Niels Bjørn Bugge Grathwohl

Parsing With Regular Expressions & Extensions to Kleene Algebra

PhD Dissertation

Department of Computer Science, University of Copenhagen
2015

Kleene
Meets
Church

Abstract

In the first part of this thesis, we investigate methods for *regular expression parsing*. The work is divided into three parts.

The first part is a two-pass algorithm for greedy regular expression parsing in a semi-streaming fashion. It executes in time $O(mn)$ for expressions of size m and input of size n. The first pass outputs a compact log of only k bits per input symbol, where k is the number of alternatives and Kleene stars in the expression. The second pass processes this log to produce a compact bit-coded parse tree, not requiring the input symbols at all. This is an improvement upon previous parsing algorithms that either do not scale linearly in the input or require more working memory for the log and use three passes instead of two.

The second part is an *optimally streaming* algorithm: bits of the bit-coded parse tree are output as soon as it is *semantically possible* to guarantee what the prefix of the final parse tree is. This can be done in time $O(2^{m \log m} + mn)$ for expressions of size m and input of size n. To be optimal, a PSPACE-complete preprocessing step is required: For a fixed regular expression, the algorithm's run-time scales linearly with the input length. For *inherently* non-streamable expressions like $\mathrm{a}^{\star}\mathrm{x} + \mathrm{a}^{\star}\mathrm{y}$ the algorithm degrades gracefully to a two-pass algorithm.

The third part is a determinization procedure for the optimally streaming algorithm where the preprocessing step has been omitted, along with a surface language, Kleenex, for expressing *general* non-deterministic finite transducers. We show how to compile Kleenex programs into deterministic streaming string transducers (SSTs), and demonstrate the performance of a Kleenex compiler that translates the SSTs to efficient C code. The resulting programs are essentially optimally streaming, run in worst-case linear time in the input size, and show consistent high performance in the 1 Gbps range on various use cases.

In the second part of this thesis, we present two extensions to Kleene algebra.

Chomsky algebra is an algebra with a structure similar to Kleene algebra, but with a generalized μ-operator for recursion instead of the Kleene star. We show that the axioms of idempotent semirings along with *continuity* of the μ-operator completely axiomatizes the equational theory of the context-free languages.

KAT + B! is an extension to Kleene algebra with tests (KAT) with mutable state. We describe a test algebra B! for mutable tests and give a commutative coproduct between KATs. The axioms of B! together with KAT and a set of commutativity conditions are shown to completely axiomatize the equational theory of an arbitrary KAT enriched with mutable state. The theory is EXPSPACE-complete. We give some examples of its use in showing the Böhm–Jacopini theorem and the folk theorem that all WHILE programs can be put into a normal form with only one WHILE loop.

Resumé

I første del af denne afhandling undersøger vi metoder til at *parse* med regulære udtryk. Dette er yderligere opdelt i tre dele.

Den første del er en to-fase-algoritme der laver semi-strømmende *grådig* parsing med regulære udtryk. Den kører i tid $O(mn)$ for udtryk af størrelse m og inddata af størrelse n. I den første fase bliver en kompakt log udskrevet, der kun består af k bits per indlæst karakter, hvor k er antallet af alternativer og Kleenestjerner i udtrykket. I den anden fase bliver denne log læst for at producere et kompakt bit-indkodet parsetræ, hvilket ikke kræver kendskab til den oprindelige inddata. Dette er en forbedring af hidtidige algoritmer til grådig parsing, da de enten ikke skalerer lineært i inddatas størrelse eller kræver mere arbejdslager til den midlertidige log-datastruktur og tre faser i stedet for to.

Den anden del er en *optimalt strømmende* algoritme: dele af det bit-indkodede parsetræ bliver udskrevet så snart det er *semantisk muligt* at garantere hvad præfixet af det endelige parsetræ er. Dette kan gøres i tid $O(2^{m \log m} + mn)$ for udtryk af størrelse m og inddata af størrelse n. For at være optimal kræves der en PSPACE-fuldkommen præberegning: For et specifikt regulært udtryk skalerer algoritmens køretid lineært i inddatas størrelse. Algoritmen falder trinvist tilbage på at være en to-fase-algoritme i tilfældet af *inherent* ikke-strømmende udtryk såsom $\mathrm{a}^\star \mathrm{x} + \mathrm{a}^\star \mathrm{y}$.

Den tredje del er en determiniseringsprocedure for den optimalt strømmende algoritme, hvor præberegningsskridtet er undladt, samt et overfladesprog, Kleenex, til at udtrykke *generelle* nondeterministiske endelige transducere. Vi viser hvordan Kleenexprogrammer kompileres ned til deterministiske strømmende strengtransducere (SSTer), og vi demonstrerer ydelsen af en Kleenexoversætter der oversætter SSTer til effektiv C-kode. De resultererende programmer er i praksis optimalt strømmende, kører i værste fald i lineær tid i inddatas størrelse, og holder et konsistent højt ydelsesniveau omkring 1 gigabit per sekund på forskellige eksempelprogrammer.

I anden del af denne afhandling præsenterer vi to udvidelser af Kleene algebra.

Chomsky algebra er en algebra med en struktur lig Kleene algebra, men med en generel μ-operator til rekursion i stedet for Kleenestjernen. Vi viser, at aksiomerne for en idempotent semiring samt *kontinuitet* af μ-operatoren aksiomatiserer lighedsteorien for kontekstfrie sprog fuldstændigt.

KAT + B! er en udvidelse af Kleene algebra med test (KAT) med muterbar tilstand. Vi beskriver en testalgebra B! for muterbare tests og giver et kommutativt koprodukt mellem KATer. Det bliver vist at B! og KATs aksiomer sammen med en mængde kommutativitetsbetingelser fuldstændigt aksiomatiserer lighedsteorien for en arbitrær KAT beriget med muterbar tilstand. Teorien er EXPSPACE-fuldstændig. Vi viser nogle eksempler af brugen af aksiomerne ved at vise Böhm–Jacopinis sætning samt den klassiske sætning der siger at alle WHILE-programmer kan omskrives til en normalform der kun har én WHILE-løkke.

Contents

List of Figures

This thesis was submitted for partial fulfillment of the PhD in Computer Science at the University of Copenhagen on August 1st 2015. It was succesfully defended on November 4th 2015. The assessment committee consisted of

- Nate Foster, Cornell University;
- Alexandra Silva, University College London;
- Andrzej Filinsky (chairman), DIKU, University of Copenhagen.

The assessment committee kindly pointed out some minor typos and error in the original manuscript. These have been fixed in this revised edition.

Acknowledgements

Being a PhD student is exciting for many reasons. One of them is that you get to do interesting and difficult things while interacting with brilliant people from all over the world.

In the course of my PhD studies, there are some people that I have interacted with especially much. All the people with whom I have co-authored papers deserve special thanks. Of course, my advisor Fritz Henglein, who is as sharp as he is humorous, has been a great influence. But especially my PhD-colleague Ulrik, with whom I have shared many travels, beers, and puzzlements at white boards, has been a great influence. He is both a good friend and researcher. The other PhD students at the APL group at DIKU, and everyone at DIKU in general, also deserve thanks for providing an always-friendly atmosphere at the department. Dexter Kozen, whom I visited at Cornell University in 2013, also deserves thanks for teaching me, both informally in his office and formally in a dedicated class at DIKU, about Kleene algebra.

Special thanks to my brother Hans for helping me with proof-reading of parts of this document, even though he was hiking in beautiful Norwegian mountains at the time.

And, most of all, thanks to the wonderful Christine for everything.

1 Introduction

This is a PhD thesis about *regular expression parsing* and *extensions to Kleene algebra*. These two seemingly disparate subjects have things in common, as the parsing techniques that we shall study are based on automata-theoretic methods, which can be reasoned about algebraically with Kleene algebra. However, the approaches to the two subjects are different. For parsing, we have a fairly practical approach—the end goal is to produce programs that can parse words under a regular expression as *fast* as possible. This is a more practical goal than the one in which the algebraic chapters are written. The goal here is not so much to produce *programs* that run, but more to study the abstract *properties* of classes of algebras.

The work presented in this thesis is based on papers that have been previously published and one paper that is under consideration for publishing. All papers are the result of work with excellent co-authors who have been humbling to work with.

- In the first part we describe three algorithms for parsing with regular expressions and a declarative language for specifying string transformations programs, a generalization of the parsing problem. We developed a compiler for the language, Kleenex, that translates Kleenex source to finite state transducers and uses our determinization algorithm to build *streaming string transducers*, a finite-state deterministic automaton with registers. The compiler produces efficient C code, as evidenced by the evaluation part of Chapter 7. Chapters in this part are based on two published papers and one paper under review [64–66].

- The second part of the thesis is about two extensions to Kleene algebra [63, 67].

  First, we describe a variant of Kleene algebra with a generalized recursion operator, μ. We call such algebras Chomsky algebras, and we show that the equational theory of the context-free languages is completely axiomatized by the axioms of *μ-continuous* Chomsky algebra.

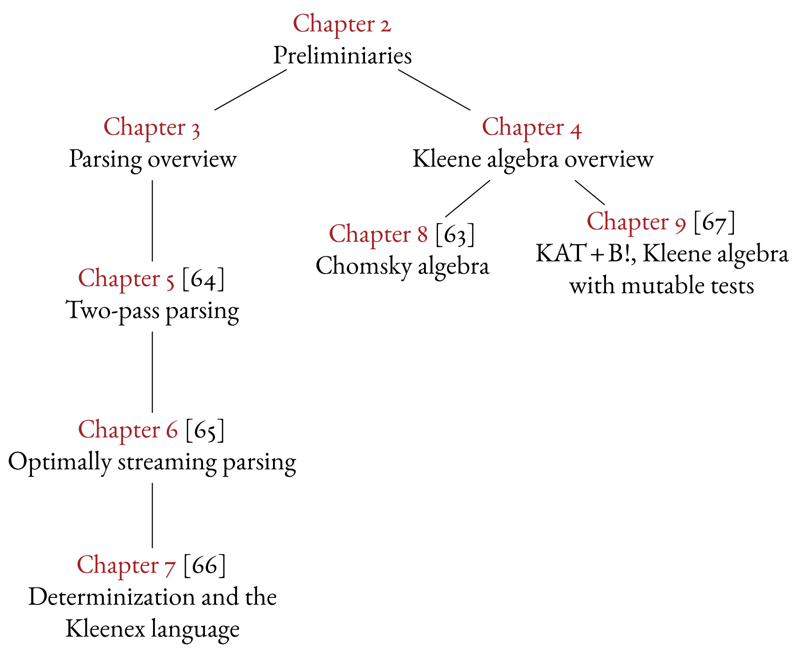
  Second, we present an extension to Kleene algebra with tests (KAT) that add mutable state. This is accomplished by defining an algebra for mutable tests that is also a KAT, and combining it with a different KAT using a commutative coproduct construction. The commutativity in this construction intuitively reflects the fact that only the test "setters" from the B! algebra may alter the state of variables.

There are thus two overarching routes through this document: the parsing related and the algebra related. The chapters regarding parsing techniques constitute a fairly straightforward narrative, where each chapter describes an advanced improvement upon the former. For the algebra chapters, this is not so. The two Kleene algebra-related systems that we introduce and discuss are not really related, and so the two chapters are insulated from each other. We have tried to alleviate this a bit by writing a common introduction that reviews some basic notions about Kleene algebra relevant to both chapters. The parsing chapters have also been equipped with an introduction that reviews some basic notions of automata theory as well as some important concepts about regular expression parsing. Both of these introduction chapters also provide an overview of the subsequent chapters.

Note that each "contribution chapter" is directly based on a corresponding paper. As a result, some parts of the chapters can look a bit like spurious restatements of things from previous chapters, which makes for a poor reading experience We have tried to minimize this effect by thorough editing, but it is probably still visible.

We start out with a very short preliminary chapter that introduces some concepts that are generally useful in the following chapters.

Overall, the structure of the thesis is the following:



2 Preliminaries

This chapter introduces some basic notions and definitions that will be of use in later chapters.

Definition 2.1. An *alphabet* Σ is a finite, non-empty set of characters.

Definition 2.2. A *language* L over some alphabet Σ is a set of words (strings) over Σ:

$$L \subseteq \Sigma^\star.$$

We write $w_1 \cdot w_2$ for the word $v = x_0 \dots x_n y_0 \dots y_m$ where $w_1 = x_0 \dots x_n$ and $w_2 = y_0 \dots y_m$. The word consisting of zero symbols is written ϵ. Often we omit the $\cdot$, writing wv for $w \cdot v$.

Definition 2.3. The *concatenation* of languages A and B is:

$$A \cdot B = \{w_1 \cdot w_2 \mid w_1 \in A, w_2 \in B\}.$$

Definition 2.4. The nth *exponentiation* of a language A is:

$$\begin{aligned} A^0 &= \{\epsilon\} \\ A^{n+1} &= A \cdot A^n \end{aligned}$$

Definition 2.5. The *Kleene asterate* of a language A is:

$$A^\star = \bigcup_{n=0}^{\infty} A^n.$$

Definition 2.6. The *disjoint union*, or *sum*, of sets A and B is:

$$A \oplus B = \{\mathsf{inl}\, x \mid x \in A\} \cup \{\mathsf{inr}\, x \mid x \in B\}.$$

Definition 2.7. The *Cartesian product* of sets A and B is:

$$A \times B = \{\langle x, y\rangle \mid x \in A, y \in B\}.$$

Definition 2.8 (Prefix). A word w is a *prefix* of w' if $w' = w \cdot w''$ for some w''. We write $w \sqsubseteq w'$ whenever w is a prefix of w'

Definition 2.9 (Longest common prefix). The *longest common prefix* of a set of words L, $\bigsqcap L$, is w such that:

$$\begin{aligned} w = \bigsqcap L \quad \text{iff} \quad & (\forall w' \in L.\, w \sqsubseteq w') \wedge \\ & (\forall w' \in L.\, \forall w'' \in L.\, w' \sqsubseteq w'' \implies w' \sqsubseteq w). \end{aligned}$$

The following property is useful:

Lemma 2.1. *For two languages A and B,*

$$A \subseteq B \implies \bigsqcap B \sqsubseteq \bigsqcap A.$$

Proof. For any word $w \in A$ we also have $w \in B$ so the longest common prefix of B must also be a prefix of the longest common prefix of A:

$$\left(\forall w' \in B. \bigsqcap B \sqsubseteq w'\right) \wedge \left(\forall w' \in B. \forall w'' \in B. w' \sqsubseteq w'' \implies w' \sqsubseteq \bigsqcap B\right),$$

so in particular

$$\left(\forall w' \in A. \bigsqcap B \sqsubseteq w'\right) \wedge \left(\forall w' \in A. \forall w'' \in A. w' \sqsubseteq w'' \implies w' \sqsubseteq \bigsqcap B\right).$$

Since $\bigsqcap A \in A$, we have $\bigsqcap B \sqsubseteq \bigsqcap A$. □

Definition 2.10 (Prefix-free language). A language S is *prefix-free* if no prefixes of any words in S are in S:

$$v \in S \implies \{v' \mid v' \sqsubseteq v\} \cap S = \emptyset.$$

Lemma 2.2. *Let X be a prefix-free language over an alphabet Σ with a lexicographical order on its elements and a minimum element. Then, for any $y \in \Sigma^\star$:*

$$(\min X)y = \min \{xy \mid x \in X\}.$$

Proof. The minimum element x' of X exists and is unique. To see

$$x'y = \min \{xy \mid x \in X\},$$

it suffices to observe that no element in X has x' as prefix, so all words in the set on the right-hand side are either $x'y$ or zy, for some $z \neq x'$:

$$x'y = \min (\{zy \mid z \in X, z \neq x'\} \cup \{x'y\}).$$

By definition, for all z, $x' \leq z$ in the lexicographical ordering, so also $x'y \leq zy$ too. The minimum element on the right-hand side are therefore $x'y$. □

Definition 2.11. An *algebra* is a pair $(A, \mathcal{F})$ where A is the *carrier set* and $\mathcal{F}$ is the *signature*, a list of functions $f_0^A, \ldots, f_k^A$ with different arities such that for each f^A

$$x_0, \ldots x_n \in A \implies f_i^A(x_0, \ldots, x_n) \in A, \quad \text{where } f_i^A \text{ is } n\text{-ary}.$$

Definition 2.12 (Homomorphism). Two algebras A and B have the same type if they have the same signature. A *homomorphism* is a mapping α between two algebras A and B with the same type such that

$$\alpha(f^A(x_0, \ldots, x_n)) = f^B(\alpha(x_0), \ldots, \alpha(x_n))$$

for each n-ary operator f^A and f^B in A and B.

Example 2.1. Let $\Sigma = \{\mathrm{a}, \mathrm{b}\}$ and let A and B be *monoids* (Definition 4.2), structures where:

- $A = (\Sigma^\star, \cdot, \epsilon)$ where for $v, w \in \Sigma^\star$, $v \cdot w \in \Sigma^\star$, and $\epsilon \cdot v = v \cdot \epsilon = v$.

- $B = (\mathbb{N}, +, 0)$ where for $i, j \in \mathbb{N}$, $i + j \in \mathbb{N}$, and $0 + i = i + 0 = v$.

Then f defined as below is a homomorphism from A to B:

$$\begin{aligned}
f(\epsilon) &= 0\\
f(\text{a}) &= 1\\
f(\text{b}) &= 2\\
f(v \cdot w) &= f(v) + f(w).
\end{aligned}$$

3 Regular Expression-Based Parsing Algorithms

In this chapter we give an overview of the work contained in this thesis that relates to *parsing techniques* for regular expressions and regular expression-like languages. Hence, the contents will be restatements of the points that are outlined in more detail in later chapters, but put into a context that makes it easier to follow the overall structure of the work as well as the insights that are gained in the process.

Parts of this chapter are based on contents from the manuscript *A Crash-Course in Regular Expression Parsing and Regular Expressions as Types*,[1] by Rasmussen, Henglein, and Grathwohl. It is unpublished, but has been used as teaching material both in a summer school course and in the supervision of several students at the University of Copenhagen. Other parts are based on the papers that form the basis for later chapters, or on a poster presented at POPL 2015 [64–66, 68].

3.1 Regular Expressions

The notion of regular expressions arose from the field of theoretical computer science, and was first described in 1956 by Stephen Cole Kleene [85] in his work on finite automata theory. The most common interpretation of regular expressions is as *patterns* denoting (possibly infinite) sets of strings. Under this interpretation, we can formulate the problem of determining whether a given string is contained in the underlying set specified by some regular expression. This generalizes the simpler string matching problem of determining whether a specific string occurs as a substring in a larger text. Concrete text search implementations emerged a relatively long while after Kleene's introduction, with one of the earliest appearances being Thompson's description of a text search algorithm based on regular expressions from 1968 [141]. The motivation for using regular expressions for simple text search is that they are computationally weak enough to provide strong guarantees of efficient running time and memory use, which is not the case for more expressive formalisms such as *context free grammars* [43].

Regular expressions are well-known among programmers and power-users, and have found use in many applications such as text editing, lexical analysis in compilers and scripting. Popular implementations include UNIX tools such as `sed`, `grep` and `awk`, text editors such as `emacs`, and some programming languages even include them as first-class constructions (Perl [30] being a notable example, although not the only one). What these implementations have in common is that they all implement the traditional regular expressions of Kleene, although some choose to add *extensions*, yielding a much more expressive (and

[1] http://diku.dk/kmc/documents/AiPL-CrashCourse.pdf

thus computationally much more complex) formalism. An example is Perl which adds *back-references* [73], making the matching problem NP-complete [125]! Additionally, the various implementations have slightly different conventions for the external syntax, mostly with regard to escaping. To avoid confusion when talking about regular expressions, we will therefore not talk about a specific implementation, but instead use a more mathematical notation.

Definition 3.1. We say that E is a *regular expression* (RE) if either:

1. E is an *atomic expression*, consisting of a single letter a from some set of letters Σ, or one of the special symbols 0 and 1 which do not occur in Σ.
2. Given REs E_1 and E_2, E is a *compound expression* of the form $E_1 + E_2$, $E_1 E_2$, or $E_1^\star$.[2]

The set Σ, called an *alphabet*, is assumed to be a finite set of "letters", which varies depending on our application. For example, for text search in a document encoded using ASCII [11], Σ will be defined as the set of printable characters in the ASCII table.

3.2 The Language Interpretation of Regular Expressions

An RE E can be interpreted as representing a set of strings, this is called the *language interpretation*. The language of E is denoted by $\mathcal{L}[\![E]\!]$. $\mathcal{L}[\![\cdot]\!]$ can simply be thought of as a function sending a regular expression to its interpretation as a set of strings. Atomic expressions denote either the language consisting of the single letter that makes up the expression, the language consisting of only the empty word, denoted ϵ, or the empty language that do not contain any words at all:

$$\begin{aligned} \mathcal{L}[\![\mathrm{a}]\!] &= \{\mathrm{a}\}\,, \\ \mathcal{L}[\![1]\!] &= \{\epsilon\}\,, \\ \mathcal{L}[\![0]\!] &= \emptyset. \end{aligned}$$

Thus, the RE a (for some $\mathrm{a} \in \Sigma$) represents a singleton set containing the one-letter string a, and the RE 1 represents the singleton set containing just the empty string. Note that this is *not* the same as the empty set, which is what the RE 0 denotes!

Compound expressions of the form $E_1 + E_2$ are defined in terms of set union:

$$\mathcal{L}[\![E_1 + E_2]\!] = \mathcal{L}[\![E_1]\!] \cup \mathcal{L}[\![E_2]\!].$$

The $+$ operator represents the merging of the languages of the expressions it combines.

Expressions of the form $E_1 E_2$ denote the language:

$$\mathcal{L}[\![E_1 E_2]\!] = \mathcal{L}[\![E_1]\!] \cdot \mathcal{L}[\![E_2]\!].$$

Thus, the interpretation of $E_1 E_2$ is the set of all words that can be formed by prepending all words from the language of E_1 to all the words from the language of E_2.

Given an RE of the form $E_1^\star$, we then define:

$$\mathcal{L}[\![E_1^\star]\!] = \mathcal{L}[\![E_1]\!]^\star$$

[2]These compound expressions are often written `E|F`, `EF`, `E*` in actual software implementations, where `E` and `F` are regular expressions.

Adding a * to an RE therefore corresponds to repetition.

For easy reference, we repeat here the full definition of the syntax and language interpretation of regular expressions:

Definition 3.2 (Regular expression syntax). Regular expressions over an alphabet Σ, Exp_Σ, are described by the grammar:

$$E ::= 0 \mid 1 \mid \mathrm{a} \mid E_1 + E_2 \mid E_1 E_2 \mid E_1^\star,$$

where $\mathrm{a} \in \Sigma$.

The juxtaposition and $+$ operator both associate to the right, so when parentheses are omitted it is a shorthand:

$$EFG = E(FG) \qquad\qquad E + F + G = E + (F + G).$$

Kleene star * binds the tightest, follows by juxtaposition, and finally $+$. The expression $E^\star FG + E' + F'^\star G'$ is parsed as $((E^\star)(FG)) + (E' + ((F'^\star)G'))$.

Definition 3.3 (Language interpretation of regular expressions). The *language interpretation* $\mathcal{L}[\![\cdot]\!]$ of a regular expression is:

$$\begin{aligned}
\mathcal{L}[\![\cdot]\!] &: \mathsf{Exp}_\Sigma \to 2^{\Sigma^\star} \\
\mathcal{L}[\![0]\!] &= \emptyset \\
\mathcal{L}[\![1]\!] &= \{\epsilon\} \\
\mathcal{L}[\![\mathrm{a}]\!] &= \{\mathrm{a}\} \\
\mathcal{L}[\![E_1 + E_2]\!] &= \mathcal{L}[\![E_1]\!] \cup \mathcal{L}[\![E_2]\!] \\
\mathcal{L}[\![E_1 E_2]\!] &= \mathcal{L}[\![E_1]\!] \cdot \mathcal{L}[\![E_2]\!] \\
\mathcal{L}[\![E^\star]\!] &= \mathcal{L}[\![E]\!]^\star
\end{aligned}$$

The type of problems one can solve with the language interpretation is *matching*. A programmer can pose the question "does the string w exist in the language $\mathcal{L}[\![E]\!]$," and it can be interpreted as the mathematical query $w \in \mathcal{L}[\![E]\!]$?

Example 3.1. The following regular expression denotes words formed by arbitrary combinations of a and b:

$$E_{ab} = (\mathrm{a} + \mathrm{b} + \mathrm{a})^\star.$$

We can verify this by unfolding the definition of the language interpretation:

$$\begin{aligned}
\mathcal{L}[\![E_{ab}]\!] &= \mathcal{L}[\![(\mathrm{a} + \mathrm{b} + \mathrm{a})^\star]\!] \\
&= \mathcal{L}[\![\mathrm{a} + \mathrm{b} + \mathrm{a}]\!]^\star \\
&= (\mathcal{L}[\![\mathrm{a}]\!] \cup \mathcal{L}[\![\mathrm{b}]\!] \cup \mathcal{L}[\![\mathrm{a}]\!])^\star \\
&= (\{\mathrm{a}, \mathrm{b}\})^\star \\
&= \{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{ba}, \mathrm{bb}, \mathrm{aaa}, \ldots\}.
\end{aligned}$$

Example 3.2. Due to the transcription between the Arabic and the Latin alphabets, there are many correct ways to spell former Libyan dictator Muammar Gadaffi's in English orthography. The following RE encodes all the possible ways to write the name:

$$E_g = (\mathrm{Q} + \mathrm{G} + \mathrm{K})(\mathrm{a} + \mathrm{u})(\mathrm{d} + \mathrm{t})(\mathrm{h} + 1)(\mathrm{d} + \mathrm{t})(\mathrm{h} + 1)\mathrm{af}(\mathrm{i} + \mathrm{y}).$$

With the language interpretation we can check that Gaddafi $\in \mathcal{L}[\![E_g]\!]$ and Qutthafy $\in \mathcal{L}[\![E_g]\!]$, whereas gardafi $\notin \mathcal{L}[\![E_g]\!]$.

3.3 Non-Deterministic Finite Automata

Regular expressions correspond to a class of abstract machines called *finite automata*. These exist in both deterministic and non-deterministic variants; here we briefly review the non-deterministic kind.

Definition 3.4 (Non-deterministic finite automaton). A *non-deterministic finite automaton* (NFA) is a five-tuple

$$(Q, \Sigma, I, F, \Delta)$$

consisting of

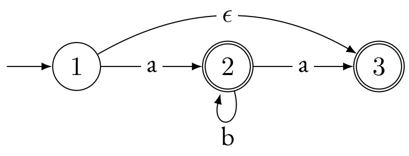
- a set of *states* Q,
- an *input alphabet* Σ,
- a set of *initial states* $I \subseteq Q$,
- a set of *final states* $F \subseteq Q$,
- a *transition relation* $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

We normally depict states as circles, the elements in the transition relation (q, a, q') as arrows, initial states with an incoming arrow, and final states with a double edge. We also write $q \xrightarrow{\mathrm{a}} q'$ and $q \xrightarrow{\epsilon} q'$ if $(q, \mathrm{a}, q') \in \Delta$ or $(q, \epsilon, q') \in \Delta$, respectively. Whenever we write $q \xrightarrow{x} q'$, x is called the *label* of the transition.

Example 3.3. The automaton

$$(\{1, 2, 3\}, \{\mathrm{a}, \mathrm{b}\}, \{1\}, \{2, 3\}, \{(1, \mathrm{a}, 2), (1, \epsilon, 3), (2, \mathrm{b}, 2), (2, \mathrm{a}, 3)\}) :$$

can be depicted graphically as:



Several methods for constructing finite automata from a regular expressions [59, 106, 141] exist, but we shall discuss *Thompson's algorithm* [141] here. Thompson's algorithm works by constructing the NFA *bottom-up*: Atomic expressions are converted first, and compound expressions are converted by combining the automata resulting from converting sub-expressions. The algorithm yields automata that has a specific form:

- the set of initial states is always a singleton set: $I = \{q^{\mathsf{in}}\}$,
- the set of final states is always a singleton set: $F = \{q^{\mathsf{fin}}\}$,
- all states either have exactly one outgoing transition in $Q \times \Sigma \times Q$, or either one or two transitions in $Q \times \{\epsilon\} \times Q$, or no outgoing transitions at all.

For this reason, we will define a specialized version of NFAs called *Thompson NFAs*:

Definition 3.5 (Thompson NFA [141]). A *Thompson NFA* is a five-tuple

$$(Q, \Sigma, q^{\mathsf{in}}, q^{\mathsf{fin}}, \Delta)$$

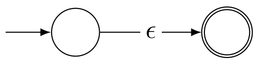
consisting of:

- a set of *states* Q,
- an *input alphabet* Σ,
- an *initial state* $q^{\mathsf{in}} \in Q$,
- a *final state* $q^{\mathsf{fin}} \in Q$,
- a *transition relation* $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

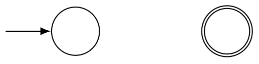
A Thompson NFA is constructed according to the rules specified below.

For any regular expression E, the conversion proceeds inductively on the syntactic structure of E. When an NFA is constructed from an expression E we call it $\mathcal{N}_E$. The base cases for atomic expressions are simple:

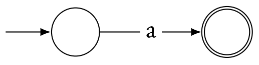
- Case $E = 1$:



- Case $E = 0$:

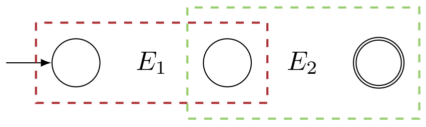


- Case $E = \mathrm{a}$:



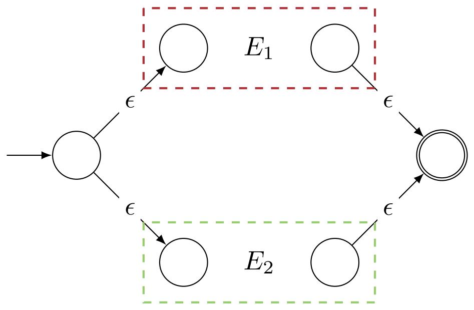
For compound expressions, we use boxes to denote the automata resulting from converting sub-expressions.

- Case $E = E_1E_2$:

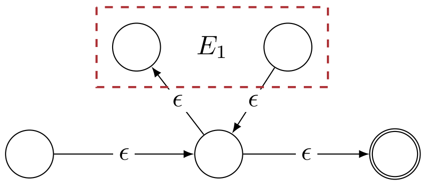


  That is, we merge the final state of the NFA for E_1 with the initial state for E_2, and let the initial state in E_1 be our new initial state, and the final state in E_2 be our new final state.

- Case $E = E_1 + E_2$:



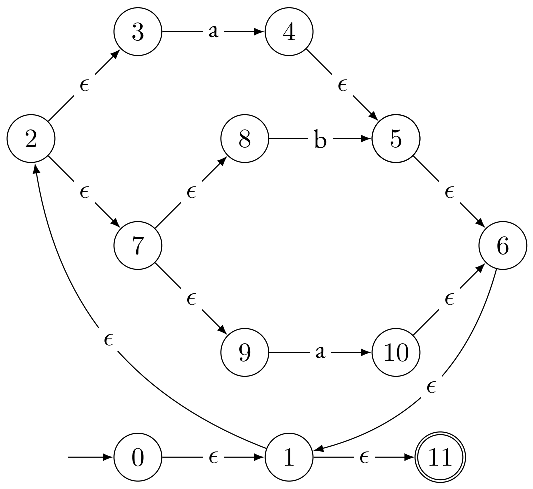
- Case $E = E_1^\star$:



Example 3.4. Constructing the Thompson NFA for the expression E_{ab}

$$E_{ab} = (\mathrm{a} + \mathrm{b} + \mathrm{a})^\star$$

from before yields the automaton $\mathcal{N}_{E_{ab}}$:



3.3.1 Simulation

An automaton specifies how an internal state of some "machine" that recognizes a language should change for each input symbol in an input word in order to halt in the accepting state for all words in the language and not for any other words. As NFAs are non-deterministic, there may be several possibilities for state change at each input symbol. By *simulating* the NFA we handle this by taking all possible transitions at the same time. In order to describe the simulation algorithm, we first need a few definitions.

Definition 3.6 (Paths). A *path* in an NFA $(Q, \Sigma, I, F, \Delta)$ is a non-empty list of states

$$q_0, q_1, \dots, q_n$$

where for each $0 \leq i < n$ and $x_i \in \Sigma \cup \{\epsilon\}$

$$q_i \xrightarrow{x_i} q_{i+1}.$$

If there is a path between q_0 and q_n with the sequence of labels $x_0, x_1, \dots, x_n$ we write

$$q_0 \overset{w}{\rightsquigarrow} q_n$$

where $w = x_0 \cdot x_1 \cdot \dots \cdot x_n$.

Definition 3.7. The language recognized by a Thompson NFA is the set of strings such that a path from the initial state to the final state exists:

$$\left\{ w \mid q^{\mathsf{in}} \overset{w}{\rightsquigarrow} q^{\mathsf{fin}} \right\}.$$

Definition 3.8 (ϵ-closure). Given an NFA $(Q, \Sigma, I, F, \Delta)$, the ϵ*-closure* of a set of states $S \in Q$ is:

$$\begin{aligned} \mathsf{Close} \colon 2^Q &\to 2^Q \\ \mathsf{Close}(S) &= \left\{ q'' \mid q' \in S, q' \overset{\epsilon}{\rightsquigarrow} q'' \right\} \end{aligned}$$

The ϵ-closure of a set of states is the set of all states reachable by traversing only edges labelled with ϵ.

Example 3.5. Recall the NFA from Example 3.4.

$$\begin{aligned} \mathsf{Close}(\{0\}) &= \{0, 1, 2, 3, 7, 8, 9, 11\}, \\ \mathsf{Close}(\{4, 10\}) &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}, \\ \mathsf{Close}(\{3\}) &= \{3\}. \end{aligned}$$

Definition 3.9. Given an NFA $(Q, \Sigma, I, F, \Delta)$ and an input symbol $\mathrm{a} \in \Sigma$, the *stepping* function takes a set of states $S \subseteq Q$ to a new set of states by following transitions labeled a:

$$\begin{aligned} \mathsf{Step} \colon 2^Q \times \Sigma &\to 2^Q \\ \mathsf{Step}(S, \mathrm{a}) &= \left\{ q' \mid q \in S, q \xrightarrow{\mathrm{a}} q' \right\} \end{aligned}$$

Example 3.6. Continuing Example 3.5:

$$\begin{aligned} \mathsf{Step}(\{0\}, \mathrm{a}) &= \emptyset, \\ \mathsf{Step}(\{0, 1, 2, 3, 7, 8, 9, 11\}, \mathrm{a}) &= \{4, 10\}, \\ \mathsf{Step}(\{0, 1, 2, 3, 7, 8, 9, 11\}, \mathrm{b}) &= \{5\}, \\ \mathsf{Step}(\{3, 8, 9\}, \mathrm{a}) &= \{4, 10\}, \\ \mathsf{Step}(\{3, 8, 9\}, \mathrm{b}) &= \{5\}. \end{aligned}$$

The two functions Step and Close provide us with the parts required to implement NFA simulation. Simulating an NFA is now just a question about executing these two functions interchangeably on an input symbol and checking if a final state is in the resulting state set.

Definition 3.10 (NFA simulation). Given an NFA $(Q, \Sigma, I, F, \Delta)$ and an input word $w \in \Sigma^\star$, the function Reach^* computes the set of states reachable from $S \subseteq Q$ by alternatively applying Close and Step on each input symbol:

$$\begin{aligned}
\mathsf{Reach}^* &: 2^Q \times \Sigma^\star \to 2^Q \\
\mathsf{Reach}^*(S, \epsilon) &= \mathsf{Close}(S) \\
\mathsf{Reach}^*(S, \mathrm{a} \cdot w') &= \mathsf{Reach}^*(\mathsf{Step}(\mathsf{Close}(S), \mathrm{a}), w').
\end{aligned}$$

With this function, deciding whether a word w is in the language of some regular expression is done simply by checking $F \cap \mathsf{Reach}^*(I, w) \neq \emptyset$.

Example 3.7. Consider the NFA $\mathcal{N}_{E_{ab}}$ of Example 3.4 and the input word aab:

$$\begin{aligned}
\mathsf{Reach}^*(\{0\}, \mathrm{aab}) &= \mathsf{Reach}^*(\mathsf{Step}(\mathsf{Close}(\{0\}), \mathrm{a}), \mathrm{ab}) \\
&= \mathsf{Reach}^*(\mathsf{Step}(\{0, 1, 2, 3, 7, 8, 9, 11\}, \mathrm{a}), \mathrm{ab}) \\
&= \mathsf{Reach}^*(\{4, 10\}, \mathrm{ab}) \\
&= \mathsf{Reach}^*(\mathsf{Step}(\mathsf{Close}(\{4, 10\}), \mathrm{a}), \mathrm{b}) \\
&= \mathsf{Reach}^*(\mathsf{Step}(\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}, \mathrm{a}), \mathrm{b}) \\
&= \mathsf{Reach}^*(\{4, 10\}, \mathrm{b}) \\
&= \mathsf{Reach}^*(\mathsf{Step}(\mathsf{Close}(\{4, 10\}), \mathrm{b}), \epsilon) \\
&= \mathsf{Reach}^*(\mathsf{Step}(\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}, \mathrm{b}), \epsilon) \\
&= \mathsf{Reach}^*(\{5\}, \epsilon) \\
&= \mathsf{Close}\{5\} \\
&= \{1, 2, 3, 5, 6, 7, 8, 9, 11\}.
\end{aligned}$$

As the final state 11 is indeed contained in the set, we conclude that aab $\in \mathcal{L}[\![E_{ab}]\!]$.

3.4 Deterministic Finite Automata

If a finite automaton is deterministic, we call it a *deterministic finite automaton*:

Definition 3.11 (Deterministic finite automaton). A *deterministic finite automaton* (DFA) is a five-tupe:

$$(Q, \Sigma, q^{\mathsf{in}}, F, \delta)$$

consisting of:

- a set of *states* Q,
- an *input alphabet* Σ,
- an *initial state* $q^{\mathsf{in}} \in Q$,
- a set of *final states* $F \subseteq Q$,
- a *transition function* $\delta : Q \times \Sigma \to Q$.

Thus, in a DFA there are no ϵ-transitions, and the transitions are encoded as a function instead of as a relation. DFAs and NFAs are equivalent: a DFA is just a special case of an NFA, and given an NFA one can obtain an equivalent DFA by using the *subset construction*. The subset construction works by memoizing the algorithm from Definition 3.10 and recording each encountered set of states as a new state in the DFA, adding transitions corresponding to the behavior of Step.

3.5 The Type Interpretation of Regular Expressions

The language interpretation of REs allow programmers to express queries related to language membership, i.e., the programmer receives one bit of information. Thus, the programmer does not speak of *how* but of *if* – the syntactic structure of the regular expression is forgotten. It turns out that many practical uses of regular expressions actually require that some degree of structure is preserved. Consider for example the scenario where we want to rewrite numerals from having the English digit grouping style to the Scandinavian digit grouping style:

$$\text{“1,234,567.89”} \quad \rightarrow \quad \text{“1.234.567,89”}.$$

Assuming given a regular expression E_{digit} for matching digits, we can easily formulate expressions for these two styles:

$$\begin{aligned} E_{numS} &= ((E_{digit}E^{\star}_{digit})\,.)^{\star}(E_{digit}E^{\star}_{digit})\,,(E_{digit}E^{\star}_{digit}) \\ E_{numE} &= ((E_{digit}E^{\star}_{digit})\,,)^{\star}(E_{digit}E^{\star}_{digit})\,.\,(E_{digit}E^{\star}_{digit}) \end{aligned}$$

These two expressions reflect the structure that we are interested in; if we apply a matching algorithm with the above inputs and expressions, however, this information is lost. One expression will report a match on the first string and fail on the other and vice versa. If instead the entire structure of the regular expressions was maintained, it would be easy to describe the transformation. If one thinks of the “meaning” of each regular expression constructor, the intuition behind this is clear:

- Concatenation expresses sequence, i.e., that something comes before something else.
- Alternation expresses a choice between the left hand side and the right hand side of the $+$.
- Kleene star expresses any finite number of repetitions of something.

The primitive symbols represent just themselves, and the special expressions 0 and 1 represent failure and unit in a manner that will be specified later.

This is called the *type interpretation* of regular expressions [57, 75, 114]. Informally, a type can be thought of as an entity representing the “shape” of a regular expression. One expression has one shape, so each regular expression has exactly one type. We shall write the types in the same way we write the regular expressions themselves; this reflects the fact that the only changed thing is how the expressions are interpreted. Instead of denoting languages, regular expressions under the type interpretation denote set of *structured values*:

Definition 3.12. The *structured values* over Σ, Val_{Σ}, are formed over the following grammar:

$$v ::= () \mid \mathsf{a} \mid \langle v_1, v_2 \rangle \mid \mathsf{inl}\, v_1 \mid \mathsf{inr}\, v_1 \mid [v_0, \ldots, v_n]$$

where $\mathsf{a} \in \Sigma$.

To each regular expression we associate a set of structured values. This is called the *type interpretation*:

Definition 3.13 (Type interpretation of regular expressions). The type interpretation associates to each regular expression E a set of structured values $\mathcal{V}[\![E]\!]$:

$$
\begin{aligned}
\mathcal{V}[\![\cdot]\!] &: \mathsf{Exp}_\Sigma \to 2^{\mathsf{Val}_\Sigma} \\
\mathcal{V}[\![0]\!] &= \emptyset \\
\mathcal{V}[\![1]\!] &= \{()\} \\
\mathcal{V}[\![\mathrm{a}]\!] &= \{\mathrm{a}\} \\
\mathcal{V}[\![E_1 E_2]\!] &= \mathcal{V}[\![E_1]\!] \times \mathcal{V}[\![E_2]\!] \\
\mathcal{V}[\![E_1 + E_2]\!] &= \mathcal{V}[\![E_1]\!] \oplus \mathcal{V}[\![E_2]\!] \\
\mathcal{V}[\![E_1^\star]\!] &= \{[v_1, \ldots, v_n] \mid v_i \in \mathcal{V}[\![E_1]\!], n \in \mathbb{N}\}\,.
\end{aligned}
$$

The above definition is similar to Definition 3.3 except that the structure of the compound expressions is reflected in the structured value. As we shall see, changing the interpretation of regular expressions to this lets a programmer ask *how* a string matches an expression instead of *if* a string matches an expression.

3.5.1 Handling Lists

At first glance it seems like we have three different ways of combining types to form new types: $+$, $\cdot$, and $[\cdot]$. However, a list is defined to be either an empty list, written $()$, or an element combined with the remainder of the list (the head and the tail of the list). Consequently, we can unfold the list type:

$$E^\star = EE^\star + 1, \tag{3.1}$$

that is, either a designated end symbol of type 1 or an element v of type E along with the rest of the list of type $E^\star$. This is equivalent to the way lists are defined in functional programming languages normally, and it is therefore enough to only define sum and product types. However, for readability reasons we will keep the "direct" notation for lists.

In Part II we will see equation 3.1 again in an algebraical setting.

Example 3.8. The type $E_a = \mathrm{a}^\star$ denotes the set of lists of as. The element of $\mathcal{V}[\![E_a]\!]$ with three repetitions can be written:

$$[a, a, a] = \mathsf{inl}\,\langle a, \mathsf{inl}\,\langle a, \mathsf{inl}\,\langle a, \mathsf{inr}\,()\rangle\rangle\rangle$$

3.5.2 Value Flattening

Note that we do not lose anything by moving from the language interpretation to the type interpretation. It is always possible to recover the underlying language from a set of structured values by erasing the structural information – called the *flattening*

Definition 3.14. The *flattening* of a structured value v is denoted $|\cdot|$ and is defined as:

$$\begin{aligned}
|\cdot| &: \mathsf{Val}_\Sigma \to \Sigma^\star \\
|()| &= \epsilon \\
|a| &= \mathsf{a} \\
|\langle v_1, v_2\rangle| &= |v_1| \cdot |v_2| \\
|\mathsf{inl}\, v_1| &= |v_1| \\
|\mathsf{inr}\, v_1| &= |v_1| \\
|[v_0, \dots, v_n]| &= |v_0| \cdot \dots \cdot |v_n| .
\end{aligned}$$

We lift the flattening function $|\cdot|$ to operate on sets of structured values. It is easily seen by induction on the structure of E that the language of E is equivalent to the set of flattened values of E. This is shown in Proposition 5.1:

Proposition. *For any regular expression E,*

$$\mathcal{L}[\![E]\!] = |\mathcal{V}[\![E]\!]| = \{|v| \mid v \in \mathcal{V}[\![E]\!]\} .$$

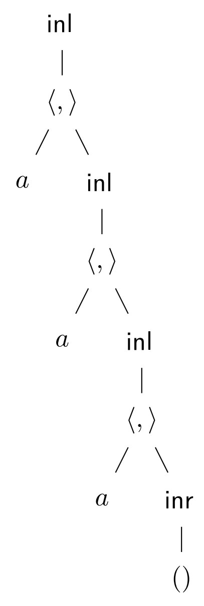
3.5.3 Structured Values are Parse Trees

One can think of the structured values as *parse trees*, which is also the reason for the term "flattening." Anything that does not take any arguments is a leaf in the tree, i.e., the primitive objects, and anything that takes $n > 0$ arguments is a node with n children.

Example 3.9. The value

$$[a, a, a] = \mathsf{inl}\,\langle a, \mathsf{inl}\,\langle a, \mathsf{inl}\,\langle a, \mathsf{inr}\,()\rangle\rangle\rangle$$

from before can be written as the following parse tree:



One obtains the flattening of the value by concatenating all the non-() leaves to get aaa.

We will use the terms *structured value*, *value*, *parse*, and *parse tree* interchangeably.

Example 3.10. Continuing Example 3.1, consider again the regular expression

$$E_{ab} = (\mathrm{a} + \mathrm{b} + \mathrm{a})^\star.$$

The structured values in $\mathcal{V}[\![E_{ab}]\!]$ encode the different ways one can combine a and b in the pattern E_{ab}:

$$\begin{aligned}
\mathcal{V}[\![E_{ab}]\!] &= \mathcal{V}[\![(\mathrm{a}+\mathrm{b}+\mathrm{a})^\star]\!] \\
&= \{[t_0,\dots,t_n] \mid t_i \in \mathcal{V}[\![\mathrm{a}+\mathrm{b}+\mathrm{a}]\!]\} \\
&= \left\{ [t_0,\dots,t_n] \mid t_i \in \left\{ \mathsf{inl}\, v_0, \mathsf{inr}\,\mathsf{inl}\, v_1, \mathsf{inr}\,\mathsf{inr}\, v_2 \;\middle|\; \begin{array}{r} v_0 \in \mathcal{V}[\![\mathrm{a}]\!], \\ v_1 \in \mathcal{V}[\![\mathrm{b}]\!], \\ v_2 \in \mathcal{V}[\![\mathrm{a}]\!] \end{array} \right\} \right\} \\
&= \left\{ [t_0,\dots,t_n] \mid t_i \in \left\{ \mathsf{inl}\, v_0, \mathsf{inr}\,\mathsf{inl}\, v_1, \mathsf{inr}\,\mathsf{inr}\, v_2 \;\middle|\; \begin{array}{r} v_0 \in \{a\}, \\ v_1 \in \{b\}, \\ v_2 \in \{a\} \end{array} \right\} \right\} \\
&= \{[t_0,\dots,t_n] \mid t_i \in \{\mathsf{inl}\, a, \mathsf{inr}\,\mathsf{inl}\, b, \mathsf{inr}\,\mathsf{inr}\, a\}\}\,.
\end{aligned}$$

By flattening the values we obtain the language of Example 3.1.

3.5.4 Bit-Coding Structured Values

Notice that in the structured values, some of the information is duplicated from the regular expression itself. For example, the expression a+b has as one of its values inl a, but the a itself is redundant; the only new information encoded in the value is that the *left* alternative was picked. Sometimes the values do not contain any new information at all. The expression ab has as its only possible value $\langle \mathrm{a}, \mathrm{b}\rangle$, but as it is the *only* possibility it contains zero bits of new information.

This observation can be used to produce a compact representation of values. As we saw above, lists can be thought of as repeated sums and products, and products do not introduce any choice to be encoded, so the only constructors that need to be encoded are $\mathsf{inl}\cdot$ and $\mathsf{inr}\cdot$. Thus, for each occurrence of a $+$ in a regular expression we need one bit to encode the choice, and for each Kleene star we need one bit for each time the inner expression is repeated, plus one bit to signal the end of the list—corresponding to picking the right alternative 1.

Definition 3.15 (Bit-coding of values). The *bit-coding* of a structured value is:

$$\begin{aligned}
\ulcorner\cdot\urcorner &: \mathsf{Val}_\Sigma \to \{\mathsf{0},\mathsf{1}\}^\star \\
\ulcorner()\urcorner &= \epsilon \\
\ulcorner a\urcorner &= \epsilon \\
\ulcorner\langle v_1, v_2\rangle\urcorner &= \ulcorner v_1\urcorner \cdot \ulcorner v_2\urcorner \\
\ulcorner\mathsf{inl}\, v_1\urcorner &= \mathsf{0}\cdot\ulcorner v_1\urcorner \\
\ulcorner\mathsf{inr}\, v_1\urcorner &= \mathsf{1}\cdot\ulcorner v_1\urcorner \\
\ulcorner[v_0,\dots,v_n]\urcorner &= \mathsf{0}\cdot\ulcorner v_0\urcorner\cdot\ldots\cdot\mathsf{0}\cdot\ulcorner v_n\urcorner\cdot\mathsf{1}.
\end{aligned}$$

As with the structured values, there may be bit-codes that do not make sense with respect to a specific regular expression—take $\mathsf{001}$ and the expression a, for example. Bit-codes that do make sense with a regular expression E can be decoded to obtain the full value:

Definition 3.16 (Type-directed decoding [75]). The *type-directed decoding* of a bit-code is (where $\mathcal{B}[\![E]\!]$ is given by Definition 3.18):

$$\begin{aligned}\llcorner \cdot \lrcorner_E &: \mathcal{B}[\![E]\!] \to \mathcal{V}[\![E]\!]\\ \llcorner bs \lrcorner_E &= \text{LET } (v, bs') = D_E(bs) \text{ IN IF } bs' = \epsilon \text{ THEN } v \text{ ELSE } \bot\end{aligned}$$

where:

$$\begin{aligned}D_1(bs) &= ((), bs)\\ D_a(bs) &= (\mathtt{a}, bs)\\ D_{E+F}(\mathtt{0} \cdot bs) &= \text{LET } (v, b') = D_E(bs) \text{ IN } (\mathsf{inl}\, v, b')\\ D_{E+F}(\mathtt{1} \cdot bs) &= \text{LET } (v, b') = D_E(bs) \text{ IN } (\mathsf{inr}\, v, b')\\ D_{EF}(bs) &= \text{LET } (v, b') = D_E(bs)\\ &\phantom{= \text{LET }} (w, b'') = D_F(b') \text{ IN } (\langle v, w\rangle, b'')\\ D_{E^\star}(bs) &= D_{EE^\star+1}(bs).\end{aligned}$$

Well-formed bit-codes with respect to a regular expression correspond to well-typed values for that expression.

Definition 3.17. The *well-formed bit-codes* for a regular expression E are those that correspond to structured values in $\mathcal{V}[\![E]\!]$:

$$\{\ulcorner v \urcorner \mid v \in \mathcal{V}[\![E]\!]\}\,.$$

Alternatively, we can define the well-formed bit-codes directly, in the same manner as we did with the values:

Definition 3.18. A regular expression E has an associated set of *bit-coded values*:

$$\begin{aligned}\mathcal{B}[\![\cdot]\!] &: \mathsf{Exp}_\Sigma \to 2^{\{0,1\}^\star}\\ \mathcal{B}[\![0]\!] &= \emptyset\\ \mathcal{B}[\![1]\!] &= \{\epsilon\}\\ \mathcal{B}[\![E_1 + E_2]\!] &= \{\mathtt{0} \cdot v \mid v \in \mathcal{B}[\![E_1]\!]\} \cup \{\mathtt{1} \cdot v \mid v \in \mathcal{B}[\![E_2]\!]\}\\ \mathcal{B}[\![E_1 E_2]\!] &= \{v \cdot w \mid v \in \mathcal{B}[\![E_1]\!], w \in \mathcal{B}[\![E_2]\!]\}\\ \mathcal{B}[\![E_1^\star]\!] &= \left\{v \cdot \mathtt{1} \mid v \in \{\mathtt{0} \cdot w \mid w \in \mathcal{B}[\![E_1]\!]\}^\star\right\}\end{aligned}$$

It can be shown by straight-forward induction on the structure of the regular expressions that the two definitions of bit-codes are equivalent:

Proposition 3.1. *Definitions 3.15 and 3.18 are equivalent:*

$$\{\ulcorner v \urcorner \mid v \in \mathcal{V}[\![E]\!]\} = \mathcal{B}[\![E]\!].$$

Definition 3.19. The set of bit-codes for values in $\mathcal{V}[\![E]\!]$ that flatten to the same word w is:

$$\mathcal{B}_w[\![E]\!] = \{\ulcorner t \urcorner \mid t \in \mathcal{V}[\![E]\!], |t| = w\}\,.$$

Using the bit-coded versions of values can be thought of as splitting the information contained in a "full" value into the static and the dynamic part. The static part is known from the regular expression itself, and the dynamic part is what is unique for this particular value. A consequence of this is that the bit-codes by themselves are useless—if we do not know which regular expression a bit-code is associated with, we cannot say anything about it.

Example 3.11. To illustrate why both the bit-string and the RE is needed, consider the bit-string 0001 again. Given RE $E_1 = (\mathrm{b}+\mathrm{a})(\mathrm{a}+\mathrm{b})(\mathrm{b}+\mathrm{a})(\mathrm{a}+\mathrm{b})$ this bit-code decodes to the value:

$$\langle \mathsf{inl}\ b, \langle \mathsf{inl}\ a, \langle \mathsf{inl}\ b, \mathsf{inr}\ b\rangle\rangle\rangle,$$

but given the RE $E_2 = a^\star$ the value is

$$[\mathrm{a},\mathrm{a},\mathrm{a}] = \mathsf{inl}\ \langle \mathrm{a}, \mathsf{inl}\ \langle \mathrm{a}, \mathsf{inl}\ \langle \mathrm{a}, ()\rangle\rangle\rangle$$

3.5.5 Ambiguity in Regular Expressions

In Example 3.1, the language interpretation of $(\mathrm{a}+\mathrm{b}+\mathrm{c})^\star$, it does not matter that there are two possibilities for matching an a in each iteration of the Kleene star; the language interpretation removes the structure from the expression in the translation to sets of strings. In the type interpretation this structure is kept, so in Example 3.10 (the type interpretation of the same expression) there is a difference between the two as in the value set. We notice that there are multiple values that flatten to the same underlying string, for example,

$$|[\mathsf{inl}\ a, \mathsf{inr}\ \mathsf{inl}\ b, \mathsf{inr}\ \mathsf{inr}\ a]| = |[\mathsf{inr}\ \mathsf{inr}\ a, \mathsf{inr}\ \mathsf{inl}\ b, \mathsf{inl}\ a]| = \mathrm{aba}.$$

This is an example of an *ambiguous* regular expression:

Definition 3.20. A regular expression E is *ambiguous* if and only if two or more values of its values in the type interpretation flatten to the same string:

$$\exists v, w \in \mathcal{V}[\![E]\!].\ |v| = |w| \wedge v \neq w.$$

To *parse* a word with a regular expression E is to produce a value in $\mathcal{V}[\![E]\!]$ that flattens to that word. For ambiguous regular expressions, we must *disambiguate* between several possible values. This should happen in a deterministic way, such that a programmer always gets the same value if she parses the same string under the same regular expression. One such disambiguation policy is called the *greedy* policy. Informally, using a greedy disambiguation policy corresponds to picking the "left-most" possibility whenever more than one is possible.

Definition 3.21 (Greedy order on values [57]). The binary relation $\lessdot$ is defined inductively on the structure of values as follows:

$$\begin{array}{rcll}
\langle v_1, v_2\rangle & \lessdot & \langle v_1', v_2'\rangle & \text{if } v_1 \lessdot v_1' \vee (v_1 = v_1' \wedge v_2 \lessdot v_2') \\
\mathsf{inl}\ v_0 & \lessdot & \mathsf{inl}\ v_0' & \text{if } v_0 \lessdot v_0' \\
\mathsf{inr}\ v_0 & \lessdot & \mathsf{inr}\ v_0' & \text{if } v_0 \lessdot v_0' \\
\mathsf{inl}\ v_0 & \lessdot & \mathsf{inr}\ v_0' & \\
[v_1, \ldots] & \lessdot & [] & \\
[v_1, \ldots] & \lessdot & [v_1', \ldots] & \text{if } v_1 \lessdot v_1' \\
[v_1, v_2, \ldots] & \lessdot & [v_1, v_2', \ldots] & \text{if } [v_2, \ldots] \lessdot [v_2', \ldots]
\end{array}$$

Now the problem of disambiguation can be formulated in terms of picking a minimum element in the greedy ordering $\lessdot$. Of course, the order is not total if one compares elements from different regular expressions E and F. However, if only elements with the same type, i.e., from the same expression, are compared, it is strict and total (Proposition 5.3). In order to ensure that a minimum element amongst possible values—and bit-coded values—exists, we have to define a subset of $\mathcal{V}[\![E]\!]$ called the *non-problematic values*. The reason for this is that for some regular expressions no least element exist in the greedy ordering, and therefore we cannot pick out the minimum element in the ordering:

Example 3.12. Let $E = (1 + \mathsf{a})^\star$. The following are all values in $\mathcal{V}[\![E]\!]$ that flatten to aa:

$$\begin{aligned} t_0 &= [\mathsf{inr}\ \mathsf{a}, \mathsf{inr}\ \mathsf{a}], \\ t_1 &= [\mathsf{inl}\ (), \mathsf{inr}\ \mathsf{a}, \mathsf{inr}\ \mathsf{a}], \\ t_2 &= [\mathsf{inl}\ (), \mathsf{inl}\ (), \mathsf{inr}\ \mathsf{a}, \mathsf{inr}\ \mathsf{a}], \\ &\ \vdots \end{aligned}$$

so no minimum element exists because of the infinite descending chain $t_0 \gtrdot t_1 \gtrdot t_2 \gtrdot \dots$

Definition 3.22 (Non-problematic values [57]). The *non-problematic values* of a regular expression E are:

$$\begin{aligned} \mathcal{V}^{\mathsf{np}}[\![\cdot]\!] &: \mathsf{Exp}_\Sigma \to 2^{\mathsf{Val}_\Sigma} \\ \mathcal{V}^{\mathsf{np}}[\![0]\!] &= \emptyset \\ \mathcal{V}^{\mathsf{np}}[\![1]\!] &= \{()\} \\ \mathcal{V}^{\mathsf{np}}[\![\mathsf{a}]\!] &= \{a\} \\ \mathcal{V}^{\mathsf{np}}[\![E_1 E_2]\!] &= \mathcal{V}^{\mathsf{np}}[\![E_1]\!] \times \mathcal{V}^{\mathsf{np}}[\![E_2]\!] \\ \mathcal{V}^{\mathsf{np}}[\![E_1 + E_2]\!] &= \mathcal{V}^{\mathsf{np}}[\![E_1]\!] \oplus \mathcal{V}^{\mathsf{np}}[\![E_2]\!] \\ \mathcal{V}^{\mathsf{np}}[\![E_1^\star]\!] &= \{[v_1, \dots, v_n] \mid v_i \in \mathcal{V}^{\mathsf{np}}[\![E_1]\!] \setminus \{w \mid w \in \mathcal{V}^{\mathsf{np}}[\![E_1]\!], |w| = \epsilon\}\}\,. \end{aligned}$$

Note that the only difference between $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{V}^{\mathsf{np}}[\![\cdot]\!]$ is that the latter do not allow "empty" elements in lists—elements that flatten to the empty string. This corresponds to what most programmers probably expect when defining iteration-like structures: programs should not be allowed to take an arbitrary number of unnecessary iterations between doing the actual work. Limiting ourselves to the non-problematic values rules out the problem from Example 3.12.

We refer to regular expressions whose type interpretation do not contain any problematic values as *non-problematic regular expressions*:

Definition 3.23 (Non-problematic regular expression [57]). A regular expression E is *non-problematic* if it does not contain any subterms of the form

$$E_0^\star$$

where $\epsilon \in E_0$.

With the refined notion of values, we can now formulate the definition of parsing to ensure that minimum elements do exist:

Definition 3.24 (Parsing). Given regular expression E and word $w \in \Sigma^\star$, to *parse* w under E is to produce the value t such that:

$$t \in \mathcal{V}^{\mathsf{np}}[\![E]\!] \wedge \forall t' \in \mathcal{V}^{\mathsf{np}}[\![E]\!].\, t \lessdot t' \vee t = t'.$$

On bit-codes we can formulate an ordering that is similar to the greedy order $\lessdot$:

Definition 3.25 (Lexicographical ordering). The *lexicographical ordering* $\prec$ between two bit sequences $d, d' \in \{0,1\}^\star$ is the least relation satisfying:

1. $\epsilon \prec d$ if $d \neq \epsilon$,
2. $0 \cdot d \prec 1 \cdot d'$,
3. $0 \cdot d \prec 0 \cdot d'$ if $d \prec d'$,
4. $1 \cdot d \prec 1 \cdot d'$ if $d \prec d'$.

The greedy order on non-problematic values correspond exactly to the lexicographical order on the bit-coded values. This is easily seen by combining the encoding function $\ulcorner\cdot\urcorner$ with the greedy order (Theorem 5.2).

Theorem. *For any RE E and values $v, v' \in \mathcal{V}[\![E]\!]$, $v \lessdot v'$ if and only if $\ulcorner v \urcorner \prec \ulcorner v' \urcorner$.*

3.6 Finite State Transducers

Just as we can use finite automata to decide language membership for regular expressions, we can use a slightly more involved class of automata to generate the set of bit-coded values of a regular expression. Recall that a structured value in $\mathcal{V}[\![E]\!]$ corresponds to a parse tree for a certain word $w \in \mathcal{L}[\![E]\!]$. That is, the value specifies *how* that word is a member of the language of E. The way we decided whether the word was in $\mathcal{L}[\![E]\!]$ or not was to follow all the paths through its corresponding NFA and checking whether one of them led to a final state. We shall see that a structured value corresponds to one of these paths.

For a given regular expression E there is a one-to-one correspondence between its structured values and their bit-codes, so it suffices to compute bit-code for an input string. Note that in the Thompson construction, whenever a node has more than one outgoing transitions it is *always* two ϵ-transitions. This happens exactly when there is a $+$ or a Kleene star in the expression. Uniquely specifying a path through the automaton requires only that each of these choice points is resolved, so it requires a sequence of bits. This is exactly the same bit sequence as the bit-codes, so by recording these we get exactly the bit-coded values.

Definition. A *finite state transducer* (FST) is a six-tuple

$$(Q, \Sigma, \Gamma, I, F, \Delta)$$

where

- Q is a set of *states*,
- Σ and Γ are the *input alphabet* and *output alphabet*, respectively,
- $I \subseteq Q$ is the set of *initial states*,
- $F \subseteq Q$ is the set of *final states*,
- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ is the *transition relation*.

Whenever there is a transition $(q, a, b, q') \in \Delta$ we write $q \xrightarrow{a|b} q'$. The pair $(a, b) \in (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$ is called the *label* of the FST transition. As with the NFAs, we shall restrict us to a certain class of FSTs that are analogous to the Thompson NFAs.

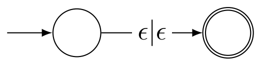
Definition 3.26. A *Thompson* FST is an FST constructed with the procedure shown below, where:

- the output alphabet is bits: $\Gamma = \{\epsilon, 0, 1\}$,
- the set of initial states is a singleton: $I = \{q^{\text{in}}\}$,
- the set of final states is a singleton: $F = \{q^{\text{fin}}\}$,
- all states either:
  - have exactly one outgoing transition labelled $a|\epsilon$, $a \in \Sigma$;
  - have exactly one outgoing transition labelled $\epsilon|\epsilon$;
  - have exactly two outgoing transitions labelled $\epsilon|0$ and $\epsilon|1$, respectively, or;
  - have zero outgoing transitions.

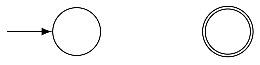
In the following, we will focus only on Thompson FSTs.

Analogous to the Thompson NFA construction, we construct Thompson FSTs. We call a Thompson FST constructed from a regular expression E $\mathcal{F}_E$:

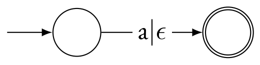
- Case $E = 1$:



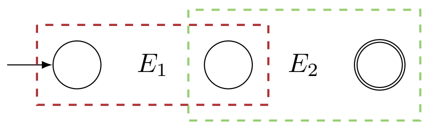
- Case $E = 0$:



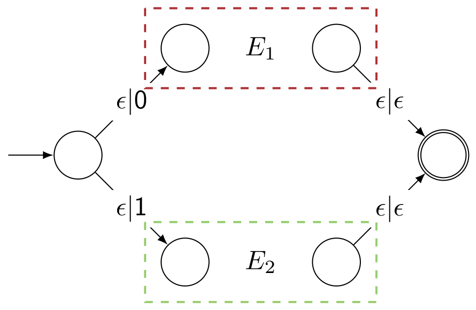
- Case $E = \text{a}$:



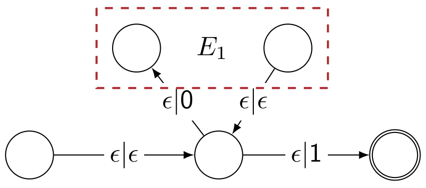
- Case $E = E_1E_2$:



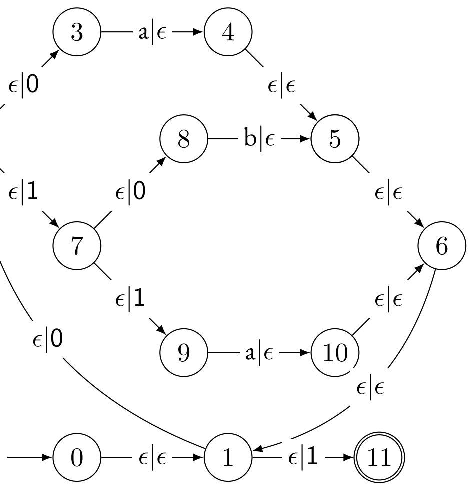
- Case $E = E_1 + E_2$:



- Case $E = E_1^\star$:



Example 3.13. Constructing the Thompson FST for the expression E_{ab} corresponding the Thompson NFA of Example 3.4 yields the FST $\mathcal{F}_{E_{ab}}$:



A path through an FST is defined in the same way as a path through an NFA, except that the sequence of labels is a sequence of pairs of input/output symbols in:

Definition 3.27 (Thompson FST paths). A *path* in a Thompson FST is a non-empty list of states

$$q_0, q_1, \dots, q_n$$

where for each $0 \le i < n$ and $(x_i, y_i) \in (\Sigma \cup \{\epsilon\}) \times \{\epsilon, 0, 1\}$

$$q_i \xrightarrow{x_i|y_i} q_{i+1}.$$

If there is a path from q_0 to q_n with the labels $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ we write

$$q_0 \overset{v|w}{\rightsquigarrow} q_n$$

where $w = x_0 \cdot x_1 \cdot \ldots \cdot x_n$ and $v = y_0 \cdot y_1 \cdot \ldots \cdot y_n$.

Just as we introduced a subset of values called the non-problematic values, we introduce a subset of paths called the *non-problematic paths*:

Definition 3.28 (Non-problematic paths [57]). A *non-problematic path* in a Thompson FST is a path with no ϵ-cycles:

$$q_0, q_1, \dots, q_n$$

such that $\neg\exists i.\, q_i \overset{\epsilon|v}{\rightsquigarrow} q_i$.

Example 3.14. Recall from Section 3.5.5 that the expression E_{ab} is *ambiguous*. That is, there exists more than one value that flatten to the same underlying string. In Example 3.13 we can see that, for the input word aab, the following is a path between $q^{\mathsf{in}} = 0$ and $q^{\mathsf{fin}} = 11$:

$$0, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 7, 8, 5, 6, 1, 11$$

with the output labels:

$$\epsilon \cdot 0 \cdot 0 \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot 0 \cdot 0 \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot 0 \cdot 1 \cdot 0 \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot 1 = 00000101.$$

This is another path from 0 to 11:

$$0, 1, 2, 7, 9, 10, 6, 1, 2, 7, 9, 10, 6, 1, 2, 7, 8, 5, 6, 1, 11$$

with the output labels:

$$\epsilon \cdot 0 \cdot 1 \cdot 1 \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot 0 \cdot 1 \cdot 1 \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot 0 \cdot 1 \cdot 0 \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot 1 = 0110110101.$$

As the above example hints at, there is a correspondence between the output labels on paths through a Thompson FST $\mathcal{F}_E$ and the bit-coded values in $\mathcal{V}[\![E]\!]$ (Theorem 5.1):

Theorem (Representation). *Given a Thompson FST $\mathcal{F}_E$ for a regular expression E, for any word $w \in \mathcal{L}[\![E]\!]$:*

$$\mathcal{B}_w[\![E]\!] = \left\{ b \mid q^{\mathsf{in}} \overset{w|b}{\rightsquigarrow} q^{\mathsf{fin}} \right\}.$$

Corollary. *The set of outputs of all paths through $\mathcal{F}_E$ is the set of bit-codes for E:*

$$\mathcal{B}[\![E]\!] = \left\{ b \mid q^{\mathsf{in}} \overset{w|b}{\rightsquigarrow} q^{\mathsf{fin}} \wedge w \in \mathcal{L}[\![E]\!] \right\}.$$

An important property of the bit-codes is that the set of bit-codes for an expression E is *prefix free* (Definition 2.10):

Lemma 3.1 (Bit-codes are prefix free). *The set of bit-codes for a regular expression E, $\mathcal{B}[\![E]\!]$, is prefix free.*

Proof. By induction on the structure of the Thompson FST $\mathcal{F}_E$. We illustrate the case for alternation. Assume that $\mathcal{B}[\![E_1]\!]$ and $\mathcal{B}[\![E_2]\!]$ are prefix free and construct the Thompson FST for $E = E_1 + E_2$. Any successful path through $\mathcal{F}_E$ must either pass through $\mathcal{F}_{E_1}$ or $\mathcal{F}_{E_2}$. The output of all paths through the former is $B_1 = \{0 \cdot b \mid b \in \mathcal{B}[\![E_1]\!]\}$ and of all paths through the latter is $B_2 = \{1 \cdot b \mid b \in \mathcal{B}[\![E_2]\!]\}$. Hence, the outputs of paths through $\mathcal{F}_E$ are in $B_1 \cup B_2$. It is clearly the case that

$$\forall b_1 \in B_1. \forall b_2 \in B_2. b_1 \not\sqsubseteq b_2,$$

so since we assumed by induction that it holds for the sub-FSTs, $\mathcal{B}[\![E]\!]$ is prefix free. □

3.7 First Algorithm (CIAA'13)

Analogously to the simulation algorithm for NFAs presented in Definition 3.10, we need a way to use the FSTs with the embedded bit-code information to produce values. Because FSTs have different outputs instead of just "success" or "failure", simulating an FST is more complicated than simulating an NFA. Furthermore, because of the ambiguity in regular expressions a simulation algorithm must ensure that the correct output is picked, i.e., the output that corresponds to the least value as defined in Section 3.5.5. Now the reason for using the restricted class Thompson FSTs instead of general FSTs becomes apparent: the complete syntactic structure of the source regular expression is contained in the FST structure. This means that the lexicographically least bit-code is the one that is output on the *leftmost* path, prioritizing transitions labeled $\epsilon|0$ over those labeled $\epsilon|1$ when a path can be continued by following either one. However, just greedily taking the left alternative every time results in a backtracking algorithm with a worst-case running time exponential in the size of the input word! The crux of the NFA simulation algorithm is that it simulates all paths *simultaneously*, thereby eliminating the need for backtracking. We mirror this in the parsing algorithm, but extended to FSTs.

