Shallow Embedding of Regular Expressions and Proof Certificates for Brzozowski's method

We axiomatize a theory of words in matching logic. Within this axiomatization, we define a shallow embedding for regular expressions and a representation for the language of DFAs. This enables us to write a certificate-generating implementation of Brzozowski's method. This certificate is a matching logic proof that only relies on the fifteen proof rules of matching logic, and twelve axioms for this theory of words.

ACM Reference Format:
. 2023. Shallow Embedding of Regular Expressions and Proof Certificates for Brzozowski's method. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 393 (January 2023), 32 pages.

1 MOTIVATION

Regular languages are foundational to Computer Science [Hopcroft et al. 2001]. They are used in diverse areas such as parsing, lexical analysis [Berglund and van der Merwe 2017], and natural language processing [Karttunen et al. 1996]. Their many representations such as NFAs, DFAs, grammars, and regular expressions as well as the transformations between them give us varied and powerful lenses through which we may tackle complex problems [Khoussainov and Nerode 2012].

While decidable, checking that an (extended) regular expression (ERE) is total (i.e. that its language includes all words over the alphabet) is a non-trivial endeavor. Decision procedures for EREs must deal with negation, constructors modulo associativity, and commutativity, as well as complex fixedpoint reasoning. It is a rich and complex arena with a lot of expressivity. All this has meant that regular-expressions have received their deserved attention from the verification community, and have been formally specified and checked several times over [Moreira et al. 2012].

However, regular expressions are not used in a void—they are used as part of larger software systems, for tasks such as parsing, lexical analysis and string manipulation. Modern software systems are large and intricate, involving complex algorithms and data-structures. Verifying each of these, as well as their interactions is time and labour intensive. Worse, retrofitting an existing system with formal guarantees is hard—it is often easier to rebuild the project from scratch with verification in mind.

An approach that avoids this issue is *verifiable computing* [Arkoudas and Bringsjord 2007]—instead of taking on the mammoth task of writing a verified system, we may write software that emits a proof of correctness that certifies that particular run of the program. Each step of each algorithm must be instrumented to emit enough information to construct such a proof. These may then be checked by a proof checker. Our trust base now becomes the proof checker—we no longer need to trust the entire system, but only the emitted proof. If we are confident in the correctness of the assumptions made, inference rules and the implementation of the proof checker, we can trust results produced by the system, regardless of the system's overall soundness.

The fewer and simpler the assumptions and inference rules that the proof checker uses, the easier it is for us to trust it. It is this principle of minimizing our trust base that has led us to make two particular choices—that of using *matching logic* [Chen and Roşu 2019b; Roşu 2017] as our foundation, and of using a *shallow embedding* [Boulton et al. 1992].

Matching logic's proof system [Chen and Roşu 2019b] is small and inspired by well-known foundational axioms. It consists of just 15 axioms, including those from propositional and first-order

Author's address:

2023. 2475-1421/2023/1-ART393 $15.00
https://doi.org/

logic and others closely related to modal logic, such as the axiom K and Park induction [Park 1969]. Despite the relative simplicity of these axioms, the proof system has proved powerful and practical. It has been used to formalize many logics and abstractions important to program verification, such as inductive datatypes, temporal logics, separation logic [Chen et al. 2020a; Chen and Roşu 2019b; Chen et al. 2020b], and most notably, through the $\mathbb{K}$ Framework [Chen and Roşu 2019a], to formalize small-step semantics of real world programming languages such as JavaScript [Park et al. 2015], C [Hathhorn et al. 2015] and x86 [Dasgupta et al. 2019]. This expressive power, despite the small size of the proof system has convinced us that matching logic is a good foundation for a proof checker.

Also with minimality in mind, we chose to formalize regular expressions as a *shallow embedding*. In a shallow embedding, expressions are translated directly to a semantically equivalent formulation in the host logic, whereas in a *deep embedding*, the grammar of the original logic is explicitly represented in the host logic, and explicit functions are provided for interpreting their meaning. This indirection, while powerful, brings in additional complexity to the formalization that we prefer to avoid.

Working in a compact logic, without the power of a deep embedding, of course, comes with downsides. Matching logic provides the (KNASTER-TARSKI) and (PRE-FIXEDPOINT) proof rules, akin to those for Park induction, that allow inductive reasoning. These, however, are intentionally very basic and minimal. It is only when combined with the other proof rules does it give us the power we need for induction and co-induction in their full generality. For example on its own, the (KNASTER-TARSKI) rule allows induction when the left hand side of the implication being proved is a least fixed point and cannot be applied when the fixedpoint is within an symbol application. In [Chen and Roşu 2019b], a theorem was proved allowing the use of two high-level proof rules (WRAP) and (UNWRAP). These rules were employed with great success in [Chen et al. 2020c] to produce the beginnings of an automated framework for fixedpoint reasoning that works for separation logic with inductive definitions, linear temporal logic and reachability logic.

These theorems, however, appear insufficient in the case of EREs. This is because these rules rely on the inductive structure of the definitions mirroring the inductive structure of the goal we are proving. We may, however, be proving that two regular expressions with completely different inductive structures are equivalent. In the case of EREs, this inductive structure is provided by the Kleene star operator. A naive application of the (KNASTER-TARSKI) proof rule (i.e. by directly using the inductive structure of the domain, or some fixedpoint in the ERE) did not get us far when trying to prove implications between expressions whose structures do not match.

To deal with this, we define a shallow representation of DFAs using matching logic's fixedpoint to capture cycles. This allows us to capture traditional algorithms such as Brzozowski's method [Brzozowski 1964], that reason about extended regular expressions via automaton theory. We believe that this technique will generalize to more complex arenas such as modal-μ calculus, where we have fixedpoint alternation (and so must employ alternating automata rather than DFAs), and also to term algebras modulo axioms based on decidablity procedures for the finite-variant property [Meseguer 2018].

Our interest in techniques for fixedpoint reasoning in matching logic is not just academic. Matching logic is also used as the foundations for $\mathbb{K}$, a language-agnostic framework for program verification. Besides the program and its formal specification, a language-agnostic verifier also takes a formal language semantics as input and uses the language-agnostic verification algorithm to prove the program correct directly using the language semantics. This avoids language specific logics, such as Hoare logic, or complex translation to an intermediate verification language à la Boogie.

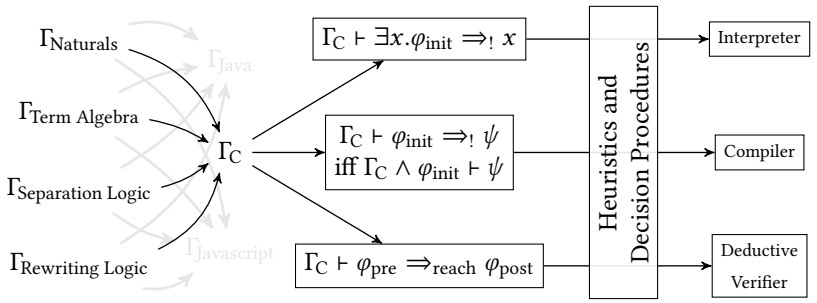

Fig. 1. The $\mathbb{K}$ Framework uses matching logic as its logical foundation, defining formal language semantics as matching logic theories. Programming language tasks, such as compilation, interpretation, and verificationn are viewed as heuristics for checking the validity particular matching logic formulae.

The $\mathbb{K}$ Framework may be viewed as a collection of best-effort heuristics for checking the validity of particular classes of matching logic formulae. For example, execution in a language L may be framed as $\Gamma_L \vdash \exists x.\, \varphi_{\text{init}} \Rightarrow_! x$ where x denotes a terminal state, while reachability may be framed as $\Gamma_L \vdash \varphi_{\text{pre}} \Rightarrow_{\text{reach}} \varphi_{\text{post}}$. Here, Γ_L is a matching logic theory formalizing the language L, $\Rightarrow_{\text{reach}}$ and $\Rightarrow_!$ represent the reachability and terminating reachability relations.

$\mathbb{K}$, unfortunately, is a complex and evolving software, implemented using around 550,000 lines of code using four different programming languages. So, despite its foundations in matching logic, formally verifying it is a gargantuan task. To resolve this, there is a nascent ecosystem for producing certificates [Chen et al. 2021; Lin et al. 2023] for the subprocedures used in verification tasks by instrumenting $\mathbb{K}$ to emit *proof hints* that may then be transformed into proof certificates for a Metamath formalization of matching logic.

This work is a natural extension to those, making headway on the problem of fixedpoint reasoning in matching logic. Here, we begin with regular expressions, and plan on extending our work to handle other types of fixedpoint reasoning such as for separation logic in [Brotherston et al. 2014] and for modal-μ calculus in [Niwiński and Walukiewicz 1996].

In this paper, we formalize the theory of words in matching logic, and develop shallow embeddings of regular expressions as well as representations of DFAs in this theory. We prove important theorems regarding Brzozowski derivatives [Brzozowski 1964], again defined as a shallow embedding. We prove Arden's rule using a small extension to Brzozowski's method for constructing DFAs, and use that to prove Salomaa's axiomatization of regular expression. Next, we use Maude [Maude Team [n.d.]] to implement Brzozowski's method, and use Maude's meta-level to produce machine-checkable Metamath Zero proofs in an evolution of the 240 line formalization of matching logic in Metamath from [Chen et al. 2021].

The rest of this paper is structured as follows: In Section 2, we discuss other formalizations of regular expressions. Section 3 introduces the various preliminaries regarding regular expressions, derivatives, Brzozowski's algorithm and matching logic. Section 4 describes the various idioms and infrastructure we use for formalizing the theory of words and regular expressions. In Section 5, we describe our theory, and outline proofs of important properties of EREs, and of Brzozowski derivatives in this formalization. We go on to show that we can certify runs of the decision procedure in matching logic, thus showing the completeness of our theory. With a small modification to Brzozowski's method, we show that we can also prove Arden's rule as a schema over regular expressions. This allows us to prove Salomaa's axiomatization. In Section 6, we describe our Maude-based proof generation for producing Metamath Zero proofs, and evalutate it against a set of

benchmarks (see Section 6.3). We conclude our paper with Section 8, describing our future plans and ideas.

> Complete proofs are presented in the appendix along with our source code, both available at [Redacted 2023].

2 RELATED WORK

Previous work on matching logic proof certificates: In [Chen et al. 2021], the authors formalize matching logic in Metamath[Megill and Wheeler 2019], and use this formalization to produce proof certificates for program execution. This is done by in a language-agnostic way, by instrumenting the $\mathbb{K}$ Framework. [Lin et al. 2023] takes this further, by producing certificates for verification tasks over some languaguages using $\mathbb{K}$'s deductive verifier. This involved producing extensive formalizations for many of $\mathbb{K}$'s abstraction such as sorts, constructors, maps, and importantly reachability logic. Reachability logic [Ştefănescu et al. 2016] is a framework for deductive verification employing co-inductive reasoning to prove partial correctness claims. While this does involve fixedpoint reasoning it is used in the proof of the reachability axioms, and does not involve structurally complex fixedpoint reasoning, and avoids complex domain reasoning. Our goal here is extend that work, by gaining a better understanding fixedpoint reasoning in matching logic. Eventually, we hope to be able to encode proof techniques, such as for separation logic in [Brotherston et al. 2011] and modal-μ calculus in [Niwiński and Walukiewicz 1996], that employ circular reasoning.

Our formalization of matching logic in Metamath *Zero* is an evolution of the *Metamath* one used in [Chen et al. 2021] and [Lin et al. 2023]. We chose Metamath Zero because it addresses some potential soundness issues in Metamath [Carneiro 2020], such as through ambiguities in user defined sugar, while also having a small trust base. Metamath Zero also has modern tooling, and is in the process of being formally verified.

Other formalizations: There are several other formalizations of regular expressions and languages. We will focus on couple of them. In [Coquand and Siles 2011], 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*. Similarly, [Krauss and Nipkow 2012] use Isabelle/HOL to formalize Brzozowski's derivatives. Interestingly, regular expressions are interpreted over binary relations and not just words.

The above formalizations define a *deep* embedding of regular expressions—that is, the syntax of regular expressions are defined as terms. The language of each expression is then defined as a membership relation, or a function from those terms to sets of lists of letters. This allows us to reason at an abstract level, such as by inducting on the structure of regular expressions and defining measures to prove termination. Theorems and proofs may be proved by appealing to the denotation of each regular expression. This allows us to express complex concepts such as termination of decision procedures.

The downside to this is that a trust base is large—our reasoning may bring in complex structures such as sets, lists, natural numbers, etc. We take an alternate approach, and use a shallow embedding. In a shallow embedding, regular expressions are themselves formulae in the logic. While we lose the ability to do many kinds of powerful meta-reasoning, we significantly reduce our trust base. We only assume the rules of the logic and the axioms of our theory. In our case, our formalization is a shallow embedding and each of the theorem proved follow from the axioms of associativity/commutatvity, etc, as well as the 15 fairly simple matching logic proof rules.

3 PRELIMINARIES

In this section, we review regular expressions and related concepts, as well as the syntax and semantics of matching logic.

3.1 Extended Regular Expressions and Brzozowski Derivatives

Let $A = \{a_1, a_2, \ldots a_n\}$ be a finite alphabet. Then extended regular expressions (ERE) over the alphabet A are defined using the following grammar:

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

The *language* for an ERE, denoted $\mathcal{L}(R)$ is defined as usual:

$$\begin{aligned}
\mathcal{L}(\emptyset) &= \emptyset \\
\mathcal{L}(\epsilon) &= \{\epsilon\} \\
\mathcal{L}(a) &= \{a\} \\
\mathcal{L}(R_1 + R_2) &= \mathcal{L}(R_1) \cup \mathcal{L}(R_2) \\
\mathcal{L}(R_1 \cdot R_2) &= \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(R_1) \text{ and } w_2 \in \mathcal{L}(R_2)\} \\
\mathcal{L}(R^*) &= \bigcup_{n=0}^{\infty} R^n \quad \text{where } R^0 = \epsilon, \text{ and } R^n = R \cdot R^{n-1} \\
\mathcal{L}(\neg R) &= A^* \setminus \mathcal{L}(R)
\end{aligned}$$

In [Brzozowski 1964], Brzozowski introduced an operation over languages called its derivative, denoted $\delta_a(R)$. This operation results in a language that is the result of "consuming" a letter from the beginning of each word in the language:

Definition 3.1 (Brzozowski Derivatives). Given a set R of sequences and a finite sequence s, the derivative of R with respect to s is denoted by $\delta_s(R)$ and is $\{t \mid s \cdot t \in R\}$.

