

Pushdown Automata

Kaan Taskin and Tobias Nipkow

February 6, 2026

Abstract

This entry formalizes pushdown automata and proves their equivalence with context-free grammars. It also shows that acceptance by empty stack and by final state are equivalent.

Contents

1	Pushdown Automata (PDA)	1
1.1	Definitions	1
1.2	Basic Lemmas	3
1.2.1	<i>step</i> and <i>step</i> ₁	3
1.2.2	<i>steps</i>	4
1.2.3	<i>step</i> _n	5
2	Equivalence of Final and Stack Acceptance	12
2.1	Stack Acceptance to Final Acceptance	12
2.2	Final Acceptance to Stack Acceptance	20
3	Equivalence of CFG and PDA	32
3.1	CFG to PDA	32
3.2	PDA to CFG	37

1 Pushdown Automata (PDA)

```
theory Pushdown_Automata
imports Main
begin
```

1.1 Definitions

In the following, we define *pushdown automata* and show some basic properties of them. The formalization is based on the Lean formalization by Leichtfried[2].

We represent the transition function δ by splitting it into two different functions $\delta_1 : Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^*$ and $\delta_2 : Q \times \Gamma \rightarrow Q \times \Gamma^*$, where $\delta_1(q, a, Z) := \delta(q, a, Z)$ and $\delta_2(q, Z) := \delta(q, \epsilon, Z)$.

```

record ('q, 'a, 's) pda = init_state    :: 'q
                        init_symbol    :: 's
                        final_states   :: 'q set
                        delta          :: 'q  $\Rightarrow$  'a  $\Rightarrow$  's  $\Rightarrow$  ('q  $\times$  's list) set
                        delta_eps      :: 'q  $\Rightarrow$  's  $\Rightarrow$  ('q  $\times$  's list) set

```

```

locale pda =
  fixes M :: ('q :: finite, 'a :: finite, 's :: finite) pda
  assumes finite_delta: finite (delta M p a Z)
  and finite_delta_eps: finite (delta_eps M p Z)
begin

```

```

notation delta ( $\delta$ )
notation delta_eps ( $\delta\epsilon$ )

```

```

fun step :: 'q  $\times$  'a list  $\times$  's list  $\Rightarrow$  ('q  $\times$  'a list  $\times$  's list) set where
  step (p, a#w, Z# $\alpha$ ) = {(q, w,  $\beta@$  $\alpha$ ) | q  $\beta$ . (q,  $\beta$ )  $\in$   $\delta$  M p a Z}
                         $\cup$  {(q, a#w,  $\beta@$  $\alpha$ ) | q  $\beta$ . (q,  $\beta$ )  $\in$   $\delta\epsilon$  M p Z}
| step (p, [], Z# $\alpha$ ) = {(q, [],  $\beta@$  $\alpha$ ) | q  $\beta$ . (q,  $\beta$ )  $\in$   $\delta\epsilon$  M p Z}
| step (_, _, []) = {}

```

```

fun step1 :: 'q  $\times$  'a list  $\times$  's list  $\Rightarrow$  'q  $\times$  'a list  $\times$  's list  $\Rightarrow$  bool
  ((_  $\rightsquigarrow$  _) [50, 50] 50) where
  (p1, w1,  $\alpha_1$ )  $\rightsquigarrow$  (p2, w2,  $\alpha_2$ )  $\iff$  (p2, w2,  $\alpha_2$ )  $\in$  step (p1, w1,  $\alpha_1$ )

```

```

definition steps :: 'q  $\times$  'a list  $\times$  's list  $\Rightarrow$  'q  $\times$  'a list  $\times$  's list  $\Rightarrow$  bool
  ((_  $\rightsquigarrow^*$  _) [50, 50] 50) where
  steps  $\equiv$  step1  $\hat{\rightsquigarrow}^*$ 

```

```

inductive stepn :: nat  $\Rightarrow$  'q  $\times$  'a list  $\times$  's list  $\Rightarrow$  'q  $\times$  'a list  $\times$  's list  $\Rightarrow$  bool
where
  refln: stepn 0 (p, w,  $\alpha$ ) (p, w,  $\alpha$ ) |
  stepn: stepn n (p1, w1,  $\alpha_1$ ) (p2, w2,  $\alpha_2$ )  $\implies$  step1 (p2, w2,  $\alpha_2$ ) (p3, w3,  $\alpha_3$ )  $\implies$ 
  stepn (Suc n) (p1, w1,  $\alpha_1$ ) (p3, w3,  $\alpha_3$ )

