Pumping, With or Without Choice[⋆]

Aquinas Hobor[1 2], Elaine Li[1✉], and Frank Stephan[2 3]

[1] Yale-NUS College
elaine.li@u.yale-nus.edu.sg
[2] School of Computing, National University of Singapore
{hobor, fstephan}@comp.nus.edu.sg
[3] Department of Mathematics, National University of Singapore

Abstract. We present the first machine-checked formalization of Jaffe and Ehrenfeucht, Parikh and Rozenberg's (EPR) pumping lemmas in the Coq proof assistant. We formulate regularity in terms of finite derivatives, and prove that both Jaffe's pumping property and EPR's block pumping property precisely characterize regularity. We illuminate EPR's classical proof that the block cancellation property implies regularity, and discover that—as best we can tell—their proof relies on the Axiom of Choice. We provide a new proof which eliminates the use of Choice. We explicitly construct a function which computes block cancelable languages from well-formed short languages.

Keywords: Pumping lemmas · Axiom of Choice · Coq

1 Overview

Pumping properties of formal languages have a rich history. Rabin and Scott provided a pumping lemma that displays properties of regular languages; Bar-Hilel, Perles, and Shamir did the same for context-free languages [17, 24]. Pumping lemmas describe how words belonging to the relevant language L can be "pumped", *e.g.* if L is regular then a word $u \in L$ can be split into parts xyz ($vwxyz$ in the context-free case) such that for all $n \in \mathbb{N}$, $xy^nz \in L$ (respectively $vw^nxy^nz \in L$). Because pumping lemmas are often stated as *necessary* conditions of a language being regular or context-free, they are often used in modus tollens form to show that certain languages are *not* regular or context-free because they do not satisfy the pumping property. Pumping lemmas are also used to prove other properties of regular or context-free languages, *e.g.* that every context-free language over the unary alphabet $\{0\}$ is regular and that every automatic function increases the length of its input by at most a constant.

The converse question of whether pumping properties can also serve as *sufficient* conditions for a language to be regular or context-free is less straightforward. Sommerhalder [29] showed that even a more restrictive "matching" form

[⋆] Aquinas Hobor is supported in part by Yale-NUS College grant R-607-265-322-121. Elaine Li is supported in part by Runtime Verification, Inc. Frank Stephan is supported in part by MOE AcRF Tier 2 grant MOE2016-T2-1-019 / R146-000-234-112. Authors are ordered alphabetically.

of Rabin-Scott's pumping lemma—that both a language *and its complement* satisfy the pumping property—fails to precisely characterize regular languages because there are non-regular languages that satisfy it. Jaffe [18] was the first to provide a pumping lemma that precisely characterizes regularity, *i.e.* that gives *both* a necessary and a sufficient condition for regularity.

1.1 Jaffe's Pumping Lemma

Theorem 1 (Jaffe). *A language L is regular iff there is a constant k s.t.*

$$\begin{aligned}&\forall x \in \Sigma^*.\ |x| = k \ \Rightarrow\ \exists u, v, w \in \Sigma^*.\\ &\quad x = uvw \ \wedge\ v \neq \epsilon \ \wedge\ \forall h \in \mathbb{N},\ z \in \Sigma^*.\ (uvwz \in L \ \Leftrightarrow\ uv^hwz \in L)\end{aligned}$$

Jaffe's pumping lemma is a reformulation of the Myhill-Nerode theorem [23]. The pumping constant k in Jaffe's pumping lemma refers to the length of word prefixes, equivalently the length of the language's derivative labels.

Definition 1 (Derivative). *The derivative of a language L with respect to a word $x \in \Sigma^*$, written L_x, is another language that accepts words y iff L accepts xy,* i.e. $xy \in L \Leftrightarrow y \in L_x$.

Theorem 2 (Myhill-Nerode). *A language L is regular iff it has a finite number of derivatives.*

1.2 The Block Pumping Lemma

Ehrenfeucht, Parikh and Rozenberg [14] provided a pumping property that gives a more sophisticated characterization of the regular languages.

Theorem 3 (EPR). *A language L is regular iff there is a constant k s.t. for any splitting of a word x into $k+2$ blocks,* i.e. $x = w, u_1, \cdots, u_k, w'$, *one can find a an interval $u_i \cdots u_j$ of blocks that can be pumped any number h of times:*

$$\begin{aligned}&\forall x, w, u_1, \cdots, u_k, w' \in \Sigma^*.\ x = wu_1 \cdots u_k w' \ \Rightarrow\\ &\quad \exists i, j \in \mathbb{N}.\ 1 \leq i < j \leq k \ \wedge\ \forall h \in \mathbb{N},\\ &\qquad wu_1 \cdots u_{i+1} \cdots u_j \cdots u_k w' \ \in L \ \Leftrightarrow\ w \cdots (u_{i+1} \cdots u_j)^h \cdots u_k w' \in L\end{aligned}$$

We call languages that satisfy EPR's "block pumping" property "block pumpable languages", and we write "L is block pumpable with k" to specify the block pumping constant. Furthermore, one can postulate that $u_1, \cdots, u_k$ are nonempty without changing the notion of block pumpable.

An advantage of block pumping over Rabin-Scott pumping is that it allows one to directly obtain block pumping constants for combined languages such as $L \cap H$, $L \cup H$ and $L \cdot H$ from the constants for L and H, as we will show in §3. Rabin-Scott pumping does not allow this: *e.g.* regular languages $L = \{0^k 1^n 2^m : k = 1 \Rightarrow (n = m \bmod h)\}$ and $H = \{0\} \cdot \{1\}^* \cdot \{2\}^*$ both have pumping constant 2, but $L \cap H$ requires pumping constant $h+1$, which is independent of the pumping constants of L and H.

