RICE UNIVERSITY

Formally Verified Algorithms for Temporal Logic and Regular Expressions

By

Agnishom Chattopadhyay

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

Konstantinos Mamouras
Konstantinos Mamouras (Aug 5, 2024 09:16 CDT)

Konstantinos Mamouras (Chair)

Assistant Professor, Computer Science

Robert Cartwright
Robert Cartwright (Aug 8, 2024 09:54 CDT)

Robert Cartwright

Professor, Computer Science

[illegible]
Kaiyuan Yang (Aug 6, 2024 22:04 CDT)

Kaiyuan Yang

Associate Professor, Electrical and Computer Engineering

HOUSTON, TEXAS

August 2024

ABSTRACT

Formally Verified Algorithms for Temporal Logic and Regular Expressions

by

Agnishom Chattopadhyay

The behavior of systems in various domains including IoT networks, cyber-physical systems and runtime environments of programs can be observed in the form of linear traces. *Temporal logic* and *regular expressions* are two core formalisms used to specify properties of such data. This thesis extends these formalisms to enable the expression of richer classes of properties in a succinct manner together with algorithms that can handle them efficiently. Using the Coq proof assistant, we formalize the semantics of our specification languages and verify the correctness of our algorithms using mechanically checked proofs. The verified algorithms have been extracted to executable code, and our emperical evaluation shows that they are competitive with state-of-the-art tools.

The first part of the thesis is focused on investigating the formalization of an online monitoring framework for past-time metric temporal logic (MTL). We employ an algebraic quantitative semantics that encompasses the Boolean and robustness semantics of MTL and we interpret formulas over a discrete temporal domain. A potentially infinite-state variant of Mealy machines, a kind of string transducers, is used as a formal model of online monitors. We demonstrate a compositional construction from formulas to monitors, such that each monitor computes (in an online fashion) the semantic values of the corresponding formula over the input stream. The time taken by the monitor to process each input item is proportional to $O(|\varphi|)$ where $|\varphi|$ is the size of the formula, and is independent of the constants that appear in the formula. The monitor uses $O(m)$ space where m is the sum of the numerical constants that appear in the formula.

The latter part of the thesis is focused on regular expressions. Regular expressions in practice often contain lookaround assertions, which can be used to refine matches based on the surrounding context. Our formal semantics of lookarounds complements the commonly used operational understanding of lookaround in terms of a backtracking implementation. Widely used regular expression matching engines take exponential time to match regular expressions with lookarounds in the worst case. Our algorithm has a worst-case time complexity of $O(m \cdot n)$, where m is the size of the regex and n is the size of the input string. The key insight is to evaluate the lookarounds in a bottom-up manner, and guard automaton transitions with oracle queries evaluating the lookarounds. We demonstrate how this algorithm can be implemented in a purely functional manner using marked regular expressions. The formal semantics of lookarounds and our matching algorithm is verified in Coq.

Finally, we investigate the formalization of a tokenization algorithm. Tokenization is the process of breaking a monolithic string into a stream of tokens. This is one of the very first steps in the compilation of programs. In this setting, the set of possible tokens is often described using an ordered list of regular expressions. Our algorithm is based on the simulation of the Thompson NFA of the given regular expressions. Two significant parts of the verification effort involve demonstrating the correctness of Thompson's algorithm and the computation of ε-closures using depth-first search. For a stream of length n and a list of regular expressions of total size m, our algorithm finds the first token in $O(m \cdot n)$ time, and tokenizes the entire stream in $O(m \cdot n^2)$ time in the worst-case.

Acknowledgments

The work presented in this thesis would not have been possible without the guidance of my advisor, Dr. Konstantinos Mamouras. Throughout the duration of my graduate studies, Dr. Mamouras has been very generous with his time and patience, and has played a major role in shaping the work presented in the next few chapters. It has been a privilege to work with a researcher with both very wide and very deep knowlege of computer science, ranging from obscure results in algorithms and formal languages to the low level tricks that make blazingly fast implementations possible.

I want to thank the two other members of my thesis committee, Dr. Robert Cartwright and Dr. Kaiyuan Yang, for taking the time to evaluate my work and attend my presentations. I greatly enjoyed working on the illuminating assignments in Dr. Cartwright's Priciples of Programming Language class, and later assisting him in teaching his class on functional programming. With Dr. Yang's team, I had the chance to work on a project that was at the intersection of hardware design and automata theory.

I am grateful to be able to collaborate with my labmates and friends Zhifu Wang, Lingkun Kong, Alexis Le Glaunec and Angela. Many of the projects presented in this thesis have benefitted from their technical expertise or the insightful discussions I had with them. Angela has helped me with the experimental evaluation of algorithms on a number of occassions. It is always comforting to have company, especially when research is stressful.

I acknowledge the support of innumerable other people who have helped with my research in direct or indirect ways. These include (but are not limited to) the researchers who have made their tool and code available online, the authors of the many helpful tutorials and blog posts, and the users on certain mailing lists and Stack Exchange forums who have answered my questions. I also want to thank our department and Dr. Tracy Volz for organizing weekly seminars where I had the chance to learn about the research conducted by my peers.

I want to thank my friend Sohail Dasgupta for being a great roommate. His nihilistic sense of humor has always been a great source of comfort, and has taught me to not take life too seriously. I want to thank my parents Rochana and Himadri Chattopadhyay for their unwavering support and care for my health and well being throughout. I want to thank my brother Amartyanil Chattopadhyay for his company and the great conversations we have shared.

Contents

Illustrations

Tables

Chapter 1

Introduction

Complex automated systems can be seen in various domains including IoT networks, cyber-physical systems, and even runtime environments of programming languages. Designers of these systems need to have confidence in their correctness. In practice, engineers can be more confident about their software via simulations and systematic testing. Our research is based on the premise that rigorous and well-developed mathematical frameworks are key to the development of sound toolchains.

In a more rigorous setting, software artifacts (including source code and binaries) must be coupled together with formal specifications that describe their intended behavior. When choosing a formal framework for specification, one has to take into account a large number of factors. Generally, they involve (1) the abstractions necessary to describe the intricacies of the system and (2) the availability of capable tools and algorithms that can manipulate objects in this framework. Thus, it is helpful to judge such a framework in light of these two factors.

If the system is reasonably sized, an approach to verification would be *model checking*. Usually, this refers to modeling the system as a transition system and then checking whether any paths in the transition system form a trace contradicting the specification using automata-theoretic techniques. (See, for example, the papers [2], [3] or [4]). Another approach is to verify the algorithms using proof assistants such as Coq, Lean, or HOL/Isabelle. Within a proof assistant, one can express correctness properties in a very expressive logic, state the algorithms as mathematical functions, and then show that the functions satisfy the properties. While the formulas and functions expressible in these systems are vastly more expressive, the verification process is largely manual. A discussion about the various proof assistants can be found in the survey by Ringer et al. [5].

Static verification techniques like these, however, may not always be feasible. These techniques do not necessarily scale well with the complexity of the system. Additionally, they may depend on assumptions that are made in the process of modeling the system. With theorem proving, additional layers of complexity are brought into the development process as the proofs need to be developed and maintained in tandem. This means that monitors, which can report events to human supervisors, need to be deployed.

The complexity of the landscape of scenarios to which monitors need to be adapted is enormous. In some cases, it is acceptable to audit traces from the system once it has completed execution, but often it may be necessary to monitor the behavior of the system in real time. In such cases, the monitors may only have extremely limited resources (e.g., when executing on a microcontroller) and may need to have a high throughput. In addition, monitors may need to respond to signals that involve uncertainty, and different events may require different responses. In many applications, the challenge of monitoring a system may arise from the heterogeneity of its components, or from being distributed over various locations.

Because of this complexity, the mainstream approach is to program monitors manually on a case-by-case basis for each application. However, this approach can also be prone to programming errors and scales poorly with respect to the complexity of the system. This is why we focus on formalisms that are grounded in mathematically sound theory and enable monitor construction across a wide range of applications.

Two formalisms that are used in the building blocks of such tools are temporal logic and regular expressions. Researchers have studied temporal logic-based specifications for monitoring in a number of applications, including Automatic Transmission Control Systems [6], Artificial Pancreas Control Systems [7], and even Engineering Education Systems [8]. Besides being used widely for text processing applications, regular expressions have also been used to inspect network packets for intrusion detection [9, 10], to detect malware signatures [11] and even to discover motifs in biological sequences [12].

1.1 Temporal Logic

Linear Temporal Logic (LTL) was introduced by Pnueli [13] as a formalism for specifying properties of reactive systems. Using temporal logic, one can specify properties of a *trace*, which is a summary of the execution (or behavior) of the system over time. Temporal logic enables the specification of obligations (or prerequisites) in the future (or past) depending on the current state of the system via temporal modalities.

Various extensions of this logic have received attention in the formal methods community in order to express properties of cyber-physical systems (see [14] for a survey). The metric extension of temporal logic (MTL) [15] allows the annotation of temporal modalities with time intervals within which the witnessing event must occur. Signal Temporal Logic (STL) [16] allows forming temporal formulas over real-valued signals.

Generally, temporal logic is stated in terms of future-oriented temporal modalities and interpreted over infinite traces. This is because a large part of the literature is focused on model-checking (i.e, static verification) of reactive systems to determine if any execution trace may have a violation in the indefinite future. However, in the context of runtime verification, we are interested either in monitoring existing traces (called *offline monitoring*) collected from executions of the system in the past, or monitoring the system in real time while it is executing (called *online monitoring*). Thus, in this context, we are only able to view a finite prefix of the trace at any given time. This thesis focuses on a past-time-only fragment of temporal logic, but our later work has considered extensions to finite-horizon future time modalities in [17, 18].

1.2 Regular Expressions

Since their introduction in the 1950s, regular expressions [19] and finite-state automata [20] have found applications in numerous domains to describe patterns over sequences. They have been used for the lexical analysis of programs [21] during compilation, the search of words and patterns in text editors [22], and bibliographic search [23]. Regular patterns are also used in network security [10] to search for intrusion signatures in network traffic, in bioinformatics [12] for describing protein, RNA, or DNA sequences, and in runtime verification [14] for specifying safety properties.

Classical regular expressions involve constructs for nondeterministic choice $r_1 + r_2$, concatenation $r_1 \cdot r_2$, and Kleene's star r^* (repetition of r zero or more times). In practice, the syntax of regular expressions is often extended with more constructs that offer convenience, such as character classes for describing sets of letters/symbols (e.g., $[ab]$ and $[0\text{–}9]$), the construct $r?$ for indicating that the pattern r is optional, and Kleene's plus r^+ (repetition of r at least once). The construct of bounded repetition, which is written as $r\{m,n\}$, describes the repetition of r from m to n times, can be translated using concatenation and ? but makes regular expressions exponentially more succinct. Certain extensions of regular expressions used in practice also extend their expressivity beyond regular patterns. For instance, the use of backreferences can ensure that the exact substring is matched in multiple occurrences. As an example, the expression $r_1\backslash 1$ matches a string of the form $w \cdot w$ where w matches r_1. Bounded repetition of the form $r\{m, n\}$ is a common occurrence in practice, which succinctly encodes repeated concatenation. Our work in [24] discusses the notion of *counter-unambiguity*, which can be viewed as the criteria to determine if simulating such a repetition using a numerical counter would be sound. In [25], the authors have examined certain other cases in which bounded repetition can be matched efficiently (in linear time-per-character) using a model of finite-state automata augmented with bit vectors. These bit vector automata have also inspired the BVAP architecture [26] for efficient hardware-based regex matching.

The focus of this thesis is regular expressions with lookaround assertions, which goes beyond classical regular expressions by allowing one to describe not only a pattern to search for, but also the context in which the pattern should appear. The lookaround assertions include *lookaheads* and *lookbehinds*. A lookahead asserts that the text that lies ahead (i.e., in the "future" relative to the current position) matches a given pattern, while a lookbehind asserts that the text that lies behind (i.e., in the "past") matches a given pattern. We also discuss the application of regular expressions to *tokenization*. Tokenization is the process of splitting a string into a stream of *tokens*. This is often an important step in the compilation of programs and in natural language processing. It is common in such applications to describe legal tokens using regular expressions.

1.3 Contributions

This thesis investigates the formalization of several efficient algorithms for temporal logic and regular expressions. The algorithms described in the subsequent chapters have been formally verified in Coq, and empirical evaluation suggests their performance is competitive. The main contributions of this thesis include:

1. A verified monitoring algorithm for temporal logic (Chapter 2, extending work published in [27])

2. An efficient algorithm for matching regular expressions with lookarounds (Chapter 3, extending work published in [28])

3. A verified tokenization procedure based on Thompson's Algorithm (Chapter 4)

1.3.1 Online Monitoring for Metric Temporal Logic

The contributions of Chapter 2 are as follows:

- We present a compositional construction of online monitors for specifications described in past-time metric temporal logic (MTL), interpreted over a discrete temporal domain. Our approach involves expressing the given formula using a family of logical and temporal connectives, and then using an inductive construction to replace each connective with a combinator.

- We employ an algebraic quantitative semantics using bounded distributive lattices that encompasses the Boolean and real-valued (robustness) [29] semantics of MTL.

- Our algorithm is verified correct in Coq, where we model online monitors using (infinite-state) Mealy machines, a form of stateful transducer which produces an output for each input. Our formalization also includes the lattice-based semantics of MTL, along with identities that are essential for our compositional construction.

- We empirically evaluate the performance of our algorithm and compare it with Reelay [30], a state-of-the-art monitoring tool.
- The time taken by the monitor to process each input item is proportional to $O(|\varphi|)$ where $|\varphi|$ is the number of connectives and atomic propositions used in the formula, and is independent of the constants that appear in the formula. The monitor uses $O(m)$ space where m is the sum of the numerical constants that appear in the formula.

1.3.2 Regular Expressions with Lookarounds

The contributions of Chapter 3 are as follows:

- We present and formalize a context-dependent semantics for regular expressions with lookarounds. This semantics complements existing definitions of the lookaround constructs that are operational, i.e., defined in terms of a backtracking matching algorithm.
- Using our formal semantics for lookaround, we prove that regular expressions with lookaround satisfy the equivalence properties of Kleene algebra [31]. Moreover, we establish a number of equivalences involving lookaround that can be used for simplifying patterns.
- We propose a novel efficient algorithm for matching lookaround expressions in $O(m \cdot n)$ time, where m is the size of the regex and n is the length of the input

string. Existing regex matching engines that support lookaround assertions are based on backtracking, and therefore take exponential time in the worst case.

- Our algorithm is purely functional. The key technical abstraction used in the algorithm is the notion of regular expressions with oracle queries. We extend the technique of marked regular expressions [32, 33] to match these expressions.
- We conduct experiments to evaluate the performance of our implementation against PCRE [34], and `java.util.regex` of the Java standard library [35] and also another verified tool [1].

1.3.3 Tokenization using Thompson's Algorithm

The contributions of Chapter 4 are as follows:

- We verify Thompson's construction for regular expressions in Coq. The construction produces an NFA represented as an array. The NFA has $O(m)$ states, where m is the size of the regular expression, and each state has at most 2 outgoing edges.
- We develop in Coq the machinery to reason about graph reachability and verify depth-first search using it. Given a graph with n vertices and m edges, our depth-first search algorithm has a time complexity of $O(n + m)$.
- We propose an approach towards using mutable arrays in OCaml code extracted from Coq, which may be helpful when arrays are manipulated in a 'linear' manner. This approach is helpful in maintaining the linear time complexity of depth-first search.

- We use the techniques above to implement a *maximal munch* lexer. Given a list of regular expressions the sum of whose size is m, and a string of length n, our lexer finds the first token in $O(m \cdot n)$ time, and tokenizes the entire string in $O(m \cdot n^2)$ time in the worst case.

- We observe that our lexer is twice as fast as Coqlex for a realistic JSON tokenization benchmark. Coqlex is exponentially slower on certain adversarially chosen inputs.

1.3.4 Other Related Contributions

In addition to the above contributions, the author has also been a part of the following works, which are closely related but not included in this thesis:

- In [36], the monitoring algorithm presented in Chapter 2 is extended to the algebraic setting of semirings. In [17], a monitoring algorithm for the continuous time setting is presented.

- In [24], the matching problem for the extension of regular expressions with bounded repetition (e.g., $r\{m, n\}$) is discussed. The key challenge here is in understanding when multiple non-deterministic runs of an NFA equipped with counters can be simulated together using a single numeric counter.

- Given a string and a regular expression, there may be multiple substrings that match the regular expression. Two popular disambiguation policies for selecting the substring are the longest-leftmost match (POSIX) and the greedy match (PCRE) policies. In [37], we discuss when the two disambiguation policies agree.

Chapter 2

Online Monitoring for Metric Temporal Logic

2.1 Introduction

Verifying cyber-physical systems statically is usually infeasible at large scales, and may require making assumptions about the behavior of the environment. In contrast, runtime verification is a lightweight technique for checking that a system exhibits the desired behavior. It is often performed in an *online* fashion, which means that the execution trace of the system is observed as it is being generated. This trace typically consists of one or more signals and event streams. A *monitor* program runs in parallel with the system, consumes the system trace incrementally, and outputs at every step a value that summarizes the current state of the system. This value can be a Boolean indication of whether an interesting event or pattern has been identified, or it can contain richer quantitative information. There is a substantial amount of existing work on formalisms for specifying monitors, as well as on algorithms for their efficient execution.

Temporal patterns are often specified using logical formalisms. Linear Temporal Logic (LTL) is one such widely utilized formalism which admits efficient algorithms. It is common to constrain the occurrence of temporal patterns using time intervals, which is a feature that gives rise to an extension of LTL called metric temporal logic (MTL) [15]. Since many applications in the domain of cyber-physical systems frequently deal with comparison between numerical signals, Signal Temporal Logic

(STL) [38], an extension of LTL with predicates allowing comparison with numerical values, is widely used.

While temporal logic facilitates the specification of temporal properties, it is equally important to have accompanying algorithms. The notion of a monitor is an algorithm that analyzes given traces for a specific temporal property. In an offline setting, the trace is available in its entirety. In contrast, online monitors are meant to be attached to running systems, so that they may report interesting (or critical) events as they happen, potentially so that a supervisor can act in real time. Thus, they must analyze system traces incrementally (fragment by fragment) as they evolve, and this must be done efficiently: each update should be handled quickly.

The standard semantics for temporal logic is qualitative, which means that monitors classify traces only in a binary pass/fail manner. However, this is not sufficiently informative for certain applications: some violations can be more serious than others, and on the other hand, some cases of satisfaction could be close to the edge of failure. In some cases, we may be able to take corrective action if we could tell that the system is approaching a potential violation. Indeed, in realistic systems with continuous dynamics, some degree of tolerance must be allowed since every value is accurate only up to the extent of measurement errors. This encourages us to consider quantitative semantics for our formalism, so that we can quantify how robustly the observed behavior fits the desired specification [29].

The variant of MTL that we consider in this chapter is interpreted over a discrete temporal domain and is a past-time-only fragment of the logic. In the setting of online monitoring we need to reactively respond to the patterns in what we have seen so far. So, using a past-time fragment makes sense and provides a clean semantics. Online monitoring with future-time temporal connectives has been considered, but these can give rise to semantic complications [39].

Using the interactive theorem prover Coq [40], we formalize the semantics of our temporal logic. The implementation of our monitoring algorithms are done within Coq, and a proof of correctness is given. Formal proofs, like the ones described in Coq, are thoroughly rigorous and machine-checkable. This gives us confidence in the correctness of our implementations. With the extraction mechanism of Coq, we can obtain executable OCaml code directly from our verified implementation.

It would be difficult to deploy an OCaml-based implementation on an embedded device in a cyber-physical system due to scarcity of runtime resources. However, our verified monitor could be used as a part of the development-level environment for such systems. An example of such an activity is the use of runtime monitoring for the purpose of falsification (see [41]). Our verified online monitor could also be used in any scenario where offline monitoring can be used. Another utility of our monitor is that it could be used as an oracle for differentially testing the correctness of other monitors.

As mentioned earlier, a strong motivation for using a quantitative semantics is to quantify how robustly a signal satisfies a given specification in view of potential perturbations. One way to do so for STL specifications is to interpret formulas over real numbers and interpret the logical connectives $\vee$ and $\wedge$ as max and min respectively [42]. In our work, we use a slightly more general framework, interpreting our formulas over arbitrary bounded distributive lattices. This abstract algebraic framework enables a simpler verification approach and, as we will discuss later, does not hurt the performance of our algorithms.

In our formalization, we model online monitors as a potentially infinite-state variant of Mealy machines. They are abstract machines whose state evolves as fragments of a trace are consumed. Each state of the machine is associated with a value that represents the current output of the monitor. We follow a compositional approach for our implementation and proofs. This is done with the help of combinators, which are constructs that compose Mealy machines in different ways (possibly with other data structures) so that their behaviors can be composed or combined. Corresponding to each Boolean or temporal connective in our specification language, we identify a combinator on Mealy machines which implements the desired behavior.

We observe that formulas in our temporal logic can be rewritten so that only a few combinators are necessary: (1) combinators which combine the output of Mealy machines running in parallel by applying a binary operation on their respective outputs, (2) combinators which compute a running aggregate on the results of a Mealy machine, (3) combinators which compute running aggregates over sliding windows, and (4) combinators which withhold the results of a machine until a given number of updates. We will see that most of these can be implemented in a straightforward way. Applying a binary operation to the current output values of two running machines can be done with a stateless construction. Computing running aggregates efficiently can be achieved by storing the aggregate of the trace seen so far. In order to withhold the results of a given machine, we can simply store them in a queue of a fixed length. Computing aggregates over sliding windows is slightly trickier. This is usually achieved with an algorithm that maintains monotonic wedges [43]. However, this assumes that the semantic values are totally ordered, which is not necessarily true in our setting of lattices. Instead, we use an algorithm that is inspired by the well-known implementation of a queue data structure using two stacks, popular in functional programming. A variant of this algorithm can be used for computing sliding-window aggregates for any associative operation in a way that every execution step of the monitor needs $O(1)$ amortized time.

Our Coq formalization and extracted code are available in a public GitHub Repository*.

Chapter Outline. In Sect. 2.2, we first introduce lattices and then present the syntax and semantics of our temporal specification language. In Sect. 2.3, we give a formal definition of Mealy machines, present a collection of Mealy combinators, and discuss in detail their implementation. In Sect. 2.4, we discuss the extraction of executable OCaml code from the Coq scripts, use it as a verification oracle and we compare its performance against the monitoring tool Reelay [30]. Finally, in Sect. 2.5, we discuss several different quantitative semantics for Signal Temporal Logic, various algorithmic approaches to online monitoring, and we also give a brief overview of related efforts to produce formally verified monitors.

2.2 Metric Temporal Logic

In this section, we review metric temporal logic (MTL), which will be the formalism that we consider here for specifying quantitative properties. We use bounded distributive lattices as semantic value domains for our logic. While this abstract algebraic setting is not usually how MTL is interpreted, we will see that the standard qualitative (Boolean) and quantitative (robustness) semantics can be obtained simply by choosing the appropriate lattice.

*`https://github.com/Agnishom/lattice-mtl`

2.2.1 Lattices

A lattice is a partial order in which every two elements have a least upper bound and a greatest lower bound. We will use an equivalent algebraic definition.

Definition 2.1. A *lattice* is a set A together with associative and commutative binary operations $\sqcap$ and $\sqcup$, called *meet* and *join* respectively, that satisfy the *absorption laws*, i.e, $x \sqcup (x \sqcap y) = x$ and $x \sqcap (x \sqcup y) = x$ for all $x, y \in A$.

Let A be a lattice. Using the absorption laws it can be shown that $\sqcup$ is idempotent: $x \sqcup x = x \sqcup (x \sqcap (x \sqcup x)) = x$ for every $x \in A$. Similarly, it can also be shown that $\sqcap$ is idempotent. Define the relation $\sqsubseteq$ as follows: $x \sqsubseteq y$ iff $x \sqcup y = y$ for all $x, y \in A$. The relation $\sqsubseteq$ is a partial order. It also holds that $x \sqsubseteq y$ iff $x \sqcap y = x$. For all $x, y \in A$, the element $x \sqcup y$ is the supremum (least upper bound) of $\{x, y\}$ and the element $x \sqcap y$ is the infimum (greatest lower bound) of $\{x, y\}$ w.r.t. the order $\sqsubseteq$.

Definition 2.2. A lattice A is said to be *bounded* if there exists a *top* element $\top \in A$ and a *bottom* element $\bot \in A$ such that $\bot \sqcup x = x$ and $x \sqcap \top = x$ (equivalently, $\bot \sqsubseteq x \sqsubseteq \top$) for every $x \in A$.

Let A be a bounded lattice. It is easy to check that $x \sqcup \top = \top$ and $\bot \sqcap x = \bot$ for every $x \in A$. For a finite subset $X = \{x_1, x_2, \ldots x_n\}$ of a bounded lattice, we write $\bigsqcup X$ for $x_1 \sqcup x_2 \sqcup \cdots \sqcup x_n$ and similarly $\bigsqcap X$ for $x_1 \sqcap x_2 \sqcap \cdots \sqcap x_n$. Moreover, we define $\bigsqcup \varnothing$ to be $\bot$ and $\bigsqcap \varnothing$ to be $\top$. So, $\bigsqcup X$ is the supremum of X and $\bigsqcap X$ is the infimum of X.

Definition 2.3. A lattice A is said to be *distributive* if $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ and $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ for all $x, y, z \in A$.

Example 2.4. Consider the two-element set $\mathbb{B} = \{\top, \bot\}$ of Boolean values, where $\top$ represents truth and $\bot$ represents falsity. The set $\mathbb{B}$, together with conjunction as meet and disjunction as join, is a bounded and distributive lattice.

Example 2.5. The set $\mathbb{R}$ of real numbers, together with min as meet and max as join, is a distributive lattice. However, $(\mathbb{R}, \min, \max)$ is not a bounded lattice. It is commonplace to adjoin the elements ∞ and $-\infty$ to $\mathbb{R}$ so that they serve as the top and bottom elements respectively.

2.2.2 Syntax and Semantics

We fix a set $\mathbb{D}$ of *data items*. We denote by $\mathbb{D}^\omega$ the set of infinite sequences over $\mathbb{D}$, which can also be thought of as functions of type $\mathbb{N} \to \mathbb{D}$. We call members of $\mathbb{D}^\omega$ *traces*. We also consider non-empty strings over $\mathbb{D}$, denoted $\mathbb{D}^+$, which we call (trace-)*prefixes*. Given a trace σ, we use $\sigma|_n$ to denote the finite string $\sigma(0)\sigma(1)\cdots\sigma(n)$.

We also fix a bounded distributive lattice $\mathbb{V}$, whose elements are *quantitative truth values* that represent degrees of truth or falsity. Given a formula, our quantitative semantics will associate a truth value (from $\mathbb{V}$) with each position of the trace. The set Φ of temporal formulas that we consider is given by the following grammar:

$$\varphi, \psi ::= f : \mathbb{D} \to \mathbb{V} \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathsf{P}_I\varphi \mid \mathsf{H}_I\varphi \mid \varphi \,\mathsf{S}_I\, \psi \mid \varphi \,\overline{\mathsf{S}}_I\, \psi,$$

where I is an interval of the form $[a, b]$ or $[a, \infty)$ with $a, b \in \mathbb{N}$. For every temporal connective $X \in \{\mathsf{P}, \mathsf{H}, \mathsf{S}, \overline{\mathsf{S}}\}$, we will write X_a as an abbreviation for $X_{[a,a]}$ and X as an abbreviation for $X_{[0,\infty)}$. We interpret formulas from Φ over traces $\mathbb{D}^\omega$ at specific positions using the *robustness* interpretation function $\rho : \Phi \times \mathbb{D}^\omega \times \mathbb{N} \to \mathbb{V}$, defined as follows:

$$
\begin{aligned}
\rho(f,\sigma,i) &= f(\sigma(i)) \\
\rho(\varphi \vee \psi, w, i) &= \rho(\varphi,\sigma,i) \sqcup \rho(\psi,\sigma,i) \\
\rho(\varphi \wedge \psi, \sigma, i) &= \rho(\varphi,\sigma,i) \sqcap \rho(\psi,\sigma,i) \\
\rho(\mathsf{P}_I\varphi,\sigma,i) &= \bigsqcup_{\substack{j\in I\\ i-j\geq 0}} \rho(\varphi,\sigma,i-j) \\
\rho(\mathsf{H}_I\varphi,\sigma,i) &= \bigsqcap_{\substack{j\in I\\ i-j\geq 0}} \rho(\varphi,\sigma,i-j) \\
\rho(\varphi\,\mathsf{S}_I\,\psi,\sigma,i) &= \bigsqcup_{\substack{j\in I\\ i-j\geq 0}} \Big(\rho(\psi,\sigma,i-j) \sqcap \bigsqcap_{k<j} \rho(\varphi,\sigma,j-k)\Big) \\
\rho(\varphi\,\overline{\mathsf{S}}_I\,\psi,\sigma,i) &= \bigsqcap_{\substack{j\in I\\ i-j\geq 0}} \Big(\rho(\psi,\sigma,i-j) \sqcup \bigsqcup_{k<j} \rho(\varphi,\sigma,j-k)\Big)
\end{aligned}
$$

Note that $\rho(\mathsf{P}_a\varphi,\sigma,i) = \bot$ and $\rho(\mathsf{H}_a\varphi,\sigma,i) = \top$ whenever $a > i$. The semantics of $\wedge$, H and $\overline{\mathsf{S}}$ can be obtained from that of $\vee$, P and S by switching the roles of $\sqcap$ and $\sqcup$. Thus, they will be referred to as dual operators.

Since we are interpreting formulas over discrete traces, our logic is expressively equivalent to LTL with a "Previous" operator. In other words, temporal connectives (including S and $\overline{\mathsf{S}}$; see Lemma 2.14) with bounded intervals can be rewritten in terms of multiple compositions of the Previous operator instead. Also, note that our temporal logic does not include negation. However, this does not limit expressiveness as we discuss in the examples below.

Example 2.6. Continuing from Example 2.4, we choose $\mathbb{D}$ to be $\mathbb{B}^k$ and we set $\mathbb{V}$ to $\mathbb{B}$. The set of functions from $\mathbb{B}^k \to \mathbb{B}$ considered may be restricted to projections $\pi_i(b_1,\dots b_i,\dots b_k) = b_i$ and negated projections $\pi_i(b_1,\dots b_i,\dots b_k) = \overline{b_i}$. This gives us the standard qualitative semantics for metric temporal logic. Formulas with negation can be expressed equivalently as formulas in negation normal form (NNF) in a fairly standard way by pushing negation inside while interchanging operators for their dual operators.

It can be illustrative to examine the interpretations of the temporal operators in this specific context, since the lattice-based case presented in this chapter is a straightforward generalization. The operators P, H and S could be informally read as 'in the **p**ast', '**h**istorically' and '**s**ince'. The formula $\mathsf{P}_{[a,b]}p$ holds at time t if p holds at some time in the interval $[t-b, t-a]$ (i.e, *in the past* of t). The formula $\mathsf{H}_{[a,b]}p$ holds at time t if p holds at every time in the interval $[t-b, t-a]$ (i.e, was *historically* true). The formula $p\,\mathsf{S}_{[a,b]}\,q$ holds at time t if q holds at some time t' where $t-b \leq t' \leq t-a$ and p holds at every time in the interval $(t', t]$ (i.e, p was true *since* q was true).

Example 2.7. We can also express a past-time version of STL interpreted over discrete time in this framework. To do so, take $\mathbb{D} = \mathbb{R}^k$. A qualitative semantics is obtained by taking $\mathbb{V}$ to be $\mathbb{B}$ and restricting the functions to comparisons of the form $(r_1, \ldots, r_i, \ldots, r_k) \mapsto r_i \sim c$ where $c \in \mathbb{R}$ and $\sim \in \{\leq, \geq, =\}$. A quantitative semantics can be obtained by taking $\mathbb{V}$ to be $\mathbb{R} \cup \{\infty, -\infty\}$ (as in Example 2.5) and considering functions of the form $(r_1, \ldots, r_i, \ldots, r_k) \mapsto r_i - c$ or $(r_1, \ldots r_i \ldots r_k) \mapsto c - r_i$. Even in the quantitative setting, STL formulas with negation can be presented in our framework by considering NNF, again by pushing negation inside while interchanging operators for their dual operators and replacing $r_i - c$ with $c - r_i$.

Our formalism is a *past-time only* logic. This means that the robustness value at a point can be determined by the trace prefix up to that position. This idea can be stated formally in the form of the following claim.

Lemma 2.8. Suppose σ, τ are traces such that $\sigma|_n = \tau|_n$ for some $n \in \mathbb{N}$. Then, for any formula $\varphi \in \Phi$ and for every $i \leq n$, $\rho(\varphi, \sigma, i) = \rho(\varphi, \tau, i)$.

This suggests a way to interpret formulas on trace-prefixes. Suppose $w \in \mathbb{D}^+$, and $\sigma \in \mathbb{D}^\omega$ is some trace such that $\sigma|_{|w|} = w$. Then, we define $\rho(\varphi, w) = \rho(\varphi, \sigma, |w|)$. Lemma 2.8 implies that this definition does not depend on the specific choice of σ.

2.3 The Monitoring Problem

Monitoring is the processing of an input trace in order to detect specified patterns. For quantitative properties, this could be thought of as applying a valuation function on a trace. In an online setting, the trace is supplied to the monitor incrementally. To elaborate, the monitor consumes fragments of the trace one at a time and the monitor is required to evaluate the quantitative property on the trace prefix seen so far. Below, we outline a compositional approach for monitoring quantitative properties denoted by MTL formulas.

2.3.1 Monitors as Mealy Machines

We will use a variant of Mealy machines, a class of string transducers, as a formal model of online monitoring algorithms.

Definition 2.9. Let A and B be sets. A *Mealy machine* with input items from A and output values in B is a tuple $(\texttt{St}, \texttt{init}, \texttt{mNext}, \texttt{mOut})$ where $\texttt{St}$ is a (possibly infinite) set of states, $\texttt{init} \in \texttt{St}$ is the initial state, $\texttt{mNext} : \texttt{St} \times A \to \texttt{St}$ is a transition function which transitions the state of the machine upon seeing an input from A, and $\texttt{mOut} : \texttt{St} \times A \to B$ provides an output at the current state, given an element from A. We write $\texttt{Mealy}(A, B)$ for the set of all Mealy machines with inputs from A and outputs from B.

While this is similar to the standard definition of Mealy Machines found in the literature, we use an equivalent, co-inductive definition in our formalization. In the co-inductive view (see [44]), the states are not explicitly expressed, but described directly in terms of their extensional behavior.

```
CoInductive Mealy (A B : Type) : Type := {
  mOut : A -> B;
  mNext : A -> Mealy A B;
}.
```

The functions `mNext` and `mOut` denote the incremental update and output of the machine, respectively, which consume traces element by element. We can extend these functions to `gNext` and `gOut` to consume non-empty strings, more generally. We can think of the function `gOut` as the quantitative property that the machine associates with the given trace-prefix.

Definition 2.10. Let $m \in \texttt{Mealy}(A, B)$. Then, $\texttt{gNext}(m) : A^* \to \texttt{Mealy}(A, B)$ is defined by $\texttt{gNext}(m, \varepsilon) = m$ and $\texttt{gNext}(m, w \cdot a) = \texttt{mNext}(\texttt{gNext}(m, w), a)$. We define $\texttt{gOut}(m) : A^+ \to B$ by $\texttt{gOut}(m, w \cdot a) = \texttt{mOut}(\texttt{gNext}(m, w), a)$.

For a quantitative property of trace-prefixes, i.e, a function $f : \mathbb{D}^+ \to \mathbb{V}$, we wish to construct a Mealy machine that computes f. In particular, we are interested in quantitative properties which arise as denotations of MTL formulas.

Definition 2.11. Let $\varphi \in \Phi$ and $m \in \texttt{Mealy}(\mathbb{D}, \mathbb{V})$. We say that the Mealy machine m *implements a monitor for* φ if $\texttt{gOut}(m, w) = \rho(\varphi, w)$ for all $w \in \mathbb{D}^+$.

Example 2.12. Following Definition 2.9, consider the machine $m : \texttt{Mealy}(\mathbb{V}, \mathbb{V})$ with states $\mathbb{V}$ (indicating that it stores one element of type $\mathbb{V}$), initial state $\bot$, $\texttt{mOut}(u, a) =$

u and $\texttt{mNext}(u,a) = a$. It holds that $\texttt{gOut}(m, v_1) = \bot$ and $\texttt{gOut}(m, v_1 v_2) = v_1$, $\texttt{gOut}(m, v_1 v_2 v_3) = v_2$, etc. The machine m implements a monitor for the formula $\mathsf{P}_1(v \mapsto v)$ in the sense of Definition 2.11.

Stated formally, the monitoring problem for MTL is to find a translation function $\texttt{toMonitor} : \Phi \to \texttt{Mealy}(\mathbb{D}, \mathbb{V})$ so that given any $\varphi \in \Phi$, $\texttt{toMonitor}(\varphi)$ implements a monitor for φ.

2.3.2 Monitor Combinators

Combinators are compositional constructs that let one define new machines in terms of existing ones. Our approach towards solving the monitoring problem is to find combinators that correspond to the temporal and Boolean connectives of MTL. With these combinators, a monitor for a given formula can be specified by induction on the structure of the formula. A similar approach for MTL with bounded future-time connectives is considered in [45, 36]. The compositional construction of transducers, called temporal testers, for temporal formulas has been studied in [46, 47, 48]. The use of combinators for specifying more general computations for stream processing has been considered in the design of the domain-specific languages StreamQRE [49] and StreamQL [50]. Quantitative regular expressions (QREs) [51, 49] (see also [52] and [53]) are particularly relevant. QREs have been used to specify complex algorithms for medical monitoring [54, 55]. Moreover, the relationship between QREs and automata-theoretic models with registers is investigated in [56, 57, 58].

Proceeding with the idea of compositional monitor construction, we identify the key constructs that are necessary in achieving the expressive power of MTL. We say that the formulas φ and ψ are *equivalent*, and we write $\varphi \equiv \psi$, if $\rho(\varphi, \sigma, i) = \rho(\psi, \sigma, i)$ for all traces $\sigma \in \mathbb{D}^\omega$ and positions $i \in \mathbb{N}$.

Lemma 2.13. The following identities hold:

$$\mathsf{P}_{[a,b]}\varphi \equiv \mathsf{P}_a\mathsf{P}_{[0,b-a]}\varphi \tag{2.1}$$

$$\mathsf{H}_{[a,b]}\varphi \equiv \mathsf{H}_a\mathsf{H}_{[0,b-a]}\varphi \tag{2.2}$$

$$\varphi \,\mathsf{S}_{[a+1,b]}\, \psi \equiv \mathsf{H}_{[0,a]}\varphi \wedge \mathsf{P}_{a+1}\left(\varphi \,\mathsf{S}_{[0,b-(a+1)]}\, \psi\right) \tag{2.3}$$

$$\varphi \,\overline{\mathsf{S}}_{[a+1,b]}\, \psi \equiv \mathsf{P}_{[0,a]}\varphi \vee \mathsf{H}_{a+1}\left(\varphi \,\overline{\mathsf{S}}_{[0,b-(a+1)]}\, \psi\right) \tag{2.4}$$

$$\mathsf{P}_{[a,\infty)}\varphi \equiv \mathsf{P}_a\mathsf{P}_{[0,\infty)}\varphi \tag{2.5}$$

$$\mathsf{H}_{[a,\infty)}\varphi \equiv \mathsf{H}_a\mathsf{H}_{[0,\infty)}\varphi \tag{2.6}$$

$$\varphi \,\mathsf{S}_{[a+1,\infty)}\, \psi \equiv \mathsf{H}_{[0,a]}\varphi \wedge \mathsf{P}_{a+1}\left(\varphi \,\mathsf{S}_{[0,\infty)}\, \psi\right) \tag{2.7}$$

$$\varphi \,\overline{\mathsf{S}}_{[a+1,\infty)}\, \psi \equiv \mathsf{P}_{[0,a]}\varphi \vee \mathsf{H}_{a+1}\left(\varphi \,\overline{\mathsf{S}}_{[0,\infty)}\, \psi\right) \tag{2.8}$$

The proofs of the identities of Lemma 2.13 are straightforward. Proving the identities involving S (or $\overline{\mathsf{S}}$) requires the distributivity axioms, which motivates the need for considering distributive lattices.

