Formally Verifying Automata for Trusted Decision Procedures

Zhaobo (Aeacus) Sheng

May 2025

Committee:

Jeremy Avigad (Chair)
Professor of Philosophy and Mathematical Sciences, Carnegie Mellon University

---

Marijn J.H. Heule
Associate Professor of Computer Science, Carnegie Mellon University

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Logic, Computation, and Methodology
to the Department of Philosophy, Carnegie Mellon University

Abstract

Automata-theoretic decision procedures date back to Büchi's work, leveraging finite automata to decide sentences in weak theories of arithmetic. An extension of the original procedure by adding automatic sequences is now implemented in the Walnut software to check and discover theorems in combinatorics and number theory. However, automata-based decision procedures do not generate efficiently checkable correctness guarantees as they run, raising concerns regarding the reliability and trustworthiness of the results they produce.

Towards addressing this issue, we are building an automata library in Lean, which is both a proof assistant and a programming language. By formalizing executable automata and proving their properties, we provide a foundation for trusted automata-theoretic decision procedures. We begin with Chapter 1, which introduces the context and motivation for this project. The basic mathematical theory of words and automata is provided in Chapter 2, with their formalization explained. Chapter 3 presents a decision procedure for automatic sequences, and reports the verification of crucial automata involved. To our knowledge, this is the first exposition of a direct decision procedure for automatic sequences in full mathematical detail, together with the first formally verified automata library designed to support a decision procedure for automatic sequences. We conclude in Chapter 4 by summarizing our contributions and propose directions for future work.

Contents

1 Walnut and Lean

1.1 Automated Reasoning in Walnut

The quest to automate mathematical reasoning traces its roots to David Hilbert's foundational *Entscheidungsproblem* (decision problem), which sought an algorithm to decide with certainty whether an arbitrary logical expression is universally valid [17]. While Turing and Church's negative resolution of the general decision problem for first-order logic marked a pivotal moment in computability theory [26], the search for decision procedures—algorithms that decide the truth of sentences in specific logical theories—has flourished. Today, decision procedures underpin many automated reasoning tools, helping computer scientists in safety-critical software verification and mathematicians in cutting-edge research. SAT solvers, which implement sophiscated decision procedures for the satisfiability problem of propositional logic formulae, have had huge success in solving century-old problems once deemed intractable [15, 16]. Recently, another decision procedure has been implemented in a software called Walnut, leveraging finite automata to decide properties of automatic sequences (roughly, sequences generated by finite automata) [23]. For example, consider the infinite Thue-Morse sequence

$$\mathbf{t} = (t_n)_{n\geq 0} = t_0t_1t_2\ldots = 01101001100101101001011 0\ldots$$

where t_n is the parity of the number of 1s in the base-2 digits of n: $t_n = 1$ if there are an odd number of 1s, and $t_n = 0$ otherwise. As we will see later, t can also be generated by an automaton. It turns out that t is not eventually periodic, and this fact can be expressed in the standard language of first-order arithmetic extended by a unary function T such that $T(i) = T_i$ when interpreted in the standard model:

$$\neg\exists n \geq 0,\ \exists p \geq 1,\ \forall i \geq n,\ T(i) = T(i+p)$$

When we give this formula to Walnut, it can tell us that this is true in under a second. Mathematicians have been using Walnut to prove theorems in combinatorics of words and

additive number theory, and as of May 2025 there has been over 100 papers, books, and theses that have used Walnut in their work [1, 24].

1.2 Formal Verification in Lean

Complex computer programs have a notorious history of containing critical bugs [21, 27]. So, when they are answering mathematical questions, concerns arise naturally regarding their trustworthiness. The SAT community addressed this issue by making the solvers produce relatively short proof certificates that can be verified by a tiny proof checker, which is formally proven to have no bugs [14]. In this way, every solution a SAT solver produces comes with a correctness guarantee. In contrast, the automata-based decision procedure in Walnut does not generate formal correctness guarantees as they execute. While Walnut's outputs are empirically reliable, its internal automata operations and number representations are prone to implementation errors, and a single bug could invalidate years of research built atop Walnut's results.

This problem calls for the formal verification of a decision procedure for automatic sequences in a suitable framework. Interactive theorem provers like Isabelle [20] and Lean [11] usually have expressive logic implemented on a small trusted kernel, providing extremely high confidence in the proofs written inside. Formalized proof is becoming the new standard of mathematical rigor, with attempts to verify proofs in interactive theorem provers leading to the discovery and correction of several errors [5, 13, 12]. Designed with the aim to bridge the gap between interactive and automated theorem proving by situating automated tools and methods in a framework that supports user interaction and the construction of fully specified axiomatic proofs [3], and being both a programming language and an interactive theorem prover [18], Lean is especially suitable for implementing and verifying a decision procedure for automatic sequences.

We take the first step towards a trusted decision procedure for automatic sequences by formalizing relevant automata and proving their properties in Lean. Chapter 2 reviews the basic mathematical theory of automata and how natural numbers are represented as words, with our formalization of them explained. Then, we present a decision procedure for automatic sequences in full mathematical detail, and report how the crucial automata involved have

been formally verified. In reporting formalization, we assume basic knowledge of dependent type theory and the syntax of Lean [4].

1.3 History and Related Works

By a decision procedure for automatic sequences, we mean a decision procedure for the theory of natural numbers with addition $\mathrm{Th}(\mathbb{N},+)$, extended by certain functions for indexing into automatic sequences, like the function T in the earlier example. $\mathrm{Th}(\mathbb{N},+)$ is also called Presburger arithmetic. It is named after Mojżesz Presburger, who proved that the theory is decidable using quantifier elimination in 1929 [22]. Later in the century, the work of Büchi, Boudet and Comon lead to an automata-theoretic decision procedure for Presburger arithmetic [9, 10, 7].

Bruyàre et. al fixed an error in Büchi's work, leading to a way to code an automatic sequence into Presburger arithmetic extended by a function V_k such that $V_k(x) = k^n$, where k^n is the largest power of k dividing x, and proved that the extension is still decidable using automata, thus giving rise to a decision procedure for automatic sequences [8]. However, the extension by the additional function and the coding of automatic sequence make the decision procedure unnecessarily complicated. Shallit sketched a very rough (one sentence) augmentation to the decision procedure for pure Presburger arithmetic that does not involve these complications by directly computing automatic sequences in his Walnut book, The Logical Approach to Automatic Sequences [23]. This book is our primary reference for Chapters 2 and 3. Mousavi, one of Walnut's authors, described the decision procedure in more detail, focusing on implementation [19]. Complementing the current literature, we will present a decision procedure for automatic sequences in full mathematical detail, emphasizing theory rather than implementation.

A formally verified implementation of the automata-theoretic decision procedure for pure Presburger arithmetic exists in the interactive theorem prover Isabelle, thanks to Berghofer and Reiter [6]. They built a library for automata on bit strings, with transitions stored as binary decision diagrams for efficiency. So, their library and decision procedure are only applicable to automata with a binary alphabet. This creates no problem for Presburger arithmetic, but severely limits the possibility of extending this procedure to decide properties of

automatic sequences, because many of them require non-binary alphabets (e.g. the Mephisto Waltz sequence) [23]. Our automata library, therefore, aims to accommodate all automatic sequences.

Lean's math library Mathlib [25] contains limited formalization of finite automata, but they use `Set` and noncomputable constructions extensively, making them undesirable for building decision procedures. Moreover, there are no formalizations of automata with output, which is essential for defining automatic sequences. Our automata library references the formalization in Mathlib, with significant additions and changes geared towards an executable decision procedure for automatic sequences, with proofs of their properties. To the best of our knowledge, this is the first automata library built for decision procedures in Lean, and the first automata library for verifying properties of automatic sequences in any interactive theorem prover. Of course, our library would also support decision procedures for pure Presburger arithmetic and Büchi's related weak second-order theories.

2 Words and Automata

2.1 Alphabets and Words

The concept of a word is essential to automata theory. Every word is defined over an alphabet.

Definition 2.1 – Alphabet

An alphabet Σ is a set of symbols (also called letters).

In general, words can be infinite. For our purposes, we reserve the word "word" for finite words.

Definition 2.2 – Word

A word over Σ is a finite sequence of symbols from Σ.

We write a word as $w = a_1 a_2 \cdots a_n$ with each $a_i \in \Sigma$. When a letter has an exponent k, it means that it is repeated k times. The length of w, denoted by $|w|$, is n. The unique word of length 0 is the empty word, denoted by ε.

Let Σ^* denote the set of all finite words over Σ. If $w \in \Sigma^*$ and $v \in \Sigma^*$, their concatenation is the word $wv \in \Sigma^*$ obtained by appending v after w. Concatenation with ε has no effect: $w\varepsilon = \varepsilon w = w$. The set of all finite words over Σ forms a monoid under the concatenation operation.

A cartesian product of two alphabets can be used as a new alphabet, so we can have a product of two words, if they have the same length.

Definition 2.3 – Product Words

For $w = a_1 \cdots a_n \in \Delta^*$ and $x = b_1 \cdots b_n \in \Sigma^*$, their product is:

$$w \times x = (a_1, b_1)(a_2, b_2) \ldots (a_n, b_n) \in (\Delta \times \Sigma)^*$$

When $\Delta = \Sigma$, we just write the product alphabet as Σ^2, this can be generalized to cartesian powers of alphabets. A word over Σ^n is just a finite sequence of $n-$tuples with elements from Σ.

We are particularly interested in alphabets that are initial segments of natural numbers, and their cartesian powers. We use $\Sigma_k = 0, 1, \ldots, k-1$ to denote the canonical k-letter alphabet for $k \geq 2$, and Σ_k^n for its n-th power.

In Lean, we take an alphabet to be a type inhabited by its symbols. The type `Fin k` containing natural numbers less than k is used for Σ_k. A word over an alphabet `α` is naturally taken to be a list of letters, of type `List α`. To access elements from tuples easily, we use `Fin n → Fin k` for Σ_k^n, as opposed to using product types.

2.2 Number Representations

Representing natural numbers as words is essential for automata-theoretic decision procedures, and we focus on base-b representations here. Recall, from elementary number theory, that every natural number can be put uniquely in base b.

> **Theorem 2.1 – Base-b digits of a natural number**
>
> Let $b \geq 2$. Every natural number n can be written uniquely as:
>
> $$n = \sum_{1 \leq i \leq t} a_i b^{t-i},$$
>
> where $t \geq 0$, $0 \leq a_i < b$ for $1 \leq i \leq t$, and $a_t \neq 0$.