To compare EPR's pumping property with Rabin-Scott's, Chak *et al.* [4] investigated languages that satisfy the block pumping property with the $\Leftrightarrow$ restricted to the $\Rightarrow$ direction only:

Definition 2 (One-sided block pumpable language). *A language L is one-sided block pumpable iff there is a constant k s.t.*

$$\begin{aligned}&\forall x \in \Sigma^*.\ x \in L \ \Rightarrow\ \forall w, u_1 \cdots u_k, w' \in \Sigma^*.\ x = w u_1 \cdots u_k w' \ \Rightarrow \\ &\quad \exists i, j \in \mathbb{N}.\ 1 \leq i < j \leq k \ \wedge\ \forall h \in \mathbb{N},\ w \cdots (u_{i+1} \cdots u_j)^h \cdots u_k w' \in L\end{aligned}$$

Chak *et al.* [4] showed that one-sided block pumpable languages not only need not be regular, but need not even be computable! Accordingly, these languages cannot be reasoned about in the same automata-theoretic way as other languages in the Chomsky hierarchy. Instead, proofs about one-sided block pumpable languages have a distinctly combinatorial flavor, relying critically on Ramsey theory.

1.3 Contributions

We present the first machine-checked proofs of the pumping lemmas of Jaffe and Ehrenfeucht, Parikh and Rozenberg in the Coq proof assistant. Jaffe's pumping lemma is straightforward to mechanize, but we present it nonetheless as a way to introduce the novelties of our setup. In particular, we use Myhill-Nerode to *define* the regularity of a language as having finitely many derivatives. We then present machine-checked proofs from block pumpable language theory that, to the best of our knowledge, are the first of formal language classes orthogonal to the Chomsky hierarchy. We introduce relevant definitions by presenting a mechanization of the closure properties of one-sided block pumpable languages: they are closed under intersection, union and concatenation [4, Thm 15].

We then proceed with EPR's more complex pumping lemma. This complexity is in part due to some omissions concerning the concept of "finiteness". We fill in the gaps of their proof and discover that it appears to require the Axiom of Choice to construct the inverse to a partial injective function.

Although not diehard constructivists, we find the Axiom of Choice a bit objectionable. One well-known consequence is the Banach-Tarski paradox [2]:

> Given a solid 3-D ball, one can decompose ("cut") it into five disjoint subsets ("pieces"), which can be reassembled using rigid motions (movements and rotations) to yield two identical copies of the original ball.

Coq offers a variety of flavors of the Axiom of Choice, but their use leads to the unfortunate (full or partial) collapse of the distinction between the set of mathematically true facts (`Prop`) and computationally decidable facts (`Type`) [3].

Accordingly, we present a new proof of EPR's pumping lemma that eliminates the Axiom of Choice by explicitly constructing the inverse function in question. This inverse function can compute block cancelable languages from well-formed input languages. The rest of this paper is organized as follows:

§2 We present our basic setup and prove Jaffe's pumping lemma.
§3 We define block pumpable languages and prove closure properties.
§4 We mechanize the original proof of EPR's pumping lemma. In the process we clarify several areas of the proof, in particular its treatment of finiteness. We show how EPR's proof uses the Axiom of Choice.

§5 We present our construction of an explicit inverse function and prove that it can enable a new choice-free proof of EPR's pumping lemma.

§6 We discuss related work before concluding in §7.

Along the way, we highlight aspects of our formalization which leverage features of Coq's type theory and/or contribute broadly applicable definitions and proofs for which we could not find existing alternatives. The present work includes results from the Capstone project of Li [21]. Our proofs are entirely machine-checked in Coq and available at

`https://github.com/atufchoice/blockpump`

2 Regularity and Jaffe's pumping lemma

Here we present the first mechanization of Jaffe's pumping lemma, and with it the basics of our formal setup. We begin with the axioms we add to CiC:

1. Functional extensionality: $\big(\forall x.\ f(x) = g(x)\big) \Rightarrow (f = g)$
2. Propositional extensionality: $(P \Leftrightarrow Q) \Rightarrow (P = Q)$
3. Law of excluded middle: $P \vee \neg P$
4. Functional choice: $(\forall a.\ \exists b.\ aRb) \Rightarrow \big(\exists f.\ \forall a.\ aR(f(a))\big)$

Specifically: Jaffe (§2), EPR's original proof (§4), and our new EPR proof (§5) use functional and propositional extensionalities to prove language equivalence and (via proof irrelevance) equality on dependent types. Proofs about block pumping, *i.e.* closure properties for block pumpable languages (§3), EPR's original proof (§4) and our new Choice-free proof (§5) use the law of excluded middle due to the fact that block pumpable languages are not Turing-decidable, and as a result, we cannot check language membership computationally. Lastly, EPR's original proof (§4) uses functional choice.

We next present the basic mathematical definitions of alphabets, words, and languages. We use Σ to refer to a finite alphabet, and σ to refer to symbols in the alphabet. For simplicity in Coq, we use a three-letter alphabet (type `T`):

```
Inductive T : Type := aa | bb | cc.
```

We use variables x, y, z, w, v to denote words; $|w|$ to denote the length of w; and the symbol ϵ to denote the empty word. In Coq, words are just lists of letters:

```
Definition word := list T.
```

We use L, H to denote languages, *i.e.* sets of words:

```
Definition language : Type := word -> Prop.
```

We use functional and propositional extensionality to prove language equality:

```
Lemma language_equality : forall (l1 l2: language),
  l1 = l2 <-> forall (w: word), l1 w <-> l2 w.
```

Instead of using the standard representations of regular languages, *i.e.* finite automata or regular expressions, we use Myhill-Nerode's Theorem *definitionally* to represent regular languages as languages with finitely many derivatives. We write L_x to denote the derivative of a language with respect to a word x (sometimes L_σ with a single alphabet symbol), *i.e.*:

```
Definition derivative_of (L: language) (x: word) : language :=
  fun w => L (x ++ w).
```

We say that L_x is a derivative of L when there exists a derivative label x such that for all words w, L_x accepts w iff L accepts xw.

```
Definition is_deriv (L L_x: language) : Prop :=
  exists (x: word), forall (w: word), L_x w <-> L (x ++ w).
```

We leverage Coq's inductively defined lists to express the finiteness of a property in terms of the existence of a list of elements satisfying that property:

```
Definition is_finite {X: Type} (P: X->Prop) : Prop :=
  exists (L : list X), forall (x: X), In x L <-> P x.
```

A language L is regular iff it has finitely many derivatives:

```
Definition regular (L: language) : Prop :=
  is_finite (is_deriv L).
```

