# $Pre^*$: The Predecessors of a Regular Language w.r.t. a Context-Free Grammar, with Applications

Tassilo Lemke and Tobias Nipkow

February 6, 2026

**Abstract**

Let $L$ be a language, $G$ a context-free grammar and let $pre^*(L)$ be the language of all predecessors (w.r.t. $G$) of words in $L$. The following fact has been rediscovered in the literature repeatedly: If $L$ is regular, so is $pre^*(L)$. Moreover, given an NFA $M$ for $L$, an NFA $M'$ for $pre^*(L)$ can be computed very elegantly from $M$. Starting from a suitable $M$, simple checks on $M'$ provide solutions to many elementary decision problems concerning $G$, such as the word-problem, emptiness problem, and more.

We formalize two algorithms to compute $pre^*(L)$ for a regular $L$ (using NFAs as representation). The first one is very simple and elegant and works for any CFG, while the second one is more efficient but is restricted to CFGs in in extended-CNF. All our algorithms are executable, allowing many elementary problems on context-free grammars to be solved automatically.

# Contents

# 1 Introduction

Given a regular language $L$ and a context-free grammar $G$, the language of predecessors of $L$ with respect to $G$, $pre^*(L)$, is also regular. This has been discovered independently by many authors [BO93, Büc59, Cau92]. We formalise the algorithm proposed by Book and Otto [BO93] which takes as input a non-deterministic finite automaton $M$, enriches it with new transitions, and yields a new automaton $M'$ such that $L(M') = pre^*(L(M))$.

This yields a unified framework for deciding many elementary properties of context-free grammars, as was first described by Esparza and Rossmanith [ER97].

These theories formalize $pre^*$, its applications to elementary CFG problems, and an improved algorithm for grammars in CNF by Bouajjani *et al.* [BEF+00].

The theories *Labeled_Transition_System* and *LTS_Automata* are auxiliary; the formalization proper starts with theory *Pre_Star*.

Closely related work:

- [SSST23] formalizes a version of $pre^*$ for pushdown systems instead of CFGs.
- [Lam09] formalizes $pre^*$ for dynamic pushdown networks, which are a generalization of pushdown systems.

# 2 Labeled Transition System

This theory could be unified with `AFP/Labeled_Transition_Systems`

**theory** `Labeled_Transition_System`
  **imports** `Main`
**begin**

Labeled Transition Systems are sets of triples of type `'s × 'a × 's`.

**type_synonym** `('s, 'l) lts = "('s × 'l × 's) set"`

The following lemma ensure that Isabelle can evaluate set comprehensions over triples.

**lemma** `Collect_triple_code[code_unfold]:`
  `"{(x,y,z) ∈ A. P x y z} = {p ∈ A. P (fst p) (fst (snd p)) (snd (snd p))}"`
  ⟨*proof*⟩

## 2.1 Step Relations

A step from a state `q` over a single symbol `c` is the set of all `q'`, such that `(q, c, q') ∈ T`:

**definition** *step_lts :: "('s, 'l) lts ⇒ 'l ⇒ 's ⇒ 's set"* **where**
*"step_lts T c s = (λ(q, c, q'). q') ` {(q, c', q') ∈ T. c = c' ∧ q = s}"*

A step of a single symbol *c* from a set of states *S* is the union of *step_lts* over *S*:

**definition** *Step_lts :: "('s, 'l) lts ⇒ 'l ⇒ 's set ⇒ 's set"* **where**
*"Step_lts T c S = ⋃(step_lts T c ` S)"*

Repeated steps of a word *w* consisting of multiple letters is achieved using a standard *fold*:

**definition** *Steps_lts :: "('s, 'l) lts ⇒ 'l list ⇒ 's set ⇒ 's set"* **where**
*"Steps_lts T w s = fold (Step_lts T) w s"*

Often, merely a single starting-state is of relevance:

**abbreviation** *steps_lts :: "('s, 'l) lts ⇒ 'l list ⇒ 's ⇒ 's set"* **where**
*"steps_lts T w s ≡ Steps_lts T w {s}"*

**lemmas** *steps_lts_defs = step_lts_def Step_lts_def Steps_lts_def*

We now prove some key properties of this step relation:

**lemma** *Step_union: "Step_lts T w ($S_1$ ∪ $S_2$) = Step_lts T w $S_1$ ∪ Step_lts T w $S_2$"*
⟨*proof*⟩

**lemma** *Steps_lts_mono: "$s_1$ ⊆ $s_2$ ⟹ Steps_lts T w $s_1$ ⊆ Steps_lts T w $s_2$"*
⟨*proof*⟩

**lemma** *Steps_lts_mono2:*
**assumes** *"$T_1$ ⊆ $T_2$"* **and** *"$q_1$ ⊆ $q_2$"*
**shows** *"Steps_lts $T_1$ w $q_1$ ⊆ Steps_lts $T_2$ w $q_2$"*
⟨*proof*⟩

**lemma** *steps_lts_mono: "$T_1$ ⊆ $T_2$ ⟹ steps_lts $T_1$ w q ⊆ steps_lts $T_2$ w q"*
⟨*proof*⟩

**lemma** *steps_lts_union: "q' ∈ steps_lts T w q ⟹ q' ∈ steps_lts (T ∪ T') w q"*
⟨*proof*⟩

**lemma** *Steps_lts_path:*
**assumes** *"$q_f$ ∈ Steps_lts T w s"*
**shows** *"∃$q_0$ ∈ s. $q_f$ ∈ steps_lts T w $q_0$"*
⟨*proof*⟩

**lemma** *Steps_lts_split:*

**assumes** *"$q_f$ ∈ Steps_lts T ($w_1$@$w_2$) $Q_0$"*
**shows** *"∃q'. q' ∈ Steps_lts T $w_1$ $Q_0$ ∧ $q_f$ ∈ steps_lts T $w_2$ q'"*
⟨*proof*⟩

**lemma** *Steps_lts_join:*
**assumes** *"q' ∈ Steps_lts T $w_1$ $Q_0$"* **and** *"$q_f$ ∈ steps_lts T $w_2$ q'"*
**shows** *"$q_f$ ∈ Steps_lts T ($w_1$@$w_2$) $Q_0$"*
⟨*proof*⟩

**lemma** *Steps_lts_split3:*
**assumes** *"$q_f$ ∈ Steps_lts T ($w_1$@$w_2$@$w_3$) $Q_0$"*
**shows** *"∃q' q''. q' ∈ Steps_lts T $w_1$ $Q_0$ ∧ q'' ∈ steps_lts T $w_2$ q' ∧ $q_f$ ∈ steps_lts T $w_3$ q''"*
⟨*proof*⟩

**lemma** *Steps_lts_join3:*
**assumes** *"q' ∈ steps_lts T $w_1$ $q_0$"* **and** *"q'' ∈ steps_lts T $w_2$ q'"* **and** *"$q_f$ ∈ steps_lts T $w_3$ q''"*
**shows** *"$q_f$ ∈ steps_lts T ($w_1$@$w_2$@$w_3$) $q_0$"*
⟨*proof*⟩

**lemma** *Steps_lts_noState: "Steps_lts T w {} = {}"*
⟨*proof*⟩

## 2.2 Reachable States

**definition** *reachable_from :: "('s, 'l) lts ⇒ 's ⇒ 's set"* **where**
*"reachable_from T q = {q'. ∃w. q' ∈ steps_lts T w q}"*

**lemma** *reachable_from_computable: "reachable_from T q ⊆ {q} ∪ (snd ` snd ` T)"*
⟨*proof*⟩

**lemma** *reachable_from_trans[trans]:*
**assumes** *"q1 ∈ reachable_from T q0"* **and** *"q2 ∈ reachable_from T q1"*
**shows** *"q2 ∈ reachable_from T q0"*
⟨*proof*⟩

