A Logical Treatment of Finite Automata

Nishant Rodrigues[1], Mircea Octavian Sebe[1],
Xiaohong Chen[1], and Grigore Roşu[1]

University of Illinois at Urbana-Champaign
{nishant2,osebe2,xc3,grosu}@illinois.edu

Abstract. We present a sound and complete axiomatization of finite words using matching logic. A unique feature of our axiomatization is that it gives a *shallow embedding* of regular expressions into matching logic, and a *logical* representation of finite automata. The semantics of both expressions and automata are precisely captured as matching logic formulae that evaluate to the corresponding language. Regular expressions are matching logic formulae *as is*, while the embedding of automata is a *structural analog*—computational aspects of automata are captured as syntactic features. We demonstrate that our axiomatization is sound and complete by showing that runs of Brzozowski's procedure for equivalence checking correspond to matching logic proofs. We propose this as a general methodology for producing machine-checkable formal proofs, enabled by capturing structural analogs of computational artifacts in logic. The proofs produced can be efficiently checked by the Metamath Zero verifier. Work presented in this paper contributes to the general scheme of achieving verifiable computing via logical methods, where computations are reduced to logical reasoning, encoded as machine-checkable proof objects, and checked by a trusted proof checker.

1 Motivation

Regular expressions are a powerful lens for studying the description, classification, and implementation of regular languages [14]. A typical presentation of the syntax of extended regular expressions (ERE) over a finite alphabet A is as follows:

$$\alpha := \emptyset \mid \epsilon \mid a \in A \mid \alpha_1 \cdot \alpha_2 \mid \alpha_1 + \alpha_2 \mid \alpha^* \mid \neg\alpha$$

where ϵ is the empty word, $\alpha_1 \cdot \alpha_2$ is concatenation, $\alpha_1 + \alpha_2$ is alternation (aka choice; sum; union), and α^* is the Kleene star. Given a regular expression α, $\mathcal{L}(\alpha)$ is the set of finite words that match α.

A second lens, of finite automata, allows us to view these languages from a *computational* perspective. [14] and [27] show that a language is regular if and only if it is accepted by a finite automaton. Besides providing deeper insight into the study of languages, this dual viewpoint has practical importance—some tasks are easier to tackle when viewed under one lens than another. For example, in the implementation of a parser, it is easier to express the desired language as an expression, whereas an automaton may be used to recognize that language. Model checking [13], and runtime monitoring [3] also exploit these dual perspectives.

Much research has been carried out in the *logical* aspects of regular *expressions*, and the *computational* aspects of finite *automata*. For example, [23] gives an axiomatization of regular expressions in terms of eleven axioms and two inference rules, while automata are used extensively to study complexity theory [13].

In this paper, we instead study *logical* aspects of automata. We present a new axiomatization of finite words using matching logic [8]. This axiomatization gives us a shallow embedding of regular expressions into matching logic where expressions are matching logic formulae *as is*. Uniquely, we can also represent automata as logical formulae. These formulae are a *structural analog* of the automaton—computational aspects such as non-determinism and cycles are captured using syntactic constructs such as logical disjunction and fixpoint operators. We will compare our shallow embedding with prior work using second-order logic, and other formalizations and axiomatizations in Section 2.

Based on our axiomatization, we propose a general technique for generating machine checkable proofs of algorithms that manipulate finite automata. We show that this technique is practical by generating proofs of equivalence between regular expressions from a derivative of Brzozowski's method [4], producing concrete proofs in matching logic's proof system realized in Metamath Zero [6]. As touched on in Section 7, an extension to this work may produce proofs for a symbolic execution based compiler [25] allowing us to trust its correctness.

Work presented here contributes to the scheme of verifiable computing [2] via logical methods: computations are reduced to logical reasoning, encoded as machine-checkable proofs, and checked by a small trusted checker, thus reducing our trust-base to the checker while avoiding the expense of full formal verification.

The rest of the paper is organized as follows:

- Section 2 briefly describes prior work in relation to our work.
- Section 3 reviews regular expressions, automata and related concepts.
- Section 4 introduces matching logic and presents a model of finite words.
- Section 5 shows how we may axiomatize this model, and prove equivalent regular expressions and automata.
- Section 6 gives a brief description of our implementation.
- Section 7 lays out some future avenues for research.

Detailed proofs may be found in the companion technical report [21].

2 Related Work

Monadic Second-order-logic (MSO) over Words There is a well-known connection between MSO and regular languages. Büchi, Elgot, and Trakhtenbrot showed that MSO formulae and regular expressions are equally expressive [5, 11, 28]. Moreover, the transformation from expressions to formulae and back is easily computable [26]. Models are sets of labeled positions, representing a word. The set of models that satisfy a formula give us its language—e.g. the MSO formula

$\bot$ defines the empty language—no word satisfies it, while $\exists x.\, P_a x \land \forall y.\, x = y$ defines the language containing the word a. Here, $P_a x$ indicates the letter at position x is a. The concatenation of languages may be defined as:

$$\exists X.\, \forall y, z.\, ((y \in X \land z \notin X) \to y < z) \land [\varphi_\alpha]_{x \in X} \land [\varphi_\beta]_{x \notin X}$$

Here, $[\varphi]_{\psi(x)}$ denotes the relativization of the formula φ to the formula ψ, a transformation that forces it to apply to a particular subdomain of the model. The translation of Kleene star is even more complex. This connection has been used, e.g. in the verification of MSO formulae [29].

One concern about this connection between MSO and regular expressions is that the translation of expressions is quite involved, including complex auxiliary clauses and quantification, as well as the relativization transformation. Our goal here is to define a shallow embedding, rather than a translation—regular expressions are directly embedded as matching logic with minimal *representational distance*.

Salomaa's Axiomatization In [23] Salomaa provides a complete axiomatization of regular expressions that may be used to prove equivalences. This axiomatization is specific to unextended regular expressions and does not support other representations such as negations in EREs, and finite automata.

Deep Embeddings of Automata and Languages There are several existing formalizations of regular expressions and automata using mechanical theorem provers, such as Coq [10] and Isabelle [16]. To the best of our knowledge, all these formalizations use deep embeddings. In [10], the authors formalize regular expressions and Brzozowski derivatives, with the denotations of regular expressions defined using a membership predicate. Besides proving the soundness of Brzozowski's method, the authors also prove that the process of taking derivatives terminates through a notion of finiteness called *inductively finite sets*. This is something that is not likely provable in a shallow embedding like ours.

Fixpoint Reasoning in Matching Logic We consider our work an extension of the work in [9], where the authors begin tackling the problem of fixpoint reasoning in matching logic. Their goal was to use matching logic as unified framework for fixpoint reasoning across domains. Using a small set of derived matching logic inference rules, they proved various results in LTL, reachability, and separation logic with inductive definitions. We employ many of the techniques first described there, but in addition deal with more complex inductive proofs and recursion schemes, besides producing formal proof certificates.

3 Preliminaries

3.1 Languages, Automata, and Expressions

A language is a set of finite sequences over letters of an alphabet. ERE and finite automata are two ways to represent a class of languages called regular languages.