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

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

$$\begin{aligned}
\emptyset|_\epsilon &= \emptyset \\
\epsilon|_\epsilon &= \epsilon \\
a|_\epsilon &= \emptyset \\
(\alpha_1 + \alpha_2)|_\epsilon &= \alpha_1|_\epsilon + \alpha_2|_\epsilon \\
(\alpha_1 \cdot \alpha_2)|_\epsilon &= \alpha_1|_\epsilon \wedge \alpha_2|_\epsilon \\
(\alpha^*)|_\epsilon &= \epsilon \\
(\neg\alpha)|_\epsilon &= \begin{cases} \epsilon & \text{if } \alpha|_\epsilon = \emptyset \\ \emptyset & \text{if } \alpha|_\epsilon = \epsilon \end{cases}
\end{aligned}$$

There are two properties of derivatives that are important to us in this paper. First, every ERE may be transformed into an equivalent form that partitions its language per the initial letter in its words:

Theorem 3.2 (Brzozowski Theorem 4.4). *Every extended regular expression R can be written in the form:*

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

where the terms in the sum are disjoint.

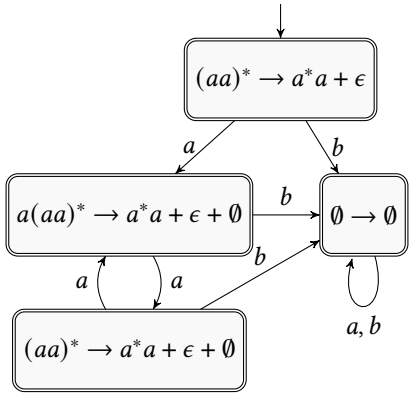

Fig. 2. DFA for the ERE $(aa)^* \longrightarrow a^*a + \epsilon$

Second, that repeatedly taking the derivative eventually converges (up to associativity, commutativity, and idemopotency of +):

Theorem 3.3 (Brzozowski Theorem 5.2). *Every extended regular expression has only a finite number of dissimilar derivatives.*

Here, two EREs are said to be dissimilar if they are not similar:

Definition 3.4. Two extended regular expressions are similar iff they are identical modulo associativity, commutativity and idemopotency of the + operator.

These two properties give rise to an algorithm for generating a DFA for the language of the expression, illustrated in Figure 2. We may construct a DFA where each state is identified with an ERE. The initial state is identified with the ERE whose validity we are checking. Each state transitions to the states identified by the derivative with respect to the input alphabet. A state identified by α is accepting iff $\alpha|_\epsilon$ is similar to ϵ. The first theorem mentioned implies that acceptance by this DFA implies membership in the the expressions languages, whereas the second shows us that this algorithm must terminate. Now, we can check if the ERE is valid by simply checking that all states are accepting.

3.2 Matching Logic

In this subsection, we will describe the syntax and semantics of matching logic formulae, called *patterns*. Matching logic patterns are defined over a matching logic signature.

Definition 3.5. Let EVar, SVar, Σ be disjoint countable sets. Here, EVar contains *element variables*, SVar contains *set variables* and Σ, called the *signature* contains a set of *symbols*.

By convention, element variables are denoted using lower case letters, whereas for set variables we use upper case letters.

Definition 3.6. A Σ-pattern over a signature Σ is defined inductively using the following grammar:

$$\varphi := \underbrace{\bot \mid \varphi_1 \longrightarrow \varphi_2}_{\text{propositional}} \mid \underbrace{\sigma \in \Sigma \mid (\varphi_1 \ \ \varphi_2)}_{\text{modal}} \mid \underbrace{x \in \text{EVar} \mid \exists x.\, \varphi}_{\text{quantification}} \mid \underbrace{X \in \text{SVar} \mid \mu X.\, \varphi}_{\text{fixedpoint}}$$

We call the pattern $(\varphi_1 \ \ \varphi_2)$ an *application*. In the case of fixedpoint patterns, $\mu X.\, \varphi$, the set variable X must occur positively in φ, that is, in a pattern, they may only occur under the left-hand side of

an implication an even number of times. The constructs $\exists$ and μ bind the element variable x and set variable X respectively. We use $\text{free}(\varphi)$ to denote the free (set or element) variables in a pattern φ.

We assume the usual syntactic sugar:

$$\neg\varphi \equiv \varphi \longrightarrow \bot \qquad \top \equiv \neg\bot$$
$$\varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \longrightarrow \varphi_2 \qquad \varphi_1 \wedge \varphi_2 \equiv \neg(\varphi_1 \longrightarrow \neg\varphi_2)$$
$$\forall x.\, \varphi \equiv \neg\exists x.\, \neg\varphi \qquad \nu X.\, \varphi \equiv \neg\mu x.\, \neg\varphi[\neg X/X]$$

Patterns are evaluated in a model:

Definition 3.7. Given a signature Σ, a Σ-*model* is the triple $(M, \_ \bullet \_, \{\sigma_M\}_{\sigma\in\Sigma})$ with:

(1) a non-empty carrier set M;
(2) a binary function $\_ \bullet \_ : M \times M \longrightarrow \mathcal{P}(M)$ called application;
(3) an interpretation $\sigma_M \subseteq M$ for each symbol $\sigma \in \Sigma$.

We extend $\_ \bullet \_$ pointwise to sets as follows:

$$\_\bar{\bullet}\_ : \mathcal{P}(M) \times \mathcal{P}(M) \longrightarrow \mathcal{P}(M)$$
$$A \bar{\bullet} B \mapsto \bigcup \{a \bullet b \mid a \in A, b \in B\}$$

Matching Logic patterns have a *pattern matching semantics*. That is, each pattern is evaluated to the set of elements in the model that match it. A matching logic model defines the evaluations of symbols and pattern applications. Element variables and set variables have evaluations determined by an variable valuation function ρ. An element variable is matched by precisely one element in the model, whereas a set variable may have arbitrary sets as their evaluation. No elements match $\bot$, while $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$ are matched by elements in the intersection and the union, respectively, of the evaluations of their sub-patterns. Formally,

Definition 3.8 (Matching logic semantics). An M-*valuation* is a function $\text{EVar} \cup \text{SVar} \longrightarrow \mathcal{P}(M)$, such that each $x \in \text{EVar}$ evaluates to a singleton. Let M be a model and ρ be an M-valuation, then the evaluations of matching logic patterns is defined inductively as follows:

$$|\bot|_{M,\rho} = \emptyset$$
$$|\sigma|_{M,\rho} = \sigma_M$$
$$|\varphi_1 \longrightarrow \varphi_2|_{M,\rho} = \left(M \setminus |\varphi_1|_{M,\rho}\right) \cup |\varphi_2|_{M,\rho}$$
$$|\varphi_1 \ \ \varphi_2|_{M,\rho} = |\varphi_1|_{M,\rho} \ \bar{\bullet} \ |\varphi_2|_{M,\rho}$$
$$|x|_{M,\rho} = \rho(x)$$
$$|\exists x.\, \varphi|_{M,\rho} = \bigcup_{a\in M} |\varphi|_{M,\rho[a/x]}$$
$$|X|_{M,\rho} = \rho(X)$$
$$|\mu X.\, \varphi|_{M,\rho} = \text{lfp}\left(A \mapsto |\varphi|_{M,\rho[A/X]}\right)$$

Since matching logic patterns are set-valued, rather than true/false as in first-order logic, defining validity is slightly more involved.

Definition 3.9. For a model M, we say that a pattern φ is *valid* in M, iff $|\varphi|^{\rho}_{M} = M$ for all M-valuations ρ. Let Γ be a set of patterns, called a *theory*. We say a model M *satisfies* Γ, if each γ in Γ is valid in M. A pattern φ is *valid* in a theory Γ, written $\Gamma \vDash \varphi$, if for every model M satisfying Γ, φ is valid.

FOL Rules	(Propositional 1)	$\varphi \to (\psi \to \varphi)$
	(Propositional 2)	$(\varphi \to (\psi \to \theta)) \to ((\varphi \to \psi) \to (\varphi \to \theta))$
	(Propositional 3)	$((\varphi \to \bot) \to \bot) \to \varphi$
	(Modus Ponens)	$\dfrac{\varphi \qquad \varphi \to \psi}{\psi}$
	($\exists$-Quantifier)	$\varphi[y/x] \to \exists x.\, \varphi$
	($\exists$-Generalization)	$\dfrac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi}$ where $x \notin FV(\psi)$
Frame Rules	(Propagation$_\bot$)	$C[\bot] \to \bot$
	(Propagation$_\vee$)	$C[\varphi \vee \psi] \to C[\varphi] \vee C[\psi]$
	(Propagation$_\exists$)	$C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$ where $x \notin FV(C)$
	(Framing)	$\dfrac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$
Fixedpoint Rules	(Prefixpoint)	$\varphi[(\mu X.\, \varphi)/X] \to \mu X.\, \varphi$
	(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{(\mu X.\, \varphi) \to \psi}$
Technical Rules	(Existence)	$\exists x.\, x$
	(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$
	(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$

Fig. 3. Matching logic proof system (where C, C_1, C_2 are application contexts and $C[\varphi] \equiv C[\varphi/\Box]$).

The matching logic proof system, shown in Figure 3, defines the provability relation, written $\Gamma \vdash \varphi$, meaning that φ can be proved using the proof system with the patterns in Γ as additional axioms. We call Γ a matching logic theory. To understand this proof system, we must first define application contexts.

Definition 3.10. A *context* is a pattern with a hole variable $\Box \in \text{EVar} \cup \text{SVar}$. We write $C[\varphi] \equiv C[\varphi/\Box]$ as the result of context plugging. We call C an application context, iff

(1) $C \equiv \Box$ is the identity context
(2) $C \equiv (\varphi \;\; C')$ or $C \equiv (C' \;\; \varphi)$, where C' is an application context and $\Box \notin \text{free}(\varphi)$.

These proof rules are sound with respect to the semantics, and fall into four categories. First, the FOL rules provide complete FOL reasoning. The frame rules allow application to commute with constructs such as disjunctions and exististentials—they allow us to reason about the union semantics of these constructs. When taken together, unary versions of (Propagation$_\bot$), (Propagation$_\vee$) and (Framing), are equivalent to modal logics axioms K and necessitation axioms. The proof rule (Knaster-Tarski) is an embodiement of the Knaster-Tarski fixedpoint theorem [Tarski et al. 1955], and together with (Prefixedpoint) correspond to the Park induction rules of modal logic [Kozen 1983; Park 1969].

Symbols: def
Axioms: $\forall x.\ \lceil x \rceil$ (DEFINEDNESS)
Notation: $\lceil \varphi \rceil \equiv (\text{def}\ \ \varphi)$
$\lfloor \varphi \rfloor \equiv \neg \lceil \neg \varphi \rceil$

Notation:
$x \in \varphi \equiv \lceil x \wedge \varphi \rceil$
$\varphi \subseteq \psi \equiv \lfloor \varphi \rightarrow \psi \rfloor$
$\varphi = \psi \equiv \lfloor \varphi \leftrightarrow \psi \rfloor$

Fig. 4. The theory of definedness

3.3 Metamath Zero

Metamath Zero (MM0) is a language for writing proofs and specifications. Proof checking frameworks must balance simplicity of mechanical verification with human readability of the specification. Inspired by Metamath and Lean, it sits between the two on this spectrum. Importantly, MM0 solves issues in Metamath that make it easy for the user to introduce unsoundness [Carneiro 2020]. At the same time, Metamath Zero has a small trusted code base of under ~2600 lines of C code. MM0 proofs are S-expressions where the first argument is the name of the applied theorem or axiom, and the remaining arguments are proofs of the antecedents for that theorem.

3.4 The Maude System

Maude [Maude Team [n.d.]] is an implementation of equational and rewriting logic. Maude modules consist of a set of sorted operators, and equations and rewriting rules over term constructed using those operators. The ease of representing transition systems using rewrite rules, and powerful matching and unification features make it ideal for representing logical proof systems proof systems. Additionally, the ease of reflection via Maude's meta-level let us easily compute the DFA and derivative tree corresponding to a regular expression.

4 MATCHING LOGIC IDIOMS

Before introducing our theory, let us look at some common idioms used when defining matching logic theories and proofs.

Notation. In order to compactly represent complex logics, matching logic allows defining syntactic sugar over patterns. Notation is used for defining "standard" sugar such as $\neg$, ν, as well as complex constructs such as equality, the Kleene star operator, and contextual implications described below.

Predicate patterns. Unlike first-order logic formulae, matching logic patterns may be interpreted as any subset of the carrier set, not just as either true or false. Predicate patterns are a special class of matching logic patterns, that can only be interpreted as either the empty set or the carrier set, depending on the valuation. Predicate patterns of course include $\top$, $\bot$ and all propositional tautologies.

Definedness. The theory of definedness is a foundational matching logic theory that allows us to succinctly capture equality and other important predicates. As shown in Figure 4, it includes one symbol, def. When applied it gives rise to the *ceiling* operator, abbreviated $\lceil \varphi \rceil$. The axiom, (DEFINEDNESS), makes $|\lceil \varphi \rceil| = M$ for any φ with non-empty evaluation.

The axiom (DEFINEDNESS) forces $\lceil \varphi \rceil$ to be a predicate pattern for any φ. In particular, if φ has a non-empty evaluation, then $|\lceil \varphi \rceil| = M$. Otherwise, it has $\emptyset$ as its evaluation. Dually, $\lfloor \varphi \rfloor$ (pronounced *floor of* φ) is evaluated as top if and only if the evaluation of φ is top. This allows us to define some important predicate patterns such as equality ($\varphi = \psi \equiv \lfloor \varphi \leftrightarrow \psi \rfloor$), and subset ($\varphi \subseteq \psi \equiv \lfloor \varphi \rightarrow \psi \rfloor$).