**lemma** *reachable_add_trans:*
**assumes** *"∀(q1, _, q2) ∈ T'. ∃w. q2 ∈ steps_lts T w q1"*
**shows** *"reachable_from T q = reachable_from (T ∪ T') q"*
⟨*proof*⟩

**definition** *states_lts :: "('s,'a)lts ⇒ 's set"* **where**
*"states_lts T = (⋃(p,a,q)∈T. {p,q})"*

**lemma** *Step_states_lts: "states_lts T ⊆ Q ⟹ Q0 ⊆ Q ⟹ Step_lts T a Q0 ⊆ Q"*

⟨*proof*⟩

**lemma** *Steps_states_lts:* **assumes** *"states_lts T ⊆ Q"* **shows** *"Q0 ⊆ Q ⟹ Steps_lts T u Q0 ⊆ Q"*
⟨*proof*⟩

**corollary** *steps_states_lts:* *"⟦ states_lts T ⊆ Q; q ∈ Q ⟧ ⟹ steps_lts T u q ⊆ Q"*
⟨*proof*⟩

**lemma** *states_lts_Un:* *"states_lts (T ∪ T') = states_lts T ∪ states_lts T'"*
⟨*proof*⟩

### 2.3 Language

**abbreviation** *accepts_lts :: "('s, 'l) lts ⇒ 's ⇒ 's set ⇒ 'l list ⇒ bool"* **where**
*"accepts_lts T s F w ≡ (steps_lts T w s ∩ F ≠ {})"*

**abbreviation** *Lang_lts :: "('s, 'l) lts ⇒ 's ⇒ 's set ⇒ ('l list) set"* **where**
*"Lang_lts T S F ≡ { w. accepts_lts T S F w }"*

**end**

## 3 LTS-based Automata

**theory** *LTS_Automata*
**imports** *Labeled_Transition_System*
**begin**

An automaton *M* is a triple *(T, S, F)*, where *T* is the transition system, *S* is the start state and *F* are the final states. This is just a thin layer on top of *lts*. NB: *T* may be infinite (but we require to finiteness in crucial places).

**record** *('s, 't) auto =*
*lts :: "('s, 't) lts"*
*start :: 's*
*finals :: "'s set"*

The language *L(M)* of an automaton *M* is defined as the set of words that reach at least one final state from the start state:

**abbreviation** *accepts_auto :: "('s, 't) auto ⇒ 't list ⇒ bool"* **where**
*"accepts_auto M ≡ accepts_lts (lts M) (start M) (finals M)"*

**abbreviation** *Lang_auto :: "('s, 't) auto ⇒ 't list set"* **where**
*"Lang_auto M ≡ Lang_lts (lts M) (start M) (finals M)"*

## 3.1 Sequential Composition of Automata

We will later provide concrete example of automata accepting specific languages. While proving that an automaton accepts a certain language often is straightforward, proving that the automaton only accepts that language is a much more difficult task. The lemma below provides a powerful tool to make these proofs manageable. It shows that if two automata over disjoint state sets are connected via a single uni-directional bridge, every word that reaches from the first set of states to a state within the second set of state must, at some point, pass this bridge, and have a prefix within the first set of states and a suffix within the second set.

**lemma** *auto_merge:*
  **assumes** "$s_A \in A$" **and** "$f_A \in A$" **and** "$s_B \in B$" **and** "$f_B \in B$" **and** "$A \cap B = \{\}$"
    **and** *sideA:* "$\forall (q,c,q') \in T_A.\ q \in A \wedge q' \in A$"
    **and** *sideB:* "$\forall (q,c,q') \in T_B.\ q \in B \wedge q' \in B$"
    **and** "$f_B \in steps\_lts\ (T_A \cup \{(f_A, c, s_B)\} \cup T_B)\ w\ s_A$"
  **shows** "$\exists w_A\ w_B.\ w = w_A@[c]@w_B \wedge f_A \in steps\_lts\ T_A\ w_A\ s_A \wedge f_B \in steps\_lts\ T_B\ w_B\ s_B$"
⟨*proof*⟩

## 3.2 Concrete Automata

We now present three concrete automata that accept certain languages.

### 3.2.1 Universe over specific Alphabet

This automaton accepts exactly the words that only contains letters from a given alphabet $\Sigma$.

**definition** *loop_lts :: "'s ⇒ 'a set ⇒ ('s × 'a × 's) set"* **where**
  *"loop_lts q Σ = {q} × Σ × {q}"*

**lemma** *loop_lts_fin: "finite Σ ⟹ finite (loop_lts q Σ)"*
  ⟨*proof*⟩

**lemma** *loop_lts_correct1: "set w ⊆ Σ ⟹ steps_lts (loop_lts q Σ) w q = {q}"*
⟨*proof*⟩

**lemma** *loop_lts_correct2: "¬ set w ⊆ Σ ⟹ steps_lts (loop_lts q Σ) w q = {}"*
⟨*proof*⟩

**lemmas** *loop_lts_correct = loop_lts_correct1 loop_lts_correct2*

**definition** *auto_univ :: "'a set ⇒ (unit, 'a) auto"* **where**
  *"auto_univ Σ = (|*

```
    lts = loop_lts () Σ,
    start = (),
    finals = {()}
  ⦆"
```

**lemma** *auto_univ_lang[simp]: "Lang_auto (auto_univ Σ) = {w. set w ⊆ Σ}"*
⟨*proof*⟩

### 3.2.2 Fixed Character with Arbitrary Prefix/Suffix

This automaton accepts exactly those words that contain a specific letter *c* at some point, and whose prefix and suffix are contained within the alphabets Σ*p* and Σ*s*.

**definition** *pcs_lts :: "'a set ⇒ 'a ⇒ 'a set ⇒ (nat × 'a × nat) set"* **where**
*"pcs_lts Σp c Σs = loop_lts 0 Σp ∪ {(0, c, 1)} ∪ loop_lts 1 Σs"*

**lemma** *pcs_lts_fin: "finite Σp ⟹ finite Σs ⟹ finite (pcs_lts Σp c Σs)"*
⟨*proof*⟩

**lemma** *pcs_lts_correct1:*
*"(∃p s. w = p@[c]@s ∧ set p ⊆ Σp ∧ set s ⊆ Σs) ⟹ 1 ∈ steps_lts (pcs_lts Σp c Σs) w 0"*
⟨*proof*⟩

**lemma** *pcs_lts_correct2:*
**assumes** *"1 ∈ steps_lts (pcs_lts Σp c Σs) w 0"*
**shows** *"∃p s. w = p@[c]@s ∧ set p ⊆ Σp ∧ set s ⊆ Σs"*
⟨*proof*⟩

**lemmas** *pcs_lts_correct = pcs_lts_correct1 pcs_lts_correct2*

**definition** *cps_auto :: "'a ⇒ 'a set ⇒ (nat, 'a) auto"* **where**
```
  "cps_auto c Σ = ⦇
    lts = pcs_lts Σ c Σ,
    start = 0,
    finals = {1}
  ⦆"
```

**lemma** *cps_auto_lang: "Lang_auto (cps_auto c U) = { α@[c]@β | α β. set α ⊆ U ∧ set β ⊆ U }"*
⟨*proof*⟩

### 3.2.3 Singleton Language

Last but not least, the automaton accepting exactly a single word can be inductively defined.

