

YaleNUSCollege

**FORMALIZING
BLOCK PUMPABLE LANGUAGE
THEORY**

ELAINE LI

**Capstone Final Report for BSc (Honours) in
Mathematical, Computational and Statistical Sciences**

Supervised by: Aquinas Hobor

AY 2018/2019

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.
2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

Unrestricted access

Make the Thesis immediately available for worldwide access.

Access restricted to Yale-NUS College for a limited period

Make the Thesis immediately available for Yale-NUS College access only from 4/2019 (mm/yyyy) to 12/2019 (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):

Potential publication in APLAS 2019. or CPP 2020.

After this period, the Thesis will be made available for worldwide access.

Other restrictions: (please specify if any part of your thesis should be restricted)

Elaine Fang Li / Cendana

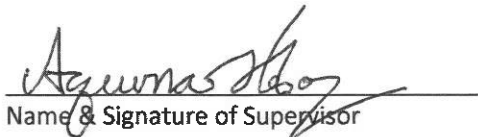
Name & Residential College of Student



Signature of Student

15/04/2019

Date



Name & Signature of Supervisor

15/4/2019

Date

Acknowledgements

This capstone project is indebted to Aquinas Hobor, for being there from the beginning of my journey into formal logic and computer science, and for indulging all of my idiosyncratic investigations from alien logic to obnoxious induction principles.

This capstone project is also indebted to Frank Stephan, for introducing me to the field through CS4232 Theory of Computation and CS5236 Advanced Automata Theory, and for generously giving his time to informally supervise this capstone.

I would like to thank the wonderful Wang Shengyi for his patience and Coq wizardry, which helped me make progress at a critical juncture.

I would like to thank Olivier Danvy for his mentorship, and Ilya Sergey for his kindness, understanding and knowledgeability.

This capstone benefited greatly from Certified Programming with Dependent Types by Adam Chlipala, in particular its spirited dedication to never throwing in the towel in the face of the dinosaur that is Coq's type universe.

Finally, I would like to thank my beloved family and friends for their presence and support throughout this journey.

Abstract

We present a mathematical formalization of results related to regularity and block pumping from automata theory, mechanized in the Coq proof assistant. Our formalization is similar in scope to existing work on formalizing regular and context-free language theory. We define formal languages and their properties, and prove a central equivalence between regularity, the block pumping property, and the block cancellation property. We also prove closure properties of block pumpable languages under union, concatenation and intersection.

Statement of Author's Contributions

All definitions used in the formal proof are standard, with the exception of my definition of regularity, which to the best of my knowledge is novel in its use of finiteness as a direct definition. Excluding explicitly-cited proofs and Ramsey's Theorem, which I admit as an axiom, all lemmas and theorems in the Coq development are stated and proven by me.

Intended Audience

The reader is assumed to have familiarity with the logic underlying the Coq proof assistant, the Calculus of Inductive Constructions, as well as elementary automata theory and combinatorics. All relevant formal and informal definitions are presented in the following section.

Notation and Definitions

We use Σ to refer to a finite alphabet, and σ to refer to symbols in the alphabet. We use variables x, y, z, w, v to denote words, and the special symbols ϵ and \emptyset to denote the empty word and empty set respectively. We use $|w|$ to denote the length of w . We use L, H to denote languages, and L_x to denote the derivative of a language with respect to a word x , and L_σ to denote the derivative of a language with respect to one symbol from the alphabet. We use bp_1, bp_2 to denote breakpoints, and i, j, k to denote natural numbers.

Definition. *Regular expressions*

The set-theoretic expressions used in regular expressions are defined as follows:

- The special symbol \emptyset denotes the empty set, or $\{\}$.
- The special symbol ϵ denotes the empty word.
- A finite list of strings in set brackets denotes the set of the corresponding elements.
- For any set L , L^* denotes the set of all strings obtained by concatenating finitely many strings from L .
- For any two sets L and H , the union of L and H , written $L \cup H$, denotes the set of all strings which are either in L or in H .
- For any two sets L and H , the intersection of L and H , written $L \cap H$, denotes the set of all strings which are in both L and H .

- For any two sets L and H , the concatenation of L and H , written $L \cdot H$, denotes the set $\{v \cdot w : v \in L \wedge w \in H\}$, the set of concatenations of members of L and H .
- For any two sets L and H , the set difference of L and H , written as $L - H$, denotes the set difference of L and H : $\{u : u \in L \wedge u \notin H\}$.

Regular expressions are recursively defined as follows:

- The constants ϵ and \emptyset are regular expressions.
- If a is any symbol, then a is a regular expression.
- If E and F are regular expressions, then $E \cup F$ is a regular expression.
- If E and F are regular expressions, then $E \cdot F$ is a regular expression.
- If E is a regular expression, then E^* is a regular expression.

Definition. *Finite automata*

Finite automata come in two forms, deterministic and non-deterministic.

- Deterministic finite automaton (DFA):

A deterministic finite automaton $(Q, \Sigma, \delta, s, F)$ is a state-based abstract machine which processes an input string and either accepts or rejects it. Q is the set of states, Σ is the finite alphabet, δ is the transition function mapping pairs from $Q \times \Sigma$ to Q , s is the starting state and F is the set of accepting states. The transition function $\delta : Q \times \Sigma \rightarrow Q$ defines a unique extension with

domain $Q \times \Sigma^*$ as follows: $\delta(q, \epsilon) = q$ for all q , and $\delta(q, wa) = \delta(\delta(q, w), a)$ for all $q \in Q, w \in \Sigma^*$ and $a \in \Sigma$. For any string $w \in \Sigma^*$, the DFA accepts w iff $\delta(s, w) \in F$.

- Non-deterministic finite automaton (NFA):

Intuitively, a non-deterministic finite automaton is a deterministic finite automaton with a multi-valued transition function. For any string $w \in \Sigma^*$, the NFA accepts w iff there exists a run of the NFA which accepts w .

Definition. *Regular grammar*

A grammar (N, Σ, P, S) consists of two disjoint sets of symbols N and Σ called non-terminals and terminals respectively, a set of derivation rules P , and a starting symbol $S \in N$. The derivation rules take the form $l \rightarrow r$, where l contains at least one symbol from N . v can be derived from w in one step iff there exist x, y and a rule $l \rightarrow r$ such that $v = xly$ and $w = xrw$. v can be derived from w in arbitrary number of steps iff there are $n \geq 0$ and $u_0, u_1, \dots, u_n \in (N \cup \Sigma)^*$ such that $u_0 = v$, $u_n = w$ and u_{m+1} can be derived from u_m in one step for each $m < n$. A regular grammar is one for which all derivation rules in P are of the form $A \rightarrow wB$ or $A \rightarrow w$, where $A, B \in N$ and $w \in \Sigma^*$.

Contents

1	Overview	12
2	Formal language theory	13
2.1	Introduction	13
2.2	An example	14
2.3	Pumping properties	15
2.4	Mr. Pumping Lemma	20
3	Formalizing automata theory	22
3.1	On an issue of representation	22
3.2	Existing work	22
3.3	Motivations	24
3.4	Principles	25
4	Our formalization	27
4.1	Coq standard library definitions	27
4.2	Our definitions	27
4.3	Ramsey's Theorem	33
4.4	Ehrenfeucht, Parikh and Rozenberg's Theorem	36

4.4.1	Lemma 2	38
4.4.2	Lemma 3	49
4.4.3	Lemma 4	50
4.5	Closure properties	54
5	Discussion	55
6	Conclusion	57
7	References	58
A	Coq project file organization	59
B	Additional definitions	61

1 Overview

In the first section, we introduce formal languages and motivate them as an interesting object of study due to their diverse representation mechanisms. We provide an overview of the existing results on a class of mathematical properties of formal languages called pumping properties.

In the second section, we motivate automata theory in general, and block pumpable language theory in particular, as an interesting object of formalization. We then provide an overview of the existing work on formalizing automata theory in various proof assistants. Finally, we give an account of the motivating principles behind our formalization, and the design choices they give rise to.

In the third section we describe our formalization. We present the definitions of the relevant mathematical objects and the statement of the central results expressed in Coq's logic. We explicate the formal proof through both a theoretical and proof engineering lens, highlighting points of divergence in the informal and formal mathematical argument and points of technical difficulty with Coq's type theory.

In the final section we discuss questions that arose in the process, and ideas for further directions.

2 Formal language theory

2.1 Introduction

Formal languages are defined as sets of words over a finite alphabet with decidable membership, and are broadly studied in theoretical computer science and discrete mathematics. Formal languages are mathematical objects that enjoy diverse representations based on the different mechanisms for checking or generating word membership:

- Algebraic expressions: formal languages are sets of words that can be *denoted* using general set-theoretic notation;
- Grammars: formal languages are sets of words that can be *generated* according to certain derivation rules;
- Automata: formal languages are sets of words *recognizable* by various kinds of abstract machines, or automata;
- Transducers: formal languages are sets of words *computed* by applying some function to sets of words.

The various representations draw inspiration from different disciplines. For example, regular expressions draw inspiration from traditional set theory. Recursive grammars were developed by linguists such as Axel Thue and Noam Chomsky[**chomsky**], who classified various formal grammars on what is now known as the Chomsky hi-

erarchy. The languages on the Chomsky hierarchy and their respective representations [hopcroft, kozen, khou, frank] are summarized in the table below. Let $A, B \in N$ denote non-terminals, $a \in \Sigma$ denote a terminal alphabet symbol, and $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$ denote any string of non-terminal and terminals.

