System Description: A Theorem-Prover for Subregular Systems: The Language Toolkit and Its Interpreter, Plebby

Dakotah Lambert(✉)

Université Jean Monnet Saint-Étienne, CNRS, Institut d Optique Graduate School, Laboratoire Hubert Curien, UMR 5516, 42023 Saint-Étienne, France
dakotahlambert@acm.org

Abstract. We introduce here a domain-specific language, PLEB. The Piecewise-Local Expression Builder interpreter (plebby) is an interactive system for defining, manipulating, and classifying regular formal languages. The interactive theorem-proving environment provides a generalization of regular expressions with which one can intuitively construct languages via constraints. These constraints retain their semantics upon extension to larger alphabets. The system allows one to decide implications and equalities, either at the language level (with a specified alphabet) or at the logical level (across all possible alphabets). Additionally, one can decide membership in a number of predefined classes, or arbitrary algebraic varieties. With several views of a language, including multiple algebraic structures, the system provides ample opportunity to explore and understand properties of languages.

Keywords: Formal language theory · Subregular analysis · Semigroup classification · Interactive theorem proving · Mathematical library

1 Introduction

The study of formal languages is fundamental to the field of theoretical computer science. The regular languages in particular correspond to finite-state automata, which model stateful systems such as neural networks [21], text processing [43], robotics [33], and much more. So fundamental are these concepts that nearly any text on the theory of computation will include chapters on the regular languages including constructions of finite automata and operations under which they are closed, *cf.* [18,25,41]. Beyond theoretical computer science, finite-state methods form a basis for much of computational linguistics, *cf.* [7,15,19].

Initially developed as a study aid over the duration of an undergraduate course in the theory of computation, the Language Toolkit (LTK) is a Haskell library for working with constraint-based descriptions of languages. It is freely available under the MIT open-source license.[1] Of the many tools it ships with,

[1] At https://github.com/vvulpes0/Language-Toolkit-2/tree/develop one finds the latest unstable version of the software, and full stable releases can be found at https://hackage.haskell.org/package/language-toolkit.

J. Gibbons and D. Miller (Eds.): FLOPS 2024, LNCS 14659, pp. 311–328, 2024.
https://doi.org/10.1007/978-981-97-2300-3_16

we focus here on the domain-specific language it defines (PLEB) and its associated interpreter, plebby. This provides a practical and pedagogical system for manipulating regular languages and finite machines, essentially a Prolog for the regular languages. The primary features of the system are the ability to decide equivalences and implications, at the logical level or at the language level, and the ability to decide which subregular classes contain a given language.

The space of regular languages is rich. McNaughton and Papert [27] discuss several of these classes, and for each class they provide a description of what kind of information is relevant to its patterns. For instance, the languages locally testable in the strict sense (often called "strictly local") distinguish words by their substrings up to some fixed length k. Rogers and Lambert [36] discuss a broader collection of classes of formal languages, with a focus on those that correspond to quantifier-free first-order systems. The PLEB programming language is particularly optimized for expressing these quantifier-free formulae, but it is powerful enough to describe any regular language. The direct mapping between these logical languages and PLEB expressions allows students and researchers to better analyze and comprehend these patterns than with basic regular expressions alone. Because of this and other functionality, plebby has been used in teaching graduate courses in computational linguistics at Stony Brook University, as well as in projects such as the machine-learning benchmark, MLRegTest [31].

Like foma [20], Pyformlang [38], and OpenFST [1], the LTK provides mechanisms for defining regular languages via the equivalence between regular expressions and finite-state automata. The core Haskell library implements all of the operations that one would expect. It offers constructions for products, concatenations, complements, and reversals, among other things. It also provides mechanisms to determinize automata or to minimize them. The PLEB language allows one to incrementally define arbitrary regular languages by describing the interaction of constraints. The semantics of constraints are maintained through all manipulations, so they need only mention relevant symbols. The alphabet grows as new symbols are encountered. We offer some degree of compatibility with foma [20] and OpenFST [1] by means of the common AT&T-style textual format for interchange. Additionally we support visualizations via the AT&T GraphViz system.

Unlike these other systems, a distinguishing feature of the LTK is the inclusion of functions for algebraic analysis. As algebraic techniques provide a simple and uniform way to characterize classes of formal languages, they form the foundation for many of our classification algorithms. Caron [5] implements tests for some of the same classes in the LANGAGE package for Maple, but the algebraic lens provides much more power and flexibility. The Semigroups package for GAP [28] provides some tooling for this kind of classification (via semigroups), but offers no simple mechanism for constructing regular languages. And while both foma and OpenFST are excellent packages for constructing regular languages, they do not offer the same level of support for classification, for exploring logical implications between systems of constraints, nor for grammatical inference [17]. We provide

all of these things, although we will not discuss grammatical inference further in this work.

In short, the Language Toolkit is not yet another automata library. Automata are a core internal representation, but the primary features of the system are as follows. First, it allows for definition of languages with an expression format built upon logical formulae involving precedence, adjacency, and relativized adjacency. This logic-based formalism replaces the more traditional regular-expression syntax. One can then use the system to prove or disprove logical claims regarding those languages. Next, languages can be classified with respect to several subregular hierarchies, indicating which kinds of logic suffice to describe them and which computational mechanisms suffice to recognize them or learn them. With the algebraic techniques, classification is extended to user-defined classes with no modification to the code. Finally, there are grammatical inference and constraint-extraction tools. These are the features that have made the Language Toolkit so useful during the past decade of its development. Not only does it provide clean implementations of textbook algorithms on automata for pedagogical use, it provides this wealth of utility for analysis of regular languages that one would not find in any other system.

We begin in Sect. 2 by detailing our generalized regular expression format, PLEB expressions. These include containment of factors, all of the operations which define regular expressions, the other Boolean operations, infiltration and shuffle products, upward and downward closures with respect to subsequences, neutral letters, and Brzozowski derivatives. Only Unicode (UTF-8) input is supported, but every operation can be expressed in pure ASCII if this is desired. The complete set of operations is listed in both forms in Table 2 on page 302.

