1 Right-Hand Side

The following are the soundness proofs of functions within the `Section` *rhs* (the right-hand side of a grammar derivation rule). They are formalized in Coq in the soundness.v file.

Let T and NT variables of type `Type`.

Lemma 1 (Soundness of *isEmpty*)**.** $\forall$ *NT T,* $\forall$ *r:rhs.t T NT, rhs.isEmpty r = false* $\vee$ *rhs.isEmpty r = true.*

Proof. This proof is straightforward from *isEmpty*'s definition. For every input it shall return either true or false. By destructing *isEmpty* two subgoals are generated, each of them for each possible output of the function.

(i) *true = false* $\vee$ *true = true.* It holds by focusing on the right side of the disjunction.

(ii) *false = false* $\vee$ *false = true.* It holds by focusing on the left side of the disjunction.

□

Lemma 2 (Soundness of *isEmpty* function)**.** $\forall$ *NT T,* $\forall$ *r:rhs.t T NT, rhs.isEmpty r = true* $\leftrightarrow$ *r = Empty.*

The function isEmpty verifies if a given rhs of a rule is of the empty kind ϵ*, where* ϵ *is the empty string character). isEmpty returns true iff the rhs is of the empty kind and false otherwise.*

Proof.
$\Rightarrow$
Let *rhs.isEmpty r = true*, *r* being of the type of *rhs.t*. We want to show that *r* is *Empty*. This proof is done by destructing *r*. Since *r* is an inductive structure, a goal is generated for each possible constructor of the inductive type. Let *rhs.isEmpty* with

Empty = true the subgoal is *Empty = Empty*, then it holds;

Single t = true the subgoal is *Single t = Empty* which by the definition of *isEmpty* is a contradiction;

Continue t nt = true the subgoal is *Continue t nt = Empty* which by the definition of *isEmpty* is a contradiction.

$\Leftarrow$
Let *r = Empty*. We want to show that *rhs.isEmpty r = true* (i) holds. By (i) we rewrite *r* as *Empty* in the goal and we have *rhs.isEmpty Empty = true* (ii). Simplifying (ii) we have that *Empty = Empty*. □

2 Regular Grammars

In this Section it is presented the soundness proofs of the functions formalized within the *reg_grammar* module, in which is our formalization of regular grammars. For the remainder of this section, the variables T and NT are from the type `Type`, the most general type in Coq.

Definition 1 (Regular grammar)**.** *A regular grammar is a tuple $G = (V, T, P, S)$, where V is a non-empty set of nonterminal symbols of the grammar (also called variable symbols), T is a non-empty set of terminal symbols, P is a set of production rules and $S \in V$ is the initial symbol of the grammar, from which all derivations of the grammar must start. It corresponds to the following definition in Coq:*

```
Record g : Type:= {
    start_symbol: NT;
    rules : set (NT × rhs.t T NT);
    terminal_symbols : set T;
    nonterminal_symbols : set NT
 }.
```

where nonterminal_symbols corresponds to V terminal_symbols corresponds to T, rules corresponds to P, in where the rules are represented by pairs (S, rhs), where S corresponds to a terminal symbol in V and rhs corresponds to a valid right hand side, described in section 1 and start_symbol corresponds to S, the initial symbol of the grammar.

Lemma 3 (Soundness of build_grammar)**.** *∀ nt:NT, ∀ rules:set (NT × rhs.t T NT), ∀ alphabet: set T, ∀ nonterminals: set NT,*

```
reg_grammar.build_grammar nt rules alphabet nonterminals =:={|
    start_symbol := nt;
    rules := add_list_to_set (rules);
    terminal_symbols := add_list_to_set (alphabet) ;
    nonterminal_symbols := add_list_to_set (nonterminal) |}.
```

This lemma is just a "sanity-check", to prove that the grammar built by using this procedure has the start symbol, the set of derivation rules, the terminal symbols set and the nonterminal symbols set the same as the ones given as parameters to this function. The function add_list_to_set is needed because we use the implementation of sets as list (as seen in the library ListSet). This function assures that the resulting fields will be sets.

Proof. This proof is straightforward from Definition 1. □

Lemma 4 (Soundness of *step_rhs*)**.** *∀ t:T, ∀ rhs: rhs.t T NT, reg_grammar.step_rhs t rhs = [None] ↔ rhs = Single t.*

This lemma states that the step_rhs *function will return [None] iff the rhs given as parameter is Single t.*

Proof.
⇒
Let *reg_grammar.step_rhs t rhs* = [*None*]. We want to prove that *rhs* has to be *Single t*, where *t* is a terminal symbol. This is done by destructing *rhs*, generating three subgoals, each for one possible constructor of the inductive type of *rhs*. Hence, let *reg_grammar.step_rhs t* with :

Empty then let H1: *reg_grammar.step_rhs t Empty* = [*None*]. The first subgoal is *Empty* = *Single t*. By the definition of the *step_rhs* function, when the *rhs* given as parameter is *Empty*, it returns []. Then, by simplifying *reg_grammar.step_rhs t Empty* = [*None*] in H1, H1 is [] = [*None*]. Since H1 is a contradiction, the subgoal is proved.

Single t_0 then let H2: *reg_grammar.step_rhs t Single* t_0 = [*None*]. The second subgoal is *Single* t_0 = *Single t*, with t_0 as a terminal symbol. By simplifying H2, it becomes (`if` *equiv_dec t* t_0 `then` [*None*] `else` []) = [*None*]. We proceed by destructing the equivalence relation for the inductive type of *t* in H2. In this case, we have to prove for two cases:

(i) Let H':$t = t_0$ and H": [*None*] = [*None*]. The subgoal is *Single* t_0 = *Single t*. By rewriting H' in the subgoal, it is *Single* t_0 = *Single* t_0, which holds.

(ii) Let H': t =\= t_0 and H": [] = [*None*]. H" is a contradiction. Then, this goal is proved.

Continue $\mathbf{t_0}$ **n** with H1: *reg_grammar.step_rhs t Continue* t_0 *n* = [*None*]. The third subgoal is *Continue* t_0 *n* = *Single t*. By simplifying H1, it is (`if` *equiv_dec t* t_0 `then` [*Some n*] `else` []) = [*None*]. We proceed by destructing *equiv_dec*, generating two subgoals:

(i) Let H': $t = t_0$ and H1: [*Some n*] = [*None*]. The subgoal is *Continue* t_0 *n* = *Single t*. H1 is a contradiction. Then, the subgoal is proved.

(ii) Let H': t =\= t_0 and H1: [] = [*None*]. The subgoal is *Continue* t_0 *n* = *Single t*. H1 is a contradiction. Then, the subgoal is proved.

⇐
Let *rhs* be *Single t*. We want to prove *reg_grammar.step_rhs t rhs*
= [*None*]. This is done by rewriting the hypothesis in the goal. The goal becomes *reg_grammar.step_rhs t Single t* = [*None*]. By simplifing the function in the goal, it is (`if` *equiv_dec t t* `then` [*None*] `else` [] *t rhs*) = [*None*]. Then, by destructing the equivalence relation (*equiv_dec*), it is generated two subgoals:

(i) Let H': Let $t = t$. The goal is [*None*] = [*None*], which holds.

(ii) Let H': t =\= t. The goal is [] = [*None*]. This subgoal is proved by contradiction by using the *exfalso* tactic, since t can't be different from itself. The goal is `False`. By applying H', the goal is $t = t$, which holds.

□

Lemma 5 (Soundness of *getRHS*)**.** $\forall$ *nt,*$\forall$ *rules, (reg_grammar .getRHS nt rules) = a::l* $\leftrightarrow$ *(rules* $\neq \emptyset$*)* $\wedge$ *(exists rule: (NT * rhs.t T NT) In (rule) (rules)* $\wedge$ *fst(rule) = nt).*

This lemma states that the return of getRHS function is a nonempty set iff rules is not an empty set and exists at least one rule in the set of rules that the left hand side of the rule equals nt.

Proof.
$\Rightarrow$
Let *H1*: (*reg_grammar.getRHS nt rules*) be a nonempty set. We want to prove that (*rules* $\neq \emptyset$) $\wedge$ ($\exists$ rule: (NT * rhs.t T NT), *In* (*rule*) (*rules*) $\wedge$ *fst(rule) = nt*)) holds. The proof is done by contradiction.