We define regularity in this way because 1) Coq formalizations of regular expressions, finite automata and their equivalence already exist [12, 13, 15, 9]; and 2) Jaffe's and EPR's proofs critically rely on the exact notion of finiteness captured in our definition. Sometimes, for proof engineering purposes we use a dependently-typed notion of finiteness as follows:

```
Definition is_finite_dep {X: Type} (P: X->Prop) :=
 exists (L: list {x | P x}), forall (dep_x : {x | P x}), In dep_x L.
```

The following equivalence lets us use one or the other as locally convenient.

```
Lemma is_finite_equiv : forall {X: Type} (P: X->Prop),
 is_finite_dep P <-> is_finite P.
```

2.1 Jaffe's Pumping Lemma

Jaffe provides the following necessary and sufficient condition for regularity. The `napp` function performs word concatenation: `napp h v` is equivalent to v^h, or v concatenated to itself h times.

```
Definition jaffe_pumpable_with (k: nat) (L: language) :=
  forall (y: word), length y = k ->
    exists (u v w: word),
      y = u ++ v ++ w /\ v <> [] /\
        forall (h: nat) (z: word),
          L (u ++ napp h v ++ w ++ z) <-> L (y ++ z).
```

Jaffe's pumping lemma amounts to proving two theorems, the first of which is:

```
Theorem reg_to_jaffe : forall (L: language),
  regular L -> exists (k: nat), jaffe_pumpable_with k L.
```

From regularity we have a finite list of derivatives LD, from which we obtain the pumping constant $|\texttt{LD}| + 1$. Given y, we construct a list of derivatives of length $|\texttt{LD}| + 1$, which by the pigeonhole principle must contain a repeated derivative. We then split y based on the two prefixes of the repeated derivative language such that they correspond to u and u++v respectively.

The second theorem is the converse:

```
Theorem jaffe_to_reg : forall (k: nat) (L: language),
  jaffe_pumpable_with k L -> regular L.
```

The following helper lemma is required to prove the converse direction, and captures the central intuition of Jaffe's pumping lemma: for any Jaffe-pumpable with k language, every derivative is equivalent to some derivative with label length shorter than k. This lemma is proven via strong induction on $|\texttt{x}|$.

```
Lemma jaffe_helper : forall (k: nat) (L: language),
  jaffe_pumpable_with k L -> forall x, exists v,
    length v <= k /\ derivative_of L x = derivative_of L v.
```

We now prove that Jaffe's pumping condition implies regularity, *i.e.* that we can construct a list of finitely many derivative languages for L. The list LD we construct is the list of derivatives labeled by all words up to length k, where k is Jaffe's pumping constant. Proving that every language in LD is a derivative of L is direct by definition. Proving that every derivative L_x of L is in LD requires jaffe_helper and case analysis on the derivative label's length, *i.e.* $|\texttt{x}|$. When $|\texttt{x}| \leq \texttt{k}$, L_x is in LD by construction; when $|\texttt{x}| > \texttt{k}$, by jaffe_helper it is equivalent to some derivative whose label is shorter than or equal to k, which is in LD by construction.

3 Block pumpable languages and their closure properties

In this section we define the block pumping and block cancellation properties, and prove that one-sided block pumpable languages are closed under union, intersection, and concatenation.

We use i, j to denote natural numbers and bp_1, bp_2 to denote breakpoints, *i.e.* indices into a word w. We define word parts in terms of indices of type nat rather than subwords of type list to allow Coq's omega tactic to automatically discharge associated proof goals. Pumping the empty word leaves it unchanged, so we can assume $|w| \geq 1$ and at least two possible breakpoints (0 and $|w|$). We use k to denote block pumping or cancellation constants. Therefore, k is at least 2, breakpoint sets are lists of k increasing, within-bounds indices into a word, and breakpoints are members of such lists. We leverage Coq's dependent types to track these technicalities as follows:

```
Definition block_pumping_constant := {p: nat | p >= 2}.
Definition breakpoint_set (k: block_pumping_constant) (w: word)
```

```
:= {bl: list nat | length bl = k
                /\ increasing bl
                /\ last bl d <= length w}.
Definition breakpoint {k: block_pumping_constant} {w: word}
  (bl: breakpoint_set k w) := {i: nat | In i bl}.
```

Recall from (§1, Theorem 3) EPR's block pumping property, *i.e.* in Coq:

```
Definition block_pumpable_matching_with (k: block_pumping_constant)
                                        (L: language) :=
  forall (w: word) (bl: breakpoint_set k w),
      exists (i j: breakpoint bl), i < j /\
        forall (m: nat),
        L w <->
        L (firstn i w++napp m (pumpable_block i j w)++skipn j w).
```

Here `firstn`, `napp`, `pumpable_block`, and `skipn` build the pumped word. EPR [14] also established a variant of the block pumping property called the block cancellation property: rather than repeating the word, we omit it. The last two lines of `block_pumpable_matching_with` are replaced with:

```
        L w <-> L (firstn i w++skipn j w).
```

While the one-sided block cancellation property is weaker than its pumping counterpart, the (two-sided) block cancellation and pumping properties are equivalent. Indeed, the block cancellation property plays a critical role in both EPR's theorem in §4, and the construction of our new Choice-free proof in §5.

3.1 Ramsey theory

Proofs about block pumpable languages use Ramsey's theorem [28], a foundational result in combinatorics. Ramsey's theorem is typically stated on graphs.