Next in Sect. 3, we detail how one might use the system to explore relationships between languages or between systems of constraints. We describe how the `:cequal` and `:cimplies` commands query logical equivalence and implication, respectively, between systems of constraints. The `:equal` and `:implies` commands operate instead at the language level, restricted to the current `universe` of symbols. This separation is possible due to our use of what we call automata with constraint semantics. In Sect. 4, we give a brief overview of the algebraic theory of formal languages and demonstrate how one might classify languages. This allows for some exploration of the relationships between language classes, as one may construct a separating language and verify that it does, indeed, separate the classes. Finally, we conclude with directions for future extension. Throughout this work, lines prefixed by `>` are code that can be run in plebby. We also include an appendix, demonstrating how one might use plebby to help answer some questions from various textbooks.

2 Generalized Regular Expressions

In this section, we note some useful operations under which the regular languages are closed. Using these, we introduce a generalized regular expression format that adds no computational power yet vastly simplifies language definition.

Kleene [21] introduced the regular expressions to describe the patterns represented by the artificial neural networks of McCulloch and Pitts [26]. Let $[\![R]\!]$ denote the meaning of the expression R, and suppose A and B are regular expressions. A regular expression over a finite alphabet Σ is defined inductively:

- $\emptyset$ is a regular expression where $[\![\emptyset]\!] = \emptyset$
- For each $\sigma \in \Sigma$, there is an expression σ where $[\![\sigma]\!] = \{\sigma\}$.
- $[\![(A|B)]\!] = [\![A]\!] \cup [\![B]\!]$ is a **union**.
- $[\![(AB)]\!] = \{ab : a \in [\![A]\!], b \in [\![B]\!]\}$ is a **concatenation**.
- $[\![A*]\!] = \varepsilon \cup [\![A]\!] \cup [\![AA]\!] \cup [\![AAA]\!] \cup \cdots$ is the **iteration closure** of A, where ε denotes the empty string. It is the fixed point of $A* = \varepsilon \cup AA*$.

As union and concatenation are associative, bracketing is often omitted. As a matter of convention, iteration binds more tightly than concatenation, which in turn binds more tightly than union. Given a finite set $S = \{s_1, s_2 \ldots, s_n\}$, we denote by S the regular expression $(s_1|s_2|\cdots|s_n)$. Thus $\Sigma*$ is the set of all, possibly empty, finite words over letters in Σ.

2.1 Factors and Symbol Sets

Following Rogers and Lambert [37], we take factors to be the fundamental unit of expressions. If $w = \sigma_1\sigma_2 \ldots \sigma_n$ for $\sigma_i \in \Sigma$, then the expression $\langle \sigma_1\ \sigma_2\ \ldots\ \sigma_n \rangle$, with whitespace between each symbol, represents the set of words which contain w as a substring. That is, it represents words of the form uwv where u and v are elements of $\Sigma*$. At any point between two symbols, a comma may be used to signify an arbitrary gap. Then $\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ is the set of words which contain w as a subsequence, words of the form $u_0\sigma_1u_1\sigma_2u_2 \ldots \sigma_nu_n$. Two modifiers anchor the factor to word boundaries: $\rtimes$ fixes the first component to the left edge while $\ltimes$ fixes the last to the right edge. These may be used together: $\rtimes\ltimes\langle \sigma_1\ \sigma_2\ \ldots\ \sigma_n \rangle$ represents the singleton set $\{w\}$. Empty sequences are allowed: $\langle\rangle$ denotes the set of all words containing the empty string as a substring. In other words, $\langle\rangle$ denotes $\Sigma*$.[2]

The individual components of a factor are not actually mere symbols, but sets of symbols. Suppose a and b are symbol sets; a symbol set is defined inductively:

- For any valid name *sym* it holds that $[\![/sym]\!] = \{sym\}$.
- Named variables are permitted. $[\![s]\!]$ is the set that had been assigned to the variable s, if such a set exists.
- $[\![\{a, b\}]\!] = [\![a]\!] \cup [\![b]\!]$.
- $[\![[a, b]]\!] = [\![a]\!] \cap [\![b]\!]$.

Assignment is expressed by $=$ *name value*. In order to save on typing slashes, it is good practice to begin a file or session with a header that declares the symbols to be used, such as:

[2] In ASCII, the word boundaries are `%|` (left) and `|%` (right), while angle-brackets are represented by less-than and greater-than signs. Other equivalences are given in Table 2 on page 302.

Table 1. Base cases and operations for regular expressions.

	Empty	Symbol	Union	Concatenate	Iterate
Reg	$\emptyset$	σ	$(e_1 \vert e_2)$	$(e_1 e_2)$	$e*$
Gen	$\neg\langle\rangle$	$\rtimes\ltimes\langle\sigma\rangle$	$\bigvee\{e_1, e_2\}$	$\bullet\{e_1, e_2\}$	$*e$

```
> =a{/a}=b{/b}=c{/c}
```

This is three assignments collapsed onto a single line, binding the symbols a, b, and c to the variables a, b, and c, respectively. Except in close proximity to such definitions, we shall continue to use the slash notation throughout, so that all examples behave properly in a fresh environment.

2.2 Booleans, Concatenation, and Iteration

Any introductory text on finite automata, such as that of Hopcroft and Ullman [18], will contain a proof that regular languages are all and only those expressible by such machines. Using this equivalence, one easily finds that the class of regular languages is closed not only under (finitary) union but also under (finitary) **intersection** and under **complement**. If $e_1, e_2, \ldots, e_n$ are PLEB-expressions, then $\bigvee\{e_1, e_2, \ldots, e_n\}$ denotes their union, $\bigwedge\{e_1, e_2, \ldots, e_n\}$ their intersection, and $\neg e_1$ the complement of e_1. For empty sequences, the neutral element of the operation is chosen. An empty union is the empty set, equivalent to $\neg\langle\rangle$, while an empty intersection is the universal language that accepts every word, equivalent to $\langle\rangle$. Union and intersection are variadic operations, taking a sequence of arguments. Complement is a monadic operator, taking just one.