Lemma 2.14. The following identities hold:

$$\varphi \,\mathsf{S}_{[0,a]}\, \psi \equiv (\varphi \,\mathsf{S}\, \psi) \wedge \mathsf{P}_{[0,a]}\psi \tag{2.9}$$

$$\varphi \,\overline{\mathsf{S}}_{[0,a]}\, \psi \equiv (\varphi \,\overline{\mathsf{S}}\, \psi) \vee \mathsf{H}_{[0,a]}\psi \tag{2.10}$$

Proof. We will only prove the first identity, since the second one can be proved by dualizing the same argument. Let $\sigma \in \mathbb{D}^\omega$ be an arbitrary trace and $n \in \mathbb{N}$ a position. We define $s_i = \rho(\varphi, \sigma, n-i)$ and $t_i = \rho(\psi, \sigma, n-i)$ for every $i \in \mathbb{N}$. Then, we have that

$$\rho(\varphi \,\mathsf{S}_{[0,a]}\, \psi, \sigma, n) = \bigsqcup_{i \le K}\Big(t_i \sqcap \prod_{j<i} s_j\Big)$$

$$\rho(\varphi \,\mathsf{S}\, \psi, \sigma, n) = \bigsqcup_{i \le n}\Big(t_i \sqcap \prod_{j<i} s_j\Big) = \rho(\varphi \,\mathsf{S}_{[0,a]}\, \psi, \sigma, n) \sqcup \bigsqcup_{K<i\le n}\Big(t_i \sqcap \prod_{j<i} s_j\Big)$$

$$\rho(\mathsf{P}_{[0,a]}\psi, \sigma, n) = \bigsqcup_{i \le K} t_i$$

where $K = \min(a, n)$. We have to prove that $L = R \sqcap Q$, where $L = \rho(\varphi \mathsf{S}_{[0,a]} \psi, \sigma, n)$, $R = \rho(\varphi \mathsf{S} \psi, \sigma, n)$ and $Q = \rho(\mathsf{P}_{[0,a]}\psi, \sigma, n)$. From $K \le n$ we obtain that $L \sqsubseteq R$. It also holds that $L \sqsubseteq Q$ because $t_i \sqcap \bigsqcap_{j<i} s_j \sqsubseteq t_i$ for every $i \le K$. It follows that $L \sqsubseteq R \sqcap Q$. It remains to show that $R \sqcap Q \sqsubseteq L$. Since

$$\begin{aligned} R \sqcap Q &= \Big(L \sqcup \bigsqcup_{K<i\le n} \Big(t_i \sqcap \bigsqcap_{j<i} s_j\Big)\Big) \sqcap Q \\ &= (L \sqcap Q) \sqcup \bigsqcup_{K<i\le n} \Big(t_i \sqcap \bigsqcap_{j<i} s_j \sqcap Q\Big) \\ &= (L \sqcap Q) \sqcup \bigsqcup_{K<i\le n} \bigsqcup_{k\le K} \Big(t_i \sqcap t_k \sqcap \bigsqcap_{j<i} s_j\Big), \end{aligned}$$

it suffices to establish that $L \sqcap Q \sqsubseteq L$ (which is true) and $t_i \sqcap t_k \sqcap \bigsqcap_{j<i} s_j \sqsubseteq L$ for every i and k with $K < i \le n$ and $k \le K$. Since $k < i$, we conclude that $t_i \sqcap t_k \sqcap \bigsqcap_{j<i} s_j \sqsubseteq t_k \sqcap \bigsqcap_{j<k} s_j \sqsubseteq L$. □ □

Remark 2.15. In the qualitative setting, the identities of Lemma 2.14 are intuitively clear, but they require a more careful argument in the quantitative setting. They have been used and proven in [59] for the lattice $(\mathbb{R}, \min, \max)$, but the given proof does not generalize to the class of lattices that we consider here. As we can see in the proof of Lemma 2.14, there is a subtlety in dealing with the terms of $\rho(\varphi \mathsf{S} \psi, \sigma, n)$ with index $i = K + 1, \dots, n$.

The first set of identities allows us to express $\mathsf{P}_{[\bullet,\bullet]}$, $\mathsf{S}_{[\bullet,\bullet]}$ in terms of $\mathsf{P}_{[0,\bullet]}$, $\mathsf{S}_{[0,\bullet]}$ and $\mathsf{P}_{\bullet}$. The second set of identities implies that $\mathsf{S}_{[0,\bullet]}$ can be replaced by S and $\mathsf{P}_{\bullet}$. Thus, the only additional constructs required in expressing the bounded temporal operators are $\mathsf{P}_{\bullet}$ and $\mathsf{P}_{[0,\bullet]}$ (and their duals).

We present in Figure 2.1 a summary of the combinators that we will consider. Each combinator can be thought of as the implementation of the corresponding Boolean or temporal connective. The key observation is that this association between combina-

$$\frac{f : \mathbb{D} \to \mathbb{V}}{\texttt{mAtomic}\ f : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad k : \mathbb{N}}{\texttt{mDelay}\ k\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad k : \mathbb{N}}{\overline{\texttt{mDelay}}\ k\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}$$

$$\frac{m_1 : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\texttt{mAnd}\ m_1\ m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m_1 : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\texttt{mOr}\ m_1\ m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}$$

$$\frac{m_1 : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\texttt{mSince}\ m_1\ m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m_1 : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\overline{\texttt{mSince}}\ m_1\ m_2 : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}$$

$$\frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\texttt{mSometime}\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}{\texttt{mAlways}\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}$$

$$\frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad k : \mathbb{N}}{\texttt{mSometimeBounded}\ k\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})} \qquad \frac{m : \texttt{Mealy}(\mathbb{D}, \mathbb{V}) \qquad k : \mathbb{N}}{\texttt{mAlwaysBounded}\ k\ m : \texttt{Mealy}(\mathbb{D}, \mathbb{V})}$$

Figure 2.1 : Summary of Mealy Combinators

tors on Mealy machines and connectives *respect* the implementation relation (Definition 2.11) between machines and formulas. E.g., if m is a monitor for φ, we expect `mSometimeBounded` $k\ m$ to be a monitor for $\mathsf{P}_{[0,k]}\varphi$.

The definition of the `toMonitor` function which constructs monitors from formulas is shown in Figure 2.2. As discussed, it proceeds by rewriting the formula into the desired form and then replacing each temporal or Boolean connective with the corresponding combinator. The main correctness claim for the monitor is stated as follows:

```
Theorem toMonitor_correctness:
  forall φ, implements (toMonitor φ) φ.
```

The proof of this theorem is done using induction on the structure of the formula. Once the identities in Lemmas 2.13 and 2.14 are proven, this can be done using the correctness properties of individual combinators.

Before we start describing each combinator in detail, we make some remarks about the general organization of our implementation and formal proofs. There is a lot of

$$[f] = \texttt{mAtomic}\ f$$

$$\begin{array}{rlrl}
[\varphi \wedge \psi] &= \texttt{mAnd}\ [\varphi]\ [\psi] & [\varphi \vee \psi] &= \texttt{mOr}\ [\varphi]\ [\psi] \\
[\mathsf{P}\varphi] &= \texttt{mSometime}\ [\varphi] & [\mathsf{H}\varphi] &= \texttt{mAlways}\ [\varphi] \\
[\mathsf{P}_{[0,a]}\varphi] &= \texttt{mSometimeBounded}\ a\ [\varphi] & [\mathsf{H}_{[0,a]}\varphi] &= \texttt{mAlwaysBounded}\ a\ [\varphi] \\
[\mathsf{P}_{a}\varphi] &= \texttt{mDelay}\ a\ [\varphi] & [\mathsf{H}_{a}\varphi] &= \overline{\texttt{mDelay}}\ a\ [\varphi] \\
[\varphi\,\mathsf{S}\,\psi] &= \texttt{mSince}\ [\varphi]\ [\psi] & [\varphi\,\overline{\mathsf{S}}\,\psi] &= \overline{\texttt{mSince}}\ [\varphi]\ [\psi] \\
[\mathsf{P}_{[a,\infty)}\varphi] &= [\mathsf{P}_{a}\mathsf{P}_{[0,\infty)}\varphi] \quad (a > 0) & [\mathsf{H}_{[a,\infty)}\varphi] &= [\mathsf{H}_{a}\mathsf{H}_{[0,\infty)}\varphi] \quad (a > 0) \\
[\mathsf{P}_{[a,b]}\varphi] &= [\mathsf{P}_{a}\mathsf{P}_{[0,b-a]}\varphi] \quad (a > 0, a \neq b) & [\mathsf{H}_{[a,b]}\varphi] &= [\mathsf{H}_{a}\mathsf{H}_{[0,b-a]}\varphi] \quad (a > 0, a \neq b) \\
[\varphi\,\mathsf{S}_{[a+1,b]}\,\psi] &= [\mathsf{H}_{[0,a]}\varphi \wedge \mathsf{P}_{a+1}(\varphi\,\mathsf{S}_{[0,b-(a+1)]}\,\psi)] & [\varphi\,\overline{\mathsf{S}}_{[a+1,b]}\,\psi] &= [\mathsf{P}_{[0,a]}\varphi \vee \mathsf{H}_{a+1}(\varphi\,\overline{\mathsf{S}}_{[0,b-(a+1)]}\,\psi)] \\
[\varphi\,\mathsf{S}_{[a+1,\infty)}\,\psi] &= [\mathsf{H}_{[0,a]}\varphi \wedge \mathsf{P}_{a+1}(\varphi\,\mathsf{S}\,\psi)] & [\varphi\,\overline{\mathsf{S}}_{[a+1,\infty)}\,\psi] &= [\mathsf{P}_{[0,a]}\varphi \vee \mathsf{H}_{a+1}(\varphi\,\overline{\mathsf{S}}\,\psi)] \\
[\varphi\,\mathsf{S}_{[0,a]}\,\psi] &= [(\varphi\,\mathsf{S}\,\psi) \wedge \mathsf{P}_{[0,a]}\psi] & [\varphi\,\overline{\mathsf{S}}_{[0,a]}\,\psi] &= [(\varphi\,\overline{\mathsf{S}}\,\psi) \vee \mathsf{H}_{[0,a]}\psi]
\end{array}$$

Figure 2.2 : The `toMonitor` function

symmetry among these combinators that can be leveraged for economy of effort. An example is the presence of dual connectives. This is why in many cases we focus on presenting these combinators in a slightly general way before instantiating them specifically to $\texttt{Mealy}(\mathbb{D}, \mathbb{V})$. As discussed before, the correctness for each combinator is phrased in terms of preserving the implementation relation – these theorems are indexed with the suffix `correctness`. These theorems are proven via lemmas indexed with the suffix `result` that characterize the most recent output of the Mealy machine at some point in the computation. The proofs proceed by induction on the trace seen so far. They require additional lemmas that establish invariants about the state of a Mealy machine as it evolves during the computation. These latter lemmas are indicated with the suffix `state`. These ideas are illustrated in the construction of `mAtomic` in Figure 2.3.

```
CoFixpoint mAtomic {A B : Type} (f : A -> B) : Mealy A B :=
  {| mOut x := f x;
     mNext _ := mAtomic f; |}.
Lemma mAtomic_state {A B : Type} (f : A -> B) (l : nonEmpty A) :
  gNext (mAtomic f) l = mAtomic f.
Lemma mAtomic_result {A B : Type} (f : A -> B) (l : nonEmpty A) :
  gOut (mAtomic f) l = f (latest l).
Lemma monAtomic_correctness :
  forall f, implements (monAtomic f) (FAtomic f).
```

Figure 2.3 : Establishing correctness of `mAtomic`.

Atomic Functions.

In order to lift functions $f : A \to \mathbb{V}$ to `Mealy`$(A, \mathbb{V})$, we define the `mAtomic` combinator, as shown in Figure 2.3. Given a function $f : A \to \mathbb{V}$, it defines a Mealy machine that applies f to the latest input element. We use the lemma `mAtomic_state` to describe the evolution of the machine when an arbitrary stream prefix is fed. Using this, we also prove `mAtomic_result`, which describes the final output of the machine after accepting an arbitrary stream prefix. The lemma titled `mAtomic_correctness` establishes that `mAtomic` correctly translates atomic functions to corresponding monitors.

Pointwise Binary Operations.

In Figure 2.4, we define the combinator `mBinOp` that combines the output of two given machines using a binary operation. By plugging in $\sqcup$ and $\sqcap$ as `op`, we can use `mBinOp` to implement the $\vee$ and $\wedge$ connectives, respectively. Like in the case of `mAtomic`, the correctness of this combinator is proven by establishing appropriate

```
CoFixpoint mBinOp {A B C D : Type} (op : B -> C -> D)
 (m : Mealy A B) (n : Mealy A C) : Mealy A D := {|
  mOut (a : A) := op (mOut m a) (mOut n a);
  mNext (a : A) := mBinOp op (mNext m a) (mNext n a);
|}.
Definition monAnd (m n : Mealy A Val) : Mealy A Val :=
  mBinOp meet m n.
Lemma monAnd_correctness m1 m2 φ1 φ2 :
  implements m1 φ1 -> implements m2 φ2
  -> implements (monAnd m1 m2) (FAnd φ1 φ2).
Definition monOr (m n : Mealy A Val) : Mealy A Val :=
  mBinOp join m n.
Lemma monOr_correctness m1 m2 φ1 φ2 :
  implements m1 φ1 -> implements m2 φ2
  -> implements (monOr m1 m2) (FOr φ1 φ2).
```

Figure 2.4 : The `mBinOp` combinator

```
Lemma delayWith_state (q : Queue) (m : Mealy A B) (l : nonEmpty A) :
    forall initSeg, initSeg = (back q) ++ rev (front q)
 -> forall k, k = length initSeg
 -> forall stream, stream = (toList (gCollect m l)) ++ initSeg
 -> forall lastSeg, lastSeg = firstn k stream
 -> exists newFront newBack,
                lastSeg = newBack ++ rev newFront
              /\ length lastSeg = k
              /\ gNext (delayWith q m) l
                 = delayWith (Build_Queue newFront newBack) (gNext m l).
```

Figure 2.5 : Delay monitors.

lemmas which describe the behavior of `mBinOp` with `gNext` and `gOut`. These let us prove, in particular, that `mAnd` and `mOr` correctly implement formulas involving $\wedge$ and $\vee$, respectively.

Delay Monitors.

We view the implementation of $\mathsf{P}_\bullet$ and $\mathsf{H}_\bullet$ as a mechanism that delays the output of a Mealy Machine. For instance, the sequence $\langle \rho(\mathsf{P}_2\varphi, a_1a_2a_3), \rho(\mathsf{P}_2\varphi, a_1a_2a_3a_4)\rangle$ is same as $\langle \rho(\varphi, a_1), \rho(\varphi, a_1a_2)\rangle$. These operators preserve the order of the outputs, but delay them by a given constant.

This can be achieved using a queue maintained at a fixed length. For instance, to implement $\mathsf{P}_a\varphi$, we maintain a queue of length a. Upon being given an input item $a \in \mathbb{D}$, we feed a to `toMonitor`(φ), enqueue the result and then return what we obtain by dequeuing. This works since the dequeued element was the result of `toMonitor`(φ) a turns ago. The queue needs to be initially filled with a instances of $\bot$ (or $\top$ in the case of H_a) since we have that $\rho(\mathsf{P}_a\varphi, w) = \bot$ (or $\rho(\mathsf{H}_a\varphi, w) = \top$) when $|w| > a$.

Since Coq is based on a functional programming environment, functional lists are the ordered collections that are the easiest for us to reason about and work with. Functional lists are typically implemented via linked lists, which means that in order to access the kth element of the list, one would have to traverse k links and would spend $O(k)$ time. This makes appending to the end of the list expensive. However, obtaining or adding elements at the head (the beginning) of the list is straightforward. Thus, these lists effectively behave as stacks and sometimes we refer to them as such. We use the well-known technique of implementing a queue with two functional lists, which we briefly discuss below.

A queue is represented by two lists `front` and `rear`. When an element is enqueued, it is added to the head of the `rear` list. Thus, the `rear` list effectively stores the elements of the queue in an order opposite to that in which they were enqueued. When dequeuing an element is required, the elements of `rear` are reversed and placed in the `front` (thus restoring the order), and the head of `front` is returned. As long as `front` is non-empty, subsequent dequeues may be directly handled by returning the head of `front`.

In our use case, the queue is maintained at a fixed length, say k, and every enqueue is followed by a subsequent dequeue. Reversing `rear` into `front` takes time $O(k)$. However, we only need to do this every k turns, since `front` is filled with k items whenever the reversal happens. Thus, every k turns, we do $O(k)$ work and only $O(1)$ work is needed otherwise. This gives us an amortized time complexity of $O(1)$.

We implement this idea in the `delayWith` combinator in Figure 2.5. The key lemma required in proving the correctness of the `delayWith` combinator shows that the queue maintained always stores the last k-many outputs of the submonitor. To

```
CoFixpoint mFold {A B : Type} (op : B -> B -> B)
 (m : Mealy A B) (init : B) : Mealy A B := {|
  mOut (a : A) := op init (mOut m a);
  mNext (a : A) := mFold op (mNext m a) (op init (mOut m a)); |}.
Definition mSometime (m : Mealy A Val) : Mealy A Val :=
  mFold join m bottom.
Definition monAlways (m : Mealy A Val) : Mealy A Val :=
  mFold meet m top.
```

Figure 2.6 : Temporal Folds

formalize this, we define $\texttt{gCollect} : \texttt{Mealy}(A, B) \times \mathbb{D}^{+} \to \mathbb{V}^{+}$ as

$$\begin{aligned}&\texttt{gCollect}(m, a_1a_2\cdots a_n) = \\ &\langle \texttt{gOut}(m, a_1), \texttt{gOut}(m, a_2), \cdots, \texttt{gOut}(m, a_1a_2\cdots a_n)\rangle.\end{aligned}$$

We may now write the mentioned invariant as in `delayWith_state`, which is established by induction on the input stream.

Temporal Folds.

The unbounded operators P and H can be thought of as a running fold on the input stream, since $\rho(\mathsf{P}\varphi, w \cdot a) = \rho(\mathsf{P}\varphi, w) \sqcup \rho(\varphi, w \cdot a)$ (and similarly for H). Thus, to evaluate these operators in an online fashion, we only need to store the robustness value for the trace seen so far. For P (resp., H), the robustness of the current trace can then be obtained by computing the join (resp., meet) of the current value and the stored one. In Figure 2.6, `mAlways` (resp., `mSometime`) computes the robustness values corresponding to the H (resp., P) connectives by computing the meet (resp., join) of

```
CoFixpoint sinceAux (m1 m2 : Mealy A Val) (pre : Val) : Mealy A Val :=
  {| mOut (a : A) :=  (mOut m2 a) ⊔ (pre ⊓ (mOut m1 a));
     mNext (a : A) :=
      sinceAux (mNext m1 a) (mNext m2 a)
        ((mOut m2 a) ⊔ (pre ⊓ (mOut m1 a)))
  |}.

Definition monSince (m1 m2 : Mealy A Val) : Mealy A Val :=
  sinceAux m1 m2 bottom.
```

Figure 2.7 : Monitoring Since

the current value with the stored one. Their proof of correctness is a straightforward induction on the trace-prefix, using the incremental equation involving the operator.

Using the following identity, we may also view the computation of S as a temporal fold, i.e, the robustness for $\varphi \mathsf{S} \psi$ may be calculated incrementally by only storing the robustness value for the stream prefix so far.

Lemma 2.16. For all $w \in \mathbb{D}^+$ and $a \in \mathbb{D}$, we have that

$$\rho(\varphi \mathsf{S} \psi, w \cdot a) = \rho(\psi, w \cdot a) \sqcup (\rho(\varphi \mathsf{S} \psi, w) \sqcap \rho(\varphi, w \cdot a)).$$

This is a well-known equality and can be proved by using distributivity in a straightforward way. A proof of this for the $(\mathbb{R}, \max, \min)$ lattice appears in [60].

Using the equality of Lemma 2.16, `mSince` can be implemented as in Figure 2.7. The correctness of `mSince` is established by proving invariants on `mSinceAux`, which is straightforward once the equality above has been established.

```
Definition aggQueue_inv (l : list B) (q : aggQueue) :=
    exists olds news,
      olds ++ news = l
      /\ new q = rev news
      /\ newAgg q = finite_op _ (rev (new q))
      /\ length (oldAggs q) = length olds
      /\ forall i , nth i (oldAggs q) unit = finite_op _ (skipn i olds).

Definition agg_inv (l : list B) (q : aggQueue) :=
    agg q = finite_op _ l.
Lemma aggQueue_agg_inv l q :
    aggQueue_inv l q -> agg_inv l q.

Lemma enqueue_aggQueue_inv l q :
    aggQueue_inv l q
    -> forall n, aggQueue_inv (l ++ [n]) (aggEnqueue n q).

Lemma aggDequeue_aggQueue_inv x xs q :
    aggQueue_inv (x :: xs) q
    -> aggQueue_inv xs (aggDequeue q).
```

Figure 2.8 : Invariants for aggQueue

Windowed Temporal Folds.

For the operators $\mathsf{P}_{[0,a]}$ or $\mathsf{H}_{[0,a]}$, the strategy above needs to be modified, since the fold is over a sliding window, rather than the entire trace. For this purpose, we use a queue like data structure (dubbed `aggQueue`, henceforth) which also maintains sliding window aggregates, in addition. An extended discussion of a similar data structure can be found in [61].

While we intend to use `aggQueue` specifically for computing sliding window join and meet on bounded lattices, the algorirthmic idea behind the data structure only uses two ideas involving $\sqcup$ (resp., $\sqcap$). Namely: (1) $\sqcup$ (resp., $\sqcap$) is associative (2) $\bot$ (resp., $\top$) are identities for $\sqcup$ (resp., $\sqcap$). These features make the lattice elements a monoid under $\sqcup$ (resp., $\sqcap$). in the remainder of this section, we will describe the algorithm for a monoid $(\mathtt{B}, \cdot, \mathbb{1})$.

As the name suggests, we can think of `aggQueue` as a data structure with a queue-like interface: it supports operations $\mathtt{aggEnqueue} : \mathtt{aggQueue} \times \mathtt{B} \to \mathtt{aggQueue}$ and $\mathtt{aggDequeue} : \mathtt{aggQueue} \to \mathtt{aggQueue}$ which allow enqueuing elements of `B` or dequeueing them. However, instead of a peek operation, we are interested in an aggregate operation $\mathtt{agg} : \mathtt{aggQueue} \to \mathtt{B}$ which reports the aggregate of all the elements in the queue.

To implement a usual queue, we maintained two lists: a rear list into which the `new` elements are enqueued, and a front list from which elements can be dequeued. Here, since we are interested in only knowing the aggregates, we replace the front list with an `oldAggs` list, which stores partial aggregates instead. Additionally, we keep track of `newAgg`, the aggregate of the values in `new`. Suppose that the contents of the represented queue are a list `l`. Then, the invariant we want to maintain suggests that `l` can be broken into two parts `olds` and `news` such that (1) `new` contains the elements of `news` in reverse order (2) `newAgg` is the aggregate of the elements of `news` (3) `oldAggs` has the same length as that of `olds` (4) The i-th element of `oldAggs` is the aggregate of the $|\texttt{olds}| - i$ elements of `olds`. Given these invariants, it is easy to see that the aggregate of the entire queue can be computed as the aggregate of `newAgg` and the head of `oldAggs`.

We maintain these invariants in the following way: Upon enqueuing $b \in \texttt{B}$, we simply add b to the head of `new` and update `newAgg` to $\texttt{newAgg} \cdot b$. Performing a dequeue is easy when `oldAggs` is non-empty: we simply remove the element at its head. When `oldAggs` is empty, the contents of `new` are added as incremental partial aggregates to `oldAggs`. In Figure 2.8, we show a formalization of the invariants that one needs to prove.

To keep a sliding window aggregate of the last k elements, `mSometimeBounded` (or `mAlwaysBounded`) initializes an `aggQueue` filled with k instances of $\mathbb{1}$ (i.e, $\bot$ (or $\top$)). When a new input is available, the monitor enqueues the result of the corresponding submonitor into the queue and dequeues the element which was enqueued k turns ago. The output of the machine is simply the aggregate of the elements in the queue. Using a similar argument as before, we can see that the invocations of `mNext` on these machines run in $O(1)$ amortized time (with a worst-case behaviour of $O(k)$ which is invoked every k turns). See Table 2.1 for an illustration of the execution of such a machine.

The correctness of the algorithm can be established via `mWinFold_state`. In essence, it states that the `contentsff` and `contentsrr` together store the last k elements of the stream, and that the invariants on `aggsff` and `aggsrr` are maintained.

Theorem 2.17. Assume that elements of $\mathbb{V}$ can be stored in constant space and the lattice operations on $\mathbb{V}$ can be computed in constant time and space. Further, let

`l`	`new`	`oldAggs`	`newAgg`	`agg`
$\langle \mathbb{1}, \mathbb{1}, \mathbb{1} \vert \rangle$	$\langle \rangle$	$\langle \mathbb{1}, \mathbb{1}, \mathbb{1} \rangle$	$\mathbb{1}$	$\mathbb{1}$
$\langle \mathbb{1}, \mathbb{1} \vert a \rangle$	$\langle a \rangle$	$\langle \mathbb{1}, \mathbb{1} \rangle$	a	$\mathbb{1} \cdot a$
$\langle \mathbb{1} \vert a, b \rangle$	$\langle b, a \rangle$	$\langle \mathbb{1} \rangle$	ab	$\mathbb{1} \cdot ab$
$\langle \vert a, b, c \rangle$	$\langle c, b, a \rangle$	$\langle \rangle$	abc	$\mathbb{1} \cdot abc$
$\langle b, c \vert d \rangle$	$\langle d \rangle$	$\langle bc, c \rangle$	d	$bc \cdot d$
$\langle c \vert d, e \rangle$	$\langle e, d \rangle$	$\langle c \rangle$	de	$c \cdot de$

Table 2.1 : A run of the sliding window algorithm that aggregates the last 3 elements. The elements a, b, c, d, e are fed in, incrementally. We use $\vert$ as a separator in `l` to indicate the old and the new parts of the queue. Note that the contents of `l` itself are not stored.

$\varphi \in \Phi$ be a formula that only uses atomic functions that can be computed in constant time and space. Then, `toMonitor`(φ) is a Mealy machine whose state can be stored in $O(2^{|\varphi|})$ space and the transition (resp., output) functions `mNext` (resp., `mOut`) can be computed in amortized $O(|\varphi|)$ time per item.

Note: The exponential in the formula stems from the fact that the constants in the formula are encoded in binary. Note that this is unavoidable since computing the value of $\mathsf{P}_a p$ would require storing the last a values of p.

Proof of Theorem 2.17. This claim can be established via a straightforward induction on the structure of the formula φ. At each step in the induction, we need to show a constant space and amortized time overhead is created.

If φ is an atomic predicate, then computing φ can be done in constant time by assumption and it requires no additional state.

If $\varphi = \alpha \bullet \beta$ for $\bullet \in \{\wedge, \vee\}$, then we may assume by induction that `toMonitor`(α) (resp., `toMonitor`(β)) use $O(2^{|\alpha|})$ (resp., $O(2^{|\beta|})$) space and amortized $O(|\alpha|)$ (resp., $O(2^{|\beta|})$) time. The machine `toMonitor`($\alpha * \beta$) uses the states of both `toMonitor`(α) and `toMonitor`(β) and can be stored in $O(2^{|\alpha|} + 2^{|\beta|}) = O(2^{|\varphi|})$ space. By assumption, the additional time required to compute the lattice operation to combine the outputs of `toMonitor`(α) and `toMonitor`(β) is $O(1)$. So, this takes $O(|\alpha|) + O(|\beta|) + O(1) = O(|\varphi|)$ amortized time.

If $\varphi = X_{[0,\infty)}\alpha$ for $X \in \{\mathsf{P}, \mathsf{H}\}$ or $\alpha S_{[0,\infty)}\beta$, then the analysis is similar. In this case, the additional state we need to store is an element of $\mathbb{V}$, which we can store in $O(1)$ space. The additional time required is just a constant number of lattice operations, which can be done in $O(1)$ time. Thus, the inductive invariant is preserved in this case.

If $\varphi = X_a\alpha$ or $X_{[0,a]}\alpha$ for $X \in \{\mathsf{P}, \mathsf{H}\}$, then it is handled using a delay buffer or a sliding window aggregator as discussed. In both of these cases, a buffer of length $O(a)$ (i.e, $O(2^{|a|})$) is used. These queue mechanisms, as discussed above, are used in an "enqueue followed by dequeue" manner. The dequeue operations generally take $O(1)$ time but every a inputs involve reversal of the buffer which takes $O(a)$ time. This amounts to an amortized time of $O(1)$ per item. $\square$

2.4 Extraction and Experiments

We use Coq's extraction mechanism to produce OCaml code for our `toMonitor` function. This gives us an OCaml library the interface of which we show in Figure 2.9. The extracted `toMonitor` function can be instantiated with arbitrary bounded distributive lattices by specifying the operations $\sqcap$ and $\sqcup$ and the corresponding identities $\top$ and $\bot$.

For our experiments, we Following Example 2.5, we wish to emulate STL and use the lattice $(\mathbb{R} \cup \{\pm\infty\}, \max, \min)$. To do this, we model $\mathbb{R}$ with the concrete OCaml type `float`, which are 64-bit floating-point numbers. We also use $\mathbb{R}$ (modelled by `float`) for the set of data items $\mathbb{D}$. We compare the performance of our monitor with Reelay [30] (a C++ library) and the implementation for semiring-based monitoring algorithms in Rust from [36].

We have observed that the rate at which these tools process items roughly approaches a constant rate. Most notably, there are periodic spikes of latency that can be observed in our monitor, which correspond to the reversal of the lists in our queue based algorithms. A similar behavior is seen in the semiring-monitor, but this is harder to observe since the Rust implementation is very fast. We summarize performance using the amortized (i.e., average) time taken to process an item. To

```
type ('v, 'a) formula =

| FAtomic of ('a -> 'v)

| FAnd of ('v, 'a) formula * ('v, 'a) formula

| FOr of ('v, 'a) formula * ('v, 'a) formula

| FSometime of int * int * ('v, 'a) formula

| FAlways of int * int * ('v, 'a) formula

| FSometimeUnbounded of int * ('v, 'a) formula

| FAlwaysUnbounded of int * ('v, 'a) formula

| FSince of int * int * ('v, 'a) formula * ('v, 'a) formula

| FSinceDual of int * int * ('v, 'a) formula * ('v, 'a) formula

| FSinceUnbounded of int * ('v, 'a) formula * ('v, 'a) formula

| FSinceDualUnbounded of int * ('v, 'a) formula * ('v, 'a) formula


type 'a lattice = { join : ('a -> 'a -> 'a); meet : ('a -> 'a -> 'a) }

type 'a boundedLattice = { bottom : 'a; top : 'a }


val toMonitor :

  'a1 lattice -> 'a1 boundedLattice

    -> ('a1, 'a2) formula -> ('a1, 'a2) monitor
```

Figure 2.9 : Extracted OCaml Code

microbenchmark the building blocks of our algorithm, we consider formulas $X_{[0,n]}$, X_n, $X_{[n,2n]}$, $X_{[n,\infty)}$ where $X \in \{\mathsf{S}, \mathsf{P}\}$ and plot their performance with respect to n in Fig. 2.10. We notice that for Reelay, the performance depends on the type of input stream provided; so, we report our findings for a stream whose elements are random, a stream whose elements form an increasing sequence and another which forms a decreasing sequence. In our tool and the semiring-monitor, the performance of the monitor seems to be roughly independent of the stream. Beyond certain values of the constants, some of the experiments with Reelay seemed to take prohibitively long time to process a given stream, preventing us from reliably measuring the performance at these values. Generally, we see that our tool has been performing better than Reelay but slower than the semiring-monitor, at least by an order of magnitude. We also see that the performance of our tool is roughly independent of the constants in the formula, as we expected from the analysis of our algorithms. The reported data is based on the mean of 6 trials of the experiments. The standard deviation is less than 15% of the mean in each case, and is indicated by whiskers.

A potential explanation for the comparatively worse performance of Reelay is that Reelay stores data values in string-indexed maps. Interval Maps are also used in Reelay's implementation of operators such as $\mathsf{P}_{[\bullet,\bullet]}$. Since our tool does not use any map-like data structure, we do not incur these costs.

We have used the profiling tool Valgrind [62] to analyze the memory consumption of the monitors. In Fig. 2.10, we plot the peak memory usage of the monitors for the same formulas as before. For Reelay, we have reported the performance for three different traces. In the case of the semiring-monitor, the memory consumption can be explained near-perfectly with the help of the description of the algorithm (which is very similar to ours). This can be attributed to the fact that Rust programs have very minimal runtime overheads. The memory usage of Reelay is somewhat hard to understand, given that it is based on the Interval Maps data structures. Our tool is implemented in OCaml and its memory consumption is hard to interpret due to the garbage-collected nature of the language, however we do see a linear trend in the memory consumption with sufficiently high values for constants. The memory measurements for all three tools seemed to be deterministic, i.e, had the same value regardless of when it was executed.

In Figure 2.11, we use formulas inspired by the Timescales [63] benchmark to see how our tool performs when the constants in the formulae are scaled. The formulas used in the Timescales benchmark are in propositional MTL, so we define the propositions p, q, r and s as $x > 0.5$, $x > 0.0$, $x > 0.25$ and $x > 0.75$ respectively, where x is the value of the current sample in the trace. For different values of n, the formulas F_0 through F_9 in Figure 2.11, in order, are: $\mathsf{H}(\mathsf{P}_{[0,n]}q \rightarrow (\neg p\,\mathsf{S}\,q))$, $\mathsf{H}(r \rightarrow \mathsf{P}_{[0,n]}(\neg p)$, $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P}q) \rightarrow (\neg p \mathsf{S}_{[n,2n]}\,q))$, $\mathsf{H}(\mathsf{P}_{[0,n]}q \rightarrow (p \mathsf{S} q))$, $\mathsf{H}(r \rightarrow \mathsf{H}_{[0,k]}p)$, $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P}q) \rightarrow (p\,\mathsf{S}_{[n,2n]}\,q))$, $\mathsf{H}\mathsf{P}_{[0,n]}p$, $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P}q) \rightarrow (\mathsf{P}_{[0,n]}(p \vee q)\,\mathsf{S}\,q))$, $\mathsf{H}((s \rightarrow \mathsf{P}_{n,2n}p) \wedge \neg(\neg s \mathsf{S}_{[n,\infty)}p))$, and $\mathsf{H}((r \wedge \neg q \wedge \mathsf{P}q) \rightarrow ((s \rightarrow \mathsf{P}_{[n,2n]}p) \wedge \neg(\neg s \mathsf{S}_{[n,\infty)}p)))$. Implications $\alpha \rightarrow \beta$ were encoded as $\neg\alpha \vee \beta$ and negations were encoded using their negation normal form. We have executed this experiment using traces with random values. The reported values are means of 10 trials, and we have used whiskers to denote the standard deviation.

All experiments were run on a computer with Intel Xeon CPUs 3.30 GHz with 16 GB memory running Ubuntu 18.04.

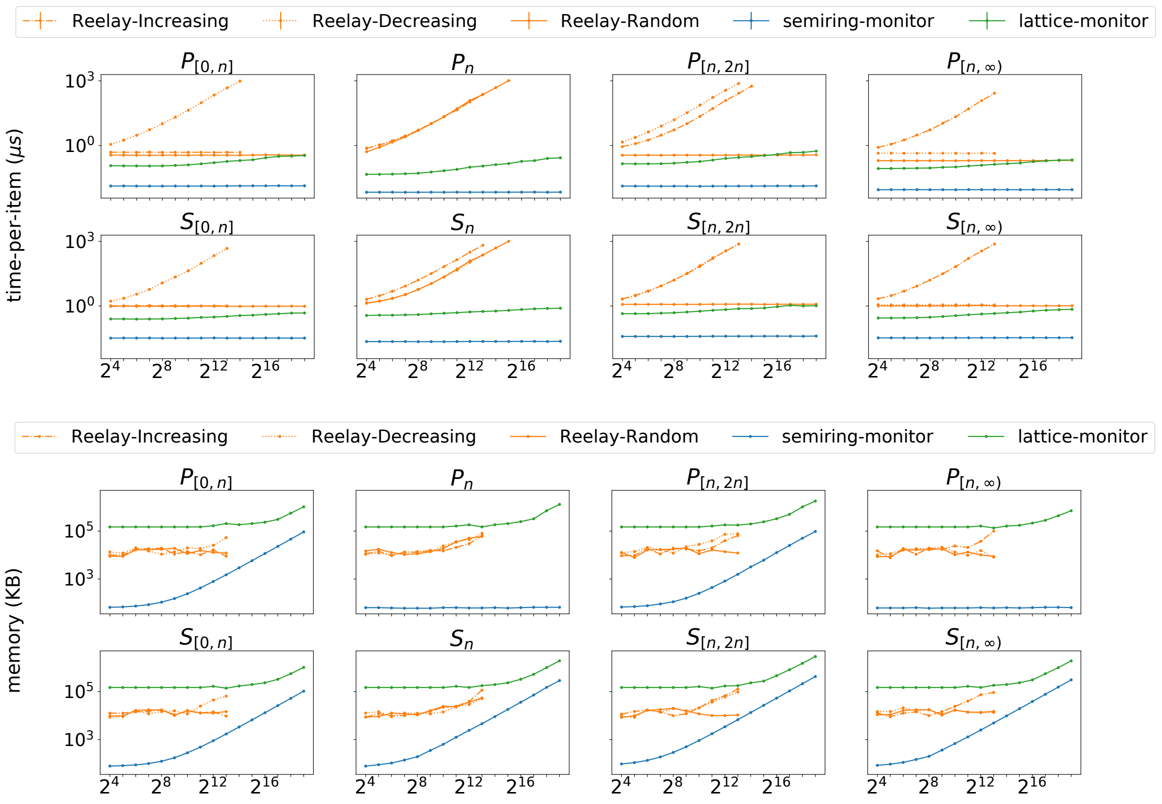


Figure 2.10 : Microbenchmarks: Formulas with large constants

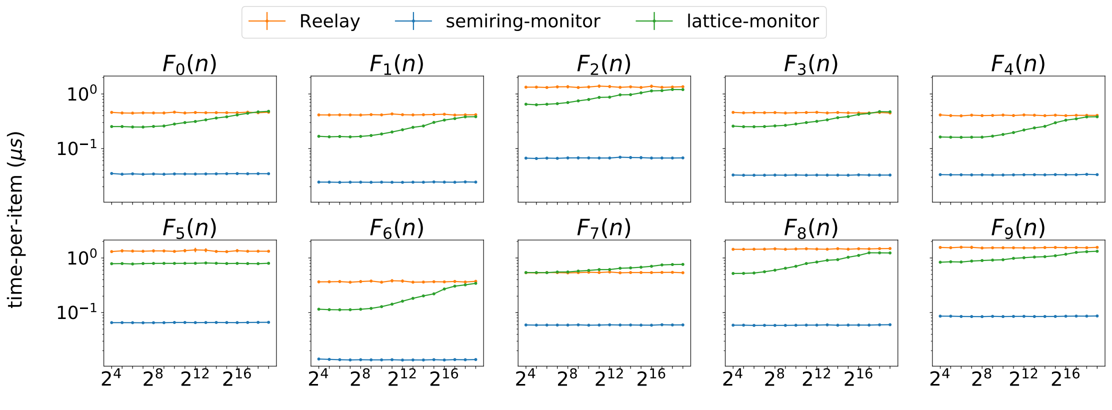


Figure 2.11 : Throughput for formulas from the Timescales benchmark

2.5 Related Work

Fainekos and Pappas [29] introduce the notion of robustness for the interpretation of temporal formulas over discrete and continuous-time signals. In their setting, signals are represented as (time-dependent) functions that take value in a metric space. The distance function of the metric space is used to endow the signal space with a metric. The robustness of satisfaction is defined to be the largest extent to which a signal can be perturbed while still satisfying (or violating, depending on the case) the specification. In the same paper, an alternative quantitative semantics is proposed with an inductive definition that replaces disjunction with max and conjunction with min. This inductive semantics computes an under-approximation of the actual robustness value. This approach is extended by Donzé and Maler [42] to include temporal robustness. The inductive semantics of [29] can be computed efficiently, and forms the basis for the semantics we use.

The inductive semantics could be slightly generalized by interpreting conjunction (resp., disjunction) with multiplication (resp., addition) in some semiring. This idea subsumes the semantics of this paper since lattices are also semirings. In [36], this semantics is explored and an abstract version of the underapproximation guarantee from [29] is presented. With our approach, we would not be able to monitor formulas with this semantics since we make crucial use of the absorption laws in Lemma 2.14. In [36], a different approach for monitoring formulas with $\mathsf{S}_{[0,a]}$ is discussed that does not rely on this property. It is also noted that a semiring in which Lemma 2.14 holds must be a bounded distributive lattice. It is worth noting that monitoring $\overline{\mathsf{S}}$ or $\overline{\mathsf{S}}_{[0,a]}$ in this inductive semiring-based framework with an analog of Lemma 2.16 would require addition to distribute over multiplication. Our lattice-based semantics is considered in a dense-time setting in [17], along with a performance analysis of its

Rust implementation.