**lemma** *steps_lts_empty_lts: "w ≠ [] ⟹ steps_lts {} w q₀ = {}"*
⟨*proof*⟩

**fun** *word_lts :: "'a list ⇒ (nat × 'a × nat) set"* **where**
*"word_lts (w#ws) = word_lts ws ∪ {(Suc (length ws), w, length ws)}"*
*|*
*"word_lts [] = {}"*

**lemma** *word_lts_domain:*
*"(q, c, q') ∈ word_lts ws ⟹ q ≤ length ws ∧ q' ≤ length ws"*
⟨*proof*⟩

**definition** *word_auto :: "'a list ⇒ (nat, 'a) auto"* **where**
*"word_auto ws = (| lts = word_lts ws, start = length ws, finals = {0} |)"*

**lemma** *word_lts_correct1:*
*"0 ∈ steps_lts (word_lts ws) ws (length ws)"*
⟨*proof*⟩

**lemma** *word_lts_correct2:*
*"0 ∈ steps_lts (word_lts ws) ws' (length ws) ⟹ ws = ws'"*
⟨*proof*⟩

**lemmas** *word_lts_correct = word_lts_correct1 word_lts_correct2*

**lemma** *word_auto_lang[simp]: "Lang_auto (word_auto w) = {w}"*
⟨*proof*⟩

**lemma** *word_auto_finite_lts: "finite (lts (word_auto w))"*
⟨*proof*⟩

**hide_const** (**open**) *lts start finals*
**term** *auto.start*

**end**

# 4 $Pre^*$

**theory** *Pre_Star*
**imports**
*Context_Free_Grammar.Context_Free_Grammar*
*LTS_Automata*
*"HOL-Library.While_Combinator"*

**begin**

This theory defines *pre*$^*$*(L)* (*pre_star* below) and verifies a simple saturation algorithm *pre_star_auto* that computes *pre*$^*$*(M)* given an NFA *M* and a finite set of context-free productions. Most of the work is on the level of finite LTS (via *pre_star_lts*).

A closely related formalization is *AFP/Pushdown_Systems* where *pre*$^*$ is computed for pushdown systems instead of CFGs.

**definition** *pre_star :: "('n,'t)Prods ⇒ ('n,'t) syms set ⇒ ('n,'t) syms set"* **where**
*"pre_star P L = {α. ∃β ∈ L. P ⊢ α ⇒* β}"*

## 4.1 Definition on LTS as Fixpoint

The algorithm works by repeatedly adding transitions to the LTS, such that at after every step, the LTS accepts the original language and its **direct** predecessors.

Since no new states are added, the number of transitions that can be added is bounded, which allow to both prove termination and the property of a fixpoint: At some point, adding another layer of direct predecessors no-longer changes anything, i.e. the LTS is saturated and *pre*$^*$ has been reached.

**definition** *pre_lts :: "('n,'t) Prods ⇒ 's set ⇒ ('s, ('n,'t) sym) lts ⇒ ('s, ('n,'t) sym) lts"*
**where**
*"pre_lts P Q T =*
*{ (q, Nt A, q') | q q' A. q ∈ Q ∧ (∃β. (A, β) ∈ P ∧ q' ∈ steps_lts T β q)}"*

**lemma** *pre_lts_code[code]: "pre_lts P Q T =*
*(⋃q ∈ Q. ⋃(A,β) ∈ P. ⋃q' ∈ steps_lts T β q. {(q, Nt A, q')})"*
⟨*proof*⟩

**definition** *pre_star_lts :: "('n, 't) Prods ⇒ 's set ⇒ ('s, ('n, 't) sym) lts ⇒ ('s, ('n, 't) sym) lts option"* **where**
*"pre_star_lts P Q = while_saturate (pre_lts P Q)"*

**lemma** *pre_star_lts_rule:*
**assumes** *"⋀T. H T ⟹ ¬ pre_lts P Q T ⊆ T ⟹ H (T ∪ pre_lts P Q T)"*
**and** *"pre_star_lts P Q T = Some T'"* **and** *"H T"*
**shows** *"H T'"*
⟨*proof*⟩

**lemma** *pre_star_lts_fp: "pre_star_lts P Q T = Some T' ⟹ pre_lts P Q T' ⊆ T'"*
⟨*proof*⟩

**lemma** *pre_star_lts_mono: "pre_star_lts P Q T = Some T' ⟹ T ⊆ T'"*

⟨*proof*⟩

## 4.2 Propagation of Reachability

No new states are added. Expressing this fact within the `auto` model is to show that the set of reachable states from any given start state remains unaltered.

**lemma** *pre_lts_reachable:*
*"reachable_from T q = reachable_from (T ∪ pre_lts P Q T) q"*
⟨*proof*⟩

**lemma** *pre_star_lts_reachable:*
**assumes** *"pre_star_lts P Q T = Some T'"*
**shows** *"reachable_from T q = reachable_from T' q"*
⟨*proof*⟩

**lemma** *states_pre_lts:* **assumes** *"states_lts T ⊆ Q"* **shows** *"states_lts (pre_lts P Q T) ⊆ Q"*
⟨*proof*⟩

## 4.3 Correctness

**lemma** *pre_lts_prod:*
**assumes** *"(A, β) ∈ P"* **and** *"q ∈ Q"* **and** *"q' ∈ Q"* **and** *"q' ∈ steps_lts T β q"*
**shows** *"q' ∈ steps_lts (T ∪ pre_lts P Q T) [Nt A] q"*
⟨*proof*⟩

**lemma** *pre_lts_pre:*
**assumes** *"P ⊢ $w_\alpha$ ⇒ $w_\beta$"* **and** *"reachable_from T q ⊆ Q"* **and** *"q' ∈ steps_lts T $w_\beta$ q"*
**shows** *"q' ∈ steps_lts (T ∪ pre_lts P Q T) $w_\alpha$ q"*
⟨*proof*⟩

**lemma** *pre_lts_fp:*
**assumes** *"P ⊢ $w_\alpha$ ⇒* $w_\beta$"* **and** *"reachable_from T q ⊆ Q"* **and** *"q' ∈ steps_lts T $w_\beta$ q"*
**and** *fp: "pre_lts P Q T ⊆ T"*
**shows** *"q' ∈ steps_lts T $w_\alpha$ q"*
⟨*proof*⟩

**lemma** *pre_lts_sub_aux:*
**assumes** *"q' ∈ steps_lts (T ∪ pre_lts P Q T) w q"*
**shows** *"∃w'. P ⊢ w ⇒* w' ∧ q' ∈ steps_lts T w' q"*
⟨*proof*⟩

**lemma** *pre_lts_sub:*
**assumes** *"∀w. (q' ∈ steps_lts T' w q) ⟶ (∃w'. P ⊢ w ⇒* w' ∧ q' ∈ steps_lts T w' q)"*

**and** *"q' ∈ steps_lts (T' ∪ pre_lts P Q T') w q"*
**shows** *"∃w'. P ⊢ w ⇒* w' ∧ q' ∈ steps_lts T w' q"*
⟨*proof*⟩

**lemma** *pre_star_lts_sub:*
**assumes** *"pre_star_lts P Q T = Some T'"*
**shows** *"(q' ∈ steps_lts T' w q) ⟹ (∃w'. P ⊢ w ⇒* w' ∧ q' ∈ steps_lts T w' q)"*
⟨*proof*⟩

**lemma** *pre_star_lts_correct:*
**assumes** *"reachable_from T $q_0$ ⊆ Q"* **and** *"pre_star_lts P Q T = Some T'"*
**shows** *"Lang_lts T' $q_0$ F = pre_star P (Lang_lts T $q_0$ F)"*
⟨*proof*⟩

## 4.4 Termination

**lemma** *pre_star_lts_terminates:*
**fixes** *P :: "('n, 't) Prods"* **and** *Q :: "'s set"* **and** *$T_0$ :: "('s, ('n, 't) sym) lts"*
**assumes** *"finite P"* **and** *"finite Q"* **and** *"finite $T_0$"* **and** *"states_lts $T_0$ ⊆ Q"*
**shows** *"∃T. pre_star_lts P Q $T_0$ = Some T"*
⟨*proof*⟩

## 4.5 The Automaton Level

**definition** *pre_star_auto :: "('n, 't) Prods ⇒ ('s, ('n, 't) sym) auto ⇒ ('s, ('n, 't) sym) auto"* **where**
*"pre_star_auto P M = (*
*let Q = {auto.start M} ∪ states_lts (auto.lts M) in*
*case pre_star_lts P Q (auto.lts M) of*
*Some T' ⇒ M (| auto.lts := T' |)*
*)"*