Theorem 4 (Ramsey's theorem for graphs). *One can always find monochromatic cliques in any edge-coloring of a sufficiently large complete graph.*

We re-express Ramsey's theorem in terms of sets by representing vertices as elements of some set and edges as pairs of elements in the set as follows:

Theorem 5 (Ramsey's theorem for sets). *For every natural number k and finite set of colors Q, 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 then formalize two-color Ramsey's theorem for sets in Coq using `bool` to represent two distinct colors:

```
Theorem Ramsey_single :
 forall (k: nat), k > 0 ->
  exists (rk: nat), rk >= k /\
   forall (l: list nat), length l = rk ->
   forall (f: nat -> nat -> bool),
     (exists (bl: list nat), length bl = k /\ subseq bl l /\
      forall (i j: nat), i < j < k ->
      f (nth i bl d) (nth j bl d) = true)
\/   (exists (bl: list nat), length bl = k /\ subseq bl l /\
      forall (i j: nat), i < j < k ->
      f (nth i bl d) (nth j bl d) = false).
```

We further specialize Ramsey to block pumping as follows:

```
Theorem Ramsey_single_prop :
 forall (k: block_pumping_constant),
  exists (rk: block_pumping_constant), rk >= k /\
   forall (w: word) (bps: breakpoint_set rk w)
          (P: nat -> nat -> Prop),
     exists (bps': breakpoint_set k w),
       sublist bps' bps /\
   ((forall (bp1 bp2: breakpoint bps'), bp1<bp2 -> (P bp1 bp2))
\/  (forall (bp1 bp2: breakpoint bps'), bp1<bp2 -> ~(P bp1 bp2))).
```

We use “Ramsey’s constant” to refer to the existential witness $r(k)$ dependent on k given by Ramsey’s theorem. Instead of a computable two-element coloring function, we use an arbitrary predicate `P`; for this reason the proof of this formulation of Ramsey requires LEM.

3.2 Closure properties of one-sided block pumpable languages

We next formalize the results from [4] that one-sided block pumpable languages are closed under union, intersection and concatenation. The proofs turn on finding the right block pumping constant for the combined language. We present the definitions for combined languages, and refer the reader to our Coq development for statements of the closure properties. We use `l1`, `l2` and `k1`, `k2` to denote two languages and their pumping constants.

For `union_lang l1 l2`, the new pumping constant is `max k1 k2`. The proof follows directly from case analysis on whether the word is in `l1` or in `l2`, and applying the block pumping property for the respective language.

For `intersection_lang l1 l2`, the new pumping constant is Ramsey’s constant `rk` for `max k1 k2`. We know from Ramsey’s theorem that every breakpoint set of size `rk` contains a subset `bl` of size `k` such that either all the breakpoint pairs form pumps for `w` into `l1` or they do not. By the one-sided block pumping property for `l1` and `l2` we know that all breakpoint sets of size `k` contain *one* pair of breakpoints which form pumps for `w` into `l1` and `l2` respectively. In the case that `bl` contains all pumps for `w` into `l1`, we apply the one-sided block pumping property for `l2` to obtain a pair of pumps for both `l1` and `l2`. Otherwise, we find a contradiction.

For `concat_lang l1 l2`, the new pumping constant is `k1+k2`. For any word `w=w1++w2`, either all of `w1`’s breakpoints are in itself, in which case we pump `w1`, or all of `w2`’s breakpoints are in itself, in which case we pump `w2`. The proof proceeds by case analysis on the above two possibilities.

4 Ehrenfeucht, Parikh and Rozenberg's pumping lemma

Having presented all the relevant formal definitions, we move on to EPR's pumping lemma. EPR's pumping lemma states the following equivalence:

Theorem 6 (EPR's pumping lemma). *The block pumping property, the block cancellation property and regularity are equivalent.*

EPR's pumping lemma amounts to the commutative triangle in Figure 1. The equivalence can be shown by proving either the clockwise or counterclockwise direction of the triangle. EPR choose regular → block pumping property → block cancellation property. We call languages that satisfy the block cancellation property (respectively the block pumping property) with pumping constant k "$BC(k)$ languages" (respectively "$BP(k)$ languages"). Of the three arrows, showing that the block cancellation property implies regularity (6) is by far the most difficult and involved.

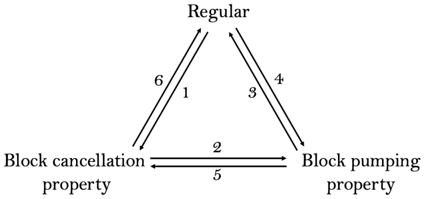

Fig. 1: The EPR commutative triangle

Lemma 1 (EPR's Lemma 1). *Block cancellation property implies regularity.*

EPR splits this proof into three sub-lemmas:

Lemma 2 (EPR's Lemma 2). *$BC(k)$ languages are finite.*

Lemma 3 (EPR's Lemma 3). *If a language is $BC(k)$, so are all of its derivatives.*

Lemma 4 (EPR's Lemma 4). *Let P be some property of languages such that (i) there are only finitely many languages that P, and (ii) forall σ in Σ, if L has P then L_σ has P. Then P implies regularity.*

EPR's Lemma 4 follows directly from Myhill-Nerode [23], and Lemma 3 is straightforward. On the other hand, EPR's proof of Lemma 2 is a bit tricky to pin down. In [14], EPR claim that the following is sufficient to show that $BC(k)$ languages are finite.

Lemma 5 (EPR's Lemma 2-ish). *Two $BC(k)$ languages that agree on words shorter than $r(k)$, where $r(k)$ is Ramsey's constant, are equal.*

EPR's proof explains how to prove this lemma, but does **not** explain why it is sufficient, *i.e.* why it implies the finiteness of block cancelable languages. We complete EPR's proof by re-interpreting Lemma 2-ish as follows, and then using a classical set-theoretic fact about finiteness and injectivity which uses Lemma 2-ish to obtain the finiteness of block cancelable languages[4].

[4] In our Coq development, we define `block_cancellable_matching_with` as the two-sided cancellation property, *i.e.* both L and L's complement satisfy it, while `block_cancellable_with` refers to the one-sided cancellation (pumping) property. The same applies for the block pumping property.

```
Definition is_short_lang (n: nat) (L: language) : Prop :=
  forall (w: word), L w -> length w <= n.
Definition filter_shortlang (n: nat) : language -> language :=
  fun L: language => (fun w: word => L w /\ length w <= n).
Theorem real_injectivity : forall (k: block_pumping_constant),
  exists (rk: block_pumping_constant),
  injective (block_cancellable_matching_with k)
            (is_short_lang rk)(filter_shortbclang k rk).
```