Definition 1. *Let $A = \{a_1, a_2, \dots a_n\}$ be a finite alphabet. Then ERE over the alphabet A are defined using the following grammar:*

$$\alpha := \emptyset \mid \epsilon \mid a \in A \mid \alpha \cdot \alpha \mid \alpha + \alpha \mid \alpha^* \mid \neg\alpha$$

The language *that an ERE represents, denoted $\mathcal{L}(\alpha)$ is defined inductively:*

$$\begin{aligned}
\mathcal{L}(\emptyset) &= \emptyset \qquad \mathcal{L}(\epsilon) = \{\epsilon\} \qquad \mathcal{L}(a) = \{a\} \\
\mathcal{L}(\alpha_1 + \alpha_2) &= \mathcal{L}(\alpha_1) \cup \mathcal{L}(\alpha_2) \\
\mathcal{L}(\alpha_1 \cdot \alpha_2) &= \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(\alpha_1) \textit{ and } w_2 \in \mathcal{L}(\alpha_2)\} \\
\mathcal{L}(\alpha^*) &= \textstyle\bigcup_{n=0}^{\infty} \mathcal{L}(\alpha^n) \quad \textit{where } \alpha^0 = \epsilon, \textit{ and } \alpha^n = \alpha \cdot \alpha^{n-1} \\
\mathcal{L}(\neg\alpha) &= A^* \setminus \mathcal{L}(\alpha)
\end{aligned}$$

Since EREs include both complement and choice, other operators like intersection, subsumption and equivalence are definable as notation. We denote these as $\alpha \wedge \beta \equiv \neg(\alpha + \neg\beta)$, $\alpha \rightarrow \beta \equiv \neg\alpha + \beta$, and $\alpha \leftrightarrow \beta \equiv (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$ respectively.

Definition 2. *A* non-deterministic finite automaton *(NFA) is a tuple $\mathcal{Q} = (Q, A, \delta, q_0, F)$, where*

- *Q is a finite set of* states*,*
- *A is a finite set of input symbols called the* alphabet*,*
- *$\delta : Q \times A \rightarrow \mathcal{P}(Q)$ is a* transition function*,*
- *$q_0 \in Q$ is the* initial state*, and*
- *$F \subseteq Q$ is the set of* accepting states*.*

If range(δ) *has only singleton sets, $\mathcal{Q}$ is a* deterministic finite automaton *(DFA).*

3.2 Brzozowski's Method

In [4], Brzozowski introduced an operation over languages called its derivative, denoted $\delta_a(\alpha)$. This operation "consumes" a prefix from each word in the language:

Definition 3. *Given a language L and a word s, the derivative of L with respect to s is denoted by $\delta_s(L)$ and is defined as $\{t \mid s \cdot t \in L\}$.*

For EREs, it turns out that the derivative can also be defined syntactically, as a recursive function, through the following equalities:

$$\begin{array}{lll}
\delta_a(\epsilon) = \emptyset & & \delta_a(\alpha_1 + \alpha_2) = \delta_a(\alpha_1) + \delta_a(\alpha_1) \\
\delta_a(\emptyset) = \emptyset & & \delta_a(\alpha_1 \cdot \alpha_2) = \delta_a(\alpha_1) \cdot \alpha_2 + \alpha_1|_\epsilon \cdot \delta_a(\alpha_2) \\
\delta_a(a) = \epsilon & & \delta_a(\alpha^*) = \delta_a(\alpha) \cdot \alpha^* \\
\delta_a(b) = \emptyset & \text{if } a \neq b. & \delta_a(\neg\alpha) = \neg\delta_a(\alpha) \\
\delta_\epsilon(\alpha) = \alpha & & \delta_{a \cdot w}(\alpha) = \delta_w(\delta_a(\alpha))
\end{array}$$

Here, $\alpha \mid_\epsilon$ is ϵ if the language of α contains ϵ and $\emptyset$ otherwise. There are two properties of derivatives that are important to us. First, every ERE may be transformed into an equivalent one partitioning its language per the initial letter:

Theorem 1 (Brzozowski Theorem 4.4). *Every ERE α can be expressed as:*

$$\alpha = \alpha|_\epsilon + \sum_{a \in A} a \cdot \delta_a(\alpha)$$

Second, repeatedly taking the derivative converges:

Theorem 2 (Brzozowski Theorem 5.2). *Two EREs are* similar *iff they are identical modulo associativity, commutativity and idempotency of the + operator. Every ERE has only a finite number of dissimilar derivatives.*

These two properties give rise to an algorithm for converting an ERE into a DFA, illustrated in Figure 1. The automaton is constructed starting from the root node, identifying each node with an ERE. The root node is identified with the original ERE. Every node has transitions for each input letter to the node identified by the derivative. A state is accepting if its language contains the empty word, a property easily checked as a syntactic function of the identifying ERE. This process must terminate by Theorem 2, giving us a DFA. We can check if the ERE is valid by simply checking that all states are accepting.

4 Matching Logic and the Standard Model of Words

In this section, we will review the syntax and semantics of matching logic and present a matching logic model $\mathbb{W}$ of finite words. We show how it may be used to embed both EREs and finite automata. Matching logic, originally proposed in [22], was revised in [8] to include a fixpoint operator. We present a variant, called polyadic matching logic, omitting sorts since we do not need them[1].

4.1 An Overview of Matching Logic

Matching logic has three parts—a syntax of formulae, also called patterns; a semantics, defining a satisfaction relation $\vDash$; and a Hilbert-style proof system, defining a provablility relation $\vdash$. We will only go over the first two, and then return to matching logic's proof system in the Section 5.

Syntax Matching logic formulae, or *patterns*, are built from propositional operators, symbol applications, variables, quantifiers, and a fixpoint binder.

Definition 4. *Let* EVar*,* SVar*, Σ be disjoint countable sets. Here,* EVar *contains* element variables*,* SVar *contains* set variables *and the* signature $\Sigma = \{\Sigma_n\}$ *is an arity-indexed set of* symbols*. A Σ-pattern over Σ is defined by the grammar:*

$$\varphi := \sigma(\varphi_1, \ldots, \varphi_n) \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 = \varphi_2 \mid \varphi_1 \subseteq \varphi_2 \mid x \mid \exists x.\, \varphi \mid X \mid \mu X.\, \varphi$$

[1] It has since been observed that sorts may be defined axiomatically, and it is unnecessary to build them into the logic. It is called *polyadic* to distinguish it from applicative matching logic with only nullary symbols but includes an explicit application operator.

Note that we have assumed more operators than necessary—equality and subset may be defined in terms of the remaining operators. Please refer to [8] for details. We assume the usual notation for operators such as $\top$, $\vee$, $\wedge$, $\forall$, ν etc. Here, ν is the greatest fixpoint operator, defined as $\nu X.\, \varphi \equiv \neg \mu X.\, \neg\varphi[\neg X/X]$.

Semantics: An Informal Overview Matching logic formulae have a pattern matching semantics. Each pattern φ *matches* a set of elements $|\varphi|$ in the model, called its interpretation. As an example, consider the naturals $\mathbb{N}$ as a model with symbols zero and succ. Here, the pattern $\top$ matches every natural, whereas $\mathrm{succ}(x)$ matches $x+1$. Conjunctions and disjunctions behave as intersections and unions—the $\varphi \vee \psi$ matches every pattern that either φ or ψ match.

Unlike first-order logic, matching logic makes no distinction between terms and formulae. We may write $\mathrm{succ}(x \vee y)$ to match both $x+1$ and $y+1$. While unintuitive at first, this syntactic flexibility allows us to shallowly embed varied and diverse logics in matching logic with ease. Examples include first-order logic, temporal logics, separation logic, and many more [8, 7]. Formulae are embedded as patterns with little to no *representational distance*, quite often verbatim.

Patterns aren't two valued as in first-order logic. We can restore the classic semantics by using the set M to indicate "true" and $\emptyset$ for "false". The operators $=$ and $\subseteq$ are *predicate patterns*—they are either true or false. For example, $x \subseteq \mathrm{succ}(\top)$ matches every natural if x is non-zero, and no element otherwise. This allows us to build *constrained patterns* of the form $\varphi_{\text{structure}} \wedge \varphi_{\text{constraints}}$. Here, $\varphi_{\text{structure}}$ defines the structure, while $\varphi_{\text{constraints}}$ places logical constraints on matched elements. For example, the pattern $x \wedge (x \subseteq \mathrm{succ}(\top))$ matches x, but only if it is the successor of some element—i.e. non-zero.

