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))}"
  by *fastforce*

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 (S₁ ∪ S₂) = Step_lts T w S₁ ∪ Step_lts T w S₂"*
unfolding *Step_lts_def* **by** *blast*

lemma *Steps_lts_mono: "s₁ ⊆ s₂ ⟹ Steps_lts T w s₁ ⊆ Steps_lts T w s₂"*
proof *(induction w arbitrary: s₁ s₂)*
case *Nil* **thus** *?case* **by** *(simp add: Steps_lts_def)*
next
case *(Cons w ws)*

define *s₁'* **where** *[simp]: "s₁' ≡ Step_lts T w s₁"*
define *s₂'* **where** *[simp]: "s₂' ≡ Step_lts T w s₂"*

have *"s₁' ⊆ s₂'"*
by *(simp add: Step_lts_def, use ‹s₁ ⊆ s₂› in blast)*
then have *"Steps_lts T ws s₁' ⊆ Steps_lts T ws s₂'"*
by *(elim Cons.IH)*
moreover have *"Steps_lts T (w#ws) s₁ = Steps_lts T ws s₁'"*
by *(simp add: Steps_lts_def)*
moreover have *"Steps_lts T (w#ws) s₂ = Steps_lts T ws s₂'"*
by *(simp add: Steps_lts_def)*
ultimately show *?case*
by *simp*
qed

lemma *Steps_lts_mono2:*

```
  assumes "T₁ ⊆ T₂" and "q₁ ⊆ q₂"
  shows "Steps_lts T₁ w q₁ ⊆ Steps_lts T₂ w q₂"
using assms(2) proof (induction w arbitrary: q₁ q₂)
  case Nil thus ?case by (simp add: Steps_lts_def)
next
  case (Cons w ws)
  have "Step_lts T₁ w q₁ ⊆ Step_lts T₂ w q₂"
    unfolding steps_lts_defs using assms(1) Cons(2) by blast
  then have "Steps_lts T₁ ws (Step_lts T₁ w q₁) ⊆ Steps_lts T₁ ws (Step_lts
T₂ w q₂)"
    by (rule Steps_lts_mono)
  then have "Steps_lts T₁ ws (Step_lts T₁ w q₁) ⊆ Steps_lts T₂ ws (Step_lts
T₂ w q₂)"
    using Cons(1) by blast
  then show ?case
    by (simp add: Steps_lts_def)
qed

lemma steps_lts_mono: "T₁ ⊆ T₂ ⟹ steps_lts T₁ w q ⊆ steps_lts T₂
w q"
  using Steps_lts_mono2[of T₁ T₂ "{q}" "{q}" w] by simp

lemma steps_lts_union: "q' ∈ steps_lts T w q ⟹ q' ∈ steps_lts (T ∪
T') w q"
using steps_lts_mono[of T "T ∪ T'"] by blast

lemma Steps_lts_path:
  assumes "q_f ∈ Steps_lts T w s"
  shows "∃q₀ ∈ s. q_f ∈ steps_lts T w q₀"
proof (insert assms; induction w arbitrary: s)
  case Nil thus ?case by (simp add: Steps_lts_def)
next
  case (Cons w ws)
  then have "q_f ∈ Steps_lts T ws (Step_lts T w s)"
    by (simp add: Steps_lts_def)
  moreover obtain q₀ where "q₀ ∈ (Step_lts T w s)" and "q_f ∈ steps_lts
T ws q₀"
    using Cons.IH calculation by blast
  ultimately obtain q' where "q₀ ∈ step_lts T w q'" and "q' ∈ s"
    unfolding steps_lts_defs by blast

  note ‹q₀ ∈ step_lts T w q'› and ‹q_f ∈ steps_lts T ws q₀›
  then have "q_f ∈ Steps_lts T ws (step_lts T w q')"
    using Steps_lts_mono[of "{q₀}"] by blast
  moreover have "Steps_lts T ws (step_lts T w q') = steps_lts T (w#ws)
q'"
    by (simp add: steps_lts_defs)
  ultimately show ?case
    using ‹q' ∈ s› by blast
```

qed

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 -
define *Q_f* **where** *[simp]: "Q_f = Steps_lts T (w_1@w_2) Q_0"*
define *Q'* **where** *[simp]: "Q' = Steps_lts T w_1 Q_0"*
have *"Q_f = Steps_lts T w_2 Q'"*
by *(simp add: Steps_lts_def)*
then obtain *q'* **where** *"q' ∈ Q'"* **and** *"q_f ∈ steps_lts T w_2 q'"*
using *assms Steps_lts_path* **by** *force*
moreover have *"q' ∈ Steps_lts T w_1 Q_0"*
using *calculation* **by** *simp*
ultimately show *?thesis*
by *blast*
qed

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 -
define *Q'* **where** *[simp]: "Q' = Steps_lts T w_1 Q_0"*
define *Q_f* **where** *[simp]: "Q_f = Steps_lts T w_2 Q'"*

have *"{q'} ⊆ Q'"*
using *assms(1)* **by** *simp*
then have *"Steps_lts T w_2 {q'} ⊆ Steps_lts T w_2 Q'"*
using *Steps_lts_mono* **by** *blast*
then have *"q_f ∈ Steps_lts T w_2 Q'"*
using *assms(2)* **by** *fastforce*
moreover have *"Q_f = Steps_lts T (w_1@w_2) Q_0"*
by *(simp add: Steps_lts_def)*
ultimately show *?thesis*
by *simp*
qed

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 -
obtain *q'* **where** *"q' ∈ Steps_lts T (w_1@w_2) Q_0 ∧ q_f ∈ steps_lts T w_3 q'"*
using *assms Steps_lts_split[***where** *w_1 = "w_1@w_2"]* **by** *fastforce*
moreover then obtain *q''* **where** *"q'' ∈ Steps_lts T w_1 Q_0 ∧ q' ∈ steps_lts T w_2 q''"*
using *Steps_lts_split* **by** *fast*
ultimately show *?thesis*

```
    by blast
qed

lemma Steps_lts_join3:
  assumes "q' ∈ steps_lts T w₁ q₀" and "q'' ∈ steps_lts T w₂ q'" and
"q_f ∈ steps_lts T w₃ q''"
  shows "q_f ∈ steps_lts T (w₁@w₂@w₃) q₀"
proof -
  have "q_f ∈ steps_lts T (w₂@w₃) q'"
    using assms(2) assms(3) Steps_lts_join by fast
  moreover then have "q_f ∈ steps_lts T (w₁@w₂@w₃) q₀"
    using assms(1) Steps_lts_join by fast
  ultimately show ?thesis
    by blast
qed

lemma Steps_lts_noState: "Steps_lts T w {} = {}"
proof (induction w)
  case Nil
  then show ?case
    by (simp add: Steps_lts_def)
next
  case (Cons w ws)
  moreover have "Steps_lts T [w] {} = {}"
    by (simp add: steps_lts_defs)
  ultimately show ?case
    by (simp add: Steps_lts_def)
qed
```

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
  fix q'
  assume "q' ∈ reachable_from T q"
  then obtain w where w_def: "q' ∈ steps_lts T w q"
    unfolding reachable_from_def by blast
  then consider "w = []" | "∃ws c. w = ws@[c]"
    by (meson rev_exhaust)
  then show "q' ∈ {q} ∪ (snd ‘ snd ‘ T)"
  proof (cases)
    case 1
    then show ?thesis
      using w_def Steps_lts_def by force
  next
```

```
    case 2
    then obtain ws c where "w = ws@[c]"
      by blast
    then obtain q1 where "q1 ∈ steps_lts T ws q" and "q' ∈ steps_lts
T [c] q1"
      using Steps_lts_split w_def by fast
    then have "(q1, c, q') ∈ T"
      by (auto simp: steps_lts_defs)
    then show ?thesis
      by force
  qed
qed

lemma reachable_from_trans[trans]:
  assumes "q1 ∈ reachable_from T q0" and "q2 ∈ reachable_from T q1"
  shows "q2 ∈ reachable_from T q0"
  using assms Steps_lts_join unfolding reachable_from_def by fast

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 (standard; standard)
  fix q'
  assume "q' ∈ reachable_from T q"
  then show "q' ∈ reachable_from (T ∪ T') q"
    unfolding reachable_from_def using steps_lts_union by fast
next
  fix q'
  assume "q' ∈ reachable_from (T ∪ T') q"
  then obtain w where "q' ∈ steps_lts (T ∪ T') w q"
    unfolding reachable_from_def by blast
  then have "∃w'. q' ∈ steps_lts T w' q"
  proof (induction w arbitrary: q)
    case Nil
    then have "q = q'" and "q ∈ steps_lts T [] q"
      unfolding Steps_lts_def by simp+
    then show ?case
      by blast
  next
    case (Cons c w)
    then obtain q1 where "q' ∈ steps_lts (T ∪ T') w q1" and "q1 ∈ steps_lts
(T ∪ T') [c] q"
      using Steps_lts_split[where w1="[c]" and w2=w] by force
    then obtain w' where w'_def: "q' ∈ steps_lts T w' q1"
      using Cons by blast

    have "q1 ∈ step_lts (T ∪ T') c q"
      using ‹q1 ∈ steps_lts (T ∪ T') [c] q› by (simp add: steps_lts_defs)
    then consider "q1 ∈ step_lts T c q" | "q1 ∈ step_lts T' c q"
```

```
    unfolding step_lts_def by blast
  then show ?case
  proof (cases)
    case 1
    then have "q1 ∈ steps_lts T [c] q"
      by (simp add: steps_lts_defs)
    then have "q' ∈ steps_lts T (c#w') q"
      using w'_def Steps_lts_join by force
    then show ?thesis
      by blast
  next
    case 2
    then have "(q, c, q1) ∈ T'"
      by (auto simp: step_lts_def)
    then obtain w'' where "q1 ∈ steps_lts T w'' q"
      using assms by blast
    then have "q' ∈ steps_lts T (w''@w') q"
      using w'_def Steps_lts_join by fast
    then show ?thesis
      by blast
  qed
 qed
 then show "q' ∈ reachable_from T q"
   by (simp add: reachable_from_def)
qed


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"
  unfolding Step_lts_def step_lts_def states_lts_def by auto

lemma Steps_states_lts: assumes "states_lts T ⊆ Q" shows "Q0 ⊆ Q ⟹
Steps_lts T u Q0 ⊆ Q"
  unfolding Steps_lts_def
  apply(induction u arbitrary: Q0)
   apply simp
  using assms by (simp add: Step_states_lts)

corollary steps_states_lts: "⟦ states_lts T ⊆ Q; q ∈ Q ⟧ ⟹ steps_lts
T u q ⊆ Q"
  using Steps_states_lts[of T Q "{q}"] by blast