Level	Language	Grammar	Automata
3	Regular	$A \rightarrow a$ or $A \rightarrow aB$	Finite state automaton
2	Context-free	$A \rightarrow \alpha$	Pushdown automaton
1	Context-sensitive	$\alpha A \beta \rightarrow \alpha \gamma \beta$	Linear-bounded automata
0	Recursively enumerable	$\alpha A \beta \rightarrow \gamma$	Turing machine

2.2 An example

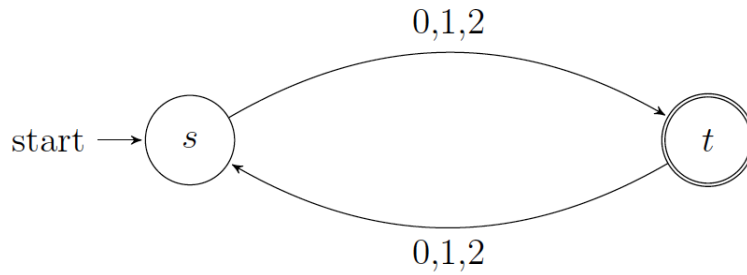
The following example of a regular language provides an intuition for the different representations of regular languages. Let L be the language over alphabet $\Sigma = \{0, 1, 2\}$ containing only words of odd length.

- The regular expression *denoting* L is

$$L = \{00, 01, 02, 10, 11, 12, 20, 21, 22\}^* \cdot \{0, 1, 2\}.$$

- The deterministic finite automaton *recognizing* L is

$$(\{s, t\}, \{0, 1, 2\}, \delta, s, \{t\}), \text{ with } \delta(s, a) = t \text{ and } \delta(t, a) = s \text{ for all } a \in \{0, 1, 2\}.$$



- The regular grammar *generating* L is $(\{S, T\}, \{0, 1, 2\}, P, S)$ with

$$P = \{S \rightarrow 0T|1T|2T|0|1|2, T \rightarrow 0S|1S|2S\}.$$

2.3 Pumping properties

Pumping properties are a well-studied class of mathematical properties of formal languages. Pumping lemmas, or iteration theorems, arise from the observation that infinite regular languages containing words of arbitrary length are recognizable by finite automata with a fixed number of states. This is enabled by the fact that repeating or omitting some parts in a word belonging to some regular language does not affect its membership. The word part which is repeated or omitted is referred to as the “pump”. This intuition is captured in the original pumping lemma [**hopcroft, kozen, khou, frank**], stated as follows:

Theorem 2.1. *Pumping Lemma*

Let $L \subseteq \Sigma^$ be a regular language. Then there is a constant k such that for every $u \in L$ of length at least k there is a representation $x \cdot y \cdot z = u$ such that $|xy| \leq k$, $y \neq \varepsilon$ and $xy^*z \subseteq L$.*

The original pumping lemma is a positive property, because it only characterizes

members of the regular language, and is silent on non-members. Further, it states a necessary but insufficient condition for regularity. For this reason, it is often used in modus tollens form to show that a language is non-regular.

Example. Let $\Sigma = \{0, 1\}$ and $L = \{0^n 1^n : n \in \mathbb{N}\}$. L is non-regular.

Proof. To show that L is non-regular, we show that L does not satisfy the pumping lemma with any constant. Assume that L satisfies the pumping lemma with some constant k . Consider the word $0^k 1^k \in L$. By assumption there is a representation $x \cdot y \cdot z = 0^k 1^k$ such that $|xy| \leq k$ and $y \neq \varepsilon$. However, $xyyz = 0^{k+n} 1^k$ for some $n > 0$, and by definition, $xyyz \notin L$. Therefore, we reach a contradiction, and such a constant k does not exist. \square

Further investigation of pumping properties for regular languages was motivated by the quest to find a pumping condition equivalent to regularity, that is, both necessary and sufficient.

John Myhill and Anil Nerode [**nerode**] formulated a property of formal languages that uniquely characterizes regularity unrelated to the notion of pumping.

Theorem 2.2. *Myhill-Nerode Theorem*

Given a language L , let $L_x = \{y \in \Sigma^ : xy \in L\}$ be the derivative of L with respect to x . The language L is regular iff the number of different derivatives L_x is finite.*

In 1978, Jaffe [**jaffe**] drew from the central idea of Myhill-Nerode – a language having finitely many derivatives – and formulated a version of the pumping property that is equivalent to regularity.

Theorem 2.3. *Jaffe's Pumping Lemma*

A language $L \subseteq \Sigma^*$ is regular iff there is a constant k such that for all $x \in \Sigma^*$ and $y \in \Sigma^k$ there are u, v, w with $y = uvw$ and $v \neq \varepsilon$ such that, for all $h \in \mathbb{N}$, $L_{xuv^h w} = L_{xy}$.

Proof. \Leftarrow Assume that L is regular. Then, by Myhill-Nerode Theorem, L has a finite number of derivatives. Let the number of derivatives of L be k . By the pigeonhole principle, two of $k + 1$ derivatives of L must be the same. Let the two derivatives be L_{xu} and L_{xuv} , where xu is a prefix of xy and $|y| = k$. Let $w \in \Sigma^*$ be $xy = xuvw$. For any $z \in \Sigma^*$, $z \in L_{xuv}$ iff wz in L_{xu} iff $wz \in L_{xuv}$ iff $z \in L_{xy}$, thus $L_{xuv} = L_{xy}$. By induction over i , we can show that $L_{xu(v^i)} = L_{xu}$, and thus $L_{xu(v^i)w} = L_{xy}$.

\Rightarrow Whenever a string z is of length at least k , L_z is equal to L_{xuv} with $|xuv| < |z|$ and therefore every derivative is equal to one of length up to k . As the alphabet is finite, there are only finitely many derivatives with word label length up to k . Therefore, L is regular by the Myhill-Nerode Theorem. \square

In 1979, Ehrenfeucht, Parikh and Rozenberg [**EPR**] showed that a more natural formulation of pumping based on “blocks” is also equivalent to regularity.

Definition. $L \subseteq \Sigma^*$ has the block pumping property if there is a k such that for all $x, x, y_1, \dots, y_k, w' \in \Sigma^*$ if $x = wy_1 \dots y_k w'$ then there exist $m, j, i \leq m < j \leq k$ such that $y_{m+1} \dots y_j$ is a pump for x relative to L .

Theorem 2.4. *EPR's Theorem*

Regularity, the block pumping property and the block cancellation property are equivalent.

The above notions of pumping focus solely pumping properties in relation to regularity. In 2016, Chak et al. [**chak**] used the positive block pumping property to characterize a class of languages called block pumpable languages. They show that the class of block pumpable languages is orthogonal to the Chomsky hierarchy. In particular, every regular language is positively block pumpable, but there exist non-regular languages that are positively block pumpable [**chak**]. We provide a proof for the example language given in [**chak**].

Definition. Let x_n, y_n be inductively defined as follows:

$$x_0 = 0$$

$$y_0 = 1$$

$$x_{n+1} = x_n y_n y_n x_n y_n x_n x_n y_n x_n y_n y_n x_n y_n x_n x_n y_n$$

$$y_{n+1} = y_n x_n x_n y_n x_n y_n y_n x_n y_n x_n x_n y_n x_n y_n y_n x_n$$

It can be shown that every x_n and y_n is cube-free, and of length 16^n .

Example. Let $L = \{z \in 0, 1^* : \forall n \in \mathbb{N}, z \neq x_n \wedge z \neq y_n\}$. L is positively block pumpable but not regular.

Proof. We first show that L is not regular by showing that its complement is not regular. By de Morgan's law, the complement $\bar{L} = \{z \in 0, 1^* : \forall n \in \mathbb{N}, z = x_n \vee z = y_n\}$. Therefore, \bar{L} is cube-free, and cannot contain any subset of the form uv^*w , and thus it cannot satisfy any pumping property. Therefore, L is not regular.

We then show that L is positively block pumpable with block pumping constant 5.

By the properties of x_n and y_n , all cube-containing words are members of L , and all words not of length 16^m for some $m \in \mathbb{N}$ are members of L . Let $x \in L$ and let the breakpoints split the word x into $w_0(1)w_1(2)w_2(3)w_3(4)w_4(5)w_5$. Now we show that either breakpoints (1,3) or breakpoints (2,3) define a pump $\forall n \in \mathbb{N}$.

- When $n \geq 3$, the resulting word with the pump repeated three times must by definition contain a cube. Because all cube-containing words are members of L , the resulting word is a member of L .
- When $n = 1$, the word remains unchanged and $w \in L$ by assumption.
- When $n = 0$ or $n = 2$, we show that omitting or twice-repeating either of the breakpoint pairs results in a word w' that is not of length 16^m for any $m \in \mathbb{N}$. By assumption, $|x| \neq 16^m$ for all $m \in \mathbb{N}$. Now either the prefix $w_0w_1w_2$ before breakpoint (3) or the suffix w_4w_5 after breakpoint (3) has length less than or equal to $|x|/2$. Assume without loss of generality that $|w_0w_1w_2| \leq |x|/2$ and $|w_3w_4w_5| \geq |x|/2$.

