-1)} & s_{1(c+1)} & \ldots & s_{1n} & j_1 \\
 & & & & \vdots & & & & \\
t'_{m1} & t''_{m1} & s_{m1} & \ldots & s_{m(c-1)} & s_{m(c+1)} & \ldots & s_{mn} & j_m \\
 & & & & \ldots & & & & \\
t'_{mk_m} & t''_{mk_m} & s_{m1} & \ldots & s_{m(c-1)} & s_{m(c+1)} & \ldots & s_{mn} & j_m \\
\hline
\end{array}$$

where $\{(t'_{i1}, t''_{i1}) \ldots (t'_{ik_i}, t''_{ik_i})\} = \{(t', t'') \mid s_{ic} \rightarrow \texttt{l[t'],t''} \in T\}$ for $i \in \{1 \ldots m\}$.

The following definition formalizes the skeleton of the matching automaton generation algorithm. It is not a complete algorithm, since it does not specify a method for selecting expansion columns in configurations. In the following section, we will discuss how to do optimal column selection. A step in the following algorithm consists of choosing an unexpanded configuration pair, selecting a column in the configurations, expanding the configurations based on the selected column, generating a fresh matching automaton state and a collection of transitions from it to the residual configurations obtained as results of expansion. This step is iterated until there are no more configuration pairs to be expanded; at that point the current incomplete matching automaton is a proper matching automaton.

6.4.3 Definition: Let $M = (Q, K, i, V, R)$ be an incomplete matching automaton, and let $(C, D) \in K$ be a configuration pair in M where C and D are configurations over tree automaton $A = (S, T)$ sharing the same tuple of variables. C is a ordinary configuration; D is an input configuration. Let c be a column of C identified by x and let d be the corresponding column in D. Let $\{p_1 \ldots p_k\} = \{\texttt{()} \mid s \in d \text{ and } s \rightarrow \texttt{()} \in T\} \cup \{\texttt{l[z],y} \mid s \in d \text{ and } \exists\, t', t''.\ s \rightarrow \texttt{l[t'],t''} \in T\}$ for some $z, y \notin \mathit{vars}(C) \backslash \{x\}$. Let $C_i = \mathit{expand}(C, c, p_i)$ and $D_i = \mathit{expand}(D, d, p_i)$ and $I_i = \mathit{results}(C_i \cap D_i)$ for each $i \in \{1 \ldots k\}$. Let q be a fresh matching automaton state such that $q \notin Q$.

A *one-step expansion* of M using the configuration pair (C, D) is an incomplete matching automaton $M' = (Q', K', i', V', R')$ where $Q' = Q \cup \{q\}$, and $K' = K \backslash (C, D) \cup \{(C_1, D_1) \ldots (C_k, D_k)\}$, and $i' = i$ if $i \in Q$, or $i' = q$ if $i \in K$, and $V' = V \cup \{q \mapsto x\}$, and $R' = R \cup \{q(x) : p_1 \xrightarrow{I_1} \{(C_1, D_1)\} \ldots q(x) : p_k \xrightarrow{I_k} \{(C_k, D_k)\}\}$. A *complete expansion* of M is a matching automaton obtained by recursively expanding the configuration pairs generated by the previous one-step expansions until there is no more unexpanded configuration pairs.

Returning to the example in Figure 6.10, we can see the algorithm at work. The end result is the same matching automaton as the one shown in Figure 6.2. Observe that configurations C_4, C_5, and C_6 are single-result configurations and hence need not be expanded any further. This example also employs the simplification technique of removing a column all of whose patterns are equivalent, as, for instance, the `u` and `v` columns of C_3 do not carry over to the residual configurations C_5 and C_6. Finally, note that we also drop the rows that contain pattern that are incompatible with the input type. This is why C_3 does not have the row with result 2 and C_4 does not have the rows with results 1 and 3.

6.4.2 Optimal Column Selection

The algorithm we have outlined does not specify how to select expansion columns in configurations for optimal performance. We now complete the algorithm's description by addressing this question. To motivate our column selection approach, we first present a series of examples.

Consider the following configuration with two columns and three results.

`y`	`z`	
`a[Any]`	`Any`	1
`Any`	`Any,c[]`	2
`c[],Any`	`Any`	3

Would it be better to test the contents of `y` or `z`? Testing `y` is sufficient to determine the outcome: depending on whether its root node is labeled by a, b, or c, the answer is 1, 2, or 3 respectively. We say that the first column determines all three results. The second column determines only result 2: if the root node of the value stored in `z` is labeled by c, we can conclude 2; if it is labeled by b, however, we cannot determine the result without testing the contents of `y`.

It would seem that testing `y` first would result in more efficient pattern matching, but, in fact, neither column is preferable as far as the optimality measure proposed above is concerned. The reason that expanding on the first column does not lead to a more efficient matching automaton than expanding on the second is that the latter matching automaton can output 2 without considering the contents of `y` at all. We say that neither column is a *better distinguisher* than the other.

If, however, the first row pattern in the second column were changed from `b[]` to `b[]|c[]`, then the second column would not determine any result and, in that case, testing `y` first would be more efficient. The first column in this case would be a better distinguisher than the second column. (Figure 6.8 shows the two target programs that correspond to choosing `y` or `z` for the initial inspection.)

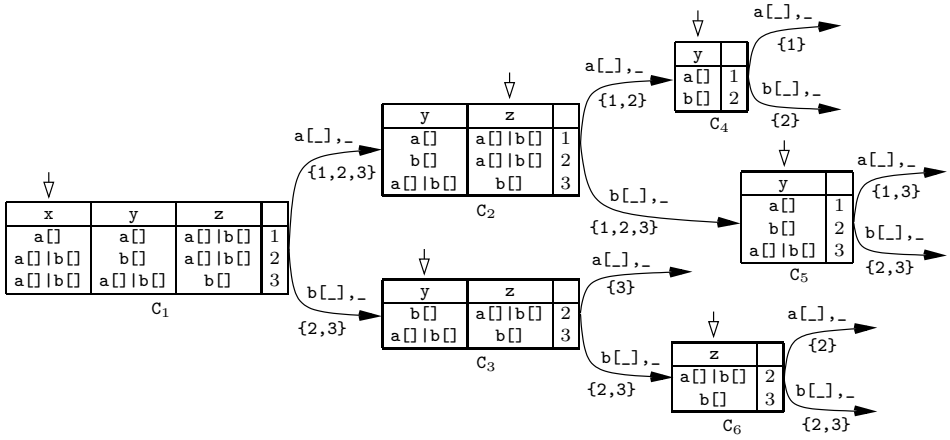


Figure 6.11: Wrong selection of a column in configuration $\mathtt{C}_1$ leads to a suboptimal matching automaton

Sometimes, no single column determines any result. Consider the following configuration.

y	z	
a[]	a[]	1
a[]|b[]	b[]	2
b[]	a[]|b[]	3

It is not possible to arrive at the result by testing the contents of either column alone. Of course, testing the contents of both y and z is sufficient to find the answer. In this case, it does not matter which column is tested first. So, as in the previous example, neither column is a better distinguisher than the other.

The following example shows that even when no column alone determines any result, it is still possible for some column to be better than another. Consider this configuration.

$C =$

x	y	z	
a[]	a[]	a[]|b[]	1
a[]|b[]	b[]	a[]|b[]	2
a[]|b[]	a[]|b[]	b[]	3

As in the previous example, testing any of the three columns alone is not sufficient to determine any result. Unlike the previous example, however, it *does* matter which variable we test first. In particular, it can be shown that testing z or y first is more beneficial than testing x first.

Figure 6.11 shows a potential run of the compilation algorithm for the above configuration. The x column is selected for expansion in the first step. When the contents of x matches a[], the list

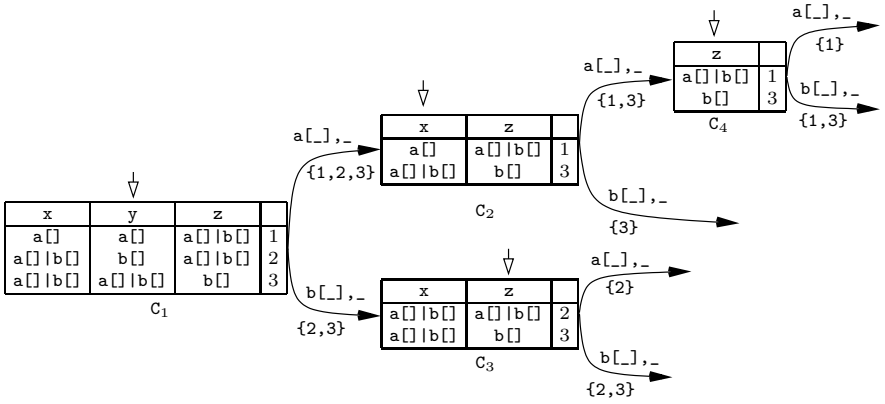


Figure 6.12: Correct column selection results in a matching automaton that is strictly more efficient than the matching automaton of Figure 6.11

of potential pattern matching outcomes cannot be narrowed and further tests must be performed on both `y` and `z`. The matching automaton generated by the algorithm tests `z` first since C_2 is expanded on the `z` column. Configuration C_3 describes the state of the matching automaton that is reached when `x` matches `b[]`. In this case, potential outcomes are reduced to 2 and 3, and it is possible to conclude 3 by testing whether `y` matches `a[]` and skipping `z` completely. If `y` matches `b[]`, however, `z` must be tested to complete pattern matching.

The matching automaton shown in Figure 6.11 is suboptimal. This can be seen by comparing it with a more efficient matching automaton shown in Figure 6.12. In it, testing `y` first allows us not only to conclude 3 without testing `z` as in the previous example but also to arrive at two other outcomes by only testing `y` and `z`—and not `x`—thus outperforming the matching automaton of Figure 6.11. Similarly, for any other matching automaton that starts by testing `x`, we can always build a strictly more efficient matching automaton that starts by testing one of the other two variables.

For this configuration, we say that both `y` and `z` are *better distinguishers* than `x`. We would like to have a formal criterion that allows us to determine whether one column is a better distinguisher than another. Furthermore, we would like this criterion to be semantic so that we can find an optimally distinguishing column without generating and comparing all possible matching automata that can arise from the current configuration.