The definition of `injective` is standard and taken from Coq's `Logic` library, while `block_cancellable_matching_with` and `is_short_lang rk` describe the properties of the domain and codomain, *i.e.* block cancelable languages and short languages. The `filter_shortbclang` function is a dependently-typed version of the simpler `filter_shortlang` shown above which transforms languages in the domain into languages in the codomain by "shearing" off the long words. Thus, in set-theoretic terms, our formulation of EPR's Lemma 2-ish states that the shearing of a $BC(k)$ language down to a language containing only "short" words of length less than $r(k)$, where $r(k)$ is Ramsey's constant, is injective. Next, we use the following set-theoretic fact:

Lemma 6. *Every injective mapping onto some finite set is from a finite set.*

```
Theorem inj_finite {X Y: Type} :
  forall (P: X->Prop) (Q: Y->Prop) (f: {x | P x}->{y | Q y}),
    inhabited {x | P x} -> injective P Q f -> is_finite_dep Q ->
      is_finite_dep P.
```

We know that the function which shears a block cancelable language down to short words is injective from EPR's 2-ish. We easily know that short languages containing length-bounded words are finite: there are exactly $2^{|\Sigma|^m}$ many of them, where $|\Sigma|$ is the size of the alphabet, and m is the length bound on words. Therefore, by the above fact, we know that block cancelable languages are finite.

We instantiate P with `block_cancellable_matching_with k`, Q with `is_short_lang` and f with our dependently-typed length-shearing function. We additionally prove that there is at least one $BC(k)$ language:

```
Lemma inhabited_bc : forall k : block_pumping_constant,
  inhabited (bc_language k).
```

This allows us to finally show that $BC(k)$ languages are finite:

```
Theorem bc_k_is_finite_dep: forall k : block_pumping_constant,
  is_finite_dep (block_cancellable_matching_with k).
```

Digression on the Axiom of Choice. The theorem `inj_finite` is classical because constructing a finite set from another finite set requires an inverse function of an injective function, which is constructed via the Axiom of Choice. As mentioned in §2, we use functional choice (`FunctionalChoice_on` from `Coq.Logic.ChoiceFacts`). It is conceivable that this is a little stronger than is required. While we cannot use constructive versions of Choice because the type of the domain is uncountable (sets of sets of words), it is plausible that with some additional gyrations we might be able to use the weaker Axiom of Description, *a.k.a.* the Axiom of No Choice (`FunctionalRelReification_on`):

$$(\forall a.\ \exists! b.\ aRb) \Rightarrow \big(\exists f.\ \forall a.\ aR(f(a))\big)$$

While weaker, relying on the Axiom of No Choice would still be unfortunate. We exorcise all forms of Choice by explicitly constructing the inverse in §5.

5 There and back again: an explicit inverse

We now present a Choice-free proof of EPR's Lemma 2. Our proof is comprised of three parts. First, we show that well-formed short languages are finite. Second, we explicitly construct a function that computes a characteristic function from a list of words. Finally, we prove correctness: that our function, when given a well-formed short language represented as a list of words, computes the block cancelable language that agrees with it on short words. We show that i) when given a well-formed short language our function returns a block cancelable language; and ii) every block cancelable language is in the image of our function.

5.1 Well-formed short languages are finite

Our function must be computable so it inputs short languages as `list word` rather than `word -> Prop`. We require two properties of such lists to be suitable for building a block cancelable language: (P1) that they contain only "short" words, *i.e.* of length less than some $r(k)$; and (P2) that they agree with some $BC(k)$ language for all words up to length $r(k)$.

Informally, the fact that there are a finite number of sets containing words bounded by some length m is obvious: the cardinality is $2^{|\Sigma|^m}$. Proving this using Coq `lists` is less straightforward. Consider the following statement:

```
Lemma is_finite_shortlang_false : forall (rk: nat),
  exists (LW: list (list word)), forall (l: list word),
    In l LW <-> forall (w: word), In w l -> length w <= rk.
```

This statement is false because lists satisfying the right-hand side of the `<->` are infinite: they can contain duplicates. We must also contend with the hassles of permutations and ordering. We circumvent constructing and reasoning about duplicate-free, length-lexicographically sorted lists of words by proving the finiteness of a stronger property, and then weakening it to obtain the length property we require. We leverage Coq's inductive definitions to define a relation `subseq`, an order-preserving sublist relation of type `list -> list -> Prop`.

```
Inductive subseq (X: Type) : list X -> list X -> Prop :=
  | subseq_nil : forall (l: list X), subseq [] l
  | subseq_hm : forall (x: X) (l1 l2: list X),
    subseq l1 l2 -> subseq (x :: l1) (x :: l2)
  | subseq_hn : forall (x: X) (l1 l2: list X),
    subseq l1 l2 -> subseq l1 (x :: l2).
```

We can show that any list has a finite number of `subseq` lists, the proof of which proceeds by induction on the `subseq` relation.

```
Theorem subseq_finite: forall (l: list word),
  is_finite (fun s => subseq s l).
```

To show that there are finitely many lists containing words up to some length `n`, we want to instantiate `l` with the list containing *all* words up to length `n`. We define a function `generate_words_of_length` with correctness property:

```
Lemma generate_words_length_correct : forall (n: nat) (w: word),
  In w (generate_words_of_length n) <-> length w = n.
```

We then use it to define a function `generate_words_upto_length` with correctness property:

```
Lemma generate_words_upto_correct : forall (n: nat) (w: word),
  In w (generate_words_upto n) <-> length w <= n.
```

Now we can state the finiteness of (P1) using `generate_words_upto`:

```
Theorem is_finite_shortwords: forall (n: nat),
  is_finite (fun lw => subseq lw (generate_words_upto n)).
```

Next, we want to prune the lists that satisfy (P1) from `is_finite_shortwords` and keep only those that also satisfy (P2), i.e. that agree with some $BC(k)$ language up to some length. We first prove that any subset of a finite list is finite:

```
Lemma p_in_list_finite {X: Type}: forall (P: X->Prop) (L: list X),
  is_finite (fun l => P l /\ In l L).
```

This allows us to state that finiteness is preserved over conjunction:

```
Lemma is_finite_conj {X: Type} : forall (P: X->Prop) (Q: X->Prop),
  is_finite (fun l => P l) ->
  is_finite (fun l => P l /\ Q l).
```

Instantiating properties `P` and `Q` with (P1) and (P2) respectively, we obtain the finiteness of well-formed short languages and are ready to define our inverse function in §5.2.

```
Theorem is_finite_subseq_wf:
 forall (k rk: block_pumping_constant),
  is_finite (fun lw => subseq lw (generate_words_upto rk) /\
          exists (l : bc_language k), agreement_upto k rk l lw 0).
```

5.2 The **`unshear`** function

Our `unshear` function inputs a list of words `lw`, and computes a characteristic function—*i.e.* a word membership decider—for a block cancelable language.

