

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55

Ley

Position1

Department1

Institution1

City1, State1, Saint-Petersburg

gkerfimf@gmail.com

56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110

56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110

56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110

• • • •

1

**Lemma 2.2.** *If  $L \neq \emptyset$  and  $L$  is regular then  $L$  is the union of regular language  $A_1, \dots, A_n$  where each  $A_i$  is accepted by a DFA with exactly one final state.*

**Theorem 2.3.** *If  $L_1$  is a context free language and  $L_2$  is a regular language then  $L_1 \cap L_2$  is context free.*

Sketch of the proof:

1. By lemma 2.1 we can assume that there is a context-free grammar  $G_{CNF}$  in Chomsky normal form, such that  $L(G_{CNF}) = L_1$
2. By lemma 2.2 we can assume that there is a set of regular languages  $\{A_1 \dots A_n\}$  where each  $A_i$  is recognized by a DFA with exactly one final state and  $L_2 = A_1 \cup \dots \cup A_n$
3. For each  $A_i$  we can explicitly define a (?) grammar of the intersection:  $L(G_{CNF}) \cap A_i$
4. Finally, we join them together with the (?) operation of union

### 3 CNF

One of important part of proof is the fact that any context-free language can be described with grammar in CNF.

We want to reuse existing proof of conversion of original context-free grammar to CNF.

We choose Smolka's version.

### 4 B-H in Coq

In this section we briefly describe motivation to use the chosen definitions, we also sketch all the fundamental parts of the proof. We also discuss advantages and disadvantages of usage side libraries/proof in ...?

Our goal is to provide step-by-step algorithm of constructing the CNF grammar of the intersection of two languages. Final formulation of the obtained theorem can be found in the last subsection(?).

All code are published on GitHub<sup>1</sup>.

#### 4.1 Smolka's code generalization

In this section we describe exact steps to ..., and discuss pros and cons of ... in this proof.

... of our proof, we need to consider nonterminals over the alphabet of triples. Therefore, it was(?) decided to simply add polymorphism over the target alphabet. Namely, let  $Tt$  and  $Vt$  be types with decidable relation of equality, then we can define the types of terminal and nonterminal over alphabets  $Tt$  and  $Vt$  respectively as follows:

```
Inductive ter : Type := | T : Tt -> ter.
Inductive var : Type := | V : Vt -> var.
```

Listing 1. TODO

<sup>1</sup>[https://github.com/YaccConstructor/YC\\_in\\_Coq](https://github.com/YaccConstructor/YC_in_Coq)

```
Lemma language_normalform G A u :
  Vs A el dom G ->
  u <> [] ->
  (language G A u <->
   language (normalize G) A u).
```

Listing 2. TODO

#### 4.2 Part ..: derivation and so on

Symbol is either a terminal or a nonterminal.

```
Inductive symbol : Type :=
| Ts : ter -> symbol
| Vs : var -> symbol.
```

Listing 3. TODO

Next we define word and phrase as lists of terminals and symbols respectively.

```
Definition word := list ter.
Definition phrase := list symbol.
```

Listing 4. TODO

TODO: add def of "terminal"

We have two different definitions because the notion of nonterminal doesn't make sense for DFA, but in order to construct derivation in grammar we need to use nonterminal in intermediate states.

Further we prove that if phrase consists only of terminals there exists save conversion between word and phrase.

We inheriting our definition of CFG from [] paper. Rule is pair of nonterminal and list of symbols. Grammar is a list of rules.

```
Inductive rule : Type :=
| R : var -> phrase -> rule.

Definition grammar := list rule.
```

Listing 5. TODO

An important step towards the definition of a language (?) governed (formed?)(?! ) by a grammar is the definition of derivability. Having  $der(G, A, p)$  — means that phrase  $p$  is derivable in grammar  $G$  starting from(?) nonterminal  $A$ .

Our proof requires grammar to be in CNF. We used statement that every grammar in convertible into CNF from Minka(?) work.

```

221 Inductive der (G : grammar)
222     (A : var) : phrase -> Prop :=
223   | vDer : der G A [Vs A]
224   | rDer l : (R A l) el G -> der G A l
225   | replN B u w v :
226     der G A (u ++ [Vs B] ++ w) ->
227     der G B v -> der G A (u ++ v ++ w).

```

Listing 6. TODO

### 4.3 General scheme of proof

General scheme of our proof is based on constructive proof presented by [?]. In the following subsections, the main steps of the proof will be presented. Overall, we will adhere to the following plan.