Existential quantification works just as in first-order logic when working over predicate patterns. Over more general patterns, it behaves as the union over a set comprehension. For example, the pattern $\exists x.\, x \wedge (x \subseteq \mathrm{succ}(\top))$ matches *every* non-zero natural. Finally, the fixpoint operator allows us to inductively build sets, as in algebraic datatypes or inductive functions. For example, the pattern $\mu X.\, \mathrm{zero} \vee \mathrm{succ}(\mathrm{succ}(X))$ defines the set of even numbers.

Semantics: A Formal Treatment We will now formally define the semantics of matching logic. In the interest of brevity we keep things concise. For a more detailed treatment please refer to [8]. Matching logic patterns are interpreted in a model, consisting of a nonempty set M of elements called the universe, and an interpretation $\sigma_M : M^n \to \mathcal{P}(M)$ for each n-ary symbol $\sigma \in \Sigma$.

Definition 5 (Matching logic semantics). *An M-*valuation *is a function* $\mathrm{EVar} \cup \mathrm{SVar} \to \mathcal{P}(M)$*, such that each $x \in$* EVar *evaluates to a singleton. For a model M and an M-valuation ρ, the interpretation of patterns is defined as:*

$$|x|^\rho_M = \rho(x),\ |X|^\rho_M = \rho(X) \qquad |\sigma(\varphi_1, \dots, \varphi_n)|^\rho_M = \bigcup_{a_i \in |\varphi_i|^\rho_M} \sigma_M(\varphi_1, \dots, \varphi_n)$$

$$|\neg\varphi|^\rho_M = M \setminus |\varphi|^\rho_M$$

$$|\exists x.\, \varphi|^\rho_M = \bigcup_{a \in M} |\varphi|_{M, \rho[a/x]} \qquad |\varphi_1 \vee \varphi_2|^\rho_M = |\varphi_1|^\rho_M \cup |\varphi_2|^\rho_M$$

$$|\mu X.\, \varphi|^\rho_M = \mathrm{lfp}\left\{A \mapsto |\varphi|_{M,\rho[A/X]}\right\}$$

$$|\varphi_1 \subseteq \varphi_2|^\rho_M = \begin{cases} M \text{ \textit{if} } |\varphi_1|^\rho_M \subseteq |\varphi_2|^\rho_M \\ \emptyset \quad \textit{otherwise} \end{cases} \qquad |\varphi_1 = \varphi_2|^\rho_M = \begin{cases} M \text{ \textit{if} } |\varphi_1|^\rho_M = |\varphi_2|^\rho_M \\ \emptyset \quad \textit{otherwise} \end{cases}$$

For the most part, this definition is as expected. For the predicate patterns, the corresponding patterns evaluate to M if they hold, otherwise to the empty set. Besides these, patterns have the obvious evaluation—set and element variables are evaluated according to ρ; logical operators are evaluated as the corresponding set operation; symbols as defined by the model; existentials as the union for x ranging over M; and μ as the fixpoint of the interpretation of the pattern.

4.2 A Model of Finite Words

Let us introduce a model $\mathbb{W}$ as the standard model of finite words. Define signature Σ_{Word} containing constants ϵ and a for each $a \in A$, and a binary symbol concat for concatenation. This model allows us to describe languages, including those of regular expressions and finite automata as patterns.

Definition 6. *Let $\mathbb{W}$ be a model for the signature Σ_{Word} with universe the set of finite sequences over alphabet A, and the following interpretations of symbols:*

- $\epsilon_{\mathbb{W}} := \{()\}$,
- *for each letter a, $a_{\mathbb{W}} := \{(a)\}$, and*
- $\mathrm{concat}_{\mathbb{W}}(s_1, s_2) := \{s_1 \cdot s_2\}$.

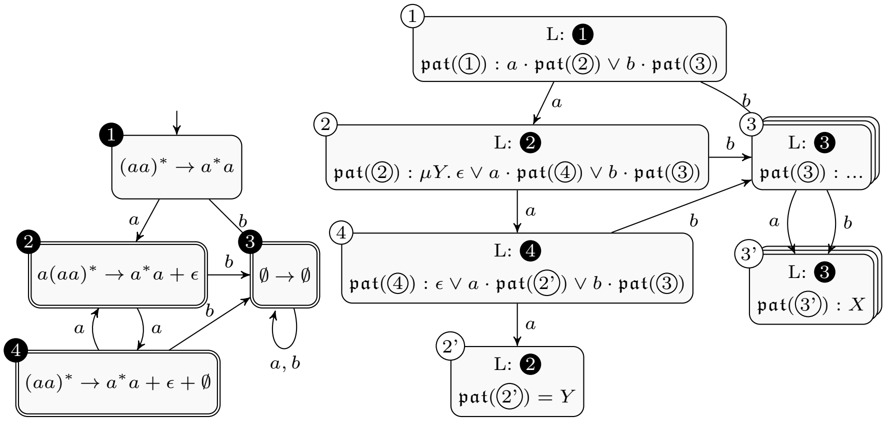
Patterns interpreted in model $\mathbb{W}$ define languages. ϵ is interpreted as the singleton set containing the zero-length word, each letter as the singleton set containing the corresponding single-letter sequence, and finally, concat as the function mapping each pair of input words to the singleton containing their concatenation.

We may define the empty language simply as $\bot$. The concatenation of two patterns gives the concatenation of their languages. Matching logic's disjunction allows us to take the union for any languages, while negation gives us the complement. Finally, we may define the Kleene closure of a language using the fixpoint operator—$\mu X.\, \epsilon \vee \varphi \cdot X$ gives us the Kleene closure of the language of φ.

A Shallow Embedding of Extended Regular Expressions It is easy to define regular expressions as patterns, once we have the following notation:

$$\emptyset \equiv \bot \qquad (\varphi + \psi) \equiv \varphi \vee \psi \qquad \varphi^* \equiv \mu X.\, \epsilon \vee (\varphi \cdot X)$$

Any ERE taken *verbatim* is interpreted in model $\mathbb{W}$ as its language.



❶
$(aa)^* \to a^*a$
a
b
❷
$a(aa)^* \to a^*a + \epsilon$
b
❸
$\emptyset \to \emptyset$
a
a
b
❹
$(aa)^* \to a^*a + \epsilon + \emptyset$
a, b
①
L: ❶
$\mathfrak{pat}(\textcircled{1}) : a \cdot \mathfrak{pat}(\textcircled{2}) \vee b \cdot \mathfrak{pat}(\textcircled{3})$
a
b
②
L: ❷
$\mathfrak{pat}(\textcircled{2}) : \mu Y.\, \epsilon \vee a \cdot \mathfrak{pat}(\textcircled{4}) \vee b \cdot \mathfrak{pat}(\textcircled{3})$
b
③
L: ❸
$\mathfrak{pat}(\textcircled{3}) : \ldots$
a
b
a
b
④
L: ❹
$\mathfrak{pat}(\textcircled{4}) : \epsilon \vee a \cdot \mathfrak{pat}(\textcircled{2'}) \vee b \cdot \mathfrak{pat}(\textcircled{3})$
3'
L: ❸
$\mathfrak{pat}(\textcircled{3'}) : X$
a
2'
L: ❷
$\mathfrak{pat}(\textcircled{2'}) = Y$

Fig. 1: A DFA $\mathcal{Q}$ for the ERE $(aa)^* \to a^*a$, and its corresponding unfolding tree. Each node n shows its label $L(n)$, and the pattern $\mathfrak{pat}(n)$. Here $\mathfrak{pat}(\textcircled{3}) \equiv \mu X.\, \epsilon \vee a \cdot \mathfrak{pat}(\textcircled{3'}) \vee b \cdot \mathfrak{pat}(\textcircled{3'})$. The pattern for the automaton, $\mathfrak{pat}_{\mathcal{Q}}$, is that of the root node $\mathfrak{pat}(\textcircled{1})$. Observe that its structure closely mirrors that of Q. Accepting nodes include ϵ as a disjunct, whereas others do not. Starting a cycle in the graph introduces a fixpoint binder, whereas completing one employs the bound variable corresponding to that cycle. The major structural differences are due to duplicate states to allow backlinks and nodes reachable via muliple paths.