3.7.1 Symmetry of Thompson FSTs

Observe that Thompson FSTs are *symmetric*: whenever a state has two *outgoing* edges there is *exactly* one corresponding state with two *incoming* edges. This can easily be verified by inspecting the cases in construction algorithm in Section 3.6: only two cases introduce states with two incoming edges, namely the cases for $E_1 + E_2$ and $E_1^\star$, and the one-to-one correspondence is maintained in both. All remaining cases trivially maintain the symmetry. Note that in the Kleene star case, one state has both two incoming and two outgoing edges, which also preserves the one-to-one correspondence. This symmetry was also shown by Giammarresi, Ponty, and Wood by associating *Dyck languages*, languages of well-nested parentheses, to "Thompson digraphs," the graph-theoretical counterpart to Thompson-style NFAs [58].

We call states with two outgoing edges *choice states*, and states with two incoming edges *join states*:

Definition 3.29. Given a Thompson FST $\mathcal{F} = (Q, \Sigma, \{\epsilon, 0, 1\}, q^{\mathsf{in}}, q^{\mathsf{fin}}, \Delta)$, let $|q|^{\mathsf{O}}$ and $|q|^{\mathsf{I}}$ be the out- and in-degree of a state q, respectively:

- $C_{\mathcal{F}} = \left\{q \mid q \in Q, |q|^{\mathsf{O}} = 2\right\}$ is the set of *choice states* in $\mathcal{F}$,
- $J_{\mathcal{F}} = \left\{q \mid q \in Q, |q|^{\mathsf{I}} = 2\right\}$ is the set of *join states* in $\mathcal{F}$,
- $S_{\mathcal{F}} = \{q \mid q \in Q, \exists q' \in Q.\, a|\epsilon q'\}$ is the set of *source* states.

3.7.2 Ordered FST Simulation

If we maintain an *order* on the state set while simulating the FST, it becomes possible to formulate that one alternative should be preferred to another. We can therefore augment the algorithm of Definition 3.10 to maintain state *lists* instead of state *sets*. We also alter the ϵ-closure function to only return a list of source states or the final state, as it unnecessary to keep track of the entire ϵ-closure when a stepping function is going to discard all states with no outgoing transitions anyway.

Using ordered variants of the ϵ-closure and the stepping function, the (reversed) lexicographically least bit-code can be obtained by: 1) generating a sequence of states for each input symbol and 2) using this list of sequences as an oracle to step backwards though the

FST and emitting the bit-codes on the transitions. This first sketch of the parsing algorithm is called the *fat log–algorithm*, and we illustrate its operation with the example below:

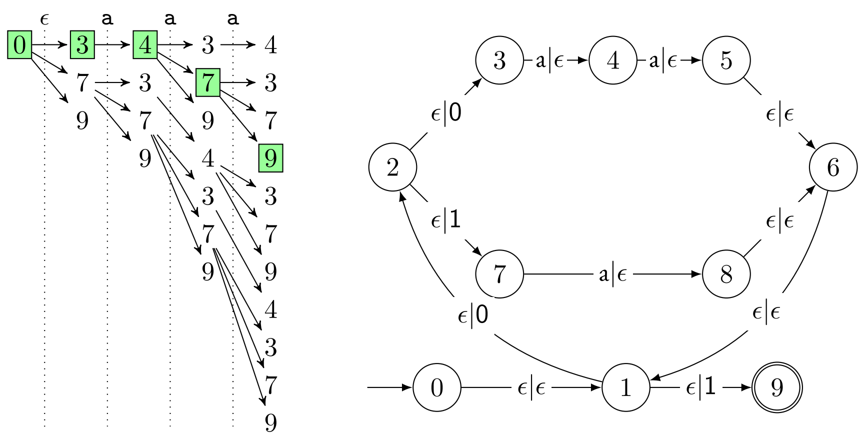
Example 3.15. The expression

$$E_{ambig} = (\mathsf{aa} + \mathsf{a})^{\star}$$

is ambiguous. Input word aaa can be obtained by flattening either of the following values:

$$\begin{array}{ll}
[\mathsf{inl}\ \langle \mathsf{a}, \mathsf{a} \rangle, \mathsf{inr}\ \mathsf{a}] & (00011), \\
[\mathsf{inr}\ \mathsf{a}, \mathsf{inl}\ \langle \mathsf{a}, \mathsf{a} \rangle] & (01001), \text{ or} \\
[\mathsf{inr}\ \mathsf{a}, \mathsf{inr}\ \mathsf{a}, \mathsf{inr}\ \mathsf{a}] & (0101011).
\end{array}$$

The least value in the greedy ordering is the first one. Pictured below is the Thompson FST $\mathcal{F}_{E_{ambig}}$ along with the list of active source states and final state in each iteration of the simulation algorithm. The state lists are ordered such that the topmost element at each instance is the highest prioritized (i.e., leftmost) state:



The final state list is $[4, 3, 7, 9, 3, 7, 9, 4, 3, 7, 9]$. Because the final state 9 occurs three times in the state list, we know that there are three separate paths from the initial state to the final state, corresponding to the three different values above. It is only the greedy left-most path through the FST we are interested in, and due to the fact that the states are ordered using the lexicographical order on the transition labels we know that the correct accepting path is the one that ends in the *topmost* final state in the list, i.e., the first 9. Hence, we follow this path backwards through the FST (marked with 🟩) and reconstruct the bit-coded by emitting bits on the traversed edges:

$$9 \overset{\mathsf{a}|1}{\leftsquigarrow} 7 \overset{\mathsf{a}|10}{\leftsquigarrow} 4 \overset{\mathsf{a}|\epsilon}{\leftsquigarrow} 3 \overset{\epsilon|00}{\leftsquigarrow} 0$$

yields 11000, the (reversed) correct bit-code. The other two paths correspond to the other two ways of parsing the input word:

$$9 \overset{\mathsf{a}|1}{\leftsquigarrow} 4 \overset{\mathsf{a}|\epsilon}{\leftsquigarrow} 3 \overset{\mathsf{a}|00}{\leftsquigarrow} 7 \overset{\epsilon|10}{\leftsquigarrow} 0 \qquad (10010)$$

$$9 \overset{\mathsf{a}|1}{\leftsquigarrow} 7 \overset{\mathsf{a}|10}{\leftsquigarrow} 7 \overset{\mathsf{a}|10}{\leftsquigarrow} 7 \overset{\epsilon|10}{\leftsquigarrow} 0 \qquad (1101010).$$

3.7.3 Two-Pass Regular Expression Parsing

A lot of unnecessary information is stored in the fat log in Example 3.15:

1. the final state list contains the final states several times, representing the fact that *all* possible paths through $\mathcal{F}_{E_{ambig}}$ is computed, even though we only realize the left-most path;
2. when walking backwards through $\mathcal{F}_{E_{ambig}}$ the only time the log is consulted as an oracle is in the join states;
3. in the join states the log is always to decide between exactly two possibilities.

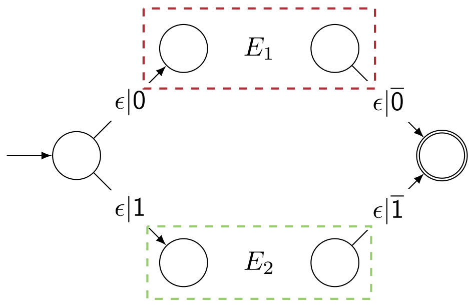
These observations lead us to the following:

1. repeated states may be deleted from state lists, keeping only the leftmost states;
2. we only need to store information about the join states in the log;
3. we only need to store one bit of information per join state in the log.

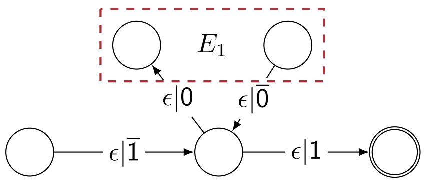
Hence, instead of storing a list of states we only need to store one bit per join state per input symbol! Now the symmetry of Thompson FSTs can be exploited: because of the one-to-one correspondence between choice and join states, it suffices to record for each join state which of its two incoming transitions was used. The symmetry ensures that paths following the left-most (right-most) incoming transition to a join state also *must* have followed the outgoing transition marked $\mathsf{0}$ ($\mathsf{1}$) of the corresponding choice state. Consequently, we decorate the Thompson FSTs with *log labels*:

Definition 3.30 (Log FST). Given a Thompson FST $\mathcal{F}_E = (Q, \Sigma, \{\epsilon, \mathsf{0}, \mathsf{1}\}, q^{\mathsf{in}}, q^{\mathsf{fin}}, \Delta)$, the *log FST* $\mathcal{F}_E^{\mathsf{L}} = (Q, \Sigma, \{\mathsf{0}, \mathsf{1}, \overline{\mathsf{0}}, \overline{\mathsf{1}}\}, q^{\mathsf{in}}, q^{\mathsf{fin}}, \Delta')$ is obtained by constructing an FST in the same way as the normal Thompson FST, except for the following cases:

- Case $E = E_1 + E_2$:


E_1
$\epsilon|\mathsf{0}$
$\epsilon|\overline{\mathsf{0}}$
$\epsilon|\mathsf{1}$
$\epsilon|\overline{\mathsf{1}}$
E_2

- Case $E = E_1^\star$:


E_1
$\epsilon|\mathsf{0}$
$\epsilon|\overline{\mathsf{0}}$
$\epsilon|\overline{\mathsf{1}}$
$\epsilon|\mathsf{1}$

In Grathwohl, Henglein, Nielsen, and Rasmussen [64] a log FST is called an *augmented NFA* (aNFA). In the remainder of this section all definitions refer to an implicit log FST $\mathcal{F}^{\mathsf{L}}$.

For each input symbol we store an object that maps join states to log labels. Such objects are called *log frames* (Definition 5.3):

$$\ell\colon \left\{\overline{0}, \overline{1}\right\}^{J_{\mathcal{F}^{\mathsf{L}}}},$$

i.e., functions from join states $J_{\mathcal{F}^{\mathsf{L}}}$ in the log FST $\mathcal{F}^{\mathsf{L}}$ to log labels $\left\{\overline{0}, \overline{1}\right\}$.

In the following we use $\odot$ to notate the point-wise operation that concatenates the first components and unions the second components (Definition 5.4):

$$([q_0, \ldots, q_n], \ell_0) \odot ([q'_0, \ldots, q'_m], \ell_1) = ([q_0, \ldots, q_n, q'_0, \ldots, q'_m], \ell_0 \cup \ell_1).$$

The final two-phase parsing algorithm can be formulated similarly to the way the NFA simulation algorithm of Section 3.3.1 is formulated: we enrich the ϵ-closure Close (Definition 5.5) and stepping function Step (Definition 5.6) to preserve the required ordering information.

Both the closure and the stepping algorithm is described in Chapter 5. Here, we demonstrate their workings on some examples.

Example 3.16. Applying Close on states 4 and 8 of $\mathcal{F}^{\mathsf{L}}_{E_{ambig}}$ from Example 3.15:

$$\begin{aligned}
\mathsf{Close}(4, \emptyset) &= ([4], \emptyset)\\
\mathsf{Close}(8, \emptyset) &= \mathsf{Close}(6, \left\{6 \mapsto \overline{1}\right\})\\
&= \mathsf{Close}(1, \left\{1 \mapsto \overline{0}, 6 \mapsto \overline{1}\right\})\\
&= \mathsf{Close}(2, \ell_0) \odot \mathsf{Close}(9, \ell_0)\\
&= \left(\mathsf{Close}(3, \ell_0) \odot \mathsf{Close}(7, \ell_0)\right) \odot ([9], \ell_0)\\
&= \left([3], \ell_0) \odot ([7], \ell_0)\right) \odot ([9], \ell_0)\\
&= ([3, 7], \ell_0) \odot ([9], \ell_0)\\
&= ([3, 7, 9], \ell_0)
\end{aligned}$$

where we set $\ell_0 = \left\{1 \mapsto \overline{0}, 6 \mapsto \overline{1}\right\}$ for notational convenience.

The stepping function respects the order of the state lists and combines the log frames from each intermediate step.

Example 3.17. We illustrate Step on $\mathcal{F}^{\mathsf{L}}_{E_{ambig}}$:

$$\begin{aligned}
&\mathsf{Step}([3, 7, 9], \mathtt{a}, ([], \emptyset))\\
&= \text{let } (S, \ell) = \mathsf{Close}(4, \emptyset) \text{ in } \mathsf{Step}([7, 9], \mathtt{a}, ([], \emptyset) \cdot (S, \ell))\\
&= \mathsf{Step}\ ([7, 9], \mathtt{a}, ([4], \emptyset))\\
&= \text{let } (S, \ell) = \mathsf{Close}(8, \emptyset) \text{ in } \mathsf{Step}\ ([9], \mathtt{a}, ([4], \emptyset) \odot (S, \ell))\\
&= \mathsf{Step}\ \left([9], \mathtt{a}, \left([4], \emptyset\right) \odot \left([3, 7, 9], \left\{1 \mapsto \overline{0}, 6 \mapsto \overline{1}\right\}\right)\right)\\
&= \mathsf{Step}\ \left([], \mathtt{a}, \left([4, 3, 7, 9], \left\{1 \mapsto \overline{0}, 6 \mapsto \overline{1}\right\}\right)\right)\\
&= \left([4, 3, 7, 9], \left\{1 \mapsto \overline{0}, 6 \mapsto \overline{1}\right\}\right)
\end{aligned}$$

The *forward pass* Fwd of the algorithm is just the combination of Close and Step analogously to Reach for NFAs (Definition 5.7). The output of this pass is a list of log frames that is read in the *backward pass* Bwd. Here, it is used as an oracle to walk from the final to the initial state and reading off the output bits of the log FST (Definition 5.8).

Finally, Fwd and Bwd is combined to obtain the *two-phase parsing algorithm* of Grathwohl, Henglein, Nielsen, and Rasmussen [64]. We show Definition 5.9 early:

Definition. The complete *two-phase parsing algorithm* is the composition of the forward and backward passes of Definitions 5.7 and 5.8:

$$\begin{aligned}\mathsf{TwoPhaseParser}\colon \Sigma^\star &\to \{0,1\}^\star \cup \{\bot\}\\ \mathsf{TwoPhaseParser}(w) &= \mathsf{Bwd}\left(\mathsf{Fwd}(w), q^{\mathsf{fin}}\right).\end{aligned}$$

Note that the algorithm does not require that the input word is stored between passes; only the log frame is needed. Furthermore, the log frames are used as a stack: first, all the log frames are pushed onto the stack, and after the last input symbol the log frames are all popped from the stack in the backward pass. The *working memory* requirement is therefore $k \cdot n$ bits for a log FST with k join states and an input word with length n. Given an FST of size m, a constant amount of work is performed at each input symbol in the forward pass and at each log frame in the backward pass, so the worst-case running time is $O(mn)$.

Example 3.18. Consider again the expression from Example 3.15. On the input word $w =$ aaa the forward pass of the lean log algorithm will produce the log L^{aaa}:

$$\begin{aligned}L^{\mathrm{aaa}} &= \mathsf{Fwd}(\mathrm{aaa})\\ &= \text{let } (S', \ell') = \mathsf{Close}(0, []) \text{ in } \mathsf{Fwd}'(\mathrm{aaa}, S', [\ell'])\\ &= \text{let } (S', \ell') = ([3,7,9], \{1 \mapsto \overline{1}\}) \text{ in } \mathsf{Fwd}'(\mathrm{aaa}, S', [\ell'])\\ &= \mathsf{Fwd}'(\mathrm{aaa}, [3,7,9], [\ell_0])\\ &= \text{let } (S', \ell') = \mathsf{Step}([3,7,9], \mathrm{a}, ([], \emptyset)) \text{ in } \mathsf{Fwd}'(\mathrm{aa}, S', [\ell', \ell_0])\\ &= \mathsf{Fwd}'(\mathrm{aa}, [4,3,7,9], [\ell_1, \ell_0])\\ &= \text{let } (S', \ell') = \mathsf{Step}([4,3,7,9], \mathrm{a}, ([], \emptyset)) \text{ in } \mathsf{Fwd}'(\mathrm{a}, S', [\ell', \ell_1, \ell_0])\\ &= \mathsf{Fwd}'(\mathrm{a}, [3,7,9,4], [\ell_2, \ell_1, \ell_0])\\ &= \text{let } (S', \ell') = \mathsf{Step}([3,7,9,4], \mathrm{a}, ([], \emptyset)) \text{ in } \mathsf{Fwd}'(\epsilon, S', [\ell', \ell_2, \ell_1, \ell_0])\\ &= \mathsf{Fwd}'(\epsilon, [4,3,7,9], [\ell_3, \ell_2, \ell_1, \ell_0])\\ &= \left[\{1 \mapsto \overline{0}, 6 \mapsto \overline{1}\}, \{1 \mapsto \overline{0}, 6 \mapsto \overline{0}\}, \{1 \mapsto \overline{0}, 6 \mapsto \overline{1}\}, \{1 \mapsto \overline{1}\}\right]\end{aligned}$$

where we have used the log frame abbreviations:

$$\begin{aligned}\ell_0 &= \{1 \mapsto \overline{1}\}\\ \ell_1 &= \{1 \mapsto \overline{0}, 6 \mapsto \overline{1}\}\\ \ell_2 &= \{1 \mapsto \overline{0}, 6 \mapsto \overline{0}\}\\ \ell_3 &= \{1 \mapsto \overline{0}, 6 \mapsto \overline{1}\}\end{aligned}$$

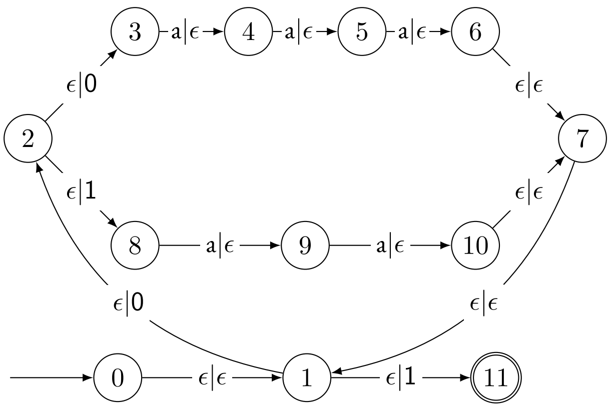
The backward pass with L^{aaa} yields:

$$
\begin{aligned}
&\mathsf{Bwd}\left(\left[\{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{0}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{1}\}\right], 9\right) \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{0}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{1}\}\right], 1\right) \cdot \mathsf{1} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{0}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{1}\}\right], 6\right) \cdot \mathsf{1} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{0}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{1}\}\right], 8\right) \cdot \mathsf{1} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{0}, 6 \mapsto \bar{0}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{1}\}\right], 7\right) \cdot \mathsf{1} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{0}, 6 \mapsto \bar{0}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{1}\}\right], 2\right) \cdot \mathsf{11} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{0}, 6 \mapsto \bar{0}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{1}\}\right], 1\right) \cdot \mathsf{011} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{0}, 6 \mapsto \bar{0}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{1}\}\right], 6\right) \cdot \mathsf{011} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{0}, 6 \mapsto \bar{0}\}, \{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{1}\}\right], 5\right) \cdot \mathsf{011} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{0}, 6 \mapsto \bar{1}\}, \{1 \mapsto \bar{1}\}\right], 4\right) \cdot \mathsf{011} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{1}\}\right], 3\right) \cdot \mathsf{011} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{1}\}\right], 2\right) \cdot \mathsf{0011} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{1}\}\right], 1\right) \cdot \mathsf{00011} \\
&= \mathsf{Bwd}\left(\left[\{1 \mapsto \bar{1}\}\right], 0\right) \cdot \mathsf{00001} \\
&= [] \cdot \mathsf{00001} = \mathsf{00001}.
\end{aligned}
$$

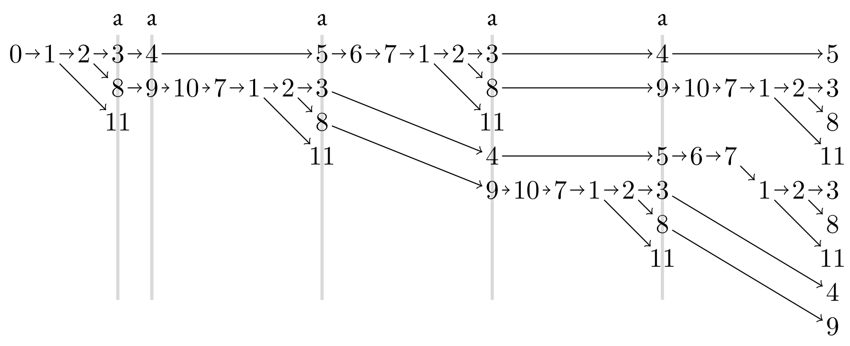
3.8 Second Algorithm (ICTAC'14)

The algorithm presented in the preceding section uses two passes to produce the bit-coded value: only after a log frame has been generated for every input symbol can the backward pass start piecing together the bit-coded value to be output. For a large class of inputs this is an undesirable behavior. Consider the expression $E = (\mathrm{a} + \mathrm{b})^\star$: after reading each input symbol a or b in an input word we are *guaranteed* that the next element in the list is either inl a or inr b. However, this fact is lost in the two-phase algorithm—even though there can be no alternative at a specific input symbol, the algorithm soldiers on until the end of the input word! The straight-forward program that parses fields in a comma separated data file exhibits the same pattern: every time the newline symbol occurs the structure of the current line is completely known and can therefore be output. Often, programs that use extant regular expression libraries are written in a way that makes this pattern very explicit by having a loop that iterates over each line and treat them separately.

It turns out that the pattern is more general than only handling newline symbols as "cut" operations. Consider the FST for the expression $(\mathrm{aaa} + \mathrm{aa})^\star$:



In Example 3.15 the entire branching structure of all paths between the initial and final state of the FST $\mathcal{F}_{E_{ambig}}$ was computed. Doing the same for the above FST on input word aaaaa and including all intermediate states in the tree yields:



There are two things to note about this figure:

1. The tree contains paths that do not continue until the end of input, either because there is no outgoing transition, as in $[0, 1, 11]$, or because the states that would have been reached by continuing from a path are already contained in the ordered states set, as in state 5 at the fifth a.

2. Moreover, consider the possible words that can be read from states 5 and 3: the set of words that results in a path between 5 and 11 is $\mathcal{L}[\![\text{a}(\text{aaa} + \text{aa})^\star]\!]$ and the words read on paths between 3 and 11 are in $\mathcal{L}[\![\text{aaa}(\text{aaa} + \text{aa})^\star]\!]$. It is not difficult to see that $\mathcal{L}[\![\text{aaa}(\text{aaa} + \text{aa})^\star]\!] \subseteq \mathcal{L}[\![\text{a}(\text{aaa} + \text{aa})^\star]\!]$, so every possible word that causes paths from 3 to reach 11 would also cause paths from 5 to reach it. Since 5 is prioritized higher in the state list, any path prefixed with $[0, 1, 2, 3, 4, 5]$ will be prioritized above any path prefixed with $[0, 1, 2, 8, 9, 10, 7, 1, 2, 3]$.

These two observations mean that we may *prune* dead paths when:

1. a path ends in a state with no outgoing transitions for the current input symbol;

2. a path ends in a state that is *covered* by another, higher-prioritized, state.

The idea from the previous section about eliminating repeated states from the state list is a special case of the second point. We will refer to the *language* of a state q as L_q:

Definition. The *language* of a state $q \in Q$ in some FST, L_q, is the set of words:

$$L_q = \left\{ w \mid q \overset{w|x}{\rightsquigarrow} q^{\mathsf{fin}} \right\}.$$

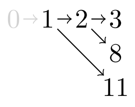
With this, we can formulate coverage (Definition 6.5) as:

Definition (Coverage). A state *q* is *covered* by a set of states Q' if and only if:

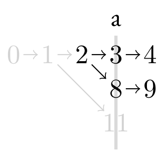
$$L_q \subseteq \bigcup \{ L_{q'} \mid q' \in Q' \}.$$

Before we discuss the new algorithm, let us follow the construction of the tree where we cut away dead paths as we go along:

1. First, an ordered ϵ-closure is performed. Any successful path *must* be prefixed with the path $[0, 1]$. Hence, output bits must be prefixed with the output bits on that path—in this case ϵ. It is not necessary to keep it in the tree, so we prune it away:

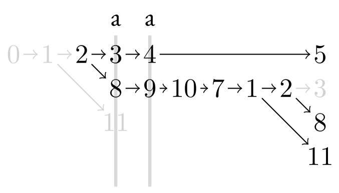


2. After the first a, the path $[1, 11]$ is dead and can be pruned. This means that any successful path through the FST must be prefixed by $[1, 2]$ (after the prefix $[0]$ from above), so $[1, 2]$ can also be pruned and its output bits be emitted: 0. The initial part of the bit-code has been emitted after only one a!

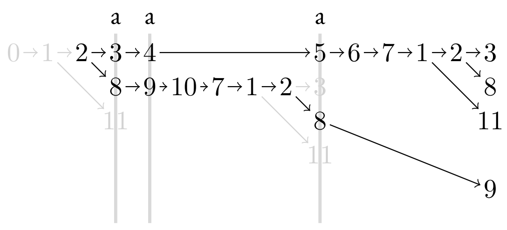


3. After the second a, the new branch that goes from 2 to 3 may be pruned, as state 3 is covered by state 5:

$$L_3 \quad = \quad \mathcal{L}[\![\text{aaa}(\text{aaa} + \text{aa})^\star]\!] \quad \subseteq \quad \mathcal{L}[\![\text{a}(\text{aaa} + \text{aa})^\star]\!] \quad = \quad L_5$$



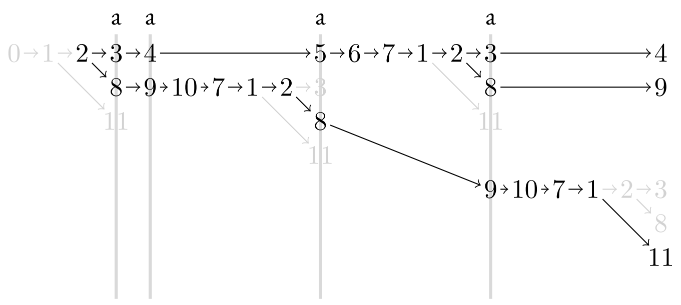
4. After the third a, the dead branch ending in 11 is pruned:



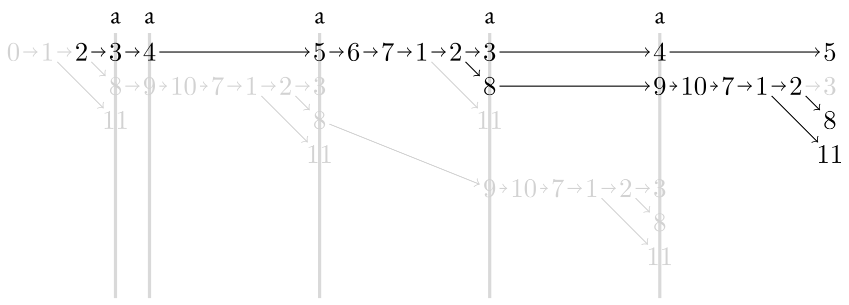
5. At the fourth a, the two paths that end in states 3 and 8 are pruned because they are covered by states 9 and 4. This can be verified by inspecting the languages associated with each state:

$$\begin{array}{rcrcrcl} L_3 & = & \mathcal{L}[\![\mathrm{aaa}(\mathrm{aaa}+\mathrm{aa})^\star]\!] & \subseteq & \mathcal{L}[\![\mathrm{a}(\mathrm{aaa}+\mathrm{aa})^\star]\!] & = & L_9 \\ L_8 & = & \mathcal{L}[\![\mathrm{aa}(\mathrm{aaa}+\mathrm{aa})^\star]\!] & \subseteq & \mathcal{L}[\![\mathrm{aa}(\mathrm{aaa}+\mathrm{aa})^\star]\!] & = & L_4 \end{array}$$

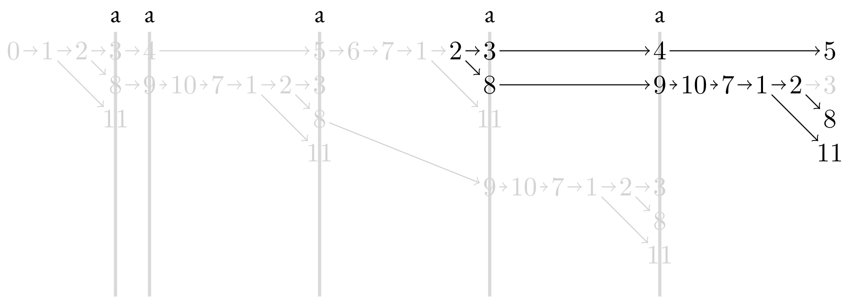
This leaves only three active states in the state list:



6. At the fifth a in the input, the branch that ends in state 11 is pruned. Because there is only one path left in the tree between the first 2 and the pruned state 11 we can safely prune the entire path. State 3 is also pruned because $L_3 \subseteq L_5$, as noted previously. The list of active states now contains three elements:



After pruning it becomes clear that any path that continues either of the three possibilities will be prefixed by $[2, 3, 4, 5, 6, 7, 1, 2]$, so the stem may also be pruned and the output bits 00 emitted:



As this illustrates, pruning stems and dead branches from the tree and emitting output bits on stems lets us do *streaming parsing*. In the particular example, one "a" means that at least one iteration in the Kleene star must be made, so the 0 is guaranteed to prefix any bit-codes. Reading a second "a" does not let us conclude anything more, because the remainder of the input word is unknown. The complete input could be "aaa", in which case the right alternative does not consume any "a"s, or it could be "aa", in which case all "a"s are consumed in the right alternative. The same is the case after three and four "a"s. However, when five "a"s have been read, it is guaranteed that no matter how many symbols follow, the first three must have been consumed by the left alternative, which accounts for the first 0 in the output at step 6, and it must also be the case that at least one more iteration is required to consume at least the remaining two characters. This accounts for the second 0 in the output at step 6.

3.8.1 Optimal Streaming

The algorithm sketched above implements what we shall call *optimally streaming* parsing. Intuitively, an optimally streaming parsing function is a function that emits prefixes of the final bit-code as soon as it is *semantically* possible or, in case the input word is not in the language of the expression being parsed, emits a designated error token as soon as this becomes apparent.

The idea of the "stems" in the trees above can be formalized as *completions* of words (Definition 6.1)

Definition (Completions). The set of *completions* $C_E(w)$ of w for a regular expression E is the set of all words in $\mathcal{L}[\![E]\!]$ that have w as a prefix:

$$C_E(w) = \{w'' \mid w \sqsubseteq w'' \wedge w'' \in \mathcal{L}[\![E]\!]\}\,.$$

Example 3.19. Let $E = (\mathrm{aaa} + \mathrm{aa})^\star$.

$$\begin{aligned}
C_E(\epsilon) &= \mathcal{L}[\![E]\!] \\
C_E(\mathrm{b}) &= \emptyset \\
C_E(\mathrm{a}) &= \mathcal{L}[\![E]\!] \setminus \{\epsilon\} \\
C_E(\mathrm{aa}) &= \mathcal{L}[\![E]\!] \setminus \{\epsilon\} \\
C_E(\mathrm{aaaaa}) &= \;(\{\mathrm{aaaaa}\} \cdot \mathcal{L}[\![E]\!]) \cup \{\mathrm{aaaaaa}\}
\end{aligned}$$

If a word has an empty set of completions under E it means that no words with that prefix exists in $\mathcal{L}[\![E]\!]$. Prefixes of that may have non-empty completions, though. The longest of these prefixes is called the *reduction* of w (Definition 6.3):

Definition (Reduction). If $C_E(w) = \emptyset$, the unique *reduction* $\overline{w}_E$ of w under E is the longest prefix of w with a non-empty completion:

$$\overline{w}_E = \text{longest } w' \text{ such that } w' \sqsubseteq w \wedge C_E(w') \neq \emptyset.$$

Example 3.20. Let $E = (\mathrm{aaa} + \mathrm{aa})^\star$. The word $w = \mathrm{aab}$ is not in $\mathcal{L}[\![E]\!]$, so its set of completions is empty. Its reduction is:

$$\begin{aligned}
\overline{\mathrm{aab}}_E &= \text{longest } w' \text{ such that } w' \sqsubseteq \mathrm{aab} \wedge C_E(w') \neq \emptyset \\
&= \text{longest } w' \in \{\epsilon, \mathrm{a}, \mathrm{aa}\} \\
&= \mathrm{aa}.
\end{aligned}$$

For any parsing function $\mathsf{P}_E(\cdot) : \mathcal{L}[\![E]\!] \to \mathcal{B}[\![E]\!]$ an optimally streaming version can be constructed (Definition 6.4). If it is semantically impossible to do perform output until the last input character has been read, the optimally streaming version of the parser will just be the same function:

Definition (Optimal streaming). The *optimally streaming* function corresponding to $\mathsf{P}_E(\cdot)$ is

$$O_E(w) = \begin{cases} \bigsqcap\{\mathsf{P}_E(w'') \mid w'' \in C_E(w)\} & \text{if } C_E(w) \neq \emptyset \\ (\bigsqcap O_E(\overline{w}))\,\sharp & \text{if } C_E(w) = \emptyset. \end{cases}$$

The symbol $\sharp$ is a failure indicator.

Example 3.21. Let $E = (\mathrm{aaa} + \mathrm{aa})^\star$ and $\mathsf{P}_E(\cdot)$ be a parsing function that implements greedy left-most parsing. The example above can be formulated in terms of the optimally streaming parsing function. Before any input is read, no output is produced:

$$\begin{aligned} \bigsqcap \mathsf{P}_E(C_E(\epsilon)) &= \bigsqcap \mathsf{P}_E(\mathcal{L}[\![E]\!]) \\ &= \bigsqcap \{1, 001, 011, 00001, \ldots\} \\ &= \epsilon. \end{aligned}$$

After the first a the function outputs 0:

$$\begin{aligned} \bigsqcap \mathsf{P}_E(C_E(\mathrm{a})) &= \bigsqcap \mathsf{P}_E(\mathcal{L}[\![E]\!] \setminus \{\epsilon\}) \\ &= \bigsqcap \{001, 011, 00001, \ldots\} \\ &= 0. \end{aligned}$$

At the second a, no new output can be produced:

$$\begin{aligned} \bigsqcap \mathsf{P}_E(C_E(\mathrm{aa})) &= \bigsqcap \mathsf{P}_E(\mathcal{L}[\![E]\!] \setminus \{\epsilon\}) \\ &= \bigsqcap \{001, 011, 00001, \ldots\} \\ &= 0. \end{aligned}$$

As we saw above, this continues until the fifth a:

$$\begin{aligned} \bigsqcap \mathsf{P}_E(C_E(\mathrm{aaaaa})) &= \bigsqcap \mathsf{P}_E((\{\mathrm{aaaaa}\} \cdot \mathcal{L}[\![E]\!]) \cup \{\mathrm{aaaaaa}\}) \\ &= \bigsqcap \{\mathsf{P}_E(w) \mid w \in \{\mathrm{aaaaa}\} \cdot \mathcal{L}[\![E]\!] \cup \{\mathrm{aaaaaa}\}\} \\ &= \bigsqcap \left\{\mathsf{P}_E(w) \mid w \in \left\{\mathrm{a}^5, \mathrm{a}^6, \mathrm{a}^7, \mathrm{a}^8, \ldots\right\}\right\} \\ &= \bigsqcap \{00011, 00001, 0001011, 000001, \ldots\} \\ &= 000. \end{aligned}$$

Example 3.22. Continuing Example 3.21, consider the output on the word aab. Its set of completions is empty, so the second case in Definition 6.4 is taken:

$$\begin{aligned} (O_E(\overline{\mathrm{aab}}))\,\sharp &= (O_E(\mathrm{aa}))\,\sharp \\ &= \left(\bigsqcap \mathsf{P}_E(C_E(\mathrm{aa}))\right)\sharp \\ &= 0\sharp. \end{aligned}$$

A parsing function is not necessarily streaming. For example, if $E = (\text{aaa} + \text{aa})^\star$, the output on a is:

$$\mathsf{P}_E\,(\text{a}) = \bot,$$

whereas the output on aa is

$$\mathsf{P}_E\,(\text{aa}) = 011,$$

but a $\sqsubseteq$ aa. The optimally streaming version of $\mathsf{P}_E\,(\cdot)$, however, *is* streaming (Theorem 6.1):

Theorem. *For a parsing function* $\mathsf{P}_E\,(\cdot)$, O_E *is a streaming function:*

$$w \sqsubseteq w' \implies O_E(w) \sqsubseteq O_E(w').$$

3.8.2 Algorithm

The algorithm presented in Grathwohl, Henglein, and Rasmussen [65] implements optimal streaming (Theorem 6.3). It proceeds as illustrated in the example above—in Chapter 6 it is described in more detail. We only sketch the main points here.

Notions related to a path tree T (Definition 6.6) are called the following:

- $\mathsf{root}(T)$ is the root node of path tree T.
- $\mathsf{path}(n, c)$ is the path from n to c, where c is a descendant of n.
- $\mathsf{init}(T)$ is the path from the root to the first binary node reachable or to the unique leaf of T if it has no binary node.
- $\mathsf{leaves}(T)$ is the *ordered list* of leaf nodes.

After each input symbol, the invariants in Definition 6.7 are established by the algorithm:

Definition. Let T_w be a path tree and w a word. Define $I(T_w)$ as the proposition that *all* of the following hold:

1. The $\mathsf{leaves}(T_w)$ have pairwise distinct node labels; all labels are *source states* or the accept state.
2. All paths from the root to a leaf read w:

$$\forall n \in \mathsf{leaves}(T_w).\, \mathsf{root}(T) \overset{w|b}{\rightsquigarrow} n.$$

3. For each leaf $n \in \mathsf{leaves}(T_w)$ there exists $w'' \in C_E(w)$ such that the bit-coded parse of w'' starts with b where $\mathsf{root}(T_w) \overset{w|b}{\rightsquigarrow} n$.
4. For each $w'' \in C_E(w)$ there exists $n \in \mathsf{leaves}(T_w)$ such that the bit-coded parse of w'' starts with b where $\mathsf{root}(T_w) \overset{w|b}{\rightsquigarrow} n$.

These invariants correspond to the observations in the beginning of this section with the caveat that the *coverage relation* must be pre-computed to guarantee optimal streaming. Unfortunately, deciding coverage is PSPACE-hard (Proposition 6.1). Deciding whether a state q is covered by states Q corresponds to deciding $L_q \subseteq \bigcup\{L_{q'} \mid q' \in Q\}$. As both L_q and all the $L_{q'}$ are regular languages, it requires us to decide language inclusion for two regular languages, a known PSPACE-complete problem [132].

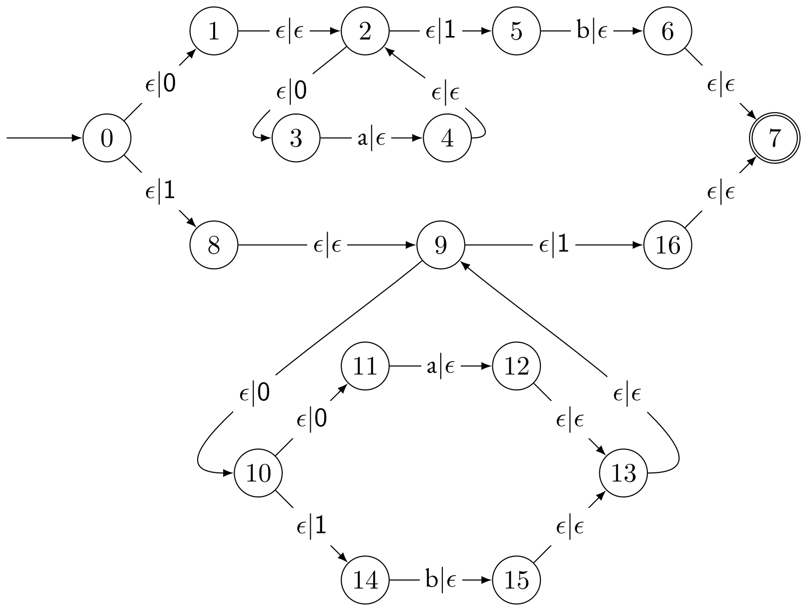
The algorithm proceeds as follows. For each input character the following steps are performed:

1. An ϵ-closure is performed and the current path tree extended. This functions in the same way as the ϵ-closure for the two-phase algorithm.
2. The invariants of Definition 6.7 are reestablished by pruning dead leaves and branches. This makes use of a precomputed coverage relation.
3. After pruning the tree, all bits from the root node to the first binary node is output, that is, all bits on the path $\mathsf{init}(T)$, where T is the current path tree.

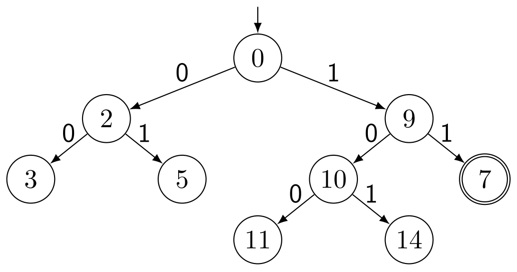
3.9 Determinization, Implementation (POPL'16)

Just as NFAs can be determinized to DFAs, the transducers of the previous sections can also be determinized. However, because they output more than just success/fail the determinization requires more structure. In normal determinization of automata, states are combined into sets of states, representing the fact that each determinized state corresponds to several non-deterministic states.

To maintain the structure that is encoded in the path trees, states in a determinized version of the Thompson FSTs are represented as trees themselves. To illustrate the idea we will focus on the Thompson FST for the expression $E = \mathsf{a}^\star \mathsf{b} + (\mathsf{a} + \mathsf{b})^\star$:



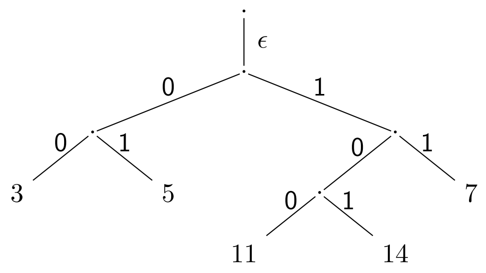
Taking the ordered ϵ-closure from the initial state 0 is and producing a path tree yields the following, where only choice states are kept as internal nodes. We have labeled the edges of the tree with the output bits on the FST paths represented by the edges:



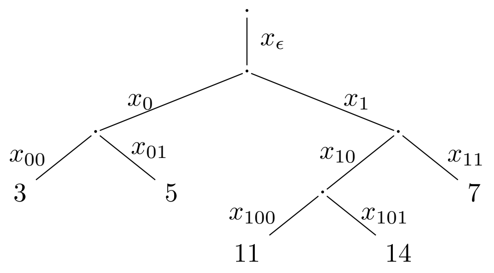
The following observations are important:

- the path tree indicates which states are active in the FST simulation and which relationship they have to each other in the ordering;
- it is not necessary to know the labels of the internal nodes, as the structure of the tree together with the leaves and the FST determine the labels uniquely.

The contents of the path tree that we need to store therefore only requires the leaves and the internal structure:



For a determinization algorithm to terminate, we must ensure that only a finite number of distinct states will be produced. If the trees are identified modulo the labels on their edges there can only be a finite number of different trees: there is a finite number of states in the FST, hence there can only be a finite number of trees with different leaves and internal nodes. Note that this is not true if the labels on the tree edges are kept: the expression $(a^\star b + a^\star c)$ requires an unbounded number of different labeled trees to represent all simulation states. Since the output bits are relevant, we split up the representation of simulation states into a *static* and a *dynamic* part. The static part is the path tree enhanced with a set of named *registers*, and the dynamic part is a valuation of those registers, i.e., a mapping between registers and bit-codes. The static part of the path tree above is:

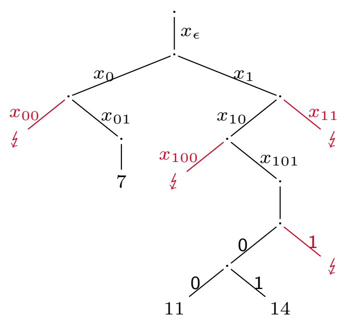


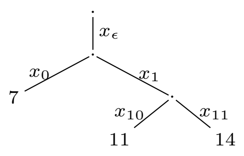
with the following valuation of the registers:

$$\begin{array}{rcl rcl rcl}
x_\epsilon & \mapsto & \epsilon & x_0 & \mapsto & 0 & x_1 & \mapsto & 1 \\
x_{00} & \mapsto & 0 & x_{01} & \mapsto & 1 & x_{10} & \mapsto & 0 \\
x_{11} & \mapsto & 1 & x_{100} & \mapsto & 0 & x_{101} & \mapsto & 1.
\end{array}$$

If we simulate the FST in the manner described in the previous section with the difference that the bit-codes are maintained in registers, we quickly see a finite set of trees that are reached. Whenever a path tree is extended by following a transition and taking the ϵ-closure on all leaves, we collapse all paths in the tree that do not contain any choice nodes. That way, the only internal nodes in the tree are choice nodes. The information stored on the edges on a path that is collapsed is kept in separate registers. After collapsing a path, all the registers are concatenated and added to the register for the new path.

For illustration, the path tree develops like pictured below when a b is consumed from the FST states in the path tree above:

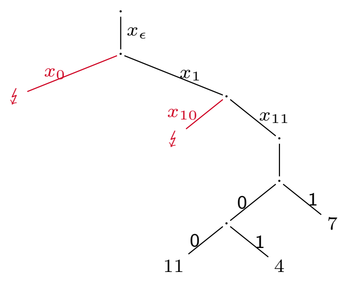


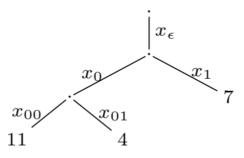


$$\begin{array}{rcl}
x_0 & := & (x_0)(x_{01}) \\
x_1 & := & (x_1)(x_{10})(x_{101})0 \\
x_{10} & := & 0 \\
x_{11} & := & 1
\end{array}$$

The paths that die are marked with red and a $\lightning$. If an a is read, the path tree steps into the same tree.

We perform one more step, now from the new path tree on an a:

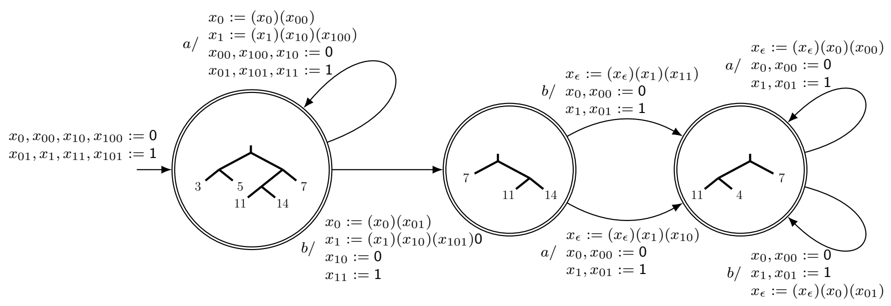




$$\begin{array}{rcl}
x_\epsilon & := & (x_\epsilon)(x_1)(x_{11}) \\
x_0, x_{00} & := & 0 \\
x_{01}, x_1 & := & 1
\end{array}$$

Had we stepped the tree on a b, the result would be the same. Stepping further from this tree always results in the same tree.

This forms the basis of our determinization algorithm: there are three distinct possibilities for the path tree to be organized, so the determinized automaton will have three states. To encode the register updates we associate a set of actions to each edge. The determinized version of the above FST looks as follows, where the path tree represented by the new states have been made explicit:


$x_0 := (x_0)(x_{00})$
$x_1 := (x_1)(x_{10})(x_{100})$
$a/$
$x_{00}, x_{100}, x_{10} := 0$
$x_{01}, x_{101}, x_{11} := 1$
$x_0, x_{00}, x_{10}, x_{100} := 0$
$x_{01}, x_1, x_{11}, x_{101} := 1$
3
5
11
14
7
$x_0 := (x_0)(x_{01})$
$b/$
$x_1 := (x_1)(x_{10})(x_{101})0$
$x_{10} := 0$
$x_{11} := 1$
7
11
14
$x_\epsilon := (x_\epsilon)(x_1)(x_{11})$
$b/$
$x_0, x_{00} := 0$
$x_1, x_{01} := 1$
$x_\epsilon := (x_\epsilon)(x_1)(x_{10})$
$a/$
$x_0, x_{00} := 0$
$x_1, x_{01} := 1$
11
4
7
$x_\epsilon := (x_\epsilon)(x_0)(x_{00})$
$a/$
$x_0, x_{00} := 0$
$x_1, x_{01} := 1$
$x_0, x_{00} := 0$
$b/$
$x_1, x_{01} := 1$
$x_\epsilon := (x_\epsilon)(x_0)(x_{01})$

3.9.1 Streaming String Transducers

This type of automaton is very similar to a class of automata known in the literature as *streaming string transducers* (Definition 7.11):

Definition (Streaming string transducers). A deterministic *streaming string transducer* (SST) over alphabets Σ and Γ is a structure

$$(Q, \Sigma, \Gamma, X, q^{\mathsf{in}}, F, \delta^1, \delta^2),$$

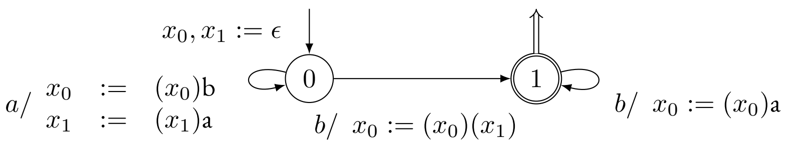
where

- Q is is a finite set of *states*;
- X is a finite set of *register variables*;
- $q^{\mathsf{in}} \in Q$ is the *initial state*;
- $F\colon Q \to (\Gamma \cup X)^\star \cup \{\emptyset\}$ is a partial function mapping each *final state* $q \in \mathrm{dom}(F)$ to a word $F(q) \in (\Gamma \cup X)^\star$ such that for each q, each $x \in X$ occurs at most once in $F(q)$;
- $\delta^1\colon Q \times \Sigma \to Q$ is the *transition function*;
- $\delta^2\colon Q \times \Sigma \times X \to (\Gamma \cup X)^\star$ is the *register update* function such that for each $q \in Q$, $a \in \Sigma$ and $x \in X$, there is at most one $y \in X$ such that x occurs in $\delta^2(q, a, y)$.

A streaming string transducer is a transducer with a set of registers that may only be *moved*, not copied.

Example 3.23. The following is an SST with two registers, x_0 and x_1, that implements the transduction

$$\mathsf{a}^n\mathsf{b}\mathsf{b}^m \mapsto \mathsf{b}^n\mathsf{a}^n\mathsf{a}^m\ :$$


$x_0, x_1 := \epsilon$
$a/\ \begin{array}{lll} x_0 & := & (x_0)\mathsf{b} \\ x_1 & := & (x_1)\mathsf{a} \end{array}$
0
1
$b/\ x_0 := (x_0)(x_1)$
$b/\ x_0 := (x_0)\mathsf{a}$

where the double arrow going out from state 1 indicates the final output (in this case nothing).

Note that the streaming string transducers that will be produced by the procedure sketched above form a subset of all SSTs. Because every register comes from an edge in a path tree, there is an ordering on registers. Due to the way path trees develop, this ordering specifies that registers can only ever be appended to the register immediately "above," i.e., to the register on the parent edge. The register x_ϵ is special, in that it corresponds to "output" in the streaming parsing algorithm of Section 3.8: appending bits to that register corresponds to lengthening the trunk of the path tree, and therefore this "append" action can be implemented by simply outputting the bits. An implementation strategy emerges: the Thompson FSTs can be determinized to SSTs, and these SSTs can easily be compiled into executable code. The registers of the SSTs can be implemented by variables in generated code.

3.10 The Kleenex Language (POPL'16)

The Thompson FSTs allow us to specify regular expression parsers in a straight-forward way. Transducers, and SSTs, are more general: there is no reason to limit ourselves to only storing bit-codes in the registers, as Example 3.23 also hints at.

Instead, we wish to be able to specify arbitrary rational functions—i.e., transductions from one regular language to another. Furthermore, since the underlying algorithm is based on the optimally streaming parsing algorithm presented earlier, it should be implemented in a streaming way, allowing for high throughput on large inputs.

A canonical example that we shall come back to is that of *syntax highlighting*: given as input a program text, emit the same program text but with color commands embedded.

We designed the language *Kleenex* as a surface language for specifying transductions [66] and implemented a compiler for it.[3] Roughly, the compiler works this way:

1. it translates Kleenex programs to FSTs;
2. those FSTs are determinized into SSTs;
3. the SSTs are represented in an intermediate language capable of expressing the register updates;
4. the intermediate language is transcribed to C and run through a C compiler (GCC [137] or Clang [138]).

3.10.1 Kleenex Syntax

Below, a simplified version of the Kleenex syntax is shown. The current version of Kleenex also supports user-specified actions, but we shall omit them here. They are implemented and described by Søholm and Tørholm in their Master's thesis [131].

Definition 3.31 (Kleenex syntax). A *Kleenex program* is a list of declarations of the format

$$N{:=}t$$

where N ranges over a set of identifiers, s is a an output word, and t is:

$$\begin{aligned} t ::= {} & \mathtt{1} \mid N \mid \mathtt{/}E\mathtt{/} \mid \mathtt{\sim}t \mid \mathtt{"}s\mathtt{"} \mid t_0 \,\mathtt{|}\, t_1 \\ & \mid t_0 t_1 \mid t\mathtt{*} \mid t\mathtt{+} \mid t\mathtt{?} \\ & \mid t\mathtt{\{}n\mathtt{\}} \mid t\mathtt{\{}n\mathtt{,\}} \mid t\mathtt{\{,}m\mathtt{\}} \mid t\mathtt{\{}n\mathtt{,}m\mathtt{\}} \end{aligned}$$

[3] https://github.com/diku-kmc/repg

The entry point of a Kleenex program is always the identifier named `main`, unless one uses the special "pipeline pragma." This is further discussed in Chapter 7.

3.10.2 Kleenex Semantics

In Chapter 7, we give the semantics of Kleenex programs as compositions of two FSTs. For a Kleenex program p, we construct a Thompson-style FST that we call the *oracle*, $\mathcal{F}_p^{\mathsf{C}}$, that outputs the greedy leftmost bit-coded parse tree, implementing the function $[\![\mathcal{F}_p^{\mathsf{C}}]\!]_{\leq}$. Then an *action machine* $[\![\mathcal{F}_p^{\mathsf{A}}]\!]$ that transduces bits to output words is constructed. Composing these two FSTs gives us the semantics of a Kleenex program (Definition 7.7):

Definition (Kleenex semantics). Let p be a Kleenex program and let $\mathcal{F}_p^{\mathsf{C}}$ and $\mathcal{F}_p^{\mathsf{A}}$ be the oracle and action machine. The program p denotes a partial function $[\![p]\!] : \Sigma^\star \to \Gamma^\star \cup \{\emptyset\}$ given by

$$[\![p]\!] = [\![\mathcal{F}_p^{\mathsf{A}}]\!] \circ [\![\mathcal{F}_p^{\mathsf{C}}]\!]_{\leq}$$

The operations of the Kleenex parts can be described as the following parts:

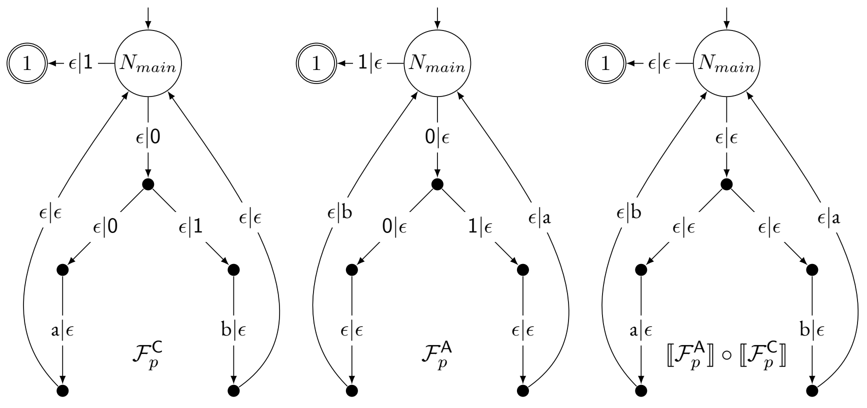
- Regular expressions, enclosed in `/ /`, act as primitives that copy matching input strings to output.
- Literal strings, enclosed in `" "`, act as unconditional output. A literal `"s"` acts as the transduction $\epsilon \mapsto s$.
- Terms prefixed with a `~T` are *suppressed*—any output that would otherwise have been produced by T are discarded.
- Finally, terms can be named and combined using the operators known from normal regular expressions—Kleene star, alternation, etc.

We specify the construction in detail in Chapter 7; here, we give some examples to establish intuition.

Example 3.24. Let p be the program that swaps as and bs in its input:

```
main := (~/a/ "b" | ~/b/ "a")*
```

Below, the oracle and the action machine for p are shown. Taking the leftmost path through the machine on the right corresponds to the composition in Definition 7.7:



Note that the syntax definition above does not rule out *ill-formed* programs. We consider programs to be ill-formed if they cannot be converted into finite state transducers. For example, this program satisfies the syntax requirements in Definition 3.31:

```
main := /a/ main /b/ | 1
```

This program *should* copy words in the non-regular language

$$\{\mathrm{a}^n\mathrm{b}^n \mid n \in \mathbb{N}\},$$

but this cannot be done with a finite state transducer due to the need for an unbounded counter. Kleenex programs are therefore restricted to be *inherently tail-recursive*. To ensure a clear distinction between valid and invalid Kleenex programs, the syntax is presented slightly differently in Chapter 7. There, a *core Kleenex* is defined, which is guaranteed to always correspond to right-regular grammars enriched with output symbols, and the rest of the constructors in the language are defined as syntactic sugar on top of this core.

3.10.3 An Example

We illustrate the pipeline of the compiler by studying an example program. Consider the transduction that translates words over $\{\mathrm{a}, \mathrm{b}\}$ as follows: All as are converted into bs if they are followed by two bs, which are deleted, and the first time a sequence of an uneven number of $2n + 1$ bs is encountered, the first $2n$ bs are removed and the last one kept. From that point on, the input is just copied to the output:

$$(\mathrm{a}^{k_i}(\mathrm{bb})^{n_i})^{m_i} \cdot s \mapsto b^{k_0} \cdot b^{k_1} \cdot \ldots \cdot s$$

where s is a word over $\{\mathrm{a}, \mathrm{b}\}$. The following are examples of input/output pairs:

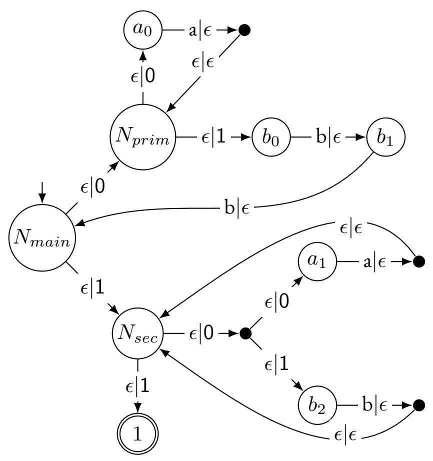
aaabb	$\mapsto$	bbb	aaabbaa	$\mapsto$	bbbaa
aaabbaab	$\mapsto$	bbbaab	abbabbbbbbbab	$\mapsto$	bbbab

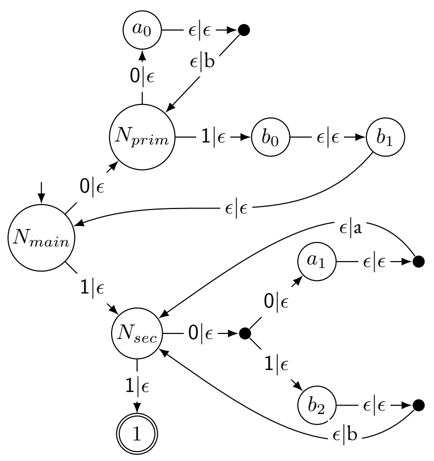
The Kleenex code below implements this transduction:

```
main := prim | sec
prim := (~/a/ "b")* ~/bb/ main
sec  := (/a/ | /b/)*
```

Kleenex implements the *greedy left-most* strategy: the left alternative will always be preferred to the right. Hence, this program can be thought of as trying to execute the `prim` branch but keeping the `sec` branch as a fall-back, in case an uneven number of bs are encountered.

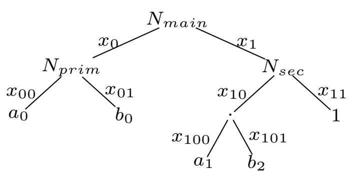
The following two FSTs are the oracle and action machine, respectively:





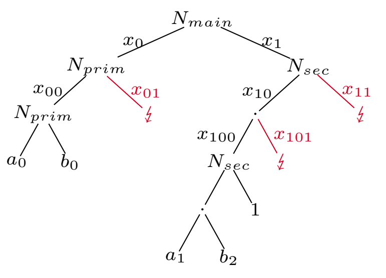
The oracle emits the bit-coded parse tree for the input string, and the right FST converts this bit-code to the input string.

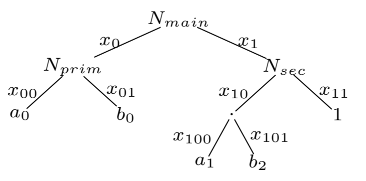
We illustrate the process of determinizing the oracle into an SST. The path tree after taking the ϵ-closure from the start state, N_{main} is:



$$\alpha_0 \begin{cases} x_\epsilon & := & \epsilon \\ x_0, x_{00}, x_{10}, x_{100} & := & 0 \\ x_{01}, x_1, x_{101}, x_{11} & := & 1 \end{cases}$$

Call this tree T_0. If an a is read, the resulting path tree is again T_0:

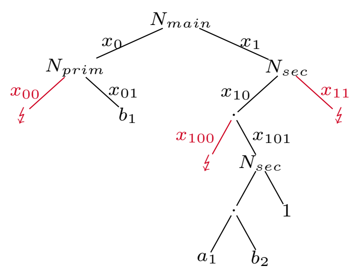


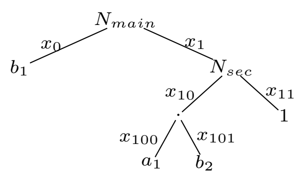


$$\alpha_1 \begin{cases} x_0 & := & (x_0)(x_{00}) \\ x_1 & := & (x_1)(x_{10})(x_{100}) \\ x_{00}, x_{100} & := & 0 \\ x_{01}, x_{101}, x_{11} & := & 1 \end{cases}$$

where the right tree is obtained by concatenating all non-splitting paths and turning it into a register concatenation; this is also reflected in the register update below it.

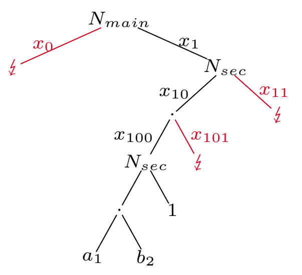
Reading a b from T_0 yields T_1:

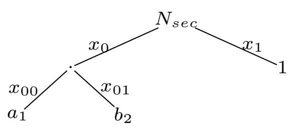




$$\alpha_2 \begin{cases} x_0 & := & (x_0)(x_{01}) \\ x_1 & := & (x_1)(x_{10})(x_{101}) \\ x_{10}, x_{100} & := & 0 \\ x_{101}, x_{11} & := & 1 \end{cases}$$

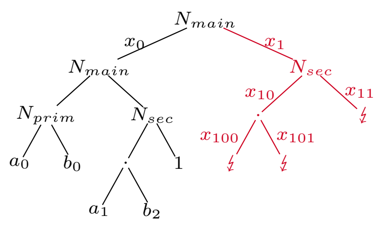
From T_1 we reach a new tree T_2 on an a, corresponding to the "mode change" that happens after an odd number of bs. This results in a trunk in the path tree, which is represented by the fact that the output buffer, x_ϵ is appended to. All other buffers are reset:

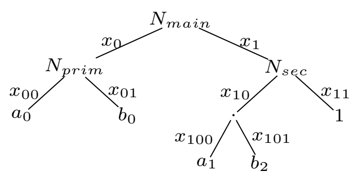




$$\alpha_3 \begin{cases} x_\epsilon & := & (x_\epsilon)(x_1)(x_{10})(x_{100}) \\ x_0, x_{00} & := & 0 \\ x_1, x_{101} & = & 1 \end{cases}$$

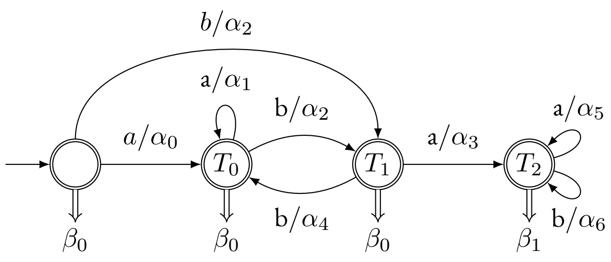
The path tree transitions back into T_0 from T_1 on reading a b. In this this case, the nodes that would have been reached in the right-hand side of the tree, from x_1 and down, are already reached by higher-prioritized paths, and they are therefore all removed from the tree. This results in the output buffer being appended to This also results in output being appended to and all other buffers being reset:





$$\alpha_4 \begin{cases} x_\epsilon & := & (x_\epsilon)(x_0) \\ x_0, x_{00} & := & 0 \\ x_{10}, x_{100} & := & 0 \\ x_1, x_{01} & := & 1 \\ x_{11}, x_{101} & := & 1 \end{cases}$$

We have now shown all the states in the SST version of the bit-code FST above: tree T_2 transitions back into tree T_2 on both a and b. Hence, the state T_2 in the SST corresponds to the situation after an odd number of bs have been read—from that point on the input is just copied. The SST obtained from the path trees above looks like:



where α_0 to α_4 are the register updates described above, and α_5, α_6 are the register updates resulting from stepping from T_2 to T_2. They have been omitted for brevity. The register updates β_0 and β_1 are the *final* register updates, and as all SST states are accepting because the accepting FST state 1 is in all the path trees, all SST states have an associated final update:

$$\begin{aligned}&\beta_0 \, \{x_\epsilon := (x_1)(x_{11})\\ &\beta_1 \, \{x_\epsilon := (x_1)\end{aligned}$$

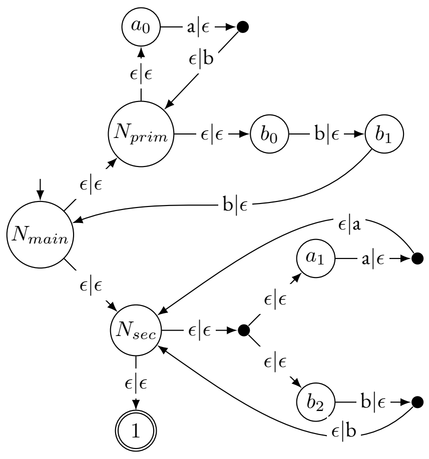
They ensure that at the end of the input string, the contents of the path from the root to the final state will be emitted.

The compiler converts this SST to a program in an intermediate language, and this program is directly translated to C code. Below, a snippet of the code produced for the oracle SST on the example program is shown. This part corresponds to the state named T_0 in the drawing: on reading an a the machine transitions back into T_0 (`l1_2`), and on reading a b it transitions to state T_1 (`l1_1`):

```
l1_2: if (!readnext(1, 1)) // End of input?
      {
         output(&buf_2);
         outputarray(const_1_3,8);
         goto accept1;
      }
      if (((avail >= 1) && ((next[0] == 'a') && 1))) // Transition #1
      {
         appendarray(&buf_2,const_1_1,16);
         appendarray(&buf_1,const_1_0,8);
         consume(1);
         goto l1_2; // Stay in this state.
      }
      if (((avail >= 1) && ((next[0] == 'b') && 1))) // Transition #2
      {
         appendarray(&buf_2,const_1_2,16);
         appendarray(&buf_1,const_1_3,8);
         consume(1);
         goto l1_1; // Go to state T_1
      }
      goto fail1;
```

Our example Kleenex program has the semantics of the composition of the oracle and action machine, as mentioned above. However, note that the output of the oracle is *prefix free*. This can be shown exactly like Lemma 3.1. The fact that the input to the action machine is a prefix free language means that all ambiguity has been resolved: it can never be the case that, when simulating an FST with a prefix free input language, the machine is in the final state but must continue, as this would violate the prefix-freeness of the input language. As a result, simulating the action machine is much simpler as there is never any need to maintain the non-determinism.