lemma states_lts_Un: "states_lts (T ∪ T') = states_lts T ∪ states_lts
T'"
  unfolding states_lts_def by auto
```

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 ∈ A" and "f_A ∈ A" and "s_B ∈ B" and "f_B ∈ B" and "A
∩ B = {}"
    and sideA: "∀(q,c,q') ∈ T_A. q ∈ A ∧ q' ∈ A"
    and sideB: "∀(q,c,q') ∈ T_B. q ∈ B ∧ q' ∈ B"
    and "f_B ∈ steps_lts (T_A ∪ {(f_A, c, s_B)} ∪ T_B) w s_A"
  shows "∃w_A w_B. w = w_A@[c]@w_B ∧ f_A ∈ steps_lts T_A w_A s_A ∧ f_B ∈
steps_lts T_B w_B s_B"
using assms(1,8) proof (induction w arbitrary: s_A)
  case Nil
  then have "steps_lts (T_A ∪ {(f_A, c, s_B)} ∪ T_B) [] s_A = {s_A}"
    by (simp add: Steps_lts_def)
  then show ?case
    using Nil.prems assms(4,5) by fast
next
  case (Cons a w)
  define T where "T ≡ T_A ∪ {(f_A, c, s_B)} ∪ T_B"

  — Obtain intermediate state after reading a:
  note ‹f_B ∈ steps_lts (T_A ∪ {(f_A, c, s_B)} ∪ T_B) (a#w) s_A›
  then obtain q where a_step: "q ∈ steps_lts T [a] s_A"
    and w_step: "f_B ∈ steps_lts T w q"
    unfolding T_def using Steps_lts_split by force

  — There are now two options:
  — 1. a directly traverses the bridge to B, so a = c.
  — 2. a remains within A and we can use the IH.
  then show ?case
  proof (cases "(s_A, a, q) ∉ T_A")
    case True
    moreover have "(s_A, a, q) ∉ T_B"
      using Cons.prems(1) assms(5,7) by fast
    moreover have "(s_A, a, q) ∈ T"
      using a_step by (auto simp: steps_lts_defs)
    ultimately have "s_A = f_A" and "a = c" and "q = s_B"
      unfolding T_def by simp+

    have inB: "s_B ∈ B ⟹ f_B ∈ steps_lts T w s_B ⟹ f_B ∈ steps_lts
T_B w s_B"
    proof (induction w arbitrary: s_B)
      case Nil
      then show ?case
        by (simp add: Steps_lts_def)
    next
      case (Cons x xs)
      then obtain q where "f_B ∈ steps_lts T xs q" and "q ∈ steps_lts
T [x] s_B"
        using Steps_lts_split by force
      then have "(s_B, x, q) ∈ T"
        by (auto simp: steps_lts_defs)
```

```
      moreover have "s_B ∈ B"
        using Cons by simp
      ultimately have "(s_B, x, q) ∈ T_B" and "q ∈ B"
        unfolding T_def using assms(2,5,6,7) by blast+
      then have "q ∈ steps_lts T_B [x] s_B"
        by (auto simp: steps_lts_defs) force
      moreover have "f_B ∈ steps_lts T_B xs q"
        using ‹f_B ∈ steps_lts T xs q› ‹q ∈ B› Cons by simp
      ultimately show ?case
        using Steps_lts_join by force
    qed

    — The bridge is directly traversed, so A can be ignored:
    have "a#w = []@[c]@w"
      by (simp add: ‹a = c›)
    moreover have "f_A ∈ steps_lts T_A [] s_A"
      by (simp add: ‹s_A = f_A› Steps_lts_def)
    moreover have "f_B ∈ steps_lts T_B w s_B"
      using w_step assms(3) inB by (simp add: ‹q = s_B›)
    ultimately show ?thesis
      by blast
  next
    case False
    then have a_step': "q ∈ steps_lts T_A [a] s_A"
      by (auto simp: steps_lts_defs) (force)
    then have "q ∈ A"
      using False Cons.prems(1) assms(6) by fast

    — Introduce the IH:
    then have "∃w_A w_B. w = w_A @[c]@w_B ∧ f_A ∈ steps_lts T_A w_A q ∧
f_B ∈ steps_lts T_B w_B s_B"
      by (rule Cons.IH; use Cons.prems w_step[unfolded T_def] in simp)
    then obtain w_A w_B where "w = w_A @[c]@w_B" and "f_A ∈ steps_lts T_A
w_A q" and "f_B ∈ steps_lts T_B w_B s_B"
      by blast
    moreover then have "f_A ∈ steps_lts T_A (a#w_A) s_A"
      using a_step' Steps_lts_join by force
    ultimately have "a#w = a#w_A@[c]@w_B ∧ f_A ∈ steps_lts T_A (a#w_A) s_A
∧ f_B ∈ steps_lts T_B w_B s_B"
      by simp
    then show ?thesis
      by (intro exI) auto
  qed
qed
```

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 Σ.

```
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)"
  by (simp add: loop_lts_def)

lemma loop_lts_correct1: "set w ⊆ Σ ⟹ steps_lts (loop_lts q Σ) w
q = {q}"
proof (induction w)
  case Nil
  then show ?case
    by (simp add: Steps_lts_def)
next
  case (Cons w ws)
  then have "steps_lts (loop_lts q Σ) [w] q = {q}"
    unfolding loop_lts_def steps_lts_defs by fastforce
  moreover have "steps_lts (loop_lts q Σ) ws q = {q}"
    using Cons by simp
  ultimately show ?case
    by (simp add: Steps_lts_def)
qed

lemma loop_lts_correct2: "¬ set w ⊆ Σ ⟹ steps_lts (loop_lts q Σ)
w q = {}"
proof (induction w)
  case Nil
  then show ?case
    by simp
next
  case (Cons w ws)
  then consider "w ∉ Σ" | "¬ set ws ⊆ Σ"
    by auto
  then show ?case
  proof (cases)
    case 1
    then have "steps_lts (loop_lts q Σ) [w] q = {}"
      by (auto simp: loop_lts_def steps_lts_defs)
    moreover have "Steps_lts (loop_lts q Σ) ws {} = {}"
      by (meson Steps_lts_path ex_in_conv)
    ultimately show ?thesis
      by (metis Steps_lts_split all_not_in_conv append_Cons append_Nil)
  next
    case 2
    then have "steps_lts (loop_lts q Σ) ws q = {}"
      using Cons by simp
```

```
    moreover have "steps_lts (loop_lts q Σ) [w] q ⊆ {q}"
      by (auto simp: loop_lts_def steps_lts_defs)
    ultimately show ?thesis
      by (metis Steps_lts_def Steps_lts_mono Un_insert_right ex_in_conv
fold_simps(1,2) insert_absorb insert_not_empty sup.absorb_iff1)
  qed
qed

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 -
  define T where "T ≡ loop_lts () Σ"
  have "⋀w. set w ⊆ Σ ⟷ () ∈ steps_lts T w ()"
    unfolding T_def using loop_lts_correct by fast
  then show ?thesis
    by (auto simp: T_def auto_univ_def)
qed
```

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 Σp and Σs.

```
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)"
  by (auto intro: loop_lts_fin simp: pcs_lts_def)

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 -
  assume "∃p s. w = p@[c]@s ∧ set p ⊆ Σp ∧ set s ⊆ Σs"
  then obtain p s where "w = p@[c]@s" and "set p ⊆ Σp" and "set s
⊆ Σs"
    by blast
  moreover hence "0 ∈ steps_lts (pcs_lts Σp c Σs) p 0"
```

```
    by (metis pcs_lts_def steps_lts_union loop_lts_correct1 singletonI)
  moreover have "1 ∈ steps_lts (pcs_lts Σp c Σs) [c] 0"
    unfolding pcs_lts_def steps_lts_defs by force
  moreover have "1 ∈ steps_lts (pcs_lts Σp c Σs) s 1"
    unfolding pcs_lts_def
    using calculation(3) steps_lts_mono loop_lts_correct1
    by (metis UnCI insert_subset subsetI)
  ultimately show "1 ∈ steps_lts (pcs_lts Σp c Σs) w 0"
    using Steps_lts_join by meson
qed

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 -
  define T_A where [simp]: "T_A ≡ loop_lts (0::nat) Σp"
  define T_B where [simp]: "T_B ≡ loop_lts (1::nat) Σs"

  have "1 ∈ steps_lts (T_A ∪ {(0, c, 1)} ∪ T_B) w 0"
    using assms by (simp add: pcs_lts_def)
  then have "∃w_A w_B. w = w_A@[c]@w_B ∧ 0 ∈ steps_lts T_A w_A 0 ∧ 1 ∈
steps_lts T_B w_B 1"
    by (intro auto_merge[where A="{0}" and B="{1}"]) (simp add: loop_lts_def)+
  then obtain w_A w_B where w_split: "w = w_A@[c]@w_B" and "0 ∈ steps_lts
T_A w_A 0" and "1 ∈ steps_lts T_B w_B 1"
    by blast
  then have "set w_A ⊆ Σp" and "set w_B ⊆ Σs"
    using loop_lts_correct2 by fastforce+
  then show ?thesis
    using w_split by blast
qed

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 }"
  using pcs_lts_correct unfolding cps_auto_def
  by (metis (lifting) disjoint_insert(2) inf_bot_right select_convs(1,2,3))
```

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 (induction w)
  case Nil
  then show ?case
    by simp
next
  case (Cons w ws)
  moreover have "Steps_lts {} [w] {q₀} = {}"
    by (simp add: steps_lts_defs)
  moreover have "Steps_lts {} ws {} = {}"
    using Steps_lts_noState by fast
  ultimately show ?case
    by (simp add: Steps_lts_def)
qed

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"
  by (induction ws) auto

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 (induction ws)
  case Nil
  then show ?case
    by (simp add: Steps_lts_def)
next
  case (Cons w ws)
  have "0 ∈ steps_lts (word_lts ws) ws (length ws)"
    using Cons.IH(1) by blast
  then have "0 ∈ steps_lts (word_lts (w#ws)) ws (length ws)"
    using steps_lts_mono by (metis word_lts.simps(1) sup_ge1 subset_iff)
  moreover have "length ws ∈ steps_lts (word_lts (w#ws)) [w] (Suc (length
ws))"
  proof -
    have "(Suc (length ws), w, length ws) ∈ word_lts (w#ws)"
      by simp
```

```
    then show ?thesis
      unfolding steps_lts_defs by force
  qed
  ultimately show ?case
    using Steps_lts_join by force
qed

lemma word_lts_correct2:
  "0 ∈ steps_lts (word_lts ws) ws' (length ws) ⟹ ws = ws'"
proof (induction ws arbitrary: ws')
  case Nil
  then show ?case
    by (simp, metis equals0D steps_lts_empty_lts)
next
  case (Cons w ws)

  — Preparation to use auto_merge:
  define T_B where [simp]: "T_B ≡ word_lts ws"
  define B where [simp]: "B ≡ {n. n ≤ length ws}"
  define T where [simp]: "T ≡ {} ∪ {(Suc (length ws), w, length ws)}
∪ T_B"

  — Apply auto_merge:
  have "0 ∈ steps_lts T ws' (length (w#ws))"
    using Cons.prems by simp
  moreover have "∀(q, c, q')∈T_B. q ∈ B ∧ q' ∈ B"
    using word_lts_domain by force
  ultimately have "∃w_A w_B. ws' = w_A@[w]@w_B ∧ (Suc (length ws)) ∈ steps_lts
{} w_A (Suc (length ws)) ∧ 0 ∈ steps_lts T_B w_B (length ws)"
    by (intro auto_merge[where A="{Suc (length ws)}" and B=B]) simp+
  then obtain w_A w_B where ws'_split: "ws' = w_A@[w]@w_B"
      and w_A_step: "(length (w#ws)) ∈ steps_lts {} w_A (length (w#ws))"
      and w_B_step: "0 ∈ steps_lts T_B w_B (length ws)"
    by force

  have "w_A = []"
    using w_A_step steps_lts_empty_lts by fast

  — Use IH to show that w_B = ws:
  have "ws = w_B"
    by (intro Cons.IH, use w_B_step in simp)

  then show ?case
    using ws'_split by (simp add: ‹w_A = []› ‹ws = w_B›)
qed

lemmas word_lts_correct = word_lts_correct1 word_lts_correct2

lemma word_auto_lang[simp]: "Lang_auto (word_auto w) = {w}"
```

```
  unfolding word_auto_def using word_lts_correct[of w] by auto

lemma word_auto_finite_lts: "finite (lts (word_auto w))"
proof -
  have "finite (word_lts w)"
    by (induction w) simp+
  then show ?thesis
    by (simp add: word_auto_def)
qed

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')})"
  unfolding pre_lts_def image_def by(auto)

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'"
  using assms unfolding pre_star_lts_def while_saturate_def by (rule
while_option_rule)

lemma pre_star_lts_fp: "pre_star_lts P Q T = Some T' ⟹ pre_lts P Q
T' ⊆ T'"
  unfolding pre_star_lts_def while_saturate_def using while_option_stop
by fast

lemma pre_star_lts_mono: "pre_star_lts P Q T = Some T' ⟹ T ⊆ T'"
  by (rule pre_star_lts_rule) blast+
```

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"
  unfolding pre_lts_def by (rule reachable_add_trans) blast

lemma pre_star_lts_reachable:
  assumes "pre_star_lts P Q T = Some T'"
  shows "reachable_from T q = reachable_from T' q"
  by (rule pre_star_lts_rule; use assms pre_lts_reachable in fast)

lemma states_pre_lts: assumes "states_lts T ⊆ Q" shows "states_lts
(pre_lts P Q T) ⊆ Q"
using steps_states_lts[OF assms] unfolding pre_lts_def states_lts_def
by auto
```