The simple dynamic-programming algorithm for offline monitoring of Linear Temporal Logic (with future modalities, in discrete-time, without metric or quantitative extensions) has been a part of the folk-lore for a long time; early references can be seen in [64]. Sen et al. [65] have considered the problem of detecting *good* and *bad* prefixes of infinite words with respect to future-time Boolean temporal logic. Good prefixes are (finite) words w such that any (infinite) extension $w \cdot \sigma$ of them would satisfy the logical formula. Similarly, bad prefixes are finite words such that no extension of them satisfies the formula. Sen and their coauthors have proven that such an automaton must have $\Omega(2^{2^{|\varphi|}})$ states. In other words, such an algorithm must use $\Omega(2^{|\varphi|})$ bits of space. The proof involves encoding the following language using an LTL formula of size $O(k^2)$:

$$L_k = \{\sigma \# w \# \sigma' \$ w \mid w \in \{0,1\}^k \text{ and } \sigma, \sigma' \in \{0,1,\#\}^*\}.$$

This technique has also been used by [66] and [67] to prove lower bounds involving temporal logic. Thati and Rosu have discussed the complexity of monitoring algorithms for MTL (with Boolean semantics) in [68], where they have shown that monitoring MTL with future modalities requires at least $\Omega(2^{\alpha c \sqrt{|\underline{\varphi}|}})$, where $\underline{\varphi}$ is φ without any numerical constants, c is the largest constant occurring in φ and α is a fixed constant. The exponential blowup in the space complexity of monitoring can be attributed to the fact that the monitor is anchored at position 0 of the trace, and must maintain an additional amount of state in order to monitor the future. This is in contrast to our approach, where the monitor only needs to summarize information on the part of the trace that is already available. The online monitoring algorithm with a similar complexity as ours was first published by Mamouras and Wang [45].

The notion of distance between traces from [29] has been generalized in [69] to a more general algebraic setting using a semiring-based semantics. While both Mamouras et al. [36] and Jaksic et al. [69] consider truth domains that are semirings, the two works consider different semantics. Jaksic et al. suggest the use of symbolic weighted automata for the purpose of monitoring. With this approach, they are able to compute the precise robustness value for a property-signal pair. The construction of a weighted automaton from a temporal formula incurs a doubly exponential blowup, if one assumes a succinct binary representation of the constants appearing in an MTL formula.

The distance between two signals can be defined to be the maximum of the distance between the values that the signals take at corresponding points of time. However, other ways to define this distance have been considered. In [70], a quantitative semantics is developed via the notion of weighted edit distance. Averaging temporal operators are proposed in [71] with the goal of introducing an explicit mechanism for temporal robustness. The Skorokhod metric [72] has been suggested as a distance function between continuous signals. In [73], another metric is considered, which compares the value taken by the signal within a neighbourhood of the current time. Another interesting view of temporal logic is in [74], where temporal connectives are viewed as linear time-invariant filters.

Signal regular expressions (SREs) [75] are another formalism for describing patterns on signals. They are based on regular expressions, rather than LTL. SREs are a variant of the timed regular expressions (TREs) of [76], which are related to timed automata [77]. A robustness semantics for SRE has been proposed in [78] along with an algorithm for offline monitoring. In [79], STL is enriched by considering a more general (and quantitative) interpretation of the Until operator and adding specific aggregation operators. They also give a semantics of their formalism using dual numbers, which are the real numbers with an adjoined element ϵ satisfying $\epsilon^2 = 0$.

In [80], a monitoring algorithm for STL is proposed and implemented in the AMT tool. A later version, AMT 2.0 [81] extends the capabilities of AMT to an extended version of STL along with TREs. In [59], an efficient algorithm for monitoring STL is discussed whose performance is linear in the length of the input trace. This is achieved by using Lemire's [43] sliding window algorithm for computing the maximum. This is implemented as a part of the monitoring tool Breach [82]. A dynamic programming approach is used in [60] to design an online monitoring algorithm. Here, the availability of a predictor is assumed which predicts the future values, so that the future modalities may be interpreted. A different approach for online monitoring is taken in [83]: they consider robustness intervals, that is, the tightest interval which covers the robustness of all possible extensions of the available trace prefix. There are also monitoring formalisms that are essentially domain-specific languages for processing data streams, such as LOLA [84] and StreamQRE [49, 52]. LOLA has recently been used as a basis for RtLOLA [85] in the StreamLAB framework [86], which adds support for sliding windows and variable-rate streams. A detailed survey on the many extensions to the syntax and semantics of STL along with their monitoring algorithms and applications is presented in [14].

In [87], a framework towards the formalization of runtime verification components are discussed. MonPoly [88] is a tool developed by Basin et al. aimed at monitoring a first-order extension of temporal logic. In [89], the authors put forward Verimon, a simplified version of MonPoly which uses the proof assistant Isabelle/HOL to formally prove its correctness. They extend this line of work in Verimon+ [90] which verifies a more efficient version of the monitoring algorithms and uses a dynamic logic, which is an extension of the temporal logic with regular expression-like constructs. A verifying compiler for LOLA specifications to rust implementations has been developed in [91]. Their toolchain generates rust code from given LOLA specifications that are decorated with annotations that can be checked by the Viper [92] toolkit to verify functional correctness. On the one hand, a Rust implementation would perform very well. On the other, the verification mechanism in this toolchain is based on SMT solvers while ours is based on the axioms of Coq's calculus, which is a much smaller system.

Chapter 3

Matching Regular Expressions with Lookarounds

3.1 Introduction

While textbook regular expressions involve only concatenation $(r \cdot s)$, alternation $(r + s)$ and Kleene iteration r^*, modern regex engines (such as PCRE [34]) support a multitude of additional features. Some of these features are purely syntactic sugar, such as the use of character classes (e.g., `[a-z]`) to indicate predicates or the use of the ? operator (e.g., `e?`) to indicate zero or one occurrence. Some of these features, like counting (e.g., `e{3,5}`) add exponential succinctness. Features like backreferences make it possible to match certain non-regular languages (e.g., `(a*)b\1` defines the language $\{a^n b a^n | n \geq 0\}$). Certain constructs, such as capture groups or lookarounds, do not add to the expressive power in terms of the overall strings that can be matched, but are best described using a different semantics (i.e., different from the simple Boolean semantics of $w \in [\![r]\!]$ and $w \notin [\![r]\!]$).

The focus of this chapter is on expressions with lookarounds. Lookahead assertions (respectively, lookbehind assertions) are used to assert that a certain pattern is satisfied in the future (respectively, in the past) of the current position in the string. For example, one could use the expression `\d+\.\d\d(?= USD )` to match a price of some item formatted as `xxx.yy USD`. Since the pattern `USD` appears within a lookahead, this part of the is string is not returned, so that the programmer can directly work with the numerical information. Similarly, if the pattern `USD` was expected to appear before the price, one could use the expression `(?<= USD )\d+\.\d\d`, where the pattern `USD` is within a lookbehind.

The semantics of lookarounds is dependent on context: given a substring w and a regular expression with lookaround r, we cannot determine whether w matches r without considering the string in which w appears. Standards such as PCRE [34] are not helpful here, since they describe the semantics in an operational style making it unsuitable to verify a given implementation. In this chapter, we use a satisfaction relation $w, [i, j] \vDash r$ (indicating that the substring $w[i, j]$ satisfies the regular expression r) which makes the ambient string w explicit.

The majority of popular regex engines bundled with the standard library of programming languages such as Java, Javascript and Python are based on backtracking, which may take exponential time in the worst-case scenario (called *catastrophic backtracking* [93]). This behavior could be used to conduct a denial-of-service attack [94]. Automata-based tools such as Google's RE2 [95] and Intel's Hyperscan [96] are guaranteed to finish execution in linear time. Despite this, backtracking engines remain popular due to their flexibility and support for non-classical features, including (but not limited to) lookarounds. To the best of our knowledge, none of the widely used automata-based engines support lookarounds.

While regular expressions with lookarounds can be compiled into NFA, doing so would incur an exponential blowup (see, for example, [97] for a lower bound) and this strategy would not be practical for matching. The algorithm we describe in this chapter approach runs in time $O(m \cdot n)$, where m is the size of the regex and n is the length of the input string. This approach does not suffer from this blowup because it does not construct a large automaton. Moreover, it processes the string using both left-to-right and right-to-left passes, unlike the typical streaming (left-to-right)

NFA simulation. The key idea behind our algorithm is to decompose expressions with lookarounds into different layers using the nesting structure of the lookarounds. The algorithm proceeds in a bottom-up fashion, first resolving the lookaround assertions at the lower layers, and then using this information to resolve lookarounds at the higher layers. Thus, when we encounter a subexpression `(?=`r`)`, we first match the subexpression r on the input string and record the results. Any subsequent requirements to resolve the lookaround assertion can be directly answered by using the recorded results. Formally, this is done by replacing lookaround assertions with *oracle queries* and enriching the strings with *oracle valuations* which are used to resolve the queries. Note that, with this approach, we construct NFAs for each expression in the decomposition instead of constructing a single large NFA for the entire expression.

We have extended algorithms based on Marked Regular Expressions [32, 33] to match certain enriched expressions. In contrast to an alternative approach using NFAs (as we have done in our paper [28]), this approach is purely functional. Marked expressions allow simulating NFAs in a manner that closely resembles the syntax of regular expressions, facilitating elegant verification without compromising performance.

Chapter Outline. In Section 3.2, we provide an informal exposition of the semantics of regular expressions with lookarounds and the matching algorithm illustrated using examples. In Section 3.3, the formal syntax and semantics are presented. In Section 3.4, we discuss how this semantics could be leveraged for equational reasoning, and illustrate this with the equational theory of anchors. In Section 3.5, we define Oracle Regular Expressions and Oracle Strings, the core abstraction used in our algorithm. The matching algorithm for oracle expressions is discussed in Section 3.6. The main algorithm is presented in Section 3.7. Performance experiments using our extracted code are presented in Section 3.8. Finally, we discuss related work in Section 3.9.

3.2 Overview of the Algorithm

In this section, we provide an informal exposition of the semantics of regular expressions with lookarounds and the algorithm for matching them, along with illustrative examples. Formal definitions of the lookaround semantics are presented in Section 3.3.

The examples in this section are over ASCII characters. In PCRE notation [34], the character-class `.` matches all characters. Thus, the *positive lookaround assertion* l_1 = `(?=.*d.*e)` asserts that to the right of the current position, there is some occurrence of the letter `d` followed by an occurrence of the letter `e`. Let's consider the string w_1 = `bcbdbebc` with $|w_1|$ = 8. In w_1, `d` occurs at position 3 and e occurs at position 5. We interpret lookaheads as zero-width assertions, meaning that their matches are considered to be windows of length 0. In this case, the windows $[0, 0]$, $[1, 1]$, $[2, 2]$ and $[3, 3]$ satisfy the assertion l_1. Given the regex r_1 = `bc(?=.*d.*e)`, the occurrence of the string `bc` at the window $[0, 2]$ matches r_1. However, the occurrence of the string `bc` at the window $[6, 8]$ does not since it fails the assertion. Note that r_1 is different from the expression $r_2 =$ `bc.*d.*e` which matches the longer window $[0, 6]$ containing the string `bcbdbe`.

The character-class `\d` refers to digits. The *negative lookahead assertion* `(?!\d\d)` asserts that the next two characters in the string are not digits. The expression r_3 = `((?!\d\d).)+` nests the assertion within a Kleene plus, thus forcing every character in the match window to satisfy it. In this case, the expression r_3 would match a string if it doesn't contain any two consecutive digits. Consider the string w_2 = `a1b1c1` and

w_3 = `a1b21c1`. The expression r_3 matches any (non-zero length) window in the string w_2 since there are no consecutive occurrences of digits. In contrast, in w_3, only the windows not containing the substring `21` would match r_3.

One could use the *positive lookbehind* l_2 = `(?<=c.*)` to indicate the presence of the character `c` in the past of the current position in the string. Lookarounds can also be nested within each other. Consider l_3 = `(?<=c((?!\d\d).)+)` which searches for an occurrence of the character `c` followed by a non-empty sequence of characters that do not contain two consecutive digits. Let w_4 = `c01c0` with $|w_4| = 5$. The expression l_3 would match only the (zero-length) windows $[4,4]$ and $[5,5]$, since the substring `01` after the first `c` disqualifies it as a witness.

Consider the regex r = `(?<=c((?!\d\d).)+)(a+)b(?=.*d.*e)` and the string w = `cabc77dcaab7dabe`. The four potential matches for the subexpression `(a+)b` are indicated below by highlighting the substrings corresponding to their windows at $[1,3]$, $[8,11]$, $[9,11]$ and $[13,15]$:

$$c\,\underline{a\,b}\,c\,7\,7\,d\,c\,\overline{a\,\underline{a\,b}}\,7\,d\,\underline{a\,b}\,e$$

The window $[1,3]$ does not satisfy the lookbehind. This is because even though there is a `c` at position 0, the substring `77` appearing at $[4,6]$ in the future disqualifies it. The window $[8,11]$ does not satisfy the lookbehind either, since the `+` requires at least one character after the occurrence of `c` before the match window. The window $[13,15]$ does not satisfy the lookahead since there is no `d` in the future. It can be checked that only the window $[9,11]$ are matches.

Our algorithm can handle arbitrarily nested lookarounds. The key idea is to decompose the regular expression with each nested layer of lookaround as its own layer. At the bottom layer, since there are no lookarounds, we can use a standard NFA-based matching algorithm. In subsequent layers, we use the matching information from the previous layers to resolve the lookaround assertions. (See 3.7) This is done by considering a slightly enriched notion of an NFA which processes strings augmented with oracle valuations (See Section 3.5).

We decompose the regex r into the following expressions:

$$\begin{aligned} \hat{r} &= \mathsf{Q}^{+}(s_1)\,\texttt{(a+)b}\,\mathsf{Q}^{+}(s_2) \\ s_1 &= (\mathsf{LookBehind}, [\,\texttt{c}((\mathsf{Q}^{-}(s_3)\,\texttt{.})^{+}\,]) \\ s_2 &= (\mathsf{LookAhead}, [\,\texttt{.*d.*e}\,]) \\ s_3 &= (\mathsf{LookAhead}, \left[\,\texttt{\textbackslash d\textbackslash d}\,\right]) \end{aligned}$$

We have used the notation Q^{+} and Q^{-} to indicate whether the lookaround is positive or negative and tagged each subexpression with a direction.

For the subexpressions s_1, s_2 and s_3, we will compute corresponding *tapes* τ_1, τ_2 and τ_3. At position i, the tape for each lookahead (respectively, lookbehind) assertion indicates whether the subexpression matches $[0, i]$ (respectively, $[i, |w|]$). We start by computing the tapes τ_2 and τ_3 since s_2 and s_3 do not depend on any other expressions.

string w	c	a	b	c	7	7	d	c	a	a	b	7	d	a	b	e	
tape τ_2	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
tape τ_3	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

The lookahead s_2 searches for the occurrence of a `d` followed by a `e` in the future. The final occurrence of a `d` is at position 12 and the final occurrence of an `e` is at position 15. Thus, the tape τ_2 contains a 1 at all positions before position 12 and all 0s afterwards. The lookahead s_3 searches for two consecutive digits in the future. Since this only occurs at the window $[4, 6]$, the tape τ_3 is marked with 1s only before position 4. Next, we turn our attention to s_1 which depends on s_3. To evaluate this, we need to use the oracle valuation τ_3 in conjunction with the string w.

string w	c	a	b	c	7	7	d	c	a	a	b	7	d	a	b	e	
tape τ_3	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

tape τ_1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

The expression s_1 looks for an occurrence of `c` in the past such that it is followed by at least one character which satisfies a negative valuation of s_3. We see that in positions 0 and 3, `c` appears, but the valuation associated with τ_3 disqualifies them. At position 7, this additional constraint is satisfied, and hence all the positions with indices greater than 8 are marked with 1 in the tape τ_1.

Combining the information from tape τ_1 and τ_2 together with w gives us the necessary information for evaluating the top-level expression $\hat{r}$.

string w	c	a	b	c	7	7	d	c	a	a	b	7	d	a	b	e	
tape τ_1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
tape τ_2	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0

Since the expression $\hat{r}$ is $\mathsf{Q}^{+}(s_1)$`(a+)b`$\mathsf{Q}^{+}(s_2)$, it will be matched at a window where the corresponding substring of w matches $a^{+}b$ and the oracle valuation for τ_1 to the left and the oracle valuation for τ_2 to the right are both 1. Inspecting the table above shows that the window $[9, 11]$ is the only one that satisfies these conditions.

3.3 Lookaround Semantics

In this section, we introduce the formal definitions used in this chapter. Note that there is a slight difference between the notation in the following definition and the PCRE notation used in the previous section. The connections will be made clear in Section 3.3.1.

Definition 3.1 (Regular Expressions with Lookaround). Let Σ be an alphabet, and $\mathcal{P}$ a set of decidable predicates over Σ (i.e., functions of type $\Sigma \to \{1,0\}$). The set LReg of regular expressions with lookaround is defined by the following grammar:

$$r, r_1, r_2 ::= \varepsilon \mid \sigma \in \mathcal{P} \mid r_1 \cdot r_2 \mid r_1 + r_2 \mid r^* \mid (?{>}r) \mid (?{\not>}r) \mid (?{<}r) \mid (?{\not<}r)$$

The expressions of the form $(?{>}r)$, $(?{\not>}r)$, $(?{<}r)$ and $(?{\not<}r)$ are called *positive lookahead*, *negative lookahead*, *positive lookbehind* and *negative lookbehind* respectively. Collectively, they are called *lookaround assertions*. Expressions of the form $(?{>}r)$ and $(?{\not>}r)$ (respectively, $(?{>}r)$ and $(?{\not>}r)]$) are called *lookahead assertions* (respectively, *lookbehind assertions*). Furthermore, $(?{>}r)$ and $(?{<}r)$ are called *positive lookarounds*, while $(?{\not>}r)$ and $(?{\not<}r)$ are called *negative lookarounds*.

We write $|w|$ to denote the length of a string w. The empty string (i.e., the string of length 0) is denoted by ε. For a string $w \in \Sigma^*$, we will call a formal pair $[i, j]$ with $0 \leq i \leq j \leq |w|$ a *window* in w.

We use some additional notation throughout the chapter. We write r^+ to denote $r \cdot r^*$, i.e., the repetition of r one or more times. We define the expression $\varnothing$ to be the predicate $\bot$ which does not match any character. This expression has the property that for any $w \in \Sigma^*$ and any window $[i, j]$ of w, we have $w, [i, j] \nvDash \varnothing$.

A *partitioning* of a window $[i, j]$ (where $i \leq j$) is a nonempty finite sequence of windows $[i_1, j_1], [i_2, j_2], \ldots, [i_n, j_n]$ with $i_1 = i$, $j_n = j$, and $j_k = i_{k+1}$ for every $k = 1, 2, \ldots, n-1$. We say that a partition has no empty blocks if $j_\alpha - i_\alpha > 0$ for every α. We can restate the semantics for Kleene Iteration in terms of partitions using the following lemma.

Lemma 3.2. Let $r \in \mathsf{LReg}$, $w \in \Sigma^*$, and $0 \leq i \leq j \leq |w|$. Then, $w, [i, j] \vDash r^*$ iff $i = j$ or there exists a partition $[i_1, j_1], [i_2, j_2], \ldots, [i_n, j_n]$ without empty blocks of $[i, j]$ such that $w, [i_k, j_k] \vDash r$ for every $k = 1, \ldots, n$.

$$
\begin{aligned}
w,[i,j] \vDash \varepsilon &\iff i = j \\
w,[i,j] \vDash \sigma &\iff j = i + 1 \text{ and } \sigma(w_i) = 1 \\
w,[i,j] \vDash r_1 + r_2 &\iff w,[i,j] \vDash r_1 \text{ or } w,[i,j] \vDash r_2 \\
w,[i,j] \vDash r_1 \cdot r_2 &\iff \text{there is } i \le k \le j \text{ s.t. } w,[i,k] \vDash r_1 \text{ and } w,[k,j] \vDash r_2 \\
w,[i,j] \vDash r^* &\iff i = j \text{ or there is } i \le k \le j \text{ s.t. } w,[i,k] \vDash r \text{ and } w,[k,j] \vDash r^* \\
w,[i,j] \vDash (?{>}r) &\iff i = j \text{ and } w,[i,|w|] \vDash r \\
w,[i,j] \vDash (?{\not>}r) &\iff i = j \text{ and } w,[i,|w|] \nvDash r \\
w,[i,j] \vDash (?{<}r) &\iff i = j \text{ and } w,[0,i] \vDash r \\
w,[i,j] \vDash (?{\not<}r) &\iff i = j \text{ and } w,[0,i] \nvDash r
\end{aligned}
$$

Figure 3.1 : Definition of the *satisfaction relation* $\vDash$ relating a string $w \in \Sigma^*$, a window $[i,j]$ with $0 \le i \le j \le |w|$, and a regular expression $r \in \mathsf{LReg}$, possibly with lookarounds.

Lemma 3.2 serves as a sanity check for the semantic definition of Fig. 3.1 for Kleene's star. A related observation is that while defining $w, [i, j] \vDash R^*$, we could have chosen a k that is strictly larger than i without loss of generality.

In Coq, we encoded LReg as an inductive data type (see the definition on the right). We use two inductively defined predicates match_regex, and not_match_regex. The Proposition match_regex r w s d stands for the proposition $w, [s, s+d] \vDash r$, and the Proposition not_match_regex r w s d stands for the negation, i.e., $w, [s, s+d] \nvDash r$. These two predicates are defined using mutual recursion.

```
Inductive LRegex : Type :=
| Epsilon : LRegex
| CharClass : (A -> bool) -> LRegex
| Concat : LRegex -> LRegex -> LRegex
| Union : LRegex -> LRegex -> LRegex
| Star : LRegex -> LRegex
| LookAhead : LRegex -> LRegex
| LookBehind : LRegex -> LRegex
| NegLookAhead : LRegex -> LRegex
| NegLookBehind : LRegex -> LRegex.
```

We define not_match_regex directly, instead of encoding it as the negation of match_regex. This is because such an encoding cannot be used in the arguments of the constructor NegLookAhead or NegLookBehind. The encoding $\sigma \to \bot$ of the negation $\neg\sigma$ as an argument of a constructor would violate strict positivity. We prove the two following lemmas which show that the two predicates behave in the manner expected.

```
Lemma match_not_match : forall r w start delta,
    ~ (match_regex r w start delta) <-> (not_match_regex r w start delta).


Lemma match_lem : forall r w start delta,
    match_regex r w start delta \/ ~ (match_regex r w start delta).
```

In our Coq formalization, we choose to work with the representation $w, [s, s+d] \vDash r$, rather than $w, [i, j] \vDash r$. This is so that we can avoid dealing with additional ordering constraints $i \leq j$ while dealing with proofs.

3.3.1 Relationship to PCRE Semantics.

The lookaround semantics presented in Fig. 3.1 slightly differs from PCRE [34]. The PCRE-style positive lookahead `(?=`r`)` is matched at position i if there is some $i \leq j \leq |w|$ such that $w, [i, j] \vDash r$. This is in contrast to our $(?\!>R)$ which matches until the end of the string, i.e., $w, [i, |w|] \vDash r$. Similarly, the PCRE lookbehind `(?<=`r`)` at position i matches if there exists some $0 \leq j \leq i$ such that $w, [j, i] \vDash r$, whereas $(?\!<)$ matches the entire prefix $[0, i]$. The negative variants `(?!`r`)` and `(?<!`r`)` in PCRE have a similar semantics.

In addition, PCRE uses `^` and `$` to denote the start and end of the string, respectively. This means that $w, [i, j] \vDash$ `^` iff $i = j = 0$, and $w, [i, j] \vDash$ `$` iff $i = j = |w|$. We show that PCRE lookarounds can be encoded in our notation, and vice-versa. The anchors `^` and `$` can be encoded as $(?\!<\varepsilon)$ and $(?\!>\varepsilon)$ respectively. The PCRE positive lookahead `(?=`r`)` can be encoded in our notation as $(?\!>r \cdot \Sigma^*)$, where the Σ^* allows the addition of an arbitrary suffix after the witness for r. Conversely, our positive lookahead $(?\!>r)$ can be encoded in PCRE notation as `(?=`r`$)`, where the `$` enforces that the match of r must be to the end of the string. The other cases (lookbehinds and the negative variants) can be encoded similarly.

3.4 Equational Reasoning

Definition 3.3 (Equivalence and Containment Relations on Regular Expressions)**.** Let $r, s \in \mathsf{LReg}$. We say that r and s are *equivalent*, and we write $r \equiv s$, if for every string $w \in \Sigma^*$ and every window $[i, j]$ of w, we have $w, [i, j] \vDash r \iff w, [i, j] \vDash s$. We also define the *containment* relation $r \sqsubseteq s$ as notational shorthand for $r + s \equiv s$.

We encode the $\equiv$ and $\sqsubseteq$ relations in Coq as follows:

```
Definition regex_eq (r1 r2 : LRegex) : Prop :=
    forall w start delta,
        match_regex r1 w start delta <-> match_regex r2 w start delta.
Definition regex_leq (r s : Regex) : Prop :=
    regex_eq (Union r s) s.
```

We notice that this is an equivalence relation and a congruence with respect to the regular expression combinators such as concatenation, union, Kleene iteration and the lookarounds. In the terminology of Coq, these congruence relations can be expressed as *Proper morphisms*. Declaring them in this manner allows standard rewrite tactics to use $\equiv$-based equations for rewriting inside proofs.

```
Instance regex_eq_equiv : Equivalence regex_eq.
Instance union_proper : Proper (regex_eq ==> regex_eq ==> regex_eq) Union.
Instance concat_proper : Proper (regex_eq ==> regex_eq ==> regex_eq) Concat.
Instance star_proper : Proper (regex_eq ==> regex_eq) Star.
```

An equivalent approach is to define $r \sqsubseteq s$ so that it holds when for every $w \in \Sigma^*$ and every window $[i, j]$ of w, we have $w, [i, j] \vDash r \implies w, [i, j] \vDash s$. We prove that these two definitions are equivalent.

```
Lemma subset_leq : forall (r s : LRegex),
    regex_leq r s <->
    (forall (w : list A) (start delta : nat),
      match_regex r w start delta -> match_regex s w start delta).
```

We have shown that $\sqsubseteq$ is a partial order and the operations of union and concatenation are monotone with respect to this order. These monotonicity theorems can also be expressed using proper morphisms.

```
Instance regex_leq_partialorder : PartialOrder regex_eq regex_leq.
Instance union_monotone : Proper (regex_leq ==> regex_leq ==> regex_leq) Union.
Instance concat_monotone : Proper (regex_leq ==> regex_leq ==> regex_leq) Concat.
```

We have proven in Coq the following identities resembling the properties of a Kleene Algebra [31].

Lemma 3.4 (**Equivalences for Regular Expressions).** The following properties hold for all regular expressions $r, r_1, r_2, r_3 \in \mathsf{LReg}(\Sigma)$:

1. $(r_1 \cdot r_2) \cdot r_3 \equiv r_1 \cdot (r_2 \cdot r_3)$ and $\varepsilon \cdot r \equiv r \cdot \varepsilon \equiv r$
2. $(r_1 + r_2) + r_3 \equiv r_1 + (r_2 + r_3)$, $r_1 + r_2 \equiv r_2 + r_1$, and $r + \varnothing \equiv r$
3. $r_1 \cdot (r_2 + r_3) \equiv r_1 \cdot r_2 + r_1 \cdot r_3$, and $(r_1 + r_2) \cdot r_3 \equiv r_1 \cdot r_3 + r_2 \cdot r_3$
4. $\varnothing \cdot r \equiv r \cdot \varnothing \equiv \varnothing$
5. $\varepsilon + rr^* \sqsubseteq r^*$
6. $\varepsilon + r^*r \sqsubseteq r^*$
7. $r_1 \cdot r_2 \sqsubseteq r_2 \implies r_1^* \cdot r_2 \sqsubseteq r_2$
8. $r_2 \cdot r_1 \sqsubseteq r_2 \implies r_2 \cdot r_1^* \sqsubseteq r_2$

Proof. As a representative case, we consider property (5) and we leave the rest of the cases to the reader. It suffices to show that $\varepsilon \sqsubseteq r^*$ and $rr^* \sqsubseteq r^*$. We omit the proof of $\varepsilon \sqsubseteq r^*$. Let w be an arbitrary string and $[i,j]$ be a location in w. Suppose that $w,[i,j] \vDash rr^*$. It follows that there exists k such that $w,[i,k] \vDash r$ and $w,[k,j] \vDash r^*$. So, there is a decomposition S of $[k,j]$ such that $w,[i',j'] \vDash r$ for every location $[i',j']$ in S. Define $S' = [i,k] \cdot S$ and notice that S' is a decomposition of $[i,j]$ witnessing that $w,[i,j] \vDash r^*$. We have thus established that $rr^* \sqsubseteq r^*$. □

We have also proven several properties involving lookarounds.

Lemma 3.5 (Algebraic Properties of Lookaround). The following properties for lookahead assertions hold (and completely symmetric properties hold for lookbehind assertions):

1. Concatenation of lookarounds is commutative: $(?{>}\,r) \cdot (?{>}\,s) \equiv (?{>}\,s) \cdot (?{>}\,r)$.
2. Idempotence: $(?{>}\,r) \cdot (?{>}\,r) \equiv (?{>}\,r)$.
3. Kleene iteration over lookarounds is equivalent to ε: $(?{>}\,r)^* \equiv \varepsilon$.
4. Union distributes over lookarounds: $(?{>}\,r + s) \equiv (?{>}\,r) + (?{>}\,s)$.
5. Lookaheads can be flattened: $(?{>}\,(?{>}\,r) \cdot s) \equiv (?{>}\,r) \cdot (?{>}\,s)$.
6. Lookaheads can be flattened in the presence of wildcards: $(?{>}\,r \cdot (?{>}\,s) \cdot \Sigma^*) \equiv (?{>}\,r \cdot s)$.
7. The union of positive and negative lookaheads can be simplified: $(?{>}\,r) + (?{\not>}\,r) \equiv \varepsilon$.
8. Positive and negative lookaheads cannot be matched together: $(?{>}\,r) \cdot (?{\not>}\,r) \equiv \varnothing$
9. For predicates p_1 and p_2: $(?{>}\,p_1 r_1) p_2 r_2 \equiv (p_1 \cap p_2)(?{>}\,r_1) r_2$.

Proof. These properties can be proved in a straightforward manner using the formal semantics of lookaround expressions. To demonstrate, we prove the first one.

Suppose $w, [i, j] \vDash (?{>}\, r) \cdot (?{>}\, s)$. Then, there exists $i \le k \le j$ such that $w, [i, k] \vDash (?{>}\, r)$ and $w, [k, j] \vDash (?{>}\, s)$. By the semantics of lookahead, it must be that $i = k$ and $k = j$. Thus, we have $w, [i, i] \vDash (?{>}\, r)$ and $w, [i, i] \vDash (?{>}\, s)$. From the semantics of $\cdot$ we obtain that $w, [i, i] \vDash (?{>}\, s) \cdot (?{>}\, r)$. Since $i = j$, this is the same as $w, [i, j] \vDash (?{>}\, s) \cdot (?{>}\, r)$. This shows that $(?{>}\, r) \cdot (?{>}\, s)) \sqsubseteq (?{>}\, s) \cdot (?{>}\, r)$. By interchanging the roles of r and s, we can prove the other direction. □

The intuition for property (5) regarding the flattening of lookarounds is that both expressions describe the requirement that both r and s have a match at location $[i, |w|]$ (if we interpret them at position i).

Note that $(?{>}\, r \cdot (?{>}\, s))$ cannot be simplified to $(?{>}\, r \cdot s)$. For example, $(?{>}\, ab \cdot (?{>}\, cd))$ cannot be true at any position i because ab has to extend to the end of the string where $(?{>}\, cd)$ cannot hold. So, this regex cannot be equivalent to $(?{>}\, abcd)$.

Another caveat is the following: Suppose $u \in [\![r]\!]$ and $v \in [\![s]\!]$. We cannot, in general, expect that $u \cdot v \in [\![r \cdot s]\!]$. This is because lookaround expressions are able to "view" the entire string. As a concrete example, consider $r = (?{>}\, (aa)^*)$, $s = a^*$, $u = \varepsilon$, $v = a$.

The purpose of verifying these properties is to streamline reasoning about regular expressions with lookarounds into an equational manner when possible. We suggest two motivating examples where such reasoning could be used.

Example 3.6 (**Rewriting Anchors).** The equational reasoning framework can be used to reason about regular expressions in which the only lookarounds that occur are the start-of-string (`^`) and end-of-string anchors (`$`). In addition to the results proven above, we need to use the additional property $a \cdot \hat{} \equiv \$ \cdot a \equiv \emptyset$, for every $a \in \Sigma$.

Given an expression r whose only lookarounds are ^ and \$, one could use equational reasoning to rewrite it as $r_1 + \hat{}r_2 + r_3\$ + \hat{}r_4\$$ (where some of them maybe $\emptyset$). This could be proven by induction on the structure of r. For example, if r and s have already been rewritten in the above manner, we could use distributivity to split the following concatenation into 16 terms.

$$(r_1 + \hat{}r_2 + r_3\$ + \hat{}r_4\$) \cdot (s_1 + \hat{}s_2 + s_3\$ + \hat{}s_4\$)$$

Terms like $r_1 \cdot s_1$, $\hat{}r_2 s_1$, $r_1 s_3\$$, $\hat{}r_2 s_3\$$ do not need to be simplified further. However, terms such as $r_1\hat{}s_2$, $r_3\$\hat{}s_2$, etc can be simplified using the additional axioms. For example, we can write

$$r_1\hat{}s_2 \equiv r_1\hat{}\nu(s_2) \equiv r_1\nu(s_2)\hat{}$$

where the notation $\nu(r)$ denotes ε if $\varepsilon \in [\![r]\!]$ and $\emptyset$ otherwise.

Example 3.7 (**Eliminating Bounded Lookarounds**)**.** An expression r is bounded if the maximum length of a string $w \in [\![r]\!]$ is finite. For example, the regular expression $a + ab$ is bounded, but the expression a^* is not. If the expressions appearing inside lookarounds are bounded, then they could be eliminated. The key equation enabling this transformation is

$$(?{>}\,\sigma_1 r_1)\sigma_2 r_2 \equiv (\sigma_1 \cap \sigma_2)(?{>}\,r_1)r_2$$

, where σ_1, σ_2 are predicates. In PCRE notation, we write `[m-n]` to denote the predicate matching characters in the interval from m to n. Consider the regex (?> `[1-5]` `[2-6]`) `[5-7]` `[6-8]`. We observe that the intersection of `[1-5]` and `[5-7]` is `5`. Thus, we can apply the equation above to rewrite the regex as 5(?> `[2-6]`) `[6-8]`. We can use it again to rewrite the regex as `56`.

One could show that if r is without lookarounds, then there exists $\sigma_i \in \mathcal{P}$ such that $r \equiv \nu(r) + \sum \sigma_i \cdot r_i$, where the notation ν is as in the last example. This form 'exposes' the predicate at the front of each expression, and allows applying the above equation. This is guaranteed to terminate if the lookaround consists of a bounded expression.

3.5 Oracles for Lookaround Assertions

In this section, we will assume that relevant information about lookaround assertions can be accessed via oracle queries. We provide an automata-theoretic framework describing how such regular expressions with oracle queries can be handled. To do this, we first make the notion of strings augmented with oracle valuations explicit. An extension of regular expressions which can query these oracles is presented. The idea is to replace lookaround expressions with these oracle queries, so that information involving lookarounds can be answered using an oracle. These oracle-based expressions can be matched using a model that is an extension of NFAs. We show that these NFAs can be simulated using a purely functional algorithm. Similar to a usual NFA, this simulation is done in a single left-to-right pass, consuming the input word (including the oracle valuations) one character at a time, taking $O(m)$ time at each step, where m is the size of the regular expression.

3.5.1 Oracle Strings and Oracle Regular Expressions

Suppose V is a finite set of (Boolean) variables. A valuation of V is a truth assignment to the elements of V. We will denote the set of valuations of V as 2^V. An *o-string* is a pair $\langle w, \beta \rangle$ where $w \in \Sigma^*$ and $\beta \in (2^V)^*$ such that $|\beta| = |w| + 1$. The set of o-strings over the alphabet Σ and variables V is denoted $\mathcal{O}(\Sigma, V)$.

These represent strings together with additional information accessible via oracle queries. Suppose $w = a_0 \cdot a_1 \cdots a_{n-1}$ and $\beta = \beta_0 \cdot \beta_1 \cdot \beta_2 \cdots \beta_n$. Informally, we wish to associate each character a_i with the oracle valuation β_i to its left and β_{i+1} to its right. Thus, the o-string $\langle w, \beta \rangle$ could be more visually represented as

$$_{\beta_0}\, a_0 \, _{\beta_1}\, a_1 \, _{\beta_2}\, a_2 \, _{\beta_3} \cdots _{\beta_{n-1}}\, a_{n-1} \, _{\beta_n}.$$

This explains the constraints on the length of the components of o-strings. The length of the o-string $\langle w, \beta \rangle$ written $|\langle w, \beta \rangle|$ is defined to be $|w|$. Note that this means that there may be more than one o-string of length 0, since such an o-string consists of a single oracle valuation.

In Coq, we define o-strings simply as pairs of lists. The additional constraints on the lengths of the components are enforced using a separate predicate. We do this in order to avoid working with dependent types which can sometimes complicate our terms and proof scripts.

```
Definition valuation : Type := list bool.
Definition ostring : Type := (list A) * (list valuation).


Definition outer_length_wf (s : ostring) : Prop :=
  length (fst s) + 1 = length (snd s).
Definition inner_length_wf (s : ostring) : Prop :=
  forall u v : valuation,
    In u (snd s) -> In v (snd s) -> length u = length v.
Definition ostring_wf (s : ostring) : Prop :=
  outer_length_wf s /\ inner_length_wf s.


Definition olength (s : ostring) : nat := length (fst s).
```

When slicing o-strings, we must take into account the oracle valuations at the boundaries. Suppose we have an $\langle w, \beta \rangle \in \mathcal{O}(\Sigma, V)$ with $w = a_0 \cdots a_{n-1}$ and $\beta = \beta_0 \cdot \beta_1 \cdots \beta_n$. For $0 \leq i \leq j \leq n$, we define the *slice* of $\langle w, \beta \rangle$ at window $[i, j]$, denoted by $\langle w, \beta \rangle[i, j]$, as the o-string $(\overline{w}, \beta')$ where $\beta' = \beta_i \cdot \beta_{i+1} \cdots \beta_{j-1} \cdot \beta_j$ and $\overline{w} = a_i \cdot a_{i+1} \cdots a_{j-1}$. In particular, when $i = j$, $\overline{w}$ is the empty string ε and β' is the singleton string β_i.

Note that the slice $[i, j]$ can be obtained by taking the slice $[0, j]$ to obtain $\langle w', \beta' \rangle$ and then selecting the slice $[i, |w'|]$. This is how we encode the notion of slicing in our Coq formalization. We define the functions oskipn and ofirstn corresponding to the slices $[i, |w|]$ and $[0, i]$ respectively. These mirror the definition of the functions skipn and firstn in the Coq standard library, and satisfy analogous lemmas. Note the slight asymmetry in the definitions below, resulting from the mismatch in the lengths of the components of o-strings.

```
Definition ofirstn (n : nat) (s : ostring) : ostring :=
  (firstn n (fst s), firstn (S n) (snd s)).
Definition oskipn (n : nat) (s : ostring) : ostring :=
  (skipn n (fst s), skipn (min n (length (fst s))) (snd s)).
```

The concatenation operation on $\mathcal{O}(\Sigma, V)$ needs to be defined carefully so that it matches the way we intend to use these strings. The concatenation of two elements of $\mathcal{O}(\Sigma, V)$ is only defined if the oracle-values agree. Formally, suppose $\langle w_1, \beta \cdot u \rangle, \langle w_2, v \cdot \gamma \rangle \in \mathcal{O}(\Sigma, V)$ with $u, v \in 2^V$, then $\langle w_1, \beta \cdot u \rangle \cdot \langle w_2, v \cdot \gamma \rangle$ is defined iff $u = v$. The concatenation $\langle w_1, \beta \cdot u \rangle \cdot \langle w_2, u \cdot \gamma \rangle$ is defined to be $\langle w_1 w_2, \beta \cdot u \cdot \gamma \rangle$. We extend this definition in a natural way to concatenation of sets of o-strings. Kleene-iteration of an o-string (or a set of o-strings) is also defined in an analogous manner, respecting the agreement of oracle valuations at concatenation boundaries.