- If we select breakpoints (1,2), when $n = 0$, $w' = w_0w_2w_3w_4w_5$, and when $n = 2$, $w' = w_0w_1w_1w_2w_3w_4w_5$.
- If we select breakpoints (1,3), when $n = 0$, $w' = w_0w_3w_4w_5$, and when $n = 2$, $w' = w_0w_1w_2w_1w_2w_3w_4w_5$.

Because the shortest possible resulting word $|w_0w_3w_4w_5| \geq |x|/2$, and the longest possible resulting word $w_0w_1w_2w_1w_2w_3w_4w_5 \leq 3|x|/2$, only one w' can have a length which is a power of 16. Therefore, the other pair of breakpoints defines a pump.

□

Further, there exist context-free languages that are positively block pumpable.

They also show the following closure properties of block pumpable languages:

Theorem 2.5. *Block pumping closure properties*

Block pumpable languages are closed under intersection, union, concatenation, but not under Kleene star and set difference.

2.4 Mr. Pumping Lemma

We present a game-semantic formulation¹ of the proof that some arbitrary language L satisfies the block pumping property to further clarify the structure of the proof. Consider a game between you and an interlocutor, Mr. Pumping Lemma. Your objective is to convince Mr. Pumping Lemma that L satisfies the block pumping property; Mr. Pumping Lemma's objective is to undermine your attempt. You and Mr. Pumping Lemma take turns to make moves: you pick the existential witnesses, and Mr. Pumping Lemma picks the universal witnesses.

1. You pick a block pumping constant k .
2. Mr. Pumping Lemma picks a word w and a set of breakpoints bps of size k .
3. You pick two breakpoints bp_1, bp_2 from the breakpoint set bps .
4. Mr. Pumping Lemma pumps the subword between bp_1 and bp_2 to his heart's content.

¹The application of game semantics to pumping lemmas is inspired by <https://math.stackexchange.com/questions/151744/application-of-pumping-lemma-for-regular-languages>

If the resulting word w' has the same membership as the original word w , then you win, and you have shown that L satisfies the block pumping property. The game semantics is recognized by L iff w is recognized by L .

The game semantics highlights the fact that your ability to win against Mr. Pumping Lemma turns on you choosing a block pumping constant in your first move and pair of breakpoints in your second move that withstand whatever word, breakpoint set and repetition number your opponent picks. Indeed, proofs involving the block pumping property centrally turn on finding the right existential witnesses for the block pumping constant and the breakpoints.

3 Formalizing automata theory

3.1 On an issue of representation

The diverse representation mechanisms of formal languages paired with proofs of their equivalence [hopcroft, kozen, khou, frank] allow informal proofs to pick the representation best suited for the argument at hand. For example, the proof that regular languages satisfy the original pumping condition is easily shown using the automata-based representation of regular languages, by appeal to the pigeonhole principle [hopcroft, khou]. The same proof using the regular expression-based representation [kozen, frank] requires structural induction, which breaks down into individual cases for the empty string, intersection, concatenation and Kleene star, and is much more involved.

This advantage is lost in formal, mechanized proofs. A formal proof environment imposes a high cost overhead on defining and commuting between multiple representations of the same mathematical object, as will be evidenced in the rest of this chapter. This representational tension makes formal languages an interesting candidate for formal verification in a proof assistant.

3.2 Existing work

Our work is broadly situated in the area of formalizing mathematics. Interest in formalizing automata theory in particular dates back to 1997, and was motivated by the question of whether constructive type theory is a natural expression for

computational mathematics as set theory is for classical mathematics. Constable et al. [**constable**] chose automata theory, and the Myhill-Nerode Theorem in particular, as the object of constructive formalization, because it is a canonical instance of extracting an algorithm from a proof. Their work consists of a formalization the Myhill-Nerode Theorem and its corollary state minimization algorithm in NuPrl using the automata-based definition of regular languages, and notably, took a team of 4 researchers 18 months to complete. Representations of formal languages, in particular regular expressions, are well-studied in computer science, and summarized by Nipkow et al. [**nipkow**], who present an overview of five different formal decision procedures for regular expression equivalence. Notably, a formalization of the Myhill-Nerode Theorem using regular expressions has been completed in HOL/Isabelle [**proofpearl**]. The authors motivate their representation choice by appealing to theoretical elegance and representational cost – HOL/Isabelle’s type system does not support higher order type quantification, making finite graphs prohibitively difficult to define. Formal results involving regular and context-free languages have been obtained in various proof assistants, including HOL4, HOL/Isabelle Isabelle, Agda and Coq.

Existing work on formalizing automata theory in Coq primarily consists of two works on regular and context-free languages respectively. Doczkal et al. [**constructive**] present a constructive theory of regular languages in Coq. They define both automata-based and regular-expression-based representations of regular languages, and prove Kleene’s theorem, which states the equivalence between them. Their development also includes constructive proofs of Myhill-Nerode’s Theorem, the regular pump-

ing lemma and closure properties of regular languages, using the automata-based representation.

Ramos et al. [**ramosresults**][**ramospump**] formalized context-free language theory using a grammar-based representation of context-free languages in Coq. They prove the pumping lemma for context-free languages, closure properties, grammar minimization properties, and reducibility into Chomsky normal form. Their paper mentions formalizing pushdown-automata-based representations of context-free languages in Coq as a plan for future development.

More recent results on regular and context-free language theory are practically and theoretically motivated by the desire to provide a reasoning framework for the two most important formal language classes on the Chomsky hierarchy. A salient practical application for regular and context-free languages are text processing and parser generation respectively. To the best of my knowledge, this work is the first formalization of a language class that is orthogonal to the Chomsky hierarchy.

3.3 Motivations

Block pumpable language theory presents itself as an interesting candidate for formalization because block pumpable languages cannot be directly defined in terms of the standard representation mechanisms for formal languages, and instead are directly defined in terms of their block pumping property. Additionally, results from block pumpable language theory utilize ideas from combinatorics such as Ramsey's Theorem and König's Lemma, which are absent from regular and context-free language theory.

3.4 Principles

Our formalization makes the salient design choice to rely solely on the inductive data types for natural numbers, lists and sigma types in Coq’s standard library to define our mathematical objects. We import no additional mathematical libraries of formalized results or syntactical extensions to Coq’s tactic language. Our reasons for doing so are twofold: 1) we believe that relying on a small set of simple algebraic data types maximizes simplicity and clarity of the proofs, and 2) we believe that developing definitions from scratch and proving properties about them incrementally provides both the prover and the reader with a more thorough understanding of the mathematical argument, as well as of Coq’s metalogic and type theory.

As a consequence, we develop extensive libraries containing supplementary lemmas about the mathematical objects we reason about in our proofs, including natural numbers, polymorphic lists, breakpoint sets, finite types and injective mappings.

Further, we utilize a relatively small subset of Coq’s tactic language for proof construction. Excluding book-keeping tactics such as `assert` for adding proven facts into the context, `remember` for naming variables and `clear` for removing used hypotheses from the context, and equality-based tactics `reflexivity` and `rewrite` for applying the reflexivity and transitivity of equality, the tactics we use directly correspond to propositional and predicate logic inference rules.

$$\begin{array}{c}
\frac{P \quad Q}{P \wedge Q} \wedge i, \text{ split} \qquad \frac{P \wedge Q}{P} \wedge e_1, \text{ destruct} \qquad \frac{P \wedge Q}{Q} \wedge e_2, \text{ destruct} \\
\\
\frac{P}{P \vee Q} \vee i_1, \text{ left} \qquad \frac{Q}{P \vee Q} \vee i_2, \text{ right} \qquad \frac{P \vee Q \quad \frac{P}{X} \quad \frac{Q}{X}}{X} \vee e, \text{ destruct} \\
\\
\frac{\frac{P}{Q}}{P \rightarrow Q} \rightarrow i, \text{ intro} \qquad \frac{P \quad P \rightarrow Q}{Q} \rightarrow e, \text{ apply} \\
\\
\frac{P(x_0)}{\forall x, P(x)} \forall i, \text{ intro} \qquad \frac{\forall x, P(x)}{P(x_{mine})} \forall e, \text{ spec} \\
\\
\frac{P(x_{mine})}{\exists x, P(x)} \exists i, \text{ exists} \qquad \frac{\exists x, P(x)}{P(x_0)} \exists e, \text{ destruct}
\end{array}$$

The only proof automation technique we employ is the omega tactic, which automatically resolves first-order Presburger arithmetic. We use omega primarily to discharge trivial proof obligations involving comparison of natural numbers.

Our design choices entail that every formal proof can be rewritten into a proof tree constructed from the above inference rules. Our motivation is to make the formal proofs as illustrative as possible.

Towards these ends, we have also adopted standard programming best practices in organizing the proof development by making use of modularity and `Makefiles`. A comprehensive description of the organization and content of the proof scripts, as well as how to build and install the development, can be found in Appendix A.

4 Our formalization

4.1 Coq standard library definitions

We first state the inductively-defined data types for natural numbers, lists and sigma-types found in Coq's standard library.

```
Inductive nat : Set := O : nat | S : nat → nat.
```

```
Inductive list (A : Type) : Type := nil : list A | cons : A → list A → list A.
```

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
```

```
  exist : ∀ x : A, P x → {x : A | P x}.
```

Coq's sigma-types are defined via a basic type paired with a logical proposition guaranteeing all inhabitants of this new sigma-type satisfy some property P . Elements of sigma-types are dependent pairs consisting of an x and the logical proposition $P x$.

4.2 Our definitions

To begin with, we define alphabets as inductive types with a finite number of constructors. The alphabet is conventionally designed to be the set $\Sigma = \{0, 1, 2\}$, and our definition follows suit.