Functional patterns. Another interesting class of patterns are *functional* patterns. Functional patterns are patterns that are interpreted as singleton sets. These behave like terms in first order logic. The semantics of matching logic ensures that element variables are functional patterns. We may add axioms to enforce that symbols have functional interpretations—i.e. they return a single output applied to a singleton input by adding the following axiom to the theory: $\forall \bar{y}.\, \exists x.\, x = (f\ \ \bar{y})$. The application of these symbols may then be used along with element variables to construct functional patterns. For example (succ zero) and (succ x) are both functional patterns in the naturals.

Fixedpoint patterns. A pattern φ, in which X occurs only positively, induces a *monotonic* function $\mathcal{F} : \mathcal{P}(M) \to \mathcal{P}(M)$, where $\mathcal{F}(A) \mapsto |\varphi|^M_{\rho[A/X]}$ [Chen and Roşu 2019b]. By the Knaster Tarski fixedpoint theorem [Tarski 1955], every monotonic function has a least- and greatest-fixedpoint. Matching logic's μ construct lets us capture this fixedpoint explicitly. For example, the axiom $\mu X.\, \text{zero} \vee (\text{succ}\, X)$ lets us capture the inductive nature of the naturals, and, assuming the other necessary axioms, precisely pins down the standard model of the naturals [Chen and Roşu 2019b]. This means, that the Löwenheim–Skolem theorem [Skolem 1879] does not hold for matching logic, and that by Godel's theorems [Gödel 1931], matching logic is *not* complete. At the same time, matching logic is strictly more expressive than first-order logic [Chen and Roşu 2019b].

Sorts. While matching logic does not provide a built-in concept of sorts, they may easily be defined axiomatically. Since, in this paper, we only deal with a simple sort hierarchy we will not go into this in detail. When dealing with a sort s, typically, its domain is represented by the pattern $\top_s$, read *inhabitants of* s. We may add an axiom to constrain that pattern to represent the domain of s. For example, the axiom $\top_{\mathbb{N}} = \mu X.\, \text{zero} \vee (\text{succ}\ \ X)$ constrains $\top_{\mathbb{N}}$ to the domain of the naturals. Further, we can constrain the domain of symbols. For instance, the axiom $\mathbb{N} + \mathbb{N} \to \mathbb{N}$ forces the symbol + to always return a natural when given two naturals as input. Note that it does not constrain the symbol + in any way for arguments outside of the naturals. For example, we could extend + to the domain of integers or even to concatenation of lists without any change to this axiom. Finally, we can quantify over the elements of a sort. For any sort s, the notation $\forall x : s.\, \varphi \equiv \forall x.\, x \in \top_s \wedge \varphi$ allows us to quantify over all elements restricted to the domain of s.

Contextual implications. In order to enable fixedpoint reasoning, matching logic includes two rules: (PREFIXEDPOINT) and (KNASTER-TARKSI). While these rules are powerful, they are also very minimalistic. For example to apply (KNASTER-TARKSI), key to any inductive proof, we *must* have a least fixedpoint pattern on the left-hand side of an implication. We may not, for example, have it within an application context. We may need this in the case of trying to prove, say even + even $\to$ even, where even $\equiv \mu X.\, \text{zero} \vee (\text{succ}\ (\text{succ}\ \ X))$.

For any application context C and pattern φ (where $\Box \notin \text{free}(\varphi)$), we call $C \multimap \varphi \equiv \exists \Box.\, \Box \wedge (C[\Box] \subseteq \varphi)$ a *contextual implication*. Informally, the pattern evaluates to the set of elements that when plugged into the context C imply φ. Using contextual implications, we may use the inferred rule (WRAP) to pull any pattern out of an applicative context, apply the (KNASTER-TARSKI) rule, and then plug the result back into the context using (UNWRAP). These two rule let us work around one of the major limitations of the (KNASTER-TARSKI) rule. This is explained in more detail in [Chen et al. 2020c].

5 THEORY OF WORDS IN MATCHING LOGIC

Now that our foundations are in place we are ready to expound on our theory of words in matching logic, shown in Figure 6. First, our theory "imports" the theory Definedness. That is, it includes all its axioms, symbols and notation. This allows us to use definedness, and thus equality in our

$$\text{(WRAP)}\quad \frac{\varphi \longrightarrow (C \multimap \psi)}{C[\varphi] \longrightarrow \psi} \qquad \frac{C[\varphi] \longrightarrow \psi}{\varphi \longrightarrow (C \multimap \psi)}\quad \text{(UNWRAP)}$$

Fig. 5. The (WRAP) and (UNWRAP) derived proof rules, first proved in [Chen and Roşu 2019b] and effectively employed in [Chen et al. 2020c] in the context of separation logic, reachability logic and LTL.

Imports: Definedness
Sorts: Letter, Word
Symbols: ϵ, $\_ \cdot \_$, and a for each $a \in A$.
Notation:

$$\emptyset \equiv \bot$$
$$(R_1 + R_2) \equiv R_1 \vee R_2$$
$$R^* \equiv \mu X.\, \epsilon \vee (X \cdot R)$$
$$\top_{\text{Letter}} \equiv \bigvee_{a \in A} a$$
$$\top_{\text{Word}} \equiv \top_{\text{Letter}}{}^*$$

Axioms:

$$\top_{\text{Word}} \qquad \text{(DOMAIN-WORD)}$$

For each $a \in A$,

$$\exists x : \text{Letter}.\, a = x \qquad \text{(FUNCTIONAL}_a\text{)}$$
$$\exists w : \text{Word}.\, \epsilon = w \qquad \text{(FUNCTIONAL}_\epsilon\text{)}$$
$$\forall w, v : \text{Word}. \exists u : \text{Word}.\, w \cdot v = u \qquad \text{(FUNCTIONAL}_\bullet\text{)}$$

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

$$a \neq b \qquad \text{(NO-CONFUSION}_a\text{)}$$
$$\epsilon \notin \top_{\text{Letter}} \qquad \text{(NO-CONFUSION}_\epsilon\text{)}$$

$\forall u, v : \text{Word}.$

$$\epsilon = u \cdot v \longrightarrow u = \epsilon \wedge v = \epsilon \qquad \text{(NO-CONFUSION}_\bullet\text{-1)}$$

$\forall x, y : \text{Letter}, u, v : \text{Word}.$

$$x \cdot u = y \cdot v \longrightarrow x = y \wedge u = v \qquad \text{(NO-CONFUSION}_\bullet\text{-2)}$$

$\forall u, v, w : \text{Word}.$

$$(u \cdot v) \cdot w = u \cdot (v \cdot w) \qquad \text{(ASSOCIATIVE)}$$
$$\forall x.\, (\epsilon \cdot x) = x \qquad \text{(IDENTITY-L)}$$
$$\forall x.\, (x \cdot \epsilon) = x \qquad \text{(IDENTITY-R)}$$

Fig. 6. The theory of words in Matching Logic.

theory. The signature introduces the expected sorts and symbols. We also define the empty set, choice and Kleene star operator as notation.

The first two axioms define the inhabitants of our two sorts. The next three axioms restrict the symbols to functional interpretations. The (NO-CONFUSION) axioms ensure that the interpretations of symbols are injective (modulo associativity, commutativity, and identity). These are similar to those in [Chen et al. 2020a] for defining term algebras, with the caveat that we must take into account the associativity of + and its identity of ϵ. That is, we must relax them as compared to those in [Chen et al. 2020a]. We must also add axioms the axioms (ASSOCIATIVE), and (IDENTITY-L), and (IDENTITY-R) in order to enforce these properties and enable their use in proofs.

We must now show that this theory in matching logic actually captures the theory of words. To do so, we will prove the following theorem.

THEOREM 5.1. *For any ERE α, the following are equivalent:*

(1) $\Gamma_{\text{Word}} \vdash \alpha$
(2) $\Gamma_{\text{Word}} \vDash \alpha$
(3) $A^* = \mathcal{L}(\alpha)$

The soundness of matching logic [Chen and Roşu 2019b] tells us that (1) implies (2). We complete the proof with the following lemmas:

LEMMA 5.2 (SOUNDNESS). *If $\Gamma_{\text{Word}} \vDash \alpha$, then $A^* = \mathcal{L}(\alpha)$.*

LEMMA 5.3 (COMPLETENESS). *If $A^* = \mathcal{L}(\alpha)$, then $\Gamma_{\text{Word}} \vdash \alpha$.*

The proof for Lemma 5.2 is straightforward, and we refer the reader to the Appendix. The rest of this section is dedicated to proving 5.3. We will first assemble a toolbox.

5.1 From Words to Languages

Now that we have a theory of words, our next task becomes immediately apparent. Each of the axioms above quantify over words. However, we it is more convenient to deal with language, i.e. sets of words. In matching logic, the natural way to deal with this is to "lift" these axioms from element variables to set variables. The proofs for these are involved and technical, but not particularly enlightening.

Theorem 5.4. *The following can be proved:*

$$\Gamma_{\text{Word}} \vdash (X \cdot Y) \cdot Z = X \cdot (Y \cdot Z) \qquad \text{(assoc)}$$
$$\Gamma_{\text{Word}} \vdash \epsilon \cdot X = X \qquad \text{(identity-l)}$$
$$\Gamma_{\text{Word}} \vdash \neg(a \cdot X \wedge b \cdot Y) \qquad \text{(no-confusion-ab-l)}$$
$$\Gamma_{\text{Word}} \vdash \neg(\epsilon \wedge a \cdot X) \qquad \text{(no-confusion-}\epsilon\text{-l)}$$
$$\Gamma_{\text{Word}} \vdash X \cdot \epsilon = X \qquad \text{(identity-right)}$$
$$\Gamma_{\text{Word}} \vdash \neg(X \cdot a \wedge Y \cdot b) \qquad \text{(no-confusion-ab-r)}$$
$$\Gamma_{\text{Word}} \vdash \neg(\epsilon \wedge X \cdot a) \qquad \text{(no-confusion-}\epsilon\text{-r)}$$

Another important tool is having alternate formulations of the domain axioms for words. This lets us induct using different bases on either the left or on the right. The proofs of these are via the (wrap) and (unrwap) rules shown earlier.

Lemma 5.5 (Inductive Domain). *The following patterns are equivalent in* Γ_{Word}*:*

(1) $\top_{\text{Word}}$ *(2)* $\mu X.\, \epsilon \vee \top_{\text{Letter}} \cdot X$ *(3)* $\mu X.\, \epsilon \vee \top_{\text{Letter}} \vee X \cdot X$

5.2 Brzozowski Derivatives in Matching Logic

In a somewhat surprising connection, Brzozowski derivatives are exactly contextual implications. For any word w and pattern ψ, we may define its Brzozowski derivative as the matching logic pattern $\delta_w(\psi) \equiv (w \cdot \Box \multimap \psi)$. This connection is quite striking and intriguing because it closely parallels the magic wand operator of separation logic: $\varphi \mathrel{-\!\!*} \psi \equiv \varphi * \Box \multimap \psi$. At first glance, this seems like a somewhat weak connection, but on closer inspection, the magic wand and derivatives are semantically quite similar—we may think of the magic want operator as taking the derivative of one head 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.

Let us review contextual implications. For any pattern ψ and context C, we define a contextual implication as the pattern $C \multimap \psi \equiv \exists \Box.\, \Box \wedge (C[\Box] \subseteq \psi)$. For any context, a contextual implication has as its evaluation the set of elements that when plugged into the context C imply the pattern ψ. So, for $C \equiv a \cdot \Box$, we get the set of words that when prefixed with a give us ψ—exactly what we need. Note that this definition also works for arbitrary words and not just single letters.

The following theorem, key to the certification, follows from properties of contextual implications: for any functional context C, we have $\vdash C[\top] \wedge \varphi \leftrightarrow C[C \multimap \varphi]$.

Lemma 5.6 (Brzozowski Derivatives). *For any regular expression* β*,*

$$\Gamma_{\text{Word}} \vdash \beta = (\epsilon \wedge \beta) \vee \bigvee_{a \in A} a \cdot \delta_a(\beta)$$

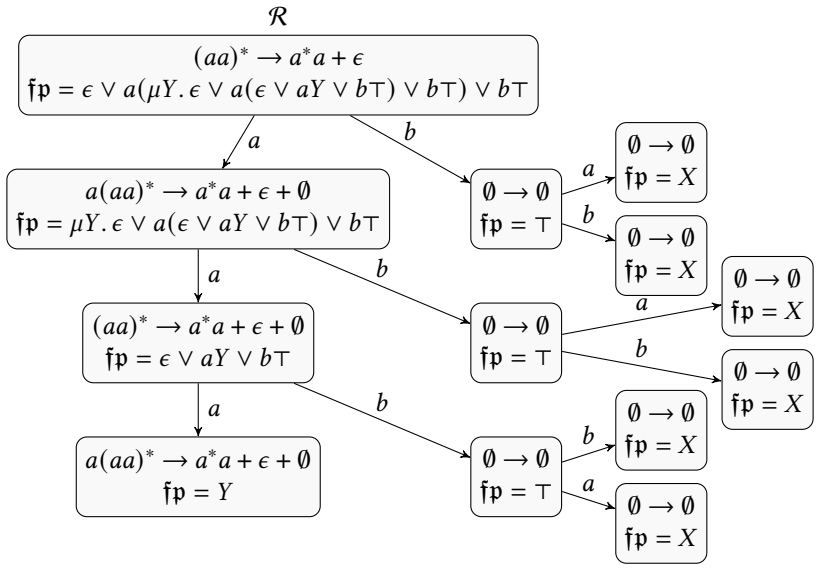

Fig. 7. The unfolding tree for the same ERE, where $\top \equiv \mu X.\, \epsilon \vee aX \vee bX$. Here, the first line of each node n is a regular expression corresponding to $L(n)$, whereas the second line shows $\mathfrak{fp}(n)$.