**lemma** *pre_star_auto_correct:*
**assumes** *"finite P"* **and** *"finite (auto.lts M)"*
**shows** *"Lang_auto (pre_star_auto P M) = pre_star P (Lang_auto M)"*
⟨*proof*⟩

**lemma** *pre_star_lts_refl:*
**assumes** *"pre_star_lts P Q T = Some T'"* **and** *"(A, []) ∈ P"* **and** *"q ∈ Q"*
**shows** *"(q, Nt A, q) ∈ T'"*
⟨*proof*⟩

**lemma** *pre_star_lts_singleton:*
**assumes** *"pre_star_lts P Q T = Some T'"* **and** *"(A, [B]) ∈ P"*
**and** *"(q, B, q') ∈ T'"* **and** *"q ∈ Q"* **and** *"q' ∈ Q"*
**shows** *"(q, Nt A, q') ∈ T'"*

⟨*proof*⟩

**lemma** *pre_star_lts_impl:*
  **assumes** *"pre_star_lts P Q T = Some T'"* **and** *"(A, [B, C]) ∈ P"*
    **and** *"(q, B, q') ∈ T'"* **and** *"(q', C, q'') ∈ T'"*
    **and** *"q ∈ Q"* **and** *"q' ∈ Q"* **and** *"q'' ∈ Q"*
  **shows** *"(q, Nt A, q'') ∈ T'"*
⟨*proof*⟩
**unused_thms**
**end**

## 4.6 $Pre^*$ Example

The algorithm is executable. This theory shows a quick example.

**theory** *Pre_Star_Example*
  **imports** *Pre_Star*
**begin**

Consider the following grammar, with *V = {A,B}* and Σ *= {a,b}*:

**datatype** *n = A | B*
**datatype** *t = a | b*

**definition** *"P ≡ {*
  *— A → a | BB*
  *(A, [Tm a]),*
  *(A, [Nt B, Nt B]),*

  *— B → AB | b*
  *(B, [Nt A, Nt B]),*
  *(B, [Tm b])*
*}"*

The following NFA accepts the regular language, whose predecessors we want to find:

**definition** *M :: "(nat, (n, t) sym) auto"* **where** *"M ≡ ⦇*
  *auto.lts = {*
    *(0, Tm a, 1),*
    *(1, Tm b, 2),*
    *(2, Tm a, 1)*
  *},*
  *start = 0 :: nat,*
  *finals = {0, 1, 2}*
*⦈"*

**lemma** *"pre_star_auto P M =*
  *⦇auto.lts =*
    *{(2, Tm a, 1), (1, Tm b, 2), (0, Tm a, 1), (0, Nt A, 1), (0, Nt A,*
*2), (0, Nt B, 2), (0, Nt A, 1),*

```
     (1, Nt A, 2), (1, Nt B, 2), (2, Nt A, 1), (2, Nt A, 2), (2, Nt B,
2), (2, Nt A, 1), (1, Nt A, 2),
     (1, Nt B, 2)},
   start = 0, finals = {0, 1, 2}⟩"
```
⟨*proof*⟩

**end**

# 5 Application to Elementary CFG Problems

**theory** *Applications*
**imports** *Pre_Star*
**begin**

This theory turns *pre_star_auto* into executable decision procedures for different CFG problems. The methos: *pre_star_auto* is applied to different suitable automata/languages. This happens behind the scenes via code equations.

These lemmas link *pre_star* to different properties of context-free grammars:

**lemma** *pre_star_term:*
```
  "x ∈ pre_star P L ⟷ (∃w. w ∈ L ∧ P ⊢ x ⇒* w)"
```
⟨*proof*⟩

**lemma** *pre_star_word:*
```
  "[Nt S] ∈ pre_star P (map Tm ‘ L) ⟷ (∃w. w ∈ L ∧ w ∈ Lang P S)"
```
⟨*proof*⟩

**lemma** *pre_star_lang:*
```
  "Lang P S ∩ L = {} ⟷ [(Nt S)] ∉ pre_star P (map Tm ‘ L)"
```
⟨*proof*⟩

## 5.1 Derivability

A decision procedure for derivability can be constructed.

**definition** *is_derivable :: "('n, 't) Prods ⇒ ('n, 't) syms ⇒ ('n, 't) syms ⇒ bool"* **where**
*[simp]: "is_derivable P α β = (P ⊢ α ⇒* β)"*

**declare** *is_derivable_def[symmetric, code_unfold]*

**theorem** *pre_star_derivability:*
  **shows** *"P ⊢ α ⇒* β ⟷ α ∈ pre_star P {β}"*
⟨*proof*⟩

**lemma** *pre_star_derivability_code[code]:*
  **fixes** *P :: "('n, 't) prods"*

**shows** *"is_derivable (set P) α β = (α ∈ Lang_auto (pre_star_auto (set P) (word_auto β)))"*
⟨*proof*⟩

## 5.2 Membership Problem

**lemma** *pre_star_membership[code_unfold]: "(w ∈ Lang P S) = (P ⊢ [Nt S] ⇒* map Tm w)"*
⟨*proof*⟩

## 5.3 Nullable Variables

**definition** *is_nullable :: "('n, 't) Prods ⇒ 'n ⇒ bool"* **where**
*"is_nullable P X = (P ⊢ [Nt X] ⇒* [])"*

— Directly follows from derivability:
**lemma** *pre_star_nullable[code]: "is_nullable P X = (P ⊢ [Nt X] ⇒* [])"*
⟨*proof*⟩

## 5.4 Emptiness Problem

**definition** *is_empty :: "('n, 't) Prods ⇒ 'n ⇒ bool"* **where**
*[simp]: "is_empty P S = (Lang P S = {})"*

**lemma** *cfg_derives_Syms:*
**assumes** *"P ⊢ α ⇒* β"* **and** *"set α ⊆ Syms P"*
**shows** *"set β ⊆ Syms P"*
⟨*proof*⟩

**lemma** *cfg_Lang_univ: "P ⊢ [Nt X] ⇒* map Tm β ⟹ set β ⊆ Tms P"*
⟨*proof*⟩

**definition** *pre_star_emptiness_auto :: "('n, 't) Prods ⇒ (unit, ('n, 't) sym) auto"* **where**
*"pre_star_emptiness_auto P =*
*(let T = Tm ‘ ⋃((λA. case A of Nt X ⇒ {} | Tm x ⇒ {x}) ‘ ⋃(set ‘ snd ‘ P)) :: ('n, 't) sym set in*
*(| auto.lts = {()} × T × {()}, start = (), finals = {()} |))"*

**theorem** *pre_star_emptiness:*
**fixes** *P :: "('n, 't) Prods"*
**shows** *"Lang P S = {} ⟷ [(Nt S)] ∉ pre_star P {w. set w ⊆ Tm ‘ Tms P}"*
⟨*proof*⟩

**lemma** *pre_star_emptiness_code[code]:*
**fixes** *P :: "('n, 't) prods"*
**shows** *"is_empty (set P) S = ([Nt S] ∉ Lang_auto (pre_star_auto (set P) (auto_univ (Tm ‘ Tms (set P)))))"*
⟨*proof*⟩

## 5.5 Useless Variables

**definition** *is_reachable_from :: "('n, 't) Prods ⇒ 'n ⇒ 'n ⇒ bool"*
*("(2_ ⊢/ (_/ ⇒^? / _))" [50, 0, 50] 50)* **where**
*"(P ⊢ X ⇒^? Y) = (∃α β. P ⊢ [Nt X] ⇒* (α@[Nt Y]@β))"*

— *X ∈ V* is useful, iff *V* can be reached from *S* and it is productive:
**definition** *is_useful :: "('n, 't) Prods ⇒ 'n ⇒ 'n ⇒ bool"* **where**
*"is_useful P S X = (P ⊢ S ⇒^? X ∧ Lang P X ≠ {})"*