Let *H2*: $\neg$ ((*rules* $\neq \emptyset$) $\wedge$ ($\exists$ rule: (NT * rhs.t T NT), *In* (*rule*) (*rules*) $\wedge$ *fst(rule) = nt*))). In other words, *H2*: (*rules* $= \emptyset$) $\wedge$ $\forall$ rule: (NT * rhs.t T NT), $\neg$ (*In* (*rule*) (*rules*)) $\vee$ *fst(rule)* $=\backslash=$ *nt*). We proceed by destructing *H2*, generating two hypothesis:

Let *H2*: *rules* $= \emptyset$. By the definition of *getRHS*, *getRHS* iterates over *rules*, if the set of rules given is an empty set, *getRHS* returns an empty set, which contradicts *H1.*

let *H2*: $\forall$ rule: (NT * rhs.t T NT), $\neg$ (*In* (*rule*) (*rules*)) $\vee$ *fst(rule)* $=\backslash=$ *nt*). By the definition of *getRHS*, if there is no rule in the set of rules that have *nt* in its left hand side, the resulting set is empty, which contradicts *H1.*

$\Leftarrow$
Let *H*: (*rules* $\neq \emptyset$) $\wedge$ ($\exists$ rule: (NT * rhs.t T NT) *In* (*rule*) (*rules*) $\wedge$ *fst(rule) = nt*). We want to prove that (*reg_grammar.getRHS nt rules*) returns a nonempty set.
By the definition of *getRHS* where the function goes through the set of production rules, searching for rules that have *nt* on its left hand side and storing them in the returning set. By destructing *H*, we generate two hypothesis: *H1*:(*rules* $\neq \emptyset$) and *H2*: ($\exists$ rule: (NT * rhs.t T NT) *In* (*rule*) (*rules*) $\wedge$ *fst(rule) = nt*). By *H1*, we have that the set of rules is not empty and by *H2*, we have that exists at least one rule in the list of rules that have *nt* in its left hand side. By *H2*, the resulting set that is returned by (*reg_grammar.getRHS nt rules*) has at least one element, being an nonempty set.

□

Lemma 6 (Soundness of *step_nt* function)**.** $\forall$ *rules,*$\forall$ *t,* $\forall$ *nt, reg_grammar .step_nt rules t nt = []* $\vee$ *In None (reg_grammar.step_nt rules t nt)* $\vee$ *(*$\exists$ *n:NT, In (Some n) (reg_grammar.step_nt rules t nt)).*

This lemma states that the function step_nt, which applies all possible derivations of a given terminal symbol and a nonterminal symbol, is either an empty list, a list containing None (if at least one of the rules that could be used had the rhs = Single t) or a list containing Some n, n being a nonterminal symbol (if at least one of the rules that could be used had the rhs = Continue t n).

Proof. The proof is done by destructing *step_nt* and analyzing each possible output of the function. It can return either an empty list or a list with elements in it. By destructing it, we generate two subgoals. Hence,let:

step_nt = [ ]. The subgoal is [] = [] $\vee$ *In None* [] $\vee$ ($\exists$ *n*:*NT*, *In* (*Some n*) []). By focusing on the left side of the first disjunction, we have [] = [] which holds.

step_nt = o::l , where o is an object of type *option NT* and *l* is a list of the same type of *o*. We proceed by destructing *o*, generating two subgoals, one for each constructor of the *option* type. Then, let *o* with:

- ***Some n*** . Then, the subgoal is *Some n :: l* = [] $\vee$ *In* (*None*) (*Some n :: l*) $\vee$ ($\exists$ n_0:*NT*, *In* (*Some* n_0) (Some n :: l)). Focusing on the far right side of the disjunction, ($\exists$ n_0:*NT*, *In* (*Some* n_0) (*Some n :: l*)), when simplifying it, we get ($\exists$ n_0:*NT*, *Some* n_0 = *Some n* $\vee$ *In* (*Some* n_0) *l*). Since *n* has been instantiated, we can use it in the existential quantifier, getting (*Some n*) = *Some n* $\vee$ *In* (*Some n*) *l*). Since the left side of the disjuntcion holds, the subgoal is proved.
- ***None*** . The subgoal now is *None :: l* = [] $\vee$ *In* (*None*) (*None :: l*) $\vee$ ($\exists$ n_0:*NT*, *In* (*Some* n_0) (None :: l)). By focusing on the right side of the first disjunction, The subgoal becomes *In* (*None*) (*None :: l*). By simplifying it, the subgoal becomes *None* = *None* $\vee$ *In None l*). Since the left side of the subgoal holds, the goal is proved.

□

Lemma 7 (Soundness of *step* function)**.** $\forall$ *rules,* $\forall$ *t,* $\forall$ *acc, reg_grammar.step rules t acc = [] $\vee$ In (None) (reg_grammar.step rules t acc) $\vee$ ($\exists$ nt:NT, In (Some nt)(reg_grammar.step rules t acc)).*

This lemma states that, for every input, the function step may return either an empty list, a list containing None or a list containing Some nt, where nt is a nonterminal symbol. From the definition of step, it applies all possible derivation steps, given a terminal symbol and a list of possible nonterminal symbols that may derive the terminal symbol or not. In order to do so, it applies the step_nt function for each possible nonterminal symbol in the derivation state, getting, for each possible nonterminal symbol, all RHS of the rules that derives the given terminal symbol.

Proof. The proof is done by destructing the *step* function. Hence, let *step* with

[ ] where the goal is [] = [] ∨
In (*None*) [] ∨ (∃ *nt*:*NT*, *In* (*Some nt*) []. By focusing on the left side of the first disjunction, the subgoal is [] = [], which holds.

o::l where the goal is *o::l* = [] ∨
In (*None*) *o::l* ∨ (∃ *nt*:*NT*, *In* (*Some nt*) *o::l*, where *o* is an element of the type *option NT* and *l* a list of type *option NT*. We proceed by destructing *o*, generating two new subgoals:

(i) *Some n :: l* = [] ∨
In (*None*) *Some n :: l* ∨ (∃ *nt*:*NT*, *In* (*Some nt*) *Some n :: l*, where *o* is an element of the type *option NT* and *l* a list of type *option NT*. By focusing on the right term of the second disjunction, the goal is (∃ *nt*:*NT*, *In* (*Some nt*) *Some n :: l*. Since *n* has already been introduced in the proof context, we can use it to instantiate the existential variable. The goal now is *In* (*Some n*) *Some n :: l*. By simplifying the function, the goal is *Some n* = *Some n* ∨ *In* (*Some n*) *l*. By focusing the left side of the goal, we have *Some n* = *Some n*, which holds.

(ii) *None :: l* = [] ∨
In (*None*) *None :: l* ∨ (∃ *nt*:*NT*, *In* (*Some nt*) *None :: l*. By focusing on the middle term of the goal, it is *In* (*None*) *None :: l*. By simplifying the goal, it is *None* = *None* ∨ *In* (*None*) *l*. By focusing on the left side of the goal, it is *None* = *None*, which holds.

□

Lemma 8 (Soundness of *parse* function)**.** ∀ *grammar*, ∀ *l*: *list T*, *reg_grammar .parse grammar l* = *true* ↔ *([Some (reg_grammar.start_symbol grammar)] | > reg_grammar.parse' (reg_grammar.rules grammar) l | > reg_grammar.is_final (reg_grammar.rules grammar)* = *true).*

This lemma states that the parse function may return true iff, after deriving the given word, the "parser" has reached a final state. In other words, if after starting the derivation from the start symbol of the grammar, when reaching the end of the word being derived, it reaches a final "derivation state". Otherwise, returns false.

Proof.
⇒
Let H: *reg_grammar.parse grammar l* = *true*. We want to show that (*reg_ grammar.is_final* (*reg_grammar.rules g*) (*reg_grammar.parse'* (*reg_grammar .rules g*) *l* [*Some* (*reg_grammar.start_symbol grammar*)]) = *true*) holds.

We proceed by rewriting H in the goal. The goal is *reg_grammar.is_final* (*reg_grammar.rules g*) (*reg_grammar.parse'* (*reg_grammar.rules g*) *l* [*Some* (*reg_grammar.start_symbol grammar*)]) = *reg_grammar.parse grammar l*

By unfolding *parse*, the goal is *reg_grammar.is_final* (*reg_grammar.rules g*) (*reg_grammar.parse'* (*reg_grammar.rules g*) *l* [*Some* (*reg_grammar.start_symbol*

grammar)]) = *reg_grammar.is_final* (*reg_grammar.rules g*) (*reg_grammar .parse'* (*reg_grammar.rules g*) *l* [*Some* (*reg_grammar.start_symbol grammar*)]), which holds.