We begin with a bird's eye view description of `unshear`'s behavior. `unshear` considers some arbitrary word `w` of length `n`. Starting with its input list `lw_init`, `unshear` incrementally considers sets of words of increasing length, adding those that pass some condition check until it has considered every word of length up to n. It then checks whether w is a member of the list computed so far, which we denote lw. The intuition behind unshear turns on the fact that block cancellation decreases word length, and that block cancelable languages are uniquely determined by a subset of words up to some length.

We follow with details of the function unshear. We accompany each computational function in bool with a correctness specification in Prop, and prove correctness: the _prop holds iff the function returns true.

A block canceled word with breakpoints i, j, is the word with the subword between the i-th and j-th symbol removed. We say that two indices cancel some word w into L if the block canceled word is a member of L. We build a block canceled word using firstn and skipn as follows:

```
Definition cancelled_word (w: word) (i j: nat) :=
  firstn i w ++ skipn j w.
```

First, we construct the function which checks words to be added to the list maintained by unshear. In particular, given some word w, we check for the existence of a k-size breakpoint set out of all possible k-size breakpoint sets for w, in which all pairs of breakpoints cancel w into some target list of words[5].

We first define a function which checks whether, for a given breakpoint set, all pairs of breakpoints cancel some word into a target list. The inner function are_all_pumps_helper takes one breakpoint hd and a list of breakpoints tl, and recursively traverses tl, pairwise checking the membership of the canceled word in the target list using a simple list membership checking function, is_member. We omit both functions for brevity.

The outer function recursively traverses a list of breakpoints and calls the inner function with each head element on the rest of the list, thus guaranteeing pairs are checked in order. We express the ordered correctness property for this function in terms of list indexing:

```
Fixpoint are_all_pumps (w: word) (l: list word)
                       (bps: list nat) :=
  match bps with
  | [] => true
  | hd :: tl => if are_all_pumps_helper w l hd tl
                then are_all_pumps w l tl
                else false end.
Definition are_all_pumps_prop (w: word) (l: list word)
                              (bps: list nat) :=
  forall (i j : nat), i < j < (length bps) ->
    In (cancelled_word w (nth i bps d) (nth j bps d)) l.
```

We want to apply this function to all possible k-size breakpoint sets for some w. We generate all k-size breakpoint sets via an order-preserving choose function which chooses n elements from a list of greater than or equal to n elements.

```
Fixpoint choose {X: Type} (L: list X) (k: nat) {struct L} :=
```

[5] We postpone discussion of why this condition works until §5.3

```
 match k with
 | 0 => nil :: nil
 | S k' => match L with
           | nil => nil
           | h :: L' => (map (fun l => h :: l) (choose L' k'))
                     ++ (choose L' k) end end.
```

The list we give to `choose` is the list of all possible breakpoints for `w`, i.e. the list starting from 0 and ending at `S (length w)`.

```
Definition get_k_bps (w: word) (k: nat) :=
  choose (iota 0 (S (length w))) k.
```

We use Coq's `existsb` function to check if there is a k-size breakpoint set in the list of all k-size breakpoint sets for which all pairs of breakpoints form pumps.

```
Definition exists_all_pumps_bps (w: word) (l: list word)
                                (k: nat) :=
  existsb (are_all_pumps w l) (get_k_bps w k).
Definition exists_all_pumps_bps_prop (w: word) (l: list word)
                                     (k: nat) :=
  exists lp : list nat,
    are_all_pumps_prop w l lp /\ In lp (get_k_bps w k).
```

Thus far, we have built the condition checker for an individual word `w` to be added to `lw` maintained by `unshear` that is parameterized by a `list word`, *i.e.* the target list in which canceled word membership is checked. However, the role of `lw` in `unshear` is twofold: not only does it accept new words, it also serves as the target list to determine the acceptance of future new words. `unshear` considers individual words in batches of a certain length. When `w` contains words of length up to some `m`, `unshear` considers all words of length `S m`. `lw` helps `unshear` determine which new words of length `S m` to add, and then accepts the ones that pass, updating itself to now contain words of length up to `S m`.

We first define the function which considers all words of some length. Here, `rk` is the length bound of our initial list, `n` is the difference between the length of the candidate word and `rk`, and `lref` is our target list.

```
Definition chuck (k rk n: nat) (lref: list word) :=
  filter (fun w=>check w lref k) (generate_words_of_length (n+rk))
++ lref.
Definition chuck_prop (k rk n: nat) (lref: list word) (w: word) :=
  (exists_all_pumps_bps_prop w lref k /\ length w = n+rk)
\/ In w lref.
```

We then define the recursive function which adds words of *up to* some length to be structurally decreasing over word length `nat`.

```
Fixpoint chuck_length (k rk n: nat) (lref: list word) :=
  match n with
  | 0 => lref
  | S n' => chuck k rk n (chuck_length k rk n' lref) end.
```

We are now ready to define unshear, with return type word->Prop, or language. We include an intermediate representation unshear_bool with return type word->bool.

```
Definition unshear_bool (k rk: nat) (lref: list word) :=
  fun w => is_member w (chuck_length k rk (length w) lref).
Definition unshear (k rk: nat) (lref: list word) :=
  fun w => unshear_bool k rk lref w = true.
```

5.3 Functional correctness of unshear

To use unshear to prove that there are finitely many block cancelable languages, we need to show that when given a well-formed short language represented as a list lw, unshear computes the block cancelable language that agrees with lw on short words. Proving the correctness of unshear thus amounts to proving the following theorem:

```
Theorem unshear_correctness: forall (k: block_pumping_constant),
 exists (rk: block_pumping_constant),
  forall (l: bc_language_dec k) (lw: list word),
   agreement_upto k rk l (chuck_length k rk lw 0) 0 ->
    (forall w, In w lw <-> (shear_language rk (unshear k rk lw)) w)
    /\ unshear k rk lw = (bc_language_dec_proj1 l).
```

The theorem states that for any decidable $BC(k)$ language L and list of words which agree with L up to length rk, (1) shear (unshear) lw = L, and (2) unshear (shear) L = lw, *i.e.* shearing an unsheared list returns us the input list, and unshearing a sheared language recovers us the language.