1. First we consider trivial cases, like DFA with no states or empty languages
2. Every CF language can be converted to CNF
3. Every DFA can be presented as an union of DFAs with single final state
4. Intersecting grammar in CNF with DFA with one final state
5. Proving that union of CF languages is CF language

### 4.4 Part one: trivial cases

Cases when one or both of the initial languages are empty we call trivial. Since in this case, the intersection language is also empty it is easy to construct the corresponding grammar.

We do the case analysis.

TODO: add some text

### 4.5 Part two: regular language and automata

In this section we describe definitions of DFA and DFA with exactly one final state, we also present function that converts any DFA to a set of DFA with one final state and lemma that states this split is well-defined(?).

A list of terminals we call word.

We assume that regular language by definition described by DFA. As the definition of an DFA, we have chosen a general definition, which does not impose any restrictions on the type of input symbols and the number of states. Thus, in our case, the DFA is a 5-tuple, (1) a state type, (2) a type of input symbols, (3) a start state, (4) a transition function, and (5) a list of final states.

Next we define a function that would evaluate in what state the automaton will end up if it starts from state  $s$  and receives a word  $w$ .

We say that the automaton accepts a word  $w$  being in state  $s$  if the function  $[final\_state\_sw]$  ends in one of the final states. Finally, we say that an automaton accepts a word  $w$ , if when(?) the DFA starts from the initial state, it ends in one of the final states.

```

276 Context {State T: Type}.
277 Record dfa: Type :=
278   mkDfa {
279     start: State;
280     final: list State;
281     next: State -> (@ter T) -> State;
282   }.

```

Listing 7. TODO

```

286 Fixpoint final_state
287   (next_d: dfa_rule)
288   (s: State)
289   (w: word): State :=
290   match w with
291   | nil => s
292   | h :: t => final_state next_d (next_d s h) t
293   end.

```

Listing 8. TODO

In order to define the DFA with exactly one final state, it is necessary to replace the list of final states by one final state in the definition of an(?) ordinary DFA. The definitions of "accepts" and "dfa\_language" vary slightly.

Alternative: In the proof we need a subset (subtype?) of all automata. Namely, automata with one finite state. We can define them as follows. We say that dfa is a single-final-state-automata, if and only if the predicate "is final state?" can be represented as "is equal to the state fin?"

```

307 Record s_dfa : Type :=
308   s_mkDfa {
309     s_start: State;
310     s_final: State;
311     s_next: State -> (@ter T) -> State;
312   }.

```

Listing 9. TODO

TODO?: add code

Similarly, we can define functions  $s\_accepts$  and  $s\_dfa\_language$  for sDFA. Since in this case, there is only one final state, to define function  $s\_accepts$  it is enough to check the state in which the automaton stopped with the finite state. The function  $s\_dfa\_language$  repeats the function  $dfa\_language$ , except that the function must now use  $s\_accepts$  instead of accepts.

Now it is easy to define a function that converts an ordinary DFA into a sequence (set?) of DFAs (?) with one final state.

Correctness of "split":

```

331 Fixpoint split_dfa_list
332   (st_d : State)
333   (next_d : dfa_rule)
334   (f_list : list State): list (s_dfa) :=
335   match f_list with
336   | nil => nil
337   | h :: t => (s_mkDfa st_d h next_d)
338               :: split_dfa_list st_d next_d t
339   end.
340
341 Definition split_dfa (d: dfa) :=
342   split_dfa_list (start d) (next d) (final d).
343

```

#### Listing 10. TODO

```

346 Lemma correct_split:
347   forall dfa w,
348     dfa_language dfa w <=>
349     exists sdfa,
350       In sdfa (split_dfa dfa) /\
351       s_dfa_language sdfa w.
352

```

#### Listing 11. TODO

### Theorem 4.1.

#### **Proof.**

TODO: add proof  
bla-bla-bla

### 4.6 Part ..: Chomsky induction

TODO: add some text

Naturally many statements about properties of language's words can be proved by induction over derivation structure. Unfortunately, grammar can derive phrase as an intermediate step, but DFA supposed to work only with words, so we can't simply apply induction over derivation structure. To tackle this problem we create custom induction-principle for grammars in CNF.

The main point is that if we have a grammar in CNF, we can always divide the word into two parts, each of which is derived only from one nonterminal. Note that if we naively take a step back, we can get nonterminal in the middle of the word. Such a situation will not make any sense for DFA.

With induction we always work with subtrees that describes some part of word. Here is a picture of subtree describing intuition behind Chomsky induction.