The version of Kleenex introduced here does not require that two separate FSTs be generated. Internally, the compiler can directly generate the composed FST, i.e., for our present Kleenex program:



The generated C code will look the same as the code shown above, but instead of bits in the registers it will maintain output words.

3.11 Further Work

In this chapter we gave an introduction and overview of the regular expression-based parsing techniques described in later chapters. There is a clear progression in insight, leading from an initial two-pass algorithm over a theoretically optimally streaming algorithm to an actual implementation that runs on commodity hardware and is competitive in performance when compared to other, similar tools (see Chapter 7). The organization of this chapter is an attempt to mirror this progression, serving the reader the important intuitions and insights that are ultimately combined in the Kleenex compiler.

There are many interesting things still to be done:

Multi-striding. Currently, input is read one character at a time and branched upon. The compiled programs could be made to use *multi-striding* techniques [112, 147] which would likely lead to increased throughput if modern CPU's data-parallel operations are explored.

FST simulation. One limitation of the current implementation of the compiler is that it forces all programs to be fully determinized. This causes the generated code size and the compilation time for some Kleenex programs to be in the tens of thousands of lines of C and in the order of half an hour, respectively. A classic technique when implementing regular expression *matching* engines, however, is to start constructing the DFA but falling back to an NFA simulation if the size of the DFA exceeds some threshold [139]. It would be a straight-forward extension to the compiler to have a FST/SST version of this mechanism: start building the SST but stop and just emit code to simulate the FST if the SST is too large. Another optimization from regular expression matchers that could address the problem of exploding code size would be to use memoization techniques.

Embedded computations. An interesting and potentially quite useful thing to explore is to use the oracle/action dichotomy to embed arbitrary computation into the action machine, e.g., by having a syntax for embedding C code in Kleenex. This is similar to the way Ragel is used [142], but with the important difference that the oracle machine of Kleenex takes care of all disambiguation. A programmer therefore does not need to worry about "undoing" work performed in an ultimately failing branch; the optimal streaming semantics makes sure that work is *only* performed if it is guaranteed that it is on a path that is a prefix of any successful paths.

4 Kleene Algebra and Extensions

In this chapter we give a brief introduction to the second part of the thesis: extensions to Kleene algebra. First some basic notions of Kleene algebra will be refreshed, and then two extensions will be introduced.

The extensions are not related to each other, and the two chapters reserved for the extensions that are based on published papers are somewhat self-contained. This chapter will therefore only give general introductions.

4.1 Kleene Algebra

The algebra of regular expressions is called *Kleene algebra*. One of the earliest authors to study this algebra was John Horton Conway in a 1971 monograph that has recently been republished in reprint [36]. Several authors have presented axiomatizations of Kleene algebra or variants thereof, including Salomaa [127], Grabmayer [62], and Kozen [88].

We present here Kozen's axiomatization [88]. The structure will be presented through a succession of algebraic structures that increase in complexity. This is inspired by the approach taken in Kozen's course notes on Kleene algebra [91].

Definition 4.1 (Semigroup). A *semigroup* is a structure $(S, \cdot)$ where S is a set and $\cdot$ is a binary operation that is associative, i.e., where the following holds for all $x, y, z \in S$:

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z. \tag{4.1}$$

Semigroups are objects with a minimal amount of structure. All the examples of semigroups we shall encounter here will also be monoids:

Definition 4.2 (Monoid). A *monoid* is a structure $(M, \cdot, 1)$ where $(M, \cdot)$ is a semigroup and 1 is the *neutral* element such that the following holds for any $x \in M$:

$$1 \cdot x = x \cdot 1 = x. \tag{4.2}$$

The standard example of a monoid is the *string monoid*:

Example 4.1. Let L be a language. The structure $(L, \cdot, \epsilon)$ is a monoid because concatenation of words is associative and concatenating ϵ on either side of a word does not change it.

The string monoid is not *commutative*, however:

Definition 4.3 (Commutative monoid). A *commutative monoid* is a monoid $(M, \cdot, 1)$ with a commutative operation:

$$x \cdot y = y \cdot x. \tag{4.3}$$

Example 4.2. Let $\mathbb{N}$ be the set of natural numbers. The structure $(\mathbb{N}, +, 0)$ is a commutative monoid because addition is both associative and commutative and adding 0 to any natural number results in the same natural number.

For monoids whose operator symbol is $\cdot$ and where there will not be any confusion we will use the established shorthand:

$$xy \overset{\text{def}}{=} x \cdot y.$$

Definition 4.4 (Semiring). A *semiring* is a structure $(S, +, \cdot, 0, 1)$ where:

- $(S, +, 0)$ is a commutative monoid;
- $(S, \cdot, 1)$ is a monoid;
- $\cdot$ distributes over $+$ on the left and right:

$$x \cdot (y + z) = x \cdot y + x \cdot z \tag{4.4}$$
$$(x + y) \cdot z = x \cdot z + y \cdot z; \tag{4.5}$$

- 0 is an *annihilator* for $\cdot$:

$$0 \cdot x = x \cdot 0 = 0. \tag{4.6}$$

Definition 4.5 (Idempotent semiring). An *idempotent semiring* is a semiring $(S, +, \cdot, 0, 1)$ where, for all $x \in S$,

$$x + x = x. \tag{4.7}$$

Idempotent semirings have a partial order $\leq$ defined as:

$$x \leq y \iff x + y = y. \tag{4.8}$$

With this, the Kleene star * can be introduced to an idempotent semiring, yielding a *Kleene algebra*:

Definition 4.6 (Kleene algebra). A *Kleene algebra* is a structure $(K, +, \cdot, ^\star, 0, 1)$ where

$$(K, +, \cdot, 0, 1)$$

is an idempotent semiring and for any $a, b, x, y \in K$ the unary operation * satisfies:

$$1 + x \cdot x^\star \leq x^\star \tag{4.9}$$
$$1 + x^\star \cdot x \leq x^\star \tag{4.10}$$
$$b + a \cdot x \leq x \implies a^\star \cdot b \leq x \tag{4.11}$$
$$b + x \cdot a \leq x \implies b \cdot a^\star \leq x. \tag{4.12}$$

Proposition 4.1. *In a Kleene algebra, axioms 4.11 and 4.12 are equivalent to the following two alternative axioms [88, 122]:*

$$a \cdot x \leq x \iff a^\star \cdot x \leq x \tag{4.13}$$
$$x \cdot a \leq x \iff x \cdot a^\star \leq x. \tag{4.14}$$

We shall use the alternative versions interchangeably with Axioms 4.11 and 4.12 when appropriate.

Proof. We show that Axioms 4.11 and 4.13 are equivalent; the other two are shown symmetrically.

Assume

$$b + a \cdot x \leq x \implies a^\star \cdot b \leq x.$$

This means that, in particular,

$$b + a \cdot b \leq b \implies a^\star \cdot b \leq b.$$

Because $x + y \leq z \iff x + y + z = z$ this means that both $x \leq z$ and $y \leq z$, so we have that

$$b + a \cdot b \leq b \implies a \cdot b \leq b \implies a^\star \cdot b \leq b,$$

which proves 4.11 $\implies$ 4.13.

For the other direction, assume

$$a \cdot x \leq x \implies a^\star \cdot x \leq x.$$

In particular,

$$b + a \cdot x \leq x \implies a \cdot x \leq x \implies a^\star \cdot x \leq x,$$

but

$$b + a \cdot x \leq x \implies b \leq x,$$

so therefore

$$b + a \cdot x \leq x \implies a^\star \cdot b \leq b,$$

which proves the other way: 4.13 $\implies$ 4.11. $\square$

Hence, a Kleene algebra K satisfies the following equations and implications:

$$\begin{array}{rcl}
x \cdot (y \cdot z) & = & (x \cdot y) \cdot z \\
1 \cdot x = x \cdot 1 & = & x \\
x + y & = & y + x \\
0 + x & = & x \\
x + x & = & x \\
x \cdot (y + z) & = & x \cdot y + x \cdot z \\
(x + y) \cdot z & = & x \cdot z + y \cdot z \\
0 \cdot x = x \cdot 0 & = & 0 \\
1 + x \cdot x^\star & \leq & x^\star \\
1 + x^\star \cdot x & \leq & x^\star \\
a \cdot x \leq x & \implies & a^\star \cdot x \leq x \\
x \cdot a \leq x & \implies & x \cdot a^\star \leq x.
\end{array}$$

Proposition 4.2 (Monotonicity [88]). *In any Kleene algebra K, the operators $+$, $\cdot$, and * are* monotone *with respect to $\leq$. That is, for $x, y, z \in K$:*

$$\begin{aligned} x \leq y &\implies x + z \leq y + z \\ x \leq y &\implies z + x \leq z + y \\ x \leq y &\implies xz \leq yz \\ x \leq y &\implies zx \leq zy \\ x \leq y &\implies x^\star \leq y^\star. \end{aligned} \tag{4.15}$$

Proof. We show implication 4.15: Assume that $x \leq y$. By the monotonicity of $+$ and $\cdot$ and by the assumption we have

$$1 + xy^\star \leq 1 + yy^\star \leq y^\star.$$

By Axiom 4.11 this means that

$$x^\star \leq y^\star. \qquad \square$$

The two axioms that state that $x^\star$ is an upper bound on the "unfoldings" $1 + xx^\star$ and $1 + x^\star x$ may be strengthened to equalities:

Proposition 4.3 ([88, Proposition 2.2]). *In any Kleene algebra, the following holds:*

$$1 + x \cdot x^\star = x^\star \tag{4.16}$$
$$1 + x^\star \cdot x = x^\star \tag{4.17}$$

The three rules known as *bisimulation*, *sliding*, and *denesting* are used in many proofs in Kleene algebra:

Proposition 4.4 ([88]). *The following all hold in any Kleene algebra:*

1. *the* bisimulation rule*: $ax = xb \implies a^\star x = xb^\star$,*
2. *the* sliding rule*: $(cd)^\star c = c(dc)^\star$,*
3. *the* denesting rule*: $(x + y)^\star = x^\star(yx^\star)^\star$.*

Proof. 1. Assume $ax \leq xb$. Then also, by monotonicity

$$axb^\star \leq xbb^\star. \tag{4.18}$$

Axiom 4.9 together with distributivity and monotonicity states that

$$x + xbb^\star = x(1 + bb^\star) \leq xb^\star.$$

Combining this with 4.18 gives us

$$x + axb^\star \leq xb + xbb^\star \leq xb^\star,$$

which by Axiom 4.11 gives

$$a^\star x \leq xb^\star.$$

The other direction is proved symmetrically, which concludes the proof.

2. This is an instance of the bisimulation rule: set $a = cd$, $b = dc$, and $x = c$. Then the premise of the bisimulation rule becomes $(cd)c = c(dc)$, and an application of the rule yields:
$$(cd)^\star c = c(dc)^\star.$$

3. We first show $(x + y)^\star \leq x^\star(yx^\star)^\star$. The following inequalities hold:
$$\begin{aligned} 1 &\leq x^\star(yx^\star)^\star \\ xx^\star(yx^\star)^\star &\leq x^\star(yx^\star)^\star \\ yx^\star(yx^\star)^\star &\leq (yx^\star)^\star \leq x^\star(yx^\star)^\star, \end{aligned}$$

   where the second and third inequalities are a consequence of the fact that $xx^\star \leq x^\star$. We then have
$$1 + (x + y)x^\star(yx^\star)^\star \leq 1 + ax^\star(yx^\star)^\star + bx^\star(yx^\star)^\star \leq x^\star(yx^\star)^\star,$$

   where the last equality is a consequence of the above three equalities and monotonicity. Applying axiom 4.11 now gives us:
$$(x + y)^\star \leq x^\star(yx^\star)^\star.$$

   The other direction $x^\star(yx^\star)^\star \leq (x+y)^\star$ follows by monotonicity. Since $x \leq x+y$ and $y \leq x + y$:
$$\begin{aligned} x^\star(yx^\star)^\star &\leq (x + y)^\star((x + y)(x + y)^\star)^\star \\ &\leq (x + y)^\star((x + y)^\star)^\star \\ &\leq ((x + y)^\star)^\star \\ &= (x + y)^\star. \end{aligned}$$

   In the last inequality we have used the fact $(x^\star)^\star = x^\star$. □

Example 4.3. Let L be a language over Σ, $L \subseteq \Sigma^\star$, that is closed under union and concatenation. Then the structure
$$(L, \cup, \cdot, \emptyset, \{\epsilon\})$$
forms an idempotent semiring—the properties in Definition 4.5 obviously hold true for this choice of operators and elements. Recall that the *Kleene asterate* (Definition 2.5) of L is the union of exponentiating L. Extending the structure with the Kleene asterate gives us a Kleene algebra
$$(L, \cup, \cdot, ^\star, \emptyset, \{\epsilon\}).$$

This example illustrates the connection with regular expressions: the languages L that satisfy these properties are exactly those languages for which there is some regular expression E where
$$L = \mathcal{L}[\![E]\!].$$

Example 4.4. A binary relation R on some set X is a set of pairs of elements from X:
$$R \subseteq X \times X.$$

Let $\mathbb{1} \overset{\text{def}}{=} \{(x, x) \mid x \in X\}$ be the identity relation that relates all elements of X to itself. Define the *composition* $\circ$ as the binary operation that combines to relations:

$$R_1 \circ R_2 \overset{\text{def}}{=} \{(x, z) \mid (x, y) \in R_1 \wedge (y, z) \in R_2\}\,.$$

This operations serves as the concatenation operator, so an exponentiation can be defined:

$$\begin{aligned} R^0 &\overset{\text{def}}{=} \mathbb{1} \\ R^{n+1} &\overset{\text{def}}{=} R \circ R^n. \end{aligned}$$

This causes the Kleene star to be the *reflexive transitive closure* of a relation:

$$R^\star = \bigcup_{n \geq 0} R^n.$$

The structure $(R, \cup, \circ, ^\star, \emptyset, \mathbb{1})$ forms a Kleene algebra.

Regular expressions are terms that can be used as syntax to denote elements in some Kleene algebra. The mapping from expressions to elements in a Kleene algebra is called an *interpretation*. The interpretation of regular expressions as regular languages from Chapter 3 is the canonical interpretation:

Definition 4.7 (Canonical interpretation). The *canonical interpretation* of a term E over an alphabet Σ in a Kleene algebra is:

$$\begin{aligned} L_\Sigma(x) &= \{x\} & L_\Sigma(e_0 + e_1) &= L_\Sigma(e_0) \cup L_\Sigma(e_1) \\ L_\Sigma(0) &= \emptyset & L_\Sigma(e_0 e_1) &= \{vw \mid v \in L_\Sigma(e_0),\ w \in L_\Sigma(e_1)\} \\ L_\Sigma(1) &= \{\epsilon\} & L_\Sigma(e^\star) &= \bigcup_{n \geq 0} L_\Sigma(e^n). \end{aligned}$$

Note that this definition is exactly the same as the $\mathcal{L}[\![\cdot]\!]$ used previously.

An important property of Kleene algebra is that it is complete with respect to equalities of regular languages:

Theorem 4.1 ([88, Theorem 5.5]). *Let E and F be two regular expressions over Σ representing the same regular languages $L_\Sigma(E) = L_\Sigma(F)$. Then $E = F$ is provable from the Kleene algebra axioms.*

Deciding equality of two regular languages is PSPACE-complete [132], so Kleene algebra is too:

Theorem 4.2. *The equational theory of Kleene algebra is PSPACE-complete.*

4.1.1 Star-Continuous Kleene algebra

Most "naturally occurring" Kleene algebras satisfy the property known as *star continuity*:

Definition 4.8 (Star-continuous Kleene algebra). A Kleene algebra is *star-continuous* if the following holds:

$$xy^\star z = \sup_{n \geq 0} xy^n z. \tag{4.19}$$

Here, sup refers the the least upper bound with respect to the idempotent semiring order 4.8.

Both the language and the relation examples above are star-continuous Kleene algebras, which can easily be seen by how the * operation is defined on both of them. Any star-continuous idempotent semiring is a Kleene algebra and therefore also a star-continuous Kleene algebra [91]. The other way does not hold: there exists Kleene algebras that are not star-continuous, although they are somewhat artificial.

Example 4.5. We repeat a standard example of a non–star-continuous Kleene algebra from Kozen's notes [91]. Write ω^2 for the set of *ordered pairs* of natural numbers. The elements of ω^2 are ordered lexicographically, such that, for example,

$$(4,6) < (5,6) \quad \text{and} \quad (1,2) < (1,3).$$

Add the two special elements $\bot$ and $\top$ to ω^2 and let them be the minimum and maximum element of the order on $\omega^2 \cup \{\bot, \top\}$, respectively. Let the $+$ operation be the supremum with respect to this order, so we have

$$\bot + x = x + \bot = x,$$

i.e., $\bot$ serves as the "zero-element" for addition. Let $\cdot$ be defined as follows:

$$\begin{array}{rcllll} x \cdot \bot & = & \bot \cdot x & = & \bot & \\ x \cdot \top & = & \top \cdot x & = & \top & (x \neq \bot) \\ (a,b) \cdot (c,d) & = & (a+c, b+d). & & & \end{array}$$

Hence, $(0,0)$ is the "one-element" for multiplication:

$$(0,0) \cdot x = x \cdot (0,0) = x.$$

If we now define the Kleene star as:

$$a^\star = \begin{cases} (0,0), & \text{if } a = \bot \text{ or } a = (0,0) \\ \top, & \text{otherwise,} \end{cases}$$

we get a Kleene algebra $(\omega^2 \cup \{\bot, \top\}, +, \cdot, ^\star, \bot, (0,0))$. This is easily verified; most cases follow directly from the associativity of addition on natural numbers, the supremum operation, or by definition. We illustrate two cases for *. First, the * is an upper bound on its unfolding (Axiom 4.9):

$$(0,0) + x \cdot x^\star = \left\{ \begin{array}{lcll} (0,0) + \bot \cdot (0,0) & = & (0,0), & x = \bot \\ (0,0) + (0,0) \cdot (0,0) & = & (0,0), & x = (0,0) \\ (0,0) + x \cdot \top & = & \top, & \text{otherwise} \end{array} \right\} = x^\star$$

Second, the * is the *least* of such bounds (Axiom 4.13). Assume that $a \cdot x \leq x$. We have

$$a^\star \cdot x = \begin{cases} (0,0) \cdot x = x, & a = \bot \\ (0,0) \cdot x = x, & a = (0,0) \\ \top \cdot x = \top, & \text{otherwise.} \end{cases}$$

In the first two cases we are done, as $a^\star \cdot x \leq x$ trivially. In the third case, it must be the case that $a = (c,d) > (0,0)$, however, this means that

$$(c,d)^\star \cdot x = \begin{cases} \top \cdot \bot = \bot, & x = \bot \\ \top \cdot \top = \top, & x = \top \\ \top \cdot (e,f) = \top, & \text{otherwise } x = (e,f). \end{cases}$$

In the first two cases, $(c,d)^\star \cdot x \le x$ and we are done. The third case cannot happen: if $x = (e,f)$ then

$$a \cdot x = (c,d) \cdot (e,f) = (c+e, d+f) > (e,f) = x,$$

which contradicts the assumption that $a \cdot x \le x$. The other two cases for * follow symmetrically.

This is not a star-continuous Kleene algebra, as Axiom 4.19 is not satisfied:

$$\sum_n (0,1)^n = \sum_n (0,n) = \sup_n (0,n) = (1,0) \neq (0,1)^\star.$$

4.2 Chomsky Algebra (FICS'13 / FI)

Kleene algebra is "the algebra of regular languages." In this section we briefly outline an extension to Kleene algebra that can be thought of as the analogue for *context-free languages* [77, 89]. This structure is dubbed *Chomsky algebra*, after the namesake of the hierarchy in which the context-free languages are the next rung on the ladder from regular languages [28, 29, 77].

The points outlined in this section are presented in greater detail in Chapter 8. Here we will only sketch the main ideas and motivations.

To provide some intuition for the extension, recall that, because, equations in Kleene algebra define regular languages, any language that can be specified with a regular expression can also be specified as the language *generated* by a *right-linear* grammar. For example, the language $L_\Sigma(E)$ where

$$E = (a+b)^\star,$$

can also be specified as the language generated by the grammar

$$E \longrightarrow \mathrm{a}E \qquad E \longrightarrow \mathrm{b}E \qquad E \longrightarrow \epsilon.$$

Any regular language can be specified as a *right-linear* grammar—a grammar where all nonterminals are in the *last* position if they occur in a production rule [77, Theorem 9.2]. If that restriction is lifted, one can write grammars that recognize the context-free languages [77].

The algebraic restriction corresponding to the right-linearity of the grammar above is the fact that in regular expressions—terms in a Kleene algebra—the only way to specify recursion is with the *. Since

$$x^\star = 1 + xx^\star,$$

this is also tail-recursion. A natural thing to consider is then what happens this restriction is lifted.

Consider the grammar

$$S \longrightarrow \mathrm{a}S\mathrm{b} \qquad S \longrightarrow \epsilon.$$

This recognizes the context-free language $\{\mathrm{a}^n\mathrm{b}^n \mid n \in \mathbb{N}\}$. By *naming* elements and referring to them one can express recursion in non-tail positions. We can think of the language as the minimal, nonempty solution to the equation

$$S \ge aSb + 1 \tag{4.20}$$

where a, b, and 1 are interpreted under the canonical interpretation (extended appropriately). This is an example of a *system of polynomial inequalities* with one inequality. Another system is

$$S \ge [S] + SS + 1 \tag{4.21}$$

that corresponds to the grammar

$$S \longrightarrow [S] \qquad\qquad S \longrightarrow SS \qquad\qquad S \longrightarrow \epsilon$$

for the language of balanced brackets. Yet another example ([89, Example 19.2]) is the pair of the grammar and polynomial inequality for the language of palindromes over $\{a, b\}$:

$$S \longrightarrow aSa \qquad S \longrightarrow bSb \qquad S \longrightarrow a \qquad S \longrightarrow b \qquad S \longrightarrow \epsilon,$$

and

$$S \geq aSa + bSb + a + b + 1. \tag{4.22}$$

We shall formulate polynomials by taking a *coproduct* of two idempotent semirings:

Definition 4.9. The *coproduct* of two idempotent semirings

$$(A, +_A, \cdot_A, 0_A, 1_A) \qquad\qquad (B, +_B, \cdot_B, 0_B, 1_B)$$

where $+_A = +_B$, $\cdot_A = \cdot_B$, $0_A = 0_B$, and $1_A = 1_B$ is the idempotent semiring

$$(A \oplus B, +, \cdot, 0, 1)$$

that consists of elements from either A or B and where $+ = +_A = +_B$, $\cdot = \cdot_A = \cdot_B$, $0 = 0_A = 0_B$, and $1 = 1_A = 1_B$.

Definition 4.10. Let X be a finite, non-empty set. The *free idempotent semiring* generated by X is the semiring that has as primitive objects all $x_i \in X$ and where all the equalities of Definition 4.5, and no other, are satisfied.

Example 4.6. Let $X = \{\alpha, \beta, \gamma\}$. The free idempotent semiring is $S = (\{\alpha, \beta, \gamma\}, +, \cdot, 0, 1)$ where, for example,

$$\alpha + \alpha = \alpha \qquad \alpha + \beta = \beta + \alpha \qquad \alpha \cdot 1 = 1 \cdot \alpha \qquad \alpha \cdot (\gamma + \beta) = \alpha \cdot \beta + \alpha \cdot \gamma,$$

but $\alpha \cdot \beta \neq \beta \cdot \alpha$.

A *polynomial* over an idempotent semiring C with *coefficients* in a set X is an element in $C[X]$, where $C[X]$ is the coproduct of C and the free idempotent semiring generated by X. This definition corresponds to the understanding of polynomials from elementary school; some scalar values (elements in C) and some variables (elements in X) blended together with some operators.

Example 4.7. The right-hand side of the inequality 4.21 is a polynomium over the free idempotent semiring B generated by {], [} and the coefficients $X = \{S\}$: $B[X]$.

The inequalities 4.20 and 4.22 are a polynomials over the free idempotent semiring generated by $\{a, b\}$ and the same set of coefficients.

In general, a system of polynomial equations with n inequalities is a set of inequalities

$$x_1 \geq p_1, x_2 \geq p_2, \ldots, x_n \geq p_n$$

where the x_i range over some set of names X and the p_i are polynomials in some idempotent semiring with coefficients in this X. If such systems always have a least solution, that is, a set of mappings $x_i \mapsto c_i$ where $c_i \in C$ such that for any other mapping that is a soltion $x_i \mapsto c_i'$, $c_i \leq c_i'$, C is called *algebraically closed*. When that is the case, we call it a *Chomsky algebra*:

Definition 4.11 (Chomsky algebra). A *Chomsky algebra* is an algebraically closed idempotent semirings.

Just as the regular expressions are terms denoting elements in a Kleene algebra, we can write expressions that denote elements in a Chomsky algebra. These are *μ-expressions*, and have been studied before [49, 50, 104]s:

Definition 4.12. Let X be a set of variables. The μ-expressions over X, $\mathsf{T}X$, are expressions formed are described by the grammar:

$$T ::= 0 \mid 1 \mid x \mid T_1 + T_2 \mid T_1 \cdot T_2 \mid \mu x.T,$$

where $x \in X$.

The definition of μ-expressions is equivalent to the definition of regular expressions, except that the Kleene star has been replaced by the binding operator μ. With this comes all the mechanics of scoping, substitution, α-conversion, etc. [14]. Substituting a term t' for a variable x in another term t is written $[x/t']t$

Regular expressions can be interpreted as objects in a Kleene algebra, and an analogous interpretation of μ-terms in a Chomsky algebra can be defined:

Definition 4.13 (Interpretation of μ-terms). An *interpretation* of μ-terms over a Chomsky algebra C, $\sigma\colon \mathsf{T}X \to C$, is a homomorphism with respect to $+$ and $\cdot$:

$$\begin{array}{llll} \sigma(0) & = 0 & \sigma(a+b) & = \sigma(a)+\sigma(b) \\ \sigma(1) & = 1 & \sigma(a \cdot b) & = \sigma(a)\cdot\sigma(b). \end{array}$$

The μ-operator is interpreted as the *least* element a in C such that replacing the variable that has been bound by the operator with a is less than a itself:

$$\sigma(\mu x.t) = \text{the least } a \in C \text{ such that } \sigma[x/a](t) \leq a.$$

Similar to the role exponentiation to nth power plays for Kleene star in star-continuous Kleene algebras, we have a notation for the n-fold composition of a term $\mu x.t$ with itself:

$$0x.t = 0 \qquad\qquad (n+1)x.t = t[x/nx.t].$$

Example 4.8. Let $t = \mu x.axb + 1$.

$$\begin{array}{llcl} \text{0-fold:} & 0x.axb+1 & = & 0 \\ \text{1-fold:} & 1x.axb+1 & = & a(0x.axb+1)b+1 \\ & & = & a0b+1 = 1 \\ \text{3-fold:} & 3x.axb+1 & = & a(2x.axb+1)b+1 \\ & & = & a(a(1x.axb+1)b+1)b+1 \\ & & = & a(a1b+1)b+1 \\ & & = & aabb+ab+1 \\ \text{4-fold:} & 4x.axb+1 & = & a(3x.axb+1)b+1 \\ & & = & aaabbb+aabb+ab+1. \end{array}$$

Note that if the μ-expression is $\mu x.1 + ax$ the $(n+1)$-fold composition is the same as the sum of the 0th to nth power of a:

$$\begin{array}{lrcl}
\text{0-fold:} & 0x.1+ax & = & 0 \\
\text{1-fold:} & 1x.1+ax & = & 1+a(0x.1+ax) \\
 & & = & 1+a0=1 \\
\text{3-fold:} & 3x.1+ax & = & 1+a(2x.1+ax) \\
 & & = & 1+a(1+a(1x.1+ax)) \\
 & & = & 1+a(1+a1) \\
 & & = & 1+a+aa \\
\text{4-fold:} & 4x.1+ax & = & 1+a(3x.1+ax) \\
 & & = & 1+a(1+a+aa) \\
 & & = & 1+a+aa+aaa.
\end{array}$$

We can now generalize star-continuity to *μ-continuity*:

Definition 4.14. A Chomsky algebra where

$$a(\mu x.t)b = \sup_{n\geq 0} a(nx.t)b \tag{4.23}$$

is a μ-continuous Chomsky algebra.

The *μ-continuous Chomsky algebras* are the analogue to the star-continous Kleene algebras. Just as there are Kleene algebras that are not star-continuous, there are Chomsky algebras that are not μ-continuous. We give an example in Chapter 8.

There is a canonical interpretation of μ-terms as the context-free languages which we repeat from Chapter 8:

Definition 4.15. The *canonical interpretation* of μ-terms over variables X is:

$$\begin{array}{ll}
L_X(x) = \{x\} & L_X(t+u) = L_X(t) \cup L_X(u) \\
L_X(0) = \emptyset & L_X(tu) = \{xy \mid x \in L_X(t),\ y \in L_X(u)\} \\
L_X(1) = \{\varepsilon\} & L_X(\mu x.t) = \bigcup_{n\geq 0} L_X(nx.t).
\end{array}$$

The canonical interpretation of μ-terms are the context-free languages over the variables in the term.

Example 4.9. Inequalities 4.20, 4.21, and 4.22 can be written as the μ-terms over setse $\{a, b, S\}$ and $\{], [, S\}$:

$$t_0 = \mu S.aSb + 1 \quad t_1 = \mu S.[S] + SS + 1 \quad t_3 = \mu S.aSa + bSb + a + b + 1.$$

The canonical interpretations of t_0, t_1, and t_2 are the context-free languages generated by the corresponding grammars above.

The main result of [63] that will be discussed in more detail in Chapter 8 is an analogue to Theorem 4.1: an equation $s = t$ holds in a μ-continuous Chomsky algebra if and only if the canonical interpretations of s and t as context-free languages are equivalent:

$$s = t \iff L_X(s) = L_X(t).$$

Note that this is the set of equations provable from the axioms of idempotent semirings, as in Kleene algebras, plus the axiom of μ-continuity above (4.23). This is an *infinitary axiom*: it is equivalent to the formulas

$$a(nx.t)b \leq a(\mu x.t)b, \quad n \geq 0$$
$$\left(\bigwedge_{n\geq 0} (a(nx.t)b \leq w) \right) \implies a(\mu x.t)b \leq w,$$

where the validity of the latter requires one to establish infinitely many premises. This is not surprising in light of the fact that equality of context-free languages is undecidable [89]!

4.3 Kleene Algebra with Tests

Before presenting the second paper that forms the basis for this part of the thesis we need some background on the extension to Kleene algebra known as *Kleene algebra with tests*.

Definition 4.16 (Boolean algebra). A *Boolean algebra* is a structure

$$(B, +, \cdot, ^{-}, 0, 1)$$

that satisfies the following:

- $(B, \cdot, 1)$ and $(B, +, 0)$ are both commutative monoids,
- $(B, +, \cdot, 0, 1)$ is a idempotent semiring,
- the following additional equalities hold:

$$x \cdot (x + y) = x \tag{4.24}$$
$$x + (x \cdot y) = x \tag{4.25}$$
$$x + (y \cdot z) = (x + y) \cdot (x + z) \tag{4.26}$$
$$x + 1 = 1 \tag{4.27}$$
$$x \cdot \overline{x} = 0 \tag{4.28}$$
$$x + \overline{x} = 1 \tag{4.29}$$

Note that, because of equation 4.27, in any Boolean algebra the following inequality holds for any x:

$$x \leq 1.$$

Example 4.10. The power set of a set X, 2^X, is a Boolean algebra:

$$(2^x, \cup, \cap, ^{-}, \emptyset, X),$$

where $\overline{Y} = 2^X \setminus Y$.

Example 4.11. Call the set with one element $\mathbb{1} = \{*\}$. The power set of $\mathbb{1}$ is $2^{\mathbb{1}} = \mathbb{2} = \{\emptyset, \{*\}\}$, so this forms the Boolean algebra with the familiar truth values "false" and "true."

Proposition 4.5. *Any Boolean algebra* $(B, +, \cdot, ^{-}, 0, 1)$ *is also a Kleene algebra* $(B, +, \cdot, ^{\star}, 0, 1)$ *where the* $+$ *and* $\cdot$ *operations are the same and the* * *is defined as*

$$x^{\star} = 1.$$

Proof. All axioms pertaining to $+$ and $\cdot$ come for free from the Boolean algebra, so only the four axioms about * need to be checked. We demonstrate one pair. Axiom 4.9:

$$1 + xx^\star = 1 + x1 = 1 + x = 1 = x^\star.$$

Axiom 4.13:

$$ax \leq x \implies a^\star x \leq x \iff 1x \leq x \iff x \leq x. \qquad \square$$

A Kleene algebra can be extended with a set of special *test symbols*. To reflect the different meaning of the symbols, we shall call the original symbols of the Kleene algebra *action symbols*. Intuitively, a "test" is something that has a truth value, so it should obey the laws of Boolean algebra. Furthermore, there must be an interplay between action symbols and test symbols to allow for desirable equations, e.g., that performing a test and then performing either of two actions is the same as either doing a test and one action, or doing the same test and the other action. This statement just means that the distributive law should be obeyed, as in a normal Kleene algebra.

The fact that any Boolean algebra is a Kleene algebra allows us to overload the operators so they can be used both in their role as Boolean algebra operators and in their role as Kleene algebra operators. Hence, we can defined a two-sorted algebra:

Definition 4.17 (Kleene algebra with tests [93, 94]). A *Kleene algebra with tests* (KAT) is a structure

$$(K, B, +, \cdot, ^\star, ^-, 0, 1)$$

where

- $(K, +, \cdot, ^\star, 0, 1)$ is a Kleene algebra,
- $(B, +, \cdot, ^-, 0, 1)$ is a Boolean algebra, and
- the test symbols are a subset of the action symbols: $B \subseteq K$.

Similar formalisms have been studied from early on [45, 82].

Expressions in KAT are normal regular expressions but extended with the test symbols and the $^-$ operator on these. KAT expressions with action symbols in Σ and test symbols in B are referred to as $\mathsf{Exp}_{\Sigma,B}$

Definition 4.18. An *interpretation* of KAT expressions $\mathsf{Exp}_{\Sigma,B}$ in a KAT K is a homomorphism $\sigma\colon \mathsf{Exp}_{\Sigma,B} \to K$:

$$\begin{array}{rcl@{\qquad}rcl}
\sigma(0) & = & 0 & \sigma(e+f) & = & \sigma(e) + \sigma(f) \\
\sigma(1) & = & 1 & \sigma(e \cdot f) & = & \sigma(e) \cdot \sigma(f) \\
\sigma(b) & = & t \in B & \sigma(\overline{b}) & = & \overline{t} \in B \\
\sigma(p) & = & x \in K & \sigma(e^\star) & = & \sigma(e)^\star,
\end{array}$$

where the interpretation of the Boolean test symbols b as t are elements of the Boolean algebra $(B, +, \cdot, ^-, 0, 1)$ and the interpretation of the primitive action symbol p is an element of the Kleene algebra $(K, +, \cdot, ^\star, 0, 1)$.

There is a canonical language interpretation of KAT expressions like the language interpretation of regular expressions. However, the languages of KAT expressions are a generalized version of the usual language, as they must accommodate the notion of tests. This is done by inserting an *atom* between each action symbol in the string. Atoms reflect the global configuration of all test symbols, and can be thought of as a valuation of all test symbols $b_0, \ldots, b_n$:

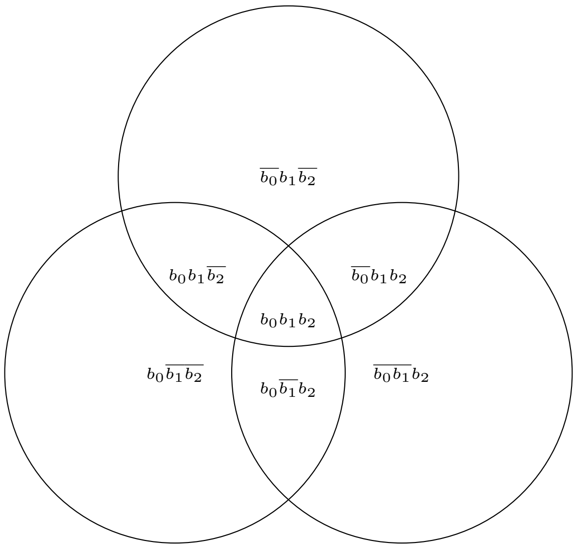
Definition 4.19 (Atoms). Let $B = \{b_0, \ldots, b_n\}$ be a set of test symbols. The *atoms* over B, At_B, are strings

$$c_0 \cdots c_n, \qquad c_i \in \{b_i, \overline{b_i}\}$$

that assigns a truth value to each test symbol.

We will usually refer to atoms with lower-case greek letters α, β, etc. When a test b is set in an atom α we write $\alpha \leq b$.

Example 4.12. Let $B = \{b_0, b_1, b_2\}$. The atoms At_B correspond to the cells in the Venn diagram:



Here, the $\leq$ corresponds to set inclusion $\subseteq$. It is easy to see that, e.g., $b_0\overline{b_1}b_2 \subseteq \overline{b_0b_1}b_2$, the atom with only b_2 set.

The guarded strings are the strings with atoms before each symbol. Kaplan used *K-expressions* and interpreted them as so-called *K-events* which he introduced in 1969 [82]. The K-expressions are closely similar to KAT expressions, and the K-events are what we now call guarded strings [82, 102]:

Definition 4.20 (Guarded strings). A *guarded string* over an action alphabet Σ and a test alphabet B is an element in $B \times (\Sigma \times B)^\star$:

$$\alpha_0 p_1 \alpha_1 p_2 \alpha_2 \cdots p_n \alpha_n,$$

where the α_i are atoms over B.

Guarded strings are thus strings that where all actions are coupled with a "certificate" that ensures the state of the tests after that action. The first atom can be thought of as the initial configuration of the tests.

Guarded strings can also be thought of as typed strings. Therefore, some care needs to be taken when combining strings, such that they "fit together."

Definition 4.21. The *fusion product* of two guarded strings $x\alpha$ and βy, where α and β are the last and initial atom, respectively, is the partial function defined by:

$$x\alpha \diamond \beta y \overset{\text{def}}{=} \begin{cases} x\alpha y & \text{if } \alpha = \beta \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Just as with normal string concatenation, we omit the $\diamond$ when the context allows it.

The fusion product can be lifted to operate on sets of guarded strings in the same way as the normal concatenation, and with it the exponentiation and Kleene asterate of sets of guarded strings are defined as usual. With these operations, the *regular languages* of guarded strings can be specified. The set of regular languages of guarded strings with action symbols Σ and test symbols B is called $\mathsf{Reg}_{\Sigma,B}$.

Definition 4.22. The *canonical interpretation* of KAT expressions $\mathsf{Exp}_{\Sigma,B}$ as regular languages of guarded strings is obtained by interpreting tests as the set of atoms wherein they hold and actions as the set of guarded strings consisting of that action:

$$G(p) \overset{\text{def}}{=} \{\alpha p \beta \mid \alpha, \beta \in \mathsf{At}_B\}$$
$$G(b) \overset{\text{def}}{=} \{\alpha \mid \alpha \in \mathsf{At}_B, \alpha \leq \beta\}\,.$$

They extend to the homomorphism $G\colon \mathsf{Exp}_{\Sigma,B} \to \mathsf{Reg}_{\Sigma,B}$:

$$\begin{array}{rcl@{\qquad}rcl} G(0) & = & \emptyset & G(e+f) & = & G(e) \cup G(f) \\ G(1) & = & \mathsf{At}_B & G(e \cdot f) & = & G(e) \diamond G(f) \\ G(\overline{b}) & = & \mathsf{At}_B \setminus G(b) & G(e^\star) & = & G(e)^\star. \end{array}$$

Using only the axioms of Kleene algebra, all equalities of regular languages of guarded strings can be shown:

Theorem 4.3 ([102, Theorem 8]). *Let E and F be two KAT expressions in $\mathsf{Exp}_{\Sigma,B}$. The canonical interpretation of E and F as regular languages of guarded strings are the same if and only of $E = F$ is provable from the axioms of Kleene algebra with tests.*

Even though Kleene algebra with test seems more complex than "normal" Kleene algebra, it is just as difficult to decide:

Theorem 4.4 ([34]). *The equational theory of Kleene algebra with tests is PSPACE-complete.*

4.3.1 A Folk Theorem—WHILE Programs

A standard demonstration of Kleene algebra with tests is to prove the "folk theorem" that every WHILE program can be simulated by a WHILE program with at most one WHILE loop [72, 93]. This program transformation can only be made if some additional state is introduced that can encode the structure of the deleted WHILE loops. One can use a "trick" to *copy* the value of tests. The term

$$bc + \overline{b}\overline{c}$$

is in effect a copy of the value in b into c.

This trick is used along with the introduction of fresh test symbols when needed to convert any WHILE program to the normal form with one WHILE loop. Intuitively, when removing WHILE loops, the control structure needs to be stored somewhere else, and this is the role that the additional variables serve. All WHILE programs can be translated into

KAT terms representing them, and it can then be shown with the axioms of KAT [93] that WHILE programs can be put into the normal form.

We will not give the entire proof here but instead illustrate the construction on a simple example program:

$$\begin{array}{l} \text{IF } a \text{ THEN WHILE } b \text{ DO } p \\ \qquad \text{ELSE WHILE } c \text{ DO } q \end{array} \tag{4.30}$$

The symbols a, b, and c are tests, and p and q are actions that may stand for other WHILE programs. This program can be encoded as the following KAT term:

$$a(bp)^\star\overline{b} + \overline{a}(cq)^\star\overline{c}.$$

To prove the folk theorem for this case, we must show that the program is equivalent to the following program in normal form:

$$\begin{array}{l} ae + \overline{ae}; \\ \text{WHILE } eb + \overline{e}c \text{ DO} \\ \qquad \text{IF } e \text{ THEN } p \text{ ELSE } q \end{array} \tag{4.31}$$

First, we add the extra variable e to the original program

$$\begin{array}{l} ae + \overline{ae}; \\ \text{IF } a \text{ THEN WHILE } b \text{ DO } p \\ \qquad \text{ELSE WHILE } c \text{ DO } q. \end{array} \tag{4.32}$$

Note that we assume that e commutes with the programs p and q. This is a way of specifying that neither p nor q can alter the value of e; it is entirely "fresh." The KAT term for this program is:

$$\begin{aligned} (ae + \overline{ae})(a(bp)^\star\overline{b} + \overline{a}(cq)^\star\overline{c}) &= aea(bp)^\star\overline{b} + ae\overline{a}(cq)^\star\overline{c} + \overline{ae}a(bp)^\star\overline{b} + \overline{ae}\overline{a}(cq)^\star\overline{c} \\ &= aea(bp)^\star\overline{b} + \overline{ae}\overline{a}(cq)^\star\overline{c} \\ &= ae(bp)^\star\overline{b} + \overline{ae}(cq)^\star\overline{c}. \end{aligned} \tag{4.33}$$

The KAT term for the normalized target program is

$$(ae + \overline{ae})((eb + \overline{e}c)(ep + \overline{e}q))^\star\overline{eb + \overline{e}c},$$

and we can rewrite it using the axioms of KAT as follows:

$$= (ae + \overline{ae})((eb + \overline{e}c)(ep + \overline{e}q))^\star(e\overline{b} + \overline{ec})$$

because $\overline{eb + \overline{e}c} = \overline{(\overline{e} + b)(e + c)} = \overline{\overline{e} + b} + \overline{e + c} = \overline{\overline{e}}\overline{b} + \overline{ec} = e\overline{b} + \overline{ec}$,

$$\begin{aligned} &= (ae + \overline{ae})(ebep + eb\overline{e}q + \overline{e}cep + \overline{e}c\overline{e}q)^\star(e\overline{b} + \overline{ec}) \\ &= (ae + \overline{ae})(ebp + \overline{e}cq)^\star(e\overline{b} + \overline{ec}) \\ &= ae(ebp + \overline{e}cq)^\star e\overline{b} + ae(ebp + \overline{e}cq)^\star\overline{ec} \\ &\quad + \overline{ae}(ebp + \overline{e}cq)^\star e\overline{b} + \overline{ae}(ebp + \overline{e}cq)^\star\overline{ec}, \end{aligned}$$

and, since $xy^\star = y^\star x$ if x and y commute [93, Lemma 2.3.2],

$$\begin{aligned}
&= aee(ebp + \overline{e}cq)^\star\overline{b} + ae\overline{e}(ebp + \overline{e}cq)^\star\overline{c} \\
&\quad + \overline{a}\overline{e}e(ebp + \overline{e}cq)^\star\overline{b} + \overline{a}\overline{e}\overline{e}(ebp + \overline{e}cq)^\star\overline{c} \\
&= ae(ebp + \overline{e}cq)^\star\overline{b} + \overline{a}\overline{e}(ebp + \overline{e}cq)^\star\overline{c}
\end{aligned}$$

and, since also $xy^\star = x(xy)^\star$ if x and y commute [93, Lemma 2.3.2],

$$\begin{aligned}
&= ae(eebp + e\overline{e}cq)^\star\overline{b} + \overline{a}\overline{e}(\overline{e}ebp + \overline{e}\overline{e}cq)^\star\overline{c} \\
&= ae(ebp)^\star\overline{b} + \overline{a}\overline{e}(\overline{e}cq)^\star\overline{c} \\
&= aee(bp)^\star\overline{b} + \overline{a}\overline{e}\overline{e}(cq)^\star\overline{c} \\
&= ae(bp)^\star\overline{b} + \overline{a}\overline{e}(cq)^\star\overline{c}
\end{aligned}$$

which is exactly equation 4.33 and hence the normalized program 4.31 with one WHILE loop is equivalent to the program 4.32, which again is equivalent to the original program 4.30, modulo the extra book keeping variables introduced.

4.4 KAT + B! (LICS'14)

The proofs of the folk theorem requires the introduction of auxiliary variables. Test symbols are immutable, but mutability is needed so the mutability is encoded by using "enough" test symbols—as discussed in Section 4.3.1.

Here we will present an extension to Kleene algebra with tests that introduces the notion of mutability into the algebra. The present section serves as a brief outline of the work, highlighting the main ideas and concepts. Details are in Chapter 9.

The mutable objects we are interested in must satisfy a certain set of equalities to capture our intuitive notion of "program variable." This is the role of the *B! algebra*:

Definition 4.23 (B! algebra). A *B! algebra* over a set of symbols B is a KAT with action symbols

$$\{b!, \overline{b}! \mid b \in B\}$$

and test symbols

$$\{b? \mid b \in B\},$$

satisfying all of the following additional equations:

$$b!b? = b! \tag{4.34}$$
$$b?b! = b? \tag{4.35}$$
$$b!\overline{b}! = \overline{b}! \tag{4.36}$$
$$t!b! = b!t!, \quad \text{if } b \neq \overline{t} \tag{4.37}$$
$$t!b? = b?t!, \quad \text{if } t \notin \{b, \overline{b}\}. \tag{4.38}$$

The $b!$ action symbols should be thought of as mutating the contents of some variable b: doing $b!$ sets b to "true," and doing $\overline{b}!$ sets it to "false." Setting a variable multiple times to the same value has no effect:

$$b! = b!b? = b!b?b! = b!b!, \tag{4.39}$$

where the equalities follow from Axioms 4.34, 4.35, and 4.34, respectively. Also, if a variable contains the value "true," it cannot contain the value "false":

$$b!\overline{b}? = b!b?\overline{b}? = b0 = 0, \tag{4.40}$$

where the first equality comes from Axiom 4.34 and the second from Axiom 4.28 of the Boolean test algebra.

The "?" symbol is not an operator but a way of syntactically discerning between tests and actions. As a B! algebra is a KAT, we have the atoms from Definition 9.2.1 over tests $\{b_0, \ldots, b_n\}$. We denote with $\alpha[b]$ the atom α if $\alpha \leq b$ and if $\alpha \leq \overline{b}$ the atom α with $\overline{b}$ replaced by b. Thus, the atom $\alpha[b]$ always has the property $\alpha[b] \leq b$.

The tests and actions can be extended to atoms:

$$\alpha? = c_0?c_1?\cdots c_n? \qquad \alpha! = c_0!c_1!\cdots c_n!$$

where $\alpha = c_0c_1\cdots c_n$ and each $c_i \in \{b_i, \overline{b_i}\}$.

Observe that each atom is an n-tuple of valuations of test symbols. Binary relations on such elements can be represented as n-by-n matrices. Theorem 9.1 ensures that the axioms of B! do not degenerate, i.e., that the *free B! algebra* F_n with n test symbols, in which no other equalities hold, does not collapse to a one-element algebra where everything is equivalent to everything. This is done by establishing that F_n is isomorphic to a KAT on binary relations on a 2^n-sized set. There are 2^n different atoms, so this property states that the free B! algebra encodes relations on atoms, i.e., on "full valuations" of test symbols.

The motivation behind extending KAT is to encode state and reason about it equationally. The B! algebra we use to encode state is also a KAT, and one can define the *commutative coproduct* of two KATs:

Definition. The *commutative coproduct* of two KATs

$$(K, B_K, +, \cdot, ^\star, ^-, 0, 1) \qquad\qquad (F, B_F, +, \cdot, ^\star, ^-, 0, 1)$$

is the coproduct of K and F, $K \oplus F$, in the sense of Definition 4.9 extended with * and $^-$, and satisfying the set of additional commutativity conditions:

$$D = \{i_K(p)i_F(s) = i_F(s)i_K(p) \mid p \in K, s \in F\},$$

where i_K and i_F are the canonical injections into $K \oplus F$ from K and F. This is equivalent to defining the commutative coproduct as the *quotient* $(K \oplus F)/D$.

The commutative coproduct is universal in the sense that for any commutative coproduct H, the injections from K and F into H are commutative (Lemma 9.1):

$$k(p)f(s) = f(s)k(p)$$

where $p \in K$, $s \in F$, and $k\colon K \to H$ and $f\colon F \to H$ are the injections into the commutative coproduct of K and F. Furthermore, the commutative coproduct between KATs K and F has the property that all elements decompose into a normal form when F is finite. It is shown in Lemma 9.2 that any element in $(K \oplus F)/D$ can be written as a sum of terms of the form $p_s s$, where $p_s \in K$ and $s \in F$:

$$\forall e \in (K \oplus F)/D.\, e = \sum_{s \in F} p_s s.$$

The reason for introducing a commutative coproduct comes from the need to combine two KATs: the KAT K of the action language, and a KAT F that can describe the mutable tests. For a finite number n of test symbols, the free B! algebra F_n is also finite. Therefore, a consequence of Lemma 9.2 is that any term in the commutative coproduct $(K \oplus F_n)/D$ is equivalent to a sum of terms where the first part represents the "action" from the KAT K and the second part represents the "state" of the mutable tests from F. Moreover, $(K \oplus F_n)/D$ is isomorphic to the matrices with indices in At and elements in K, $\mathsf{Mat}(\mathsf{At}, K)$ (Theorem 9.2) which formalizes the intuition that the terms containing mutable tests encode state changes.

Example 4.13. Let α_0, α_1, α_2, and α_3 be the atoms over the tests $\{b_0?, b_1?\}$. The term

$$\alpha_0?p\alpha_1! + \alpha_1?(p+qr)\alpha_2! + \alpha_2?(r+q)\alpha_1! + \alpha_3 r\alpha_1!$$

over $(K \oplus F_n)/D$ is by Lemma 9.2 equivalent to

$$p\alpha_0?\alpha_1! + (p+qr)\alpha_1?\alpha_2! + (r+q)\alpha_2?\alpha_1! + r\alpha_3\alpha_1!.$$

The primitive action symbols p, q, and r do not alter the state, this can only be done by $\alpha!$. Hence, the tests $\alpha?$ can be moved to after the action symbol, and we get the normal form from Lemma 9.2. By Theorem 9.2, this term corresponds to the matrix

$$\begin{array}{c|cccc} & \alpha_0 & \alpha_1 & \alpha_2 & \alpha_3 \\ \hline \alpha_0 & 0 & p & 0 & 0 \\ \alpha_1 & 0 & 0 & p+qr & 0 \\ \alpha_2 & 0 & r+q & 0 & 0 \\ \alpha_3 & 0 & r & 0 & 0 \end{array}$$

wehere we represent the elements in K by the reqular expressions for brevity.

Example 4.14. Consider the term

$$b_0?pqb_0!\overline{b_1}! + b_1?qpr\overline{b_0}!$$

over tests $\{b_0?, b_1?\}$ and some KAT K. Let the atoms be:

$$\alpha_0 = b_0 b_1 \qquad \alpha_1 = b_0\overline{b_1} \qquad \alpha_2 = \overline{b_0}b_1 \qquad \alpha_2 = \overline{b_0 b_1}.$$

If we replace each test b with the sum of atoms such that $\alpha \leq$ we get:

$$\begin{gathered}(\alpha_0? + \alpha_1?)pq\alpha_1! + (\alpha_0? + \alpha_2?)qpr(\alpha_2! + \alpha_3!) = \\ \alpha_0?pq\alpha_1! + \alpha_1?pq\alpha_1! + \alpha_0?qpr\alpha_2! + \alpha_0?qpr\alpha_3! + \alpha_2?qpr\alpha_2! + \alpha_2?qpr\alpha_3! = \\ pq\alpha_0?\alpha_1! + pq\alpha_1?\alpha_1! + qpr\alpha_0?\alpha_2! + qpr\alpha_0?\alpha_3! + qpr\alpha_2?\alpha_2! + qpr\alpha_2?\alpha_3!\end{gathered}$$

in $(K \oplus F_n)/D$, which corresponds to the matrix

$$\begin{array}{c|cccc} & \alpha_0 & \alpha_1 & \alpha_2 & \alpha_3 \\ \hline \alpha_0 & 0 & pq & qpr & qpr \\ \alpha_1 & 0 & pq & 0 & 0 \\ \alpha_2 & 0 & 0 & qpr & qpr \\ \alpha_3 & 0 & 0 & 0 & 0 \end{array}$$

The set of equations that hold in $(K \oplus F_n)/D$ are completely axiomatized by the axioms of KAT and B! along with any additional equations from K, Δ_K:

$$\mathrm{KAT} + \mathrm{B!} + \Delta_K \vdash e_1 = e_2 \iff (K \oplus F_n)/D \vDash e_1 = e_2,$$

where KAT refers to the axioms of KAT and B! refers to the axioms of B! algebra. This is the contents of Theorem 9.3. The additional equations of the KAT K are encoded in the *diagram* Δ_K: the set of equations that hold in K. If K is the free KAT, no such diagram is needed, as there are no equations that hold except for those that can be proved from the axioms of KAT. Hence, when K is the free KAT,

$$\mathrm{KAT} + \mathrm{B!} \vdash e_1 = e_2 \iff (K \oplus F_n)/D \vDash e_1 = e_2,$$

which is the contents of Corollary 9.3.

Surprisingly, even though deciding equality in a B! algebra is PSPACE complete (Theorem 9.4), equality in KAT + B! is EXPSPACE-complete (Theorem 9.5).

The folk theorem of Section 4.3.1 is shown in full in Chapter 9 using KAT + B!.

4.5 Future Work

In this section we have outlined the main points of two extensions to Kleene algebra: Chomsky algebra and KAT + B!. There are more interesting questions one could investigate:

- Is there an elegant way of formulating the automata-theoretic model for context-free languages, pushdown automata, with Chomsky algebra? This would be analogous to encode the transition relation of finite state machines with matrices over Kleene algebra terms [88].

- Recent work investigates a *probabilistic* version of NetKAT [54], an extension to Kleene algebra that originally motivated the development of KAT + B! [7, 95]. It would be interesting to investigate a probabilistic KAT + B!.

- In the same vein, one could investigate extending existing coalgebraic decision procedures for NetKAT [55] to KAT + B!.

Part I

Parsing With Regular Expressions

5 Two-Pass Greedy Regular Expression Parsing

This chapter is based on the paper “Two-Pass Greedy Regular Expression Parsing” [64].

5.1 Introduction

Regular expression parsers can be built using Perl-style backtracking or general context-free parsing techniques. What the backtracking parser produces is the *greedy* parse amongst potentially many parses. General context-free parsing and backtracking parsing are not scalable since they have cubic, respectively exponential worst-case running times. REs can be and often are grammatically ambiguous and can require arbitrary long look-ahead, making limited look-ahead context-free parsing techniques inapplicable. Kearns [84] describes the first linear-time algorithm for RE parsing. In a streaming context it consists of three passes: reverse the input, perform backward NFA-simulation, and construct parse tree. Frisch and Cardelli [57] formalize greedy parsing and use the same strategy to produce a greedy parse. Dubé and Feeley [42] and Nielsen and Henglein [114] produce parse trees in linear time for fixed RE, the former producing internal data structures and their serialized forms, the latter parse trees in bit-coded form; neither produces a greedy parse.

In this chapter we present the following contributions:

1. Specification and construction of symmetric nondeterministic finite automata (NFA) with maximum in- and out-degree 2, whose paths from initial to final state are in one-to-one correspondence with the parse trees of the underlying RE; in particular, the greedy parse for a string corresponds to the lexicographically least path accepting the string.

2. NFA simulation with *ordered state sets*, which gives rise to a two-pass greedy parse algorithm using $\lceil m \lg m \rceil$ bits per input symbol and the original input string, with m the size of the underlying RE. No input reversal is required.

3. NFA simulation optimized to require only $k \leq \lceil 1/3m \rceil$ bits per input symbol, where the input string need not be stored at all and the second pass is simplified. Remarkably, this *lean-log algorithm* requires fewest log bits, and neither state set nor even the input string need to be stored.

4. An empirical evaluation, which indicates that our prototype implementation of the optimized two-pass algorithm outperforms also in practice previous RE parsing tools and is sometimes even competitive with RE tools performing limited forms of RE matching.

We first briefly recall key components related to the algorithm:

- the type interpretation of REs;
- the definition of greedy parses and their bit-coding;
- NFAs with bit-labeled transitions, called *Thompson FSTs* and *log FSTs* here, but in the paper that forms the basis for this chapter they are called *augmented Thompson NFAs* [64].

We then describe NFA simulation with ordered sets for greedy parsing and finally the optimized algorithm, which only logs join state bits.

Finally, we conclude with an empirical evaluation of a straightforward prototype to gauge the competitiveness of full greedy parsing with regular-expression based tools yielding less information for Kleene-stars.

5.2 Symmetric NFA Representation of Parse Trees

Regular expressions are finite terms of the form $0, 1, \mathrm{a}, E_1E_2, E_1 + E_2$ or $E_1^\star$, where E_1, E_2 are REs (Definition 3.2). For simplicity and brevity we henceforth assume non-problematic REs that do not contain sub-REs of the form $E^\star$, where E is nullable (Definition 3.23). All results reported here can be and have been extended to such problematic REs in the style of Frisch and Cardelli [57]. In particular, our implementation BitC handles problematic REs.

REs can be interpreted as types built from singleton, product, sum, and list type constructors [57, 74] (Definition 3.12). They denote structured values, or parse trees, in their type interpretation $\mathcal{V}[\![E]\!]$. For convenience, we repeat the relevant definitions from Chapter 3 here. The type interpretation (Definition 3.13) of a regular expression is:

$$\begin{aligned}
\mathcal{V}[\![0]\!] &= \emptyset \\
\mathcal{V}[\![1]\!] &= \{()\} \\
\mathcal{V}[\![\mathrm{a}]\!] &= \{a\} \\
\mathcal{V}[\![E_1E_2]\!] &= \mathcal{V}[\![E_1]\!] \times \mathcal{V}[\![E_2]\!] \\
\mathcal{V}[\![E_1 + E_2]\!] &= \mathcal{V}[\![E_1]\!] \oplus \mathcal{V}[\![E_2]\!] \\
\mathcal{V}[\![E_1^\star]\!] &= \{[v_1, \ldots, v_n] \mid v_i \in \mathcal{V}[\![E_1]\!], n \in \mathbb{N}\}\,.
\end{aligned}$$

The flattening (Definition 3.14) of a value is:

$$\begin{aligned}
|()| &= \epsilon \\
|a| &= \mathrm{a} \\
|\langle v_1, v_2\rangle| &= |v_1| \cdot |v_2| \\
|\mathsf{inl}\, v_1| &= |v_1| \\
|\mathsf{inr}\, v_1| &= |v_1| \\
|[v_0, \ldots, v_n]| &= |v_0| \cdot \ldots \cdot |v_n|.
\end{aligned}$$

The interpretation as structured values preserves structural information from the RE: Therefore :

Proposition 5.1. *For any regular expression E:*

$$\mathcal{L}[\![E]\!] = \{|v| \mid v \in \mathcal{V}[\![E]\!]\}\,.$$

Proof. By structural induction on E. We demonstrate the case for $E_0 + E_1$.

- First we show $\mathcal{L}[\![E_0 + E_1]\!] \subseteq \{|v| \mid v \in \mathcal{V}[\![E_0 + E_1]\!]\}$. Assume

  $$s \in \mathcal{L}[\![E_0 + E_1]\!] = \mathcal{L}[\![E_0]\!] \cup \mathcal{L}[\![E_1]\!].$$

  Then either $s \in \mathcal{L}[\![E_0]\!]$ or $s \in \mathcal{L}[\![E_1]\!]$. Without loss of generality, we may assume that $s \in \mathcal{L}[\![E_0]\!]$. By the induction hypothesis, we also have $s \in \{|v| \mid v \in \mathcal{V}[\![E_0]\!]\}$. Any v from $\mathcal{V}[\![E_0]\!]$ becomes $\mathsf{inl}\ v$ in $\mathcal{V}[\![E_0 + E_1]\!]$, and since $|v| = |\mathsf{inl}\ v|$ we must have $v \in \{|v| \mid v \in \mathcal{V}[\![E_0 + E_1]\!]\}$.

- We then have to show $\{|v| \mid v \in \mathcal{V}[\![E_0 + E_1]\!]\} \subseteq \mathcal{L}[\![E_0 + E_1]\!]$. Let

  $$v \in \mathcal{V}[\![E_0 + E_1]\!].$$

  Either $v = \mathsf{inl}\ v'$ where $v' \in \mathcal{V}[\![E_0]\!]$, or vice versa for E_1. Assume without loss of generality $v = \mathsf{inl}\ v'$. We then have by the induction hypothesis that $v' \in \mathcal{L}[\![E_0]\!]$, and therefore also $v' \in \mathcal{L}[\![E_0 + E_1]\!]$. Since $|v| = |\mathsf{inl}\ v'|$ we also have $v \in \mathcal{L}[\![E_0 + E_1]\!]$. □

We recall bit-coding from Nielsen and Henglein [114], and presented in the introduction (Definition 3.15). The bit-code $\ulcorner v \urcorner$ of a parse tree $v \in \mathcal{V}[\![E]\!]$ is a sequence of bits uniquely identifying v within $\mathcal{V}[\![E]\!]$:

$$\begin{aligned}
\ulcorner () \urcorner &= \epsilon \\
\ulcorner a \urcorner &= \epsilon \\
\ulcorner \langle v_1, v_2 \rangle \urcorner &= \ulcorner v_1 \urcorner \cdot \ulcorner v_2 \urcorner \\
\ulcorner \mathsf{inl}\ v_1 \urcorner &= \mathsf{0} \cdot \ulcorner v_1 \urcorner \\
\ulcorner \mathsf{inr}\ v_1 \urcorner &= \mathsf{1} \cdot \ulcorner v_1 \urcorner \\
\ulcorner [v_0, \ldots, v_n] \urcorner &= \mathsf{0} \cdot \ulcorner v_0 \urcorner \cdot \ldots \cdot \mathsf{0} \cdot \ulcorner v_n \urcorner \cdot \mathsf{1}.
\end{aligned}$$

That is, there exists a function $\llcorner \cdot \lrcorner_E$ such that $\llcorner \ulcorner v \urcorner \lrcorner_E = v$. The decoding function $\llcorner \cdot \lrcorner_E$ is given in Definition 3.16. It returns the left component of the following auxiliary function if all bits are consumed:

$$\begin{aligned}
D_1(bs) &= ((), bs) \\
D_{\mathrm{a}}(bs) &= (\mathrm{a}, bs) \\
D_{E+F}(\mathsf{0} \cdot bs) &= \textsc{let}\ \ (v, b') = D_E(bs)\ \ \textsc{in}\ \ (\mathsf{inl}\ v, b') \\
D_{E+F}(\mathsf{1} \cdot bs) &= \textsc{let}\ \ (v, b') = D_E(bs)\ \ \textsc{in}\ \ (\mathsf{inr}\ v, b') \\
D_{EF}(bs) &= \textsc{let}\ \ (v, b') = D_E(bs) \\
&\qquad\qquad (w, b'') = D_F(b')\ \ \textsc{in}\ \ (\langle v, w \rangle, b'') \\
D_{E^\star}(bs) &= D_{EE^\star+1}(bs).
\end{aligned}$$

We write $\mathcal{B}[\![E]\!]$ instead of $\mathcal{V}[\![E]\!]$ whenever we want to refer to the bit-codings of E rather than the parse trees of E (Definition 3.18). We use subscripts to discriminate parses with a specific flattening: $\mathcal{V}_s[\![E]\!] = \{v \in \mathcal{V}[\![E]\!] \mid |v| = s\}$. We extend the notation $\mathcal{B}_s[\![\ldots]\!]$ similarly.

Note that a bit string by itself does not carry enough information to deduce which parse tree it represents. Indeed this is what makes bit strings a compact representation of strings where the underlying RE is statically known.

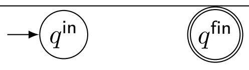
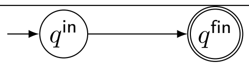
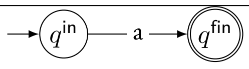
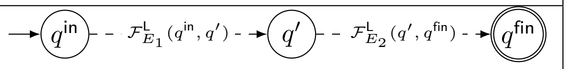
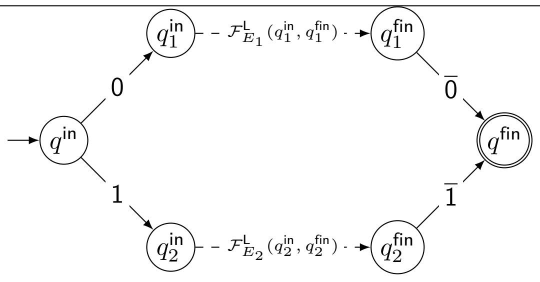
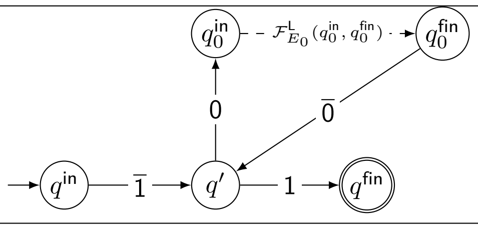
E	$\mathcal{F}^{\mathsf{L}}_{E}(q^{\mathsf{in}}, q^{\mathsf{fin}})$
0	
1	
a	
E_1E_2	 q^{in} $\mathcal{F}^{\mathsf{L}}_{E_1}(q^{\mathsf{in}}, q')$ q' $\mathcal{F}^{\mathsf{L}}_{E_2}(q', q^{\mathsf{fin}})$ q^{fin}
$E_1 + E_2$	 q_1^{in} $\mathcal{F}^{\mathsf{L}}_{E_1}(q_1^{\mathsf{in}}, q_1^{\mathsf{fin}})$ q_1^{fin} 0 $\overline{0}$ q^{in} q^{fin} 1 $\overline{1}$ q_2^{in} $\mathcal{F}^{\mathsf{L}}_{E_2}(q_2^{\mathsf{in}}, q_2^{\mathsf{fin}})$ q_2^{fin}
$E_0^{\star}$	 q_0^{in} $\mathcal{F}^{\mathsf{L}}_{E_0}(q_0^{\mathsf{in}}, q_0^{\mathsf{fin}})$ q_0^{fin} 0 $\overline{0}$ q^{in} $\overline{1}$ q' 1 q^{fin}

Figure 5.1: Log FST construction schema (Definition 3.30). For simplicity, we have omitted ϵ on the labels: edge labels $\epsilon|0$, $\epsilon|1$, $\epsilon|\overline{0}$, $\epsilon|\overline{1}$, and a$|\epsilon$ are labeled just 0, 1, $\overline{0}$, $\overline{1}$, and a, respectively. There is a one-to-one correspondence between the output labels $\{0, 1\}$ and the log labels $\{\overline{0}, \overline{1}\}$, as well as between the choice and join states. Therefore, recording the log label on the transition into each join state indicates which outgoing transition from the corresponding join state was taken.

The set $\mathcal{B}[\![E]\!]$ for an RE E can be compactly represented by *log FSTs*, a variant of enhanced NFAs [114] that has in- and outdegree at most two and carries a label on each transition. In Figure 5.1, the construction presented in the introduction is recalled. Note that in the paper that this chapter is based upon we use the term *augmented NFA* (aNFA) [64].

Recall that a log FST is a Thompson FST with some input alphabet Σ and the output alphabet $\{0, 1, \overline{0}, \overline{1}\}$ (Definition 3.30). For convenience, the construction is repeated from the introduction in Figure 5.1. In this chapter we will use a shorthand for the log FST labels:

- $\epsilon|0$ and $\epsilon|1$ are referred to as 0 and 1,
- $\epsilon|\overline{0}$ and $\epsilon|1$ are referred to as $\overline{0}$ and $\overline{1}$, and
- a$|\epsilon$ is referred to as a.

We call transition labels in Σ *input labels*; labels in $\{0, 1\}$ *output labels*; and labels in $\{\overline{0}, \overline{1}\}$ *log labels*.

