Constructive Formalization of Regular Languages

Jan-Oliver Kaiser
Advisors: Christian Doczkal, Gert Smolka
Supervisor: Gert Smolka

UdS

November 7, 2012

Table of Contents

Motivation

- **Goal:** Build an extensive, elegant, constructive formalization of regular languages that includes
  1. regular expressions,
  2. the decidability of equivalence of regular expressions,
  3. finite automata,
  4. and the Myhill-Nerode theorem.
- **How:** Coq with SSReflect. No sacrifices for performance.

Regular Expressions

Definition

- We use extended Regular Expressions (RE) over an alphabet Σ:

$$r; s ::= \emptyset \mid \varepsilon \mid a \mid rs \mid r \ + \ s \mid r \,\&\, s \mid r^* \mid \neg r$$

$$\mathcal{L}(\emptyset) = \emptyset \qquad \mathcal{L}(r^*) = \mathcal{L}(r)^*$$

$$\mathcal{L}(\varepsilon) = \{\varepsilon\} \qquad \mathcal{L}(r + s) = \mathcal{L}(r) \cup \mathcal{L}(s)$$

$$\mathcal{L}(.) = \Sigma \qquad \mathcal{L}(r\&s) = \mathcal{L}(r) \cap \mathcal{L}(s)$$

$$\mathcal{L}(a) = \{a\} \qquad \mathcal{L}(rs) = \mathcal{L}(r) \cdot \mathcal{L}(s)$$

$$\mathcal{L}(\neg r) = \Sigma^* \backslash \mathcal{L}(r)$$

- Implementation taken from Coquand and Siles[1]. **This saved us quite a bit of work**.
- $\approx$ 200 lines of code including an implementation of regular languages and lots of useful lemmas.

[1] Thierry Coquand and Vincent Siles. "A Decision Procedure for Regular Expression Equivalence in Type Theory". In: *CPP*. 2011, pp. 119–134.

Finite Automata

- Our Finite Automata (FA) over an alphabet Σ consist of
  1. a set of states Q,
  2. a starting state $s \in Q$,
  3. a set of final states $F \subseteq Q$,
  4. a transition relation $\delta \in Q \times \Sigma \times Q$.
- Two types: one for non-deterministic FA (δ may be non-functional), one for deterministic FA (δ is functional).
- For our deterministic FA, δ is also total and, thus, a function. This helped, but maybe not a lot.

Finite Automata

Formalization

- Our definition is very close to the textbook definition.
- $\approx$ 120 lines of code including some general lemmas for later proofs.

```
Record nfa : Type :=
  { nfa_state :> finType;
    nfa_s : nfa_state;
    nfa_fin : pred nfa_state;
    nfa_step : nfa_state -> char
               -> pred nfa_state }.
```

```
Record dfa : Type :=
  { dfa_state :> finType;
    dfa_s : dfa_state;
    dfa_fin : pred dfa_state;
    dfa_step : dfa_state -> char
               -> dfa_state }.
```

RE $\implies$ FA

- Structure of proof given by inductive definition of RE.
- **Plan:** Construct FA for every RE constructor.
- Sounds simple enough ..
- .. $\approx$ 800 lines of code.
- $\approx$ 100 lines of that needed for equivalence of DFA and NFA.
- Another $\approx$ 100 lines of code for **extended** regular expressions.
- This is a candidate for improvement.

FA $\Longrightarrow$ RE

- We use the "Transitive Closure method", Kleene's original proof[2].
- This method recursively builds a regular expression $R^X_{x,y}$ that recognizes words whose runs starting in x only pass through states in X and end in y.
- The previous version constructed $R^k_{x,y}$ which translates to $R^{\{z \mid \#(z) < k\}}_{x,y}$ where $\#$ is an ordering on Q.
- Instead of **nat**, we now recurse on the size of a **finite subset** of Q.[3]
- This also avoids cumbersome conversions from nat to SSReflect's ordinals and, finally, to states.

[2] S. C. Kleene. "Representation of Events in Nerve Nets and Finite Automata". In: *Automata Studies* (1965).

[3] Dexter Kozen. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997, pp. I–XIII, 1–400. ISBN: 978-0-387-94907-9.

FA $\implies$ RE

Proof Outline

- We introduce a decidable language $L^X_{x,y}$ that directly encodes the idea behind $R^X_{x,y}$ as a boolean predicate.
- The proof of FA $\implies$ RE consists of three steps:
  1. We show that $L^X_{x,y}$ respects the defining recursive equation of $R^X_{x,y}$.
  2. We show that for all $w \in \Sigma^*$, $w \in L^X_{x,y} \iff w \in R^X_{x,y}$.
  3. We show that $\bigcup_{f \in F} L^Q_{s,f} = \mathcal{L}(A)$ and thus $\sum_{f \in F} R^Q_{s,f} = \mathcal{L}(A)$.

FA $\implies$ RE

- After some restructuring: $\approx$ 550 lines of code, $\approx$ 150 of which are general infrastructure.
- Previous version: $\approx$ 800 lines of code, much harder to read.

```
Lemma L_split k' i j a w:
    let k := k_ord k' in
    (a :: w) \in L^k'.+1 i j ->
    (a :: w) \in L^k' i j \/
    exists w1, exists w2,
    a :: w = w1 ++ w2 /\
    w1 != [::] /\
    w1 \in L^k' i (enum_val k) /\
    w2 \in L^k'.+1 (enum_val k) j.
```

```
Lemma L_split X x y z w:
    w \in L^(z |: X) x y ->
    w \in L^X x y \/
    exists w1, exists w2,
        w = w1 ++ w2 /\
        size w2 < size w /\
        w1 \in L^X x z /\
        w2 \in L^(z |: X) z y
```

Figure: Previous and current version of the same lemma

Myhill-Nerode Theorem

- It turns out that there are two different concepts: Myhill relations and the Nerode relation.
- We also consider a related characterization: weak Nerode relations.
- All these are equivalence relations which we require to be of finite index, i.e. to have a finite number of equivalence classes.
- However, COQ has no notion of quotient types.