⇐

Let *H*: *reg_grammar.is_final* (*reg_grammar.rules g*) (*reg_grammar.parse'* (*reg_grammar.rules g*) *l* [*Some* (*reg_grammar.start_symbol grammar*)] = *true*. We want to show that *reg_grammar.parse grammar l* = *true* holds. We proceed by rewriting *H* in the goal. The goal then is *reg_grammar.parse grammar l* = *reg_grammar.is_final* (*reg_grammar.rules g*) (*reg_grammar.parse'* (*reg_grammar.rules g*) *l* [*Some* (*reg_grammar.start_symbol grammar*)]. By unfolding the *parse* function, the goal is *reg_grammar.is_final* (*reg_grammar.rules g*) (*reg_grammar.parse'* (*reg_grammar.rules g*) *l* [*Some* (*reg_grammar.start_symbol grammar*)] = *reg_grammar.is_final* (*reg_grammar.rules g*) (*reg_grammar.parse'* (*reg_grammar.rules g*) *l* [*Some* (*reg_grammar.start_symbol grammar*)], which holds.

□

3 Deterministic Finite Automata

This section discuss the soundness of deterministic finite automata (DFA) formalization and its main functions. Some of the lemmas can be found in *soundness.v* file, while the definitions can be found in *main.v*.

Our definition of DFA follows the same definition as in [Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman]. A finite deterministic automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the finite alphabet that the automata recognizes, q_0 is the starting state of the automaton, $q_0 \in Q$ and F is the set of final states, $F \in Q$.
Our approach defines Q as a set of states, Σ as a set of terminal symbols, q_0 is of the type of the states of the automaton and F is defined as a function that maps a state to a boolean, checking whether a state is final or not. We formalize DFA as follows, where A is the type of the symbols of the alphabet of DFA and S is the type of the states of DFA:

```
Record t := DFA {
    initial_state : S;
    is_final : S → bool;
    next : S → A → S;
    states: set S;
    alphabet: set A
 }.
```

Lemma 9 (Soundness of *dfa.run*)**.** ∀ *m: dfa.t NT T,* ∀ *l:list T,*
dfa.run m l = *true* ↔
(dfa.is_final m

(dfa.run' (dfa.next m) l
(dfa.initial_state m)) = true).

This lemma states that the run of a DFA on a word will return true iff, after checking the whole word, starting from the starting state of the automaton, it reaches a final state of the automaton.

Proof.
⇒
Let H: *dfa.run m l* returns *true*. We want to prove (*dfa.is_final m* (*dfa.run' (dfa.next m) l* (*dfa.initial_state m*))) returns *true*. We proceed by rewriting H in the goal. The goal is *dfa.run m l* = (*dfa.is_final m* (*dfa.run' (dfa.next m) l* (*dfa.initial_state m*))). By unfolding the definition of *run* in the goal, it is (*dfa.is_final m* (*dfa.run' (dfa.next m) l* (*dfa.initial_state* (*m*))) = (*dfa.is_final m* (*dfa.run' (dfa.next m) l* (*dfa.initial_state m*)), which holds.

⇒

Let H: (*dfa.is_final m* (*dfa.run' (dfa.next m) l* (*dfa.initial_state m*)) returns *true*. We want to prove that *dfa.run m l* returns *true*. By rewriting H in the goal, it is
dfa.run m l true = *dfa.is_final m* (*dfa.run' (dfa.next m) l* (*dfa.initial_state m*)). By unfolding *dfa.run*, the goal is
dfa.is_final m (*dfa.run' (dfa.next m) l* = *dfa.is_final m* (*dfa.run' (dfa.next m) l*, which holds.

□

3.1 Conversion of NFA to DFA

Let M a DFA of type *dfa.t ST A*, where ST is the type of the states and A the type of symbols of the alphabet of the DFA M.

Lemma 10 (Soundness of *list_states*)**.** *this lemma states that the function list_states with a state S and a set of terminal symbols A of a DFA M returns all next states from S that are in M.*

Proof. The proof is straightforward from *list_states*'s definition. By unfolding *list_states*, it is:

```
match A with
  | [] ⇒ []
  | a::t ⇒ if (dfa.next S a) then
       set_add equiv_dec (dfa_next s a) (list_states S t)
       else list_states S t
end.
```

Hence, *list_states* applies the transition relation defined in M for the state S with every symbol in M's alphabet. □

Lemma 11 (Soundness of *s1_in_s2*)**.** *The function s1_in_s2 with two sets A and B returns true iff all elements of A are in B and false otherwise.*

Proof. The proof is straightforward from *s1_in_s2*'s definition. By unfolding *s1_in_s2*, it is

```
match A with
| [] ⇒ true
| a::t ⇒ set_mem equiv_dec a B && s1_in_s2 t B
end.
```

where *set_mem* is a function within `Module` *ListSet* that returns true if a given element a is in a given set B and false otherwise. By applying *set_mem* a B for every element a in A as a conjunction between all executions of *set_mem* a B, we have that *s1_in_s2* returns true iff all elements of A are in B, proving this lemma. □

Lemma 12 (Soundness of *set_eq*)**.** *The function set_eq returns true iff given two sets A and B, all elements of A are in B and all elements of B are in A*

Proof. The proof is straightforward from *set_eq*'s definition. It suffices to unfold *set_eq*'s definition and verify that it returns *true* if both sets have the same cardinality and if all elements of A are in B (and vice-versa) which is verified by applying *s1_in_s2* A B and *s1_in_s2* B A. Therefore by Lemma 11, this lemma is proved.

□

The following lemma's function is defined in the module *dfa_to_nfa*, where states are defined as sets of variables of ST type. Therefore, a state in *dfa_to_nfa* is of the kind `set` ST.

Lemma 13 (Soundness of *bounded_search*)**.** *The function bounded_search checks if, given two states A and B and a integer n it returns true if B can be reached from A in at maximum n steps using $\sigma(A, x)$ where x is any symbol in M's alphabet. In other words, from A, the function checks whether B is a state in the neighborhood of A with any symbol in M's alphabet: if it is, then it returns true, otherwise it calls bounded_search for all states in the neighborhood of A recursively. This function uses a natural number n to provide a bounded search in the state space (it is the most intuitive way to implement such function in Coq.)*

Proof. let H be "The function *bounded_search* returns true iff a given state B can be reached from A in at maximum n steps with any symbol defined in M's alphabet". The proof is done by strong induction on the number of steps n given as parameter. Therefore, let:

(i) Induction basis: Let H be "The function *bounded_search* returns true iff a given state B can be reached from A in at maximum 0 steps. It is the case that no steps are taken. Hence, by *bounded_search*'s definition, it returns *true* iff $A = B$ and false otherwise.

(ii) Induction Hypothesis: Let H now be "The function *bounded_search* returns true iff a given state B can be reached from A in at maximum k steps, where $1 \leq k \leq n$, with any symbol in M's alphabet.

(iii) Inductive Step: We want to prove that "The function *bounded_search* returns true iff a given state B can be reached from A in at maximum $n+1$ steps" holds. From the inductive hypothesis, we have that H holds where the number of steps taken is n. Then, (*bounded_search* A B n) returns true iff B is reachable from A in at maximum n steps and false otherwise. Let us call C the set of states that contains the last states visited by *bounded_search* A B n. By adding 1 to the number of steps n taken by *bounded_search*, *bounded_search* will check if there is a state in the neighborhood of some state in C that equals to B. This is equivalent to (*bounded_search* A B $n+1$) (starting the search from A with $n+1$ as the number of maximum steps taken). Therefore the hypothesis holds for $n+1$.