Paths are defined as in Definition 3.27, but we use the shorthand notation here. If there is a path from p to q that is labeled x we write

$$p \xrightarrow{x} q.$$

The sequences $\mathsf{read}(p)$, $\mathsf{write}(p)$, and $\mathsf{log}(p)$ are the subsequences of input labels, output labels, and log labels of p, respectively.

For a log FST $\mathcal{F}^{\mathsf{L}}$ with state set Q and transition relation Δ we write (Definition 3.29):

- $J^{\mathsf{L}}_{\mathcal{F}}$ for the *join states* $\{q \in Q \mid \exists q_1, q_2.\, (q_1, \overline{0}, q), (q_2, \overline{1}, q) \in \Delta\}$;
- $S^{\mathsf{L}}_{\mathcal{F}}$ for the *source states* $\{q \in Q \mid \exists q' \in Q, \mathsf{a} \in \Sigma.\, (q, \mathsf{a}, q') \in \Delta\}$; and
- $C^{\mathsf{L}}_{\mathcal{F}}$ for the *choice states* $\{q \in Q \mid \exists q_1, q_2.\, (q, 0, q_1), (q, 1, q_2) \in \Delta\}$.

If $\mathcal{F}^{\mathsf{L}}$ is a log FST, then $\overline{\mathcal{F}^{\mathsf{L}}}$ is the FST obtained by *flipping* all transitions and exchanging the start and finishing states. That is, all transitions are reversed and all output labels have been interchanged with their corresponding log labels.

Our algorithm for constructing a log FST from an RE is a standard Thompson-style NFA generation algorithm modified to accomodate output and log labels. It is described in the introduction, but we restate it here:

Definition 5.1. We write $M = \mathcal{F}^{\mathsf{L}}_{E}(q^{\mathsf{in}}, q^{\mathsf{fin}})$ when M is a log FST constructed according to the rules in Figure 5.1 with initial state q^{in} and final state q^{fin}.

As discussed in Section 3.7.1, log FSTs are dual under reversal; that is, flipping a log FST produces the log FST for the reverse of the regular language of the original log FST.

Proposition 5.2. *Let $\overline{E}$ be canonically constructed from E to denote the reverse of $\mathcal{L}[\![E]\!]$. Let $M = \mathcal{F}^{\mathsf{L}}_{E}(q^{\mathsf{in}}, q^{\mathsf{fin}})$. Then $\overline{M} = \mathcal{F}^{\mathsf{L}}_{\overline{E}}(q^{\mathsf{fin}}, q^{\mathsf{in}})$.*

Proof. By induction on the structure of E. We illustrate the case for $E = E_0 E_1$. The log FST M must have been constructed by combining $\mathcal{F}^{\mathsf{L}}_{E_0}(q^{\mathsf{in}}, q')$ and $\mathcal{F}^{\mathsf{L}}_{E_1}(q', q^{\mathsf{fin}})$ (Figure 5.1). Therefore, the flipped log FST $\overline{M}$ must be constructed by combining $\mathcal{F}^{\mathsf{L}}_{\overline{E_1}}(q^{\mathsf{fin}}, q')$ and $\mathcal{F}^{\mathsf{L}}_{\overline{E_0}}(q', q^{\mathsf{in}})$ which by the induction hypothesis are the log FSTs for $\overline{E_1}$ and $\overline{E_0}$, respectively. As $\overline{E} = \overline{E_0 E_1} = \overline{E_1}\,\overline{E_0}$ we are done. □

This is useful since we will be running log FSTs in both forward and backward (reverse) directions.

Well-formed log FSTs—and Thompson-style NFAs in general—are canonical representations of regular expressions in the sense that they not only represent their language interpretation, but their type interpretation:

Theorem 5.1 (Representation). *Given a Thompson FST $M = \mathcal{F}_E(q^{\mathsf{in}}, q^{\mathsf{fin}})$, M outputs the bit-codings of E:*

$$\mathcal{B}_w[\![E]\!] = \left\{ b \mid q^{\mathsf{in}} \overset{w|b}{\rightsquigarrow} q^{\mathsf{fin}} \right\}.$$

Proof. Left to right: by induction on the structure of E. Right to left: by induction on the structure of $\mathcal{F}_E$. □

Corollary 5.1. *The set of outputs of all paths through $\mathcal{F}_E$ is the set of bit-codes for E:*

$$\mathcal{B}[\![E]\!] = \left\{ b \mid q^{\mathsf{in}} \overset{w|b}{\rightsquigarrow} q^{\mathsf{fin}} \wedge w \in \mathcal{L}[\![E]\!] \right\}.$$

Corollary 5.2. *The log FST $M = \mathcal{F}^{\mathsf{L}}_E(q^{\mathsf{in}}, q^{\mathsf{fin}})$ outputs the bit-codings of E:*

$$\mathcal{B}_w[\![E]\!] = \left\{\mathsf{write}(p) \mid q^{\mathsf{in}} \overset{p}{\rightsquigarrow} q^{\mathsf{fin}} \wedge \mathsf{read}(p) = w\right\}.$$

5.3 Greedy Parsing

The *greedy parse* of a string s under an RE E is what a backtracking parser returns that tries the left operand of an alternative first and backtracks to try the right alternative only if the left alternative does not yield a successful parse. The name comes from treating the Kleene star $E^\star$ as $EE^\star + 1$, which "greedily" matches E against the input as many times as possible. A "lazy" matching interpretation of $E^\star$ corresponds to treating $E^\star$ as $1 + EE^\star$. (In practice, multiple Kleene-star operators are allowed to make both interpretations available; e.g. $E^\star$ and $E^{\star\star}$ in PCRE.)

Greedy parsing can be formalized by an order $\lessdot$ on parse trees, where $v_1 \lessdot v_2$ means that v_1 is "more greedy" than v_2. We recall from Definition 3.21 that the binary relation $\lessdot$ is defined inductively on the structure of values as follows [57]:

$$\begin{array}{rcll}
\langle v_1, v_2\rangle & \lessdot & \langle v_1', v_2'\rangle & \text{if} \quad v_1 \lessdot v_1' \vee (v_1 = v_1' \wedge v_2 \lessdot v_2') \\
\mathsf{inl}\, v_0 & \lessdot & \mathsf{inl}\, v_0' & \text{if} \quad v_0 \lessdot v_0' \\
\mathsf{inr}\, v_0 & \lessdot & \mathsf{inr}\, v_0' & \text{if} \quad v_0 \lessdot v_0' \\
\mathsf{inl}\, v_0 & \lessdot & \mathsf{inr}\, v_0' & \\
[v_1, \ldots] & \lessdot & [] & \\
[v_1, \ldots] & \lessdot & [v_1', \ldots] & \text{if} \quad v_1 \lessdot v_1' \\
[v_1, v_2, \ldots] & \lessdot & [v_1, v_2', \ldots] & \text{if} \quad [v_2, \ldots] \lessdot [v_2', \ldots]
\end{array}$$

The relation $\lessdot$ is not a total order; consider for example the incomparable elements $\langle \mathrm{a}, \mathsf{inl}\, ()\rangle$ and $\langle \mathrm{b}, \mathsf{inr}\, ()\rangle$. The parse trees of any particular RE are totally ordered, however:

Proposition 5.3. *For each E, the order $\lessdot$ is a strict total order on $\mathcal{V}[\![E]\!]$.*

Proof. By induction on the structure of E. We illustrate the case for $E = E_0 + E_1$. Let $v, v' \in \mathcal{V}[\![E_0 + E_1]\!]$. There are four possible possibilities for v and v'. If $v = \mathsf{inl}\, w$ and $v' = \mathsf{inr}\, w'$ or vice versa we are done. Consider the case when $v = \mathsf{inl}\, w$ and $v' = \mathsf{inl}\, w'$. We then must have $w \in \mathcal{V}[\![E_0]\!]$ and $w' \in \mathcal{V}[\![E_0]\!]$, so by the induction hypothesis we can order them $w \lessdot w'$ or vice versa. Hence, by the definition of the greedy ordering, we must either have $v \lessdot v'$ or $v' \lessdot v$. The case for $v = \mathsf{inr}\, w$ and $v' = \mathsf{inr}\, w'$ is symmetric. □

In the following, we will show that there is a correspondence between the structural order on values and the lexicographical ordering on their bit-codings (Definition 3.25). Recall that the lexicographical ordering of two bit sequences d, d is given by:

1. $\epsilon \prec d$ if $d \neq \epsilon$,
2. $0 \cdot d \prec 1 \cdot d'$,
3. $0 \cdot d \prec 0 \cdot d'$ if $d \prec d'$,
4. $1 \cdot d \prec 1 \cdot d'$ if $d \prec d'$.

Theorem 5.2. *For any RE E and values $v, v' \in \mathcal{V}[\![E]\!]$, $v \lessdot v'$ if and only if $\ulcorner v \urcorner \prec \ulcorner v' \urcorner$.*

Proof. By induction on the structure of E. We illustrate the case for $E = E_0 + E_1$. Let $v, v' \in \mathcal{V}[\![E_0 + E_1]\!]$. There are four combinations of v, v'; we consider the case $v = \mathsf{inl}\ w$, $v' = \mathsf{inl}\ w'$: The bit-codes are

$$\ulcorner \mathsf{inl}\ w \urcorner = 0 \cdot \ulcorner w \urcorner \qquad\qquad \ulcorner \mathsf{inl}\ w' \urcorner = 0 \cdot \ulcorner w' \urcorner.$$

Assume $w \lessdot w'$: then by the induction hypothesis $\ulcorner w \urcorner \prec \ulcorner w' \urcorner$, but then we also have $\ulcorner v \urcorner = 0 \cdot \ulcorner w \urcorner \prec 0 \cdot \ulcorner w' \urcorner = \ulcorner v' \urcorner$. For the other direction, assume that $\ulcorner w \urcorner \prec \ulcorner w' \urcorner$: then by the induction hypothesis we have $w \lessdot w'$, so therefore $v \lessdot v'$. The other cases for v, v' follow similarly. □

Recall that we have limited ourselves to *non-problematic* REs. We therefore have:

Corollary 5.3. *For any RE E with log FST $M = \mathcal{F}^{\mathsf{L}}_E(q^{\mathsf{in}}, q^{\mathsf{fin}})$, and for any string s, $\min_{\lessdot} \mathcal{V}_s[\![E]\!]$ exists and is the decoding of the minimum bit-code in the lexicograpical order:*

$$\min_{\lessdot} \mathcal{V}_s[\![E]\!] = \llcorner \min_{\prec} \left\{ \mathsf{write}(p) \mid q^{\mathsf{in}} \overset{p}{\rightsquigarrow} q^{\mathsf{fin}} \wedge \mathsf{read}(p) = s \right\} \lrcorner_E .$$

Proof. Follows from Corollary 5.2 and Theorem 5.2. □

We can now characterize greedy RE parsing as follows: Given an RE E and string s, find bit sequence b such that there exists a path p from start to finishing state in the log FST for E such that:

1. $\mathsf{read}(p) = s$,
2. $\mathsf{write}(p) = b$,
3. b is lexicographically least among all paths satisfying 1 and 2.

This is easily done by a backtracking algorithm that tries 0-labeled transitions before 1-labeled ones. It is atrociously slow in the worst case, however: exponential time. How to do it faster?

5.4 NFA-Simulation with Ordered State Sets

Our first algorithm is basically an NFA-simulation. We only sketch its key idea, which is the basis for the more efficient algorithm in the following section.

Recall from Definition 3.10 that a standard NFA-simulation consists of computing the set $\mathsf{Reach}^*(S, s)$, where

$$\begin{aligned} \mathsf{Reach}^*(S, \epsilon) &= \mathsf{Close}(S) \\ \mathsf{Reach}^*(S, \mathrm{a} \cdot w') &= \mathsf{Reach}^*(\mathsf{Step}(\mathsf{Close}(S), \mathrm{a}), w'). \end{aligned}$$

Checking $q^{\mathsf{fin}} \in \mathsf{Reach}^*(\{q^{\mathsf{in}}\}, s)$ determines whether s is accepted or not. But how to construct an accepting *path* and in particular the one corresponding to the greedy parse?

We can *log* the set of states reached after each symbol during the NFA-simulation. After forward NFA-simulation, let S_i be the FST-states reached after processing the first i symbols of input $s = a_1 \dots a_n$. Given a list of logged state sets, the input string s and the final state q^{fin}, the nondeterministic algorithm Path^* constructs a path from q^{in} to q^{fin} through the state sets:

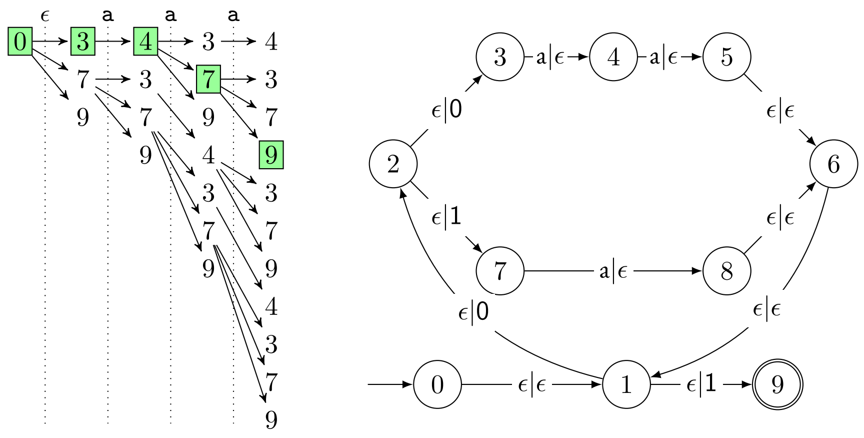


Figure 5.2: The automaton and fat log from Example 3.15. The backwards trace of the lexicographical least path is highlighted. Note that we have not abbreviated the labels in this figure.

Definition 5.2 (Path recovery). Given a list of logged state sets that have been active in an NFA-simulation on an input string s, $S_0, \ldots, S_n$, we can reconstruct a path through the FST that reads s:

$$\mathsf{Path}(S_i, q) = (q', p) \text{ where } q' \in S_i, q' \xrightarrow{p} q, \mathsf{read}(p) = a_i$$

$$\mathsf{Path}^*(S_0, q) = p' \cdot p \text{ where } (q', p) = \mathsf{Path}(S_0, q),\ \ q^{\mathsf{in}} \overset{p'}{\rightsquigarrow} q', \mathsf{read}(p') = \epsilon$$
$$\mathsf{Path}^*(S_i, q) = p' \cdot p \text{ where } (q', p) = \mathsf{Path}(S_i, q),\ \ p' = \mathsf{Path}^*(S_{i-1}, q').$$

Calling $\mathsf{write}(\mathsf{Path}^*(S_n, q^{\mathsf{fin}}))$ gives a bit-coded parse tree, though not necessarily the lexicographically least.

We can adapt the NFA-simulation by keeping each state set S_i in a particular order: If $\mathsf{Reach}^*(\{q^s\}, a_1 \ldots a_i) = \{q_{i1}, \ldots q_{ij_i}\}$ then order the q_{ij} according to the lexicographic order of the paths reaching them. Intuitively, the highest ranked state in S_i is on the greedy path if the remaining input is accepted from this state; if not, the second-highest ranked is on the greedy path, if the remaining input is accepted; and so on. This is illustrated in Example 3.15, which we have illustrated again in Figure 5.2.

The NFA-simulation can be refined to construct properly ordered state sequences instead of sets without asymptotic slow-down. The log, however, is adversely affected by this. We need $\lceil m \lg m \rceil$ bits per input symbol, for a total of $\lceil mn \lg m \rceil$ bits.

The key property for allowing us to list a state at most once in an order state sequence is this:

Lemma 5.1. *Let s, t_1, t_2, and t be states in a log FST M, and let p_1, p_2, q_1, q_2 be paths in M such that*

- $s \overset{p_1}{\rightsquigarrow} t_1$ *and* $s \overset{p_2}{\rightsquigarrow} t_2$,
- $t_1 \overset{q_1}{\rightsquigarrow} t$ *and* $t_2 \overset{q_2}{\rightsquigarrow} t$.

where p_1 is not a prefix of p_2. If $\mathsf{write}(p_1) \prec \mathsf{write}(p_2)$ *then* $\mathsf{write}(p_1 q_1) \prec \mathsf{write}(p_2 q_2)$.

Proof. Application of the lexicographical ordering on paths. ☐

5.5 Lean-Log Algorithm

After the ordered forward NFA-simulation with logging, the algorithm Path above can be refined to always yield the greedy parse whend traversing the log FST in backwards direction. Since the join states J_M of a log FST M become the choice states $C_{\overline{M}}$ of the reverse FST $\overline{M}$ we only need to construct one "direction" bit for each join state at each input string position. It is not necessary to record any states in the log at all, and we do not even have to store the input string. This results in an algorithm that requires only k bits per input symbol for the log, where k is the number of Kleene-stars and alternatives occurring in the RE. It can be shown that $k \leq \frac{1}{3}m$, the size of the log FST; in practice we can observe $k \ll m$.

We shall refer to the sequence of bits that is stored per input symbol as the *log frame*:

Definition 5.3 (Log frame). A *log frame* ℓ is a map between join states and log labels:

$$\ell\colon \left\{\overline{\mathsf{0}}, \overline{\mathsf{1}}\right\}^{J^{\mathsf{L}}_{\mathcal{F}}}$$

Definition 5.4. Define $\odot$ to be the point-wise operation on two pairs that concatenates the first components and unions the second components:

$$([q_0, \dots, q_n], \ell_0) \odot ([q'_0, \dots, q'_m], \ell_1) = ([q_0, \dots, q_n, q'_0, \dots, q'_m], \ell_0 \cup \ell_1).$$

Our optimized algorithm is obtained by combining a group of algorithms. We first augment the ϵ-closure to be ordered and output log frames:

Definition 5.5 (Ordered ϵ-closure). The *ordered ϵ-closure* of a state is:

$$\mathsf{Close}\colon Q \times \left\{\overline{\mathsf{0}}, \overline{\mathsf{1}}\right\}^{J^{\mathsf{L}}_{\mathcal{F}}} \to Q^\star \times \left\{\overline{\mathsf{0}}, \overline{\mathsf{1}}\right\}^{J^{\mathsf{L}}_{\mathcal{F}}}$$

$$\mathsf{Close}(q, \ell) = \begin{cases} \mathsf{Close}(q_0, \ell) \odot \mathsf{Close}(q_1, \ell) & q \xrightarrow{\epsilon|\mathsf{0}} q_0, q \xrightarrow{\epsilon|\mathsf{1}} q_1 \\ \mathsf{Close}(q', \ell \cup \{q' \mapsto t\}) & q \xrightarrow{\epsilon|t} q', t \in \left\{\overline{\mathsf{0}}, \overline{\mathsf{1}}\right\}, q' \notin \mathrm{dom}(L) \\ ([q], \ell) & \text{otherwise.} \end{cases}$$

Next, the stepping function is extended to respect the ordering of states and combine log frames with $\odot$:

Definition 5.6 (Ordered step function). The *ordered stepping function* is:

$$\mathsf{Step}\colon Q^\star \times \Sigma \times (Q^\star \times \left\{\overline{\mathsf{0}}, \overline{\mathsf{1}}\right\}^{J^{\mathsf{L}}_{\mathcal{F}}}) \to Q^\star \times \left\{\overline{\mathsf{0}}, \overline{\mathsf{1}}\right\}^{J^{\mathsf{L}}_{\mathcal{F}}}$$

$$\mathsf{Step}\,([], a, (S, \ell)) = (S, L)$$

$$\mathsf{Step}\,(q \cdot qs, a, (S, \ell)) = \begin{cases} \mathsf{Step}(qs, a, (S, \ell) \odot \mathsf{Close}(q', \ell)) & q \xrightarrow{\mathsf{a}|\epsilon} q' \\ \mathsf{Step}(qs, a, (S, \ell)) & \text{otherwise.} \end{cases}$$

With these two function, the *forward pass* of out algorithm is described as:

Definition 5.7 (Forward pass). The *forward pass algorithm* is:

$$\mathsf{Fwd}\colon \Sigma^\star \to \left(\left\{\overline{\mathsf{0}}, \overline{\mathsf{1}}\right\}^{J^{\mathsf{L}}_{\mathcal{F}}}\right)^\star \cup \{\bot\}$$

$$\mathsf{Fwd}(s) = \text{let } (S_0, \ell_0) = \mathsf{Close}(q^{\mathsf{in}}, []) \text{ in } \mathsf{Fwd}'(s, S_0, [\ell_0])$$

where we have made use of auxiliary function Fwd':

$$\begin{aligned}
\mathsf{Fwd}' \colon \Sigma^\star \times Q^\star \times \left(\{\overline{0},\overline{1}\}^{J_\mathcal{F}}\right)^\star &\to \left(\{\overline{0},\overline{1}\}^{J_\mathcal{F}}\right)^\star \cup \{\bot\} \\
\mathsf{Fwd}'([], S, L) &= \begin{cases} L & \text{if } q^{\mathsf{in}} \in S \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{Fwd}'(a \cdot s', S, L) &= \begin{cases} \mathsf{Fwd}'(s', S', \ell \cdot L) & \text{if } (S', \ell) = \mathsf{Step}(S, a, ([], \emptyset)), S' \neq [] \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

The function Fwd yields a list of log frames L. Finally, to produce the parse tree we must walk backwards through the log FST and use the list of log frames as an oracle. Hence, in this pass there is no more non-determinism, as it is resolved by the log frames:

Definition 5.8 (Backward pass). The *backward pass algorithm* is:

$$\begin{aligned}
\mathsf{Bwd} \colon \left(\{\overline{0},\overline{1}\}^{J^{\mathsf{L}}_\mathcal{F}}\right)^\star \times Q &\to \{0,1\}^\star \cup \{\bot\} \\
\mathsf{Bwd}(\bot, \cdot) &= \bot \\
\mathsf{Bwd}(\ell \cdot L, q) &= \begin{cases} [] & \text{if } q = q^{\mathsf{in}} \\ \mathsf{Bwd}(\ell \cdot L, q') \cdot b & \text{if } q' \xrightarrow{\epsilon \mid b} q, b \in \{0,1\} \\ \mathsf{Bwd}(L, q') & \text{if } q' \xrightarrow{a \mid \epsilon} q, a \in \Sigma \\ \mathsf{Bwd}(\ell \cdot L, q') & \text{if } q' \xrightarrow{\epsilon \mid \ell(q)} q, \ell(q) \in \{\overline{0},\overline{1}\} \end{cases}
\end{aligned}$$

The final parsing algorithm is obtained by combining the forward and the backward pass:

Definition 5.9 (Two-phase parsing algorithm). The complete *two-phase parsing algorithm* is the composition of the forward and backward passes of Definitions 5.7 and 5.8:

$$\begin{aligned}
\mathsf{TwoPhaseParser} \colon \Sigma^\star &\to \{0,1\}^\star \cup \{\bot\} \\
\mathsf{TwoPhaseParser}(w) &= \mathsf{Bwd}\left(\mathsf{Fwd}(w), q^{\mathsf{fin}}\right).
\end{aligned}$$

The forward pass keeps the log FST and the current character in memory. This requires $O(m)$ words of random access memory (RAM). It writes writing $n \cdot k$ bits to the log and discards the each character of the input string after it is read. Finally, the backward pass also requires $O(m)$ words of RAM and reads from the log in reverse write order. The log is thus a two-phase stack: In the first pass it is only pushed to, in the second pass popped from.

Both Close and Step run in time $O(m)$ per input symbol, hence the forward pass requires time $O(m \cdot n)$. Likewise, the backward pass requires time $O(m \cdot n)$.

Close keeps track of visited states and returns the states reached ordered lexicographically according to the paths reaching them. Hence, the following theorem can be proved:

Theorem 5.3. *For any regular expression E and symbol sequence s, if $L_l = \mathsf{Fwd}(s)$, and $d = \mathsf{Bwd}(L_l, q^{\mathsf{fin}})$, then $\llcorner d \lrcorner_E = \min_\lessdot \mathcal{V}_s[\![E]\!]$.*

Proof. The forward pass maintains the states in the lexicographical order of output strings on the paths. Because the bit for a join state q is only set the first time q is reached, it is only set on the lexicographically least path from q^{in} to q. Therefore, the oracle bits used when tracing backwards with Bwd causes the program to take the greedy leftmost path (in the non-flipped FST), which has the lexicographically least output bit-code. By Corollary 5.3 this is the value $\min_{\lessdot} \mathcal{V}_s[\![E]\!]$. □

5.6 Evaluation

We have implemented the optimized algorithms in C and in Haskell and compared the performance of the C implementation with the following existing RE tools:

RE2: Google's RE implementation [139].

Tcl: The scripting language Tcl [119].

Perl: The scripting language Perl [30].

Grep: The UNIX tool `grep` [136].

Rcp: The implementation of the algorithm "*DFASIM*" [114]. It is based on Dubé and Feeley's method [42], but altered to produce a bit-coded parse tree.

FrCa: The implementation of the algorithm"*FrCa*" algorithm used in [114]. It is based on Frisch and Cardelli's method [57].

In the subsequent plots, our implementation of the lean-log algorithm is referred to as *BitC*. The tests have been performed on an Intel Xeon 2.5 GHz machine running GNU/Linux 2.6.

Note that we use the shorthand $E?$ for the RE $E + 1$.

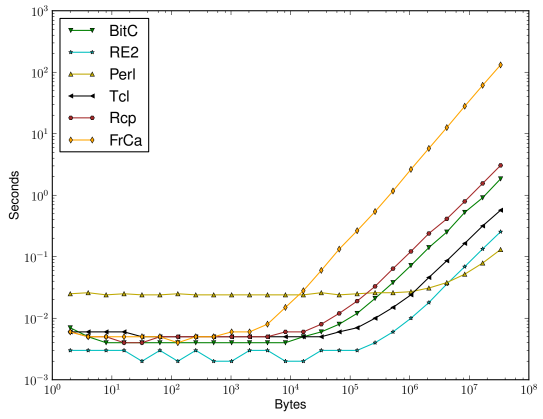
5.6.1 Pathological Expressions

To get an indication of the "raw" throughput for each tool, $\mathrm{a}^\star$ was run on sequences of as (Figure 5.3(a)). (Note that the plots use log scales on both axes, so as to accommodate the dramatically varying running times.) Perl outperforms the rest, likely due to a strategy where it falls back on a simple scan of the input. FrCa stores each position in the input string from which a match can be made, which in this case is every position. As a result, FrCa uses significantly more memory than the rest, causing a dramatic slowdown.

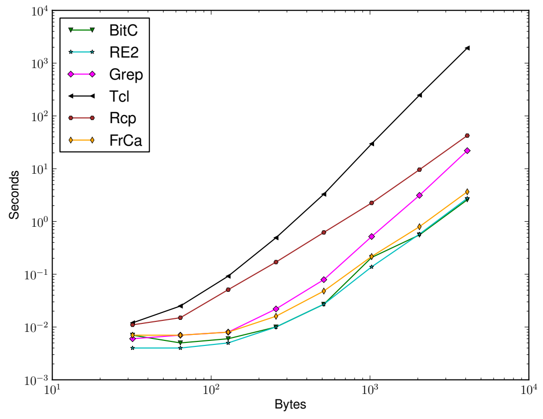
The expression $(\mathrm{a}+\mathrm{b})^\star\mathrm{a}(\mathrm{a}+\mathrm{b})^n$ with the input $(\mathrm{ab})^{n/2}$ is a worst-case for DFA-based methods, as it results in a number of states exponential in n. Perl has been omitted from the plots, as it was prohibitively slow. Tcl, Rcp, and Grep all perform orders of magnitude slower than FrCa, RE2, and BitC (Figure 5.3(b)), indicating that Tcl and Grep also use a DFA for this expression. If we fix n to 25, it becomes clear that FrCa is slower than the rest, likely due to high memory consumption as a result of its storing all positions in the input string (Figure 5.3(c)). The asymptotic running times of the others appear to be similar to each other, but with greatly varying constants.

For the backtracking worst-case expression $(\mathrm{a}?)^n\mathrm{a}^n$ in Figure 5.4(a), BitC performs roughly like RE2.[1] In contrast to Rcp and FrCa, which are both highly sensitive to the *direction* of non-determinism, BitC has the same performance for both $(\mathrm{a}?)^n\mathrm{a}^n$ and $\mathrm{a}^n(\mathrm{a}?)^n$ (Figure 5.4(b)).

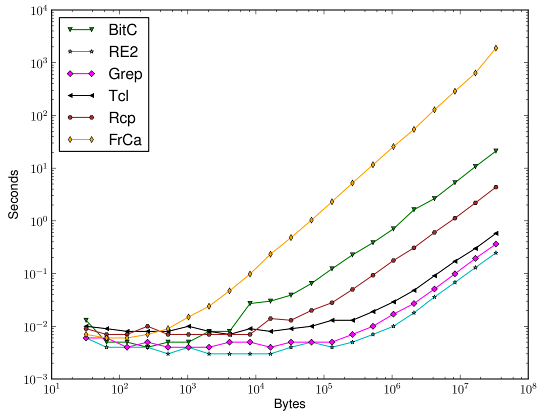
[1]The expression parser in BitC failed for the largest expressions, which is why they are not on the plot.



(a) $a^\star$, input a^n.

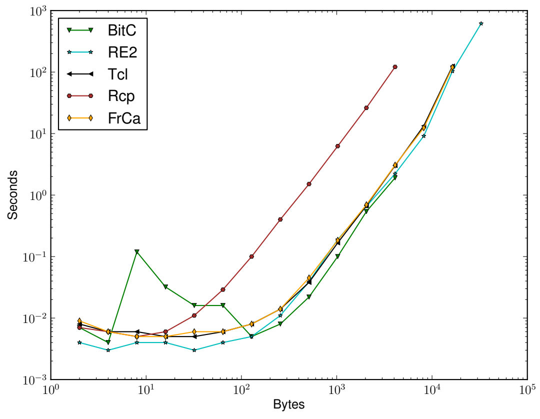


(b) $(a+b)^\star a(a+b)^n$, input $(ab)^{n/2}$.

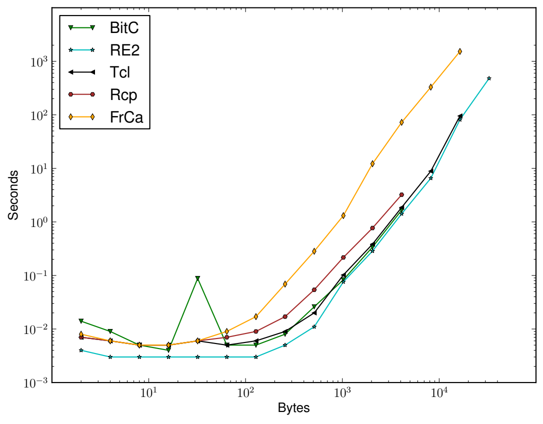


(c) $(a+b)^\star a(a+b)^{25}$, input $(ab)^{n/2}$.

Figure 5.3: Comparisons with different REs.



(a) $(a?)^n a^n$, input a^n.



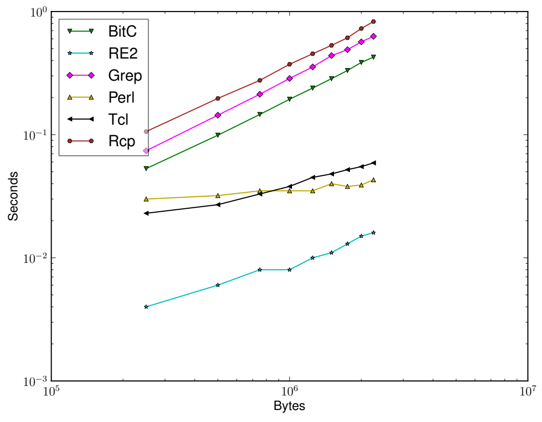
(b) $a^n (a?)^n$, input a^n.

Figure 5.4: Backtracking worst-case expressions and its reverse.

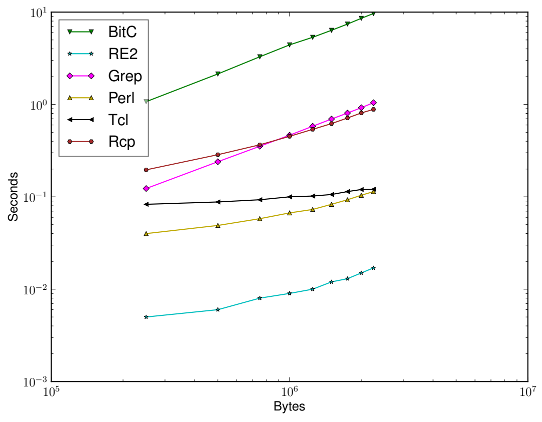
5.6.2 Practical Examples

We have run the comparisons with various "real-life" examples of REs taken from [145], all of which deal with expressions matching e-mail addresses. In Figure 5.5(b), BitC is significantly slower than in the other examples. This can likely be ascribed to heavy use of bounded repetitions in this expression, as they are currently just rewritten into concatenations and alternations in our implementation. A more efficient implementation would use a dedicated construction for repetitions, using a simple counter instead of n states for an expression like a^n.

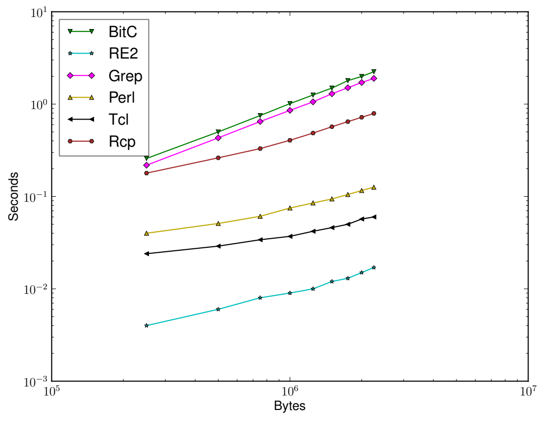
In the other two cases, BitC's performance is roughly like that of Grep. This is promising for BitC since Grep performs only RE *matching*, not full *parsing*. RE2 is consistently ranked as the fastest program in our benchmarks, presumably due to its aggressive optimizations and ability to dynamically choose between several strategies. Recall that RE2 performs greedy leftmost subgroup matching, not full parsing. Our present prototype of BitC is coded in less than 1000 lines of C. It uses only standard libraries and performs no optimizations such as NFA-minimization, DFA-construction, cached or parallel NFA-simulation,



(a) #4



(b) #7



(c) #8

Figure 5.5: “Real-life” examples of REs.

etc.: this is future work.

5.7 Related Work

The known RE parsing algorithms can be divided into four categories.

- The first category is Perl-style backtracking used in many tools and libraries for RE subgroup matching [39]; it has an exponential worst case running time, but always produces the greedy parse and enables some extensions to REs such as backreferences.

- Another category consists of context-free parsing methods, where the RE is first translated to a context-free grammar, before a general context-free parsing algorithm such as Earley's [43] using cubic time is applied. An interesting CFG method is derivatives-based parsing [107]. While efficient parsers exist for subsets of unambiguous context-free languages, this restriction propagates to REs, and thus these parsers can only be applied for subsets of unambiguous REs.

- The third category contains RE scalable parsing algorithms that do not always produce the greedy parse. This includes NFA and DFA based algorithms provided by Dubé and Feeley [42] and Nielsen and Henglein [114], where the RE is first converted to an NFA with additional information used to parse strings or to create a DFA preserving the additional information for parsing. While the method presented here uses a Thompson-style NFA generation in order to exploit the symmetry that enables the efficient greedy parsing, these algorithms do not necessarily require the same properties, so other techniques such as ϵ-free automata could in theory be applied. This category also includes the algorithm by Fischer, Huch and Wilke [52]; it is left out of our tests since its Haskell-based implementation often turned out not to be competitive with the other tools.

- The last category consists of the algorithms that scale well and always produce greedy parse trees. Kearns [84] and Frisch and Cardelli [57] reverse the input; perform backwards NFA-simulation, building a log of NFA-states reached at each input position; and construct the greedy parse tree in a final forward pass over the input. They require storing the input symbol plus m bits per input symbol for the log. This can be optimized to storing bits proportional to the number of NFA-states reached at a given input position [114], although the worst case remains the same. Our lean log algorithm uses only two passes, does not require storing the input symbols and stores only $k < \frac{1}{3}m$ bits per input symbol in the string.

  We observe that the maximal munch tokenization problem [126] corresponds to greedy parsing for the RE T^* where T is an RE for tokens. Our algorithm solves this in time $O(mn)$ using two passes and $O(m)$ random access memory, whereas Reps' solution solves it in time $O(mn)$ in one pass using $O(mn)$ random access memory [126].

6 Optimally Streaming Greedy Regular Expression Parsing

This chapter is based on the paper "Optimally Streaming Greedy Regular Expression Parsing" [65].

6.1 Introduction

In programming, regular expressions are often used to extract information from an input, which requires an intensional interpretation of regular expressions as denoting parse trees, and not just their ordinary language-theoretic interpretation as denoting strings.

This is a nontrivial change of perspective. We need to deal with grammatical ambiguity—*which* parse tree to return, not just that it has one—and memory requirements become a critical factor: Deciding whether a string belongs to the language denoted by $(ab)^\star + (a + b)^\star$ can be done in constant space, but outputting the first bit, whether the string matches the first alternative or only the second, may require buffering the whole input string. This is an instructive case of deliberate grammatical ambiguity to be resolved by the prefer-the-left-alternative policy of greedy disambiguation: Try to match the left alternative; if that fails, return a match according to the right alternative as a fallback. Straight-forward application of automata-theoretic techniques does not help: $(ab)^\star + (a+b)^\star$ denotes the same *language* as $(a + b)^\star$, which is unambiguous and corresponds to a small DFA, but is also useless: it does not represent any more when a string consists of a sequence of ab-groups. If a programmer writes $(ab)^\star + (a+b)^\star$, his/her *intention* is that a sequence of ab is a special token, but arbitrary strings over a and b should still be handled. Compiling the expression to a DFA ignores this intent.

Previous parsing algorithms [42, 57, 83, 114, 135], and the one presented in Chapter 5 [64], require at least one full pass over the input string before outputting any output bits representing the parse tree. This is the case even for regular expressions requiring only bounded lookahead such as one-unambiguous regular expressions [25].

In this chapter we study the problem of *optimally streaming* parsing. Consider

$$(ab)^\star + (a + b)^\star,$$

which is ambiguous and in general requires unbounded input buffering, and consider the particular input string

$$ab \ldots ab\ aa\ bababab \ldots .$$

An *optimally* streaming parsing algorithm needs to buffer the prefix ab . . . ab in some form because the complete parse might match either of the two alternatives in the regular expression, but once encountering aa, only the right alternative is possible. At this point it outputs this information and the output representation for the buffered string as parsed by the second alternative. After this, it outputs a bit for each input symbol read, with no internal buffering: input symbols are discarded before reading the next symbol. Optimality means that output bits representing the eventual parse tree must be produced *earliest possible*: as soon as they are *semantically determined* by the input processed so far under the assumption that the parse will succeed.

Outline. In Section 6.2 we recall some notions from Chapter 3 about regular expressions, their type interpretation, and the bit-coding of parse trees.

The class of Thompson-style machines called *Thompson FSTs* in Chapter 3 are briefly recalled in Section 6.3. Paths and their output labels in such FSTs naturally represent *complete* parse trees, and paths to intermediate states represent *partial* parse trees for prefixes of an input string. Note that we called these machines augmented NFAs in the original paper that this chapter is based upon [65]. This is similar to the previous chapter; we have changed the name to Thompson FST to better normalize naming conventions between Chapters 3, 5, 6, and 7.

The *greedy disambiguation strategy* is used to specify a deterministic mapping of accepted strings to paths in the FST; this is recalled in Section 6.4.

We then formally define the concept of optimal streaming parsing in Section 6.5 and shows some useful properties. A requirement for optimal streaming is the concept of *covering*. This has been briefly introduced in the introduction (Section 3.8). In Section 6.6 we discuss it in more detail and show that it is a computationally hard problem to decide.

Finally, the parsing algorithm is described in Section 6.7. It is based on the *path trees* already discussed in Chapter 3, which will be treated more formally in the present chapter. With these, an optimally streaming parsing algorithm is specified, and its asymptotic runtime complexity analyzed.

Finally, in Section 6.8, we continue the thread from Section 3.8 and demonstrate the algorithm by illustrative examples alluding to its expressive power and practical utility.

6.2 Preliminaries

In the following section, we recall the definitions of regular expressions and their interpretation as types from Chapter 3. The regular expressions over Σ are given by the grammar (Definition 3.2):

$$E ::= 0 \mid 1 \mid \mathrm{a} \mid E_1 + E_2 \mid E_1 E_2 \mid E_1^\star,$$

where $\mathrm{a} \in \Sigma$. Concatenation (juxtaposition) and alternation ($+$) associates to the right; parentheses may be inserted to override associativity. Kleene star ($\star$) binds tightest, followed by concatenation and alternation.

The standard interpretation of a regular expression E is as a denotation of a regular language $\mathcal{L}[\![E]\!] \subseteq \Sigma^\star$ (Definition 3.3):

$$\begin{aligned}
\mathcal{L}[\![0]\!] &= \emptyset \\
\mathcal{L}[\![1]\!] &= \{\epsilon\} \\
\mathcal{L}[\![\mathrm{a}]\!] &= \{\mathrm{a}\} \\
\mathcal{L}[\![E_1 + E_2]\!] &= \mathcal{L}[\![E_1]\!] \cup \mathcal{L}[\![E_2]\!] \\
\mathcal{L}[\![E_1 E_2]\!] &= \mathcal{L}[\![E_1]\!] \cdot \mathcal{L}[\![E_2]\!] \\
\mathcal{L}[\![E^\star]\!] &= \mathcal{L}[\![E]\!]^\star,
\end{aligned}$$

In the remainder of this chapter we shall restrict ourselves to regular expressions E such that $\mathcal{L}[\![E]\!] \neq \emptyset$.

For regular expression parsing, we are interested in the interpretation of regular expressions as *types*, denoting sets of structured values, or parse trees. The *values* over Σ are elements formed over the grammar (Definition 3.12):

$$v ::= () \mid \mathsf{a} \mid \langle v_1, v_2 \rangle \mid \mathsf{inl}\, v_1 \mid \mathsf{inr}\, v_1 \mid [v_0, \dots, v_n],$$

where $\mathsf{a} \in \Sigma$. The regular expressions interpreted as types denote sets of these values (Definition 3.13):

$$\begin{aligned}
\mathcal{V}[\![0]\!] &= \emptyset \\
\mathcal{V}[\![1]\!] &= \{()\} \\
\mathcal{V}[\![\mathsf{a}]\!] &= \{\mathsf{a}\} \\
\mathcal{V}[\![E_1 E_2]\!] &= \mathcal{V}[\![E_1]\!] \times \mathcal{V}[\![E_2]\!] \\
\mathcal{V}[\![E_1 + E_2]\!] &= \mathcal{V}[\![E_1]\!] \oplus \mathcal{V}[\![E_2]\!] \\
\mathcal{V}[\![E_1^\star]\!] &= \{[v_1, \dots, v_n] \mid v_i \in \mathcal{V}[\![E_1]\!], n \in \mathbb{N}\}\,.
\end{aligned}$$

Recall that $|v|$ is the the *flattening* of a value v, defined as the word obtained by doing an in-order traversal of v and writing down all the symbols in the order they are visited (Definition 3.14). We write $\mathcal{V}_w[\![E]\!]$ for the restricted set $\{v \in \mathcal{V}[\![E]\!] \mid |v| = w\}$. Regular expression *parsing* is a generalization of the acceptance problem of determining whether a word w belongs to the language of some RE E, where additionally we produce a parse tree from $\mathcal{V}_w[\![E]\!]$. We say that an expressoin E is *ambiguous* iff there exists a w such that $|\mathcal{V}_w[\![E]\!]| > 1$ (Definition 3.20).

Any value can be serialized into a sequence of bits. The bit-code of a value v is written $\ulcorner v \urcorner$ (Definition 3.15):

$$\begin{aligned}
\ulcorner () \urcorner &= \epsilon \\
\ulcorner a \urcorner &= \epsilon \\
\ulcorner \langle v_1, v_2 \rangle \urcorner &= \ulcorner v_1 \urcorner \cdot \ulcorner v_2 \urcorner \\
\ulcorner \mathsf{inl}\, v_1 \urcorner &= \mathsf{0} \cdot \ulcorner v_1 \urcorner \\
\ulcorner \mathsf{inr}\, v_1 \urcorner &= \mathsf{1} \cdot \ulcorner v_1 \urcorner \\
\ulcorner [v_0, \dots, v_n] \urcorner &= \mathsf{0} \cdot \ulcorner v_0 \urcorner \cdot \ldots \cdot \mathsf{0} \cdot \ulcorner v_n \urcorner \cdot \mathsf{1}.
\end{aligned}$$

The set $\{\ulcorner v \urcorner \mid v \in \mathcal{V}[\![E]\!]\}$ is written $\mathcal{B}[\![E]\!]$ (Definition 3.18), and the set of bit-codes that flatten to a string s is written $\mathcal{B}_s[\![E]\!]$.

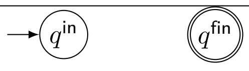
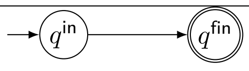
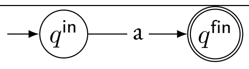
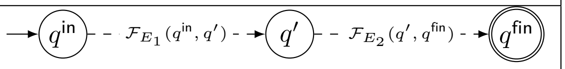
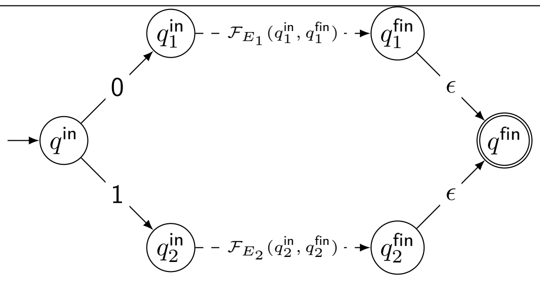
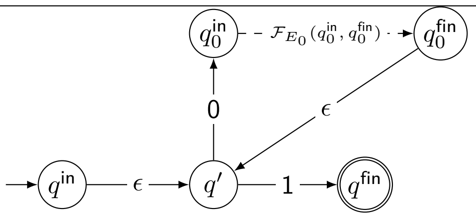
Note that for a fixed regular expression E, bit-coding is an isomorphism when seen as a function $\ulcorner \cdot \urcorner_E : \mathcal{V}[\![E]\!] \to \mathcal{B}[\![E]\!]$. Its inverse is the *decoding* $\llcorner \cdot \lrcorner_E : \mathcal{B}[\![E]\!] \to \mathcal{V}[\![E]\!]$ (Definition 3.16).

6.3 Thompson FSTs

In this section we recall the definition of *Thompson FSTs* from Section 3.6. Our construction is similar to that of Thompson [141], but augmented with extra annotations on some of the non-deterministic ϵ-transitions. The resulting automata can are a non-deterministic transducers which for each accepted input string in the language of the underlying regular expression outputs the bit-codes for the corresponding parse trees. Thus, a Thompson FST is a six-tuple

$$(Q, \Sigma, \Gamma, \{q^{\mathsf{in}}\}, \{q^{\mathsf{fin}}\}, \Delta)$$

where the output alphabet is fixed to $\Gamma = \{\epsilon, 0, 1\}$ and Σ is the input alphabet (Definition 3.26). Note that in the paper on which this chapter is based, the augmented NFAs did not have a formal distinction between input and output alphabets; this was instead accomplished by adjoining the special symbols 0 and 1 to the input alphabet of the Thompson NFA. The end result is equivalent, but we find it cleaner to separate the alphabets, making the notation more aligned with standard literature on automata and transducers. A Thompson FST must have been constructed by the extended Thompson construction from Section 3.6. We recall it in "cheat-sheet form" here:

E	$\mathcal{F}_E(q^{\mathsf{in}}, q^{\mathsf{fin}})$
0	
1	
a	
E_1E_2	 q^{in} $\mathcal{F}_{E_1}(q^{\mathsf{in}}, q')$ q' $\mathcal{F}_{E_2}(q', q^{\mathsf{fin}})$ q^{fin}
$E_1 + E_2$	 q_1^{in} $\mathcal{F}_{E_1}(q_1^{\mathsf{in}}, q_1^{\mathsf{fin}})$ q_1^{fin} 0 ϵ q^{in} q^{fin} 1 ϵ q_2^{in} $\mathcal{F}_{E_2}(q_2^{\mathsf{in}}, q_2^{\mathsf{fin}})$ q_2^{fin}
$E_0^\star$	 q_0^{in} $\mathcal{F}_{E_0}(q_0^{\mathsf{in}}, q_0^{\mathsf{fin}})$ q_0^{fin} 0 ϵ q^{in} ϵ q' 1 q^{fin}

Here, the notation $\mathcal{F}_E(q^{\mathsf{in}}, q^{\mathsf{fin}})$ refers to the Thompson FST constructed for the regular expression E with inital and final states q^{in} and q^{fin}. Note that, like in Chapter 5, we use the shorthands:

- 0 and 1 for the FST label pairs $\epsilon|0$ and $\epsilon|1$, respectively,
- a for the FST label a$|\epsilon$,
- ϵ for the FST label $\epsilon|\epsilon$.

Figure 6.1 shows a small example of the construction.

A *path* an a Thompson FST with states Q and transition relation Δ is a non-empty sequence of states

$$q_0, q_1, \ldots, q_n$$

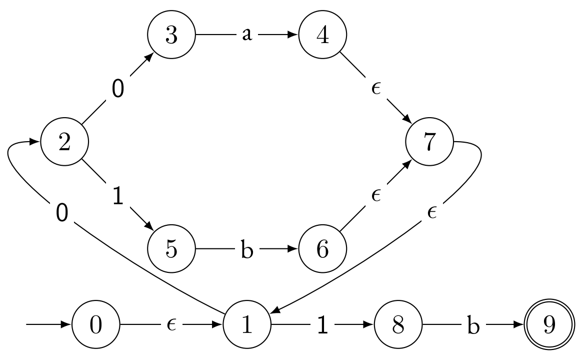


Figure 6.1: The FST $\mathcal{F}_{(\mathrm{a}+\mathrm{b})^\star\mathrm{b}}(0,9)$ for the regular expression $(\mathrm{a}+\mathrm{b})^\star\mathrm{b}$.

such that each state in the sequence has a transition in Δ to the next (Definition 3.27). We call a path between q_0 and q_n α and write

$$q_0 \overset{\alpha}{\rightsquigarrow} q_n$$

when such a path exists.

Each path α is associated with a (possibly empty) sequence of input/output label pairs (Definition 3.27). As in Chapter 5, we use the following notation:

- $\mathsf{read}(\alpha)$ is the sequence of *input labels* for a path α,
- $\mathsf{write}(\alpha)$ is the sequence of *output labels* for a path α.

With this notation, the fact that an FST accepts a word w can be formulated as

$$q^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} q^{\mathsf{fin}} \quad \wedge \quad \mathsf{read}(\alpha) = w,$$

i.e., that there is a path from initial to final state whose input labels form the word w.

Recall that there is a one-to-one correspondence between bit-codes and accepting paths (Theorem 5.1). We restate this with the syntactic shorthands used in this chapter:

Theorem. *For a regular expression E with Thompson FST $\mathcal{F}_E$, we have for each $w \in \mathcal{L}[\![E]\!]$:*

$$\begin{aligned}\mathcal{B}_w[\![E]\!] &= \left\{ b \mid q^{\mathsf{in}} \overset{w|b}{\rightsquigarrow} q^{\mathsf{fin}} \right\} \\ &= \left\{ \mathsf{write}(\alpha) \mid q^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} q^{\mathsf{fin}}, \mathsf{read}(\alpha) = w \right\}.\end{aligned}$$

Determinization. Given a state set Q, define its *closure* as the set

$$\mathrm{close}(Q) = \left\{ q' \mid q \in Q \wedge \exists\alpha.\mathsf{read}(\alpha) = \epsilon \wedge q \overset{\alpha}{\rightsquigarrow} q' \right\}.$$

For any Thompson FST $\mathcal{F} = (Q, \Sigma, \{\epsilon, 0, 1\}, \{q^{\mathsf{in}}\}, \{q^{\mathsf{fin}}\}, \Delta)$, let

$$D(\mathcal{F}) = (\mathsf{DState}_{\mathcal{F}}, I_{\mathcal{F}}, F_{\mathcal{F}}, \Delta_{\mathcal{F}})$$

be the deterministic automaton obtained by applying the standard subset sum construction: Here, $I_{\mathcal{F}} = \text{close}(\{q^{\text{in}}\})$ is the *initial state*, and $\mathsf{DState}_{\mathcal{F}} \subseteq 2^Q$ is the set of states, defined to be the smallest set containing $I_{\mathcal{F}}$ and closed under the transition function

$$\Delta_{\mathcal{F}}(Q, \text{a}) = \text{close}(\{q' \mid (q, \text{a}, q') \in \Delta, q \in Q\}).$$

The set of *final states* $F_{\mathcal{F}}$ is the set $\{Q \in \mathsf{DState}_M \mid q^{\text{fin}} \in Q\}$.

6.4 Disambiguation

A regular expression parsing algorithm has to produce a parse tree for an input word whenever the word is in the language for the underlying RE. In the case of ambiguous REs, the algorithm has to choose one of several candidates. We do not want the choice to be arbitrary, but rather a parse tree which is uniquely identified by a *disambiguation policy*. As there is a one-to-one correspondence between words in the language of an RE E and accepting paths in $\mathcal{F}_E$, a disambiguation policy can be seen as a deterministic choice between paths that recognize the same words.

We will focus on greedy disambiguation, which corresponds to choosing the first result that would have been found by a backtracking regular expression parsing algorithm such as the one found in the Perl programming language [30]. The greedy strategy has successfully been implemented in previous work [57, 64], and was also the subject of Chapter 5. It is simpler to define and implement than other strategies such as POSIX [56, 80] whose known parsing algorithms are technically more complicated [116, 134, 135].

Greedy disambiguation can be seen as picking the accepting path with the lexicographically least bitcode (Chapter 5). A well-known problem with backtracking parsing is non-termination in the case of regular expressions with nullable subexpressions under Kleene star, which means that the lexicographically least path is not always well-defined (see Example 3.12). This problem can easily be solved by not problematic paths, as in [57]. Problematic paths are paths that contain non-productive loops, i.e., a subpath

$$q_n \overset{\alpha'}{\rightsquigarrow} q_n$$

such that $\mathsf{read}(\alpha') = \epsilon$ (Definition 3.28). In practice, when implementing NFA simulation algorithms, such paths are discarded by maintaining a set of previously visited states. This is sometimes referred to as "black hole detection."

6.5 Optimal Streaming

In this section we specify what it means to be an *optimally streaming* implementation of a function from sequences to sequences.

Recall from Definition 2.8 that when a word w is a *prefix* of another word w' it means that there exists a w'' such that $ww'' = w'$. Whenever w is a prefix of w' we write:

$$w \sqsubseteq w'.$$

Note that $\sqsubseteq$ is a partial order with greatest lower bounds for nonempty sets L. This is the *longest common prefix* $\bigsqcap L$ (Definition 2.9), and it is the word w that is a prefix of all words in L such that, for any other word w' that is also a prefix of all words in L, $w' \sqsubseteq w$.

Definition 6.1 (Completions). The set of *completions* $C_E(w)$ of w for a regular expression E is the set of all words in $\mathcal{L}[\![E]\!]$ that have w as a prefix:

$$C_E(w) = \{w'' \mid w \sqsubseteq w'' \wedge w'' \in \mathcal{L}[\![E]\!]\}.$$

Note that $C_E(w)$ may be empty.

Lemma 6.1. *For words* w, w'*:*

$$w \sqsubseteq w' \implies C_E(w) \supseteq C_E(w').$$

Proof. By expanding Definition 6.1:

$$\begin{aligned} C_E(w) = \; & \{v \mid w \sqsubseteq v \wedge v \in \mathcal{L}[\![E]\!]\} \\ \supseteq & \{v \mid w \sqsubseteq w' \sqsubseteq v \wedge v \in \mathcal{L}[\![E]\!]\} = C_E(w'). \end{aligned}$$ □

Definition 6.2 (Extension). For $C_E(w) \neq \emptyset$, the unique *extension* $\widehat{w}_E$ of w under E is the longest extension of w with a suffix such that all successful extensions of w to an element of $\mathcal{L}[\![E]\!]$ are also extensions of $\widehat{w}$:

$$\widehat{w}_E = \bigsqcap C_E(w).$$

Word w is *extended* under E if $w = \widehat{w}$; otherwise it is unextended.

Extension is a closure operation: $\widehat{\widehat{w}} = \widehat{w}$; in particular, extensions are extended.

Example 6.1. Let $E = \mathsf{a}(\mathsf{ab} + \mathsf{ac} + \mathsf{ad})$. Then,

$$\begin{aligned} \widehat{\epsilon}_E &= \mathsf{aa}, \\ \widehat{\mathsf{a}}_E &= \mathsf{aa}, \\ \widehat{\mathsf{aab}}_E &= \mathsf{aab}. \end{aligned}$$

Definition 6.3 (Reduction). If $C_E(w) = \emptyset$, the unique *reduction* $\overline{w}_E$ of w under E is the longest prefix of w with a non-empty completion:

$$\overline{w}_E = \text{longest } w' \text{ such that } w' \sqsubseteq w \wedge C_E(w') \neq \emptyset.$$

Given parse function $\mathsf{P}_E(\cdot) : \mathcal{L}[\![E]\!] \to \mathcal{B}[\![E]\!]$ for complete input strings, we can now define what it means for an implementation of it to be optimally streaming:

Definition 6.4 (Optimal streaming). The *optimally streaming* function corresponding to $\mathsf{P}_E(\cdot)$ is

$$O_E(w) = \begin{cases} \bigsqcap \{\mathsf{P}_E(w'') \mid w'' \in C_E(w)\} & \text{if } C_E(w) \neq \emptyset \\ (\bigsqcap O_E(\overline{w}))\,\sharp & \text{if } C_E(w) = \emptyset. \end{cases}$$

The first condition expresses that after seeing prefix w the function must output *all* bits that are a common prefix of all bit-coded parse trees of words in $\mathcal{L}[\![E]\!]$ that w can be extended to. That is, the partial parse tree for the longest possible word that is guaranteed to prefix the input word in case of a successful parse. The second condition expresses that as soon as it is clear that a prefix has no extension to an element of $\mathcal{L}[\![E]\!]$, an indicator $\sharp$ of failure must be emitted, with no further output after that. The bits in the bit-code for the prefix string up until it became clear that a successful path to the final state is impossible will be output, followed by the failure indicator. In this sense O_E is *optimally streaming*: it produces output bits at the semantically earliest possible time during input processing.

The function O_E is always a streaming variant of its corresponding parsing function $\mathsf{P}_E(\cdot)$:

Theorem 6.1. *For a parsing function* $\mathsf{P}_E(\cdot)$, O_E *is a streaming function:*

$$w \sqsubseteq w' \implies O_E(w) \sqsubseteq O_E(w').$$

Proof. There are four cases to consider.

- $C_E(w) \neq \emptyset$ and $C_E(w') \neq \emptyset$: By Lemma 6.1, $C_E(w') \subseteq C_E(w)$, so we must have:
$$\mathsf{P}_E(C_E(w')) \subseteq \mathsf{P}_E(C_E(w))$$
and therefore, by Lemma 2.1:
$$O_E(w) = \bigsqcap \mathsf{P}_E(C_E(w)) \sqsubseteq \bigsqcap \mathsf{P}_E(C_E(w')) = O_E(w').$$
- $C_E(w) \neq \emptyset$ and $C_E(w') = \emptyset$: Let v be the unique reduction of w': $v = \overline{w'}$. Then $w \sqsubseteq v \sqsubseteq w'$ and $C_E(v) \neq \emptyset$, so by the above case we have
$$\begin{aligned} O_E(w) \sqsubseteq O_E(v)\sharp &= \left(\bigsqcap \mathsf{P}_E(C_E(v))\right)\sharp \\ &= \left(\bigsqcap\left(\bigsqcap \mathsf{P}_E(C_E(v))\right)\right)\sharp \\ &= \left(\bigsqcap O_E(v)\right)\sharp = O_E(w'). \end{aligned}$$
- $C_E(w) = \emptyset$ and $C_E(w') \neq \emptyset$: If $C_E(w) = \emptyset$ then it must also be the case that $C_E(w') = \emptyset$, since $w \sqsubseteq w'$. Consequently, this case cannot happen.
- $C_E(w) = \emptyset$ and $C_E(w') = \emptyset$: Let v and v' be the reductions of w and w': $v = \overline{w}$, $v' = \overline{w'}$. Then $v \sqsubseteq w \sqsubseteq v' \sqsubseteq w'$ and we get $O_E(w) \sqsubseteq O_E(w')$ by combining the above cases. □

The definition of O_E has the, at first glance, surprising consequence that O_E may output bits for parts of the input it has not even read yet.

Lemma 6.2. *The completions of a word* w *under regular expression* E *equals the completions of the extended* w*:*

$$C_E(w) = C_E(\widehat{w}).$$

Proof. Follows from the fact that $\widehat{w}$ is the longest common prefix of $C_E(w)$, so it must admit the same completions as w. □

Theorem 6.2. *For a regular expression* E*, the optimally streaming output on words* w *equals the optimally streaming output on their completions:*

$$O_E(w) = O_E(\widehat{w}).$$

Proof. There are two cases: $C_E(w) \neq \emptyset$ and $C_E(w) = \emptyset$. In the latter case $\widehat{w}$ is not defined, so nothing needs to be shown. For the first case, we have, by Lemma 6.2:

$$\begin{aligned} O_E(w) &= \bigsqcap \{\mathsf{P}_E(w'') \mid w'' \in C_E(w)\} \\ &= \bigsqcap \{\mathsf{P}_E(w'') \mid w'' \in C_E(\widehat{w})\} \\ &= O_E(\widehat{w}). \end{aligned}$$ □

Example 6.2. Let $E = (\mathsf{a} + \mathsf{a})(\mathsf{a} + \mathsf{a})$. We have $O_E(\epsilon) = \mathsf{00}$; that is, O_E outputs $\mathsf{00}$ off the bat, before reading any input symbols, in anticipation of aa being the *only* possible successful extension. Assume the input is ab. After reading a, O_E does not output anything, and after reading b it outputs $\sharp$ to indicate a failed parse, the total output being $\mathsf{00}\sharp$.

6.6 Coverage

Our algorithm is based on simulating Thompson FSTs in lock-step, maintaining a set of partial paths reading the prefix w of the input that has been consumed so far. In order to be optimally streaming, we have to identify partial paths which are guaranteed not be a prefixes of a greedy parse for a word in $C_E(w)$.

In this section, we define a *coverage relation* which our parsing algorithm relies on in order to detect the aforementioned situation. In the following, fix a regular expression E and its Thompson FST $\mathcal{F}_E = (Q_E, \Sigma_E, \{\epsilon, 0, 1\}, \{q_E^{\mathsf{in}}\}, \{q_E^{\mathsf{fin}}\}, \Delta_E)$.

Definition 6.5 (Coverage). Let $p \in Q_E$ be a state and $Q \subseteq Q_E$ a state set. We say that Q *covers* p, written $Q \succcurlyeq p$, if and only if

$$\left\{\mathsf{read}(\alpha) \mid q \overset{\alpha}{\rightsquigarrow} q^{\mathsf{fin}}, q \in Q\right\} \supseteq \left\{\mathsf{read}(\beta) \mid p \overset{\beta}{\rightsquigarrow} q^{\mathsf{fin}}\right\} \tag{6.1}$$

Coverage can be seen as a slight generalization of language inclusion. That is, if $Q \succcurlyeq p$, then every word suffix read by a path from p to the final state can also be read by a path from one of the states in Q to the final state.

Let $\overline{\mathcal{F}_E}$ refer to the automaton obtained by reversing the direction of all transitions and swapping the initial and final states. It can easily be verified that if (6.1) holds for some Q and p, then the following property also holds in the *reverse* automaton $\overline{\mathcal{F}_E}$:

$$\left\{\mathsf{read}(\alpha) \mid q^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} q, q \in Q\right\} \supseteq \left\{\mathsf{read}(\beta) \mid q^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} p\right\} \tag{6.2}$$

If we consider $D(\overline{\mathcal{F}_E})$, the deterministic automaton generated from $\overline{\mathcal{F}_E}$, then we see that (6.2) is satisfied if and only if:

$$\forall S \in \mathsf{DState}_{\overline{\mathcal{F}_E}}.\ p \in S \implies Q \cap S \neq \emptyset. \tag{6.3}$$

This is true since a DFA state S is reachable by reading a word w in $D(\overline{\mathcal{F}_E})$ if and only if every $q \in S$ is reachable by reading w in $\overline{\mathcal{F}_E}$, by the subset construction. Since a DFA accepts the same language as the underlying Thompson FST, this implies that condition (6.2) holds if and only if Q has a non-empty intersection with *all* DFA states containing p.

The equivalence of (6.1) and (6.3) gives us a method to decide $\succcurlyeq$ in a Thompson FST $\mathcal{F}_E$, provided that we have computed $D(\overline{\mathcal{F}_E})$ beforehand. Checking (6.3) for a particular Q and p can be done by intersecting all states of $\mathsf{DState}_{\overline{\mathcal{F}_E}}$ with Q, using time $O(|Q||\mathsf{DState}_{\overline{\mathcal{F}_E}}|) = O(|Q|2^{O(m)})$, where m is the size of the regular expression E.

The exponential cost appears to be unavoidable—the problem of deciding coverage is inherently hard to compute:

Proposition 6.1. *The problem of deciding coverage, that is the set*

$$\{(E, Q, p) \mid Q \subseteq Q_E \wedge Q \succcurlyeq p\},$$

is PSPACE-hard.

Proof. We can reduce regular expression equivalence to coverage: Given regular expressions E and F, produce a Thompson FST $\mathcal{F}_{E+F}$ for $E + F$ and observe that $\mathcal{F}_E$ and $\mathcal{F}_F$ are subautomata. Now observe that there is a path

$$q_{E+F}^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} q_E^{\mathsf{fin}} \quad \text{(respectively } q_{E+F}^{\mathsf{in}} \overset{\beta}{\rightsquigarrow} q_F^{\mathsf{fin}}\text{)}$$

in $\mathcal{F}_{E+F}$ if and only if there is a path

$$q_E^{\mathsf{in}} \overset{\alpha'}{\rightsquigarrow} q_E^{\mathsf{fin}}$$

with $\mathsf{read}(\alpha) = \mathsf{read}(\alpha')$ in $\mathcal{F}_E$ (respectively $q_F^{\mathsf{in}} \overset{\beta'}{\rightsquigarrow} q_F^{\mathsf{fin}}$ with $\mathsf{read}(\beta) = \mathsf{read}(\beta')$ in M_F). Hence, in $\mathcal{F}_{E+F}$ we have

$$\{q_F^{\mathsf{in}}\} \succcurlyeq q_E^{\mathsf{in}} \iff \mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!].$$

Regular expression containment is PSPACE-complete [132], so therefore coverage must be PSPACE-hard. $\square$

Even after having computed a determinized automaton, the decision version of the coverage problem is still NP-complete, which we show by reduction to and from MIN-COVER, a well-known NP-complete problem [37, Chapter 35.3]. Let STATE-COVER refer to the problem of deciding membership for the language

$$\{(\mathcal{F}, D(\mathcal{F}), p, k) \mid \exists Q.\ |Q| = k \wedge p \notin Q \wedge Q \succcurlyeq p \text{ in } \mathcal{F}\}.$$

Recall that MIN-COVER is the problem of deciding membership for the language

$$\left\{(X, \mathcal{H}, k) \mid \exists \mathcal{C} \subseteq \mathcal{H}.\ |\mathcal{C}| = k \wedge X = \bigcup \mathcal{C}\right\}.$$

Proposition 6.2. *STATE-COVER is NP-complete.*

Proof. STATE-COVER $\Longrightarrow$ MIN-COVER: Let $(\mathcal{F}, D(\mathcal{F}), p, k)$ be given. Define $X = \{S \in \mathsf{DState}_{\mathcal{F}} \mid p \in S\}$ and $\mathcal{H} = \{R_q \mid q \in \bigcup X\}$ where $R_q = \{S \in X \mid q \in S\}$. Then any k-sized set cover

$$\mathcal{C} = \{R_{q_1}, ..., R_{q_k}\}$$

gives a state cover $Q = \{q_1, ..., q_k\}$ and vice-versa.

MIN-COVER $\Longrightarrow$ STATE-COVER: Let $(X, \mathcal{H}, k)$ be given, where $|X| = m$ and $|\mathcal{H}| = n$. Construct a Thompson FST $\mathcal{F}_{X,\mathcal{H}}$ over the alphabet $\Sigma = X \uplus \{\$\}$. Define its states to be the set

$$\{q^{\mathsf{in}}, q^{\mathsf{fin}}, p\} \cup \{H_1, \ldots, H_n\}.$$

For each H_i, add transitions

$$H_i \xrightarrow{\$} q^{\mathsf{fin}} \text{ and } q^{\mathsf{in}} \overset{x_{ij}}{\rightsquigarrow} H_i$$

for each $x_{ij} \in H_i$. Finally, add transitions $p \xrightarrow{\$} q^{\mathsf{fin}}$ and $q^{\mathsf{in}} \xrightarrow{x} p$ for each $x \in X$.

Observe that $D(\mathcal{F}_{X,\mathcal{H}})$ will have states $\{\{q^{\mathsf{in}}\}, \{q^{\mathsf{fin}}\}\} \cup \{S_x \mid x \in X\}$ where

$$S_x = \{H \in \mathcal{H} \mid x \in H\} \cup \{p\},$$

and $\Delta_{\mathcal{F}_{X,\mathcal{H}}}(\{q^{\mathsf{in}}\}, x) = S_x$. The time to compute $D(\mathcal{F}_{X,\mathcal{F}})$ is bounded by $O(|X||\mathcal{H}|)$. Then any k-sized state cover $Q = \{H_1, \ldots, H_k\}$ is also a set cover. $\square$

6.7 Algorithm

Our parsing algorithm produces a bit-coded parse tree from an input string w for a given regular expression E. We will simulate $\mathcal{F}_E$ in lock-step, reading a symbol from w in each step. The simulation maintains a set of all partial paths that read the prefix of w that has been consumed so far; there are always only finitely many paths to consider, since we restrict ourselves to paths without non-productive loops (non-problematic paths). When a path reaches a non-deterministic choice, it will "fork" into two paths with the same prefix. Thus, the path set can be represented as a tree of states, where the root is the initial state, the edges are transitions between states, and the leaves are the reachable states.