```
Inductive T : Type :=
```

```
  | aa : T
```

```
  | bb : T
```

| $cc : T$.

We define words as lists of elements from the alphabet.

Definition $word := list\ T$.

Our definition of languages reflects the notion of some membership condition for a set of words. We define languages as a type $word \rightarrow Prop$, which refers to a type which takes some word and returns a proposition stating some property about that word. Here, the property is that of being a member of the given language.

Definition $language : Type := word \rightarrow Prop$.

One feature of Coq's higher order type theory is the distinction between type equality and propositional equivalence. Type equality is uniquely defined for each inductively-defined data type, whereas propositional equivalence captures the notion of provability. For example, two natural numbers are equal iff they are syntactically equal by repeated application of their inductive constructors. Two propositions are equivalent iff we can prove one from the other, and vice versa. While equality is usually taken for granted in informal mathematical proofs, it is often a subtle and contentious topic in type theory. In first-order logic, proofs are not treated as first-class citizens, and therefore only uses the notion of type-based language equality. We introduce into Coq's base logic the axiom of functional extensionality and propositional extensionality, which allows us to commute between type-based equality and propositional equivalence for languages.

Axiom $functional_extensionality : \forall X\ Y\ (f\ g : X \rightarrow Y),$

$(\forall x : X, f\ x = g\ x) \rightarrow$

$$f = g.$$

Axiom *prop_ext*: $\forall P Q : \text{Prop}, P \leftrightarrow Q \rightarrow P = Q.$

Language equality amounts to two languages having the same membership condition for all words. We formulate language equality as follows:

Lemma *language_equality* : $\forall l1 l2 : \text{language},$

$$l1 = l2 \leftrightarrow \forall w : \text{word}, l1 w \leftrightarrow l2 w.$$

We define derivatives of a language both as a type and as a predicate. A derivative language of some language L with respect to some word label x is defined as:

Definition *derivative_of* ($L : \text{language}$) ($t : \text{word}$) : $\text{language} :=$

$$\text{fun } w \Rightarrow L (t ++ w).$$

The property of being a derivative of some language L with respect to word label x is defined as follows:

Definition *is_derivative* ($L L' : \text{language}$) : $\text{word} \rightarrow \text{Prop} :=$

$$\text{fun } (x : \text{word}) \Rightarrow \forall (w : \text{word}), L' w \leftrightarrow L (x ++ w).$$

Defining block pumping and block cancellation properties requires the definition of block pumping constants and word pumps. We choose to define the pumps in the block pumping property using index-based breakpoints instead of subwords to allow for clear reasoning via algebraic manipulation of inductively defined data types. Therefore, we are required to define breakpoint sets and breakpoints.

Mathematically, breakpoints for a given word are an ordered, duplicate-free set of indices of fixed size, where the indices refer to positions within the word, and can

fall on both the leftmost and rightmost end of the word. We represent breakpoints as natural numbers, and breakpoint sets as lists of natural numbers. Lists are inductively defined in Coq as a data type with zero arity and two constructors: `nil` and `cons`. Primitive lists differ from mathematical breakpoint sets in several significant ways: lists are ordered, can contain duplicates, and do not place any constraint on its elements besides requiring that they are of some type X .

Therefore, we leverage Coq's sigma-types to define breakpoint sets by augmenting the `list` type with additional logical information. Breakpoint sets are thus defined as a type `list nat` which depends on an argument k , the block pumping constant, and satisfies the following properties:

1. The list is of length k ;
2. All elements in the list are strictly increasing;
3. The last element in the list is less than or equal to the length of the word.

Definition *breakpoint_set_predicate* ($l : list\ nat$) ($w : word$) ($p : block_pumping_constant$)

: Prop :=

$$length\ l = p$$

$\wedge \forall n\ m : nat,$

$$n < m < (length\ l) \rightarrow$$

$$nth\ n\ l\ d < nth\ m\ l\ d$$

$\wedge last\ l\ d \leq length\ w.$

Definition *breakpoint_set* ($p : block_pumping_constant$) ($w : word$) : Type :=

$\{bl : list\ nat \mid breakpoint_set_predicate\ bl\ w\ p\}$.

Element j in the list directly corresponds to the j -th position in the list, which then corresponds to the breakpoint between the j -th and $(j+1)$ -th symbol in the word. Therefore, element 0 refers to the breakpoint at the left-most end of the word and element $length\ w$ refers to the breakpoint at the right-most end of the word, both of which are valid breakpoints. The former breakpoint is implicitly made possible by breakpoints being of type nat , and the latter breakpoint is explicitly made possible in the third conjunctive clause of the breakpoint set predicate.

We define block pumping constants, which are natural numbers greater than or equal to 2, in a similar way, by augmenting Coq's natural number data type with a logical proposition guaranteeing that the number is greater than or equal to 2.

Definition $p_predicate\ (p : nat) : Prop :=$

$p \geq 2.$

Definition $block_pumping_constant : Type :=$

$\{ p : nat \mid p_predicate\ p\}$.

We then define breakpoints as members of the breakpoint set. Therefore, the breakpoint type depends on a set of breakpoints bps , a word w and a block pumping constant k .

Definition $breakpoint_predicate\ (i : nat)\ (p : block_pumping_constant)\ (w : word)\ (bl : breakpoint_set\ p\ w) : Prop :=$

$In\ i\ (bl_proj1\ bl).$

Definition $breakpoint\ \{p : block_pumping_constant\}\ \{word : list\ T\}\ (bl : break-$

$point_set\ p\ word) : Type :=$

$\{i : nat \mid breakpoint_predicate\ i\ p\ word\ bl\}$.

We choose for breakpoints to depend on breakpoint sets and not vice versa because the statement of the block pumping property nests existential quantification of individual breakpoints inside universal quantification of breakpoint sets.

Definition $block_pumpable_matching\ (L : language) :=$

$\exists k : block_pumping_constant,$

$\forall w : word,$

$\forall bl : breakpoint_set\ k\ w,$

$\exists i\ j : breakpoint\ bl,$

$i < j$

$\wedge \forall m : nat, L\ (firstn\ i\ w\ ++\ napp\ m\ (pumpable_block\ i\ j\ w)\ ++\ skipn\ j\ w) \leftrightarrow$

$L\ w.$

Definition $block_cancellable_matching\ (L : language) :=$

$\exists k : block_pumping_constant,$

$\forall w : word,$

$\forall bl : breakpoint_set\ k\ w,$

$\exists i\ j : breakpoint\ bl,$

$i < j$

$\wedge L\ (firstn\ i\ w\ ++\ skipn\ j\ w) \leftrightarrow L\ w.$

We postpone our definition of regularity for later discussion. For now, suffice it to consider the representational obligations accompanying two standard ways of

defining regularity:

1. A language is regular iff all words in the language match some regular expression.
2. A language is regular iff all words in the language are accepted by some finite automaton.

Both versions of regularity require us to define new inductive data types: regular expressions, and finite automata. We refer the reader to Appendix B for the formal definitions in Coq.

4.3 Ramsey's Theorem

Proofs involving breakpoint sets make use of a foundational result in combinatorics, Ramsey's Theorem [ramsey]. Ramsey's Theorem enjoys wide applications in various areas of mathematics and can be stated informally in several different ways. Formally stating Ramsey's Theorem to reason about block pumping properties further requires attention to its 1) data types and 2) predicate logic quantifier nesting. We start with the most general graph-based statement of Ramsey's Theorem, followed by a set-based version, and finally present the version we use which is specific to breakpoint sets, and explicate how it is obtained from the original statement.

Theorem 4.1. *Ramsey's Theorem for graphs*

One can always find monochromatic cliques in any edge-labeling with colors of a sufficiently large complete graph.

Defining variable k to be the size of the monochromatic clique, variable $r(k)$ to be the size of the sufficiently large complete graph, and representing the vertices in the graph as a set I , we can express the notion of edge-coloring with a coloring function C of type $vertex \rightarrow vertex \rightarrow color$, which in turn corresponds to $T \rightarrow T \rightarrow color$, where T refers to the type of the representative in the set of vertices. Further, we define that C maps every single pair (i, j) in the set of vertices I , thus capturing the notion of a complete graph – every two vertices are connected and colored.

Theorem 4.2. *Ramsey’s Theorem for sets*

For every natural number k and finite set Q of colors, there exists a natural number $r(k)$ such that for every ordered set I with $r(k)$ elements and for every function mapping each pair (i, j) to a color $C(i, j)$, there exists a subset $J \subset I$ with k elements such that all pairs in J are mapped to the same color.