We can then represent a natural number n as a word $(n)_b = a_1 a_2 \cdots a_t$ over Σ_b, called the canonical base-b word of n. For example, $(22)_2 = 10110$. The empty word ε represents 0. Note that adding leading zeros in base b digits does not change the natural number, so infinitely many words over Σ_b represent the same natural number. For example, 10110 and 0010110 both represent 22 over Σ_2. By the theorem above, number representations in base b are unique up to adding leading zeros.

To obtain the canonical base-b word of a natural number n in Lean, we make use of Mathlib's `Nat.digits` function, which computes the base b digits of n into a list. This function puts the least significant digits first, so we reverse it in defining our `toBase` function.

```
def toBase (b n : ℕ) : List ℕ := (Nat.digits b n).reverse
```

We can also represent m-tuples of natural numbers as words over Σ_b^m. The idea is to take the product over base-b words of every individual natural number. Since different natural numbers might have different canonical word length, we need to find the longest word and add leading zeros to all shorter words, before taking the product. This gives us the canonical base-b word of a tuple. Similar to the case for a single natural number, adding tuples of zeros at the beginning of a word does not change the tuple of natural numbers it represents.

We type m-tuples of natural numbers in Lean using the vector type `Fin m → ℕ`. To obtain the canonical base-b word of a tuple v, we first compute the canonical base-b word for every element in the tuple.

```
def mapToBase (b : ℕ) (v : Fin m → ℕ) : Fin m → List ℕ :=
  fun i => toBase b (v i)
```

Then, we add necessary leading zeros so that every element is represented by a word with uniform length. The names of `addZeros` and `maxLenFin` suggest their functions.

```
def stretchLen (ls: Fin m → List ℕ) : Fin m → List ℕ :=
  fun i => addZeros (maxLenFin ls - (ls i).length) (ls i)
```

Finally, we zip the vector of lists into a list of vectors by recursion on the length of the words, so they become words over Σ_b^m. Note that although we represent words as lists and tuples as vectors, it is also possible to represent words as vectors and tuples as lists, because mathematically they are both sequences. In deciding between lists and vectors for representing sequences in Lean, the key is whether we want to fix the length of the sequence in the type. With a cartesian power alphabet Σ^n, we need letters to have a fixed length n. If we use lists of elements from Σ to represent letters, then every letter needs to come with a proposition stating that its length is n. Using vectors avoids this by directly typing letters by `Fin n → Σ`. On the other hand, using vectors to represent words is not ideal because we often need to concatenate or decompose words, requiring frequent type casting for the length of vectors.

Using lists avoids this problem, and can significantly simplify proofs throughout the library.

```
def zip (ls: Fin m → List ℕ) (hlb: ∀ i, ∀ x ∈ ls i, x < (b + 2)) (hls : ∀ i
    , (ls i).length = l) : List (Fin m → Fin (b + 2)) :=
  match l with
  | 0 => []
  | m + 1 =>
     (fun i =>
       have : 0 < (ls i).length := by
         rw[hls]
         omega
       ⟨(ls i)[0], by apply hlb; exact List.getElem_mem this⟩) :: (zip (fun i
          => (ls i).tail)
        (by apply zipTailHlb; exact hlb)
        (by apply zipTailHls; exact hls))
```

Starting here, we work with base `b + 2` in Lean. This simplifies the verification effort because Lean can automatically infer that the base we are working with is at least 2. So if we want to use base $b = 2$, we set `b` to 0. Unless otherwise specified, we will stick with using b in our mathematical presentation, and use `b + 2` for b in Lean. The proofs and theorems appearing in this definition are there in order to correctly type the final word. Bringing all of the steps together, we have a function that computes the canonical base-b word for an m-tuple of natural numbers.

```
def toWord (v: Fin m → ℕ) (b: ℕ) : List (Fin m → Fin (b + 2)) :=
  zip (stretchLen (mapToBase (b + 2) v)) (stretchLen_of_mapToBase_lt_base
    _ _ (by omega)) (fun i => stretchLen_uniform _ _)
```

2.3 Finite Automata

Finite-state automata are at the heart of automata-theoretic decision procedures. We begin with DFAO, a basic type of finite automata.

> **Definition 2.4 – Deterministic Finite Automata with Output**
>
> Let Σ and Γ be nonempty alphabets. A deterministic finite automaton with output (DFAO) is a sextuple
>
> $$M = (\Sigma, Q, q_0, \delta, \Gamma, \omega),$$
>
> where
>
> - Σ is the input alphabet
> - Q is a finite, nonempty set of states;
> - $q_0 \in Q$ is the distinguished start state;
> - $\delta : Q \times \Sigma \to Q$ is the transition function;
> - Γ is the output alphabet
> - $\omega : Q \to \Gamma$ is the output function.

We formalize DFAO in a straightforward way, where `α` is Σ, `state` is Q, `out` is Γ, `transition` is the curried δ, `start` is q_0, and `output` is ω. Whenever we need to use the fact that Q is finite, we supply an instance `[Fintype state]`. We often need the `[DecidableEq state]` instance as well for reasoning about states. If a finite automaton has n states, we usually take the canonical n-element type `Fin n` as its state type.

```
structure DFAO (α  state out: Type) where
  (transition : α → state → state)
  (start : state)
  (output : state → out)
```

We can evaluate words over the input alphabet of an automaton.

Definition 2.5 – Evaluating Words

Given $M = (\Sigma, Q, q_0, \delta, \Gamma, \omega)$, we can evaluate words over Σ by extending δ:

$$\hat{\delta}(q, \varepsilon) = q, \qquad \hat{\delta}(q, aw) = \hat{\delta}\big(\delta(q, a), w\big) \quad \text{for } w \in \Sigma^*,\ a \in \Sigma,$$

and compute the function $f_M : \Sigma^* \to \Gamma,\ f_M(w) = \omega\big(\hat{\delta}(q_0, w)\big)$, which terminates because w is finite.

Intuitively, the automaton transitions from a state to another (not necessarily distinct) state after reading each letter. When it finishes reading the word, the state it ends up in will tell us the output of this run. The function $\hat{\delta}$ is defined in Lean accordingly.

```
def DFAO.transFrom (dfao : DFAO α state out) (x : List α ) (s : state) :
    state :=
  match x with
    | [] => s
    | a::as => DFAO.transFrom dfao as (dfao.transition a s)
```

And the definition for f_M, `DFAO.eval`, follows.

```
def DFAO.evalFrom (dfao : DFAO α state out) (x : List α ) (q : state) : out
    :=
  dfao.output (DFAO.transFrom dfao x q)

def DFAO.eval (dfao : DFAO α state out) (x : List α ) : out :=
  DFAO.evalFrom dfao x dfao.start
```

DFA are very similar to DFAO, with the output alphabet and output function replaced by a set of accepting states.

Definition 2.6 – Deterministic Finite Automata

Let Σ be a nonempty alphabet. A deterministic finite automaton (DFA) is a quintuple

$$M = (\Sigma, Q, q_0, \delta, F),$$

where

- Σ is the input alphabet
- Q is a finite, nonempty set of states
- $q_0 \in Q$ is the distinguished start state
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $F \subseteq Q$ is the set of accepting states

δ can be extended to $\hat{\delta}$ in the same way for DFAO. A word x is said to be accepted by M if $\hat{\delta}(q_0, x) \in F$, and rejected otherwise. The set of all accepted words $\{x \in \Sigma^* | \hat{\delta}(q_0, x) \in F\}$ is called the language recognized by M, and is written $L(M)$.

We observe that a DFA is essentially equivalent to a DFAO where $\Gamma = \{\top, \bot\}$ and $\omega = \chi_F$ with

$$\chi_F(q) = \begin{cases} \top & \text{if } q \in F, \\ \bot & \text{otherwise.} \end{cases}$$

in the sense that F and χ_F uniquely specify each other. In Lean, we implement DFA as DFAO by using `out` in the DFAO as the characteristic function of the accepting set.

```
-- A DFA is a DFAO where the output is a boolean
   abbrev DFA (α  state : Type) := DFAO α state Bool
```

A celebrated theorem that we will use is the pumping lemma for DFA.

Theorem 2.2 – Pumping Lemma

Given a DFA M, let $L = L(M)$ be the set of words it accepts. Then there is a constant n such that for all $z \in L$ with $|z| \geq n$, there exists a decomposition $z = uvw$ with $|uv| \leq n$ and $|v| \geq 1$ such that $uv^i w \in L$ for all $i \geq 0$.

Proof. Say M has n states. This n is the constant in the statement of the theorem. If $|z| \geq n$, consider the states encountered when reading the first n symbols of z, formally $\{\hat{\delta}(q_0, z') \mid z' \text{ is an initial segment of } z \text{ with length } \leq n\}$. Including the initial state, at least $n+1$ not necessarily distinct states are encountered, so by the pigeonhole principle, some

state is repeated. This means that reading a nonempty segment v of z on this state loops back to itself. Write $z = uvw$, where u is the word that precedes the loop and w the word that follows the loop. We can repeat looping (or not go through the loop at all) and still get an acceptance path for the word $uv^i w$. □

There is actually a more general version for DFAO, which we proved in Lean. The `{b}*` means arbitrary exponent of `b`, in the sense that `{a} * {b}* * {c}` means the set $\{ab^n c | n \in \mathbb{N}\}$. We will use this notation in the next chapter. When reporting theorems, we will not show their proofs.

```
theorem DFAO.pumping_lemma_eval [Fintype state] {dfao : DFAO α state out} {x
    : List α } {o : out} (hx : dfao.eval x = o)(hlen : Fintype.card state ≤ x
    .length) :
    ∃ a b c, x = a ++ b ++ c ∧ a.length + b.length ≤ Fintype.card state ∧ b
       ≠[]
     ∧ ∀ y ∈ ({a} * {b}* * {c} : Language α ), dfao.eval y = o
```

So far, we have been looking at deterministic automata, where we transition to a single fixed state upon reading a letter. There are also nondeterministic automata, where we can transition to multiple states.

