Verifying a real-world regex implementation

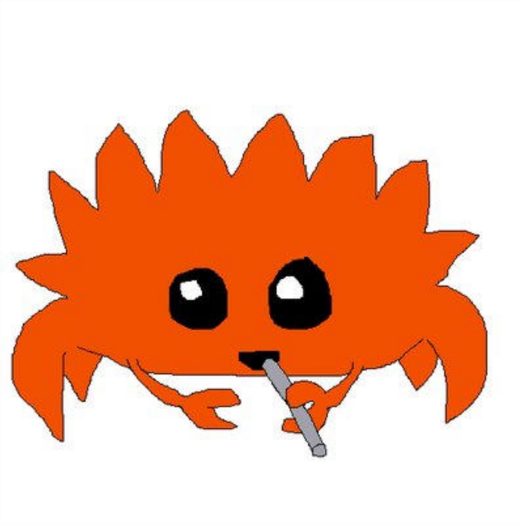
Yulu Pan

2025-10-02

Software Engineer at Indeed Technologies Japan

Self Introduction

- I'm Yulu Pan from Japan👋
  - **pandaman** in the Lean Zulip
- Software Engineer at Indeed Technologies Japan
- Started Lean around 2023, mainly interested in **software verification**
- Author of lean-regex, a formally verified regex engine
- Functional induction is my favorite feature🤗



Overview of lean-regex

What is lean-regex?

- lean-regex is a regex engine for **Lean 4 as a programming language**
- It provides **regex features programmers expect**
  - Substring search
  - Submatches (capture groups)
  - Character classes (`\d`, `\w`, `\s`, `[a-z]`, etc.)
  - Anchors (`^`, `$`, `\b`, `\B`)
- All features are given a **formal operational semantics**
  - The matcher implementation is **proved correct** with respect to the semantics
- **Linear-time matching** via nondeterministic finite automaton (NFA)
  - Optimizations are verified correct
- **Looking for contributors!**

Formal verification of lean-regex

Scope of formal verification

- **Formally specified regex semantics**, including position-aware features
  - Anchors (`^`, `$`, `\b`, `\B`) matches the current position without consuming inputs
  - Capture groups record positions of submatches
- Prove **soundness** and **completeness** of the matcher
- Limitations today:
  - Parser and preprocessing are terminating, but not yet verified for correctness
    - We had a bug in preprocessing😓
  - Disambiguation policy is not specified or verified

Specifying regex semantics

- Computer science usually talks about **regular expressions and regular languages**
  - Strings are treated as sequences of characters
  - A regular expression denotes a regular language
  - The focus is on a **set membership problem**: whether a string belongs to the language
- **Distinct features of real-world regex engines**
  - Operate on UTF-8 encoded strings and iterators
  - Perform substring search: find a match **inside** a string, not necessarily the whole string
  - **Position-aware features** requires tracking positions and submatches
- We defined the semantics of real-world regexes as a (big-step) **operational semantics**

- Regex syntax

$$e := \emptyset \mid \varepsilon \mid c \mid e_1 \cdot e_2 \mid e_1 \cup e_2 \mid e^* \\ \mid \hat{} \mid \$ \mid (e)_i$$

- Semantics: $\mathbf{it} \xrightarrow{\boldsymbol{e}} \mathbf{it'} \mid \boldsymbol{M}$
  - "Regex e matches the substring from position it to position it', with captures M"
  - $\mathbf{it} := \langle \boldsymbol{w_1}, \boldsymbol{w_2} \rangle$
    - Valid iterator representing a position in $w = w_1 \cdot w_2$
  - $\boldsymbol{M} := \boldsymbol{\emptyset} \mid \boldsymbol{M}[\boldsymbol{i} \mapsto (\mathbf{it}, \mathbf{it'})]$
    - Sequence of captured submatches
    - $M_1 + M_2$ concatenates captured submatches