We first modify the data types in the set-based statement to be specific to the data types that define block pumping. Breakpoint sets are ordered sets with the additional properties that 1) they contain greater than or equal to two elements and 2) they are strictly increasing and their last element is less than the length of some constant. The first property requires us to quantify over block pumping constants, which are natural numbers that are greater than or equal to two, instead of ordinary natural numbers, for k . Further, “colors” of pairs of breakpoints correspond to “properties” of pairs of breakpoints, and therefore we set the type of our coloring function to $breakpoint \rightarrow breakpoint \rightarrow \text{Prop}$.

The notion of colors implicitly requires that each color is distinct, and each edge, or pair of elements, can only be assigned one unique color. When representing colors

in terms of decidable propositions, which must be either true or false, the number of conjunctive propositions n gives rise to 2^n different colors, corresponding to the number of rows in the truth table, where each color has a unique truth assignment for each proposition. In the two-color case, we have one property P and we assign one color to be “P” and the other color “not P”. In the four-color case, we have two properties P and P , and the four colors must strictly be “P and Q”, “P and not Q”, “not P and Q”, and “not P and not Q”.

Expressing Ramsey’s Theorem in predicate logic further requires special attention to quantifier nesting. Firstly, Ramsey’s constant $r(k)$ must directly depend on constant k , therefore the first \forall and the second \exists are tightly bound. Secondly, breakpoint sets depend on some word and a block pumping constant, therefore the third \forall and the fourth \exists are tightly bound. Finally, the theorem quantifies over all possible coloring functions, or in this case, all possible property-assignments to pairs of breakpoints. Therefore, the fifth \forall must precede the sixth \exists because the choice of property must not depend on the sublist of size k . Finally, the condition that the two breakpoints are ordered must appear as an implicative antecedent in the statement of the theorem itself, and not in the definition of the properties about breakpoints.

This gets us the following block pumping-specific, formal statement of Ramsey’s Theorem required in our proofs:

Axiom *Theorem_of_Ramsey* :

$\forall (k : \text{block_pumping_constant}),$
 $\exists rk : \text{block_pumping_constant},$

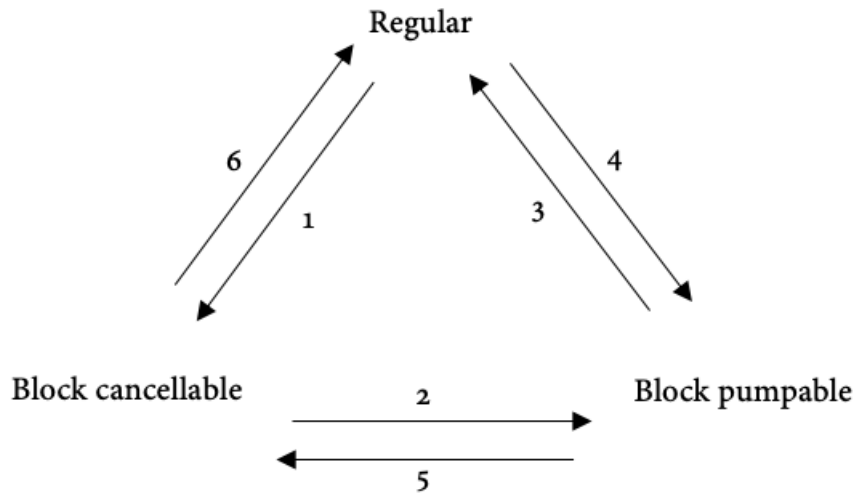
$$\begin{aligned}
& \forall w : \text{word}, \\
& \forall \text{bps} : \text{breakpoint_set } rk \ w, \\
& \quad \forall (P \ Q : \text{nat} \rightarrow \text{nat} \rightarrow \text{PROP}), \\
& \quad \exists \text{bps}' : \text{breakpoint_set } k \ w, \\
& \quad \quad \text{sublist } \text{bps}' \ \text{bps} \wedge \\
& \quad \quad ((\forall \text{bp1 } \text{bp2} : \text{breakpoint } \text{bps}', \\
& \quad \quad \quad \text{bp1} < \text{bp2} \rightarrow (P \ \text{bp1} \ \text{bp2}) \wedge (Q \ \text{bp1} \ \text{bp2})) \\
& \quad \vee (\forall \text{bp1 } \text{bp2} : \text{breakpoint } \text{bps}', \\
& \quad \quad \quad \text{bp1} < \text{bp2} \rightarrow \neg (P \ \text{bp1} \ \text{bp2}) \wedge (Q \ \text{bp1} \ \text{bp2})) \\
& \quad \vee (\forall \text{bp1 } \text{bp2} : \text{breakpoint } \text{bps}', \\
& \quad \quad \quad \text{bp1} < \text{bp2} \rightarrow (P \ \text{bp1} \ \text{bp2}) \wedge \neg (Q \ \text{bp1} \ \text{bp2})) \\
& \quad \vee (\forall \text{bp1 } \text{bp2} : \text{breakpoint } \text{bps}', \\
& \quad \quad \quad \text{bp1} < \text{bp2} \rightarrow \neg (P \ \text{bp1} \ \text{bp2}) \wedge \neg (Q \ \text{bp1} \ \text{bp2}))).
\end{aligned}$$

4.4 Ehrenfeucht, Parikh and Rozenberg's Theorem

EPR's theorem states that the block pumping property, the block cancellation property and regularity are equivalent. In other words, any language which satisfies one of these three properties can also be shown to satisfy the other two.

EPR's theorem breaks down into six directions, as illustrated in the figure below. They prove the triangle commutes along directions (4), (5) and (6). Direction (5) is trivial, because block cancellation is just a special case of block pumping for when the pump is repeated 0 times, i.e. cancelled out. EPR's proof of (4) informally appeals to the pigeonhole principle. The proof of (6) is the most involved, and

receives detailed treatment in the section below.



In principle, showing either the clockwise or counterclockwise direction suffices. In the interest of completeness and of demonstrating the compactness of our definition of regularity, we construct formal proofs of all six directions.

In the following section we give a sketch of the proof of (6) as given in [EPR]. We then explicate three key ingredients in the mathematical argument made explicit by constructing the formal proof. We point out an interesting phenomenon: the major omission in the informal proof of (6) marks the point at which we require the axiom of choice to be added to Coq’s base logic.

EPR breaks down the proof of their theorem into three lemmas.

Lemma 4.3. *EPR’s Lemma 2*

There are only finitely many languages that are block cancellable with k .

Lemma 4.4. *EPR’s Lemma 3*

If L is in C_k then so is L_σ for all σ in Σ .

Lemma 4.5. *EPR's Lemma 4*

Let P be some property of languages such that (i) there are only finitely many languages that have P ; (ii) for all σ in Σ , if L has P then L_σ has P . Then P implies regularity.

In this case, the property P is instantiated with the block cancellable property for some k .

4.4.1 Lemma 2

EPR claims that it suffices to show that for any two languages that are block cancellable with k , if the two languages agree on word membership for all words up to Ramsey's constant $r(k)$, then these two languages are equal.

The formal proof of Lemma 2 articulates the following three central ingredients: 1) the four-color version of Ramsey's Theorem for pairs, 2) strong induction on word length, and 3) a set-theoretic fact stating that every finite list is injectively mapped onto from another finite list.

Ramsey's Theorem for four colors EPR's proof of Lemma 2 is completed with Ramsey's Theorem for two colors, with the first color being the property "is a breakpoint for L_1 ", and the second color being the property "is not a breakpoint for L_1 ". It then argues that the disjoint sets Z and Z' are the same for L_1 and L_2 . In a formal, constructive setting, the proof requires Ramsey's Theorem for four colors. This is because the breakpoints are extracted using existential quantifiers, and existential elimination does not guarantee any additional properties about the arbitrary witness

it produces. EPR's claim requires that for two identical propositions in predicate logic, "for all breakpoint sets of size k , there exist breakpoints bp_1 and bp_2 , such that. . .", if we eliminate the universal quantifier with a particular breakpoint set of size k , then we can destruct the existential quantifiers to produce the exact same two witnesses. This is plainly false. Consider the following statement:

“For every natural number n , there exists a natural number m such that $m \geq n$.”

Suppose we eliminate the universal \forall by supplying two identical statements with the natural number 7. Our statement changes to:

“There exists a natural number m such that $m \geq 7$.”

If we apply existential elimination to this statement, we are given a witness m_0 . The only information we have about m_0 is that it satisfies the greater than equal to relation with regards to 7. There are an infinite number of possible m'_0 s, and therefore we have no way of showing that the two m'_0 s extracted from these two identical statements are the same one.

The four colors for Ramsey's Theorem in the formal proof correspond to the properties “are breakpoints for both L_1 and L_2 ”, “are breakpoints for L_1 but not L_2 ”, “are breakpoints for L_2 but not L_1 ”, and “are breakpoints for neither L_1 nor L_2 ”. The first three cases are discharged in similar ways, and the final case is discharged through contradiction.

Strong induction The proof of Lemma 2 argues that for any word w of arbitrary length, one can shorten it to a word w' which both L_1 and L_2 agree on the mem-

bership of. We perform case analysis on $|w|$, and trivially discharge the case where $|w| \leq r(k)$ by assumption. In the case that $|w| > r(k)$, we want to show that w can be pumped down to a word w' where $|w'| \leq r(k)$, and then use the same reasoning pattern as above. This cannot be shown directly using the fact that $|w| > r(k)$, because removing a non-empty pump from a word arbitrarily longer than $r(k)$ does not necessarily result in a word that is shorter than $r(k)$. Therefore, inductive reasoning is required. The induction principle at work differs from the standard induction principle that syntactically mirrors the inductive definition of natural numbers:

Theorem *nat_ind* : $\forall P : \text{nat} \rightarrow \text{Prop}$,

$P\ 0 \rightarrow$

$(\forall n : \text{nat}, P\ n \rightarrow P\ (S\ n)) \rightarrow$

$\forall n : \text{nat}, P\ n.$

This proof instead requires a stronger induction principle:

Theorem *strong_induction* :

$\forall P : \text{nat} \rightarrow \text{Prop}$,

$P\ 0 \rightarrow$

$(\forall n : \text{nat}, (\forall m, (m < n \rightarrow P\ m)) \rightarrow P\ n) \rightarrow$

$(\forall n : \text{nat}, P\ n).$

Strong induction is frequently used to reason about natural numbers, and we omit the details of the proof of `strong_induction`¹. Setting up strong induction on word length in the proof of Lemma 2 requires some proof engineering:

¹The complete proof can be found in `stronginduction.v`.