Contrast this to the MSO translation of concatenation, shown in Section 2, and especially of Kleene star.

Theorem 3. *Let α be an ERE. Then $\mathcal{L}(\alpha) = |\alpha|_{\mathbb{W}}$*

4.3 Embedding Automata

While it is obvious how to embed expressions the representation of automata, being computational rather than logical, is less clear. Here, we define a pattern $\mathfrak{pat}_{\mathcal{Q}}$ whose interpretation is the language of a finite automaton $\mathcal{Q}$, either deterministic or non-deterministic. Crucially, this pattern captures not just the language of the automaton (in Section 2 we mentioned that it is possible to do this in MSO as well), but also its *structure*—as shown in Table 1, structural elements of the automata map to syntactic elements of the pattern—non-determinism maps to logical disjunctions; cycles map to fixpoints. This allows us to represent transformations of automata, such as making a transition, union, or complementation, as *logical manipulations* of this pattern in a proof system. This is imperative to capturing the execution of an algorithm employing these in a formal proof. To define $\mathfrak{pat}_{Q}$, we must first define the *unfolding tree* of the automaton $\mathcal{Q}$.

Definition 7. *For a finite automaton $\mathcal{Q} = (Q, A, \delta, q_0, F)$, its* unfolding tree *is a labeled tree (N, E, L) where N is the set of nodes, $E \subseteq A \times N \times N$ is a*

Computational aspect of $\mathcal{Q}$	Syntactic aspect of $\mathfrak{pat}_{\mathcal{Q}}$
Node n is accepting	ϵ is a subclause of $\mathfrak{pat}(n)$
Non-determinism, union of FAs	Logical union
Graph cycles	Fixpoint binder and its bound variable
Changing the initial node	Unfolding, framing

Table 1: Structural aspects of $\mathcal{Q}$ become syntactic aspects of $\mathfrak{pat}_Q$. This is crucial to capturing the traces of algorithms as proofs.

labeled edge relation, and $L : N \to Q$ is a labeling function. It is the tree defined inductively:

- *the root node has label q_0,*
- *if a node n has label q with no ancestors also labeled q, then for each $a \in A$ and $q' \in \delta(q, a)$, there is a node $n' \in N$ with $L(n') = q'$, and $(a, n, n') \in E_a$.*

When $\mathcal{Q}$ is a DFA, we use n_a to denote the unique child of node n along edge a. All leaves in this tree are labeled by states that complete a cycle in the automaton. We define a secondary labeling function, $\mathfrak{pat} : N \to \mathsf{Pattern}$ over this tree.

Definition 8. *Let (N, E, L) be an unfolding tree for $\mathcal{Q} = (Q, A, \delta, q_0, F)$. Let $X : Q \to \mathsf{SVar}$ be an injective function. Then, we define $\mathfrak{pat}$ recursively as follows:*

1. *For a leaf node n, $\mathfrak{pat}(n) := X(L(n))$.*
2. *For a non-leaf node,*
   a. *if n doesn't have a descendant with the same label, then:*

$$\mathfrak{pat}(n) = \begin{cases} \epsilon \ \vee \displaystyle\bigvee_{(a,n,n')\in E} a \cdot \mathfrak{pat}(n') & \textit{if } L(n) \textit{ is accepting.} \\ \displaystyle\bigvee_{(a,n,n')\in E} a \cdot \mathfrak{pat}(n') & \textit{otherwise.} \end{cases}$$

   b. *if n has a descendant with the same label, then:*

$$\mathfrak{pat}(n) = \begin{cases} \mu X(L(n)).\, \epsilon \ \vee \displaystyle\bigvee_{(a,n,n')\in E} a \cdot \mathfrak{pat}(n') & \textit{if } L(n) \textit{ is accepting.} \\ \mu X(L(n)). \displaystyle\bigvee_{(a,n,n')\in E} a \cdot \mathfrak{pat}(n') & \textit{otherwise.} \end{cases}$$

Finally, define $\mathfrak{pat}_{\mathcal{Q}} := \mathfrak{pat}(\mathcal{R})$, where $\mathcal{R}$ is the root of this tree.

For nodes of the form (2b), we "name" them by binding the variable $X(L(n))$ using the fixpoint operator. When we return to that state we use the bound variable to complete a cycle. The use of fixpoints allows us to clearly embody the inductive structure as a pattern. Figure 1 shows an example of unfolding tree. The following theorem shows that this representation of automata is as expected.

Theorem 4. *Let $\mathcal{Q}$ be a finite automaton. Then $\mathcal{L}(\mathcal{Q}) = |\mathfrak{pat}_{\mathcal{Q}}|_{\mathbb{W}}$*

4.4 Embedding Brzozowski's Derivative

Besides regular languages, other important constructs may be defined using this model. Let us look at derivatives, needed to capture Brzozowski's method as a proof. The Brzozowski derivative of a language L w.r.t. a word w, is the set of words obtainable from a word in L by removing the prefix w. Defining this is quite simple in matching logic—for any word w and pattern ψ, we may define its Brzozowski derivative as the pattern $\delta_w(\psi) \equiv \exists x.\, x \wedge (w \cdot x \subseteq \psi)$.

This definition is quite interesting because it closely parallels the embedding of separation logic's magic wand in matching logic: $\varphi \mathrel{-\!\!*} \psi \equiv \exists x.\, x \wedge (\varphi * x \subseteq \psi)$. At first glance, this seems like a somewhat weak connection, but on closer inspection, magic wand and derivatives are semantically quite similar—we may think of magic wand as taking the derivative of one heap with respect to the other.

It is these connections between seemingly disparate areas of program verification that matching logic seeks to bring to the foreground. In fact, both derivates and magic wand generalize to a matching logic operator called *contextual implication*: $C \multimap \psi \equiv \exists \square.\, \square \wedge (C[\square] \subseteq \psi)$ for any pattern ψ and application context C [9]. Using this notation, derivatives and magic wand become $\delta_w(\varphi) \equiv w \cdot \square \multimap \varphi$ and $\varphi \mathrel{-\!\!*} \psi \equiv \varphi * \square \multimap \varphi$ respectively. This operator has proven key to many techniques for fixpoint reasoning in matching logic, especially the derived rules (WRAP) and (UNWRAP) that enable applying Park induction within contexts [9]:

$$\vdash C[\varphi] \to \psi \qquad \xrightleftharpoons[\text{(WRAP)}]{\text{(UNWRAP)}} \qquad \vdash \varphi \to (C \multimap \psi)$$

5 Proof Generation

In the previous section, we showed how we may capture languages as matching logic patterns. Specifically, automata are captured as patterns that are structural analogs. In this section, we will demonstrate how we capture runs of algorithms that manipulate automata as proofs. In particular, we capture runs of Brzozowski's method using matching logic's Hilbert style proof system.