Definition 2.7 – Nondeterministic Finite Automata (NFA)

An nondeterministic finite automaton is a quintuple

$$M = (Q, \Sigma, \delta, Q_0, F)$$

where:

- Q: Finite set of states
- Σ: Input alphabet
- $\delta : Q \times \Sigma \to 2^Q$: Transition function
- $Q_0 \subseteq Q$: Initial state set
- $F \subseteq Q$: Accepting states

In formalizing NFA, we use lists without duplicates instead of sets for efficiency. The type `ListND state` contains all lists of elements from `state`, with no duplicates. If `state` is a finite type, then there is a bound on the length of elements of `ListND state`, making the type finite as well. Formalizing 2^Q as `ListND state` makes the state sets ordered, so the total number of state sets is larger than needed. However, it allows the use of list operation and related theorems in Mathlib, simplifying implementation and verification.

```
abbrev ListND (α  : Type) := {l : List α // l.Nodup}

structure NFA (α  state : Type) where
  (transition : α → state → ListND state)
  (start : ListND state)
  (output : state → Bool)
```

For an NFA, the transition function δ is extended to $\hat{\delta} : 2^Q \times \Sigma^* \to 2^Q$ recursively:

- For $Q' \subseteq Q$ and $w = \varepsilon$,
$$\hat{\delta}(Q', \varepsilon) = Q'$$
- For $Q' \subseteq Q$, $w \in \Sigma^*$, and $a \in \Sigma$,
$$\hat{\delta}(Q', aw) = \hat{\delta}(\bigcup_{q \in Q'} \delta(q, a), w)$$

and M accepts a word w iff $\hat{\delta}(Q_0, w) \cap F \neq \emptyset$.

In Lean, these are implemented in a way similar to the ones for DFAO above, but using list operations. `NFA.transList` computes $\bigcup_{q \in qs} \delta(q, a)$.

```
def NFA.transList (nfa : NFA α state) (a : α ) (qs : ListND state) [
    DecidableEq state] : ListND state :=
  ⟨(qs.val.flatMap fun q => nfa.transition a q).dedup, (by apply List.
      nodup_dedup)⟩

def NFA.transFrom (nfa : NFA α state) (s : List α ) (qs : ListND state) [
    DecidableEq state] : ListND state :=
  match s with
    | [] => qs
    | a::as => NFA.transFrom nfa as (NFA.transList nfa a qs)

def NFA.evalFrom (nfa : NFA α state) (s : List α ) (qs : ListND state) [
    DecidableEq state] : Bool :=
  (nfa.transFrom s qs).val.any nfa.output

def NFA.eval (nfa : NFA α state) (s : List α ) [DecidableEq state] : Bool :=
  NFA.evalFrom nfa s nfa.start
```

It turns out that for every NFA, we can construct an equivalent DFA, in the sense that they accept the same words.

Definition 2.8 – Subset Construction

Let $M = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. The subset construction produces a DFA

$$M' = (2^Q, \Sigma, \delta', Q_0, F')$$

where:

- States: 2^Q (all subsets of Q)
- Initial state: Q_0
- Transitions: For $S \subseteq Q$ and $a \in \Sigma$,

$$\delta'(S,a) = \bigcup_{q \in S} \delta(q,a)$$

- Accepting states: $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$

For an NFA with n states, the subset-constructed DFA has 2^n states.

Using the `transList` function earlier, the subset construction is elegantly formalized in Lean.

```
def NFA.toDFA (nfa : NFA α state) [DecidableEq state] : DFA α ListND state
    where
  transition := fun a qs => NFA.transList nfa a qs
  start := nfa.start
  output := fun qs => qs.val.any nfa.output
```

The equivalence between the NFA and its corresponding DFA can be proven.

Theorem 2.3 – Correctness of Subset Construction

Let M be an NFA and M' its subset-constructed DFA. For all $w \in \Sigma^*$,
M accepts w iff M' accepts w

Proof. By induction on w, prove that $\hat{\delta}'(Q', w) = \hat{\delta}(Q', w)$ for any $Q' \subseteq Q$. Base case: $w = \varepsilon$, we have $\hat{\delta}'(Q', \varepsilon) = Q' = \hat{\delta}(Q', \varepsilon)$.
Inductive step: For $w = ax$, we have $\hat{\delta}'(Q', ax) = \hat{\delta}'(\delta'(Q', a), x) = \hat{\delta}'(\bigcup_{q \in Q'} \delta(q, a), x)$, which, by IH, is the same as $\hat{\delta}(\bigcup_{q \in Q'} \delta(q, a), x) = \hat{\delta}(Q', ax)$.

In particular, we have $\hat{\delta}'(Q_0, w) = \hat{\delta}(Q_0, w)$. Then, M accepts w iff $\hat{\delta}(Q_0, w) \cap F \neq \emptyset$ iff M' accepts w.

□

We have established this equivalence in Lean.

```
theorem NFA.toDFA_eval (nfa : NFA α state) (s : List α ) [DecidableEq state]
  : (nfa.toDFA).eval s = nfa.eval s
```

As a corollary, we have the pumping lemma for NFA.

Corollary 2.1 – Pumping Lemma for NFA

Given an NFA M, let $L = L(M)$ be the set of words it accepts. Then there is a constant n such that for all $z \in L$ with $|z| \geq n$, there exists a decomposition $z = uvw$ with $|uv| \leq n$ and $|v| \geq 1$ such that $uv^i w \in L$ for all $i \geq 0$.

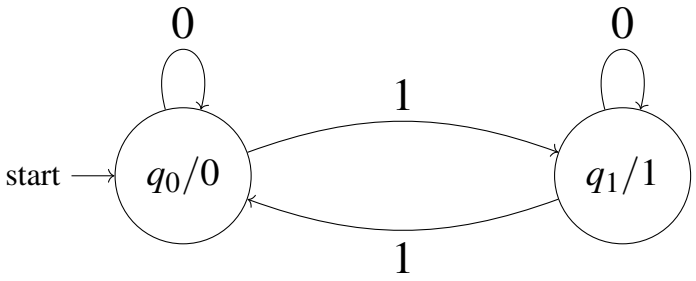
Proof. Using the subset construction, obtain an equivalent DFA for M, then apply the pumping lemma for the DFA. □

2.4 Automatic Sequences

Definition 2.9 – b-automatic sequence

Let $b \geq 2$. An infinite sequence $a = a_0 a_1 a_2 \ldots$ taking values in a finite alphabet $\Delta \subset \mathbb{N}$ is called b-automatic if there exists a DFAO $M = (Q, \Sigma_b, \delta, q_0, \Delta, \omega)$, such that for every $n \geq 0$, for any base b word w of n, the output of $f_M(w) = a_n$.

If a sequence s is b-automatic for some b, then it is automatic. There are more automatic sequences produced related to different number representation systems, which we do not discuss here. The Thue-Morse sequence is 2-automatic, generated by the following DFAO. Automata are often depicted in transition diagrams like this, where states are displayed as circles with their names and output inside, and transitions are displayed as labeled arrows. For automata without output, only state names are displayed and accepting states are marked with double circles.



3 A Decision Procedure for Automatic Sequences

Before delving into the decision procedure, we have to understand the syntax of the formal language[1] in which problems are stated.

3.1 Presburger Arithmetic Extended by an Automatic Sequence

Instead of reviewing general first-order logic, we provide a minimal introduction to the language of Presburger arithmetic extended by a function S for indexing into a b-automatic sequence s, in classical first-order logic with equality.

First-order logic with equality contains the connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$; the quantifiers $\exists$, $\forall$; a countably infinite collection of variables x_1, x_2, ..., y, z, ... the two parentheses), (; and the equality symbol $=$. In Presburger arithmetic, we also have the binary function symbol $+(\cdot,\cdot)$. Adding the unary function symbol S completes the building blocks we need. We call this language $\mathscr{L}_S$.

Definition 3.1 – $\mathscr{L}_S$ Terms

The set of $\mathscr{L}_S$-terms $\mathscr{T}$ is generated inductively by the following clauses

- For any variable v, $v \in \mathscr{T}$
- For any $t_1, t_2 \in \mathscr{T}$, then $+(t_1,t_2) \in \mathscr{T}$
- For any $t \in \mathscr{T}$, $S(t) \in \mathscr{T}$

For example, $+(x,S(x))$ is a $\mathscr{L}_S$-term. When writing down terms, we can use the function symbol $+$ as in common mathematical practice, replacing $+(x,y)$ as $(x+y)$. Syntactically, it is important to keep the parentheses, although semantically $+$ is associative. The set of formulae is defined in the standard way.

[1] Unfortunately, the word "language" is used in both mathematical logic and automata theory, and they mean different things. In this section, a language is just a collection of symbols, which we use to build terms and formulae.

Definition 3.2 – $\mathscr{L}_S$-formula

Finite strings of the following forms are called atomic formulae:

- $t_1 = t_2$, where t_1 and t_2 are $\mathscr{L}_S$-terms

The set of $\mathscr{L}_S$-formulae $\mathscr{F}$ is generated inductively by the following clauses:

- if $\phi \in \mathscr{F}$, then $\neg(\phi) \in \mathscr{F}$.
- if $\phi \in \mathscr{F}$ and $\psi \in \mathscr{F}$, then $(\phi) \vee (\psi), (\phi) \wedge (\psi), (\phi) \rightarrow (\psi)$, and $(\phi) \leftrightarrow (\psi) \in \mathscr{F}$.
- if $\phi \in \mathscr{F}$ and v is a variable, then $\exists v(\phi)$ and $\forall v(\phi) \in \mathscr{F}$. We say ϕ is the scope of the $\exists$ or $\forall$ quantifier.

A formula is quantifier-free if it does not contain any quantifiers. When no ambiguity could arise, parentheses enclosing formulae can be dropped.

We need to distinguish free and bound variables in a formula.

Definition 3.3 – Free and bound variables, and sentence

A variable v is free in a formula if it is not in the scope of any $\exists$ quantifier. Otherwise it is bound. If a formula contains no free variables, it is a sentence.

When talking about the number of free variables in a formula, we mean the number of distinct free variables. For a formula ϕ with k free variables, we often write out $\phi(v_1, v_2, ..., v_k)$ to display the free variables explicitly. It is also common to write $\phi(\vec{v})$ as a shorthand of $\phi(v_1, v_2, ..., v_k)$. For the sake of clarity, we do not allow the same variable to occur as both a free variable and a bound variable in a formula (like in $x = y \wedge \exists x(x + y = z)$) or meaningless quantification (like in $\exists x(y = z)$). The decision procedure will construct automata for a restricted class of $\mathscr{L}_S$-formulae, those in automata normal forms.

Definition 3.4 – Automata Atom

An atomic $\mathscr{L}_S$-formula is an automata atom if it has one of the following forms:

- $v_1 = v_2$
- $S(v_1) = v_2$,
- $v_1 + v_2 = v_3$

where v_1, v_2 and v_3 must be variables, not general terms.