Definition 3.8 (**Oracle Regular Expressions).** The set OReg of *oracle regular expressions* is defined with the following grammar:

$$R, S ::= \varepsilon \mid \sigma \in \mathcal{P} \mid \mathsf{Q}^{+}(v \in V) \mid \mathsf{Q}^{-}(v \in V) \mid R \cdot S \mid R + S \mid R^{*}.$$

Instead of lookaheads as in LReg, oracle regular expressions have positive queries of the form $\mathsf{Q}^{+}(v)$ or negative queries of the form $\mathsf{Q}^{-}(v)$, where $v \in V$. These queries express assertions on the accompanying oracle valuations. The semantics of a OReg expression R is given in terms of $[\![R]\!]$, a subset of $\mathcal{O}(\Sigma, V)$. This language is defined inductively as follows:

$$\begin{aligned}
[\![\varepsilon]\!] &= \{\langle \varepsilon, \beta_0 \rangle \mid \beta_0 \in 2^V\} \\
[\![\sigma]\!] &= \{\langle a, \beta_0 \cdot \beta_1 \rangle \mid a \in \Sigma, \beta_0, \beta_1 \in 2^V, \sigma(a) = 1\} \\
[\![\mathsf{Q}^{+}(v)]\!] &= \{\langle \varepsilon, \beta_0 \rangle \mid \beta_0 \in 2^V, \beta_0[v] = 1\} \\
[\![\mathsf{Q}^{-}(v)]\!] &= \{\langle \varepsilon, \beta_0 \rangle \mid \beta_0 \in 2^V, \beta_0[v] = 0\} \\
[\![R + S]\!] &= [\![R]\!] \cup [\![S]\!] \\
[\![R \cdot S]\!] &= [\![R]\!] \cdot [\![S]\!] \\
[\![R^{*}]\!] &= [\![R]\!]^{*}
\end{aligned}$$

Example 3.9. Consider the oregex $r_1 = \mathsf{Q}^{+}(v_0) \cdot a^{+} b \cdot \mathsf{Q}^{+}(v_1)$. Here, the queries express the constraint that the valuation corresponding to the variable v_0 at the beginning of the string, and the valuation corresponding to the variable v_1 at the end of the string are both true. Concretely, we have $\langle w, \beta \rangle \in [\![r]\!]$ iff $w \in [\![a^{+}b]\!]$ and $\beta_0[v_0] = \beta_{|w|}[v_1] = 1$.

Consider the oregex $r_2 = c \cdot (\mathsf{Q}^{-}(v_2)\Sigma)^{+}$. Here, the negative query $\mathsf{Q}^{-}(v_2)$ asserts that the valuation corresponding to the variable v_2 be false before each character that matches the subexpression Σ. More concretely, suppose $\langle w, \beta \rangle \in [\![r_2]\!]$. Then, we have that $w_0 = c$ and $|w| > 1$ (enforced by the Kleene plus). For $1 \le i \le |w| - 1$, we must have $\beta_i[v_2] = 0$. Note that no constraints are placed on β_0 or $\beta_{|w|}$.

```
Inductive ORegex : Type :=
| OEpsilon : ORegex
| OCharClass : (A -> bool) -> ORegex
| OConcat : ORegex -> ORegex -> ORegex
| OUnion : ORegex -> ORegex -> ORegex
| OStar : ORegex -> ORegex
| OQueryPos : nat -> ORegex
| OQueryNeg : nat -> ORegex.
```

In our Coq formalization, we represent objects of type OReg as an inductive type, with constructors shown on the left. In order to represent membership in the set $[\![R]\!]$, we work with a satisfaction relation `match_oregex`. This is similar to our handling of LReg, except that the complication involving handling negation does not appear here. The case for concatenation and Kleene iteration, however, is interesting. This is because concatenation on o-strings is defined only when the oracle valuations agree. We sidestep this issue by phrasing the concatenation in terms of the slicing operation. In particular, we say that $\langle w, \beta \rangle \in [\![R \cdot S]\!]$ iff there exists an i such that $\langle w, \beta \rangle[0, i] \in [\![R]\!]$ and $\langle w, \beta \rangle[i, |w|] \in [\![S]\!]$. The Kleene iteration is handled in a similar manner. These two cases are shown below.

```
...
| omatch_concat : forall (r1 r2 : ORegex) (os : ostring) (n : nat),
    match_oregex r1 (ofirstn n os) -> match_oregex r2 (oskipn n os)
    -> match_oregex (OConcat r1 r2) os
...
| omatch_star_cons : forall (r : ORegex) (os : ostring) (n : nat),
    match_oregex r (ofirstn n os) -> match_oregex (OStar r) (oskipn n os)
    -> match_oregex (OStar r) os
...
```

3.5.2 Choosing appropriate oracle valuations

In this section, we will make concrete the connection between LReg and OReg. In particular, we will show that if the lookaround assertions are pre-evaluated, then they could be used as valuations for o-strings which could be supplied to an oracle regular expression obtained by replacing the lookarounds with queries.

Definition 3.10 (Maximal Lookarounds, Arity). Let $r \in \mathsf{LReg}$ be an expression. The maximal lookarounds of r, written $\mathsf{maxLk}(r)$ is a list of tuples of type $\{\triangleright, \triangleleft\} \times \mathsf{LReg}$. It is formally defined as follows:

$$\mathsf{maxLk}((?{>}\,r)) = [(\triangleleft, r)] \quad \mathsf{maxLk}((?{<}\,r)) = [(\triangleright, r)]$$

$$\mathsf{maxLk}((?{\ngtr}\,r)) = [(\triangleleft, r)] \quad \mathsf{maxLk}((?{\nless}\,r)) = [(\triangleright, r)]$$

$$\mathsf{maxLk}(r) = [\,],\ \text{if } r \in \{\varepsilon, (\sigma \in \mathcal{P})\}$$

$$\mathsf{maxLk}(r_1 \circ r_2) = \mathsf{maxLk}(r_1) \mathbin{+\!\!+} \mathsf{maxLk}(r_2),\ \text{if } \circ \in \{\cdot, +\}$$

$$\mathsf{maxLk}(r^*) = \mathsf{maxLk}(r)$$

where the notation ++ refers to list concatenation.

If $(\triangleleft, s)$ or $(\triangleright, s)$ appears in $\mathsf{maxLk}(r)$, then we say that s is a *maximal lookaround* of r. The *arity* of an expression r is the number of its maximal lookarounds, i.e., $\mathsf{arity}(r) = |\mathsf{maxLk}(r)|$.

Note that the directions associated with the maximal lookarounds are counter-intuitive. As we will see in subsection 3.7.1, lookbehinds need the scanning of the string from left to right, while lookaheads need the scanning from right to left.

Example 3.11. Consider the regular expression $r =$ `(?<=c((?!\d\d).)+)(a+)b(?=.*d.*e)` in PCRE notation. In our notation, we can write this as

$$(?{<}\,\Sigma^* c \cdot ((?{\ngtr}\ \texttt{\textbackslash d\textbackslash d}\,\Sigma^*)\Sigma)^+) \cdot a^+ b \cdot (?{>}\,\Sigma^* d \Sigma^* e \Sigma^*).$$

The maximal lookarounds of r are $\mathsf{maxLk}(r) = [m_0, m_1]$ where $m_0 = (\triangleright, \Sigma^* c\cdot((?\not> \texttt{\\d\\d}\Sigma^*)\Sigma)^+)$ and $m_1 = (\triangleleft, \Sigma^* d \Sigma^* e \Sigma^*)$. Therefore, we have $\mathsf{arity}(r) = 2$. Note that even though $(?\not> \texttt{\\d\\d}\Sigma^*)$ is a lookahead that appears as a subexpression, it is not considered *maximal*.

Definition 3.12 (Abstraction). The abstraction $\mathsf{abstract}(r)$ of a regular expression r is obtained by replacing each maximal lookaround with $\mathsf{Q}^+(v)$ or $\mathsf{Q}^-(v)$. The variables here are chosen from the set $V = \{v_i \mid i < |\mathsf{arity}(r)|\}$. Formally, $\mathsf{abstract}(r)$ is defined to be $\mathsf{abstract}_0(r)$, and given $i \in \mathbb{N}$, we define $\mathsf{abstract}_i(r)$ as follows:

$$\begin{aligned}
\mathsf{abstract}_i(r) &= r, \text{ if } r \in \{\varepsilon, (\sigma \in \mathcal{P})\} \\
\mathsf{abstract}_i(r_1 \circ r_2) &= \mathsf{abstract}_i(r_1) \circ \mathsf{abstract}_{i+\mathsf{arity}(r_1)}(r_2), \text{ if } \circ \in \{\cdot, +\} \\
\mathsf{abstract}_i(r^*) &= \mathsf{abstract}_i(r)^* \\
\mathsf{abstract}_i(r) &= \mathsf{Q}^+(v_i), \text{ if } r \in \{(?> s), (?< s)\} \\
\mathsf{abstract}_i(r) &= \mathsf{Q}^-(v_i), \text{ if } r \in \{(?\not> s), (?\not< s)\}
\end{aligned}$$

Example 3.13. Let us consider again the expression r from Example 3.11. The abstraction of r is given by $\mathsf{abstract}(r) = \mathsf{abstract}_0(r) = \mathsf{Q}^+(v_0) \cdot a^+ b \cdot \mathsf{Q}^+(v_1)$. If we consider the subexpression $s = \Sigma^* c \cdot ((?\not> \texttt{\\d\\d}\Sigma^*)\Sigma)^+$ of r, then we have $\mathsf{abstract}(s) = \Sigma^* c \cdot (\mathsf{Q}^-(v_0) \cdot \Sigma)^+$.

Next, we define the notion of *tapes* for expressions and lookarounds, which are sequences of truth values obtained by incrementally scanning the string.

Definition 3.14 (Tapes, Oracle Valuations). Let $r \in \mathsf{LReg}$ be an expression and $w \in \Sigma^*$ be a string such that $|w| = n$. We define the left-to-right tape $\mathsf{tape}(\triangleright, r, w)$ and the right-to-left tape $\mathsf{tape}(\triangleleft, r, w)$ as sequences of length $n+1$ over $\{0, 1\}$ such that for each $0 \le i \le n$, the i-th entry of the tape satisfies the following:

$$\mathsf{tape}(\triangleright, r, w)[i] = \begin{cases} 1 & \text{if } w,[0,i] \vDash r \\ 0 & \text{if } w,[0,i] \nvDash r \end{cases} \qquad \mathsf{tape}(\triangleleft, r, w)[i] = \begin{cases} 1 & \text{if } w,[i,|w|] \vDash r \\ 0 & \text{if } w,[i,|w|] \nvDash r \end{cases}$$

When there is no confusion, we will write tape to mean $\mathsf{tape}(\triangleright)$.

We also extend the notion of tapes to OReg and maximal lookarounds. For $s \in$ OReg with $\langle w, \beta\rangle \in \mathcal{O}(\Sigma, V)$, we define $\mathsf{tape}(s, \langle w, \beta\rangle)$ in a similar manner as above, indicating the truth values of the slices $\langle w, \beta\rangle[0,i]$ for $\mathsf{tape}(\triangleright)$ and $\langle w, \beta\rangle[i, |w|]$ for $\mathsf{tape}(\triangleleft)$. Given a maximal lookaround of the form $m = (d, r)$, where $d \in \{\triangleright, \triangleleft\}$ and $r \in \mathsf{LReg}$, we define $\mathsf{tape}((d,r))$ in a natural way as $\mathsf{tape}((\triangleright, r), w) = \mathsf{tape}(\triangleright, r, w)$ and $\mathsf{tape}((\triangleleft, r), w) = \mathsf{tape}(\triangleleft, r, w)$.

Let $r \in \mathsf{LReg}$ be of arity k. Let T be a collection of k tapes such that for each $0 \le i < k$, tape $T[i] = \mathsf{tape}(\mathsf{maxLk}(r)[i], w)$. Let $V = v_0, v_1, \dots v_{k-1}$. The oracle valuations for r on w, written $\mathsf{oval}(r, w)$ is the sequence of length $|w| + 1$ over 2^V obtained by transposing T. In other words,

$$\mathsf{oval}(r, w)[i][v_j] = T[j][i], \text{ for } 0 \le i \le |w|, 0 \le j < k.$$

Example 3.15. Consider r from Example 3.11. The maximal lookaround m_0 searches for the character c in the past followed by a non-empty sequence of characters satisfying certain conditions, while the maximal lookaround m_1 searches for an occurrence of the character d followed by the character e in the future. Below, we show the tapes for $\mathsf{tape}(m_0, w)$ and $\mathsf{tape}(m_1, w)$ for an illustrative string w. The oracle valuations $\mathsf{oval}(r, w)$ for r on w can be obtained by reading the table below in a column-wise manner.

w	c	a	b	c	7	7	d	c	a	a	b	7	d	a	b	e
$\mathsf{tape}(m_0, w)$	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$\mathsf{tape}(m_1, w)$	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0

In our Coq Formalization, we use the relation `is_lookaround_tape` to encode the notion of tape for maximal lookarounds, (i.e., pairs $\{\rhd, \lhd\} \times \mathsf{LReg}$). The definition of oval is instrumented using an additional index s which corresponds to the subscript $\mathsf{abstract}_s$. This is necessary in order to formulate a stronger inductive hypothesis we can induct on. Note also that we phrase this definition as a relation rather than a function.

```
Definition oval_tapes_aux (e : @LRegex A) (w : list A)
  (s : nat) (ts : list tape) : Prop :=
  length ts >= s + arity e
  /\ (forall t, In t ts -> length t = length w + 1)
  /\ forall i t r, i < arity e
    -> nth_error ts (s + i) = Some t
    -> nth_error (maximal_lookarounds e) i = Some r
    -> is_lookaround_tape r w t.

Definition is_oval_aux (r : @LRegex A) (w : list A)
  (s : nat) (vs : list valuation) : Prop :=
  exists ts, oval_tapes_aux r w s ts /\ vs = transpose (length w + 1) ts.

Definition is_oval (r : @LRegex A) (w : list A) (vs : list valuation) : Prop :=
  is_oval_aux r w 0 vs.
```

The following lemma establishes the connection between LReg and OReg by choosing the appropriate oracle valuations.

Lemma 3.16. Let $r \in \mathsf{LReg}$ and $w \in \Sigma^*$. Let $\beta = \mathsf{oval}(r, w)$. Then, the following holds.

$$w, [i, j] \vDash r \iff \langle w, \beta \rangle [i, j] \in [\![\mathsf{abstract}(r)]\!]$$

Thus, in order to match an expression r with lookarounds on w, we can first evaluate the lookarounds to obtain the oracle valuations $\beta = \mathsf{oval}(r, w)$, and then match the oracle regular expression $\mathsf{abstract}(r)$ on $\langle w, \beta \rangle$. This formalizes the key idea behind our approach.

In Coq, this lemma is stated in terms of the `ofirstn` and `oskipn` functions. Recall that the index s is used to keep track of the subscript of $\mathsf{abstract}_s$.

```
Lemma oracle_compose_aux (r : @LRegex A) (w : list A)
    (s : nat) (vs : list valuation) :
  is_oval_aux r w s vs
  -> forall start delta, start + delta <= length w
    -> match_regex r w start delta
    <-> match_oregex (abstractAux s r) (ofirstn delta (oskipn start (w, vs))).
```

3.6 Purely Functional Matching of Oracle Expressions

In this section, we discuss how the oracle regular expressions can be matched on oracle strings, i.e., the computation of tapes (from Definition 3.14 for the case of OReg. This is very similar to the matching algorithm for ordinary regular expressions (without queries or lookarounds), except that the queries need to be checked against the oracle valuations.

The main difference between handling character-predicates and queries can be understood by analyzing the case of concatenation. For example, consider the regular expression $\sigma_1 \cdot \sigma_2$ obtained by concatenating two predicates σ_1 and σ_2. This matches a length 2 string where the first character satisfies σ_1 and the second character satisfies σ_2. In contrast, the oracle regular expression $\mathsf{Q}^{+}(v_1) \cdot \mathsf{Q}^{+}(v_2)$ matches an o-string $\langle \varepsilon, \beta_0 \rangle$, where $\beta_0[v_1] = \beta_1[v_2] = 1$. Thus, the concatenated queries match *the same oracle valuation.* An alternative approach would be to use Oracle NFAs, in which the transitions are guarded by queries (see [28]).

We use a purely functional approach based on marked regular expressions. The notion of marked regular expressions for classical regular expressions is discussed in [33] and [32]. In [98], the authors formalize this approach in Isabelle, and utilize them in order to decide equivalence. Here, we extend this approach to oracle regular expressions, and formalize it in Coq.

The marked regular expression approach directly deals with the syntax tree of regular expressions instead of building an explicit NFA. This is particularly pleasant in functional programming languages where we can represent the expressions as inductive types and use pattern matching to access the subexpressions. The marks placed on the character classes of the regular expression are used to keep track of the active states of the corresponding NFA.

Definition 3.17 (Marked Oracle Regular Expressions). The set MReg of *marked oracle regular expressions* is defined with the following grammar:

$$M, N ::= \varepsilon \mid \sigma \mid \underline{\sigma} \mid \mathsf{Q}^{+}(v) \mid \mathsf{Q}^{-}(v) \mid M \cdot N \mid M + N \mid M^{*}$$

where $\sigma \in \mathcal{P}$ and $v \in V$. We call $\underline{\sigma}$ (respectively, σ) a *marked* (respectively, *unmarked*) predicate.

We extend the terminology above to say that an expression is *unmarked* if it contains no marked predicates. Given $r \in$ OReg, we can view it as an element of

MReg in which none of the predicates are marked. This expression is written as $\mathsf{toMarked}(r)$. Conversely, given $m \in \mathsf{MReg}$, we can remove all the marks from it to obtain the expression $\mathsf{strip}(m) \in \text{OReg}$. For each $m \in \mathsf{MReg}$, we define a subset $\langle\!\langle r \rangle\!\rangle$ of $\mathcal{O}(\Sigma, V)$ as follows:

$$\begin{aligned}
\langle\!\langle r \rangle\!\rangle &= \varnothing \text{ if } r \in \{\varepsilon, \sigma, \mathsf{Q}^{+}(v), \mathsf{Q}^{-}(v)\} \\
\langle\!\langle \underline{\sigma} \rangle\!\rangle &= [\![\sigma]\!] \\
\langle\!\langle e_1 \cup e_2 \rangle\!\rangle &= \langle\!\langle e_1 \rangle\!\rangle \cup \langle\!\langle e_2 \rangle\!\rangle \\
\langle\!\langle e_1 \cdot e_2 \rangle\!\rangle &= \langle\!\langle e_1 \rangle\!\rangle \cdot [\![\mathsf{strip}(e_2)]\!] \cup \langle\!\langle e_2 \rangle\!\rangle \\
\langle\!\langle e^{*} \rangle\!\rangle &= \langle\!\langle e \rangle\!\rangle \cdot [\![\mathsf{strip}(e)]\!]^{*}
\end{aligned}$$

Example 3.18. The intuition for the language of $\langle\!\langle r \rangle\!\rangle$ is that it represents the set of o-strings that could be matched by starting from the current mark. For example, consider the expressions $m_1 = \underline{a}bc$, $m_2 = a\underline{b}c$ and $m_3 = \underline{a}b\underline{c}$. Let $\langle w, \beta \rangle$ be an o-string. We have $\langle w, \beta \rangle \in \langle\!\langle m_1 \rangle\!\rangle$ iff $w = abc$, $\langle w, \beta \rangle \in \langle\!\langle m_2 \rangle\!\rangle$ iff $w = bc$ and $\langle w, \beta \rangle \in \langle\!\langle m_3 \rangle\!\rangle$ iff $w = abc$ or $w = c$. On the other hand, if $m_4 = abc$ with no marks, then $\langle\!\langle m_4 \rangle\!\rangle = \varnothing$. This can be understood by realizing that this represents a scenario where there are no active states in the simulated NFA. In general, we can observe that for unmarked m, the language $\langle\!\langle m \rangle\!\rangle$ is $\varnothing$.

The queries in expressions in MReg do not carry any marks, but they can still express constraints on valuations if they occur after a marked predicate. For instance, consider $m_5 = \underline{a}\mathsf{Q}^{+}(v)bc$ and $m_6 = \mathsf{Q}^{+}(v)\underline{a}bc$. Given $\langle w, \beta \rangle \in \langle\!\langle m_5 \rangle\!\rangle$, we must have $w = abc$ and $\beta_1[v] = 1$. However, for $\langle w, \beta \rangle \in \langle\!\langle m_6 \rangle\!\rangle$, we have no constraints on β.

In Coq, we define an inductive type `MRegex` to represent the set MReg. Similar to the case of LReg and OReg, we represent the notion of $\langle\!\langle \cdot \rangle\!\rangle$ using a satisfaction relation encoded as the inductive type `match_mregex`.

3.6.1 Operations on Marked Expressions

Our matching algorithm operates by transforming marked regular expressions. The functions $\mathsf{nullable}$ and final are used to extract information about the state machine that is being simulated. The functions follow and read presented in Figure 3.2 manipulate the marks without changing the underlying regex.

Let us start by characterizing the functions $\mathsf{nullable}$ and final. In the following, we call a marked predicate $\underline{\sigma}$ *spurious* if the predicate σ is unsatisfiable. If m has no such predicates, we say m has no spurious marks.

Lemma 3.19. Let $m \in \mathsf{MReg}$. Then, the following holds.

1. For any $\beta_0 \in 2^V$, $\mathsf{nullable}(\beta_0, m) = 1$ iff $\langle \varepsilon, \beta_0 \rangle \in [\![\mathsf{strip}(m)]\!]$
2. Suppose m has no spurious marks and $\beta_1 \in 2^V$. If $\mathsf{final}(\beta_1, m) = 1$, then there exists $a \in \Sigma$ such that for all $\beta_0 \in 2^V$, $\langle a, \beta_0\beta_1 \rangle \in \langle\!\langle m \rangle\!\rangle$
3. Let $a \in \Sigma$ and $\beta_0, \beta_1 \in 2^V$. If $\langle a, \beta_0\beta_1 \rangle \in \langle\!\langle m \rangle\!\rangle$, then $\mathsf{final}(\beta_1, m) = 1$

Example 3.20. Consider the expressions $m_6 = (a + \varepsilon)\mathsf{Q}^{+}(v)$, and $m_7 = (\underline{a} + \varepsilon)\mathsf{Q}^{+}(v)$. Since $\mathsf{strip}(m_6) = \mathsf{strip}(m_7)$, we can see that $\mathsf{nullable}(\beta_0, m_6) = \mathsf{nullable}(\beta_0, m_7)$ for any choice of β_0. In particular, $\mathsf{nullable}(\beta_0, m_6) = 1$ iff $\beta_0[v] = 1$. Here, the function nullable tests whether the o-string $\langle \varepsilon, \beta_0 \rangle$ is in $[\![\mathsf{strip}(m_6)]\!]$.

Informally, the function final checks if there is a mark on a predicate that is in the final position, and whether the subsequent queries can be satisfied by the valuation provided. Thus, if $\beta_1[v] = 1$, we do have $\mathsf{final}(\beta_1, m_7) = 1$. For m_6, we have $\mathsf{final}(\beta_0, m_6) = 0$ for any β_0 since there are no marks in m_6. Similarly, $\mathsf{final}(\beta_0, (\underline{a}bc + \varepsilon)\mathsf{Q}^{+}(v)) = 0$, since the marked character a is not in a final position. Also, $\mathsf{final}(\beta_0, (\underline{a} + \varepsilon)) = 1$ for any β_0 since there are no queries imposing constraints on the valuations.

$$\mathsf{nullable} : 2^V \times \mathsf{MReg} \to \mathbb{B}$$

$$\mathsf{nullable}(\beta, \varepsilon) = 1 \qquad \mathsf{nullable}(\beta, m^*) = 1$$

$$\mathsf{nullable}(\beta, \sigma) = 0 \qquad \mathsf{nullable}(\beta, \underline{\sigma}) = 0$$

$$\mathsf{nullable}(\beta, \mathsf{Q}^+(v)) = \beta[v] \quad \mathsf{nullable}(\beta, \mathsf{Q}^-(v)) = \neg\beta[v]$$

$$\mathsf{nullable}(\beta, m_1 + m_2) = \mathsf{nullable}(\beta, m_1) \vee \mathsf{nullable}(\beta, m_2)$$

$$\mathsf{nullable}(\beta, m_1 \cdot m_2) = \mathsf{nullable}(\beta, m_1) \wedge \mathsf{nullable}(\beta, m_2)$$

$$\mathsf{final} : 2^V \times \mathsf{MReg} \to \mathbb{B}$$

$$\mathsf{final}(\beta, \varepsilon) = 0 \qquad \mathsf{final}(\beta, m^*) = \mathsf{final}(\beta, m)$$

$$\mathsf{final}(\beta, \sigma) = 0 \qquad \mathsf{final}(\beta, \underline{\sigma}) = 1$$

$$\mathsf{final}(\beta, \mathsf{Q}^+(v)) = 0 \quad \mathsf{final}(\beta, \mathsf{Q}^-(v)) = 0$$

$$\mathsf{final}(\beta, m_1 + m_2) = \mathsf{final}(\beta, m_1) \vee \mathsf{final}(\beta, m_2)$$

$$\mathsf{final}(\beta, m_1 \cdot m_2) = (\mathsf{final}(\beta, m_1) \wedge \mathsf{nullable}(\beta, m_2)) \vee \mathsf{final}(\beta, m_2)$$

$$\mathsf{follow} : \mathbb{B} \times 2^V \times \mathsf{MReg} \to \mathsf{MReg}$$

$$\mathsf{follow}(\beta, \varepsilon) = \varepsilon$$

$$\mathsf{follow}(\beta, \mathsf{Q}^+(v)) = \mathsf{Q}^+(v) \quad \mathsf{follow}(\beta, \mathsf{Q}^-(v)) = \mathsf{Q}^-(v)$$

$$\mathsf{follow}(b, \beta, \sigma) = \mathsf{follow}(b, \beta, \underline{\sigma}) = \begin{cases} \underline{\sigma} & \text{if } b = 1 \\ \sigma & \text{otherwise} \end{cases}$$

$$\mathsf{follow}(b, \beta, m_1 + m_2) = \mathsf{follow}(b, \beta, m_1) + \mathsf{follow}(b, \beta, m_2)$$

$$\mathsf{follow}(b, \beta, m_1 \cdot m_2) = \mathsf{follow}(b, \beta, m_1) \cdot \mathsf{follow}(b', \beta, m_2)$$

$$\text{where } b' = \mathsf{final}(\beta, m_1) \vee (b \wedge \mathsf{nullable}(\beta, m_1))$$

$$\mathsf{follow}(b, \beta, m^*) = (\mathsf{follow}(b \vee \mathsf{final}(\beta, m), \beta, m))^*$$

$$\mathsf{read} : \Sigma \times \mathsf{MReg} \to \mathsf{MReg}$$

$$\mathsf{read}(a, \varepsilon) = \varepsilon$$

$$\mathsf{read}(a, \mathsf{Q}^+(v)) = \mathsf{Q}^+(v) \quad \mathsf{read}(a, \mathsf{Q}^-(v)) = \mathsf{Q}^-(v)$$

$$\mathsf{read}(a, \sigma) = \sigma \quad \mathsf{read}(a, \underline{\sigma}) = \begin{cases} \underline{\sigma} & \text{if } \sigma(a) = 1 \\ \sigma & \text{otherwise} \end{cases}$$

$$\mathsf{read}(a, m_1 + m_2) = \mathsf{read}(a, m_1) + \mathsf{read}(a, m_2)$$

$$\mathsf{read}(a, m_1 \cdot m_2) = \mathsf{read}(a, m_1) \cdot \mathsf{read}(a, m_2)$$

$$\mathsf{read}(a, m^*) = \mathsf{read}(a, m)^*$$

Figure 3.2 : Operations on MReg expressions

Next, we characterize the function read. The function $\mathsf{read}(a, \cdot)$ has the effect of removing the marks from marked predicates $\underline{\sigma}$ that do not satisfy $\sigma(a) = 1$. Roughly, this has the effect of removing from $\langle\!\langle m \rangle\!\rangle$ the o-strings that do not start with the character a.

Lemma 3.21. Let $m \in \mathsf{MReg}$ and $a \in \Sigma$. Then, the following holds.

1. $\langle\!\langle \mathsf{read}(a, m) \rangle\!\rangle \subseteq \langle\!\langle m \rangle\!\rangle$
2. Suppose $\langle a \cdot w, \beta \rangle \in \langle\!\langle m \rangle\!\rangle$. Then, $\langle a \cdot w, \beta \rangle \in \langle\!\langle \mathsf{read}(a, m) \rangle\!\rangle$
3. Suppose $a' \in \Sigma$ and $\langle a' \cdot w, \beta \rangle \in \langle\!\langle \mathsf{read}(a, m) \rangle\!\rangle$. Then, $\langle a \cdot w, \beta \rangle \in \langle\!\langle m \rangle\!\rangle$
4. The expression $\mathsf{read}(a, m)$ has no spurious marks

Example 3.22. Let us use the predicate `\d` to denote the set of digits. Consider the expression $m = \underline{\texttt{\textbackslash d}}abc$. We have $\mathsf{read}(7, m) = m$ since the digit 7 satisfies the predicate `\d`. On the other hand, $\mathsf{read}(a, m) = \texttt{\textbackslash d}abc$ – the mark is lost since a does not satisfy the predicate.

In order to prove the properties of the function follow, it is helpful to define two other functions init and shift, whose definitions we have provided in Figure 3.3. These two functions are related in the following way:

$$\mathsf{follow}(0, \beta_0, m) = \mathsf{shift}(\beta_0, m)$$

$$\mathsf{follow}(1, \beta_0, m) = \mathsf{init}(\beta_0, \mathsf{shift}(\beta_0, m))$$

These equivalences can be proven using a straightforward induction once the idempotence of the function init is shown. Informally, the function init is used to place marks at initial positions in the expression, and the function shift is used to move the marks to the next positions. This intuition is formalized in the following lemmas.

$$
\begin{aligned}
&\mathsf{init} : 2^V \times \mathsf{MReg} \to \mathsf{MReg}\\
\mathsf{init}(\beta, r) &= r, \text{ if } r \in \{\varepsilon, \mathsf{Q}^{+}(v), \mathsf{Q}^{-}(v)\}\\
\mathsf{init}(\beta, \sigma) &= \underline{\sigma}\\
\mathsf{init}(\beta, \underline{\sigma}) &= \underline{\sigma}\\
\mathsf{init}(\beta, m_1 + m_2) &= \mathsf{init}(\beta, m_1) + \mathsf{init}(\beta, m_2)\\
\mathsf{init}(\beta, m^*) &= \mathsf{init}(\beta, m)^*\\
\mathsf{init}(\beta, m_1 \cdot m_2) &= \mathsf{init}(\beta, m_1) \cdot m_2' \text{ ,where}\\
m_2' &= \begin{cases} \mathsf{init}(\beta, m_2) & \text{if } \mathsf{nullable}(\beta, m_1) = 1\\ m_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{shift} : 2^V \times \mathsf{MReg} \to \mathsf{MReg}\\
\mathsf{shift}(\beta, r) &= r, \text{ if } r \in \{\varepsilon, \mathsf{Q}^{+}(v), \mathsf{Q}^{-}(v)\}\\
\mathsf{shift}(\beta, \sigma) &= \sigma\\
\mathsf{shift}(\beta, \underline{\sigma}) &= \sigma\\
\mathsf{shift}(\beta, m_1 + m_2) &= \mathsf{shift}(\beta, m_1) + \mathsf{shift}(\beta, m_2)\\
\mathsf{shift}(\beta, m_1 \cdot m_2) &= \mathsf{shift}(\beta, m_1) \cdot m_2'', \text{ where}\\
m_2' &= \mathsf{shift}(\beta, m_2), \text{ and}\\
m_2'' &= \begin{cases} \mathsf{init}(\beta, m_2') & \text{if } \mathsf{final}(\beta, m_1) = 1\\ m_2' & \text{otherwise} \end{cases}\\
\mathsf{shift}(\beta, m^*) &= (m'')^*, \text{ where}\\
m' &= \mathsf{shift}(\beta, m), \text{ and}\\
m'' &= \begin{cases} \mathsf{init}(\beta, m') & \text{if } \mathsf{final}(\beta, m) = 1\\ m' & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.3 : The init and shift functions for MReg

Lemma 3.23. Let $m \in \mathsf{MReg}$ and $\beta_0, \beta_* \in 2^V$, $\beta \in (2^V)^*$ and $w \in \Sigma^*$. Then, the function init satisfies the following properties.

1. $\langle\!\langle m \rangle\!\rangle \subseteq \langle\!\langle \mathsf{init}(\beta_0, m) \rangle\!\rangle$
2. Suppose $w \neq \varepsilon$. Suppose $\langle w, \beta_0 \cdot \beta \rangle \in [\![\mathsf{strip}(m)]\!]$. Then, $\langle w, \beta_0 \cdot \beta \rangle \in \langle\!\langle \mathsf{init}(\beta_0, m) \rangle\!\rangle$.
3. Suppose $\langle w, \beta_* \cdot \beta \rangle \in \langle\!\langle \mathsf{init}(\beta_0, m) \rangle\!\rangle$. Then either $\langle w, \beta_* \cdot \beta \rangle \in \langle\!\langle m \rangle\!\rangle$ or $\langle w, \beta_0 \cdot \beta \rangle \in [\![\mathsf{strip}(m)]\!]$.

Next, let $\beta_1 \in 2^V$ and $a_0 \in \Sigma$. Then, the function shift satisfies the following properties.

1. Suppose $w \neq \varepsilon$ and $\langle a_0 \cdot w, \beta_0\beta_1 \cdot \beta \rangle \in \langle\!\langle m \rangle\!\rangle$. Then, $\langle w, \beta_1 \cdot \beta \rangle \in \langle\!\langle \mathsf{shift}(\beta_1, m) \rangle\!\rangle$.
2. Suppose $\langle w, \beta_* \cdot \beta \rangle \in \langle\!\langle \mathsf{shift}(\beta_1, m) \rangle\!\rangle$. Then, there exists some $a_0 \in \Sigma$ such that forall $\beta_0 \in 2^V$, $\langle a_0 \cdot w, \beta_0\beta_1 \cdot \beta \rangle \in \langle\!\langle m \rangle\!\rangle$.

Example 3.24. We have $\mathsf{init}(\beta_0, abc + de) = \underline{a}bc + \underline{d}e$ for all β_0. Shifting would move each mark to the next position, i.e., $\mathsf{shift}(\beta_0, \underline{a}bc + \underline{d}e) = a\underline{b}c + d\underline{e}$. When there are queries guarding the expression, the supplied valuation must satisfy the query in order to place the mark. For example, $\mathsf{init}(\beta_0, \mathsf{Q}^+(v)abc) = \mathsf{Q}^+(v)\underline{a}bc$ only $\beta_0[v] = 1$. Similarly, $\mathsf{shift}(\beta_0, \underline{a}\mathsf{Q}^+(v)bc) = a\mathsf{Q}^+(v)\underline{b}c$ only if $\beta_0[v] = 1$.

3.6.2 Caching **final** and **nullable** for Marked Expressions

We wish to compute $\mathsf{follow}(m)$ in $O(|m|)$ time. However, translating the definitions from Figure 3.2 will result in an algorithm that would take $O(|m|^2)$ time in the worst case. This is because the functions final and $\mathsf{nullable}$ are called repeatedly on the same subexpressions. This can be avoided by augmenting the type of expressions to store these additional results. We call these expressions *marked expressions with caching*, denoted by CMReg. In Coq, we define the mutually inductive types `CMRegex` and `CMRe` as follows. Note that with this definition, the functions cFinal and $\mathsf{cNullable}$ are simply

```
Inductive CMRegex : Type :=
  | MkCMRegex : bool -> bool -> CMRe -> CMRegex
with CMRe : Type :=
  | CMEpsilon : CMRe
  | CMCharClass : (A -> bool) -> CMRe
  | CMQueryPos : nat -> CMRe
  | CMQueryNeg : nat -> CMRe
  | CMConcat : CMRegex -> CMRegex -> CMRe
  | CMUnion : CMRegex -> CMRegex -> CMRe
  | CMStar : CMRegex -> CMRe.

Definition cNullable (r : CMRegex) : bool :=
  match r with | MkCMRegex b _ _ => b end.
Definition cFinal (r : CMRegex) : bool :=
  match r with | MkCMRegex _ b _ => b end.
Definition cRe (r : CMRegex) : CMRe :=
  match r with | MkCMRegex _ _ re => re end.
```

field accessors which run in $O(1)$ time, which store the results of final and nullable respectively.

An interesting consideration here is the induction principle that allows us to prove properties involving `CMRegex` and `CMRe` which are mutually recursive. The auto-generated induction principle is insufficient since the subexpressions of `CMRe` (respectively, `CMRegex`) are nested inside an additional layer of `CMRegex` (respectively, `CMRe`). We write our custom induction principle in the term language, which is later used to prove properties.

Given $c \in \mathsf{CMReg}$, we write $\mathsf{unCache}(c) \in \mathsf{MReg}$ to denote the underlying marked regular expression. We say that c is *synced* to a valuation $\beta_0 \in V$ if $\mathsf{cNullable}(c) = \mathsf{nullable}(\beta_0, \mathsf{unCache}(c))$ and $\mathsf{cFinal}(c) = \mathsf{final}(\beta_0, \mathsf{unCache}(c))$. In order to streamline our functions and proofs, we define smart constructors mkEps, mkCharClass, mkQPos, mkQNeg, mkConcat, mkUnion and mkStar which propagate the caching information. These have the property that if the supplied arguments are synced with respect to some valuation, then so is the resulting expression. The explicit definition and property for mkConcat are shown. Compare this with the cases of concatenation in the definition of nullable and final in Figure 3.2.

```
Definition mkConcat (r1 r2 : CMRegex) : CMRegex := MkCMRegex
    (cNullable r1 && cNullable r2)
    ((cFinal r1 && cNullable r2) || cFinal r2)
    (CMConcat r1 r2).
```

We manipulate the cached expressions using the functions sync, cFollow and cRead as defined in Figure 3.4. Their relationship with the corresponding functions for MReg is summarized in the following lemmas.

$$
\begin{aligned}
\mathsf{sync} &: 2^V \times \mathsf{CMReg} \to \mathsf{CMReg}\\
\mathsf{sync}(\beta,\varepsilon) &= \mathsf{mkEps}\\
\mathsf{sync}(\beta,\sigma) &= \mathsf{mkCharClass}(0,\sigma)\\
\mathsf{sync}(\beta,\underline{\sigma}) &= \mathsf{mkCharClass}(1,\sigma)\\
\mathsf{sync}(\beta,\mathsf{Q}^{+}(v)) &= \mathsf{mkQPos}(\beta[v],v)\\
\mathsf{sync}(\beta,\mathsf{Q}^{-}(v)) &= \mathsf{mkQNeg}(\beta[v],v)\\
\mathsf{sync}(\beta,m_1+m_2) &= \mathsf{mkUnion}(\mathsf{sync}(\beta,m_1),\mathsf{sync}(\beta,m_2))\\
\mathsf{sync}(\beta,m_1\cdot m_2) &= \mathsf{mkConcat}(\mathsf{sync}(\beta,m_1),\mathsf{sync}(\beta,m_2))\\
\mathsf{sync}(\beta,m^*) &= \mathsf{mkStar}(\mathsf{sync}(\beta,m))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{cRead} &: \Sigma \times \mathsf{CMReg} \to \mathsf{CMReg}\\
\mathsf{cRead}(a,\varepsilon) &= \mathsf{mkEps}\\
\mathsf{cRead}(a,\sigma) &= \mathsf{mkCharClass}(0,\sigma)\\
\mathsf{cRead}(a,\underline{\sigma}) &= \begin{cases}\mathsf{mkCharClass}(1,\sigma) & \text{if } \sigma(a)=1\\ \mathsf{mkCharClass}(0,\sigma) & \text{otherwise}\end{cases}\\
\mathsf{cRead}(a,\mathsf{Q}^{+}(v)) &= \mathsf{mkQPos}(\mathsf{cNullable}(\mathsf{Q}^{+}(v)),v)\\
\mathsf{cRead}(a,\mathsf{Q}^{-}(v)) &= \mathsf{mkQNeg}(\mathsf{cNullable}(\mathsf{Q}^{-}(v)),v)\\
\mathsf{cRead}(a,m_1+m_2) &=\\
&\qquad \mathsf{mkUnion}(\mathsf{cRead}(a,m_1),\mathsf{cRead}(a,m_2))\\
\mathsf{cRead}(a,m_1\cdot m_2) &=\\
&\qquad \mathsf{mkConcat}(\mathsf{cRead}(a,m_1),\mathsf{cRead}(a,m_2))\\
\mathsf{cRead}(a,m^*) &= \mathsf{mkStar}(\mathsf{cRead}(a,m))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{cFollow} &: \mathbb{B} \times \mathsf{CMReg} \to \mathsf{CMReg}\\
\mathsf{cFollow}(b,\varepsilon) &= \mathsf{mkEps}\\
\mathsf{cFollow}(b,\sigma) &= \mathsf{cFollow}(b,\underline{\sigma}) = \mathsf{mkCharClass}(b,\sigma)\\
\mathsf{cFollow}(b,\mathsf{Q}^{+}(v)) &= \mathsf{mkQPos}(\mathsf{cNullable}(\mathsf{Q}^{+}(v)),v)\\
\mathsf{cFollow}(b,\mathsf{Q}^{-}(v)) &= \mathsf{mkQNeg}(\mathsf{cNullable}(\mathsf{Q}^{-}(v)),v)\\
\mathsf{cFollow}(b,m_1+m_2) &= \mathsf{mkUnion}(\mathsf{cFollow}(b,m_1),\mathsf{cFollow}(b,m_2))\\
\mathsf{cFollow}(b,m_1\cdot m_2) &= \mathsf{mkConcat}(\mathsf{cFollow}(b,m_1),\mathsf{cFollow}(b',m_2))\\
\text{where } b' &= \mathsf{cFinal}(m_1) \vee (b \wedge \mathsf{cNullable}(m_1))\\
\mathsf{cFollow}(b,m^*) &= \mathsf{mkStar}(\mathsf{cFollow}(b \vee \mathsf{cFinal}(m),m))
\end{aligned}
$$

Figure 3.4 : Operations on CMReg expressions

Lemma 3.25. Let $m \in \mathsf{MReg}$ and $\beta_0 \in 2^V$. Then, the following holds.