This technique is only possible because of the structural similarity between an automata $\mathcal{Q}$, and its pattern $\mathfrak{pat}_{\mathcal{Q}}$. It gives us the ability to represent computational transformations on automata as *logical* transformations of these patterns using matching logic's proof system. This section focuses on the theory and proofs involved. The subsequent section, Section 6, will present our concrete implementation producing matching logic proofs that can be checked using Metamath Zero. Let us first introduce matching logic's proof system, and a theory Γ_{Word} within which we do our reasoning.

5.1 Matching Logic's Proof System

The third component to matching logic is its proof system, shown in Figure 2. It defines the provability relation, written $\Gamma \vdash \varphi$, meaning that φ can be proved using the proof system using the theory Γ as additional axioms.

(Propos. 1)	$\varphi \to (\psi \to \varphi)$	(Propag$_\bot$)	$C[\bot] \to \bot$
(Propos. 2)	$(\varphi \to (\psi \to \theta))$	(Propag$_\vee$)	$C[\varphi \vee \psi] \to C[\varphi] \vee C[\psi]$
	$\to((\varphi \to \psi) \to (\varphi \to \theta))$	(Propag$_\exists$)	$C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$
(Propos. 3)	$((\varphi \to \bot) \to \bot) \to \varphi$		where $x \notin FV(C)$
(MP)	$\dfrac{\varphi \qquad \varphi \to \psi}{\psi}$	(Framing)	$\dfrac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$
($\exists$-Quant.)	$\varphi[y/x] \to \exists x.\, \varphi$	($\exists$-Gen.)	$\dfrac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi}$ where $x \notin FV(\psi)$
(Pre-fp)	$\varphi[(\mu X.\, \varphi)/X] \to \mu X.\, \varphi$	(kt)	$\dfrac{\varphi[\psi/X] \to \psi}{(\mu X.\, \varphi) \to \psi}$
(Existence)	$\exists x.\, x$	(Subst)	$\dfrac{\varphi}{\varphi[\psi/X]}$
(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$		

Fig. 2: Matching logic proof system. Here C, C_1, C_2 are application contexts, a pattern in which a distinguished element variable $\square$ occurs exactly once, and only under applications. We use the notation $C[\varphi] \equiv C[\varphi/\square]$.

These proof rules fall into four categories. First, the FOL rules provide complete FOL and propositional reasoning. The (propagation) rules allow applications to commute through constructs with a "union" semantics, such as disjunction and existentials. The proof rule (knaster-tarski) is an embodiement of the Knaster-Tarski fixpoint theorem [24], and together with (prefixedpoint) correspond to the Park induction rules of modal logic [18, 15]. Finally, (existence), (singleton), and (subst) are technical rules, needed to work with variables.

5.2 A Theory of Finite Words

We may use a theory Γ, a set of patterns called *axioms*, to restrict the models we consider to those in which every axiom is "true". We say a pattern φ *holds* in a model M, or that φ is *valid* in M, written $M \vDash \varphi$ if its interpretation is M under all evaluations. For a theory Γ, we write $M \vDash \Gamma$ if every axiom in Γ is valid in M. For a pattern ψ, we write $\Gamma \vDash \psi$ if for every model where $M \vDash \Gamma$ we have $M \vDash \psi$. These axioms also extend the provability relation $\vdash$ defined by the proof system, allowing us to proof additional theorems. The soundness of matching logic guarantees that each proved theorem holds in every model of the theory.

Figure 3 defines a theory, Γ_{Word}, of finite words. The first set of the axioms in Γ_{Word}, (func$_\sigma$), gives each symbol a functional interpretation: for an n-ary symbol σ, the axiom $\forall x_1, \ldots, x_n.\, \exists y.\, \sigma(x_1, \ldots, x_n) = y$, forces the interpretation σ_M to return a single output for any input. This is because element variables are always interpreted as singleton sets. Next, the (no-conf) axioms ensure that

Signature: ϵ, $\_\cdot\_$, and a for each $a \in A$.

Axioms:

For each $a \in A$,

$$\exists w.\, a = w \qquad (\textsc{func}_a)$$

$$\exists w.\, \epsilon = w \qquad (\textsc{func}_\epsilon)$$

$$\forall u, v.\, \exists w.\, u \cdot v = w \qquad (\textsc{func}_\bullet)$$

$$\forall u, v, w.\, (u \cdot v) \cdot w = u \cdot (v \cdot w) \qquad (\textsc{assoc})$$

$$\forall x.\, (\epsilon \cdot x) = x \qquad (\textsc{id}_L)$$

$$\forall x.\, (x \cdot \epsilon) = x \qquad (\textsc{id}_R)$$

For each distinct $a, b \in A$,

$$a \neq b \qquad (\textsc{no-conf}_a)$$

$$\epsilon \not\subseteq a \vee b \qquad (\textsc{no-conf}_\epsilon)$$

$$\forall u, v. \epsilon = u \cdot v \rightarrow u = \epsilon \wedge v = \epsilon \qquad (\textsc{no-conf}_\bullet\text{-1})$$

$$\forall x, y : \text{Letter}. \forall u, v.\; x \cdot u = y \cdot v \rightarrow x = y \wedge u = v \qquad (\textsc{no-conf}_\bullet\text{-2})$$

$$\mu X. \epsilon \vee \bigvee_{a \in A} a \cdot X \qquad (\textsc{domain})$$

Fig. 3: $\mathit{\Gamma}_{\text{Word}}$: A theory of finite words in matching logic. This theory is complete for proving equivalence between representations of both automata and extended regular expressions. Here, $\forall x : \text{Letter}.\, \varphi$ is notation for $\forall x.\, x \in \left(\bigvee_{a \in A} a\right) \rightarrow \varphi$.

interpretations of symbols are injective modulo AU—they have distinct interpretations unless their arguments are equal modulo associativity of concatenation with unit ϵ. Here, $\forall x : \text{Letter}.\, \varphi$ is notation for $\forall x.\, x \in (\bigvee_{a \in A} a) \rightarrow \varphi$, i.e. we quantify over letters. The axioms ($\textsc{assoc}$), and ($\textsc{id}_L$), and ($\textsc{id}_R$) enforce the corresponding properties and allow their use in proofs. The final axiom ($\textsc{domain}$) defines our domain to be inductively constructed from ϵ, concatenation and letters. It is easy to see the standard model $\mathbb{W}$ satisfies these axioms, giving us the theorem, proved in the appendix [21]:

Theorem 5. $\mathbb{W} \vDash \mathit{\Gamma}_{\text{Word}}$

The rest of this section is dedicated to showing that $\mathit{\Gamma}_{\text{Word}}$ is complete with respect to both equivalence of automata and EREs—if two automata or expressions have the same language their representations are provably equivalent.

5.3 Proving Equivalence between EREs

We are now ready to demonstrate our proof generation method. We will use it to capture equivalence of expressions using matching logic's Hilbert-style proof system. Brzozowski's method consists of two parts—converting an ERE into a DFA $\mathcal{Q}$, and checking that $\mathcal{Q}$ is total. Mirroring this, the proof for equivalence between EREs $\mathit{\Gamma}_{\text{Word}} \vdash \alpha \leftrightarrow \beta$ has two parts. First, we prove that $\mathit{\Gamma}_{\text{Word}} \vdash \mathfrak{pat}_{\mathcal{Q}} \rightarrow (\alpha \leftrightarrow \beta)$—the language of $\alpha \leftrightarrow \beta$ subsumes that of $\mathcal{Q}$. Second, that $\mathit{\Gamma}_{\text{Word}} \vdash \mathfrak{pat}_Q$—the language of $\mathcal{Q}$ is total. We put these together using ($\textsc{modus-ponens}$), giving us $\mathit{\Gamma}_{\text{Word}} \vdash \alpha \leftrightarrow \beta$—the EREs are provably equivalent.