Definition 3.5 – Automata Normal Form

An $\mathscr{L}_S$ formula is said to be in automata normal form (ANF) if it is in prenex form (where all quantifiers appear at the beginning of the formula, followed by a quantifier-free formula), and every atomic formula in it is an automata atom.

We now show that this restriction is superficial in the sense that for any $\mathscr{L}_S$ formula, there exists an equivalent one that is in ANF. We do not discuss semantics of first-order logic here. Intuitively, the formulae are supposed to be talking about natural numbers and the sequence s. When we say that two formulae (with the same free variables) are equivalent, we mean that under identical assignment of free variables to natural numbers, one is true iff the other is true. Every first-order formula can be put into prenex form, so it suffices to show that for any atomic $\mathscr{L}_S$-formula, there exists an equivalent $\mathscr{L}_S$ formula in which all atomic formulae are automata atoms. Then for any formula, we can eliminate non-automata atoms by replacing them with formulae that have only automata atoms. Then, we put the formula in prenex form, turning it into ANF. For more details about semantics of first-order logic, logical equivalence, and prenex form, see [2].

Theorem 3.1 – Eliminating Non-Automata Atoms

For any atomic $\mathscr{L}_S$-formula, there exists an equivalent $\mathscr{L}_S$ formula in which all atoms are automata atoms.

Proof. Consider an arbitrary atomic $\mathscr{L}_S$-formula $t_1 = t_2$. Notice that it is equivalent to the formula $\exists x, (t_1 = x \wedge t_2 = x)$. So, we only need to show that for every atomic $\mathscr{L}_S$-formula of the form $t = x$, there exists an equivalent $\mathscr{L}_S$ formula with only automata atoms.

Consider a term t, we know S occurs in it finitely many times. By induction on this finite number k, we prove that $t = x$ is equivalent to an $\mathscr{L}_S$ formula in which all non-automata atomic formulae do not contain S. Base case is trivial. Inductive step: Consider the first S in t that is not applied to a variable, it is applied to some term r, and $S(r)$ is a subterm of t. Now take fresh variables u, v and consider the formula $\exists v \exists u(t' = x \wedge (S(u) = v \wedge u = r))$, where t' is the term obtained by replacing all the occurrences of $S(r)$ with v in t. This is equivalent to $t = x$. But notice that $S(u) = v$ is compliant, and the other two atomic terms have fewer

k many S: We took at least one S away from t', and r has one less S than $S(r)$. Applying inductive hypothesis completes this subproof.

So, we are left with the task of showing that for every term t not containing S, there exists an $\mathscr{L}_S$ formula with only automata atoms that is equivalent to $t = x$. This can be carried out by induction on the number of $+$ in t. Base case is trivial. Inductive step: Find a subterm $t_1 + t_2$ that is not the same as t (if there are none, then there is only a single $+$, and $t = x$ is an automata atom already). Take a fresh variable v and consider the formula $\exists v(t' = x \wedge t_1 + t_2 = v)$, where t' is obtained by replacing all occurrences of $t_1 + t_2$ by v in t. This is equivalent to $t = x$, but both t' and $t_1 + t_2$ have less $+$ than t, so rewriting the formula with inductive hypothesis closes the proof. □

Corollary 3.1 – ANF Theorem

Every $\mathscr{L}_S$-formula is equivalent to one in automata normal form.

In the rest of this chapter, we present a decision procedure for the b-automatic sequence s. For a $\mathscr{L}_S$-formula $\phi(\vec{v})$ in automata normal form with m free variables $\vec{v}$, we build a DFA with alphabet Σ_b^m that accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $\phi(\vec{a})$ is true. Because ANF formulae are in prenex form, we consider the case where $\phi(\vec{v})$ is quantifier-free first. Fixing Σ_b^m as the input alphabet, we construct an automaton for each of its subformula $\psi(\vec{v'})$. Note that every variable in $\vec{v'}$ is in $\vec{v}$, but it could be the case that some variables in $\vec{v}$ are not in $\vec{v'}$. If that happens, with v' containing $n < m$ variables, we have an injection $I : \{1,2,...,n\} \to \{1,2,...,m\}$ such that $v'_i = v_{I(i)}$ for every $i \leq n$. The automaton we build for $\psi(\vec{v'})$ accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $\psi(\vec{a'})$ is true, where $\vec{a'}$ is the n-tuple such that $a'_i = a_{I(i)}$. So, the automaton ignores all irrelevant indices (corresponding to variables not in $\vec{v'}$) when reading input, and functions essentially as an automaton over Σ_b^n. For convenience, we generalize the notation for formulae with free variables, to allow $\psi(\vec{v})$ even when some variables in $\vec{v}$ do not appear in ψ. So, $\psi(\vec{v})$ is the same as $\psi(\vec{v'})$. In contexts like the one above where the assignment of natural numbers to variables is clear, we also take $\psi(\vec{a})$ to be the same as $\psi(\vec{a'})$. So, we can say that we want to build a DFA for $\psi(\vec{v})$ that accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $\psi(\vec{a})$ is true, even when ψ actually does not contain some variables in $\vec{v}$. We call such a DFA $M_{\psi(\vec{v})}$. We now build $M_{\phi(\vec{v})}$ by induction on ϕ.

3.2 Boolean Automata

Before dealing with atomic formulae, we first build automata for boolean connectives. To build $M_{\neg\phi(\vec{v})}$ from $M_{\phi(\vec{v})}$, we take the complement of the accepting set.

> **Definition 3.6 – Complement Automaton**
> Given a DFA $M = (\Sigma, Q, q_0, \delta, F)$, its complement is the DFA $\overline{M} = (\Sigma, Q, q_0, \delta, Q \setminus F)$

Because the accepting set is represented as a function from Q to $\{\top, \bot\}$ in Lean, we can formalize this by flipping the value of the function.

```
def DFA.negate (dfa: DFA α state) : DFA α state := {
  transition := dfa.transition
  start := dfa.start
  output := fun x => ! dfa.output x
}
```

It is obvious that whenever $\overline{M}$ accepts, M does not.

> **Theorem 3.2 – Correctness of Complement**
> Fix $M = (\Sigma, Q, q_0, \delta, F)$. For any $w \in \Sigma^*$, $\overline{M}$ accepts w if and only if M does not accept w.

Proof. $\overline{M}$ accepts w iff $\delta(q_0, x) \in Q \setminus F$ iff $\delta(q_0, x) \notin F$ iff M does not accept w. □

The proof is also straightforward in Lean.

```
theorem negate_iff (dfa : DFA α state) (s : List α ) :
  (dfa.negate).eval s = true ↔ ¬dfa.eval s = true
```

Let $M_{\neg\phi(\vec{v})}$ be $\overline{M_{\phi(\vec{v})}}$. The correctness of $M_{\neg\phi(\vec{v})}$ follows from the correctness of the complement automaton.

Corollary 3.2 – Correctness of $M_{\neg\phi(\vec{v})}$

If $M_{\phi(\vec{v})}$ accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $\phi(\vec{a})$ is true, then $M_{\neg\phi(\vec{v})}$ accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $\neg\phi(\vec{a})$ is true.

We also need conjunction and disjunction. To build $M_{(\phi\wedge\psi)(\vec{v})}$ or $M_{(\phi\vee\psi)(\vec{v})}$ from $M_{\phi(\vec{v})}$ and $M_{\psi(\vec{v})}$, we need to take their product.

Definition 3.7 – Product Construction for DFA

Let

$$M_1 \;=\; (\Sigma,\, Q_1,\, q_1,\, \delta_1,\, F_1), \qquad M_2 \;=\; (\Sigma,\, Q_2,\, q_2,\, \delta_2,\, F_2)$$

be deterministic finite automata over the same input alphabet Σ. Their product automaton is the DFA

$$M_1 \times M_2 \;=\; \big(\Sigma,\, Q_1 \times Q_2,\, (q_1,q_2),\, \delta_\times,\, F_\times\big),$$

where, for every $a \in \Sigma$ and $(r_1,r_2) \in Q_1 \times Q_2$,

$$\delta_\times\big((r_1,r_2),a\big) \;=\; \big(\delta_1(r_1,a),\, \delta_2(r_2,a)\big).$$

The accepting set $F_\times$ is chosen to realise the desired Boolean combination of $L(M_1)$ and $L(M_2)$:

$$\text{(Intersection)} \quad F_\cap \;=\; F_1 \times F_2$$

$$\text{(Union)} \quad F_\cup \;=\; (F_1 \times Q_2) \cup (Q_1 \times F_2).$$

If $|Q_1| = m$ and $|Q_2| = n$, the product automaton has mn states.

We formalize the intersection product and the union product separately.

```
def DFA.intersection (dfa1 : DFA α state1) (dfa2 : DFA α state2) : DFA α (
    state1 × state2) := {
  transition := fun a q => ⟨dfa1.transition a q.1, dfa2.transition a q.2⟩,
  start := (dfa1.start, dfa2.start),
  output := fun (q1, q2) => dfa1.output q1 && dfa2.output q2
}

def DFA.union (dfa1 : DFA α state1) (dfa2 : DFA α state2) : DFA α (state1
    × state2) := {
  transition := fun a q => ⟨dfa1.transition a q.1, dfa2.transition a q.2⟩,
  start := (dfa1.start, dfa2.start),
  output := fun (q1, q2) => dfa1.output q1 || dfa2.output q2
}
```

Theorem 3.3 – Correctness of Product Construction

The product construction is correct. In other words, depending on the choice of $F_\times$,

$$F_1 \times F_2 \implies M_1 \times M_2 \text{ accepts } w \text{ iff } (M_1 \text{ accepts } w \text{ and } M_2 \text{ accepts } w),$$
$$(F_1 \times Q_2) \cup (Q_1 \times F_2) \implies M_1 \times M_2 \text{ accepts } w \text{ iff } (M_1 \text{ accepts } w \text{ or } M_2 \text{ accepts } w).$$

Proof. This is immediate from the way that the state set and transition function are defined for the product. By induction, we have $\hat{\delta}_\times\big((r_1, r_2), w\big) = \big(\hat{\delta}_1(r_1, w),\ \hat{\delta}_2(r_2, w)\big)$. for any word w in the shared alphabet. It then follows that if the accepting set is $F_1 \times F_2$, we have $f_{M_1 \times M_2} = \top$ iff ($f_{M_1} = \top$ and $f_{M_2} = \top$). Similarly, if the accepting set is $(F_1 \times Q_2) \cup (Q_1 \times F_2)$, we have $f_{M_1 \times M_2} = \top$ iff ($f_{M_1} = \top$ or $f_{M_2} = \top$). □

We also write $M_1 \cap M_2$ and $M_1 \cup M_2$ for the products. The correctness of them has been established in Lean.

```
theorem intersection_iff (dfa1 : DFA α state1) (dfa2 : DFA α state2) (s :
    List α ) :
  (dfa1.intersection dfa2).eval s = true ↔ (dfa1.eval s = true ∧ dfa2.eval s
      = true)

theorem union_iff (dfa1 : DFA α state1) (dfa2 : DFA α state2) (s : List α )
    :
  (dfa1.union dfa2).eval s = true ↔ (dfa1.eval s = true ∨ dfa2.eval s = true
     )
```

Take $M_{\phi(\vec{v})} \cap M_{\psi(\vec{v})}$ for $M_{(\phi \wedge \psi)(\vec{v})}$, the correctness of product construction implies the correctness of $M_{(\phi \wedge \psi)(\vec{v})}$.