TODO: add picture

TODO: add Lemma derivability\_backward\_step.

More formally: Let  $G$  be a grammar in CNF. Consider arbitrary nonterminal  $N \in G$  and phrase which consists only on terminals  $w$ . If  $w$  is derivable from  $N$  and  $|w| \geq 2$ , then there exists nonterminals  $N_1, N_2$  and subphrases of  $w$  —

$w_1, w_2$  such that:  $N \rightarrow N_1 N_2 \in G$ ,  $der(N_1, w_1)$ ,  $der(N_2, w_2)$ ,  $|w_1| \geq 1$ ,  $|w_2| \geq 1$  and  $w_1 + w_2 = w$ .

#### **Proof.**

The next step is to prove the following statement:

Let  $G$  be a grammar in CNF. And  $P$  be a predicate on non-terminals and phrases (i.e.  $P : var \rightarrow phrase \rightarrow Prop$ ). Let us also assume that the following two hypotheses are satisfied: (1) for every terminal production (i.e. in the form  $N \rightarrow a$ ) of grammar  $G$ ,  $P(r, [Tsr])$  and (2) for every  $N, N_1, N_2 \in G$  and two phrases which consist only of terminals  $w_1, w_2$ , if  $P(N_1, w_1)$ ,  $P(N_2, w_2)$ ,  $der(G, N_1, w_1)$  and  $der(G, N_2, w_2)$  then  $P(N, w_1 + w_2)$ . Then for any nonterminal  $N$  and any phrase consisting only of terminals  $w$ , the fact that  $w$  is derivable from  $N$  implies  $P(N, w)$ .

Basically, this principle says that if some  $P$  holds for two basic situations, then  $P$  hold for any derivable word.

**Proof?**. There is a constant  $n$  such that  $|w| \leq n$ . We prove the statement by induction on  $n$ .

Base:  $n = 0$ ,

Induction step:

TODO: add some text

As one might notice, TODO

### 4.7 Part ..: intersection

Since bla-bla-bla, we can assume that we have (1) DFA with exactly one final state —  $dfa$  and (2) grammar in CNF —  $G$ .

Let  $G_{INT}$  be the grammar of intersection. In  $G_{INT}$  nonterminals presented as triples  $(from \times var \times to)$  where  $from$  and  $to$  are states of  $dfa$ , and  $var$  is a nonterminal of  $G$ .

#### 4.7.1 Function

Next we present adaptation of the algorithm given in [].

Since  $G$  is a grammar in CNF, it has only two type of productions: (1)  $N \rightarrow a$  and (2)  $N \rightarrow N_1 N_2$ , where  $N, N_1, N_2$  are nonterminals and  $a$  is a terminal.

For every production  $N \rightarrow N_1 N_2$  in  $G$  we generate a set of productions of the form  $(from, N, to) \rightarrow (from, N_1, m)(m, N_2, to)$  where:  $from, m, to$  — goes through all  $dfa$  states.

For every production of the form  $N \rightarrow a$  we add a set of productions  $(from, N, (dfa\_step(from, a))) \rightarrow a$  where:  $from$  — goes through all  $dfa$  states and  $dfa\_step(from, a)$  is the state in which the  $dfa$  appears after receiving terminal  $a$  in state  $from$ .

TODO: add some text

Next we join the functions above to get a generic function which works for both types of productions. Note that since the grammar is in CNF, (?) the third alternative is never called.

Note that at this point we do not have any manipulations with starting rules. Nevertheless(?), the hypothesis of the uniqueness of the final state of the DFA, will help us unambiguously introduce the starting nonterminal of the grammar of intersection.



```

441 Definition convert_nonterm_rule_2
442   (r r1 r2: _)
443   (state1 state2 : _) :=
444   map (fun s3 => R (V (s1, r, s3))
445         [Vs (V (s1, r1, s2));
446           Vs (V (s2, r2, s3))])
447   list_of_states.
448
449 Definition convert_nonterm_rule_1
450   (r r1 r2: _)
451   (s1 : _) :=
452   flat_map (convert_nonterm_rule_2 r r1 r2 s1)
453   list_of_states.
454
455 Definition convert_nonterm_rule (r r1 r2: _) :=
456   flat_map (convert_nonterm_rule_1 r r1 r2)
457   list_of_states.

```

#### Listing 12. TODO

```

461 Definition convert_terminal_rule
462   (next: _)
463   (r: _)
464   (t: _): list TripleRule :=
465   map (fun s1 => R (V (s1, r, next s1 t)) [Ts t])
466   list_of_states.