Proving $\boldsymbol{\mathit{\Gamma}_{\text{Word}} \vdash \mathfrak{pat}_{\mathcal{Q}} \rightarrow (\alpha \leftrightarrow \beta)}$ To prove this, we prove a more general, inductive, lemma:

Lemma 1. *Let n be a node in the unfolding tree of the DFA $\mathcal{Q}$ of the regular expression $\alpha \leftrightarrow \beta$, where α and β have the same language. Then,*

$$\Gamma \vdash \mathfrak{pat}(n)[\Lambda_n] \to \delta_{\mathrm{path}(n)}(\alpha \leftrightarrow \beta)$$

where,

$$\Lambda_n = \begin{cases} \lambda, \textit{ the empty substitution} & \textit{if } n \textit{ is the root node} \\ \Lambda_p[\delta_{\mathrm{path}(p)}(\alpha \leftrightarrow \beta)/X(p)] & \textit{if } n \textit{ has parent } p, \textit{ and } \mathfrak{pat}(p) \textit{ binds } X(p) \\ \Lambda_p & \textit{otherwise.} \end{cases}$$

The substitution Λ_n provides the inductive hypothesis—as we use the (KNASTER-TARSKI) rule on each μ-binder in $\mathfrak{pat}_{\mathcal{Q}}$, it replaces the bound variable with the right-hand side of the goal. The left-hand side then becomes a disjuntion of the form $\epsilon \vee a \cdot \mathfrak{pat}(n_a)[\Lambda_{n_a}] \vee b \cdot \mathfrak{pat}(n_b)[\Lambda_{n_b}]$. We decompose the right-hand side into a similar structure using an important property of derivatives, proved in Γ_{Word}:

Lemma 2. *For any pattern φ, $\Gamma_{\mathrm{Word}} \vdash \varphi = \big((\epsilon \wedge \varphi) \vee \bigvee_{a \in A} a \cdot \delta_a(\varphi)\big)$*

The derivatives are reduced to expressions using proved syntactic simplifications:

Lemma 3. *For EREs α, β and distinct letters a and b, the following hold:*

- $\Gamma_{\mathrm{Word}} \vdash \delta_a(\emptyset) = \emptyset$; $\Gamma_{\mathrm{Word}} \vdash \delta_a(\epsilon) = \emptyset$;
- $\Gamma_{\mathrm{Word}} \vdash \delta_a(b) = \emptyset$; $\Gamma_{\mathrm{Word}} \vdash \delta_a(a) = \epsilon$;
- $\Gamma_{\mathrm{Word}} \vdash \delta_a(\alpha_1 + \alpha_2) = \delta_a(\alpha_1) + \delta_a(\alpha_1)$;
- $\Gamma_{\mathrm{Word}} \vdash \delta_a(\alpha_1 \cdot \alpha_2) = \delta_a(\alpha_1) \cdot \alpha_2 + (\alpha_1 \wedge \epsilon) \cdot \delta_a(\alpha_2)$;
- $\Gamma_{\mathrm{Word}} \vdash \delta_a(\neg\alpha) = \neg\delta_a(\alpha)$;
- $\Gamma_{\mathrm{Word}} \vdash \delta_a(\alpha^*) = \delta_a(\alpha) \cdot \alpha^*$.

Proving $\boldsymbol{\Gamma_{\mathrm{Word}} \vdash \mathfrak{pat}_{\mathcal{Q}}}$ The next part of the proof is a bit more technical, requiring us to exploit the equivalence $\Gamma_{\mathrm{Word}} \vdash (\mu X.\, \epsilon \vee X \cdot \varphi) \leftrightarrow (\mu X.\, \epsilon \vee \varphi \cdot X)$, and induct using the (DOMAIN) axiom. This reduces our goal to $\Gamma_{\mathrm{Word}} \vdash \mathfrak{pat}_{\mathcal{Q}} \cdot (\bigvee_{a \in A} a) \to \mathfrak{pat}_{\mathcal{Q}}$, a consequence of the following inductive lemma:

Lemma 4. *Let n be a node in the unfolding tree of a total DFA $\mathcal{Q}$. Then,*

$$\Gamma_{\mathrm{Word}} \vdash \mathfrak{pat}(n)[\Theta_n] \cdot \big(\textstyle\bigvee_{a \in A} a\big) \to \mathfrak{pat}(n)[\mathrm{U}_n]$$

where,

$$\Theta_n = \begin{cases} \lambda, \textit{ the empty substitution} & \textit{if } n \textit{ is the root node} \\ \Theta_p[\Psi_p/X_{L(p)}] & \textit{if } n \textit{ has parent } p, \textit{ and } \mathfrak{pat}(p) \textit{ binds } X(p) \\ \Theta_p & \textit{otherwise} \end{cases}$$

$$\Psi_p = \Box \cdot \big(\textstyle\bigvee_{a \in A} a\big) \multimap \mathfrak{pat}(p)[\mathrm{U}_p]$$

$$\mathrm{U}_n = \begin{cases} \lambda, \textit{ the empty substitution} & \textit{if } n \textit{ is the root node} \\ \mathrm{U}_p[\mathfrak{pat}(p)[\mathrm{U}_p]/X_{L(p)}] & \textit{if } n \textit{ has parent } p, \textit{ and } \mathfrak{pat}(p) \textit{ binds } X(p) \\ \mathrm{U}_p & \textit{otherwise} \end{cases}$$

```
def checkValid($\varphi$: Regex, prev: set[Regex] = $\emptyset$) $\rightarrow$ bool:
    if $\varphi \in$ prev: return True
    if $\neg$hasEWP($\varphi$): return False
    return checkValid(canonicalize($\delta_a(\varphi)$, prev $\cup$ $\{\varphi\}$))
       and checkValid(canonicalize($\delta_b(\varphi)$, prev $\cup$ $\{\varphi\}$))
```

Fig. 4: The algorithm instrumented to generate proofs. The `canonicalize` function reduces the pattern $\delta_a(\varphi)$ to an ERE, simplifying it to a form where choice is left-associative, and the idemopotency and unit identities have been applied.

Again, Θ_n gives us the inductive hypothesis, this time in the form of a contextual implication. To apply it, we leverage a general property about contextual implications: $\vdash C[C \multimap \varphi] \rightarrow \varphi$, allowing us combine framing with Park induction. This gives us our main theorem, showing that our axiomatization is complete with respect to extended regular expressions:

Theorem 6. *For any EREs α and β with the same language, $\Gamma_{\text{Word}} \vdash \alpha \leftrightarrow \beta$.*

5.4 From Expressions to Automata

Our uniform treatment of automata and expressions as patterns allows us to apply Brzozowski's method not just to EREs but also to more general patterns. For example, it can be used to determinize NFAs, or take the complement, union, or intersection of DFAs. The general principle is the same as above, except instead of $\alpha \leftrightarrow \beta$, we use a pattern corresponding to the operation we wish to perform. For example, to prove that the DFA $\mathcal{Q}$ has the same language as the intersection of those of $\mathcal{A}$ and $\mathcal{B}$, we prove $\Gamma_{\text{Word}} \vdash \mathfrak{pat}_{Q} \leftrightarrow (\mathfrak{pat}_{\mathcal{A}} \wedge \mathfrak{pat}_{\mathcal{B}})$. All we need is the ability to take the derivative of arbitrary fixpoint patterns enabled by the equivalence $\Gamma_{\text{Word}} \vdash \delta_a(\mu X.\, \varphi) \leftrightarrow \delta_a(\varphi[\mu X.\, \varphi/X])$.