Concatenation, denoted $\bullet(e_1, e_2, \ldots, e_n)$, is another variadic operator. There is also gapped concatenation, denoted $\bullet\bullet(e_1, e_2, \ldots, e_n)$ and equivalent to concatenation interspersed with arbitrary content: $\bullet(e_1, \langle\rangle, e_2, \langle\rangle, \ldots, \langle\rangle, e_n)$. The Kleene star operator signifying the iteration closure is yet another monadic operator, denoted $*e_1$. We provide $+e_1$ as syntactic sugar for $\bullet(e_1, *e_1)$. All operators are prefixes; the monadic operators attach directly to the expression upon which they act, while variadic operators attach to braces (or, equivalently, parentheses) embracing a comma-separated sequence of operands. At this point, we can represent any regular expression. A summary of equivalences is provided in Table 1. The union and iteration operators would work identically if concatenation were right-to-left rather than left-to right. So regular languages are closed under reversal as well. We offer a monadic operator, $\rightleftarrows$, for this task.

2.3 Subsequences and Shuffle Ideals

In Sect. 2.1 we noted that if $w = \sigma_1\sigma_2\ldots\sigma_n$ then $\langle\sigma_1, \sigma_2, \ldots, \sigma_n\rangle$ is the set of all words that contain w as a subsequence. This is a **shuffle ideal**. In general, given an arbitrary expression e, we can define $\uparrow e$ as the set of all words which contain any word in $[\![e]\!]$ as a subsequence. That this **upward closure** is regular is a

consequence of Higman's Lemma and the resulting language is in the one-half level of the Straubing hierarchy, *cf.* [30]. One constructs $\uparrow e$ by adding self-loops on each symbol to each state of the automaton represented by e. This operation is idempotent.

We will see more on the `:cequal` command in the next section. Essentially, it indicates whether two constraints are logically equivalent. With it, we can verify the equivalence between an upward closure and a subsequence-factor:

```
> :cequal ↑⋊⋉⟨/a /b /b /a⟩ ⟨/a,/b,/b,/a⟩
True
```

We may also close in the other direction: the **downward closure** of e, denoted $\downarrow e$, is the set of all words which are contained as a subsequence by some word in e. That is, $\downarrow e$ is the set of words obtained by beginning from some word in e and deleting zero or more instances of zero or more symbols. One constructs $\downarrow e$ by adding edges consuming no input in parallel with all edges of the automaton represented by e. This operation is idempotent. Languages closed under subsequence, that is, languages L such that $\downarrow L \equiv L$, have been studied by Haines [14] for their interesting mathematical properties as well as by Rogers *et al.* [35] for their linguistic relevance.

Upward closure is a specific case of the shuffle product. The shuffle product of two words is defined inductively as follows, where a and b are symbols in Σ and u and v are words in $\Sigma*$ [24].

$$u \sqcup\!\sqcup \varepsilon = u = \varepsilon \sqcup\!\sqcup u$$
$$au \sqcup\!\sqcup bv = a(u \sqcup\!\sqcup bv) \cup b(au \sqcup\!\sqcup v)$$

Given two languages A and B, their shuffle product is the set $A \sqcup\!\sqcup B = \{a \sqcup\!\sqcup b : a \in A, b \in B\}$. For a given expression e, it is the case that $\uparrow e$ is logically equivalent to $\sqcup\!\sqcup\{e, \langle\rangle\}$. The infiltration product, denoted $\Uparrow$, is defined similarly [8].

$$u \Uparrow \varepsilon = u = \varepsilon \Uparrow u$$
$$au \Uparrow bv = \begin{cases} a(u \Uparrow bv) \cup b(au \Uparrow v) \cup a(u \Uparrow v) & \text{if } (a = b \\ a(u \Uparrow bv) \cup b(au \Uparrow v) & \text{otherwise.} \end{cases}$$

We provide monadic $\uparrow$ and $\downarrow$ operators as well as variadic $\Uparrow$ and $\sqcup\!\sqcup$ operators. The variadic operators require some caution. As discussed in Sect. 3, subexpressions have their alphabets semantically extended when used in larger expressions. When computing shuffle products, it may be wise to fix the alphabet of each subexpression to a desired set T by intersecting with $*\rtimes\ltimes\langle\{T\}\rangle$.

2.4 Tiers and Neutral Letters

Subsequences provide a simple mechanism to describe long-distance dependencies, but they are not the only available mechanism. Another possibility, which has been useful in computational linguistics [16] and in robotic control [33] hinges on the notion of a **tier** of salient symbols. If symbols are not salient to the

constraint, then they are ignored entirely. Neither inserting them nor deleting them can influence whether a word is accepted [23]. In other words, symbols not salient are **neutral**. Using this notion of salience and neutrality, one can describe long-distance constraints as if they were local.

Given a symbol set T, we provide two monadic operators. The first, $[T]e$, restricts the alphabet of e to the symbols in T, then adds self-loops on each other symbol to each state. This yields the inverse tier-projection from T of e, the words which satisfy e on the T-tier. The other operator, $|T|e$, makes each element of T neutral in e. Edges which consume no input are added in parallel to each edge labeled by an element of T, and then self-loops labeled by each such element are added to each state. $|T|e$ is equivalent to $\sqcup\{\bigwedge\{e, *\rtimes\ltimes\langle T\rangle\}, \neg\langle T\rangle\}$. For convenience, if T is a union of multiple symbol sets then the outermost braces may be omitted.