From the above notation for derivatives, we may also prove the syntactic simplifications, representing $\alpha|_\epsilon$ quite simply as $\alpha \wedge \epsilon$:

Lemma 5.7 (Derivatives Simplification). *For regular expressions α and β and a letter a, and every $b \neq a$, the following hold:*

(1) $\Gamma_{\text{Word}} \vdash \delta_a(\epsilon) = \emptyset$
(2) $\Gamma_{\text{Word}} \vdash \delta_a(\emptyset) = \emptyset$
(3) $\Gamma_{\text{Word}} \vdash \delta_a(b) = \emptyset$
(4) $\Gamma_{\text{Word}} \vdash \delta_a(a) = \epsilon$
(5) $\Gamma_{\text{Word}} \vdash \delta_a(\alpha_1 + \alpha_2) = \delta_a(\alpha_1) + \delta_a(\alpha_1)$
(6) $\Gamma_{\text{Word}} \vdash \delta_a(\alpha_1 \cdot \alpha_2) = \delta_a(\alpha_1) \cdot \alpha_2 + (\alpha_1 \wedge \epsilon) \cdot \delta_a(\alpha_2)$
(7) $\Gamma_{\text{Word}} \vdash \delta_a(\neg\alpha) = \neg\delta_a(\alpha)$
(8) $\Gamma_{\text{Word}} \vdash \delta_a(\alpha^*) = \delta_a(\alpha) \cdot \alpha^*$

5.3 Representing DFAs

In this subsection, we will define a pattern $\mathfrak{fp}_Q$ that captures the language of a DFA Q as a shallow embedding—that is a pattern whose denotation is the same as the language of Q. The algorithm described in Section 3 for checking the validity of EREs above produces a DFA. These DFAs are represented as directed graphs that may have cycles. We must find some way of encoding these. These cycles correspond to the inductive structure of the language they define, so the natural way to do so is to represent them using fixedpoint patterns. A state in the cycle may be represented a fixedpoint pattern that binds a variable, whereas returning to that state may be represented as the corresponding set variable.

Definition 5.8. A *deterministic finite automata* is a tuple (Q, A, δ, q_0, F), where

(1) Q is a finite set of *states*,
(2) A is a finite set of input symbols called the *alphabet*,

(3) $\delta : Q \times A \to Q$ is a *transition function*,
(4) $q_0 \in Q$ is the *initial state*, and
(5) $F \subseteq Q$ is the set of *accepting states*.

By duplicating certain nodes, the *unfolding tree* of a DFA allows us to represent a DFA as a tree with backlinks.

Definition 5.9. For a DFA $\mathcal{Q} = (Q, A, \delta, q_0, F)$, its *unfolding tree* is a labeled tree (N, E, L) where N is the set of nodes, E is the edge relation, and $L : N \to Q$ is a labeling function. It is defined inductively as follows:

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

We will now define a secondary labeling function, $\mathfrak{fp} : N \to \mathsf{Pattern}$ over this tree, and use that to define $\mathfrak{fp}_{\mathcal{Q}}$.

Definition 5.10. Let (N, E, L) be an unfolding tree for a DFA (Q, A, δ, q_0, F). Let $X : Q \to \mathsf{SVar}$ be an injective function. Then, we define $\mathfrak{fp}$ recursively as follows:

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

$$\mathfrak{fp}(n) = \begin{cases} \epsilon \vee \bigvee_{a \in A} a \cdot \mathfrak{fp}(n_a) & \text{if } L(n) \text{ is accepting.} \\ \bigvee_{a \in A} a \cdot \mathfrak{fp}(n_a) & \text{otherwise.} \end{cases}$$

   (b) if n has a descendant with the same label, then:

$$\mathfrak{fp}(n) = \begin{cases} \mu X(L(n)).\, \epsilon \vee \bigvee_{a \in A} a \cdot \mathfrak{fp}(n_a) & \text{if } L(n) \text{ is accepting.} \\ \mu X(L(n)).\quad \bigvee_{a \in A} a \cdot \mathfrak{fp}(n_a) & \text{otherwise.} \end{cases}$$

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

Note that the case for (2a) is quite similar to (2b), except that we "name" the node by binding the variable $X(L(n))$. This distinction isn't strictly necessary—we could name all nodes while only superficially affecting the proof. The bound variable wouldn't occur under the fixedpoint pattern. However, we find that making this distinction allows us to clearly embody the inductive structure as a pattern. Figure 7 shows the unfolding tree for $\beta = (aa)^* \to a^*a + \epsilon$.

Note that all leaves in this tree have as labels states that complete a cycle in the original DFA. These have $\mathfrak{fp}(n) = X(L(n))$, a set variable. This set variable is always bound pattern by the $\mathfrak{fp}(a)$, where a is the ancestor node also labeled by $L(n)$. For a node n, we use n_a to refer to the child whose label is $\delta(L(n), a)$. The pattern $\mathfrak{fp}_\beta$ captures the recursive structure of the DFA as a fixedpoint pattern, and allows us to use the (Knaster-Tarski) to easily.

5.4 Producing Proof Certificates

Producing a proof certificate for a valid regular expression β involves proving two lemmas. First, that $\Gamma_{\mathsf{Word}} \vdash \top_{\mathsf{Word}} \to \mathfrak{fp}_{\mathcal{Q}}$, i.e. that the domain implies the representation of the DFA, and second, proving that $\Gamma_{\mathsf{Word}} \vdash \mathfrak{fp}_{\mathcal{Q}} \to \beta$. Here $\mathcal{Q}$ is the DFA for β produced via Brzozowski's method.

5.4.1 Phase 1: Proving: $\top_{\text{Word}} \longrightarrow \mathfrak{fp}_Q$. This part of the certification is straight-forward. Once we have proved Lemma 5.11, the proof becomes its application in a structure reflecting that of the unfolding tree, with minor additions for proving that applications of (KNASTER-TARSKI) are well-formed (i.e. that the set variable occurs only positively), and for propagating substitutions in.

LEMMA 5.11. *Let n in be a node in the unfolding of tree of a* valid *regular expression. Then:*

(1) if n is a leaf node, $\Gamma_{\text{Word}} \vdash \mathfrak{fp}_\beta(n)[\Theta_n] \cdot \top_{\text{Letter}} \longrightarrow \mathfrak{fp}_\beta(n)[\mathrm{U}_n]$, and
(2) if n is an interior node,

$$\frac{\mathfrak{fp}_\beta(n_a)[\Theta_{n_a}] \cdot \top_{\text{Letter}} \longrightarrow \mathfrak{fp}_\beta(n_a)[\mathrm{U}_{n_a}] \text{ for each } a \in A}{\mathfrak{fp}_\beta(n)[\Theta_n] \cdot \top_{\text{Letter}} \longrightarrow \mathfrak{fp}_\beta(n)[\mathrm{U}_n]}$$

where,

$$\Theta_n = \begin{cases} \lambda & \text{when } n = \mathcal{R} \\ \Theta_p[\Psi_p/X_{L(p)}] & \text{when } \mathfrak{fp}_\beta(p) \text{ binds } X_{L(p)} \\ \Theta_p & \text{otherwise.} \end{cases}$$

$$\Psi_p = \square \cdot a \multimap \mathfrak{fp}_\beta(p)[\mathrm{U}_p]$$

$$\mathrm{U}_n = \begin{cases} \lambda & \text{when } n = \mathcal{R} \\ \mathrm{U}_p[\mathfrak{fp}_\beta(p)/X_{L(p)}] & \text{when } \mathfrak{fp}_\beta(p) \text{ binds } X_{L(p)} \\ \mathrm{U}_p & \text{otherwise.} \end{cases}$$

5.4.2 Phase 2: Proving: $\mathfrak{fp}_Q \longrightarrow \beta$. This second part of the proof is more complex since it involves showing that the DFA representation is equivalent to the regular expression. For this, we need more details than the unfolding tree provides—we need to show that it is not just any DFA, but rather a representation of the Brzozowski DFA correpsonding to that particular expression.

Let us represent this as a structure called a *proof hint*. This tree has three types of nodes,

(1) *Derivative nodes* correspond to the application of Lemma 5.6. They are labeled only with the regular expression α, corresponding to the language of that node and have a child node for each letter a in A each labeled with the matching logic pattern $\delta_a(\alpha)$. It is important to note that the children's labels are precisely this pattern and not simplifications of it. That is, if $\alpha = b$, then the label would be $\delta_a(b)$, and not $\emptyset$.
(2) *Simplification nodes* store information regarding the application of simplification rules—the type of simplification applied (that is, the theorem/lemma applied), the context in which they were applied, as well as the substitution with which they were applied. These simplifications may be those from Theorem 5.7, associativity, commutativity and identity of choice, as well as additional simplifications, such as removing double negations and double Kleenes, used for reducing the size of the DFA.
(3) Finally, *backlink nodes* represent the completion of cycles.

The translation of derivative nodes and backlink nodes is as in the previous case: we simply apply Lemma 5.12 for each interior and leaf node of the proof hint.

LEMMA 5.12. *Let n in be a node in the unfolding of tree of a* valid *regular expression. Then:*

(1) if n is a leaf node, $\Gamma_{\text{Word}} \vdash \mathfrak{fp}(n)[\Lambda_n] \longrightarrow L(n)$, and
(2) if n is an interior node,

$$\frac{\epsilon \longrightarrow L(n) \qquad \mathfrak{fp}(n_a)[\Lambda_{n_a}] \longrightarrow L(n_a) \text{ for each } a \in A}{\mathfrak{fp}(n)[\Lambda_n] \longrightarrow L(n)}$$

$$\text{where, } \Lambda_n = \begin{cases} \lambda & \text{when } n = \mathcal{R} \\ [L(p)/X_{L(p)}] & \text{when } \mathfrak{fp}_\beta(p) \text{ binds } X_{L(p)} \\ \Lambda_p & \text{otherwise.} \end{cases}$$

Simplification nodes are more complex. In this case, we need to lift each of the simplification theorems into the context in which they are applied. We do so using a set of theorems, called *congruence of equivalence*. These theorems are of the form: $\varphi \leftrightarrow \psi$ implies $\Gamma_{\text{Word}} \vdash C[\varphi] \leftrightarrow C[\psi]$ and allow replacing equivalent patterns in arbitary contexts. While these theorems were proved as a single theorem on paper in [Chen and Roşu 2019c], at our level of represention it becomes a meta-theorem, requiring induction on the structure of patterns for its proof. So, we need to turn each of the base and inductive cases into theorems of their own. For example, if we want to apply the simplification (DER-CHOICE), $\delta_a(\varphi + \psi) \leftrightarrow \delta_a(\varphi) + \delta_a(\psi)$, to the pattern $a \cdot \delta_a(\varphi + \psi) \rightarrow \gamma$. This simplification happens within the context $a \cdot \Box \rightarrow \gamma$. So, we must first apply (DER-OR) to the theorems (CONG-OF-EQUIV-IMPLICATION-LEFT), and (CONG-OF-EQUIV-CONCAT-RIGHT) to give us $\Gamma_{\text{Word}} \vdash (a \cdot \delta_a(\varphi + \psi) \rightarrow \rho) \leftrightarrow (a \cdot \delta_a(\varphi) + a \cdot \delta_a(\psi) \rightarrow \rho)$. This equivalence may then be used to apply the desired simplification.

5.5 Proving Arden's Rule and Salomaa's Axiomatization

Arden's rule is a foundational theorem regarding languages. In a sense, it captures the initiality of words, and distinguishes it from say ω-words or streams. So, even though it does not relate directly to our completeness result, we will show that we can prove it, and use it to prove Salomaa's complete axiomatization..

THEOREM 5.13 (ARDEN'S RULE). *Let A and B be languages over an alphabet Σ. Then the equation $X = A \cdot X + B$ has a solution $X_0 = A^* \cdot B$. If $\epsilon \notin A$, then this solution is unique. In any case X_0 is the smallest solution.*

Arden's rule can be stated quite succinctly in matching logic:

THEOREM 5.14. *$\alpha \cdot \beta^*$ is the least solution for $X = \alpha \vee \beta \cdot X$. Further, if $\epsilon \notin \beta$, this solution is unique. We may formalize this in matching logic as:*

$$\Gamma_{\text{Word}} \vdash \beta^* \cdot \alpha = \mu X.\, \alpha \vee \beta \cdot X \qquad \text{(Least Solution)}$$

$$\Gamma_{\text{Word}} \vdash \epsilon \notin \beta \rightarrow \beta^* \cdot \alpha = \nu X.\, \alpha \vee \beta \cdot X \qquad \text{(Greatest Solution)}$$

This proof is particularly interesting because it involves proving that a greatest fixedpoint implies a least fixedpoint (in Greatest Solution), something the matching logic doesn't give us the tools to do out of the box. To prove this, we extend derivatives to handle the above greatest fixedpoint pattern and then use our previous completeness result to prove is as a schema over β. When $\beta|_\epsilon = \emptyset$, we have:

$$\begin{aligned} \delta_a(\nu X.\, \alpha \vee \beta \cdot X) &= \delta_a(\alpha \vee \beta \cdot (\nu X.\, \alpha \vee \beta \cdot X)) \\ &= \delta_a(\alpha) \vee \delta_a(\beta) \cdot (\nu X.\, \alpha \vee \beta \cdot X) \end{aligned}$$

Note that when $\epsilon \in \beta$, this simplification becomes non-terminating. It is easy to see that $(\nu X.\, \alpha \vee \beta \cdot X) = \alpha \vee \beta \cdot (\nu X.\, \alpha \vee \beta \cdot X)$, and so this extension preserves Lemma 5.6. It is also clear that Brzozowski's Theorem 5.2, stating that the derivatives of a regular expression form a finite set, is preserved. Instances of Arden's rule when α and β are regular expressions may now be proved easily through using the technique in the previous section. This proof gives us the last piece of the puzzle needed to prove Salomaa's axiomatization of (unextended) regular expressions shown in the Appendix.