6 Implementation and Evaluation

In this section, we describe the implementation of our method. The algorithm implemented is shown in Figure 4. It recursively checks that the expression and its derivatives have the empty-word property, keeping track of when it has already visited an expression. Here, $\delta_a(\varphi)$ represents a *pattern* using the derivative notation, and not the fully simplified regular expression. This notation is simplified away in the (also instrumented) canonicalize function that also normalizes the choice operator to be left-associative and commutes subterms into lexicographic order, allowing the application of the idemopotency and unit axioms. This results in a canonnical representation of expressions modulo similarity.

The intrumentation of successful runs of this method produces a *proof-hint*, an example of which is shown in Figure 5. A proof-hint is an informal artifact

$$
\begin{array}{rll}
\multicolumn{3}{l}{(\texttt{der}\ (a+b)^*,} \\
a: & (\texttt{simpl}\ \delta_a((a+b)^*), & \texttt{der-}{*},\ \Box,\ \alpha \mapsto (a+b); l \mapsto a, \\
 & (\texttt{simpl}\ (\delta_a((a+b)) \cdot (a+b)^* & \texttt{der-}{\vee},\ \Box \cdot (a+b)^*,\ \ldots, \\
 & (\texttt{simpl}\ (\delta_a(a) + \delta_a(b))(a+b)^*, & \texttt{der-same-letter},\ (\Box + \delta_a(b)) \cdot (a+b)^*,\ \ldots, \\
 & (\texttt{simpl}\ (\epsilon + \delta_a(b))(a+b)^*, & \texttt{der-diff-letter}(\ \epsilon + \Box) \cdot (a+b)^*,\ \ldots, \\
 & (\texttt{simpl}\ (\epsilon + \bot)(a+b)^*, & \texttt{choice-identity-right},\ \Box \cdot (a+b)^*,\ \ldots, \\
 & (\texttt{simpl}\ \epsilon \cdot (a+b)^*, & \texttt{concat-identity-left},\ \Box,\ \ldots, \\
 & (\texttt{backlink}\ (a+b)^*)))))))), & \\
b: & (\texttt{simpl}\ \delta_b((a+b)^*), \ldots)) &
\end{array}
$$

Fig. 5: An snippet of a proof-hint for expression $(a+b)^*$ produced by the instrumentation. Most substitutions are omitted for brevity. The lemma id `der-*` corresponds to the metamath theorem for $\Gamma_{\text{Word}} \vdash \delta_l(\alpha^*) = \delta_l(\alpha) \cdot \alpha^*$

containing all the information necessary to produce a formal proof. It is a term defined by the following grammar.

$$
\begin{aligned}
\text{Node} := {} & (\texttt{backlink}\ \text{Pattern}) \\
& \mid (\texttt{der}\ \text{Pattern}, a : \text{Node}, b : \text{Node}) \\
& \mid (\texttt{simpl}\ \text{Pattern}, \text{LemmaID}, \text{Context}, \text{Subst}, \text{Node})
\end{aligned}
$$

These terms are more detailed structures than unfolding trees—if we ignore the simplification nodes, we get an unfolding tree. Each `backlink` and `der` node is labeled by a regular expression, and correspond to the leaf and interior nodes of an unfolding tree. In addition, `der` nodes have child nodes labeled by the patterns $\delta_a(\varphi)$ and $\delta_b(\varphi)$. Note that these are patterns and *not* regular expressions—they use the matching logic notation for derivative, and are distinct from the fully simplified EREs. Each `simpl` node keeps track of equational simplifications needed to reduce the derivative notation, and employs associativity, commutativity, and idempotency of choice to reduce the expression into a canonical form, allowing the construction of unfolding tree to terminate. The `simpl` nodes contain the name of the simplification applied, the context in which it was applied, as well as the substitutions with which it was applied. The LemmaID corresponds to a hand-proved lemma in the Metamath Zero formalization.

To produce the proof of validity, proof-hints are used in three contexts. First, to produce the pattern $\mathfrak{pat}_Q$; next, to produce an instance of Lemma 1; and finally, to produce an instance of Lemma 4. For each lemma, we inductively build up the proof from two manually proven Metamath Zero theorems, one for the `backlink` node case, and another for the `der` node case. In the case of Lemma 1, the `simpl` nodes are ignored. In the case of Lemma 4 we use them to reduce the patterns to their canonical form. This is done by lifting a manually

Benchmark	Nodes	`.mmb` size	Gen. time	Check time
Manual Lemmas		307		3
$(a+b)^*$	3	2	64	3
$a^{**} \to a^*$	5	4	82	3
$(aa)^* \to a^*a + \epsilon$	9	15	179	3
$\neg(\top \cdot a \cdot \top) + \neg(b^*)$	5	5	90	3
$\mathsf{match}_l(2)$ / $\mathsf{match}_l(8)$	19 /43	13 / 266	273 / 27483	3 / 4
$\mathsf{match}_r(2)$ / $\mathsf{match}_r(8)$	19 /43	13 / 228	337 / 21085	3 / 4
$\mathsf{eq}_l(2)$ / $\mathsf{eq}_l(8)$	13 /37	15 / 446	374 / 91661	3 / 5
$\mathsf{eq}_r(2)$ / $\mathsf{eq}_r(8)$	13 /37	15 / 330	368 / 31489	3 / 5

Table 2: Statistics for certificate generation. Sizes are in KiB, times in milliseconds. We show the unfolding tree nodes, proof size, generation and checking time.

proven theorem corresponding to the LemmaId into the context, and applying the substitution, all supplied by the `simpl` node.

Trust Base Our trust base consists of the Metamath Zero formalization of matching logic proof system, including its syntax and meta-operations for its sound application such as substitution, freshness (272 lines); the theory of words instantiated with $A = \{a, b\}$, (13 lines); and the Metamath Zero proof checker, `mm0-c`. Each of these are defined in `.mm0` files in our repository [20, 19]. From these, we prove by hand 354 supporting general theorems and 163 specific to Γ_{Word}, such as Lemmas 1 and 4, and those about derivatives and their simplification.

Evaluation We have evaluated our work against handcrafted tests, as well as standard benchmarks for deciding equivalence presented in [17]. Some statistics are shown in Table 2. Each $\mathsf{match}_{\{l,r\}}(n)$ test, by [12], is an ERE asserting that a^n matches $(a + \epsilon) \cdot a^n$, that is, $a^n \to (a + \epsilon) \cdot a^n$. Here α^n indicates n-fold concatenation of α, with the l version using concatenation from the left, and the r version on the right. That is, α^3 may be either $((\alpha \cdot \alpha) \cdot \alpha)$ or $(\alpha \cdot (\alpha \cdot \alpha))$. Each $\mathsf{eq}_{\{l,r\}}(n)$ test, by [1], checks if a^* and $(a^0 + \cdots + a^n) \cdot (a^n)^*$ are equivalent. We also include property testing using the Hypothesis testing framework. We randomly generate an ERE α, and check that $\alpha \to \alpha$. Our procedure does not optimize for this, so it allows testing correctness for a variety of expressions, augmenting the few handcrafted ones, and the structurally monotonous benchmarks.

Performance In this work, our goal was to prove that this process is feasible—we have not focused on performance. In fact, we find the performance numbers here are quite poor. There are a number of reasons for this.

First, we made some poor implementation choices with reference to instrumentation. The prototype uses Maude and its meta-level to produce the instrumentation. While Maude's search command *collects* all the information needed for the proof hint, it does not make it accessible. This forced us to repeatedly enter and exit the meta-level to collect this information, bringing the running time of, e.g., $\mathsf{match}_l(8)$ to 27 seconds, compared to 3ms when implemented idiomatically.