```

remember (length w) as n.
assert (H_move := Nat.eq_refl (length w)).
rewrite ← Heqn in H_move at 2. clear Heqn.
generalize dependent w.
induction n as [|n IHn] using strong_induction;

```

We first introduce a new variable `length w` into Coq’s context, which only contains the variable `w`, for some arbitrary word. Next, we appeal to the reflexivity of the \leq relation to move the tautological proposition “ $n \leq n$ ” to the antecedent of our goal. Finally, we perform strong induction on the second occurrence of `n` this tautology, thus preserving the less than equal to relation between our word length and the inductive variable. This generates the following inductive hypothesis in the successor case, which gives that the conclusion holds for *all* word lengths up to $|n|$.

$$IHn : \forall m : nat, m < n \rightarrow \forall w : word, length\ w = m \rightarrow L1\ w \leftrightarrow L2\ w$$

Finite injectivity and the Curry-Howard correspondence EPR’s proof omits the rest of the argument for the finiteness of block cancellable languages with k , and directly obtains Lemma 2 from the fact that two block cancellable languages which agree on words of shorter length are equal. We first complete the informal proof with the following fact from set theory:

Theorem 4.6. *If there exists an injective mapping from some finite, inhabited set to an arbitrary set, then that set is finite.*

Proof. Let L_1, L_2 be two languages which satisfy the block cancellation property for constant k . Let L'_1, L'_2 be the subset of L_1 and L_2 containing only words shorter

than length Ramsey’s constant for k , or $r(k)$. By EPR’s claim from Lemma 2, two block cancellable languages which agree on words of shorter length are equal, therefore L_1 and L_2 are the same language, and the mapping from a block cancellable language with constant k to a language containing only words shorter than $r(k)$ is an injection. Because the set of languages containing words shorter than any constant $r(k)$ is finite with cardinality $2^{|\Sigma|^{r(k)}}$, and the mapping from the set of languages block cancellable with k to this set is an injection, by Theorem 4.6 the set of languages block cancellable with k is finite. \square

Completing the formal proof turns on the following theorem:

Theorem *inj_finite* $\{X\ Y : \text{Type}\}$:

$\forall (P : X \rightarrow \text{Prop}) (Q : Y \rightarrow \text{Prop}) (f : \{x \mid P\ x\} \rightarrow \{y \mid Q\ y\}),$

inhabited $\{x \mid P\ x\} \rightarrow$

injective $P\ Q\ f \rightarrow$

is_finite_dep $Q \rightarrow$

is_finite_dep $P.$

The predicates *inhabited* and *injective* are defined in the standard way. We again leverage the expressive power of Coq’s dependently-typed lambda calculus to capture the notion of mathematical sets using sigma-types: we define sets as list types paired with some property P that implicitly defines the set membership condition. Our *injective* function from set to set is then expressed as a function from sigma-typed list to sigma-typed list. Notably, proving *inj_finite* requires adding the axiom of functional choice into Coq’s base logic:

Definition *FunctionalChoice* : Type → Type → Prop :=

```

fun A B : Type =>
  ∀ R : A → B → Prop,
    (∀ x : A, ∃ y : B, R x y) →
      ∃ f : A → B, ∀ x : A, R x (f x).

```

Functional choice is used to obtain the inverse function of our injective function, in order to construct a list of elements satisfying P from a list of elements satisfying Q , given an injection.

We leverage the intrinsic finiteness guarantee of inductive lists to define the property of finiteness as follows:

Definition *is_finite* {X : Type} (P : X → Prop) : Prop :=

```

∃ L : list X, ∀ (x : X), In x L ↔ P x.

```

In other words, a set with membership condition P is finite iff we can explicitly construct a list such that all elements of the generic type that satisfy P , and all members of the list satisfy P . However, this notion of finiteness does not suffice for our finite injectivity theorem. The type of our witness list is of generic X rather than sigma-type X , whereas the type of our *injective* function is from sigma-type to sigma-type. Our finite injectivity theorem instead requires the following dependently-typed version of finiteness:

Definition *is_finite_dep* {X : Type} (P : X → Prop) : Prop :=

```

∃ L : list {x | P x}, ∀ (dep_x : {x | P x}), In dep_x L.

```

This commits us to proving the equivalence between these two notions of finite-

ness. While the two definitions sound identical articulated in natural language, the fundamental difference between them lies in where the propositional information capturing set membership is contained: in the dependent version, that x satisfies P is contained at the level of `Type`, whereas in the non-dependent version, that x satisfies P is contained at the level of `Prop`. The proof of equivalence elegantly showcases the Curry-Howard correspondence between proofs and programs, which allows for logical propositions to act as function arguments.

Theorem *is_finite_equiv* : $\forall \{X : \text{Type}\} (P : X \rightarrow \text{Prop}),$

$$(\exists L : \text{list } \{x \mid P x\}, \forall (dep_x : \{x \mid P x\}), \text{In } dep_x L)$$

$$\leftrightarrow (\exists L : \text{list } X, \forall (x : X), \text{In } x L \leftrightarrow P x).$$

The proof of the left direction requires us to, given a list of dependent pairs of x and $P x$, construct a list of x 'es that uniquely satisfy P . The list construction is trivially completed by mapping a projection of the first element of the dependent pair onto the list of dependent pairs:

$$\exists (\text{map } (\text{fun } x \Rightarrow \text{proj1_sig } x) L).$$

The following lemma completes the proof of the left direction:

Lemma *inj_in_map_iff* :

$$\forall \{X : \text{Type}\} (P : X \rightarrow \text{Prop}) (f : \{x \mid P x\} \rightarrow X) (L : \text{list } \{x \mid P x\})$$

$$(x : \{x \mid P x\}),$$

$$\text{In } (\text{proj1_sig } x) (\text{map } (\text{fun } x \Rightarrow \text{proj1_sig } x) L) \leftrightarrow \text{In } x L.$$

The proof of the right direction comprises one of the most technically challenging endeavors in our work. It requires us to, given a list of generic x 'es, construct a list

of dependent pairs of x and $P x$. We provide an intuition for the difficulty in doing so through a comparison with the following function:

Definition *build_dep_impl* :

```

∀ {X : Type}
  (x : X)
  (P Q : X → Prop)
  (H : ∀ x, P x ↔ Q x)
  (H_P : ∀ x : X, P x), {x | Q x}.

intros; spec H x; apply H in H_P; refine (exist Q x H_P).

```

Defined.

Fixpoint *build_dep_impl_list*

```

{X : Type}
(P Q : X → Prop)
(l : list {x : X | P x})
(H : ∀ x, P x ↔ Q x) : list {x : X | Q x} :=

match l with
| [] ⇒ []
| hd :: tl ⇒ (build_dep_impl P Q hd H) :: build_dep_impl_list P Q tl H

end.

```

The function above takes a list of dependent pairs of x 'es that satisfy P , alongside a *logical proposition* H which states that all x 'es satisfying P satisfy Q , and constructs a list of dependent pairs of x 'es that satisfy Q . This amounts to recursively traversing l , and at each head element which is a dependent pair consisting of

x and logical proposition $P x$, destructing it to obtain x , then using x to appeal to H to obtain $Q x$, and finally pairing x and $Q x$ together to make a new dependent pair which is then appended onto the return list.

In the above example, P and Q are predicates of type $X \rightarrow \text{Prop}$ whose truth values are universally guaranteed by H . For the proof of the right direction of our equivalence, P corresponds to the property of being in some list, and Q corresponds to the property characterizing the finite set we want to construct. However, the list we have to work with is not a list of dependent pairs, but rather a list of generic x 'es. The first challenge consists of not being able to directly obtain the proposition to pass to H as an implicative antecedent, as in the example above. The second challenge consists in the fact that the proposition we need to obtain is self-referential: each x can only “know” that it is in the list it is in when it is being matched as the head of the list:

Definition *in_eq* :=

```
fun (A : Type) (a : A) (l : list A) =>
```

```
  or-introl eq_refl : ∀ (A : Type) (a : A) (l : list A), In a (a :: l).
```

Therefore, heavy modifications must be made to the function above in order to construct our desired witness list. The function which accomplishes this is as follows:

```
Fixpoint build_dep_impl_list {X: Type} (P: X → Prop) (L: list X)
```

```
  (Hfn : ∀ x, In x L → P x) : list {x | P x} :=
```

```
  match L as l return (l = L → list {x | P x}) with
```

```

| nil ⇒ fun _ ⇒ nil
| hd :: tl ⇒ fun h ⇒ cons (eq_rect (hd :: tl)
                                   -
                                   (fun Hfin0 : ∀ x, In x (hd :: tl) → P x ⇒
                                   exist _ hd (Hfin0 hd (in_eq hd tl)))
                                   L h Hfin)
                                   (build_dep_impl_list P tl (rest_fin P L hd tl Hfin h))
end (eq_refl L).

```

The function uses dependent return types to include an additional argument which “remembers” that the current head of the list is indeed a member of the list by applying *in_eq* above. It then passes the proposition obtained from *in_eq* along to *Hfin* in order to obtain $P x$, where x is the current head of the list. Finally, it pairs x and $P x$ together to construct a list of dependent pairs.

The above equivalence allows us to construct the following building blocks for our *inj_finite* theorem:

P is the property of being block cancellable with some constant k :

Definition *bc_sigma* ($k : \text{block_pumping_constant}$) : *language* → Prop :=