```

#### Listing 13. TODO

```

470 Definition convert_rule (next: _) (r: _ ) :=
471   match r with
472   | R r [Vs r1; Vs r2] =>
473     convert_nonterm_rule r r1 r2
474   | R r [Ts t] =>
475     convert_terminal_rule next r t
476   | _ => [] (* Never called *)
477   end.

```

```

479 Definition convert_rules
480   (rules: list rule) (next: _): list rule :=
481   flat_map (convert_rule next) rules.

```

```

483 (* Maps grammar and s_dfa to grammar over triples *)

```

```

484 Definition convert_grammar grammar s_dfa :=
485   convert_rules grammar (s_next s_dfa).

```

#### Listing 14. TODO

### 4.7.2 Correctness

TODO: add some text

In the interest of clarity of exposition, we skip some auxiliary lemmas, such as "we can get the initial grammar from the grammar of intersection by projecting the triples back to

terminals/nonterminals". Also note that the grammar after the conversion remains in CFN. Since the transformation of rules does not change the structure of the rules, but only replaces one(?!!) terminals and nonterminals with others.

Next we prove the two main lemmas. Namely, the derivability in the initial grammar and the  $s\_dfa$  implies the derivability in the grammar of intersection. And the other way around, the derivability in the grammar of intersection implies the derivability in the initial grammar and the  $s\_dfa$ .

Let  $G$  be a grammar in CNF. In order to use Chomsky Induction we also assume that syntactic analysis is possible.

**Theorem 4.2.** *Let  $s\_dfa$  be an arbitrary DFA, let  $r$  be a non-terminal of grammar  $G$ , let  $from$  and  $to$  be two states of the DFA. We also pick an arbitrary word  $w$ . If in grammar  $G$  it is possible to derive  $w$  out of  $r$  and starting from the state  $from$  when  $w$  is received, the  $s\_dfa$  ends up in state  $to$ , then word  $w$  is also derivable in grammar  $(convert\_rules\ G\ next)$  from the nonterminal  $(V\ (from, r, to))$ .*

**Proof.** TODO. In another case, it would be logical to use induction on the derivation structure in  $G$ . But as it was discussed earlier, this is not the case, otherwise we will get a phrase (list of terminals and nonterminals) instead of a word. Let's apply chomsky induction principle with  $P := funrphr => \forall(next : dfa\_rule)(fromto : DfaState), final\_state \rightarrow to \rightarrow der(convert\_rulesGnext)(V(from, r, to))phr$ . We will get the bla-bla, bla-bla, bla-bla-bla

Since a language is just a bla-bla-bla, we use the lemma above to prove bla-bla-bla

### 4.8 Part ..: union

After the previous step, we have a list of grammars of CF languages, in this section, we provide a function by which we construct a grammar of the union of languages.

For this, we need nonterminals from every language to be from different nonintersecting sets. To achieve this we add labels to nonterminals. Thus, each grammar of the union would have its own unique ID number, all nonterminals within one grammar will have the same ID which coincides with the ID of a grammar. In addition, it is necessary to introduce a new starting nonterminal of the union.

```

Inductive labeled_Vt : Type :=
| start : labeled_Vt
| lV : nat -> Vt -> labeled_Vt.

```

```

Definition label_var (label: nat)
  (v: @var Vt): @var
  labeled_Vt :=
V (lV label v).

```

#### Listing 15. TODO

Construction of new grammar is quite simple. The function that constructs the union grammar takes a list of grammars, then, it (1) splits the list into head  $[h]$  and tail  $[tl]$ , (2) labels  $[length\ tl]$  to  $h$ , (3) adds a new rule from the start nonterminal of the union to the start nonterminal of the grammar  $[h]$ , finally (4) the function is recursively called on the tail  $[tl]$  of the list.

**Definition** label\_grammar label grammar := ...

**Definition** label\_grammar\_and\_add\_start\_rule  
label  
grammar :=

let '(st, gr) := grammar in  
(R (V start) [Vs (V (lV label st))])  
:: label\_grammar label gr.

**Fixpoint** grammar\_union  
(grammars : seq (@var Vt \* (@grammar Tt Vt)))  
: @grammar  
Tt  
labeled\_Vt :=  
match grammars with  
| [] => []  
| (g::t) =>  
label\_grammar\_and\_add\_start\_rule  
(length t)  
g ++ (grammar\_union t)  
end.