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"
  using assms unfolding pre_lts_def Steps_lts_def Step_lts_def step_lts_def
by force

lemma pre_lts_pre:
  assumes "P ⊢ w_α ⇒ w_β" and "reachable_from T q ⊆ Q" and "q' ∈ steps_lts
T w_β q"
  shows "q' ∈ steps_lts (T ∪ pre_lts P Q T) w_α q"
proof -
  obtain w_p w_s A β where prod: "(A, β) ∈ P"
      and w_α_split: "w_α = w_p@[Nt A]@w_s"
      and w_β_split: "w_β = w_p@β@w_s"
    using assms(1) by (meson derive.cases)

  obtain q1 q2 where step_w_p: "q1 ∈ steps_lts T w_p q"
      and step_β: "q2 ∈ steps_lts T β q1"
      and step_w_s: "q' ∈ steps_lts T w_s q2"
    using Steps_lts_split3 assms(3)[unfolded w_β_split] by fast
  then have q1_reach: "q1 ∈ reachable_from T q" and "q2 ∈ reachable_from
T q1"
    using assms(2) unfolding reachable_from_def by blast+
  then have q2_reach: "q2 ∈ reachable_from T q"
    using assms(2) reachable_from_trans by fast

  have "q2 ∈ steps_lts (T ∪ pre_lts P Q T) [Nt A] q1"
    by (rule pre_lts_prod; use q1_reach q2_reach assms(2) prod step_β
in blast)
  moreover have "q1 ∈ steps_lts (T ∪ pre_lts P Q T) w_p q"
      and "q' ∈ steps_lts (T ∪ pre_lts P Q T) w_s q2"
    using step_w_p step_w_s steps_lts_union by fast+
  ultimately have "q' ∈ steps_lts (T ∪ pre_lts P Q T) w_α q"
    unfolding w_α_split using Steps_lts_join3 by fast
  then show ?thesis .
qed

lemma pre_lts_fp:
  assumes "P ⊢ w_α ⇒* w_β" and "reachable_from T q ⊆ Q" and "q' ∈ steps_lts
T w_β q"
    and fp: "pre_lts P Q T ⊆ T"
  shows "q' ∈ steps_lts T w_α q"
proof (insert assms, induction rule: converse_rtranclp_induct[where r="derive
P"])
  case base thus ?case by simp
next
  case (step y z)
  then show ?case
    using pre_lts_pre by (metis sup.order_iff)
```

```
qed

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 (insert assms, induction w arbitrary: q)
  case Nil
  then show ?case
    by (simp add: Steps_lts_def)
next
  case (Cons c w)
  then obtain q1 where step_w: "q' ∈ steps_lts (T ∪ pre_lts P Q T) w
q1"
      and step_c: "q1 ∈ steps_lts (T ∪ pre_lts P Q T) [c] q"
    using Steps_lts_split by (metis (no_types, lifting) append_Cons append_Nil)

  obtain w' where "q' ∈ steps_lts T w' q1" and "P ⊢ w ⇒* w'"
    using Cons step_w by blast

  have "∃c'. q1 ∈ steps_lts T c' q ∧ P ⊢ [c] ⇒* c'"
  proof (cases "q1 ∈ steps_lts T [c] q")
    case True
    then show ?thesis
      by blast
  next
    case False
    then have "q1 ∈ steps_lts (pre_lts P Q T) [c] q"
      using step_c unfolding Steps_lts_def Step_lts_def step_lts_def by
force
    then have "(q, c, q1) ∈ pre_lts P Q T"
      by (auto simp: Steps_lts_def Step_lts_def step_lts_def)
    then obtain A β where "(A, β) ∈ P" and "c = Nt A" and "q1 ∈ steps_lts
T β q"
      unfolding pre_lts_def by blast
    moreover have "P ⊢ [c] ⇒* β"
      using calculation by (simp add: derive_singleton r_into_rtranclp)
    ultimately show ?thesis
      by blast
  qed
  then obtain c' where "q1 ∈ steps_lts T c' q" and "P ⊢ [c] ⇒* c'"
    by blast

  have "q' ∈ steps_lts T (c'@w') q"
    using ‹q1 ∈ steps_lts T c' q› ‹q' ∈ steps_lts T w' q1› Steps_lts_join
by fast
  moreover have "P ⊢ (c#w) ⇒* (c'@w')"
    using ‹P ⊢ [c] ⇒* c'› ‹P ⊢ w ⇒* w'›
    by (metis (no_types, opaque_lifting) Cons_eq_appendI derives_append_decomp
self_append_conv2)
```

```
  ultimately show ?case
    by blast
qed

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 -
  obtain w' where "P ⊢ w ⇒* w'" and "q' ∈ steps_lts T' w' q"
    using pre_lts_sub_aux assms by fast
  then obtain w'' where "P ⊢ w' ⇒* w''" and "q' ∈ steps_lts T w'' q"
    using assms(1) by blast
  moreover have "P ⊢ w ⇒* w''"
    using ‹P ⊢ w ⇒* w'› calculation(1) by simp
  ultimately show ?thesis
    by blast
qed

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 -
  let ?I = "λT'. ∀w. (q' ∈ steps_lts T' w q) ⟶ (∃w'. P ⊢ w ⇒* w' ∧
q' ∈ steps_lts T w' q)"
  have "⋀T'. ?I T' ⟹ ?I (T' ∪ pre_lts P Q T')"
    by (simp add: pre_lts_sub[where T=T])
  then have "?I T'"
    by (rule pre_star_lts_rule[where T=T and T'=T']; use assms in blast)
  then show "(q' ∈ steps_lts T' w q) ⟹ (∃w'. P ⊢ w ⇒* w' ∧ q' ∈
steps_lts T w' q)"
    by simp
qed

lemma pre_star_lts_correct:
  assumes "reachable_from T q₀ ⊆ Q" and "pre_star_lts P Q T = Some T'"
  shows "Lang_lts T' q₀ F = pre_star P (Lang_lts T q₀ F)"
proof (standard; standard)
  fix w
  assume "w ∈ Lang_lts T' q₀ F"
  then obtain q_f where "q_f ∈ steps_lts T' w q₀" and "q_f ∈ F"
    by blast
  then obtain w' where "P ⊢ w ⇒* w'" and "q_f ∈ steps_lts T w' q₀"
    using pre_star_lts_sub assms by fast
  moreover have "w' ∈ Lang_lts T q₀ F"
    using calculation ‹q_f ∈ F› by blast
  ultimately show "w ∈ pre_star P (Lang_lts T q₀ F)"
```

```
    unfolding pre_star_def by blast
next
  fix w
  assume "w ∈ pre_star P (Lang_lts T q₀ F)"
  then obtain w' where "P ⊢ w ⇒* w'" and "w' ∈ Lang_lts T q₀ F"
    unfolding pre_star_def by blast
  then obtain q_f where "q_f ∈ steps_lts T w' q₀" and "q_f ∈ F"
    by blast
  then have "q_f ∈ steps_lts T' w' q₀"
    using steps_lts_mono pre_star_lts_mono assms by (metis in_mono)
  moreover have "reachable_from T' q₀ ⊆ Q"
    using assms pre_star_lts_reachable by fast
  moreover have "pre_lts P Q T' ⊆ T'"
    by (rule pre_star_lts_fp; use assms(2) in simp)
  moreover note ‹P ⊢ w ⇒* w'›
  ultimately have "q_f ∈ steps_lts T' w q₀"
    by (elim pre_lts_fp) simp+
  with ‹q_f ∈ F› show "w ∈ Lang_lts T' q₀ F"
    by blast
qed
```

4.4 Termination

```
lemma pre_star_lts_terminates:
  fixes P :: "('n, 't) Prods" and Q :: "'s set" and T₀ :: "('s, ('n,
't) sym) lts"
  assumes "finite P" and "finite Q" and "finite T₀" and "states_lts
T₀ ⊆ Q"
  shows "∃T. pre_star_lts P Q T₀ = Some T"
proof -
  define b :: "('s, ('n, 't) sym) lts ⇒ bool" where
    [simp]: "b = (λT. ¬ pre_lts P Q T ⊆ T)"
  define f :: "('s, ('n, 't) sym) lts ⇒ ('s, ('n, 't) sym) lts" where
    [simp]: "f = pre_lts P Q"
  then have "mono f"
    unfolding mono_def pre_lts_def
    by (smt (verit, ccfv_threshold) UnCI UnE in_mono mem_Collect_eq Steps_lts_mono2
subsetI)

  define U :: "('s, ('n, 't) sym) lts" where
    "U = { (q, Nt A, q') | q q' A. q ∈ Q ∧ (∃β. (A, β) ∈ P ∧ q' ∈ Q)}
∪ T₀"
  have "T₀ ⊆ U" by(simp add: U_def)
  have "⋀p a q. (p,a,q) ∈ T₀ ⟹ p ∈ Q ∧ q ∈ Q"
    using assms(4) unfolding states_lts_def by auto
  then have "pre_lts P Q T ⊆ U" if asm: "T ⊆ U" for T
    using asm steps_states_lts[of T Q] unfolding U_def pre_lts_def states_lts_def
    by fastforce
  then have U_bounds: "⋀X. X ⊆ U ⟹ f X ⊆ U"
```

```
    by simp

  have "finite U"
  proof -
    define U' :: "('s, ('n, 't) sym) lts" where
      [simp]: "U' = Q × ((λ(A,_). Nt A) ‘ P) × Q"
    have "finite ((λ(A,_). Nt A) ‘ P)"
      using assms(1) by simp
    then have "finite U'"
      using assms(2) U'_def by blast

    define T' :: "('s, ('n, 't) sym) lts" where
      [simp]: "T' = { (q,Nt A,q') | q q' A. q ∈ Q ∧ (∃β. (A, β) ∈ P
∧ q' ∈ Q)}"
    then have "T' ⊆ U'"
      unfolding T'_def U'_def using assms(1) by fast
    moreover note <finite U'>
    ultimately have "finite T'"
      using rev_finite_subset[of U' T'] by blast
    then show "finite U"
      by (simp add: U_def assms)
  qed

  show ?thesis
    unfolding pre_star_lts_def
    using while_saturate_finite_subset_Some[of f U, OF <mono f> U_bounds
<finite U> <T₀ ⊆ U>]
    by (simp)
qed
```

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 -
  define T where "T ≡ auto.lts M"
  define Q where "Q ≡ {auto.start M} ∪ states_lts T"
  then have "finite Q"
    unfolding T_def states_lts_def using assms(2) by auto
  have MQ: "states_lts (auto.lts M) ⊆ Q" unfolding Q_def T_def by (force)
```

```
  have "reachable_from T (auto.start M) ⊆ Q"
    using reachable_from_computable unfolding Q_def states_lts_def by
fastforce
  moreover obtain T' where T'_def: "pre_star_lts P Q T = Some T'"
    using pre_star_lts_terminates[OF assms(1) <finite Q> assms(2) MQ]
T_def by blast
  ultimately have "Lang_lts T' (auto.start M) (auto.finals M)
    = pre_star P (Lang_lts T (auto.start M) (auto.finals M))"
    by (rule pre_star_lts_correct)
  then have "Lang_auto (M (| auto.lts := T' |)) = pre_star P (Lang_auto
M)"
    by (simp add: T_def)
  then show ?thesis
    unfolding pre_star_auto_def using Q_def T'_def T_def
    by(force)
qed

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 -
  have "q ∈ steps_lts T' [] q"
    unfolding Steps_lts_def using assms by force
  then have "(q, Nt A, q) ∈ pre_lts P Q T'"
    unfolding pre_lts_def using assms by blast
  moreover have "T' = T' ∪ pre_lts P Q T'"
    using pre_star_lts_fp assms(1) by blast
  ultimately show ?thesis
    by blast
qed

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 -
  have "q' ∈ steps_lts T' [B] q"
    unfolding steps_lts_defs using assms by force
  then have "(q, Nt A, q') ∈ pre_lts P Q T'"
    unfolding pre_lts_def using assms by blast
  moreover have "T' = T' ∪ (pre_lts P Q T')"
    using pre_star_lts_fp assms(1) by blast
  ultimately show ?thesis
    by blast
qed

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 -
  have "q'' ∈ steps_lts T' [B, C] q"
    unfolding steps_lts_defs using assms by force
  then have "(q, Nt A, q'') ∈ pre_lts P Q T'"
    unfolding pre_lts_def using assms by blast
  moreover have "T' = T' ∪ pre_lts P Q T'"
    using pre_star_lts_fp assms(1) by blast
  ultimately show ?thesis
    by blast
qed
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}|)"
by *eval*

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)"*
unfolding *pre_star_def* **by** *blast*

lemma *pre_star_word:*
"[Nt S] ∈ pre_star P (map Tm ‘ L) ⟷ (∃w. w ∈ L ∧ w ∈ Lang P S)"
unfolding *Lang_def pre_star_def* **by** *blast*

lemma *pre_star_lang:*
"Lang P S ∩ L = {} ⟷ [(Nt S)] ∉ pre_star P (map Tm ‘ L)"
using *pre_star_word*[**where** *P=P*] **by** *blast*

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}"
  by (simp add: Lang_def pre_star_def)

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 -
  define M where [simp]: "M ≡ word_auto β"
  have "Lang_auto (pre_star_auto (set P) M) = pre_star (set P) (Lang_auto
M)"
    by (intro pre_star_auto_correct; simp add: word_auto_finite_lts)
  then show ?thesis
    using pre_star_derivability by force
qed
```