Select rules from the operational semantics

$$\frac{}{\langle w_1, cw_2\rangle \xrightarrow{c} \langle w_1 c, w_2\rangle \mid \emptyset}$$

$$\frac{\text{it} \xrightarrow{e_1} \text{it'} \mid M}{\text{it} \xrightarrow{e_1 \cup e_2} \text{it'} \mid M}$$

$$\frac{\text{it} \xrightarrow{e_2} \text{it'} \mid M}{\text{it} \xrightarrow{e_1 \cup e_2} \text{it'} \mid M}$$

$$\frac{\text{it} \xrightarrow{e} \text{it'} \mid M}{\text{it} \xrightarrow{(e)_i} \text{it'} \mid M[i \mapsto (\text{it}, \text{it'})]}$$

$$\frac{\text{it.pos} = 0}{\text{it} \xrightarrow{\hat{}} \text{it} \mid \emptyset}$$

$$\frac{\text{it.atEnd}}{\text{it} \xrightarrow{\$} \text{it} \mid \emptyset}$$

$$\frac{\text{it} \xrightarrow{e_1} \text{it'} \mid M_1 \quad \text{it'} \xrightarrow{e_2} \text{it''} \mid M_2}{\text{it} \xrightarrow{e_1 \cdot e_2} \text{it''} \mid M_1 + M_2}$$

Correctness properties

- Proved that the matcher is sound and complete with respect to the operational semantics
  - **Soundness**: if the matcher returns `.some m`, `m` is a match after the starting position
  - **Completeness**: if a match exists after the starting position, the matcher returns `.some m`
    - Contraposition: if the matcher returns `.none`, no match exists
- The matcher operates on `String.Iterator` and correctness holds only for "valid" iterators
  - `ValidFor` from Batteries allowed `List`-based resoning for valid iterators

Proof strategy

$$\text{Regex Semantics} \xleftrightarrow{\text{Equivalent}} \text{NFA Path} \xleftrightarrow{\text{Sound/Complete}} \text{NFA Simulation}$$

1. **Correctness of compilation: compiled NFA has a path iff the regex matches**
   - Mostly textbook proofs
   - Challenge 1: Reusing proofs for intermediate data
   - Challenge 2: Reasoning about NFAs with different sizes
2. **Correctness of search: NFA simulation finds a path iff one exists**
   - Proved invariants about paths and capture groups for graph traversal algorithms
   - The search may find a better match if multiple matches exist

Challenge 1: Reusable proofs with ProofData

- NFA compilation involves intermediate NFAs
- Example: compiling $e_1 \cup e_2$

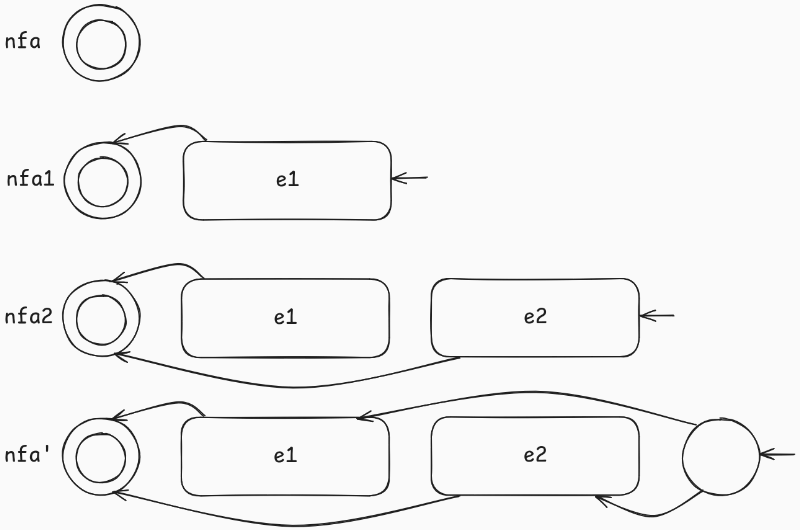


Figure 1: Construction of an NFA for $e_1 \cup e_2$

- Problem: it's hard to reuse proofs about intermediate NFAs
  - `nfa₁` and `nfa₂` are not available in the top-level `theorem` command
- Required to prove the same lemmas (e.g., lifting paths across NFAs) over and over again

Challenge 1: Reusable proofs with ProofData

- Adapted the ProofData idiom from the Carleson project
- Defined a class for each `Node` variant to represent the compilation inputs

```
class ProofData where -- inputs for the NFA compilation
  nfa : NFA
  next : Nat
  e : Expr

class Alternate extends ProofData where -- inputs specific to `e₁ ∪ e₂`
  e₁ : Expr
  e₂ : Expr
  expr_eq : e = .alternate e₁ e₂

-- intermediate NFAs and their properties
def Alternate.nfa₁ [Alternate] : NFA := ...
theorem Alternate.liftPath₁ [Alternate] : Path nfa₁ ... := ...
```