**definition** *pre_star_reachable_auto :: "('n, 't) Prods ⇒ 'n ⇒ (nat, ('n, 't) sym) auto"* **where**
*"pre_star_reachable_auto P X = (*
*let T = ⋃(set ‘ snd ‘ P) in*
*(| auto.lts = ({0} × T × {0}) ∪ ({1} × T × {1}) ∪ {(0, Nt X, 1)}, start = 0, finals = {1} |)*
*)"*

**theorem** *pre_star_reachable:*
**fixes** *P :: "('n, 't) Prods"*
**shows** *"(P ⊢ S ⇒^? X) ⟷ [Nt S] ∈ pre_star P { α@[Nt X]@β | α β. set α ⊆ Syms P ∧ set β ⊆ Syms P }"*
⟨*proof*⟩

**lemma** *pre_star_reachable_code[code]:*
**fixes** *P :: "('n, 't) prods"*
**shows** *"(set P ⊢ S ⇒^? X) = ([Nt S] ∈ Lang_auto (pre_star_auto (set P) (cps_auto (Nt X) (Syms (set P)))))"*
⟨*proof*⟩

## 5.6 Disjointness and Subset Problem

**theorem** *pre_star_disjointness: "Lang P S ∩ L = {} ⟷ [(Nt S)] ∉ pre_star P (map Tm ‘ L)"*
⟨*proof*⟩

**theorem** *pre_star_subset: "Lang P S ⊆ L ⟷ [(Nt S)] ∉ pre_star P (map Tm ‘ (-L))"*
⟨*proof*⟩

**end**

## 5.7 Examples

**theory** *Applications_Example*
**imports** *Applications*
**begin**

Consider the following grammar, with *V = {A,B,C,D}* and Σ *= {a,b,c,d}*:

**datatype** *n = A | B | C | D*

```
datatype t = a | b | c | d

definition P :: "(n, t) Prods" where "P ≡ {
  — A → a | BB | C
  (A, [Tm a]),
  (A, [Nt B, Nt B]),
  (A, [Nt C]),

  — B → AB | b
  (B, [Nt A, Nt B]),
  (B, [Tm b]),

  — C → c | ε
  (C, [Tm c]),
  (C, []),

  — D → d
  (D, [Tm d])
}"
```

Checking whether a symbol is nullable is straight-forward:

```
value "is_nullable P A"
— True

value "is_nullable P B"
— False

value "is_nullable P C"
— True

value "is_nullable P D"
— False
```

Instead of using `value`, it can also be proven by `eval` in theorems:

```
lemma "is_nullable P A" ⟨proof⟩

lemma "¬ is_nullable P B" ⟨proof⟩

lemma "is_nullable P C" ⟨proof⟩

lemma "¬ is_nullable P D" ⟨proof⟩
```

Similarily, derivability can also be checked and proven as simple:

```
lemma "P ⊢ [Nt A] ⇒* [Nt A, Nt B, Nt B]"
  ⟨proof⟩
lemma "¬ P ⊢ [Nt A] ⇒* [Nt A, Nt B]"
  ⟨proof⟩
```

Following derivability, the membership problem is straight-forward:

**lemma** *"[a] ∈ Lang P A"*
⟨*proof*⟩
**lemma** *"[b] ∉ Lang P A"*
⟨*proof*⟩
**lemma** *"[b,b] ∈ Lang P A"*
⟨*proof*⟩

To check if the accepted language is empty, one first needs to unfold *is_empty ?P ?S = (Lang ?P ?S = {})*, from which automatic evaluation is again possible:

**lemma** *"¬ Lang P A = {}"*
⟨*proof*⟩

Similar to derivability, reachability (i.e., derivability with an arbitrary prefix and suffix), can also be automated:

**lemma** *"P ⊢ A ⇒? B"*
⟨*proof*⟩

**lemma** *"P ⊢ B ⇒? A"*
⟨*proof*⟩

**lemma** *"P ⊢ A ⇒? C"*
⟨*proof*⟩

**lemma** *"P ⊢ B ⇒? C"*
⟨*proof*⟩

**lemma** *"¬ P ⊢ C ⇒? A"*
⟨*proof*⟩

**lemma** *"¬ P ⊢ C ⇒? A"*
⟨*proof*⟩

**end**

# 6 Finiteness of Context-Free Languages

**theory** *Finiteness*
**imports** *Applications*
**begin**

Another interesting application, particularly for context-free grammars in chomsky normal-form (CNF), is the detection of "cyclic" non-terminals.

Particularly, if all non-terminals are reachable (can be reached from the starting symbol) and productive (i.e., a terminal word can be derived from each symbol), the following holds:

*L(C)* = ∞ ⟷ ∃*X* *α* *β*. *X* ⇒* *αXβ* ∧ a*β* ≠ *ε*

Since we have a decision-procedure for derivability, we can work towards also automating this process. However, to keep proofs simple, this theory only focuses on grammars in CNF, meaning a conversion is required a priori.

## 6.1 Preliminaries and Assumptions

**locale** *CFG =*
  **fixes** *P :: "('n, 't) Prods"* **and** *S :: 'n*
  **assumes** *cnf: "⋀p. p ∈ P ⟹ (∃A a. p = (A, [Tm a]) ∨ (∃A B C. p = (A, [Nt B, Nt C])))"*
**begin** — begin-context CFG

**definition** *is_useful_all :: "bool"* **where**
  *"is_useful_all ≡ (∀X::'n. is_useful P S X)"*

**definition** *is_non_nullable_all :: "bool"* **where**
  *"is_non_nullable_all ≡ (∀X::'n. ¬ is_nullable P X)"*

**lemma** *derives_concat:*
  **assumes** *"P ⊢ X₁ ⇒* w₁"* **and** *"P ⊢ X₂ ⇒* w₂"*
  **shows** *"P ⊢ (X₁@X₂) ⇒* (w₁@w₂)"*
  ⟨*proof*⟩

**lemma** *derives_split:*
  **assumes** *"P ⊢ X ⇒* w"*
  **shows** *"∃X₁ X₂ w₁ w₂. X = X₁@X₂ ∧ w = w₁@w₂ ∧ P ⊢ X₁ ⇒* w₁ ∧ P ⊢ X₂ ⇒* w₂"*
  ⟨*proof*⟩

**lemma** *derives_step:*
  **assumes** *"P ⊢ X ⇒* (α@w₁@β)"* **and** *"P ⊢ w₁ ⇒* w₂"*
  **shows** *"P ⊢ X ⇒* (α@w₂@β)"*
⟨*proof*⟩

**lemma** *is_useful_all_derive:*
  **assumes** *"is_useful_all"*
  **shows** *"∃w. P ⊢ xs ⇒* map Tm w"*
⟨*proof*⟩

**lemma** *is_non_nullable_all_derive:*
  **assumes** *"is_non_nullable_all"* **and** *"P ⊢ xs ⇒* w"*
  **shows** *"xs = [] ⟷ w = []"*
⟨*proof*⟩

## 6.2 Criterion of Finiteness

Finally, we introduce the definition *is_infinite*, which instead of making use of the language set, uses the criterion introduced above.