We will satisfy the above concerns as follows. First, we will introduce *decision trees*, which have the same semantics as matching automata but are higher level. We will define what it means for one decision tree to be strictly more efficient than another. Then, after establishing a correspondence between decision trees and configurations, we will derive the notion of an optimal expansion column.

6.4.4 Definition: A *decision tree* is a tree whose nodes are labeled by variables, whose edges are labeled by regular types, and whose leaves are sets of integer results. A path from the root to a leaf may not contain duplicate variables. We say that an environment E is *accepted* by a decision tree t with result j, written $E \in t \Rightarrow j$, if there exists a path $x_1 \stackrel{p_1}{\rightarrow} x_2 \stackrel{p_2}{\rightarrow} \ldots x_k \stackrel{p_k}{\rightarrow} J$ from the root to a leaf, where $x_1 \ldots x_k$ are the variables labeling nodes of the path starting from the root, $p_1 \ldots p_k$ are the regular types labeling the edges of the path, and J is the leaf result set, such that $j \in J$ and $E(x_i) \in p_i$ for all $i \in \{1 \ldots k\}$.

One decision tree is strictly more efficient than another if it accepts any environment by testing a subset of the variables that must be tested by the other decision tree to accept the same environment.

6.4.5 Definition: A decision tree t_1 is *strictly more efficient* than an equivalent decision tree t_2 if, for any path $x_1 \stackrel{p_1}{\rightarrow} x_2 \stackrel{p_2}{\rightarrow} \ldots x_k \stackrel{p_k}{\rightarrow} J$ in t_2, there exists a path $y_1 \stackrel{q_1}{\rightarrow} y_2 \stackrel{q_2}{\rightarrow} \ldots y_m \stackrel{q_m}{\rightarrow} J$ in t_1 such that, for any $i \in \{1 \ldots m\}$, there exists $j \in \{1 \ldots k\}$ with $y_i = x_j$ and $q_i = p_j$, and, furthermore, there exists a t_2 path for which the corresponding t_1 path is strictly shorter.

A configuration can give rise to a finite number of decision trees. To help identify the set of all decision trees corresponding to a configuration, we first introduce an auxiliary notion of a partition of a set of regular types.

6.4.6 Definition: Let T be an input regular type and $S = \{p_1 \ldots p_m\}$ a set of regular types such that T is a subtype of $p_1 \cup \ldots \cup p_m$. A *partition* of S is a set of mutually disjoint regular types $\{t_1 \ldots t_k\}$ such that $T \cap (p_1 \cup \ldots \cup p_m)$ is a subtype of $t_1 \cup \ldots \cup t_k$ and, for any $i \in \{1 \ldots k\}$ and $j \in \{1 \ldots m\}$, if $t_i \cap p_j$ is non-empty, then t_i is a subtype of p_j.

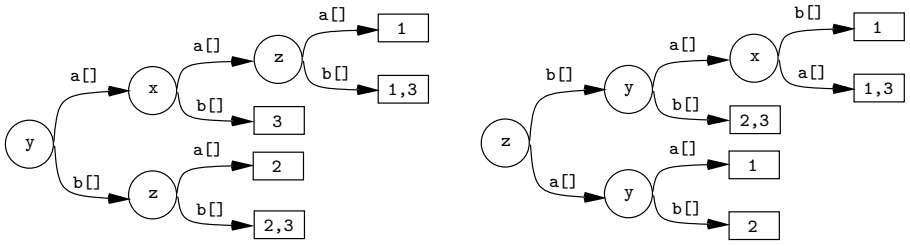
The idea is to use the elements of a partition to indicate which of the original patterns match a given input value. For example, $\{\texttt{a[]}, \texttt{b[]}\}$ is a partition for the input type $T = \texttt{a[]|b[]}$ and the collection of patterns $S = \{\texttt{a[],b[],a[]|b[]}\}$. If a value v is in `a[]`, then it is in the first and third but not in the second patterns of S; if v is in `b[]`, then it is in the second and third but not in the first patterns of S.

A partition of S with respect to T can be obtained by taking all the non-empty types of the form $T \cap p'_1 \cap \ldots \cap p'_m$ where each p'_i is either p_i or $T \backslash p_i$. We say that this is the *minimal partition* of S with respect to T.

6.4.7 Definition: A decision tree t is said to *correspond* to a configuration C with respect to some input configuration C_0 if two conditions hold: 1) edges from a node x are labeled by regular types

x	y	z	
a[]	a[]	a[]|b[]	1
a[]|b[]	b[]	a[]|b[]	2
a[]|b[]	a[]|b[]	b[]	3

(a)



(b) (c)

Figure 6.13: A configuration (a) and two optimal corresponding decision trees (b) and (c)

each of which is a union of some types from the minimal partition of C's column corresponding to x with respect to the union of types in x's column in C_0; and 2) t and C are semantically equivalent; i.e. for any environment $E \in C_0$, we have $E \in t \Rightarrow j$ iff $E \in C \Rightarrow j$.

Given a configuration $C$and an input configuration C_0, it is possible—albeit very time consuming—to generate all decision trees that satisfy the first condition. It is then easy to check whether any such decision tree is semantically equivalent to C. Combining these two steps, we can obtain an algorithm that produces all of C's decision trees.

6.4.8 Definition: Let C be a configuration, C_0 an input configuration, and c one of C's columns associated with variable x. This column is said to be an *optimal distinguisher* if there exists a decision tree corresponding to C with respect to C_0 whose root is labeled by x such that there does not exist a strictly more efficient decision tree corresponding to C with respect to C_0.

Figure 6.13 shows a configuration discussed earlier and two optimal decision trees corresponding to it. The columns associated with z and y are both optimal distinguishers for this configuration. The matching automaton shown in Figure 6.12 was generated using decision tree (b) as a witness of its optimality.

6.4.3 Optimality

Since the compilation algorithm introduced above can be viewed as an instantiation of the type-insensitive algorithm presented in Section 4.5—here the method of selecting expansion columns is specified while there it was left unspecified—the same correctness and termination arguments can be carried over for the algorithm of this paper. Additionally, we can show that the column selection principle introduced above ensures generation of optimal matching automata.

6.4.9 Lemma: Let C be a configuration and C_0 an input configuration over finite regular types. Let q be the complete expansion of C with respect to C_0, and let M be the associated matching automaton. Then there does not exist a matching automaton with a state that is equivalent to C with respect to C_0 and is strictly more efficient than q.

Proof: Assume that there exists a matching automaton M' with a state q' that is equivalent to C with respect to C_0 and is strictly more efficient than q. If both q and q' use the same test variable, follow the equivalent transitions in both M and M' until a pair of states with different test variables is found. Otherwise, q and q' use different test variables. In any case, let q_1 and q'_1 be the corresponding states with different test variables where q_1 is in M and q'_1 is in M' and q'_1 is strictly more efficient than q_1 for q_1's input configuration D_0.

Suppose x and y are q_1's and q'_1's test variables respectively. Let t be the decision tree with x at the root that was used to identify the column associated with x as the expansion column and to generate the transitions originating in q_1 and all the subsequent states of M reachable from q_1. Since q'_1 is strictly more efficient than q_1, we can convert the fragment of M' starting at q'_1 into a decision tree t' with y at the root that is strictly more efficient than t for input D_0. This is a contradiction, since according to the column selection principle, there cannot be a strictly more efficient equivalent decision tree than t. □

The following *monotonicity* property is a corollary of the above lemma. It states that for any matching problem, given a more specific input type, our compilation algorithm generates a matching automaton that is not worse than the one it generates for the same matching problem with a less specific input type.

6.4.10 Corollary: Let C be a configuration, and let C' and C'' be input configurations such that C' is a subconfiguration of C''. Let q' and q'' be the complete expansion of C with respect to C' and C'' respectively. Then q'' is not strictly more efficient than q' for C'.

Proof: By the correctness property of the compilation algorithm, q'' is equivalent to C with respect to C''. Since C' is a subconfiguration of C'', it is also the case that q'' is equivalent to C with respect to C'. Then, by Lemma 6.4.9, q'' cannot be strictly more efficient than q' for C'. □

6.5 Compilation of Recursive Patterns

The algorithm described in the previous section works only for finite patterns; for recursive patterns, it can go into an infinite loop since expansion may not necessarily produce "smaller" residual configurations. Section 4.5 introduced the notion of *loop breakers*—those tree automaton states that may lead to infinite expansion. For configurations with loop breakers, our algorithm used a different expansion technique that produced subroutine transitions and guaranteed termination. The algorithm described in that section was type-insensitive; as a result, it generated a conservatively large set of loop breakers and less efficient matching automata with many subroutine transitions.

Here, we address several issues of type-based optimization in the presence of recursive patterns. First, we demonstrate that using the input type is beneficial for reducing the number of loop breakers. We then point out that selecting a proper set of loop breakers among a number of candidate sets can make a substantial difference in the efficiency of the resulting matching automaton.

6.5.1 Input Types and Loop Breakers

Computing loop breaker sets without regard for the input type can result in an unnecessarily large number of loop breakers. This can lead to generating superfluous subroutine transitions in cases where simple transitions present a more efficient alternative. Consider, for instance the following configuration that is generated by the `match` expression of the program shown in Figure 6.1.

x	
`Any,a[]`	1
`Any`	2

The pattern in the first row is recursive; so, if we were to generate a matching automaton that emulates the above configuration for arbitrary input, we would have to treat that pattern as a loop breaker and resort to using subroutine transitions. Knowing that input values values are restricted to the `a[],(a[]|b[])`, however, allows us to avoid recursion. This can be seen if we intersect the input type with the original patterns thus obtaining the following non-recursive configuration.

```
fun f(T x) : Any =
  match x with
    | a[Any,a[]],Any,a[] → 1
    | a[Any],Any → 2
```

(a)

```
fun f(T x) : Any =
  case x of
    | ~[y],z →
      case y of
        | ~[_],u →
          case u of
            | a[_],_ →
              let pr = A(z) in
              if π1(pr) then 1
              else 2
            else 2
```

(b)

```
fun A(Any x):[bool,bool] =
  case x of
    | () → [false,true]
    | a[y],z →
      case y of
        | () →
          case z of
            | () →
              [true,true]
            else A(z)
        else A(z)
    | ~[_],z → A(z)
```

(c)

Figure 6.14: An example with recursive patterns: a source program (a); an equivalent target program with a subroutine call (b); subroutine function (c); input type `T = a[(a[]|b[]),(a[]|b[])],Any`

x	
a[],a[]	1
a[],(a[]|b[])	2

In the type-insensitive algorithm, loop breakers were computed once and for all before matching automaton generation. As we have shown above, such a strategy does not work efficiently in the framework of the type-propagation algorithm described in Section 6.4 since it leads to unnecessarily identifying many tree automaton states as loop breakers. In the new setting, the loop breaker analysis must be done at each iteration of the algorithm right before the current configuration is expanded. The algorithm computes the intersection of the current configuration and the current input configuration and then finds an appropriate loop breaker set in the obtained configuration.