For example, the constraint $[/a, /b]\neg\langle /a\ /b\rangle$ over projected substrings is logically equivalent to the constraint $\neg\langle /a, /b\rangle$ over subsequences. As we will see in the next section, one can verify this:

```
> :cequal [/a,/b]¬⟨/a /b⟩ ¬⟨/a,/b⟩
True
```

Both $[T]$ and $|T|$ are idempotent operations. Further, $\neg[T]e$ is equivalent to $[T]\neg e$ and $\neg|T|e$ is equivalent to $|T|\neg e$ [23].

2.5 Brzozowski Derivatives and Quotients

Given a language L and a prefix s, one might wish to know which strings t act as valid completions where $st \in L$. This **Brzozowski derivative** is sometimes denoted $s^{-1}L$, so named as Brzozowski used the operation in finding the derivatives of regular expressions [3]. A generalization of this is the **left-quotient** $A\backslash B$, the set of strings t that can be appended to a string in A to yield a string in B.

Similarly, the **right-quotient** B/A is the set of strings s that can be prepended to a string in A to yield a string in B. The expression B/A is clearly equivalent to $(A^R\backslash B^R)^R$, where x^R denotes the reversal of x.

Hopcroft and Ullman [18] provide a nonconstructive proof that if B is regular then B/A (and, of course, $A\backslash B$) is regular for any language A. For regular A, we may use a simple construction on automata. In order to compute $A\backslash B$, first compute the concatenation $C = A\Sigma*$. Then, compute the product $(A \times C) \times B$. The accepting states are those whose B- and C-components are both accepting, and the initial states are those whose A-components are accepting. This then begins computation at any state where A could end, and accepts only strings that would be valid continuations in B.

We provide variadic functions for both quotients. They are not associative operations, and so they are best used only dyadically: $[\![\backslash\backslash(A, B)]\!] = [\![A]\!]\backslash[\![B]\!]$ and $[\![//(B, A)]\!] = [\![B]\!]/[\![A]\!]$.

An example, suppose that B is the set of words that do not contain an ab substring and that A is a set of words such that every word in A ends on a.

Table 2. Monadic (left, $\oplus e$), and variadic (right, $\oplus\{e_1, e_2, \ldots, e_n\}$) operators.

Syntax	ASCII	Meaning
$\neg$	`!`	complement
$*$	`*`	iteration closure
$+$	`+`	iteration (nonempty)
$\rightleftarrows$	`-`	reversal
$\uparrow$	`^`	upward closure
$\downarrow$	`$`	downward closure
$[T]$	`[T]`	salience restriction
$\lvert T\rvert$	`|T|`	neutralizing

Syntax	ASCII	Empty	Meaning
$\bigvee$	`\/`	$\neg\langle\rangle$	union
$\bigwedge$	`/\`	$\langle\rangle$	intersection
$\bullet$	`@`	$\rtimes\ltimes\langle\rangle$	concatenation
$\bullet\bullet$	`@@`	$\rtimes\ltimes\langle\rangle$	gapped concatenation
$\Uparrow$	`.^.`	$\rtimes\ltimes\langle\rangle$	infiltration product
$\sqcup\!\sqcup$	`|_|_|`	$\rtimes\ltimes\langle\rangle$	shuffle product
$\backslash\backslash$	`\\`	$\rtimes\ltimes\langle\rangle$	left-quotient
$//$	`//`	$\rtimes\ltimes\langle\rangle$	right-quotient

Note that $A\backslash B$ is the set of all words that neither begin with b nor contain the ab substring. We shall see more on the `:cimplies` command in the next section, but for now we notice that in this quotient no word begins with b.

```
> :cimplies \\(⋉⟨/a⟩,¬⟨/a /b⟩) ¬⋊⟨/b⟩
True
```

We can also use these quotients to construct the prefix closure $//(e, \langle\rangle)$ or suffix closure $\backslash\backslash(\langle\rangle, e)$ of an expression e. Then the substring closure is $\backslash\backslash(\langle\rangle, //(e, \langle\rangle))$.

2.6 Summary

Table 2 lists the available operators, both in Unicode syntax and in ASCII syntax. They are listed in the order introduced in the text. Monadic operators are written directly before their operand. Variadic operators take zero or more comma-separated operands surrounded by either curly braces or parentheses and are written before the opening delimiter. Factors also have ASCII syntax: use less-than and greater-than signs in place of the angle-brackets, and replace the anchor symbols with `%|` (left) and `|%` (right).

Like with symbol sets, expressions may be assigned to variables using the syntax $=$ *name value*. A bare expression acts as an assignment to the special variable *it*. And finally, all assignments of both symbol sets and expressions update a special variable *universe*, a symbol set containing all symbols used so far in bound variables.

With the tools discussed so far, one can easily define regular languages using generalized regular expressions known as PLEB expressions. In the next section we discuss how to check for equalities or implications and how one might minimize constraint-based descriptions.

We close this section with a final example. Krebs *et al.* [22] describe a language U_2 that has been instrumental to their work on characterizing classes of languages definable with fragments of first-order logic restricted to two variables. In their work, U_2 is defined over the alphabet $\Sigma = \{a, b, c\}$ as follows.

$$U_2 = (\Sigma* - (\Sigma*ac*a\Sigma*)) \cup (\Sigma* - (\Sigma*bc*b\Sigma*))ac*a\Sigma*$$

Already this expression is extended to include (relative) complements. In this language, c is a neutral letter. After ignoring c, U_2 is a language in which either no aa substring occurs, or there is an aa substring not preceded at any distance by a bb substring. An equivalent PLEB expression is as follows.

```
> =a{/a}=b{/b}=c{/c}
> = U2 [a,b] ⋁ {¬⟨a a⟩,•(¬⟨b b⟩,⋊⟨a a⟩)}
```

3 Constraint Analysis

In the previous section, we briefly mentioned the `:cequal` and `:cimplies` commands. This section introduces the mechanism behind them and describes a few of the other commands available. We begin by distinguishing constraints from the languages they yield.