□

Lemma 14 (Soundness of *get_all_reachable_states*)**.** *The function get_all_reachable_states with a set of states S returns all states that are reachable from the initial state of M that are in s in at maximum length(dfa.states M) steps.*

Proof. It is sufficient to unfold *get_all_reachable_states* and verify that *get_all_reachable states* adds all states in S that are reachable from the initial state of M (using *bounded_search*) in the returning set of states returned by *get_all_reachable_states*. Therefore, by Lemma 13 this lemma is proved. □

Lemma 15 (Soundness of *dfa_states*)**.** *The function dfa_states returns a set of set of states of M that are reachable from the initial state of M. In other words, it returns all states in the power set of Q (where Q is the set of states of M) that are reachable from M's initial state.*

Proof. This proof is completely straightforward from *dfa_states*. By unfolding its definition, *dfa_states* calls *get_all_reachable_states* with the power set of M's states. □

Lemma 16 (Soundness of the function *build_dfa_from_nfa*)**.** *This lemma states the soundness of the function build_dfa_from_nfa, which implements the conversion of a NFA to a DFA. We implement the algorithm described in [Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman].*

Proof. By the definition of *build_dfa_from_nfa*, The initial state of the resulting DFA is the same of the NFA. We define a valid state of the DFA as a set of states of the NFA. We then proceed by creating all possible new states of the DFA by constructing the powerset of Q (where Q is the set of states of the NFA) since states of the DFA are all subsets of the powerset of the set of states

of the NFA. Then, we define the transition function of the resulting DFA being the transition function of states of the NFA that are part of a state in the DFA (for every state of the DFA there is a transition from it iff there is a transition from one state of the NFA that is part of it). The set of final states is modeled as a function which verifies if a state in the DFA contains at least a final state in the NFA and the DFA's alphabet is the same of the NFA. We build the set of states of the resulting DFA by listing all states that are reachable from the DFA's initial state by checking in the powerset of Q states that can be reached from the initial state. The complete proof of the algorithm can be found in [Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman].

□

3.2 Conversion of Regular Grammars to DFA

Lemma 17 (Soundness of the conversion of a regular grammar to a DFA)**.** *The conversion of a regular grammar to a DFA is implemented according to the* Construction Algorithm 1 *in [Zhang and Qian(2013)] and is implemented in the section powerset_construction. We generate the corresponding NFA with the described algorithm and convert it to a DFA "on the fly" using the conversion of a DFA to a NFA algorithm described in [Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman]. A state in the DFA is formed by zero or more nonterminal symbols of the grammar. Therefore, for every $a \in T$ and $A, B \in V$:*

$A \to a \in P$ holds, $\sigma(A, a) = f$ holds in the DFA, where f is a newly created state, $f \notin V$ and $f \in F$ (F being the set of final states of the DFA). We use the None as this newly created state where all transitions $\sigma(A, a) = None$ holds. By the definition of step_rhs, None is a derivation state that represents a rule of the kind $A \to a$ could be used to end the derivation process.Hence, having $\sigma(A, a) = None$ holds, None being a final state in the DFA.

$A \to aB \in P$ holds, $\sigma(A, a) = B$ holds in the DFA. If $B \to \epsilon \in P$ holds, then $B \in F$ holds.

For both cases of rules, for any $C \in V$ if there is any rule $A \to aB \in P$ and $A \to aC \in P$ the corresponding state in the DFA is a set formed by both B and C. Also, if there is any rule $A \to aB \in P$ and $A \to a \in P$, the corresponding state in the DFA is a set formed by both B and the new state formed, represented by None.

The set of states of the DFA is built by the same way described in 16

3.3 Conversion of DFA to Regular Grammars

Lemma 18 (Soundness of the conversion of a DFA to a regular grammar)**.** *The conversion of a regular grammar to a DFA is implemented according to*

Construction Algorithm 4 *in [Zhang and Qian(2013)] where its proof can be found. Thus, for every* $a \in \Sigma$ *and* $A, B \in S$*:*

$\delta(A, a) = B$ *holds, we proceed by using the function get_transitions_from_a_state, which gets all the transition rules from a given state and the DFA's alphabet and then generate rules of the kind* $A \to aB$*, where* A *is the state,* a *is a symbol of the alphabet and* B *is the next state in the transition. Then, the function add_rules calls get_transitions_from_a_state with all states of the DFA, returning a set of all possible rules of the kind* $A \to aB$ *that could be extracted from the DFA.*

***get_transitions_from_a_state* is sound** *: By its definition, given a state and the alphabet of the DFA, get_transitions_from_a_state creates a rule of the kind* $A \to aB$ *whenever* $\delta(A, a) = B$ *holds, returning a set of rules of this kind for all* $a \in \Sigma$*.*

$B \in F$ *holds, we create a rule* $B \to \epsilon$ *with the function get_empty_rules. Therefore, we proceed by using the function get_empty_rules, which generate rules of the kind* $A \to \epsilon$*, where* ϵ *is the empty string character,* A *is a state reachable from the state given to get_empty_rules with any symbol of the DFA's alphabet and* A *is a final state of the DFA.*

***get_empty_rules* is sound** *: given a state and the DFA's alphabet, the function get_empty_rules returns a set with all empty rules from all final states that can be reached from this terminal with any symbol from the NFA's alphabet.*

Then we use the function add_rules to generate all possible empty rules, using add_empty_rules with every state of the DFA, returning a set with all possible derivation rules extracted from the DFA.

Lemma 19 (The run on an automaton built from a regular grammar may return the same result as the parser for that grammar)**.** $\forall$ *l,* $\forall$ *g,*
reg_grammar.parse g l = dfa.run (powerset_construction.build_dfa g) l .

Proof. The proof is straightforward from the way the conversion algorithm is implemented: By the definition of *powerset_construction.build_dfa*, we have that the starting state of the automaton is the start symbol of the grammar, final states of derivation are the same of the grammar and the transition function of the automaton is the derivation step done by the grammar. From lemma 19, we have that the equality holds (this is proved in *Coq* using the `reflexivity` tactic.) □

3.4 Conversion of DFA to NFA

Lemma 20 (All final states of a DFA are the same final states of an NFA built from a DFA)**.** $\forall$ *l,* $\forall$ *s,*

dfa.is_final m (dfa.run' (dfa.next m) l (s)) =
nfa.verify_final_state (dfa.dfa_to_nfa m) (nfa.run' (dfa.dfa_to_nfa m) l [s]).

This lemma states that for all states of a DFA and for all lists of terminal symbols, if after a run of the list on a DFA, the automaton reaches a final state, then the NFA built from this DFA may also be in a final state of the NFA (the same happens for the case of not reaching a final state).

Proof. The proof is done by induction on *l*. Therefore, let l be:

[ ] where the first subgoal is $\forall$ *s*,
dfa.is_final m (dfa.run' (dfa.next m) [] (s)) =
nfa.verify_final_state (dfa.dfa_to_nfa m) (nfa.run' (dfa.dfa_to_nfa m) [] [s]). By simplifying the goal, it is $\forall$ *s*,
dfa.is_final m s = (`if` *dfa.is_final m s* `then` *true* `else` *false*).
We proceed by introducing *s* at this point only, so the induction hypothesis generated will be more general. The goal is *dfa.is_final m s* = (`if` *dfa.is_final m s* `then` *true* `else` *false*). By destructing *dfa.is_final*, it is generated two subgoals:

(i) *true* = *true* which holds.

(ii) *false* = *false* which holds.

a::l where the second subgoal is $\forall$ *s*,
dfa.is_final m (dfa.run' (dfa.next m) (a::l) (s)) =
nfa.verify_final_state (dfa.dfa_to_nfa m) (nfa.run' (dfa.dfa_to_nfa m) (a::l) [s]), where *a* is an element of the type of the terminal symbols, *l* is a list of terminal symbols and *a::l* represents a list with *a* as the head of the list entailed by *l*. We proceed by introducing *s* as a fresh variable of the type of the states of the DFA and then simplifying the goal. It is *dfa.is_final m (dfa.run' (dfa.next m) (l) (dfa.next m s a)) =*
nfa.verify_final_state (dfa.dfa_to_nfa m) (nfa.run' (dfa.dfa_to_nfa m) (l) [dfa.next m s a]). Since the induction hypothesis, *IH*: $\forall$ *s*,
dfa.is_final m (dfa.run' (dfa.next m) l (s)) =
nfa.verify_final_state (dfa.dfa_to_nfa m) (nfa.run' (dfa.dfa_to_nfa m) l [s]) holds for any state, it also holds if the state is *dfa.next m s a*. Then, we can rewrite *IH* in the goal. The goal is *nfa.verify_final_state (dfa.dfa_to_nfa m) (nfa.run' (dfa.dfa_to_nfa m) l [dfa.next m s a]) = nfa.verify_final_state (dfa.dfa_to_nfa m) (nfa.run' (dfa.dfa_to_nfa m) l [dfa.next m s a])*, which holds.