Challenge 1: Reusable proofs with ProofData

- Proofs introduce ProofData via e.g., `let pd := Alternate.intro nfa next e₁ e₂`
- Good
  - Proofs about intermediate NFAs can be reused across different theorems
- Bad
  - More boilerplate
  - `nfa` and `pd.nfa` are def-eq but not syntactically equal; `rw` and `simp` often fail
- Hard to discover these kinds of idioms
  - Wrote a blog post about this: https://zenn.dev/pandaman64/articles/lean-proof-data-en
  - While we already have many great resources and posts, I'd love to see more posts about **idioms and techniques you've found useful in Lean**!

Challenge 2: Type of automaton state indices

- NFA states stored in an array; each node embeds transitions
- States identified by indices

```
inductive Node where
  | done                                 -- accepting state
  | char (c : Char) (next : Nat)         -- transition on a character
  | split (next₁ next₂ : Nat)            -- ε-transition to two states
  -- and others

structure NFA where
  nodes : Array NFA.Node  -- array of states
  start : Nat             -- start state
```

Challenge 2: Type of automaton state indices

- `Nat` vs `Fin` for indices when defining a path through an NFA
- Correctness of compilation:
  - Compilation involves pushing states to the end of the `nodes` array
  - `Fin` indices required a lot of casts when reasoning about **NFAs with different sizes**
  - **Ended up using `Nat` as the index type**
- Correctness of search:
  - The algorithm operates on a single NFA
  - **`Fin` over the fixed NFA was more convenient**
- Defined `Nat`-indexed and `Fin`-indexed paths and proved equivalence

Performance of lean-regex

Complexity

- **Linear-time matching** thanks to nondeterministic finite automaton (NFA) simulation
  - This complexity bound itself is not formally verified
- Lean `Array` allows constant-time node access and cheap construction!

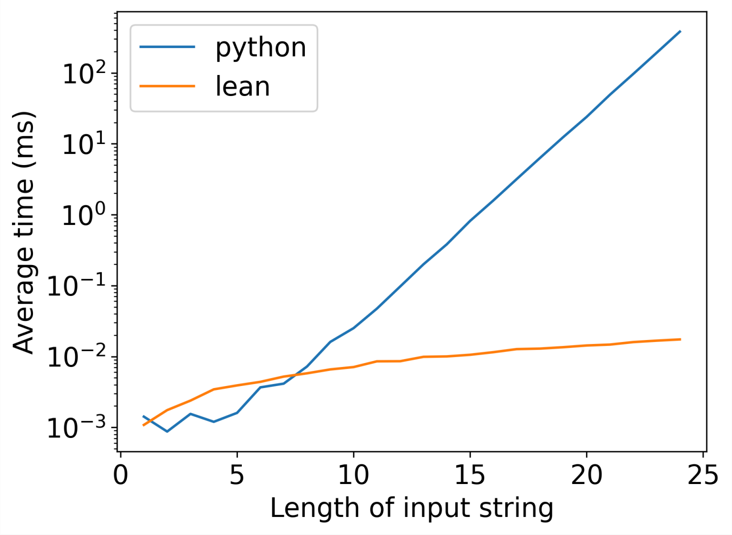


Figure 2: Matching `^(a+)+$` against `a^nX`

Absolute performance today

- **3000+ times** slower than highly-optimized engines (e.g., `rust-lang/regex`)
  - The best engines uses DFAs, SIMD-acclerated searches, etc.
- **7-10 times** (or more) slower than the same algorithm in Rust
- Profiling indicates **heavy allocation and deallocation** (around 50% of the time)