**definition** *is_reachable_step :: "'n ⇒ 'n ⇒ bool"* **(infix** *"→^?" 80)* **where**
*"(X →^? Y) ≡ (∃α β. P ⊢ [Nt X] ⇒* (α@[Nt Y]@β) ∧ α@β ≠ [])"*

**definition** *is_infinite :: "bool"* **where**
*"is_infinite ≡ (∃X. X →^? X)"*

**fun** *is_infinite_derives :: "'n ⇒ ('n, 't) sym list ⇒ ('n, 't) sym list ⇒ nat ⇒ ('n, 't) sym list"* **where**
*"is_infinite_derives X α β (Suc n) = α@(is_infinite_derives X α β n)@β"*
*|*
*"is_infinite_derives X α β 0 = [Nt X]"*

**fun** *is_infinite_words :: "'t list ⇒ 't list ⇒ 't list ⇒ nat ⇒ 't list"* **where**
*"is_infinite_words $w_X$ $w_\alpha$ $w_\beta$ (Suc n) = $w_\alpha$@(is_infinite_words $w_X$ $w_\alpha$ $w_\beta$ n)@$w_\beta$" |*
*"is_infinite_words $w_X$ $w_\alpha$ $w_\beta$ 0 = $w_X$"*

**definition** *reachable_rel :: "('n × 'n) set"* **where**
*"reachable_rel ≡ {($X_2$, $X_1$). ∃α β. ($X_1$, α@[Nt $X_2$]@β) ∈ P}"*

**lemma** *cnf_implies_pumping:*
**assumes** *"(Y, α@[Nt X]@β) ∈ P"*
**shows** *"Y →^? X"*
⟨*proof*⟩

**lemma** *reachable_rel_tran: "(X, Y) ∈ reachable_rel^+ ⟹ Y →^? X"*
⟨*proof*⟩

**lemma** *reachable_rel_wf:*
**assumes** *"finite P"*
**and** *cnf: "⋀p. p ∈ P ⟹ (∃A a. p = (A, [Tm a]) ∨ (∃A B C. p = (A, [Nt B, Nt C])))"*
**and** *loopfree: "⋀X. ¬ X →^? X"*
**shows** *"wf reachable_rel"*
⟨*proof*⟩

**lemma** *is_infinite_implies_finite:*
**assumes** *"finite P"*
**and** *loopfree: "⋀X. ¬ X →^? X"*
**shows** *"finite {w. P ⊢ [Nt X] ⇒* w}"*
⟨*proof*⟩

**theorem** *is_infinite_correct:*
**assumes** *"is_useful_all"* **and** *"is_non_nullable_all"* **and** *"finite P"*
**shows** *"¬ finite (Lang P S) ⟷ is_infinite"*
⟨*proof*⟩
**no_notation** *is_reachable_step* **(infix** *"→^?" 80)*

### 6.3 Finiteness Problem

**lemma** *is_infinite_check:*
*"is_infinite ⟷ (∃X. [Nt X] ∈ pre_star P { α@[Nt X]@β | α β. α@β ≠ [] })"*
⟨*proof*⟩

**theorem** *is_infinite_by_prestar:*
**assumes** *"is_useful_all"* **and** *"is_non_nullable_all"* **and** *"finite P"*
**shows** *"finite (Lang P S) ⟷ (∀X. [Nt X] ∉ pre_star P { α@[Nt X]@β | α β. α@β ≠ [] })"*
⟨*proof*⟩

**end** — end-context CFG

**end**

## 7 $Pre^*$ Optimized for Grammars in CNF

**theory** *Pre_Star_CNF*
**imports** *Pre_Star*
**begin**

Bouajjani et al. [BEF+00] have proposed in an improved algorithm for grammars in extended Chomsky Normal Form. This theory proves core properties (correctness and termination) of the algorithm.

### 7.1 Preliminaries

Extended Chomsky Normal Form:

**definition** *CNF1 :: "('n, 't) Prods ⇒ bool"* **where**
*"CNF1 P ≡ (∀(A, β) ∈ P.*
— 1. *A → ε*
*(β = []) ∨*
— 2. *A → a*
*(∃a. β = [Tm a]) ∨*
— 3. *A → B*
*(∃B. β = [Nt B]) ∨*
— 4. *A → BC*
*(∃B C. β = [Nt B, Nt C])*
*)"*

**type_synonym** *('s, 'n, 't) tran = "'s × ('n, 't) sym × 's"* — single transition
**type_synonym** *('s, 'n, 't) trans = "('s, 'n, 't) tran set"* — set of auto.trans
**type_synonym** *('s, 'n, 't) directT = "('s, 'n, 't) tran ⇒ ('s, 'n, 't) trans"*

**type_synonym** *('s, 'n, 't) implT = "('s, 'n, 't) tran ⇒ (('s, 'n, 't) tran × ('s, 'n, 't) tran) set"*

**record** *('s, 'n, 't) alg_state =*
*rel :: "('s, 'n, 't) trans"*
*trans :: "('s, 'n, 't) trans"*
*direct :: "('s, 'n, 't) directT"*
*impl :: "('s, 'n, 't) implT"*

## 7.2 Procedure

**definition** *alg_state_new :: "('n, 't) Prods ⇒ 's set ⇒ ('s, 'n, 't) trans ⇒ ('s, 'n, 't) alg_state"* **where**
*"alg_state_new P Q T ≡ ⦇*
*rel = {},*
*trans = T*
*∪ { (q, Nt A, q) | q A. (A, []) ∈ P ∧ q ∈ Q }*
*∪ { (q, Nt A, q') | q q' A. ∃a. (A, [Tm a]) ∈ P ∧ (q, Tm a, q') ∈ T ∧ q ∈ Q ∧ q' ∈ Q },*
*direct = (λ(q, X, q'). case X of*
*Nt B ⇒ { (q, Nt A, q') | A. (A, [Nt B]) ∈ P ∧ q ∈ Q ∧ q' ∈ Q } |*
*Tm b ⇒ {}*
*),*
*impl = (λ(q, X, q'). case X of*
*Nt B ⇒ { ((q', Nt C, q''), (q, Nt A, q'')) | q'' A C. (A, [Nt B, Nt C]) ∈ P ∧ q ∈ Q ∧ q' ∈ Q ∧ q'' ∈ Q } |*
*Tm b ⇒ {}*
*)*
*⦈"*

**definition** *alg_inner_pre :: "('s, 'n, 't) alg_state ⇒ ('s, 'n, 't) tran ⇒ ('s, 'n, 't) alg_state"* **where**
*"alg_inner_pre S t ≡ S ⦇*
— *t* is added to *rel*:
*rel := (rel S) ∪ {t},*
— *t* is removed, and *direct(t)* is added to *trans*:
*trans := ((trans S) - {t}) ∪ direct S t,*
— *direct(t)* is cleared:
*direct := (direct S) (t := {})*
*⦈"*

**definition** *alg_inner_post :: "('s, 'n, 't) alg_state ⇒ ('s, 'n, 't) tran ⇒ ('s, 'n, 't) alg_state"* **where**
*"alg_inner_post S t ≡ (*
*let i = impl S t in*
*S ⦇*
— If *(t', t'') ∈ impl(t)* and *t' ∈ rel*, then *t'' ∈ trans*:
*trans := (trans S) ∪*

```
    snd ` { (t', t'') ∈ i. t' ∈ rel S },
    — If (t', t'') ∈ impl(t) and t' ∉ rel, then t'' ∈ direct(t'):
    direct := (λt'. direct S t' ∪
    snd ` { (t'2, t'') ∈ i. t' = t'2 ∧ t' ∉ rel S }
    ),
    — Inner while-loop removes everything from impl(t):
    impl := (impl S) (t := {})
  ⦇
)"
```

**definition** *alg_outer_step :: "('s, 'n, 't) alg_state ⇒ ('s, 'n, 't) tran ⇒ ('s, 'n, 't) alg_state"* **where**
*"alg_outer_step S t ≡ alg_inner_post (alg_inner_pre S t) t"*

**abbreviation** *"alg_outer_step_lts S t ≡ rel S ∪ {t}"*
**abbreviation** *"alg_outer_step_trans S t ≡ (trans S) - {t} ∪ direct S t ∪ snd ` { (t', t'') ∈ impl S t. t' ∈ rel S ∪ {t} }"*
**abbreviation** *"alg_outer_step_trans' S t ≡ (trans S) - {t} ∪ direct S t ∪ {t''. ∃t'. (t', t'') ∈ impl S t ∧ t' ∈ rel S ∪ {t} }"*
**abbreviation** *"alg_outer_step_direct S t ≡ (λt'. ((direct S) (t := {})) t' ∪ snd ` { (t'2, t'') ∈ impl S t. t' = t'2 ∧ t' ∉ (rel S) ∪ {t} })"*
**abbreviation** *"alg_outer_step_direct' S t ≡ (λt'. ((direct S) (t := {})) t' ∪ {t''. (t', t'') ∈ impl S t ∧ t' ∉ (rel S) ∪ {t} })"*
**abbreviation** *"alg_outer_step_impl S t ≡ (impl S) (t := {})"*