□

Lemma 21 (Soundness of a run on a DFA and on the NFA built from this DFA)**.** $\forall$ *l*,
dfa.run m l = nfa.run (dfa.dfa_to_nfa m) l.

This lemma states that for any list of terminal symbols, the result of a run in a DFA and a run on the NFA built from the given DFA may return the same result.

Proof. We proceed by first unfolding the definition of *nfa.run* and *dfa.run*. The goal is *dfa.is_final m* (*dfa.run'* (*dfa.next m*) *l* (*dfa.initial_state m*)) = *nfa.verify_final_state* (*dfa.dfa_to_nfa m*) (*nfa.run'* (*dfa.dfa_to_nfa m*) *l* [*dfa.initial_state m*]). Then, the proof is done by induction on *l*, generating two subgoals. Hence, let l be:

[ ] where the first subgoal is
dfa.is_final m (*dfa.run'* (*dfa.next m*) [] (*dfa.initial_state m*)) = *nfa.verify_final_state* (*dfa.dfa_to_nfa m*) (*nfa.run'* (*dfa.dfa_to_nfa m*) [] [*dfa.initial_state m*]). By simplifying the goal, it is *dfa.is_final m* (*dfa.initial_state m*) = (`if` *dfa.is_final m dfa.initial_state* `then` *true* `else` *false*). We proceed by destructing *dfa.is_final*, generating two new subgoals:

(i) *true* = *true* which holds.

(ii) *false* = *false* which holds.

a::l , the subgoal is
dfa.is_final m (*dfa.run'* (*dfa.next m*) *a::l* (*dfa.initial_state m*)) = *nfa.verify_final_state* (*dfa.dfa_to_nfa m*) (*nfa.run'* (*dfa.dfa_to_nfa m*) *a::l* [*dfa.initial_state m*]), where *a* is a terminal symbol, *l* a list of terminal symbols and *a::l* stands for a list headed by *a* and entailed by *l*.
Therefore, let *H*: *dfa.is_final m* (*dfa.run'* (*dfa.next m*) *l* (*dfa.initial_state m*)) =
nfa.verify_final_state (*dfa.dfa_to_nfa m*) (*nfa.run'* (*dfa.dfa_to_nfa m*) *l* [*dfa.initial_state m*]) as the induction hypothesis. This goal can be proved by rewriting lemma 21: since this lemma states that a run starting at any state of a DFA may return the same thing that a run starting at any state of the NFA built from this DFA and we want to prove that a run with a list of terminal symbols on a DFA, starting in its initial state, will return the same value as the run for the same list on a NFA, starting in its initial state, built from the DFA. Then, after rewriting Lemma 21, the goal is *nfa.verify_final_state* (*dfa.dfa_to_nfa m*) (*nfa.run'* (*dfa.dfa_to_nfa m*) *a::l* [*dfa.initial_state m*] = *nfa.verify_final_state* (*dfa.dfa_to_nfa m*) (*nfa.run'* (*dfa.dfa_to_nfa m*) *a::l* [*dfa.initial_state m*], which holds.

□

3.5 Minimal DFA verification

The notion of a minimal automaton implemented follows the same as described in [Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman]: two states p and q of an automaton are equivalent iff, for each symbol a in the alphabet of the automaton, $\delta(p, a) = \delta(q, a)$. In other words, p and q are equivalent if, for each symbol in the alphabet of the automaton, p and q go to the same state. To check if an automaton is minimal, we formalized three functions: *check_pair_states*, which is the function that check if two given states are equivalent, *check_a_pair_states*,

which is the function that checks for a given state and all the other states of the automaton if there is a pair of states (a, t), where a is the given state and t is a state in the set of states of the automaton, and *has_no_equivalent_states*, which is the function that checks, for all states of the automaton, if there is a pair of states that are equivalent.

Lemma 22 (Soundness of *check_pair_states*)**.** *The function check_pair_states returns true iff the pair of states (A,B) is equivalent (A and B being any state in Q, the set of states of the DFA).*

Proof. The definition of this function follows exactly the checking of such states described in [Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman]. □

Lemma 23 (Soundness of *check_a_pair_states*)**.** *The function check_a_pair_states returns true iff, given a state of the DFA and the set of states, exists a pair of equivalent states (A, B) where A is the state given as parameter and $B \in Q$ (Q the set of states of the DFA).*

Proof. The function described above calls *check_pair_states* for a given state A of the DFA and all states of the DFA (different from A). By *check_a_pair_states*'s definition, it will return true iff it finds at least a pair of states (A, B) where A and B are equivalent (as defined in *check_pair_states*). Therefore, by Lemma 22, this lemma is proved. □

Lemma 24 (Soundness of *has_equivalent_states*)**.** *∀ m: dfa.t NT T, ∀ s: list NT, ∀ l: list T, dfa.has_equivalent_states = true ↔ ∃ a, In a s ∧ dfa.check_a _pair_states m (a) s l = true.*

This lemma states that the function dfa.has_equivalent_states with m as a DFA will return true iff there is a equivalent state in the list of states (which is checked by the check_pair_states function).

Proof. The proof is straightforward from *has_equivalent_states*'s definition. By unfolding *has_equivalent_states*, it is

```
fix rec s1 s2 l :=
      match s1 with
      | [] ⇒ false
      | a::t ⇒ check_a_pair_states m (a) s2 l || rec t s2 l
      end.
```

Hence, *has_equivalent_states* uses *check_a_pair_states* to retrieve all pair of states in the two sets given as parameter. Therefore, it returns true only if it retrieves at least one pair of equivalent states. Thus, by Lemma 23, this Lemma is proved.

□

Lemma 25 (Soundness of *is_minimal*)**.** *∀ m: dfa.t NT T, dfa.is_minimal (m) = true ↔ has_no_equivalent_states (dfa.states m) (dfa.states m) (dfa.alphabet m) = false.*

This lemma states that the function dfa.is_minimal m returns true iff the function has_no_equivalent_states, which checks over m's set of states and the alphabet, checking for every possible pair of states of the automaton and for every symbol of the alphabet, if there is a pair of states that are equivalent.

Proof.
⇒

Let H: *dfa.is_minimal m = true*. We want to show that *dfa.has_no_equivalent_states m (dfa.states m) (dfa.states m) (dfa.alphabet m) = false*. By unfolding the definition of *dfa.is_minimal*, H is *negb (dfa.has_no_equivalent_states m (dfa.states m) (dfa.states m) (dfa.alphabet m)) = true*. By *negb*'s definition we have that *negb true = false* and *negb false = true*, where *negb* is the boolean negation. Then, by H, we have that *negb (dfa.has_no_equivalent_states m (dfa.states m) (dfa.states m) (dfa.alphabet m)) = true* iff *dfa.has_no_equivalent_states m (dfa.states m) (dfa.states m) (dfa.alphabet m) returns false*, so *negb false = true*. Then, we have that the goal holds.

⇐

Let H : *dfa.has_no_equivalent_states m (dfa.states m) (dfa.states m) (dfa.alphabet m) = false*. We want to show that *dfa.is_minimal m = true* holds. By unfolding *dfa.is_minimal* in the goal, it is *negb (dfa.has_no_equivalent_states m (dfa.states m) (dfa.states m) (dfa.alphabet m)) = true*. From the goal and the definition of *negb*, *negb dfa.has_no_equivalent_states m (dfa.states m) (dfa.states m) (dfa.alphabet m) returns true* iff *dfa.has_no_equivalent_states m (dfa.states m) (dfa.states m) (dfa.alphabet m) returns false*, so *negb false = true*. Then, the goal holds.

□

Lemma 26 (Soundness of *get_equivalent_states*)**.** *The function get_equivalent_states returns pairs of equivalent states, using check_pair_states to verify if two given states of the automaton are equivalent.*

Proof. The proof is straightforward from *get_equivalent_states*'s definition. Given a state, a set of states and a set of terminal symbols, this function returns a set containing pairs of the form (S, X), where S is a given state of the DFA and X is a state, $X \in Q$ and $S \neq X$. For each state of the DFA and the state given as parameter, *get_equivalent_states* will check if a state of the DFA and the given state are equivalent using *check_pair_state*. The resulting set contains equivalent states of the DFA which are equivalent to the given state as parameter. □

Lemma 27 (Soundness of *get_all_equivalent_states*)**.**

The function get_all_equivalent_states is the function that verifies there are equivalent states in sets of states, returning a set with all pairs of equivalent states.