A (formal) language is merely a set of words. A constraint is a logical formula that might be satisfied by one or more words, or which might be unsatisfiable. For example $\langle /a\ /b \rangle$ expresses a constraint that the ab substring appears somewhere, and $\langle /a, /b \rangle$ expresses a constraint that the ab subsequence appears somewhere. These are not logically equivalent, but the first does logically imply the second. And if the alphabet is merely $\{a, b\}$, then the language they express is the same.

```
> = substr ⟨/a /b⟩
> = subseq ⟨/a,/b⟩
> :cequal substr subseq
False
> :cimplies substr subseq
True
> :cimplies subseq substr
False
> :equal substr subseq
True
> =c{/c}
> :equal substr subseq
False
```

This example demonstrates the above observations. The presence of an ab substring logically implies the presence of an ab subsequence, but the reverse does not hold. The `:cequal` and `:cimplies` commands operate at the constraint level, comparing logical semantics. However, the `:equal` and `:implies` commands operate at the language level, restricting the domain to the current `universe` of symbols. If the alphabet is exactly the set $\{a, b\}$, then the two expressions yield the same language, but if instead it were $\{a, b, c\}$ then they would not.

This works because factors are constructed in such a way that their semantics are preserved. Expressions are compiled to finite-state automata using not only the symbols they mention, but also a special symbol, ⍰, which represents all others. This acts as the `@` of [19]. When combining expressions, their alphabets

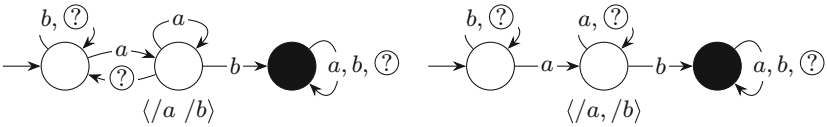

Fig. 1. Automata with constraint semantics.

must be **semantically extended** by inserting edges on any new symbols in parallel with these ?-edges. We designate these as automata with **constraint semantics**. Figure 1 depicts our example substring and subsequence constraints. When displaying automata with the `:display` command, they are first **desemantified** by stripping the wildcard symbols and normalizing the result.

One might use these tools to construct expressions that recreate a given pattern. Given a language, be that one constructed from a PLEB expression, one read from an OpenFST-compatible automaton file using `:readATT`, or one imported using the grammatical inference commands (not described in this work), one can hypothesize constraints and ask if the language `:implies` those constraints. Keeping any that are successfully implied, eventually one reaches a point where the cooccurrence (intersection) of the proposed constraints is `:equal` to the language. The set may be large. Removing one constraint at a time, one may ask if the cooccurrence of the remaining constraints `:implies` the removed constraint. If this implication holds, then the constraint is redundant and need not be included. At times, it may be useful to `:display` the difference between systems of constraints, in order to see what is accepted that should not be or vice versa. Finally one is left with a minimal set of constraints that describes the language. There may well be other such sets.

This has been a sampling of ways in which plebby can help explore formal languages through verifying accurate factorization and minimizing systems of constraints. The next section describes classification techniques. Different classes of languages correspond to different kinds of constraints, so the techniques presented ahead may also be useful for such analysis.

4 Algebra and Complexity Analysis

Chomsky's [9] hierarchy includes no classes more restrictive than the regular languages. However, there are several well-motivated subclasses of this class. Every language is associated with a semigroup called its **syntactic semigroup**, and the regular languages are all and only those whose syntactic semigroups are finite [32]. We offer commands to display the algebraic structure of a language in various ways. One can view a Cayley graph using `:synmon`, or an egg-box diagram in the sense of [10] using `:eggbox`. Additionally, a Hasse diagram of the syntactic order in the sense of Pin [29] can be displayed with `:synord`.

Eilenberg's theorem established a formal correspondence between classes of regular languages and classes defined by collections of equations, called pseudovarieties, of finite semigroups [12,13]. A **pseudovariety**, henceforth simply called

a **variety**, is a class of semigroups closed under division and finitary products, where a semigroup S is said to divide another semigroup T if S is a quotient of a subsemigroup of T. Varieties of (finite) semigroups are called $+$-varieties, while varieties of (finite) monoids are $*$-varieties. We provide three commands: `:isVarietyM` for $*$-varieties, `:isVarietyS` for $+$-varieties, and `:isVarietyT` for what would be $+$-varieties after removing any neutral letters.

These commands take two arguments. The first is a description of the variety, and the second is the expression to test. A variety is a semicolon-separated collection of universally-quantified weak inequalities, all wrapped in square brackets. Inequalities are over the syntactic order of Pin [29]. All variables in these relations are a single letter, and concatenation (multiplication in the semigroup) is denoted by adjacency. Reiterman describes another operator, denoted $\pi(x)$, which maps x to the unique idempotent element in the subsemigroup generated by x [34]. Since then, this has more typically been denoted x^ω, *cf.* [29], although in plebby we denote it $x*$ for ease of entry. This operator allows varieties to be defined by a single conjunction of equations rather than being ultimately defined by a series of such conjunctions [34].

For example we might ask whether a language has a syntactic semigroup which is both commutative ($ab = ba$) and idempotent ($xx = x$). For concreteness, we will perform this test against two languages: the language which contains an ab substring, and the language which contains both a and b. In both cases, the alphabet shall be $\Sigma = \{a, b, c\}$.

```
> =a{/a}=b{/b}=c{/c}
> :isVarietyS [ab=ba;xx=x] ⟨a b⟩
False
> :isVarietyS [ab=ba;xx=x] ⋀{⟨a⟩,⟨b⟩}
True
```

This particular class is well-studied and so there is a shortcut, `:isCB` (for "commutative band"), which performs the same operation. In many cases, the shortcut commands employ faster algorithms than the general variety check.

Because a language and its complement share the same syntactic semigroup, a class not closed under complement cannot be a variety. Pin uses the concept of a syntactic order to capture some such classes as varieties of ordered semigroups [29]. It is for this reason that our variety-testing commands also allow the use of the weak inequalities, $\leq$ and $\geq$, under the syntactic order. As all variables are universally-quantified, strict inequalities are meaningless. Thus $<$ and $>$ are synonyms for $\leq$ and $\geq$, respectively.