6 IMPLEMENTATION

The realization of the theory presented in the previous section as concrete proof certificates involves a few parts. First, we must formalize matching logic and the theory of words in Metamath Zero. Next, we must prove each of the lemmas and theorems using that formalization. Finally, we must implement Brzozowski's method and intrument it to produce proof hints. Let us discuss each of these in brief.

6.1 Metamath Zero formalization

Our formalization of matching logic is an evolution of the one in [Chen et al. 2021]. We chose to port this formalization to Metamath Zero [Carneiro 2020] in order to deal with some of the concerns with metamath discussed previously, to take advantage of the modern tool chain surrounding Metamath Zero, while also exploring different possible foundations for our ecosystem.

Our trust base consists of the matching logic proof system (255 lines; 84 axioms), the theory of definedness (17 lines; 1 axiom), and the theory of words instantiated with $A = \{a, b\}$, Γ_{Word} (13 lines; 12 axioms) as well as the `mm0-c` proof checker. Each of the these theories are defined in the `.mm0` files at [Redacted 2023].

Additionally, we also prove by hand 354 general matching logic theorems and 163 specific to the theory of words and also employ propositional lemmas taken from `examples/peano.mm` from the Metamath Zero examples directory. Note that although these proofs may be quite complex, they are not part of the trust base, since these theorems follow from the matching logic formalization and are checked by the Metamath Zero proof checker. These proofs and theorems are written in a higher level language, `mm1`, that is then compiled down to the `mmb` proof format using the `mm0-rs` tool, which is then verified using the `mm0-c` verifier. Again, note that `mm0-c` is part of our trust-base, while `mm0-rs` is not. While `mm0-c` is around 2800 lines of C code, this is smaller than the trust bases of Coq, Lean, and the official (vanilla) metamath distribution. There is also an on-going effort to formally verify this program.

6.2 Generating proof hints and proofs

To generate proof hints, we define regular expressions as maude module. First, we fully evaluate the expression with simplification rules, corresponding to Lemma 5.7, and associativity, commutativity and idemopotency of choice. For example, the rule

$$\texttt{rl [regex\_eq\_der\_concat]:}\ \ \delta_A(\alpha \cdot \beta) \Rightarrow \delta_A(\alpha) \cdot \beta \vee (\epsilon \wedge \alpha) \cdot \delta_A(\beta).$$

corresponds to the Metamath Zero theorem:

theorem `regex_eq_der_concat` $(\alpha\ \beta : \text{Pattern}) : \$\delta_a(\alpha \cdot \beta) \leftrightarrow \delta_a(\alpha) \cdot \beta + (\epsilon \wedge \alpha) \cdot \delta_a(\beta))\$ = \ldots;$

If the resultant expression's language does not contain epsilon (i.e. $\alpha \mid_\epsilon = \emptyset$), then the expression is not valid, and the algorithm halts, returning failure. Otherwise, a set of non-deterministic rules apply, taking the derivative with respect to each letter in the alphabet. We use the function `metaXApply` from Maude's `META-LEVEL` to apply these rules, allowing us to capture the context in which the rule applied as well as the substitution used, and build the proof hint.

6.3 Evaluation

We have evaluated our work against some hand crafted tests intended to exercise each of the simplification rules used, as well as standard benchmarks for evaluating decision procedures for regular expressions presented in [Nipkow and Traytel 2014]. Some statistics are shown in Table 8.

Each $\text{match}_{\{l,r\}}(n)$ test, by [Fischer et al. 2010], is an extended regular expression that asserts that a^n matches $(a + \epsilon) \cdot a^n$, that is, $a^n \rightarrow (a + \epsilon) \cdot a^n$. Here α^n indicates n-fold concatenation of

Benchmark	PH time	mm1 Size	mm1 time	mmb Size	mmb time	Nodes	Th$_1$	Th$_2$	Smpl.	cong	oh%
Base		205		307	79						
$(a+b)^*$	44	4	63	2	0	3	91	45	13	6	29
$a^{**} \longrightarrow a^*$	61	6	80	4	0	5	149	99	27	19	27
$(aa)^* \longrightarrow a^*a + \epsilon$	141	19	177	15	9	9	561	428	110	140	16
$(a^*b)^* + (b^*a)^*$	497	50	547	25	11	11	967	1692	385	650	17
even $+(a+b)\cdot$even	346	19	367	13	11	3	91	840	211	394	3
$\neg(\top \cdot a \cdot \top) + \neg(b^*)$	68	7	91	5	0	5	145	160	43	49	16
match$_l(1)$	100	15	127	7	2	15	433	287	76	67	24
match$_l(2)$	233	26	270	13	8	19	547	757	207	253	12
match$_l(4)$	1450	95	1566	45	27	27	775	4178	1061	1921	4
match$_l(8)$	26913	790	27835	266	488	43	1231	40006	8341	23087	1
match$_r(1)$	119	15	150	7	20	15	433	287	76	67	24
match$_r(2)$	270	26	308	13	21	19	547	757	207	253	12
match$_r(4)$	1374	83	1504	42	46	27	775	3547	920	1575	4
match$_r(8)$	20481	566	20833	228	263	43	1231	27934	6110	15498	1
eq$_l(1)$	124	11	157	7	17	9	263	272	73	72	20
eq$_l(2)$	315	26	349	15	20	13	377	914	229	370	10
eq$_l(4)$	3080	151	3109	70	70	21	605	7106	1504	3939	3
eq$_l(8)$	92433	1704	91165	446	1491	37	1061	85652	14652	56007	1
eq$_r(1)$	121	11	149	7	16	9	263	272	73	72	20
eq$_r(2)$	309	26	351	15	27	13	377	914	229	370	10
eq$_r(4)$	2067	120	2134	60	55	21	605	5419	1143	2974	3
eq$_r(8)$	31008	929	31497	330	770	37	1061	44675	7023	30288	1

Fig. 8. Statistics for certificate generation. Sizes are in KiB, and times are in milliseconds. From left to right, we show the time taken to generate the proof hint, the size of and time taken to generate the `.mm0` proof using Maude, the size of and time taken to generate the `.mmb` proof by `mm0-rs`, the number of nodes in the unfolding tree, number of theorem applications for phase 1, number of theorem applications for phase 2, number of simplifications applied, number of congruence of equivalence theorems applied, percentage of "overhead" theorems. Node that the size and time are reported relative to proving the initial manually written theorems and lemmas.

α, with the l version of the test using concatenation from the left, whereas the r version uses concatenation on the right. That is, α^4 may be either $(((\alpha \cdot \alpha)\alpha)\alpha)$ or $(\alpha(\alpha(\alpha \cdot \alpha)))$. Similarly, each eq$_{\{l,r\}}(n)$ test, by [Antimirov 2005], checks if a^* and $(a^0 + \cdots + a^n) \cdot (a^n)^*$ are equivalent. We also include a QuickCheck-like randomized test generation using the Hypothesis testing framework in python. We randomly generate an ERE α, and check that $\alpha \longrightarrow \alpha$. While any production procedure for deciding expressions would include an optimization that solves this in a single step, ours does not include such an optimization, allowing us to check that our construction is correct for a wide variety of expressions, rather than the small set of structurally similar ones used in the hand crafted tests or the other benchmarks.

Observations. First, let us discuss proof generation time. We observe that a majority of proof generation time is spent generating the proof hint in the instrumented algorithm. While there is

some overhead from proof generation, it is not so great as to be impractical, especially in cases where high-assuredness is necessary. While the runtime for proof hint generation itself is somewhat high, this largely because of of how we are employing the metalevel. We discuss this in more detail in the following section.

Next, let us look at proof size. While proof size appears to grow quite fast with respect to the number of nodes, it is roughly linear with respect to the number of simplifications applied and the depth of the term. In fact, most the proof rules applied are logically necessary—taking the derivatives, applying simplifications, lifting these into their contexts. The minimum reasonable number of this is one theorem application for each node, one for applying the simplification, plus the depth of each simplification. We use one additional lemma for applying each simplification, as well as a few theorems for overhead—applying substitutions, and proving positivity. This overhead depends on the depth of the expressions, and the size of the unfolding tree. For smaller proofs, their percentage may be quite large, but for larger ones makes up an insignificant part of the proof.

7 LIMITATIONS & FUTURE WORK

Our work thus far has mainly dealt with first-steps in understanding fixedpoint reasoning in matching logic and producing completely certified proofs, rather than building an optimized procedure for deciding expressions. There is work that needs to be done with respect to both optimization and handling more complex fixedpoint reasoning. Let us talk about some of these aspects.

7.1 Optimizations

Additional simplifications. Brzozowski's method on its own does not produce a minimal DFA. In fact, it is far from minimal. A common practice, when implementing Brzozowski derivatives, is to use additional simplifications besides those needed to ensure termination. In our case, we only use a 8 of these, for simplifying concatenation with epsilon and the emptyset, removing double negations and double Kleenes, and simlifying $\emptyset^*$, and ϵ^*. Using additional simplifications will reduce the size of the size of the DFA, producing smaller proofs.

Use of lemmas. Another low-hanging optimization is to break our proof into lemmas for each node. When transforming the DFA into an unfolding, several nodes of the DFA may be duplicated. By breaking our proof into lemmas, one for each node, since each duplicated node can use the same lemma it reduce some of the redundancy introduced by the unfolding tree.

Optimizing contexts. As can be seen in our benchmarks, as the size of the proof increases, a large majority of the proof becomes the application of congruence of equivalence theorems. This is because as the depth of the expressions increases, each simplification requires more and more theorem applications to lift the simplification into the context. We alread reduce this by combining these lifting operations between simplifiations whose contexts share a prefix. A further optimization would be to use Maude's evaluation strategies to order simplifications so that the above optimization can be applied more often.

Using Antimirov derivatives and NFAs. Another optimization is to use Antimirov's *partial derivatives* instead of Brzozowski's. In the case of partial derivatives, each regular expression has a set of derivatives for each letter rather than a single one. So, this procedure thus yields an NFA rather than a DFA. However, since each partial derivative may be smaller and simpler than the total one, the overall automaton size is smaller.

Conveniently, since the embeddings of set union and regular expression choice coincide, the matching logic theorems for regarding partial derivatives follow from small extensions to those for

Brzozowski's. While our technique for representing a DFA easily extends to representing an NFA, showing that the representation is total will likely be a more involved proof. For example, we may need to convert it to a DFA to show totality. Still, it is the size of the second part of the proof that we are concerned about, so this optimization may still be a win.

Another consideration is the application of the idemopotency simplifications. To apply the idemopotency theorem, $\Gamma_{\text{Word}} \vdash \varphi + \varphi \leftrightarrow \varphi$, the patterns must occur in the same AST node. To allow this, we sort the patterns in lexicographical order to enable the application of this theorem. for the case when they are in neighbouring AST nodes. These commutativity and associativity rules add to the overall size of the certificate, and are applied even when idemopotency isn't needed. Skipping the application of idemopentency when not needed would simplify this procedure.

7.2 Generality

The use of derivatives is not restricted to just regular expressions and languages. They may also be used in checking membership of context free grammars [Might et al. 2011]. This would require representing pushdown automata in matching logic, and would be more involved that DFAs.

In a similar vein, we are also interested in formalizing variant-based satisfiability [Meseguer 2018] to matching logic. Similar to how derivatives decomposes an expression's language by its initial letter, this technique decomposes a *symbolic term* into its most general unifiers modulo equations. If these equations have the *finite variant property*, then this becomes a decision procedure when combined with *folding variant narrowing*.

Our motivation for studying shallow embeddings of regular expressions, was to learn techniques for fixedpoint reasoning. However, modal μ-calculus is a much richer arena for this. Besides allowing greatest fixedpoint, needed to express properties over infinite words, it also allows *fixedpoint alternation*, alternating greatest- and least-fixedpoints, allow us to expresses properties like "infinitely often", and leading to an infinite heirarchy of fixedpoint formula with increasing expressivity [Arnold 1999]. If we can capture a decision procedure for modal-μ calculus in matching logic, this would greatly increase the complexity fixedpoint reasoning we may capture in matching logic. In [Niwiński and Walukiewicz 1996], Niwiński shows a decision procedure for modal μ calculus. This procedure produces a witness, called a *refutation*, when a formula is unsatisfiable. This is an infinite tree data-structure built using tableaux rules. Each rule either simplifies the formula or reduces it via a step similar to taking the derivative. Later results [Wilke 2001] show that every unsatisfiable formula has a finitely representable refutation, via "memoryless strategy". Extending our technique for representing DFAs to enable representing refutations would allow us to capture these proofs in matching logic.

8 CONCLUSION

In this work, we formalize the theory of words in matching logic. We show that despite only using a shallow embedding, we can prove several foundational results regarding extended regular expressions and derivatives. Further, our theory is complete—we are able to certify a decision procedure for extended regular expressions.

REFERENCES

Valentin Antimirov. 2005. 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, 455–466.

Konstantine Arkoudas and Selmer Bringsjord. 2007. Computers, justification, and mathematical knowledge. *Minds and Machines* 17 (2007), 185–202.

André Arnold. 1999. The μ-calculus alternation-depth hierarchy is strict on binary trees. *RAIRO-Theoretical Informatics and Applications* 33, 4-5 (1999), 329–339.

Martin Berglund and Brink van der Merwe. 2017. On the semantics of regular expression parsing in the wild. *Theoretical Computer Science* 679 (2017), 69–82.

Richard J Boulton, Andrew D Gordon, Michael JC Gordon, John Harrison, John Herbert, and John Van Tassel. 1992. Experience with embedding hardware description languages in HOL.. In *TPCD*, Vol. 10. 129–156.

James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. 2011. Automated Cyclic Entailment Proofs in Separation Logic.. In *CADE*, Vol. 11. Springer, 131–146.

James Brotherston, Carsten Fuhs, Juan A. Navarro Pérez, and Nikos Gorogiannis. 2014. A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates. 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)* (Vienna, Austria) *(CSL-LICS '14)*. ACM, New York, NY, USA, Article 25, 10 pages. https://doi.org/10.1145/2603088.2603091

Janusz A Brzozowski. 1964. Derivatives of regular expressions. *Journal of the ACM (JACM)* 11, 4 (1964), 481–494.