Consider the example shown in Figure 6.14. The initial configuration and the initial input configuration corresponding to the `match` expression in the source program are as follows:

$C_1 =$

x	
a[Any,a[]],Any,a[]	1
a[Any],Any	2

$C_2 =$

x
a[(a[]|b[]),(a[]|b[])],Any

Since initial configuration pairs can only be expanded by label, there is no need to perform loop breaker analysis at this point yet. The following configurations C_3 and C_4 are the results of expanding by label the above configurations C_1 and C_2 respectively.

```
def A = a[] | a[B]
def B = b[] | a[A]
def C = c[] | a[C]
def D = ()
def T = a[A],B | a[C]
```

(a)

```
fun f(T x) : Any =
  match x with
    | a[A],B → 1
    | a[C] → 2
```

(b)

```
fun f(T x) : Any =
  case x of
    | a[y],z →
      let pr1 = AC(y) in
      let pr2 = BD(z) in
      if π1(pr1) && π1(pr2) then 1
      else 2
```

(c)

```
fun f(T x) : Any =
  case x of
    | a[_],z →
      case z of
        | () → 2
        else 1
```

(d)

Figure 6.15: Effect of selecting loop breakers on optimality: source types (a); a target language processing function (b); an equivalent inefficient target program (c); an optimal target program (d)

$C_3 =$

y	z	
Any,a[]	Any,a[]	1
Any	Any	2

$C_4 =$

y	z
(a[]|b[]),(a[]|b[])	Any

At first glance, both columns of C_3 contain recursive patterns, but if we intersect C_3 with the input configuration C_4, we obtain the following configuration in which only the second column contains loop breakers.

y	z	
(a[]|b[]),a[]	Any,a[]	1
(a[]|b[]),(a[]|b[])	Any	2

The result of expanding this configuration is the program shown in Figure 6.14(b) in which the contents of `z`—corresponding to the second column—is passed to the subroutine, and the contents of `y`—corresponding to the first column—is inspected inline in the body of `f`.

6.5.2 Selecting Among Alternative Loop Breaker Sets

Once the current configuration is intersected with the current input configuration and obviously non-recursive patterns are disregarded as potential loop breakers, there still may be multiple ways of choosing a loop breaker set among the rest of the patterns. It is essential for the compiler to choose a loop breaker set that will not lead to an obviously inefficient target program.

```
def T = () | ~[T]
def A = () | a[A]

fun f(T x) : Any =
  match x with
    | A → 1
    | Any → 2
```

(a)

```
fun f(T x) : Any =
  case x of
    | () → 1
    | a[x],_ → f(x)
    else 2
```

(b)

```
fun f(T x) : Any =
  case x of
    | () → 1
    | a[x],_ →
      case x of
        | () → 1
        | a[y],_ → f(y)
        else 2
    else 2
```

(c)

Figure 6.16: A source program (a); equivalent target program with one recursive call (b); equivalent target program with the recursive call unrolled one level (c)

As an example, let us evaluate a recursive configuration that arises from the source program shown in Figure 6.15(b). (Since the input type in this program equals the union of the `match` expression patterns, we will omit discussing input configurations—they are always equivalent to the current configurations.) After expanding the initial configuration, the compiler will encounter the following configuration.

y	z	
A	B	1
C	D	2

There are two minimal loop breaker sets: {A, C} and {B, C}. If the latter is selected, the above configuration will not have columns without loop breakers and, hence, must be expanded by state. This results in the program shown in Figure 6.15(c). A much more efficient program can be generated if A and C are selected. In this case, the above configuration can be expanded by label on its second column. This will produce single result configurations that need not be expanded further hence avoiding subroutine calls altogether (d). Clearly it is beneficial to select loop breakers that reside in the same column of the current configuration, or, in general, minimize the number of the resulting recursive columns.

6.6 Related Work

Frisch was the first to publish a description of a type-based optimization approach for a language with regular pattern matching [16]. His algorithm is based on a special kind of tree automata called *non-uniform automata*. Like matching automata, non-uniform automata incorporate the notion of "results" of pattern matching (i.e., a match yields a value, not just success or failure). Also, like

matching automata, non-uniform automata support sequential traversal of subtrees. This makes it possible to construct a deterministic non-uniform automaton for any regular language. Unlike matching automata, non-uniform automata impose a left to right traversal of the input value. Whereas it is possible for a matching automaton to scan a fragment of the left subtree, continue on with a fragment of the right, come back to the left and so on, a non-uniform automaton must traverse the left subtree fully before moving on to the right subtree.

Frisch proposes an algorithm that uses type propagation. His algorithm differs from the tree automaton simplification algorithm in that it must traverse several patterns simultaneously (whereas the latter handles one pattern at a time) and generate result sets that will be used in the transitions of the constructed automaton. Frisch's algorithm does not always achieve optimality. In particular, it generates an automaton that tries to learn as much information from the left subtree as possible, even if this information will not be needed in further pattern matching.

In his dissertation [17], Frisch presents a more flexible form of non-uniform automata that allow arbitrary, rather than strictly left-to-right, order of traversal. There is no formal discussion of optimality however.

Outside of the XDuce family, a lot of work has been done in the area of XPath query optimization. Several subsets of XPath have been considered. Wood describes a polynomial algorithm for finding a unique minimal XPath query that is equivalent to the given query [71]. The minimization problem is solved for the set of all documents regardless of their schema. When the schema is taken into account, the problem is coNP-hard. Flesca, Furfaro, and Masciari consider a wider subset of XPath and show that the minimization problem for it is also coNP-hard [13]. They then identify an subset of their subset for which an ad-hoc polynomial minimization is possible.

Genevès and Vion-Dury describe a logic-based XPath optimization framework [26] in which a collection of rewrite rules is used to transform a query in a subset of XPath into a more efficient, but not necessarily optimal, form.

Optimizing full XPath has also been investigated. Gottlob, Koch, and Pichler observe that many XPath evaluation engines are exponential in the worst case. They propose an algorithm that works for full XPath and that is guaranteed to process queries in polynomial time and space. Furthermore, they define a useful subset of XPath for which processing time and space are reduced to quadratic and linear respectively [27, 28]. Fokoue [14] describes a type-based optimization technique for XPath queries. The idea is to evaluate a given query on the schema of the input value obtaining as a result some valuable information that can be used to simplify the query.

At this point, we hesitate to draw deeper analogies between the above XPath-related work and our type-based optimization algorithm since the nature of XPath pattern matching is quite

different from that of regular pattern matching.

Chapter 7

Run-Time System

This chapter addresses the lower-level issue of how to compile XTATIC values and value-constructing primitives into C♯-based run-time representations. We explore several alternative representation choices and analyze them with respect to their support for efficient pattern matching, common XTATIC programming idioms, and safe integration with foreign XML representations such as the standard Document Object Model (DOM). We describe 1) a lazy data structure for sequences of XML trees that efficiently supports repeated concatenation on both ends of a sequence; 2) a representation of textual data (PCDATA) that allows regular pattern matching over character sequences (i.e., statically typed string grep) to be compiled into calls on native .NET regular expression libraries; 3) a type-tagging scheme allowing fast dynamic revalidation of XML values whose static types have been lost, e.g., by upcasting to `object` for storage in a generic collection; and 4) a proxy scheme allowing foreign XML representations such as DOM [67] to be manipulated by XTATIC programs without first translating them to our representation.

We have implemented these designs and measured their performance both against some natural variants and against other implementations of XML processing languages. The results show that a declarative statically typed embedding of XML transformation operations into a stock object-oriented language compares well with existing mainstream XML processing frameworks.

7.1 Representing Trees

We now turn to the design of efficient representations for XML trees. First, we select a tag representation that supports separate compilation and XML namespaces (Section 7.1.1). Next, we design a tree representation that supports XTATIC's view of trees as shared and immutable structures (Section 7.1.2). The main constraint on the design is that the programming style favored

by XTATIC involves a great deal of appending (and consing) of sequences. To avoid too much recopying of sub-sequences, we enhance the naive design to do this appending lazily (Section 7.1.3). Finally, XTATIC needs to inter-operate with other XML representations available in .NET, in particular DOM. We show how DOM structures can masquerade as instances of our XTATIC trees in a type-safe manner(Section 7.1.4).

7.1.1 Tags

Our implementation defines a class `Tag`, and every particular XML tag is an *object* of this class. A tag object has a string field for the tag's local name and a field for its namespace URI. We use memoisation (interning) to ensure that there is a single run-time object for each known tag, making tag matching a simple matter of physical object comparison. Separate compilation is supported by allocating these tags at start-up time: each separately compiled library adds its tags to a common hash table when loaded. Hence, every library associates the same object to a given tag. Moreover, this tag representation simplifies the recovery of a tag name, needed to print it.

We considered several other run-time representations of tags. One, directly corresponding to the formal definition of the XTATIC data model [23], encodes XML tags by different *classes*. This approach does not work in the context of separate compilation since the same XML tag occurring in different compilation units would be mapped to *distinct* classes sharing the same name but residing in different assemblies. Representing tags by values of an enumeration type offers the ability to compile pattern-matching into efficient `switch` statements, but, like the class-based approach, does not work with separate compilation, since we cannot guarantee that the same tag will correspond to the same enumeration value in every compilation unit. One might also represent an XML tag by a single string containing both the namespace and the tag name separated by some special character, and hash this string using a fixed function carefully chosen as to minimize collisions. This approach, similar to the one used for the implementation of labels in OCaml [25], is not applicable in our setting as the name of the tag is no longer available at runtime.