```

```

abbreviation stepsn ((_ / $\rightsquigarrow'$ (_) / _) [50, 0, 50] 50) where
  c  $\rightsquigarrow$ (n) c'  $\equiv$  stepn n c c'

```

The language accepted by empty stack:

```

definition accept_stack :: 'a list set where
  accept_stack  $\equiv$  {w.  $\exists$  q. (init_state M, w, [init_symbol M])  $\rightsquigarrow^*$  (q, [], [])}

```

The language accepted by final state:

```

definition accept_final :: 'a list set where

```

$accept_final \equiv \{w. \exists q \in final_states\ M. \exists \gamma. (init_state\ M, w, [init_symbol\ M]) \rightsquigarrow^* (q, [], \gamma)\}$

1.2 Basic Lemmas

1.2.1 *step* and *step*₁

lemma *card_trans_step*: $card\ (\delta\ M\ p\ a\ Z) = card\ \{(q, w, \beta @ \alpha) \mid q\ \beta. (q, \beta) \in \delta\ M\ p\ a\ Z\}$

by (*rule* *bij_betw_same_card*[**where** $?f = \lambda(q, \beta). (q, w, \beta @ \alpha)$]) (*auto simp*: *bij_betw_def inj_on_def*)

lemma *card_eps_step*: $card\ (\delta\varepsilon\ M\ p\ Z) = card\ \{(q, w, \beta @ \alpha) \mid q\ \beta. (q, \beta) \in \delta\varepsilon\ M\ p\ Z\}$

by (*rule* *bij_betw_same_card*[**where** $?f = \lambda(q, \beta). (q, w, \beta @ \alpha)$]) (*auto simp*: *bij_betw_def inj_on_def*)

lemma *card_empty_step*: $card\ (step\ (p, [], Z\#\alpha)) = card\ (\delta\varepsilon\ M\ p\ Z)$

by (*rule* *sym*) (*simp add*: *card_eps_step*)

lemma *finite_delta_step*: $finite\ \{(q, w, \beta @ \alpha) \mid q\ \beta. (q, \beta) \in \delta\ M\ p\ a\ Z\}$ (**is finite** $?A$)

using *bij_betw_finite*[*of* $\lambda(q, \beta). (q, w, \beta @ \alpha)\ \delta\ M\ p\ a\ Z\ ?A$] **by** (*fastforce simp add*: *bij_betw_def inj_on_def finite_delta*)

lemma *finite_delta_eps_step*: $finite\ \{(q, w, \beta @ \alpha) \mid q\ \beta. (q, \beta) \in \delta\varepsilon\ M\ p\ Z\}$ (**is finite** $?A$)

using *bij_betw_finite*[*of* $\lambda(q, \beta). (q, w, \beta @ \alpha)\ \delta\varepsilon\ M\ p\ Z\ ?A$] **by** (*fastforce simp add*: *bij_betw_def inj_on_def finite_delta_eps*)

lemma *card_nonempty_step*: $card\ (step\ (p, a\#\ w, Z\#\alpha)) = card\ (\delta\ M\ p\ a\ Z) + card\ (\delta\varepsilon\ M\ p\ Z)$

apply (*simp only*: *step.simps*)

apply (*subst card_trans_step*)

apply (*subst card_eps_step*)

apply (*rule card_Un_disjoint*)

apply (*auto simp*: *finite_delta_step finite_delta_eps_step*)

done

lemma *finite_step*: $finite\ (step\ (p, w, Z\#\alpha))$

by (*cases w*) (*auto simp*: *finite_delta_step finite_delta_eps_step*)

lemma *step1_nonempty_stack*: $(p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \implies \exists Z' \alpha'. \alpha_1 = Z' \#\alpha'$

by (*cases* α_1) *auto*

lemma *step1_empty_stack*: $\neg (p_1, w_1, []) \rightsquigarrow (p_2, w_2, \alpha_2)$

by *simp*

lemma *step1_rule*: $(p_1, w_1, Z\#\alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \iff (\exists \beta. w_2 = w_1 \wedge \alpha_2 =$