Definition 6.6 (Path trees). A *path tree* is a rooted, ordered, binary tree with *internal nodes* of outdegrees 1 or 2. Nodes are labeled by Thompson FST-states and edges by $\Gamma = \Sigma \cup \{\epsilon, 0, 1\}$. Binary nodes have a pair of 0- and 1-labeled edges (in this order only), respectively.

We use the following notation:

- $\mathsf{root}(T)$ is the root node of path tree T.
- $\mathsf{path}(n, c)$ is the path from n to c, where c is a descendant of n.
- $\mathsf{init}(T)$ is the path from the root to the *first* binary node reachable or to the unique leaf of T if it has no binary node.
- $\mathsf{leaves}(T)$ is the *ordered list* of leaf nodes.
- $\mathsf{Tr}_{\mathsf{empty}}$ is the empty tree.

As a notational convenience, the tree with a root node labeled q and no children is written $q\langle\cdot\rangle$, where q is a state in a Thompson FST. Similarly, a tree with a root labeled q with children l and r is written $q\langle 0 : l, 1 : r\rangle$, where q is a state in a Thompson FST and l and r are path trees and the edges from q to l and r are labeled 0 and 1, respectively. Unary nodes are labelled by $\Sigma \cup \{\epsilon\}$ and are written $q\langle \ell : c\rangle$, denoting a tree rooted at q with only one ℓ-labelled child c.

In the following we shall use T_w to refer to a path tree created after processing input word w and T to refer to path trees in general, where the input string giving rise to the tree is irrelevant.

Definition 6.7 (Path tree invariant). Let T_w be a path tree and w a word. Define $I(T_w)$ as the proposition that *all* of the following hold:

1. The $\mathsf{leaves}(T_w)$ have pairwise distinct node labels; all labels are *symbol sources*, that is states with a single symbol transition, or the accept state.
2. All paths from the root to a leaf read w:
$$\forall n \in \mathsf{leaves}(T_w).\ \mathsf{read}(\mathsf{path}(\mathsf{root}(T_w), n)) = w.$$
3. For each leaf $n \in \mathsf{leaves}(T_w)$ there exists $w'' \in C_E(w)$ such that the bit-coded parse of w'' starts with $\mathsf{write}(\mathsf{path}(\mathsf{root}(T_w), n))$.
4. For each $w'' \in C_E(w)$ there exists $n \in \mathsf{leaves}(T_w)$ such that the bit-coded parse of w'' starts with $\mathsf{write}(\mathsf{path}(\mathsf{root}(T_w), n))$.

```
Require: A Thompson FST $\mathcal{F}$, a coverage relation $\succcurlyeq$, and an input stream $S$.
Ensure: The greedy leftmost parse tree of $S$, emitted in an optimally-streaming fashion.
 1: function STREAM-PARSE($\mathcal{F}$, $\succcurlyeq$, $S$)
 2:     $w \leftarrow \epsilon$
 3:     $(T_\epsilon, \_\_) \leftarrow \textsc{closure}(\mathcal{F}, \emptyset, q^{\mathsf{in}})$  ▷ Initialize path tree as the output of CLOSURE
 4:     while $S$ has another input symbol $a$ do
 5:         if $C_E(wa) = \emptyset$ then
 6:             return $\mathsf{write}(\mathsf{init}(T_w))$ followed by $\sharp$ and exit.
 7:         end if
 8:         $T_{wa} \leftarrow \textsc{Establish-Invariant}(T_w, a, \mathcal{F}, \succcurlyeq)$
 9:         Output new bits on the path to the first binary node in $T_{wa}$, if any.
10:         $w \leftarrow wa$
11:     end while
12:     if $q^{\mathsf{fin}} \in \mathsf{leaves}(T_w)$ then
13:         return $\mathsf{write}(\mathsf{path}(\mathsf{root}(T_w), q^{\mathsf{fin}}))$
14:     else
15:         return $\mathsf{write}(\mathsf{init}(T_w))$ followed by $\sharp$
16:     end if
17: end function
```

Algorithm 1: Optimally streaming greedy regular expression parsing algorithm.

```
Require: A path tree $T_w$ satisfying invariant $I(T_w)$, a character $a$, a Thompson FST $\mathcal{F}$,
    and a coverage relation $\succcurlyeq$.
Ensure: A path tree $T_{wa}$ satisfying invariant $I(T_{wa})$.
 1: function ESTABLISH-INVARIANT($T_w$, $a$, $\mathcal{F}$, $\succcurlyeq$)
 2:     Remove leaves from $T_w$ that do not have a transition on $a$.
 3:     Extend $T_w$ to $T_{wa}$ by following all $a$-transitions.
 4:     for each leaf $n$ in $T_{wa}$ do
 5:         $(T', \_\_) \leftarrow \textsc{closure}(\mathcal{F}, \emptyset, n)$.
 6:         Replace the leaf $n$ with the tree $T'$ in $T_{wa}$.
 7:     end for
 8:     return PRUNE($T_{wa}$, $\succcurlyeq$)
 9: end function
```

Algorithm 2: Establishing invariant $I(T_{wa})$.

The path tree invariant is maintained by Algorithm 2: line 2 establishes part 1; line 3 establishes part 2; and lines 4–7 establish part 3 and 4.

Theorem 6.3 (Optimal streaming property). *Assume extended $w = \widehat{w}$, $C_E(w) \neq \emptyset$. Consider the path tree T_w after reading w upon entry into the while-loop of the algorithm in Algorithm 1. Then* $\mathsf{write}(\mathsf{init}(T_w)) = O_E(w)$.

In other words, the initial path from the root of T_w to the first binary node in T_w is the *longest common prefix* of all paths that accept words that are prefixed by w. Operationally, whenever that path gets longer by pruning branches, we output the bits on the extension.

Proof. Assume w extended, that is $w = \widehat{w}$; assume $C_E(w) \neq \emptyset$, that is there exists w'' such that $w \sqsubseteq w''$ and $w'' \in \mathcal{L}[\![E]\!]$.

```
Require: A path tree $T$ and a covering relation $\succcurlyeq$.
Ensure: A pruned path tree $T'$ where all leaves are alive.
 1: function PRUNE($T, \succcurlyeq$)
 2:     for each $l$ in $\mathsf{reverse}(\mathsf{leaves}(T))$ do
 3:         $S \leftarrow \{n \mid n$ comes before $l$ in $\mathsf{leaves}(T)\}$
 4:         if $S \succcurlyeq l$ then
 5:             $p \leftarrow \mathsf{parent}(l)$
 6:             Delete $l$ from $T$
 7:             $T \leftarrow$ CUT($T, p$)
 8:         end if
 9:     end for
10:     return $T$
11: end function
12: function CUT($T, n$)                                  ▷ Cuts a chain of 1-ary nodes.
13:     if $|\mathsf{children}(n)| = 0$ then
14:         $p \leftarrow \mathsf{parent}(n)$
15:         $T' \leftarrow T$ with $n$ removed
16:         return CUT($T', p$)
17:     else
18:         return $T$
19:     end if
20: end function
```

Algorithm 3: Pruning algorithm.

Claim: $|\mathsf{leaves}(T_w)| \geq 2$ or the unique node in $\mathsf{leaves}(T_w)$ is labeled by the accept state. Proof of claim: Assume otherwise, that is $|\mathsf{leaves}(T_w)| = 1$, but its node is not the accept state. By (1) of $I(T_w)$, this means the node must have a symbol transition on some symbol a. In this case, all accepting paths $C_E(wa) = C_E(w)$ and thus $\widehat{w} = \widehat{wa}$; in particular $\widehat{w} \neq w$, which, however, is a contradiction to the assumption that w is extended.

This means we have two cases. The case $|\mathsf{leaves}(T_w)| = 1$ with the sole node being labeled by the accept state is easy: It spells a single path from initial to accept state. By (2) and (3) of $I(T_w)$ we have that this path is correct for w. By (4) and since the accept state has no outgoing transitions, we have $C_E(w) = \{w\}$, and the theorem follows for this case.

Consider the case $|\mathsf{leaves}(T_w)| \geq 2$. Recall that $C_E(w) \neq \emptyset$ by assumption. By (4) of $I(T_w)$ the accepting path of every $w'' \in C_E(w)$ starts with $\mathsf{path}(\mathsf{root}(T_w), n)$ for some $n \in \mathsf{leaves}(T_w)$, and by (3) each path from the root to a leaf is the start of some accept path. Since $|\mathsf{leaves}(T_w)| \geq 2$ we know that there exists a binary node in T_w. Consider the first binary node on the path from the root to a leaf. It has both $\mathsf{0}$- and $\mathsf{1}$-labeled out-edges. Thus the longest common prefix of

$$\{\mathsf{write}(p) \mid n \in \mathsf{leaves}(T_w), p \in \mathsf{path}(\mathsf{root}(T_w), n)\}$$

is $\mathsf{write}(\mathsf{init}(T_w))$, the bits on the initial path from the root of T_w to its first binary node. □

The algorithm as given is only optimally streaming for extended prefixes. It can be made to work for all prefixes by enclosing it in an outer loop that for each prefix w computes $\widehat{w}$ and calls the Algorithm 1 with $\widehat{w}$. The outer loop then checks that subsequent symbols

```
Require: A Thompson FST $\mathcal{F}$, a set of visited states $V$, and a state $q$
Ensure: A path tree $T$ and a set of visited states $V'$
 1: function CLOSURE($\mathcal{F}, V, q$)
 2:     if $q \xrightarrow{0} q_l$ and $q \xrightarrow{1} q_r$ then
 3:         $(T^l, V_l) \leftarrow$ CLOSURE($\mathcal{F}, V \cup \{q\}, q_l$)        ▷ Try left option first.
 4:         $(T^r, V_{lr}) \leftarrow$ CLOSURE($\mathcal{F}, V_l, q_r$)     ▷ Use $V_l$ to skip already-visited nodes.
 5:         return $(q\langle T^l : T^r\rangle, V_{lr})$
 6:     end if
 7:     if $q \xrightarrow{\epsilon} p$ then
 8:         if $p \in V$ then                                        ▷ Stop loops.
 9:             return $(\mathsf{Tr}_{\mathsf{empty}}, V)$
10:         else
11:             $(T', V') \leftarrow$ CLOSURE($\mathcal{F}, V \cup \{q\}, p$)
12:             return $(q\langle \epsilon : T'\rangle, V')$
13:         end if
14:     else                                       ▷ $q$ is a symbol source or the final state.
15:         return $(q\langle\cdot\rangle, V)$
16:     end if
17: end function
```

Algorithm 4: Ordered ϵ-closure with path tree construction.

match until $\widehat{w}$ is reached. By Theorem 6.2, the resulting algorithm gives the right result for all input prefixes, not only extended ones.

Theorem 6.4. *The optimally streaming algorithm can be implemented to run in time*

$$O(2^{m \log m} + mn),$$

where $m = |E|$ *and* $n = |w|$.

Proof. As shown in Section 6.6, we can decide coverage in time $O(m2^{O(m)})$. The set of ordered lists $\mathsf{leaves}(T)$ for any T reachable from the initial state can be precomputed and covered states marked in it. (This requires unit-cost random access since there are $O(2^{m \log m})$ such lists.) The ϵ-closure can be computed in time $O(m)$ for each input symbol, and pruning can be amortized over ϵ-closure computation by charging each edge removed to its addition to a tree path. □

For a fixed regular expression E this is linear time in n and thus asymptotically optimal. An exponential in m as an additive preprocessing cost appears practically unavoidable since we require the coverage relation, which is inherently hard to compute (Proposition 6.1).

6.8 Examples

We revisit the example from Section 3.8. Consider the RE $(\mathrm{aaa}+\mathrm{aa})^\star$. A simplified version of its Thompson FST is shown in Figure 6.2. The following two observations are requirements for an earliest parse of this expression:

- After *one* a has been read, the algorithm *must* output a 0 to indicate that one iteration of the Kleene star has been made, but:

- *five* consecutive as determine that the leftmost possibility in the Kleene star choice was taken, meaning that the first *three* as are consumed in that branch.

The first point can be seen by noting that any parse of a non-zero number of as must follow a path through the Kleene star. This guarantees that *if* a successful parse is eventually performed, it must be the case that at least one iteration was made.

The second point can be seen by considering the situation where only four input as have been read: it is not known whether these are the only four or more input symbols in the stream. In the former case, the correct (and only) parse is two iterations with the right alternative, but in the latter case, the first three symbols are consumed in the left branch instead.

These observations correspond intuitively to what "earliest" parsing is: as soon as it is impossible that an iteration was *not* made, a bit indicating this fact is emitted, and as soon as the first three symbols *must* have been parsed in the left alternative, this fact is output. Furthermore, a 0-bit is emitted to indicate that (at least) another iteration is performed.

Figure 6.2 shows the evolution of the path tree during execution with the RE $(\mathrm{aaa}+\mathrm{aa})^\star$ on the input aaaaa.

By similar reasoning as above, after five as it is safe to commit to the left alternative after every third a. Hence, for the inputs $\mathrm{aaaaa}(\mathrm{aaa})^n$, $\mathrm{aaaaa}(\mathrm{aaa})^n\mathrm{a}$, and $\mathrm{aaaaa}(\mathrm{aaa})^n\mathrm{aa}$ the "commit points" are placed as follows ($\cdot$ indicates end-of-input):

$$\underset{0}{\mathrm{a}} \mid \underset{00}{\mathrm{aaaa}} \mid \underbrace{\left(\underset{00}{\mathrm{aaa}} \mid \cdots \mid \underset{00}{\mathrm{aaa}}\right)}_{n\text{ times}} \mid \underset{11}{\cdot} \qquad \underset{0}{\mathrm{a}} \mid \underset{00}{\mathrm{aaaa}} \mid \underbrace{\left(\underset{00}{\mathrm{aaa}} \mid \cdots \mid \underset{00}{\mathrm{aaa}}\right)}_{n\text{ times}} \mid \underset{01}{\mathrm{a}\cdot}$$

$$\underset{0}{\mathrm{a}} \mid \underset{00}{\mathrm{aaaa}} \mid \underbrace{\left(\underset{00}{\mathrm{aaa}} \mid \cdots \mid \underset{00}{\mathrm{aaa}}\right)}_{n\text{ times}} \mid \underset{1011}{\mathrm{aa}\cdot}$$

6.8.1 Complex Coverage

The previous example does not exhibit any non-trivial coverage, i.e., situations where a state n is covered by $k > 1$ other states. One can construct an expression that contains non-trivial coverage relations by observing that if each symbol source s in the FST is associated with the RE representing the language recognized from s, coverage can be expressed as a set of (in)equations in Kleene algebra (see also Section 4.1). Thus, the coverage $\{n_0, n_1\} \succcurlyeq n$ becomes $RE(n_0) + RE(n_1) \geq RE(n)$ in Kleene algebra, where $RE(\cdot)$ is the function that yields the regular expression for the language recognized from a symbol source in a Thompson FST.

Any expression of the form $x_1zy_1 + x_2zy_2 + x_3z(y_1 + y_2)$ satisfies the property that two subterms cover a third. If the coverage is to play a role in the algorithm, however, the languages denoted by x_1 and x_2 must not subsume that of x_3, otherwise the subterm (subautomaton) starting with x_3 would never play a role due to the greedy leftmost disambiguation.

Choose $x_1 = x_2 = (\mathrm{aa})^\star$, $x_3 = \mathrm{a}^\star$, $y_1 = \mathrm{a}$, and $y_2 = \mathrm{b}$. Figure 6.3 shows the expression

$$(\mathrm{aa})^\star\mathrm{za} + \mathrm{aa}^\star\mathrm{zb} + \mathrm{a}^\star\mathrm{za} + \mathrm{b} = (\mathrm{aa})^\star(\mathrm{za} + \mathrm{zb}) + \mathrm{a}^\star\mathrm{z}(\mathrm{a} + \mathrm{b}).$$

The earliest point where any bits can be output is when the z is reached. Then it becomes known whether there was an even or odd number of as. Due to the coverage $\{8, 13\} \succcurlyeq 20$,

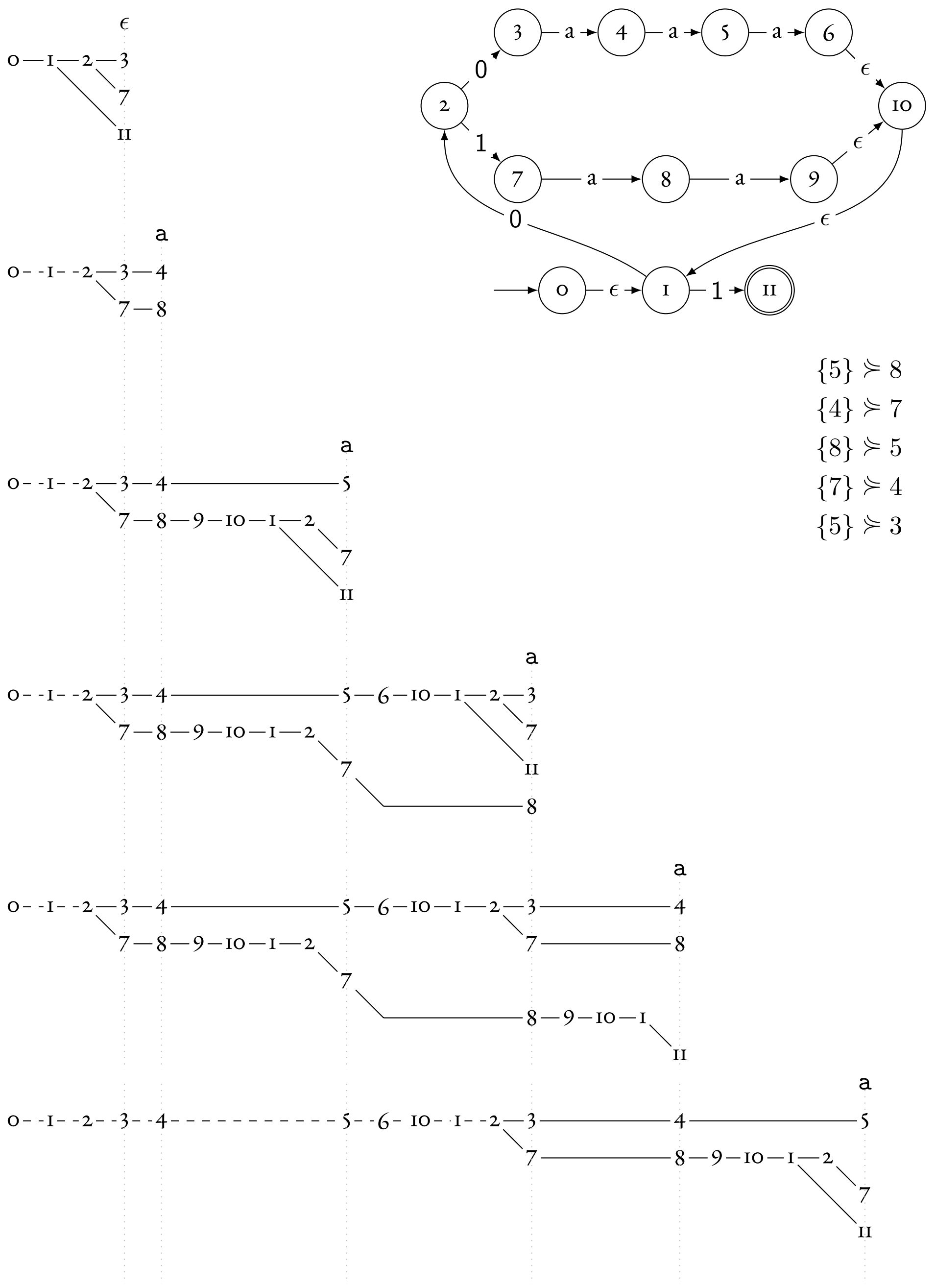


Figure 6.2: Example run of the algorithm on the regular expression $E = (\text{aaa} + \text{aa})^\star$ and the input string aaaaa. The dashed edges represent the partial parse trees that can be emitted: thus, after one a we can emit a 0, and after five as we can emit 00 because the bottom "leg" of the tree has been removed in the pruning step. The automaton for E and its associated minimal covering relation are shown in the inset.

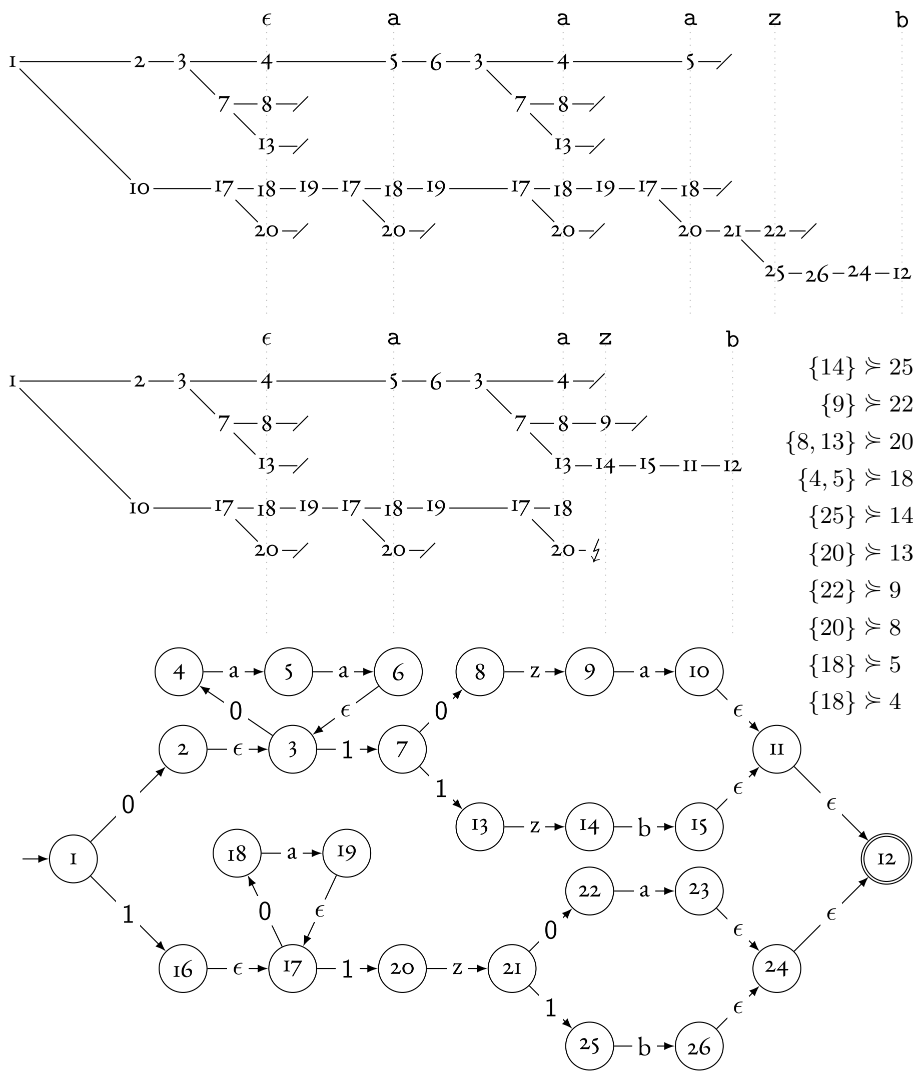


Figure 6.3: Example run of the algorithm on $E = (\mathrm{aa})^\star(\mathrm{za} + \mathrm{zb}) + \mathrm{a}^\star\mathrm{z}(\mathrm{a} + \mathrm{b})$. Note that state 20 is covered by *the combination of* states 8 and 13. The earliest time the algorithm can do a commit is when a z is encountered, which decides whether there is an even or odd number of as. The topmost figure shows the evolution of the path tree on the input aaazb. There is a long "trunk" from state 1 to state 21 after reading z, as the rest of the branches have been pruned (not shown). The desired output, corresponding to taking the rightmost option in the sum, can be read off the labels on the edges. Likewise in the second figure, we see that if the z comes after an even number of as, a binary-node-free path from 1 to 7 emerges. Due to the cover $\{8, 13\} \succcurlyeq 20$, the branch starting from 20 is not expanded further, even though there could be a z-transition on it. This is indicated with $\lightning$. Overall, the resulting parse tree corresponds to the leftmost option in the sum.

state 20 is pruned away on the input aazb, thereby causing the path tree to have a large trunk that can be output.

6.8.2 CSV Files

The expression $((a+b)^\star(;(a+b)^\star)^\star n)^\star$ defines the format of a simple semicolon-delimited data format with data consisting of words over $\{a, b\}$ and rows separated by the newline character, n. Our algorithm emits the partial parse trees after each letter has been parsed, as illustrated on the example input below:

```
a;ba;a
b;;b
```

a	;	b	a	;	a	n	b	;	;	a	n	·
000	10	01	00	10	00	11	001	10	10	00	11	1

Due to the star-height of three, many widespread implementations would not be able to meaningfully handle this expression using only the RE engine. Capturing groups under Kleene stars return either the first or last match, but not a *list* of matches—and certainly not a list of lists of matches! Hence, if using an implementation like Perl's [30], one is forced to rewrite the expression by removing the iteration in the outer Kleene star and reintroduce it as a looping construct in Perl.

6.9 Related and Future Work

Parsing regular expressions is not new [42, 57, 64, 114, 134], and streaming parsing of XML documents has been investigated for more than a decade in the context of XQuery and XPath—see, e.g., [41, 70, 152]. However, *streaming regular expression* parsing appears to be new.

In earlier work [64], presented in Chapter 5, we described a compact "lean log" format for storing intermediate information required for two-pass regular expression parsing. The algorithm presented here may degenerate to two passes, but requires often just one pass in the sense being effectively streaming, using only $O(m)$ work space, independent of n. The preprocessing of the regular expression and the intermediate data structure during input string processing are more complex, however. It may be possible to merge the two approaches using a tree of lean log frames with associated counters, observing that edges in the path tree that are *not* labeled 0 or 1 are redundant. This is an area for future work. In Chapter 7 we shall see a method for determinizing FSTs while getting mostly-optimal streaming, based on the algorithm presented here.

7 Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers

This chapter is based on the paper "Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers" that has been submitted to POPL 2016 [66].

7.1 Introduction

Imagine you want to implement syntax highlighting. This can be thought of as parsing the input into its tokens and processing each token according to its class. For illustration, assume we have one keyword, "for," and alphabetic identifiers as the only tokens. The lexical structure of the input is essentially described by the regular expression $((\text{for} + [\text{a} - \text{z}]^\star)\ )^\star$, where whitespace is, for simplicity, represented by the single blank between the two closing parentheses. This scenario highlights the following:

Ambiguity by design. The RE is ambiguous. The intended semantics is that the left alternative has higher priority than the right. This is *greedy* disambiguation: Choose the left alternative if possible, treating $E^\star$ as its *unfolding* $EE^\star + 1$. Accordingly, in our example "for" matches the left alternative, not the right.

Regular expression parsing. Note that the RE has star height two; in particular, we need to *parse* the input under multiple Kleene stars. For our RE the parse of a string is a list of segments (corresponding to the outer Kleene star), with each segment represented by a pair, a token and whitespace (corresponding to concatenation), where each token is tagged (corresponding to the alternation) to indicate that it is either the keyword "for" or an identifier; an identifier, in turn, consists of a list (corresponding to the inner Kleene star) of characters.

Output actions. We need to output something, the highlighted tokens, not just *accept* or *reject* a string as is done by finite *automata*. Note that output actions are not specified in our RE.

We would like to do the highlighting in a *streaming* fashion, using as little internal storage as possible and performing output actions as early as they are determined by the input prefix read so far, at a high sustained input processing rate, in particular in worst-case linear-time in the length of the input stream with a low factor depending linearly on the size of the RE. We would like to accomplish this automatically for *arbitrary* REs (or similar input format specification) and output actions, with speeds that in practice adapt to how much output actually needs to be produced; in particular, performance should gracefully approach pure acceptance testing as more and more output actions are removed. How?

As discussed in Chapter 3, it turns out that the set of parses are exactly the elements of the RE read as a *type* [57, 75]: Kleene star is the (finite) list type constructor, concatenation the Cartesian product, alternation the sum type and an individual character the singleton type containing that character. A *Thompson automaton* [141] represents an RE in a strong sense: the complete paths—paths from initial to final state—are in one-to-one correspondence with the parses (Theorem 5.1) [64]. If a string has four parses (e.g. "for for "), then there are exactly four complete paths accepting it. Let us look at bit closer at a Thompson automaton (Definition 3.5): It is non-deterministic, with ϵ-transitions, easily constructed, having $O(m)$ states and transitions from an RE of size m. It has exactly one initial and one accepting state. Every state is either *non-deterministic*: it has two outgoing ϵ-transitions ("left" or "right"); or it is *deterministic*: it has exactly one outgoing transition labeled by ϵ or an input symbol, or it is the final state, which has no outgoing transition. Every complete path is determined by a sequence of bits used as an oracle (Chapter 5) [64]. Starting with the initial state, follow all outgoing transitions from deterministic states; upon arriving at a non-deterministic state query the oracle to determine whether to go left or right, until the final state is reached. The bit sequence of query responses yields a prefix-free binary code for the string accepted on the designated path. This *bit-code* can also be computed directly from the RE underlying the Thompson NFA [75, 114]. The key observation is that only alternation needs to be coded by a bit; symbols, concatenation and unfolding of Kleene-star expresssions require zero bits. Since a bit-code represents a particular parse, a string can have multiple bit-codes if and only if the RE (and thus Thompson automaton) is ambiguous (Definition 3.20): The *greedy* parse of a string, which we are interested in, corresponds to the *lexicographically least* amongst its bit-codes (Chapter 5) [64].

The *greedy RE parsing problem* is producing this lexicographically least bit-code for a string matching a given RE. As we saw in Chapter 6, this can be done by an *optimally streaming* algorithm, running in time linear in the size of the input string for fixed RE [65]: The bits in the output are produced as soon as they are uniquely determined by the input prefix read so far, assuming the input string will eventually be accepted. The algorithm maintains an ordered *path tree* from the initial state to all the automata states reachable by the input prefix read so far. A branching node represents both sides of an alternation that are both still viable. The (possibly empty) path segment from the initial state to the first branching node is what can be output based on the input prefix processed so far, without knowing which of the presently reached states will eventually accept the rest of the input. This works for all regular expressions and all inputs; e.g., it automatically results in constant memory space consumption for REs that are deterministic modulo finite look-ahead, e.g. one-unambiguous REs [25].

Let us step back a bit. It is possible to aggressively ("earliest possible") and efficiently stream out the bit-code of the greedy parse of an input string under a given regular expression as the input is streaming in: worst-case linear time in the input string size, no backtracking and each input symbol can be processed in time $O(m)$, linear in the size of the RE and of its Thompson NFA. (Here it is critical that Thompson NFAs have ϵ-transitions since equivalent ϵ-free automata require $\Omega(m \log m)$ transitions [128] and standard ϵ-free NFA-constructions [9, 59, 81] even $\Omega(m^2)$.)

Coming back to our syntax highlighting problem, we can use this algorithm to parse the input, build the parse tree from the bit-code and recursively descend it to perform the syntax highlighting. We might (correctly) suspect that the highlighting can be done by piping the bit-code into a separate highlighter process, eliding the materialization of the bit-code. In this chapter we show that we can do better yet: The algorithm can be generalized to simulating arbitrary non-deterministic finite-state transducers, NFAs with output actions. Furthermore, we can compile their non-determinism away by producing theoretically and practically very efficient streaming string transducers [3–5].

7.1.1 Contributions

We present the following contributions in this chapter:

- An aggressively *streaming* algorithm for *non-deterministic finite state transducers (FSTs)* for ordered output alphabets, which emits the lexicographically least output sequence generated by all accepting paths of an input string. It runs in $O(mn)$ time, for automata of size m and inputs of size n.

- An effective determinization of FSTs into a subclass of *streaming string transducers (SST)* [3], finite state machines with string registers that are updated linearly when entering the state upon reading an input symbol. The number of registers required adapts to the number of output actions in the FST: The fewer output actions the fewer registers. In particular, without special-casing, no registers are generated—yielding a deterministic finite automata (DFA).

- An expressive declarative language, *Kleenex*, for specifying FSTs with full support for and clear semantics of unrestricted nondeterminism by greedy disambiguation. A basic Kleenex program is a context-free grammar with embedded semantic output actions, but syntactically restricted to ensure that the input is regular.[1] Basic Kleenex programs can be functionally composed into pipelines. The central technical aspect of Kleenex is its semantic support for unbridled (regular) non-determinism and its effective determinization and compilation to SSTs, thus both highlighting and complementing their significance.

- An implementation, including some empirically evaluated optimizations, of Kleenex that generates SSTs and sequential machines rendered as standard single-threaded C-code, which is eventually compiled to X86 machine code. The optimizations, which are neither conclusive nor final, illustrate the design robustness obtained by the underlying theories of ordered FSTs and SSTs.

- Use cases and examples that illustrate the expressive power of Kleenex, and a performance comparison with related tools, including Ragel [142], RE2 [139], and specialized string processing tools. These document Kleenex's consistently high performance (typically around 1 Gbps, single core, on stock hardware) even when compared to expressively more specialized tools with special-cased algorithms and tools with no or limited support for non-determinism.

7.2 Transducers

The semantics of Kleenex will be given by translation to non-deterministic *finite state transducers*, which are finite automata extended with output in a free monoid. In this section, we will recall the standard definition (see also Chapter 3 and the standard work by Berstel [18]).

[1]This facilitates avoiding the $\Omega(M(n))$ lower bound for context-free grammar parsing, where $M(n)$ is the complexity of multiplying $n \times n$ matrices.

Since Kleenex is deterministic, we also need to define a disambiguated semantics which allows us to interpret any non-deterministic transducer as a partial function, even when it may have more than one possible output for a given input string.

In the following, an *alphabet* is understood to be a finite subset $\{0, 1, ..., n-1\} \subseteq \mathbb{N}$ of consecutive natural numbers with their usual ordering. The alphabets Σ and Γ called the *input* and *output* alphabets, respectively. We recall some definitions about finite state transducers and paths through them from Chapter 3.

Definition 7.1 (Finite state transducer). A *finite state transducer* (FST) over Σ and Γ is a six-tuple

$$\mathcal{F} = (Q, \Sigma, \Gamma, q^{\mathsf{in}}, q^{\mathsf{fin}}, \Delta)$$

where

- Q is a finite set of *states*;
- $q^{\mathsf{in}}, q^{\mathsf{fin}} \in Q$ are the *initial* and *final* states, respectively;
- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ is the *transition relation*.

We write $q \xrightarrow{x|y} q'$ whenever $(q, x, y, q') \in \Delta$.

Definition 7.2. The *support* of $q \in Q$ is:

$$\mathrm{supp}(q) = \left\{ x \in \Sigma \cup \{\epsilon\} \mid \exists q', v.\, q \overset{x|v}{\rightsquigarrow} q' \right\}.$$

A *path* in $\mathcal{F}$ is a sequence of transitions (Definition 3.27):

$$q_0 \xrightarrow{x_1|y_1} q_1 \xrightarrow{x_2|y_2} \ldots \xrightarrow{x_n|y_n} q_n.$$

It has *input label* $u = x_1 x_2 ... x_n$ and *output label* $v = y_1 y_2 ... y_n$. We write $q_0 \overset{u|v}{\rightsquigarrow} q_n$ if a path from q_0 to q_n with input label u and output label v exists.

Definition 7.3 (Normalized FST). An FST $\mathcal{F}$ is *normalized* if for every state $q \in Q_{\mathcal{F}}$, either

$$\mathrm{supp}(q) = \{\epsilon\} \quad \text{or} \quad \mathrm{supp}(q) \subseteq \Sigma,$$

and $\mathrm{supp}(q^f) \subseteq \Sigma$. If $\mathrm{supp}(q) \subseteq \Sigma$ we write $q\downarrow$.

In other words, a q such that $q\downarrow$ is a state with no outgoing ϵ-transitions but possibly with outgoing symbol transitions.

The formulation of our simulation algorithm in Section 7.4 becomes simpler when restricting our attention to normalized transducers, since we can take advantage of the following separation property:

Proposition 7.1. *If $\mathcal{F}$ is normalized, then*

$$p \overset{uv|z}{\rightsquigarrow} r\downarrow,$$

for $v \neq \epsilon$, if and only if there exists a q such that $z = xy$ and

$$p \overset{u|x}{\rightsquigarrow} q\downarrow \overset{v|y}{\rightsquigarrow} r\downarrow.$$

Proof. Let a path in a normalized FST $\mathcal{F}$ be given:

$$p \xrightarrow{uv|z} r \downarrow .$$

As $v \neq \epsilon$ it must be the case that after reading u the state r from which the first symbol in v is read has a character transition. Because $\mathcal{F}$ is normalized, if r has a character transition it has only character transitions: $r \downarrow$. Therefore,

$$p \xrightarrow{u|x} q \downarrow \xrightarrow{v|y} r \downarrow .$$

The other direction follows immediately. □

Definition 7.4 (Relational semantics). An FST $\mathcal{F}$ denotes a relation

$$[\![\mathcal{F}]\!] \subseteq \Sigma^\star \times \Gamma^\star$$

with $(u, v) \in [\![\mathcal{F}]\!]$ if and only if $q^{\mathsf{in}} \xrightarrow{u|v} q^{\mathsf{fin}}$.

The relations definable as FSTs are the *rational relations* [18]. In the special case where for any $u \in \Sigma^\star$ there is at most one $v \in \Gamma^\star$ such that $(u, v) \in [\![\mathcal{F}]\!]$, the transducer computes a partial function.

Any FST $\mathcal{F}$ can be translated to an equivalent normalized FST: For any state q in $\mathcal{F}$ that has both ϵ- and character-transitions, add an extra state q', move the character transitions from q to q' and make a new ϵ-transition from q to q'. Now q has only ϵ-transitions and q' has only character transitions.

In the following we give a refined semantics which allows us to interpret any FST as denoting a partial function, using the assumed ordering on alphabets to disambiguate between outputs. As usual, our semantics requires restricting paths to be non-problematic [57] (Definition 3.28): If a path contains a non-empty loop

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

with empty input label, then the path is said to be *problematic*. If not, it is *non-problematic*. If there is a non-problematic path from q to q' with labels u, v, then we write a subscript on the arrow:

$$q \xrightarrow{u|v}_{\mathsf{np}} q' .$$

The output words (elements of $\Gamma^\star$) are lexicographically ordered. That is, $w_1 \leq w_2$ if either:

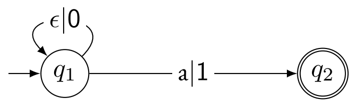
- $w_1 \sqsubseteq w_2$, w_1 is a prefix of w_2, or
- there exist words w', w_1', w_2' and symbols $b_1, b_2 \in \Gamma$ such that $w_1 = w' b_1 w_1'$, $w_2 = w' b_2 w_2'$ and $b_1 < b_2$.

We use the ordering on output words to choose a single path from a non-empty set of paths:

Definition 7.5 (Functional semantics). Any transducer $\mathcal{F}$ denotes a partial function

$$[\![\mathcal{F}]\!]_{\leq} : \Sigma^\star \to \Gamma^\star \cup \{\emptyset\}$$
$$[\![\mathcal{F}]\!]_{\leq}(u) = \min \left\{ v \mid q^{\mathsf{in}} \xrightarrow{u|v}_{\mathsf{np}} q^{\mathsf{fin}} \right\} .$$

Why the restriction to non-problematic paths? Consider the following transducer:



Then

$$\min \left\{ v \mid q_1 \overset{a|v}{\rightsquigarrow} q_2 \right\} = \emptyset,$$

as evidenced by the following infinitely descending chain of outputs: $1 \geq 01 \geq 001 \geq 0001 \geq \ldots$. Operationally, such a chain corresponds to a non-terminating backtracking search. On the other hand, the number of non-problematic paths with a given input label is always finite, ensuring well-foundedness of the lexicographic order. Every problematic path has a corresponding non-problematic path with the same input label. We therefore have the following property:

Proposition 7.2. $\mathrm{dom}(\llbracket\mathcal{F}\rrbracket_{\leq}) = \mathrm{dom}(\llbracket\mathcal{F}\rrbracket)$.

Proof. For any input $w \in \mathrm{dom}(\llbracket\mathcal{F}\rrbracket)$ the set of paths through $\mathcal{F}$ will either only contain non-problematic paths or it will contain some problematic paths. In the former case we are done, and in the latter case we can just use the fact that $w \cdot \epsilon = \epsilon \cdot w = w$ to remove the non-problematic paths. □

7.3 Kleenex

The core syntax of Kleenex is essentially that of right linear grammars extended with *output actions* and choice operators. Semantically, a Kleenex program denotes a function which transforms an input string from a regular language into a sequence of action symbols, with the caveat that if the input grammar is ambiguous, then the production rules are chosen according to a greedy leftmost disambiguation strategy.

We will first present the abstract syntax of core Kleenex, which is given a semantics in terms of the transducers introduced in Section 7.2.

Definition 7.6 (Kleenex syntax). Let Σ and Γ be two alphabets. A Kleenex program is a non-empty list $p = d_0 d_1 \ldots d_n$ of *definitions* d_i, each of the form $N_i := t_i$, where t_i is a *term* generated by the grammar:

$$t ::= 1 \mid N \mid a\,t \mid \texttt{"}b\texttt{"}\,t \mid t_0 \,|\, t_1$$

In the above, N is assumed to range over some set of non-terminal identifiers $\{N_1, \ldots, N_n\}$, $a \in \Sigma$ over *input symbols* and $b \in \Gamma$ over *output actions*.

We restrict the valid Kleenex programs to those where there is at most one definition for each non-terminal identifier.

Let p be a Kleenex program over non-terminals $\{N_1, \ldots, N_n\}$. We define a set of states Q_p and two transition relations $E_p^{\mathsf{A}}, E_p^{\mathsf{C}}$ as the smallest sets closed under the following rules:

$$\frac{}{N_1 \in Q_p} \qquad \frac{N_i \in Q_p}{t_i \in Q_p \quad (N_i, \epsilon, \epsilon, t_i) \in E_p^{\mathsf{A}} \cap E_p^{\mathsf{C}}}(N_i := t_i)$$

$$\frac{a\,t \in Q_p}{t \in Q_p \quad (a\,t, a, \epsilon, t) \in E_p^{\mathsf{C}} \quad (a\,t, \epsilon, \epsilon, t) \in E_p^{\mathsf{A}}}$$

```
main    := odd  ~/a/
         | even ~/a/
odd     := ~/aa/ "bb" odd
         | "c"
even    := ~/a/ "c" even
         | "b"
```

$$\begin{aligned} N_{\text{main}} &:= N_{\text{odd}} \mid N_{\text{even}} \\ N_{\text{odd}} &:= a\, a\, \texttt{"}b\texttt{"}\, \texttt{"}b\texttt{"}\, N_{\text{odd}} \\ &\quad \mid \texttt{"}\, c\, \texttt{"}\, a\, 1 \\ N_{\text{even}} &:= a\, \texttt{"}c\texttt{"}\, N_{\text{even}} \mid \texttt{"}b\texttt{"}\, a\, 1 \end{aligned}$$

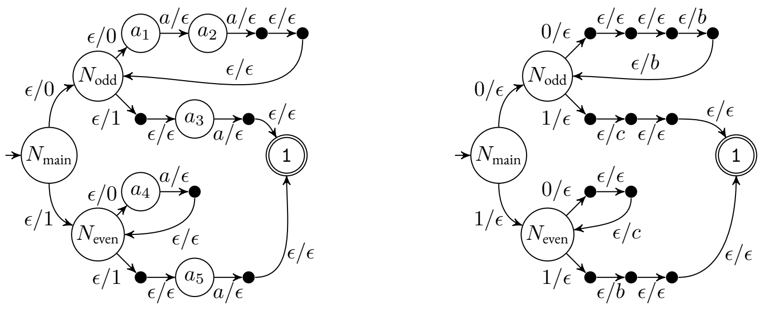


Figure 7.1: In the top left is a Kleenex program in the surface syntax and on the right is the desugared version. Below, the oracle transducer and action machine is shown, from left to right. The transduction realized by the program maps a^{2n+1} to $b^{2n}c$, and a^{2n+2} to $c^{2n}b$, respectively.

$$\frac{\texttt{"}b\texttt{"}t \in Q_p}{t \in Q_p \quad (\texttt{"}b\texttt{"}\, t, \epsilon, \epsilon, t) \in E_p^{\mathsf{C}} \quad (\texttt{"}b\texttt{"}\, t, \epsilon, b, t) \in E_p^{\mathsf{A}}}$$

$$\frac{t_0 \mid t_1 \in Q_p}{\begin{array}{ll} \{t_0, t_1\} \subseteq Q_p & (t_0 \mid t_1, \epsilon, 0, t_0), (t_0 \mid t_1, \epsilon, 1, t_1) \in E_p^{\mathsf{C}} \\ & (t_0 \mid t_1, 0, \epsilon, t_0), (t_0 \mid t_1, 1, \epsilon, t_1) \in E_p^{\mathsf{A}} \end{array}}$$

The sets are easily seen to be finite. They define two transducers, an *oracle*

$$\mathcal{F}_p^{\mathsf{C}} = (\Sigma, \{0, 1\}, Q_p, N_1, 1, E_p^{\mathsf{C}})$$

and an *action machine*

$$\mathcal{F}_p^{\mathsf{A}} = (\{0, 1\}, \Gamma, Q_p, N_1, 1, E_p^{\mathsf{A}}),$$

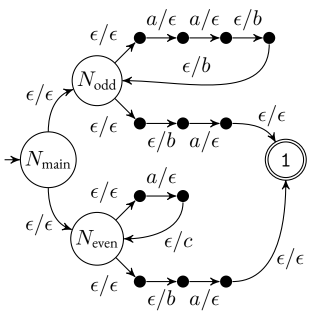
where $\mathcal{F}_p^{\mathsf{A}}$ is easily seen to be deterministic, and $\mathcal{F}_p^{\mathsf{C}}$ is non-deterministic and possibly ambiguous. Intuitively, the oracle translates an input string to a set of codes for the possible paths through p which reads the given string. The action machine translates a code to a sequence of actions.

Disambiguating according to the greedy leftmost strategy corresponds to choosing the lexicographically least code (Chapter 5), and we can thus define the semantics as follows:

Definition 7.7 (Kleenex semantics). Let p be a Kleenex program and let $\mathcal{F}_p^{\mathsf{C}}$ and $\mathcal{F}_p^{\mathsf{A}}$ be defined as above. The program p denotes a partial function

$$\begin{aligned} &[\![p]\!] : \Sigma^* \to \Gamma^* \cup \{\emptyset\} \\ &[\![p]\!] = [\![\mathcal{T}_p^{\mathsf{A}}]\!] \circ [\![\mathcal{T}_p^{\mathsf{C}}]\!]_{\leq}. \end{aligned}$$

Figure 7.1 contains a small example of the oracle and action machines. Combining the machines in the figure corresponds to following the leftmost path through this automaton:



7.3.1 Syntactic Sugar

The full syntax of our language is obtained by extending the syntax of core Kleenex with the following term-level constructors:

$$\begin{aligned} t ::= \ldots &\mid \texttt{"}v\texttt{"} \mid \texttt{/}e\texttt{/} \mid \texttt{\textasciitilde}t \mid t_0 \cdot t_1 \mid t\texttt{*} \mid t\texttt{+} \mid t\texttt{?} \\ &\mid t\texttt{\{}n\texttt{\}} \mid t\texttt{\{}n\texttt{,\}} \mid t\texttt{\{,}m\texttt{\}} \mid t\texttt{\{}n\texttt{,}m\texttt{\}} \end{aligned}$$

where $v \in \Gamma^\star$, $n, m \in \mathbb{N}$, and e is a *regular expression*. The term `"xyz..."` is just shorthand for a sequence of action symbols. The regular expressions are special versions of Kleenex terms that do not contain identifiers. They always desugar to terms that output the matched input string: The sugared term /e/ adds a default action "$\alpha(a)$" after every input symbol a in e using a given default action map $\alpha \colon \Sigma \to \Gamma$. For example, the regular expression `/a*|b{n,m}|c?/` becomes the term $(a\texttt{"}a\texttt{"})\texttt{*|}(b\texttt{"}b\texttt{"})\texttt{\{}n\texttt{,}m\texttt{\}|}(c\texttt{"}c\texttt{"})\texttt{?}$. A *suppressed* subterm is written ~t, and it desugars into t with all action symbols removed. Composition $t_0 \cdot t_1$ allows general sequential composition instead of the strict cons-like form of the core syntax. The operators ·*, ·+ and ·? desugar to their usual meaning as regular operators, as do operators $\cdot\texttt{\{}n\texttt{\}}$, $\cdot\texttt{\{}n\texttt{,\}}$, $\cdot\texttt{\{,}m\texttt{\}}$, and $\cdot\texttt{\{}n\texttt{,}m\texttt{\}}$.

By convention, the nonterminal named `main` is the entry point to a Kleenex program.

The desugaring can be described more precisely by a desugaring operator $\mathcal{D}[\![\cdot,\cdot]\!]$. The result of desugaring a program $p = d_1 \ldots d_n$ with initial term $N_1 := t_1$ is a program with initial term $N_1' := \mathcal{D}[\![t_1, 1]\!]$ which furthermore is a solution to the equations in Figure 7.2. The system does not always have a well-defined solution: The generalized composition operator of sugared Kleenex allows one to write non-regular grammars, for example:

```
main := "a" main "b" | 1
```

A program that does not have a well-defined desugaring is not considered to be well-formed, and we will not attempt to give it a semantics.

7.3.2 Custom Register Updates

The full Kleenex language as implemented in the Kleenex compiler[2] also supports *register actions*:

$$\begin{aligned} t ::= \ldots &\mid R\,\texttt{@}\,t \mid \texttt{!}R \\ &\mid \texttt{[}\,R\,\texttt{<-}\,(R \mid \texttt{"}v\texttt{"})^\star\,\texttt{]} \\ &\mid \texttt{[}\,R\,\texttt{+=}\,(R \mid \texttt{"}v\texttt{"})^\star\,\texttt{]} \end{aligned}$$

[2]https://github.com/diku-kmc/repg

$$
\begin{aligned}
\mathcal{D}[\![\texttt{1}, k]\!] &= \mathcal{D}[\![\texttt{\char`~1}, k]\!] = k \\
\mathcal{D}[\![\texttt{"}b_1 \ldots b_n\texttt{"}, k]\!] &= \texttt{"}b_1\texttt{"} \ldots \texttt{"}b_n\texttt{"}\ k \\
\mathcal{D}[\![\texttt{\char`~(}\texttt{"}b\texttt{"}\ t\texttt{)}, k]\!] &= \mathcal{D}[\![\texttt{\char`~}t, k]\!] \\
\mathcal{D}[\![a\, t, k]\!] &= a\, \mathcal{D}[\![t, k]\!] \\
\mathcal{D}[\![\texttt{\char`~(}a\, t\texttt{)}, k]\!] &= a\, \mathcal{D}[\![\texttt{\char`~}t, k]\!] \\
\mathcal{D}[\![\texttt{\char`~(}t_0 \,\texttt{|}\, t_1\texttt{)}, k]\!] &= \mathcal{D}[\![\texttt{\char`~}t_0, k]\!] \,\texttt{|}\, \mathcal{D}[\![\texttt{\char`~}t_1, k]\!] \\
\mathcal{D}[\![N, k]\!] &= N_{\mathcal{D}[\![t,k]\!]} && \text{(where } N \texttt{ := } t\text{)} \\
\mathcal{D}[\![\texttt{\char`~}N, k]\!] &= N_{\mathcal{D}[\![\texttt{\char`~}t,k]\!]} && \text{(where } N \texttt{ := } t\text{)} \\
\mathcal{D}[\![\texttt{/}e\texttt{/}, k]\!] &= \mathcal{D}[\![t_e, k]\!] \\
\mathcal{D}[\![t_0 \cdot t_1, k]\!] &= \mathcal{D}[\![t_0, \mathcal{D}[\![t_1, k]\!]]\!] \\
\mathcal{D}[\![t_0 \,\texttt{|}\, t_1, k]\!] &= \mathcal{D}[\![t_0, k]\!] \,\texttt{|}\, \mathcal{D}[\![t_1, k]\!] \\
\mathcal{D}[\![t\texttt{*}, k]\!] &= \mathcal{D}[\![t, \mathcal{D}[\![t\texttt{*}, k]\!]]\!] \,\texttt{|}\, k \\
\mathcal{D}[\![t\texttt{+}, k]\!] &= \mathcal{D}[\![t, \mathcal{D}[\![t\texttt{*}, k]\!]]\!] \\
\mathcal{D}[\![t\texttt{?}, k]\!] &= \mathcal{D}[\![t, k]\!] \,\texttt{|}\, k \\
\mathcal{D}[\![t\texttt{\{}n\texttt{\}}, k]\!] &= \mathcal{D}[\![\underbrace{t \cdot \ldots \cdot t}_{n}, k]\!] \\
\mathcal{D}[\![t\texttt{\{}n\texttt{,\}}, k]\!] &= \mathcal{D}[\![\underbrace{t \cdot \ldots \cdot t}_{n} \cdot t\texttt{*}, k]\!] \\
\mathcal{D}[\![t\texttt{\{}n\texttt{,}m\texttt{\}}, k]\!] &= \mathcal{D}[\![\underbrace{t \cdot \ldots \cdot t}_{n} \cdot \underbrace{t\texttt{?} \cdot \ldots \cdot t\texttt{?}}_{m}, k]\!]
\end{aligned}
$$

Figure 7.2: Desugaring operator. A non-terminal name N_t on the right-hand side of an equation implies the presence of a definition $N_t \texttt{ := } t$, and the term t_e corresponds to the regular expression e.

where R is a lower-case register name. They allow a rudimentary support for expressing "variables" in Kleenex programs without breaking the semantic link to right-linear grammars: Intuitively, these constructs allow one to store actions and perform them later. Writing $R \texttt{ @ } t$ redirects all actions that would have resulted from running t into the register R, which can be performed later by writing $\texttt{!}\, R$. The register R can be either set to a sequence of actions $(R \mid \texttt{"}v\texttt{"})^\star$ or appended with them, using the `<-` and `+=` construct, respectively.

Register actions were developed as part of Søholm and Tørholm's Master's thesis [131]. The semantics of register are specified in detail in their work using what they call *ordered finite action transducers* (OFATs)—here we will just outline the general ideas. The main point with OFATs is to lay bare the underlying layer of SST registers at the FST stage. As outlined in Chapter 3 and discussed later in this chapter, Thompson FSTs can be determinized into streaming string transducers, an automaton model with a concept of register. However, the registers are introduced only as artefacts of the determinization procedure, not as a reflection of a programmer's direct intention. This is the difference to OFATs: they contain a set of registers that are carried over into the SST in the SST construction. As a result, the addition of these custom register updates do not change the underlying mathematical model of

Kleenex.

An example of a Kleenex program that uses the register actions is the following that swaps two input lines by storing them in registers a and b and outputting them in reverse order:

```
main := a@line b@line !b !a
line := /[^\n]*\n/
```

7.4 Simulation and Determinization

In this section, we specify an algorithm for simulating FSTs under the functional semantics, allowing us to efficiently evaluate the oracle transducer defined in Section 7.3. We also show how the simulation algorithm can be implemented by finite deterministic *streaming string transducers* [4] whose states are identified by equivalence classes of simulation states. The latter construction gives a deterministic machine model for Kleenex programs which can be compiled to efficient code that can be executed on hardware.

We note that non-deterministic transducers are strictly more powerful than their deterministic counterparts, and can thus not always be determinized. Determinization procedures exist [16, 115] which result in a deterministic transducer with an infinite state set in the general case, and a finite state set if and only if the underlying transduction is *subsequential* [18, 129], i.e., if it is realized by a *subsequential transducer*, a transducer without ϵ-transitions and with final output words associated to the accepting states. The oracle transducers of Kleenex programs are not subsequential in general—the oracle in Figure 7.1 is an example of this. Our simulation algorithm is also different from the existing methods for determinizing transducers by also taking disambiguation into account.

In the following we fix a transducer $\mathcal{F} = (Q, \Sigma, \Gamma, q^{\mathsf{in}}, q^{\mathsf{fin}}, \Delta)$. We will assume that $\mathcal{F}$ is normalized, and that it furthermore satisfies the following property:

Definition 7.8 (Prefix-free transducer). A transducer $\mathcal{F}$ is said to be *prefix-free* if for all $p, q, q' \in Q_{\mathcal{F}}$ where $\mathrm{supp}(q), \mathrm{supp}(q') \subseteq \Sigma$ we have that if

$$p \overset{x|y}{\rightsquigarrow} q \quad \text{and} \quad p \overset{x|y'}{\rightsquigarrow} q'$$

then $y \not\sqsubset y'$.

It is easy to verify that the oracle transducers constructed Section 7.3 are both normalized and prefix-free. Note that they will always have $\Delta = \{0, 1\}$, but our construction generalizes to oracle alphabets of all sizes.

7.4.1 Generalized State Set Simulation

Let D be a finite and totally ordered set, and write $S(D, Q)$ for the set of partial functions $Q \to D^\star \cup \{\emptyset\}$. Elements $A \in S(D, Q)$ can be seen as generalized subsets of Q where every member q is labeled by some element $A(q) \in D^\star$, and every non-member has $A(q) = \emptyset$. We extend word concatenation in $D^\star$ to the set $D^\star \cup \{\emptyset\}$ by setting

$$x \cdot \emptyset = \emptyset = \emptyset \cdot x.$$

For $u, v \in D^\star \cup \{\emptyset\}$, write $u \sqsubseteq v$ if u is a *prefix* of v (Definition 2.8). Write $u \sqsubset v$ if $u \sqsubseteq v$ and $u \neq v$. Let $u \wedge v$ refer to the longest common prefix of u and v:

$$u \wedge v \overset{\text{def}}{=} \bigsqcap \{u, v\},$$

i.e., the longest p such that $u = pu'$ and $v = pv'$ for some u', v'. Note that in view of this definition, $\emptyset$ becomes a neutral element with $u \wedge \emptyset = u = \emptyset \wedge u$.

We define a *right action* on the generalized state sets as follows:

Definition 7.9 (Right action). Let $A \in S(D, Q)$ and $u \in \Sigma^\star$. The *right action* on $S(D, Q)$ is:

$$\_ \cdot \_ \colon S(D, Q) \times \Sigma^\star \to S(D \cup \Gamma, Q)$$
$$(A \cdot u)(q) = \min \left\{ A(p)v \mid p \xrightarrow{u|v}_{\mathsf{np}} q \downarrow \right\}.$$

When $D = \Gamma$ the right action is a map $S(\Gamma, Q) \times \Sigma^\star \to S(\Gamma, Q)$. It is easily seen that the right action is related to the functional semantics in the following way:

Proposition 7.3. *Let $A(q) = \epsilon$ if $q = q^{\mathsf{in}}$ and $A(q) = \emptyset$ otherwise. Then*

$$(A \cdot u)(q^{\mathsf{fin}}) = [\![\mathcal{F}]\!]_{\leq}(u).$$

A generalized subset $A \in S(D, Q)$ is *prefix-free* if $A(p) \not\sqsubseteq A(q)$ for all $p, q \in Q$ (Definition 2.10). When $\mathcal{F}$ is normalized and prefix-free, the right action preserves prefix-freeness of generalized subsets and commutes with word concatenation:

Proposition 7.4. *If $\mathcal{F}$ is normalized and prefix-free and A is prefix-free, then for all $u, v \in \Sigma^\star$,*

1. *$A \cdot u$ is prefix-free, and*
2. *$(A \cdot u) \cdot v = A \cdot uv$.*

Proof. The first property follows directly by A and $\mathcal{F}$ being prefix-free. For the second, we have for $r \in Q$,

$$\begin{aligned}
((A \cdot u) \cdot v)(r) &= \min \left\{ (A \cdot u)(q)y \mid q \xrightarrow{v|y}_{\mathsf{np}} r \downarrow \right\} \\
&= \min \left\{ \min \left\{ A(p)x \mid p \xrightarrow{u|x}_{\mathsf{np}} q \downarrow \right\} y \mid q \xrightarrow{v|y}_{\mathsf{np}} r \downarrow \right\} \\
&= \min \left\{ \min \left\{ A(p)xy \mid p \xrightarrow{u|x}_{\mathsf{np}} q \downarrow \right\} \mid q \xrightarrow{v|y}_{\mathsf{np}} r \downarrow \right\} && (7.1) \\
&= \min \left\{ A(p)xy \mid p \xrightarrow{u|x}_{\mathsf{np}} q \downarrow \xrightarrow{v|y}_{\mathsf{np}} r \downarrow \right\} && (7.2) \\
&= \min \left\{ A(p)z \mid p \xrightarrow{uv|z}_{\mathsf{np}} r \downarrow \right\} && (7.3) \\
&= (A \cdot uv)(r).
\end{aligned}$$

Equality (7.1) is a consequence of the fact that A and $\mathcal{F}$ are prefix-free and Lemma 2.2. Equality (7.2) is just associativity of minimum, and equality (7.3) follows from Proposition 7.1 and the fact that $\mathcal{F}$ is normalized. □

For $x \in D^\star$ and $A \in S(D, Q)$, define $xA \in S(D, Q)$ by $(xA)(q) = x(A(q))$. We say that x is a prefix of A if $A = xA'$ for some A', which is equivalent to x being a prefix of every $A(q)$. The right action commutes with the prefix operation:

Proposition 7.5. *Let $x \in D^\star$, then $(xA) \cdot u = x(A \cdot u)$ for all $u \in \Sigma^\star$.*

Proof. Follows from the fact that lexicographic ordering satisfies

$$\min \{xy \mid y \in Y\} = x \min Y. \qquad \square$$

7.4.2 Streaming Simulation Algorithm

A streaming simulation algorithm on $\mathcal{F}$ processes an input from left to right and may write zero or more symbols to the output in each step.

Definition 7.10 (Streaming FST simulation). Let $\mathcal{F}$ be a normalized and prefix-free transducer, and let the input $u = a_1 a_2 \cdots a_n$ be given. Let $A_0 \in S(\Gamma, Q)$ be defined as in Proposition 7.3. Reading symbol a_i, compute $B_i = A_i \cdot a_{i+1}$. Append $p_i = \bigwedge_{q \in Q} B_i(q)$ to the output stream and set $A_{i+1} = B_i'$, where the equality $B_i = p_i B_i'$ defines B_i'. When there are no more input symbols left, append $(A_n \cdot \epsilon)(q^{\mathsf{fin}})$ to the output and return, or fail if $(A_n \cdot \epsilon)(q^{\mathsf{fin}}) = \emptyset$.

By Propositions 7.3, 7.4, and 7.5, the algorithm computes $[\![\mathcal{F}]\!]_{\leq}(u)$. This algorithm is essentially the same as Algorithm 1 from Chapter 6, with the caveat that no coverage relation is used here. Therefore, it is not *optimally* streaming in the sense of Definition 6.4, which Algorithm 1 is. However, the full coverage relation is expensive to compute (Proposition 6.1), and we conjecture that it will only make a difference in pathological cases, such as the example in Figure 6.3.

7.4.3 A Deterministic Computation Model

We wish to translate Kleenex programs to completely deterministic programs without a simulation overhead.

As discussed above, the oracle transducers that are constructed from Kleenex programs are not subsequential in general, and so they cannot be determinized to finite state transducers [18, 129].

We turn instead to *streaming string transducers* [4] (SSTs), a deterministic model of computation which generalizes subsequential transducers by allowing copy-free updates to a finite set of word registers. It turns out that every transducer that can be simulated by our generalized state set algorithm can be expressed as an SST.

Definition 7.11 (Streaming string transducer [3–6]). A deterministic *streaming string transducer* (SST) over alphabets Σ and Γ is a structure

$$(Q, \Sigma, \Gamma, X, q^{\mathsf{in}}, F, \delta^1, \delta^2),$$

where

- Q is is a finite set of *states*;
- X is a finite set of *register variables*;
- $q^{\mathsf{in}} \in Q$ is the *initial state*;
- $F : Q \rightharpoonup (\Gamma \cup X)^\star \cup \{\emptyset\}$ is a partial function mapping each *final state* $q \in \mathrm{dom}(F)$ to a word $F(q) \in (\Gamma \cup X)^\star$ such that for each q, each $x \in X$ occurs at most once in $F(q)$;

- $\delta^1 \colon Q \times \Sigma \to Q$ is the *transition function*;
- $\delta^2 \colon Q \times \Sigma \times X \to (\Gamma \cup X)^\star$ is the *register update* function such that for each $q \in Q$, $a \in \Sigma$ and $x \in X$, there is at most one $y \in X$ such that x occurs in $\delta^2(q, a, y)$.

The semantics are defined as follows.

Definition 7.12 (Configuration, valuation of SST). A *configuration* of an SST $\mathcal{S}$ is a pair (q, ρ) where $q \in Q_\mathcal{S}$ is a state, and

$$\rho \colon X_\mathcal{S} \to \Gamma^\star$$

is a *valuation*. A valuation extends as a monoid homomorphism to a map

$$\widehat{\rho} \colon (X_\mathcal{S} \cup \Gamma)^\star \to \Gamma^\star$$

by setting $\rho(x) = x$ for $x \in \Gamma$. The *initial configuration* is $(q^{\mathsf{in}}, \rho^{\mathsf{in}})$ where $\rho^{\mathsf{in}}(x) = \epsilon$ for all $x \in X_\mathcal{S}$.

A configuration steps to a new configuration given an input symbol:

$$\delta_\mathcal{S}((q, \rho), a) \overset{\text{def}}{=} (\delta^1_\mathcal{S}(q, a), \rho'),$$

where $\rho'(x) = \widehat{\rho}(\delta^2_\mathcal{S}(q, a, x))$. The transition function extends to a transition function $\delta^\star_\mathcal{S}$ on words by

$$\begin{aligned} \delta^\star_\mathcal{S}((q, \rho), \epsilon) &= (q, \rho) \\ \delta^\star_\mathcal{S}((q, \rho), au) &= \delta^\star_\mathcal{S}(\delta_\mathcal{S}((q, \rho), a), u). \end{aligned}$$

With these definitions, we can formulate the semantics of an SST.

Definition 7.13 (SST semantics). The partial function denoted by an SST $\mathcal{S}$ is:

$$\llbracket \mathcal{S} \rrbracket \colon \Sigma^\star \to \Gamma^\star \cup \{\emptyset\}$$

$$\llbracket \mathcal{S} \rrbracket(u) = \begin{cases} \widehat{\rho'}(F_\mathcal{S}(q')) & \text{if } \delta^\star((q^{\mathsf{in}}, \rho^{\mathsf{in}}), u) = (q', \rho') \\ & \text{and } q' \in \mathrm{dom}(F_\mathcal{S}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

7.4.4 Tabulation

We need to come up with a representation of our streaming simulation algorithm as an SST with a designated register used for streaming output. Our representation needs to satisfy the property of being finite state as well as the property that the *output register* contains the output $p_1 p_2 ... p_i$ of the algorithm in Definition 7.10 after reading the ith input symbol a_i. The latter requirement means that we must somehow statically encode the prefix structure of all potential outputs in the states of the SST, since SSTs cannot access the contents of registers. It turns out that this is possible by letting the states of the SST be equivalence classes of generalized state sets, where the equivalence relates state sets that agree on state ordering and prefix structure.

Trees

We will call a prefix-free generalized state set A an *ordered tree* with node set

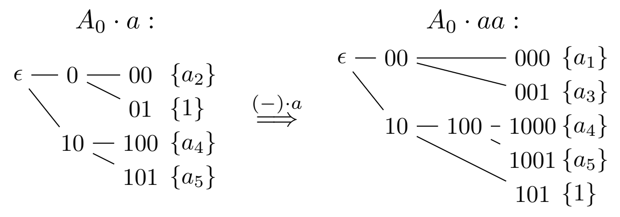
$$N_A = \{A(p) \wedge A(q) \mid p, q \in Q, A(p) \wedge A(q) \neq \emptyset\}\,.$$

Under this view, the leaves of A seen as a tree is the subset of nodes

$$L_A = \{A(q) \mid q \in Q, A(q) \neq \emptyset\} \subseteq N_A,$$

and the leaves are labeled by $A^{-1}\colon L_A \to \{0,1\}^Q$. Because A is assumed prefix-free, we have for any nodes $x, y \in N_A$ that $x \sqsubseteq y$ if and only if there is a $z \in N_A$ such that $x = y \wedge z$. In this case x is called an *ancestor* of y and z, which in turn are called the *descendants* of x. Importantly, the root node of any (sub)tree is the longest common prefix of its descendants.

Example 7.1. We illustrate the tree interpretation as follows. Consider the oracle transducer from Figure 7.1. Let A_0 be the generalized state set that maps N_{main} to ϵ and every other state to $\emptyset$. Then the state sets $A_0 \cdot a$ and $A_0 \cdot aa$ can be seen as trees in the following way:



We will consider two generalized state sets to be equivalent if they are indistinguishable as ordered trees.

Definition 7.14 (Ordered tree isomorphism). Let D_1, D_2 be totally ordered and let $A_1 \in S(D_1, Q)$ and $A_2 \in S(D_2, Q)$ be trees. An *ordered tree isomorphism* between A_1 and A_2 is a bijective map $h\colon N_{A_1} \to N_{A_2}$ such that for all $p, q \in Q$:

1. $h(A_1(p) \wedge A_1(q)) = A_2(p) \wedge A_2(q)$; and
2. $A_1(p) \leq A_1(q)$ if and only if $A_2(p) \leq A_2(q)$.