Proof. The proof is straightforward from *get_all_equivalent_states*. Given two set of states and a set of nonterminal symbols, it uses *get_equivalent_states* for each element in both sets of states, returning a set with all equivalent states within the sets of states given. □

Lemma 28 (Soundness of *check_equivalent_states*)**.** *Given a DFA, the function check_equivalent_states will return a set with pairs that contains all equivalent states that belongs to the DFA.*

Proof. The function *check_equivalent_states* uses *get_equivalent_states*, passing to *get_equivalent states* the DFA's set of states in both arguments and the DFA's alphabet. Then, *get_equivalent_states* will return all pairs with all equivalent states of the DFA. This set is therefore returned by *check_equivalent_states*. □

4 Nondeterministic finite automata

This section presents lemmas about nondeterministic finite automata (NFA) and its formalization. We formalize NFA according to
[Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman]

A NFA is a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q, Σ, δ, q_0 and F have all the same meaning as for DFA, but δ is a function which support is $(Q \times \Sigma) \to 2^Q$, where 2^Q is the power set of Q. We formalize NFA as follows:

```
Record t := NFA {
    initial_state : S;
    is_final : S → bool;
    next : A → S → set S;
    states: set S;
    alphabet: set A
}.
```

Lemma 29 (Soundness of *nfa.run*)**.** ∀ *l:list A, nfa.run nfa l = true* ↔ *nfa.verify_final_state nfa (nfa.run' (nfa) l ([(nfa.initial_state nfa)])) = true.*

This lemma states that after a run of the NFA in a list of terminal symbols, it will return true iff it reaches a final state, starting the run in the starting state of the NFA.

Proof.
⇒
Let *H*: *nfa.run nfa l* returns *true*. We want to prove *nfa.verify_final_state nfa* (*nfa.run'* (*nfa*) *l* ([(*nfa.initial_state nfa*)]) returns *true*. We proceed by rewriting *H* in the goal. The goal is *nfa.verify_final_state nfa* (*nfa.run'* (*nfa*) *l* ([(*nfa.initial_state nfa*)]) = *nfa.run nfa l*. By unfolding the definition of *nfa.run*,

the goal is *nfa.verify_final_state nfa (nfa.run' (nfa) l ([(nfa.initial_state nfa)]) = nfa.verify_final_state nfa (nfa.run' (nfa) l ([(nfa.initial_state nfa)])*, which holds.

$\Leftarrow$

Let H: *nfa.verify_final_state nfa (nfa.run' (nfa) l ([(nfa.initial_state nfa)])* returns *true*. We want to prove that *nfa.run nfa l* returns *true*. We proceed by rewriting H in the goal. The goal is *nfa.run nfa l = nfa.verify_final_state nfa (nfa.run' (nfa) l ([(nfa.initial_state nfa)])*. By unfolding the definition of *nfa.run*, the goal is *nfa.verify_final_state nfa (nfa.run' (nfa) l ([(nfa.initial_state nfa)]) = nfa.verify_final_state nfa (nfa.run' (nfa) l ([(nfa.initial_state nfa)])*, which holds. □

Lemma 30 (Soundness of building a regular grammar from NFA)**.** *The conversion of a regular grammar to a NFA is implemented in Coq by the name build_grammar_from_nfa in accordance with* Construction Algorithm 4 *in [Zhang and Qian(2013)], where the proof of the algorithm can be found. Thus, for every $a \in \Sigma$ and $A, B \in S$:*

$\delta(A, a) = B$ *holds, we create a rule $A \to aB$ with the function get_every_state. This function implements the presented idea for a single state. To build all possible nonempty rules, the function get_all_rules is used, which uses get_every_state for each state of the NFA.*

***get_every_state* is sound** *: from the definition of NFA the support of the transition relation is $(Q \times \Sigma) \to 2^Q$, where 2^Q is the power set of Q. Therefore, get_every_state creates a valid rule of the kind $A \to aB$, for every possible B reachable from A with all $a \in \Sigma$, extracting all possible rules from the NFA.*

$B \in F$ *holds, we create a rule $B \to \epsilon$ with the function get_empty_rules.*

***get_empty_rules* is sound** *: given a state and the NFA's alphabet, the function get_empty_rules returns a set with all empty rules from all final states that can be reached from this terminal with any symbol from the NFA's alphabet.*

The function dfa_transitions_to_grammar_rules uses both functions to create the set of all production rules of the resulting grammar, using get_all_rules (a function that only applies get_every_state to all states of the DFA) and the function add_empty_rules (a function that only applies get_empty_rules to all states of the DFA).

Lemma 31 (Soundness of building a NFA from a regular grammar)**.** *The conversion of a regular grammar to a NFA is implemented according to* Construction Algorithm 1 *in [Zhang and Qian(2013)] where its proof can be found. Therefore, for every $a \in T$ and $A, B \in V$:*

$A \to a \in P$ *holds,* $\sigma(A, a) = f$ *holds in the NFA, where* f *is a newly created state,* $f \notin V$ *and* $f \in F$ *(*F *being the set of final states of the NFA). We use the None from the Option type in Coq as this newly created state where all transitions* $\sigma(A, a) = f = \sigma(A, a) = None$ *holds. By the definition of step_rhs, None is a derivation state that represents a rule of the kind* $A \to a$ *could be used to end the derivation process. Hence, having* $\sigma(A, a) = None$ *holds, None being a final state in the DFA also holds.* $\sigma(A, a) = None$ *is extracted directly from the derivation process by using the step function.*

$A \to aB \in P$ *holds,* $\sigma(A, a) = B$ *holds in the NFA.* $\sigma(A, a) = B$ *is extracted directly from the derivation process by using the step function.*

The set of states in the NFA is V *from the given grammar (with None added iff there is at least a rule* $A \to a \in P$*), the alphabet of the NFA is* T *and the starting state of the NFA is* S*.*

5 Nondeterministic Finite Automata with ϵ-transitions

This section discuss the formalization of NFA with ϵ-transitions (NFA-ϵ) which are NFA with transitions in the empty string ϵ. In other words, let q, q' states of a given NFA. if $\delta(q, \epsilon) = q'$ holds in the NFA, then the NFA may go from q to q' without consuming the symbol in the input tape.

A NFA is defined as a quintuple $(Q, \Sigma, \delta, q_0, F)$ where all elements of the quintuple have the same meaning as in Section 4, but δ, the transition function, maps Q x $(\Sigma \cup \{\epsilon\})$ to 2^Q. It corresponds to the following definition in *Coq* (defined according to [Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman]):

```
Record t := NFA_e {
  initial_state : ST;
  is_final : ST → bool;
  next : ST → set (nfa_epsilon_transitions.ep_trans ST A);
  states : set ST;
  alphabet : set A
}.
```

where ST stands for the type of the states of the NFA-ϵ and A stands for the type of its alphabet.

The main difference in the formalization of NFA and NFA-ϵ here consists in the *next* function (which represents the transition function of both): for NFA, it is defined as $ST \to A \to A$ whereas in NFA-ϵ it is defined as $ST \to$ `set` (*nfa_epsilon_transitions.ep_trans* ST A). This way, it is possible to capture both transitions of the NFA (ϵ-transitions, which does not consume symbols from the input table and transitions defined for NFA without ϵ-transitions and DFA) at the same time.

Hence, *nfa_epsilon_transitions.ep_trans* is an inductive type defined in the module *nfa_epsilon_transitions* which captures all possible transitions of this class of NFA:

```
Inductive ep_trans S A :=
| Epsilon : S → ep_trans S A
| Goes : A → S → ep_trans S A.
```

where *Epsilon* the constructor of the inductive type *ep_trans* denotes an ϵ-transition (note that there is no symbol of the alphabet defined for this transition) and *Goes* is the constructor of the inductive type that denotes transitions on symbols of the alphabet.

To define a run in this kind of NFA, one must first convert the NFA with ϵ-transitions to a NFA without those transitions which simulates the given ϵ-NFA. This is done by the algorithm described in
[Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman] by calculating the ϵ-closure of all states using functions which soundness are proved below.