5.2 Membership Problem

```
lemma pre_star_membership[code_unfold]: "(w ∈ Lang P S) = (P ⊢ [Nt S]
⇒* map Tm w)"
  by (simp add: Lang_def)
```

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] ⇒* [])"
  by (simp add: is_nullable_def)
```

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"
  using assms proof (induction rule: converse_rtranclp_induct[where r="derive
P"])
  case base
  then show ?case
    by simp
next
  case (step y z)
  then have "set z ⊆ Syms P"
    using derives_set_subset by blast
  then show ?case
    using step by simp
```

qed

lemma *cfg_Lang_univ: "P ⊢ [Nt X] ⇒* map Tm β ⟹ set β ⊆ Tms P"*
proof -
  assume *"P ⊢ [Nt X] ⇒* map Tm β"*
  moreover have *"Nt X ∈ Syms P"*
    using *Syms_def calculation derives_start1* **by** *fastforce*
  ultimately have *"set (map Tm β) ⊆ Syms P"*
    using *cfg_derives_Syms* **by** *force*
  moreover have *"⋀t. (t ∈ Tms P) ⟷ Tm t ∈ Syms P"*
    unfolding *Tms_def Syms_def Tms_syms_def* **by** *blast*
  ultimately show *"set β ⊆ Tms P"*
    by *force*
qed

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 -
  have *"Lang P S = {} ⟷ (∄w. P ⊢ [Nt S] ⇒* map Tm w)"*
    by *(simp add: Lang_def)*
  also have *"... ⟷ (∄w. P ⊢ [Nt S] ⇒* map Tm w ∧ set w ⊆ Tms P)"*
    using *cfg_Lang_univ* **by** *fast*
  also have *"... ⟷ (∄w. P ⊢ [Nt S] ⇒* w ∧ set w ⊆ Tm ‘ Tms P)"*
    by *(smt (verit, best) cfg_Lang_univ ex_map_conv imageE image_mono list.set_map subset_iff)*
  also have *"... ⟷ [Nt S] ∉ pre_star P {w. set w ⊆ Tm ‘ Tms P}"*
    unfolding *pre_star_def* **by** *blast*
  finally show *?thesis* **.**
qed

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 -
  define *M :: "(unit, ('n, 't) sym) auto"* **where** *[simp]: "M ≡ auto_univ (Tm ‘ Tms (set P))"*
  have *"finite (Tm ‘ Tms (set P))"*
    using *finite_Tms* **by** *blast*
  then have *"Lang_auto (pre_star_auto (set P) M) = pre_star (set P) (Lang_auto*

```
M)"
    by (intro pre_star_auto_correct; auto simp: auto_univ_def intro: loop_lts_fin)
  then show ?thesis
    using pre_star_emptiness unfolding M_def auto_univ_lang by fastforce
qed
```

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))"
```

— $X \in V$ 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 -
  define L where "L ≡ { (α::('n, 't) syms)@[Nt X]@β | α β. set α ⊆
Syms P ∧ set β ⊆ Syms P }"
  have "[Nt S] ∈ pre_star P L ⟷ (∃w. w ∈ L ∧ P ⊢ [Nt S] ⇒* w)"
    by (simp add: pre_star_term)
  also have "... ⟷ (∃α β. P ⊢ [Nt S] ⇒* (α@[Nt X]@β) ∧ set α ⊆ Syms
P ∧ set β ⊆ Syms P)"
    unfolding L_def by blast
  also have "... ⟷ (∃α β. P ⊢ [Nt S] ⇒* (α@[Nt X]@β))"
  proof -
    have "⋀w. P ⊢ [Nt S] ⇒ w ⟹ set w ⊆ Syms P"
      by (smt (verit, best) Syms_def UN_I UnCI case_prod_conv derive_singleton
subset_eq)
    then have "⋀w. w ≠ [Nt S] ⟹ P ⊢ [Nt S] ⇒* w ⟹ set w ⊆ Syms
P"
      by (metis cfg_derives_Syms converse_rtranclpE)
    then have "⋀α β. P ⊢ [Nt S] ⇒* (α@[Nt X]@β) ⟹ set α ⊆ Syms P
∧ set β ⊆ Syms P"
      by (smt (verit) Cons_eq_append_conv append_is_Nil_conv empty_set
empty_subsetI le_supE list.discI set_append)
    then show ?thesis
```

```
    by blast
  qed
  finally show ?thesis
    by (simp add: is_reachable_from_def L_def)
qed

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 -
  define M :: "(nat, ('n, 't) sym) auto" where [simp]: "M ≡ cps_auto
(Nt X) (Syms (set P))"
  have "finite (Syms (set P))"
    unfolding Syms_def by fast
  then have "Lang_auto (pre_star_auto (set P) M) = pre_star (set P) (Lang_auto
M)"
    by (intro pre_star_auto_correct; auto simp: cps_auto_def intro: pcs_lts_fin)
  then show ?thesis
    using pre_star_reachable unfolding M_def cps_auto_lang by fastforce
qed
```

5.6 Disjointness and Subset Problem

```
theorem pre_star_disjointness: "Lang P S ∩ L = {} ⟷ [(Nt S)] ∉ pre_star
P (map Tm ‘ L)"
  by (simp add: pre_star_lang)

theorem pre_star_subset: "Lang P S ⊆ L ⟷ [(Nt S)] ∉ pre_star P (map
Tm ‘ (-L))"
proof -
  have "Lang P S ⊆ L ⟷ Lang P S ∩ -L = {}"
    by blast
  then show ?thesis
    by (simp add: pre_star_disjointness)
qed

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" by eval

lemma "¬ is_nullable P B" by eval

lemma "is_nullable P C" by eval

lemma "¬ is_nullable P D" by eval
```

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

```
lemma "P ⊢ [Nt A] ⇒* [Nt A, Nt B, Nt B]"
  by eval

— But A ⇒* AB is not:
lemma "¬ P ⊢ [Nt A] ⇒* [Nt A, Nt B]"
  by eval
```

Following derivability, the membership problem is straight-forward:

lemma *"[a] ∈ Lang P A"*
  by *eval*

— While *b ∈ L(G)*:
lemma *"[b] ∉ Lang P A"*
  by *eval*

— But *bb ∈ L(G)* again holds:
lemma *"[b,b] ∈ Lang P A"*
  by *eval*

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 = {}"*
  unfolding *is_empty_def[symmetric]* **by** *eval*

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

lemma *"P ⊢ A ⇒^? B"*
  by *eval*

lemma *"P ⊢ B ⇒^? A"*
  by *eval*

lemma *"P ⊢ A ⇒^? C"*
  by *eval*

lemma *"P ⊢ B ⇒^? C"*
  by *eval*

lemma *"¬ P ⊢ C ⇒^? A"*
  by *eval*

lemma *"¬ P ⊢ C ⇒^? A"*
  by *eval*

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:

$L(C) = \infty \longleftrightarrow \exists X\ \alpha\ \beta.\ X \Rightarrow^* \alpha X \beta \wedge a\beta \neq \varepsilon$

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)"*
  using *assms derives_append_decomp* **by** *blast*

lemma *derives_split:*
  assumes *"P ⊢ X ⇒* w"*
  shows *"∃X₁ X₂ w₁ w₂. X = X₁@X₂ ∧ w = w₁@w₂ ∧ P ⊢ X₁ ⇒* w₁ ∧ P ⊢ X₂ ⇒* w₂"*
  using *assms* **by** *blast*

lemma *derives_step:*
  assumes *"P ⊢ X ⇒* (α@w₁@β)"* **and** *"P ⊢ w₁ ⇒* w₂"*
  shows *"P ⊢ X ⇒* (α@w₂@β)"*
proof -
  have *"P ⊢ w₁@β ⇒* w₂@β"*
    using *assms(2)* **by** *(simp add: derives_concat)*
  then have *"P ⊢ α@w₁@β ⇒* α@w₂@β"*
    by *(simp add: derives_concat)*
  then show *?thesis*
    using *assms(1)* **by** *simp*
qed

lemma *is_useful_all_derive:*
  assumes *"is_useful_all"*

```
  shows "∃w. P ⊢ xs ⇒* map Tm w"
using assms proof (induction xs)
  case Nil
  moreover have "P ⊢ [] ⇒* map Tm []"
    by simp
  ultimately show ?case
    by (elim exI)
next
  case (Cons a xs)
  then obtain w' where w'_def: "P ⊢ xs ⇒* map Tm w'"
    by blast

  have "∃w. P ⊢ [a] ⇒* map Tm w"
  proof (cases a)
    case (Nt X)
    then have "Lang P X ≠ {}"
      using Cons(2) by (simp add: is_useful_all_def is_useful_def)
    then show ?thesis
      by (simp add: Nt Lang_def)
  next
    case (Tm c)
    then have "P ⊢ [Tm c] ⇒* map Tm [c]"
      by simp
    then show ?thesis
      using Tm by blast
  qed
  then obtain w where w_def: "P ⊢ [a] ⇒* map Tm w"
    by blast

  from w_def w'_def have "P ⊢ (a#xs) ⇒* map Tm (w@w')"
    using derives_concat by fastforce
  then show ?case
    by blast
qed

lemma is_non_nullable_all_derive:
  assumes "is_non_nullable_all" and "P ⊢ xs ⇒* w"
  shows "xs = [] ⟷ w = []"
proof -
  have "⋀X. ¬ P ⊢ [Nt X] ⇒* []"
    using assms(1) by (simp add: is_non_nullable_all_def is_nullable_def)
  moreover have "⋀c. ¬ P ⊢ [Tm c] ⇒* []"
    by simp
  ultimately have nonNullAll: "⋀x. ¬ P ⊢ [x] ⇒* []"
    using sym.exhaust by metis

  have thm1: "xs = [] ⟹ w = []"
    using assms(2) derives_from_empty by blast
```

```
  have thm2: "xs ≠ [] ⟹ w ≠ []"
  proof
    assume "xs ≠ []"
    then obtain x xs' where "xs = x#xs'"
      using list.exhaust by blast
    moreover have "P ⊢ ([x]@xs') ⇒* [] ⟹ (P ⊢ [x] ⇒* [] ∧ P ⊢ xs'
⇒* [])"
      using derives_split by (metis Nil_is_append_conv derives_append_decomp)
    moreover have "¬ P ⊢ [x] ⇒* []"
      by (simp add: nonNullAll)
    ultimately show "w = [] ⟹ False"
      using assms(2) by simp
  qed

  show ?thesis
    using thm1 thm2 by blast
qed
```

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_α w_β (Suc n) = w_α@(is_infinite_words w_X w_α
w_β n)@w_β" |
  "is_infinite_words w_X w_α w_β 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 -
  consider "∃a. (α@[Nt X]@β) = [Tm a]" | "∃B C. (α@[Nt X]@β) = [Nt B,
Nt C]"
```