(1) amounts to showing unshear does not remove words from its input list or add words of length less than rk, and is straightforward.

(2) amounts to showing unshear recovers the block cancelable language L. This direction requires us to show the correctness of our chucking condition described above [§5.2], and involves Ramsey's theorem. In particular, we need to show that chuck preserves language agreement between lw and L, with language agreement defined as follows:

```
Definition agreement_upto (k rk: block_pumping_constant)
                          (l: bc_language_dec k)
                          (lw: list word) (m : nat) :=
  forall w, In w lw <-> (length w <= m + rk
                      /\ bc_language_dec_proj1 l w).
```

First, we show that given a list of words lw which agrees with some block cancelable language L up to length m, chucking in words of length m+1 results in a list which agrees with L up to length m+1. This further breaks down into two directions: 1) any word added by chuck must be in L and of length no more than m+1, and 2) any word in L and of length no more than m+1 must pass chuck's condition check.

```
Lemma IH_chuck_step: forall (k: block_pumping_constant),
 exists (rk: block_pumping_constant),
  forall (l: bc_language_dec k) (lw: list word) (m: nat),
   agreement_upto k rk l lw m ->
    agreement_upto k rk l (chuck k rk (S m) lw) (S m).
```

For the first direction, we have a word `w` that is either in `lw` or newly `chucked` in, and we must show (i) $|\texttt{w}| \leq$ `S m + rk` and (ii) `L w`. In the case that `w` is in `lw`, we are done. In the case that `w` is newly `chucked`, it satisfies the length requirement by definition. By our `chucking` condition, there exists a k-size breakpoint set `lp` with *all* breakpoint pairs forming pumps for `w` into `lw`. We apply `L`'s block cancellation property with `w` and `lp` to obtain a cancelled word `w'` which agrees with `w` on membership in `L`, use the induction hypothesis to obtain that `w'` is in `L`, and thus complete the proof that `w` is in `L`.

For the second direction, we have a word `w` with (i) $|\texttt{w}| \leq$ `S m + rk` and (ii) `L w`, and we must show that it is `chucked` in. This amounts to showing that it satisfies the `chucking` condition: that there exists a k-size breakpoint set containing all cancelable pumps for `w` into `lw`. This direction turns on Ramsey's theorem, as presented in (§2). From Ramsey's theorem, we know that for any r(k)-size breakpoint set, there exists a k-size breakpoint set with all pairs either forming cancelable pumps for `L` or cancelable pumps for `L`'s complement. In the first case, we have exactly the `chucking` condition. In the negative case, we have a contradiction from `L`'s block cancellation property.

`IH_chuck_step` can be seen as the inductive step for `chuck`'s correctness proof. We use it to prove `chuck_length`'s correctness theorem, which shows by induction that `chuck_length` preserves language agreement up to length `m + rk` for any arbitrary `m`, where `rk` is the length bound of `lw`.

```
Lemma IH_chuck:
 forall (k: block_pumping_constant),
  exists (rk: block_pumping_constant),
   forall (l: bc_language_dec k) (lw: list word) (m: nat),
    agreement_upto k rk l (chuck_length k rk 0 lw) 0 ->
     agreement_upto k rk l (chuck_length k rk m lw) m.
```

This completes the proof of the second obligation for `unshear`'s correctness: `unshear` adds exactly the same words that its associated block cancelable language `L` accepts up to some length `m + rk`. Therefore, by language equality, the resulting language is equivalent to `L`.

6 Related work

Automata theory. Automata and formal languages have been foundational topics to computing since Turing's introduction of his Machine [30]. Chomsky, together with Marcel P. Schützenberger, introduced the Chomsky hierarhcy [5, 6] of regular, context-free, context-sensitive and recursively enumerable sets of strings. These classes of languages have been extensively studied over the decades since to yield results of both practical and theoretical interest [17].

Pumping lemmas. Pumping lemmas connect the finite automata mechanism to the words such mechanisms can accept. The best-known pumping lemmas are by Rabin and Scott for regular languages and by Bar-Hilel, Perles, and Shamir for context-free languages [17, 24]. Jaffe [18] and Ehrenfeucht, Parikh and Rozenberg [14] pioneered the study of pumping properties that characterize the regular languages. Follow-up work [29] provided evidence that other pumping conditions are insufficient to give a characterization. Varrichio [31] solved an open problem of EPR by establishing that the positive block pumping property (the pump can be repeated but not canceled) also characterizes regular languages. Chak, Freivalds, Stephan and Tan [4] studied the class of languages that are block pumpable but whose complement is not.

Constructive mathematics. Brouwer originated the ideas of intuitionistic mathematics [16], which removes the Law of Excluded Middle as a universal reasoning principle. The generalized Axiom of Choice is not admitted by intuitionistic logic: Diaconescu's theorem shows that it leads to the Law of Excluded Middle [11].

Martin-Löf developed intuitionistic type theory [22] and the notion of dependent types, thereby contributing to many associated mechanized proof environments. Thierry Coquand took these ideas and built the calculus of constructions [8], which in turn led to the calculus of inductive constructions [27], the underlying logic of the Coq proof assistant [7]. Coq separates computation (*i.e.* `Type`) from mathematical truths (*i.e.* `Prop`). The Axiom of Choice, in a type-theoretical context, essentially erases this distinction.

Mechanizations of automata theory. Interest in mechanizing automata theory began over thirty years ago [20]. Existing work in formalizing automata theory focuses on languages in the Chomsky hierarchy: Kreitz [20] and Constable *et al.* [10] formalize finite automata-based regular language theory in NuPRL, Dockzal *et al.* [12, 13] formalize regular language theory in Coq, Ramos *et al.* [26] formalize context-free language theory in Coq and Zhang *et al.* [32] formalize the Myhill-Nerode theorem using only regular expressions in Isabelle/HOL.