Myhill-Nerode Theorem

- We use functions of finite domain to represent equivalence relations of finite index.
- Think of the domain as the set of equivalence classes.
- We also need to have a representative of every equivalence class. Thus, we require surjectivity.

```
Record Fin_Eq_Cls :=
      { fin_type   : finType;
        fin_func   :> word -> fin_type;
        fin_surjective   :  surjective  fin_func  }.
```

- With constructive choice, we can then give a canonical representative of every every equivalence class.

```
Definition cr (f: Fin_Eq_Cls) x := xchoose ( fin_surjective  f x).
```

Myhill-Nerode Theorem

Myhill relations, weak Nerode relations, Nerode relation

- An equivalence relation $\equiv$ of finite index is a **Myhill relation**[4] for L iff
  1. $\equiv$ is **right-congruent**, i.e. $\forall u, v \in \Sigma^*. \forall a \in \Sigma.\ u \equiv v \implies ua \equiv va$,
  2. and $\equiv$ **refines L**, i.e. $\forall u, v \in \Sigma^*.\ u \equiv v \implies (u \in L \iff v \in L)$.
- An equivalence relation $\equiv$ of finite index is a **weak Nerode relation** for L iff

$$\forall u, v \in \Sigma^*.\ u \equiv v \implies \forall w \in \Sigma^*. (uw \in L \iff vw \in L).$$

- The **Nerode relation**[5] $\doteq_L$ for L is defined such that

$$\forall u, v \in \Sigma^*.\ u \doteq_L v \iff \forall w \in \Sigma^*. (uw \in L \iff vw \in L).$$

[4] John R. Myhill. *Finite Automata and the Representation of Events*. Tech. rep. WADC TR-57-624. Wright-Paterson Air Force Base, 1957.

[5] Anil Nerode. "Linear Automaton Transformations". In: *Proceedings of the American Mathematical Society* 9.4 (1958), pp. 541–544.

Myhill-Nerode Theorem

Formalization of Myhill, weak Nerode and Nerode relation

- For all three relations, we build a record that consists of an equivalence relation of finite index and proofs of the properties of the respective relation.
- Example:

```
Record Myhill_Rel :=
        { myhill_func :> Fin_Eq_Cls;
          myhill_congruent : right_congruent myhill_func;
          myhill_refines : refines myhill_func }.
```

Myhill-Nerode Theorem

- Our version of the Myhill-Nerode theorem states that the following are equivalent
  1. language L is accepted by a DFA,
  2. we can construct a Myhill relation for L,
  3. we can construct a weak Nerode relation for L,
  4. the Nerode relation for L is of finite index.
- (1) $\implies$ (2) is easy. (Map word w to the last state of its run on the automaton.)
- (2) $\implies$ (3) is a trivial inductive proof.
- (4) $\implies$ (1) is also straight-forward. (Use equivalence classes as states.)

Myhill-Nerode Theorem

(3) $\implies$ (4)

- A weak Nerode relation (given as a function f) has at least as many equivalence classes as the Nerode relation.
- Some of them are redundant w.r.t. to L, i.e. we may have equivalence classes s.t.

$$\forall uv \in \Sigma^*.\ fu \neq fu \wedge \forall w \in \Sigma^*.\ (uw \in L \iff vw \in L).$$

- Our goal is to merge these equivalence classes.
- In fact, we construct an equivalence relation on these equivalence classes.
- Equivalence classes are contained in that relation iff they are equivalent w.r.t. to L.

Myhill-Nerode Theorem

Finding equivalent equivalence classes

- We use the table-filling algorithm[6] for minimizing DFA.
- Our fixed-point algorithm finds all equivalence classes whose representatives are distinguishable w.r.t. to L.
- The remaining equivalence classes are then equivalent w.r.t. to L.
- Start: $\{(x, y) \mid cr(x) \notin L \iff cr(y) \in L\}$.
- Step: if the previous result is d, the new result is $d \cup \{(x, y) \mid \exists a.\ (f(cr(x)a), f(cr(y)a)) \in d\}$.
- Due to finiteness of the domain and monotonicity of the algorithm, it has a fixed point which we can compute.

[6] D.A. Huffman. "The synthesis of sequential switching circuits". In: *Journal of the Franklin Institute* 257.3 (1954), pp. 161 –190.

Myhill-Nerode Theorem

- We construct a function f_{min} that maps every equivalence class to the set of equivalence classes equivalent to it w.r.t. L.
- We then show that f_{min} is surjective and encodes an equivalence relation of finite index.
- Finally, we show that f_{min} is equivalent to the Nerode relation.

Myhill-Nerode Theorem

- The lemmas of this chapter are rather abstract, which makes for nice and short statements.
- The proofs also received more refinement than the other chapters. They are very concise.
- The entire chapter consists of ≈ 430 lines of code.

Summary

- All in all we have $\approx$ 2100 lines of code.
- Mostly very close to the mathematical definitions.
- Code produced in the beginning of the project might be improved quite a bit.

Thank you for your attention!