4.1 Some Varieties with Shortcuts

As one might imagine, there are boundless varieties of interest. We provide shortcut commands for many of them. In this section we list a few of them by name alongside their equivalent commands and their language-theoretic characterizations. There are several others; for a full list, see the `:help` in the interpreter.

Locally Testable. See [27] or [4]. A language is locally testable iff there is some integer k such that the set of substrings of length k of a word is sufficient information to determine whether the word is accepted. The commands for deciding this class are `:isLT` and `:isVarietyS [a*xa*ya*=a*ya*xa*;(a*xa*)*=a*xa*]`.

Tier-Based Locally Testable. See [23]. A language is tier-based locally testable iff after removing its neutral letters it is locally testable. This class is decided by `:isTLT` and `:isVarietyT [a*xa*ya*=a*ya*xa*;(a*xa*)*=a*xa*]`. In either case, the set, T, of nonneutral letters is reported.

Piecewise Testable. See [40]. The subsequence analogue of locally testable, a language is piecewise testable iff there is some integer k such that the set of subsequences of length k of a word is sufficient information to determine whether the word is accepted. The commands for deciding this class are `:isPT` and `:isVarietyM [y(xy)*=(xy)*=(xy)*x]`.

Strictly Piecewise. See [14] or [35]. A restriction of the piecewise testable languages, the strictly piecewise languages are those definable by a finite set of forbidden subsequences. This class is decided by the `:isSP` or `:isVarietyM [1≤x]` commands. The size, k, of forbidden subsequences is reported when using `:isSP`. Equivalently, given an expression `e`, one can decide whether `e` is strictly piecewise using `:equal e ↓e`. The complements of strictly piecewise languages correspond precisely to the one-half level of the Straubing hierarchy [30].

Locally Threshold Testable. See [27] or [2]. A language is locally threshold testable iff it is definable by Boolean combinations of constraints that a particular substring occurs at least some fixed finite number n of times. These are the languages first-order definable with successor but without general precedence [42]. The commands for deciding this class are `:isLTT` and `:isVarietyS [e*af*be*cf*=e*cf*be*af*;xx*=x*]`.

Tier-Based Locally Threshold Testable. See [23]. A language is tier-based locally testable iff after removing its neutral letters it is locally threshold testable. The commands for deciding this class are `:isTLTT` or `:isVarietyT [e*af*be*cf*=e*cf*be*af*;xx*=x*]`. In either case, the set, T, of nonneutral letters is reported.

Star-Free. See [39]. A language is star-free if and only if it is definable by a regular expression generalized to allow intersection and complement but restricted by disallowing the use of the iteration operator. These are the languages first-order definable with general precedence [27]. The commands for deciding this class are `:isSF` and `:isVarietyM [xx*=x*]`. Notice that this equation is one of the equations for locally threshold testable. This is in general an easy way to

construct sub- and supervarieties of a variety: simply add or remove equations, add or remove constraints.

4.2 Some Other Well-Studied Classes

While a number of language classes correspond precisely to varieties of semigroups or monoids, this is not always the case. As noted in the previous section, inequalities allow for the capture of classes that are not closed under complement. However, even these ordered varieties cannot capture classes not closed under both union and intersection. This section discusses two such classes which see wide application.

Strictly Local. See [27]. A restriction of the locally testable languages, analogous to that which derives strictly piecewise from piecewise testable, a language is strictly local if and only if it is definable by a finite set of forbidden substrings. These are decided by `:isSL` using an algorithm implied by the work of Caron [5] and by Edlefsen *et al.* [11]. The size, k, of forbidden substrings is reported.

Tier-Based Strictly Local. See [16] or [23]. A language is tier-based strictly local if and only if after removing its neutral letters it is strictly local. This class is decided by `:isTSL`. The size, k, of forbidden substrings is reported, as is the set T of nonneutral letters.

4.3 Summary

This section has discussed a sampler of the classification algorithms offered by plebby and, in general, by the Language Toolkit. A full list is available in the interpreter's help system (see `:help classification`), or arbitrary varieties may be tested. (Note that all of these decision problems operate at the language level, not at the constraint level.) Knowing which set of classes contain a given language can offer insight regarding the properties of the language. This can assist in factoring the language, as one knows what types of constraints to try to find.

This system is also useful in exploring relationships between classes. While it cannot at this moment automatically determine whether a subclass relationship exists, one can manually construct a separating example language and verify that the separation holds. If a language is in class C but not in class C', then C is not a subclass of C'.

5 Conclusion

We have introduced plebby, the interactive theorem-prover built atop and packaged with the Language Toolkit, and demonstrated its use in defining, manipulating, and classifying regular languages. The project is freely available under the

MIT open-source license. Functionality goes well beyond what has been discussed here; all available commands are documented in the included manual pages or the interpreter's `:help` system. We only briefly touched on the visualization capabilities, and did not even mention the file I/O or grammatical inference capabilities. There are additionally stand-alone programs to `classify`, `display`, or automatically `factorize` regular languages.

As these tools were created for the purposes of education and research in mathematical and computational linguistics, performance was never the greatest concern. However, in order to be more widely useful, one key area of future work will be to improve performance to scale to industrial operations, where the automata under consideration might have large alphabets and several thousand states. Part of this work will involve changing the underlying representation of some of the core data structures; this work has already begun through splitting off some of the algebraic procedures into a separate `finite-semigroups` package.[3] The tradeoff is that the representation in this package strips much of the information that is pedagogically useful, such as which elements correspond to which strings. Thus care must be taken to avoid diminishing pedagogical utility when constructing representations for speed. For the classification task alone, we have also created AMALGAM[4] in the C programming language, which similarly discards information for better performance.