We write $h\colon A_1 \equiv A_2$ and say that A_1 and A_2 are *equivalent* when h is an ordered tree isomorphism between A_1 and A_2. Tree equivalence is preserved by the right action:

Proposition 7.6. *If $A \in (D_1, Q), B \in (D_2, Q)$ and $h\colon A \equiv B$ then for all $a \in \Sigma$, we have $A \cdot a \equiv B \cdot a$.*

Proof sketch. Since h is an order isomorphism and since A and B are prefix-free, we have for all $q \in Q$ exists $p_q \in Q$ and $y_q \in \Gamma^\star$ such that $(A \cdot a)(q) = A(p_q)y_q$ and

$$(B \cdot a)(q) = h(A(p_q))y_q.$$

Observe that for any $n \in N_{A \cdot a}$ there exists $q_1, q_2 \in Q$ such that

$$\begin{aligned} n &= (A \cdot a)(q_1) \wedge (A \cdot a)(q_2) \\ &= \begin{cases} A(q_1)(y_{q_1} \wedge y_{q_2}) & \text{if } A(q_1) = A(q_2) \\ A(q_1) \wedge A(q_2) & \text{otherwise} \end{cases} \end{aligned}$$

Furthermore, there does not exist $q_1, q_2, r_1, r_2 \in Q$ such that

$$A(q_1)(y_{q_1} \wedge y_{q_2}) = A(r_1) \wedge A(r_2),$$

since that would imply that $A(q_1)$ is a prefix of $A(r_1)$ and $A(r_2)$. We define a map such that for all $q_1, q_2 \in Q$,

$$h' \colon N_{A \cdot a} \to N_{B \cdot a}$$

$$h'((A \cdot a)(q_1) \wedge (A \cdot a)(q_2)) = \begin{cases} h(A(q_1)(y_{q_1} \wedge y_{q_2})) & \text{if } A(q_1) = A(q_2) \\ h(A(q_1) \wedge A(q_2)) & \text{otherwise.} \end{cases}$$

This is a well-defined function by the previous observations, and a tree isomorphism by the fact that h is a tree isomorphism. □

Canonical Representatives

Definition 7.15 (Canonical generalized set). A generalized set $A \in S(D, Q)$ is *canonical* if

1. $\mathrm{rng}(A)$ is *prefix closed*: if $y \in \mathrm{rng}(A)$ and $x \sqsubseteq y$ then $x \in \mathrm{rng}(A)$; and
2. $\mathrm{rng}(A)$ is *downwards closed*: if $xb \in \mathrm{rng}(A)$ for $b' < b$ then $xb' \in \mathrm{rng}(A)$ (for $b, b' \in \Gamma$).

Write $\widetilde{S}(D, Q)$ for the subset of canonical trees. The set is finite, as every canonical tree A has a prefix closed node set, so the longest word in N_A is bounded by $|\mathrm{dom}(A)| - 1$ (the maximum depth of a tree with $|\mathrm{dom}(A)|$ leaves).

Any tree has a canonical representative:

Proposition 7.7. *For any set D and tree $A \in S(D, Q)$, there is a unique $C \in \widetilde{S}(\mathbb{N}, Q)$ with $A \equiv C$.*

As a consequence, there is a reduction map

$$[\cdot] \colon S(D, Q) \to \widetilde{S}(\mathbb{N}, Q)$$

such that $A \equiv B$ if and only if $[A] = [B]$, implying that the quotient set $S(D, Q)/\equiv$ must be finite. Any $A \in S(D, Q)$ is thus canonically represented by a homomorphism $h_A \colon N_{[A]} \to N_A$ such that $A = h_A \circ [A]$.

In view of Proposition 7.6, this means that we can statically enumerate all possible trees up to tree isomorphism by computing with the canonical representatives. Any concrete tree reachable by the simulation algorithm is an instance of a canonical tree composed with a suitable homomorphism. An SST implementing the simulation algorithm can thus take the set of canonical trees as its states, and will then need to maintain the associated homomorphism via register updates.

Paths

We need to represent tree homomorphisms using SST registers such that the effect of computing right actions on the underlying tree can be expressed as SST updates.

For a tree $A \in S(D, Q)$, any node $x \in N_A$ has a unique maximal decomposition $x = x_0 x_1 \cdots x_n$ such that each $x_0 x_1 \cdots x_i \in N_A$ for all $0 \leq i \leq n$. Intuitively, this reflects the full path from the root node to the node x, and we can define the map

$$\begin{aligned} \mathsf{path}_A &: N_A \to N_A^\star \\ \mathsf{path}_A(x) &= (x_0, x_0 x_1, \ldots, x_0 x_1 \cdots x_n), \end{aligned}$$

which maps nodes to their maximal path decomposition (we use the tuple notation to distinguish between the two levels of monoids). In view of this and the fact that homomorphisms must preserve descendants, for any homomorphism $h \colon A \equiv B$ there is a unique $\kappa_h \colon N_A \to N_B$ such that

$$h(x) = \kappa_h(t_0)\kappa_h(t_1) \cdots \kappa_h(t_n), \tag{7.4}$$

where $\mathsf{path}_A(x) = (t_0, t_1, ..., t_n)$. Intuitively, κ can be seen as a "differential" representation of h, representing the change of h between a node and its immediate ancestor. By viewing κ_h as a map $N_A \to D_B^\star$ which extends uniquely to a monoid homomorphism

$$\widehat{\kappa_h} \colon N_A^\star \to D_B^\star,$$

we obtain $h = \widehat{\kappa_h} \circ \mathsf{path}_A$. Considering the unique isomorphism $h_A \colon [A] \equiv A$, write κ_A for the associated decomposition satisfying (7.4), and we thus have

$$A = \widehat{\kappa_A} \circ \mathsf{path}_{[A]} \circ [A]. \tag{7.5}$$

The path-operator is easily seen to be a tree isomorphism since it preserves node ordering and prefix structure. That is, for any $A \in S(D, Q)$, we have $\mathsf{path}_A \colon A \equiv A^\sharp$ where $A^\sharp \in S(N_D, Q)$ is defined by $A^\sharp = \mathsf{path}_A \circ A$. Using this notation, (7.5) becomes

$$A = \widehat{\kappa_A} \circ [A]^\sharp. \tag{7.6}$$

SST Construction

We construct an SST implementing the FST simulation algorithm and sketch a proof of its correctness.

Theorem 7.1. *For any normalized prefix-free transducer $\mathcal{F} = (Q, \Sigma, \Gamma, q^{\mathsf{in}}, q^{\mathsf{fin}}, \Delta)$, there is an SST $\mathcal{S}$ such that $[\![\mathcal{S}]\!] = [\![\mathcal{F}]\!]_{\leq}$.*

Proof. We define $\mathcal{S}$ as follows. Let A_0 be defined as in Definition 7.10, and observe that $A_0 \in \widetilde{S}(\mathbb{N}, Q)$. The states are the canonical trees labeled by Q:

$$\begin{aligned} Q_{\mathcal{S}} &= \{[A] \mid A \in S(\Gamma, Q)\} \cup \{A_0\} \subseteq \widetilde{S}(\mathbb{N}, Q), \\ q_{\mathcal{S}}^{\mathsf{in}}(q) &= A_0(q) \end{aligned}$$

The registers will be identified by canonical tree nodes:

$$X_{\mathcal{S}} = \bigcup \{N_C \mid C \in Q_{\mathcal{S}}\}.$$

The final output and the transition maps are given as follows:

$$\begin{aligned}
F_{\mathcal{S}}(C) &= (C^\sharp \cdot \epsilon)(q^{\mathsf{fin}}),\\
\delta^1_{\mathcal{S}}(C, a) &= [C \cdot a],\\
\delta^2_{\mathcal{S}}(C, a, x) &= \begin{cases} \kappa_{C^\sharp \cdot a}(x) & \text{if } x \in N_{[C^\sharp \cdot a]}\\ \epsilon & \text{otherwise}\end{cases}
\end{aligned}$$

We claim that $\mathcal{S}$ computes the same function as $\mathcal{F}$ under the functional semantics.

For $u \in \Sigma^\star$ let (C_u, ρ_u) refer to the value $\delta^\star_{\mathcal{S}}((q^{\mathsf{in}}_{\mathcal{S}}, \rho^{\mathsf{in}}), u) = (C_i, \rho_i)$. We show that for any $u \in \Sigma^\star$, we have $\widehat{\rho_u} \circ (C^\sharp_u \cdot \epsilon) = A_0 \cdot u$. Suppose that this holds. Then for any $u \in \Sigma^\star$, we have by the above and Proposition 7.3 that

$$[\![\mathcal{S}]\!](u) = \widehat{\rho_u}(F_{\mathcal{S}}(C_u)) = \widehat{\rho_u} \circ (C^\sharp_u \cdot \epsilon)(q^{\mathsf{fin}}) = (A_0 \cdot u)(q^f) = [\![\mathcal{F}]\!]_{\leq}(u).$$

Our claim follows as a special case of Lemma 7.1. □

Lemma 7.1. *Let* $A \in S(\Gamma, Q)$ *and* $\rho \colon X_{\mathcal{S}} \to \Gamma^\star$ *such that* $A = \widehat{\rho} \circ [A]^\sharp$. *Then for any* $u \in \Sigma^+$ *with* $\delta^\star_{\mathcal{S}}(([A], \rho), u) = (C, \rho')$ *we have* $\widehat{\rho'} \circ C^\sharp = A \cdot u$.

Proof. By induction on u. For $u = a$ we have $C = [[A] \cdot a] = [A \cdot a]$ and $\rho' = \widehat{\rho} \circ \kappa_{[A]^\sharp \cdot a}$. We can easily verify that $\widehat{\rho'} = \widehat{\rho} \circ \widehat{\kappa_{[A]^\sharp \cdot a}}$ so for any $q \in Q$,

$$\begin{aligned}
\widehat{\rho'} \circ [A \cdot a]^\sharp(q) &= \widehat{\rho} \circ \widehat{\kappa_{[A]^\sharp \cdot a}} \circ [A \cdot a]^\sharp(q)\\
&= \widehat{\rho} \circ ([A]^\sharp \cdot a)(q)\\
&= \widehat{\rho}\left(\min\left\{[A]^\sharp(p)y \mid p \overset{a|y}{\rightsquigarrow}_{\mathsf{np}} q \downarrow\right\}\right)\\
&= \min\left\{\widehat{\rho} \circ [A]^\sharp(p)y \mid p \overset{a|y}{\rightsquigarrow}_{\mathsf{np}} q \downarrow\right\}\\
&= \min\left\{A(p)y \mid p \overset{a|y}{\rightsquigarrow}_{\mathsf{np}} q \downarrow\right\}\\
&= (A \cdot a)(q).
\end{aligned}$$

The second equality follows by observing that $A \equiv [A] \equiv [A]^\sharp$, so by Proposition 7.6, we have $A \cdot a \equiv [A]^\sharp \cdot a$ and thus $[A \cdot a] = [[A]^\sharp \cdot a]$. Therefore,

$$\widehat{\kappa_{[A]^\sharp \cdot a}} \circ [A \cdot a]^\sharp = \widehat{\kappa_{[A]^\sharp \cdot a}} \circ [[A]^\sharp \cdot a]^\sharp = [A]^\sharp \cdot a$$

by using the identity (7.6). The fourth equality is justified by the fact that $[A]^\sharp(p) \leq [A]^\sharp(q)$ if and only if $A(p) \leq A(q)$.

For $u = au'$ where $u' \neq \epsilon$, we have $(C, \rho') = \delta^\star_{\mathcal{S}}(([A \cdot a], \widehat{\rho} \circ \kappa_{[A]^\sharp \cdot a})$. By the previous argument we can apply the induction hypothesis, and we obtain $C = [(A \cdot a) \cdot u']$ and $\widehat{\rho'} \circ C^\sharp = (A \cdot a) \cdot u'$. The result then follows by Proposition 7.4. □

Example 7.2. We illustrate how the construction works by showing how Example 7.1 is implemented as an SST update between states $[A_0 \cdot a]$ and $[A \cdot aa]$. The register update is obtained by computing $\kappa_{[A_0 \cdot a]^\sharp \cdot a}$. The tree $[A_0 \cdot a]^\sharp$ looks as follows:

$$\begin{tikzcd}
(\overline{\epsilon}) \arrow[r, dash] \arrow[ddr, dash] & (\overline{\epsilon}, \overline{0}) \arrow[r, dash] \arrow[dr, dash] & (\overline{\epsilon}, \overline{0}, \overline{00})\,\{a_2\} \\
& & (\overline{\epsilon}, \overline{0}, \overline{01})\,\{1\} \\
& (\overline{\epsilon}, \overline{1}) \arrow[r, dash] \arrow[dr, dash] & (\overline{\epsilon}, \overline{1}, \overline{10})\,\{a_4\} \\
& & (\overline{\epsilon}, \overline{1}, \overline{11})\,\{a_5\}
\end{tikzcd}$$

Recall that each node is a full path in the canonical tree $[A_0 \cdot a]$. The node names from $N_{[A_0 \cdot a]}$ are overlined and elements of the path monoid $N^{\star}_{[A_0 \cdot a]}$ are written $(x_1, x_2, \ldots)$. The tree $[A_0 \cdot a]^{\sharp} \cdot a$ looks as follows:

$$\begin{tikzcd}
(\overline{\epsilon}) \arrow[r, dash] \arrow[ddr, dash] & (\overline{\epsilon}, \overline{0}, \overline{00}) \arrow[rr, dash] \arrow[drr, dash] & & (\overline{\epsilon}, \overline{0}, \overline{00}, 0)\,\{a_1\} \\
& & & (\overline{\epsilon}, \overline{0}, \overline{00}, 1)\,\{a_3\} \\
& (\overline{\epsilon}, \overline{1}) \arrow[r, dash] \arrow[ddrr, dash] & (\overline{\epsilon}, \overline{1}, \overline{10}) \arrow[r, dash] \arrow[dr, dash] & (\overline{\epsilon}, \overline{1}, \overline{10}, 0)\,\{a_4\} \\
& & & (\overline{\epsilon}, \overline{1}, \overline{10}, 1)\,\{a_5\} \\
& & & (\overline{\epsilon}, \overline{1}, \overline{11})\ \{1\}
\end{tikzcd}$$

Note that symbols that are not overlined are output symbols from Γ. The map

$$\kappa' = \kappa_{[A_0 \cdot a]^{\sharp} \cdot a} \colon N_{[A_0 \cdot aa]} \to (N_{[A_0 \cdot a]} \cup \Gamma)^{\star}$$

gives us the relevant SST update strings:

$$\begin{array}{rrr}
\kappa'(\overline{\epsilon}) = (\overline{\epsilon}) & \kappa'(\overline{0}) = (\overline{0}, \overline{00}) & \kappa'(\overline{00}) = 0 \\
\kappa'(\overline{01}) = 1 & \kappa'(\overline{1}) = (\overline{1}) & \kappa'(\overline{10}) = (\overline{10}) \\
\kappa'(\overline{100}) = 0 & \kappa'(\overline{101}) = 1 & \kappa'(\overline{11}) = (\overline{11})
\end{array}$$

The full construction of an SST from the oracle transducer in Figure 7.1 can be seen in Figure 7.3.

7.5 Implementation

Our implementation compiles a Kleenex program to machine code by implementing the transducer constructions described in the earlier sections. We have also implemented several optimizations to decrease the size of the generated SSTs and improve the performance of the generated code. We will briefly describe these in the following section, and we note that they are all orthogonal to the underlying principles behind our compilation.

The possible compilation paths of our implementation can be seen in Fig. 7.4.

7.5.1 Transducer Pipeline

It is possible to chain together several Kleenex programs in a pipeline, letting the output of one serve as the input of the next. This can for example be used to strip unwanted characters before performing a transformation. By using the optional *pipeline pragma*:

$$\texttt{start:}\ t_1 \texttt{>>} \ldots \texttt{>>} t_n,$$

a programmer can specify that the entry point to the program is t_1 (instead of `main`) and that the output should be chained together as specified, with the final output being that of t_n. The current implementation does this by building separate transducers for each t_i running them as separate processes connected with UNIX pipes.

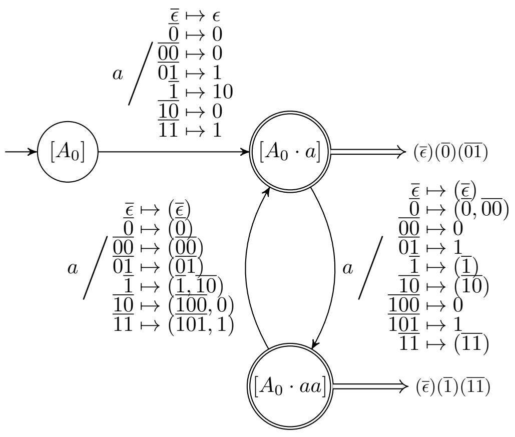


Figure 7.3: Example of SST computing the same function as the oracle transducer in Figure 7.1. Each transition is tagged by a register update, and the nodes of the canonical tree identifying the destination state make up the registers. The wide arrows exiting the accepting states indicate the final output string. Note that this always includes the root variable $(\overline{\epsilon})$ which thus acts as an interface for streaming output (although for this particular example, nothing can output until the end of the input).

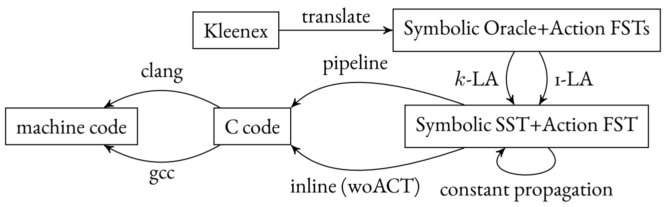


Figure 7.4: Compilation paths. 1-LA is symbolic SST construction with single-symbol transitions; k-LA is construction of SST with up to k symbols of lookahead for some k determined by the program. The "pipeline" translation path indicates that the resulting program keeps the oracle SST and action FST separate, with data being piped from the SST to the FST at runtime. The "inline" path indicates that the action FST is fused into the oracle SST. Programs compiled with this path have the suffix "woACT" in the performance plots.

7.5.2 Inlining the Action Transducer

When we have constructed the oracle SST we end up with two deterministic machines which need to be composed. We can either do this at runtime by piping the output of the oracle SST into the action FST, or we can apply a form of *deforestation* to inline the output of the action FST directly in the SST (this is straightforward since the action FST is deterministic by construction). The former approach is advantageous if the Kleenex program produces a lot of output and is highly nondeterministic. We have implemented both approaches, and discuss their strengths and weaknesses in Section 7.6.3.

7.5.3 Constant Propagation

The SSTs generated by our construction contains quite a lot of trivial register updates which can be eliminated in order to achieve better run-time efficiency. Consider the SST in Fig. 7.3, where all registers but $(\bar{0})$ and $(\bar{1})$ are easily seen to have a constant known value in each state. Eliminating the redundant registers means that we only have to maintain two registers at run-time.

We achieve this by *constant propagation*: computing reaching definitions by solving a set of data-flow constraints. As this is a quite standard technique, we will discuss it further here, but refer to existing text books [10].

7.5.4 Symbolic Representation

Text transformation programs often contain idioms with a very redundant representation as pure transducers. For example, a program might match against a whole range of characters and proceed in the same way regardless of which one was matched. This will, however, lead to a transition for each concrete character in the generated FST, even though all transitions have the same source and destination states. For large ranges, like all ASCII characters, this causes poor performance.

A more succinct representation can be obtained by using a symbolic representation of the transition relation by introducing transitions whose input labels are *predicates*, and whose output labels are *terms* indexed by input symbols. Replacing input labels with predicates was described by Watson [148]. Such symbolic transducers have been developed further and have recently received a lot of attention, with applications in verification and verifiable string transformations [115, 144, 146, 147].

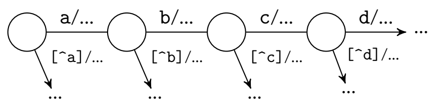
Our implementation of Kleenex uses a symbolic representation for basic ranges of symbols in order to get rid of most redundancies. Both the simulation algorithm and the SST construction can be generalized in a fairly straight-forward way to the symbolic case without altering the fundamental structure, so we have omitted the details of this optimization. Instead, we refer the reader to the cited literature for the technical details of symbolic transducers.

7.5.5 Finite Lookahead

A common pattern in Kleenex programs are definitions of the form

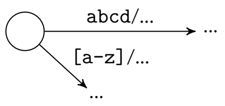
```
token := ~/abcd/ commonCase | ~/[a-z]+/ fallback
```

that is, a specific pattern appearing with higher priority than a more general fallback pattern. Patterns of this form will result in (symbolic) SSTs containing the following kind of structure:



The primary case and the fallback pattern are simulated in lockstep, and in each state there is a transition for when the common case fails after reading 0, 1, 2, etc., symbols. The problem is that we are effectively *specializing* the fallback pattern to all of the prefixes `a`, `ab`, and `abc`. With constant propagation applied, this can result in successively longer output strings from the fallback case being present in the updates on the transitions along the primary branch, leading to a quadratic size blow-up.

If the SST was able to look more than one symbol ahead before determining the next state, we would be able to tabulate a much coarser set of simulation states and do away with the fine-grained interleaving. For the above example, we would like a transition structure like the following:



If the first four symbols of the input are `abcd`, the upper transition is taken. If this is not the case, but the first symbol is `a`, then the lower transition is taken. The idea is that any string successfully matched by the primary case will satisfy the test `abcd`, so if the transition with `[a-z]` is taken, then the FST states corresponding to the primary case can be removed from the generalized state set and tabulation can continue with a simpler simulation state.

The semantics of SSTs with lookahead are still deterministic despite the seeming overlap of patterns, as the model requires that any pair of tests are either disjoint (no string will satisfy both at the same time), or one test is completely contained in another (if a string satisfies the first test, it also satisfies the second). This restriction gives a total order between tests, specifying their priority—the most specific test must be tried first.

7.6 Benchmarks

We have run comparisons with different combinations of the following tools:

RE2, Google's automata-based regular expression C++ library [139].

RE2J, a recent re-implementation of RE2 in Java [140].

GNU **`AWK`**, GNU **`grep`**, and GNU **`sed`**, programming languages and tools for text processing and extraction [136].

Oniglib, a regular expression library written in C++ with support for different character encodings [86].

Ragel, a finite state machine compiler with multiple language backends [142].

In addition, we implemented test programs using the standard regular expression libraries in the scripting languages Perl [30], Python [105], and Tcl [149]. Version numbers of the tested software are shown in Table 7.1.

Name	Version
gcc	4.8.4
clang	3.5.0
Perl	5.20.1
Python	2.7.9
Tcl	8.6.3
GNU `AWK`	4.0.2
GNU `grep`	2.21
GNU `sed`	4.2.1
GNU coreutils	8.21
Oniguruma	5.9.6
RE2	GitHub: `bdb5058`
RE2J	1.0
Ragel	6.9
DRex	20150114

Table 7.1: Version numbers of tools, libraries, etc.

Meaning of plot labels. The plot labels for Kleenex programs indicate the compilation path. They follow the format `[<0|3>[-la] | woACT] [clang|gcc]`. `0/3` indicates whether constant propagation was disabled/enabled. `la` indicates whether lookahead was enabled. `clang/gcc` indicates which C compiler was used. The last part `woACT` indicates that custom register updates are disabled, in which case we generate a single fused SST as described in 7.6.3. These are only run with constant propagation and lookahead enabled.

Experimental setup. The benchmark machine runs Linux, has 32 GB RAM and an eight-core Intel Xeon E3-1276 3.6 GHz CPU with 256 KB L2 cache and 8 MB L3 cache. Each benchmark program was run 15 times, after first doing two warm-up rounds. All C and C++ files have been compiled with `-O3` on both GCC and Clang.

Difference between Kleenex and the other implementations. Unless otherwise stated, all the non-Kleenex implemetations follow the same structure: a loop that reads the input line by line and applies a regular expression to the line. Hence, in these implementations there is an interplay between the regular expression library used and the external language. For instance, in the RE2 programs, C++ control structures are used to achieve the desired behavior. In Kleenex, line breaks do not carry any special significance, so the multi-line programs can be formulated entirely within Kleenex.

Ragel optimization levels. Ragel is a state-machine compiler that generates C code. It implements several different compilation techniques; we have compiled all Ragel programs with the most optimized versions of each compilation technique. These are called "T1," "F1," and "G2." "T1" and "F1" causes the generated C code to be based on a lookup-table, and "G2" means that it should be based on C `goto` statements.

Kleenex compilation timeout. Some of the plots do not contain all versions of the Kleenex implementations. This is because the C compiler has timed out (after 30 seconds). Because our compiler performs a determinization of a non-deterministic machine, the set of states

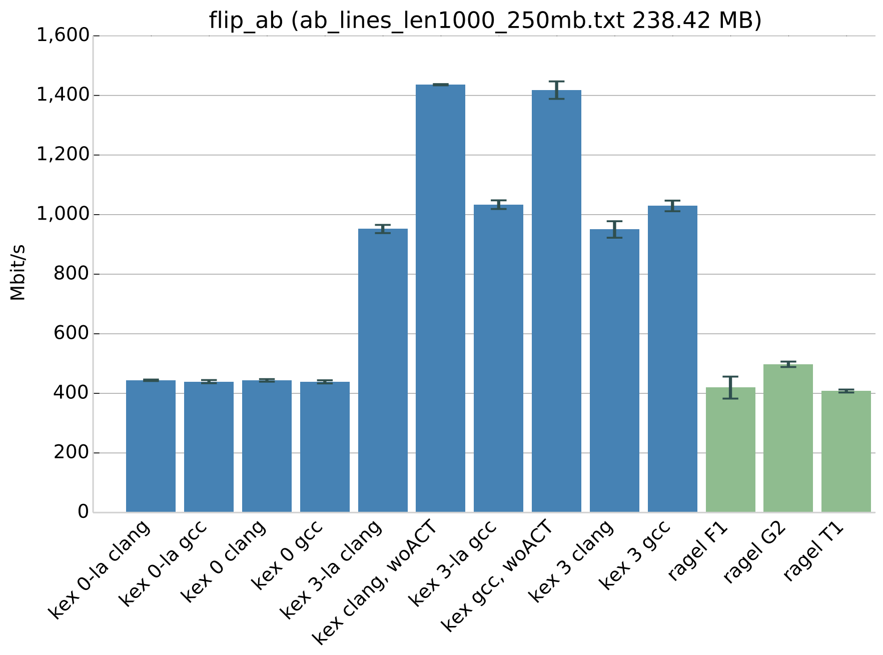


Figure 7.5: The program `flip_ab` run on lines with average length 1000.

can be very large in programs with a high degree of overlap. Here, the C code that is generated becomes very large, which causes the C compiler to spend more time than we have allowed. This is a an area for future research.

7.6.1 Baseline

The following three programs are intended to give a baseline impression of the performance of Kleenex programs.

Flipping "a"s and "b"s

The program `flip_ab` swaps "a"s and "b"s on all its input lines. Thus, any implementation of this program is forced to inspect every single input character. In Kleenex it looks like this:

```
main := ("b" ~/a/ | "a" ~/b/ | /\n/)*
```

We made a corresponding implementation with Ragel, using a `while`-loop in C to get each new input line and feed it to the automaton code generated by Ragel.

Implementing this functionality with regular expression libraries in the other tools would be an unnatural use of them, so we have not measured those.

The performance of the two implementations run on input with an average line length of 1000 characters is shown in Figure 7.5.

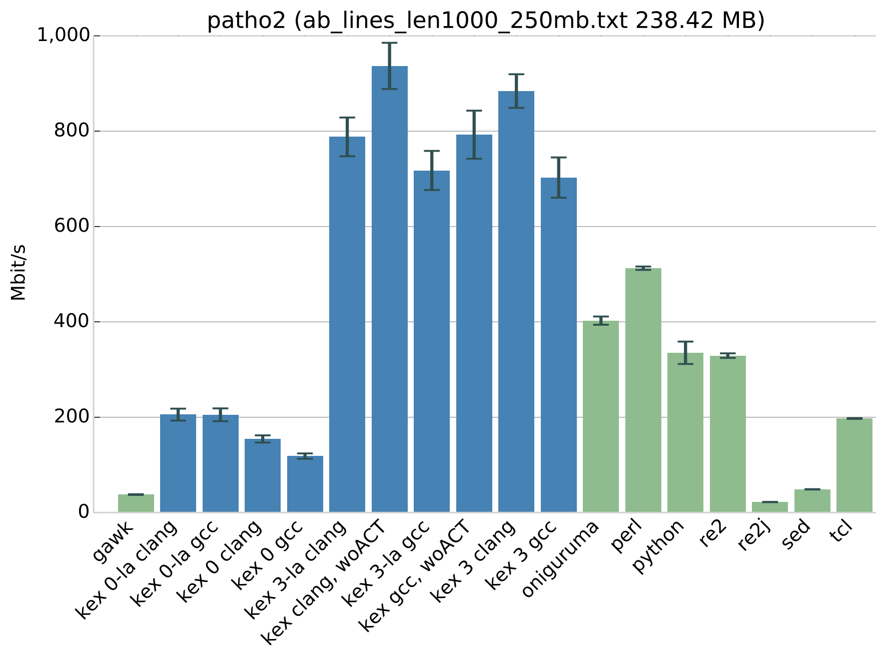


Figure 7.6: The non-streaming program `patho2` run on lines with average length 1000.

A Pathological Program

The program `patho2` is inherently non-streaming: for each line of input, output can *only* be made after the last character on that line has been read.

```
main := ((~/[a-z]*a/ | /[a-z]*b/)? /\n/)+
```

In this benchmark, the constant propagation makes a big difference, as Figure 7.6 shows. Due to the high degree of interleaving and the lack of keywords, in this program the lookahead optimization reduces overall performance.

This benchmark was not run with Ragel because Ragel requires the programmer to do all disambiguation manually when writing the program; the C code that Ragel generates does not handle ambiguity in a predictable way. It is very difficult to implement highly ambiguous text transformations with Ragel because the programmer must play the part of the disambiguator! This exemplifies one benefit of Kleenex, namely that it is no more difficult for a programmer point of view to express ambiguous transformations, for example to express fallback cases.

7.6.2 Rewriting

In this section we study some programs that perform slightly more interesting transformations than above.

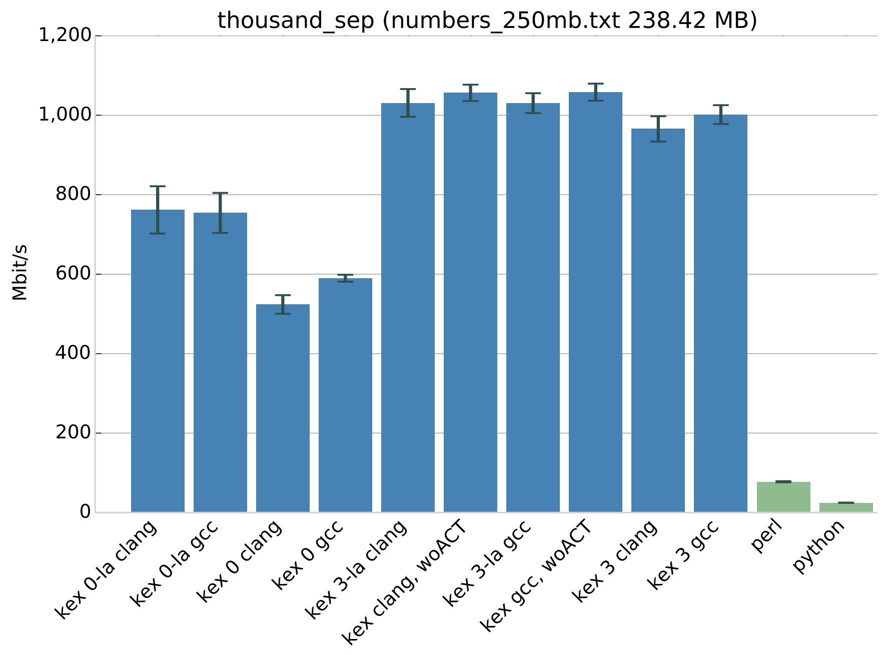


Figure 7.7: Inserting thousand separators on random numbers with average length 1000.

Thousand Separators

The following Kleenex program inserts thousand separators in a sequence of digits:

```
main  := (num /\n/)*
num   := digit{1,3} ("," digit{3})*
digit := /[0-9]/
```

We evaluated the Kleenex implementation along with two other implementations using Perl and Python. The performance can be seen in Figure 7.7. Both Perl and Python are significantly slower than all of the Kleenex implementations; this is a problem that is tricky to formulate with normal regular expressions. Interestingly, if one reads the input right-to-left, it becomes trivial to formulate, as a simple greedy strategy would suffice then.

CSV Rewriting

The program `csv_project3` deletes all columns except number two and five from a CSV file. It outputs those two columns with a separating tabulation character:

```
main := (row /\n/)*
col  := /[^,\n]*/
row  := ~(col /,/) col "\t" ~/,/ ~(col /,/)
        ~(col /,/) col ~/,/      ~col
```

Various specialized tools exist that can handle this transformation are included in Figure 7.8; GNU `cut` is a command that splits its input on certain characters, and GNU `AWK` has built-in support for this type of transformation.

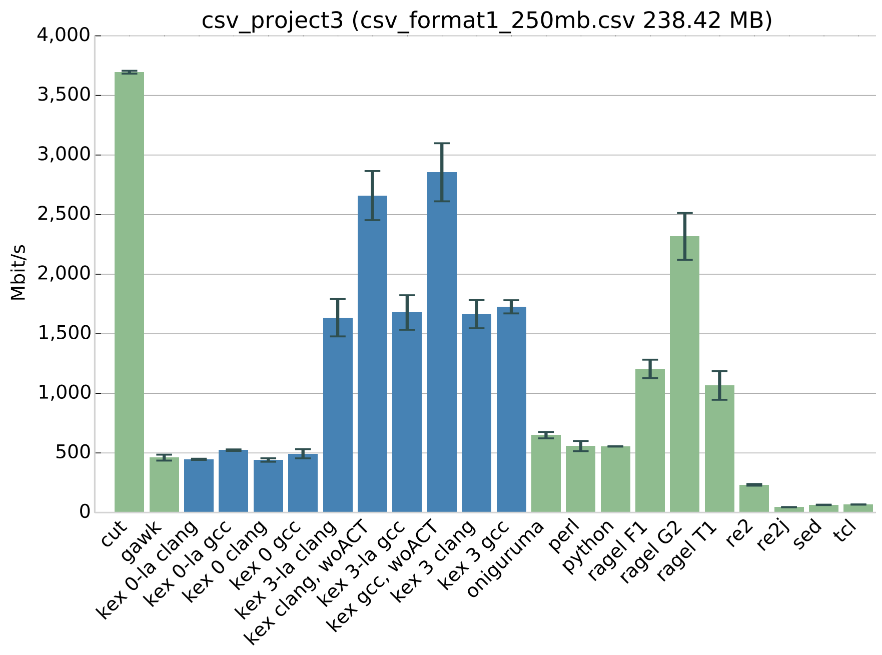


Figure 7.8: The program `csv_project3` reads in a CSV file with six columns and outputs columns two and five with a tabulation character as separator. The label "gawk" refers to GNU AWK. The tool "cut" is a command from GNU coreutils that is designed to split up lines.

Apart from `cut`, which is really fast for its own use-case, the Kleenex implementation is the fastest. The performance of Ragel is slightly lower, but this is likely due to the way the implementation produces output: In a Kleenex program, output strings are automatically put in an output buffer which is flushed routinely, whereas a programmer has to manually handle buffering when writing a Ragel program.

IRC Protocol Handling

The following Kleenex program parses the IRC protocol as specified in RFC 2812.[3] It follows roughly the output style described in part 2.3.1 of the RFC. Note that the Kleenex source code and the BNF grammar in the RFC are almost identical. Figure 7.9 shows the throughput on 250 MiB data.

```
main := (message | "Malformed line: " /[^\r\n]*\r?\n/)*
message := (~/:/ "Prefix: " prefix "\n"  ~/ /)?
           "Command: " command "\n"
           "Parameters: " params? "\n"
           ~crlf
command := letter+ | digit{3}
prefix := servername
```

[3] https://tools.ietf.org/html/rfc2812

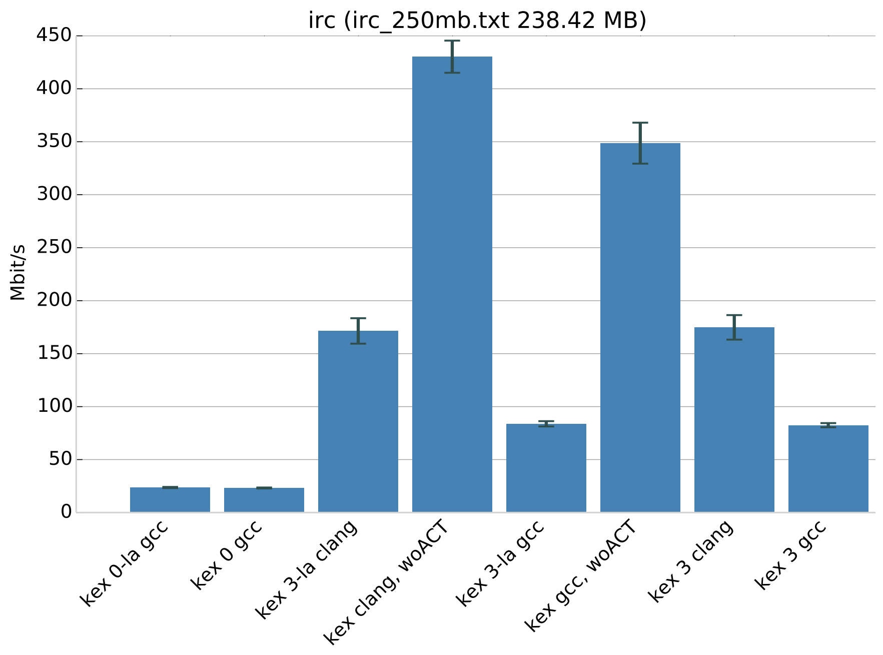


Figure 7.9: Throughput when parsing 250 MiB random IRC data.

```
        | nickname ((/!/ user)? /@/ host )?
user := /[^\n\r @]/+ // Missing \x00
middle := nospcrlfcl ( /:/ | nospcrlfcl )*
params := (~/ / middle ", "){,14} ( ~/ :/ trailing )?
        | ( ~/ / middle ){14} ( / / /:/?  trailing )?
trailing := (/:/ | / / | nospcrlfcl)*
nickname := (letter | special)
            (letter | special | digit){,10}
host := hostname | hostaddr
servername := hostname
hostname := shortname ( /\./ shortname)*
hostaddr := ip4addr
shortname := (letter | digit) (letter | digit | /-/)*
             (letter | digit)*
ip4addr := (digit{1,3} /\./ ){3} digit{1,3}
```

7.6.3 The Effects of Action-Separation

One can choose to use the machine resulting in combining the oracle and the action machine when compiling Kleenex. Doing so results in only one process doing both the disambiguation and outputting, which in some cases is faster and in other slower. Figures 7.8, 7.11, 7.15, and 7.17 illustrate both situations. It depends on the structure of the problem whether it pays off to split up the work in two processes; if all the work happens in the oracle and the action machine nearly does not do anything, then the added overhead incurred by the process context switches becomes noticeable. On the other hand, in cases where both machines do much work, the fact that two CPU cores can be utilized speeds up the program. This would be more likely if Kleenex had support for actions which could perform arbitrary computation, e.g. in the form of embedded C code.

Furthermore, the fact that two processes communicate over a pipe adds the factor that the size of the bit-code transferred is not too large. In their current form, the oracle machines only have a binary sum operator, no $\sum$ operator that can range over n cases. This means that in a choice such as

```
m := a | b | c | d | e | f | g
```

there will be six direction bits per input character that should be parsed by the g case. For implementation reasons, each direction bit is really a whole byte, so the above causes a six-fold increase in the size of the transmitted data between oracle and action machine. This is an effect visible in the `rot13` program (Figure 7.17).

7.7 Use Cases and Example Programs

In this section we will briefly touch upon various interesting use cases for Kleenex and show some example programs.

7.7.1 JSON logs to SQL

We have implemented a Kleenex program that transforms a JSON log file into an SQL insert statement. The program works on the logs provided by Issuu.[4] The code is shown in Figure 7.10.

The Ragel version we implemented outperforms Kleenex by about 50% (Figure 7.11), indicating that further optimizations of our SST construction should be possible.

7.7.2 Apache CLF to JSON

The Kleenex program in Figure 7.12 rewrites Apache CLF[5] log files into a list of JSON records.

This is a re-implementation of a Ragel program.[6] Figure 7.13 contains the benchmark results. The versions compiled with clang are not included, as the compilation timed out after 30 seconds. Curiously, the non-optimized Kleenex program is the fastest in this case.

7.7.3 ISO Date/Time Objects to JSON

Inspired by a regular expression "cookbook" [61], the program `iso_datetime_to_json` converts date and time stamps in an ISO standard format to JSON objects. Its source code is shown in Figure 7.14 and Figure 7.15 shows the performance compared to other implementations of the transformations.

[4]The line-based data set consists of 30 compressed parts and part one is available from `http://labs.issuu.com/anodataset/2014-03-1.json.xz`. The script on `https://github.com/diku-kmc/repg/blob/master/test/data/issuu/download.sh` can be used to fetch parts of it.

[5]`https://httpd.apache.org/docs/trunk/logs.html#common`

[6]`https://engineering.emcien.com/2013/04/5-building-tokenizers-with-ragel`

```
main :=
  "INSERT INTO issuu_log (ts, visitor_uuid, "
  "visitor_useragent, visitor_country) VALUES\n"
  json2sql

json2sql := object ",\n" ws json2sql
          | object ";\n" ws

object := "(" ~/\{/ ws keyVals ws ~/}/ ")"
keyVals := (ws keyVal)+

keyVal :=
    ~/"ts"/                 sep someInt             keepComma
  | ~/"visitor_uuid"/       sep stringReplaceQuotes keepComma
  | ~/"visitor_useragent"/ sep stringReplaceQuotes keepComma
  | ~/"visitor_country"/   sep stringReplaceQuotes dropComma
  | fb

fb := ~(/"/ someString /"/   sep ( /"/ someString /"/
                                 | someInt
                                 ) (dropComma | ""))


stringReplaceQuotes := qt someString qt
qt := "'" ~/"/ // replace double with single quote
sep := ws ~/:/ ws
someString := /[^"\n]*/
someInt := /-?[0-9]*/
someNumber := someInt /\./ someInt
contryCode := /[A-Z]{2}/

// Skip whitespace
ws := ~/[ \n]*/
keepComma := ws /,/
dropComma := ws ~/,/
```

Figure 7.10: The Kleenex code for the Issuu JSON log file to SQL transformation.

7.7.4 The ROT13 Text Transformation

The `rot13` program shifts letters in the English alphabet by 13 places. In Kleenex it can be implemented as shown in Figure 7.16.

Figure 7.17 shows a performance comparison of Kleenex and Ragel implementations of ROT13. The fact that the deforestation optimization (`woACT`) makes such a big difference on this plot is caused, as discussed in Section 7.6.3, by the fact that the input data is random, so for all input characters except "a" the data transferred between oracle and action machine will be multiple times larger than the input size.

7.7.5 BibTeX Rewriting

DReX is another tool for specifying string rewriting programs based on streaming string transducers [3]. We have implemented some of the example programs used by the authors

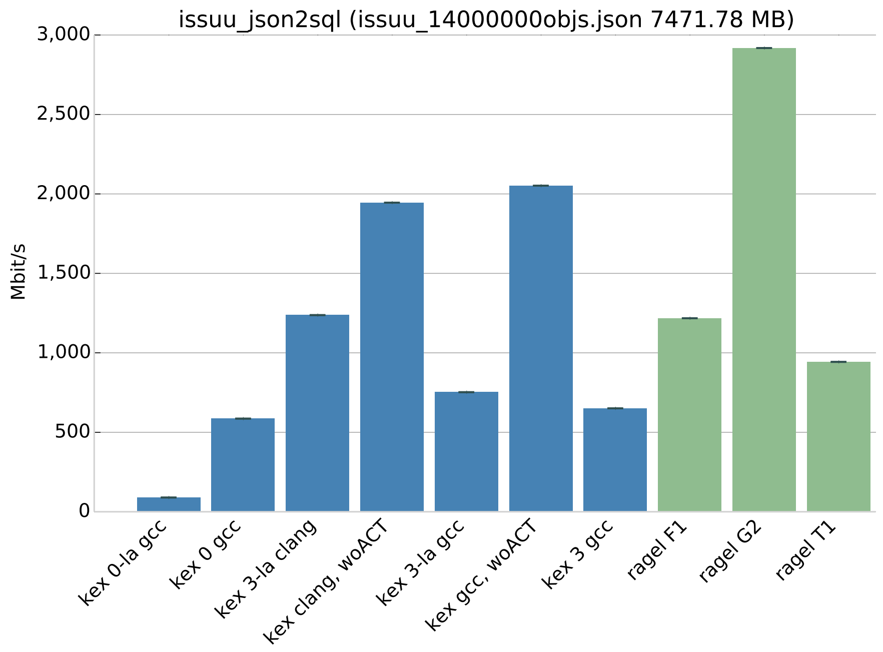


Figure 7.11: The speeds of transforming JSON objects to SQL INSERT statements using Ragel and Kleenex.

```
main := "[" loglines? "]\n"
loglines := (logline "," /\n/)* logline /\n/
logline := "{" host ~sep ~userid ~sep ~authuser sep
               timestamp sep request sep code sep
               bytes sep referer sep useragent "}"
host := "\"host\":\"" ip "\""
userid := "\"user\":\"" rfc1413 "\""
authuser := "\"authuser\":\"" /[^ \n]+/ "\""
timestamp := "\"date\":\"" ~/\[/ /[^\n\]]+/ ~/]/ "\""
request := "\"request\":" quotedString
code := "\"status\":\"" integer "\""
bytes := "\"size\":\"" (integer | /-/) "\""
referer := "\"url\":" quotedString
useragent := "\"agent\":" quotedString
ws := /[\t ]+/
sep := "," ~ws
quotedString := /"([^"\n]|\\")*"/
integer := /[0-9]+/
ip := integer (/\./ integer){3}
rfc1413 := /-/
```

Figure 7.12: A Kleenex program that transforms Apache log files into a JSON format.

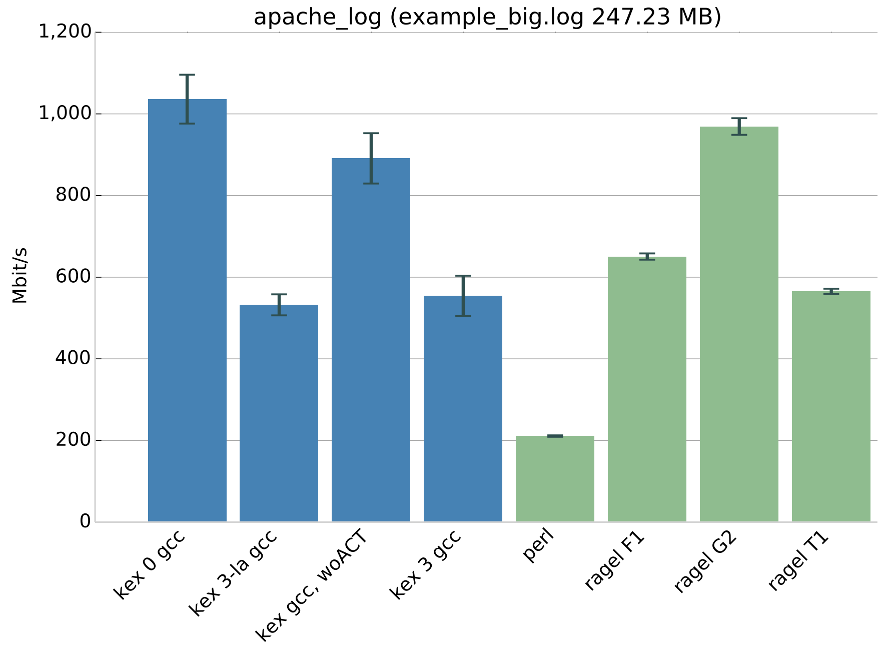


Figure 7.13: Speed of the conversion from the Apache Common Log Format to JSON.

```
// Kleenex program to transform datetime corresponding to
// the xml schema "datetime" object.  Outputs a JSON-like format.
// More or less completely taken from
// "Regular Expressions Cookbook", p. 237.
start: dateTimes

dateTimes := (dateTime ~/\n/)+

dateTime := "{'year'='" year ~/-/ "'"
            "', 'month'='" month ~/-/ "'"
            "', 'day'='" day ~/T/ "'"
            "', 'hours'='" hours ~/:/ "'"
            "', 'minutes'='" minutes ~/:/ "'"
            "', 'seconds'='" seconds "'"
            "', 'tz'='" timezone "'"
            "}\n"

year := /(?:[1-9][0-9]*)?[0-9]{4}/
month := /1[0-2]|0[1-9]/
day := /3[0-1]|0[1-9]|[1-2][0-9]/
hours := /2[0-3]|[0-1][0-9]/
minutes := /[0-5][0-9]/
seconds := /[0-5][0-9]/
timezone := /Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9]/
```

Figure 7.14: The Kleenex source for `iso_datetime_to_json`.

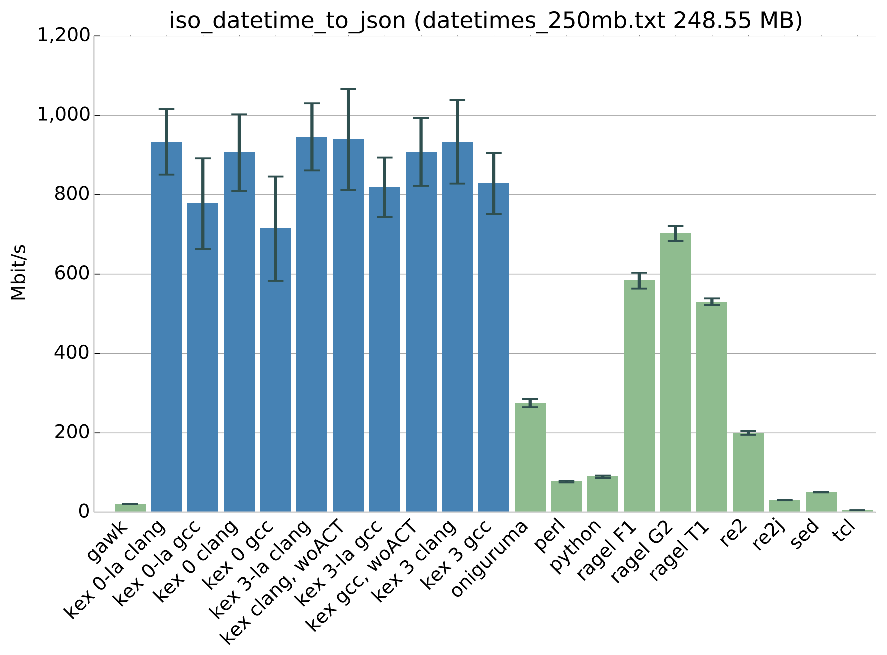


Figure 7.15: The performance of the conversion of ISO time stamps into JSON format.

```
main := (rot13 | /./)*
rot13  := ~/a/ "n" |  ~/b/ "o" |  ~/c/ "p" |  ~/d/ "q"
        | ~/e/ "r" |  ~/f/ "s" |  ~/g/ "t" |  ~/h/ "u"
        | ~/i/ "v" |  ~/j/ "w" |  ~/k/ "x" |  ~/l/ "y"
        | ~/m/ "z" |  ~/n/ "a" |  ~/o/ "b" |  ~/p/ "c"
        | ~/q/ "d" |  ~/r/ "e" |  ~/s/ "f" |  ~/t/ "g"
        | ~/u/ "h" |  ~/v/ "i" |  ~/w/ "j" |  ~/x/ "k"
        | ~/y/ "l" |  ~/z/ "m"
```

Figure 7.16: An implementation of the ROT13 text tranformation in Kleenex.

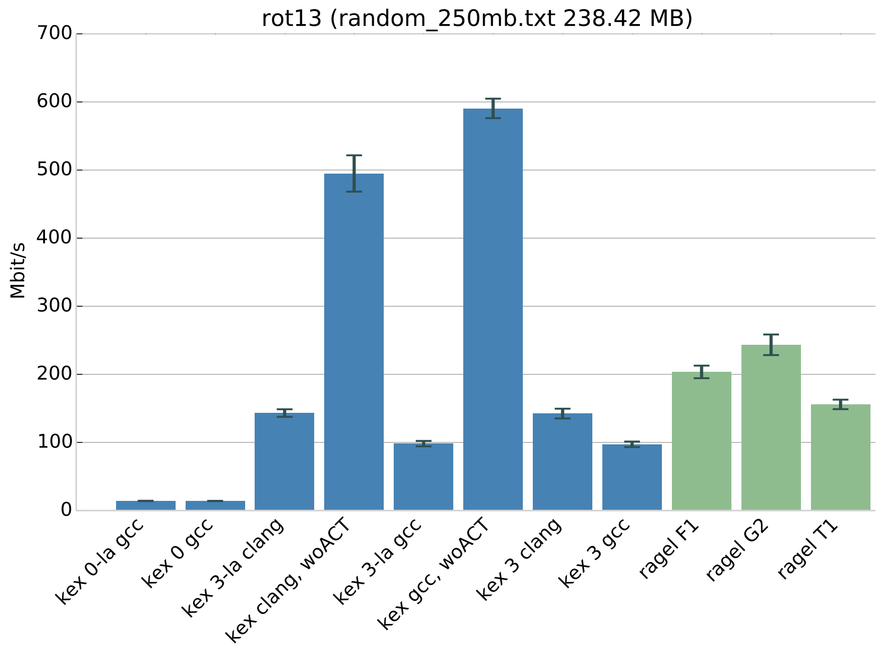


Figure 7.17: The speed of the `rot13` program. The choice of C compiler and the constant propagation make a big difference.

```
main := (~/\/\// ~line | line)*
line := /[^\n]*\n/
```

Figure 7.18: A Kleenex program that deletes comments from BibTeX files.

```
//Concatenate all XML tags, ignore things in between.
main := (tag | ~/./)*
tag  := /<[^>]*>/
```

Figure 7.19: A Kleenex program that strips tags from an XML file.

of DReX here; Figures 7.18, 7.19, and 7.20 contains the corresponding Kleenex versions of their examples.

7.7.6 Highlighting Kleenex Code

The Kleenex code examples in this chapter have been highlighted with a Kleenex program that emits LaTeX-commands for coloring. Figure 7.21 contains a variant of this program that emits ANSI color codes.

```
// Moves all title entries up to the previous entry
// in a bibtex file. The last entry is deleted.

// Uncomment to also swap title to the top
//start: align >> swap
start: align
align := head@header field* foot@footer
         ( !head head@header put_rest  field*
           !title !foot foot@footer)*
swap := (header field* !title put_rest footer)*
field := title@(sp /title/ sp /=/ sp /\{[^}]*},?\n/)
     | f@(sp word sp /=/ sp /\{[^}]*},?\n/) [ rest += f ]
header  := /@/ word sp /\{/ sp alnum /,\n/
footer  := /}/ (sp|/\n/)*
put_rest  := !rest [ rest <- "" ]
word  := /[A-Za-z_]+/
alnum  := /[A-Za-z0-9_]+/
sp := /[ \t]/*
```

Figure 7.20: A Kleenex program that shifts the title field in a BibTeX file to the entry one position up.

```
main := ( escape | comment | term
        | symbol | ignored | ws* )*
term := black /~/ (constant | match | ident) end
    | (teal constant | yellow match | blue ident) end
ignored  := /[]()|{},:[]/
ident    := (letter | /[0-9_]/)+
symbol   := yellow /<-|\+=|:=|>>|\*|\?|\+/ end
constant := /"/ ( /\\./ | /[^\\"]/ )* /"/
comment  := black ( /\/\/[^\n]*\n/
                  | /\/\*[^*\/]*\*\// ) end
match    := /\// ( /[^\/\n]/ | /\\./ )+ /\//
escape := /\\\\/
        | blue /\\x[0-9a-fA-F]{2}/ end
        | /\\[tnr]/
sp := / /*
letter := /[a-zA-Z]/
word   := letter+
ws := /[\t\r\n ]/
red    := "\x1b[31m"
green  := "\x1b[32m"
yellow:= "\x1b[33m"
blue   := "\x1b[34m"
end    := "\x1b[39;49m"
black  := "\x1b[30m"
teal   := "\x1b[36m"
```

Figure 7.21: A Kleenex program that highlights Kleenex syntax and emits ANSI color codes. A modified version of this was used to highlight the code in this chapter.

```
// Parses a list of RFC1738 generic URLs.
// https://www.ietf.org/rfc/rfc1738.txt
main := (genericurl /\n/)*
genericurl := scheme ~/:/ schemepart
scheme := "Scheme: " /[a-z0-9.+-]+/ "\n"
schemepart := ip_schemepart
            | "Scheme-part: " xchars "\n"
ip_schemepart := ~/\/\// login (~/\// urlpath)?
login := (user (~/:/ password)? ~/@/)? hostport
hostport := host (~/:/ port)?
host := "Host: " (hostname | hostnumber) "\n"
hostname := domainlabels toplabel
domainlabels := (domainlabel /\./)*
domainlabel := alphadigit (alphadashdigits alphadigit)?
toplabel := alpha (alphadashdigits alphadigit)?
alphadashdigits := (alphadigit | /-/)*
alphadigit := alpha | digit
hostnumber := digits (/\./ digits){3}
port := "Port: " digits "\n"
user := "User: " userstr "\n"
password := "Password: " userstr "\n"
userstr := (uchar | /[;?&=]/)*
urlpath := "Path: " xchars "\n"
xchars := xchar*

alpha    := /[a-zA-Z]/
digit    := /[0-9]/
digits   := /[0-9]+/
safe     := /[$_.+-]/
extra    := /[!*'(),]/
reserved := /[;\/?:@&=]/
escape   := /%[0-9A-Fa-f]{2}/
unreserved := alpha | digit | safe | extra
uchar := unreserved | escape
xchar := unreserved | reserved | escape
```

Figure 7.22: RFC1738 generic URL parser in Kleenex.

7.7.7 URL Parsing

Figure 7.22 contains a Kleenex implementation of a URL parser.

7.7.8 HTML Comments

The Kleenex program in Figure 7.23 finds HTML comments with basic formatting commands and renders them in HTML after the comment. For example,

```
<!-- doc: *Hello* world -->
```

becomes

```
<!-- doc: *Hello* world --><div> <b>Hello</b> world </div>.
```

```
main := (comment | /./)*
comment := /<!-- doc:/ clear doc* !orig /-->/
           "<div>" !render "</div>"
doc := ~/\*/ t@/[^*]*/ ~/\*/
       [ orig += "*" t "*" ] [ render += "<b>" t "</b>" ]
     | t@/./ [ orig += t ] [ render += t ]
clear := [ orig  <- "" ] [ render <- "" ]
```

Figure 7.23: A Kleenex program that finds and rewrites comments in HTML documents.

7.8 Related Work

We discuss related work in the context of current and future work.

7.8.1 Regular Expression Matching

Regular expression *matching* has different meanings in the literature.

For *acceptance testing*, which corresponds to classical automata theory, Bille and Thorup [19] improve on Myers' [111] log-factor improved RE-membership testing of classical NFA-simulation, based on tabling. They design a $O(kn)$ algorithm [20] with word-level parallelism, where $k \leq m$ is number of strings occurring in a regular expression. The tabling technique may be promising in practice; the algorithms have not been implemented and evaluated empirically, though.

In *subgroup matching* as in PCRE [73], an input is not only classified as accepting or not, but a substring is returned for each sub-RE in an RE designated to be of interest. Subgroup matching is often implemented by backtracking over alternatives, which yields the greedy match.[7] It may result in exponential-time behavior, however. Consequently, considerable human effort is expended to engineer REs to perform well. REs resulting in exponential run-time behavior are used in algorithmic attacks, leading to proposals for countermeasures to such attacks by classifying REs with slow backtracking performance [124, 133], where the countermeasures in turn appear to be attackable. Even in the absence of inherently hard matching with backreferences [1], backtracking implementations with avoidable performance blow-ups are amazingly wide-spread. This may be due to a combination of their good best-case performance and PCRE-embellishments driven by use cases. Some submatch libraries with guaranteed worst-case linear-time performance, notably RE2 [139], are making inroads, however. Both Myers, Oliva and Guimaraes [110] and Okui, Suzuki [117] describe POSIX-disambiguated matching algorithms, running in $O(mn)$ and $O(m^2n)$, respectively. Sulzmann and Lu [135] use Brzozowski and Antimirov derivatives [9, 27] for Perl-style subgroup matching for greedy and POSIX disambiguation.

Full RE *parsing* generalizes submatching: it returns a list of matches for each Kleene-star, also for nested ones. Kearns [83] and Frisch and Cardelli [57] devise three-pass linear-time *greedy* RE parsing; they require two passes over the input, the first consisting of reversing the entire input, before generating output in the third pass. Grathwohl, Henglein, Nielsen, Rasmussen devise a two-pass [64] and an optimally streaming [65] greedy regular expression parsing algorithm, described in Chapers 5 and 6. *Streaming* guarantees that line-by-line RE matching can be coded as a single RE matching problem. Sulzman and Lu [134] remark

[7] Committing to the left alternative before checking that the remainder of the input is accepted is the essence of parsing expression grammars [53].

that POSIX is notoriously difficult to implement correctly and show how to use Brzozowski derivatives [27] for POSIX RE parsing.

There are specialized RE matching tools and techniques too numerous to review comprehensively. We mention a few employing automaton optimization techniques applicable to Kleenex, but presently unexplored. Yang, Manadhata, Horne, Rao, Ganapathy [153] propose an OBDD representation for subgroup matching and apply it to intrusion detection REs; the cycle counts per byte appear a bit high, but are reported to be competitive with RE2. Sidhu and Prasanna [130] implement NFAs directly on an FPGA, essentially performing NFA-simulation in parallel; it outperforms GNU `grep`. Brodie, Taylor, Cytron [24] construct a multistride DFA, which processes multiple input symbols in parallel, and devise a compressed implementation on stock FPGA, also achieving very high throughput rates. Likewise, Ziria employs tabled multistriding to achieve high throughput [60]. Navarro and Raffinot [113] show how to code DFAs compactly for efficient simulation.

7.8.2 Ambiguity

Regular expressions may be ambiguous, which is irrelevant for acceptance testing, but problematic for submatching and parsing since the output depends on which amongst possibly multiple matches is to be returned. Brüggemann-Klein [26] provides an efficient $O(m^2)$ RE ambiguity testing algorithm. Vansummeren [143] illustrates differences between POSIX, first/longest and greedy matches. Colcombet [35] analyzes notions of (non)determinism of automata.

7.8.3 Transducers

From RE parsing it is a surprisingly short distance to the implementation of arbitrary non-deterministic finite state transducers (FSTs) [18, 109]. In contrast to the situation for *automata*, non-deterministic transducers are strictly more powerful than deterministic transducers; this, together with observable ambiguity, highlights why RE parsing is more challenging than RE acceptance testing.

As we have seen, efficient RE parsing algorithms operate on arbitrary NFAs, not only those corresponding to REs. Indeed, REs are not a particularly convenient or compact way of specifying regular languages: they can be represented by *certain* small NFAs with low tree-width, but may be inherently quadratically bigger even for DFAs [44, Theorem 23]. This is why Kleenex employs context-free grammars restricted to denote regular languages, with embedded output actions, to denote FSTs.

We have shown that FSTs, in particular unambiguous FSTs, can be implemented by a *subclass* of streaming string transducers (SSTs). SSTs extensionally correspond to regular transductions, functions implementable by 2-way deterministic finite-state transducers [4], MSO-definable string transductions [46] and a combinator language analogous to regular expressions [6]. The implementation techniques used in Kleenex appear to be directly applicable to all SSTs, not just the ones corresponding to FSTs.

Allender and Mertz [2] show that the functions computable by register automata, which generalize output strings to arbitrary monoids, are in NC and thus inherently parallelizable. This is achievable by performing relational FST-composition by matrix multiplication on the matrix representation of FSTs [18], which can be performed by parallel reduction. This is tantamount to running an FST from all states, not just the input state, on input string fragments. Mytkowicz, Musuvathi, Schulte [112] observe that there is often a small set of cut states sufficient to run each FST. This promises to be an interesting parallel harness for a suitably adapted Kleenex implementation running on fragments of very large inputs.

Veanes, Molnar, Mytkowics [147] employ symbolic transducers [21, 40, 146] and a data-parallel intermediate language in the implementation of BEK for multicore execution.

7.9 Conclusions

We have presented Kleenex, a convenient language for specifying (non-deterministic) finite state transducers; and its compilation to machine code representations of streaming state transducers, which emit the output.

Kleenex is comparatively expressive and performs consistently well—for complex regular expressions with nontrivial amounts of output almost always better in the evaluated use cases—vis-à-vis text processing tools such as RE2, Ragel, `grep`, `AWK`, `sed` and standard regular expression-libraries in Perl, Python, and Tcl.

We believe that Kleenex's clean semantics, streaming optimality, algorithmic generality, worst-case guarantees and absence of tricky code and special casing provide a useful basis for

- extensions to deterministic visible push-down automata, restricted versions of back-references and approximate/probabilistic matching;
- known, but so far unexplored optimizations, such as multicharacter input processing, automata minimization and symbolic representation, hybrid FST-simulation/SST-construction (analogous to NFA-simulation with NFA-state set memoization to implement on-demand DFA-construction);
- massively parallel (log-depth, linear work) input processing.

We thank Issuu for releasing their data set to the research community.

Part II

Extensions to Kleene Algebra

8 Infinitary Axiomatization of the Equational Theory of Context-Free Languages

This chapter is based on the paper "Infinitary Axiomatization of the Equational Theory of Context-Free Languages," published as FICS 2013 and under review for a Fundamenta Informaticae special issue [63].

8.1 Introduction

Algebraic reasoning about programming language constructs has been a popular research topic for many years. At the propositional level, the theory of flowchart programs and linear recursion are well handled by such systems as Kleene algebra and iteration theories, systems that characterize the equational theory of the regular sets. To handle more general forms of recursion including procedures with recursive calls, one must extend to the context-free languages, and here the situation is less well understood. One reason for this is that, unlike the equational theory of the regular sets, the equational theory of the context-free languages is not recursively enumerable. This has led some researchers to declare its complete axiomatization an insurmountable task [104].

Whereas linear recursion can be characterized with the star operator * of Kleene algebra or the dagger operation † of iteration theories [22], the theory of context-free languages requires a more general fixpoint operator μ. The characterization of the context-free languages as least solutions of algebraic inequalities involving μ goes back to a 1971 paper of Gruska [69]. More recently, several researchers have given equational axioms for semirings with μ and have developed fragments of the equational theory of context-free languages [38, 49, 50, 78, 79, 104].

In this chapter we consider another class of models satisfying a condition called *μ-continuity* analogous to the star-continuity condition of Kleene algebra:

$$a(\mu x.p)b = \sum_{n\geq 0} a(nx.p)b,$$

where the summation symbol denotes supremum with respect to the natural order in the semiring, and

$$0x.p = 0 \qquad\qquad (n{+}1)x.p = p[x/nx.p].$$

This infinitary axiom combines the assertions that $\mu x.p$ is the supremum of its finite approximants $nx.p$ and that multiplication in the semiring is continuous with respect to these suprema. Analogous to a similar result for star-continuous Kleene algebra, we show that all context-free languages over a μ-continuous idempotent semiring have suprema. Our main result is that the μ-continuity condition, along with the axioms of idempotent semirings, completely axiomatize the equational theory of the context-free languages. This is the first completeness result for the equational theory of the context-free languages, answering a question of Leiß [104].

8.1.1 Related Work

Courcelle [38] investigates *regular systems*, finite systems of fixpoint equations over first-order terms over a ranked alphabet with a designated symbol $+$ denoting set union, thereby restricting algebras to power set algebras. He stages their interpretation by first interpreting recursion over first-order terms as infinite trees, essentially as the final object in the corresponding coalgebra, then interpreting the signature symbols in ω-complete algebras. He provides soundness and completeness for transforming regular systems that preserve all solutions and soundness, but not completeness for preserving their least solutions. Courcelle's approach is syntactic since it employs unfolding of terms in fixpoint equations.

Leiß [104] investigates three classes of idempotent semirings with a syntactic least fixpoint operator μ. The three classes are called KAF, KAR, and KAG in increasing order of specificity. All these classes are assumed to satisfy the fundamental *Park axioms*

$$p[x/\mu x.p] \le \mu x.p \qquad\qquad p \le x \implies \mu x.p \le x,$$

which say that $\mu x.p$ is the least solution of the inequality $p \le x$. The classes KAR and KAG further assume

$$\mu x.(b + ax) = \mu x.(1 + xa)\cdot b \qquad \mu x.(b + xa) = b \cdot \mu x.(1 + ax)$$

and

$$\mu x.(s + rx) = \mu x.(\mu y.(1 + yr)\cdot s) \qquad \mu x.(s + xr) = \mu x.(s \cdot \mu y.(1 + ry)),$$

respectively. These axioms can be viewed as imposing continuity properties of the semiring operators with respect to μ. All standard interpretations, including the context-free languages over an alphabet X, are continuous and satisfy the KAG axioms. Ésik and Leiß [49, 50] show that conversion to Greibach normal form can be performed purely algebraically under these assumptions.

Ésik and Kuich [48] introduce *continuous semirings*, which are required to have suprema for all directed sets, and they employ domain theory to solve polynomial fixpoint equations.

8.1.2 Outline

In Section 8.2 we lay the foundations of our completeness result. In Sections 8.2.1–8.2.3 we introduce *Chomsky algebras*, our name for algebraically closed idempotent semirings, and develop a few of their basic properties. In Sections 8.2.4–8.2.5 we review the μ-notation, a well-known syntax for describing least solutions of systems of polynomial inequalities explicitly, and recall *Bekić's theorem*, which states that the μ operator is sufficient to describe the least solution of a finite system of simultaneous inequalities. In Section 8.2.6 we define the notion of *μ-continuity*, which is the cornerstone of our axiomatization. We also give an example of a Chomsky algebra that is not μ-continuous. In Section 8.2.7 we give several results establishing the relationship of our axiomatization to others in the literature.