1. The expression $\mathsf{sync}(\beta_0, m)$ is synced with respect to β_0 and it preserves the underlying marked expression, i.e., $\mathsf{unCache}(\mathsf{sync}(\beta_0, m)) = m$.

2. If m is synced with respect to β_0, then so is $\mathsf{cRead}(\beta_0, m)$.

3. The function cRead simulates read on the underlying MReg expression, i.e., $\mathsf{unCache}(\mathsf{cRead}(a, m)) = \mathsf{read}(a, \mathsf{unCache}(m))$

4. Suppose m is synced to β_0. Then, the function $\mathsf{cFollow}$ simulates follow on the underlying MReg expression, i.e.,

$$\mathsf{unCache}(\mathsf{cFollow}(b, m)) = \mathsf{follow}(b, \beta_0, \mathsf{unCache}(m)).$$

These lemmas establish the correspondence between the operations on MReg and their cached version CMReg.

3.6.3 Consuming Oracle Strings

We define the function $\mathsf{consume}$, whose effect is to simulate moving tokens in the corresponding NFA whose paths are labelled by the supplied string. We write $\mathsf{toCached} : \mathsf{MReg} \to \mathsf{CMReg}$ to denote the function which initializes the caching information to 0 for a given expression in MReg.

Definition 3.26 (Consuming OStrings). Let $r \in \mathrm{OReg}$, and $w = a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1} \in \Sigma^*$, and $\beta = \beta_0 \cdot \beta_1 \cdot \ldots \cdot \beta_n \in (2^V)^*$. We define the series of expressions $m_0, m_1 \ldots m_n \in$

CMReg as follows.

$$m_0 = \mathsf{sync}(\beta_0, \mathsf{toCached}(\mathsf{toMarked}(r)))$$

$$m_1 = \mathsf{sync}(\beta_1, \mathsf{cRead}(a_0, \mathsf{cFollow}(1, m_0)))$$

$$m_{i+1} = \mathsf{sync}(\beta_{i+1}, \mathsf{cRead}(a_i, \mathsf{cFollow}(0, m_i)))$$

We define $\mathsf{consume}(r, \langle w, \beta\rangle) = m_n$.

Note that in the above definition, we apply $\mathsf{cFollow}$ with 1 in order to obtain m_1 but with 0 for the subsequent expressions. This is because applying $\mathsf{cFollow}$ with 1 has the effect of placing the marks at the initial positions, as explained in subsection 3.6.1. For the subsequent expressions, we use 0 since we only need to move the marks ahead.

The function oMatch defined below computes, in a streaming manner, the tape obtained by scanning the input oracle-string from left-to-right.

Definition 3.27 (oMatch on OStrings)**.** Let $r \in$ OReg, and $w = a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1} \in \Sigma^*$, and $\beta = \beta_0 \cdot \beta_1 \cdot \ldots \cdot \beta_n \in (2^V)^*$. Define $b_0 = \mathsf{cNullable}(\mathsf{consume}(r, \langle \varepsilon, \beta_0\rangle))$, $m_1 = \mathsf{consume}(r, \langle a_0, \beta_0 \cdot \beta_1\rangle)$ and $b_1 = \mathsf{cFinal}(m_1)$. Define $\mathsf{oMatch}(\langle w, \beta\rangle) \in \mathbb{B}^{n+1}$ as

$$\begin{aligned}
&\mathsf{oMatch}(\langle w, \beta\rangle) = [b_0] && \text{if } |w| = 0\\
&\mathsf{oMatch}(\langle w, \beta\rangle) = [b_0, b_1] && \text{if } |w| = 1\\
&\mathsf{oMatch}(\langle w, \beta\rangle) = [b_0, b_1] \mathbin{+\!\!+} \mathsf{oMatch}'(m_1, a_1 \cdot a_2 \cdots, \beta_2 \cdot \beta_3 \cdots) && \text{if } |w| > 1
\end{aligned}$$

where oMatch' is defined as follows.

$$\mathsf{oMatch}'(m, [], []) = [\mathsf{cFinal}(m)]$$

$$\mathsf{oMatch}'(m, a \cdot w, \beta_* \cdot \beta) = [\mathsf{cFinal}(m)] \mathbin{+\!\!+} \mathsf{oMatch}'(m', w, \beta)$$

$$\text{where } m' = \mathsf{sync}(\beta_*, \mathsf{cRead}(a, \mathsf{cFollow}(0, m)))$$

w_i		a		a		b	
$\beta_i[v]$	1		1		0		0

m'_i	$Q\underline{a}a^*bQ$		$Qa(\underline{a})^*\underline{b}Q$		$Qa(\underline{a})^*\underline{b}Q$		Qaa^*bQ
m_{i+1}		$Q\underline{a}a^*bQ$		$Qa(\underline{a})^*bQ$		$Qaa^*\underline{b}Q$	
b_i	0		0		0		0

Table 3.1 : Matching the ORegex $\mathsf{Q}^+(v)a \cdot a^* \cdot b\mathsf{Q}^+(v)$ on the ostring $\langle aab, 1100\rangle$ using Marked Regular Expressions

Theorem 3.28 (**Correctness of oMatch**)**.** Let $r \in$ OReg, and $\langle w, \beta\rangle \in \mathcal{O}(\Sigma, V)$. Then, the list $\mathsf{oMatch}(\langle w, \beta\rangle)$ records whether $\langle w, \beta\rangle[0, i] \in [\![r]\!]$, i.e.,

$$\mathsf{oMatch}(r, \langle w, \beta\rangle) = \mathsf{tape}(r, \langle w, \beta\rangle).$$

Proof. We can establish by induction that $b_0 = \mathsf{cNullable}(consume(r, \langle w, \beta\rangle[0, 0]))$ and $b_{i+1} = \mathsf{cFinal}(consume(r, \langle w, \beta\rangle[0, i+1]))$. The other key facts needed are the following: (1) If $|w| = 0$, then $\langle w, \beta\rangle \in [\![r]\!]$ iff $\mathsf{cNullable}(m) = 1$, and (2) If $|w| > 0$, then $\langle w, \beta\rangle \in [\![r]\!]$ iff $\mathsf{cFinal}(m) = 1$. These would follow from the properties of $\mathsf{nullable}$ and final (Lemma 3.19), and the fact that $\mathsf{cFollow}$ and cRead simulate the functions $\mathsf{cFollow}$ and cRead from subsection 3.6.1 (Lemma 3.25). $\square$

Example 3.29. In order to illustrate the working of oMatch, we choose the expression $r = \mathsf{Q}^+(v)a \cdot a^* \cdot b\mathsf{Q}^+(v)$ and use the following set of equations which illustrates every step of the computation.

$$m_0 = r \qquad m'_0 = \mathsf{follow}(1, \beta_0, m_0) \qquad b_0 = \mathsf{nullable}(\beta_0, m_0)$$

$$m_{i+1} = \mathsf{read}(w_i, m'_i) \qquad m'_{i+1} = \mathsf{follow}(0, \beta_{i+1}, m_{i+1}) \qquad b_{i+1} = \mathsf{final}(\beta_{i+1}, m_{i+1})$$

We will choose $w = aab$ and the valuations of v to be 1100. In Table 3.1, we show the values of m_i, m'_i and b_i at each step. We abbreviate $\mathsf{Q}^{+}(v)$ as Q, for the sake of brevity.

To form m'_0, we need the valuation of $\beta_0[v]$ in order to check whether the query $\mathsf{Q}^{+}(v)$ is satisfied. Each m'_{i+1} is obtained by shifting the marks to the next positions in the expression. For example, notice that m'_1 has two marks. This is because the expression a^* is nullable, and the subsequent character could have been either an a or a b. Each m_{i+1} is obtained by removing the marks on the predicates of m'_i which do not satisfy the character w_i. For example, we see that the mark on the predicate b present in m'_1 is removed in m_2. This is because the character $w_2 = a$ does not satisfy the predicate b. The mark on b in m_3 is lost when shifting to form m'_3. This is because there are no subsequent characters left in the expression. The final result is b_3, which is obtained from the valuation β_3 and the expression m_3. Even though there is a mark at the final character b in m_3, b_3 is still 0 since the query $\mathsf{Q}^{+}(v)$ is not satisfied by β_3.

Theorem 3.30 (Resource Usage of oMatch). Suppose $r \in \text{OReg}$ with $|r| = m$ and $\langle w, \beta \rangle \in \mathcal{O}(\Sigma, V)$ with $|w| = n$. Then, one can compute $\mathsf{oMatch}(r, \langle w, \beta \rangle)$ in $O(m \cdot n)$ time in a streaming manner that requires an additional $O(m)$ state space.

Proof. With the caching trick explained in subsection 3.6.2, we know that reading off the cFinal and cNullable fields can be done in $O(1)$ time. Similarly, applications of the smart constructors mkEps, mkCharClass, etc also take $O(1)$ time. Using this information, we can show that the functions sync, cFollow and cRead in Figure 4 can be computed in $O(m)$ time, since they access each of their subexpressions exactly once. The function applies cRead using each character in w, sync using each valuation in β, and uses cFollow exactly $|w|$ times. Thus, the functions of complexity $O(m)$ are iterated $O(n)$ times, giving us a total time complexity of $O(m \cdot n)$.

Definition 3.27 shows how to compute oMatch in a streaming manner. At each step, we only need to store the additional state represented via the argument m of oMatch'. This requires $O(m)$ space. □

3.7 Efficient Layerwise Matching

In Section 3.5, we have shown that oracle regular expressions can be used to evaluate regular expressions with lookahead if the truth values of the lookaheads are supplied as oracle valuations. In Section 3.6, we have demonstrated how oracle regular expressions can be matched on oracle strings. Now, we will show how to combine these two results to match regular expressions with lookahead in an efficient manner.

3.7.1 Computing Tapes

In Lemma 3.16, we see that the appropriate oracle valuations for the oracle regular expression $\mathsf{abstract}(r)$ is given in terms of $\mathsf{oval}(r, w)$, which is defined in terms of tapes of the maximal lookarounds. Here, we show how these can be computed using other existing concepts. One important detail here is that we need to scan the strings in reverse order in order to compute oracle valuations for lookahead.

Definition 3.31 (Reversal of LReg and OReg expressions). Let $r \in \mathsf{LReg}$ be an expression. The reversal of r, written $\mathsf{rev}(r)$ is defined as follows:

$$\begin{aligned} \mathsf{rev}(r) &= r \text{ if } r \in \{\varepsilon\} \cup \mathcal{P} & \mathsf{rev}((?{>}\,r)) &= (?{<}\,\mathsf{rev}(r)) \\ \mathsf{rev}(r_1 \cdot r_2) &= \mathsf{rev}(r_2) \cdot \mathsf{rev}(r_1) & \mathsf{rev}((?{<}\,r)) &= (?{>}\,\mathsf{rev}(r)) \\ \mathsf{rev}(r_1 + r_2) &= \mathsf{rev}(r_1) + \mathsf{rev}(r_2) & \mathsf{rev}((?{\not>}\,r)) &= (?{\not<}\,\mathsf{rev}(r)) \\ \mathsf{rev}(r^*) &= \mathsf{rev}(r)^* & \mathsf{rev}((?{\not<}\,r)) &= (?{\not>}\,\mathsf{rev}(r)) \end{aligned}$$

If $s \in \text{OReg}$, then $\mathsf{rev}(s)$ is defined in a similar manner by replacing the lookaround clauses with $\mathsf{rev}(\mathsf{Q}^+(v)) = \mathsf{Q}^+(v)$ and $\mathsf{rev}(\mathsf{Q}^-(v)) = \mathsf{Q}^-(v)$.

The reversals satisfy the following useful properties:

Lemma 3.32. Let $r \in \mathsf{LReg}$, $s \in \text{OReg}$, $\hat{r} = \mathsf{abstract}(r)$, $w \in \Sigma^*$, $\beta = \mathsf{oval}(r, w)$ and $\beta' \in (2^V)^{|w|+1}$. Then, the following hold:

1. $w, [i, j] \vDash r \iff \mathsf{rev}(w), [|w| - j, |w| - i] \vDash \mathsf{rev}(r)$
2. $\langle w, \beta' \rangle, [i, j] \vDash s \iff \langle \mathsf{rev}(w), \mathsf{rev}(\beta') \rangle, [|w| - j, |w| - i] \vDash \mathsf{rev}(s)$
3. $\mathsf{tape}(\lhd, r, w) = \mathsf{rev}(\mathsf{tape}(\rhd, \mathsf{rev}(r), \mathsf{rev}(w)))$
4. $\mathsf{tape}(\mathsf{rev}(r), \mathsf{rev}(w)) = \mathsf{tape}(\mathsf{rev}(\hat{r}), \langle \mathsf{rev}(w), \mathsf{rev}(\beta) \rangle)$

Part (1) and (2) of the lemma are characteristic properties of reversals of LReg and OReg expressions respectively. Part (3) gives us a way to express $\mathsf{tape}(\lhd)$ in terms of $\mathsf{tape}(\rhd)$. It is a direct consequence of (1). Part (4) follows from Part (1) and Lemma 3.16. This tells us that we can work with the reversal of $\mathsf{abstract}(r)$ instead of r itself, and saves us from repeatedly reversing subexpressions, which is crucial for the linear time complexity of the algorithm.

3.7.2 Matching Algorithm

In this section, we will describe the function eval which given $r \in \mathsf{LReg}$ and $w \in \Sigma^*$ computes $\mathsf{abstract}(r)$ and $\mathsf{oval}(r, w)$ together using oMatch from Definition 3.27.

The definition of the function eval shown in Figure 3.5, is defined in terms of the auxiliary function $\mathsf{evalAux}$. The function $\mathsf{evalAux}$ is given five arguments: w, $\overline{w}$, r, i and T. The first two arguments are the string and its reversal. We compute the reversal in advance in the function eval so that we do not repeatedly reverse the string

```
// Computes $\langle \mathsf{abstract}(r), \mathsf{oval}(r, w)\rangle$

Definition $\mathit{eval}\ (r : \mathit{LReg})\ (w : \mathit{list}\ \Sigma) : \mathit{OReg} \times \mathit{list}\ \mathbb{B}$ :=
    let $(\hat{r}, \_, T) = \mathsf{evalAux}(w, \mathsf{rev}(w), r, 0, [\,])$ in
      let $\beta = \mathsf{transpose}(\mathsf{rev}(T))$ in
      $(\hat{r}, \beta)$

// Ensure $\overline{w} = \mathsf{rev}(w)$
// Appends the tapes for the maximal lookarounds of $r$ to $T$
Fixpoint $\mathit{evalAux}(w, \overline{w} : \mathit{list}\ \Sigma)\ (r : \mathit{LReg})\ (i : \mathbb{N})\ (T : \mathit{list}\ \mathit{list}\ \mathbb{B}) : \mathit{OReg} \times \mathbb{N} \times \mathit{list}\ \mathit{list}\ \mathbb{B}$ :=
    match $r$ with
        $\varepsilon \Longrightarrow (\varepsilon, 0, T)$
        $\sigma \Longrightarrow (\sigma, 0, T)$
        $r_1 \circ r_2$ where $\circ \in \{\cdot, +\} \Longrightarrow$
            let $(s_1, n_1, T') = \mathsf{evalAux}(w, \overline{w}, r_1, i, T)$ in
            let $(s_2, n_2, T'') = \mathsf{evalAux}(w, \overline{w}, r_2, i + n_1, T')$ in
            $(s_1 \circ s_2, n_1 + n_2, T'')$
        $r^* \Longrightarrow$
            let $(s, n, T') = \mathsf{evalAux}(w, \overline{w}, r, i, T)$ in
            $(s^*, n, T')$
        $(?{<}\,r) \Longrightarrow$
            let $(s, \beta) = \mathsf{eval}(r, w)$ in
            let $\mathit{tape} = \mathsf{oMatch}(s, \langle w, \beta\rangle)$ in
            let $q = \mathsf{Q}^+(v_i)$ in
            $(q, 1, [\mathit{tape}] + T)$
        $(?{\not<}\,r) \Longrightarrow$
            let $(s, \beta) = \mathsf{eval}(r, \overline{w})$ in
            let $\mathit{tape} = \mathsf{rev}(\mathsf{oMatch}(s, \langle \overline{w}, \beta\rangle))$ in
            let $q = \mathsf{Q}^+(v_i)$ in
            $(q, 1, [\mathit{tape}] + T)$
        $(?{>}\,r) \Longrightarrow$
            let $(s, \beta) = \mathsf{eval}(r, \overline{w})$ in
            let $\mathit{tape} = \mathsf{rev}(\mathsf{oMatch}(\mathsf{rev}(s), \langle \overline{w}, \mathsf{rev}(\beta)\rangle))$ in
            let $q = \mathsf{Q}^+(v_i)$ in
            $(q, 1, [\mathit{tape}] + T)$
        $(?{\not>}\,r) \Longrightarrow$
            let $(s, \beta) = \mathsf{eval}(r, w)$ in
            let $\mathit{tape} = \mathsf{oMatch}(s, \langle w, \beta\rangle)$ in
            let $q = \mathsf{Q}^+(v_i)$ in
            $(q, 1, [\mathit{tape}] + T)$
```

Figure 3.5 : Definition of eval, which computes $\langle \mathsf{abstract}(r), \mathsf{oval}(r, w)\rangle$

in recursive calls of $\mathsf{evalAux}$. The third argument is the expression r that we want to abstract. The fourth argument is the index i used for computing indices of the queries. The fifth argument is a list of tapes, which have already been computed. The output has three components, s, n and T. The first component s is intended to be $\mathsf{abstract}(r)$. The second component n is $\mathsf{arity}(r)$. And the last component T consists of $\mathsf{tape}(m_i, w)$ for each maximal lookarounds m_i in r. However, these tapes appear in reverse order. This is because we add each new tape to the front of the list T for the sake of efficiency. The formal connection is stated in the following lemma.

Lemma 3.33 (Behavior of evalAux and eval). Suppose $w \in \Sigma^*$, $\overline{w} = \mathsf{rev}(w)$, $r \in \mathsf{LReg}$, $i \in \mathbb{N}$, and T is a list of sequences each of length $|w| + 1$. Define $(s, n, T') = \mathsf{evalAux}(w, \overline{w}, r, i, T)$. Then, the following holds:

1. $s = \mathsf{abstract}_i(r)$ (see Definition 3.12).
2. $n = \mathsf{arity}(r)$, the number of maximal lookarounds in r.
3. $T' = T'' \mathbin{+\!\!+} T$, where T'' is a list of $|w| + 1$ length sequences added to T. There are $n = \mathsf{arity}(r)$ sequences in T''. Furthermore, T'' consists of the tape of the maximal lookarounds stacked in reverse order, i.e., $T''[n - j + 1] = \mathsf{tape}(\mathsf{maxLk}(r)[j], w)$ for all $0 \le j < n$.

As a result, if $(\hat{r}, \beta) = \mathsf{eval}(r, w)$, then, $\hat{r} = \mathsf{abstract}(r)$ and $\beta = \mathsf{oval}(r, w)$.

The proof of this theorem makes use of the correctness of oMatch, (i.e., Theorem 3.28) along with Lemma 3.16 (about the connection of oval and $\mathsf{abstract}$) and Lemma 3.32 (about the connection of $\mathsf{tape}(\rhd)$ and $\mathsf{tape}(\lhd)$).

The results of $\mathsf{eval}(r, w)$ can be passed to oMatch to check if we have a match.

Definition 3.34 (**Matching LReg**)**.** Let $r \in \mathsf{LReg}$, $w \in \Sigma^*$, $[i, j]$ a window in w and let $(\hat{r}, \beta) = \mathsf{eval}(r, w[i, j])$. Define $\mathsf{match}(r, w, i, j) \in \mathbb{B}^{j-i+1}$ as $\mathsf{match}(r, w, i, j) = \mathsf{oMatch}(\hat{r}, \langle w, \beta \rangle [i, j])$

The following theorem follows from Lemma 3.33 (establishing the correctness of eval), Theorem 3.28 (establishing the correctness of oMatch) and Lemma 3.16 (establishing the connection between $\mathsf{abstract}(r)$ and $\mathsf{oval}(r, w)$).

Theorem 3.35 (**Correctness of match**)**.** Let $r \in \mathsf{LReg}$, $w \in \Sigma^*$ and $[i, j]$ a window in w. Then,

$$\mathsf{match}(r, w, i, j)[k] = \begin{cases} 1 & \text{if } w, [i, i+k] \vDash r \\ 0 & \text{if } w, [i, i+k] \nvDash r \end{cases}$$

for all $0 \le k \le j - i$.

The following theorem says that the function match can be computed in linear time, and requires a linear amount of space.

Theorem 3.36 (**Resource Usage of match**)**.** Suppose $r \in \mathsf{LReg}$ with $|r| = m$ and $w \in \Sigma^*$ with $|w| = n$. Then, one can compute $\mathsf{match}(r, w, i, j)$ in $O(m \cdot n)$ time using $O(m \cdot n)$ space.

The main intuition behind the result is as follows: the algorithm effectively decomposes the main expression r into subexpressions $r_1, r_2, \ldots r_k$ such that $|r| = |r_1| + |r_2| + \ldots + |r_k|$, where the decomposition is based on the nesting structure of layers of lookarounds. For each subexpression r_i, we run oMatch from Section 3.6, whose resource consumptions are bounded by Theorem 3.30. The space usage arises from the fact that we need to store tapes for each maximal lookaround. We have a maximum of $O(m)$ tapes, each of which stores $n + 1$ bits.

We note some important implementation details for the function eval. Consider a regular expression r of size m, which has $O(m)$ layers of lookaheads. In the computation of eval, we would need to reverse the given string $O(m)$ times. However, we have avoided this by reversing the string in advance before passing it to evalAux. Similarly, we may have to reverse the r itself $O(m)$ times. However, after processing each lookaround r_i of r, we work instead with an abstracted version of r in which r_i has been replaced by a query. If $|r_i| = m_i$ then, the abstracted version of r has size $m - m_i$, lowering the cost of subsequent reversals. These implementation details keep the time complexity of eval linear, instead of quadratic.

3.7.3 Leftmost Longest Match Extraction

The previous subsection answers the classical membership problem: given $r \in \mathsf{LReg}$ and $w \in \Sigma^*$, and a window $[i, j]$ in w, decide if $w, [i, j] \vDash r$. However, regular expressions with lookaround are sometimes used specifically so that one can specify additional constraints about the context in which a substring appears without *capturing* the context itself.

Example 3.37. One can use the regular expression [0–9]{3}–[0–9]{3}–[0–9]{4} to find a telephone number. Usually, the first three digits of the phone number are the area code. To extract just the area code from a telephone number, one can use the regular expression [0–9]{3}(?>–[0–9]{3}–[0–9]{3}) that contains a lookahead assertion.

Another example that shows the usefulness of lookbehind expressions is the extraction of an email address domain. Suppose $\alpha = [0\text{–}9A\text{–}Za\text{–}z]$ is a predicate that contains the alphanumeric characters. One can use the regular expression $\alpha^*@\alpha^*.\alpha^*$ to match email addresses. To extract the domain of the email address, one can write the expression $(?<\alpha^*@)\alpha^*.\alpha^*$. (Interestingly, this regex is not allowed by the PCRE standard, which disallows lookbehinds that could extend over a location of unbounded length. The algorithm we present in this chapter does not have this limitation.)

A match for r on w is a window $[i,j]$ such that $w,[i,j] \vDash r$. Call a match $[i,j]$ *maximal* if it is not subsumed by any other match $[i',j']$ where $i' \le i \le j \le j'$ with either $i' < i$ or $j < j'$. The *leftmost longest* match is a maximal match $[i,j]$ where i has the least possible value. For a given regular expression r and a word w, the left-most maximal match is uniquely defined. The problem of extracting the leftmost-longest match has been discussed before (see, e.g., the notes by Russ Cox [99]) and is considered the POSIX standard. We discuss below how to extract the leftmost longest match for regular expressions with lookarounds.

Definition 3.38. Let $r \in \mathsf{LReg}$ and $w \in \Sigma^*$. Define $\mathsf{llMatch}(r,w)$ as follows.

1. Compute $(\hat{r},\beta) = (\mathsf{abstract}(r), \mathsf{oval}(r,w)) = \mathsf{eval}(r,w)$.

2. Compute $t_1 = \mathsf{oMatch}(\Sigma^* \cdot \mathsf{rev}(\hat{r}), \langle \mathsf{rev}(w), \mathsf{rev}(\beta)\rangle)$. Let i' be the largest index of t_1 that is 1. If such i' does not exist, let $\mathsf{llMatch}(r,w)$ be undefined.

3. Let $i = |w| - i'$. Compute $t_2 = \mathsf{oMatch}(\hat{r}, \langle w,\beta\rangle[i,|w|])$. Let d be the largest index of t_2 that is true. If there is no such d, let $\mathsf{llMatch}(r,w)$ be undefined. Otherwise, define $\mathsf{llMatch}(r,w) = [i, i+d]$.

Theorem 3.39 (Correctness of llMatch). Let $r \in \mathsf{LReg}$ and $w \in \Sigma^*$. If $\mathsf{llMatch}(r,w)$ is undefined, then for any window $[i,j]$ in w, $w,[i,j] \nvDash r$. If, $\mathsf{llMatch}(r,w) = [i,j]$, then $w,[i,j] \vDash r$ and $[i,j]$ is the leftmost maximal match for r on w.

In Coq, we state this theorem as follows.

```
Theorem llmatch_correct (r : @LRegex A) (w : list A) :
  (forall n d, llmatch r w = Some (n, d) (* when defined *)
    -> match_regex r w n d (* match *)
    /\ (forall n', n' < n (* leftmost *)
    -> ~ (exists d', n' + d' <= length w /\ match_regex r w n' d'))
    /\ (forall d', d' > d -> n + d' <= length w (* maximal *)
    -> ~ match_regex r w n d'))
  /\
  (llmatch r w = None (* when undefined *)
    -> forall n d, n <= n + d <= length w -> ~ match_regex r w n d).
```

3.8 Experiments

We conducted experiments to empirically assess the performance of our formally verified algorithm. We compare it with industrially used tools PCRE [34], and `java.util.regex` of the Java standard library [35] and the tool developed and verified in Lean presented in [1]. The Lean tool is based on Brzozowski derivatives [100], and the two industrial tools are backtracking-based.

We have chosen three families of synthetic regexes (Table 3.2) which we have called `DNLA`, `NX` and `ND`. The family `DNLA` consists of increasingly many disjuncts of negative lookaheads, while `NX` and `ND` consist of increasingly nested lookaheads. For the families `NX` and `ND`, we use input strings of the form a^n; and for the family `DNLA`, we use input strings of the form $e^n \cdot abcd$. These expressions and inputs are deliberately chosen to be computationally challenging. We wish to force the algorithms to explore the whole string (instead of exiting early), possibly multiple times.

Figure 3.6 shows the performance of the four tools on the three different families. The rows from top to bottom represent the different tools: `extracted` (representing

Table 3.2 : Families of Regular Expressions in Experiments

Family	Inst. 1	Inst. 2	Inst. 3	⋯
`DNLA`	`(?!.*a.*).*`	`((?!.*a.*)|(?!.*b.*)).*`	`(((?!.*a.*)|(?!.*b.*))|(?!.*c.*)).*`	⋯
`NX`	`a(?=.*c)`	`a(?=.*(?=.*c))`	`a(?=.*(?=.*(?=.*c)))`	⋯
`ND`	`a(?=.*b)`	`a((?=.*c)|.*a(?=.*b))`	`a((?=.*d)|(.*a((?=.*c)|.*a(?=.*b))))`	⋯

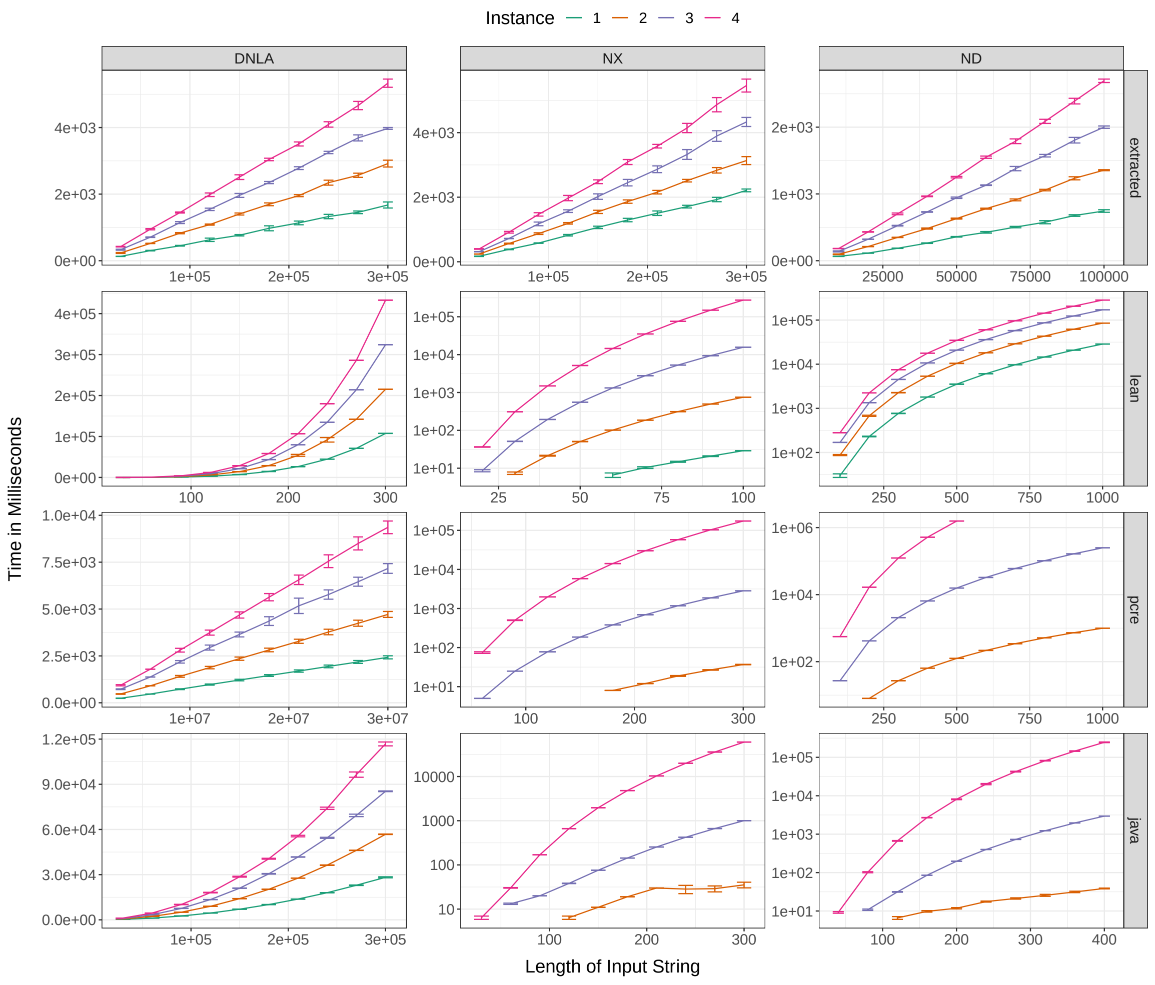


Figure 3.6 : Performance of our extracted Haskell code (`extracted`), the Lean tool in [1] (`lean`), PCRE (`pcre`), and Java (`java`) on the three different regex families `DNLA`, `NX`, and `ND`

our Haskell code extracted from Coq), `lean`, `pcre` and `java`. The columns (left to right) constitute the regex families: `DNLA`, `NX`, and `ND`. The colors represent the different instances of the families, as shown in the legend. The x-axis is the length of the input string, and the y-axis represents the running time in milliseconds. The times reported are the averages of five trials; standard deviations are shown using error bars. Note that the scales of both the x- and y-axes of the subfigures differ due to the difference in capacities of the tools.

Inspecting the first row confirms our claim that our algorithm has a linear running time. Generally, our extracted tool is able to handle longer inputs compared to the other tools, even for higher instances. An exception is the `DNLA` family, over which the `pcre` tool has the best performance. However, for the hardest instance and the longest input in `DNLA`, the running time of our tool is only 5% of that of Java. For the families `NX` and `ND`, we have plotted the y-axis in log-scale for all the tools other than `extracted`. Both the industrial tools seem to be showing super-linear growth for these families. We conjecture that the quick blowup in the running times in `pcre` and `java` is caused by 'catastrophic backtracking'; and in `lean` due to the explosion in the size of the derivatives. An interesting observation is that the derivative-based implementation of `lean` seems to outperform `java` and `pcre` for the family `ND`.

Data points with extremely short running times (i.e., $\leq$ 5ms) have been discarded due to their unreliability. For `pcre`, we set the parameter `match_limit`, which indicates the allowed amount of backtracking, to the maximum possible value of $2^{32} - 1$, and the tool aborts on workloads that exceed this threshold.

Experimental Setup. The experiments were conducted on a MacBook Air M2 (8 cores) running Ventura 13.6.1 with 8 GB of RAM. We used PCRE version 10.42 and Apple clang version 14.0.3 for compilation. The Lean code and proof-scripts are taken from [1] and built with Lean 4 (version 4.5.0-rc1). The runtime environment used for Java is Java SE version 20.0.2. Our extracted Haskell code is compiled with `ghc` version 9.4.8.

3.9 Related Work

An early example of using lookarounds in the context of parsing is [101], where one-character lookarounds are used. In [102], deterministic transducers are obtained through lookaheads. Positive and negative lookarounds are part of Parsing Expression Grammars [103]. Lookarounds are often used to specify the context from which data should be extracted, as seen in CDuce [104] (which operates on XML documents using regular patterns) and Kleenex [105] (which uses regular-expression-based grammars to specify transductions). Recently, extensions of context-free grammars with lookaheads have been studied in [106, 107].

At the time of writing, the widely used regex engines that do support lookarounds are all based on backtracking, and can thus take exponential time to match in the worst-case scenario (even in the absence of lookarounds). Some of them, including PCRE and Python, do not allow unbounded lookbehinds. The linear time algorithm for matching regular expressions with lookarounds we have presented in this chapter was originally published in [28]. This chapter presents the implementation in a purely functional approach, in particular the use of Marked Regular Expressions instead of explicit NFAs. The approach of using marked regular expressions to simulate NFAs was discussed in [33, 108]. Formalized versions of Marked Expressions in Isabelle/HOL are seen in [98] and [32]. A discussion on two variants for the marking of regular expressions can be found in [98].

The matching of regexes with lookaround assertions is also considered in [109] and [110]. Barriere and Pit-Claudel [109] represent the NFAs using Virtual Machine instructions, a technique introduced by Rob Pike [111] and popularized by Russ Cox [99]. Fujinami and Hasuo [110] use a memoization approach to filling in the oracle valuations, and use a formalism based on 'NFA with sub-automata'.

The backtracking-based greedy strategy of matching regexes prefers the first option r_1 in a union $r_1 + r_2$ and prefers to lengthen the number of blocks matching r in a Kleene iteration r^*. Compatibility with the greedy strategy is a consideration in [110] and [112]. The greediest parse trees can be constructed, without backtracking, in linear time as demonstrated in [113, 114]. In our case, we have shown how to extract the leftmost-longest match, specified in the POSIX [115] standard.

Alternative semantics for lookaheads can be found in [97] and [116]. Miyazaki and Minamide [97] associate lookahead expressions with a pair of strings of the form (s, t) where s is the match and t is the lookahead. Berglund et al. [116] define the semantics of lookaheads using alternating finite automata (AFA). A consequence of Berglund's approach is that the string could be matched in $O(m \cdot n)$ time by running the AFA on the string in reverse. Note that neither of these approaches handles lookbehinds.

A streaming construction that scans the string from left to right (see, e.g., Morihata [117]) would produce a DFA of doubly exponential size (see [97] for a lower bound). This issue is avoided in our algorithm, since we may do right-to-left passes (for lookaheads) in addition to left-to-right passes. An optimization proposed in [28] shows that when the regex only has lookbehinds, the matching could be done in a streaming manner by pipelining a network of NFA.

Limited forms of intersection and complementation can be simulated by expressions with lookarounds. For example, the expression $(?{>}\, r_1) \cdot (?{>}\, r_2) \cdot \Sigma^*$ would match the strings that would match both r_1 and r_2, and the expression $(?{\not>}\, r_1)\Sigma^*$ only matches strings that do not match r_1. However, this encoding would not necessarily work when the negation or intersection needs to be nested inside concatenation or Kleene iteration. Expressions with negation are known to be non-elementarily succinct [118] while expressions with intersection are exponentially succinct [119]. Some algorithmic improvements in matching such expressions have been found in [120].

Derivatives are popular for functional and verified implementations of regex matching. Brzozowski derivatives [100] are the simplest approach to doing so, but a direct implementation may be impractical since the size of the derivatives can grow very large. Coquand and Siles [121] have formalized the notion of Brzozowski derivatives using Coq. Verbatim++ [122] is a recent verified lexing tool which uses Brzowzowsksi derivatives, which leverages memoization to prevent recomputation. A related approach is Antimorov's [123] partial derivatives which correspond to states of an NFA. Partial derivatives have been verified by [124] and [125].

Stanford et al. [126] have defined symbolic derivatives to deal with Boolean combinations of derivatives. This work has been extended in [127, 128] and used by Moseley et al. [128] to develop a matching library in the .NET framework. Zhuchko et al. [1] have verified an algorithm for matching regular expressions with lookarounds in Lean based on the idea of location-based (i.e., context-dependent) derivatives from [128]. Urban and their coauthors have verified POSIX lexing based on derivatives in Isabelle/HOL [129, 130, 131].

An extensive theory of regular languages involving regular expressions, DFAs and NFAs were formalized in Coq by Doczkal et al. [132, 133]. A matrix-based NFA formalization in Agda is seen in [134]. Kammar and Marek [135] have formalized a parser based on typed regular expressions in Idris. A Coq formalization of finite state automata is also discussed in [136], which they have verified in the context of Kleene

Algebras.

Kleene observed some of the algebraic properties in his seminal work on regular expressions [19]. The algebraic theory was further developed by Conway [137] in his extensive monograph. A particular problem of interest is that of finding a sound and complete axiomatization of Kleene Algebras. Conway's axiomatization is infinitary. Salomaa [138] gave an axiomatization, but it relied on inference rules that were unsound for other models (i.e, other than that of regular languages). Redko [139] proved the negative result that no finite *equational* axiomatization is possible. In 1994, Kozen [31] was the first to give a finitary axiomatization of Kleene Algebras (which involves equational *implications* in addition to equations). These are the axioms we have considered in 3.4 of this chapter. The development of a complete axiomatization for the theory of regular expressions with lookarounds is left for future work. Techniques such as the extension of Kleene Algebra with additional equations [140] might be useful in this regard.