7.1.2 Simple Sequences

Every XTATIC value with a regular type is a *sequence* of trees. XTATIC's pattern-matching algorithms, based on tree automata, require access to the label of the first tree in the sequence, its children, and its following sibling. This access style is naturally supported by a simple singly linked structure.

Figure 7.1 summarizes the classes implementing sequences. `Seq` is an abstract superclass representing all sequences regardless of their form. As the exact class of a `Seq` object is often needed

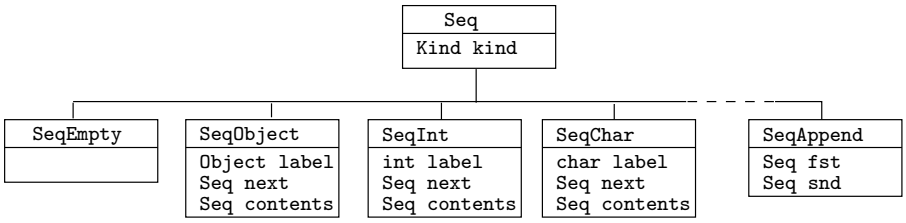


Figure 7.1: Classes used for representing sequences.

by XTATIC-generated code, such as pattern matching, it is stored as an enumeration value in the field `kind` of every `Seq` object. Maintaining this field allows us to use a `switch` statement (which should be implemented by a good .NET JIT compiler using a jump table) instead of a chain of `if-then-else` statements relying on the "`is`" operator to test class membership.

The subclass `SeqObject` includes two fields, `next` and `contents`, that point to the rest of the sequence—the right sibling—and the first child of the node. The field `label` holds a C♯ object. Empty sequences are represented using a single, statically allocated object of class `SeqEmpty`. (Using `null` would require an extra test before `switch`ing on the kind of the sequence—in effect, optimizing the empty-sequence case instead of the more common non-empty case.)

In principle, the classes `SeqEmpty` and `SeqObject` can encode all XTATIC trees. But to avoid downcasting when dealing with labels containing primitive values (most critically, characters), we also include specialized classes `SeqBool`, `SeqInt`, `SeqChar`, etc. for storing values of base types.

XML data is encoded using `SeqObject`s that contain, in their `label` field, instances of the special class `Tag` that represent XML tags, as described in Section 7.1.1.

Briefly, pattern matching of labels is implemented as follows. The object (or value) in a label matches a label pattern when: the pattern is a class `C` and the object belongs to a subclass of `C`, the pattern is a tag and the object is physically equal to the tag, the pattern is a base value `v` and the label holds a value equal to `v`.

7.1.3 Lazy Sequences

In the programming style encouraged by XTATIC, sequence concatenation is a pervasive operation. Unfortunately, the run-time representation outlined so far renders concatenation linear in the size of the first sequence, leading to unacceptable performance when elements are repeatedly appended at the end of a sequence, as in the assignment of `res` in the `addrbook` example in Chapter 3.

This observation naturally suggests a lazy approach to concatenation:[1] we introduce a new kind

[1]The problem of efficient list concatenation has, of course, been studied in the functional programming community,

```
Seq lazy_norm(Seq node) {
  switch (node.kind) {
    case Append: return norm_rec(node.fst, node.snd);
    default:     return node;   }  }

Seq norm_rec(Seq node, Seq acc) {
  switch (node.kind) {
    case Append: return norm_rec(node.fst, new SeqAppend(node.snd, acc));
    case Object:
      switch node.next.kind {
        case Empty: return new SeqObject(node.label, node.contents, acc);
        default:    return new SeqObject(node.label, node.contents,
                                         new SeqAppend(node.next, acc));
      }
    /* similar cases for SeqInt, SeqBool, ... */   }  }
```

Figure 7.2: Lazy Normalization Algorithm.

of sequence node, `SeqAppend`, that contains two fields, `fst` and `snd`. The concatenation of (non-empty) sequences `Seq1` and `Seq2` is now compiled into the constant time creation of a `SeqAppend` node, with `fst` pointing to `Seq1`, and `snd` to `Seq2`. We preserve the invariant that neither field of a `SeqAppend` node points to the empty sequence.

To support pattern matching, we need a *normalization* operation that exposes at least the first element of a sequence. The simplest approach, *eager* normalization, just transforms the whole sequence so that it does not contain any top-level `SeqAppend` nodes (children of the nodes in the sequence are not normalized). However, there are cases when it is not necessary to normalize the whole sequence, e.g. when a program inspects only the first few elements of a long list. To this end we introduce a *lazy* normalization algorithm, given in pseudocode form in Figure 7.2.

The algorithm fetches the first concrete element—that is, the leftmost non-`SeqAppend` node of the tree—copies it (so that the contexts that possibly share it are not affected), and makes it the first element of a new sequence consisting of (copies of) the traversed `SeqAppend` nodes arranged into an equivalent, but right-skewed tree. Figure 7.3 illustrates this algorithm, normalizing the sequence starting at node $\texttt{SeqAppend}_6$ to the equivalent sequence starting at node $\texttt{SeqObject}'_4$.

Since parts of sequence values are often shared, it is not uncommon to process (and normalize) the same sequence several times. As described so far, the normalization algorithm returns a new sequence, e.g. $\texttt{SeqObject}'_4$, but leaves the original lazy sequence unchanged. To avoid redoing the same work during subsequent normalizations of the same sequence, we also modify *in-place* the root `SeqAppend` node, setting the `snd` field to `null` (indicating that this `SeqAppend` has been

and a number of techniques have been proposed; see Section 7.5. We describe here our adaptation of these ideas to the specifics of XTATIC—for example, in-place updates will turn out to be critical for the correctness of pattern variable binding.

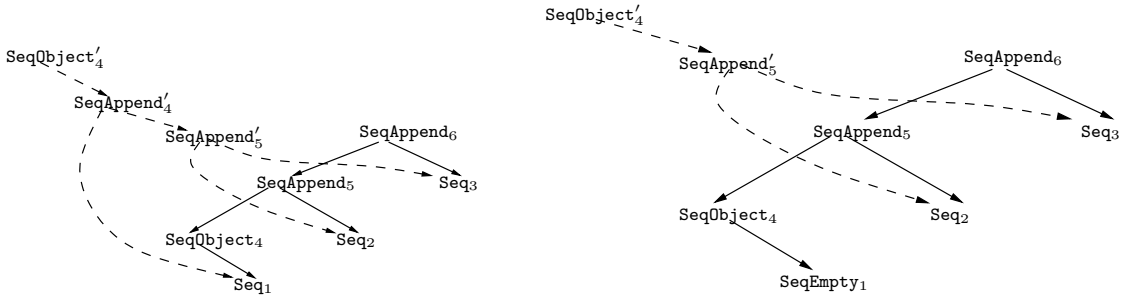


Figure 7.3: Lazy normalization of lazy sequences. In (a), the leftmost concrete element has a right sibling; in (b) it does not. Dotted pointers and their source objects are created during normalization.

normalized), and the `fst` field to the result of normalization:

```
Seq lazy_norm_in_place(Seq node) {
  switch (node.kind) {
    case Append:
      if (node.snd == null) return node.fst;
      node.fst = norm_rec(node.fst, node.snd); node.snd = null;
      return node.fst;
    default: return node;   }   }
```

Interestingly, this in-place modification is required for the *correctness* of binding of non-tail variables in patterns. The pattern matching algorithm [35] naturally supports only those pattern variables that bind to tails of sequence values; variables binding to non-tail sequences are handled by a trick. Namely, binding a non-tail variable `x` is accomplished in two stages. The first stage performs pattern matching and—as it traverses the input sequence—sets auxiliary variables $\mathtt{x}_b$ and $\mathtt{x}_e$ to the beginning and end of the subsequence. The second stage computes `x` from $\mathtt{x}_b$ and $\mathtt{x}_e$ by traversing the sequence beginning at $\mathtt{x}_b$ and copying nodes until it reaches $\mathtt{x}_e$. In both stages, the program traverses the same sequence, performing normalization along the way. In-place modification guarantees that during both traversals we will encounter *physically* the same concrete nodes, and so, in the second stage, we are justified in detecting the end of the subsequence by checking physical equality between the current node and $\mathtt{x}_e$.

Because of creation of fresh `SeqAppend` nodes, the lazy normalization algorithm can allocate more memory than its eager counterpart. However, we can show that this results in no more than a constant factor overhead, as follows. A node is said to be a *left node* if it is pointed by the `fst` pointer of a `SeqAppend`. There are two cases when the algorithm creates a new `SeqAppend` node:

	eager concatenations	eager normalization	lazy normalization
back appending	∞	1,050 ms	1,050 ms
front appending	950 ms	950 ms	950 ms

Figure 7.4: Running times for two variants of the phone book application.

when it traverses a left `SeqAppend` node, and when it reaches the leftmost concrete element. In both cases, the newly created nodes are *not* left nodes and so will not lead to further creation of `SeqAppend` nodes during subsequent normalizations. Hence, lazy normalization allocates at most twice as much memory as eager normalization.

We now present some measurements quantifying the consequences of this overhead on running time. Figure 7.4 shows running times for two variants of the phone book application from Chapter 3, executed on an address book of $250,000$ entries. (Our experimental setup is described below in Section 7.4.) The first variant constructs the result as in Chapter 3 to the end. The second variant constructs the result by by appending to the front:

```
res = person[n,t], res;
```

This variant favors the non-lazy tree representation from the previous subsection, which serves as a baseline for our lazy optimizations. Since our implementation recognizes prepending singleton sequences as a special case, no lazy structures are created when the second program is executed, and, consequently all concatenation approaches behave the same. For the back-appending program, the system runs out of memory using eager concatenation, while both lazy concatenation approaches perform reasonably well. Indeed, the performance of the lazy representations for the back-appending program is within 10% of the performance of the non-lazy representation for the front-appending program, which favors such a representation.