> **Corollary 3.3 – Correctness of $M_{(\phi \wedge \psi)(\vec{v})}$**
>
> If $M_{\phi(\vec{v})}$ accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $\phi(\vec{a})$ is true and $M_{\psi(\vec{v})}$ accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $\psi(\vec{a})$ is true, then $M_{(\phi \wedge \psi)(\vec{v})}$ accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $(\phi \wedge \psi)(\vec{a})$ is true.

Similarly, we can take $M_{\phi(\vec{v})} \cup M_{\psi(\vec{v})}$ for $M_{(\phi \vee \psi)(\vec{v})}$, but this is not necessary, because $\{\neg, \wedge\}$ is already a complete set of boolean connectives, allowing us to express $\vee$, $\rightarrow$, and $\leftrightarrow$.

3.3 Atomic Automata

We now provide automata for automata atoms. The first case is $M_{(v_i = v_j)(\vec{v})}$ for $i, j \leq m$, where the input alphabet is Σ_b^m. The automaton merely has to remember whether it has already seen a pair of unequal digits in positions i and j.

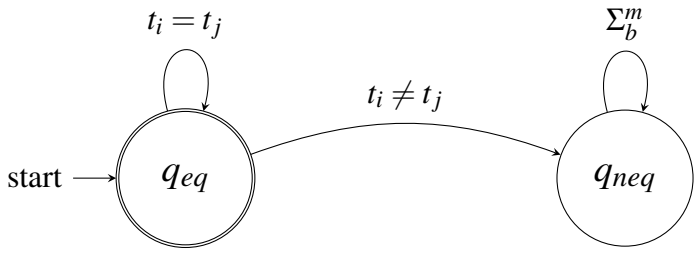
Let

$$M_{(v_i = v_j)(\vec{v})} = \left(\Sigma_b^m,\ \{q_{eq}, q_{neq}\},\ q_{eq},\ \delta,\ \{q_{eq}\}\right),$$

where the transition function

$$\delta(q_{eq},t) = \begin{cases} q_{eq} & \text{if } t_i = t_j, \\ q_{neq} & \text{otherwise,} \end{cases} \qquad \delta(q_{neq},t) = q_{neq} \quad \text{for all } t.$$

This automaton is illustrated in the following transition diagram.



It is easily implemented in Lean.

```
def eqBase (b m: ℕ) (i j : Fin m): DFA (Fin m → Fin (b + 2)) (Fin 2) := {
  transition := fun v s => match s with
    | 0 => if (v i).val == v j then 0 else 1
    | 1 => 1
  start := 0
  output := fun x => x == 0
}
```

Theorem 3.4 – Correctness of the Equality Automaton

$M_{(v_i=v_j)(\vec{v})}$ accepts base-b words of an m-tuple of natural numbers $\vec{a}$ iff $(v_i = v_j)(\vec{a})$ is true, i.e. $a_i = a_j$.

Proof. ($\Rightarrow$) If $a_i = a_j$, then they will have the same base-b digits, so in the word, every $t_i = t_j$, and the automaton will never leave q_{eq}. Therefore, the automaton will accept in the end. ($\Leftarrow$) If the automaton accepts, that means it has been staying at q_{eq} (once it reaches q_{neq}, it cannot come back), so every $t_i = t_j$ in the word, meaning the base-b digits of a_i and a_j are the same, so $a_i = a_j$. □

This has been formally verified, for the canonical (no leading zeros) word of $\vec{v}$.

```
theorem eqBase_iff (b m: ℕ) (v : Fin m → ℕ) (i j : Fin m):
  v i = v j ↔ (eqBase b m i j).eval (toWord v b)
```

But we should show this for not only the canonical base-b word of $\vec{v}$, but for every possible base-b word. More generally, every automata we construct should not distinguish between different base-b words of a tuple. Recall that all base-b words of a tuple differ only by leading zeros.

> **Definition 3.8 – Zero-respecting Automata**
>
> We say that a DFA M over Σ_b^m is zero-respecting when for every word z, M accepts z if and only if M also accepts 0^*z. Here, 0 denotes the m-tuple of zeros.

This is formalized as a proposition in Lean.

```
def DFA.respectZero (dfa : DFA (Fin m → Fin (b + 2)) state) : Prop := ∀ (x :
    List (Fin m → Fin (b + 2))), ∀ n, dfa.eval x ↔ dfa.eval (padZeros n x)
```

To completely verify the correctness of the equality automaton, we have proven in Lean that it is zero-respecting.

```
theorem equality_respectZero : (eqBase b m i j).respectZero
```

Moreover, we also verified that if $M_{\phi(\vec{v})}$ and $M_{\psi(\vec{v})}$ are zero-respecting, then so are $M_{\neg\phi(\vec{v})}$, $M_{(\phi\wedge\psi)(\vec{v})}$, and $M_{(\phi\vee\psi)(\vec{v})}$, for the boolean automata we constructed previously.

```
theorem DFA.negate_respectZero (dfa : DFA (Fin m → Fin (b + 2)) state) (h:
    dfa.respectZero) : dfa.negate.respectZero

theorem DFA.intersection_respectZero (dfa1 : DFA (Fin m → Fin (b + 2))
    state1) (dfa2 : DFA (Fin m → Fin (b + 2)) state2) (h1: dfa1.respectZero)
    (h2: dfa2.respectZero) : (dfa1.intersection dfa2).respectZero

theorem DFA.union_respectZero (dfa1 : DFA (Fin m → Fin (b + 2)) state1) (
    dfa2 : DFA (Fin m → Fin (b + 2)) state2) (h1: dfa1.respectZero) (h2: dfa2
    .respectZero) : (dfa1.union dfa2).respectZero
```

We now move on to build $M_{(S(v_i)=v_j)(\vec{v})}$, by exploiting the fact that s only takes values in a finite $\Delta \subset \mathbb{N}$. Observe that if we extend the language of $\mathscr{L}_S$ by elements[1] of $\Delta \subset \mathbb{N}$, referring to those natural numbers, then $S(v_i) = v_j$ is equivalent to $\bigvee_{k\in\Delta}(S(v_i) = k \wedge v_j = k)$. Because we already have boolean automata, it suffices to build $M_{(S(v_i)=k)(\vec{v})}$ and $M_{(v_j=k)(\vec{v})}$ for every $k \in \Delta$.

Building $M_{(v_j=k)(\vec{v})}$ requires the base-b digits of k. Let $(k)_b = d_1 d_2 \cdots d_t$

Because leading zeros are irrelevant, the automaton only needs to verify that, after stripping an arbitrary number of leading zeros on the j-th index, the remaining word is exactly $(k)_b$.

Let

$$M_{(v_j=k)(\vec{v})} = \left(\Sigma_b^m, Q, q_0, \delta, \{q_t\}\right),$$

where the state set is

$$Q = \{q_0, q_1, \ldots, q_t, q_\perp\}.$$

Intuitively q_r means "matched the first r significant digits of $(k)_b$"; q_0 processes leading

[1] We are actually adding constants which are being interpreted as those elements, but it is convenient to blur the difference here.

zeros and $q_\perp$ is the rejecting state. For a letter $t = (t_0, \ldots, t_{m-1}) \in \Sigma_b^m$ put

$$\delta(q_r, t) = \begin{cases} q_r & \text{if } r = 0 \text{ and } t_j = 0, \\ q_{r+1} & \text{if } 0 \leq r < t \text{ and } t_j = d_{r+1}, \\ q_0 & \text{if } k = 0 \text{ and } t_j = 0, \\ q_\perp & \text{otherwise} \end{cases} \qquad \delta(q_\perp, t) = q_\perp.$$

Theorem 3.5 – Correctness of $M_{(v_j=k)(\vec{v})}$

$M_{(v_j=k)(\vec{v})}$ accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $v_j = k$.

Proof. By construction, the only way for $M_{(v_j=k)(\vec{v})}$ to end up in its accepting state q_t is for it to stay at q_0, then visit $q_1, q_2, ..., q_k$ sequentially without repetition. In other words, $M_{(v_j=k)(\vec{v})}$ accepts a word iff the j-th element of every tuple consists of exactly the digits of k in base b, with leading zeros, iff $v_j = k$. □

To construct $M_{(S(v_i)=k)(\vec{v})}$, let $M_s = (Q, \Sigma_b, \delta, q_0, \Delta, \omega)$ be the DFAO that generates the b-automatic sequence s. We obtain $M_{S(v_i)} = (Q, \Sigma_b^m, \delta^i, q_0, \Delta, \omega)$ by letting $\delta^i(t, q) = \delta(t_i, q)$, and then take $M_{(S(v_i)=k)(\vec{v})} = (Q, \Sigma_b^m, \delta^i, q_0, F)$ where

$$F = \{q \in Q | \omega(q) = k\}$$

We claim that this $M_{(S(v_i)=k)(\vec{v})}$ is correct.

Theorem 3.6 – Correctness of $M_{(S(v_i)=k)(\vec{v})}$

$M_{(S(v_i)=k)(\vec{v})}$ accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $S(a_i) = k$.

Proof. Let w be a base-b word of $\vec{a}$. $M_{(S(v_i)=k)(\vec{v})}$ accepts w iff $f_{M_{S(v_i)}}(w) = k$ iff $S(a_i) = k$. □

Verification of this depends on the specific choice of s that one want to implement the decision procedure on. However, the last step of building $M_{(S(v_i)=k)(\vec{v})}$ out of $M_{S(v_i)}$ involves an interesting general operation that turns a DFAO into a DFA, which we have verified.

```
def DFAO.toDFA (dfao : DFAO α state out) (o: out) [DecidableEq out]: DFA α
    state := {
  transition := dfao.transition,
  start := dfao.start,
  output := fun s => (dfao.output s) == o
}

theorem DFAO.toDFA_eval (dfao : DFAO α state out) (o: out) (s : List α ) [
    DecidableEq out] :
    (dfao.toDFA o).eval s = (dfao.eval s == o)
```

We are left with building $M_{(v_i+v_j=v_k)(\vec{v})}$. The automaton checks addition digit by digit, considering carries.