Chapter 4

Tokenization using Thompson's Algorithm

4.1 Introduction

Even if its source is available, an important aspect of trusting the security of software binaries would still rely on the ability to trust the compiler used to compile it, as noted by Ken Thompson in his Turing Award lecture [141]. Indeed, examples of compiler-induced security vulnerabilities do exist in real code [142]. Compilers are large pieces of software, and ensuring their safety and correctness is a non-trivial task. Efforts such as CompCert [143, 144] aim to formally verify the correctness of a C compiler using the proof assistant Coq. The CakeML project is a similar effort, with the goal of developing an ML ecosystem with a verified compiler [145].

Lexing source code into a stream of tokens is one of the first steps the front-end of a compiler performs. In order to have an end-to-end verified compiler, it is thus important to verify lexing. Currently, the lexing phase in the CompCert compiler is unverified. In the CakeML project, a handcrafted verified lexer is used that can process a fixed list of tokens. It is thus desirable to have a more flexible lexer that could be utilized in such systems.

Tokens in most programming languages are typically described using regular expressions. A lexer takes a list of regular expressions describing potential tokens, and splits the source code into tokens based on their matches. Ties are broken using the maximal munch rule (section 4.5), which favors the longest match and the earliest expression. Note that using unsuitably crafted regex matching engines, such as those which use backtracking, could lead to security vulnerabilities in certain cases [94]. Industry standard lexer generators such as Flex [146] compile the regular expressions into deterministic finite state automata (DFA), which can be simulated very efficiently.

Existing efforts towards verified lexing include Verbatim [147], Verbatim++ [122] and Coqlex [148]. All of these are based on Brzozowski derivatives. Despite their elegance, the difficulty with this approach is the potential exponential growth in the size of derivatives, as we demonstrate in Section 4.6. The manipulation of regexes for the computation of derivatives involves the allocation and deallocation of memory for the syntax trees, which can also be an overhead.

Alexee, the verified lexing tool we describe in this chapter, is based on Thompson NFAs [149]. Thompson NFAs are exponentially succinct compared to DFAs, but can still be simulated efficiently (in linear time per character). Unlike a DFA simulation, which has one active state, NFA simulations must maintain a set of active states. Since our automata have ε-transitions, we need to use a graph reachability algorithm to compute the transitions for a set of active states. Having ε-transitions in the automaton is crucial to make sure that it has a linear number of edges, which is important for the efficiency of the simulation.

Since our lexing algorithm is based on NFAs, we have to develop a system for reasoning about graphs and reachability. This is a significant challenge in this formalization effort that does not arise in derivative-based approaches. A substantial part of our effort is spent on characterizing the paths in the Thompson NFA (see Section 4.3). We choose to represent the states of the automata using integer identifiers to produce an efficient implementation. Another choice we have made towards obtaining an efficient impelementation is to use mutable arrays (see Section 4.3.1): we do this by axiomatizing arrays and then extracting them into a (slightly unfaithful) OCaml representation which modifies the array in place. This is useful in implementing depth-first search in an efficient manner (see Section 4.4.1).

Chapter Outline. This chapter has two main components, the verification of Thompson's algorithm and its use for maximal munch lexing. In Section 4.2, we introduce the syntax and semantics of regular expressions and automata. Thompson's NFA Construction is discussed in Section 4.3. In Section 4.4, we discuss the simulation of these NFA to match strings. This involves a depth-first search to compute the ε-closures of a set of states which is discussed in Section 4.4.1. The main lexing algorithm is discussed in 4.5. The emperical performance of our lexer is compared against that of other tools in Section 4.6. Finally, we discuss related work in Section 4.7.

4.2 Regular Expressions and Automata

Let Σ be an alphabet, and Σ^* the set of strings over Σ. Given a string $w \in \Sigma^*$, we denote its length by $|w|$. The empty string (i.e., the string of length 0) is denoted by ε. For a string $w \in \Sigma^*$, we will call a formal pair $[i, j]$ with $0 \leq i \leq j \leq |w|$ a *window* in w. We write $w[i, j]$ for the substring $w_i w_{i+1} \ldots w_{j-1}$ (note the omission of j) of w.

Definition 4.1 (Regular Expressions)**.** Let $\mathcal{P}$ be a set of decidable predicates over Σ (i.e., functions of type $\Sigma \to \{1, 0\}$). The set Reg of regular expressions is defined by the following grammar:

$$r, r_1, r_2 ::= \varepsilon \mid \sigma \in \mathcal{P} \mid r_1 \cdot r_2 \mid r_1 + r_2 \mid r^*$$

We associate with each regular expression $r \in \mathsf{Reg}$ a language $[\![r]\!] \subseteq \Sigma^*$ in the following usual way:

$$
\begin{aligned}
[\![\varepsilon]\!] &= \{\varepsilon\} \\
[\![\sigma]\!] &= \{w \in \Sigma^* \mid \sigma(w_0) = 1\} \\
[\![r_1 \cdot r_2]\!] &= \{w_1 \cdot w_2 \mid w_1 \in [\![r_1]\!], w_2 \in [\![r_2]\!]\} \\
[\![r_1 + r_2]\!] &= [\![r_1]\!] \cup [\![r_2]\!] \\
[\![r^*]\!] &= \{w_1 \cdots w_n \mid n \geq 0, w_i \in [\![r]\!]\}
\end{aligned}
$$

In our Coq formalization, strings over a type `A` are represented using the type `list A`. We use an inductive type to represent regular expressions, and the denotation function $[\![\cdot]\!]$ is defined as a `Fixpoint` producing a `list A` → `Prop`.

```
Inductive Regex : Type :=
| Epsilon : Regex
| CharClass : (A -> bool) -> Regex
| Concat : Regex -> Regex -> Regex
| Union : Regex -> Regex -> Regex
| Star : Regex -> Regex.

Fixpoint regex_language (r : Regex) (w : list A) : Prop :=
  match r with
  | Epsilon => w = []
  | CharClass f => exists a, w = [a] /\ f a = true
  | Concat r1 r2 => exists w1 w2, w = w1 ++ w2
        /\ regex_language r1 w1 /\ regex_language r2 w2
  | Union r1 r2 => regex_language r1 w \/ regex_language r2 w
  | Star r => exists ws, w = concat ws
        /\ (forall w', In w' ws -> regex_language r w')
  end.
```

Throughout our formalization, we use the type `X → Prop` to represent sets of elements of type `X`. We define the membership relations $\in$ and $\notin$, the set operations $\cup, \cap$, and the set relations $\subseteq$ and $\equiv$ (set equality). The definition we use for set equality (for $\equiv$) is extensional, which is not the same as Coq's standard equality (i.e, $=$). However, by defining appropriate setoid morphisms, Coq's generalized rewriting mechanism [150] can be used to rewrite equations using our extensional equality.

4.2.1 Non-deterministic Finite Automata

As we will see in the Section 4.3, we can restrict ourselves to a class of NFA which lends itself to a certain well-behaved representation.

Definition 4.2 (Transition systems, Automata)**.** A *transition system* over an alphabet Σ, using the predicates $\mathcal{P}$ of type $\Sigma \to \{1,0\}$) is a function $\Delta : \mathbb{N} \to \text{None} \oplus \mathsf{Char}(\mathcal{P}) \oplus \mathsf{Jump}(\mathbb{N}) \oplus \mathsf{Split}(\mathbb{N} \times \mathbb{N})$ (where $\oplus$ denotes a disjoint union). If $\Delta(p)$ has the value $\mathsf{Char}(\sigma)$, or $\mathsf{Jump}(q)$, or $\mathsf{Split}(q_1, q_2)$, we say that $\{p+1\}$, or $\{q\}$, or $\{q_1, q_2\}$ are the *successors* of p in Δ, respectively.

An *automaton* is a transition system Δ together with a 'final state' $F \in \mathbb{N}$, that satisfies the following conditions: (1) for all $q \geq F$, $\Delta(q) = \text{None}$, and (2) for each successor q of p in Δ, $q \leq F$. The set of automata is denoted by $\mathcal{A}$.

The function Δ associates edges with each state of the transition system. If $\Delta(p) = \text{None}$, there are no outgoing edges from the state p. If $\Delta(p) = \mathsf{Char}(\sigma)$, then p is associated with the edge $p \to^{\sigma} p+1$. If $\Delta(p) = \mathsf{Jump}(q)$, then p is associated with the edge $p \to q$. If $\Delta(p) = \mathsf{Split}(q_1, q_2)$, then p is associated with the edges $p \to q_1$ and $p \to q_2$. Note that only the edges of the form $p \to^{\sigma} p+1$ emanating from p with $\Delta(p) = \mathsf{Char}(\sigma)$ are *guarded* with a predicate. The edges emanating from Jump or Split states are *guarded*.

A coherent sequence of these edges forms a path. The set of paths in the transition system T starting from p and ending at q is written as $\mathsf{Paths}_T(p,q)$. Given a path ρ, the label $\pi(\rho) \in \mathcal{P}^*$ can be obtained by concatenating the labels on its edges. This determines a set of strings as follows: $[\![\sigma_1 \cdot \sigma_2 \cdots \sigma_n]\!] = \{a_1 \cdot a_2 \cdots a_n \mid \bigwedge_{i=1}^{n} \sigma_i(a_i) = 1\}$. Sometimes, we refer to sequences of predicates (such as $\sigma_1 \cdot \sigma_2 \cdots \sigma_n$) as stringoids (often abbreviated as `stroid` in the code listings).

We implicitly assume that 0 is the initial state of an automaton. Thus, a run of an automaton $A = (\Delta, F)$ is a path ρ which starts at 0 and ends at F. We denote the set of runs of A by $\mathsf{Runs}(A) = \mathsf{Paths}_A(0,F)$. The language of the automaton A is defined as

$$[\![A]\!] = \{w \mid \rho \in \mathsf{Runs}(A), w \in [\![\pi(\rho)]\!]\}\,.$$

In Coq, we represent transition systems as a function, and automata as a record type consisting of a transition function and the final state.

```
Inductive Successor : Type :=
 | SNone : Successor (* p -/-> *)
 | SBlkd : (A -> bool) -> Successor (* p -a-> p + 1 *)
 | SSplt : nat -> nat -> Successor (* p -> q1, q2 *)
 | SJump : nat -> Successor (* p -> q *).

Definition Graph := nat -> Successor.

Record automaton : Type := {
    final : nat;
    delta: @Graph A;
  }.
```

```
Definition is_run_of (M : automaton) pp : Prop :=
    path_from_to 0 (final M) pp /\ path_in_graph (delta M) pp.

Definition automaton_language (M : automaton) (w : list A) : Prop :=
    exists pp, is_run_of M pp /\ stroidAccept (pathToStroid pp) w = true.
```

In our formalization, we view paths as either a `list edge` or a `list nat` (where the `nat`s represent states). There are additional predicates that check the validity of a path. For example, the proposition `path_from_to p q pp` asserts that the path `pp` is a path from state `p` to `q`, the proposition `path_in T pp` asserts that the edges present in the path `pp` respect the edges of the transition system `T`, and the proposition `consecutivity pp` asserts that in the path `pp` the destination of each edge is the source of the next edge. We also have predicates `vpath_from_to` and `vpath_in` which express similar assertions for paths represented as a list of vertices. A significant part of our formalization involves several lemmas about these predicates which allow us to reason about paths.

4.3 Thompson's Construction

In this section, we describe Thompson's inductive construction which produces an automaton $\mathcal{A}(r)$ given a regular expression $r \in \mathsf{Reg}$ which recognizes the language $[\![r]\!]$.

Definition 4.3 (Thompson's Construction)**.** Given automata $A_1 = (\Delta_1, F_1)$ and $A_2 = (\Delta_2, F_2)$, we define the operations of concatenation, union and Kleene iteration on them as in Figure 4.1. The automata $\mathcal{A}(\varepsilon)$ and $\mathcal{A}(\sigma)$ are also defined in the Figure 4.1. Given a regular expression r, $\mathcal{A}(r)$ is defined inductively by replacing occurrences of ε

$$\mathcal{A}(\varepsilon) = (\Delta, 0) \text{ where, } \Delta(p) = \text{None}$$

$$\mathcal{A}(\sigma) = (\Delta, 1) \text{ where, } \Delta(p) = \begin{cases} \mathsf{Char}(\sigma) & \text{if } p = 0 \\ \text{None} & \text{otherwise} \end{cases}$$

$$A_1 + A_2 = (\Delta, F_1 + F_2 + 2) \text{ where,}$$

$$\Delta(p) = \begin{cases} \mathsf{Split}(1, F_1 + 2) & \text{if } p = 0 \\ \mathsf{Jump}(F_1 + F_2 + 2) & \text{if } p = F_1 + 1 \\ \Delta_1(p-1) + 1 & \text{if } 1 \leq p \leq F_1 \\ \Delta_2(p - F_1 - 2) + F_1 + 2 & \text{if } F_1 + 2 \leq p \leq F_1 + F_2 + 1 \\ \text{None} & \text{otherwise} \end{cases}$$

$$A_1 \cdot A_2 = (\Delta, F_1 + F_2) \text{ where,}$$

$$\Delta(p) = \begin{cases} \Delta_1(p) & \text{if } p \leq F_1 \\ \Delta_2(p - F_1) + F_1 & \text{if } F_1 \leq p \leq F_1 + F_2 \\ \text{None} & \text{otherwise} \end{cases}$$

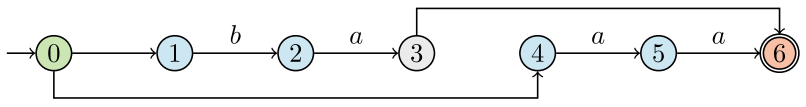
$$A_1^* = (\Delta, F_1 + 2) \text{ where,}$$

$$\Delta(p) = \begin{cases} \mathsf{Split}(1, F_1 + 2) & \text{if } p = 0 \\ \mathsf{Jump}(0) & \text{if } p = F_1 + 1 \\ \Delta_1(p-1) + 1 & \text{if } 1 \leq p \leq F_1 \\ \text{None} & \text{otherwise} \end{cases}$$

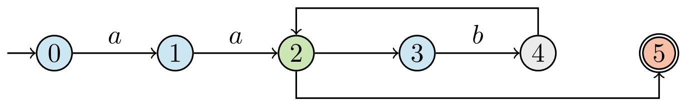
Figure 4.1 : Thompson's Construction

and $\sigma \in \mathcal{P}$ with $\mathcal{A}(\varepsilon)$ and $\mathcal{A}(\sigma)$, and recursively apply the operations of concatenation, union and Kleene iteration.

Example 4.4. Consider the regular expression $r_1 = ba + aa$. The Thompson NFA $\mathcal{A}(r_1)$ is as shown below. In $\mathcal{A}(r_1)$, the final state is $F_1 = 6$. State 0 is a Split state, states $1, 2, 4$ and 5 are Char states and state 3 is a Jump state, as represented by the color coding.



We show the automaton $\mathcal{A}(r_2)$ for the regex $r_2 = aab^*$ below. This automaton has 6 states. States 0, 1 and 3 are Char states, state 2 is a Split state and state 4 is a Jump state.



We associate with each of the constructs above a lemma which states that construction on the automata respect the operation on the regular expressions.

Lemma 4.5. Suppose r_1, r_2 and r are regular expressions. Then, the automata $\mathcal{A}(r_1)$, $\mathcal{A}(r_2)$ and $\mathcal{A}(r)$ satisfy the following properties:

1. $[\![r_1 + r_2]\!] = [\![\mathcal{A}(r_1) + \mathcal{A}(r_2)]\!]$
2. $[\![r_1 \cdot r_2]\!] = [\![\mathcal{A}(r_1) \cdot \mathcal{A}(r_2)]\!]$
3. $[\![r^*]\!] = [\![\mathcal{A}(r)^*]\!]$

Additionally, $[\![\varepsilon]\!] = [\![\mathcal{A}(\varepsilon)]\!]$ and for any $\sigma \in \mathcal{P}$, $[\![\sigma]\!] = [\![\mathcal{A}(\sigma)]\!]$.

The proof of these lemmas involve reasoning about runs. In our Coq formalization, we have a collection of lemmas labelled `union_run_fwd`, `union_run_bwd`, `concat_run_fwd`, `concat_run_bwd`, etc. The lemmas with the suffix `_fwd` provide a way to take runs in $\mathcal{A}(r_1)$ or $\mathcal{A}(r_2)$ and produce a run in $\mathcal{A}(r_1)+\mathcal{A}(r_2)$ or $\mathcal{A}(r_1)\cdot\mathcal{A}(r_2)$, while the lemmas with the suffix `_bwd` provide a way to take runs in $\mathcal{A}(r_1) + \mathcal{A}(r_2)$ or $\mathcal{A}(r_1) \cdot \mathcal{A}(r_2)$ and produce runs in $\mathcal{A}(r_1)$ or $\mathcal{A}(r_2)$. While proving these lemmas informally using a visual representation of the Thompson construction is straightforward, the formalization in Coq requires long and tedious arguments involving paths in our transition systems.

As an illustration, we show below the lemmas for the star operation. Taken together, these two lemmas state that a run in the automaton $\mathcal{A}(r)^*$ consists of a sequence of runs in $\mathcal{A}(r)$.

```
Lemma star_automaton_run_fwd (M : @automaton A) : forall pps,
    (forall pp, In pp pps -> is_run_of M pp)
    -> let pps' :=
        flat_map (fun pp => ESplt0 0 1 :: (path_up 1 pp)
            ++ [EJump (1 + final M) 0]) pps in
    is_run_of (star_automaton M) (pps' ++ [ESplt1 0 (final (star_automaton M))]).

Lemma star_automaton_run_bwd (M : @automaton A) : forall pp,
        is_run_of (star_automaton M) pp
    ->  exists pps, (forall ppA, In ppA pps -> is_run_of M (path_down 1 ppA))
    /\  pp = (flat_map (fun ppA => ESplt0 0 1 :: ppA
              ++ [EJump (1 + final M) 0]) pps)
            ++ [ESplt1 0 (final (star_automaton M))].
```

The key theorem of this section can be stated as follows:

Theorem 4.6. Let $r \in \mathsf{Reg}$ be a regular expression. Then, the automaton $\mathcal{A}(r)$ recognizes the language $[\![r]\!]$.

The proof is a straightforward induction on the structure of the regular expression r using Lemma 4.5. The theorem can be stated in a succinct manner in Coq as follows, using ≡ to denote extensional equality of sets of strings represented as `list A → Prop`.

```
Theorem regexAutomaton_language (r : @Regex A) :
    automaton_language (regexAutomaton r) ≡ regex_language r.
```

The following lemma establishes a bound on the number of states in the automaton $\mathcal{A}(r)$. This can be proven using a straightforward induction on the structure of the regular expression r by noticing that during each step of the construction only at most a constant number of additional states are added.

Lemma 4.7. Let $r \in \mathsf{Reg}$ be a regular expression. Let $\mathcal{A}(r) = (\Delta, F)$ be as defined in Definition 4.3. Then, F is $O(|r|)$, where $|r|$ is the size of the syntax tree of the regular expression r.

4.3.1 Using Arrays to Represent the Transition Function

In the above discussion, as well as throughout most of our Coq formalization, we have represented the transition function Δ as a function of type `nat → Successor`. Extracting this directly to OCaml would result in the use of closures to represent this function at runtime. We discuss the usage of arrays to represent the transition function, so that the function calls can be evaluated in constrant time. An opaque type $\mathsf{BArray}\ A$ is defined, together with operations $\mathsf{baLength}$, baGet, baSet, and baNew for obtaining the length of the array, accessing elements, setting elements, and creating a new array, respectively. The relationship between them is defined using a list of axioms, as shown in Figure 4.2. During extraction to OCaml, the type BArray is implemented as an OCaml array.

$$\begin{aligned}
\mathsf{baLength}(\mathsf{baNew}(n,x)) &= n \\
\mathsf{baLength}(\mathsf{baSet}(a,i,x)) &= \mathsf{baLength}(a) \\
\mathsf{baGet}(\mathsf{baNew}(n,x),i) &= \begin{cases} \mathrm{Some}(x) & \text{if } 0 \le i < n \\ \mathrm{None} & \text{otherwise} \end{cases} \\
\mathsf{baGet}(\mathsf{baSet}(a,i,x),j) &= \begin{cases} \mathrm{Some}(x) & \text{if } i = j \\ \mathsf{baGet}(a,j) & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 4.2 : Axioms for the BArray type

```
Extract Constant BArray "'a" => "'a array".
Extract Constant baNew => "fun n x -> Array.make n x".
Extract Constant baLength => "Array.length".
Extract Constant baGet =>
  "fun a i -> if i < Array.length a then Some (Array.get a i) else None".
Extract Constant baSet => "fun a i x -> Array.set a i x; a".
```

Notice that the implementation of baSet is given using `Array.set`, which mutates the array in place. This idea is not reflected in the axiomatization, but is necessary for the efficient use of arrays. Whenever there is an array a_2 defined as $\mathsf{baSet}(a_1, i, x)$, in order for the program to be safe, we must ensure that a_1 is never used again. Fortunately, this happens to be the case in all the places where we have made use of arrays.

To store the transition function as an array, we define a function arrayify which takes in a function f whose domain is a bounded set of natural numbers $\{0, 1, \dots n-1\}$, and returns an array of length n such that the i-th element of the array is $f(i)$. This can be done in a straightforward manner by iterating over the various possible values of i.

```
Definition iterSetArray (l : list nat) (init : BArray A) : BArray A :=
    fold_left (fun a i => baSet a i (F i)) l init.

Definition arrayify : BArray A :=
    iterSetArray (seq 0 n) (baNew n (F 0)).

Lemma arrayify_correct : forall i,
    i < n -> baGet arrayify i = Some (F i).
```

We will revisit the use of arrays in Section 4.4.1, where we need to maintain sets of visited states for the depth-first search algorithm. In a standard functional setting, these would be represented via Sets or Maps, with an implementation based on balanced binary search trees (such as Coq Standard Library's `Coq.MSets.MSetRBT` [151]). Standard operations (such as lookup and insertion) on these data structures would incur an additional logarithmic factor.

4.4 Simulating NFAs using Depth-First Search

In order to determine whether an NFA accepts the prefix fed so far, we maintain the set of states that are reachable on the prefix. Suppose $A = (\Delta, F)$ is an automaton. Then, this set is denoted by the following notation:

$$\mathsf{Reach}_A(w) = \{q \mid \exists \rho \in \mathsf{Paths}_A(0, q), w \in [\![\pi(\rho)]\!]\}.$$

Checking membership (i.e, whether $w \in [\![A]\!]$) can be performed easily if the set $\mathsf{Reach}_A(w)$ is available. The key observation here is that for all $w \in \Sigma^*$, $w \in [\![A]\!]$

if and only if $F \in \mathsf{Reach}_A(w)$. Thus, the remainder of the section is devoted to computing $\mathsf{Reach}_A(w \cdot a)$ given $\mathsf{Reach}_A(w)$.

In order to do this, we define two more concepts. Given a set of states S of A, and a character $a \in \Sigma$, we define $\mathsf{adv}(a, S) = \{p+1 \mid p \in S, \Delta(p) = \mathsf{Char}(\sigma), \sigma(a) = 1\}$. We also define the set $\mathsf{Closure}(S) = \{q \mid p \in S, \exists \rho \in \mathsf{Paths}(p,q), \pi(\rho) = \varepsilon\}$. In other words, $\mathsf{Closure}(S)$ is the set of states reachable from a state of S without encountering any guarded edges.

In Coq, we represent $\mathsf{Reach}_A(w)$ and $\mathsf{Closure}(S)$ using elements of type `nat` → `Prop`. The function adv is expressed on a list of states as a `filterMap`.

```
Definition reachOf (M : automaton) (w : list A) : nat -> Prop :=
  fun q => exists pp,
       path_in_graph (delta M) pp
    /\ path_from_to 0 q
    /\ stroidAccept (pathToStroid pp) w = true.

Definition epsClosure (M : automaton) (S : nat -> Prop) : nat -> Prop :=
  fun q => exists p pp, p ∈ S
    /\ path_in_graph (delta M) pp
    /\ path_from_to p q pp
    /\ pathToStroid pp = [].

Definition advance (G : Graph) (S : list nat) (a : A) : list nat :=
    filterMap (fun p => match G p with
                        | SBlkd pred => if pred a then Some (p + 1) else None
                        | _ => None
                        end) S.
```

The following lemma relates $\mathsf{Reach}_A(w \cdot a)$ with $\mathsf{Reach}_A(w)$.

Lemma 4.8. Let $A = (\Delta, F)$ be an automaton. Then, for any word $w \in \Sigma^*$ and character $a \in \Sigma$, we have

$$\mathsf{Reach}_A(w \cdot a) = \mathsf{Closure}(\mathsf{adv}(a, \mathsf{Reach}_A(w))).$$

Proof. The proof requires the following observation: Suppose that $\rho \in \mathsf{Paths}_A(0, q)$ with $\pi(\rho) = s \cdot \sigma$ where $s \in \mathcal{P}^*$ and $\sigma \in \mathcal{P}$. Then, ρ could be split up into $\rho = \rho_s \cdot (p \to^\sigma p+1) \cdot \rho_\varepsilon$, for some p where $\rho_s \in \mathsf{Paths}_A(0, p)$ with $\pi(\rho_s) = s$, and $\rho_\varepsilon \in \mathsf{Paths}_A(p+1, q)$ with $\pi(\rho_\varepsilon) = \varepsilon$. Note that the condition on ρ_ε implies that $q \in \mathsf{Closure}(\{p+1\})$.

Now suppose that $q \in \mathsf{Reach}_A(w \cdot a)$. Then there is a path $\rho \in \mathsf{Paths}_A(0, q)$ with $\pi(\rho) = s \cdot \sigma$ such that $w \in [\![s]\!]$ and $\sigma(a) = 1$. Using the fact above, we split up ρ into $\rho_s \in \mathsf{Paths}_A(0, p)$ where $\rho_s \in \mathsf{Paths}_A(0, p)$ and $\rho_\varepsilon \in \mathsf{Paths}_A(p+1, q)$. Thus, $p \in \mathsf{Reach}(w)$, and since $\sigma(a) = 1$, $p + 1 \in \mathsf{adv}(a, \mathsf{Reach}(w))$. Finally, since $\pi(\rho_\varepsilon) = \varepsilon$, $q \in \mathsf{Closure}(\mathsf{adv}(a, \mathsf{Reach}(w)))$.

Conversly, suppose $q \in \mathsf{Closure}(\mathsf{adv}(a, \mathsf{Reach}(w)))$. By unwinding the definitions, we see that there is some p and a path $\rho_s \mathsf{Paths}_A(0, p)$, with $w \in [\![s]\!]$, and an edge $p \to^\sigma p+1$ with $\sigma(a) = 1$, and a path $\rho_\varepsilon \in \mathsf{Paths}_A(p+1, q)$ with $\pi(\rho_\varepsilon) = \varepsilon$. Thus, the path $\rho_s \cdot (p \to^\sigma p+1) \cdot \rho_\varepsilon$ is a path in $\mathsf{Paths}(0, q)$ witnesses $q \in \mathsf{Reach}_A(w \cdot a)$. $\square$

Consider the subset $\mathsf{ReachG}_A(w) = \mathsf{Reach}_A(w) \cap \{q \mid \Delta(q) = None \vee \Delta(q) = \mathsf{Char}(\sigma)\}$, consisting only of all those states of $\mathsf{Reach}_A(w)$ which are not Jump or Split states. It is easy to see that $\mathsf{adv}(a, \mathsf{ReachG}_A(w)) = \mathsf{adv}(a, \mathsf{Reach}_A(w))$, since the Jump and Split states are ignored by the adv function. Defining $\mathsf{ClosureG}(S) = \mathsf{Closure}(S) \cap \{q \mid \Delta(q) = None \vee \Delta(q) = \mathsf{Char}(\sigma)\}$ in a similar manner, we have the following corollary.

Corollary 4.9. Let $A = (\Delta, F)$ be an automaton. Then, for any word $w \in \Sigma^*$ and character $a \in \Sigma$, we have

$$\mathsf{ReachG}_A(w \cdot a) = \mathsf{ClosureG}(\mathsf{adv}(a, \mathsf{ReachG}_A(w))).$$

Using the functions ReachG and $\mathsf{ClosureG}$ is slightly more efficient since the function adv has to iterate over a fewer number of states. Note that our observation that $w \in [\![A]\!]$ if and only if $F \in \mathsf{Reach}_A(w)$ can additionally be viewed as $w \in [\![A]\!] \iff F \in \mathsf{ReachG}_A(w)$.

4.4.1 Depth-First Search

While the straightforward implementation of adv is efficient, formulating an implementation of $\mathsf{Closure}$ in an efficient manner is trickier. In this subsection, we discuss our formalization of the depth-first search algorithm to tackle the computation of $\mathsf{Closure}$.

The key to the efficiency of DFS is that it avoids exploring paths that pass through states that have already been visited. To encapsulate this idea, we define the predicate

$$R^{\circ}_A(S, X) = \{q \mid p \in S, \rho \in \mathsf{Paths}_A(p, q), \pi(\rho) = \varepsilon, \rho \cap X = \varnothing\}.$$

That is, $R^{\circ}_A(S, X)$ is the set of states reachable from any state in S via an unlabelled path without passing through any state in X. We drop the subscript A when the automaton is clear from the context. Additionally, when $S = \{p\}$, we write $R^{\circ}(p, X)$ instead of $R^{\circ}(\{p\}, X)$ for convenience. Using this notation, we have that $\mathsf{Closure}(S) = R^{\circ}(S, \varnothing)$.

The following simple properties about the relation R° follow from its definition.

Lemma 4.10. Let $A = (\Delta, F)$ be an automaton, and let $p \in \mathbb{N}$ and $X \subseteq \mathbb{N}$. Then, the following properties hold:

1. If $p \in X$, then $R^\circ(p, X) = \varnothing$.

2. If $p \notin X$, and $\Delta(p) = \text{None}$, then $R^\circ(p, X) = \{p\}$.

3. If $p \notin X$, and $\Delta(p) = \mathsf{Char}(\sigma)$, then $R^\circ(p, X) = \{p\}$.

When we have a Jump state p, we can avoid exploring paths whose 'tails' pass through p. The following lemma makes this precise.

Lemma 4.11. Let $A = (\Delta, F)$ be an automaton, and let $p \in \mathbb{N}$ and $X \subseteq \mathbb{N}$ such that $p \notin X$ and $\Delta(p) = \mathsf{Jump}(q)$. Then, $R^\circ(p, X) = \{p\} \cup R^\circ(q, X \cup \{p\})$.

Proof. Using the definition of R°, we can already see that $\{p\} \cup R^\circ(q, X \cup \{p\}) \subseteq R^\circ(p, X)$. We focus on the other direction.

Suppose $r \in R^\circ(p, X)$. Then, there is a path $\rho \in \mathsf{Paths}(p, r)$ which does not pass through any elements of X. If the path ρ contains only one edge, then it must be q, and we would be done. If none of the later edges of ρ happen to be incident on p, then we are also done. Otherwise, let ρ' be the suffix of ρ starting from the last occurrence of p. Then, ρ' must have the form $\rho' = p \to q \to \cdots \to r$, where p does not occur in the path from q to r. Thus, $r \in \{p\} \cup R^\circ(q, X \cup \{p\})$. $\square$

The following lemma is crucial in justifying avoiding rexploration of parts of the graph which have already been visited.

Lemma 4.12. Let $S_1, S_2, X \subseteq \mathbb{N}$. Then, the following relation holds.

$$R^\circ(S_1 \cup S_2, X) = R^\circ(S_1, X) \cup R^\circ(S_2, X) = R^\circ(S_1, X) \cup R^\circ(S_2, X \cup R^\circ(S_1, X))$$

Proof. The first equality is trivial and follows from the definition of R°. As for the second equality, notice first that since $X \subseteq X \cup R^\circ(S_1, X)$, it is clear that

$R^\circ(S_2, X \cup R^\circ(S_1, X)) \subseteq R^\circ(S_2, X)$. To prove the conclusion, it is enough to show that $R^\circ(S_2, X) \subseteq R^\circ(S_1, X) \cup R^\circ(S_2, X \cup R^\circ(S_1, X))$.

Take $r \in R^\circ(S_2, X)$, and let $\rho \in \mathsf{Paths}(p, r)$ be a path which does not pass through any elements of X. If ρ does not pass through $R^\circ(S_1, X)$, then we have $r \in R^\circ(S_2, X \cup R^\circ(S_1, X))$, and we are done. Let us examine the case when ρ does pass through $R^\circ(S_1, X)$. Let p be a vertex of ρ such that $p \in R^\circ(S_1, X)$. Since $p \in R^\circ(S_1, X)$, there is a path $\rho_1 \in \mathsf{Paths}(u, p)$ for some $u \in S_1$ which does not pass through any elements of X. Consider the suffix ρ_2 of ρ starting from an occurrence of p. Then, ρ_2 is an element of $\mathsf{Paths}(p, r)$ which does not pass through any elements of X. Concatenating the paths, we have $\rho_1 \cdot \rho_2 \in \mathsf{Paths}(u, r)$ which does not pass through any elements of X. Thus, $r \in R^\circ(S_1, X)$. $\square$

This allows us to characterize the set $R^\circ(p, X)$ for Split states p.

Corollary 4.13. Let $A = (\Delta, F)$ be an automaton, and let $p \in \mathbb{N}$ and $X \subseteq \mathbb{N}$ such that $p \notin X$ and $\Delta(p) = \mathsf{Split}(q_1, q_2)$. Then, $R^\circ(p, X) = \{p\} \cup R^\circ(q_1, X \cup \{p\}) \cup R^\circ(q_2, X \cup R^\circ(q_1, X \cup \{p\}))$.

There is a difficulty that arises when formalizing Lemma 4.12 in Coq due to its constructive logic. The argument showing that $R^\circ(S_2, X) \subseteq R^\circ(S_1, X) \cup R^\circ(S_2, X \cup R^\circ(S_1, X))$ requires us to show whether the witness belongs in the left hand or the right hand side of the union. This in turn requires us to determine whether the witness path passes through $R^\circ(S_1, X)$ or not. We assume this is a part of the hypothesis for the lemma, and this hypothesis is supplied later when proving the correctness of DFS. Indeed, determining whether a path passes through $R^\circ(S_1, X)$ must involve computing the R° relation, which is done using DFS. The Coq formulation of this lemma is shown below.

```
**Function** DFS $(fuel : \mathbb{N})\ (u : \mathbb{N})\ (X : Set\,\mathbb{N}) : Set\,\mathbb{N}$ :=
    if $fuel = 0 \vee u \in X$ then
        $X$
    else
        let $X' = \{u\} \cup X$ in
        let $\texttt{fuel}' = \texttt{fuel} - 1$ in
        match $\Delta(u)$ with
            None $\rightarrow X'$
            Char$(\sigma) \rightarrow X'$
            Jump $v_1 \rightarrow DFS\ \texttt{fuel}'\ v_1\ X'$
            Split $v_1\ v_2 \rightarrow DFS\ \texttt{fuel}'\ v_2\ (DFS\ \texttt{fuel}'\ v_1\ X')$
        end
```

Figure 4.3 : Depth First Search Algorithm using a `fuel` parameter

```
Lemma vreach_avoid_incr_set (G : @Graph A) (Avd Src1 Src2 : nat -> Prop)
  (Vdec : forall v,
      {v ∈ (vreach_without G Src1 Avd)} + {~ v ∈ (vreach_without G Src1 Avd)}) :
  (vreach_without G Src1 Avd) ∪ (vreach_without G Src2 Avd)
  ≡ (vreach_without G Src1 Avd)
    ∪ (vreach_without G Src2 (Avd ∪ (vreach_without G Src1 Avd))).
```

The DFS algorithm is listed in Figure 4.3. To satisfy Coq's termination checker, an additional parameter `fuel` corresponding to the number of recursive calls necessary is supplied. This parameter is also useful in the proofs, as we will see in Lemma 4.14. In our formalization, the set X of states is represented as BArray `bool`, an array of $F + 1$ booleans, where the value `true` at index i indicates that the state $i \in X$. This enables checking whether $u \in X$ holds in $O(1)$ time.

The characteristic lemma about the correctness of DFS can be stated as follows.

Lemma 4.14. Let $A = (\Delta, F)$ be an automaton, and let $p \leq F$ and $X \subseteq \{0, 1, \ldots F\}$. Suppose that $\texttt{fuel} \geq (F+1) - |X|$. Then,

$$\texttt{dfs}(\texttt{fuel}, p, X) = X \cup R^{\circ}(p, X).$$

Proof. The proof proceeds by induction on `fuel`. If $\texttt{fuel} = 0$, then the function `dfs` simply returns X. But the condition of the hypothesis ensures that $X = \{0, 1, \ldots F\}$. By Lemma 4.10, we have that $R^{\circ}(p, X) = \varnothing$, and thus $X = X \cup R^{\circ}(p, X)$. The case where $\texttt{fuel} > 0$ and $p \in X$, the reasoning is similar, since we still return X. Below, we assume that $\texttt{fuel} > 0$ and $p \notin X$.

Consider the cases where p is a None or $\mathsf{Char}(\sigma)$ state. As noted in Lemma 4.10, we have that $R^{\circ}(p, X) = \{p\}$ in this cases. Indeed, the function returns $X \cup \{p\} = X \cup R^{\circ}(p, X)$ in these cases.

Suppose $\Delta(p) = \mathsf{Jump}(q)$. In this case, we define $X' = X \cup \{p\}$, and the algorithm returns $\texttt{dfs}(\texttt{fuel} - 1, q, X')$. Indeed, $|X'| = |X| + 1$ and hence the hypotheses $\texttt{fuel} - 1 \geq (F+1) - |X'|$ holds. Since $\texttt{fuel} - 1 < \texttt{fuel}$, we can invoke the inductive hypothesis. By the inductive hypothesis, we have that $\texttt{dfs}(\texttt{fuel} - 1, q, X') = X' \cup R^{\circ}(q, X')$. By Lemma 4.11, this is indeed what we want.

When $\Delta(p) = \mathsf{Split}(q_1, q_2)$, we define $X' = X \cup \{p\}$. By the induction hypothesis, we have that $\texttt{dfs}(\texttt{fuel} - 1, q_1, X') = X' \cup R^{\circ}(q_1, X')$ and $\texttt{dfs}(\texttt{fuel} - 1, q_2, \texttt{dfs}(\texttt{fuel} - 1, q_1, X')) = \texttt{dfs}(\texttt{fuel} - 1, q_1, X') \cup R^{\circ}(q_2, \texttt{dfs}(\texttt{fuel} - 1, q_1, X'))$. Using Lemma 4.12, we notice that this is indeed $X \cup R^{\circ}(p, X)$. □

We write $\texttt{dfs}(p, X)$, dropping the `fuel` argument to mean $\texttt{dfs}(F+1, p, X)$, i.e, giving the DFS algorithm the maximum needed fuel.

The time taken by the function `dfs` can be described as follows. This lemma can be proven by induction on $(F+1) - |X|$, the number of states in the automaton which have not been visited yet. An important aspect of the proof that allows us to keep the linear complexity in the number of states is that each of them has at most two outgoing edges. If we allowed an unstructured automaton with an arbitrary number of outgoing transitions, the complexity would be $O(F + |\Delta|)$, where $|\Delta|$, the number of transitions, could be as large as $O(F^2)$.

Lemma 4.15. Let $A = (\Delta, F)$ be an automaton, and let $p \leq F$ and $X \subseteq \{0, 1, \dots F\}$. Then, the time taken to compute function `dfs`(p, X) is $O(F - |X|)$.

While the `dfs` function above takes one vertex as input, it could be called repeatedly to compute $R^{\circ}(S, \varnothing)$ for a finite set of vertices S. Given a list of vertices $p_1, p_2 \dots p_n, p_{n+1}$, we define the set $\underline{R}(\dots)$ as follows. Define $\underline{R}() = \varnothing$, and $\underline{R}(p) = R^{\circ}(p, \varnothing)$. For $n \geq 1$, define

$$\underline{R}(p_1, p_2, \dots p_n, p_{n+1}) = \underline{R}(p_1, \dots p_n) \cup R^{\circ}(p_{n+1}, \underline{R}(p_1, \dots p_n)).$$

Lemma 4.16. Let $p_1, p_2, \dots p_n$ be a list of states. Then,

$$\underline{R}(p_1, p_2, \dots p_n) = R^{\circ}(\{p_1, p_2, \dots p_n\}, \varnothing).$$

Proof. The proof is by induction on n, the number of elements in the list. When $n = 0$ or $n = 1$, the result is clear by unfolding the definition of $\underline{R}$.