This comparison does not show any difference between the lazy and eager normalization approaches. We have also compared performance of eager vs. lazy normalization on the benchmarks discussed below in Section 7.4. Their performance is always close, with slight advantage for one or the other depending on workload. On the other hand, for programs that explore only part of a sequence, lazy normalization can be *arbitrarily* faster, making it a clear winner overall.

Our experience suggests that, in common usage patterns, our representation exhibits constant amortized time for all operations. It is possible, however, to come up with scenarios where repeatedly accessing the first element of a sequence may take linear time for each access. Consider the following program fragment:

```
Any res1 = ();
```

```
Any res2 = ();
while true do
  res1 = res1, a[];
  res2 = res1, b[];
  match res2 with
    (Tag x)[], Any → ...use x...
```

Since the pattern matching expression extracts only the first element of `res2`, only the top-level `SeqAppend` object of the sequence stored in `res2` is modified in-place during normalization. The `SeqAppend` object of the sequence stored in `res1` is not modified in-place, and, consequently, is completely renormalized during each iteration of the loop.

Kaplan, Tarjan and Okasaki [42, 43, 60, 41] describe *catenable steques*, which provide all the functionality required by XTATIC pattern-matching algorithms with operations that run in constant amortized time in the presence of sharing. We have implemented their algorithms in C♯ and compared their performance with that of our representation using the lazy normalization algorithm. The steque implementation is slightly more compact—on average it requires between 1.5 and 2 times less memory than our representation. For the above tricky example, catenable steques are also fast, while XTATIC's representation fails on sufficiently large sequences. For more common patterns of operations, however, our representation is more efficient. The following table shows running times of a program that builds a sequence by back-appending one element at a time and fully traverses the constructed sequence. We ran the experiment for sequences of four different sizes.

	Steques	XTATIC
n = 10,000	70 ms	6 ms
n = 20,000	140 ms	12 ms
n = 30,000	230 ms	19 ms
n = 40,000	325 ms	31 ms

The implementation using catenable steques is significantly slower than our much simpler representation because of the overhead arising from the complexity of the steque data structures.

7.1.4 DOM Interoperability

XTATIC modules are expected to be useful in applications built in other .NET languages with the use of extensive .NET libraries. The latter already contain support for XML, collected in the `System.Xml` namespace. It can greatly enhance the usefulness of XTATIC if its XML manipulation facilities can be applied to native .NET XML representations and, conversely, if XTATIC XML

data can be accessed from vanilla C♯ code. We have explored the former direction of this two-sided interoperability problem by implementing support for DOM, one popular XML representation available in .NET.

A straightforward solution for accessing DOM from XTATIC would be to translate any DOM data of interest into our representation in its entirety. This is wasteful, however, if an XTATIC program ends up accessing only a small portion of the document. A better idea is to wrap a DOM fragment in a lazy structure (using another subclass, called `SeqDom`, of `Seq`) and investigate its contents only as needed during pattern matching. However, since DOM structures are mutable, we need to take some care to maintain type safety. We do this by investigating the underlying DOM structure just once and copying the parts we have seen into immutable `Seq` nodes.

Concretely, a `SeqDom` object has two fields, `dom` and `seq`, one of which is always `null`. When a `SeqDom` object is created, its `dom` field points to a DOM element node, meaning that the object represents the XML fragment consisting of the DOM node, its following siblings, and its children nodes. (The DOM element node's pointers to its parent and previous siblings are not relevant for the meaning a `SeqDom` wrapper, since XTATIC never needs to traverse them.)

The actual inspection of the underlying DOM nodes happens during normalization. In the most common case, normalization of a `SeqDom` object considers its underlying DOM element node (call it `e`) and creates a new `SeqObject` object with the `label` field corresponding to `e.Name` and fields `contents` and `next` pointing to newly created `SeqDom` objects corresponding to the DOM nodes `e.FirstChild` and `e.NextSibling`. In cases where either of the latter two DOM nodes is not an element node—i.e. it is either `null` or a DOM text node—the normalization transforms it directly to a `SeqEmpty` or an appropriate XTATIC `pcdata` representation. Finally, the normalization modifies the original `SeqDom` object by setting the `dom` field to `null` and pointing the `seq` field to the newly created `SeqObject`.

A `SeqDom` object can be initially obtained by applying a special library function to a DOM node. The return type of this function is `xml`, the least precise XML type. A successful pattern match of such a value against a pattern more detailed than `xml` has the side-effect of transforming, via the normalizations that get invoked along the way, some initial "spine" of the value into our native XTATIC representation. If the pattern contains subpatterns typed as `xml`, the corresponding value fragments may safely be put in `SeqDom` wrappers; it does not matter if the underlying DOM structures are later modified, since no assumptions about their actual type have yet been made by XTATIC code.

As a consequence, the parts of the original DOM structure may be destructively updated at any time after it was wrapped in `DomSeq`. The results of such updates are visible to XTATIC

only if they happen before pattern matching reaches them. (Caveat: This statement applies to a single-threaded context. It remains true in a multithreaded setup only when methods of the DOM implementation are thread safe—something that .NET DOM implementation does not guarantee.)

Using `SeqDom` wrappers in the context of lazy concatenation with subsequent normalization of the resulting structures into concrete XTATIC sequences is crucial to the efficiency of our approach. An alternative design that implemented concatenation of DOM wrappers as a DOM wrapper and at the same time wanted to support shared-structure view of data would have to do extensive deep cloning of DOM fragments. Otherwise, the doubly-linked nature of DOM structures could lead to unintended sharing violations.

We plan to address the other direction of the interoperability problem—efficiently *exporting* XML trees created by XTATIC for use in native C♯ code—by implementing one of the `System.Xml` access interfaces on top of XTATIC sequences.

7.2 Representing Text

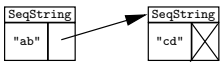
In this section we describe several ways of representing `pcdata` and weigh the merits of each approach.

The definition of the type `pcdata` as `<(char)/>*` immediately suggests a naive representation using linked lists of `SeqChar` objects. For example, the text `‘abcd‘` would be represented as:

The primary advantage of this representation is that we can directly use our sequence representation and tree pattern matching algorithms to inspect textual data. The primary disadvantage is also obvious: extremely inefficient use of memory, as each character is represented by a `SeqChar` object.

A more compact alternative is to use native C♯ strings for representing `pcdata`. We have explored several variants.

The simplest of these is to extend the `Seq` class hierarchy with class `SeqString`, whose objects store a `string` value plus a pointer to the next `Seq` object. Here is one possible representation for the text `'abcd'`:



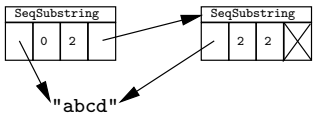
As the example suggests, a single `SeqString` node does not have to encapsulate an entire run of consecutive text. In the case where characters are added to the sequence one by one, this actually results in memory usage less efficient than the naive `SeqChar` representation, as a one-character

string takes more space than a single `char`. Moreover, the following program illustrates a common case where the new representation results in worse behavior than the naive representation. It scans a given chunk of text character by character and replaces every occurrence of consecutive 'a's by a 'b'.

```
fun process(pcdata txt) : pcdata =
  pcdata res = ();
  while true do
    match txt with
      () →
        return res;
      'a'+, Any rest →
        res = res, 'b';
        txt = rest;
      (char)[] x, Any rest →
        res = res, x;
        txt = rest;
```

In each iteration of the loop, the content of `rest` is a string that needs to be extracted—hence copied—from the current value of `txt`, resulting in quadratic space and time behavior. For this program, the naive `SeqChar` implementation does not need to allocate any new sequences for `rest`.

We can avoid this problem by refining the `SeqString` representation to point to a shared string buffer and maintain a starting offset and length. The new `pcdata` representation is encoded by class `SeqSubstring` as illustrated in the following figure.



There are cases where the `SeqSubstring` representation is less efficient than the `SeqString` scheme because of the two extra memory slots. Therefore, it is advantageous to have the three schemes coexist and choose adaptively (at instantiation time) which one to use: `SeqChar` when we find ourselves appending a single character to a sequence, `SeqString` when we append a whole string, and `SeqSubstring` when we append a piece extracted from an existing string.

An interesting side effect of the string representation is that it allows us to use the .NET regular expression library `System.Text.RegularExpressions` for pattern matching over pure PCDATA. Some advantages for this choice are ease of implementation, leveraging the highly optimized .NET library (which, for example, translates regular expressions into CLR bytecodes at run time and JIT-compiles them), and the size reduction of XTATIC-generated C♯ code that results from delegating

	SeqChar	SeqString	SeqSubstring	Adaptive
process	47 Mb / 2,500 ms	∞	52 Mb / 4,800 ms	44 Mb / 4,500 ms
addrbook	249 Mb / 250 ms	77 Mb / 180 ms	68 Mb / 180 ms	68 Mb / 180 ms

Figure 7.5: Comparison of `pcdata` representations.

to library calls all the character matching code that otherwise would have to be generated. One drawback is that it requires that lazy sequences of characters be coalesced into a single `string` object that can be passed to the regular expression engine. When a string is needed for .NET regular expression matching, the sequence is (eagerly) normalized to eliminate `SeqAppends` up to the first non-character representing item and the resulting list of strings is concatenated.

We conclude this section with performance measurements from two experiments (Figure 7.5). The first is the program `process` given earlier in this section, evaluated on a 0.5 Mb `pcdata` file; the second is the `addrbook` example run on an XML document containing 250,000 `APers` elements. In both experiments, we measure the memory usage and processing time of the program. In the first experiment, we take a memory checkpoint right after `process` finished its work and built the result; in the second, right after the input document is loaded in memory.

In the first experiment, the `SeqString` representation did not complete because of its quadratic behavior during suffix extraction. At the time when we take the memory measurement, the document is fully fragmented and each character is boxed in a `SeqChar` or `SeqSubstring` object. Since the latter is larger, the `SeqChar` representation is more compact than the `SeqSubstring` representation for this program. The `SeqChar` scheme is also faster. The main reason of this is that suffix extraction does not perform allocation whereas in the `SeqSubstring` representation, a suffix is obtained by creating a new `SeqSubstring` object.