```
    using assms cnf by blast
  then show ?thesis
  proof (cases)
    case 1
    then have "False"
      by (simp add: append_eq_Cons_conv)
    then show ?thesis
      by simp
  next
    case 2
    then obtain B C where BC_def: "(α@[Nt X]@β) = [Nt B, Nt C]"
      by blast
    then have "X = B ∨ X = C"
      by (metis Nil_is_append_conv append_Cons in_set_conv_decomp in_set_conv_decomp_first
set_ConsD sym.inject(1))
    then have "P ⊢ [Nt Y] ⇒ []@[Nt X]@[Nt C] ∣ P ⊢ [Nt Y] ⇒ [Nt B]@[Nt
X]@[]"
      using BC_def assms(1) derive_singleton by force
    then show ?thesis
      unfolding is_reachable_step_def by (rule disjE) blast+
  qed
qed

lemma reachable_rel_tran: "(X, Y) ∈ reachable_rel⁺ ⟹ Y →^? X"
proof (induction rule: trancl.induct)
  case (r_into_trancl X Y)
  then show "Y →^? X"
    using cnf cnf_implies_pumping by (auto simp: reachable_rel_def)
next
  case (trancl_into_trancl X Y Z)
  then have "Z →^? Y"
    using cnf cnf_implies_pumping by (auto simp: reachable_rel_def)
  with trancl_into_trancl(3) have "Z →^? X"
  proof -
    assume "Z →^? Y" and "Y →^? X"

    obtain α_Z β_Z where z_der: "P ⊢ [Nt Z] ⇒* (α_Z@[Nt Y]@β_Z)" and "α_Z@β_Z
≠ []"
      using ‹Z →^? Y›[unfolded is_reachable_step_def] by blast
    obtain α_Y β_Y where y_der: "P ⊢ [Nt Y] ⇒* (α_Y@[Nt X]@β_Y)" and
"α_Y@β_Y ≠ []"
      using ‹Y →^? X›[unfolded is_reachable_step_def] by blast

    have "P ⊢ [Nt Z] ⇒* (α_Z@α_Y@[Nt X]@β_Y@β_Z)"
      using z_der y_der by (metis append.assoc derives_step)
    moreover have "α_Z@α_Y@β_Y@β_Z ≠ []"
      using ‹α_Z@β_Z ≠ []› ‹α_Y@β_Y ≠ []› by simp
    ultimately show "Z →^? X"
      unfolding is_reachable_step_def by (metis append.assoc)
```

```
  qed
  then show ?case
    by simp
qed

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 -
  define Nt2 :: "'n × 'n ⇒ ('n, 't) sym × ('n, 't) sym"
    where "Nt2 ≡ (λ(a,b). (Nt a, Nt b))"
  define S :: "(('n, 't) sym × ('n, 't) sym) set"
    where "S ≡ ⋃(set ‘ snd ‘ P) × (Nt ‘ fst ‘ P)"

  have "finite (⋃(set ‘ snd ‘ P))"
    by (rule finite_Union; use assms(1) in blast)
  moreover have "finite (fst ‘ P)"
    using assms(1) by simp
  ultimately have "finite S"
    unfolding S_def by blast
  moreover have "(Nt2 ‘ reachable_rel) ⊆ S"
    unfolding reachable_rel_def Nt2_def S_def by (auto split: prod.splits
sym.splits, force)
  ultimately have "finite (Nt2 ‘ reachable_rel)"
    using finite_subset by blast
  moreover have "inj_on Nt2 reachable_rel"
    unfolding inj_on_def Nt2_def by fast
  ultimately have finite: "finite reachable_rel"
    using finite_image_iff by blast

  have "acyclic reachable_rel"
  unfolding acyclic_def using loopfree reachable_rel_tran by blast

  from finite_acyclic_wf[OF finite this] show "wf reachable_rel" .
qed

lemma is_infinite_implies_finite:
  assumes "finite P"
    and loopfree: "⋀X. ¬ X →^? X"
  shows "finite {w. P ⊢ [Nt X] ⇒* w}"
proof -
  have "wf reachable_rel"
    using assms cnf by (simp add: reachable_rel_wf)
  then show ?thesis
  proof (induction)
    case (less X)
```

```
    have "{w. ∃a. (X, [Tm a]) ∈ P ∧ P ⊢ [Tm a] ⇒* w} = snd ‘ {(Y, β)
∈ P. X = Y ∧ (∃a. β = [Tm a])}"
      by force
    then have finA: "finite {w. ∃a. (X, [Tm a]) ∈ P ∧ P ⊢ [Tm a] ⇒*
w}"
      using assms(1) by (metis (no_types, lifting) case_prod_conv finite_imageI
mem_Collect_eq old.prod.exhaust rev_finite_subset subsetI)

    have "⋀B C. (X, [Nt B, Nt C]) ∈ P ⟹ finite {w. P ⊢ [Nt B, Nt C]
⇒* w}"
    proof -
      fix B and C
      assume "(X, [Nt B, Nt C]) ∈ P"
      then have "(X, []@[Nt B]@[Nt C]) ∈ P" and "(X, [Nt B]@[Nt C]@[])
∈ P"
        by simp+
      then have "(B, X) ∈ reachable_rel" and "(C, X) ∈ reachable_rel"
        unfolding reachable_rel_def by blast+
      then have "finite {w. P ⊢ [Nt B] ⇒* w}" and "finite {w. P ⊢ [Nt
C] ⇒* w}"
        using less by simp+
      moreover have "{w. P ⊢ [Nt B, Nt C] ⇒* w} = (λ(b,c). b@c) ‘ ({w.
P ⊢ [Nt B] ⇒* w} × {w. P ⊢ [Nt C] ⇒* w})"
      proof (standard; standard)
        fix w
        assume "w ∈ {w. P ⊢ [Nt B, Nt C] ⇒* w}"
        then have "P ⊢ [Nt B]@[Nt C] ⇒* w"
          by simp
        then obtain b c where "P ⊢ [Nt B] ⇒* b" and "P ⊢ [Nt C] ⇒*
c" and "w = b@c"
          using derives_append_decomp by blast
        then show "w ∈ (λ(b,c). b@c) ‘ ({w. P ⊢ [Nt B] ⇒* w} × {w.
P ⊢ [Nt C] ⇒* w})"
          by blast
      next
        fix w
        assume "w ∈ (λ(b,c). b@c) ‘ ({w. P ⊢ [Nt B] ⇒* w} × {w. P ⊢
[Nt C] ⇒* w})"
        then obtain b c where "P ⊢ [Nt B] ⇒* b" and "P ⊢ [Nt C] ⇒*
c" and "w = b@c"
          by fast
        then have "P ⊢ [Nt B]@[Nt C] ⇒* w"
          using derives_concat by blast
        then show "w ∈ {w. P ⊢ [Nt B, Nt C] ⇒* w}"
          by simp
      qed
      ultimately show "finite {w. P ⊢ [Nt B, Nt C] ⇒* w}"
        by simp
```

```
    qed
    moreover have "finite {(B, C). (X, [Nt B, Nt C]) ∈ P}"
    proof -
      define S :: "('n × ('n, 't) sym list) set" where
          "S ≡ ((λ(B,C). (X, [Nt B, Nt C])) ‘ {(B, C). (X, [Nt B, Nt
C]) ∈ P})"
      have subP: "S ⊆ P"
        unfolding S_def by fast
      with assms(1) have "finite S"
        by (elim finite_subset)
      then show ?thesis
        unfolding S_def by (rule finite_imageD, simp add: inj_on_def)
    qed
    ultimately have "finite (⋃((λ(B,C). {w. P ⊢ [Nt B, Nt C] ⇒* w})
‘ {(B,C). (X, [Nt B, Nt C]) ∈ P}))"
      by (intro finite_Union; fast)
    moreover have "{w. ∃B C. (X, [Nt B, Nt C]) ∈ P ∧ P ⊢ [Nt B, Nt
C] ⇒* w}
        = (⋃((λ(B,C). {w. P ⊢ [Nt B, Nt C] ⇒* w}) ‘ {(B,C). (X, [Nt
B, Nt C]) ∈ P}))"
      by blast
    ultimately have finB: "finite {w. ∃B C. (X, [Nt B, Nt C]) ∈ P ∧ P
⊢ [Nt B, Nt C] ⇒* w}"
      by simp

    let ?P = "λw β. (X, β) ∈ P ∧ P ⊢ β ⇒* w"
    have un: "{w. ∃β. ?P w β} = {w. ∃a. ?P w [Tm a]} ∪ {w. ∃B C. ?P
w [Nt B, Nt C]}"
      using cnf by blast
    have "finite {w. ∃β. (X, β) ∈ P ∧ P ⊢ β ⇒* w}"
      unfolding un by (intro finite_UnI; use finA finB in simp)
    moreover have "⋀X. {w. P ⊢ [Nt X] ⇒* w} = {[Nt X]} ∪ {w. ∃β. (X,
β) ∈ P ∧ P ⊢ β ⇒* w}"
      by (auto split: prod.splits simp: derives_Cons_decomp)
    ultimately show ?case
      by simp
  qed
qed

theorem is_infinite_correct:
  assumes "is_useful_all" and "is_non_nullable_all" and "finite P"
  shows "¬ finite (Lang P S) ⟷ is_infinite"
proof (standard, erule contrapos_pp)
  assume "¬ is_infinite"
  then have finA: "finite {w. P ⊢ [Nt S] ⇒* w}"
    using is_infinite_implies_finite assms(3) by (simp add: is_infinite_def)
  have "finite (map Tm ‘ {w. P ⊢ [Nt S] ⇒* map Tm w}::('n, 't) sym list
set)"
    by (rule finite_subset[where B="{w. P ⊢ [Nt S] ⇒* w}"]; use finA
```

```
in blast)
  moreover have "inj_on (map Tm) {w. P ⊢ [Nt S] ⇒* map Tm w}"
    by (simp add: inj_on_def)
  ultimately have "finite {w. P ⊢ [Nt S] ⇒* map Tm w}"
    using finite_image_iff[where f="map Tm"] by blast
  then show "¬ infinite (Lang P S)"
    by (simp add: Lang_def)
next
  assume "is_infinite"
  then obtain X where "X →^? X"
    unfolding is_infinite_def by blast
  then obtain α β where deriveX: "P ⊢ [Nt X] ⇒* (α@[Nt X]@β)" and
"α@β ≠ []"
    unfolding is_reachable_step_def by blast

  obtain w_X where w_X_def: "P ⊢ [Nt X] ⇒* map Tm w_X"
    using assms(1) is_useful_all_derive by blast

  obtain w_α w_β where w_α_def: "P ⊢ α ⇒* map Tm w_α" and w_β_def: "P ⊢
β ⇒* map Tm w_β"
    using assms(1) is_useful_all_derive by blast+
  then have "w_α@w_β ≠ []"
    using ‹α@β ≠ []› by (simp add: assms(2) is_non_nullable_all_derive)

  define f_d where "f_d ≡ is_infinite_derives X α β"
  define f_w where "f_w ≡ is_infinite_words w_X w_α w_β"

  have "P ⊢ S ⇒^? X"
    using assms(1) by (simp add: is_useful_all_def is_useful_def)
  then obtain p s where "P ⊢ [Nt S] ⇒* (p@[Nt X]@s)"
    unfolding is_reachable_from_def by blast
  moreover obtain w_p where w_p_def: "P ⊢ p ⇒* map Tm w_p"
    using assms(1) is_useful_all_derive by blast
  moreover obtain w_s where w_s_def: "P ⊢ s ⇒* map Tm w_s"
    using assms(1) is_useful_all_derive by blast
  ultimately have fromS: "P ⊢ [Nt S] ⇒* (map Tm w_p@[Nt X]@map Tm w_s)"
    by (meson local.derives_concat rtranclp.rtrancl_refl rtranclp_trans)

  have "⋀i. P ⊢ [Nt X] ⇒* f_d i"
    subgoal for i
      apply (induction i; simp_all add: f_d_def)
      apply (meson deriveX local.derives_concat rtranclp.rtrancl_refl
rtranclp_trans)
      done
    done
  moreover have "⋀i. P ⊢ f_d i ⇒* map Tm (f_w i)"
    subgoal for i
      by (induction i; simp add: f_d_def f_w_def w_X_def w_α_def w_β_def
derives_concat)
```

```
    done
  ultimately have "⋀i. P ⊢ [Nt X] ⇒* map Tm (f_w i)"
    using rtranclp_trans by fast
  then have "⋀i. P ⊢ [Nt S] ⇒* (map Tm w_p@map Tm (f_w i)@map Tm w_s)"
    using fromS derives_step by presburger
  then have "⋀i. P ⊢ [Nt S] ⇒* (map Tm (w_p@(f_w i)@w_s))"
    by simp
  moreover define f_w' where f_w'_def: "f_w' = (λi. w_p @ (f_w i) @ w_s)"
  ultimately have "⋀i. P ⊢ [Nt S] ⇒* map Tm (f_w' i)"
    by simp
  then have "⋀i. f_w' i ∈ Lang P S"
    by (simp add: Lang_def)
  then have "range f_w' ⊆ Lang P S"
    by blast

  have "⋀i. length (f_w i) < length (f_w (i+1))"
    subgoal for i
      by (induction i; use f_w_def ‹w_α@w_β ≠ []› in simp)
    done
  then have x: "⋀i. length (f_w' i) < length (f_w' (i+1))"
    by (simp add: f_w'_def)
  then have "⋀i n. 0 < n ⟹ length (f_w' i) < length (f_w' (i+n))"
    subgoal for i n
      apply (induction n, auto)
      apply (metis Suc_lessD add_cancel_left_right gr_zeroI less_trans_Suc)
      done
    done
  then have f_w'_order: "⋀i_1 i_2. i_1 < i_2 ⟹ length (f_w' i_1) < length
(f_w' i_2)"
    using less_imp_add_positive by blast

  then have "inj f_w'"
    unfolding inj_def by (metis nat_neq_iff)

  have "infinite (Lang P S)"
    using ‹range f_w' ⊆ Lang P S› ‹inj f_w'› infinite_iff_countable_subset
by blast
  then show "¬ finite (Lang P S)"
    by simp
qed
```

— Notation only used in this theory.