5.1 Helpers

In this subsection it is presented the soundness of helpers that will be used in the main functions that manipulates NFA with epsilon-transitions (ϵ-NFA) For all lemmas that succeed, the variable m stands for a given ϵ-NFA of type *nfa_epsilon.t*, where t is the record where our formalization of a ϵ-NFA is done.

Lemma 32 (Soundness of *next_state_w_e*)**.** *The function next_state_w_e returns, given a set of transitions of type* `set` *(nfa_epsilon_transitions.ep_trans ST A), all next states that are reachable only with ϵ-transitions. The returning set of states are all obtained only from Epsilon transitions.*

Proof. The proof is straightforward from *next_state_w_e*'s definition. Let s be a set. Then, by destructing the set, we have two goals:

the returning set of states with *next_state_w_e* $\emptyset$ contains only states reachable by a ϵ-transition holds: by *next_state_w_e*'s definition, since it goes through a set of possible transitions adding to the resulting set only states that are reachable by ϵ-transitions, all states in the resulting set (in this case, zero states) are reached by ϵ-transitions. By simplification of *next_state_w_e*, the goal holds.

the returning set of states with *next_state_w_e a::l* contains only states reachable by ϵ-transitions] holds, where a is an element of the set and l is the rest of the set. By *next_state_w_e*'s definition,since it goes through the given set of transitions, adding to the resulting set only states that are reachable by ϵ-transitions. Therefore, the returning set will have at maximum n elements (where n) is the length of the given set of transitions, in which case all transitions are of the epsilon kind and at minimum

0 elements (in which case all transitions are not of the epsilon kind).By simplification of *next_state_w_e*, the goal holds.

□

Lemma 33 (Soundness of *get_goes_transitions*)**.** *The function get_goes_transitions returns, given a set of transitions of type* `set` *(nfa_epsilon_transitions.ep_trans ST A), transitions defined for any symbol in a given ϵ-NFA. In other words, the returning set of states is made by transitions of the kind Goes, one of the inductive constructors of the type (nfa_epsilon_transitions.ep_trans).*

Proof. The proof is straightforward from *get_goes_transitions*'s definition. We want to prove that *get_goes_transitions s* returns always set of states reachable with transitions that are not ϵ-transitions. Let s be a set. Then, by destructing the set, we have two goals:

the returning set of states by *get_goes_transitions* $\emptyset$ contains only sta tes reachable by a ϵ-transition holds: by *get_goes_transitions*'s definition, since it goes through a set of transitions adding to the resulting set states that are reachable by transitions that are not ϵ-transitions, all states in the resulting set (in this case, zero states) met this condition. By simplification of *get_goes_transitions*, the goal holds.

the returning set of states by *get_goes_transitions a::l* contains only sta tes reachable by ϵ-transitions] holds, where a is an element of the set and l is the rest of the set. By *get_goes_transitions*'s definition it goes through the given set of transitions, adding to the resulting set states that are reachable by non ϵ-transitions. Therefore, the returning set will have at maximum n elements where n is the length of the given set of transitions, in which case all transitions are of the epsilon kind and at minimum 0 elements (in which case all transitions are not of the epsilon kind). By simplification of *get_goes_transitions*, the goal holds.

□

Lemma 34 (Soundness of *ep_trans_is_epsilon*)**.** $\forall$ *s:nfa_epsilon_transitions. ep_trans ST A,*

nfa_epsilon.ep_trans_is_epsilon s = true $\leftrightarrow$ *($\exists$ t:ST, s = Epsilon t).*

This lemma states that the function ep_trans_is_epsilon returns true iff the given transition is of the kind Epsilon x, where x is of the same type of the states of the NFA with ϵ-transitions.

Proof.
$\Rightarrow$
Let H: *nfa_epsilon.ep_trans_is_epsilon s = true.* We want to show that ($\exists$ t:*ST*, s = *Epsilon* t) holds. We proceed by destructing s. Then, let *nfa_epsilon.ep_trans_is_epsilon* with

Epsilon s = true , the goal is ∃ *t*:*ST*, *Epsilon s* = *Epsilon t*. By destructing *s*, a new variable is introduced in the proof context of type ST (which is named *s*). Hence, we can use it to instantiate in the existential quantifier. The goal is *Epsilon s* = *Epsilon s*, which holds.

Goes a s = true is a contradiction. Then, the goal being ∃ *t*:*ST*, *Goes a s* = *Epsilon t* is discharged.

⇒
Let *H*: ∃ *t*:*ST*, *s* = *Epsilon t*. We want to prove that *nfa_epsilon.ep_trans _is_epsilon s* = *true* holds. By destructing *H*, we can obtain another hypothesis, say *H1*: *s* = *Epsilon x*. Then, by rewriting *H1* in the goal it is *nfa_epsilon.ep_trans*
_is_epsilon Epsilon x = *true* which by simplification holds. □

Lemma 35 (Soundness of *has_no_e_transitions*)**.** *This lemma states that the function has_no_e_transitions returns true iff a given set of transitions S does not contain any transitions that are of the Epsilon kind.*

Proof.
The proof is straightforward from *has_no_e_transitions*'s definition: The function goes recursively through a set of transitions of type (*set nfa_epsilon_trans itions.ep_trans ST A*) and checks for every element if there is one that is of *Epsilon* kind using the function *ep_trans_is_epsilon*. It only returns true iff it reaches the end of the set (it has already checked all elements of the set) and no one is of the *Epsilon* and returns false otherwise. □

Lemma 36 (Soundness of *create_transition*)**.** *The function create_transition returns , given a state x and a set of transitions of the ϵ-NFA S (where those transitions have type nfa_epsilon_transitions.ep_trans ST A, ST being the type of the states of m and A being the type of the elements of the alphabet of m), it returns transitions (x,a) where a is a transition with a ∈ S. We want to prove that create_transition returns either an empty list or a list with elements, where those elements are pairs of transitions (x,a).*

Proof. This proof is straightforward from *create_transition*'s definition. We proceed by destructing the set *S* of transitions given as input. Therefore, let *create_transition x* with

the empty set ∅ then by *create_transition*'s definition it will return a set with all transitions that can be built from the set (in this case, 0 transitions).

a set with elements *a::t* where *a* is a transition of the type *nfa_epsilon_transitions.ep_trans ST A*, *a* ∈ *S* and *t* is the remainder of the set *S*. By unfolding *create_transitions*'s definition, one can see that it adds to the resulting set a transition (*x*,*a*) for every *a* ∈ *S* recursively, calling *create_transition x* with *t* .

□

Lemma 37 (Soundness of *bounded_search*)**.** *The function bounded_search (defined in the module nfa_epsilon) checks if, given two state A and B and a integer n it returns true if B can be reached from A in at maximum N steps with only ϵ-transitions. From A, the function checks whether B is a state in the ϵ-neighborhood of A. If it is, then it returns true, otherwise it calls bounded_search for all states in the neighborhood of A recursively. This function uses a natural number n to provide a bounded search in the state space (it is the most intuitive way to implement such function in Coq.)*

Proof. let H be "The function *bounded_search* returns true iff a given state B can be reached from A in at maximum n steps". The proof is done by strong induction on the number of steps n given as parameter. Therefore, let:

(i) Induction basis: Let H be "The function *bounded_search* returns true iff a given state B can be reached from A in at maximum 0 steps. It is the case that no steps are taken. Hence, by *bounded_search*'s definition, it returns *true* iff x $A = B$ and false otherwise.

(ii) Induction Hypothesis: Let H now be "The function *bounded_search* returns true iff a given state B can be reached from A in at maximum k steps, where $1 \leq k \leq n$.

(iii) Inductive Step: We want to prove that "The function *bounded_search* returns true iff a given state B can be reached from A in at maximum $n+1$ steps" holds. From the inductive hypothesis, we have that H holds where the number of steps taken is n. By *bounded_search*'s definition, we have that if H holds for n steps. Let us call C the last state reached by *bounded_search* A B n. Then, (*bounded_search* A B n) returns true iff B is reachable from A with only ϵ-transitions in at maximum n steps and false otherwise. By adding 1 to the number of steps n taken by *bounded_search*, it will return true iff A equals B and otherwise it will check in the set of the next states reachable from A there is a state that equals B. This is equivalent to (*bounded_search* A B $n+1$). Then, the hypothesis holds for $n+1$.