**lemma** *alg_outer_step_trans_eq[simp]:*
*"alg_outer_step_trans S t = alg_outer_step_trans' S t"*
⟨*proof*⟩

**lemma** *alg_outer_step_direct_eq[simp]:*
*"alg_outer_step_direct S t = alg_outer_step_direct' S t"*
⟨*proof*⟩

**lemma** *alg_outer_step_simps[simp]:*
**shows** *"rel (alg_outer_step S t) = alg_outer_step_lts S t"*
**and** *"trans (alg_outer_step S t) = alg_outer_step_trans S t"*
**and** *"direct (alg_outer_step S t) = alg_outer_step_direct S t"*
**and** *"impl (alg_outer_step S t) = alg_outer_step_impl S t"*
⟨*proof*⟩

**definition** *alg_outer :: "('s, 'n, 't) alg_state ⇒ ('s, 'n, 't) alg_state option"* **where**
*"alg_outer ≡ while_option (λS. trans S ≠ {}) (λS. alg_outer_step S (SOME x. x ∈ trans S))"*

**lemma** *alg_outer_rule:*
**assumes** *"⋀S x. P S ⟹ x ∈ trans S ⟹ P (alg_outer_step S x)"*
**and** *"alg_outer S = Some S'"*
**shows** *"P S ⟹ P S'"*

⟨*proof*⟩

## 7.3 Correctness

### 7.3.1 Subset

**definition** *pre_star_alg_sub_inv :: "('s, 'n, 't) trans ⇒ ('s, 'n, 't) alg_state ⇒ bool"* **where**
  *"pre_star_alg_sub_inv T' S ≡ (*
    *(trans S) ⊆ T' ∧ (rel S) ⊆ T' ∧*
    *(∀t' ∈ T'. ∀t ∈ direct S t'. t ∈ T') ∧*
    *(∀t ∈ T'. ∀(t', t'') ∈ impl S t. t' ∈ T' ⟶ t'' ∈ T')*
  *)"*

**lemma** *alg_state_new_inv:*
  **assumes** *"pre_star_lts P Q T = Some T'"*
  **shows** *"pre_star_alg_sub_inv T' (alg_state_new P Q T)"*
⟨*proof*⟩

**lemma** *alg_outer_step_inv:*
  **assumes** *"pre_star_lts P Q T = Some T'"* **and** *"t ∈ trans S"* **and** *"pre_star_alg_sub_inv T' S"*
  **shows** *"pre_star_alg_sub_inv T' (alg_outer_step S t)"*
⟨*proof*⟩

**lemma** *alg_outer_inv:*
  **assumes** *"pre_star_lts P Q T = Some T'"* **and** *"pre_star_alg_sub_inv T' S"*
    **and** *"alg_outer S = Some S'"*
  **shows** *"pre_star_alg_sub_inv T' S'"*
⟨*proof*⟩

**lemma** *pre_star_alg_sub:*
  **fixes** *P* **and** *T*
  **assumes** *"alg_outer (alg_state_new P Q T) = Some S'"* **and** *"pre_star_lts P Q T = Some T'"*
  **shows** *"rel S' ⊆ T'"*
⟨*proof*⟩

### 7.3.2 Super-Set

**lemma** *alg_outer_fixpoint: "alg_outer S = Some S' ⟹ alg_outer S' = Some S'"*
  ⟨*proof*⟩

**lemma** *pre_star_alg_trans_empty: "alg_outer S = Some S' ⟹ trans S' = {}"*
  ⟨*proof*⟩

**lemma** *alg_outer_step_direct: "t ≠ t' ⟹ direct S t' ⊆ direct (alg_outer_step S t) t'"*
⟨*proof*⟩

**lemma** *alg_outer_step_impl: "(impl S) (t := {}) = impl (alg_outer_step S t)"*
⟨*proof*⟩

**lemma** *alg_outer_step_impl_to_trans[intro]:*
**assumes** *"(t', t'') ∈ impl S t"* **and** *"t' ∈ rel S ∨ t = t'"*
**shows** *"t'' ∈ trans (alg_outer_step S t)"*
⟨*proof*⟩

**lemma** *alg_outer_step_impl_to_direct[intro]:*
**assumes** *"(t', t'') ∈ impl S t"* **and** *"t' ∉ rel S"* **and** *"t ≠ t'"*
**shows** *"t'' ∈ direct (alg_outer_step S t) t'"*
⟨*proof*⟩
**lemma** *pre_star_alg_trans_to_lts:*
**assumes** *"alg_outer S = Some S'"*
**shows** *"trans S ⊆ rel S'"*
⟨*proof*⟩
**lemma** *pre_star_alg_direct_to_lts:*
**fixes** $S_0$ *:: "('s, 'n, 't) alg_state"*
**assumes** *"alg_outer* $S_0$ *= Some S'"*
**and** *"t ∉ rel* $S_0$*"* **and** *"t ∈ rel S'"*
**shows** *"direct* $S_0$ *t ⊆ rel S'"*
⟨*proof*⟩
**lemma** *pre_star_alg_impl_to_lts:*
**fixes** $S_0$ *:: "('s, 'n, 't) alg_state"*
**assumes** *"alg_outer* $S_0$ *= Some S'"*
**and** *"t ∉ rel* $S_0$*"* **and** *"t' ∉ rel* $S_0$*"*
**and** *"(t', t'') ∈ impl* $S_0$ *t"*
**and** *"t ∈ rel S'"* **and** *"t' ∈ rel S'"*
**shows** *"t'' ∈ rel S'"*
⟨*proof*⟩
**lemma** *pre_star_alg_new_refl_to_trans:*
**assumes** *"S = alg_state_new P Q T"* **and** *"(A, []) ∈ P"* **and** *"q ∈ Q"*
**shows** *"(q, Nt A, q) ∈ trans S"*
⟨*proof*⟩

**lemma** *pre_star_alg_refl_to_lts:*
**assumes** *"alg_outer (alg_state_new P Q T) = Some S'"* **and** *"(A, []) ∈ P"* **and** *"q ∈ Q"*
**shows** *"(q, Nt A, q) ∈ rel S'"*
⟨*proof*⟩
**lemma** *pre_star_alg_singleton_nt_to_lts:*
**assumes** *"alg_outer (alg_state_new P Q T) = Some S'"*
**and** *"(A, [Nt B]) ∈ P"* **and** *"q ∈ Q"* **and** *"q' ∈ Q"*
**shows** *"(q, Nt B, q') ∈ rel S' ⟹ (q, Nt A, q') ∈ rel S'"*

⟨*proof*⟩

**lemma** *pre_star_alg_tm_only_from_delta:*
**fixes** *S' :: "('s, 'n, 't) alg_state"*
**assumes** *"alg_outer (alg_state_new P Q T) = Some S'"*
**and** *"(q, Tm b, q') ∈ rel S'"* **and** *"q ∈ Q"* **and** *"q' ∈ Q"*
**shows** *"(q, Tm b, q') ∈ T"*
⟨*proof*⟩

**lemma** *pre_star_alg_singleton_tm_to_lts:*
**assumes** *"alg_outer (alg_state_new P Q T) = Some S'"* **and** *"(A, [Tm b]) ∈ P"*
**and** *"(q, Tm b, q') ∈ rel S'"* **and** *"q ∈ Q"* **and** *"q' ∈ Q"*
**shows** *"(q, Nt A, q') ∈ rel S'"*
⟨*proof*⟩