```
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
≠ [] })"
```

unfolding *is_infinite_def is_reachable_step_def* **by** *(auto simp: pre_star_term)*

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 ≠ [] })"*
using *assms is_infinite_correct is_infinite_check* **by** *blast*

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"
  by (standard; force)

lemma alg_outer_step_direct_eq[simp]:
  "alg_outer_step_direct S t = alg_outer_step_direct' S t"
  by force

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 -
  define R where "R ≡ rel S ∪ {t}"
  define T where "T ≡ ((trans S) - {t}) ∪ direct S t"
  define D where "D ≡ (direct S) (t := {})"
  define I where "I ≡ impl S"
  note defs = R_def T_def D_def I_def

  have R_subst: "rel (alg_inner_pre S t) = R"
    by (simp add: R_def alg_inner_pre_def)
  have T_subst: "trans (alg_inner_pre S t) = T"
    by (simp add: T_def alg_inner_pre_def)
  have D_subst: "direct (alg_inner_pre S t) = D"
    by (simp add: D_def alg_inner_pre_def)
  have I_subst: "impl (alg_inner_pre S t) = I"
    by (simp add: I_def alg_inner_pre_def)
  note substs = R_subst T_subst D_subst I_subst
```

```
  have "rel (alg_inner_post (alg_inner_pre S t) t) = R ∪ {t}"
    unfolding alg_inner_post_def substs
    by (metis (no_types, lifting) R_def R_subst Un_absorb Un_insert_right
alg_state.select_convs(1) alg_state.surjective alg_state.update_convs(2,3,4)
sup_bot.right_neutral)
  then show "rel (alg_outer_step S t) = rel S ∪ {t}"
    by (simp add: substs defs alg_outer_step_def)

  have "trans (alg_inner_post (alg_inner_pre S t) t) = T ∪ snd ‘ { (t',
t'') ∈ I t. t' ∈ R }"
    unfolding alg_inner_post_def substs
    by (metis (no_types, lifting) alg_state.select_convs(2) alg_state.surjective
alg_state.update_convs(2,3,4))
  then show "trans (alg_outer_step S t) = alg_outer_step_trans S t"
    by (simp add: substs defs alg_outer_step_def)

  have "direct (alg_inner_post (alg_inner_pre S t) t) = (λt'. D t' ∪
snd ‘ { (t'2, t'') ∈ I t. t' = t'2 ∧ t' ∉ R })"
    unfolding alg_inner_post_def substs
    using alg_state.select_convs(3) alg_state.surjective alg_state.update_convs(1,2,3,4)
  proof -
    have "∀p. direct (alg_inner_pre S t (|trans := T ∪ snd ‘ {(pa, p).
(pa, p) ∈ I t ∧ pa ∈ R}, direct := λp. D p ∪ snd ‘ {(pb, pa). (pb, pa)
∈ I t ∧ p = pb ∧ p ∉ R}, impl := I(t := {})|)) p = D p ∪ snd ‘ {(pb,
pa). (pb, pa) ∈ I t ∧ p = pb ∧ p ∉ R}"
      by simp
    then show "direct (let r = I t in alg_inner_pre S t (|trans := T ∪
snd ‘ {(pa, p). (pa, p) ∈ r ∧ pa ∈ R}, direct := λp. D p ∪ snd ‘ {(pb,
pa). (pb, pa) ∈ r ∧ p = pb ∧ p ∉ R}, impl := I(t := {})|)) = (λp. D p
∪ snd ‘ {(pb, pa). (pb, pa) ∈ I t ∧ p = pb ∧ p ∉ R})"
      by meson
  qed
  then show "direct (alg_outer_step S t) = alg_outer_step_direct S t"
    by (simp add: substs defs alg_outer_step_def)

  have "impl (alg_inner_post (alg_inner_pre S t) t) = I (t := {})"
    unfolding alg_inner_post_def substs
    by (metis (no_types, lifting) alg_state.select_convs(4) alg_state.surjective
alg_state.update_convs(4))
  then show "impl (alg_outer_step S t) = alg_outer_step_impl S t"
    by (simp add: substs defs alg_outer_step_def)
qed

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 -
  let ?b = "λS. trans S ≠ {}"
  let ?c = "λS. alg_outer_step S (SOME x. x ∈ trans S)"
  have "⋀S. P S ⟹ trans S ≠ {} ⟹ P (alg_outer_step S (SOME x. x
∈ trans S))"
    by (simp add: assms some_in_eq)
  with assms(2) show "P S ⟹ P S'"
    unfolding alg_outer_def using while_option_rule[where b="?b" and
c="?c"] by blast
qed
```

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 -
  define S where "S = alg_state_new P Q T"

  have invR: "(rel S) ⊆ T'"
    by (simp add: S_def alg_state_new_def)

  have invT: "(trans S) ⊆ T'"
    using pre_star_lts_mono[OF assms] pre_star_lts_refl[OF assms] pre_star_lts_singleton[OF
assms]
    by(auto simp add: S_def alg_state_new_def)

  have "⋀q q' X t. (q, X, q') ∈ T' ⟹ t ∈ direct S (q, X, q') ⟹ t
∈ T'"
  proof -
    fix t and q X q'
    assume "(q, X, q') ∈ T'" and t_in: "t ∈ direct S (q, X, q')"
    show "t ∈ T'" proof (cases X)
      case (Nt B)
      then have "direct S (q, X, q') = { (q, Nt A, q') | A. (A, [Nt B])
∈ P ∧ q ∈ Q ∧ q' ∈ Q }"
        by (simp add: S_def alg_state_new_def)
```

```
      then obtain A where t_split: "t = (q, Nt A, q')"
          and "(A, [Nt B]) ∈ P"
          and inQ: "q ∈ Q ∧ q' ∈ Q"
        using prod_cases3 t_in by auto
      moreover have "(q, Nt B, q') ∈ T'"
        using ‹(q, X, q') ∈ T'› Nt by blast
      moreover note assms
      ultimately have "(q, Nt A, q') ∈ T'"
        by (intro pre_star_lts_singleton) (use inQ in blast)+
      then show ?thesis
        by (simp add: t_split)
    next
      case (Tm b)
      then have "direct S (q, X, q') = {}"
        by (simp add: S_def alg_state_new_def)
      then show ?thesis
        using t_in by blast
    qed
  qed
  then have invD: "∀t' ∈ T'. ∀t ∈ direct S t'. t ∈ T'"
    by fast

  have "⋀t t' t''. t ∈ T' ⟹ (t', t'') ∈ impl S t ⟹ t' ∈ T' ⟹
t'' ∈ T'"
  proof -
    fix t t' t''
    assume "t ∈ T'" and "(t', t'') ∈ impl S t" and "t' ∈ T'"
    obtain q q' X₁ where t_split: "t = (q, X₁, q')"
      by (elim prod_cases3)
    show "t'' ∈ T'" proof (cases X₁)
      case (Nt B)
      have "impl S t = {((q', Nt C, q''), (q, Nt A, q'')) |q'' A C.
          (A, [Nt B, Nt C]) ∈ P ∧ q ∈ Q ∧ q' ∈ Q ∧ q'' ∈ Q}"
        by (simp add: S_def t_split Nt alg_state_new_def)
      then obtain q'' A C where t'_split: "t' = (q', Nt C, q'')"
          and t''_split: "t'' = (q, Nt A, q'')" and "(A, [Nt B, Nt C])
∈ P"
          and inQ: "q ∈ Q ∧ q' ∈ Q &   q'' ∈ Q"
        using ‹(t', t'') ∈ impl S t› by force

      note ‹(A, [Nt B, Nt C]) ∈ P› and assms
      moreover have "(q', Nt C, q'') ∈ T'"
        using ‹t' ∈ T'› by (simp add: t'_split)
      moreover have "(q, Nt B, q') ∈ T'"
        using ‹t ∈ T'› by (simp add: t_split Nt)
      ultimately have "(q, Nt A, q'') ∈ T'"
        by (intro pre_star_lts_impl) (use inQ in blast)+
      then show ?thesis
        unfolding t''_split by assumption
```

```
    next
      case (Tm b)
      have "impl S t = {}"
        by (simp add: S_def t_split Tm alg_state_new_def)
      then show ?thesis
        using ‹(t', t'') ∈ impl S t› by simp
    qed
  qed
  then have invI: "∀t ∈ T'. ∀(t', t'') ∈ impl S t. t' ∈ T' ⟶ t''
∈ T'"
    by fast

  from invR invT invD invI show ?thesis
    unfolding pre_star_alg_sub_inv_def S_def by blast
qed

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 -
  note inv[simp] = assms(3)[unfolded pre_star_alg_sub_inv_def]
  have [simp]: "t ∈ T'"
    using assms(2) assms(3) unfolding pre_star_alg_sub_inv_def by blast
  moreover have invi: "∀(t', t'') ∈ impl (alg_outer_step S t) t. t'
∈ T' ⟶ t'' ∈ T'"
    by simp
  moreover have invR: "rel (alg_outer_step S t) ⊆ T'"
    by simp
  moreover have invT: "trans (alg_outer_step S t) ⊆ T'"
    unfolding alg_outer_step_simps(2) alg_outer_step_trans_eq
    using inv invi ‹t ∈ T'› by blast
  moreover have invD: "∀t' ∈ T'. ∀t ∈ direct (alg_outer_step S t) t'.
t ∈ T'"
    unfolding alg_outer_step_simps(3) alg_outer_step_direct_eq using inv
invi ‹t ∈ T'›
    by (metis (no_types, lifting) Un_iff case_prod_conv empty_iff fun_upd_apply
mem_Collect_eq)
  moreover have invI: "∀t₂ ∈ T'. ∀(t', t'') ∈ impl (alg_outer_step
S t) t₂. t' ∈ T' ⟶ t'' ∈ T'"
    by simp
  ultimately show ?thesis
    unfolding pre_star_alg_sub_inv_def by blast
qed

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 -
  note assms' = assms(1,2) assms(3)[unfolded alg_outer_def]
  have "⋀s. pre_star_alg_sub_inv T' s ⟹ trans s ≠ {} ⟹
      pre_star_alg_sub_inv T' (alg_outer_step s (SOME x. x ∈ trans s))"
    by (rule alg_outer_step_inv; use assms someI_ex in fast)
  then show ?thesis
    by (rule while_option_rule[where P="pre_star_alg_sub_inv T'"]) (use
assms' in blast)+
qed

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 -
  have "pre_star_alg_sub_inv T' (alg_state_new P Q T)"
    using assms by (elim alg_state_new_inv)
  with assms have "pre_star_alg_sub_inv T' S'"
    by (intro alg_outer_inv[where S="alg_state_new P Q T" and T'=T'
and S'=S']; simp)
  then show ?thesis
    unfolding pre_star_alg_sub_inv_def by blast
qed
```

7.3.2 Super-Set

```
lemma alg_outer_fixpoint: "alg_outer S = Some S' ⟹ alg_outer S' = Some
S'"
  unfolding alg_outer_def by (metis (lifting) while_option_stop while_option_unfold)

lemma pre_star_alg_trans_empty: "alg_outer S = Some S' ⟹ trans S' =
{}"
  using while_option_stop unfolding alg_outer_def by fast

lemma alg_outer_step_direct: "t ≠ t' ⟹ direct S t' ⊆ direct (alg_outer_step
S t) t'"
  by simp

lemma alg_outer_step_impl: "(impl S) (t := {}) = impl (alg_outer_step
S t)"
  by simp

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)"
  using assms unfolding alg_outer_step_simps alg_outer_step_trans_eq by
blast
```