Another reason is that we targeted simplicity, rather than even the most basic optimizations. For example, when multiple identical nodes occur in an unfolding tree, we do not reuse the subproofs for identical notes in the derivative tree, and instead re-prove the result each time. This causes a significant blow up in proof size. We believe that a relatively small engineering effort would greatly improve performance both in terms of proof size and generation time.

Another issue is that handling machine generated proofs is not one of Metamath Zero's design goals. It is intended as a human-readable language, for human-written proofs. We would rather output a succinct binary representation of proofs. Although Metamath Zero does allow generation of proofs directly in the mmb format, this seems closer to an embedded systems format than a formal language.

7 Future Work and Conclusion

Study of Languages Definable in $\boldsymbol{\Gamma}_{\mathbf{Word}}$ While this paper has focused on regular languages in Γ_{Word}, we can define more languages. For example, the context-free language $\{a^n \cdot b^n | n \in \mathbb{N}\}$ may be defined as $a^n \cdot b^n \equiv \mu X. \epsilon \vee a \cdot X \cdot b$. Extending this, we may define $a^n \cdot b^n \cdot c^i$, and $a^i \cdot b^n \cdot c^n$ for $n, i \in \mathbb{N}$ as the patterns $a^n \cdot b^n \cdot c^*$ and $a^* \cdot b^n \cdot c^n$ respectively. Finally, since patterns are closed under intersection we may define the *context-sensitive* language $a^n \cdot b^n \cdot c^n \equiv (a^n \cdot b^n \cdot c^*) \wedge (a^* \cdot b^n \cdot c^n)$. Extensive research has been done regarding languages definable in fragments of MSO. A corresponding effort for matching logic would be interesting. Likely, quantifiers and fixpoint operators will allow defining most computable languages.

Application to Control Flow Graphs (CFGs) Through the $\mathbb{K}$ Framework, the transition systems of programming languages are defined in matching logic. The CFGs of programs in these languages may be viewed as automata. Our technique would allow formal proofs of correctness of algorithms over the CFGs of programs, such as the semantics-based compiler in [25].

Acknowledgements We warmly thank Mario Carneiro for his invaluable feedback on the usage of Metamath Zero. We are indebted to the anonymous reviewers for their kind input and suggestions.

References

[1] Valentin Antimirov. “Partial derivatives of regular expressions and finite automata constructions”. In: *STACS 95: 12th Annual Symposium on Theoretical Aspects of Computer Science Munich, Germany, March 2–4, 1995 Proceedings*. Springer. 2005, pp. 455–466.

[2] Konstantine Arkoudas and Selmer Bringsjord. “Computers, justification, and mathematical knowledge”. In: *Minds and Machines* 17 (2007), pp. 185–202.

[3] Ezio Bartocci et al. “Introduction to runtime verification”. In: *Lectures on Runtime Verification: Introductory and Advanced Topics* (2018), pp. 1–33.

[4] Janusz A Brzozowski. “Derivatives of regular expressions”. In: *Journal of the ACM (JACM)* 11.4 (1964), pp. 481–494.

[5] J. Richard Buchi. “Weak Second-Order Arithmetic and Finite Automata”. In: *Mathematical Logic Quarterly* 6.1-6 (1960), pp. 66–92. DOI: https://doi.org/10.1002/malq.19600060105. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19600060105. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.19600060105.

[6] Mario Carneiro. “Metamath Zero: Designing a theorem prover prover”. In: *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings 13*. Springer. 2020, pp. 71–88.

[7] Xiaohong Chen and Grigore Roşu. “A general approach to define binders using matching logic”. In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), pp. 1–32.

[8] Xiaohong Chen and Grigore Roşu. *Matching μ-logic*. Tech. rep. http://hdl.handle.net/2142/102281. University of Illinois at Urbana-Champaign, Jan. 2019.

[9] Xiaohong Chen et al. “Towards a unified proof framework for automated fixpoint reasoning using matching logic”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–29.

[10] Thierry Coquand and Vincent Siles. “A decision procedure for regular expression equivalence in type theory”. In: *Certified Programs and Proofs: First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings 1*. Springer. 2011, pp. 119–134.

[11] Calvin C Elgot. “Decision problems of finite automata design and related arithmetics”. In: *Transactions of the American Mathematical Society* 98.1 (1961), pp. 21–51.

[12] Sebastian Fischer, Frank Huch, and Thomas Wilke. “A play on regular expressions: functional pearl”. In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 2010, pp. 357–368.

[13] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. “Introduction to automata theory, languages, and computation”. In: *Acm Sigact News* 32.1 (2001), pp. 60–65.

[14] SC Kleene. “Representation of events in nerve nets and finite automata”. In: *Automata Studies: Annals of Mathematics Studies. Number 34* 34 (1956), p. 3.

[15] Dexter Kozen. “Results on the propositional μ-calculus”. In: *Theoretical computer science* 27.3 (1983), pp. 333–354.

[16] Alexander Krauss and Tobias Nipkow. “Proof pearl: Regular expression equivalence and relation algebra”. In: *Journal of Automated Reasoning* 49 (2012), pp. 95–106.

[17] Tobias Nipkow and Dmitriy Traytel. “Unified decision procedures for regular expression equivalence”. In: *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 5*. Springer. 2014, pp. 450–466.

[18] David Park. “Fixpoint induction and proofs of program properties”. In: *Machine intelligence* 5 (1969).

[19] Nishant Rodrigues and Mircea Sebe. *A Logical Treatment of Finite Automata (Artifact)*. Dec. 2023. DOI: 10.5281/zenodo.10431211. URL: https://doi.org/10.5281/zenodo.10431211.

[20] Nishant Rodrigues and Mircea Sebe. *Matching Logic in MM0*. Oct. 2023. URL: https://github.com/formal-systems-laboratory/matching-logic-in-mm0 (visited on 04/14/2023).

[21] Nishant Rodrigues et al. *Technical Report: A Logical Treatment of Finite Automata*. Tech. rep. https://hdl.handle.net/2142/121770. 2024.

[22] Grigore Roşu. “Matching Logic”. In: *Logical Methods in Computer Science* Volume 13, Issue 4 (Dec. 2017). DOI: 10.23638/LMCS-13(4:28)2017. URL: https://lmcs.episciences.org/4153.

[23] Arto Salomaa. “Two complete axiom systems for the algebra of regular events”. In: *Journal of the ACM (JACM)* 13.1 (1966), pp. 158–169.

[24] Alfred Tarski et al. “A lattice-theoretical fixpoint theorem and its applications.” In: *Pacific journal of Mathematics* 5.2 (1955), pp. 285–309.

[25] The K Team. *KSummarizer*. 2022. URL: https://research.runtimeverification.com/#the-k-summarizer (visited on 10/16/2023).

[26] Wolfgang Thomas. “Languages, automata, and logic”. In: *Handbook of Formal Languages: Volume 3 Beyond Words*. Springer, 1997, pp. 389–455.

[27] Ken Thompson. “Programming Techniques: Regular Expression Search Algorithm”. In: *Commun. ACM* 11.6 (June 1968), pp. 419–422. ISSN: 0001-0782. DOI: 10.1145/363347.363387. URL: https://doi.org/10.1145/363347.363387.

[28] Boris Avraamovich Trakhtenbrot. “Finite automata and the logic of one-place predicates”. In: *Sibirskii Matematicheskii Zhurnal* 3.1 (1962), pp. 103–131.

[29] Dmitriy Traytel and Tobias Nipkow. “Verified decision procedures for MSO on words based on derivatives of regular expressions”. In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 3–12.