By way of induction, let us assume that $\underline{R}(p_1, p_2, \dots p_n) = R^{\circ}(\{p_1, p_2, \dots p_n\}, \varnothing)$. Instantiating Lemma 4.12 with $S_1 = \{p_1, p_2, \dots p_n\}$ and $S_2 = \{p_{n+1}\}$, and $X = \varnothing$, we find that

$$R^{\circ}(\{p_1, p_2, \dots p_n, p_{n+1}\}, \varnothing) = R^{\circ}(\{p_1, p_2, \dots p_n\}, \varnothing) \cup R^{\circ}(p_{n+1}, R^{\circ}(\{p_1, p_2, \dots p_n\}, \varnothing)).$$

We may use the inductive hypothesis to replace $R^{\circ}(\{p_1, \dots p_n\}, \varnothing)$ with $\underline{R}(p_1, \dots p_n)$, and get

$$R^{\circ}(\{p_1, p_2, \dots p_n, p_{n+1}\}, \varnothing) = \underline{R}(p_1, p_2, \dots p_n) \cup R^{\circ}(p_{n+1}, \underline{R}(p_1, p_2, \dots p_n)).$$

We observe that the right hand side is exactly $\underline{R}(p_1, p_2, \ldots p_n, p_{n+1})$, and this completes the proof. $\square$

This gives us a way to repeatedly apply `dfs` to compute $R^\circ(S, \varnothing)$ for a finite set of states S. The proof of the following theorem can be given by first using Lemma 4.14 to notice that the chained `dfs` calls are equivalent to the $\underline{R}$ function, and then using Lemma 4.16.

Theorem 4.17. Let $S = \{p_1, p_2, \ldots p_n\}$ be a finite set of states. Then,

$$\mathsf{Closure}(S) = R^\circ(S, \varnothing) = \texttt{dfs}(p_n, \ldots \texttt{dfs}(p_2, \texttt{dfs}(p_1, \varnothing)) \ldots).$$

This computation can be easily expressed as a fold as shown below.

```
Definition dfsMany (lsrc : list nat) : BSet :=
  fold_left (fun SS u => dfs (1 + (final M)) u SS) lsrc (emptyBSet (S (final M))).
```

The time required to compute $\mathsf{Closure}(S)$ using `dfsMany` is linear in F. This is because with each call of `dfs`, the number of 'unvisited states' in X decreases, and this decreases the total amount of work done across all calls of `dfs`.

Theorem 4.18. Let $S = \{p_1, p_2, \ldots p_n\}$ be a finite set of states. Then, the time taken to compute $\mathsf{Closure}(S)$ by repeated calls of `dfs` (as shown in Theorem 4.17) is $O(F)$.

As discussed earlier in this section, it is more efficient to work with the set ReachG and $\mathsf{ClosureG}$ instead of Reach and $\mathsf{Closure}$, respectively. Thus, we modify the `dfs` function so that it produces the list $\mathsf{ClosureG}$ directly while computing $\mathsf{Closure}$. In the modified function, there is an additional argument supplied to `dfs` which stores only the Char and None states in a list. In the skeleton code provided below, the function `dfsMany` computes $\mathsf{Closure}$, while the function `dfsManyG` computes $\mathsf{ClosureG}$.

```
Fixpoint dfsInner (fuel : nat) (u : nat)
    (X : BSet) (qs : list nat) : (BSet * list nat) :=
  ...

Definition dfsManyInner (lsrc : list nat) : BSet * list nat :=
  fold_left (fun '(SS, gs) u => dfsInner
    (1 + final M) u SS gs) lsrc (emptyBSet (1 + final M), []).

Definition dfsMany (lsrc : list nat) : BSet :=
  fst (dfsManyInner lsrc).

Definition dfsManyG (lsrc : list nat) : list nat :=
  snd (dfsManyInner lsrc).
```

4.5 Maximal Munch Tokenization

The process of *tokenization* or *lexing* is to partition a string into a stream of *tokens*. Typically, a list of regular expressions is used to specify what the tokens could be. When multiple regular expressions could match, the following rule is used to disambiguate.

Definition 4.19 (Maximal Munch Tokenization)**.** Let R be a list of n regular expressions $r_0, \ldots, r_{n-1} \in \mathsf{Reg}$, and $w \in \Sigma^*$ be a string. We say that $\mathsf{maxMunch}(R, w) =$ Some(i, j) if $w[0, i] \in [\![r_j]\!]$ and additionally the following conditions hold:

1. No prefix longer than $w[0, i]$ is recognized, i.e, for any $i' > i$, $w[0, i'] \notin [\![r_k]\!]$ for any $0 \leq k \leq n$.

2. The expression r_j is the first expression that recognizes $w[0, i]$, i.e., for any $0 \leq k < j$, $w[0, i] \notin [\![r_k]\!]$.

If no such (i,j) exist, then $\mathsf{maxMunch}(R,w) = \text{None}$. In this case, we say that w cannot be tokenized by R, or that the maximal munch of w with respect to R is not defined.

A tokenization of w with respect to a list of regular expressions R is an ordered list of pairs $(j_1,t_1),\ldots,(j_k,t_k)$ such that $w = t_1 \cdot t_2 \cdots t_k$ and for all $1 \le \alpha \le k$, $\mathsf{maxMunch}(R, t_\alpha t_{\alpha+1} \cdots t_k) = \text{Some}(|t_\alpha|, j_\alpha)$. In other words, the tokenization is obtained by collecting every maximal prefix. Note that the tokenization may not necessarily exist.

The lexer takes a list of regular expressions R, each describing a potential token, and a string w, and splits w into $w[0,i] \cdot w[i,|w|]$ such that $\mathsf{maxMunch}(R,w) = \text{Some}(i,j)$ for some j. Executing the lexer repeatedly on the suffix $w[i,|w|]$ in this manner gives us a stream of tokens which can be used for further processing.

Example 4.20. In the case of a string representing a JSON object [152], the set of tokens include strings, numbers, three literal names (`true`, `false` and `null`), six structural characters (`[`, `]`, `{`, `}`, `,` and `:`), and whitespace (consisting of spaces, tabs, newlines and carriage returns). Each of these tokens, including strings and numbers, could be expressed as a regular expression. For example, the string `[{"abc":-1.2}]` would be tokenized as the tokens `[`, `{`, `"abc"`, `:`, `-1.2`, `}` and `]`. These tokens have the types `begin-array`, `begin-object`, `string`, `name-separator`, `number`, `end-object` and `end-array` respectively. Knowing the types of these tokens would allow the parser consuming them to construct the appropriate tree represented by the JSON serialization.

Example 4.21. Let $R = [r_1, r_2] = [ba + aa, aab^*]$. The automata $\mathcal{A}(r_1)$ and $\mathcal{A}(r_2)$ are shown in Example 4.4. Let $w = aabaa$ be the string being tokenized. This string would be tokenized as $(2, aab), (1, aa)$. Note that the first token is $\underline{aab}aa$ (by matching r_2) rather than $\underline{aa}baa$ (by matching r_1) since the former option is longer. For the second token, $aab\underline{aa}$ is matched by both r_1 and r_2. We classify this token as type 1 instead of 2 since the maximal-munch rule prioritizes earlier options.

If the input string was $w' = aaba$ instead, the maximal munch tokenization would not exist. Note that $(1, aa), (2, ba)$ is not a maximal-munch tokenization, since selecting $\underline{aa}baa$ would violate the maximal munch principle.

In the remainder of this section, we demonstrate how the $\mathsf{maxMunch}(R, w)$ function can be computed by simulating the Thompson NFA constructed in the previous sections. The main idea is the following: Given $R = \langle r_1, r_2, \ldots, r_n \rangle$, we construct the corresponding automata $A = \langle m_1, m_2, \ldots m_n \rangle$; we maintain a list of sets of *active states* $S = \langle S_1, S_2, \cdots S_n \rangle$ and update each S_i when we read a character from the input string. This retains enough information to determine if any of the expressions r_i have recognized the consumed prefix.

Definition 4.22 (Lexer Configuration)**.** A *lexer configuration* C is an ordered tuple (ℓ, τ, β) where $\ell \in \mathbb{N}$, $\beta \in \mathsf{Option}(\mathbb{N} \times \mathbb{N})$ and $\tau \in \mathsf{list}\ (\mathbb{N} \times \mathcal{A} \times \mathsf{list}\ \mathbb{N})$. The value ℓ indicates the length of the string fed in so far, β indicates the maximal munch so far, and τ contains the list of active states for each automaton. Each element of τ is of the form (i, m_i, S_i) where m_i is the automaton corresponding to the expression r_i of R, and S_i is the set of 'active states' $\mathsf{ReachG}_{m_i}(w)$.

In Figure 4.4, we define certain invariants which express propositions relating the configuration to the input string w. Clearly, if $\chi(C, w)$ holds, then $\mathsf{maxMunch}(R, w) = \beta$. Thus, we define $\mathsf{initCfg}$ such that $\chi(\mathsf{initCfg}, \varepsilon)$ holds, and $\mathsf{stepCfg}$ such that if $\chi(C, w)$ holds, then $\chi(\mathsf{stepCfg}(C, a), w \cdot a)$ holds.

$$\chi_{\mathsf{len}}(\ell, w) \triangleq \ell = |w|$$

$$\chi_{\beta}(\beta, w) \triangleq \beta = \mathsf{maxMunch}(R, w)$$

$$\chi_{\mathsf{idx}}(\tau) \triangleq \forall (i, m_i, S_i) \in \tau, m_i = \mathcal{A}(r_i)$$

$$\chi_{\mathsf{sorted}}(\tau) \triangleq \forall i < j,\ \text{if } \tau[i] = (\alpha, m_\alpha, S_\alpha) \text{ and } \tau[j] = (\beta, m_\beta, S_\beta),\ \text{then } \alpha < \beta$$

$$\chi_{\mathsf{Reach}}(\tau, w) \triangleq \forall (i, m_i, S_i) \in \tau,\ \text{then } S_i = \mathsf{ReachG}_{m_i}(w)$$

$$\chi_{\mathsf{missing}}(\tau, w) \triangleq \forall i,\ \text{if } (i, m_i, \_) \notin \tau,\ \text{then } \mathsf{ReachG}_{m_i}(w) = \varnothing$$

$$\chi_{\mathsf{threads}}(\tau, w) \triangleq \chi_{\mathsf{idx}}(\tau) \wedge \chi_{\mathsf{sorted}}(\tau) \wedge \chi_{\mathsf{Reach}}(\tau, w) \wedge \chi_{\mathsf{missing}}(\tau, w)$$

$$\chi(C, w) \triangleq \chi_{\mathsf{len}}(\ell, w) \wedge \chi_{\beta}(\beta, w) \wedge \chi_{\mathsf{threads}}(\tau, w),\ \text{where } C = (\ell, \beta, \tau)$$

Figure 4.4 : Invariants for the Lexer Configuration

We define the function firstAccept on τ to be the first i such that $(i, m_i, S_i) \in \tau$ and $F_{m_i} \in S_i$. The function bmUpdate is used to calculate the maximal munch for the updated string $w{\cdot}a$. We show below their Coq definitions. The function `find` is taken from the Coq standard library, which produces the first element of a list satisfying a predicate by scanning it in a linear manner.

```
Definition firstAccept
  (threads : list (nat * automaton * list nat)) : option nat :=
  match find (fun '(i, mi, si) => existsb (fun q => q =? final mi) si) thrds with
  | None => None
  | Some (i, _, _) => Some i
  end.

Definition bestMatchUpdate (oldBM : option (nat * nat))
  (strPtr : nat) (new : option nat) : option (nat * nat) :=
  match new with
  | None => oldBM
  | Some j => Some (strPtr, j)
  end.
```

Lemma 4.23. Suppose τ' is such that $\chi_{\mathsf{threads}}(\tau', w \cdot a)$, and β is such that $\chi_\beta(\beta, w)$ holds. Define

$$\beta' = \mathsf{bmUpdate}(\beta, |w \cdot a|, \mathsf{firstAccept}(\tau')).$$

Then the following hold:

1. If $\mathsf{firstAccept}(\tau') = \text{None}$, then for all r_j, $w \cdot a \notin [\![r_j]\!]$.
2. If $\mathsf{firstAccept}(\tau') = \text{Some}(j)$, then j is the least value such that $w \cdot a \in [\![r_j]\!]$.
3. $\chi_\beta(\beta', w \cdot a)$ holds.

Proof. For any automaton $A = (\Delta, F)$, we know that $w \in [\![A]\!]$ if and only if $F \in \mathsf{Reach}_A(w)$. The invariant $\chi_{\mathsf{Reach}}(\tau', w \cdot a)$ implies that for each $(j, m_j, S_j) \in \tau'$, $S_j = \mathsf{ReachG}_{m_j}(w \cdot a)$. Thus, if the function $\mathsf{firstAccept}$ did not find any m_j in τ' such that $F_{m_j} \in \mathsf{ReachG}_{m_j}(w \cdot a)$, then $w \cdot a \notin [\![m_j]\!]$ for any m_j among those which are present in τ'. Additionally, the invariant $\chi_{\mathsf{missing}}(\tau', w \cdot a)$ implies that for each j such that $(j, m_j, \_)$ do not appear in τ', $\mathsf{ReachG}_{m_j}(w \cdot a) = \varnothing$. This means that $w \cdot a \notin [\![m_j]\!]$ for any m_j which are not present in τ'. Thus, the first condition holds.

The proof of the second condition is similar, and makes use of the invariants χ_{Reach} and χ_{missing}. Additionally, the invariant χ_{sorted} is needed to establish that the first automaton found while scanning the list τ' indeed has the least possible value of j.

The third part is a direct consequence of the definition of the maximal munch. If $w \cdot a \notin [\![r_j]\!]$ for any j, then $\mathsf{maxMunch}(R, w \cdot a) = \mathsf{maxMunch}(R, w)$. Otherwise, if j is the least value such that $w \cdot a \in [\![r_j]\!]$, then $\mathsf{maxMunch}(R, w \cdot a) = \text{Some}(|w \cdot a|, j)$. $\square$

Next, we define the function $\mathsf{stepThreads}$ which updates the list of active states in τ given a character a. While doing so, it drops the tuples (i, m_i, S_i) for which $S_i = \varnothing$, since these cannot accept any further extensions of the current input string. Given the character $a \in \Sigma$, the function $\mathsf{stepCfg}$ produces the updated configuration corresponding to the extended string $w \cdot a$.

```
Definition stepThreads (a : A) (oldThreads : list (nat * @automaton A * list nat))
  : list (nat * @automaton A * list nat) :=
    filterMap ( fun '(i, m, states) =>
      let new_states := dfsMany m (advance (delta m) states a) in
      if nilb new_states then None else Some (i, m, new_states)
    ) oldThreads .
```

Lemma 4.24. The function $\mathsf{stepThreads}$ preserves χ_{threads}. Suppose $\chi_{\mathsf{threads}}(\tau, w)$ holds and $\tau' = \mathsf{stepThreads}(a, \tau)$. Then $\chi_{\mathsf{threads}}(\tau', w \cdot a)$ holds.

Proof. We show why each of χ_{idx}, χ_{sorted}, χ_{Reach} and χ_{missing} hold for τ'.

Since the fields i, m_i in each entry of the list τ do not change, χ_{idx} continues to hold. Similarly, since the list τ is sorted, and the function `filterMap` does not change the order of the elements, χ_{sorted} continues to hold for τ'.

For each $(i, m_i, S_i) \in \tau$, the we have $S_i = \mathsf{ReachG}_{m_i}(w)$ by the invariant χ_{Reach}. Using Lemma 4.8, we know that $\mathsf{ReachG}_{m_i}(w \cdot a) = \mathsf{Closure}(\mathsf{adv}(a, S_i))$. This shows that χ_{Reach} holds for τ'.

The function `filterMap` drops the entries (i, m_i, S_i) for which $S_i = \varnothing$. This is consistent with the invariant χ_{missing}. Additionally, consider the cases where (i, m_i, S_i) was already missing in τ. Then, $\mathsf{ReachG}_{m_i}(w) = \varnothing$, and one can check that $\mathsf{ReachG}_{m_i}(w \cdot a) = \varnothing$ as well. These entries are excluded from τ' as well, and thus χ_{missing} holds for τ'. □

In Fig 4.5, we describe the $\mathsf{initCfg}$ configuration and the $\mathsf{stepCfg}$ function, using the building blocks defined above. The following lemmas characterizing the $\mathsf{initCfg}$ configuration and the $\mathsf{stepCfg}$ function can be proven using Lemma 4.24.

Lemma 4.25. The following holds:

1. $\chi(\mathsf{initCfg}, \varepsilon)$ holds.

2. If $\chi(C, w)$ holds, then $\chi(\mathsf{stepCfg}(C, a), w \cdot a)$ holds.

The function $\mathsf{foldCfg}$ is used to compute the final configuration given a string w. The following theorem states that the $\mathsf{maxMunch}$ function in Figure 4.5 computes the maximal munch correctly.

Theorem 4.26. Let $w \in \Sigma^*$ be a string and $R = \langle r_1, r_2, \ldots, r_n \rangle$ be a list of regular expressions. Define the configuration $(\ell, \tau, \beta) = \mathsf{foldCfg}(w, \mathsf{initCfg})$. Then $\beta = \mathsf{maxMunch}(R, w)$.

Proof. We will prove by induction on w that the invariant χ holds for the configuration $(|w|, \tau, \beta)$. This is sufficient to prove the conclusion, since χ_β implies that $\beta = \mathsf{maxMunch}(R, w)$.

If $w = \varepsilon$, then this is immediate from Lemma 4.25. If $w = w' \cdot a$, and the invariant $\chi(C, w)$ held then, so does $\chi(\mathsf{stepCfg}(C, a), w \cdot a)$ by Lemma 4.25. Thus, the induction step holds when τ is nonempty.

Let us consider the case where τ is empty and our function $\mathsf{foldCfg}$ 'returns early'. Assume that χ holds for $(|w|, [], \beta)$. We wish to show that χ still holds for $(|w \cdot a|, [], \beta)$. The key observation is that using χ_{missing}, we know that for all i, $\mathsf{ReachG}_{m_i}(w) = \varnothing$. Thus, $\mathsf{ReachG}_{m_i}(w \cdot a) = \varnothing$ as well. This implies that neither the best match, nor the list of threads need to be updated. $\square$

```
// Lexer Configuration - $(\ell, \tau, \beta)$
type Cfg = $\mathbb{N} \times \mathsf{list}\ (\mathbb{N} \times \mathcal{A} \times \mathsf{list}\ \mathbb{N}) \times \mathsf{Option}(\mathbb{N} \times \mathbb{N})$
// Produces an initial lexer configuration
Function initCfg $(R : \mathit{list\ Reg}) : \mathsf{Cfg}$ :=
    // Use Thompson's Construction for each $r_i$ in $R$
    let $M = [(\Delta_i, F_i) \mid 0 \le i < |R|, (\Delta_i, F_i) = \mathcal{A}(R[i])]$ in
    // Compute $\mathsf{ReachG}_{m_i}(\{\varepsilon\})$ for each $m_i$
    let $\tau = [(i, m_i, \mathsf{ClosureG}_{m_i}(\{0\})) \mid 0 \le i < |R|, m_i = M[i]]$ in
    let $\beta$ = match $\mathsf{firstAccept}(\tau)$ with
        $\mathrm{None} \to \mathrm{None}$
        $\mathrm{Some}(j) \to \mathrm{Some}(0, j)$
    end in
    $(0, \tau, \beta)$
// Steps the configuration by consuming one character
Function stepCfg $(a : \Sigma)\ (C : \mathsf{Cfg}) : \mathsf{Cfg}$ :=
    // For each $(i, m_i, S_i) \in \tau$, define $S_i' = \mathsf{ClosureG}_{m_i}(\mathsf{adv}(a, S_i))$
    // $\tau'$ contains the tuples $(i, m_i, S_i')$
    // except those for which $S_i' = \varnothing$
    let $\tau' = \mathsf{stepThreads}\ a\ (\tau\ C)$ in
    let $\ell' = 1 + (\ell\ C)$ in
    // if a new match has been found, update $\beta$
    let $\beta' = \mathsf{bmUpdate}\ (\beta\ C)\ \ell'\ (\mathsf{firstAccept}\ \tau')$ in
    $(\ell', \tau', \beta')$
// Steps the configuration character-by-character
Function foldCfg $(w : \Sigma^*)\ (C : \mathsf{Cfg}) : \mathsf{Cfg}$ :=
    // If $w = \varepsilon$, we are done
    // If $\tau$ is empty, no active states in any automata remain
    if $w = \varepsilon \lor \tau\ C = []$ then
        $C$
    else
        let $a \cdot w' = w$ in foldCfg $w'$ (stepCfg $a$ $C$)
// Computes the maximal munch
Function maxMunch $(R : \mathit{list\ Reg})\ (w : \Sigma^*) : \mathsf{Option}(\mathbb{N} \times \mathbb{N})$ :=
    let $C = \mathsf{initCfg}\ R$ in
    $\beta$ (foldCfg $w$ $C$)
```

Figure 4.5 : Algorithm for Computing the first token using the *maximal-munch* principle

Table 4.1 : Evolution of the lexer configuration for $\langle ba + aa, aab^* \rangle$ on $aabaa$

Consumed Prefix	ε	a	aa	aab	$aaba$	$aabaa$
ℓ	0	1	2	3	4	5
τ	$(\mathcal{A}_1, [1,4])$ $(\mathcal{A}_2, [0])$	$(\mathcal{A}_1, [5])$ $(\mathcal{A}_2, [1])$	$(\mathcal{A}_1, [6])$ $(\mathcal{A}_2, [3,5])$	$(\mathcal{A}_2, [3,5])$	$\varnothing$	$\varnothing$
β	None	None	Some$(2,1)$	Some$(3,2)$	Some$(3,2)$	Some$(3,2)$

Example 4.27. Let $R = \langle r_1, r_2 \rangle = \langle ba+aa, aab^* \rangle$. The automata $\mathcal{A}(r_1)$ and $\mathcal{A}(r_2)$ are shown in Example 4.4. We show in Table 4.1 the first token of $w = \underline{aab}aa$ is obtained by our algorithm.

When the consumed prefix was ε or just a, we had β = None, since neither of the regexes matched. When the prefix was aab, both r_1 and r_2 were matched, which could be known by checking that the final states 6 and 5 of the automata $\mathcal{A}(r_1)$ and $\mathcal{A}(r_2)$ respectively were in the set of 'active states'. This is when the best match β found so far is also updated. Upon consuming the next b, there are no remaining active states in the first automaton, so it is dropped from τ. The value of β is updated as well to reflect the new match. When the next letter is consumed, there are no active tokens left in either automata. Thus, the foldCfg function would not process the rest of the string and return early.

The following theorem characterizes the time complexity of the foldCfg function, and thus the time it requires to compute maxMunch(R, w).

Theorem 4.28. Let $R = \langle r_1, r_2, \ldots, r_n \rangle$ be a list of regular expressions, and let $w \in \Sigma^*$ be a string. Let $s = |r_1| + \cdots + |r_n|$ be the sum of the sizes of the regular expressions in R. Then, the time taken to compute maxMunch(R, w) using foldCfg is $O(s \cdot |w|)$.

Proof. Using Lemma 4.7, each of the automata $\mathcal{A}(r_i)$ have size $O(|r_i|)$. Each step of the function foldCfg consists of calling stepThreads, which involves computing ClosureG for each automaton. Theorem 4.18 shows that the time taken to compute ClosureG is $O(|r_i|)$ for each i. Additional time is necessary to check each entry in τ and discard the empty ones. This takes time $O(n)$, where n is the number of expressions in R, and is also $O(s)$. Thus, the time taken to compute stepThreads is $O(s)$. Iterating this over $|w|$ characters takes $O(s \cdot |w|)$ time. □

This procedure can be used in a straightforward manner to compute the maximal munch tokenization of a given string.

Corollary 4.29. Let $R = \langle r_1, r_2, \ldots, r_n \rangle$ be a list of regular expressions, and let $w \in \Sigma^*$ be a string. Suppose s is the sum of the sizes of the regular expressions in R. Then, the *maximal-munch* tokenization, as defined in Definition 4.19, can be computed by repeatedly using foldCfg in $O(s \cdot |w|^2)$ time.

Proof. The procedure consists of calling foldCfg to determine each token. If $w = \varepsilon$, we are done. If $\beta = \text{None}$ or if $\beta = \text{Some}(0, j)$, the string cannot be tokenized. Otherwise, if $\beta = \text{Some}(i, j)$, then we add the token $(j, w[0, i])$ to the stream of tokens and recurse on the remaining string $w[i + 1, j]$.

Theorem 4.28 shows that the time taken to compute maxMunch(R, w) is $O(s \cdot |x|)$, when the string supplied is x. Since with each call to foldCfg, we shave off one character from w, there are at most $|w|$ calls to foldCfg. Each of these calls use a string of size $\leq |w|$. Thus, the time needed to compute the tokenization is $O(s \cdot |w|^2)$. □

Example 4.30. Suppose we have $R = \langle ab, (ab)^*\# \rangle$, and the input is $(ab)^n$ for some $n \in \mathbb{N}$. Each call of foldCfg would scan the entire string each time to see if $(ab)^*\#$ can be matched and take time proportional to the size of the string. Since no match of $(ab)^*\#$ would be found, each token would be ab (of size 2) and the tokenization algorithm would have to scan the remainder of the string again. Thus, this algorithm would elicit the worst-case running time of $O(n^2)$.

4.5.1 Pre-computing results of Depth-First Search

The function $\mathsf{stepThreads}$ requires computing ReachG for each set of active states, which is usually done using the `dfsMany` function explained in Section 4.4.1. In practice, it is very often the case that the set of active states is very often either a singleton or a set containing two elements. Thus, for these two cases, we precompute the results of DFS and store them in arrays, by utilizing a technique similar to Section 4.3.1. With this modification, the each element of the list τ in the lexer configuration has the following form: $(j, m_j, a_j^1, a_j^2, S_j)$. The entries j, m_j and S_j are as before, while the arrays a_j^1 and a_j^2 satisfy the following properties: $a_j^1[p] = \mathsf{ClosureG}(\{p\})$ and $a_j^2[p][q] = \mathsf{ClosureG}(\{p, q\})$. Thus, while computing $\mathsf{ClosureG}$, if we encounter a set of size ≤ 2, we can look up the result in the arrays a_j^1 and a_j^2 directly.

4.6 Experimental Results

We conducted experiments to empirically evaluate the performance of our verified lexer, in comparison with other verified and non-verified tools. In particular, for verified tools, we consider Coqlex [148], Verbatim [122]. For non-verified lexer generators, we consider Flex [146], and Ocamllex [153]. Further, both Coqlex and Verbatim are based on Brzozowski derivatives [100]. Flex and OCamllex are based on DFAs, and Verbatim uses an optimization that constructs a partially complete DFA before beginning lexing. Our tool is the only one which is based on Thompson NFAs.

For realistic benchmarks, we consider the JSON ruleset, which distinguishes be-

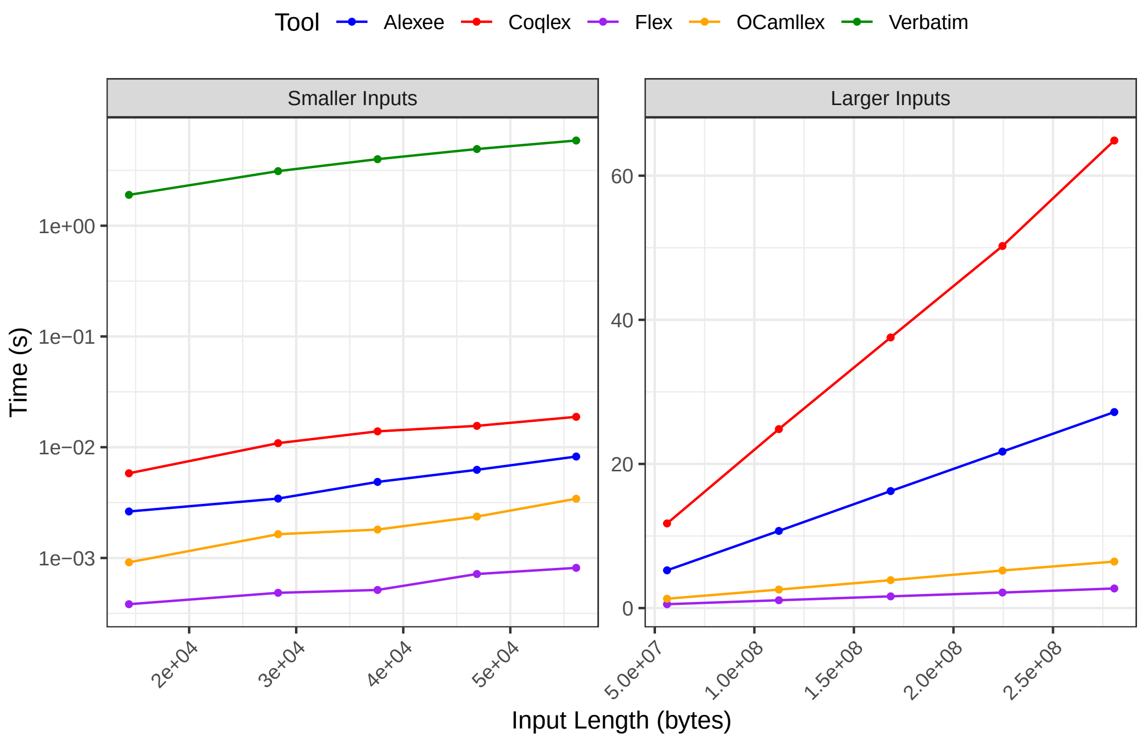


Figure 4.6 : Performance of Alexee, Coqlex, Verbatim, Flex, and Ocamllex on JSON tokenization

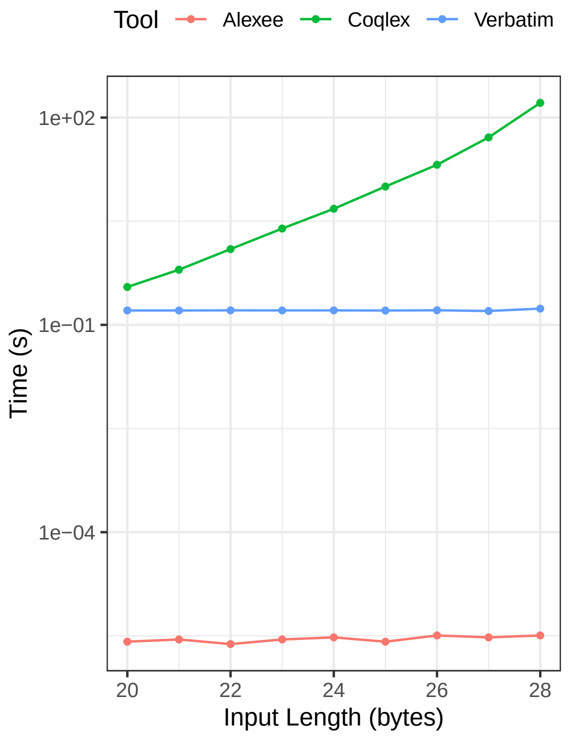
tween the different tokens in a JSON file (see Example 4.20). Following the evaluation in [148] and [122], the tools are evaluated on the gross domestic product (GDP) data retrieved from World Bank Open Data [154]. Note that in order to demonstrate the growth of running time with respect to input text length, we manipulated the JSON file to create inputs of various lengths. To produce smaller inputs, we have taken a prefix of the input data. For larger inputs, we have copied the input data multiple times to produce larger strings.

Figure 4.6 shows the performance of the five tools on the GDP JSON data. Each data point shown in the plots is the average of three trials. The standard deviation of the three trials is almost negligible across the board and is thus omitted. We noticed that Verbatim frequently ran into segmentation faults when given large inputs that the other tools were able to reasonably handle. Thus, the subfigure on the left, which is plotted using a log-scale transformation on the y-axis, involves smaller inputs on which Verbatim was able to process the entire input. On the other hand, to better distinguish the performance of the other four tools, the subfigure on the right involves larger inputs, and the difference in performance can be observed clearly.

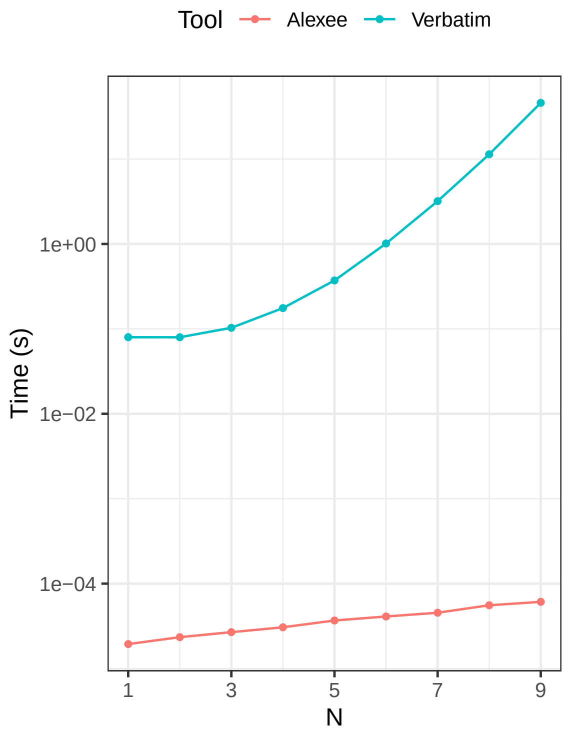
We observe that all five tools exhibit linear growth of running time with respect to input length. However, Verbatim is much slower than the other four. Despite being slower than the non-verified tools, which is to be expected, with a throughput of around 10 MB/sec, Alexee is roughly twice as fast as Coqlex. The throughputs of Flex, OCamllex, Coqlex and Verbatim are approximately 103 MB/sec, 44 MB/sec, 4 MBps 4 MB/sec and 9 KB/sec respectively.

The two microbenchmarks (shown in Fig 4.7) are based on synthetic regexes which elicit the worst-case behavior in Coqlex and Verbatim. The first microbenchmark (Fig 4.7a) uses the ruleset $[(a^*b?)^*, a, b]$ on the input $a^N b$, with varying values of N. Noting that the y-axis is in log scale, we observe that Coqlex takes an exponential amount of time (in terms of N) to tokenize this string. This behavior occurs because taking a-derivatives of the regex $(a^*b?)^*$ causes it to grow exponentially. Indeed, we have $\partial_a[(a^*b?)^*] = (a^*b?) \cdot (a^*b?)^*$, and $\partial_a[(a^*b?) \cdot (a^*b?)^*] = (a^*b?) \cdot (a^*b?)^* + (a^*b?) \cdot (a^*b?)^*$. Thus, the derivative of the regex using the string a^{N+1} would have 2^N copies of $(a^*b?) \cdot (a^*b?)^*$. This behavior does not show up in Verbatim since it prepares a (partially complete) DFA. Similarly, our tool does not suffer from this worst-case behavior since it is based on NFA simulation.

The second microbenchmark (shown in Fig 4.7b) considers a ruleset with a single



(a) Lexing Time for the string $a^n b$ on the ruleset $\langle (a^* b?)^*, a, b \rangle$



(b) Precomputation Time for the ruleset $\langle (a+b)^* a (a+b)^N \rangle$

Figure 4.7 : Two microbenchmarks comparing Alexee, Verbatim and Coqlex

lexical rule $\langle (a+b)^\star a(a+b)^N \rangle$, for varying values of N. The figure shows the precomputation time taken by Alexee and Verbatim. In the case of Alexee, this time is used to compute the Thompson NFA and some results of the depth-first search (see 4.5.1). In the case of Verbatim, the tool computes a partial DFA table where the states of the DFA correspond to derivatives. Since Coqlex does not have any particular precomputation step, we exclude it from this comparison. In the case of our tool, the precomputation time was obtained by instrumenting our code, and in the case of Verbatim, the precomputation time was obtained by parsing the two character string *ab*. We notice that the time taken to finish the necessary precomputation for Verbatim grows exponentially with the size of the regex. This is because the minimal DFA for this family of regexes is of size $\Omega(2^n)$. For the regex $(a+b)^\star a(a+b)^9$, Verbatim takes 45+ seconds to finish the preprocessing phase and segfaults for larger sizes.

Experimental Setup. The experiments were conducted on a MacBook Air M2 (8 cores) running Ventura 13.6.1 with 8 GB of RAM. We used flex version 2.6.4 and Apple clang version 14.0.3 for compilation. As required by the build instructions of Verbatim, it was built using version 4.10.2 of the OCaml compiler (Coq version 8.11). Version 4.14.0 of the OCaml compiler was used to build Coqlex, OCamllex and Alexee. Alexee and Coqlex were extracted using Coq version 8.19 and version 8.15, respectively.

4.7 Related Work

The maximal-munch rule mentioned in this paper is related to the notion of longest-leftmost match of POSIX [115] regular expressions. Early mentions of the 'maximal-munch' principle can be found in Cattell's thesis [155]. Yang et al. have discussed the applicability of this rule in the context of lexical analysis in [156].

A description of a lexing algorithm using automata such as ours can be found in standard books on compiler construction, such as [157]. We discussed that our algorithm may take quadratic time in the worst case. Reps [158] discusses an algorithm which executes this process in linear time by keeping track of pairs of positions in the string and states which do not lead to accepting states. This approach would require an additional $O(M \cdot n)$ bits to store this information, where M is the number of states of the DFA in consideration and n is the length of the string being tokenized.

A widely used approach for tools such as Flex [146] and Lex [159] is to behave as lexer generators, i.e, generate C code from a grammar which can be compiled down to an efficient lexer. In contrast, our tool behaves more like an interpreter since no code generation step is involved. The OCamllex tool [153] is also a lexer generator that emits OCaml code.

These tools are all based on deterministic automata (DFA), which has the advantage that one needs to maintain only one state identifier instead of a set of active states. For a grammar with regular expressions that are sufficiently small, this approach may be more efficient. Another lexer generator is RE2C [160] which is based on Tagged DFAs [161]. On the other hand, our approach is based on Thompson's NFA [149]. A notable regular expression engine (although not a lexer) that is based on this technique is Google's RE2 [95], whose implementation details are discussed on Russ Cox's website [99]. Other interesting work in the area of lexing includes Plex [162] which discusses how using a prescanning phase to identify token boundaries can be used to parallelize lexing and Chakravarty's work which discusses an approach to lexing based on combinators in a lazy functional programming language [163].

Nipkow [164] has provided a formalization of DFA-based lexing in Isabelle. While this formalization is mathematically sound, it leaves out some important implementation details: most importantly, while it formalizes a notion of determinization and removal of ε-transitions, these notions are based on an abstract formalism of sets rather than a concrete implementation. While we do not determinize, computing the ε-closure is indeed a non-trivial step in our formalism, as outlined in Section 4.4.1. As an intermediate step towards the DFA, Nipkow also builds the Thompson NFA. A difference here between this work and ours is the way the states are labelled: we use integer identifiers for our state labels, while a list of Booleans is used in Nipkow's formalization. The idea is that the additional bit in a concatenation or an union automaton is used to indicate the component of the automaton that is being used. A similar approach is used in the work of Doczkal et al. [132]: they use operations on finite types to represent the state space of the automata. For example, the state space of the union automaton is represented in their work as the sum type of the two finite types that represent the state spaces of the two underlying automata. While our approach produces a more efficient runtime representation, it makes the verification of the construction in Section 4.3 more challenging.

Recent efforts in verified lexing include Verbatim [147], Verbatim++ [122] and Coqlex [148]. All of these are based on Brzozowski derivatives [100] and thus face potential inefficiency arising from the size of the derivative. Verbatim++ improves upon its predecessor Verbatim by using a number of optimizations, which relate to the idea of memoizing the derivatives to avoid recomputation. However, as [148] notes, this still incurs a substantial overhead. Coqlex improves upon this mainly by managing the size of the derivatives in an efficient manner: they use smart constructors to simplify derivatives on the fly, and stop scanning the string early when the derivative happens to be the empty regular expression. Coqlex's approach to simplifying derivatives is still not immune to exponential blowup, however, as we have demonstrated in our experiments.

Since derivatives lend themselves to elegant manipulation in functional languages and are easier to verify, many verified implementations of regular expression matching are based on them. Coquand and Siles have formalized Brzozowski derivatives in [121]. A related approach is the use of partial derivatives [123] which have also been considered for formalization [125, 165, 124]. Derivatives have been used by Zhuchko et al. [1] to verify a matching procedure for regular expressions with lookaround in Lean, and by Urban and co-authors [129, 130, 131] to verify POSIX lexing based in Isabelle/HOL. Notable formalizations of regular languages that do not use derivatives include the work of Braibant and Pous [136] and Firsov and Uustalu [134], both of which use matrices.