```
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'"
  using assms unfolding alg_outer_step_simps alg_outer_step_direct_eq
by blast
```

— Everything from *trans* is eventually added to *rel*:

```
lemma pre_star_alg_trans_to_lts:
  assumes "alg_outer S = Some S'"
  shows "trans S ⊆ rel S'"
proof
  fix x
  assume "x ∈ trans S"
  have "x ∈ trans S' ∨ x ∈ rel S'"
    by (rule alg_outer_rule[where P="λS. x ∈ trans S ∨ x ∈ rel S"];
use assms ‹x ∈ trans S› in auto)
  then show "x ∈ rel S'"
    using assms pre_star_alg_trans_empty by blast
qed
```

— If *t* is added to *rel*, then so is *direct(t)*:

```
lemma pre_star_alg_direct_to_lts:
  fixes S₀ :: "('s, 'n, 't) alg_state"
  assumes "alg_outer S₀ = Some S'"
    and "t ∉ rel S₀" and "t ∈ rel S'"
  shows "direct S₀ t ⊆ rel S'"
proof -
  let ?I = "λS. (t ∉ rel S ∧ direct S₀ t ⊆ direct S t) ∨ (direct S₀
t ⊆ rel S ∪ trans S)"
  have "⋀S t. ?I S ⟹ t ∈ trans S ⟹ ?I (alg_outer_step S t)"
  proof -
    fix S :: "('s, 'n, 't) alg_state" and t'
    assume assm1: "(t ∉ rel S ∧ direct S₀ t ⊆ direct S t) ∨ (direct
S₀ t ⊆ rel S ∪ trans S)"
      and assm2: "t' ∈ trans S"

    show "?I (alg_outer_step S t')"
    proof (cases "t = t'")
      case True
      then show ?thesis
        using assm1 by auto
    next
      case False
      consider "t ∉ rel S ∧ direct S₀ t ⊆ direct S t" | "direct S₀ t
⊆ rel S ∪ trans S"
        using assm1 by blast
      then show ?thesis
        by (cases; auto)
```

```
    qed
  qed
  with assms have "?I S'"
    by (elim alg_outer_rule[where P="?I"]) simp+
  then show ?thesis
    using assms pre_star_alg_trans_empty by blast
qed

— If t and t' are added to rel, then so are all t'' from (t', t'') ∈ impl(t):
lemma pre_star_alg_impl_to_lts:
  fixes S₀ :: "('s, 'n, 't) alg_state"
  assumes "alg_outer S₀ = Some S'"
    and "t ∉ rel S₀" and "t' ∉ rel S₀"
    and "(t', t'') ∈ impl S₀ t"
    and "t ∈ rel S'" and "t' ∈ rel S'"
  shows "t'' ∈ rel S'"
proof -
  let ?I = "λS. (t ∉ rel S ∧ (t', t'') ∈ impl S t)
      ∨ (t' ∉ rel S ∧ t'' ∈ direct S t')
      ∨ (t'' ∈ rel S ∪ trans S)"

  have "⋀S x. ?I S ⟹ x ∈ trans S ⟹ ?I (alg_outer_step S x)"
  proof -
    fix S :: "('s, 'n, 't) alg_state" and x
    assume "?I S" and "x ∈ trans S"
    then show "?I (alg_outer_step S x)"
    proof (elim disjE)
      assume assm1: "x ∈ trans S" and assm2: "t ∉ rel S ∧ (t', t'')
∈ impl S t"
      then show "?I (alg_outer_step S x)"
      proof (cases "x = t")
        case True
        then show ?thesis
        proof (cases "t' ∈ rel S ∨ t = t'")
          case True
          with assm2 have "t'' ∈ trans (alg_outer_step S t)"
            by (intro alg_outer_step_impl_to_trans[of t' t'' S t]; simp)
          then show ?thesis
            by (simp add: ‹x = t›)
        next
          case False
          then have "t' ∉ rel S ∪ {t}"
            using False by blast
          then have "t' ∉ rel (alg_outer_step S t)"
            by simp
          moreover with False assm2 have "t'' ∈ direct (alg_outer_step
S t) t'"
            by (intro alg_outer_step_impl_to_direct[of t' t'' S t]; simp)
          ultimately show ?thesis
```

```
            by (simp add: ‹x = t›)
        qed
      next
        case False
        then show ?thesis
          using alg_outer_step_impl assm2 by simp
      qed
    next
      assume assm1: "x ∈ trans S" and assm2: "t' ∉ rel S ∧ t'' ∈ direct
S t'"
      then show "?I (alg_outer_step S x)"
        by (cases "x = t'"; simp)
    next
      assume "x ∈ trans S" and "t'' ∈ rel S ∪ trans S"
      then show "?I (alg_outer_step S x)"
        by force
    qed
  qed
  with assms have "?I S'"
    by (elim alg_outer_rule[where P="?I"]) simp+
  then show ?thesis proof (elim disjE)
    assume "t ∉ rel S' ∧ (t', t'') ∈ impl S' t"
    then show "t'' ∈ rel S'"
      using assms(5) by blast
  next
    assume "t' ∉ rel S' ∧ t'' ∈ direct S' t'"
    moreover have "alg_outer S' = Some S'"
      using assms(1) by (rule alg_outer_fixpoint)
    ultimately show "t'' ∈ rel S'"
      using pre_star_alg_direct_to_lts assms(6) by blast
  next
    assume " t'' ∈ rel S' ∪ trans S'"
    then show "t'' ∈ rel S'"
      using assms(1) pre_star_alg_trans_empty by blast
  qed
qed
```

— Reflexive auto.trans are eventually added to `rel`:

```
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"
  using assms by (simp add: alg_state_new_def)

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'"
  using assms pre_star_alg_new_refl_to_trans pre_star_alg_trans_to_lts
by fast
```

— Lemmas for singleton productions, i.e. $A \to B$ or $A \to b$:

```
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 -
  have "(q, Nt A, q') ∈ direct (alg_state_new P Q T) (q, Nt B, q')"
    using assms by (simp add: alg_state_new_def)
  moreover have "(q, Nt B, q') ∉ rel (alg_state_new P Q T)"
    by (simp add: alg_state_new_def)
  ultimately show "(q, Nt B, q') ∈ rel S' ⟹ (q, Nt A, q') ∈ rel S'"
    using assms(1) pre_star_alg_direct_to_lts by blast
qed

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 -
  define i where "i ≡ (λt. t = (q, Tm b::('n, 't) sym, q') ⟶ t ∈ T)"
  define I :: "('s, 'n, 't) alg_state ⇒ bool"
    where "I ≡ (λS. (∀t ∈ rel S. i t) ∧ (∀t ∈ trans S. i t)
      ∧ (∀t. ∀t' ∈ direct S t. i t') ∧ (∀t. ∀(t', t'') ∈ impl S t.
i t' ∧ i t''))"

  have "I (alg_state_new P Q T)"
    unfolding alg_state_new_def I_def i_def
    by (auto split: sym.splits intro: sym.exhaust)
  moreover have "⋀S t. I S ⟹ t ∈ trans S ⟹ I (alg_outer_step S
t)"
    unfolding I_def i_def alg_outer_step_simps
    by (auto split: sym.splits; blast)
  ultimately have "I S'"
    using assms(1) by (elim alg_outer_rule)
  then show ?thesis
    using assms(2) by (simp add: I_def i_def)
qed

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 -
  have "(q, Tm b, q') ∈ T"
    using assms pre_star_alg_tm_only_from_delta by fast
  then have "(q, Nt A, q') ∈ trans (alg_state_new P Q T)"
```

```
    by (auto simp: alg_state_new_def assms)
  then show ?thesis
    using pre_star_alg_trans_to_lts assms(1) by blast
qed

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'"
  using assms pre_star_alg_singleton_nt_to_lts pre_star_alg_singleton_tm_to_lts
by (cases X; fast)

— Lemmas for dual productions, i.e. A → AB:
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 -
  define S where [simp]: "S ≡ alg_state_new P Q T"
  have "(q, Nt B, q') ∉ rel S" and "(q', Nt C, q'') ∉ rel S"
    by (simp add: alg_state_new_def)+
  moreover have "((q', Nt C, q''), (q, Nt A, q'')) ∈ impl S (q, Nt B,
q')"
    using assms by (simp add: alg_state_new_def)
  moreover have "alg_outer S = Some S'"
    by (simp add: assms(1))
  ultimately show ?thesis
    using assms(3,4) by (elim pre_star_alg_impl_to_lts; force)
qed

lemma pre_star_alg_sup:
  fixes P and T :: "('s, 'n, 't) trans" and q₀
  defines "Q ≡ {q₀} ∪ 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 -
  — If t ∈ T, then t is eventually added to rel:
  have base: "T ⊆ rel S'" and "Q = {q₀} ∪ states_lts T"
  proof
    fix t
    assume "t ∈ T"
    then have "t ∈ trans S"
      by (simp add: S_def alg_state_new_def)
    then show "t ∈ rel S'"
```

```
    using assms(3) pre_star_alg_trans_to_lts by blast
  next
    show "Q = {q0} ∪ states_lts T"
      by (simp add: Q_def)
  qed

  define b where "b ≡ (λT::('s, 'n, 't) trans. ¬ pre_lts P Q T ⊆ T)"
  define c where "c ≡ (λT::('s, 'n, 't) trans. T ∪ pre_lts P Q T)"

  have "⋀t T. Q = {q0} ∪ states_lts T ⟹ T ⊆ rel S' ⟹ t ∈ pre_lts
P Q T ⟹ t ∈ rel S'"
  proof -
    fix T and t
    assume q_reach: "Q = {q0} ∪ states_lts T" "T ⊆ rel S'" and t_src:
"t ∈ pre_lts P Q T"
    then obtain q q' A β where t_split: "t = (q, Nt A, q')" and "(A,
β) ∈ P" and "q' ∈ steps_lts T β q"
      unfolding pre_lts_def by blast
    moreover have q_in: "q ∈ Q ∧ q' ∈ Q"
      using t_src calculation steps_states_lts[of T] unfolding pre_lts_def
q_reach(1)
      using fst_conv by blast
    ultimately consider "β = []" | "∃X. β = [X]" | "∃B C. β = [Nt B,
Nt C]"
      using assms(5)[unfolded CNF1_def] by fast
    then have "(q, Nt A, q') ∈ rel S'" proof (cases)
      case 1
      then have "q = q'"
        using ‹q' ∈ steps_lts T β q› by (simp add: Steps_lts_def)
      moreover have "(A, []) ∈ P"
        using ‹(A, β) ∈ P›[unfolded 1] by assumption
      ultimately show ?thesis
        using assms(2,3) q_in pre_star_alg_refl_to_lts by fast
    next
      case 2
      then obtain X where β_split: "β = [X]"
        by blast
      then have "(q, X, q') ∈ rel S'"
        using ‹q' ∈ steps_lts T β q› ‹T ⊆ rel S'› by (auto simp: Steps_lts_def
Step_lts_def step_lts_def)
      moreover have "(A, [X]) ∈ P"
        using ‹(A, β) ∈ P›[unfolded β_split] by assumption
      ultimately show ?thesis
        using assms(2,3) q_in pre_star_alg_singleton_to_lts by fast
    next
      case 3
      then obtain B C where β_split: "β = [Nt B, Nt C]"
        by blast
      then obtain q'' where  "q' ∈ steps_lts T [Nt C] q''" and "q''
```

