Chapter 1

Finite Automata

Another way of characterizing regular languages are finite automata. We will show that the languages of finite automata are exactly REG. Furthermore, we will also derive a decision procedure for equivalence of regular expressions.

1.1 Definition

A finite automaton consists of

1. finite set of states Q,
2. an alphabet Σ,
3. a starting state $s_0 \in Q$,
4. a set of final states $F \subseteq Q$
5. and a state-transition relation δ. **?**

We define a **run** of a word $w \in \Sigma^*$ on an automaton $A = (\Sigma, Q, s_0, F, \delta)$ as any sequence of states σ such that $\forall\, 0 <= i < |\sigma| - 2.\, (\sigma_i, w_i, \sigma_{i-1}) \in \delta$. A word w is **accepted** by A if and only if there exists a run σ of w on A such that $\sigma_0 = s_0 \wedge \sigma_{|\sigma|-1} \in F$. The **language** of A is exactly the set of words accepted by A and is denoted $\mathcal{L}(A)$. It will later be useful to also have a acceptance criterion defined by runs starting in a given state $x \in Q$, for which we will denote the resulting language $\mathcal{L}_x(A)$.

Introduce section-wide variables

1.1.1 Determinism and Non-Determinism

Finite automata can be non-deterministic in the sense that there exist multiple distinct runs for a word. This is the case if and only if δ is not functional.

Listing 1.1: Non-Deterministic Finite Automata

```
Record nfa : Type :=
  nfaI {
    nfa_state :> finType;
    nfa_s0 : nfa_state ;
    nfa_fin : pred nfa_state ;
    nfa_step : nfa_state -> char -> pred nfa_state
    }.

Fixpoint nfa_lpath x (xs : seq A) (w: word) {struct xs} :=
match xs,w with
  | y :: xs', a::w' => nfa_step A x a y && nfa_lpath y xs' w'
  | [::] , [::] => true
  | _ , _ => false
end.
```

For functional δ, we speak of **deterministic finite automata**. In this case, we also assume δ to be total and write it as a function. This allows us to directly define the acceptance criterion in terms of the unique run of a word on the automaton.

Listing 1.2: Deterministic Finite Automata

```
Record dfa : Type :=
  dfaI {
    dfa_state :> finType;
    dfa_s0 : dfa_state ;
    dfa_fin : pred dfa_state ;
    dfa_step : dfa_state -> char -> dfa_state
    }.

Fixpoint dfa_run' (x: A) (w: word) : seq A :=
match w with
  | [::] => [::]
  | a::w => (dfa_step A x a) ::dfa_run' (dfa_step A x a) w
end.
```

Equivalence between DFA and NFA

Deterministic and non-deterministic finite automata are equally powerful. One direction is trivial since every DFA is also a NFA. We prove the other direction using the powerset construction. Given NFA A, we construct an equivalent DFA A_{det} in the following way: The new set of states is the powerset of the given NFA's set of states. The new starting state is the singleton set containing the original starting state. A state is final if and

only if it contains an original final state. The transition function on powerset states is defined as follows:

$$(P, a, Q) \in \delta_{det} \iff Q = \bigcup_{p \in P} \{q | (p, a, q) \in \delta\}.$$

Listing 1.3: Powerset Construction

```
Definition powerset_state : finType := [finType of {set A}].
Definition powerset_s0 : powerset_state := set1 (nfa_s0 A).
Definition nfa_to_dfa :=
  dfaI
    powerset_state
    powerset_s0
    [ pred X: powerset_state | existsb x: A, (x \in X) && nfa_fin A x]
    [ fun X a => \bigcup_(x | x \in X) finset (nfa_step A x a) ]
.
```

Theorem 1.1.1. *The powerset automaton A_{det} accepts the same language as A, i.e.*

$$\mathcal{L}(A) = \mathcal{L}(A_{det}).$$

Proof. We first prove that for every powerset state X and every state $x \in X$ we have that $\mathcal{L}_x(A) \subseteq \mathcal{L}_X(A_{det})$. Applying this to $X = \{s_0\}$ yields $\mathcal{L}(A) \subseteq \mathcal{L}(A_{det})$. We then show that for every powerset state X and word w with $w \in \mathcal{L}_X(A_{det})$ there exists a state x such that $x \in X$ and $w \in \mathcal{L}_x(A)$. For $X = \{s_0\}$, this shows $\mathcal{L}(A_{det}) \subseteq \mathcal{L}(A)$. Both proofs are done by induction on word. □

The formalization of this proof is straight-forward and follows exactly the plan laid out above. The corresponding Lemmas are:

```
Lemma nfa_to_dfa_complete (x: A) w (X: nfa_to_dfa ):
  x \in X -> nfa_accept A x w -> dfa_accept nfa_to_dfa X w.
Lemma nfa_to_dfa_sound (X: nfa_to_dfa) w:
  dfa_accept nfa_to_dfa X w -> existsb x, (x \in X) && nfa_accept A x w.
Lemma nfa_to_dfa_correct w : nfa_lang A w = dfa_lang nfa_to_dfa w.
```

1.2 Connected Components

Finite automaton can have isolated subsets of states that are not reachable from the starting state. These states can not contribute to the language of the automaton. It will later be useful to have automata that only contain reachable states. We define a procedure to extract the connected component from a given automaton.