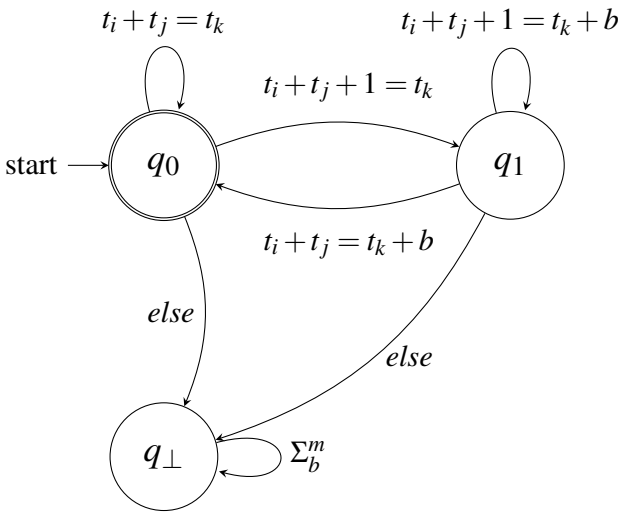
Let

$$M_{(v_i+v_j=v_k)(\vec{v})} = \left(\Sigma_b^m,\, Q,\, q_0,\, \delta,\, \{q_0\}\right),$$

where $Q = \{q_0, q_1, q_\perp\}$, q_0 means "no carry needed", and q_1 means "need carry 1". with transition function defined for $t = (t_0, \ldots, t_{m-1}) \in \Sigma_b^m$, as

$$\delta(q_0,t) = \begin{cases} q_0 & t_i + t_j = t_k, \\ q_1 & t_i + t_j + 1 = t_k, \\ q_\perp & \text{otherwise;} \end{cases} \qquad \delta(q_1,t) = \begin{cases} q_1 & t_i + t_j + 1 = t_k + b, \\ q_0 & t_i + t_j = t_k + b, \\ q_\perp & \text{otherwise,} \end{cases} \qquad \delta(q_\perp,t) = q_\perp.$$

Intuitively, in state q_0 we check the digit sum without an incoming carry; if it needs a carry we go to q_1 for the next more significant digits. In state q_1 we go to the rejecting state if no carry is produced, and check if another carry is needed from the next digits.



The addition-checking automaton is implemented in Lean exactly as above:

```
def addBase (b m : ℕ) (i j k : Fin m) :
    DFA (Fin m → Fin (b + 2)) (Fin 3) :=
{ transition := fun f s =>
    match s with
    | 0 =>
        if (f i).val + f j == f k then 0 else
        if (f i).val + f j + 1 == f k then 1 else 2
    | 1 =>
        if (f i).val + f j + 1 == f k + b then 1 else
        if (f i).val + f j == f k + b then 0 else 2
    | 2 => 2,
  start := 0,
  output := fun s => s == 0 }
```

Theorem 3.7 – Correctness of the Addition Automaton

For every m-tuple of natural numbers $\vec{a}$,

$$M_{(v_i+v_j=v_k)(\vec{v})} \text{ accepts a base-}b \text{ word of } \vec{a} \text{ iff } a_i + a_j = a_k.$$

Proof. Leading zeros clearly do not affect acceptance of the automaton, so we can as-

sume that we are working with a canonical word of a tuple of natural numbers. Write n for the length of the input word, and let the digits in the i,j,k-th indices in the tuples be $x_{n-1}\dots x_0$, $y_{n-1}\dots y_0$, and $z_{n-1}\dots z_0$. The automaton reads the word left-to-right ($x_{n-1}, y_{n-1}, z_{n-1}$ first).

($\Rightarrow$) Because the automaton accepts in the end, we can assume it never goes to $q_\perp$. By induction, we prove the following invariant:

After reading the first ℓ letters ($0 \le \ell \le n$) the automaton is in state q_r, $r \in \{0,1\}$ if and only if

$$\sum_{p=n-\ell}^{n-1} b^p \big(x_p + y_p\big) \;=\; \sum_{p=n-\ell}^{n-1} b^p z_p \;+\; r b^{n-\ell}. \tag{$\star_\ell$}$$

Base case is $\ell = 0$. Before the first letter the automaton is in q_0, we have the empty sum on both sides and the initial state is q_0, i.e. $r = 0$, so $(\star_0)$ holds.

Inductive step: Assume we are currently at state q_r and the next letter supplies digits $x := x_{n-\ell-1}$, $y := y_{n-\ell-1}$, $z := z_{n-\ell-1}$. The transition function does an exhaustive case analysis on the next state r':

$$q_{r'} = \delta(q_r, (\dots, x, \dots, y, \dots, z, \dots)) = \begin{cases} 0 & \text{if } x+y = z \text{ and } r = 0, \\ 1 & \text{if } x+y+1 = z \text{ and } r = 0, \\ 0 & \text{if } x+y = z+b \text{ and } r = 1, \\ 1 & \text{if } x+y+1 = z+b \text{ and } r = 1, \\ q_\perp & \text{otherwise.} \end{cases}$$

A direct check of the four non-rejecting cases shows that it is always the case that

$$x b^{n-\ell-1} + y b^{n-\ell-1} \;=\; z b^{n-\ell-1} + r b^{n-\ell} - r' b^{n-\ell-1}.$$

Adding this to $(\star_\ell)$, which we have by IH, yields $(\star_{\ell+1})$. After reading the final letter and setting $r = 0$, $(\star_\ell)$ is equivalent to $a_i + a_j = a_k$.

($\Leftarrow$) We prove the contrapositive. Suppose the automaton does not accept a base-b word of $\vec{a}$. This means that in reading the word it transitioned into $q_\perp$. Assume it first reached $q_\perp$

after reading the $l+1$-th letter (this $+1$ is safe because it cannot reach $q_\perp$ without reading anything), then the invariant in the forward direction

$$\sum_{p=n-\ell}^{n-1} b^p \big(x_p + y_p\big) \;=\; \sum_{p=n-\ell}^{n-1} b^p z_p \;+\; r\,b^{n-\ell}. \tag{$\star_\ell$}$$

still holds after reading the l-th letter and in state q_r, $r \in \{0,1\}$. We case on r, and abbreviate $x := x_{n-\ell-1}$, $y := y_{n-\ell-1}$, $z := z_{n-\ell-1}$. When $r = 0$, we know neither $x+y=z$ nor $x+y+1=z$, so $x+y>z$ or $x+y<z-1$. Together with the simplified $(\star_\ell)$, a straightforward computation shows that it is impossible for the rest less significant digits to make up the difference, so $a_i + a_j \neq a_k$. The case where $r=1$ is similar.

□

It is easy to check that $M_{(v_j=k)(\vec{v})}$, $M_{(S(v_i)=k)(\vec{v})}$, and $M_{(v_i+v_j=v_k)(\vec{v})}$ are zero-respecting. In particular, $M_{(S(v_i)=k)(\vec{v})}$ is zero-respecting because M_s is, by definition of b-automatic sequences. At the time of writing this thesis, we have not yet verified these atomic automata.

3.4 Projection and Fixing Leading Zeros

With the boolean and atomic automata in place, we are done with the quantifier-free part of $\mathscr{L}_S$ formulae. Now, given a formula $\phi(\vec{v})$ with $m+1$ free variables $\vec{v}$, its automaton $M_{\phi(\vec{v})}$ with alphabet Σ_b^{m+1}, and a variable v_i in $\vec{v}$, we want to build a DFA $M_{\exists v_i(\phi(\vec{v}))}$ with alphabet Σ_b^m that accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $\exists v_i(\phi(\vec{a_i^v}))$ is true, where $\phi(\vec{a_i^v})$ is $\phi(a_1,...,a_{i-1},v_i,a_i,...,a_m)$, assigning the $\vec{a}$ to the variables in $\vec{v}$ other than v_i sequentially. The automata operation we are going to use is called projection.

Definition 3.9 – Projection

Given a DFA $M \;=\; \big(\Sigma_b^{m+1},\, Q,\, q_0,\, \delta,\, F\big)$, Let

$$\pi_i(M) \;=\; \big(\Sigma_b^m,\, Q,\, \{q_0\},\, \Delta_i,\, F\big),$$

where $\pi_i(M)$ is an NFA whose transition relation is $\Delta_i \colon Q \times \Sigma_b^m \to 2^Q$ is

$$\Delta_i(q,t) \;=\; \{\delta(q, \mathsf{insert}_i(x,t)) \;\mid\; x \in \Sigma_b\},$$

and $\mathsf{insert}_i(x,t)$ means inserting the symbol x as the i-th component of the m-tuple t, moving $t_i,\ldots,t_m$ one position towards the right, resulting in an $m+1$ tuple. Intuitively, $\pi_i(M)$ nondeterministically guesses the digits of the i-th natural number, and simulates M on the completed word.

This requires a careful formalization in Lean.

```
def project (i : Fin (m + 1)) (dfa : DFA (Fin (m + 1) → Fin (b + 2)) state)
    [DecidableEq state] :
  NFA (Fin m → Fin (b + 2)) state := {
  transition :=
  fun a q => ⟨(List.map (fun (x : Fin (b + 2)) => dfa.transition (Fin.
      insertNth i x a) q)
    (FinEnum.toList (Fin (b + 2)))).dedup, by apply List.nodup_dedup⟩
  start := ⟨[dfa.start], List.nodup_singleton dfa.start⟩
  output := dfa.output
}
```

The words that $\pi_i(M)$ accepts satisfy the following property.

Theorem 3.8 – Projection Property

Let $M = \left(\Sigma_b^{m+1}, Q, q_0, \delta, F\right)$ be a DFA and fix an index $i \le m$. For a letter $t \in \Sigma_b^{m+1}$ write $\pi_i(t) \in \Sigma_b^m$ for the tuple obtained by deleting the i-th component, and extend π_i to words by mapping it letter-wise. Then for every word $w \in (\Sigma_b^m)^*$,

$\pi_i(M)$ accepts w iff there exists $w' \in (\Sigma_b^{m+1})^*$ such that $w = \pi_i(w')$ and M accepts w'.