To see how performance of XTATIC's pattern matching over `pcdata` compares with performance of native string manipulation, we hand-coded a C$^\sharp$ program that implements the behavior of the `process` example. Instead of scanning each character of the input string, this program goes directly to the next substring matching `a+` and concatenates the intermediate substrings using C$^\sharp$'s `StringBuilder`. This program completes in 60 ms and uses 3.5 Mb. This shows that XTATIC pays a heavy price for treating `pcdata` generically and doing a lot of boxing and for forcing character for character traversal.

The results of the second experiment contrast with the first. Since `addrbook` does not pattern match over `pcdata`, no fragmentation takes place, and we can benefit from a compact `pcdata` representation. As expected, the `SeqChar` scheme proves to be substantially less efficient than the others. Observe that `SeqSubstring` is more memory efficient than `SeqString` even though

`SeqString` objects are smaller. The reason of this is that with `SeqSubstring`, we load all `pcdata` chunks into a single string buffer and avoid creating a large number of string objects.

The approaches presented in this section are in no way exhaustive. We are planning to extend the XTATIC pattern matching algorithm to experiment with several strategies to directly match strings and to compare these with the native .NET regular expression approach.

7.3 Calling Hell From Heaven

The homogeneous translation scheme raises some issues related to calls between XTATIC and C♯. One is static in nature and is concerned with overloaded methods whose signatures are different in XTATIC but are mapped to indistinguishable C♯ signatures. Another is a dynamic issue addressing efficient retrieval of XTATIC values from homogeneous C♯ containers. We consider these issues in turn.

7.3.1 Method Overloading

It is natural for XTATIC to extend C♯ method overloading to support method signatures that contain regular types. This conflicts somewhat with our homogeneous translation scheme, which would translate all such types to `Seq`, possibly resulting in shortage of signatures that could differentiate the original methods.

We resolve this problem by generating new names for all methods having arguments of types more precise than `Seq` (or, equivalently, `[[any]]`). (The renaming scheme, however, has to take into account method signatures so that, e.g., an overriding method receives the same modified name as the method it overrides). As it follows, this name mangling is not applied to methods whose regular type arguments are all of class `Seq`—this is the case that can be handled by C♯ overloading itself.

Similar treatment, for identical safety reasons, is applied to class fields.

This approach fits well with the needs of separate compilation, when a pre-compiled XTATIC library is to be used for programming larger applications, either using XTATIC or pure C♯:

- In addition to translating XTATIC code to C♯ our compiler also preserves in a separate structure signatures of methods with regular argument types.[2] Then, an XTATIC library consists of this information together with an assembly generated by a C♯ compiler.

[2]Currently this structure is a separate file, but we explore if the corresponding information can be stored directly in .NET assemblies.

- When an XTATIC program is compiled against the library, the preserved signature information is used to resolve overloaded method calls into appropriate mangled method names.
- A C♯ program compiled against the XTATIC library is expected to refer only to non-mangled method names,[3] that is method names with regular type arguments at most as precise as `Seq`.

Consequently, if a creator of an XTATIC library needs to expose to pure C♯ code a method operating on values of a regular type, say `[[Person]]`, he needs to explicitly create a method accepting `Seq` values, casting them to `[[Person]]`, and then performing the intended functionality.

The latter scenario can be contrasted with an entirely different (hypothetical) support for method overloading that would translate a method with regular type arguments to a method with `Seq` arguments, automatically generating appropriate casting code. We believe that our simpler solution, by exposing to the programmer need to perform potentially expensive cast operations, can result in better program designs, as well as more customized error reporting and recovery.

7.3.2 Fast Downcasting

Part of the appeal of XTATIC is that it allows programmers to use familiar C♯ libraries to store and manipulate XML values; in particular, XML values can be stored in generic collections such as `Hashtable` and `Stack`. However, extracting values from such containers requires a downcast from `object` to the intended type of the value. In pure C♯ this operation incurs only a small time overhead, but in XTATIC, downcasting to a regular type may involve an expensive structural traversal of the entire value. To avoid this overhead, we need a way to *stamp* sequence values with a representation of their type and perform a run-time type comparison rather than full re-validation during downcasting. Our design places this stamping under programmer control.

We begin by extending the source language with a stamping construct, written `<[[T]]>e` ("stamp `e` with regular type `T`"). An expression of this form is well typed if `e` has static type `T`; in this case, the result type of the whole expression is `object`. At run-time, stamped sequences are represented by objects of class `StampedSeq`, with two fields: `stamp`, of type `Typestamp`, and `contents`, of type `Seq`. (Giving stamped values type `object` ensures that, at run-time, such values will never appear as part of other sequences. This makes sequence operations such as concatenation, normalization, and prefix extraction simpler and more efficient.)

For each regular type `T` appearing in a stamp or cast expression in the program, the compiler generates initialization code that hashes `T` type to an object of class `Typestamp`.

[3] Technically, it is possible to violate safety by finding out the precise mangled names of methods. We are not aware, however, about protection for mangled methods more substantial than security through obscurity.

A cast expression of the form `([[T]])e` is executed by first checking whether the value of `e` has class `StampedSeq`; if so, it extracts the stamp, checks whether it is identical to the hash of `T`, and returns the sequence stored in the `contents` field of the `StampedSeq` object; if this fails, it falls back to the general pattern-matching algorithm, which dynamically re-validates the value.

A small experiment demonstrates the benefits of type-stamping. Consider an obfuscated program for reversing sequences belonging to the type `APers*` introduced in Chapter 3. We traverse the sequence and put each element in a C♯ queue. Then we dequeue one element at a time, cast it to `APers`, and add it to the result. We ran this program on two XML documents, each containing 30,000 `APers` elements. In the first document, each `APers` element has exactly one `email` child, in the second, twenty. For each document we tried two versions of the program: one with type-stamping, and one without.

	without type-stamping	with type-stamping
1 email	33 ms	28 ms
20 emails	89 ms	28 ms

On the document with single `email`s, type stamping yields only a small performance improvement, since the overhead of adding and checking the type stamps is roughly equivalent to the cost of pattern-matching a small `APers` element. In the other case, the benefits of type-stamping are clear—the type-stamping version of the program is three times faster.

In this design, the burden of type stamping is placed on the programmer. We have experimented with alternative designs in which stamping is performed silently—either by adding a stamp whenever a sequence value is upcast to type `object` or by including a type stamp in every sequence object. However, we have not found a design in which the performance costs of stamping and stamp checking seem acceptably predictable. The difficulty is that there are infinitely many equivalent representations of the static type of a given sequence value. Because the process of stamping is invisible, the programmer has no way of predicting which of these representations will actually appear in the stamp. Thus, rather than the simple syntactic-identity check used above, we must ensure that *any* representation will produce the same effect—i.e., we must perform full equivalence checking between stamps at run time. Indeed, to avoid requiring the programmer to calculate the minimal types of sequence values (because only this type will satisfy an equivalence check), we would need to perform full subtype testing at run time. This is problematic, since—although common cases of subtype checking for regular types can be optimized sufficiently to make the compiler's front end acceptably fast in practice [37]—in general subtype testing can take time exponential in the size of the types. Such a potentially costly operation should not be applied automatically.

7.4 Measurements

This section describes performance measurements comparing XTATIC with some other XML processing systems. Our goal in gathering these numbers has been to verify that our current implementation gives reasonable performance on a range of tasks and datasets, rather than to draw detailed conclusions about relative speeds of the different systems. (Differences in implementation platforms and languages, XML processing styles, etc. make the latter task well nigh impossible!)

Our tests were executed on a 2GHz Pentium 4 with 512MB of RAM running Windows XP. The XTATIC and DOM experiments were executed on Microsoft .NET version 1.1. The CDUCE interpreter (CVS version of November 25th, 2003) was compiled natively using ocamlopt 3.07+2. QIZX/OPEN and Xalan XSLTC were executed on SUN JAVA version 1.4.2. Since this chapter is concerned with run-time data structures, our measurements do not include static costs of typechecking and compilation. Also, since the current implementation of XTATIC's XML parser is inefficient and does not reveal much information about the performance of our data model, we factor out parsing and loading of input XML documents from our analysis. Each measurement was obtained by running a program with given parameters ten times and averaging the results. We selected sufficiently large input documents to ensure low variance of time measurements and to make the overhead of just-in-time compilation negligible. The XTATIC programs were compiled using the hybrid `pcdata` encoding described in Section 7.2 and the lazy append with lazy normalization policy described in Section 7.1.

We start by comparing XTATIC with the QIZX/OPEN [15] implementation of XQUERY. Our test is a small query named `shake` that counts the number of distinct words in the complete Shakespeare plays, represented by a collection of XML documents with combined size of 8Mb.

	`shake`
XTATIC	7,500 ms
QIZX/OPEN	3,200 ms

The core of the `shake` implementation in XQUERY is a call to a function `tokenize` that splits a chunk of character data into a collection of white-space-separated words. In XTATIC, this is implemented by a generic pattern matching statement that extracts the leading word or white space, processes it, and proceeds to handle the remainder of the `pcdata`. Each time, this remainder is boxed into a `SeqSubstring` object, only to be immediately unboxed during the next iteration of the loop. We believe this superfluous manipulation is the main reason why XTATIC is more than twice slower than QIZX/OPEN in this example.

We also implemented several XQUERY examples from the XMark suite [61], and ran them

on an 11MB data file generated by XMark (at "factor 0.1"). XTATIC substantially outperforms QIZX/OPEN on all of these benchmarks—by 500 times on `q01`, by 700 times on `q02`, by six times on `q02`, and by over a thousand times on `q08`. This huge discrepancy appears to be a consequence of two factors. Firstly, QIZX/OPEN, unlike its commercial counterpart, does not use indexing, which for examples such as `q01` and `q02` can make a dramatic performance improvement. Secondly, we are translating high-level XQUERY programs into low-level XTATIC programs—in effect, performing manual query optimization. This makes a comparison between the two systems problematic, since the result does not provide much insight about the underlying representations.