Mario Carneiro. 2020. 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, 71–88.

Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. 2021. Towards a trustworthy semantics-based language framework via proof generation. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*. Springer, 477–499.

Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. 2020a. Initial algebra semantics in matching logic. (2020).

Xiaohong Chen and Grigore Roşu. 2019a. *Applicative Matching Logic: Semantics of K*. Technical Report http://hdl.handle.net/2142/104616. University of Illinois at Urbana-Champaign.

Xiaohong Chen and Grigore Roşu. 2019b. Matching μ-logic. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*. ACM, Vancouver, Canada, 1–13.

Xiaohong Chen and Grigore Roşu. 2019c. *Matching μ-logic*. Technical Report http://hdl.handle.net/2142/102281. University of Illinois at Urbana-Champaign.

Xiaohong Chen, Minh-Thai Trinh, Nishant Rodrigues, Lucas Peña, and Grigore Roşu. 2020b. Towards a unified proof framework for automated fixpoint reasoning using matching logic. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

Xiaohong Chen, Minh-Thai Trinh, Nishant Rodrigues, Lucas Peña, and Grigore Roşu. 2020c. Towards a unified proof framework for automated fixpoint reasoning using matching logic. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

Thierry Coquand and Vincent Siles. 2011. 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, 119–134.

Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, 74–91. https://doi.org/10.1145/2983990.2984027

Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S Adve, and Grigore Roşu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1133–1148.

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

Kurt Gödel. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für mathematik und physik* 38 (1931), 173–198.

Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the undefinedness of C. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 336–345. https://doi.org/10.1145/2813885.2737979

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schille. 1996. Regular expressions for language engineering. *Natural Language Engineering* 2, 4 (1996), 305–328.

Bakhadyr Khoussainov and Anil Nerode. 2012. *Automata theory and its applications*. Vol. 21. Springer Science & Business Media.

Dexter Kozen. 1983. Results on the propositional μ-calculus. *Theoretical computer science* 27, 3 (1983), 333–354.

Alexander Krauss and Tobias Nipkow. 2012. Proof pearl: Regular expression equivalence and relation algebra. *Journal of Automated Reasoning* 49 (2012), 95–106.

Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Roşu. 2023. Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 77 (apr 2023), 29 pages.

https://doi.org/10.1145/3586029

Maude Team. [n.d.]. *The Maude System.* maude.cs.illinois.edu/

Norman Megill and David A Wheeler. 2019. *Metamath: a computer language for mathematical proofs.* Lulu. com.

José Meseguer. 2018. Variant-based satisfiability in initial algebras. *Science of Computer Programming* 154 (2018), 3–41.

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. *Acm sigplan notices* 46, 9 (2011), 189–195.

Nelma Moreira, David Pereira, and Simão Melo de Sousa. 2012. Deciding regular expressions (in-) equivalence in Coq. In *Relational and Algebraic Methods in Computer Science: 13th International Conference, RAMiCS 2012, Cambridge, UK, September 17-20, 2012. Proceedings.* Springer, 98–113.

Tobias Nipkow and Dmitriy Traytel. 2014. 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, 450–466.

Damian Niwiński and Igor Walukiewicz. 1996. Games for the μ-calculus. *Theoretical Computer Science* 163, 1 (1996), 99–116. https://doi.org/10.1016/0304-3975(95)00136-0

David Park. 1969. Fixpoint induction and proofs of program properties. *Machine intelligence* 5 (1969).

Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 346–356.

Redacted. 2023. *OOPSLA 2023 submission.* https://github.com/oopsla-23-submission-393/oopsla-23-submission-393

Grigore Roşu. 2017. Matching logic. *Logical Methods in Computer Science* 13, 4 (Dec. 2017), 1–61. https://doi.org/10.23638/lmcs-13(4:28)2017

Thoralf Skolem. 1879. Logico-combinatorial investigations in the satisfiability or provability of mathematical propositions: a simplified proof of a theorem by L. Löwenheim and generalizations of the theorem. *From Frege to Gödel. A Source Book in Mathematical Logic* 1931 (1879), 252–263.

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. (1955).

Alfred Tarski et al. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics* 5, 2 (1955), 285–309.

Thomas Wilke. 2001. Alternating tree automata, parity games, and modal mu-calculus. *Bulletin of the Belgian Mathematical Society-Simon Stevin* 8, 2 (2001), 359–391.

A PROOF OF LEMMA 5.2 SOUNDNESS: IF $\Gamma_{\text{Word}} \vdash \alpha$, THEN $A^* = \mathcal{L}(\alpha)$.

Proof. To prove soundness, we show that for any finite alphabet A, the set of words over A, that is, A^* correspond to a matching logic model of Γ_{Word}. We have signature $\Sigma = A \cup \{\epsilon, \cdot, \text{def}\}$. Let $\mathfrak{def}$ and $\mathfrak{concat}$ be distinguished elements. For every alphabet A, we define a matching logic model

$$M_A = (A \cup \{\mathfrak{def}, \mathfrak{concat}\}, \_ \bullet \_, \{\sigma_M \mid \text{for } \sigma \in \Sigma\})$$

where,

(1) $a_M = \{a\}$ for each $a \in A$,
(2) $\text{def}_M = \{\mathfrak{def}\}$,
(3) $\text{concat}_M = \{\mathfrak{concat}\}$
(4) (a) For any element e, $\mathfrak{def} \bullet e = M$
    (b) For any word w, $\mathfrak{concat} \bullet w = \{w\}$
    (c) For any pair of words w, v, $w \bullet v = \{wv\}$
    (d) For any other elements e_1, e_2, $e_1 \bullet e_2 = \emptyset$

Briefly, the def and concatenation symbols are interpreted as distinguished elements $\mathfrak{def}$, $\mathfrak{concat}$. The $\mathfrak{concat}$ element applied to any word results in the word itself, and the application of a word to another word is their concatenation.

The validity of the (FUNCTIONAL$_a$) and (FUNCTIONAL$_\epsilon$) axioms follows from the interpretations of the symbols ϵ and each a—they are interpreted as the singleton sets, while (FUNCTIONAL.) follows holds since the concatenation of every pair of words is a unique word. It is also clear that (ASSOCIATIVITY), (ASSOCIATIVITY) and (IDENTITY) hold by those properties of concatentation. The validity of the various (NO-CONFUSION) axioms follow from the freeness of words. □

B PROOF OF THEOREM 5.4 (LIFTING VARIOUS AXIOMS TO PATTERNS)

The axioms we defined above are quantified over element variables of the sort Word. This means that we can only instantiate them with other element variable or things we can prove are evaluated to singleton sets (aka functional patterns). To make them more generally applicable we show that they imply formulations that use free set variables. These can then be instantiated to any pattern using the (SET VARIABLE SUBSTITUTION) matching logic proof rule. We start by proving a general lemma that allows us to lift any element-variable quantified equalites to set variable ones .

Lemma B.1. *For any symbol contexts C and D, $\Gamma_{\text{Word}} \vdash C[x] = D[x]$ iff $\Gamma_{\text{Word}} \vdash C[X] = D[X]$*

Proof. The backward direction is easily proved using the (SET VARIABLE SUBSTITUTION) proof rule. By properties of equality, we only need to show $\Gamma_{\text{Word}} \vdash C[X] \longrightarrow D[X]$ and vice verca. We

only show one direction. The other follows by symmetery.

$$
\begin{aligned}
& C[x] = D[x] \\
\Rightarrow\ & C[x] \leftrightarrow D[x] \\
\Rightarrow\ & C[x] \rightarrow D[x] \\
\Rightarrow\ & z \in (C[x] \rightarrow D[x]) && \text{by (MEMBERSHIP INTRODUCTION)} \\
\Rightarrow\ & z \in C[x] \rightarrow z \in D[x] && \text{by (MEMBERSHIP}_{\neg}\text{), (MEMBERSHIP}_{\wedge}\text{)} \\
\Rightarrow\ & x \in X \wedge z \in C[x] \rightarrow x \in X \wedge z \in D[x] \\
\Rightarrow\ & \exists w.\, x \in X \wedge z \in C[x] \rightarrow w \in X \wedge z \in D[w] \\
\Rightarrow\ & \forall x.\, \exists w.\, x \in X \wedge z \in C[x] \rightarrow w \in X \wedge z \in D[w] \\
\Rightarrow\ & (\exists x.\, x \in X \wedge z \in C[x]) \rightarrow (\exists w.\, w \in X \wedge z \in D[w]) \\
\Rightarrow\ & z \in C[X] \rightarrow z \in D[X] && \text{by (MEMBERSHIP SYMBOL), with } \varphi = X \\
\Rightarrow\ & z \in C[X] \rightarrow D[X] && \text{by (MEMBERSHIP}\neg\text{), (MEMBERSHIP}\wedge\text{)} \\
\Rightarrow\ & C[X] \rightarrow D[X] && \text{by (MEMBERSHIP-ELIMINATION)}
\end{aligned}
$$

□

PROOF OF THEOREM 5.4. The first three may be proved by the application of lemma B.1 (NO-CONFUSION-AB-LEFT):

$$
\begin{aligned}
& \neg(a \cdot u \wedge b \cdot u) \\
\Leftarrow\ & a \cdot u \wedge b \cdot v \rightarrow \bot \\
\Leftarrow\ & (\exists x.\, a \cdot u = x) \rightarrow a \cdot u \wedge b \cdot u \rightarrow \bot \\
\Leftarrow\ & a \cdot u = x \rightarrow x \wedge b \cdot u \rightarrow \bot \\
\Leftarrow\ & (\exists y.\, b \cdot u = y) \rightarrow a \cdot u = x \rightarrow x \wedge b \cdot u \rightarrow \bot \\
\Leftarrow\ & b \cdot u = y \rightarrow a \cdot u = x \rightarrow x \wedge y \rightarrow \bot \\
\Leftarrow\ & b \cdot u = a \cdot u \rightarrow \bot
\end{aligned}
$$

(NO-CONFUSION-EPSILON-LEFT):

$$
\begin{aligned}
& \neg(\epsilon \wedge a \cdot u) \\
\Leftarrow\ & \epsilon \wedge a \cdot u \rightarrow \bot \\
\Leftarrow\ & (\exists y.\, \epsilon = y) \rightarrow \epsilon \wedge a \cdot u \rightarrow \bot \\
\Leftarrow\ & \epsilon = y \rightarrow \epsilon \wedge a \cdot u \rightarrow \bot \\
\Leftarrow\ & \epsilon = y \rightarrow y \wedge a \cdot u \rightarrow \bot \\
\Leftarrow\ & (\exists z.\, a \cdot u = z) \wedge \epsilon = y \rightarrow y \wedge a \cdot u \rightarrow \bot \\
\Leftarrow\ & a \cdot u = z \wedge \epsilon = y \rightarrow y \wedge z \rightarrow bot \\
\Leftarrow\ & a \cdot u = z \wedge \epsilon = y \rightarrow y = z \rightarrow \bot \\
\Leftarrow\ & \epsilon = a \cdot u \rightarrow \bot \\
\Leftarrow\ & \epsilon = a \wedge \epsilon = u \rightarrow \bot \\
\Leftarrow\ & \bot \wedge \epsilon = u \rightarrow \bot
\end{aligned}
$$

The proofs for (NO-CONFUSION-AB-RIGHT) and (NO-CONFUSION-EPSILON-RIGHT) is symmetric. □

C PROOF OF LEMMA 5.5 (INDUCTIVE DOMAIN)

We prove this by proving a more general lemma about the Kleene star operator:

LEMMA C.1. *For any pattern α, the following are equivalent in Γ_{Word}:*

(1) $\alpha^* \equiv (\mu X.\, \epsilon \vee X \cdot \alpha)$
(2) $\mu X.\, \epsilon \vee \alpha \cdot X$
(3) $\mu X.\, \epsilon \vee \alpha \vee X \cdot X$

Proof of Lemma C.1. We prove this as a set of circular implications, each implying the next.

(1) $\Longrightarrow$ (3):

$$(\mu X.\, \epsilon \vee X \cdot \alpha) \to \mu X.\, \epsilon \vee \alpha \cdot X$$
$$\Leftarrow \epsilon \vee (\mu X.\, \epsilon \vee \alpha \cdot X) \cdot \alpha \to \mu X.\, \epsilon \vee \alpha \cdot X$$

Epsilon case is easy.

$$(\mu X.\, \epsilon \vee \alpha \cdot X) \cdot \alpha \to \mu X.\, \epsilon \vee \alpha \cdot X$$
$$\Leftarrow (\mu X.\, \epsilon \vee \alpha \cdot X) \to \overbrace{\Box \cdot \alpha \multimap \mu X.\, \epsilon \vee \alpha \cdot X}^{\Psi}$$
$$\Leftarrow \epsilon \vee \alpha \cdot \Psi \to \Box \cdot \alpha \multimap \mu X.\, \epsilon \vee \alpha \cdot X$$

Epsilon case is easy.

$$\Leftarrow \alpha \cdot \Psi \to \Box \cdot \alpha \multimap \mu X.\, \epsilon \vee \alpha \cdot X$$
$$\Leftarrow \alpha \cdot \Psi \cdot \alpha \to \mu X.\, \epsilon \vee \alpha \cdot X$$
$$\Leftarrow \alpha \cdot (\mu X.\, \epsilon \vee \alpha \cdot X) \to \mu X.\, \epsilon \vee \alpha \cdot X \qquad \text{Property of contextual implication.}$$
$$\Leftarrow \alpha \cdot (\mu X.\, \epsilon \vee \alpha \cdot X) \to \epsilon \vee \alpha \cdot (\mu X.\, \epsilon \vee \alpha \cdot X)$$

Proved.