```

fun L : language ⇒
  ∀ (s : list T),
  ∀ (bl : breakpoint_set k s),
  ∃ (i j : breakpoint bl),
  i < j ∧ (L (firstn i s ++ skipn j s) ↔ (L s)).

```

Definition *bc_language* ($k : \text{block_pumping_constant}$) : Type :=

$\{ l \mid \text{bc_sigma } k \ l \}$.

We then prove that there always exists some language that is block cancellable with k :

Lemma *inhabited_bc* : $\forall k : \text{block_pumping_constant}, \text{inhabited } (\text{bc_language } k)$.

Q is the property of only containing words shorter than some constant n .

Definition *is_short_lang* ($n : \text{nat}$) ($L : \text{language}$) : Prop :=

$\forall w : \text{word}, L \ w \rightarrow \text{length } w \leq n$.

Definition *short_lang* ($n : \text{nat}$) : Type :=

$\{ l \mid \text{is_short_lang } n \ l \}$.

We then constructively prove that short languages are finite:

Lemma *is_finite_sheared* : $\forall n : \text{nat},$

$\text{is_finite } (\text{is_short_lang } n)$.

Lemma *is_finite_sheared_dep* : $\forall n : \text{nat},$

$\text{is_finite_dep } (\text{is_short_lang } n)$.

Finally, we supply the above lemmas to *inj_finite* to obtain that the class of languages block cancellable with k is finite:

Theorem *bc_k_is_finite_dep* :

$\forall k : \text{block_pumping_constant}, \text{is_finite_dep } (\text{bc_sigma } k)$.

Theorem *bc_k_is_finite* :

$\forall k : \text{block_pumping_constant}, \text{is_finite } (\text{bc_sigma } k)$.

4.4.2 Lemma 3

The next step to showing that the block cancellation property implies regularity consists of the proof that all derivatives of block cancellable languages are also block cancellable. EPR's proof [EPR] for this lemma is brief:

Proof. Suppose that $z \in \Sigma^*$ and $z = wy_1 \dots y_k w'$. Consider $\sigma z = w' y_z \dots y_k w'$ where $w' = \sigma w$. Then, since $L \in C_k$, there exist $m, j, 1 \leq m \leq j \leq k$ such that:

$$w'' y_1 \dots y_m y_{j+1} \dots y_k w' \in L \text{ iff } \sigma z \in L.$$

But then,

$$wy_1 \dots y_m y_{j+1} \dots y_k w' \in L_\sigma \text{ iff } \sigma z \in L_\sigma.$$

Hence L_σ is also in C_k . □

The informal reasoning elides many proof obligations that are explicitly required in the formal proof, including:

- Case distinction on whether z is the empty word ϵ ;
- Case distinction on whether the derivative word label d is ϵ , in which case L_d and L are the same language;
- The proof that any set of breakpoints for a longer word is also a set of breakpoints for a shorter word.

Coq's constructive type theory requires us to explicitly construct a set of breakpoints for a word in L_x from a set of breakpoints for a word in L . In this case, given an arbitrary set of breakpoints for a word w in L , we increment each breakpoint in the

set by $|x|$, where x is the label of the derivative language. Our dependently-typed definition of breakpoint sets requires us to show that the breakpoint set predicate is preserved for w' in L_x from the breakpoint set predicate for w in L . One key proof engineering takeaway from working extensively with dependent types in Coq is that we must take care not to lose information in the process of destruction and reconstruction. In the case of this proof, when we map the plus n operation to a list of natural numbers, the list of natural numbers we obtain is uniformly greater than n . However, this information is not contained in the data type `list nat`, which only tells us that its elements are greater than or equal to 0.

4.4.3 Lemma 4

Up to this point, we have two central results about classes of block cancellable languages: that they are finite, and that they are closed under derivative. The final step to showing that the block cancellation property implies regularity consists of Lemma 4, which quantifies over properties of languages.

Lemma 4.7. *Lemma 4*

Let P be some property of languages such that (i) there are only finitely many languages that have P ; (ii) for all σ in Σ , if L has P then L_σ has P . Then P implies regularity.

An analysis of the explanatory role of each of the components in Lemma 4 render the following conclusions:

1. The role of the property P is to leverage its finiteness to supply a finite set of

states for the automaton;

2. The role of preservation over derivatives is to construct the transition relation and the acceptance condition;
3. L_0 supplies the start state, representing some arbitrary language that has property P ;
4. The final automaton can be shown to recognize all words belonging to regular languages, and by construction recognizes all words satisfying property P .

We perform a similar analysis on EPR's argument that regular languages satisfy the block pumping property, which is informally and briefly given by appealing to the pigeonhole principle. We obtain the following conclusions:

1. The pigeonhole principle compares the length of the word and the number of automaton states, implicitly, the number of breakpoints;
2. Final states and start states can be considered as regular states with a special state label;

We combine the insights from the conclusions above to formulate the following notion of regularity:

Definition *equiv_classes_pred* ($ls : list\ language$) :=

$\forall L, In\ L\ ls \rightarrow \forall w, In\ (derivative_of\ L\ w)\ ls.$

Definition *equiv_classes* : Type :=

$\{ls : list\ language \mid equiv_classes_pred\ ls\}.$

Definition *lang_class* ($LS : equiv_classes$) : Type :=

$\{ l : language \mid In\ l\ (equiv_classes_proj1\ LS) \}$.

Definition *regular* ($L : language$) : Prop :=

$\exists\ LS : equiv_classes, In\ L\ LS$.

We define equivalence classes of languages as finite lists that are closed under language derivative. We then define a language to be regular iff it belongs to an equivalence class of languages that is closed under derivative. We verify the correctness of our definition by proving that it obeys the same notion of acceptance as an automata-based definition.

We then use our definition of regularity to complete the equivalence between the block cancellation property and regularity. The proof illustrates how our characterization of regularity screens off the representational redundancy in the definition of a finite automaton.

Proof. Because L is regular, there exists a list of languages LS that is closed under derivative, meaning that for every language in the list, all of its derivatives are also in the list. We supply the length of LS to be the pumping constant k . We then introduce an arbitrary word w , and do case distinction on the length of the word. In the case that w is shorter than k , then it is trivially block pumpable, because one cannot insert $k + 1$ breakpoints into a word that is shorter than k . In the case that the word is longer than k , we obtain an arbitrary set of breakpoints of size $k + 1$. Because breakpoints correspond to positions within the word, we can obtain a list of prefixes of w corresponding to mapping the first n symbols in the word to the list of

breakpoints. By the definition of derivative languages, each prefix w' is recognized by some derivative $L_{w'}$ with respect to L . Therefore, there are a total of $k + 1$ derivatives of L which recognize $k + 1$ prefixes. Further, we know that each of these derivatives occur in the finite list of equivalence classes which defines the property of regularity for L – this information is carried in the dependent type signature of LS . However, because the length of the finite number of derivatives is k , by the pigeonhole principle we know that there must exist two derivatives which are the same. We can destruct the existential witnesses for the indices of two breakpoints which mark two prefixes recognized by the same derivative. We then pass these indices to the two breakpoint existentials. Finally, the proof is completed by the application simple algebraic properties of derivatives. \square

A further advantage of this notion of regularity is that it naturally gives rise to an easy proof that regularity implies block pumpability. The reasoning process is very similar, and the central idea is that it reduces state repetition and the pigeonhole principle over states to simple algebraic manipulation of language derivatives. The manipulation of derivatives in the argument relies only on the following four facts about derivatives:

Lemma *chomp_deriv* : $\forall (w : \text{word}) (L : \text{language}),$

$$L w = (\text{derivative_of } L w) [].$$

Lemma *cat_deriv* : $\forall (x y w : \text{word}) (L : \text{language}),$

$$(\text{derivative_of } (\text{derivative_of } L x) y) w = (\text{derivative_of } L (x ++ y)) w.$$

Lemma *cat_cat_deriv* : $\forall (x y z w : \text{word}) (L : \text{language}),$

$$\begin{aligned}
& (\text{derivative_of } (\text{derivative_of } L \ x) \ (y \ ++ \ z)) \ w = \\
& (\text{derivative_of } (\text{derivative_of } L \ (x \ ++ \ y)) \ z) \ w.
\end{aligned}$$

We further press our definition for concision by using it to complete the proof triangle, in particular, to prove that regular languages are block pumpable.

This section has shown how an alternative formulation of regularity has allowed us to avoid the costly representational overhead of defining finite automata, and make the proof by pigeonhole principle more direct and elegant. To the best of our knowledge, our definition is novel in its utilization of the notion of finiteness in defining regularity. It also showcases well the benefits of working in a formal proof environment: being forced to think like a representational minimalist also forces one to refine their definitions and arguments until they admit no further interrogation.

4.5 Closure properties

The proofs of closure properties for block pumpable languages under intersection, union and concatenation fall upon constructing suitable witnesses for the block pumping constant, and suitable witnesses for the two breakpoints. The former relies on Ramsey’s Theorem, and the latter relies on arguing that the breakpoint sets of two languages L and H can be used to construct a breakpoint set for $L \cap H$, $L \cup H$ and $L \cdot H$ respectively. We follow the proofs of closure properties in [chak], which are generally free from significant proof omissions. The proof engineering required to construct the formal proofs are similar to what has been described above. Therefore, in the interest of brevity, we omit detailed descriptions of these formal proofs and instead refer the reader to `closure.v` of the Coq development.

5 Discussion

Our initial point of departure for formally investigating block pumpable language theory stemmed from a broader curiosity about whether properties that are true of some mathematical object are provable on all possible representations of that object. In particular, we wondered whether a proof that regular languages satisfy the positive block pumping property which proceeds on direct induction over regular expressions was possible. We have yet to find an affirmative answer to this question, as it seems like we must go from regular expressions to finite automata via Kleene’s theorem, and then argue using the pigeonhole principle on the number of states in the finite automaton, in order to complete the proof.

Another point of interest which arose throughout the course of our formalization effort concerns the constructive vs. classical divide in formalizing mathematics. In our development, we used classical axioms as a natural way to formalize set theory. However, we wonder whether the axiom of choice is strictly necessary for the specific purpose of completing the proof of block cancellable to regular. At initial glance, forgoing the axiom of choice entails coming up with a terminating algorithm that computes a block cancellable language from a language that contains only short strings.

Finally, our extensive engagement with Coq’s dependent types gives rise to the following question: how can one formally characterize the expressive power of dependent types in Coq’s metalogic? Our usage of dependent types is restricted to sigma-types, which consist of an element and a `Prop`. If we assume that all of our

propositions are decidable, is carrying the propositional information inside `Type` provably equivalent to carrying the propositional information inside `Prop`, in the form of an implicative antecedent? How does the picture change when we expand our consideration to dependent types which consist of an element and a `Type` instead of `Prop`?

In addition to investigating the questions listed in the section above, we plan to further our development by including formal results of Jaffe [[jaffe](#)] and Varricchio's [[varricchio](#)] necessary and sufficient pumping conditions for regularity.

6 Conclusion

We present the first known formalization of results from automata theory about languages orthogonal to the Chomsky hierarchy. In constructing formal proofs of the equivalence between the block pumping property, the block cancellation property and regularity, as well as closure properties of positively block pumpable languages, we make use of various features of Coq's higher-order, dependently-typed lambda calculus to define mathematical objects and reason classically about sets. Our process instances an interesting contravariance between the concerns of informal and formal mathematical proofs: the bulk of the informal reasoning can be reduced to algebraic manipulation of inductively defined data structures that safely reside within the Calculus of Inductive Constructions; precisely where the informal proof ends does the formal proof take off, eliciting a departure from constructive type theory to the realm of classical logic.

7 References

A Coq project file organization

The Coq development consists of the following files, and can be found at <https://github.com/rooibosri>

- `stronginduction.v` contains a proof of the strong induction principle from the inductive definition of natural numbers and its standard induction principle;
- `finite.v` contains a formal theory of finite sets in two flavors, regular and dependent types, including the foundational result that finite sets map injectively onto finite sets, and a proof of equivalence between the two versions of finiteness;
- `lib.v` contains a large collection of helper lemmas which supplement `Nat` and `List` in Coq's standard library;
- `definitions.v` includes all of the inductively defined data types required of block pumpable language theory, and basic facts about derivatives;
- `blockpump.v` contains a library of lemmas about breakpoint sets, and the proof of EPR's Lemma 2 and Lemma 3;
- `regular.v` contains our alternative characterization of regularity, and the proof from regular to block cancellable, regular to block pumpable, and regular to block cancellable;
- `sixwayinn.v` contains all six proof arrows in the commutative triangle between regularity, block pumpability and block cancellability

- `dfa.v` contains obsolete definitions of deterministic finite automata;
- `closure.v` contains proofs of the closure properties of block pumpable languages under intersection, union and concatenation.

B Additional definitions

Regular expressions are inductively defined as follows ¹:

```
Inductive reg_exp : Type :=  
| EmptySet : reg_exp  
| EmptyStr : reg_exp  
| Char : T → reg_exp  
| App : reg_exp → reg_exp → reg_exp  
| Union : reg_exp → reg_exp → reg_exp  
| Star : reg_exp → reg_exp .
```

The matching relation between regular expressions and words is defined inductively as follows:

```
Inductive exp_match : word → reg_exp → Prop :=  
| MEmpty : exp_match [] EmptyStr  
| MChar : ∀ x, exp_match [x] (Char x)  
| MApp : ∀ s1 re1 s2 re2,  
    exp_match s1 re1 →  
    exp_match s2 re2 →  
    exp_match (s1 ++ s2) (App re1 re2)  
| MUnionL : ∀ s1 re1 re2,  
    exp_match s1 re1 →
```

¹The definition of regular expressions is taken from chapter `IndProp.v` of Benjamin Pierce et al.'s textbook `Software Foundations` [sf].

$$\begin{aligned}
& \text{exp_match } s1 \text{ (Union } re1 \text{ } re2) \\
| \text{ MUnionR : } \forall re1 \ s2 \ re2, \\
& \quad \text{exp_match } s2 \ re2 \rightarrow \\
& \quad \text{exp_match } s2 \text{ (Union } re1 \ re2) \\
| \text{ MStar0 : } \forall re, \text{ exp_match [] (Star } re) \\
| \text{ MStarApp : } \forall s1 \ s2 \ re, \\
& \quad \text{exp_match } s1 \ re \rightarrow \\
& \quad \text{exp_match } s2 \text{ (Star } re) \rightarrow \\
& \quad \text{exp_match (s1 ++ s2) (Star } re).
\end{aligned}$$

Coq automatically equips all inductive types with an induction principle. The induction principle for regular expression matching is as follows:

Definition *exp_match_ind* :=

$$\begin{aligned}
& \forall P : \text{word} \rightarrow \text{reg_exp} \rightarrow \text{Prop}, \\
& \quad P \text{ [] EmptyStr} \rightarrow \\
& \quad (\forall x : T, P [x] (\text{Char } x)) \rightarrow \\
& \quad (\forall (s1 : \text{word}) (re1 : \text{reg_exp}) (s2 : \text{word}) (re2 : \text{reg_exp}), \\
& \quad \quad s1 =^{\sim} re1 \rightarrow P s1 re1 \rightarrow s2 =^{\sim} re2 \rightarrow P s2 re2 \rightarrow P (s1 ++ s2) (\text{App} \\
& \quad \quad re1 \ re2)) \rightarrow \\
& \quad (\forall (s1 : \text{word}) (re1 \ re2 : \text{reg_exp}), \\
& \quad \quad s1 =^{\sim} re1 \rightarrow P s1 re1 \rightarrow P s1 (\text{Union } re1 \ re2)) \rightarrow \\
& \quad (\forall (re1 : \text{reg_exp}) (s2 : \text{word}) (re2 : \text{reg_exp}), \\
& \quad \quad s2 =^{\sim} re2 \rightarrow P s2 re2 \rightarrow P s2 (\text{Union } re1 \ re2)) \rightarrow \\
& \quad (\forall re : \text{reg_exp}, P \text{ [] (Star } re)) \rightarrow
\end{aligned}$$

$(\forall (s1\ s2 : \text{word}) (re : \text{reg_exp}),$

$s1 =^{\sim} re \rightarrow$

$P\ s1\ re \rightarrow s2 =^{\sim} \text{Star}\ re \rightarrow P\ s2\ (\text{Star}\ re) \rightarrow P\ (s1\ ++\ s2)\ (\text{Star}\ re))$

\rightarrow

$\forall (w : \text{word}) (r : \text{reg_exp}), w =^{\sim} r \rightarrow P\ w\ r.$

Deterministic finite automata are defined as follows:

Definition $run\ (s : \text{dfa_states}) : \text{Type} := \text{list}\ (\text{state}\ s \times T).$

Record $dfa : \text{Type} :=$

$DFA\ \{\ \text{set} : \text{dfa_states};$

$s0 : \text{state}\ \text{set};$

$f : \text{transition}\ \text{set}\ \}.$

Definition $accepting' (w : \text{word}) (dfa : \text{dfa}) :=$

$\exists r : \text{run}\ (\text{set}\ dfa),$

$w = (\text{map}\ \text{snd}\ r)\ ++\ [\text{epsilon}]$

$\wedge\ \text{hd}\ \text{bogus}\ (\text{map}\ \text{fst}\ r) = (s0\ dfa)$

$\wedge\ (\forall i : \text{nat}, 0 \leq i < \text{length}\ w - 1 \rightarrow$

$((f\ dfa)\ (\text{nth}\ i\ (\text{map}\ \text{fst}\ r)\ \text{bogus})$

$(\text{nth}\ i\ (\text{map}\ \text{snd}\ r)\ \text{epsilon})$

$(\text{nth}\ (S\ i)\ (\text{map}\ \text{fst}\ r)\ \text{bogus})))$

$\wedge\ \text{state_proj1_snd}\ (\text{fst}\ (\text{last}\ r\ (\text{bogus}, \text{epsilon}))) = \text{true}.$