```
∈ steps_lts T [Nt B] q"
        using β_split ‹q' ∈ steps_lts T β q› Steps_lts_split by force
      then have "(q, Nt B, q'') ∈ rel S'" and "(q'', Nt C, q') ∈ rel
S'"
        using ‹q' ∈ steps_lts T β q› ‹T ⊆ rel S'› by (auto simp: steps_lts_defs)
      moreover have "(q, Nt B, q'') ∈ T"
        using ‹q'' ∈ steps_lts T [Nt B] q› by (auto simp: steps_lts_defs)
      moreover have "q'' ∈ Q"
        using q_reach(1) steps_states_lts[of T Q q] q_in ‹q'' ∈ steps_lts
T [Nt B] q› by blast
     moreover have "(A, [Nt B, Nt C]) ∈ P"
        using ‹(A, β) ∈ P›[unfolded β_split] by assumption
      ultimately show ?thesis
        using assms(2,3) q_in pre_star_alg_dual_to_lts by fast
    qed
    then show "t ∈ rel S'"
      by (simp add: t_split)
  qed
  moreover have "⋀T. Q = {q0} ∪ states_lts T ⟹ Q = {q0} ∪ states_lts
(T ∪ pre_lts P Q T)"
    using states_pre_lts unfolding states_lts_Un
    by (metis Un_assoc Un_upper2 sup.order_iff)
  ultimately have step: "⋀T. (T ⊆ rel S' ∧ Q = {q0} ∪ states_lts T)
⟹ ¬ pre_lts P Q T ⊆ T
      ⟹ (T ∪ pre_lts P Q T ⊆ rel S' ∧ Q = {q0} ∪ states_lts (T ∪ pre_lts
P Q T))"
    by (smt (verit, del_insts) Un_iff subset_eq)

  note base step
  moreover note assms(4)[unfolded pre_star_lts_def] b_def c_def
  ultimately have "T' ⊆ rel S' ∧ Q = {q0} ∪ states_lts T'"
    by (elim pre_star_lts_rule; use assms in simp)
  then show "T' ⊆ rel S'"
    by simp
qed
```

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"
  by (auto simp: alg_state_m_i_def)

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"
```

```
  using assms by (auto simp: alg_state_m_i_def)

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"
  using assms by (auto simp: alg_state_m_d_def)

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"
  using assms by (auto simp: alg_state_m_d_def)

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"
  using assms by auto

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)"
  unfolding alg_state_comp_def by (intro wf_inv_image) blast

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)"
  unfolding alg_state_fin_inv_def
proof (intro conjI)
  show "finite (rel (alg_outer_step S t))"
    by (simp add: assms[unfolded alg_state_fin_inv_def])
next
  have "{t''. ∃t'. (t', t'') ∈ impl S t ∧ t' ∈ alg_outer_step_lts S
t} ⊆ snd ` impl S t"
    by force
```

```
  moreover have "finite (snd ‘ impl S t)"
    using assms[unfolded alg_state_fin_inv_def] by blast
  ultimately have "finite {t''. ∃t'. (t', t'') ∈ impl S t ∧ t' ∈ alg_outer_step_lts
S t}"
    by (elim finite_subset)
  then show "finite (trans (alg_outer_step S t))"
    using assms[unfolded alg_state_fin_inv_def]
    unfolding alg_outer_step_simps alg_outer_step_trans_eq by blast
next
  have "⋀t'. {t''. (t', t'') ∈ impl S t ∧ t' ∉ alg_outer_step_lts S
t} ⊆ snd ‘ impl S t"
    by force
  moreover have "finite (snd ‘ impl S t)"
    using assms[unfolded alg_state_fin_inv_def] by blast
  ultimately have "⋀t'. finite {t''. (t', t'') ∈ impl S t ∧ t' ∉ alg_outer_step_lts
S t}"
    using finite_subset by blast
  moreover have "⋀t'. finite (((direct S)(t := {})) t')"
    using assms[unfolded alg_state_fin_inv_def] by (auto simp: alg_state_m_d_def)
  ultimately show "∀t'. finite (direct (alg_outer_step S t) t')"
    unfolding alg_outer_step_simps alg_outer_step_direct_eq by blast
next
  have "alg_state_m_d (alg_outer_step S t) ⊆ alg_state_m_d S ∪ fst ‘
impl S t"
    unfolding alg_outer_step_simps alg_state_m_d_def by (auto, force)
  moreover have "finite (alg_state_m_d S ∪ fst ‘ impl S t)"
    using assms[unfolded alg_state_fin_inv_def] by blast
  ultimately show "finite (alg_state_m_d (alg_outer_step S t))"
    using finite_subset by blast
next
  show "∀t'. finite (impl (alg_outer_step S t) t')"
    by (simp add: assms[unfolded alg_state_fin_inv_def])
next
  have "finite (alg_state_m_i S)"
    by (simp add: assms(1)[unfolded alg_state_fin_inv_def])
  moreover have "alg_state_m_i (alg_outer_step S t) ⊆ alg_state_m_i
S"
    using assms(2) by (rule alg_state_m_i_step_weak)
  ultimately show "finite (alg_state_m_i (alg_outer_step S t))"
    by (elim finite_subset)
qed

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))"
  using assms alg_state_fin_inv_step by (metis some_in_eq)

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 -
  have "⋀S t. t ∈ trans S ⟹ alg_state_fin_inv S ⟹ b S ⟹ (alg_outer_step
S t, S) ∈ inv_image alg_state_comp alg_state_measure"
  proof -
    fix S and t
    assume "t ∈ trans S" and inv: "alg_state_fin_inv S" and "b S"

    obtain n1 n2 n3 where n_def: "alg_state_measure S = (n1, n2, n3)"
      using prod_cases3 by blast
    then have n1_def: "n1 = card (alg_state_m_i S) ∧ finite (alg_state_m_i
S)"
        and n2_def: "n2 = card (alg_state_m_d S) ∧ finite (alg_state_m_d
S)"
        and n3_def: "n3 = card (trans S) ∧ finite (trans S)"
      using inv by (simp add: alg_state_measure_def alg_state_fin_inv_def)+

    define S' where "S' ≡ alg_outer_step S t"

    obtain m1 m2 m3 where m_def: "alg_state_measure S' = (m1, m2, m3)"
      using prod_cases3 by blast
    moreover have "alg_state_fin_inv S'"
      using inv ‹t ∈ trans S› alg_state_fin_inv_step unfolding S'_def
b_def by blast
    ultimately have m1_def: "m1 = card (alg_state_m_i S') ∧ finite (alg_state_m_i
S')"
        and m2_def: "m2 = card (alg_state_m_d S') ∧ finite (alg_state_m_d
S')"
        and m3_def: "m3 = card (trans S') ∧ finite (trans S')"
      by (simp add: S'_def alg_state_measure_def alg_state_fin_inv_def)+

    consider (red1) "impl S t ≠ {}"
      | (red2) "impl S t = {} ∧ direct S t ≠ {}"
      | (red3) "impl S t = {} ∧ direct S t = {}"
      by blast
    then have "((m1, m2, m3), (n1, n2, n3)) ∈ alg_state_comp"
    proof (cases)
      case red1
      with ‹t ∈ trans S› have "alg_state_m_i S' ⊂ alg_state_m_i S"
        by (simp add: alg_state_m_intros[where t=t and S=S] S'_def)+
      then have "m1 < n1"
        using m1_def n1_def by (simp add: psubset_card_mono)
      then show ?thesis
        by (simp add: alg_state_comp_def)
    next
      case red2
      with ‹t ∈ trans S› have "alg_state_m_i S' ⊆ alg_state_m_i S"
          and "alg_state_m_d S' ⊂ alg_state_m_d S"
```

```
      by (simp add: alg_state_m_intros[where t=t and S=S] S'_def)+
    then have "m1 ≤ n1" and "m2 < n2"
      using m1_def m2_def n1_def n2_def
      by (simp add: psubset_card_mono card_mono)+
    then show ?thesis
      by (auto simp: alg_state_comp_def)
  next
    case red3
    then have "alg_state_m_i S' ⊆ alg_state_m_i S"
      and "alg_state_m_d S' ⊆ alg_state_m_d S"
      and "trans S' ⊂ trans S"
      using ‹t ∈ trans S› alg_state_m_intros[where t=t and S=S] by
(simp add: S'_def)+
    then have "m1 ≤ n1" and "m2 ≤ n2" and "m3 < n3"
      using m1_def m2_def m3_def n1_def n2_def n3_def
      by (simp add: psubset_card_mono card_mono)+
    then show ?thesis
      by (auto simp: alg_state_comp_def)
  qed
  then show "(alg_outer_step S t, S) ∈ inv_image alg_state_comp alg_state_measure"
    using m_def n_def by (simp add: S'_def)
qed
then have "{(t, s). (alg_state_fin_inv s ∧ b s) ∧ t = c s} ⊆ inv_image
alg_state_comp alg_state_measure"
  unfolding c_def b_def
  by (smt (verit, ccfv_SIG) all_not_in_conv mem_Collect_eq old.prod.case
some_eq_imp subrelI)
with wf_alg_state_comp show ?thesis
  by (rule wf_subset)
qed

lemma alg_outer_terminates:
  assumes "alg_state_fin_inv S"
  shows "∃S'. alg_outer S = Some S'"
  unfolding alg_outer_def
  by (intro wf_while_option_Some; use wf_alg_outer_step alg_state_fin_inv_step'
assms in fast)

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)"
  unfolding alg_state_fin_inv_def
proof (intro conjI)
  show "finite (rel (alg_state_new P Q T))"
    by (simp add: alg_state_new_def)
next
  note assms(3)
  moreover have "finite {(q, Nt A, q) |q A. (A, []) ∈ P ∧ q ∈ Q}"
```

```
    by (rule finite_subset[where B="Q × (Nt ‘ fst ‘ P) × Q"]; use assms
in force)
  moreover have "finite {(q, Nt A, q') |q q' A. ∃a. (A, [Tm a]) ∈ P
∧ (q, Tm a, q') ∈ T ∧ q ∈ Q ∧ q' ∈ Q}"
    by (rule finite_subset[where B="Q × (Nt ‘ fst ‘ P) × Q"]; use assms
in force)
  ultimately show "finite (trans (alg_state_new P Q T))"
    by (simp add: alg_state_new_def)
next
  have "⋀q q' B. finite {(q, Nt A, q') |A. (A, [Nt B]) ∈ P ∧ q ∈ Q ∧
q' ∈ Q}"
    by (rule finite_subset[where B="Q × (Nt ‘ fst ‘ P) × Q"]; use assms
in force)
  then show "∀t. finite (direct (alg_state_new P Q T) t)"
    unfolding alg_state_new_def by (auto split: sym.split)
next
  have "alg_state_m_d (alg_state_new P Q T) ⊆ Q × hd ‘ snd ‘ P × Q"
    unfolding alg_state_new_def alg_state_m_d_def by (auto split: sym.splits)
force
  moreover have "finite (hd ‘ snd ‘ P )"
    using assms(1) by simp
  ultimately show "finite (alg_state_m_d (alg_state_new P Q T))"
    using assms(2) finite_subset by blast
next
  have "⋀t. impl (alg_state_new P Q T) t ⊆ (Q × hd ‘ tl ‘ snd ‘ P ×
Q) × (Q × Nt ‘ fst ‘ P × Q)"
    unfolding alg_state_new_def by (auto split: sym.splits) force+
  moreover have "finite ((Q × hd ‘ tl ‘ snd ‘ P × Q) × (Q × Nt ‘ fst
‘ P × Q))"
    using assms(1,2) by simp
  ultimately show "∀t. finite (impl (alg_state_new P Q T) t)"
    using finite_subset by blast
next
  have "alg_state_m_i (alg_state_new P Q T) ⊆ Q × hd ‘ snd ‘ P × Q"
    unfolding alg_state_new_def alg_state_m_i_def by (auto split: sym.splits)
force
  moreover have "finite (hd ‘ snd ‘ P )"
    using assms(1) by simp
  ultimately show "finite (alg_state_m_i (alg_state_new P Q T))"
    using assms(2) finite_subset by blast
qed
```

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 -
  define Q where "Q ≡ {auto.start M} ∪ states_lts (auto.lts M)"
  have "finite Q"
    using assms(2) by (auto simp add: states_lts_def Q_def)

  define S where "S ≡ alg_state_new P Q (auto.lts M)"
  have "alg_state_fin_inv S"
    using alg_state_new_fin_inv assms(1,2) ‹finite Q› by (simp add: S_def)
  then obtain S' where S'_def: "alg_outer S = Some S'"
    using alg_outer_terminates by blast

  obtain T' where T'_def: "pre_star_lts P Q (auto.lts M) = Some T'"
    using pre_star_lts_terminates assms(1,2) ‹finite Q›
    by (metis Q_def sup_ge2)
  moreover have "rel S' ⊆ T'"
    using S'_def T'_def pre_star_alg_sub unfolding S_def by blast
  moreover have "T' ⊆ rel S'"
    using S'_def T'_def cnf pre_star_alg_sup unfolding S_def Q_def by
fast
  ultimately have "rel S' = T'"
    by simp

  have "pre_star_auto P M = pre_star_code_cnf P M"
    unfolding pre_star_auto_def pre_star_code_cnf_def
    using T'_def S'_def ‹rel S' = T'› unfolding S_def Q_def by simp
  then show ?thesis
    using pre_star_auto_correct assms(1,2) by metis
qed

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.