Theorem 1.2.1. *The language of the connected automaton A_c is identical to that of the original automaton A, i.e.*

$$\mathcal{L}(A) = \mathcal{L}(A_c).$$

Proof. By definition, unreachable states have no influence on the language of an automaton because there is no run from the starting state that contains such a state. □

We make use of SSReflect's *connect* predicate to extract a sequence of all states reachable from s_0. From this, we construct a finite type and use that as the new set of states. These new states carry a proof of reachability. We also have to construct a new transition function that ensures transitions always end in reachable states. Theorem **??** is trivially solved by induction on word.

Add implementation

1.3 Emptiness

Given an automaton A, we can check if $\mathcal{L}(A) = \emptyset$. We simply obtain the connected automaton of A and check if there are any final states left.

Theorem 1.3.1. *We can decide emptiness of $\mathcal{L}(A)$ by computing the cardinality of A_c's set of final states, i.e.*

$$F_c = \emptyset \iff \mathcal{L}(A) = \emptyset.$$

Proof. This is correct since $F_c = \emptyset \Leftrightarrow \mathcal{L}(A_c) = \emptyset$ and $\mathcal{L}(A_c) = \mathcal{L}(A)$ by theorem **??**. □

Add implementation

1.4 Deciding Equivalence of Finite Automata

Given finite automata A_1 and A_2, we construct DFA A such that the language of A is the symmetric difference of the languages of A_1 and A_2, i.e.,

$$\mathcal{L}(A) = \mathcal{L}(A_1) \ominus \mathcal{L}(A_2) = \mathcal{L}(A_1) \cap \neg\mathcal{L}(A_2) \cup \mathcal{L}(A_2) \cap \neg\mathcal{L}(A_1).$$

Theorem 1.4.1. *The equivalence of A_1 and A_2 is decidable, i.e.*

$$\mathcal{L}(A_1) = \mathcal{L}(A_2) \text{ if and only if } \mathcal{L}(A) \text{ is empty.}$$

Proof. The correctness of this procedure follows from the properties of the symmetric difference operator, i.e.

$$\mathcal{L}(A_1) \ominus \mathcal{L}(A_2) = \emptyset \Leftrightarrow \mathcal{L}(A_1) = \mathcal{L}(A_2).$$

The decidability of this procedure follows directly from theorem **??**. □

Add implementation

1.5 Regular Expressions and Finite Automata

We prove that there is a finite automaton for every extended regular expression and vice versa. In fact, we can give a standard regular expression for every finite automaton. With this, we will prove that extended regular expressions are equivalent to standard regular expressions, thereby proving closure under intersection and negation.

1.5.1 Regular Expressions to Finite Automata

We prove that there exists an equivalent automaton for every extended regular expressions. The structure of this proof is given by the inductive nature of regular expressions. For every constructor, we provide an equivalent automaton.

Include all proofs?

1.5.2 Deciding Equivalence of Regular Expressions

Based on our procedure to construct an equivalent automaton from a regular expression, we can decide equivalence of regular expressions. Given r_1 and r_2, we construct equivalent DFA A_1 and A_2 as above.

1.5.3 Finite Automata to Regular Expressions

We prove that there is an equivalent standard regular expression for every finite automaton.

Since we are given an automaton it is not obvious how to partition our proof obligations into smaller parts. We use Kleene's original proof, the transitive closure method. This method recursively constructs a regular expression that is equivalent to the given automaton. Given a DFA A, we first assign some ordering to its states. We then define $R^k_{i,j}$ such that $\mathcal{L}(R^k_{i,j})$ is the set of all words that have a run on A starting in state i that ends in state j without ever leaving a state smaller than k. The base case $R^0_{i,j}$ is the set of all singleton words that are edges between state i and j, and ε if $i = j$. Given $R^k_{i,j}$ we can easily define $R^{k+1}_{i,j}$ based on the observation that only one new state has to be considered:

Insert complete formal definition

$$R^{k+1}_{i,j} = R^k_{i,k} \cdot (R^k_{k,k})^* \cdot R^k_{k,j} + R^k_{i,j}.$$

We make use of SSReflect's ordinals to get an ordering on states. We chose to employ ordinals for i and j, but not for k. This simplifies the inductive definitions on k. It does, however, lead to explicit conversions when k is used in place of i or j. In fact, i and j are states in our Coq implementation. We only rely on ordinals for comparison to k.