Pottier [166] discusses a Coq-based formalization where DFS forests are defined and used to prove the correctness of Kosaraju's strongly connected components algorithm. Their work is parameterized using a runtime representation of the state of the algorithm, which is formalized as a record consisting of a type representing the data structure, a function returning the set of vertices (in a `V → Prop` representation), a function which checks the presence of a given vertex, and a function which adds a vertex to the set. Lammich and Neumann [167] have verified depth-first search algorithms in Isabelle/HOL. Their development is a framework on top of which different DFS-based algorithms can be built by specifying functions to 'hook' into certain extension points. They demonstrate this by verifying Tarjan's SCC algorithm in their framework. Chen and Levy [168] have produced a formal proof for the DFS algorithm in Why3 [169].

In Section 4.4.1, we used Boolean arrays to represent sets of vertices for DFS. Because of how we extract these arrays, set operations on them mutate them in place.

Inside Coq's representation, however, this is not how they behave, and thus we rely on ourselves to check that the older copies of the arrays are not reused. Sakaguchi [170] has developed a monadic domain-specific language that facilitates the use of computations using a mutable array, which requires the use of an additional plugin. Recently, the 'Functional-But-In-Place' paradigm has seen some interest, notably in the implementation of the Lean theorem prover [171, 172, 173]. In this framework, functional programs are analyzed to determine if older copies of data structures are reused, and if not, the program reuses the existing space instead of reallocating. This problem has also been studied under the title of the 'aggregate update problem' in older literature, such as [174].

Chapter 5

Conclusion

Temporal logic and regular expressions are two core formalisms used in a number of domains to specify properties of sequential data. Their usage ranges from searching for patterns in text files and tokenization of programs to the specification of properties of cyber-physical systems. Their significance and widespread use make it important for us to have trusted and efficient algorithms for their application. In this thesis, we have explored the use of proof assistants to formalize the semantics of certain aspects of temporal logic and regular expressions, and to verify the correctness of certain algorithms.

In Chapter 2, we have discussed a novel quantitative semantics for metric temporal logic and proposed an efficient algorithm for its online monitoring. We have presented a formalization in the Coq proof assistant of our semantics and verified the construction of these monitors.

In Chapter 3, we have proposed an efficient algorithm for the time matching of regular expressions with lookarounds. Existing engines that support lookarounds are based on backtracking, and can take exponential time to match in the worst case. Engines that are based on the construction of automata do not support lookarounds. Our novel algorithm has a linear time-complexity, and is based on a decomposition of the regular expression into subexpressions. We have verified the correctness of our algorithm in Coq.

In Chapter 4, we have investigated the problem of tokenization, the splitting of a string into tokens based on a ruleset described by regular expressions. We have formalized an efficient algorithm based on Thompson NFAs. The two main challenges in the verification effort are related to establishing the path lemmas for the Thompson construction, and the formalization of depth-first search. We have also proposed a simple way to modify the extraction process so that arrays are mutated in place.

We have extracted each of our verified algorithms as executable code, and evaluated their performance against other state-of-the-art tools. Our empirical results show that our algorithms obey the theoretical complexity bounds we have established, and are also competitive with existing tools on realistic benchmarks.

5.1 Future Work

While the performance of our formalized algorithms is generally competitive, they could be improved further using careful implementation techniques. While our extraction targets are Haskell or OCaml, one could produce a verified program in a lower-level language such as C using, for example, a toolchain such as Verified C [175]. This would allow us to reduce a number of overheads, such as the cost of memory allocation and garbage collection, and would also make it more suitable for deployment into resource-constrained environments such as embedded systems, microcontrollers or IoT devices.

The construction of our formalization of our verified monitors for temporal logic could be further streamlined using dataflow combinators (serial, parallel and feedback composition), as seen in [176, 45]. Our future work has extended the monitoring algorithm from lattice-based semantics to semiring-based semantics [36], and from past-only discrete time to bounded-future continuous time semantics [17]. These extended algorithms are yet to be verified in Coq. Another interesting direction would be to extend our quantitative semantics and monitoring algorithm to dynamic logic, as has been done in [90] in a qualitative setting.

Our work in [28] discusses a number of optimizations that could be applied to improve the performance of our algorithm for regular expressions with lookarounds for a wide range of practical scenarios. For example, in the case that a regular expression consists solely of lookaheads or lookbehinds, the evaluation could be done in a streaming manner in a single pass, and would require only an $O(m)$ amount of memory (instead of $O(m \cdot n)$). When the lookaheads are bounded and sufficiently small, they could be also eliminated by rewriting the expression. It would also be interesting to investigate the formalization of some of the aspects of the complexity results. This could be done by, for instance, establishing bounds on the size of the intermediate regular expressions. It may be possible to give a complete algebraic axiomatization of simple cases of lookaround (e.g., anchors and word boundaries) using the approach of Kleene algebra with extra equations [177, 140, 178, 179, 180]. The axiomatization of general lookaround is more challenging, as it can encode some kinds of intersection.

Industrial strength tools such as Flex and OCamllex remain faster than our verified tokenization tool, Alexee. Future work would involve incorporating lower-level tricks, such as code generation, which would improve throughput. In a number of practical scenarios where the NFAs are small enough, determinization would improve the throughput since the need for exploration using depth-first search would be eliminated. We also expect a number of realistic tokenization rulesets to use deterministic regular expressions [181, 182], in which case the determinization step would be unnecessary, and the depth-first search could also be eliminated. A related scenario would arise when the regular expressions express k-lookahead determinism [183], in which case looking at the next k characters would be sufficient to determine the next state.

Bibliography

[1] E. Zhuchko, M. Veanes, and G. Ebner, “Lean formalization of extended regular expression matching with lookarounds,” in *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2024, (New York, NY, USA), p. 118–131, ACM, 2024.

[2] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili, “Regular model checking,” in *Computer Aided Verification* (E. A. Emerson and A. P. Sistla, eds.), (Berlin, Heidelberg), pp. 403–418, Springer, 2000.

[3] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking,” *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.

[4] E. M. Clarke, O. Grumberg, and K. Hamaguchi, “Another look at LTL model checking,” *Formal Methods in System Design*, vol. 10, no. 1, pp. 47–71, 1997.

[5] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock, “QED at large: A survey of engineering of formally verified software,” *Foundations and Trends in Programming Languages*, vol. 5, p. 102–281, sep 2019.

[6] B. Hoxha, H. Abbas, and G. Fainekos, “Benchmarks for temporal logic requirements for automotive systems,” in *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems* (G. Frehse and M. Althoff, eds.), vol. 34 of *EPiC Series in Computing*, pp. 25–30, EasyChair, 2015.

[7] F. Cameron, G. Fainekos, D. M. Maahs, and S. Sankaranarayanan, "Towards a verified artificial pancreas: Challenges and solutions for runtime verification," in *Runtime Verification* (E. Bartocci and R. Majumdar, eds.), (Cham), pp. 3–17, Springer, 2015.

[8] G. Juniwal, A. Donzé, J. C. Jensen, and S. A. Seshia, "CPSGrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory," in *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, (New York, NY, USA), ACM, 2014.

[9] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, (USA), pp. 229–238, USENIX Association, 1999.

[10] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '06, (New York, NY, USA), pp. 93–102, ACM, 2006.

[11] T. T. Ngoc, T. T. Hieu, H. Ishii, and S. Tomiyama, "Memory-efficient signature matching for clamav on FPGA," in *2014 IEEE Fifth International Conference on Communications and Electronics (ICCE)*, pp. 358–363, 2014.

[12] I. Roy and S. Aluru, "Discovering motifs in biological sequences using the Micron Automata Processor," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 13, no. 1, pp. 99–111, 2016.

[13] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57, 1977.

[14] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, "Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications," in *Lectures on Runtime Verification* (E. Bartocci and Y. Falcone, eds.), vol. 10457 of *LNCS*, pp. 135–175, Cham: Springer, 2018.

[15] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Systems*, vol. 2, no. 4, pp. 255–299, 1990.

[16] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *FTRTFT 2004, FORMATS 2004* (Y. Lakhnech and S. Yovine, eds.), vol. 3253 of *LNCS*, (Heidelberg), pp. 152–166, Springer, 2004.

[17] K. Mamouras, A. Chattopadhyay, and Z. Wang, "A compositional framework for quantitative online monitoring over continuous-time signals," in *RV 2021* (L. Feng and D. Fisman, eds.), vol. 12974 of *LNCS*, (Cham), pp. 142–163, Springer, 2021.

[18] K. Mamouras, A. Chattopadhyay, and Z. Wang, "A compositional framework for algebraic quantitative online monitoring over continuous-time signals," *International Journal on Software Tools for Technology Transfer*, vol. 25, no. 4, pp. 557–573, 2023.

[19] S. C. Kleene, "Representation of events in nerve nets and finite automata," in *Automata Studies* (C. E. Shannon and J. McCarthy, eds.), no. 34 in Annals of Mathematics Studies, pp. 3–41, Princeton University Press, 1956.

[20] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM Journal of Research and Development*, vol. 3, no. 2, pp. 114–125, 1959.

[21] W. L. Johnson, J. H. Porter, S. I. Ackley, and D. T. Ross, "Automatic generation of efficient lexical processors using finite state techniques," *Communications of the ACM*, vol. 11, no. 12, pp. 805–813, 1968.

[22] K. Thompson, "Programming techniques: Regular expression search algorithm," *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[23] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[24] L. Kong, Q. Yu, A. Chattopadhyay, A. Le Glaunec, Y. Huang, K. Mamouras, and K. Yang, "Software-hardware codesign for efficient in-memory regular pattern matching," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, (New York, NY, USA), p. 733–748, ACM, 2022.

[25] A. Le Glaunec, L. Kong, and K. Mamouras, "Regular expression matching using bit vector automata," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 92:1–92:30, 2023.

[26] Z. Wen, L. Kong, A. Le Glaunec, K. Mamouras, and K. Yang, "BVAP: Energy and memory efficient automata processing for regular expressions," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, (New York, NY, USA), pp. 151–166, ACM, 2024.

[27] A. Chattopadhyay and K. Mamouras, "A verified online monitor for metric temporal logic with quantitative semantics," in *Runtime Verification* (J. Deshmukh and D. Ničković, eds.), (Cham), pp. 383–403, Springer, 2020.

[28] K. Mamouras and A. Chattopadhyay, "Efficient matching of regular expressions with lookaround assertions," *Proceedings of the ACM on Programming Languages*, vol. 8, no. POPL, pp. 92:1–92:31, 2024.

[29] G. E. Fainekos and G. J. Pappas, "Robustness of temporal logic specifications for continuous-time signals," *Theoretical Computer Science*, vol. 410, no. 42, pp. 4262–4291, 2009.

[30] D. Ulus, "The Reelay monitoring tool." `https://doganulus.github.io/reelay/`, 2021. [Online; accessed August 20, 2021].

[31] D. Kozen, "A completeness theorem for Kleene algebras and the algebra of regular events," *Information and Computation*, vol. 110, no. 2, pp. 366–390, 1994.

[32] A. Asperti, "A compact proof of decidability for regular expression equivalence," in *Interactive Theorem Proving* (L. Beringer and A. Felty, eds.), (Berlin, Heidelberg), pp. 283–298, Springer, 2012.

[33] S. Fischer, F. Huch, and T. Wilke, "A play on regular expressions: Functional pearl," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, (New York, NY, USA), p. 357–368, ACM, 2010.

[34] The PCRE2 Developers, "Perl-compatible regular expressions (revised API: PCRE2)." `https://pcre2project.github.io/pcre2/doc/html/index.html`, 2022. [Online; accessed Feb 26, 2023].

[35] Oracle, "Java regex matching." `https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html`. [Online; accessed Feb 28, 2024].

[36] K. Mamouras, A. Chattopadhyay, and Z. Wang, "Algebraic quantitative semantics for efficient online temporal monitoring," in *TACAS 2021* (J. F. Groote and K. G. Larsen, eds.), vol. 12651 of *LNCS*, (Cham), pp. 330–348, Springer, 2021.

[37] K. Mamouras, A. Le Glaunec, W. A. Li, and A. Chattopadhyay, "Static analysis for checking the disambiguation robustness of regular expressions," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 231:1–231:25, 2024.

[38] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *FTRTFT/FORMATS 2004* (Y. Lakhnech and S. Yovine, eds.), vol. 3253 of *LNCS*, (Berlin, Heidelberg), pp. 152–166, Springer, 2004.

[39] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout, "Reasoning with temporal logic on truncated paths," in *CAV 2003* (W. A. Hunt and F. Somenzi, eds.), (Berlin, Heidelberg), pp. 27–39, Springer, 2003.

[40] The Coq development team, "The Coq proof assistant." `https://coq.inria.fr`, 2021. [Online; accessed August 20, 2021].

[41] G. Fainekos, B. Hoxha, and S. Sankaranarayanan, "Robustness of specifications and its applications to falsification, parameter mining, and runtime monitoring with S-TaLiRo," in *Runtime Verification* (B. Finkbeiner and L. Mariani, eds.), (Cham), pp. 27–47, Springer, 2019.

[42] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *FORMATS 2010* (K. Chatterjee and T. A. Henzinger, eds.),

vol. 6246 of *LNCS*, (Heidelberg), pp. 92–106, Springer, 2010.

[43] D. Lemire, “Streaming maximum-minimum filter using no more than three comparisons per element,” *Nordic Journal of Computing*, vol. 13, no. 4, pp. 328–339, 2006.

[44] A. Chlipala, *Certified Programming with Dependent Types : a pragmatic introduction to the Coq proof assistant*, ch. 5. Cambridge, MA: The MIT Press, 2013.

[45] K. Mamouras and Z. Wang, “Online signal monitoring with bounded lag,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3868–3880, 2020.

[46] A. Pnueli and A. Zaks, “On the merits of temporal testers,” in *25 Years of Model Checking: History, Achievements, Perspectives* (O. Grumberg and H. Veith, eds.), vol. 5000 of *LNCS*, pp. 172–195, Heidelberg: Springer, 2008.

[47] O. Maler, D. Nickovic, and A. Pnueli, “Real time temporal logic: Past, present, future,” in *FORMATS 2005* (P. Pettersson and W. Yi, eds.), vol. 3829 of *LNCS*, (Heidelberg), pp. 2–16, Springer, 2005.

[48] T. Ferrère, O. Maler, D. Ničković, and A. Pnueli, “From real-time logic to timed automata,” *Journal of the ACM*, vol. 66, no. 3, pp. 19:1–19:31, 2019.

[49] K. Mamouras, M. Raghothaman, R. Alur, Z. G. Ives, and S. Khanna, “StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’17, (New York, NY, USA), pp. 693–708, ACM, 2017.

[50] L. Kong and K. Mamouras, “StreamQL: A query language for processing streaming time series,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 183:1–183:32, 2020.

[51] R. Alur, D. Fisman, and M. Raghothaman, “Regular programming for quantitative properties of data streams,” in *ESOP 2016* (P. Thiemann, ed.), vol. 9632 of *LNCS*, (Berlin, Heidelberg), pp. 15–40, Springer, 2016.

[52] R. Alur and K. Mamouras, “An introduction to the StreamQRE language,” *Dependable Software Systems Engineering*, vol. 50, pp. 1–24, 2017.

[53] R. Alur, K. Mamouras, and D. Ulus, “Derivatives of quantitative regular expressions,” in *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday* (L. Aceto, G. Bacci, G. Bacci, A. Ingólfsdóttir, A. Legay, and R. Mardare, eds.), vol. 10460 of *LNCS*, pp. 75–95, Cham: Springer, 2017.

[54] H. Abbas, R. Alur, K. Mamouras, R. Mangharam, and A. Rodionova, “Real-time decision policies with predictable performance,” *Proceedings of the IEEE, Special Issue on Design Automation for Cyber-Physical Systems*, vol. 106, no. 9, pp. 1593–1615, 2018.

[55] H. Abbas, A. Rodionova, K. Mamouras, E. Bartocci, S. A. Smolka, and R. Grosu, “Quantitative regular expressions for arrhythmia detection,” *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 16, no. 5, pp. 1586–1597, 2019.

[56] R. Alur, K. Mamouras, and C. Stanford, “Automata-based stream processing,” in *Proceedings of the 44th International Colloquium on Automata, Languages,*

and Programming (ICALP 2017) (I. Chatzigiannakis, P. Indyk, F. Kuhn, and A. Muscholl, eds.), vol. 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 112:1–112:15, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[57] R. Alur, K. Mamouras, and C. Stanford, "Modular quantitative monitoring," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 50:1–50:31, 2019.

[58] R. Alur, D. Fisman, K. Mamouras, M. Raghothaman, and C. Stanford, "Streamable regular transductions," *Theoretical Computer Science*, vol. 807, pp. 15–41, 2020.

[59] A. Donzé, T. Ferrère, and O. Maler, "Efficient robust monitoring for STL," in *CAV 2013* (N. Sharygina and H. Veith, eds.), vol. 8044 of *LNCS*, (Heidelberg), pp. 264–279, Springer, 2013.

[60] A. Dokhanchi, B. Hoxha, and G. Fainekos, "On-line monitoring for temporal logic robustness," in *RV 2014* (B. Bonakdarpour and S. A. Smolka, eds.), vol. 8734 of *LNCS*, (Cham), pp. 231–246, Springer, 2014.

[61] D. Basin, F. Klaedtke, and E. Zalinescu, "Greedily computing associative aggregations on sliding windows," *Information Processing Letters*, vol. 115, no. 2, pp. 186–192, 2015.

[62] The Valgrind Developers, "Valgrind: An instrumentation framework for building dynamic analysis tools." `https://valgrind.org/`, 2021. [Online; accessed August 20, 2021].

[63] D. Ulus, “Timescales: A benchmark generator for MTL monitoring tools,” in *RV 2019* (B. Finkbeiner and L. Mariani, eds.), vol. 11757 of *LNCS*, (Cham), pp. 402–412, Springer, 2019.

[64] N. Markey and P. Schnoebelen, “Model checking a path,” in *CONCUR 2003 - Concurrency Theory* (R. Amadio and D. Lugiez, eds.), (Berlin, Heidelberg), pp. 251–265, Springer, 2003.

[65] K. Sen, G. Roşu, and G. Agha, “Generating optimal linear temporal logic monitors by coinduction,” in *Advances in Computing Science – ASIAN 2003. Progamming Languages and Distributed Computation Programming Languages and Distributed Computation* (V. A. Saraswat, ed.), (Berlin, Heidelberg), pp. 260–275, Springer, 2003.

[66] O. Kupferman and M. Y. Vardi, “Model checking of safety properties,” *Formal Methods in System Design*, vol. 19, no. 3, pp. 291–314, 2001.

[67] O. Kupferman and M. Y. Vardi, “Freedom, weakness, and determinism: from linear-time to branching-time,” in *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.98CB36226)*, pp. 81–92, 1998.

[68] P. Thati and G. Rosu, “Monitoring algorithms for metric temporal logic specifications,” *Electronic Notes in Theoretical Computer Science*, vol. 113, pp. 145–162, 2005. Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).

[69] S. Jakšić, E. Bartocci, R. Grosu, and D. Ničković, “An algebraic framework for runtime verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2233–2243, 2018.

[70] S. Jakšić, E. Bartocci, R. Grosu, T. Nguyen, and D. Ničković, "Quantitative monitoring of STL with edit distance," *Formal Methods in System Design*, vol. 53, no. 1, pp. 83–112, 2018.

[71] T. Akazaki and I. Hasuo, "Time robustness in MTL and expressivity in hybrid system falsification," in *CAV 2015* (D. Kroening and C. S. Păsăreanu, eds.), vol. 9207 of *LNCS*, (Cham), pp. 356–374, Springer, 2015.

[72] J. V. Deshmukh, R. Majumdar, and V. S. Prabhu, "Quantifying conformance using the Skorokhod metric," *Formal Methods in System Design*, vol. 50, no. 2-3, pp. 168–206, 2017.

[73] H. Abbas and R. Mangharam, "Generalized robust MTL semantics for problems in cardiac electrophysiology," in *2018 Annual American Control Conference (ACC)*, pp. 1592–1597, IEEE, 2018.

[74] A. Rodionova, E. Bartocci, D. Nickovic, and R. Grosu, "Temporal logic as filtering," in *International Conference on Hybrid Systems: Computation and Control (HSCC 2016)*, pp. 11–20, ACM, 2016.

[75] D. Ulus, T. Ferrère, E. Asarin, and O. Maler, "Timed pattern matching," in *FORMATS 2014* (A. Legay and M. Bozga, eds.), vol. 8711 of *LNCS*, (Cham), pp. 222–236, Springer, 2014.

[76] E. Asarin, P. Caspi, and O. Maler, "Timed regular expressions," *Journal of the ACM*, vol. 49, no. 2, pp. 172–206, 2002.

[77] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[78] A. Bakhirkin, T. Ferrère, O. Maler, and D. Ulus, "On the quantitative semantics of regular expressions over real-valued signals," in *FORMATS 2017* (A. Abate and G. Geeraerts, eds.), vol. 10419 of *LNCS*, (Cham), pp. 189–206, Springer, 2017.

[79] A. Bakhirkin and N. Basset, "Specification and efficient monitoring beyond STL," in *TACAS 2019* (T. Vojnar and L. Zhang, eds.), vol. 11428 of *LNCS*, (Cham), pp. 79–97, Springer, 2019.

[80] O. Maler and D. Ničković, "Monitoring properties of analog and mixed-signal circuits," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 3, pp. 247–268, 2013.

[81] D. Ničković, O. Lebeltel, O. Maler, T. Ferrère, and D. Ulus, "AMT 2.0: Qualitative and quantitative trace analysis with Extended Signal Temporal Logic," in *TACAS 2018* (D. Beyer and M. Huisman, eds.), (Cham), pp. 303–319, Springer, 2018.

[82] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *CAV 2010* (T. Touili, B. Cook, and P. Jackson, eds.), vol. 6174 of *LNCS*, (Heidelberg), pp. 167–170, Springer, 2010.

[83] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh, "Efficient guiding strategies for testing of temporal properties of hybrid systems," in *NFM 2015* (K. Havelund, G. Holzmann, and R. Joshi, eds.), vol. 9058 of *LNCS*, (Cham), pp. 127–142, Springer, 2015.

[84] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "LOLA: Runtime monitoring of

synchronous systems," in *12th International Symposium on Temporal Representation and Reasoning (TIME 2005)*, pp. 166–174, IEEE, 2005.

[85] P. Faymonville, B. Finkbeiner, M. Schwenger, and H. Torfah, "Real-time stream-based monitoring," *CoRR*, vol. abs/1711.03829, 2017.

[86] P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, and H. Torfah, "StreamLAB: Stream-based monitoring of cyber-physical systems," in *CAV 2019* (I. Dillig and S. Tasiran, eds.), vol. 11561 of *LNCS*, (Cham), pp. 421–431, Springer, 2019.

[87] J. O. Blech, Y. Falcone, and K. Becker, "Towards certified runtime verification," in *ICFEM 2012* (T. Aoki and K. Taguchi, eds.), vol. 7635 of *LNCS*, (Heidelberg), pp. 494–509, Springer, 2012.

[88] D. Basin, F. Klaedtke, and E. Zalinescu, "The MonPoly monitoring tool," in *RV-CuBES 2017* (G. Reger and K. Havelund, eds.), vol. 3 of *Kalpa Publications in Computing*, pp. 19–28, EasyChair, 2017.

[89] J. Schneider, D. Basin, S. Krstić, and D. Traytel, "A formally verified monitor for metric first-order temporal logic," in *RV 2019* (B. Finkbeiner and L. Mariani, eds.), vol. 11757 of *LNCS*, (Cham), pp. 310–328, Springer, 2019.

[90] D. Basin, T. Dardinier, L. Heimes, S. Krstić, M. Raszyk, J. Schneider, and D. Traytel, "A formally verified, optimized monitor for metric first-order dynamic logic," in *IJCAR 2020* (N. Peltier and V. Sofronie-Stokkermans, eds.), vol. 12166 of *LNCS*, (Cham), pp. 432–453, Springer, 2020.

[91] B. Finkbeiner, S. Oswald, N. Passing, and M. Schwenger, "Verified Rust

monitors for Lola specifications," in *Runtime Verification* (J. Deshmukh and D. Ničković, eds.), (Cham), pp. 431–450, Springer, 2020.

[92] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation* (B. Jobstmann and K. R. M. Leino, eds.), (Berlin, Heidelberg), pp. 41–62, Springer, 2016.

[93] M. Berglund, F. Drewes, and B. van der Merwe, "Analyzing catastrophic backtracking behavior in practical regular expression matching," in *Automata and Formal Languages 2014 (AFL 2014)* (Z. Ésik and Z. Fülöp, eds.), vol. 151 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pp. 109–123, Open Publishing Association, 2014.

[94] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, (New York, NY, USA), p. 246–256, ACM, 2018.

[95] "RE2: Google's regular expression library." `https://github.com/google/re2`, 2023.

[96] "Intel's Hyperscan: A high-performance multiple regex matching library." `https://github.com/intel/hyperscan`, 2023.

[97] T. Miyazaki and Y. Minamide, "Derivatives of regular expressions with lookahead," *Journal of Information Processing*, vol. 27, pp. 422–430, 2019.

[98] T. Nipkow and D. Traytel, "Unified decision procedures for regular expression equivalence," in *Interactive Theorem Proving* (G. Klein and R. Gamboa, eds.), (Cham), pp. 450–466, Springer, 2014.

[99] R. Cox, "Regular expression matching in the wild." `https://swtch.com/~rsc/regexp/regexp3.html`, 2010.

[100] J. A. Brzozowski, "Derivatives of regular expressions," *Journal of the ACM*, vol. 11, p. 481–494, oct 1964.

[101] S. M. Kearns, "Extending regular expressions with context operators and parse extraction," *Softw. Pract. Exper.*, vol. 21, p. 787–804, jul 1991.

[102] Y. Sakuma, Y. Minamide, and A. Voronkov, "Translating regular expression matching into transducers," *Journal of Applied Logic*, vol. 10, no. 1, pp. 32–51, 2012. Special issue on Automated Specification and Verification of Web Systems.

[103] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, (New York, NY, USA), pp. 111–122, ACM, 2004.

[104] V. Benzaken, G. Castagna, and A. Frisch, "CDuce: An XML-centric general-purpose language," in *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, (New York, NY, USA), pp. 51–63, ACM, 2003.

[105] B. B. Grathwohl, F. Henglein, U. T. Rasmussen, K. A. Søholm, and S. P. Tørholm, "Kleenex: Compiling nondeterministic transducers to deterministic

streaming transducers," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, (New York, NY, USA), pp. 284–297, ACM, 2016.

[106] T. Miyazaki and Y. Minamide, "Context-free grammars with lookahead," in *LATA 2021* (A. Leporati, C. Martín-Vide, D. Shapira, and C. Zandron, eds.), vol. 12638 of *LNCS*, (Cham), pp. 213–225, Springer, 2021.

[107] T. Miyazaki and Y. Minamide, "Derivatives of context-free grammars with lookahead," *Journal of Information Processing*, vol. 31, pp. 421–431, 2023.

[108] A. Asperti, C. S. Coen, and E. Tassi, "Regular expressions, au point," 2010.

[109] A. Barrière and C. Pit-Claudel, "Linear matching of JavaScript regular expressions," *Proceedings of the ACM on Programming Languages*, vol. 8, jun 2024.

[110] H. Fujinami and I. Hasuo, "Efficient matching with memoization for regexes with look-around and atomic grouping," in *Programming Languages and Systems* (S. Weirich, ed.), (Cham), pp. 90–118, Springer Nature Switzerland, 2024.

[111] R. Pike, "The text editor sam," *Software: Practice and Experience*, vol. 17, no. 11, pp. 813–845, 1987.

[112] A. Barrière and C. Pit-Claudel, "Linear matching of javascript regular expressions," 2023.

[113] A. Frisch and L. Cardelli, "Greedy regular expression matching," in *ICALP 2004* (J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, eds.), vol. 3142 of *LNCS*, (Berlin, Heidelberg), pp. 618–629, Springer, 2004.

[114] L. Nielsen and F. Henglein, "Bit-coded regular expression parsing," in *LATA 2011* (A.-H. Dediu, S. Inenaga, and C. Martín-Vide, eds.), vol. 6638 of *LNCS*, (Berlin, Heidelberg), pp. 402–413, Springer, 2011.

[115] "IEEE standard for information technology - portable operating system interface (POSIX(R))," *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pp. 1–3874, 2008.

[116] M. Berglund, B. van der Merwe, and S. van Litsenborgh, "Regular expressions with lookahead," *JUCS - Journal of Universal Computer Science*, vol. 27, no. 4, pp. 324–340, 2021.

[117] A. Morihata, "Translation of regular expression with lookahead into finite state automaton," *Computer Software*, vol. 29, no. 1, pp. 147–158, 2012.

[118] L. J. Stockmeyer, *The complexity of decision problems in automata theory and logic.* PhD thesis, Massachusetts Institute of Technology, 1974.

[119] G. Roşu and M. Viswanathan, "Testing extended regular language membership incrementally by rewriting," in *Rewriting Techniques and Applications* (R. Nieuwenhuis, ed.), (Berlin, Heidelberg), pp. 499–514, Springer, 2003.

[120] G. Roşu, "An effective algorithm for the membership problem for extended regular expressions," in *Foundations of Software Science and Computational Structures* (H. Seidl, ed.), (Berlin, Heidelberg), pp. 332–345, Springer, 2007.

[121] T. Coquand and V. Siles, "A decision procedure for regular expression equivalence in type theory," in *CPP 2011* (J.-P. Jouannaud and Z. Shao, eds.), vol. 7086 of *LNCS*, (Berlin, Heidelberg), pp. 119–134, Springer, 2011.

[122] D. Egolf, S. Lasser, and K. Fisher, “Verbatim++: Verified, optimized, and semantically rich lexing with derivatives,” in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, (New York, NY, USA), pp. 27–39, ACM, 2022.

[123] V. Antimirov, “Partial derivatives of regular expressions and finite automaton constructions,” *Theoretical Computer Science*, vol. 155, no. 2, pp. 291–319, 1996.

[124] V. Komendantsky, “Reflexive toolbox for regular expression matching: Verification of functional programs in Coq+Ssreflect,” in *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, PLPV ’12, (New York, NY, USA), pp. 61–70, ACM, 2012.

[125] N. Moreira, D. Pereira, and S. Melo de Sousa, “Deciding regular expressions (in-)equivalence in Coq,” in *RAMiCS 2012* (W. Kahl and T. G. Griffin, eds.), vol. 7560 of *LNCS*, (Berlin, Heidelberg), pp. 98–113, Springer, 2012.

[126] C. Stanford, M. Veanes, and N. Bjørner, “Symbolic boolean derivatives for efficiently solving extended regular expression constraints,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, (New York, NY, USA), p. 620–635, ACM, 2021.

[127] I. E. Varatalu, M. Veanes, and J.-P. Ernits, “Derivative based extended regular expression matching supporting intersection, complement and lookarounds,” 2023.

[128] D. Moseley, M. Nishio, J. Perez Rodriguez, O. Saarikivi, S. Toub, M. Veanes,

T. Wan, and E. Xu, “Derivative based nonbacktracking real-world regex matching with backtracking semantics,” *Proceedings of the ACM on Programming Languages*, vol. 7, jun 2023.

[129] C. Urban, “POSIX lexing with derivatives of regular expressions,” *Journal of Automated Reasoning*, vol. 67, jul 2023.

[130] C. Tan and C. Urban, “POSIX lexing with bitcoded derivatives,” in *14th International Conference on Interactive Theorem Proving (ITP 2023)* (A. Naumowicz and R. Thiemann, eds.), vol. 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 27:1–27:18, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[131] F. Ausaf, R. Dyckhoff, and C. Urban, “Posix lexing with derivatives of regular expressions (proof pearl),” in *Interactive Theorem Proving* (J. C. Blanchette and S. Merz, eds.), (Cham), pp. 69–86, Springer, 2016.

[132] C. Doczkal, J.-O. Kaiser, and G. Smolka, “A constructive theory of regular languages in Coq,” in *Certified Programs and Proofs* (G. Gonthier and M. Norrish, eds.), (Cham), pp. 82–97, Springer, 2013.

[133] C. Doczkal and G. Smolka, “Two-way automata in Coq,” in *Interactive Theorem Proving* (J. C. Blanchette and S. Merz, eds.), (Cham), pp. 151–166, Springer, 2016.

[134] D. Firsov and T. Uustalu, “Certified parsing of regular languages,” in *Certified Programs and Proofs* (G. Gonthier and M. Norrish, eds.), (Cham), pp. 98–113, Springer, 2013.

[135] O. Kammar and K. Marek, "Idris tyre: a dependently typed regex parser," 2023.

[136] T. Braibant and D. Pous, "Deciding Kleene Algebras in Coq," in *ITP*, vol. 6172 of *LNCS*, (Edinburgh, United Kingdom), pp. 163–178, Springer, Aug. 2010.

[137] J. H. Conway, *Regular Algebra and Finite Machines.* London: Chapman and Hall, 1971.

[138] A. Salomaa, "Two complete axiom systems for the algebra of regular events," *Journal of the ACM*, vol. 13, p. 158–169, jan 1966.

[139] V. N. Redko, "On defining relations for the algebra of regular events," *Ukrainskii Matematicheskii Zhurnal*, vol. 16, pp. 120–126, 1964.

[140] D. Kozen and K. Mamouras, "Kleene algebra with equations," in *ICALP 2014* (J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, eds.), vol. 8573 of *LNCS*, (Berlin, Heidelberg), pp. 280–292, Springer, 2014.

[141] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, p. 761–763, aug 1984.

[142] J. Xu, K. Lu, Z. Du, Z. Ding, L. Li, Q. Wu, M. Payer, and B. Mao, "Silent bugs matter: A study of compiler-introduced security bugs," in *32nd USENIX Security Symposium (USENIX Security 23)*, (Anaheim, CA), pp. 3655–3672, USENIX Association, Aug. 2023.

[143] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "CompCert - A Formally Verified Optimizing Compiler," in *ERTS 2016: Em-*

bedded Real Time Software and Systems, 8th European Congress, (Toulouse, France), SEE, Jan. 2016.

[144] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, (New York, NY, USA), p. 42–54, ACM, 2006.

[145] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: a verified implementation of ML," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), p. 179–191, ACM, 2014.

[146] Vern Paxson, "Flex: The fast lexical analyzer." `https://github.com/westes/flex`. [Online; accessed May 22, 2024].

[147] D. Egolf, S. Lasser, and K. Fisher, "Verbatim: A verified lexer generator," in *2021 IEEE Security and Privacy Workshops (SPW)*, pp. 92–100, 2021.

[148] W. Ouedraogo, G. Scherer, and L. Straßburger, "Coqlex: Generating formally verified lexers," *Art Sci. Eng. Program.*, vol. 8, no. 1, 2023.

[149] K. Thompson, "Programming techniques: Regular expression search algorithm," *Communications of the ACM*, vol. 11, p. 419–422, jun 1968.

[150] M. Sozeau, "Generalized rewriting." `https://coq.inria.fr/doc/V8.19.2/refman/addendum/generalized-rewriting.html`, 2023. [Online; accessed Aug 7, 2024].

[151] A. W. Appel, "Efficient verified red-black trees," 2011.

[152] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format.” RFC 8259, Dec. 2017.

[153] J. B. Smith, *Ocamllex and Ocamlyacc*, pp. 193–211. Berkeley, CA: Apress, 2007.

[154] W. Bank, “United states annual GDP data [data file],” 2020.

[155] R. G. G. Cattell, *Formalization and Automatic Derivation of Code Generators.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1978.

[156] W. Yang, C.-W. Tsay, and J.-T. Chan, “On the applicability of the longest-match rule in lexical analysis,” *Computer Languages, Systems & Structures*, vol. 28, pp. 273–288, oct 2002.

[157] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques & Tools.* Pearson Education, 2007.

[158] T. Reps, ““Maximal-munch” tokenization in linear time,” *ACM Transactions on Programming Languages and Systems*, vol. 20, p. 259–273, mar 1998.

[159] J. R. Levine, T. Mason, and D. Brown, *Lex & yacc.* ” O’Reilly Media, Inc.”, 1992.

[160] U. Trofimovich, “RE2C: A lexer generator based on lookahead-TDFA,” *Software Impacts*, vol. 6, p. 100027, 2020.

[161] V. Laurikari, “NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions,” in *Proceedings Seventh International Symposium on String Processing and Information Retrieval (SPIRE 2000)*, (USA), pp. 181–187, IEEE, 2000.

[162] L. Li, S. Sato, Q. Liu, and K. Taura, “Plex: Scaling parallel lexing with backtrack-free prescanning,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 693–702, 2021.

[163] M. M. T. Chakravarty, “Lazy lexing is fast,” in *Functional and Logic Programming* (A. Middeldorp and T. Sato, eds.), (Berlin, Heidelberg), pp. 68–84, Springer, 1999.

[164] T. Nipkow, “Verified lexical analysis,” in *Theorem Proving in Higher Order Logics* (J. Grundy and M. Newey, eds.), (Berlin, Heidelberg), pp. 1–15, Springer, 1998.

[165] J. B. Almeida, N. Moreira, D. Pereira, and S. M. de Sousa, “Partial derivative automata formalized in Coq,” in *Implementation and Application of Automata* (M. Domaratzki and K. Salomaa, eds.), (Berlin, Heidelberg), pp. 59–68, Springer, 2011.

[166] F. Pottier, “Depth-First Search and Strong Connectivity in Coq,” in *Vingt-sixièmes journées francophones des langages applicatifs (JFLA 2015)* (D. Baelde and J. Alglave, eds.), (Le Val d’Ajol, France), Jan. 2015.

[167] P. Lammich and R. Neumann, “A framework for verifying depth-first search algorithms,” in *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP ’15, (New York, NY, USA), p. 137–146, ACM, 2015.

[168] R. Chen and J.-J. Levy, “Readable semi-automatic formal proofs of Depth-First Search in graphs using Why3.” Nov. 2015.

[169] J.-C. Filliâtre and A. Paskevich, “Why3 — where programs meet provers,”

in *Programming Languages and Systems* (M. Felleisen and P. Gardner, eds.), (Berlin, Heidelberg), pp. 125–128, Springer, 2013.

[170] K. Sakaguchi, "Program extraction for mutable arrays," *Science of Computer Programming*, vol. 191, jun 2020.

[171] S. Ullrich and L. de Moura, "Counting immutable beans: reference counting optimized for purely functional programming," in *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, IFL '19, (New York, NY, USA), ACM, 2021.

[172] A. Reinking, N. Xie, L. de Moura, and D. Leijen, "Perceus: garbage free reference counting with reuse," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, (New York, NY, USA), p. 96–111, ACM, 2021.

[173] A. Lorenzen, D. Leijen, and W. Swierstra, "FP2: fully in-place functional programming," *Proceedings of the ACM on Programming Languages*, vol. 7, aug 2023.

[174] P. Hudak and A. Bloss, "The aggregate update problem in functional programming systems," in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, (New York, NY, USA), p. 300–314, ACM, 1985.

[175] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel, "VST-Floyd: A separation logic tool to verify correctness of c programs," *Journal of Automated Reasoning*, vol. 61, p. 367–422, jun 2018.

[176] K. Mamouras, "Semantic foundations for deterministic dataflow and stream processing," in *ESOP 2020* (P. Müller, ed.), vol. 12075 of *LNCS*, (Heidelberg), pp. 394–427, Springer, 2020.

[177] D. Kozen, "Kleene algebra with tests," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 3, pp. 427–443, 1997.

[178] 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)*, CSL-LICS '14, (New York, NY, USA), pp. 44:1–44:10, ACM, 2014.

[179] K. Mamouras, *Extensions of Kleene Algebra for Program Verification.* PhD thesis, Cornell University, Ithaca, NY, August 2015.

[180] K. Mamouras, "Equational theories of abnormal termination based on Kleene algebra," in *FoSSaCS 2017* (J. Esparza and A. S. Murawski, eds.), vol. 10203 of *LNCS*, (Berlin, Heidelberg), pp. 88–105, Springer, 2017.

[181] A. Brüggemann-Klein and D. Wood, "Deterministic regular languages," in *STACS 92*, (Heidelberg), pp. 173–184, Springer, 1992.

[182] A. Brüggemann-Klein and D. Wood, "One-unambiguous regular languages," *Infromation and Computation*, vol. 140, no. 2, pp. 229–253, 1998.

[183] Y.-S. Han and D. Wood, "Generalizations of 1-deterministic regular languages," *Information and Computation*, vol. 206, no. 9, pp. 1117–1125, 2008. Special Issue: 1st International Conference on Language and Automata Theory and Applications (LATA 2007).