Next, we compare XTATIC with two XSLT implementations: .NET XSLT and Xalan XSLTC. The former is part of the standard C♯ library; the latter is an XSLT compiler that generates a JAVA class file from a given XSLT template.

We have implemented several transformations from the XSLTMark benchmark suite [9]. The `backwards` program traverses the input document and reverses every element sequence; `identity` copies the input document; `dbonerow` searches a database of person records for a particular entry, and `reverser` reads a `PCDATA` fragment, splits it into words, and outputs a new `PCDATA` fragment in which the words are reversed. The first three programs are run on a 2MB XML document containing 10,000 top-level elements; the last program is executed on a small text fragment.

	`backwards`	`identity`	`dbonerow`	`reverser`
XTATIC	450 ms	450 ms	13 ms	2.5 ms
.NET XSLT	2,500 ms	750 ms	300 ms	9 ms
Xalan XSLTC	2,200 ms	250 ms	90 ms	0.5 ms

XTATIC exhibits equivalent speed for `backwards` and `identity` since the cost of reversing is approximately equal to the cost of copying a sequence in the presence of lazy concatenation. The corresponding XSLT programs behave differently since `backwards` is implemented by copying *and* sorting every sequence according to the position of the elements. The XSLT implementations are relatively efficient on `identity`. This may be partially due to the fact that they use a much more compact read-only representation of XML documents. XTATIC is substantially slower than Xalan XSLTC on the `pcdata`-intensive `reverser` example. We believe the reason for this is, as in the case of `shake` in the comparison with QIZX/OPEN, the overhead of our `pcdata` implementation for performing text traversal. Conversely, XTATIC is much faster on `dbonerow`. As with QIZX/OPEN, this can be explained by the difference in the level of programming detail—a single XPATH line in the XSLTC program corresponds to a low-level XTATIC program that specifies how to search the input document efficiently.

In the next pair of experiments, we compare XTATIC with CDUCE [4] on two programs:

`addrbook` and `split`. The first of these was introduced in Chapter 3 (the CDuce version was coded to mimic the Xtatic version, i.e., we did not use CDuce's higher-level `transform` primitive); it is run on a 25MB data file containing 250,000 `APers` elements. The second program traverses a 5MB XML document containing information about people and sorts the children of each person according to gender.

	`split`	`addrbook`
Xtatic	950 ms	1,050 ms
CDuce	650 ms	1,300 ms

Although it is difficult to compare programs executed in different run-time frameworks and written in different source languages, we can say that, to a rough first approximation, Xtatic and CDuce exhibit comparable performance. An important advantage of CDuce is a very memory-efficient representation of sequences. This is compensated by the fact that Xtatic programs are (just-in-time) compiled while CDuce programs are interpreted.

The next experiment compares Xtatic with Xact [47]. We use two programs that are part of the Xact distribution—`recipe` processes a database of recipes and outputs its HTML presentation; `sortedaddrbook` is a version of the address book program introduced in Chapter 3 that sorts the output entries. We ran `recipe` on a file containing 525 recipes and `sortedaddrbook` on a 10,000 entry address book.[4]

	`recipe`	`sortedaddrbook`
Xtatic	250 ms	1,600 ms
Xact	60,000 ms	10,000 ms

For both programs Xtatic is substantially faster. As with XQuery, this comparison is not precise because of a mismatch between XML processing mechanisms of Xtatic and Xact. In particular, the large discrepancy in the case of `recipe` can be partly attributed to the fact that its style of processing in which the whole document is traversed and completely rebuilt in a different form is foreign to the relatively high level XML manipulation primitives of Xact but is quite natural to the relatively low level constructs of Xtatic.

The last experiment compares Xtatic with a C♯ program using DOM and the .NET XPath library, again using the `addrbook` example on the 25MB input file. The C♯ program employs XPath to extract all the `APers` elements with `tel` children, destructively removes their `email` children, and returns the obtained result.

[4]Because of problems installing Xact under Windows, unlike the other experiments, comparisons with Xact were executed on a 1GHz Pentium III with 256MB of RAM running Linux.

	addrbook
XTATIC	1,050 ms
DOM/Xpath	5,100 ms

This experiment confirms that DOM is not very well-suited for the kind of functional manipulation of sequences prevalent in XTATIC. The DOM data model is geared for destructive modification and random access traversal of elements and, as a result, is much more heavyweight.

7.5 Related Work

We have concentrated here on the runtime representation issues that we addressed while building an implementation of XTATIC that is both efficient and tightly integrated with C♯.

There is considerable current research and development activity aimed at providing convenient support for XML processing in both general-purpose and domain-specific languages. In the latter category, XQUERY [74] and XSLT [68] are special-purpose XML processing languages specified by W3C that have strong industrial support, including a variety of implementations and wide user base. In the former, the CDUCE language of Benzaken, Castagna, and Frisch [19, 4] generalizes XDUCE's type system with intersection and function types. The XEN language of Meijer, Schulte, and Bierman [54, 55] is a proposal to significantly modify the core design of C♯ in order to integrate support for objects, relations, and XML (in particular, XML itself simply becomes a syntax for serialized object instances). XACT [47, 6] extends JAVA with XML processing, proposing an elegant programming idiom: the creation of XML values is done using XML templates, which are immutable first-class structures representing XML with named gaps that may be filled to obtain ordinary XML trees. XJ [29] is another extension of JAVA for native XML processing that uses W3C Schema as a type system and XPATH as a navigation language for XML. XOBE [46] is a source to source compiler for an extension of JAVA that, from language design point of view, is very similar to XTATIC. SCALA is a developing general-purpose web services language that compiles into JAVA bytecode; it is currently being extended with XML support [11].

So far, most of the above projects have concentrated on developing basic language designs; there is little published work on serious implementations. (Even for XQUERY and XSLT, we have been unable to find detailed descriptions of their run-time representations.) We summarize here the available information.

Considerable effort, briefly sketched in [4], has been put into making the CDUCE's OCAML-based interpreter efficient. They address similar issues of text and tree representations and use similar solutions. CDUCE's user-visible datatype for strings is also the character list, and they also

implement its optimized alternatives—the one described in the paper resembles our `SeqSubstring`. CDuce uses lazy list concatenation, but apparently only with eager normalization. Another difference is the object-oriented flavor of our representations.

Xact's implementation, developed independently and in parallel with Xtatic but driven by similar needs (supporting efficient sharing, etc.) and targeting a similar (object-oriented) runtime environment, has strong similarities to ours; in particular, lazy data structures are used to support efficient gap plugging. Our preliminary performance measurements may be viewed as validating the representation choices of both implementations. Xtatic's special treatment of `pcdata` (Section 7.2)does not appear to be used in Xact.

The current implementations of XOBE and XJ are based on DOM, although the designs are amenable to alternative back-ends.

Kay [44] describes the implementation of Version 6.1 of his XSLT processor Saxon. The processor is implemented in Java and, like in our approach, does not rely on a pre-existing Java DOM library for XML data representation, since DOM is again too heavyweight for the task at hand: e.g., it carries information unnecessary for XPath and XSLT (like entity nodes) and supports updates. Saxon comes with two variants of run time structures. One is object-oriented and is similar in spirit to ours. Another represents tree information as arrays of integers, creating node objects only on demand and destroying them after use. This model is reportedly more memory efficient and quicker to build, at the cost of slightly slower tree navigation. Overall, it appears to perform better and is provided as the default in Saxon.

In the broader context of functional language implementations, efficient support for list (and string) concatenation has long been recognized as an important issue. An early paper by Morris, Schmidt and Wadler [57] describes a technique similar to our eager normalization in their string processing language Poplar. Sleep and Holmström [63] propose a modification to a lazy evaluator that corresponds to our lazy normalization. Keller [45] suggests using a lazy representation without normalization at all, which behaves similarly to database B-trees, but without balancing. We are not aware of prior studies comparing the lazy and eager alternatives, as we have done here.

More recently, the algorithmic problem of efficient representation for lists with concatenation has been studied in detail by Kaplan, Tarjan and Okasaki [41, 42, 43, 60]. They describe *catenable steques* which support constant amortized time sequence operations. We opted for the simpler representations described here out of concern for excessive constant factors in running time arising from the complexity of their data structures (see Section 7.1.3.)

Another line of work, started by Hughes [38] and continued by Wadler [70] and more recently Voigtlander [66] considers how certain uses of list concatenation (and similar operations) in an

applicative program can be eliminated by a systematic program transformation, sometimes resulting in improved asymptotic running times. In particular, these techniques capture the well-known transformation from the quadratic to the linear version of the reverse function. It is not clear, however, whether the techniques are applicable outside the pure functional language setting: e.g., they transform a recursive function f that uses append to a function f' that uses only list construction, while in our setting problematic uses of append often occur inside imperative loops.

Prolog's difference lists [64] is a logic programming solution to constant time list concatenation. Using this technique requires transforming programs operating on regular lists into programs operating on difference lists. This is not always possible. Marriott and Søndergaard [53] introduce a dataflow analysis that determines whether such transformation is achievable and define the automatic transformation algorithm. We leave a more detailed comparison of our lazy concatenation approach and the difference list approach for future work.

Chapter 8

Conclusions and Future Work

This dissertation has described an efficient implementation of XTATIC—a mixture of the general purpose object-oriented language C♯ with constructs for type-safe processing of XML documents. We have introduced the framework of matching automata, developed compilers based on it, and demonstrated that they generate efficient low-level programs. Along the way, we have discussed an approach for enhancing matching automata resulting in simple and efficient disambiguation policy of pattern matching involving patterns with variable binding; an efficient type-based compiler optimization and a theoretic analysis of the corresponding optimization problem, and a C♯-based run-time system supporting fast and compact operations on XML sequences. We have shown that XTATIC's performance is quite competitive compared to the performance of existing mainstream XML processing frameworks.

The next step in the evolution of XTATIC concerns graduating from being a research project and moving toward being accepted in a wider community of programmers. For this to become reality, several criteria must be satisfied.

The first group of requirements deals with linguistic extensions that are necessary for XTATIC to become a *practical* general-purpose programming language.