Other directions for future work are numerous. Some of our classification procedures return a description of the class parameters in addition to the Boolean response. We would like to provide such parameterizations for more classes in the future. For nonmembership, in some cases it might be nice to generate parameterized words as evidence. For some classes, this would be easy and would add to the utility as a theorem-prover. We would also like to be able to automatically generate semigroups satisfying given conditions, which may help in disproving a subset relationship between two varieties. Extending our current system, or perhaps creating a companion system, for similar analysis of finite-state transducers is a more involved goal. Using symbolic predicate-based symbols like the Microsoft Automata toolkit would increase utility in computational linguistics, especially with Carpenter-style feature systems [6]. Finally, we would like to add the capacity to translate foma scripts into PLEB expressions, or otherwise import automata with constraint semantics from such files.

We hope that this system will continue to enlighten all who study formal languages and their connections to algebra and logic.

Acknowledgments. The system described in this work owes its creation to the wonderful Theory of Computation course taught by Jim Rogers at Earlham College. Further enhancements arose from work with Jeffrey Heinz at Stony Brook University. And much gratitude is extended to the anonymous reviewers for their helpful suggestions.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

[3] Available at https://hackage.haskell.org/package/finite-semigroups.
[4] Available at https://github.com/vvulpes0/amalgam.

Appendix

This appendix contains selected worked exercises from various textbooks.

Exercise 2.1 from McNaughton and Papert [27]

"Decide whether each of the Figures 2.2–2.8 represents a locally testable event. Decide further whether it is locally testable in the strict sense." We cover only figures 2.4, 2.7 and 2.8. These figures are represented by the following AT&T files, named `mp-2-1-4.att`, `mp-2-1-7.att` and `mp-2-1-8.att`, respectively.

`mp-2-1-4.att`

1	4	a
1	2	b
1		
2	6	a
2	3	b
3	1	a
3	6	b
4	5	a
4	6	b
5	6	a
5	1	b
6	6	a
6	6	b

`mp-2-1-7.att`

1	2	a
1	1	b
1		
2	3	a
2	1	b
2		
3	3	a
3	4	b
4	3	a
4	5	b
5	6	a
5	7	b
6	3	a
6	7	b
7	6	a
7	1	b

`mp-2-1-8.att`

1	2	a
1	5	b
2	2	a
2	3	b
3	2	a
3	4	b
4	2	a
4	4	b
4		
5	6	a
5	5	b
6	7	a
6	5	b
7	7	a
7	5	b
7		

```
> :readATT mp-2-1-4.att _ _
> :isLT it
True
> :isSL it
True: k=5
> :readATT mp-2-1-7.att _ _
> :isLT it
False
> :isSL it
False
> :readATT mp-2-1-8.att _ _
> :isLT it
True
> :isSL it
False
```

Here, the tool directly answers the exercises, even providing additional information regarding the factor size k for the language locally testable in the strict sense.

5.1 Exercises from Sipser [41]

In the third edition of "Introduction to the Theory of Computation", Sipser [41] asks students to construct state diagrams for various regular languages. Exercise 1.4 focuses on intersections, 1.5 on complements, and 1.6 has assorted other languages. We select a small sample to cover here, all over the alphabet $\Sigma = \{a, b\}$:

1.4e $\{w | w$ starts with an a and has at most one b$\}$

1.5c $\{w | w$ contains neither the substrings ab nor ba$\}$

1.6n All strings except the empty string

As an aside, exercise 1.6 uses $\Sigma = \{0, 1\}$ in the original.

```
> =a{/a}=b{/b}
> :display ⋀{⋊⟨a⟩,¬⟨b,b⟩}    # 1.4e
> :display ¬⋁{⟨a b⟩,⟨b a⟩}    # 1.5c
> :display ¬⋊⋉⟨⟩              # 1.6n
```

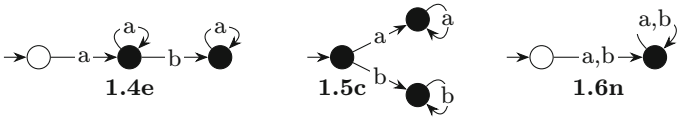

Fig. 2. State diagrams for Sipser, with node labels omitted.

Figure 2 depicts the results. Rejecting sink states are omitted from the display and must be filled in by hand.

References

1. Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., Mohri, M.: OpenFst: a general and efficient weighted finite-state transducer library. In: Holub, J., Žďárek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 11–23. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76336-9_3
2. Beauquier, D., Pin, J.-E.: Factors of words. In: Ausiello, G., Dezani-Ciancaglini, M., Della Rocca, S.R. (eds.) ICALP 1989. LNCS, vol. 372, pp. 63–79. Springer, Heidelberg (1989). https://doi.org/10.1007/BFb0035752
3. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964). https://doi.org/10.1145/321239.321249
4. Brzozowski, J.A., Simon, I.: Characterizations of locally testable events. Discret. Math. **4**(3), 243–271 (1973). https://doi.org/10.1016/S0012-365X(73)80005-6
5. Caron, P.: LANGAGE: a maple package for automaton characterization of regular languages. In: Wood, D., Yu, S. (eds.) WIA 1997. LNCS, vol. 1436, pp. 46–55. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0031380

