Formally verifying automata for trusted decision procedures

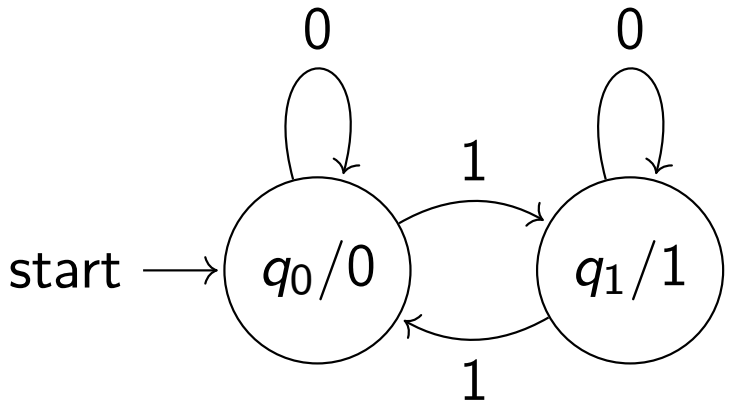
Aeacus Sheng, advised by Jeremy Avigad

A sequence and an automaton

The infinite Thue-Morse sequence, 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.

$$\mathrm{t} = (t_n)_{n\geq 0} = t_0 t_1 t_2 \ldots = 01101001100101101001011 0 \ldots$$

It can be generated by a finite automaton.



Put an index into base-2, and feed its digits into the automaton.

Walnut

Question: Is it ultimately periodic?
Answer: No, initially by an induction proof that takes pages.

In Presburger arithmetic $Th(\mathbb{N}, +)$ extended by a unary function T such that $T(i) = t_i$, we can express this as:

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

A software, Walnut, runs an automata-theoretic decision procedure to answer questions like this. It is well-known that Presburger arithmetic (LIA) is decidable, using quantifier elimination (By Presburger). This extension, however, needs automata.

Decision procedure using automata

(By Büchi) For Presburger arithmetic, build inductively, for every formula $A(\vec{x})$, an automation that accepts a representation of $\vec{a}$ iff $A(\vec{a})$ is true.

> Represent natural numbers as words (sequences) over $\Sigma_k = \{0, 1, ..., k-1\}$. n-tuples of natural numbers are words over Σ_k^n
>
> Build automata for atomic formulas, like equality and addition.
>
> Parsing a complex formula corresponds to performing operations on automata.

After the language is extended by a function for automatic sequences, the general idea still works. Just need to handle more atomic formulas.

Problem

Walnut has been used in more than 100 papers and books, proving theorems in combinatorics of words and additive number theory.

Unlike SAT solvers, the algorithm does not generate efficiently-checkable proof certificates.

Goal: A trusted decision procedure for automatic sequences in Lean.

The first step (ongoing work): Implement and verify key automata involved in the decision procedure.

Current formalization of automata in Lean's Mathlib is not designed for computation. It's also missing lots of stuff we need, like automata with output.

Finite Automata

A deterministic finite automaton with output (DFAO) is a tuple $M = (Q, \Sigma, \delta, q_0, \Delta, \tau)$ as follows:

Q is a finite nonempty set of states.

Σ is the input alphabet (usually $\Sigma_k^n = \{0, 1, \ldots, k-1\}^n$).

$\delta : Q \times \Sigma \to Q$ is the transition function.

q_0 is the initial state.

Δ is the output alphabet.

$\tau : Q \to \Delta$ is the output map.

For DFA without output, there is no output alphabet or output map, but a set of accepting states: the DFA "accepts" an input iff the transitions end in an accepting state. For NFA, δ allows multiple states to be reached from a single state upon reading a single letter. It accepts if any reached state is accepting.

Finite Automata in Lean

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

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

```
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)
```

VS. Mathlib

Here's the Mathlib definition of a crucial construction that we need, turning an NFA into an equiavlent DFA (in the sense that they accept the same words) using subset construction.

```
toDFA : DFA α (Set σ ) where
  step := M.stepSet
  start := M.start
  accept := { S | ∃s ∈S, s ∈M.accept }
```

In our library, it is much more computation-friendly.

```
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
```

Number representations

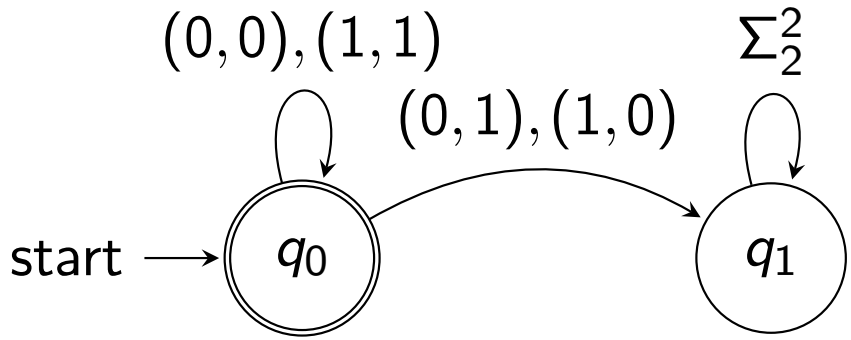
Automata evaluate words recursively, so we naturally take words to be lists. Tuples of natural numbers are represented as lean vectors. To verify automata in the decision procedure, we need to represent natural number tuples as a word. Wrong choice of data structure here will make verification much more difficult.

```
def toWord (v: Fin m →ℕ) (k: ℕ) : List (Fin m →Fin (k +
    2)) :=
  zip (stretchLen (mapToBase (k + 2) v)) (by
    apply stretchLen_of_mapToBase_lt_base
    omega
  ) (by
    intro i
    apply stretchLen_uniform
  )
```

Automata for Equality and Inequality

Equality Automaton:

The automaton for checking equality reads two inputs in parallel (e.g. representing two numbers digit-by-digit) and accepts if every pair of corresponding digits is equal.



Not-Equal Automaton:

Swap the roles of the states: q_0 becomes non-accepting and q_1 is now accepting.

Note that this construction is general, which simulates $\neg$.

Correctness Proofs

```
theorem eqBase_iff_equal (k m: ℕ) (v : Fin m →ℕ) (a b :
    Fin m):
  v a = v b ↔(eqBase k m a b).eval (toWord v k)
```

```
theorem DFA.negate_eval (dfa : DFA α state) (s : List α )
     :
  (dfa.negate).eval s = ! dfa.eval s
```

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

Now & Future

We have verified all automata construction for the "first-order logic with equality" fragment of the language. But there's still a long way to go.

- Verify addition and functions for automatic sequences.
- Meta-programming on Lean expressions.
- Verify optimizations to the procedure, such as minimization.
- Verify other number representations (Fibonacci).

Now, a toy demo in Lean.

https://github.com/Aeacu2/Automata

Check Walnut out at
https://cs.uwaterloo.ca/~shallit/walnut.html