Proof. ($\Rightarrow$) Assume $\pi_i(M)$ accepts w. By definition of the NFA $\pi_i(M)$ there is a sequence of symbols $x_0 x_1 \ldots x_{|w|-1}$ in Σ_b such that if we insert x_k as the i-th component of the k-th letter w_k we obtain a word $w' := \mathsf{insert}_i(x_0, w_0)\,\mathsf{insert}_i(x_1, w_1)\ldots$ which is clearly accepted by M. By construction $w = \pi_i(w')$, establishing the existence.

($\Leftarrow$) Conversely, suppose there exists w' with $w = \pi_i(w')$ and M accepts w'. By induction on w', we can prove that $\hat{\delta}(q_0, w') \in \hat{\Delta}_i(\{q_0\}, w)$, because for every letter a in w and every state q, we have that $\{\delta(q, \mathsf{insert}_i(x,a)) \mid x \in \Sigma_b\} = \Delta_i(q,a)$. Therefore, $\pi_i(M)$ accepts w. □

And we have this property proven in Lean.

```
theorem project_eval_iff [DecidableEq state](dfa : DFA (Fin (m + 1) → Fin (b
    + 2)) state) (i : Fin (m + 1)) (l : List (Fin m → Fin (b + 2))) : (
   project i dfa).eval l ↔ ∃ (l : List (Fin (m + 1) → Fin (b + 2))) , l = l
   .map (Fin.removeNth i) ∧ dfa.eval l
```

By the subset construction, we can treat $\pi_i(M)$ as a DFA. In the rest of this section, we do not distinguish DFA and NFA. However, $\pi_i(M_{\phi(\vec{v})})$ is not yet $M_{\exists v_i(\phi(\vec{v}))}$, because it is not zero-respecting. Consider the formula $\exists v_3, v_1 + v_2 = v_3$. $M_{\exists v_3, v_1+v_2=v_3}$ in base-2 should accept the word $(1,1)$ because $1+1=2$. Consider, on the other hand, the projection $\pi_{v_3}(M_{v_1+v_2=v_3})$ During its simulation it has to guess one symbol for the v_3-position at every letter. If w has length 1 the guess is forced to a single digit $x \in \{0,1\}$. But addition automaton for $M_{v_1+v_2=v_3}$ in the previous section shows that with inputs $a=1$ and $b=1$ no single digit x yields an accepting transition:

$$1+1=0 \text{ (carry 1)}, \qquad 1+1=2 \text{ (not a base-2 digit)}.$$

Consequently, every run of the projected automaton on w is trapped in the rejecting state, so $\pi_{v_3}(M_{v_1+v_2=v_3})$ rejects $(1,1)$. On the other hand, if the input word is $(0,0)(1,1)$, then $\pi_{v_3}(M_{v_1+v_2=v_3})$ would accept, because the additional leading zeros allows the automaton to guess $(2)_2 = 10$. More generally, we have the following theorem.

> **Theorem 3.9 – $\pi_i(M_{\phi(\vec{v})})$ is Partially Correct**
>
> If $M_{\phi(\vec{v})}$ accepts the base-b word of the $m+1$ tuple of natural numbers $\vec{a}$, then $\pi_i(M_{\phi(\vec{v})})$ will accept some base-b word of the m-tuple $\mathsf{remove}_i(\vec{v})$, where $\mathsf{remove}_i(\vec{v}) = (v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{m+1})$

Proof. By removing the i-th index of the base-b word of $\vec{v}$, we obtain a word that is a base-b word of the m-tuple $\mathsf{remove}_i(\vec{v})$, which $\pi_i(M_{\phi(\vec{v})})$ accepts by the projection property. □

Intuitively, this means that if we pad enough number of leading zeros, then $\pi_i(M_{\phi(\vec{v})})$ will accept a a base-b word of the m-tuple $\text{remove}_i(\vec{v})$. We now show that it is possible to fix the problem of $\pi_i(M_{\phi(\vec{v})})$. Although $\pi_i(M_{\phi(\vec{v})})$ is generally not zero-respecting, it has a weaker property that we can exploit.

> **Definition 3.10 – Zero-accepting Automata**
>
> We say that a DFA M over Σ_b^m is zero-accepting when for every word z, if M accepts z then M also accepts 0^*z. Here, 0 denotes the m-tuple of zeros.

```
def DFA.acceptZero (dfa : DFA (Fin m → Fin (b + 2)) state) : Prop := ∀ (x :
    List (Fin m → Fin (b + 2))), (dfa.eval x → ∀ n, dfa.eval (padZeros n x))
```

As demonstrated in the last section, we can assume that $M_{\phi(\vec{v})}$ is zero-respecting, and show that $\pi_i(M_{\phi(\vec{v})})$ is zero-accepting.

> **Theorem 3.10 – $\pi_i(M_{\phi(\vec{v})})$ is Zero-accepting**
>
> If a DFA M with alphabet Σ_b^{m+1} is zero-respecting, then $\pi_i(M)$ is zero-accepting. So, given $M_{\phi(\vec{v})}$ is zero-respecting, $\pi_i(M_{\phi(\vec{v})})$ is zero-accepting.

Proof. Suppose $\pi_i(M)$ accepts a word w, then by the projection property there exists a corresponding word w' that M accepts. Because M is zero-respecting, it accepts $0^*w'$ Applying the projection property again proves the theorem. □

In Lean, we have a similar `acceptZero` proposition for NFA, and this theorem.

```
theorem project_acceptZero [DecidableEq state] (dfa : DFA (Fin (n + 1) → Fin
     (b + 2)) state) (h: DFA.acceptZero dfa) (m : Fin (n + 1)) : NFA.
    acceptZero (project m dfa)
```

The crucial insight now is that there is a bounded number of leading zeros we need to pad, for $\pi_i(M_{\phi(\vec{v})})$ to accept a a base-b word of the m-tuple $\text{remove}_i(\vec{v})$, where the word of $\vec{v}$ is accepted by $M_{\phi(\vec{v})}$.

Theorem 3.11 – Bounded Acceptance

Suppose a DFA M with alphabet Σ_b^m is zero-accepting. Then there exists a natural number n, such that if M accepts $0^m z$ for some arbitrary m and z, then M accepts $0^n z$.

Proof. Use the n given by the pumping lemma.

Case 1: If $m \leq n$, then by assumption M accepts $0^n z$.

Case 2: If $m > n$, by pumping lemma there exists a decomposition $0^m z = uvw$ with $|uv| \leq n$ and $|v| \geq 1$ such that M accepts $uv^i w$ for all $i \geq 0$. Now since $|uv| \leq n$ and $m > n$, u and v must contain only zeros. Take $i = 0$, we have that M accepts $0^{m-|v|}z$, where $|v| \geq 1$. Repeating this process will eventually reduce to case 1. □

With the pumping lemma at hand, we are able to prove this in Lean.

```
theorem DFA.bounded_accept [Fintype state] (dfa : DFA (Fin m → Fin (b + 2))
    state) (x : List (Fin m → Fin (b + 2)))(h: dfa.acceptZero): ∀ z, (∃  k,
    z = padZeros k x) ∧ dfa.eval z → dfa.eval (padZeros (Fintype.card state)
     x)
```

We observe that padding leading zeros to all inputs of an automaton is equivalent to a change of initial state.

Definition 3.11 – Fix Leading Zeros

Suppose a DFA $M = \left(Q, \Sigma_b^m, \delta, q_0, F\right)$ is zero-accepting. Let n be given by the previous theorem. Define the DFA M_0 to be $(Q, \Sigma_b^m, \delta, q_0', F)$ where $q_0' = \hat{\delta}(q_0, 0^n)$.

In Lean, we define the equivalent of this for NFA.

```
def NFA.fixLeadingZeros [Fintype state] [DecidableEq state] (nfa : NFA (Fin m
     → Fin (b + 2)) state) : NFA (Fin m → Fin (b + 2)) state := {
  transition := nfa.transition
  start := nfa.transFrom (padZeros (Fintype.card ListND state) []) nfa.start
  output := nfa.output
}
```

We can show that M_0 behaves in the way we want.

> **Theorem 3.12 – Correctness of M_0**
>
> Suppose a DFA $M = \left(Q, \Sigma_b^m, \delta, q_0, F\right)$ is zero-accepting. Then M_0 satisfies the following:
>
> 1. If $0^*x \cap L(M) \neq \emptyset$, then $0^*x \subseteq L(M_0)$.
> 2. If $0^*x \cap L(M) = \emptyset$, then $0^*x \cap L(M_0) = \emptyset$

Proof. Note that by definition, M' accepts x if and only if M accepts $0^n x$.

1. For every $x \in \Sigma_k$, if $0^*x \cap L(M) \neq \emptyset$ then M accepts $0^n x$ by the previous theorem, so M' accepts x. Then by assumption $0^*x \subseteq L(M')$.

2. If $0^*x \cap L(M) = \emptyset$, then M does not accept $0^n y$ for any $y \in 0^*x$ since $0^n y \in 0^*x$. Therefore, M' does not accept any $y \in 0^*x$. □

This means that M_0 accepts all representations of a tuple of natural numbers, iff it originally accepts any. It follows immediately that M_0 is zero-respecting. In particular, $(\pi_i(M_{\phi(\vec{v})}))_0$ is zero-respecting.

> **Corollary 3.4 – $M_{\exists v_i(\phi(\vec{v}))}$ is Zero-respecting**
>
> $(\pi_i(M_{\phi(\vec{v})}))_0$ is zero-respecting.

And we have this in Lean.

```
theorem project_fix_respectZero [Fintype state][DecidableEq state] (dfa : DFA
    (Fin (m + 1) → Fin (b + 2)) state) (i : Fin (m + 1)) (h: dfa.respectZero
   ) : (project i dfa).fixLeadingZeros.respectZero
```

The final thing we need is its correctness of $M_{\exists v_i(\phi(\vec{v}))}$.

> **Theorem 3.13 – Correctness of $M_{\exists v_i(\phi(\vec{v}))}$**
>
> Suppose $M_{\phi(\vec{v})}$ is correct. Let $M_{\exists v_i(\phi(\vec{v}))} := (\pi_i(M_{\phi(\vec{v})}))_0$,
>
> i.e. projection followed by the leading–zero fix described above. For every m-tuple of

> natural numbers $\vec{a} = (a_1, \ldots, a_m)$ and for every base-b word w of $\vec{a}$ we have
>
> $M_{\exists v_i(\phi)}$ accepts w iff there exists $n \in \mathbb{N}$ such that $\phi(a_1, \ldots, a_{i-1}, n, a_i, \ldots, a_m)$ is true.