6. Carpenter, B.: The Logic of Typed Feature Structures. Cambridge Tracts in Theoretical Computer Science, vol. 32. Cambridge University Press (1992). https://doi.org/10.1017/CBO9780511530098
7. Chandlee, J.: Strictly local phonological processes. Ph.D. thesis, University of Delaware (2014). https://chandlee.sites.haverford.edu/wp-content/uploads/2015/05/Chandlee_dissertation_2014.pdf
8. Chen, K.T., Fox, R.H., Lyndon, R.C.: Free differential calculus IV. The quotient groups of the lower central series. Ann. Math. **68**(1), 81–95 (1958). https://doi.org/10.2307/1970044
9. Chomsky, N.: On certain formal properties of grammars. Inf. Control **2**(2), 137–167 (1959). https://doi.org/10.1016/S0019-9958(59)90362-6
10. Clifford, A.H., Preston, G.B.: The Algebraic Theory of Semigroups, Mathematical Surveys and Monographs, vol. 7. American Mathematical Society, Providence (1961)
11. Edlefsen, M., Leeman, D., Myers, N., Smith, N., Visscher, M., Wellcome, D.: Deciding strictly local (SL) languages. In: Breitenbucher, J. (ed.) Proceedings of the 2008 Midstates Conference for Undergraduate Research in Computer Science and Mathematics, pp. 66–73 (2008)
12. Eilenberg, S.: Automata, Languages, and Machines, vol. B. Academic Press, New York (1976)
13. Eilenberg, S., Schützenberger, M.P.: On pseudovarieties. Adv. Math. **19**(3), 413–418 (1976). https://doi.org/10.1016/0001-8708(76)90029-3
14. Haines, L.H.: On free monoids partially ordered by embedding. J. Comb. Theory **6**(1), 94–98 (1969). https://doi.org/10.1016/s0021-9800(69)80111-0
15. Heinz, J.: Inductive learning of phonotactic patterns. Ph.D. thesis, University of California, Los Angeles (2007)
16. Heinz, J., Rawal, C., Tanner, H.G.: Tier-based strictly local constraints for phonology. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Short Papers, Portland, Oregon, vol. 2, pp. 58–64. Association for Computational Linguistics (2011). https://aclanthology.org/P11-2011
17. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press (2010). https://doi.org/10.1017/CBO9781139194655
18. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
19. Hulden, M.: Finite-state machine construction methods and algorithms for phonology and morphology. Ph.D. thesis, The University of Arizona (2009). https://hdl.handle.net/10150/196112
20. Hulden, M.: Foma: a finite-state compiler and library. In: Proceedings of the Demonstrations Session at EACL 2009, Athens, Greece, pp. 29–32. Association for Computational Linguistics (2009). https://aclanthology.org/E09-2008
21. Kleene, S.C.: Representation of events in nerve nets and finite automata. In: Shannon, C.E., McCarthy, J. (eds.) Automata Studies, Annals of Mathematics Studies, vol. 34, pp. 3–42. Princeton University Press (1956). https://doi.org/10.1515/9781400882618-002
22. Krebs, A., Lodaya, K., Pandya, P.K., Straubing, H.: Two-variable logics with some betweenness relations: expressiveness, satisfiability, and membership. Logical Methods Comput. Sci. **16**(3), 1–41 (2020). https://doi.org/10.23638/LMCS-16(3:16)2020
23. Lambert, D.: Relativized adjacency. J. Logic Lang. Inform. **32**(4), 707–731 (2023). https://doi.org/10.1007/s10849-023-09398-x

24. Lothaire, M.: Combinatorics on Words. Cambridge University Press, New York (1983)
25. MacCormick, J.: What Can Be Computed? A Practical Guide to the Theory of Computation. Princeton University Press (2018)
26. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biol. **5**, 115–133 (1943). https://doi.org/10.1007/bf02478259
27. McNaughton, R., Papert, S.A.: Counter-Free Automata. MIT Press, Cambridge (1971)
28. Mitchell, J., et al.: Semigroups – GAP Package, 5.1.0 edn. (2022). https://doi.org/10.5281/zenodo.592893
29. Pin, J.-E.: Syntactic semigroups. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, pp. 679–746. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-642-59136-5_10
30. Pin, J.E., Weil, P.: Polynomial closure and unambiguous product. Theory Comput. Syst. **30**(4), 383–422 (1997). https://doi.org/10.1007/bf02679467
31. van der Poel, S., et al.: MLRegTest: a benchmark for the machine learning of regular languages (2023). https://doi.org/10.48550/arXiv.2304.07687
32. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM J. Res. Dev. **3**(2), 114–125 (1959). https://doi.org/10.1147/rd.32.0114
33. Rawal, C., Tanner, H.G., Heinz, J.: (Sub)regular robotic languages. In: 2011 19th Mediterranean Conference on Control & Automation (MED), pp. 321–326 (2011). https://doi.org/10.1109/MED.2011.5983140
34. Reiterman, J.: The Birkhoff theorem for finite algebras. Algebra Universalis **14**, 1–10 (1982). https://doi.org/10.1007/BF02483902
35. Rogers, J., et al.: On languages piecewise testable in the strict sense. In: Ebert, C., Jäger, G., Michaelis, J. (eds.) MOL 2007/2009. LNCS (LNAI), vol. 6149, pp. 255–265. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14322-9_19
36. Rogers, J., Lambert, D.: Extracting subregular constraints from regular stringsets. J. Lang. Model. **7**(2), 143–176 (2019). https://doi.org/10.15398/jlm.v7i2.209
37. Rogers, J., Lambert, D.: Some classes of sets of structures definable without quantifiers. In: Proceedings of the 16th Meeting on the Mathematics of Language, Toronto, Canada, pp. 63–77. Association for Computational Linguistics (2019). https://doi.org/10.18653/v1/W19-5706
38. Romero, J.: Pyformlang: an educational library for formal language manipulation. In: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, pp. 576–582. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3408877.3432464
39. Schützenberger, M.P.: On finite monoids having only trivial subgroups. Inf. Control **8**(2), 190–194 (1965). https://doi.org/10.1016/s0019-9958(65)90108-7
40. Simon, I.: Piecewise testable events. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 214–222. Springer, Heidelberg (1975). https://doi.org/10.1007/3-540-07407-4_23
41. Sipser, M.: Introduction to the Theory of Computation, 3rd edn. Cengage Learning, Boston (2013)
42. Thomas, W.: Classifying regular events in symbolic logic. J. Comput. Syst. Sci. **25**, 360–376 (1982). https://doi.org/10.1016/0022-0000(82)90016-2
43. Thompson, K.: Programming techniques: regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968). https://doi.org/10.1145/363347.363387