(2) $\Longrightarrow$ (3):

$$(\mu X.\, \epsilon \vee \alpha \cdot X) \to (\mu X.\, \epsilon \vee \alpha \vee X \cdot X)$$
$$\Leftarrow \epsilon \vee \alpha \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X) \to (\mu X.\, \epsilon \vee \alpha \vee X \cdot X)$$
$$\Leftarrow \epsilon \vee \alpha \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X) \to (\mu X.\, \epsilon \vee \alpha \vee X \cdot X)$$
$$\Leftarrow \epsilon \vee \alpha \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X) \to \epsilon \vee \alpha \vee (\mu X.\, \epsilon \vee \alpha \vee X \cdot X) \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X))$$

Epsilon case is easy.

$$\Leftarrow \alpha \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X) \to (\mu X.\, \epsilon \vee \alpha \vee X \cdot X) \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X)$$
$$\Leftarrow \alpha \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X) \to (\epsilon \vee \alpha \vee (\mu X.\, \epsilon \vee \alpha \vee X \cdot X) \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X)) \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X)$$
$$\Leftarrow \alpha \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X) \to \alpha \cdot (\mu X.\, \epsilon \vee \alpha \vee X \cdot X)$$

(3) $\Longrightarrow$ (1):

$$\mu X.\, \epsilon \vee \alpha \vee X \cdot X \to \alpha^*$$
$$\Leftarrow \epsilon \vee \alpha \vee (\alpha^*) \cdot (\alpha^*) \to \alpha^*$$

First two cases proved by unfolding.

$$\Leftarrow (\alpha^*) \cdot (\alpha^*) \to \alpha^*$$
$$\Leftarrow \alpha^* \to \overbrace{(\alpha^*) \cdot \Box \multimap \alpha^*}^{\Psi}$$
$$\Leftarrow \epsilon \vee \Psi \cdot \alpha \to (\alpha^*) \cdot \Box \multimap \alpha^*$$

Epsilon case:

$$\epsilon \to (\alpha^*) \cdot \Box \to \alpha^*$$
$$\Leftarrow (\alpha^*) \cdot \epsilon \to \alpha^*$$
$$\Leftarrow \alpha^* \to \alpha^*$$

$\Psi \cdot \alpha$ case:

$$
\begin{aligned}
& \Psi \cdot \alpha \to (\alpha^*) \cdot \Box \multimap \alpha^* \\
\Leftarrow\ & (\alpha^*) \cdot \Psi \cdot \alpha \to \alpha^* \\
\Leftarrow\ & \Gamma_{\text{Word}}, (\alpha^*) \cdot \Box \to \alpha^* \vdash (\alpha^*) \cdot \Box \cdot \alpha \to \alpha^* \\
\Leftarrow\ & \alpha^* \cdot \alpha \to \alpha^* \\
\Leftarrow\ & \alpha^* \cdot \alpha \to \epsilon \vee \alpha^* \alpha
\end{aligned}
$$

□

Let $\alpha = \top_{\text{Letter}}$. This gives use the following theorem:

D BRZOZOWSKI DERIVATIVES

D.1 Proof of Lemma 5.6

Forward direction.

$$
\begin{aligned}
& \beta \to (\epsilon \wedge \beta) \vee \bigvee a \cdot \delta_a(\beta) \\
\Leftarrow\ & \top_{\text{Word}} \wedge \beta \to (\epsilon \wedge \beta) \vee \bigvee a \cdot \delta_a(\beta) \\
\Leftarrow\ & (\epsilon \vee \bigvee_{a \in A} a \cdot \top_{\text{Word}}) \wedge \beta \to (\epsilon \wedge \beta) \vee \bigvee a \cdot \delta_a(\beta) \\
\Leftarrow\ & (\epsilon \wedge \beta) \vee \bigvee_{a \in A} (a \cdot \top_{\text{Word}} \wedge \beta) \to (\epsilon \wedge \beta) \vee \bigvee a \cdot \delta_a(\beta) \\
\Leftarrow\ & a \cdot \top_{\text{Word}} \wedge \beta \to a \cdot \delta_a(\beta) \\
\Leftarrow\ & x \in a \cdot \top_{\text{Word}} \wedge \beta \to x \in a \cdot \delta_a(\beta) \\
\Leftarrow\ & \exists z.\, x \in a \cdot z \wedge x \in \beta \wedge z \in \top_{\text{Word}} \to \exists y.\, x \in a \cdot y \wedge y \in \delta_a(\beta) \\
\Leftarrow\ & x \in a \cdot z \wedge x \in \beta \wedge z \in \top_{\text{Word}} \to x \in a \cdot z \wedge z \in \delta_a(\beta) \\
\Leftarrow\ & x \in a \cdot z \wedge x \in \beta \wedge z \in \top_{\text{Word}} \to z \in \delta_a(\beta) \\
\Leftarrow\ & a \cdot z \in \beta \to z \in \exists \Box.\, \Box \wedge \lfloor a \cdot \Box \to \beta \rfloor \\
\Leftarrow\ & a \cdot z \in \beta \to \lfloor a \cdot z \to \beta \rfloor
\end{aligned}
$$

□

Backward direction.

$$(\epsilon \wedge \beta) \vee \bigvee a \cdot \delta_a(\beta) \to \beta$$

$\epsilon \wedge \beta$ case is trivial.

$$
\begin{aligned}
& \Leftarrow a \cdot \delta_a(\beta) \to \beta \\
& \Leftarrow a \cdot (a \cdot \Box \multimap \beta) \to \beta \\
& \Leftarrow \beta \to \beta \qquad \text{Property of contextual implication.}
\end{aligned}
$$

□

D.2 Proof of Lemma 5.6

The (1), (2), and (3) hold since $\Gamma_{\text{Word}} \vdash \lfloor \emptyset \rfloor \to \bot$, $\Gamma_{\text{Word}} \vdash \lfloor a \cdot \Box \to \epsilon \rfloor \to \bot$, and $\Gamma_{\text{Word}} \vdash \lfloor a \cdot \Box \to b \rfloor \to \bot$. The goal (4) follows from the no-confusion and idemopotency axioms.

Note that for functional patterns t, we have $t \in \varphi = t \subseteq \varphi$. That is, $\lceil t \wedge \varphi \rceil = \lfloor t \to \varphi \rfloor$. This lets us proof the following:

For (5), we have the following provable equality:

$$\begin{aligned}
&\delta_a(\alpha_1 \vee \alpha_2) \\
&= \exists \Box.\, \Box \wedge \lfloor a \cdot \Box \to (\alpha_1 \vee \alpha_2) \rfloor \\
&= \exists \Box.\, \Box \wedge \lceil a \cdot \Box \wedge (\alpha_1 \vee \alpha_2) \rceil \\
&= \exists \Box.\, \Box \wedge \lceil (a \cdot \Box \wedge \alpha_1) \vee (a \cdot \Box \wedge \alpha_2) \rceil \\
&= \exists \Box.\, \Box \wedge \lceil (a \cdot \Box \wedge \alpha_1) \rceil \vee \lceil (a \cdot \Box \wedge \alpha_2) \rceil \\
&= \delta_a(\alpha_1) \vee \delta_a(\alpha_2)
\end{aligned}$$

For (6), we first reduce our goal to $\lfloor a \cdot x \to a \cdot \beta \rfloor \leftrightarrow \lfloor a \cdot x \to a \cdot (\epsilon \wedge \alpha) \cdot \delta_a(\beta) \rfloor \vee \lfloor a \cdot x \to a \cdot \delta_a(\alpha) \cdot \beta \rfloor$. This can be done using the membership axioms.

$$\begin{aligned}
&\lfloor a \cdot x \to \alpha \cdot \beta \rfloor \\
\leftrightarrow\ &(a \cdot x \in \alpha \cdot \beta) \\
\leftrightarrow\ &\left(a \cdot x \in ((\epsilon \wedge \alpha) \vee \bigvee_{b \in A} b \cdot \delta_b(\alpha)) \cdot \beta\right) \\
\leftrightarrow\ &(a \cdot x \in (\epsilon \wedge \alpha) \cdot \beta) \vee \bigvee_{b \in A} (a \cdot x \in b \cdot \delta_b(\alpha) \cdot \beta)
\end{aligned}$$

When $a \neq b$, we have $a \cdot x \in b \cdot \psi = \bot$

$$\begin{aligned}
\leftrightarrow\ &(a \cdot x \in (\epsilon \wedge \alpha) \cdot \beta) \vee (a \cdot x \in a \cdot \delta_a(\alpha) \cdot \beta) \\
\leftrightarrow\ &\left(a \cdot x \in (\epsilon \wedge \alpha) \cdot \left((\epsilon \wedge \beta) \vee \bigvee_{b \in A} b \cdot \delta_b(\beta)\right)\right) \vee (a \cdot x \in a \cdot \delta_a(\alpha) \cdot \beta) \\
\leftrightarrow\ &(a \cdot x \in (\epsilon \wedge \alpha \cdot \beta)) \vee (a \cdot x \in (\epsilon \wedge \alpha) \cdot \vee_{b \in a} b \cdot \delta_b(\beta)) \vee (a \cdot x \in a \cdot \delta_a(\alpha) \cdot \beta) \\
\leftrightarrow\ &\bot \vee \left(a \cdot x \in \bigvee_{b \in A} b \cdot (\epsilon \wedge \alpha) \cdot \delta_b(\beta)\right) \vee (a \cdot x \in a \cdot \delta_a(\alpha) \cdot \beta) \qquad \text{Since } \epsilon \wedge \alpha \text{ commutes within concat.} \\
\leftrightarrow\ &(a \cdot x \in a \cdot (\epsilon \wedge \alpha) \cdot \delta_a(\beta)) \vee (a \cdot x \in a \cdot \delta_a(\alpha) \cdot \beta) \\
\leftrightarrow\ &(a \cdot x \to a \cdot (\epsilon \wedge \alpha) \cdot \delta_a(\beta)) \vee (a \cdot x \to a \cdot \delta_a(\alpha) \cdot \beta)
\end{aligned}$$

For (7), we have the following provable equality:

$$\begin{aligned}
&\exists \Box.\, \Box \wedge \lfloor a \cdot \Box \to \neg \varphi \rfloor \\
&= \exists \Box.\, \Box \wedge a \cdot \Box \in \neg \varphi \\
&= \exists \Box.\, \Box \wedge \neg (a \cdot \Box \in \varphi)
\end{aligned}$$

Using the various (MEMBERSHIP) theorems, we may show that $\exists \Box.\, \Box \wedge \neg \psi = \neg \exists \Box.\, \Box \wedge \psi$, for a predicate ψ.

For (8), we have the following provable equality:

$$
\begin{aligned}
&\delta_a(\alpha^*)\\
&= \delta_a(\epsilon \vee (\neg\epsilon \wedge \alpha)\alpha^*)\\
&= \delta_a(\epsilon) \vee \delta_a(\neg\epsilon \wedge \alpha)\alpha^*)\\
&= \delta_a(\epsilon) \vee \delta_a(\neg\epsilon \wedge \alpha) \cdot \alpha^* \vee (\neg\epsilon \wedge \alpha \wedge \epsilon) \cdot \delta_a(\alpha^*)\\
&= \bot \vee \delta_a(\neg\epsilon \wedge \alpha) \cdot \alpha^* \vee \bot\\
&= \neg\delta_a(\epsilon) \wedge \delta_a(\alpha) \cdot \alpha^*\\
&= \top \wedge \delta_a(\alpha) \cdot \alpha^*\\
&= \delta_a(\alpha) \cdot \alpha^*
\end{aligned}
$$

E PROOF OF LEMMAS 5.11 AND 5.12

Lemma E.1. *Let n in be a node in the unfolding of tree of a* valid *regular expression. Then:*

(1) if n is a leaf node, $\Gamma_{\text{Word}} \vdash \mathfrak{fp}(n)[\Theta_n] \cdot \top_{\text{Letter}} \longrightarrow \mathfrak{fp}(n)[\mathrm{U}_n]$, and
(2) if n is an interior node,

$$\frac{\mathfrak{fp}(n_a)[\Theta_{n_a}] \cdot \top_{\text{Letter}} \longrightarrow \mathfrak{fp}(n_a)[\mathrm{U}_{n_a}] \textit{ for each } a \in A}{\mathfrak{fp}(n)[\Theta_n] \cdot \top_{\text{Letter}} \longrightarrow \mathfrak{fp}(n)[\mathrm{U}_n]}$$

where,

$$\Theta_n = \begin{cases} \lambda & \textit{when } n = \mathcal{R} \\ \Theta_p[\Psi_p/X_{L(p)}] & \textit{when } \mathfrak{fp}(p) \textit{ binds } X_{L(p)} \\ \Theta_p & \textit{otherwise.} \end{cases}$$

$$\Psi_p = \square \cdot \top_{\text{Letter}} \multimap \mathfrak{fp}(p)[\mathrm{U}_p]$$

$$\mathrm{U}_n = \begin{cases} \lambda & \textit{when } n = \mathcal{R} \\ \mathrm{U}_p[\mathfrak{fp}(p)/X_{L(p)}] & \textit{when } \mathfrak{fp}(p) \textit{ binds } X_{L(p)} \\ \mathrm{U}_p & \textit{otherwise.} \end{cases}$$

Proof. (1) Since n is a leaf node, we have $\mathfrak{fp}(n) = X(L(n)) = X(L(p))$, where p is n's parent. So, $\mathfrak{fp}(n)[\Theta_n] = \mathfrak{fp}(n)[\Psi_p/X(L(p))] = (\square \cdot \top_{\text{Letter}} \multimap \mathfrak{fp}(p))[\mathrm{U}_p] = \square \cdot \top_{\text{Letter}} \multimap \mathfrak{fp}(p)[\mathrm{U}_p]$. Our goal is thus:

$$(\square \cdot \top_{\text{Letter}} \multimap \mathfrak{fp}(p)[\mathrm{U}_p]) \cdot \top_{\text{Letter}} \longrightarrow \mathfrak{fp}(p)[\mathrm{U}_p]$$

and follows from the property $C[C \multimap \psi] \longrightarrow \psi$.