Some of these proofs are constructive, although in a few cases the authors assume that they are working in the Chomsky hierarchy to begin with. For example, Dockzal *et al.* [12] use the Coq type `word -> bool` to represent languages, rather than `word -> Prop` as we do, and then go on to prove Myhill-Nerode constructively. In a certain sense this begs the question, however.

There is also substantial existing work focusing on verified translation and decision procedures for representations of regular languages. Filliatre [15] constructively proves the expressive equivalence of regular expressions and finite automata in Coq, and extracts a functional program which translates a regular expression to a finite automata, Almeida *et al.* [1] prove the correctness of a partial derivative automata construction from regular expressions in Coq, Coquand and Nipkow *et al.* [9, 25] verify a decision procedure for regular expression equivalence in Coq, and Krauss *et al.* [19] verify a regular expression equivalence checker in HOL/Isabelle etc.

7 Conclusion

To the best of our knowledge, the present work is the first mechanization of language classes, namely the one-sided block pumpable and one-sided block cancelable languages, that are orthogonal to the Chomsky hierarchy and furthermore, cannot be characterized algebraically or automata-theoretically. We have formalized two important and significantly different pumping lemmas which both characterize regularity: Jaffe's pumping lemma and EPR's block pumping lemma. We have also formalized closure properties of one-sided block pumpable languages. We have presented a new Choice-free proof of EPR's theorem by defining an inverse function from block cancelable to well-formed short languages.

References

1. Jose Carlos Bacelar Almeida, Nelma Moreira, David Perira and Simao Melo de Sousa. Partial Derivative Automata Formalized in Coq. *The 15th International Conference on Implementation and Application of Automata*, 10:59–68, 2010.
2. Stefan Banach and Alfred Tarski. Sur la décomposition de ensebles de points en parties respectivement congruentes. *Fundamenta Mathematicae* 6:244–277, 1924.
3. Stefano Berardi, Marc Bezem and Thierry Coquand. On the Computational Content of the Axiom of Choice. *The Journal of Symbolic Logic*, 63(2):600–622, 1998.
4. Christopher Hanrui Chak, Rusins Freivalds, Frank Stephan and Henrietta Tan Wan Yik. On Block Pumpable Languages. *Theoretical Computer Science*, 2016.
5. Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 19859.
6. Noam Chomsky and Marcel P. Schützenberger. The algebraic theory of context free languages. *Computer Programming and Formal Languages*, 1963.
7. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.10.0.* http://coq.inria.fr, 2019.
8. Thierry Coquand, Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.
9. Theirry Coquand and Vincent Siles. A Decision Procedure for Regular Expression Equivalence in Type Theory. *Certified Programs and Proofs*, 11:119–134, 2011.
10. Robert Constable, Paul B. Jackson, Pavel Naumov and Juan Uribe. Constructively formalizing automata theory. *Proof, Language and Interaction*, 1998.
11. Radu Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, 51:176–178, 1975.
12. Christian Doczkal, Jan-Oliver Kaiser and Gert Smolka. A Constructive Theory of Regular Languages in Coq. *Certified Programs and Proofs*, 82–97, 2013.
13. Christian Doczkal, Gert Smolka. Regular Language Representations in the Constructive Type Theory of Coq. *Journal of Automated Reasoning*, 2018.
14. Andrzej Ehrenfeucht, Rohit Parikh and Grzegorz Rozenberg. Pumping lemmas for regular sets. *SIAM Journal on Computing*, 10:536–541, 1981.
15. Jean-Christophe Filliatre. Finite Automata Theory in Coq: A constructive proof of Kleene's theorem. *Ecole Normale Supérieure de Lyon Research Report*, 1997.
16. Jean Van Heijenoort. *From Frege to Godel: A Source Book in Mathematical Logic, 1879-1931.* Harvard University Press, 1967.
17. John E. Hopcroft, Ravjeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Third edition. Addison Wesley, 2007.

18. Jeffrey Jaffe. A necessary and sufficient pumping lemma for regular languages. *ACM SIGACT News*, 10(2):48–49, January 1978.
19. Alexander Krauss and Tobias Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *Journal of Automated Reasoning*, 49(1):95–106, 2012.
20. Cristoph Kreitz. Constructive Automata Theory Implemented with the Nuprl Proof Development System. *Computer Science Technical Reports*, 1986.
21. Elaine Li. *Formalizing Block Pumpable Language Theory.* Capstone Final Report for BSc. (Honours) in Mathematical, Computational and Statistical Sciences,Yale-NUS College, 2019.
22. Per Martin-Löf. An intuitionistic theory of types. *Twenty-Five Years of Constructive Type Theory*, 1998.
23. Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society* 9(4):541–544, 1958.
24. Anton Nijholt. An annotated bibliography of pumping. *Bulletin of the EATCS*, 17:34–53, June 1982.
25. Tobias Nipkow and Dmitriy Traytel. Unified Decision Procedures for Regular Expression Equivalence. *Interactive Theorem Proving*, 450–466, 2014.
26. Marcus Vinicius Midena Ramos, Ruy J.G.B. de Queiroz, Nelma Moreira and Jose Carlos Bacelar Almeida. On the Formalization of Some Results of Context-Free Language Theory. *Lecture Notes in Computer Science*, 9803:338–357, 2016.
27. Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. *Studies in Logic (Mathematical logic and foundations)*, 978-1-84890-166-7, 2015.
28. F.P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, s2(30):264–286, 1930.
29. Rudolph Sommerhalder. Classes of languages proof against regular pumping. *RAIRO Informatique théorique*, 14:169–180, 1980.
30. A.M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem: A correction. *Proceedings of the London Mathematical Society*, 43(6):544–546, 1937.
31. Stefano Varricchio. A pumping condition for regular sets. *SIAM Journal on Computing*, 26(3):764–771, 1997.
32. Chunhan Wu, Xingyuan Zhang and Christian Urban. A Formalisation of the Myhill-Nerode Theorem Based on Regular Expressions (Proof Pearl). *ITP*, 2011.