Total (samples)		Self		
16%	1,804	1,804	lean_dec_ref_cold	lean_runner
9.6%	1,112	1,112	mi_free	lean_runner
8.2%	942	942	inl lean_dec_ref	/home/pandaman/.elan/toolchains/leanprover--lean4---v4.23.0-
7.0%	809	809	mi_malloc_small	lean_runner
6.1%	707	707	inl lean_inc	/home/pandaman/.elan/toolchains/leanprover--lean4---v4.23.0-rc2/i
5.1%	583	583	l_Regex_VM__u03b5Closure___redArg	/home/pandaman/rebar/engines/lean/.l
4.1%	473	473	inl lean_is_st	/home/pandaman/.elan/toolchains/leanprover--lean4---v4.23.0-rc2
3.9%	446	446	inl lean_ctor_set	/home/pandaman/.elan/toolchains/leanprover--lean4---v4.23.0
3.5%	399	399	lean_copy_expand_array	lean_runner
3.2%	371	371	l_Regex_VM_stepChar___redArg	/home/pandaman/rebar/engines/lean/.lake/p
3.1%	363	363	l_Regex_Data_SparseSet_insert___redArg	/home/pandaman/rebar/engines/lea
2.6%	303	303	inl lean_ctor_get	/home/pandaman/.elan/toolchains/leanprover--lean4---v4.23.0

Figure 3: Profiling of the matcher. Top five functions come from allocation and deallocation.

Avenues for verified optimizations

- **Prefilters**: extract string literals from a regex and perform fast substring search
  - A simple prefilter doubled the speed in the best case
    - The optimization is verified correct!
  - Integrate Knuth-Morris-Pratt from Batteries?
- **Deterministic finite automaton (DFA) compilation**
  - Requires a lot more verification effort, though
- **Reduce allocations**
  - Avoiding `structure` and nested pairs improved the performance by 5-20%
  - https://github.com/pandaman64/lean-regex/pull/131
- **Eager to learn more performance tricks!**
  - Goal: only 3x slower than the Rust counterpart

Future directions

Future directions

- **Performance improvements**
  - Extending the prefilter to handle more cases (thanks Michiel!)
  - Certified elaboration of a regex to Lean functions (à la re2c)
- **Feature compatibility**
  - Unicode classes
  - Modes: multiline, case-insensitive, etc
- **Formal verification of disambiguation policy**
- NFA visualization with ProofWidget (thanks Krishna!)

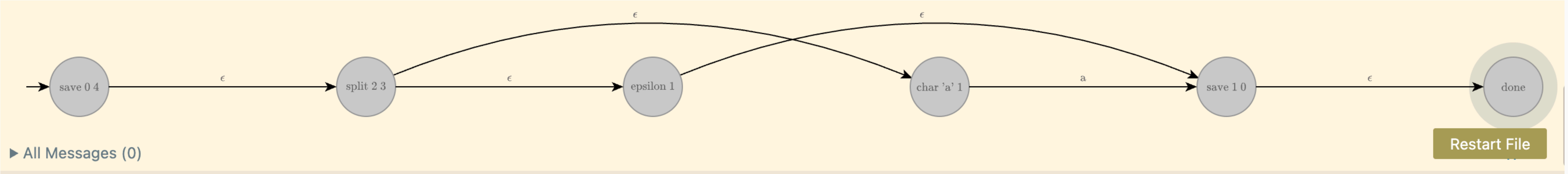


Figure 4: NFA visualization of $(a|\varepsilon)$

Summary

- `lean-regex` is **a real-world regex engine with formal guarantees**
  - NFA-based linear-time matching with position-aware features
- Formal proofs for **soundness** and **completeness**
  - Disambiguation policy is not verified yet; even a formal specification is not easy
- It's not very fast, but there is room for **verified optimizations**
- **We are looking for contributors!**

https://github.com/pandaman64/lean-regex

Appendix: Disambiguation policy

- Regex engines implement **disambiguation policies** (e.g., greedy, POSIX)
  - A disambiguation policy selects a single match from a set of possible matches
  - e.g., matching `foo|foobar` against `foobar` gives `foo` in the greedy policy
- `lean-regex` intends to implement **the greedy policy**
  - Not verified yet
  - Specifying the policy itself is subtle

Appendix: Edge cases in the greedy policy

- **Kleene star of empty matches**
- Example: matching `(^|a)*` against `aaa`
  - In the beginning of the string, both `^` and `a` can match
  - Greedy policy prioritizes `^`, which doesn't advance the position
  - Matching `^` again will loop indefinitely
    - Popular engines like `rust-lang/regex` match `^` just once, never prioritizing `a`

References

- Alain Frisch, Luca Cardelli. Greedy regular expression matching. ACM POPL 2004