One extension that is of immediate importance involves providing support for full C♯ including generics. This would necessitate a substantial redesign of XTATIC's compiler and run-time environment. A promising starting point with respect to generics is Hosoya, Frisch, and Castagna's recent proposal for adding polymorphism to XDUCE [32].

The extended language must have an efficient and robust implementation. For this, further improvement of the type-based optimization algorithm is essential. In particular, my goal is to refine the algorithm in order to achieve a more general optimality property. For memory efficiency,

XTATIC must support a streaming XML parsing model. The current approach used in XTATIC is not applicable for large data sets since it relies on keeping entire documents in main memory during processing. To solve this problem, we would like to re-engineer the XTATIC compiler by combining the non-backtracking compilation algorithm developed in the framework of matching automata with a streaming parser.

The next group of requirements concerns the aspects of language design that provide for a smoother integration between native C♯, XML-related constructs, and existing industry standards.

It is essential that there be a more intuitive correspondence between traditional C♯ data structures such as objects and arrays and XML values; XTATIC must provide data binding capabilities for a well-typed mapping between the two worlds and tools for specifying relationships between XML schemas and C♯ types. Other useful tools can help in defining complex XML types from scratch, importing them from outside sources, and inferring them from existing documents.

To achieve broader acceptance, XTATIC must be well-integrated with existing standards. This particularly involves bringing closer the XTATIC type system and XML schema standards and providing easy conversion mechanism between them. A tricky question is what to do with schema features—such as referential integrity constraints—that do not easily translate to types.

The last kind of extensions is geared toward enhancing the expressive power of XTATIC's pattern matching. There are two complimentary directions. The first one involves extending patterns to match not only XML data but also C♯ objects and their structure. The second introduces new pattern matching constructs with better iterative capabilities. There has been some initial investigation of iteration constructs that employ ambiguous patterns and collect all of their bindings [24]. Other approaches may exploit non-linear variables in patterns and new language-level mapping, filtering, and folding constructs. All these extensions present considerable challenges and design choices both to the type checker and to the compiler.

Bibliography

[1] A. W. Appel and T. Jim. Shrinking lambda Expressions in Linear Time. *Journal of Functional Programming*, 7(5):515–540, 1997.

[2] L. Augustsson. Compiling pattern matching. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381. Springer-Verlag, Berlin, DE, 1985.

[3] M. Baudinet and D. MacQueen. Tree pattern matching for ML. unpublished paper, 1985.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, pages 51–63, 2003.

[5] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree languages over non-ranked alphabets, 1998.

[6] A. S. Christensen, C. Kirkegaard, and A. Møller. A runtime system for XML transformations in Java. In Z. Bellahsène, T. Milo, and e. a. Michael Rys, editors, *Database and XML Technologies: International XML Database Symposium (XSym)*, volume 3186 of *Lecture Notes in Computer Science*, pages 143–157. Springer, Aug. 2004.

[7] J. Clark and M. Murata. RELAX NG. http://www.relaxng.org, 2001.

[8] H. Common, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Draft book; http://www.grappa.univ-lille3.fr/tata.

[9] I. DataPower Technology. XSLTMark. http://www.datapower.com/xml_community/xsltmark.html, 2001.

[10] S. Dawson, C. R. Ramakrishnan, S. Skiena, and T. Swift. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems*, 18(5):528–563, 1996.

[11] B. Emir. Extending pattern matching with regular tree expressions for XML processing in Scala. Diploma thesis, EPFL, Lausanne; http://lamp.epfl.ch/ buraq, 2003.

[12] F. L. Fessant and L. Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

[13] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of xpath queries. In *VLDB*, pages 153–164, 2003.

[14] A. Fokoue. Improving the performance of XPath query engines on large collections of XML data, 2002.

[15] X. Franc. Qizx. http://www.xfra.net/qizxopen, 2003.

[16] A. Frisch. Regular tree language recognition with static information. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2004.

[17] A. Frisch. *Théorie, conception et réalisation d'un langage adapté á XML*. PhD thesis, Ecole Normale Supérieure, Paris, Paris, France, 2004.

[18] A. Frisch and L. Cardelli. Greedy regular expression matching. In *31st International Colloquium on Automata, Languages and Programming (ICALP)*, July 2004.

[19] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2002.

[20] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. Xml goes native: Run-time representations for xtatic. http://www.cis.upenn.edu/ milevin/compiler.ps, Jan. 2004.

[21] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for Xtatic. In *14th International Conference on Compiler Construction*, Apr. 2005.

[22] 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.

[23] V. Gapeyev and B. C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

[24] V. Gapeyev and B. C. Pierce. Paths into patterns. Technical Report MS-CIS-04-25, University of Pennsylvania, Oct. 2004.

[25] J. Garrigue. Programming with polymorphic variants, Sept. 1998.

[26] P. Genevès and J.-Y. Vion-Dury. Logic-based XPath optimization. In *International ACM Symposium on Document Engineering*, 2004.

[27] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing xpath queries. In *VLDB*, pages 95–106, 2002.

[28] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space efficiency, 2003.

[29] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: facilitating XML processing in Java. In *International World Wide Web Conference*, pages 278–287, 2005.

[30] H. Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, Japan, 2000.

[31] H. Hosoya. Regular expression filters for XML. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004.

[32] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, 2005.

[33] 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. Preliminary version in PLAN-X 2002.

[34] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In D. Suciu and G. Vossen, editors, *International Workshop on the Web and Databases (WebDB)*, May 2000. Reprinted in *The Web and Databases, Selected Papers*, Springer LNCS volume 1997, 2001.

[35] H. Hosoya and B. C. Pierce. Regular expression pattern matching. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), London, England*, 2001. Full version in *Journal of Functional Programming*, 13(6), Nov. 2003, pp. 961–1004.

[36] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[37] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, Jan. 2005. Preliminary version in ICFP 2000.

[38] J. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, Mar. 1986.

[39] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Oct. 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.

[40] S. P. Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler Inliner. In *IDL*, 1999. revised version to appear in Journal of Functional Programming.

[41] H. Kaplan, C. Okasaki, and R. E. Tarjan. Simple confluently persistent catenable lists. *SIAM Journal on Computing*, 30(3):965–977, 2000.

[42] H. Kaplan and R. E. Tarjan. Persistent lists with catenation via recursive slow-down (preliminary version). In *ACM Symposium on Theory of Computing*, pages 93–102, 1995.

[43] H. Kaplan and R. E. Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM*, 46:577–603, 1999.

[44] M. H. Kay. Saxon: Anatomy of an xslt processor, Feb. 2001. http://www-106.ibm.com/developerworks/library/x-xslt2/.

[45] R. M. Keller. Divide and CONCer: Data structuring in applicative multiprocessing systems. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 196–202, 1980.

[46] M. Kempa and V. Linnemann. On XML objects. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2003.

[47] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, Mar. 2004.

[48] A. Krall. Implementation techniques for Prolog. In N. E. Fuchs, editor, *10. Workshop Logische Programmierung*, Bericht, pages 1–15, Zürich, 1994. Universität Zürich.

[49] V. Laurikari. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, 2001.

[50] M. Y. Levin. Compiling regular patterns. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, 2003.

[51] M. Y. Levin and B. C. Pierce. Regular pattern matching with binding. Technical Report MS-CIS-XX-XX, University of Pennsylvania, Aug. 2005.

[52] M. Y. Levin and B. C. Pierce. Type-based optimization for regular patterns. In *Database Programming Languages (DBPL)*, Aug. 2005.

[53] K. Marriott and H. Søndergaard. Difference-list transformation for prolog. *New Generation Computing*, 11:125–157, 1993.

[54] E. Meijer and W. Schulte. Unifying tables, objects and documents. In *Declarative Programming in the Context of OO Languages (DP-COOL)*, Sept. 2003.

[55] E. Meijer, W. Schulte, and G. Bierman. Programming with circles, triangles and rectangles. In *XML Conference and Exposition*, Dec. 2003.

[56] A. Møller and M. I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proc. Tenth International Conference on Database Theory, ICDT '05*, LNCS. Springer-Verlag, January 2005.

[57] J. H. Morris, E. Schmidt, and P. Wadler. Experience with an applicative string processing language. In *ACM Symposium on Principles of Programming Languages (POPL), Las Vegas, Nevada*, pages 32–46, 1980.

[58] M. Murata. Hedge Automata: a Formal Model for XML Schemata. Web page, 2000.

[59] A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In *Foundations of Software Technology and Theoretical Computer Science*, pages 134–145, 1998.

[60] C. Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *IEEE Symposium on Foundations of Computer Science*, pages 646–706, Oct. 1995.

[61] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, China, Aug. 2002. See also http://www.xml-benchmark.org/.

[62] P. Sestoft. ML pattern match compilation and partial evaluation. *Lecture Notes in Computer Science*, 1110:446–??, 1996.

[63] M. R. Sleep and S. Holmström. A short note concerning lazy reduction rules for append. *Software Practice and Experience*, 12(11):1082–4, Nov. 1982.

[64] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[65] S. Vansummeren. Type inference for unique pattern matching. *ACM Transactions on Programming Languages and Systems*, to appear.

[66] J. Voigtländer. Concatenate, reverse and map vanish for free. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Pittsburgh, Pennsylvania*, pages 14–25, 2002.

[67] W3C. Document object model (dom) technical reports, 1997–2003. http://www.w3c.org/DOM/DOMTR.

[68] W3C. XSL Transformations (XSLT), 1999. `http://www.w3.org/TR/xslt`.

[69] O. Waddell and R. K. Dybig. Fast and Effective Procedure Inlining. In *Static Analysis Symposium*, pages 35–52, 1997.

[70] P. Wadler. The concatenate vanishes. Note, University of Glasgow, 1987. (revised 1989).

[71] P. T. Wood. Minimising simple xpath expressions. In *WebDB*, pages 13–18, 2001.

[72] Extensible Markup Language (XML™), Feb. 1998. XML 1.0, W3C Recommendation, `http://www.w3.org/XML/`.

[73] XML Schema Part 1: Structures, W3C Recommendation, May 2001. http://www.w3c.org/TR/xmlschema-1/.

[74] XQuery 1.0: An XML Query Language, W3C Working Draft, Apr. 2005. http://www.w3.org/TR/xquery/.