(2) Assume n has the form (2b), that is $fp(n) = \mu X(L(n)).\, \epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_a)$. Our goal becomes:

$$
\begin{aligned}
& (\mu X(L(n)).\, \epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_a))[\Theta_n] \cdot \top_{\text{Letter}} \longrightarrow \mathfrak{fp}(n)[\mathrm{U}_n] \\
\Leftarrow\ & (\mu X(L(n)).\, \epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_a))[\Theta_n] \longrightarrow \Box \cdot \top_{\text{Letter}} \multimap \mathfrak{fp}(n)[\mathrm{U}_n] \\
\Leftarrow\ & (\mu X(L(n)).\, \epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_a)[\Theta_n]) \longrightarrow \Box \cdot \top_{\text{Letter}} \multimap \mathfrak{fp}(n)[\mathrm{U}_n] \\
\Leftarrow\ & (\epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_a)[\Theta_{n_a}]) \longrightarrow \Box \cdot \top_{\text{Letter}} \multimap \mathfrak{fp}(n)[\mathrm{U}_n] \\
\Leftarrow\ & (\epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_a)[\Theta_{n_a}]) \cdot \top_{\text{Letter}} \longrightarrow \mathfrak{fp}(n)[\mathrm{U}_n] \qquad (\star) \\
\Leftarrow\ & \top_{\text{Letter}} \vee \bigvee_{a\in A} \left(a \cdot \mathfrak{fp}(n_a)[\Theta_{n_a}] \cdot \top_{\text{Letter}}\right) \longrightarrow \mathfrak{fp}(n)[\mathrm{U}_n]
\end{aligned}
$$

It is clear that $\top_{\text{Letter}} \longrightarrow \mathfrak{fp}(n)[\mathrm{U}_n]$, since n is accepting.

$$
\begin{aligned}
& a \cdot \mathfrak{fp}(n_a)[\Theta_{n_a}] \cdot \top_{\text{Letter}} \longrightarrow \mathfrak{fp}(n)[\mathrm{U}_n] \\
\Leftarrow\ & a \cdot \mathfrak{fp}(n_a)[\Theta_{n_a}] \cdot \top_{\text{Letter}} \longrightarrow \epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_{n_a})[\mathrm{U}_{n_a}] \qquad \text{By unfolding the RHS} \\
\Leftarrow\ & a \cdot \mathfrak{fp}(n_a)[\Theta_{n_a}] \cdot \top_{\text{Letter}} \longrightarrow a \cdot \mathfrak{fp}(n_{n_a})[\mathrm{U}_{n_a}]
\end{aligned}
$$

For the case, where n has form (2a) the proof is the same as above, but starting at ($\star$). □

Lemma E.2. *Let n in be a node in the unfolding of tree of a* valid *regular expression. Then:*

(1) if n is a leaf node, $\Gamma_{\text{Word}} \vdash \mathfrak{fp}(n)[\Lambda_n] \longrightarrow L(n)$, and
(2) if n is an interior node,

$$
\frac{\epsilon \longrightarrow L(n) \qquad \mathfrak{fp}(n_a)[\Lambda_{n_a}] \longrightarrow L(n_a) \text{ for each } a \in A}{\mathfrak{fp}(n)[\Lambda_n] \longrightarrow L(n)}
$$

$$
\text{where, } \Lambda_n = \begin{cases} \lambda & \text{when } n = \mathcal{R} \\ [L(p)/X_{L(p)}] & \text{when } \mathfrak{fp}_\beta(p) \text{ binds } X_{L(p)} \\ \Lambda_p & \text{otherwise.} \end{cases}
$$

Proof. Suppose n is a leaf node, then $\mathfrak{fp}(n) = X(L(n))$, and our goal becomes $L(n) \longrightarrow L(n)$. This is trivial.

Suppose n is an interior node of form (2b), then $\mathfrak{fp}(n) = \mu X.\, \epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_a)$, and our goal becomes $\mu X.\, \epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_a)[\Lambda_n] \longrightarrow L(n)$. We need to apply (KNASTER-TARSKI)

$$
\begin{aligned}
& \mu X.\, \epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_a)[\Lambda_n] \longrightarrow L(n) \\
\Leftarrow\ & \epsilon \vee \bigvee_{a\in A} a \cdot \mathfrak{fp}(n_a)[\Lambda_{n_a}] \longrightarrow L(n)
\end{aligned}
$$

The ϵ case is the first antecedent.

$$
\begin{aligned}
& a \cdot \mathfrak{fp}(n_a)[\Lambda_{n_a}] \rightarrow L(n) \\
\Leftarrow\ & a \cdot \mathfrak{fp}(n_a)[\Lambda_{n_a}] \rightarrow (\epsilon \wedge L(n)) \vee \bigvee_{b \in A} b \cdot \delta_b(L(n)) && \text{By Lemma 5.6} \\
\Leftarrow\ & a \cdot \mathfrak{fp}(n_a)[\Lambda_{n_a}] \rightarrow a \cdot \delta_a(L(n)) \\
\Leftarrow\ & \mathfrak{fp}(n_a)[\Lambda_{n_a}] \rightarrow \delta_a(L(n)) \\
\Leftarrow\ & \mathfrak{fp}(n_a)[\Lambda_{n_a}] \rightarrow L(n_a)
\end{aligned}
$$

□

F ARDEN'S RULE

Proof. (Least Solution) Let $\Psi_\mu \equiv \mu X.\, \alpha \vee \beta \cdot X$ and $\Psi_\nu \equiv \nu X.\, \alpha \vee \beta \cdot X$. We shall prove each direction separately.

$$
\begin{aligned}
& \beta^* \cdot \alpha \rightarrow \Psi_\mu \\
\Leftarrow\ & \beta^* \rightarrow \Box \cdot \alpha \multimap \Psi_\mu \\
\Leftarrow\ & (\mu X.\, \epsilon \vee \beta \cdot X) \rightarrow \Box \cdot \alpha \multimap \Psi_\mu \\
\Leftarrow\ & \epsilon \vee \beta \cdot (\Box \cdot \alpha \multimap \Psi_\mu) \rightarrow \Box \cdot \alpha \multimap \Psi_\mu
\end{aligned}
$$

For the epsilon case we must prove:

$$
\begin{aligned}
& \epsilon \rightarrow \Box \cdot \alpha \multimap \Psi_\mu \\
\Leftarrow\ & \epsilon \cdot \alpha \rightarrow \Psi_\mu \\
\Leftarrow\ & \alpha \rightarrow \Psi_\mu \\
\Leftarrow\ & \alpha \rightarrow \alpha \vee (\beta \cdot \Psi_\mu)
\end{aligned}
$$

We are left with the recursive case:

$$
\begin{aligned}
& \beta \cdot (\Box \cdot \alpha \multimap \Psi_\mu) \rightarrow \Box \cdot \alpha \multimap \Psi_\mu \\
\Leftarrow\ & \beta \cdot (\Box \cdot \alpha \multimap \Psi_\mu) \cdot \alpha \rightarrow \Psi_\mu \\
\Leftarrow\ & \beta \cdot \Psi_\mu \rightarrow \Psi_\mu \\
\Leftarrow\ & \beta \cdot \Psi_\mu \rightarrow \alpha \vee \beta \cdot \Psi_\mu
\end{aligned}
$$

Now for the other direction:

$$
\begin{aligned}
& \mu X.\, \alpha \vee X \cdot \beta \rightarrow \alpha \cdot \beta^* \\
\Leftarrow\ & \alpha \cdot \beta^* \rightarrow \mu X.\, \alpha \vee X \cdot \beta \\
\Leftarrow\ & \mu X.\, \alpha \vee X \cdot \beta \rightarrow \alpha \cdot \beta^* \\
\Leftarrow\ & \alpha \vee \alpha \cdot \beta^* \cdot \beta \rightarrow \alpha \cdot \beta^* \\
\Leftarrow\ & \alpha \cdot (\epsilon \vee \beta^* \cdot \beta) \rightarrow \alpha \cdot \beta^* \\
\Leftarrow\ & \epsilon \vee \beta^* \cdot \beta \rightarrow \beta^*
\end{aligned}
$$

□

Proof. (Greatest Solution). We have $\beta^* \cdot \alpha = \alpha \vee \beta \cdot \beta^* \cdot \alpha = \alpha \vee \beta \cdot \beta^* \cdot \alpha$, by unfolding β^*. This makes it easy to see that $(\nu X.\, \alpha \vee \beta \cdot X) \leftrightarrow \alpha \cdot \beta^*$ has a total DFA by Brzozowski's method. This is because the derivative of $\alpha \cdot \beta^*$ has the same structure as the greatest fixedpoint:

$$\delta_a(\alpha \vee \beta \cdot \beta^* \cdot \alpha) = \delta_a(\alpha) \vee \delta_a(\beta) \cdot (\beta^* \cdot \alpha)$$

Then, by Theorem **??**, we get (Greatest Solution). □

G SALOMAA'S COMPLETE AXIOMATIZATION

Salomaa gave a complete axiom system for RE:

Axioms:

$$(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma) \quad \text{(ASSOC-OR)}$$
$$(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma) \quad \text{(ASSOC-CONCAT)}$$
$$\alpha + \beta = \beta + \alpha \quad \text{(COMM-OR)}$$
$$(\alpha + \beta) \cdot \gamma = (\alpha \cdot \gamma) + (\beta \cdot \gamma) \quad \text{(DIST-1)}$$
$$\alpha \cdot (\beta + \gamma) = (\alpha \cdot \beta) + (\alpha \cdot \gamma) \quad \text{(DIST-2)}$$
$$\alpha + \alpha = \alpha \quad \text{(IDEM-OR)}$$
$$\epsilon \cdot \alpha = \alpha \quad \text{(IDENT-EPS)}$$
$$\emptyset \cdot \alpha = \emptyset \quad \text{(EMPTY-CONCAT)}$$
$$\alpha + \emptyset = \alpha \quad \text{(EMPTY-OR)}$$
$$\alpha^* = \epsilon + \alpha \cdot \alpha^* \quad \text{(UNFOLD)}$$
$$\alpha^* = (\epsilon + \alpha)^* \quad \text{(EPSILON-UNFOLD)}$$

Inference rules:

$$\frac{\alpha = \beta}{C[\alpha] = C[\beta]} \quad \text{(SUBST-1)}$$
$$\frac{\alpha = \beta \qquad C[\beta] = \gamma}{C[\alpha] = \gamma} \quad \text{(SUBST-2)}$$
$$\frac{\epsilon \notin \beta \qquad \alpha = \alpha \cdot \beta + \gamma}{\alpha = \gamma \cdot \beta^*} \quad \text{(SOLUTION)}$$

For each axiom, we will prove the corresponding statement in matching logic treating each α, β and γ as set variables, which can then be instantiated to any regular expression (or indeed, any pattern) using (SET VARIABLE SUBSTITUTION). We will prove the three inference rules as schemas.

Proof. Axioms COMM-OR ASSOC-OR, EMPTY-OR, IDEM-OR, EMPTY-CONCAT, DIST-1 and DIST-2 are general properties of matching logic symbols, disjunction and equality and can be proved using Lemma B.1. Axioms ASSOC-CONCAT and IDENT-EPS may be proved easily from the corresponding axioms in Γ_{Word} along with Lemma B.1. Inference rules (SUBST-1) and (SUBST-2) are general properties for matching logic patterns.

This leaves us with UNFOLD, EPSILON-UNFOLD and SOLUTION.

Axiom (UNFOLD). :

$$\epsilon + X \cdot X^* \to X^*$$
$$\Leftarrow \epsilon \vee X \cdot (\mu Y.\, \epsilon \vee Y \cdot X) \to (\mu Y.\, \epsilon \vee Y \cdot X)$$

Proved by fixedpoint.

$$X^* \to \epsilon + X \cdot X^*$$
$$\Leftarrow X^* \to \epsilon + X \cdot X^*$$
$$\Leftarrow \epsilon \vee (\epsilon + X \cdot X^*) \cdot X \to \epsilon + X \cdot X^*$$
$$\Leftarrow \epsilon \vee X \vee X \cdot X^* \cdot X \to \epsilon \vee X \cdot X^*$$
$$\Leftarrow \epsilon \vee X \vee X \cdot X^* \cdot X \to \epsilon \vee X \vee X \cdot X^* \cdot X$$

by Lemma **??**, (PRE-FIXEDPOINT)

Axiom (EPSILON-UNFOLD). :

$$
\begin{array}{ll}
(\epsilon + X)^* \to X^* & X^* \to (\epsilon + X)^* \\
\Leftarrow \epsilon + X^* \cdot (\epsilon + X) \to X^* & \Leftarrow \epsilon + (\epsilon + X)^* X \to (\epsilon + X)^* \\
\Leftarrow \epsilon + X^* + X^* \cdot X \to X^* & \Leftarrow (\epsilon + X)^* X \to (\epsilon + X)^* \\
\Leftarrow \epsilon + X^* \cdot X \to X^* & \Leftarrow (\epsilon + X)^* X \to (\epsilon + X)(\epsilon + X)^* \\
\Leftarrow \epsilon + X^* \cdot X \to \epsilon \vee X^* & \Leftarrow (\epsilon + X)^* X \to (\epsilon + X)^*
\end{array}
$$

Inference rule (SOLUTION). : For any regular expression β and patterns α and β, $\epsilon \notin \beta$ and $\alpha + \gamma \cdot \beta = \gamma$ implies $\alpha \cdot \beta^* = \gamma$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\alpha \vee Y \cdot \beta = Y}{\alpha \vee Y \cdot \beta \leftrightarrow Y}}{\alpha \vee Y \cdot \beta \to Y}}{\mu X.\, \alpha \vee X \cdot \beta \to Y} \qquad \dfrac{\dfrac{\dfrac{\alpha \vee Y \cdot \beta = Y}{\alpha \vee Y \cdot \beta \leftrightarrow Y}}{Y \to \alpha \vee Y \cdot \beta}}{Y \to \nu X \cdot \alpha \vee X \cdot \beta}\ \text{kt} \qquad \epsilon \notin \beta}{\dfrac{\alpha \cdot \beta^* \to Y \qquad\qquad Y \to \alpha \cdot \beta^*}{Y = \alpha \cdot \beta^*}}\ \text{Arden's Rule}
$$

□