Our main result, that our axiomatization exactly characterizes the equational theory of the context-free languages, is presented in Section 8.3. Finally, Section 8.4 contains discussion and conclusions.

8.2 Chomsky Algebras

In this section we introduce Chomsky algebras and the notion of *μ-continuity* and develop some of their properties. Intuitively, a Chomsky algebra is an idempotent semiring in which all systems of polynomial inequalities have unique least solutions.

8.2.1 Polynomials

Recall that an idempotent semiring (Definition 4.5) is a structure with binary operations $+$ and $\cdot$ and constants 0 and 1 satisfying the following equations:

$$\begin{aligned} a+(b+c) &= (a+b)+c & a(bc) &= (ab)c \\ a+b &= b+a & 1a &= a1 = a \\ a+0 &= a+a = a & a0 &= 0a = 0 \\ a(b+c) &= ab+ac & (a+b)c &= ac+bc. \end{aligned}$$

The adjective *idempotent* refers to the axiom $a+a=a$. Every idempotent semiring has a natural partial order $a \le b \iff a+b=b$.

Let $(C,\ +,\ \cdot,\ 0,\ 1)$ be an idempotent semiring and X a fixed set of variables. A *polynomial over indeterminates X with coefficients in C* is an element of $C[X]$, where $C[X]$ is the coproduct of C and the free idempotent semiring on generators X in the category of idempotent semirings. For example, if $a, b, c \in C$ and $x, y \in X$, then the following are polynomials:

$$0 \qquad a \qquad axbycx+1 \qquad ax^2byx+by^2xc \qquad 1+x+x^2+x^3.$$

The elements of $C[X]$ are not purely syntactic, as they satisfy all the equations of idempotent semirings and identities of C. For example, if $a^2 = b^2 = 1$ in C, then

$$(axa+byb)^2 = ax^2a+axabyb+bybaxa+by^2b.$$

Every polynomial can be written as a finite sum of *monomials* of the form

$$a_0x_0a_1x_1\cdots a_{n-1}x_{n-1}a_n,$$

where each $a_i \in C-\{0\}$ and $x_i \in X$. The *free variables* of such an expression p are the elements of X appearing in it and are denoted $\mathsf{FV}(p)$. The representation is unique up to associativity of multiplication and associativity, commutativity, and idempotence of addition.

8.2.2 Polynomial Functions and Evaluation

Let $C[X]$ be the semiring of polynomials over indeterminates X and let D be an idempotent semiring containing C as a subalgebra. By general considerations of universal algebra, any valuation $\sigma\colon X \to D$ extends uniquely to a semiring homomorphism $\hat{\sigma}\colon C[X] \to D$ preserving C pointwise. Formally, the functor $X \mapsto C[X]$ is left adjoint to a forgetful functor that takes an idempotent semiring D to its underlying set:

$$\begin{tikzcd} C[X] \arrow[r, "\hat{\sigma}"] & D \arrow[d, "|-|"] \\ X \arrow[u, "C[-]"] \arrow[r, "\sigma"'] & |D| \end{tikzcd}$$

Intuitively, $\hat{\sigma}$ is the *evaluation morphism* that evaluates a polynomial at the point $\sigma \in D^X$. Thus each polynomial $p \in C[X]$ determines a *polynomial function* $[\![p]\!] : D^X \to D$, where $[\![p]\!](\sigma) = \hat{\sigma}(p)$.

The set of all functions $D^X \to D$ with the pointwise semiring operations is itself an idempotent semiring with C as an embedded subalgebra under the embedding $c \mapsto \lambda\sigma.c$. The map $[\![\cdot]\!] : C[X] \to (D^X \to D)$ is actually $\hat{\tau}$, where $\tau(x) = \lambda f.f(x)$.

For the remainder of this chapter, we write σ for $\hat{\sigma}$, as there is no longer any need to distinguish them.

8.2.3 Algebraic Closure and Chomsky Algebras

A *system of polynomial inequalities over C* is a set

$$p_1 \le x_1,\ p_2 \le x_2,\ \ldots,\ p_n \le x_n \tag{8.1}$$

where $x_i \in X$ and $p_i \in C[X]$, $1 \le i \le n$. A *solution* of (8.1) in C is a valuation $\sigma : X \to C$ such that $\sigma(p_i) \le \sigma(x_i)$, $1 \le i \le n$. The solution σ is a *least solution* if $\sigma \le \tau$ pointwise for any other solution τ. If a least solution exists, then it is unique.

An idempotent semiring C is said to be *algebraically closed* if every finite system of polynomial inequalities over C has a least solution in C.

The category of *Chomsky algebras* consists of algebraically closed idempotent semirings along with semiring homomorphisms that preserve least solutions of systems of polynomial inequalities.

The canonical example of a Chomsky algebra is the family of context-free languages $\mathsf{CF}\,X$ over an alphabet X. A system of polynomial inequalities (8.1) can be regarded as a context-free grammar, and the least solution of the system is the context-free language generated by the grammar. For example, the set of strings in $\{a,b\}^\star$ with equally many a's and b's is generated by the grammar

$$S \to \varepsilon \mid aB \mid bA \qquad A \to aS \mid bAA \qquad B \to bS \mid aBB, \tag{8.2}$$

which corresponds to the system

$$1 + aB + bA \le S \qquad aS + bAA \le A \qquad bS + aBB \le B, \tag{8.3}$$

where the symbols a, b are interpreted as the singleton sets $\{a\}, \{b\}$, the symbols S, A, B are variables ranging over sets of strings, and the semiring operations $+, \cdot, 0$, and 1 are interpreted as set union, set product $AB = \{xy\}\ x \in A,\ y \in B$, $\emptyset$, and $\{\epsilon\}$, respectively.

Continuous idempotent semirings are also μ-continuous Chomsky algebras. These include Boolean semirings, the tropical semiring, the powerset of strings over an alphabet X, and the binary relations $\mathsf{Rel}\,X$ over X [48, p. 44]. Specifically, $\mathsf{Rel}\,X$ consists of all binary relations $R \subseteq X \times X$, where $+$ is set union, $\cdot$ relational composition, 0 the empty set, and 1 the identity relation $\{(x,x) \mid x \in X\}$.

Consider for example the relation $P = \bigcup_{n\ge 0} R^n S^n$, where $R^n = \underbrace{R \cdots R}_{n}$.

$$\begin{tikzcd} & \bullet \arrow[dr, dotted, "S^n"] & \\ \bullet \arrow[ur, dotted, "R^n"] \arrow[rr, bend right, "P"'] & & \bullet \end{tikzcd}$$

Then P is the least solution in $\mathsf{Rel}\, X$ to the system $RxS + 1 \leq x$.

8.2.4 μ-Expressions

Let X be a set of indeterminates. Leiß [104] and Ésik and Leiß [49, 50] consider *μ-expressions* defined by the grammar

$$t ::= x \mid t + t \mid t \cdot t \mid 0 \mid 1 \mid \mu x.t$$

where $x \in X$. These expressions provide a syntax with which least solutions of polynomial systems can be named. Scope, bound and free occurrences of variables, α-conversion, and safe substitution are defined as usual (see e.g. [14]). We denote by $t[x/u]$ the result of substituting u for all free occurrences of x in t, renaming bound variables as necessary to avoid capture. Let $\mathsf{T}X$ denote the set of μ-expressions over indeterminates X.

Let C be a Chomsky algebra and X a set of indeterminates. An *interpretation* over C is a map $\sigma\colon \mathsf{T}X \to C$ that is a homomorphism with respect to the semiring operations and such that

$$\sigma(\mu x.t) = \text{the least } a \in C \text{ such that } \sigma[x/a](t) \leq a, \tag{8.4}$$

where $\sigma[x/a]$ denotes σ with x rebound to a. The element a exists and is unique: Informally, by Bekić's theorem, each μ-expression t can be associated with a system of polynomial inequalities such that $\sigma(t)$ is a designated component of its least solution, which exists by algebraic closure.

Every set map $\sigma\colon X \to C$ extends uniquely to such a homomorphism. An interpretation σ *satisfies* the equation $s = t$ if $\sigma(s) = \sigma(t)$ and satisfies the inequality $s \leq t$ if $\sigma(s) \leq \sigma(t)$. All interpretations over Chomsky algebras satisfy the axioms of idempotent semirings, α-conversion (renaming of bound variables), and the *Park axioms* [47, 120]

$$t[x/\mu x.t] \leq \mu x.t \qquad\qquad t \leq x \implies \mu x.t \leq x. \tag{8.5}$$

Intuitively, the Park axioms say that $\mu x.t$ is the least solution of the single inequality $t \leq x$. It follows easily form (8.4) and (8.5) that

$$t[x/\mu x.t] = \mu x.t. \tag{8.6}$$

Thus, Chomsky algebras are essentially the ordered Park μ-semirings of [49] with the additional restriction that $+$ is idempotent and the order is the natural order $x \leq y \iff x + y = y$.

8.2.5 Bekić's Theorem

It is well known that the ability to name least solutions of single inequalities with μ gives the ability to name least solutions of all finite systems of inequalities. This is known as Bekić's theorem [17]. The construction is analogous to the definition of $M^\star$ for a matrix M over a Kleene algebra.

Bekić's theorem can be proved by regarding a system of inequalities as a single inequality on a Cartesian product, partitioning into two systems of smaller dimension, then applying the result for the 2×2 case inductively. See also [151] or [49] for a comprehensive treatment.

Proposition 8.1 (Bekić). *The* 2×2 *system*

$$p(x, y) \le x \qquad\qquad q(x, y) \le y$$

has least solution a_0, b_0*, where*

$$a(y) = \mu x.p(x, y) \qquad b_0 = \mu y.q(a(y), y) \qquad a_0 = a(b_0).$$

Proof. From (8.6) we have

$$a(y) = p(a(y), y) \qquad b_0 = q(a(b_0), b_0) = q(a_0, b_0)$$
$$a_0 = a(b_0) = p(a(b_0), b_0) = p(a_0, b_0).$$

Thus a_0, b_0 is a solution of the system. To show it is the least solution, suppose

$$p(c, d) \le c \qquad\qquad q(c, d) \le d.$$

By definition of a,

$$p(c, d) \le c \implies a(d) = \mu x.p(x, d) \le c.$$

By monotonicity, assumptions on c, d and definition of b_0,

$$q(a(d), d) \le q(c, d) \le d \implies b_0 \le d.$$

Again by monotonicity and assumptions on c, d and definition of a_0,

$$p(c, b_0) \le p(c, d) \le c \implies a_0 \le c.$$ □

For example, in the context-free languages, the set of strings in $\{a, b\}^\star$ with equally many a's and b's is represented by the term

$$\mu S.(1 + a \cdot \mu B.(bS + aBB) + b \cdot \mu A.(aS + bAA)) \tag{8.7}$$

obtained from the system (8.2) by this construction.

8.2.6 μ-Continuity

Let $nx.t$ be an abbreviation for the n-fold composition of t applied to 0, defined inductively by

$$0x.t = 0 \qquad\qquad (n{+}1)x.t = t[x/nx.t].$$

A Chomsky algebra is called *μ-continuous* if it satisfies the *μ-continuity axiom*:

$$a(\mu x.t)b = \sum_{n \ge 0} a(nx.t)b, \tag{8.8}$$

where the summation symbol denotes supremum with respect to the natural order $x \le y \iff x + y = y$. Note that the supremum of a and b is $a + b$.

The family $\mathsf{CF}\, X$ of context-free languages over an alphabet X forms a μ-continuous Chomsky algebra. The *canonical interpretation* over this algebra is $L_X : \mathsf{T}X \to \mathsf{CF}\, X$, where

$$\begin{aligned} L_X(x) &= \{x\} & L_X(t + u) &= L_X(t) \cup L_X(u) \\ L_X(0) &= \emptyset & L_X(tu) &= \{xy\}\ x \in L_X(t),\ y \in L_X(u) \\ L_X(1) &= \{\varepsilon\} & L_X(\mu x.t) &= \bigcup_{n \ge 0} L_X(nx.t). \end{aligned} \tag{8.9}$$

Under L_X, every term in $\mathsf{T}X$ represents a context-free language over its free variables (note that x is not free in $nx.t$). In the example (8.7) of Section 8.2.5, the free variables are a, b and the bound variables are S, A, B, corresponding to the terminal and nonterminal symbols, respectively, of the grammar (8.2) of Section 8.2.3.

Not all Chomsky algebras are μ-continuous. As with Kleene algebra [99] (Example 4.5), we can construct a Chomsky algebra that is not μ-continuous. Consider the set of ordered pairs of natural numbers ordered lexicographically, extended with adjoined least and greatest elements $\bot$ and $\top$. Define $+$ as supremum and

$$\begin{aligned} x \cdot \bot = \bot \cdot x &= \bot \\ x \cdot \top = \top \cdot x &= \top \quad (x \neq \bot) \\ (a,b)\cdot(c,d) &= (a+c, b+d). \end{aligned}$$

This is a Chomsky algebra, but it is not μ-continuous. We have

$$\mu x.(0,1)\cdot x + (0,1) = \top$$

since $(0,1)\cdot\top + (0,1) \leq \top$ and neither $\bot$ nor any (k,l) satisfy the inequality $(0,1)\cdot x + (0,1) \leq x$:

$$\begin{array}{lclclcl} (0,1)\cdot\bot + (0,1) & = & \bot + (0,1) & = & (0,1) & \not\leq & \bot \\ (0,1)\cdot(k,l) + (0,1) & = & (k,l+1) + (0,1) & = & (k,l+1) & \not\leq & (k,l). \end{array}$$

On the other hand,

$$\sum_{n\geq 0}(nx.(0,1)\cdot x + (0,1)) = \sup\{(0,n) \mid n \geq 1\} = (1,0).$$

We have $\mu x.t \neq \sum_{n\geq 0} nx.t$ for $t = (0,1)\cdot x + (0,1)$, which shows that this Chomsky algebra is not μ-continuous.

8.2.7 Relation to Other Axiomatizations

In this section we show that the axiomatizations considered in [49, 50, 104] are valid in all μ-continuous Chomsky algebras.

Definition 8.1 (μ-semiring [49]). A *μ-semiring* [49] is a semiring $(A, +, \cdot, 0, 1)$ satisfying the *μ-congruence* and *substitution* properties:

$$t = u \implies \mu x.t = \mu x.u \qquad\qquad \sigma(t[y/u]) = \sigma[y/\sigma(u)](t).$$

Idempotence is not assumed.

Lemma 8.1. *Every Chomsky algebra is a μ-semiring.*

Proof. The μ-congruence property is immediate from the definition of the μ operation (8.4). The substitution property is a general property of systems with variable bindings; see [14, Lemma 5.1.5]. It can be proved by induction. For the case of $\mu x.t$, we assume without loss of generality that $y \neq x$ (otherwise there is nothing to prove) and that x is not free in u.

$$\begin{aligned} \sigma((\mu x.t)[y/u]) &= \sigma(\mu x.(t[y/u])) \\ &= \text{least } a \text{ such that } \sigma[x/a](t[y/u]) \leq a \\ &= \text{least } a \text{ such that } \sigma[x/a][y/\sigma(u)](t) \leq a \qquad (8.10) \\ &= \text{least } a \text{ such that } \sigma[y/\sigma(u)][x/a](t) \leq a \\ &= \sigma[y/\sigma(u)](\mu x.t), \end{aligned}$$

where we have used the induction hypothesis in (8.10). □

We now consider various axioms proposed in [104].

Lemma 8.2. *In all μ-continuous Chomsky algebras,*

$$\mu x.(1 + ax) = \mu x.(1 + xa), \quad x \notin \mathsf{FV}(a).$$

Proof. By μ-continuity, it suffices to show that $nx.(1 + ax) = nx.(1 + xa)$ for all n. We show by induction that for all n, $nx.(1 + ax) = nx.(1 + xa) = \sum_{i=0}^{n} a^i$. The basis $n = 0$ is trivial:

$$0x.(1 + ax) = 0x.(1 + xa) = \sum_{i=0}^{0} a^i = 0.$$

For the inductive case,

$$\begin{aligned}(n{+}1)x.(1 + ax) &= 1 + a(nx.(1 + ax)) \\ &= 1 + a(\textstyle\sum_{i=0}^{n} a^i) \\ &= \textstyle\sum_{i=0}^{n+1} a^i \\ &= 1 + (\textstyle\sum_{i=0}^{n} a^i)a \\ &= 1 + (nx.(1 + xa))a \\ &= (n{+}1)x.(1 + xa).\end{aligned}$$ □

Lemma 8.3. *The following two equations hold in all μ-continuous Chomsky algebras:*

$$a(\mu x.(1 + xb)) = \mu x.(a + xb) \qquad (\mu x.(1 + bx))a = \mu x.(a + bx).$$

Proof. We show the first equation only; the second follows from a symmetric argument. By μ-continuity, we need only show that the equation holds for any n. We do this by induction over n. The basis $n = 0$ is trivial:

$$a(0x.(1 + xb)) = a0 = 0 = 0x.(a + xb).$$

For the inductive case,

$$\begin{aligned}a((n{+}1)x.(1 + xb)) &= a + a(nx.(1 + xb))b \\ &= a + (nx.(a + xb))b \\ &= (n{+}1)x.(a + xb),\end{aligned}$$

where the induction hypothesis has been used in the second step. □

These properties also show that μ-continuous Chomsky algebras are algebraically complete semirings, and therefore also algebraic Conway semirings in the sense of [49, 50].

Lemma 8.4. *The* Greibach inequalities

$$\mu x.s(\mu y.(1 + ry)) \leq \mu x.(s + xr) \qquad \mu x.(\mu y.(1 + yr))s \leq \mu x.(s + rx)$$

of KAG *[104] hold in all μ-continuous Chomsky algebras.*

Proof. For the left-hand inequality, let $u = \mu x.(s + xr)$. By the Park axioms, it suffices to show that $s(\mu y.(1 + ry))[x/u] \leq u$. But

$$\begin{aligned} s(\mu y.(1 + ry))[x/u] &= s[x/u](\mu y.(1 + r[x/u]y)) \\ &= s[x/u](\mu y.(1 + yr[x/u])) \\ &= \mu y.(s[x/u] + yr[x/u]) \\ &= \mu x.(s + xr), \end{aligned}$$

where Lemmas 8.2 and 8.3 have been used.

The right-hand inequality can be proved by a symmetric argument. □

Definition 8.2 (Algebraic Conway semiring [49]). An *algebraic Conway semiring* is a μ-semiring satisfying

$$\mu x.t[x/t'] = t[x/\mu x.t'[x/t]] \tag{8.11}$$

$$\mu x.\mu y.t = \mu x.t[y/x] \tag{8.12}$$

$$(\mu x.1 + ax)b = \mu x.b + ax \tag{8.13}$$

$$b(\mu x.1 + xa) = \mu x.b + xa \tag{8.14}$$

$$\mu x.1 + ax = \mu x.1 + xa \tag{8.15}$$

Lemma 8.5. *Any Chomsky algebra is an algebraic Conway semiring.*

Proof. Any Chomsky algebra is a μ-semiring by Lemma 8.1, and (8.13)–(8.15) hold by Lemmas 8.2 and 8.3. For (8.11), we have

$$\begin{aligned} \sigma(t[x/\mu x.t'[x/t]]) &= \text{least } a \text{ such that } \sigma(t[x/t'[x/t[x/a]]]) \leq a \\ &= \text{least } a \text{ such that } \sigma(t[x/t'][x/t[x/a]]) \leq a \\ &= \text{least } b \text{ such that } \sigma(t[x/t'][x/b]) \leq b \\ &= \sigma(\mu x.t[x/t']), \end{aligned}$$

and for (8.12) only need to show $nx.\mu y.t = nx.t[y/x]$ for all n by μ-continuity. We use induction on n. The base case $n = 0$ is trivial: $0x.\mu y.t = 0 = 0x.t[y/x]$, and for $n > 0$ we have

$$\begin{aligned} (n{+}1)x.\mu y.t &= \mu y.t[x/nx.\mu y.t] \\ &= \mu y.t[x/nx.t[y/x]] \\ &= (n{+}1)x.t[y/x]. \end{aligned}$$ □

Various other axioms of [49, 50, 104] follow from the Park axioms.

The μ-continuity condition (8.8) implies the Park axioms (8.5), but we must defer the proof of this fact until Section 8.3. For now we just observe a related property of the canonical interpretation L_X.

Lemma 8.6. *For any $s, t \in \mathsf{T}X$ and $y \in X$,*

$$L_X(s[y/\mu y.t]) = \bigcup_{n \geq 0} L_X(s[y/ny.t]).$$

Proof. We proceed by induction on the structure of s. The base cases are straightforward:

$$\begin{aligned}
L_X(0[y/\mu y.t]) &= L_X(0) = \bigcup_n L_X(0[y/ny.t]) \\
L_X(1[y/\mu y.t]) &= L_X(1) = \bigcup_n L_X(1[y/ny.t]) \\
L_X(x[y/\mu y.t]) &= L_X(x) = \bigcup_n L_X(x[y/ny.t]) \quad (x \neq y) \\
L_X(y[y/\mu y.t]) &= L_X(\mu y.t) = \bigcup_n L_X(y[y/ny.t]).
\end{aligned}$$

The remaining cases are shown as follows:

$$\begin{aligned}
L_X((p+q)[y/\mu y.t]) &= L_X(p[y/\mu y.t]) \cup L_X(q[y/\mu y.t]) \\
&= \bigcup_m L_X(p[y/my.t]) \cup \bigcup_n L_X(q[y/ny.t]) \\
&= \bigcup_n L_X(p[y/ny.t]) \cup L_X(q[y/ny.t]) \\
&= \bigcup_n L_X((p+q)[y/ny.t]). \\
L_X((pq)[y/\mu y.t]) &= L_X(p[y/\mu y.t]) \cdot L_X(q[y/\mu y.t]) \\
&= \bigcup_m L_X(p[y/my.t]) \cdot \bigcup_n L_X(q[y/ny.t]) \\
&= \bigcup_n L_X(p[y/ny.t]) \cdot L_X(q[y/ny.t]) \\
&= \bigcup_n L_X((pq)[y/ny.t]).
\end{aligned}$$

For $\mu x.s$, assume without loss of generality that $y \neq x$ and x is not free in t.

$$\begin{aligned}
L_X((\mu x.s)[y/\mu y.t]) &= \bigcup_m L_X((mx.s)[y/\mu y.t]) \\
&= \bigcup_m \bigcup_n L_X((mx.s)[y/ny.t]) \\
&= \bigcup_n \bigcup_m L_X((mx.s)[y/ny.t]) \\
&= \bigcup_n L_X((\mu x.s)[y/ny.t]).
\end{aligned}$$

□

8.3 Main Result

Our main result depends on an analog of a result of [96] (see [99]). It asserts that the supremum of a context-free language over a μ-continuous Chomsky algebra K exists, interpreting strings over K as products in K. Moreover, multiplication is continuous with respect to suprema of context-free languages.

Lemma 8.7. *Let $\sigma\colon \mathsf{T}X \to K$ be any interpretation over a μ-continuous Chomsky algebra K. Let $\tau\colon \mathsf{T}X \to \mathsf{CF}\,X$ be any interpretation over the context-free languages $\mathsf{CF}\,X$ such that for all $x \in X$ and $s, u \in \mathsf{T}X$,*

$$\sigma(sxu) = \sum_{y \in \tau(x)} \sigma(syu).$$

Then for any $s, t, u \in \mathsf{T}X$,

$$\sigma(stu) = \sum_{y \in \tau(t)} \sigma(syu).$$

In particular,

$$\sigma(stu) = \sum_{y \in L_X(t)} \sigma(syu), \tag{8.16}$$

where L_X is the canonical interpretation defined in §8.2.6.

Note carefully that the lemma does not assume *a priori* knowledge of the existence of the suprema. The equations should be interpreted as asserting that the supremum on the right-hand side exists and is equal to the expression on the left-hand side.

Proof. The proof is by induction on the structure of t, that is, by induction on the subexpression relation $t + u \succ t, t + u \succ u, t \cdot u \succ t, t \cdot u \succ u, \mu x.t \succ nx.t$, which is well-founded [97].

All cases are similar to the proof in [99, Lemma 7.1] for star-continuous Kleene algebra, with the exception of the case $t = \mu x.p$.

For variables $t = x \in X$, the desired property holds by assumption. For the constants $t = 0$ and $t = 1$,

$$\begin{aligned}
\sigma(s0u) &= 0 = \sum \emptyset = \sum_{y \in \emptyset} \sigma(syu) = \sum_{y \in \tau(0)} \sigma(syu)\\
\sigma(s1u) &= \sigma(su) = \sum_{y \in \{\varepsilon\}} \sigma(syu) = \sum_{y \in \tau(1)} \sigma(syu).
\end{aligned}$$

For sums $t = p + q$,

$$\begin{aligned}
\sigma(s(p+q)u) &= \sigma(spu) + \sigma(squ)\\
&= \sum_{x \in \tau(p)} \sigma(sxu) + \sum_{y \in \tau(q)} \sigma(syu) && (8.17)\\
&= \sum_{z \in \tau(p) \cup \tau(q)} \sigma(szu) && (8.18)\\
&= \sum_{z \in \tau(p+q)} \sigma(szu). && (8.19)
\end{aligned}$$

Equation (8.17) is by two applications of the induction hypothesis. Equation (8.18) is by the properties of supremum. Equation (8.19) is by the definition of sum in CF X.

For products $t = pq$,

$$\sigma(spqu) = \sum_{x \in \tau(p)} \sum_{y \in \tau(q)} \sigma(sxyu) \tag{8.20}$$

$$= \sum_{z \in \tau(p)\cdot\tau(q)} \sigma(szu) \tag{8.21}$$

$$= \sum_{z \in \tau(pq)} \sigma(szu). \tag{8.22}$$

Equation (8.20) is by two applications of the induction hypothesis. Equations (8.21) and (8.22) are by the definition of product in CF X.

Finally, for $t = \mu x.p$,

$$\sigma(s(\mu x.p)u) = \sum_n \sigma(s(nx.p)u) \tag{8.23}$$

$$= \sum_n \sum_{y \in \tau(nx.p)} \sigma(syu) \tag{8.24}$$

$$= \sum_{y \in \bigcup_n \tau(nx.p)} \sigma(syu) \tag{8.25}$$

$$= \sum_{y \in \tau(\mu x.p)} \sigma(syu). \tag{8.26}$$

Equation (8.23) is just the μ-continuity property (8.8). Equation (8.24) is by the induction hypothesis, observing that $\mu x.p \succ nx.p$. Equation (8.25) is a basic property of suprema. Finally, equation (8.26) is by the definition of $\tau(\mu x.p)$ in CF X.

The result (8.16) for the special case of $\tau = L_X$ is immediate, observing that L_X satisfies the assumption of the lemma: for $x \in X$,

$$\sigma(sxu) = \sum_{y\in\{x\}} \sigma(syu) = \sum_{y\in L_X(x)} \sigma(syu). \qquad \square$$

At this point we can show that the μ-continuity condition implies the Park axioms.

Theorem 8.1. *The μ-continuity condition* (8.8) *implies the Park axioms* (8.5).

Proof. We first show $p \le x \implies \mu x.p \le x$ in any idempotent semiring satisfying the μ-continuity condition. Let σ be a valuation such that $\sigma(\mu x.p) = \sum_n \sigma(nx.p)$. Suppose that $\sigma(p) \le \sigma(x)$. We show by induction that for all $n \ge 0$, $\sigma(nx.p) \le \sigma(x)$. This is certainly true for $0x.p = 0$. Now suppose it is true for $nx.p$. Using monotonicity,

$$\sigma((n{+}1)x.p) = \sigma(p[x/nx.p]) \le \sigma(p[x/x]) = \sigma(p) \le \sigma(x).$$

By μ-continuity, $\sigma(\mu x.p) = \sum_n \sigma(nx.p) \le \sigma(x)$.

Now we show that $p[x/\mu x.p] \le \mu x.p$. This requires the stronger property that a μ-expression is chain-continuous with respect to suprema of context-free languages as a function of its free variables. Using Lemmas 8.6 and 8.7,

$$\begin{aligned}
\sigma(p[x/\mu x.p]) &= \sum \{\sigma(y) \mid y \in L_X(p[x/\mu x.p])\} \\
&= \sum \left\{\sigma(y) \mid y \in \bigcup_n L_X(p[x/nx.p])\right\} \\
&= \sum_n \sum \{\sigma(y) \mid y \in L_X(p[x/nx.p])\} \\
&= \sum_n \sigma(p[x/nx.p]) \\
&= \sum_n \sigma((n{+}1)x.p) \\
&= \sigma(\mu x.p).
\end{aligned}$$

□

The following is our main theorem.

Theorem 8.2. *Let X be a set and let $s, t \in \mathsf{T}X$. The following are equivalent:*

(i) The equation $s = t$ holds in all μ-continuous Chomsky algebras; that is, $s = t$ is a logical consequence of the axioms of idempotent semirings and the μ-continuity condition

$$a(\mu x.t)b = \sum_{n\geq 0} a(nx.t)b, \tag{8.27}$$

or equivalently, the universal formulas

$$a(nx.t)b \leq a(\mu x.t)b, \quad n \geq 0 \tag{8.28}$$

$$\left(\bigwedge_{n\geq 0} (a(nx.t)b \leq w)\right) \implies a(\mu x.t)b \leq w. \tag{8.29}$$

(ii) The equation $s = t$ holds in the semiring of context-free languages $\mathsf{CF}\,Y$ over any set Y.

(iii) $L_X(s) = L_X(t)$, where $L_X : \mathsf{T}X \to \mathsf{CF}\,X$ is the standard interpretation mapping a μ-expression to a context-free language of strings over its free variables.

Thus the axioms of idempotent semirings and μ-continuity are sound and complete for the equational theory of the context-free languages.

Proof. The implication (i) $\implies$ (ii) holds since $\mathsf{CF}\,Y$ is a μ-continuous Chomsky algebra. The implication (ii) $\implies$ (iii) holds because (iii) is a special case of (ii). Finally, if (iii) holds, then by two applications of Lemma 8.7, for any interpretation $\sigma : \mathsf{T}X \to K$ over a μ-continuous Chomsky algebra K,

$$\sigma(s) = \sum_{x\in L_X(s)} \sigma(x) = \sum_{x\in L_X(t)} \sigma(x) = \sigma(t),$$

which proves (i). □

Corollary 8.1. *The context-free languages over the alphabet X form the free μ-continuous Chomsky algebra on generators X.*

Proof. Let K be a μ-continuous Chomsky algebra. Any map $\sigma : X \to K$ extends uniquely to an interpretation $\sigma : \mathsf{T}X \to K$. By Lemma 8.7, this decomposes as

$$\begin{tikzcd}
\mathsf{CF}\,X \arrow[r, "\mathsf{CF}\,\sigma"] & \mathsf{CF}\,K \arrow[d, "\Sigma"] \\
\mathsf{T}X \arrow[u, "L_X"] \arrow[r, "\sigma"'] & K
\end{tikzcd}$$

where

$$L_X : \mathsf{T}X \to \mathsf{CF}\,X$$

is the canonical interpretation in the context-free languages over X,

$$\mathsf{CF}\,\sigma : \mathsf{CF}\,X \to \mathsf{CF}\,K$$

is the map $\mathsf{CF}\,\sigma(A) = \{\sigma(x) \mid x \in A\}$, and

$$\Sigma : \mathsf{CF}\,K \to K$$

takes the supremum of a context-free language over K, which is guaranteed to exist by Lemma 8.7. The unique morphism $\mathsf{CF}\,X \to K$ corresponding to σ is $\Sigma \circ \mathsf{CF}\,\sigma$. Thus CF is left adjoint to the forgetful functor from μ-continuous Chomsky algebras to Set. The maps $x \mapsto \{x\} : X \to \mathsf{CF}\,X$ and $\Sigma : \mathsf{CF}\,K \to K$ are the unit and counit, respectively, of the adjunction. $\square$

8.4 Conclusion

We have given a natural complete infinitary axiomatization of the equational theory of the context-free languages:

$$a + (b + c) = (a + b) + c \tag{8.30}$$
$$a + b = b + a \tag{8.31}$$
$$a + 0 = a \tag{8.32}$$
$$a + a = a \tag{8.33}$$
$$a(bc) = (ab)c \tag{8.34}$$
$$1a = a \tag{8.35}$$
$$a1 = a \tag{8.36}$$
$$a(b + c) = ab + ac \tag{8.37}$$
$$(a + b)c = ac + bc \tag{8.38}$$
$$0a = 0 \tag{8.39}$$
$$a0 = 0 \tag{8.40}$$
$$a(nx.t)b \leq a(\mu x.t)b, \quad n \geq 0 \tag{8.41}$$
$$\left(\bigwedge_{n \geq 0} (a(nx.t)b \leq w) \right) \Rightarrow a(\mu x.t)b \leq w. \tag{8.42}$$

Leiß [104] states as an open problem:

> Are there natural equations between μ-regular expressions that are valid in all continuous models of KAF, but go beyond KAG?

Here we have identified such a system, thereby answering Leiß's question. He does not state axiomatization as an open problem, but observes that the set of pairs of equivalent context-free grammars is not recursively enumerable, then goes on to state:

> Since there is an effective translation between context-free grammars and μ–regular expressions ..., the equational theory of context-free languages in terms of μ-regular expressions is not axiomatizable at all.

Nevertheless, we have given an axiomatization. How do we reconcile these two views? Leiß is apparently using "axiomatization" in the sense of "recursive axiomatization." But observe that the axiom (8.42) is an infinitary Horn formula. To use it as a rule of inference, one needs to establish infinitely many premises of the form $x(ny.p)z \leq w$. But this in itself is a Π^0_1-complete problem. One can show that it is Π^0_1-complete to determine whether a given context-free grammar G over a two-letter alphabet generates all strings [89]. By coding G as a μ-expression w, the problem becomes $\mu x.(1 + ax + bx) \leq w$, which by (8.27) is equivalent to showing that $nx.(1 + ax + bx) \leq w$ for all n.

Acknowledgments

We thank Zoltán Ésik, Hans Leiß, and the anonymous referees for helpful comments.

9 KAT+B!

This chapter is based on the paper “KAT + B!” [67].

9.1 Introduction

Kleene algebra with tests (KAT) is a propositional equational system that combines Kleene algebra (KA) with Boolean algebra. It has been shown to be an effective tool for many low-level program analysis and verification tasks involving communication protocols, safety analysis, source-to-source program transformation, concurrency control, and compiler optimization [8, 15, 31–33, 93, 101]. A notable recent success is its adoption as a basis for NetKAT, a foundation for software-defined networks (SDN) [7, 95].

One advantage of KAT is that it allows a clean separation of the theory of the domain of computation from the program restructuring operations. The former typically involves first-order reasoning, whereas the latter is typically propositional. It is often advantageous to separate the two, because the theory of the domain of computation may be highly undecidable. With KAT, one typically isolates the needed properties of the domain as premises in a Horn formula

$$s_1 = t_1 \wedge \cdots \wedge s_n = t_n \to s = t,$$

where the conclusion $s = t$ expresses a more complicated equivalence between, say, an unoptimized or unannotated version of a program and its optimized or annotated version. The premises are verified once and for all using the properties of the domain, and the conclusion is then verified propositionally in KAT under those assumptions.

Certain premises that arise frequently in practice can be incorporated as part of the theory using a technique known as *elimination of hypotheses*, in which Horn formulas with premises of a certain form can be reduced to the equational theory without loss of efficiency [31, 71, 102]. However, there are a few useful ones that cannot. In particular, it is known that there are certain program transformations that cannot be effected in pure KAT, but require extra structure. Two paradigmatic examples are the Böhm–Jacopini theorem [23] (see also [12, 118, 121, 123, 150]) and the folklore result that all WHILE programs can be transformed to a program with a single WHILE loop [72, 108].

The Böhm–Jacopini theorem states that every deterministic flowchart can be written as a while program. The construction is normally done at the first-order level and introduces auxiliary variables to remember values across computations. It has been shown that the construction is not possible without some kind of auxiliary structure of this type [12, 87, 103].

Akin to the Böhm–Jacopini theorem, and often erroneously conflated with it, is the folklore theorem that every while program can be written with a single while loop. Like the proof of the Böhm–Jacopini theorem, the proofs of [76, 108], as reported in [72], are normally done at the first-order level and use auxiliary variables. It was a commonly held belief that this result had no purely propositional proof [72], but a partial refutation of this view was given in [93] using a construction that foreshadows the construction we present here.

One can carry out these constructions in an uninterpreted first-order version of KAT called *schematic KAT* (SKAT) [8, 98], but as SKAT is undecidable in general [92], one would prefer a less radical extension.

In this chapter we investigate the minimal amount of structure that suffices to perform these transformations and show how to incorporate it in KAT without sacrificing deductive completeness or decidability. Our main results are:

- We show how to extend KAT with a set of independent *mutable tests*. The construction is done axiomatically with generators and additional equational axioms. We formulate the construction as a general *commutative coproduct* construction that satisfies a certain universality property. The generators are abstract *setters* of the form $b!$ and $\bar{b}!$ and *testers* $b?$ and $\bar{b}?$ for a test symbol b. We can think of these intuitively as operations that set and test the value of a Boolean variable, although we do not introduce any explicit notion of storage or variable assignment.

- We prove a representation theorem (Theorem 9.2) for the commutative coproduct of an arbitrary KAT K and a KAT of binary relations on a finite set, namely that it is isomorphic to a certain matrix algebra over K.

- As a corollary to the representation theorem, we show that the extension is *conservative*; that is, an arbitrary KAT K can be augmented with mutable tests without affecting the theory of K. This is captured formally by a general property of the commutative coproduct, namely *injectivity*. It is not known whether the coproduct of KATs is injective in general, but we show that it is injective if at least one of the two cofactors is a finite relational KAT, which is the case in our application.

- We show that the free mutable test algebra on generators b_i, $1 \leq i \leq n$, is isomorphic to the KAT of all binary relations on a set of 2^n states. We also characterize the primitive operations in terms of a tensor product of n copies of a 2-state system. We show that the equational theory of this algebra is PSPACE-complete, thus no easier or harder to decide than KAT.

- We show that the equational theory of an arbitrary KAT K augmented with mutable tests is axiomatically reducible to the theory of K. In particular, the free KAT, augmented with mutable tests, is completely axiomatized by the KAT axioms plus the axioms for mutable tests.

- We show that the equational theory of KAT with mutable tests, KAT+B!, is EXPSPACE-complete.

- We demonstrate that the program transformations mentioned above, namely the Böhm–Jacopini theorem and the folklore result about WHILE programs, can be carried out in KAT with mutable tests.

Balbiani et al. [13] present a related system, DL-PA, a variant of propositional dynamic logic (PDL) with mutable tests only. Their system corresponds most closely to our free mutable test algebra, which is PSPACE-complete. The semantics of DL-PA is restricted to relational models, and they show that model checking and satisfiability are EXPTIME-complete. The added complexity is partly due to the presence of the modal operators in PDL, which are absent in KAT.

This chapter is organized as follows. In Section 9.2 we introduce the theory of mutable tests and prove that the free mutable test algebra on n generators is isomorphic to the KAT of all binary relations on a set of size 2^n. We also introduce the commutative coproduct construction and prove our representation theorem for the commutative coproduct of an arbitrary KAT K and a finite relational KAT. In Section 9.3 we prove our main completeness and complexity results. In Section 9.4 we apply the theory to give an axiomatic treatment of two applications involving program transformations. In Section 9.5 we present conclusions and open problems.

9.2 KAT and Mutable Tests

Recall that a *Kleene algebra* is a structure $(K, +, \cdot, ^\star, 0, 1)$ that is an idempotent semiring (Definition 4.5) with an iteration operator * satisfying

$$\begin{aligned} 1 + pp^\star &= p^\star & \qquad q + pr \leq r &\implies p^\star q \leq r \\ 1 + p^\star p &= p^\star & \qquad q + rp \leq r &\implies qp^\star \leq r \end{aligned}$$

where $\leq$ refers to the natural partial order on K given by (4.8):

$$x \leq y \iff x + y = y.$$

All KA operations are monotone with respect to $\leq$ (Proposition 4.2).

The following are some typical KA identities:

$$(p^\star q) \star p^\star = (p+q)^\star \tag{9.1}$$

$$p(qp)^\star = (pq)^\star p \tag{9.2}$$

$$p^\star = (p^n)^\star(1 + p + \cdots + p^{n-1}). \tag{9.3}$$

A *Kleene algebra with tests* (KAT) is a Kleene algebra with an embedded Boolean subalgebra (Definition 4.17). That is, it is a two-sorted structure $(K, B, +, \cdot, ^\star, ^-, 0, 1)$ such that

- $(K, +, \cdot, ^\star, 0, 1)$ is a Kleene algebra,
- $(B, +, \cdot, ^-, 0, 1)$ is a Boolean algebra, and
- B as a semiring is a subalgebra of K.

The Boolean complementation operator $^-$ is defined only on B. Elements of B are called *tests*. The letters p, q, r, s denote arbitrary elements of K and a, b, c denote tests. The operators $+, \cdot, 0, 1$ each play two roles: applied to arbitrary elements of K, they refer to nondeterministic choice, composition, fail, and skip, respectively; and applied to tests, they take on the additional meaning of Boolean disjunction, conjunction, falsity, and truth, respectively. These two usages do not conflict; for example, sequential testing of b and c is the same as testing their conjunction.

Conventional imperative programming constructs and Hoare partial correctness assertions can be encoded, and propositional Hoare logic is subsumed. The deductive completeness and complexity results for KA and KAT [34, 88, 102] say that the axioms are complete for the equational theory of standard language and relational models and that the equational theory is decidable in PSPACE.

9.2.1 Mutable Tests

Let $T_n = \{t_1, \ldots, t_n\}$ be a set of primitive test symbols. Consider a set of primitive actions $\{t!, \bar{t}! \mid t \in T_n\}$ (note that $\bar{\bar{t}}! = t!$). We write $t?$ for the test t to emphasize the distinction between $t?$ and $t!$. Let F_n be the free KAT over primitive actions $\{t!, \bar{t}! \mid t \in T_n\}$ and primitive tests T_n modulo the following equations:

(i) $t!t? = t!$

(ii) $t?t! = t?$

(iii) $t!\bar{t}! = \bar{t}!$

(iv) $s!t! = t!s!$, provided $s \neq \bar{t}$.

(v) $s!t? = t?s!$, provided $s \notin \{t, \bar{t}\}$.

Intuitively, axiom (i) says that the action $t!$ makes a subsequent test $t?$ true, (ii) says that if $t?$ is already true, then the action $t!$ is redundant, (iii) says that setting a value overrides a previous such action on the same value, and (iv) and (v) say that actions and tests on different values are independent.

The theory B! refers to the equational consequences of (i)–(v) along with the axioms of KAT on terms over T_n (Definition 4.23). Two immediate such consequences are (4.39) and (4.40):

(vi) $t!t! = t!$

(vii) $t!\bar{t}? = 0$.

Recall that an *atom* of T_n is a sequence $s_1 s_2 \cdots s_n$, where each s_i is either t_i or $\bar{t}_i$ (Definition). Atoms are denoted $\alpha, \beta, \ldots$, and the set of atoms is denoted At. We write $\alpha \leq t$ if t appears in α. Let $\alpha[t]$ denote the atom α if $\alpha \leq t$ and α with $\bar{t}$ replaced by t if $\alpha \leq \bar{t}$. Each atom $\alpha = s_1 \cdots s_n$ determines a *complete test* $\alpha? = s_1?s_2? \cdots s_n? \in F_n$ and a *complete assignment* $\alpha! = s_1!s_2! \cdots s_n! \in F_n$. The following are elementary consequences of B!:

$$t? = \sum_{\alpha \leq t} \alpha?\alpha! \qquad t! = \sum_{\alpha} \alpha?\alpha[t]! \tag{9.4}$$

$$\alpha!\alpha? = \alpha! \quad \alpha?\alpha! = \alpha? \quad \alpha!\beta! = \beta! \quad \alpha?\beta? = 0 \text{ if } \alpha \neq \beta. \tag{9.5}$$

9.2.2 Mutable Tests and Binary Relations

The following theorem characterizes the free B! algebra F_n. The theorem shows that B! is sound in the sense that the free model does not trivialize to the one-element algebra.

Theorem 9.1. *The algebra F_n is isomorphic to the KAT of all binary relations on a set of size 2^n.*

Proof. The set At is of size 2^n. Consider the KAT of binary relations on At. This algebra is isomorphic to $\mathrm{Mat}(\mathsf{At}, \mathbb{2})$, the KAT of $\mathsf{At} \times \mathsf{At}$ matrices over the two-element KAT with the usual Boolean matrix operations. We will construct an isomorphism $h_n : F_n \to \mathrm{Mat}(\mathsf{At}, \mathbb{2})$.

For the generators, let $h_n(t?)$ and $h_n(t!)$ be $\mathsf{At} \times \mathsf{At}$ matrices with components

$$h_n(t?)_{\alpha\beta} = \begin{cases} 1 & \text{if } \beta = \alpha \leq t \\ 0 & \text{otherwise,} \end{cases} \qquad h_n(t!)_{\alpha\beta} = \begin{cases} 1 & \text{if } \beta = \alpha[t] \\ 0 & \text{otherwise.} \end{cases}$$

One can show without difficulty that the axioms (i)–(v) of B! are satisfied under the interpretation h_n. We show (i) and (ii):

$$\begin{aligned}
(h_n(t!)h_n(t?))_{\alpha\beta} &= \sum_\gamma h_n(t!)_{\alpha\gamma} h_n(t?)_{\gamma\beta} \\
&= \begin{cases} 1 & \text{if } \gamma = \alpha[t] \text{ and } \beta = \gamma \leq t \\ 0 & \text{otherwise} \end{cases} \\
&= \begin{cases} 1 & \text{if } \beta = \alpha[t] \\ 0 & \text{otherwise} \end{cases} \\
&= h_n(t!)_{\alpha\beta}. \\
(h_n(t?)h_n(t!))_{\alpha\beta} &= \sum_\gamma h_n(t?)_{\alpha\gamma} h_n(t!)_{\gamma\beta} = h_n(t?)_{\alpha\alpha} h_n(t!)_{\alpha\beta} \\
&= \begin{cases} 1 & \text{if } \alpha \leq t \text{ and } \beta = \alpha[t] \\ 0 & \text{otherwise} \end{cases} \\
&= \begin{cases} 1 & \text{if } \alpha \leq t \text{ and } \beta = \alpha \\ 0 & \text{otherwise} \end{cases} \\
&= h_n(t?)_{\alpha\beta}.
\end{aligned}$$

Since F_n is the free B! algebra on generators T_n, h_n extends uniquely to a KAT homomorphism $h_n : F_n \to \mathrm{Mat}(\mathsf{At}, \mathbb{2})$. Under this extension, $h_n(\alpha?\beta!)$ is the matrix with 1 in location $\alpha\beta$ and 0 elsewhere. As every matrix in $\mathrm{Mat}(\mathsf{At}, \mathbb{2})$ is a sum of such matrices, h_n is surjective.

We wish also to show that h_n is injective. To do this, we show that every element of F_n is a sum of elements of the form $\alpha?\beta!$. This is true for primitive tests $t?$ and primitive actions $t!$ by (9.4). The constants 1 and 0 are equivalent to $\sum_\alpha \alpha?\alpha!$ and the empty sum, respectively.

For sums, the conclusion is trivial. For products, we observe using (9.5) that

$$\alpha?\beta!\gamma?\delta! = 0 \quad \text{if} \quad \beta \neq \gamma \quad \text{and} \quad \alpha?\beta!\beta?\delta! = \alpha?\delta!.$$

By distributivity, this allows the product of two sums of elements of the form $\alpha?\beta!$ to be reduced to a sum of the same form. For *, any element of the form e^* where e is a sum of elements of the form $\alpha?\beta!$ is equivalent to $1 + e + e^2 + \cdots + e^m$ for some m, since At is finite.

Now if $A \subseteq \mathsf{At}^2$, then $h_n(\sum_{\alpha\beta\in A} \alpha?\beta!)$ is the matrix with 1 in locations $\alpha\beta \in A$ and 0 elsewhere. Thus if $A, B \subseteq \mathsf{At}^2$ and $h_n(\sum_{\alpha\beta\in A} \alpha?\beta!) = h_n(\sum_{\alpha\beta\in B} \alpha?\beta!)$, then $A = B$, therefore $\sum_{\alpha\beta\in A} \alpha?\beta! = \sum_{\alpha\beta\in B} \alpha?\beta!$. □

9.2.3 The Commutative Coproduct

Let K and F be KATs. The *commutative coproduct* of K and F is the coproduct of K and F modulo extra commutativity conditions $\{ps = sp \mid p \in K,\ s \in F\}$ that say that elements of K and F commute multiplicatively. The commutativity conditions model the idea that operations in K and F are independent of each other. We will give an explicit construction below.

The usual coproduct $K \oplus F$ comes equipped with canonical coprojections $i_K : K \to K \oplus F$ and $i_F : F \to K \oplus F$. The coprojections are often called *injections*, although they need not be injective.[1] The coproduct is said to be *injective* if i_K and i_F are injective.

Injectivity is important because it means the extension of an algebra K with extra features F is *conservative* in the sense that it does not introduce any new equations. The coproduct of KATs is not known to be injective in general; however, we shall show that if F is a finite relational KAT, then the coproduct and commutative coproduct are injective.

Our proof relies on an explicit coproduct construction from universal algebra that holds for any variety or quasivariety V (class of algebras defined by universally quantified equations or equational implications) over any signature Σ. We briefly review the construction here.

Let T_K be the set of Σ-terms over K. The identity function $K \to K$ extends uniquely to a canonical homomorphism $T_K \to K$. The *diagram* of K, denoted Δ_K, is the kernel of this homomorphism; this is the set of equations between Σ-terms over K that hold in K. It follows from general considerations of universal algebra that $T_K/\Delta_K \cong K$, where T/E denotes the quotient of T modulo the V-congruence generated by equations E; that is, the smallest Σ-congruence on T containing E and closed under the equations and equational implications defining V.

Now let $T_{K,F}$ denote the set of mixed Σ-terms over the disjoint union of the carriers of K and F. The coproduct is

$$K \oplus F = T_{K,F}/(\Delta_K \cup \Delta_F).$$

The canonical injection $i_K : K \to K \oplus F$ is obtained from the identity embedding $T_K \to T_{K,F}$ reduced modulo Δ_K on the left and $\Delta_K \cup \Delta_F$ on the right; the map is well-defined on Δ_K-classes since Δ_K refines $\Delta_K \cup \Delta_F$. This construction satisfies the usual universality property for coproducts, namely that for any pair of homomorphisms $k : K \to H$ and $f : F \to H$, there is a unique homomorphism $\langle k, f\rangle : K \oplus F \to H$ such that $k = \langle k, f\rangle \circ i_K$ and $f = \langle k, f\rangle \circ i_F$.

Now let K and F be KATs, and let D be the set of commutativity conditions

$$D = \{i_K(p)i_F(s) = i_F(s)i_K(p) \mid p \in K,\ s \in F\}.$$

on $K \oplus F$. The *commutative coproduct* is the quotient $(K \oplus F)/D$. Composed with the canonical map $[\cdot] : K \oplus F \to (K \oplus F)/D$, i_K and i_F inject K and F, respectively, into $(K \oplus F)/D$. The following universality property is satisfied:

Lemma 9.1. *For any pair of homomorphisms $k : K \to H$ and $f : F \to H$ such that*

$$\forall p \in K\ \forall s \in F\ k(p)f(s) = f(s)k(p), \tag{9.6}$$

there is a unique universal arrow $\langle k, f\rangle_D : (K \oplus F)/D \to H$ such that $k = \langle k, f\rangle_D \circ [\cdot] \circ i_K$ and $f = \langle k, f\rangle_D \circ [\cdot] \circ i_F$.

Proof. Property (9.6) implies that D refines the kernel of $\langle k, f\rangle : K \oplus F \to H$, therefore $\langle k, f\rangle$ factors uniquely as $\langle k, f\rangle_D \circ [\cdot]$, as shown in Figure 9.1. ☐

[1]For example, $\mathbb{Z}_m \oplus \mathbb{Z}_n \cong \mathbb{Z}_{\gcd(m,n)}$ in the category of commutative rings.

$$\begin{tikzcd}
K \arrow[r, "i_K"] \arrow[rdd, bend right, "k"'] & K \oplus F \arrow[d, "{[\cdot]}"] \arrow[dd, bend right=60, "{\langle k, f\rangle}"'] & F \arrow[l, "i_F"'] \arrow[ldd, bend left, "f"] \\
& (K \oplus F)/D \arrow[d, "{\langle k, f\rangle_D}"] & \\
& H &
\end{tikzcd}$$

Figure 9.1: Universality property of the commutative coproduct.

Our main results depend on the following key lemma.

Lemma 9.2. *Let K and F be KATs. If F is finite, then every element of $(K \oplus F)/D$ can be expressed as a finite sum $\sum_{s \in F} p_s s$, where $p_s \in K$.*

Note that the lemma is not true in general without the assumption of finiteness. For example, it can be shown that the commutative coproduct of two copies of the free KA on one generator does not satisfy the lemma.

Proof. The lemma is certainly true of individual elements of K and F. We show that the property is preserved under the KAT operations. The cases of $+$ and $\cdot$ are quite easy, using commutativity and distributivity:

$$\begin{aligned}
\left(\sum_s p_s s\right) + \left(\sum_s p'_s s\right) &= \sum_s (p_s + p'_s) s \\
\left(\sum_s p_s s\right) \cdot \left(\sum_s p'_s s\right) &= \sum_s \left(\sum_s p_s s\right) \cdot p'_s s \\
&= \sum_s \sum_s p_s s p'_s s \\
&= \sum_s \sum_s p_s p'_s s s \\
&= \sum_s p_s p'_s s s.
\end{aligned}$$

The only difficult case is that of *. We wish to show that $(\sum_{s \in F} p_s s)^\star$ is equivalent to a finite sum of the form $\sum_{t \in F} q_t t$. Let $\Sigma = \{a_s \mid s \in F\}$ be a finite alphabet with one letter for each element of F, and let Reg_Σ be the free KA on generators Σ. Consider the following homomorphisms generated by the indicated actions on Σ:

$$\begin{array}{lll}
f \colon \Sigma^\star \to F & g \colon \mathsf{Reg}_\Sigma \to K \oplus F & h \colon \mathsf{Reg}_\Sigma \to K \\
f(a_s) = s & g(a_s) = p_s s & h(a_s) = p_s.
\end{array}$$

For each $t \in F$, the set $f^{-1}(t) = \{x \in \Sigma^\star \mid f(x) = t\}$ is a regular set, as it is the set accepted by the deterministic finite automaton with states F, start state 1, accept state t, and transitions $\delta(s, a) = s \cdot f(a)$. It is easily shown by induction that for all $x \in \Sigma^\star$, $\delta(s, x) = s \cdot f(x)$. Thus the automaton accepts x exactly when $t = \delta(1, x) = f(x)$, that is, when $x \in f^{-1}(t)$.

Let A be the $F \times F$ transition matrix of this automaton: $A_{st} = \sum_{sr=t} a_r$. Then $(A^\star)_{st}$ represents the set of strings x such that $s \cdot f(x) = t$. Moreover,

$$\left(\sum_{s\in F} a_s\right)^\star = \sum_{t\in F}(A^\star)_{1t} \tag{9.7}$$

since every string is accepted at some state t.

Let M be the $F \times F$ diagonal matrix with diagonal elements $M_{ss} = s$ and off-diagonal elements $M_{st} = 0$ for $s \neq t$. The homomorphisms g and h lift to $F \times F$ matrices over Reg_Σ with

$$g(A)_{st} = \sum_{sr=t} p_r r \qquad\qquad h(A)_{st} = \sum_{sr=t} p_r.$$

Then for any $s, t \in F$,

$$\begin{aligned}(M \cdot g(A))_{st} &= \sum_r M_{sr} g(A)_{rt} = M_{ss} g(A)_{st} = s \sum_{sr=t} p_r r \\ &= \sum_{sr=t} p_r sr = \sum_{sr=t} p_r t = h(A)_{st} M_{tt} \\ &= \sum_r h(A)_{sr} M_{rt} = (h(A) \cdot M)_{st}.\end{aligned}$$

Since s, t were arbitrary, $M \cdot g(A) = h(A) \cdot M$. By the bisimulation rule of KA (Proposition 4.4),

$$M \cdot g(A^\star) = M \cdot g(A)^\star = h(A)^\star \cdot M = h(A^\star) \cdot M,$$

thus for all $s, t \in F$,

$$\begin{aligned}sg(A^\star)_{st} = M_{ss} g(A^\star)_{st} &= \sum_{r\in F} M_{sr} g(A^\star)_{rt} = (M \cdot g(A^\star))_{st} \\ &= (h(A^\star) \cdot M)_{st} = \sum_{r\in F} h(A^\star)_{sr} M_{rt} \\ &= h(A^\star)_{st} M_{tt} = h(A^\star)_{st} t.\end{aligned}$$

In particular, setting $s = 1$ and summing over $t \in F$,

$$\sum_{t\in F} g(A^\star)_{1t} = \sum_{t\in F} h(A^\star)_{1t} t. \tag{9.8}$$

Using (9.7) and (9.8),

$$\begin{aligned}\left(\sum_{s\in F} p_s s\right)^\star = \left(\sum_{s\in F} g(a_s)\right)^\star = g\left(\left(\sum_{s\in F} a_s\right)^\star\right) = g\left(\sum_{t\in F} (A^\star)_{1t}\right) \\ = \sum_{t\in F} g(A^\star)_{1t} = \sum_{t\in F} h(A^\star)_{1t} t.\end{aligned}$$

Setting $q_t = h(A^\star)_{1t}$, we have expressed $(\sum_{s\in F} p_s s)^\star$ in the desired form. $\square$

Theorem 9.2. *If K is a KAT and F is the KAT of all binary relations on a finite set S, then* $(K \oplus F)/D \cong \mathrm{Mat}(S, K)$.

Proof. For $p \in K$, let $k(p) \in \mathrm{Mat}(S, K)$ be the $S \times S$ diagonal matrix with p on the main diagonal and 0 elsewhere. For $s \in F$, let $f(s)$ be the standard representation of the binary relation s as an $S \times S$ Boolean matrix. The maps $k\colon K \to \mathrm{Mat}(S, K)$ and $f\colon F \to \mathrm{Mat}(S, K)$ are injective KAT homomorphisms and embed K and F isomorphically in $\mathrm{Mat}(S, K)$. The image of F under f is $\mathrm{Mat}(S, \mathbb{2})$, a subalgebra of $\mathrm{Mat}(S, K)$. By the universality property for coproducts, we have that

$$\langle k, f\rangle\colon K \oplus F \to \mathrm{Mat}(S, K)$$

and k and f factor as $k = \langle k, f\rangle \circ i_K$ and $f = \langle k, f\rangle \circ i_F$.

Moreover, because $k(p)$ is a diagonal matrix for $p \in K$ and $f(s)$ is a Boolean matrix for $s \in F$, the commutativity conditions D are satisfied in the sense that $k(p)f(s) = f(s)k(p)$, thus Lemma 9.1 applies and we have a KAT homomorphism

$$\langle k, f\rangle_D\colon (K \oplus F)/D \to \mathrm{Mat}(S, K).$$

That this homomorphism is an isomorphism follows from Lemma 9.2 by an argument similar to that of Theorem 9.1. For $\alpha, \beta \in S$, let $n_{\alpha\beta} \in F$ such that $h(n_{\alpha\beta})_{\alpha\beta} = 1$ and all other entries are 0. Then for all $s \in F$,

$$n_{\alpha\alpha} s n_{\beta\beta} = \begin{cases} n_{\alpha\beta} & \text{if } h(s)_{\alpha\beta} = 1 \\ 0 & \text{if } h(s)_{\alpha\beta} = 0 \end{cases} \qquad\qquad \sum_\alpha n_{\alpha\alpha} = 1.$$

We have

$$h\left(\sum_s p_s s\right)_{\alpha\beta} = \sum_s p_s h(s)_{\alpha\beta} = \sum_{h(s)_{\alpha\beta}=1} p_s \tag{9.9}$$

$$\begin{aligned} \sum_s p_s s &= \sum_s p_s \left(\sum_\alpha n_{\alpha\alpha}\right) s \left(\sum_\beta n_{\beta\beta}\right) \\ &= \sum_{\alpha,\beta} \sum_s p_s n_{\alpha\alpha} s n_{\beta\beta} = \sum_{\alpha,\beta} \sum_{h(s)_{\alpha\beta}=1} p_s n_{\alpha\beta} \end{aligned} \tag{9.10}$$

If $h(\sum_s p_s s) = h(\sum_s q_s s)$, then for all $\alpha, \beta \in S$, $h(\sum_s p_s s)_{\alpha\beta} = h(\sum_s q_s s)_{\alpha\beta}$. By (9.9) and (9.10),

$$\sum_s p_s s = \sum_{\alpha,\beta} \sum_{h(s)_{\alpha\beta}=1} p_s n_{\alpha\beta} = \sum_{\alpha,\beta} \sum_{h(s)_{\alpha\beta}=1} q_s n_{\alpha\beta} = \sum_s q_s s.$$

The construction is illustrated in Figure 9.2. ☐

Corollary 9.1. *If K is a KAT and F is any KAT of binary relations on a finite set S, then $(K \oplus F)/D$ is isomorphic to a subalgebra of* $\mathrm{Mat}(S, K)$.

Proof. Compose an embedding of F into the KAT of all binary relations on S with the map f of Theorem 9.2. ☐

The following corollary says that the extension of an arbitrary KAT with mutable tests is conservative.

$$\begin{tikzcd}
K \arrow[r, "i_K"] \arrow[dr, "k"'] & K \oplus F \arrow[d, "\langle k, f\rangle"] & F \arrow[l, "i_F"'] \arrow[dl, "f"] \\
& \begin{array}{c}(K \oplus F)/D \\ \cong \mathrm{Mat}(S, K)\end{array} &
\end{tikzcd}$$

Figure 9.2: Matrix representation of the commutative coproduct.

Corollary 9.2. *If K is a KAT and F is any KAT of binary relations on a finite set S, then the commutative coproduct $(K \oplus F)/D$ is injective.*

Proof. The maps $k = \langle k, f\rangle \circ i_K : K \to \mathrm{Mat}(S, K)$ and $f = \langle k, f\rangle \circ i_F : F \to \mathrm{Mat}(S, K)$ are injective. By Theorem 9.2, $(K \oplus F)/D \cong \mathrm{Mat}(S, K)$, and k and f compose with this isomorphism to give the canonical injections from K and F, respectively, to $(K \oplus F)/D$. □

9.3 Completeness and Complexity

In Section 9.2, we showed that an arbitrary KAT K can be conservatively extended with a small amount of state in the form of a finite set of mutable tests and their corresponding mutation actions. As shown in Theorem 9.2, the resulting algebra is isomorphic to $\mathrm{Mat}(\mathsf{At}, K)$, where At is the set of atoms of the free Boolean algebra generated by the mutable tests.

In this section we prove three results. First, the KAT axioms along with the axioms B! for mutable tests and the commutativity conditions D are complete for the equational theory of $(K \oplus F_n)/D$ relative to the equational theory of K. This is quite a strong result in the sense that it holds for an arbitrary KAT K, regardless of its nature. In particular, for the special case in which K is the free KAT on some set of generators, the model $(K \oplus F_n)/D$ is the free KAT with mutable tests T_n. Most of the work for this result has already been done in Section 9.2.

The second result is that the equational theory B! is complete for PSPACE. This complexity class is characterized by alternating polynomial-time Turing machines [100].

The third result is that the equational theory of a free KAT augmented with mutable tests is complete for EXPSPACE, deterministic exponential space. This result is quite surprising, as both KAT and B! separately are complete for PSPACE, yet their combination is exponentially more complex in the worst case.

9.3.1 Completeness

Let K be an arbitrary KAT. Let KAT+B! denote the deductive system consisting of the axioms of KAT, the axioms for mutable tests B!, and the commutativity conditions D over a language of KAT terms with primitive action and test symbols interpreted in K as well as a set of mutable tests T_n. Let Δ_K be the diagram of K.

Theorem 9.3. *The axioms KAT+B!+Δ_K are complete for the equational theory of $(K \oplus F_n)/D$. In other words, the axioms KAT+B! are complete for the equational theory of $(K \oplus F_n)/D$ relative to the equational theory of K.*

Proof. Let e_1 and e_2 be expressions denoting elements of $(K \oplus F_n)/D$. By Theorem 9.1 and Lemma 9.2, we have

$$\mathrm{KAT}+\mathrm{B!}+\Delta_K \vdash e_1 = \sum_{\alpha,\beta\in\mathsf{At}} p_{\alpha\beta}\alpha?\beta!$$
$$\mathrm{KAT}+\mathrm{B!}+\Delta_K \vdash e_2 = \sum_{\alpha,\beta\in\mathsf{At}} q_{\alpha\beta}\alpha?\beta!.$$

If $(K \oplus F_n)/D \vDash e_1 = e_2$, we have under the canonical interpretation $\langle k, i\rangle$ that the matrices $\langle k, i\rangle(e_1)$ and $\langle k, i\rangle(e_2)$ are equal, thus for all $\alpha, \beta \in \mathsf{At}$,

$$p_{\alpha\beta} = \langle k, i\rangle(e_1)_{\alpha\beta} = \langle k, i\rangle(e_2)_{\alpha\beta} = q_{\alpha\beta},$$

and conversely. □

Corollary 9.3. *The axioms KAT+B! are complete for the equational theory of $(K \oplus F_n)/D$, where K is the free KAT on some set of generators.*

9.3.2 Complexity

Theorem 9.4. *The equational theory B! is PSPACE-complete.*

We note that neither the upper nor the lower bound follows from previous results. The upper bound does not follow from results on elimination of hypotheses [31, 71, 102], as axioms (i) and (ii) can be eliminated by these results, but not the others.

Proof. We first show that the problem of deciding $\alpha?\beta! \leq e$, where $\alpha, \beta \in \mathsf{At}$, is in PSPACE. We give an alternating polynomial-time algorithm that operates inductively on the structure of e.

To decide $\alpha?\beta! \leq t?$ or $\alpha?\beta! \leq t!$, using (9.4) we can ask whether $\alpha = \beta \leq t$ or $\beta = \alpha[t]$, respectively.

For addition, we have $\alpha?\beta! \leq e_1 + e_2$ if and only if $\alpha?\beta! \leq e_1$ or $\alpha?\beta! \leq e_2$. We nondeterministically choose one of these alternatives and check it recursively.

For multiplication, we have $\alpha?\beta! \leq e_1e_2$ if and only if there exists γ such that $\alpha?\gamma! \leq e_1$ and $\gamma?\beta! \leq e_2$. We guess γ nondeterministically using existential branching and check both conditions recursively using universal branching.

Finally, to check $\alpha?\beta! \leq e^\star$, by Theorem 9.1 it suffices to check that $\alpha?\beta! \leq e^k$ for some $0 \leq k < 2^n$. We guess k nondeterministically using existential branching. To check $\alpha?\beta! \leq e^k$, we guess γ nondeterministically using existential branching, and for each such γ, we check recursively using universal branching that $\alpha?\gamma! \leq e^{\lfloor k/2\rfloor}$ and $\gamma?\beta! \leq e^{\lceil k/2\rceil}$.

To decide the equational theory in PSPACE, we note that $e_1 \leq e_2$ if for all $\alpha, \beta \in \mathsf{At}$, if $\alpha?\beta! \leq e_1$, then $\alpha?\beta! \leq e_2$. The α and β can be chosen universally and the implication $\alpha?\beta! \leq e_1 \implies \alpha?\beta! \leq e_2$ checked in PSPACE.

To show PSPACE-hardness, we encode the membership problem for deterministic linear-bounded automata, a well known PSPACE-complete problem. Let M be a deterministic linear-bounded automaton with states Q and tape alphabet Γ. Let $x = x_1 \cdots x_n$ be an input string of length n over M's input alphabet. For $a \in \Gamma$, $q \in Q$, and $0 \leq i \leq n+1$, introduce mutable tests P_i^a and Q_i^q with the following intuitive meanings:

P_i^a = the symbol currently occupying tape cell i is a,
Q_i^q = M is currently in state q scanning tape cell i.

The operation of the machine is governed by a transition function $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{+1, -1\}$. Intuitively, the transition $\delta(p, a) = (q, b, d)$ means, "When in state p scanning symbol a, print b on that cell, move the head in direction d, and enter state q." For each such transition, consider the expressions

$$P_i^a?Q_i^p?\bar{P}_i^a!\bar{Q}_i^p!P_i^b!Q_{i+d}^q! \tag{9.11}$$

for all i. The part $P_i^a?Q_i^p?$ tests whether the machine is currently scanning a on cell i in state p. If so, $\bar{P}_i^a!\bar{Q}_i^p!P_i^b!Q_{i+d}^q!$ effects the transition to the new configuration as dictated by the transition function δ. The truth values of variables not mentioned do not change.

Assume that the input is delimited by left and right endmarkers $\vdash$ and $\dashv$, that M starts in its start state s scanning the left endmarker $\vdash$, that M never overwrites the endmarkers, and that before accepting, M erases its tape by writing a blank symbol $\sqcup$ on all tape cells except for the endmarkers, moves its head all the way to the left, and enters state t. The start and accept configurations are atoms

$$\begin{aligned} \text{start} &= Q_0^s\, P_0^{\vdash}\, P_1^{x_1}\, P_2^{x_2} \cdots P_n^{x_n}\, P_{n+1}^{\dashv}\, U \\ \text{accept} &= Q_0^t\, P_0^{\vdash}\, P_1^{\sqcup}\, P_2^{\sqcup} \cdots P_n^{\sqcup}\, P_{n+1}^{\dashv}\, V \end{aligned}$$

where U and V are the negations of the remaining variables. Let e be the sum of all expressions (9.11). Then M accepts x if and only if start?accept! $\le e^\star$. □

Let K be the free KAT on some set of generators. As shown in Corollary 9.3, the equational theory of $(K \oplus F_n)/D$ is completely axiomatized by KAT+B!.