Add implementation of R

Furthermore, we define $L^k_{i,j} \subseteq \mathcal{L}(A)$ in terms of runs on the automaton. The relation of $L^k_{i,j}$ to $\mathcal{L}(A)$ can be proven very easily. We will also prove it equivalent to $R^k_{i,j}$. This allows us to connect $R^k_{i,j}$ to $\mathcal{L}(A)$.

```
Definition allbutlast p : pred (seq X) :=
  fun xs => all p (belast xs).
Definition L :=
  [fun k: nat =>
    [fun x y: A =>
      [pred w |
       (last x (dfa_run' A x w) == y)
         && allbutlast (<_k) (dfa_run' A x w)
      ]
    ]
  ].
```

Theorem 1.5.1. *We can express $\mathcal{L}(A)$ in terms of L. L is equivalent to R.*

$$\mathcal{L}(A) = \bigcup_{f \in F} L^{|Q|}_{s_0,f} = \mathcal{L}(\sum_{f \in F} R^{|Q|}_{s_0,f}).$$

Proof. By definition, every $w \in \mathcal{L}(A)$ has a run that ends in some $f \in F$. Then, by definition, $w \in L^{|Q|}_{s_0,f}$.

It remains to show that $\mathcal{L}(R^k_{i,j}) = L^k_{i.j}$. This claim can be proven by induction over k. We begin with the inclusion of $\mathcal{L}(R^k_{i,j})$ in $L^k_{i,j}$. For $k = 0$, we do a case distinction on $i == j$ and unfold R. The resulting three cases ($i == j \wedge w = \varepsilon$, $i == j \wedge |w| = 1$, $i <> j \wedge |w| = 1$) are easily closed.

The inductive step has two cases: A triple concatenation and a simple recursion. The second case is solved by the inductive hypothesis. In the firs case, we split up the concatenation such that

$$w = w_1 \cdot w_2 \cdot w_3 \wedge w_1 \in \mathcal{L}(R^k_{i,k}) \wedge w_2 \in \mathcal{L}((R^k_{k,k})^*) \wedge w_3 \in \mathcal{L}(R^k_{k,j}).$$

The induction hypothesis is applied to w_1 and w_3 to get $w_1 \in L^k_{i,k}$ and $w_3 \in L^k_{k,j}$. We use a lemma by Coquand and Siles that splits w_2 into a sequence of words from $\mathcal{L}(R^k_{k,k})$ to which we can apply the induction hypothesis. Two concatenation lemmas for L are used to merge the sequence of words proven to be in $L^k_{k,k}$, w_1 and w_3. This shows $\mathcal{L}(R^k_{i,j}) \subseteq L^k_{i,j}$.

Next, we show the inclusion of $L^k_{i,j}$ in $\mathcal{L}(R^k_{i,j})$, again by induction over k. The base case is solved by case distinction on $i == j$. The inductive step requires a **splitting lemma** for L which shows that every non-empty word in $L^{k+1}_{i,j}$ is either in $L^k_{i,j}$ or has a non-empty prefix in $L^k_{i,k}$ and a corresponding

suffix in $L_{k,j}^{k+1}$. The In the first case, we can apply the induction hypothesis. In the second case, we use size induction on the word, apply the original induction hypothesis to the prefix and the size induction hypothesis to the suffix. We use two concatenation lemmas for R to merge the sub-expression. This finishes the proof. □

Formalizing theorem **??** requires infrastructure to deal with *allbutlast*. Once this is in place, we can formalize the concatenation lemmas for R and L. These are required later to connect sub-results.

```
Lemma R_catL k i j w1 w2:
      w1 \in R^k i  (k_ord k) ->
      w2 \in R^k.+1 (k_ord k) j ->
      w1++w2 \in R^k.+1 i j.

Lemma L_catL k i j w1 w2:
      w1 \in L^k i (enum_val (k_ord k)) ->
      w2 \in L^k.+1 (enum_val (k_ord k)) j ->
      w1++w2 \in L^k.+1 i j.

Lemma L_catL k i j w1 w2:
      w1 \in L^k i (enum_val (k_ord k)) ->
      w2 \in L^k.+1 (enum_val (k_ord k)) j ->
      w1++w2 \in L^k.+1 i j.
```

We also need the splitting lemma mentioned earlier. This is quite intricate. We could split right after the first character and thereby simplify the lemma. However, the current form has the advantage of requiring simple concatenation lemmas.

```
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.
```

These lemmas suffice to show the claim of theorem **??**.

```
Lemma R_L_star k vv:
    ( forall (i j : 'I_#|A|) (w : word char),
        w \in R^k i j -> w \in L^k (enum_val i) (enum_val j)) ->
      all [predD mem_reg (R^k (k_ord k) (k_ord k)) &
           eps (symbol:=char)] vv ->
      flatten vv \in L^k.+1 (enum_val (k_ord k)) (enum_val (k_ord k)).
```

```
Lemma R_L k i j w: w \in R^k i j -> w \in L^k (enum_val i) (enum_val j).

Lemma L_R_1 k i j w:
      ( forall (i j : 'I_#|A|) (w : automata.word char),
       w \in L^k (enum_val i) (enum_val j) -> w \in R^k i j) ->
       w \in L^k.+1 (enum_val i) (enum_val j) -> w \in R^k.+1 i j.

Lemma L_R k i j w: w \in L^k (enum_val i) (enum_val j) -> w \in R^k i j.
```

Fix this mess