#### Listing 16. TODO

#### 4.8.1 Equivalence proof

In this section, we prove that function *grammar\_union* constructs a correct grammar of union language indeed. Namely, we prove the following theorem.

**Theorem 4.3.** *Let grammars be a sequence of pairs of starting nonterminals and grammars. Then for any word  $w$ , the fact that  $w$  belongs to language of union is equivalent to the fact that there exists a grammar  $(st, gr) \in grammars$  such that  $w$  belongs to language generated by  $(st, gr)$ .*

**Proof of theorem 4.3.** Since the statement is formulated as an equivalence, we divide the proof into two parts:

1. If  $w$  belongs to the union language, then  $w$  belongs to one of the initial language.
2. If  $w$  belongs to one of the initial language, then  $w$  belongs to the union language.

The fact that  $(st, grammar) \in grammars$  implies that there exist  $gr1$  and  $gr2$  such that:  $gr1 ++ (st, grammar) :: gr2 = grammars$ .

Proof. This proved through induction over  $l$ . assume  $l = h :: t$ , then either word accepted by  $h$  or tail. If word accepted

**Variable** grammars: seq (var \* grammar).

**Theorem** correct\_union:  
forall word,  
language (grammar\_union grammars)  
(V (start Vt)) (to\_phrase word) <->  
exists s\_l,  
language (snd s\_l) (fst s\_l)  
(to\_phrase word) /\  
In s\_l grammars.

#### Listing 17. TODO

by  $h$  If word accepted by  $l$ . We just proving that adding one more language to union preserves word derivability. Which is equivalent to proving that adding new rules to grammar preserves word derivability

2. If we have derivation for some word in new grammar lanager we can provide derivate in for some language from union.

Proof. Here we converting derivability procedure for language union into derivability procedure of one of language. Then we proving that in derivation we can use rules from only one language at time. Finally we converting derivation by simple relabelling back all nonterminals.

#### 4.9 Part N: taking all parts together

TODO: add some text

**Theorem 4.4.** *For any two decidable types Terminal and Non-terminal for type of terminals and nonterminals correspondingly. If there exists bijection from Nonterminal to  $\mathbb{N}$  and syntactic analysis in the sense of definition TODO is possible, then for any DFA dfa which accepts Terminal and any grammar  $G$ , there exists the grammar of intersection  $L(DFA)$  and  $G$ .*

Proof.

## 5 Related Work

All results you use in your work. All relevant results in this field (excluded this work). Smolka, smb else [2–4].

Firsov in Agda: CYK, Chomsky Normal Form, etc.

Sertified parsers.

In HOL4.

Thmb else

## 6 Conclusion

Short resume of main part (main results formulation). We present mechanization of Bar-Hillel theorem on closure of contex-free languages under intersection with regular.

Other algorithms on regular and context-free languages intersection. One of direction of future reserch is mechanization of practical algorithms which are just implementation of

Bar-Hillel theorem. For example, context-free path querying algorithm, based on GLL [10] parsing algorithm [5]. Other problems on language intersection [8, 9].

Acknowledgments

The research was supported by the Russian Science Foundation grant No. 18-11-00100 from JetBrains Research.

References

[1] Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. 1961. On formal properties of simple phrase structure grammars. *Sprachtypologie und Universalienforschung* 14 (1961), 143–172.

[2] Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. 2013. A constructive theory of regular languages in Coq. In *International Conference on Certified Programs and Proofs*. Springer, 82–97.

[3] Christian Doczkal and Gert Smolka. 2017. Regular Language Representations in the Constructive Type Theory of Coq. (2017).

[4] Denis Firsov. 2016. Certification of Context-Free Grammar Algorithms. (2016).

[5] Semyon Grigorev and Anastasiya Ragozina. 2016. Context-Free Path Querying with Structural Representation of Result. *arXiv preprint arXiv:1612.08872* (2016).

[6] J. Hellings. 2014. Conjunctive context-free path queries. (2014).

[7] Jelle Hellings. 2015. Querying for Paths in Graphs using Context-Free Path Queries. *arXiv preprint arXiv:1502.02242* (2015).

[8] Mark-Jan Nederhof and Giorgio Satta. 2002. Parsing non-recursive context-free grammars. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 112–119.

[9] Mark-Jan Nederhof and Giorgio Satta. 2004. The language intersection problem for non-recursive context-free grammars. *Information and Computation* 192, 2 (2004), 172–184.

[10] Elizabeth Scott and Adrian Johnstone. 2010. GLL parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177–189.

A Appendix

Text of appendix ...