Definition 9.1 (Automata on guarded strings [90]). An *automaton on guarded strings* over Σ and B is a four-tuple $M = (Q, \Delta, \mathsf{start}, \mathsf{final})$ where

- Q is the set of states;
- $\mathsf{start} \subseteq Q$ is the set of *start states*;
- $\mathsf{final} \subseteq Q$ is the set of *final states*;
- $\Delta \subseteq Q \times (\Sigma \cup \mathcal{B}) \times Q$ is the *transition relation*, where $\mathcal{B}$ is the set of composite tests built from the atomic tests B.

Theorem 9.5. *The set of equational consequences of KAT+B! (that is, the equational theory of a free KAT augmented with mutable tests) is EXPSPACE-complete.*

Proof. Let K be the free KAT on generators Σ and B. The atomic tests B are ordinary KAT tests and are not mutable. The set of equational consequences of KAT+B! coincides with the equational theory of the structure $(K \oplus F_n)/D$ (Corollary 9.3). This structure is isomorphic to the matrix algebra $\mathrm{Mat}(\mathsf{At}, K)$ (Theorem 9.2), where At is the set of 2^n atoms generated by the mutable tests T_n. Every element of $\mathrm{Mat}(\mathsf{At}, K)$ is an $\mathsf{At}\times\mathsf{At}$ matrix, each entry of which is a regular set of guarded strings over Σ, B. Regular sets of guarded strings are recognized by non-deterministic automata on guarded strings (Definition 9.1). In such an automaton, a transition of the form (s, p, t) with $p \in \Sigma$ is called an *action transition*, and one of the form (s, b, t) with $b \in \mathcal{B}$ is called a *test transition*. In particular, a test transition of the form $(s, 1, t)$ is called an ϵ-transition. We refer the reader to [90] for a definition of how these automata compute on guarded strings. Let $\mathcal{L}(s, t)$ be the set of the guarded strings x so that there is some computation on x starting from state s that ends in state t. The guarded automaton M recognizes the language of guarded strings

$$\bigcup_{s \in \mathsf{start}} \bigcup_{t \in \mathsf{final}} \mathcal{L}(s,t).$$

We extend this automaton model so that it recognizes matrices of regular sets of guarded strings. A *matrix automaton* is a tuple

$$M = (Q \times \mathsf{At}, \Delta, \mathsf{start}, \mathsf{final}),$$

where $\mathsf{start}, \mathsf{final} \colon \mathsf{At} \to 2^Q$ and $\Delta \subseteq (Q \times \mathsf{At}) \times (\Sigma \cup \mathcal{B}) \times (Q \times \mathsf{At})$. The automaton recognizes the $\mathsf{At} \times \mathsf{At}$ matrix L, each entry of which is a regular language of guarded strings:

$$L(\alpha, \beta) = \bigcup_{s \in \mathsf{start}(\alpha)} \bigcup_{t \in \mathsf{final}(\beta)} \mathcal{L}(\langle s, \alpha\rangle, \langle t, \beta\rangle).$$

We will now describe a construction similar to Kleene's theorem. Given a KAT + B! expression e over Σ, B, T_n we will give a matrix automaton that recognizes the matrix of languages denoted by e under its standard interpretation in the structure $\mathrm{Mat}(\mathsf{At}, K)$. For all base cases $p, b, t?, t!$ we define the set $Q = \{s_1, s_2\}$, the start states $\mathsf{start}(\alpha) = \{s_1\}$, and the accepting states $\mathsf{final}(\alpha) = \{s_2\}$, for every $\alpha \in \mathsf{At}$. We give the set Δ of transitions separately for each of these base cases:

- Case: action letter p in Σ. For every atom α, we put a transition $\langle s_1, \alpha\rangle \xrightarrow{p} \langle s_2, \alpha\rangle$.
- Case: arbitrary test b in $\mathcal{B}$. For every atom α, we have a transition $\langle s_1, \alpha\rangle \xrightarrow{b} \langle s_2, \alpha\rangle$.
- Case: mutable test $t?$. We put the transitions $\langle s_1, \alpha\rangle \xrightarrow{1} \langle s_2, \alpha\rangle$ for every $\alpha \leq t$.
- Case: primitive action $t!$. The automaton has the transitions $\langle s_1, \alpha\rangle \xrightarrow{1} \langle s_2, \alpha[t]\rangle$ for every α. Recall that $\alpha[t]$ is the modification of α so that t holds.

The remaining base cases are for 1 and 0. We define the corresponding automata as follows:

- For the case of 1, we have the trivial automaton with $Q = \{s\}$, $\mathsf{start}(\alpha) = \{s\}$, $\mathsf{final}(\alpha) = \{s\}$, and $\Delta = \emptyset$.
- The automaton for 0 is defined as $Q = \{s\}$, $\mathsf{start}(\alpha) = \{s\}$, $\mathsf{final}(\alpha) = \emptyset$, and $\Delta = \emptyset$.

Suppose that $M_1 = (Q_1 \times \mathsf{At}, \Delta_1, \mathsf{start}_1, \mathsf{final}_1)$ and $M_2 = (Q_2 \times \mathsf{At}, \Delta_2, \mathsf{start}_2, \mathsf{final}_2)$ are the matrix automata for the expressions e_1 and e_2 respectively. Without loss of generality the sets Q_1 and Q_2 are disjoint.

- For the expression $e_1 + e_2$ we define the automaton $M = (Q \times \mathsf{At}, \Delta, \mathsf{start}, \mathsf{final})$ by $Q = Q_1 \cup Q_2$,

$$\begin{aligned}\mathsf{start}(\alpha) &= \mathsf{start}_1(\alpha) \cup \mathsf{start}_2(\alpha)\\ \mathsf{final}(\alpha) &= \mathsf{final}_1(\alpha) \cup \mathsf{final}_2(\alpha)\end{aligned}$$

  and $\Delta = \Delta_1 \cup \Delta_2$.
- For the expression $e_1 \cdot e_2$ define $M = (Q \times \mathsf{At}, \Delta, \mathsf{start}, \mathsf{final})$ by $Q = Q_1 \cup Q_2$, $\mathsf{start}(\alpha) = \mathsf{start}_1(\alpha)$, $\mathsf{final}(\alpha) = \mathsf{final}_2(\alpha)$, and $\Delta = \Delta_1 \cup \Delta_2 \cup \Delta'$, where

$$\Delta' = \left\{\langle s, \alpha\rangle \xrightarrow{1} \langle t, \alpha\rangle \mid s \in \mathsf{final}_1(\alpha),\ t \in \mathsf{start}_2(\alpha)\right\}.$$

Now, suppose that $M = (Q \times \mathsf{At}, \Delta, \mathsf{start}, \mathsf{final})$ is the matrix automaton for the expression e. The automaton for $e \cdot e^\star$ results from M by adding ϵ-transitions from the final states back to the start states:

$$\langle s, \alpha\rangle \xrightarrow{1} \langle t, \alpha\rangle,$$

where $s \in \mathsf{final}(\alpha)$, $t \in \mathsf{start}(\alpha)$, and $\alpha \in \mathsf{At}$. Finally, the automaton for $e^\star = 1 + e \cdot e^\star$ can be obtained using the constructions for 1 and + that we have already described.

Consider now two KAT+B! expressions e_1, e_2 and the problem of checking whether they denote the same matrix in the structure Mat(At, K). We can construct effectively the corresponding matrix automata M_1 and M_2, as described in the previous paragraph. We can have an explicit representation of these automata, since exponential space suffices for this. Let L_1 and L_2 be the matrices of languages accepted by M_1 and M_2 respectively. For every pair of atoms α, β we have to check whether $L_1(\alpha, \beta) = L_2(\alpha, \beta)$. This problem amounts to checking the equivalence of automata on guarded strings, which can be done in space polynomial in the size of the automata [90]. It follows that we can decide whether $e_1 = e_2$ in exponential space.

For the lower bound, we encode the membership problem for exponential-space bounded Turing machines. Given such a machine M and an input x of length n, we use n mutable tests to construct an integer counter that can count up to $2^n - 1$, as illustrated below:

```
$\bar{t}_0!; \bar{t}_1!; \cdots ; \bar{t}_{n-1}!;$
while $\bar{t}_0? + \bar{t}_1? + \cdots + \bar{t}_{n-1}?$ {
  if $\bar{t}_0?$ then $t_0!;$
  else if $\bar{t}_1?$ then $\bar{t}_0!; t_1!;$
  else if $\bar{t}_2?$ then $\bar{t}_0!; \bar{t}_1!; t_2!;$
  else ...
  else if $\bar{t}_{n-1}?$ then $\bar{t}_0!; \bar{t}_1!; \cdots ; \bar{t}_{n-2}!; t_{n-1}!;$
  else skip;
}
```

We use the counter as a "yardstick" to construct an expression e that simulates a non-deterministic automaton which accepts all strings that are not valid computation histories of M on input x. The automaton decides nondeterministically where to look for an incorrect move of M. It remembers a few symbols of the input string, then starts the counter. With each iteration of the counter, it skips over an input symbol (not shown above). In this way it can compare symbols a distance 2^n apart to check whether the transition rules of M are followed. The expression e generates all strings if and only if M does not accept x. This construction is quite standard (see for example [51, 100, 132]), so we omit further details. □

9.4 Applications

9.4.1 The Böhm-Jacopini Theorem

A well-studied problem in program schematology is that of transforming unstructured flowgraphs to structured form. An early seminal result is the Böhm–Jacopini theorem [23], which states that any deterministic flowchart program is equivalent to a deterministic WHILE program. This theorem has reappeared in many contexts and has been reproved by many different methods [12, 118, 121, 123, 150].

Like most early work in program schematology, the Böhm–Jacopini theorem is usually formulated at the first-order level. This allows auxiliary individual or Boolean variables to be introduced to preserve information across computations. This is an essential ingredient of the Böhm–Jacopini construction, and they asked whether it was strictly necessary. This question was answered affirmatively by Ashcroft and Manna [12] and Kosaraju [87].

In [103], a purely propositional account of this negative result was given. A class of automata called *strictly deterministic automata* was presented, an abstraction of deterministic flowchart schemes. The three-state strictly deterministic automaton of Figure 9.3 was

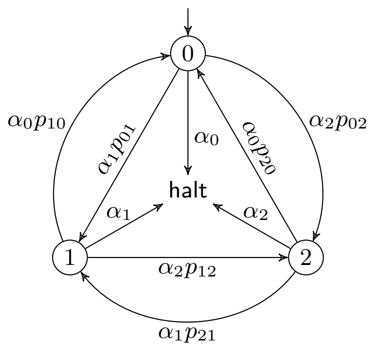


Figure 9.3: A strictly deterministic automaton not equivalent to any WHILE program [103].

```
$t_0$!; $\bar{t}_1$!; $\bar{t}_2$!; //start state is 0
while true {
  if $t_0$? then
    $\bar{t}_0$!; if $\alpha_1$ then $p_{01}$; $t_1$!; else if $\alpha_2$ then $p_{02}$; $t_2$!; else halt;
  else if $t_1$? then
    $\bar{t}_1$!; if $\alpha_2$ then $p_{12}$; $t_2$!; else if $\alpha_0$ then $p_{10}$; $t_0$!; else halt;
  else //must be $t_2$
    $\bar{t}_2$!; if $\alpha_0$ then $p_{20}$; $t_0$!; else if $\alpha_1$ then $p_{21}$; $t_1$!; else halt;
}
```

Figure 9.4: A WHILE program with mutable tests equivalent to Figure 9.3.

shown not to be equivalent to any deterministic WHILE program, where the α_i are mutually exclusive and exhaustive tests and the p_{ij} are primitive actions.

With strictly deterministic automata, Boolean values are provided by the environment in the form of an input string consisting of an infinite sequence of atoms, and the program responds with actions, including halting or failing. This is the correct propositional semantics: it allows all possible interpretations of the actions that could cause tests to become true or false. Two strictly deterministic automata are considered equivalent if they generate the same set of finite guarded strings (see [103] for formal definitions and details).

The Böhm–Jacopini theorem is true in the presence of mutable tests. The technique is well known, so rather than give a general account, we illustrate with the strictly deterministic automaton of Figure 9.3. We introduce mutable tests t_0, t_1, and t_2, which serve as program counters. An equivalent deterministic while program with mutable tests is shown in Figure 9.4.

The major difference here is that the mutable tests are under the control of the program instead of the environment.

We have not given the formal definition of the set of guarded strings generated by a strictly deterministic automaton with mutable tests, but under the appropriate definition, it can be shown that this WHILE program and the strictly deterministic automaton of Figure 9.3 generate the same set of guarded strings.

9.4.2 A Folk Theorem

In this section we illustrate how KAT + B! can be used in practice. We will show, reasoning equationally in KAT + B!, a classical result of program schematology: Every WHILE program can be simulated by a WHILE program with at most one WHILE loop, assuming that we allow extra Boolean variables. An example of part of the proof done using only KAT is shown in Section 4.3.1.

We work with a programming language that has atomic programs (written $a, b, \ldots$), the constant programs skip and fail, atomic tests, as well as the constructs: sequential composition $f; g$, conditional test if p then f else g, and iteration while p do f. These constructs are modeled in KAT as follows:

$$\mathsf{skip} = 1 \qquad \mathsf{fail} = 0 \qquad f; g = fg$$
$$\mathsf{if}\ e\ \mathsf{then}\ f\ \mathsf{else}\ g = ef + \bar{e}g \qquad \mathsf{while}\ e\ \mathsf{do}\ f = (ef)^\star \bar{e}$$

There is a semantic justification for these translations, using the standard relation-theoretic semantics for the input-output behavior of WHILE programs. Intuitively, to show the result we introduce extra Boolean variables that encode the control structure of the program. These variables are modeled in KAT + B! using mutable tests $t_1, t_2, \ldots$, which are taken to be disjoint from any mutable tests that might already appear in the program.

Commutativity axioms: KAT + B! has axioms that say that primitive actions commute with the mutable test symbols, that is, $t?a = at?$ and $t!a = at!$. Moreover, $t!p = pt!$ and $t!\bar{p} = \bar{p}t!$ for every atomic KAT test p, since tests commute. The following lemmas establish that using the axioms of KAT + B! more commutativity equations can be shown.

Lemma 9.3. *If the mutable test symbols $t, \bar{t}$ do not appear in the KAT + B! test term p, then we have that $t!p = pt!$ and $t!\bar{p} = \bar{p}t!$.*

Proof. By induction on p. If p is an atomic KAT test, then the claim follows directly from the axioms. The cases for the constants 0 and 1 are trivial. If p is a mutable test $s?$, then by our assumption we have that $s \neq t, \bar{t}$ and therefore $t!s? = s?t!$ and $t!\bar{s}? = \bar{s}?t!$ are axioms of B!. For the induction step, consider the case $p + q$:

$$t!(p + q) = t!p + t!q = pt! + qt! = (p + q)t!$$
$$t!(\overline{p + q}) = t!\bar{p}\bar{q} = \bar{p}t!\bar{q} = \bar{p}\bar{q}t! = (\overline{p + q})t!$$

The case pq is similar. For the case of $\bar{p}$, the equation $t!\bar{p} = \bar{p}t!$ follows from the induction hypothesis for p. Similarly, $t!\bar{\bar{p}} = t!p = pt! = \bar{\bar{p}}t!$. □

Lemma 9.4. *If the mutable test symbols $t, \bar{t}$ do not appear in the KAT + B! term f, then $t?f = ft?$ and $t!f = ft!$.*

Proof. We only show the part involving $t!$, for $t?$ the proof is essentially the same. We argue by induction on the structure of f. If f is a test, then the result follows from Lemma 9.3. If f is an atomic program a, then from the axioms we have that $t!a = at!$. For composition and choice we have using the induction hypothesis: $t!fg = ft!g = fgt!$, and

$$t!(f + g) = t!f + t!g = ft! + gt! = (f + g)t!.$$

It remains to show that $t!f^\star = f^\star t!$. By virtue of the bisimulation rule, it suffices to see that $t!f = ft!$, which is the induction hypothesis. □

The theorem that follows is a normal form theorem, from which the result we want to show follows immediately. Working in a bottom-up fashion, every WHILE program term is put into the normal form. That the transformed program in normal form is equivalent to the original one is shown in KAT + B!.

Theorem 9.6. *For any WHILE program f, there are WHILE-free u, p, ϕ and a finite collection $t_1, \ldots, t_k$ of extra mutable tests such that*

$$f; z = u; \mathsf{while}\, p\, \mathsf{do}\, \phi; z,$$

where $z = \bar{t}_1!; \ldots; \bar{t}_k!$.

Proof. In the normal form given above, the pre-computation u, the WHILE-guard p, and the WHILE-body ϕ may involve the extra mutable test symbols $t_1, \ldots, t_k, \bar{t}_1, \ldots, \bar{t}_k$. These symbols do not appear in f. The post-computation $z = \bar{t}_1!; \ldots; \bar{t}_k$ "zeroes out" all the extra mutable Boolean variables. Its role is in some sense to simply project out this extra finite state. The proof proceeds by induction on the structure of the WHILE program term f.

For the base case, suppose that f is a WHILE-free program term, and let t be a fresh mutable test symbol. Intuitively, $t?$ holds if f has not been executed yet, and $\bar{t}?$ holds when f has been executed. The base case follows from Lemma 9.5.

From the induction hypothesis, we can bring the programs f and g in normal form so that

$$f; z = u; \mathsf{while}\, p\, \mathsf{do}\, \phi; z$$

and

$$g; z = v; \mathsf{while}\, q\, \mathsf{do}\, \psi; z,$$

where z sets to zero all the mutable tests that appear in the transformations of f and g. For the cases of a conditional test $\mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g$ and composition $f; g$, we introduce a fresh mutable test symbol t.

- Case $\mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g$: The Boolean variable corresponding to the symbol t records the branch to be taken. So, $t?$ holds when f should be executed, and $\bar{t}?$ holds when g should be executed. The case follows from Lemma 9.6.

- Case $f; g$: The Boolean variable for the symbol t records the current position of execution. So, $t?$ holds when we are executing f, and $\bar{t}?$ when we are executing g. The case follows from Lemma 9.7.

It remains to handle the case of the WHILE loop $\mathsf{while}\, e\, \mathsf{do}\, f$. First, we have by Lemma 9.8 that

$$\mathsf{while}\, e\, \mathsf{do}\, f; z = \mathsf{while}\, e\, \mathsf{do}\, (f; z); z.$$

With this, we can put the program in a more convenient form with Lemma 9.9:

$$\mathsf{if}\, e\, \mathsf{then}\, \Big(u; \mathsf{while}\, (e + p)\, \mathsf{do}\, \mathsf{if}\, p\, \mathsf{then}\, \phi\, \mathsf{else}\, (z; u)\Big); z.$$

But we already know how to transform conditional statements, so we apply that transformation to bring the term in the desired normal form. □

Lemma 9.5. $f; z = t!; \mathsf{while}\, t?\, \mathsf{do}\, (f; \bar{t}!); z$, *where* $z = \bar{t}!$.

Proof. First, we unravel the expression $(f?f\bar{t}!)^\star$ twice and observe that

$$\begin{aligned}(t?f\bar{t}!)^\star &= 1 + t?f\bar{t}!(t?f\bar{t}!)^\star\\ &= 1 + t?f\bar{t}!(1 + t?f\bar{t}!(t?f\bar{t}!)^\star)\\ &= 1 + t?f\bar{t}! + t?f\bar{t}!t?f\bar{t}!(t?f\bar{t}!)^\star\\ &= 1 + t?f\bar{t}!,\end{aligned}$$

because $\bar{t}!t? = \bar{t}!\bar{t}?t? = 0$. So, we conclude that

$$\begin{aligned}\mathrm{RHS} &= t!(t?f\bar{t}!)^\star\bar{t}?\bar{t}!\\ &= t!(1 + t?f\bar{t}!)\bar{t}?\\ &= t!\bar{t}? + t!t?f\bar{t}!\bar{t}?,\end{aligned}$$

which is equal to $t!f\bar{t}! = ft!\bar{t}! = f\bar{t}! = f; z$, since t was chosen to be fresh (Lemma 9.4). □

Lemma 9.6 (Normal form for conditional). *The program* (if e then f else g); z; $\bar{t}!$ *is equal to*

if e then $(t!; u)$ else $(\bar{t}!; v)$;
while $((t? \wedge p) \vee (\bar{t}? \wedge q))$ do (if $t?$ then ϕ else ψ);
$z; \bar{t}!$.

Proof. The WHILE-free pre-computation in the normal form translation is equal to $et!u + \bar{e}\bar{t}!v$. The guard of the WHILE loop is $t?p + \bar{t}?q$, and the body is $t?\phi + \bar{t}?\psi$. So,

$$\begin{aligned}&((t? \wedge p) \vee (\bar{t}? \wedge q)); (\text{if } t? \text{ then } \phi \text{ else } \psi) =\\ &(t?p + \bar{t}?q)(t?\phi + \bar{t}?\psi) =\\ &t?p\phi + \bar{t}?q\psi.\end{aligned}$$

The negation of the guard of the loop is $\neg(t?p+\bar{t}?q) = (\bar{t}?+\bar{p})(t?+\bar{q}) = \bar{t}?\bar{q}+t?\bar{p}+\bar{p}\bar{q}$.

First, we claim that $t?(t?p\phi)^\star = t?(p\phi)^\star$. Since $t? \leq 1$ and * is monotone, we have that $(t?p\phi)^\star \leq (p\phi)^\star$, and therefore $t?(t?p\phi)^\star \leq t?(p\phi)^\star$. In order to show that $t?(p\phi)^\star \leq t?(t?p\phi)^\star$, it suffices to see that $t? \leq t?(t?p\phi)^\star$, and that

$$\begin{aligned}t?(t?p\phi)^\star p\phi &= t?(1 + (t?p\phi)^\star t?p\phi)p\phi\\ &= t?p\phi + t?(t?p\phi)^\star t?p\phi p\phi\\ &= t?t?p\phi + t?(t?p\phi)^\star t?t?p\phi p\phi\\ &= t?t?p\phi + t?(t?p\phi)^\star t?p\phi t?p\phi\\ &= t?(1 + (t?p\phi)^\star t?p\phi)t?p\phi\\ &= t?(t?p\phi)^\star t?p\phi \leq t?(t?p\phi)^\star.\end{aligned}$$

Now, we want to show that $t?(t?p\phi + \bar{t}?q\psi)^\star = t?(t?p\phi)^\star$. By monotonicity of *, the right-hand side is less than or equal to the left-hand side. For the other direction, we need to show that

$$\begin{aligned}&t?(t?p\phi)^\star(t?p\phi + \bar{t}?q\psi) = &&\\ &t?t?(t?p\phi)^\star(t?p\phi + \bar{t}?q\psi) = && [\text{prev. claim}]\\ &t?t?(p\phi)^\star(t?p\phi + \bar{t}?q\psi) = && [t \text{ not in } p, \phi]\\ &t?(p\phi)^\star t?(t?p\phi + \bar{t}?q\psi) = &&\\ &t?(p\phi)^\star t?p\phi = && [\text{prev. claim}]\\ &t?(t?p\phi)^\star t?p\phi, &&\end{aligned}$$

which is $\leq t?(t?p\phi)^\star$.

Let W abbreviate the entire WHILE loop of the normal form translation. We have already seen that

$$W = (t?p\phi + \bar{t}?q\psi)^\star(\bar{t}?\bar{q} + t?\bar{p} + \bar{p}\bar{q})$$

and therefore

$$\begin{aligned} t?W &= t?(t?p\phi)^\star(\bar{t}?\bar{q} + t?\bar{p} + \bar{p}\bar{q}) \\ &= t?(p\phi)^\star(\bar{t}?\bar{q} + t?\bar{p} + \bar{p}\bar{q}) \\ &= (p\phi)^\star t?(\bar{t}?\bar{q} + t?\bar{p} + \bar{p}\bar{q}) \\ &= (p\phi)^\star(t?\bar{p} + t?\bar{p}\bar{q}) \\ &= (p\phi)^\star t?\bar{p}, \end{aligned}$$

because $t?\bar{p}\bar{q} \leq t?\bar{p}$. So, we have

$$\begin{aligned} e\text{RHS} &= e(et!u + \bar{e}\bar{t}!v)Wz\bar{t}! = et!uWz\bar{t}! \\ &= et!t?uWz\bar{t}! = et!ut?Wz\bar{t}! \\ &= et!u(p\phi)^\star t?\bar{p}z\bar{t}! = eu(p\phi)^\star\bar{p}z\bar{t}!, \end{aligned}$$

which is equal to $efz\bar{t}!$ by the induction hypothesis. Similarly, it can be shown $\bar{e}\text{RHS} = \bar{e}gz\bar{t}!$. We thus conclude that

$$\begin{aligned} \text{RHS} &= (e + \bar{e})\text{RHS} = e\text{RHS} + \bar{e}\text{RHS} \\ &= efz\bar{t}! + \bar{e}gz\bar{t}! = (ef + \bar{e}g)z\bar{t}!, \end{aligned}$$

which is equal to (if e then f else g); z; $\bar{t}!$, namely the left-hand size of the equation we wanted to show. □

Lemma 9.7 (Normal form for composition). *The program $f; g; z; \bar{t}!$ is equal to*

```
$t!; u;$
while $(t? \vee (\bar{t}? \wedge q))$ do
    if $t?$ then (if $p$ then $\phi$ else $(z; \bar{t}!; v)$) else $\psi$;
$z; \bar{t}!.$
```

Proof. The negation of the guard of the while loop is $\neg(t? + \bar{t}?q) = \bar{t}?(t? + \bar{q}) = \bar{t}?\bar{q}$. The body of the loop is equal to $t?(p\phi + \bar{p}z\bar{t}!v) + \bar{t}?\psi = t?p\phi + t?\bar{p}z\bar{t}!v + \bar{t}?\psi$. So, the encoding of the WHILE loop is

$$\begin{aligned} &((t? + \bar{t}?q)(t?p\phi + t?\bar{p}z\bar{t}!v + \bar{t}?\psi))^\star\,\bar{t}?\bar{q} \\ &= (t?p\phi + t?\bar{p}z\bar{t}!v + \bar{t}?q\psi)^\star\bar{t}?\bar{q} \\ &= (A + \bar{t}?q\psi)^\star\bar{t}?\bar{q} \\ &= A^\star(\bar{t}?q\psi A^\star)^\star\bar{t}?\bar{q}, \end{aligned}$$

where we put $A = t?p\phi + t?\bar{p}z\bar{t}!v$.

From $\bar{t}?A = \bar{t}?(t?p\phi + t?\bar{p}z\bar{t}!v) = 0 \leq \bar{t}?$ we obtain that $\bar{t}?A^\star \leq \bar{t}?$. Moreover, $\bar{t}? \leq \bar{t}?A$ and hence $\bar{t}?A^\star = \bar{t}?$. It follows that $\bar{t}?q\psi A^\star = q\psi\bar{t}?A^\star = q\psi\bar{t}?$. Now, we claim that $(q\psi\bar{t}?)^\star\bar{t}? = \bar{t}?(q\psi)^\star$. The inequality $(q\psi\bar{t}?)^\star\bar{t}? \leq \bar{t}?(q\psi)^\star$ follows from monotonicity of *. For the inequality $\bar{t}?(q\psi)^\star \leq (q\psi\bar{t}?)^\star\bar{t}?$ we need to show that

$$(q\psi\bar{t}?)^\star\bar{t}?q\psi = (q\psi\bar{t}?)^\star q\psi\bar{t}? = (q\psi\bar{t}?)^\star q\psi\bar{t}?\bar{t}?,$$

which is $\leq (q\psi\bar{t}?)^\star\bar{t}?$. We have thus shown that the WHILE loop is equal to

$$A^\star(q\psi\bar{t}?)^\star\bar{t}?q = A^\star\bar{t}?(q\psi)^\star\bar{q}.$$

Now, we focus on simplifying the expression $t?A^\star\bar{t}? = t?(t?p\phi + t?\bar{p}z\bar{t}!v)^\star\bar{t}?$. First, we observe that unfolding $(t?\bar{p}z\bar{t}!v)^\star$ twice gives us the equation

$$(t?\bar{p}z\bar{t}!v)^\star = 1 + t?\bar{p}z\bar{t}!v.$$

Moreover, $\bar{t}?(t?p\phi)^\star = \bar{t}?(1 + t?p\phi(t?p\phi)^\star) = \bar{t}?$. Therefore, using the denesting rule, we obtain that $t?A^\star\bar{t}?$ is equal to

$$\begin{aligned}
&t?(t?p\phi)^\star(t?\bar{p}z\bar{t}!v(t?p\phi)^\star)^\star\bar{t}? = \\
&t?(t?p\phi)^\star(t?\bar{p}z\bar{t}!v\bar{t}?(t?p\phi)^\star)^\star\bar{t}? = \\
&t?(t?p\phi)^\star(t?\bar{p}z\bar{t}!v\bar{t}?)^\star\bar{t}? = \\
&t?(t?p\phi)^\star(t?\bar{p}z\bar{t}!v)^\star\bar{t}? = \\
&t?(t?p\phi)^\star(1 + t?\bar{p}z\bar{t}!v)\bar{t}? = \\
&t?(t?p\phi)^\star\bar{t}? + t?(t?p\phi)^\star t?\bar{p}z\bar{t}!v\bar{t}? = \\
&t?(t?p\phi)^\star t?\bar{p}z\bar{t}!v\bar{t}? = \\
&t?(p\phi)^\star\bar{p}z\bar{t}!v.
\end{aligned}$$

Finally, we can work on the right-hand side of the equation we want to establish:

$$\begin{aligned}
\text{RHS} &= t!uA^\star\bar{t}?(q\psi)^\star\bar{q}z\bar{t}! = t!ut?A^\star\bar{t}?(q\psi)^\star\bar{q}z\bar{t}! \\
&= t!ut?(p\phi)^\star\bar{p}z\bar{t}!v(q\psi)^\star\bar{q}z\bar{t}! = u(p\phi)^\star\bar{p}zv(q\psi)^\star\bar{q}z\bar{t}!,
\end{aligned}$$

which is equal by the induction hypothesis to $fzgz\bar{t}! = fgzz\bar{t}! = f; g; z; \bar{t}!$. □

Lemma 9.8. $\mathsf{while}\ e\ \mathsf{do}\ f; z = \mathsf{while}\ e\ \mathsf{do}\ (f; z); z$.

Proof. The left-hand side is equal to $(ef)^\star\bar{e}z$, and the right-hand side is equal to $(efz)^\star\bar{e}z$. It suffices to show that $(ef)^\star z = (efz)^\star z$.

$$(efz)^\star z \leq (ef)^\star z \Longleftarrow efz(ef)^\star z \leq (ef)^\star z,$$

which holds because $efz(ef)^\star z = ef(ef)^\star zz \leq (ef)^\star z$. Now, we observe that $(efz)^\star z = z(efz)^\star$ by the bisimulation rule, because $efzz = zefz$ (both are equal to efz). So,

$$(ef)^\star z \leq (efz)^\star z \Longleftarrow ef(efz)^\star z \leq (efz)^\star z,$$

which holds because $ef(efz)^\star z = ef(efz)^\star zz = efz(efz)^\star z \leq (efz)^\star z$. □

Lemma 9.9 (Normal form for loop). *The program* $(\mathsf{while}\ e\ \mathsf{do}\ f); z$ *is equal to*

$$\mathsf{if}\ e\ \mathsf{then}\ \Big(u; \mathsf{while}\ (e+p)\ \mathsf{do}\ \mathsf{if}\ p\ \mathsf{then}\ \phi\ \mathsf{else}\ (z; u)\Big); z.$$

Proof. The above program is equal to

$$\begin{aligned}
&\bar{e}z + eu((e+p)(p\phi + \bar{p}zu))^\star(\overline{e+p})z = \\
&\bar{e}z + eu(ep\phi + e\bar{p}zu + p\phi)^\star\bar{e}\bar{p}z = \\
&\bar{e}z + eu(p\phi + e\bar{p}zu)^\star\bar{e}\bar{p}z,
\end{aligned}$$

because $ep\phi \leq p\phi$. Using the denesting rule (9.1) and then the sliding rule (9.2), we see that this is equal to

$$\begin{aligned}
&\bar{e}z + eu(p\phi)^\star(e\bar{p}zu(p\phi)^\star)^\star\bar{e}\bar{p}z = \\
&\bar{e}z + eu(p\phi)^\star(\bar{p}zeu(p\phi)^\star)^\star\bar{e}\bar{p}z = \\
&\bar{e}z + (eu(p\phi)^\star\bar{p}z)^\star eu(p\phi)^\star\bar{e}\bar{p}zz = \\
&\bar{e}z + (efz)^\star eu(p\phi)^\star\bar{p}z\bar{e}z = \\
&\bar{e}z + (efz)^\star(efz)\bar{e}z = \\
&(1 + (efz)^\star(efz))\bar{e}z,
\end{aligned}$$

which is equal to $(efz)^\star\bar{e}z = \mathsf{while}\ e\ \mathsf{do}\ (f;z); z = (\mathsf{while}\ e\ \mathsf{do}\ f); z$. □

9.5 Conclusion

We have shown how to axiomatically extend Kleene algebra with tests with a finite amount of mutable state. This extra feature allows certain program transformations to be effected at the propositional level without passing to a full first-order system. The extension is conservative and deductively complete relative to the theory of the underlying algebra. The full theory is decidable and complete for EXPSPACE. We have given a representation theorem of the free models in terms of matrices.

An intriguing open problem is whether the coproduct of two KATs is injective. We have shown that it is if one of the two cofactors is a KAT of binary relations on a finite set.

9.6 Acknowledgments

Thanks to Bob Constable, Nate Foster, Fritz Henglein, Mark Reitblatt, Ross Tate, and Laure Thompson for valuable conversations and insights.

Bibliography

[1] A. V. Aho. Algorithms for finding patterns in strings. In J. v. Leeuwen, editor, *Handbook of theoretical computer science*. Vol. Algorithms and Complexity (A), pp. 255–300. Elsevier and MIT Press, 1990.
ISBN: 0-444-88071-2 (cited on p. 142).

[2] E. Allender and I. Mertz. Complexity of regular functions. In A.-H. Dediu, E. Formenti, C. Martín-Vide, and B. Truthe, editors, *Language and automata theory and applications*. Vol. 8977, in Lecture Notes in Computer Science, pp. 449–460. Springer International Publishing, 2015.
DOI: 10.1007/978-3-319-15579-1_35 (cited on p. 143).

[3] R. Alur, L. D'Antoni, and M. Raghothaman. DReX: a declarative language for efficiently evaluating regular string transformations. In *Proc. 42nd ACM symposium on principles of programming languages (POPL'15)*, 2015.
DOI: 10.1145/2676726.2676981 (cited on pp. 109, 118, 135).

[4] R. Alur and P. Černý. Expressiveness of streaming string transducers. In *Proc. foundations of software technology and theoretical computer science (FSTTCS)*, 2010.
DOI: 10.4230/LIPIcs.FSTTCS.2010.1 (cited on pp. 109, 116, 118, 143).

[5] R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. *ACM SIGPLAN notices*, 46(1):599–610, 2011.
DOI: 10.1145/1925844.1926454 (cited on pp. 109, 118).

[6] R. Alur, A. Freilich, and M. Raghothaman. Regular combinators for string transformations. In *Proc. of the joint meeting of the 23rd EACSL annual conference on computer science logic (CSL) and the 29th annual ACM/IEEE symposium on logic in computer science (LICS'14)*. In CSL-LICS '14. ACM, Vienna, Austria, 2014, 9:1–9:10.
DOI: 10.1145/2603088.2603151 (cited on pp. 118, 143).

[7] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT symposium on principles of programming languages*. In POPL '14. ACM, San Diego, California, USA, 2014, pp. 113–126.
DOI: 10.1145/2535838.2535862 (cited on pp. 70, 163).

[8] A. Angus and D. Kozen. Kleene algebra with tests and program schematology. Tech. rep. (TR2001-1844). Computer Science Department, Cornell University, July 2001.
URL: http://hdl.handle.net/1813/5831 (cited on pp. 163, 164).

[9] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. comput. sci.*, 155(2):291–319, 1996.
DOI: http://dx.doi.org/10.1016/0304-3975(95)00182-4 (cited on pp. 108, 142).

[10] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
ISBN: 0521582741.
URL: http://dl.acm.org/citation.cfm?id=522388 (cited on p. 126).

[11] ASCII subset of unicode.
URL: http://www.unicode.org/charts/PDF/U0000.pdf (cited on p. 8).

[12] E. Ashcroft and Z. Manna. The translation of GOTO programs into WHILE programs. In *Proceedings of IFIP congress 71*. C. Freiman, J. Griffith, and J. Rosenfeld, editors. Vol. 1. North-Holland, 1972, pp. 250–255.
ISBN: 0-917072-14-6 (cited on pp. 163, 176).

[13] P. Balbiani, A. Herzig, and N. Troquard. Dynamic logic of propositional assignments: a well-behaved variant of PDL. In *Proc. 28th symp. logic in computer science (LICS'13)*. ACM/IEEE, 2013, pp. 143–152.
DOI: 10.1109/LICS.2013.20 (cited on p. 164).

[14] H. Barendregt. *The lambda calculus: its syntax and semantics*. Vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
ISBN: 978-0-444-87508-2 (cited on pp. 60, 151, 153).

[15] A. Barth and D. Kozen. Equational verification of cache blocking in LU decomposition using Kleene algebra with tests. Tech. rep. (TR2002-1865). Computer Science Department, Cornell University, June 2002.
URL: http://hdl.handle.net/1813/5848 (cited on p. 163).

[16] M.-P. Béal and O. Carton. Determinization of transducers over finite and infinite words. *Theoretical computer science*, 289(1):225–251, Oct. 2002.
DOI: 10.1016/S0304-3975(01)00271-7 (cited on p. 116).

[17] H. Bekić. Definable operations in general algebras, and the theory of automata and flowcharts. In C. Jones, editor, *Programming languages and their definition*. Vol. 177, in Lecture Notes in Computer Science, pp. 30–55. Springer Berlin Heidelberg, 1984.
DOI: 10.1007/BFb0048939 (cited on p. 151).

[18] J. Berstel. *Transductions and Context-Free Languages*. Teubner Stuttgart, 1979.
URL: http://www-igm.univ-mlv.fr/~berstel/LivreTransductions/LivreTransductions.html (cited on pp. 109, 111, 116, 118, 143).

[19] P. Bille and M. Thorup. Faster regular expression matching. In S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. Nikoletseas, and W. Thomas, editors, *Automata, languages and programming*. Vol. 5555, in Lecture Notes in Computer Science, pp. 171–182. Springer Berlin Heidelberg, 2009.
DOI: 10.1007/978-3-642-02927-1_16 (cited on p. 142).

[20] P. Bille and M. Thorup. Regular expression matching with multi-strings and intervals. In *Proc. 21st ACM-SIAM symposium on discrete algorithms (SODA'10)*, 2010.
URL: http://dl.acm.org/citation.cfm?id=1873601.1873705 (cited on p. 142).

[21] N. Bjørner and M. Veanes. Symbolic transducers. Tech. rep. (MSR-TR-2011-3). Microsoft Research, 2011 (cited on p. 144).

[22] S. L. Bloom and Z. Ésik. *Iteration theories*. Springer, 1993.
DOI: 10.1007/978-3-642-78034-9 (cited on p. 147).

[23] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*:366–371, May 1966.
DOI: 10.1145/355592.365646 (cited on pp. 163, 176).

[24] B. Brodie, D. Taylor, and R. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *Acm sigarch computer architecture news*, 34(2):202, 2006.
DOI: 10.1145/1150019.1136500 (cited on p. 143).

[25] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and computation*, 140(2):229–253, 1998.
DOI: 10.1006/inco.1997.2688 (cited on pp. 89, 108).

[26] A. Brüggemann-Klein. Regular expressions into finite automata. *Theor. comput. sci.*, 120(2):197–213, 1993.
DOI: http://dx.doi.org/10.1016/0304-3975(93)90287-4 (cited on p. 143).

[27] J. A. Brzozowski. Derivatives of regular expressions. *J. acm*, 11(4):481–494, 1964.
DOI: 10.1145/321239.321249 (cited on pp. 142, 143).

[28] N. Chomsky. Three models for the description of language. *IEEE transactions on information theory*, 2(3):113–124, 1956.
DOI: 10.1109/TIT.1956.1056813 (cited on p. 58).

[29] N. Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959.
DOI: 10.1016/S0019-9958(59)90362-6 (cited on p. 58).

[30] T. Christiansen, B. D. Foy, L. Wall, and J. Orwant. *Programming Perl*. O'Reilly Media, 4th edition ed., 2012.
ISBN: 978-0-596-00492-7 (cited on pp. 7, 83, 94, 106, 127).

[31] E. Cohen. Hypotheses in Kleene algebra. Tech. rep. (Technical Report TM-ARH-023814). Bellcore, 1993.
URL: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.6067 (cited on pp. 163, 173).

[32] E. Cohen. Lazy caching in Kleene algebra. Tech. rep. Bellcore, 1994.
URL: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.5074 (cited on p. 163).

[33] E. Cohen. Using Kleene algebra to reason about concurrency control. Tech. rep. Telcordia, 1994 (cited on p. 163).

[34] E. Cohen, D. Kozen, and F. Smith. The complexity of Kleene algebra with tests. Tech. rep. (TR96-1598). Computer Science Department, Cornell University, July 1996.
URL: http://hdl.handle.net/1813/7253 (cited on pp. 65, 165).

[35] T. Colcombet. Forms of determinism for automata. In *Proc. 29th symposium on theoretical aspects of computer science (stacs)*. Vol. 14. LIPIcs, 2012, pp. 1–23.
DOI: 10.4230/LIPIcs.STACS.2012.1 (cited on p. 143).

[36] J. H. Conway. *Regular algebra and finite machines*. Dover Publications, 2012.
ISBN: 978-0-486-31058-9 (cited on p. 51).

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. Of *The MIT Electrical Engineering and Computer Science Series*. MIT Press and McGraw-Hill, 2nd edition ed., 2001.
ISBN: 0-262-53196-8 (cited on p. 98).

[38] B. Courcelle. Equivalences and transformations of regular systems – applications to recursive program schemes and grammars. *Theoretical computer science*, 42:1–122, 1986.
DOI: 10.1016/0304-3975(86)90050-2 (cited on pp. 147, 148).

[39] R. Cox. Regular expression matching can be simple and fast.
URL: http://swtch.com/~rsc/regexp/regexp1.html (cited on p. 87).

[40] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *Proceedings of the 41th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL'14)*. ACM Press, San Diego, California, Jan. 2014.
DOI: 10.1145/2535838.2535849 (cited on p. 144).

[41] D. Debarbieux, O. Gauwin, J. Niehren, T. Sebastian, and M. Zergaoui. Early nested word automata for XPath query answering on XML streams. In S. Konstantinidis, editor, *Proc. 18th international conference on implementation and application of automata (CIAA'13)*. Vol. 7982, in Lecture Notes in Computer Science, pp. 292–305. Springer Berlin Heidelberg, July 2013.
DOI: 10.1007/978-3-642-39274-0_26 (cited on p. 106).

[42] D. Dubé and M. Feeley. Efficiently Building a Parse Tree From a Regular Expression. *Acta informatica*, 37(2):121–144, 2000.
DOI: 10.1007/s002360000037 (cited on pp. 73, 83, 87, 89, 106).

[43] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
DOI: 10.1145/362007.362035 (cited on pp. 7, 87).

[44] K. Ellul, B. Krawetz, J. Shallit, and M.-w. Wang. Regular expressions: new results and open problems. *Journal of automata, languages and combinatorics*, 10(4):407–437, 2005.
URL: http://dl.acm.org/citation.cfm?id=1103362.1103368 (cited on p. 143).

[45] E. Engeler. Algorithmic properties of structures. *Mathematical systems theory*, 1(2):183–195, 1967.
DOI: 10.1007/BF01705528 (cited on p. 63).

[46] J. Engelfriet and H. Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM transactions on computational logic (TOCL)*, 2(2):216–254, 2001.
DOI: 10.1145/371316.371512 (cited on p. 143).

[47] Z. Ésik. Completeness of Park induction. *Theoretical computer science*, 177(1):217–283, 1997.
DOI: 10.1016/S0304-3975(96)00240-X (cited on p. 151).

[48] Z. Ésik and W. Kuich. Modern automata theory. Unpublished manuscript. 2007 (cited on pp. 148, 150).

[49] Z. Ésik and H. Leiß. Algebraically complete semirings and Greibach normal form. *Annals of pure and applied logic*, 133:173–203, 2005.
DOI: 10.1016/j.apal.2004.10.008 (cited on pp. 60, 147, 148, 151, 153–155).

[50] Z. Ésik and H. Leiß. Greibach normal form in algebraically complete semirings. In *CSL '02: proceedings of the 16th international workshop and 11th annual conference of the eacsl on computer science logic*. Springer-Verlag, London, UK, 2002, pp. 135–150.
DOI: 10.1007/3-540-45793-3_10 (cited on pp. 60, 147, 148, 151, 153–155).

[51] J. Ferrante and C. W. Rackoff. *The computational complexity of logical theories*. Vol. 718 of *Lecture Notes in Mathematics*. Springer Berlin Heidelberg, 1979.
DOI: 10.1007/BFb0062837 (cited on p. 176).

[52] S. Fischer, F. Huch, and T. Wilke. A play on regular expressions: functional pearl. In *Proc. of the 15th ACM SIGPLAN international conference on functional programming (ICFP'10)*. ACM, Baltimore, Maryland, USA, 2010, pp. 357–368.
DOI: 10.1145/1863543.1863594 (cited on p. 87).

[53] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on principles of programming languages*. In POPL '04. ACM, Venice, Italy, 2004, pp. 111–122.
DOI: 10.1145/964001.964011 (cited on p. 142).

[54] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva. Probabilistic NetKAT. Tech. rep. Computing and Information Science, Cornell University, July 2015.
URL: http://hdl.handle.net/1813/40335 (cited on p. 70).

[55] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for NetKAT. In *Proc. 42nd acm sigplan-sigact symp. principles of programming languages (POPL'15)*. ACM. Mumbai, India, Jan. 2015, pp. 343–355.
DOI: 10.1145/2676726.2677011 (cited on p. 70).

[56] G. Fowler. An interpretation of the POSIX regex standard. Jan. 2003.
URL: http://www2.research.att.com/~astopen/testregex/re-interpretation.html (cited on p. 94).

[57] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. 31st international colloquium on automata, languages and programming (ICALP'04)*. Vol. 3142. In Lecture Notes in Computer Science (LNCS). Springer, July 2004, pp. 618–629.
DOI: 10.1007/978-3-540-27836-8_53 (cited on pp. 15, 20, 21, 24, 73, 74, 78, 83, 87, 89, 94, 106, 108, 111, 142).

[58] D. Giammarresi, J.-L. Ponty, and D. Wood. Thompson digraphs: a characterization. In O. Boldt and H. Jürgensen, editors, *Automata implementation*. Vol. 2214, in Lecture Notes in Computer Science, pp. 91–100. Springer Berlin Heidelberg, 2001.
DOI: 10.1007/3-540-45526-4_9 (cited on p. 26).

[59] V. Glushkov. The abstract theory of automata. *Russian mathematical surveys*, 16(5):1–53, 1961.
DOI: 10.1070/RM1961v016n05ABEH004112 (cited on pp. 10, 108).

[60] M. Gowda, G. Stewart, G. Mainland, B. Radunović, D. Vytiniotis, and D. Patterson. Ziria: language for rapid prototyping of wireless PHY. In *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 359–362.
DOI: 10.1145/2619239.2631427 (cited on p. 143).

[61] J. Goyvaerts and S. Levithan. *Regular expressions cookbook*. O'Reilly, 2009.
ISBN: 978-0-596-52068-7 (cited on p. 134).

[62] C. Grabmayer. Using proofs by coinduction to find "traditional" proofs. In J. L. Fiadeiro, N. Harman, M. Roggenbach, and J. Rutten, editors, *Algebra and coalgebra in computer science*. Vol. 3629, in Lecture Notes in Computer Science, pp. 175–193. Springer Berlin Heidelberg, 2005.
DOI: 10.1007/11548133_12 (cited on p. 51).

[63] N. B. B. Grathwohl, F. Henglein, and D. Kozen. Infinitary axiomatization of the equational theory of context-free languages. In *Proc. 9th workshop fixed points in computer science (FICS'13)*. D. Baelde and A. Carayol, editors. Vol. 126. In Electronic Proceedings in Theoretical Computer Science. Torino, Italy, Sept. 2013, pp. 44–55.
DOI: 10.4204/EPTCS.126.4 (cited on pp. 1, 2, 61, 147).

[64] N. B. B. Grathwohl, F. Henglein, L. Nielsen, and U. T. Rasmussen. Two-pass greedy regular expression parsing. In S. Konstantinidis, editor, *Proc. 18th international conference on implementation and application of automata (CIAA'13)*. Vol. 7982, in Lecture Notes in Computer Science, pp. 60–71. Springer Berlin Heidelberg, July 2013.
DOI: 10.1007/978-3-642-39274-0_7 (cited on pp. 1, 2, 7, 29, 30, 73, 74, 76, 89, 94, 106, 108, 142).

[65] N. B. B. Grathwohl, F. Henglein, and U. T. Rasmussen. Optimally streaming greedy regular expression parsing. In *Theoretical aspects of computing – ICTAC'14*. G. Ciobanu and D. Méry, editors. Vol. 8687. In Lecture Notes in Computer Science. Springer International Publishing, Sept. 2014, pp. 224–240.
DOI: 10.1007/978-3-319-10882-7_14 (cited on pp. 1, 2, 7, 37, 89, 90, 108, 142).

[66] N. B. B. Grathwohl, F. Henglein, U. T. Rasmussen, K. A. Søholm, and S. P. Tørholm. Kleenex: compiling nondeterministic transducers to deterministic streaming transducers. Submitted for publication to: 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16). 2016 (cited on pp. 1, 2, 7, 42, 107).

[67] N. B. B. Grathwohl, D. Kozen, and K. Mamouras. KAT + B! In *Proceedings of the joint meeting of the twenty-third eacsl annual conference on computer science logic (csl) and the twenty-ninth annual acm/ieee symposium on logic in computer science (lics)*. In CSL-LICS '14. ACM, Vienna, Austria, 2014, 44:1–44:10.
DOI: 10.1145/2603088.2603095 (cited on pp. 1, 2, 163).

[68] N. B. B. Grathwohl, U. T. Rasmussen, and F. Henglein. Kleene Meets Church: regular expressions as types. Poster presented at POPL 2015. Mumbai, India, 2015.
URL: http://diku.dk/kmc/documents/kmcposter.pdf (cited on p. 7).

[69] J. Gruska. A characterization of context-free languages. *J. comput. syst. sci.*, 5(4):353–364, Aug. 1971.
DOI: 10.1016/S0022-0000(71)80023-5 (cited on p. 147).

[70] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. 2003 ACM SIGMOD international conference on management of data*. In SIGMOD '03. ACM, San Diego, California, 2003, pp. 419–430.
DOI: 10.1145/872757.872809 (cited on p. 106).

[71] C. Hardin and D. Kozen. On the elimination of hypotheses in Kleene algebra with tests. Tech. rep. (TR2002-1879). Computer Science Department, Cornell University, Oct. 2002.
URL: http://hdl.handle.net/1813/5855 (cited on pp. 163, 173).

[72] D. Harel. On folk theorems. *Communications of the ACM*, 23(7):379–389, July 1980.
DOI: 10.1145/358886.358892 (cited on pp. 65, 163, 164).

[73] P. Hazel. PCRE – Perl-compatible regular expressions.
URL: http://www.pcre.org/pcre.txt (cited on pp. 8, 142).

[74] F. Henglein and L. Nielsen. Declarative coinductive axiomatization of regular expression containment and its computational interpretation (preliminary version). TOPPS D-Report (612). Department of Computer Science, University of Copenhagen (DIKU), Feb. 2010.
URL: http://www.diku.dk/hjemmesider/ansatte/henglein/papers/henglein2010a.pdf (cited on p. 74).

[75] F. Henglein and L. Nielsen. Regular expression containment: coinductive axiomatization and computational interpretation. In *Proc. 38th ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL'11)*. Vol. 46. In SIGPLAN Notices. ACM Press, Jan. 2011, pp. 385–398.
DOI: 10.1145/1926385.1926429 (cited on pp. 15, 19, 108).

[76] K. Hirose and M. Oya. General theory of flowcharts. In *Comment. math. univ. st. paul*, 1972 (cited on p. 163).

[77] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison–Wesley, 1979.
ISBN: 0-201-02988-X (cited on p. 58).

[78] M. Hopkins. The algebraic approach I: the algebraization of the Chomsky hierarchy. In *Proc. 10th int. conf. relational methods in computer science and 5th int. conf. applications of kleene algebra (relmics/aka 2008)*. R. Berghammer, B. Möller, and G. Struth, editors. Vol. 4988. In Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg, Apr. 2008, pp. 155–172.
DOI: 10.1007/978-3-540-78913-0_13 (cited on p. 147).

[79] M. Hopkins. The algebraic approach II: dioids, quantales and monads. In *Proc. 10th int. conf. relational methods in computer science and 5th int. conf. applications of kleene algebra (relmics/aka 2008)*. R. Berghammer, B. Möller, and G. Struth, editors. Vol. 4988. In Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg, Apr. 2008, pp. 173–190.
DOI: 10.1007/978-3-540-78913-0_14 (cited on p. 147).

[80] IEEE Computer Society. *Standard for information technology - portable operating system interface (posix), base specifications, issue 7*. IEEE Std 1003.1. IEEE, 2008.
DOI: 10.1109/IEEESTD.2008.4694976 (cited on p. 94).

[81] L. Ilie and S. Yu. Follow automata. *Information and computation*, 186(1):140–162, 2003.
DOI: 10.1016/S0890-5401(03)00090-7 (cited on p. 108).

[82] D. M. Kaplan. Regular expressions and the equivalence of programs. *Journal of computer and system sciences*, 3(4):361–386, 1969.
DOI: 10.1016/S0022-0000(69)80027-9 (cited on pp. 63, 64).

[83] S. Kearns. Extending regular expressions with context operators and parse extraction. *Software - practice and experience*, 21(8):787–804, 1991.
DOI: 10.1002/spe.4380210803 (cited on pp. 89, 142).

[84] S. M. Kearns. Extending regular expressions. PhD thesis. Columbia University, 1990 (cited on pp. 73, 87).

[85] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata studies*:3–42, 1956.
URL: http://www.dlsi.ua.es/~mlf/nnafmc/papers/kleene56representation.pdf (cited on p. 7).

[86] K. Kosako. The Oniguruma regular expression library. 2014.
URL: http://www.geocities.jp/kosako3/oniguruma/ (cited on p. 127).

[87] S. R. Kosaraju. Analysis of structured programs. In *Proceedings of the fifth annual acm symposium on theory of computing*. In STOC '73. ACM, Austin, Texas, USA, 1973, pp. 240–252.
DOI: 10.1145/800125.804055 (cited on pp. 163, 176).

[88] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and computation*, 110(2):366–390, May 1994.
DOI: 10.1006/inco.1994.1037 (cited on pp. 51, 52, 54, 56, 70, 165).

[89] D. Kozen. *Automata and computability*. Springer Verlag, 1997.
ISBN: 978-1-4612-7309-7 (cited on pp. 58, 59, 62, 161).

[90] D. Kozen. Automata on guarded strings and applications. *Matématica contemporânea*, 24:117–139, 2003.
URL: http://hdl.handle.net/1813/5821 (cited on pp. 174, 176).

[91] D. Kozen. Course notes: introduction to kleene algebra. 2004.
URL: http://www.cs.cornell.edu/Courses/cs786/2004sp/ (cited on pp. 51, 57).

[92] D. Kozen. Halting and equivalence of schemes over recursive theories. Tech. rep. (TR2002-1881). Computer Science Department, Cornell University, Oct. 2002.
URL: http://hdl.handle.net/1813/5857 (cited on p. 164).

[93] D. Kozen. Kleene algebra with tests. *ACM transactions on programming languages and systems*, 19(3):427–443, May 1997.
DOI: 10.1145/256167.256195 (cited on pp. 63, 65–67, 163, 164).

[94] D. Kozen. Kleene algebra with tests and commutativity conditions. In T. Margaria and B. Steffen, editors, *Tools and algorithms for the construction and analysis of systems (TACAS'96)*. Vol. 1055, in Lecture Notes in Computer Science, pp. 14–33. Springer Berlin Heidelberg, 1996.
DOI: 10.1007/3-540-61042-1_35 (cited on p. 63).

[95] D. Kozen. NetKAT: a formal system for the verification of networks. In *Proc. 12th asian symposium on programming languages and systems (APLAS'14)*. J. Garrigue, editor. Vol. 8858. In Lecture Notes in Computer Science. Asian Association for Foundation of Software (AAFS). Springer, Singapore, Nov. 2014.
DOI: 10.1007/978-3-319-12736-1_1 (cited on pp. 70, 163).

[96] D. Kozen. On induction vs. *-continuity. In *Proc. logics of programs*. Vol. 131. In Lecture Notes in Computer Science (LNCS). Springer, May 1981, pp. 167–176.
DOI: 10.1007/BFb0025769 (cited on p. 156).

[97] D. Kozen. Results on the propositional μ-calculus. *Theoretical computer science*, 27(3):333–354, 1983.
DOI: 10.1016/0304-3975(82)90125-6 (cited on p. 157).

[98] D. Kozen. Some results in dynamic model theory. *Science of computer programming*, 51(1–2):3–22, 2004. Mathematics of Program Construction (MPC 2002).
DOI: http://dx.doi.org/10.1016/j.scico.2003.09.004 (cited on p. 164).

[99] D. Kozen. *The design and analysis of algorithms*. Springer-Verlag, New York, 1991.
DOI: 10.1007/978-1-4612-4400-4 (cited on pp. 153, 156, 157).

[100] D. C. Kozen. *Theory of computation*. Springer Publishing Company, Incorporated, 1st ed., 2010.
ISBN: 1849965714 (cited on pp. 172, 176).

[101] D. Kozen and M.-C. Patron. Certification of compiler optimizations using kleene algebra with tests. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Pereira, Y. Sagiv, and P. Stuckey, editors, *Computational logic — cl 2000*. Vol. 1861, in Lecture Notes in Computer Science, pp. 568–582. Springer Berlin Heidelberg, 2000.
DOI: 10.1007/3-540-44957-4_38 (cited on p. 163).

[102] D. Kozen and F. Smith. Kleene algebra with tests: completeness and decidability. In D. v. Dalen and M. Bezem, editors, *Computer science logic*. Vol. 1258, in Lecture Notes in Computer Science, pp. 244–259. Springer Berlin Heidelberg, 1997.
DOI: 10.1007/3-540-63172-0_43 (cited on pp. 64, 65, 163, 165, 173).

[103] D. Kozen and W.-L. Tseng. The Böhm–Jacopini theorem is false, propositionally. In P. Audebaud and C. Paulin-Mohring, editors, *Mathematics of program construction*. Vol. 5133, in Lecture Notes in Computer Science, pp. 177–192. Springer Berlin Heidelberg, 2008.
DOI: 10.1007/978-3-540-70594-9_11 (cited on pp. 163, 176, 177).

[104] H. Leiß. Towards Kleene algebra with recursion. In *CSL '91: proceedings of the 5th workshop on computer science logic*. Springer-Verlag, London, UK, 1992, pp. 242–256.
DOI: 10.1007/BFb0023771 (cited on pp. 60, 147, 148, 151, 153–155, 160).

[105] M. Lutz. *Programming python*. Vol. 8. O'Reilly, 4th edition ed., Dec. 2010.
ISBN: 978-0-596-15810-1 (cited on p. 127).

[106] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE trans. on electronic comput.*, EC-9(1):38–47, 1960.
DOI: 10.1109/TEC.1960.5221603 (cited on p. 10).

[107] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: a functional pearl. In *Proc. of the 16th ACM SIGPLAN international conference on functional programming (ICFP'11)*. ACM, 2011, pp. 189–195.
DOI: 10.1145/2034773.2034801 (cited on p. 87).

[108] G. Mirkowska. Algorithmic logic and its applications. PhD thesis. University of Warsaw, 1972 (cited on p. 163).

[109] M. Mohri. Finite-state transducers in language and speech processing. *Computational linguistics*, 23(2):269–311, 1997.
URL: http://dl.acm.org/citation.cfm?id=972695.972698 (cited on p. 143).

[110] E. W. Myers, P. Oliva, and K. Guimarães. Reporting exact and approximate regular expression matches. In M. Farach-Colton, editor, *Combinatorial pattern matching*. Vol. 1448, in Lecture Notes in Computer Science, pp. 91–103. Springer Berlin Heidelberg, 1998.
DOI: 10.1007/BFb0030783 (cited on p. 142).

[111] G. Myers. A four Russians algorithm for regular expression pattern matching. *J. acm*, 39(2):432–448, 1992.
DOI: http://doi.acm.org/10.1145/128749.128755 (cited on p. 142).

[112] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th international conference on architectural support for programming languages and operating systems (ASPLOS'14)*. ACM, 2014, pp. 529–542.
DOI: 10.1145/2541940.2541988 (cited on pp. 48, 143).

[113] G. Navarro and M. Raffinot. Compact DFA representation for fast regular expression search. In G. S. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Algorithm engineering*. Vol. 2141, in Lecture Notes in Computer Science, pp. 1–13. Springer Berlin Heidelberg, 2001.
DOI: 10.1007/3-540-44688-5_1 (cited on p. 143).

[114] L. Nielsen and F. Henglein. Bit-coded regular expression parsing. In *Proc. 5th int'l conf. on language and automata theory and applications (LATA'11)*. Vol. 6638. In Lecture Notes in Computer Science (LNCS). Springer, May 2011, pp. 402–413.
DOI: 10.1007/978-3-642-21254-3_32 (cited on pp. 15, 73, 75, 76, 83, 87, 89, 106, 108).

[115] G. v. Noord and D. Gerdemann. Finite State Transducers with Predicates and Identities. *Grammars*, 4(3):263–286, 2001.
DOI: 10.1023/A:1012291501330 (cited on pp. 116, 126).

[116] S. Okui and T. Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. In M. Domaratzki and K. Salomaa, editors, *Implementation and application of automata*. Vol. 6482, in Lecture Notes in Computer Science, pp. 231–240. Springer Berlin Heidelberg, 2011.
DOI: 10.1007/978-3-642-18098-9_25 (cited on p. 94).

[117] S. Okui and T. Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. Tech. rep. (2013-002). The University of Aizu, June 2013 (cited on p. 142).

[118] G. Oulsnam. Unraveling unstructured programs. *The computer journal*, 25(3):379–387, 1982.
DOI: 10.1093/comjnl/25.3.379 (cited on pp. 163, 176).

[119] J. Ousterhout. Tcl: An Embeddable Command Language. In *Proc. USENIX winter conference*, Jan. 1990, pp. 133–146 (cited on p. 83).

[120] D. M. R. Park. Fixpoint induction and proofs of program properties. In *Machine intelligence*. D. Michie and B. Meltzer, editors. Vol. 5. Edinburgh University Press, 1969, pp. 59–78 (cited on p. 151).

[121] W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat, and exit statements. *Comm. assoc. comput. mach.*, 16(8):503–512, 1973.
DOI: 10.1145/355609.362337 (cited on pp. 163, 176).

[122] V. Pratt. Dynamic algebras as a well-behaved fragment of relation algebras. In C. H. Bergman, R. D. Maddux, and D. L. Pigozzi, editors, *Algebraic logic and universal algebra in computer science*. Vol. 425, in Lecture Notes in Computer Science, pp. 77–110. Springer New York, 1990.
DOI: `10.1007/BFb0043079` (cited on p. 52).

[123] L. Ramshaw. Eliminating goto's while preserving program structure. *Journal of the ACM*, 35(4):893–920, 1988.
DOI: `10.1145/48014.48021` (cited on pp. 163, 176).

[124] A. Rathnayake and H. Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *Corr*, abs/1405.7058, 2014.
URL: `http://arxiv.org/abs/1405.7058` (cited on p. 142).

[125] Reduction of 3-CNF-SAT to Perl regular expression matching.
URL: `http://perl.plover.com/NPC/NPC-3SAT.html` (cited on p. 8).

[126] T. Reps. "Maximal-munch"tokenization in linear time. *ACM trans. program. lang. syst.*, 20(2):259–273, 1998.
DOI: `http://doi.acm.org/10.1145/276393.276394` (cited on p. 87).

[127] A. Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, Jan. 1966.
DOI: `10.1145/321312.321326` (cited on p. 51).

[128] G. Schnitger. Regular expressions and NFAs without ε-transitions. In B. Durand and W. Thomas, editors, *STACS 2006*. Vol. 3884, in Lecture Notes in Computer Science, pp. 432–443. Springer, 2006.
DOI: `10.1007/11672142_35` (cited on p. 108).

[129] M. Schützenberger. Sur une variante des fonctions sequentielles. *Theoretical computer science*, 4(1):47–57, Feb. 1977.
DOI: `10.1016/0304-3975(77)90055-X` (cited on pp. 116, 118).

[130] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *Proc. 9th annual IEEE symposium on field-programmable custom computing machines (FCCM'01)*, 2001, pp. 227–238 (cited on p. 143).

[131] K. A. Søholm and S. P. Tørholm. Ordered finite action transducers for high-performance stream processing. Master's Thesis. University of Copenhagen, 2015 (cited on pp. 42, 115).

[132] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Proceedings of the fifth annual acm symposium on theory of computing*. In STOC '73. ACM, Austin, Texas, USA, 1973, pp. 1–9.
DOI: `10.1145/800125.804029` (cited on pp. 38, 56, 98, 176).

[133] S. Sugiyama and Y. Minamide. Checking time linearity of regular expression matching based on backtracking. In *IPSJ transactions on programming*. (3) in 7, 2014, pp. 1–11.
DOI: `10.2197/ipsjtrans.7.82` (cited on p. 142).

[134] M. Sulzmann and K. Z. M. Lu. POSIX regular expression parsing with derivatives. In *Proc. 12th international symposium on functional and logic programming*. In FLOPS '14. Kanazawa, Japan, June 2014.
DOI: `10.1007/978-3-319-07151-0_13` (cited on pp. 94, 106, 142).

[135] M. Sulzmann and K. Z. M. Lu. Regular expression sub-matching using partial derivatives. In *Proc. 14th symposium on principles and practice of declarative programming*. In PPDP '12. ACM, Leuven, Belgium, 2012, pp. 79–90.
DOI: 10.1145/2370776.2370788 (cited on pp. 89, 94, 142).

[136] The GNU Project. 2015.
URL: http://www.gnu.org/software/coreutils/coreutils.html (cited on pp. 83, 127).

[137] The GNU project. GCC, the GNU conpiler collection.
URL: http://gcc.gnu.org/ (cited on p. 42).

[138] The LLVM project. Clang: a C language frontend for LLVM.
URL: http://clang.llvm.org/ (cited on p. 42).

[139] The RE2 authors. RE2.
URL: https://code.google.com/p/re2/ (cited on pp. 48, 83, 109, 127, 142).

[140] The RE2J authors. RE2J. 2015.
URL: https://github.com/google/re2j (cited on p. 127).

[141] K. Thompson. Programming techniques: regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.
DOI: 10.1145/363347.363387 (cited on pp. 7, 10, 11, 91, 108).

[142] A. Thurston. Ragel state machine compiler. 2015.
URL: http://www.colm.net/open-source/ragel/ (cited on pp. 49, 109, 127).

[143] S. Vansummeren. Type inference for unique pattern matching. *Acm trans. program. lang. syst.*, 28(3):389–428, 2006.
DOI: http://doi.acm.org/10.1145/1133651.1133652 (cited on p. 143).

[144] M. Veanes. Symbolic String Transformations with Regular Lookahead and Rollback. In *Ershov informatics conference (PSI'14)*. Vol. 8974. In Lecture Notes in Computer Science. Springer Verlag, 2014.
DOI: 10.1007/978-3-662-46823-4_27 (cited on p. 126).

[145] M. Veanes, P. d. Halleux, and N. Tillmann. Rex: symbolic regular expression explorer. In *Proc. 3rd international conference on software testing, verification and validation (icst'10)*. IEEE Computer Society Press, Paris, France, Apr. 2010, pp. 498–507.
DOI: 10.1109/ICST.2010.15 (cited on p. 85).

[146] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: algorithms and applications. In *Proceedings of the 39th annual symposium on principles of programming languages*. In POPL '12. Philadelphia, PA, USA, 2012, pp. 137–150.
DOI: 10.1145/2103656.2103674 (cited on pp. 126, 144).

[147] M. Veanes, D. Molnar, T. Mytkowicz, and B. Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL'15)*. ACM, 2015.
DOI: 10.1145/2676726.2677014 (cited on pp. 48, 126, 144).

[148] B. W. Watson. Implementing and using finite automata toolkits. *Natural language engineering*, 2(04):295–302, 1996.
DOI: 10.1017/S135132499700154X (cited on p. 126).

[149] B. B. Welch, K. Jones, and J. Hobbs. *Practical programming in tcl and tk*. Prentice Hall, 4th edition ed., 2003.
ISBN: 0130385603 (cited on p. 127).

[150] M. Williams and H. Ossher. Conversion of unstructured flow diagrams into structured form. *The computer journal*, 21(2):161–167, 1978.
DOI: 10.1093/comjnl/21.2.161 (cited on pp. 163, 176).

[151] G. Winskel. *The formal semantics of programming languages*. MIT Press, 1993.
ISBN: 9780262231695 (cited on p. 151).

[152] X. Wu and D. Theodoratos. A survey on XML streaming evaluation techniques. *The VLDB journal*, 22(2):177–202, Apr. 2013.
DOI: 10.1007/s00778-012-0281-y (cited on p. 106).

[153] L. Yang, P. Manadhata, W. Horne, P. Rao, and V. Ganapathy. Fast submatch extraction using obdds. In *Proceedings of the eighth acm/ieee symposium on architectures for networking and communications systems*. In ANCS '12. ACM, Austin, Texas, USA, 2012, pp. 163–174.
DOI: 10.1145/2396556.2396594 (cited on p. 143).