□

Lemma 38 (Soundness of *get_all_reachable_states_w_e*)**.** *The function get_all_reachable_states_w_e returns, given a state x and a set of states S, all reachable states from S only by ϵ-transitions that are in S. It is worth noting that A must be a state in the same automaton M where the states in S are also states of M.*

Proof. By unfolding *get_all_reachable_states_w_e* x S, where x is a state and S is a set of states, we have:

```
match S with
  | [] ⇒ []
  | a::t ⇒ if (x≠a) then
                if (bounded_search (length(nfa_epsilon.states m))
                                   (x) a)
                 then set_add equiv_dec (a) (get_all_reachable_states_w_e x
t)
                 else get_all_reachable_states_w_e x t
            else get_all_reachable_states_w_e x t
                                   end.
```

Therefore, it is possible to see that it uses *bounded_search* to retrieve all states that are reachable from x in at maximum *length(nfa_epsilon.states m)* steps (which is the size of the space state of m). In the context of a ϵ-NFA, we have that in the worst case a state can be accessible only by passing through all other states in the ϵ-NFA exactly once. Hence, it would take exactly *length(nfa_epsilon .states m) - 1* (where S is the set of states of the ϵ-NFA) steps to reach this state, and in the best case, a state a is reachable from another state x if a is in the neighborhood of x (in other words, $\delta(x, \epsilon) = a$). Specifically for this function, we are interested in states that are only reachable from a given state with ϵ-transitions (which does not change the idea just presented). Therefore, by using *bounded_search x a* (with $a \in S$) we have that *get_all_reachable_states_w_e x S* returns all states from the ϵ-NFA that are reachable from x with only ϵ-transitions. □

Lemma 39 (Soundness of *epsilon_clos*)**.** *The function epsilon_clos calculates the ϵ-closure of a given state. Given a state S, it returns all states reachable by ϵ-transitions from S.*

Proof. Let H be "The function *epsilon_clos* with A given as parameter returns all states reachable from A with only ϵ-transitions. By unfolding the function *epsilon_clos*, its definition is (*get_all_reachable_states_w_e x S length*(*nfa_epsilon.states m*)), where S is a state and *nfa_epsilon.states m* is the set of states of the same automaton x is a state. Therefore, by *get_all_reachable_states_w_e* it returns all states reachable from a given state that are in the set of states given as parameter. S is *nfa_epsilon.states m* (as seen in *next_nfa*'s definition). Hence, *epsilon_clos* will return all states in the automaton m that can be reached from A with ϵ-transitions.

□

Lemma 40 (Soundness of *next_from_state*)**.** *The function next_from_state returns all transitions that are not ϵ-transitions of states that are in the neighborhood of a given state. This function is used later to retrieve nonϵ-transitions of states that are in the ϵ-closure of a state.*

Proof. By unfolding *next_from_state*'s definition.Hence, unfolding *next_from_state* x with x a state given as parameter, we have *get_goes_*

transitions (nfa_epsilon.next m x), where m is a ϵ-NFA of type nfa_epsilon.t. Therefore, by lemma 33, *get_goes_transitions (nfa_epsilon.next m)* returns all transitions that are not ϵ-transitions in the set of states defined in the transition relation for x. □

Lemma 41 (Soundness of *next_from_states*)**.** *The function next_from_states returns, given a set of states S, all transitions that are not ϵ-transitions from states that are defined in the transition relation of the automaton m for each state $x \in S$.*

Proof. This proof is straightforward from *next_from_states*'s definition. By unfolding *next_from_states s* where s is a set of states, it is *flat_map (next_from_state) s*. In other words, it applies the function *next_from_state* to all states in the set s. Therefore it goes by Lemmas 33 and 40. □

After calculating the ϵ-closure of all states in the ϵ-NFA, one is ready to build the NFA without ϵ-transitions that simulates the original ϵ-NFA. In our formalization, we get the corresponding rules for the NFA without ϵ-transitions and formalize them as a set of pairs (x,t) where x of type ST (where ST stands for the inductive type of the states of m)is a state of m and t is a transition of the type (*nfa_epsilon_transitions.ep_trans ST A*).

Lemma 42 (Soundness of *next_from_epsilon_clos*)**.** *The function next_from_epsilon_clos returns, given a set of states S, the transitions defined for any $z \in \Sigma$ (Σ the alphabet of m) from the epsilon closure of all states in the ϵ-NFA m. In other words, it returns the transition relation of the NFA without ϵ-transitions correspondent to m by calculating the ϵ-closure of all states in m and then applying the algorithm presented in Theorem 2.2 in [Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman]*

Proof. Let H be "*next_from_epsilon_clos S* returns all transitions of the equivalent NFA without ϵ-transitions from a given ϵ-NFA m." Therefore, by destructing S, let *next_from_epsilon_clos* with

the empty set $\emptyset$ by *next_from_epsilon_clos* we have that this function will return an empty set, which corresponds to the set of the transition relation that can be built from all states in the set of states given as input (0 states).

a nonempty set a::t where a is a set and t is the remainder of the set given as input. By *next_from_epsilon_clos*'s definition, it calculates the ϵ-closure of a with the function *epsilon_clos*, gets all transitions defined for the states in the ϵ-closure of a using the function *next_from_states* and finally defines transition rules for the state a with all transitions defined for the states in the ϵ-closure of a using the function *create_transition* making the union of this operation with *next_from_epsilon_clos t* by calling the function recursively for the remaining elements in the set of states. Then, by lemmas 36, 39 and 41 this lemma is proved.

□

Lemma 43 (Soundness of *next_nfa*)**.** *The function next_nfa returns a set with all transitions of the NFA without ϵ-transitions corresponding to m.*

Proof. By unfolding *next_nfa*'s definition, it is *set_union equiv_dec (*next_from_epsilon_clos (nfa_epsilon.states m)) where *(*next_from_epsilon_clos (nfa_epsilon.states m)) returns all rules that can be extracted from the ϵ-NFA with all transitions defined for some $z \in m$'s alphabet. Hence, by Lemma 42, this lemma is proved. □

After building the set of transitions of the corresponding NFA without ϵ-transitions from m, We define a function to formalize the run in this NFA named *step* of type $ST \rightarrow A \rightarrow$ (`set` ($ST \times$ *nfa_epsilon_transitions.ep_trans ST A*)) which makes possible the run on this NFA.

Lemma 44 (Soundness of *step*)**.** *The function step returns all next states defined in the transition relation of the NFA built from the ϵ-NFA for a given state s of type ST, a given symbol a in the alphabet of m of type A and a set of transitions S of type (*`set` *($ST \times$ nfa_epsilon_transitions.ep_trans ST A))*

Proof. The proof is done by contradiction. Let s of type ST be a state, a a symbol in the alphabet of m of type A and S of type (`set` ($ST \times$ *nfa_epsilon_transitions.ep_trans ST A*)) a set of transitions (a set of pairs (x,y), x being a state and y a transition of type *nfa_epsilon_transitions.ep_trans ST A* of kind *Goes*). Then, let H: be "The function *step* does not return all next states defined in the transition relation of the NFA built from the ϵ-NFA with s, a and S.

From *step*'s definition, with s, a and S it may return all states z where s is a state in (s,b), where $b = Goes\ a\ z$ for every pair in S. Therefore, *step*'s definition with s, a and S contradicts H. □

Lemma 45 (Soundness of *nfa_step*)**.** *The function nfa_step defines a valid transition relation for the definition of NFAs presented in Section 4 (of type* `A → S → (set S)` *), S being a state and A a symbol in the NFA's alphabet). In other words, nfa_step given a state s and a a symbol of m's alphabet, calls the step function with such arguments and with the set of transitions extracted from the NFA-ϵ.*

Proof. The proof is straightforward from *nfa_step*'s definition.By unfolding *nfa_step*, it is *step s x next_nfa*. Therefore, *nfa_step* is a function that uses *step* to apply the transition relation of the corresponding NFA without ϵ-transitions extracted from m by using *next_nfa*. Therefore, by Lemmas 44 and 43, this lemma is proved. □

References

[Hopcroft et al.(2006)Hopcroft, Motwani, and Ullman] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Automata theory, languages, and computation. *International Edition*, 24, 2006.

[Zhang and Qian(2013)] Jielan Zhang and Zhongsheng Qian. The equivalent conversion between regular grammar and finite automata. *Journal of Software Engineering and Applications*, 6(01):33, 2013.