**lemma** *pre_star_alg_singleton_to_lts:*
**assumes** *"alg_outer (alg_state_new P Q T) = Some S'"*
**and** *"(A, [X]) ∈ P"* **and** *"q ∈ Q"* **and** *"q' ∈ Q"*
**shows** *"(q, X, q') ∈ rel S' ⟹ (q, Nt A, q') ∈ rel S'"*
⟨*proof*⟩
**lemma** *pre_star_alg_dual_to_lts:*
**assumes** *"alg_outer (alg_state_new P Q T) = Some S'"* **and** *"(A, [Nt B, Nt C]) ∈ P"*
**and** *"(q, Nt B, q') ∈ rel S'"* **and** *"(q', Nt C, q'') ∈ rel S'"*
**and** *"q ∈ Q"* **and** *"q' ∈ Q"* **and** *"q'' ∈ Q"*
**shows** *"(q, Nt A, q'') ∈ rel S'"*
⟨*proof*⟩

**lemma** *pre_star_alg_sup:*
**fixes** *P* **and** *T :: "('s, 'n, 't) trans"* **and** $q_0$
**defines** *"Q ≡ {$q_0$} ∪ states_lts T"*
**defines** *"S ≡ alg_state_new P Q T"*
**assumes** *"alg_outer S = Some S'"*
**and** *"pre_star_lts P Q T = Some T'"*
**and** *"CNF1 P"*
**shows** *"T' ⊆ rel S'"*
⟨*proof*⟩

## 7.4 Termination

**definition** *"alg_state_m_d S ≡ ({t. direct S t ≠ {}})"*
**definition** *"alg_state_m_i S ≡ ({t. impl S t ≠ {}})"*

**lemma** *alg_state_m_i_step_weak:*
**assumes** *"t ∈ trans S"*
**shows** *"alg_state_m_i (alg_outer_step S t) ⊆ alg_state_m_i S"*
⟨*proof*⟩

**lemma** *alg_state_m_i_step:*
**assumes** *"t ∈ trans S"* **and** *"impl S t ≠ {}"*
**shows** *"alg_state_m_i (alg_outer_step S t) ⊂ alg_state_m_i S"*
⟨*proof*⟩

**lemma** *alg_state_m_d_step_weak:*
**assumes** *"t ∈ trans S"* **and** *"impl S t = {}"*
**shows** *"alg_state_m_d (alg_outer_step S t) ⊆ alg_state_m_d S"*
⟨*proof*⟩

**lemma** *alg_state_m_d_step:*
**assumes** *"t ∈ trans S"* **and** *"impl S t = {}"* **and** *"direct S t ≠ {}"*
**shows** *"alg_state_m_d (alg_outer_step S t) ⊂ alg_state_m_d S"*
⟨*proof*⟩

**lemma** *alg_state_m_trans_step:*
**assumes** *"t ∈ trans S"* **and** *"impl S t = {}"* **and** *"direct S t = {}"*
**shows** *"trans (alg_outer_step S t) ⊂ trans S"*
⟨*proof*⟩

**lemmas** *alg_state_m_intros = alg_state_m_i_step_weak alg_state_m_i_step*
*alg_state_m_d_step_weak alg_state_m_d_step alg_state_m_trans_step*

**definition** *"alg_state_comp ≡ lex_prod less_than (lex_prod less_than less_than)"*

**definition** *alg_state_measure :: "('s, 'n, 't) alg_state ⇒ (nat × nat*
*× nat)"* **where**
*"alg_state_measure S ≡ (card (alg_state_m_i S), card (alg_state_m_d*
*S), card (trans S))"*

**lemma** *wf_alg_state_comp: "wf (inv_image alg_state_comp alg_state_measure)"*
⟨*proof*⟩

**definition** *alg_state_fin_inv :: "('s, 'n, 't) alg_state ⇒ bool"* **where**
*"alg_state_fin_inv S ≡ (*
*finite (rel S) ∧ finite (trans S) ∧*
*(∀t. finite (direct S t)) ∧ finite (alg_state_m_d S) ∧*
*(∀t. finite (impl S t)) ∧ finite (alg_state_m_i S)*
*)"*

**lemma** *alg_state_fin_inv_step:*
**assumes** *"alg_state_fin_inv S"*
**and** *"t ∈ trans S"*
**shows** *"alg_state_fin_inv (alg_outer_step S t)"*
⟨*proof*⟩

**lemma** *alg_state_fin_inv_step':*
**assumes** *"alg_state_fin_inv s"* **and** *"trans s ≠ {}"*
**shows** *"alg_state_fin_inv (alg_outer_step s (SOME x. x ∈ trans s))"*

⟨*proof*⟩

**lemma** *wf_alg_outer_step:*
**defines** *"b ≡ (λS. trans S ≠ {})"*
**and** *"c ≡ (λS. alg_outer_step S (SOME x. x ∈ trans S))"*
**shows** *"wf {(t, s). (alg_state_fin_inv s ∧ b s) ∧ t = c s}"*
⟨*proof*⟩

**lemma** *alg_outer_terminates:*
**assumes** *"alg_state_fin_inv S"*
**shows** *"∃S'. alg_outer S = Some S'"*
⟨*proof*⟩

**lemma** *alg_state_new_fin_inv:*
**fixes** *T :: "('s, 'n, 't) trans"*
**assumes** *"finite P"* **and** *"finite Q"* **and** *"finite T"*
**shows** *"alg_state_fin_inv (alg_state_new P Q T)"*
⟨*proof*⟩

### 7.5 Final Algorithm

**definition** *pre_star_code_cnf :: "('n, 't) Prods ⇒ ('s, ('n, 't) sym) auto ⇒ ('s, ('n, 't) sym) auto"* **where**
*"pre_star_code_cnf P M ≡ (*
— Construct the set of "interesting" states:
*let Q = {auto.start M} ∪ states_lts (auto.lts M) in*
*let S = alg_state_new P Q (auto.lts M) in*
*case alg_outer S of*
*Some S' ⇒ M (| auto.lts := (rel S') |)*
*)"*

**lemma** *pre_star_code_cnf_correct:*
**assumes** *"finite P"* **and** *"finite (auto.lts M)"* **and** *cnf: "CNF1 P"*
**shows** *"Lang_auto (pre_star_code_cnf P M) = pre_star P (Lang_auto M)"*
⟨*proof*⟩

**end**

## References


[BEF+00] Ahmed Bouajjani, Javier Esparza, Alain Finkel, Oded Maler, Peter Rossmanith, Bernard Willems, and Pierre Wolper. An efficient automata approach to some problems on context-free grammars. *Information Processing Letters*, 74(5-6):221–227, 2000. URL: https://doi.org/10.1016/S0020-0190(00)00055-7.

[BO93] Ronald V Book and Friedrich Otto. *String-rewriting systems*. Springer, 1993.

[Büc59] J. Richard Büchi. Regular canonical systems. Technical Report 3105 2794-7-T, Univ. of Michigan, 1959.

[Cau92] Didier Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, 1992.

[ER97] Javier Esparza and Peter Rossmanith. An automata approach to some problems on context-free grammars. In Christian Freksa, Matthias Jantzen, and Rüdiger Valk, editors, *Foundations of Computer Science: Potential - Theory - Cognition, to Wilfried Brauer on the occasion of his sixtieth birthday*, volume 1337 of *Lecture Notes in Computer Science*, pages 143–152. Springer, 1997. URL: https://doi.org/10.1007/BFb0052083.

[Lam09] Peter Lammich. Formalization of dynamic pushdown networks in Isabelle/HOL. 2009. URL: https://www21.in.tum.de/~lammich/isabelle/dpn-document.pdf.

[SSST23] Anders Schlichtkrull, Morten Konggaard Schou, Jiri Srba, and Dmitriy Traytel. Pushdown systems. *Archive of Formal Proofs*, October 2023. https://isa-afp.org/entries/Pushdown_Systems.html, Formal proof development.