Proof. Recall that $M_{\exists v_i(\phi)}$ is zero-respecting. Hence it is enough to argue for one base-b representation of $\vec{a}$. But we have shown that $\pi_i(M_{\phi(\vec{v})})$ is partially correct. □

This is the major achievement of our verification effort in Lean.

```
theorem project_iff [Fintype state] [DecidableEq state] (v : Fin m → ℕ) (i :
    Fin (m + 1)) (dfa : DFA (Fin (m + 1) → Fin (b + 2)) state) (hres: dfa.
   respectZero):
  (project i dfa).fixLeadingZeros.eval (toWord v b) ↔ (∃ (x : ℕ), dfa.eval (
     toWord (Fin.insertNth i x v) b))
```

Replacing every $\forall$ with the equivalent $\neg\exists\neg$ in an ANF formula, we can now build $M_{\phi(\vec{v})}$ for any $\phi(\vec{v})$ by repeatedly projecting, fixing leading zeros, and complementing on automata.

Note that it is possible to reorder the variables in the input tuple according to the order in which they are quantified over in the formulae, so that in building the final automaton, it is always the first variable in the input tuple that gets projected away. In this way, it suffices to implement a simplified projection function that projects out the first element in an input tuple. As a result, we can avoid dealing with inserting or removing things from a vector, and just use `Matrix.vecCons`, `Matrix.vecHead`, and `Matrix.vecTail`.

```
def project (dfa : DFA (Fin (n + 1) → Fin (b + 2)) state) [DecidableEq state
   ] : NFA (Fin n → Fin (b + 2)) state := {
   transition := fun a q => ⟨(List.map (fun (x : Fin (b + 2)) => dfa.
       transition (Matrix.vecCons x a) q)(FinEnum.toList (Fin (b + 2)))).
       dedup, by apply List.nodup_dedup⟩,
   start := ⟨[dfa.start], List.nodup_singleton dfa.start⟩,
   output := dfa.output
 }
```

4 Conclusion and Future Work

We have established the following decision procedure for sentences in $\mathscr{L}_S$, the language of Presburger arithmetic extended by a function S for indexing into a b-automatic sequence s:

1. Put a sentence ϕ in automata normal form.
2. Build an automaton for its quantifier-free fragment, by combining atomic automata for its atoms using boolean automata constructions.
3. Build automata for each quantifier using projection, complement, and fixing leading zeros.

The final automaton M_ϕ has an empty input alphabet Σ_b^0. We know that for any $\mathscr{L}_S$-formula $\psi(\vec{v})$ in automata normal form with m free variables $\vec{v}$, the automaton we build accepts base b-words of an m-tuple of natural numbers $\vec{a}$ iff $\psi(\vec{a})$ is true. So, M_ϕ accept the trivial base b-word of the trivial 0-tuple of natural numbers iff ϕ is true. We simply inspect whether the starting state is accepting for an answer.

The decision procedure can be easily extended to include multiple b-automatic sequences, by taking care of atomic formulae properly. We have verified all automata constructions in the decision procedure that corresponds to the semantic components of first-order logic with equality. This means that our library already supports some basic theorem-proving in Lean. For example, we can show that for every natural number, there exists a natural number that is equal to it, by building and substituting automata for the Lean expression inductively. In the end, we use `rfl'` or `native_decide` to check whether the final automaton is accepting.

```
theorem demo : ¬∃ x : ℕ, ¬∃ y, x = y := by
-- Build x = y
  have : ∀ x y : ℕ, x = y ↔ (eqBase 0 2 1 0).eval (toWord ![y, x] 0 ) := by
    intro x y
    rw[eqBase_iff]
    simp_all
  simp_rw [this]
```

```
-- Build ∃ y, x = y
  have : ∀ x, ((∃  y, (eqBase 0 2 1 0).eval (toWord ![y, x] 0)) ↔ (project (
      eqBase 0 2 1 0)).fixLeadingZeros.toDFA.eval (toWord ![x] 0)) := by
    intro x
    rw[project_iff]
    exact equality_respectZero
  simp_rw [this]
-- Build ¬∃ y, x = y
  have : ∀ x, ((¬ (project (eqBase 0 2 1 0)).fixLeadingZeros.toDFA.eval (
      toWord ![x] 0) = true) ↔ (project (eqBase 0 2 1 0)).fixLeadingZeros.
      toDFA.negate.eval (toWord ![x] 0) = true) := by
    intro x
    rw[negate_iff]
  simp_rw [this]
--Build ∃ x, ¬∃ y, x = y
  have : (∃  x, DFAO.eval (project (eqBase 0 2 1 0)).fixLeadingZeros.toDFA.
      negate (toWord ![x] 0) = true) ↔ ((project (project (eqBase 0 2 1 0)).
      fixLeadingZeros.toDFA.negate).fixLeadingZeros.toDFA.eval (toWord ![] 0)
      = true) := by
      rw[project_iff]
      apply DFA.negate_respectZero
      apply project_fix_toDFA_respectZero
      exact equality_respectZero
  simp_rw[this]
-- Finally, build ¬∃ x, ¬∃ y, x = y
  have : ¬(project (project (eqBase 0 2 1 0)).fixLeadingZeros.toDFA.negate).
      fixLeadingZeros.toDFA.eval (toWord ![] 0) =
    true ↔ (project (project (eqBase 0 2 1 0)).fixLeadingZeros.toDFA.negate).
        fixLeadingZeros.toDFA.negate.eval (toWord ![] 0) = true := by
    rw[negate_iff]
  simp_rw[this]
-- Check acceptance
  rfl' -- or native_decide
```

There is much more work to be done towards a practically useful decision procedure for automatic sequences in Lean. Atomic automata other than the equality checking one need to be verified. Some meta-programming on Lean expressions to automatically generate proofs like the one shown above is desirable. The naive procedure described here also has an astronomical worst-case runtime, because the subset construction exponentiates the number of states whenever a quantifier is processed. In practice, Walnut's runtime is often much faster than the theoretical worst-case [23]. To have a relatively efficient tactic, we need to implement and verify optimizations such as the minimization algorithm for automata, used in Walnut. There are also automatic sequences depending on other number representation systems, e.g. the Fibonacci system. To prepare decision procedures for these automatic sequences, we need to implement their number representation systems.

Despite all of these remaining challenges, we hope that our library provides a solid, verified foundation for future work. It is publicly accessible at `https://github.com/Aeacu2/Automata`.

Acknowledgments

This thesis was advised by Professor Jeremy Avigad, who provided invaluable guidance both in shaping the overall direction and in attending to specific details. In addition to him, Professor Marijn J. H. Heule and Hannah Fechtner read the earlier versions of this thesis and provided many valuable feedback.

References

[1] J.-P. Allouche, N. Rampersad, and J. Shallit. Periodicity, repetitions, and orbits of an automatic sequence. *Theoretical Computer Science*, 410(30):2795–2803, 2009. A bird's eye view of theory.

[2] J. Avigad. *Mathematical Logic and Computation*. Cambridge University Press, 2022.

[3] J. Avigad. Automated reasoning for mathematics. In *Automated Reasoning: 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3–6, 2024, Proceedings, Part I*, page 3–20, Berlin, Heidelberg, 2024. Springer-Verlag.

[4] J. Avigad, L. de Moura, S. Kong, and S. Ullrich. Theorem proving in Lean 4, 2024. With contributions from the Lean Community.

[5] J. Avigad and J. Harrison. Formally verified mathematics. *Commun. ACM*, 57(4):66–75, Apr. 2014.

[6] S. Berghofer and M. Reiter. Formalizing the logic–automaton connection. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 2009.

[7] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Trees in Algebra and Programming — CAAP '96*, pages 30–43, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[8] V. Bruyàre, G. Hansel, C. Michaux, and R. Villemaire. Logic and p-recognizable sets of integers. *Bulletin of the Belgian Mathematical Society - Simon Stevin*, 1(2):191 – 238, 1994.

[9] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[10] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science (Proceedings of the 1960 International Congress)*, pages 1–11. Stanford University Press, 1962.

[11] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The lean theorem prover (system description). In *2015 Conference on Automated Deduction*, pages 378–388. Springer, Cham, July 2015.

[12] S. Gouëzel and V. Shchur. A corrected quantitative version of the Morse lemma, 10 2018.

[13] T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the Kepler conjecture, 2009.

[14] M. J. H. Heule, W. Hunt, M. Kaufmann, and N. Wetzler. Efficient, verified checking of propositional proofs. In *Interactive Theorem Proving: 8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings*, page 269–284, Berlin, Heidelberg, 2017. Springer-Verlag.

[15] M. J. H. Heule, O. Kullmann, and V. W. Marek. *Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer*, page 228–245. Springer International Publishing, 2016.

[16] M. J. H. Heule and M. Scheucher. Happy ending: An empty hexagon in every set of 30 points. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024)*, volume 14570 of *Lecture Notes in Computer Science*, pages 61–80. Springer, 2024.

[17] D. Hilbert and W. Ackermann. Grundzüge der theoretischen logik. 1928.

[18] L. d. Moura and S. Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag.

[19] H. Mousavi. Automatic theorem proving in Walnut, 2021.

[20] T. Nipkow, L. Paulson, and M. Wenzel. Isabelle/HOL: A proof assistant for higher-order logic. *LNCS*, 2002.

[21] B. Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, 1997.

[22] M. Presburger. Über die Vollständigkeit eines gewissen systems der Arithmetik ganzer Zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du I^{er} Congrès des Mathématiciens des Pays Slaves*, pages 92–101, Warszawa, 1929. Addendum p. 395.

[23] J. Shallit. *The Logical Approach to Automatic Sequences: Exploring Combinatorics on Words with Walnut*. London Mathematical Society Lecture Note Series. Cambridge University Press, 2022.

[24] J. Shallit. Rarefied Thue-Morse sums via automata theory and logic. *Journal of Number Theory*, 257:98–111, 2024.

[25] The Mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, pages 367–381. Association for Computing Machinery, 2020.

[26] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 1937.

[27] U.S. Securities and Exchange Commission. Order instituting administrative and cease-and-desist proceedings, pursuant to sections 15(b) and 21C of the Securities Exchange Act of 1934, making findings, and imposing remedial sanctions and a cease-and-desist order, in the matter of Knight Capital Americas LLC. Exchange Act Release No. 34-70694, Oct. 2013. Administrative Proceeding File No. 3-15570.