$\beta @ \alpha_1 \wedge (p_2, \beta) \in \delta \varepsilon M p_1 Z$
 $(p_2, \beta) \in \delta M p_1 a Z$
by (*cases* w_1) *auto*

lemma *step1_rule_ext*: $(p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \longleftrightarrow (\exists Z' \alpha'. \alpha_1 = Z' \# \alpha' \wedge$
 $((\exists \beta. w_2 = w_1 \wedge \alpha_2 = \beta @ \alpha' \wedge (p_2, \beta) \in \delta \varepsilon M p_1 Z')$
 $(p_2, \beta) \in \delta M p_1 a Z'))$ (**is** $?l \longleftrightarrow ?r$)
apply (*rule iffI*)
apply (*metis step1_nonempty_stack step1_rule*)
apply (*use step1_rule in force*)
done

lemma *step1_stack_app*: $(p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \implies (p_1, w_1, \alpha_1 @ \gamma) \rightsquigarrow$
 $(p_2, w_2, \alpha_2 @ \gamma)$
using *step1_rule_ext* **by** *auto*

1.2.2 steps

lemma *steps_refl*: $(p, w, \alpha) \rightsquigarrow^* (p, w, \alpha)$
by (*simp add: steps_def*)

lemma *steps_trans*: $\llbracket (p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2, \alpha_2); (p_2, w_2, \alpha_2) \rightsquigarrow^* (p_3, w_3, \alpha_3) \rrbracket$
 $\implies (p_1, w_1, \alpha_1) \rightsquigarrow^* (p_3, w_3, \alpha_3)$
unfolding *steps_def* **using** *rtranclp_trans* [**where** $?r = \text{step1}$] **by** *blast*

lemma *step1_steps*: $(p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \implies (p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2,$
 $\alpha_2)$
by (*simp add: steps_def r_into_rtranclp*)

lemma *steps_empty_stack*: $(p_1, w_1, []) \rightsquigarrow^* (p_2, w_2, \alpha_2) \implies p_1 = p_2 \wedge w_1 = w_2$
 $\wedge \alpha_2 = []$
unfolding *steps_def* **using** *converse_rtranclpE2* **by** *fastforce*

lemma *steps_induct2* [*consumes 1*]:
assumes $x1 \rightsquigarrow^* x2$
and $\bigwedge p w \alpha. P (p, w, \alpha) (p, w, \alpha)$
and $\bigwedge p_1 w_1 \alpha_1 p_2 w_2 \alpha_2 p_3 w_3 \alpha_3. (p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \implies (p_2, w_2,$
 $\alpha_2) \rightsquigarrow^* (p_3, w_3, \alpha_3) \implies$
 $P (p_2, w_2, \alpha_2) (p_3, w_3, \alpha_3) \implies P (p_1, w_1, \alpha_1) (p_3, w_3, \alpha_3)$
shows $P x1 x2$
using *assms* [*unfolded steps_def*]
proof (*induction rule: converse_rtranclp_induct*)
case *base* **thus** $?case$ **by** (*metis prod_cases3*)
next
case *step* **thus** $?case$ **by** *simp* (*metis prod_cases3 step1.simps*)
qed

```

lemma steps_induct2_bw[consumes 1, case_names base step]:
  assumes steps x1 x2
    and  $\bigwedge p w \alpha. P (p, w, \alpha) (p, w, \alpha)$ 
    and  $\bigwedge p_1 w_1 \alpha_1 p_2 w_2 \alpha_2 p_3 w_3 \alpha_3. (p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2, \alpha_2) \implies (p_2, w_2, \alpha_2) \rightsquigarrow (p_3, w_3, \alpha_3) \implies$ 
       $P (p_1, w_1, \alpha_1) (p_2, w_2, \alpha_2) \implies P (p_1, w_1, \alpha_1) (p_3, w_3, \alpha_3)$ 
    shows P x1 x2
  using assms[unfolded steps_def]
proof(induction rule: rtranclp_induct)
  case base
    then show ?case by (metis prod_cases3)
  next
    case (step)
    then show ?case by simp (metis prod_cases3 step1.simps)
qed

```

```

lemmas converse_rtranclp_induct3_aux =
  converse_rtranclp_induct [of step1 (ax, ay, az) (bx, by, bz), split_rule]
lemmas steps_induct =
  converse_rtranclp_induct3_aux [of M, folded steps_def, consumes 1, case_names refl step]

```

```

lemma step1_word_app:  $step_1 (p_1, w_1, \alpha_1) (p_2, w_2, \alpha_2) \longleftrightarrow step_1 (p_1, w_1 @ w, \alpha_1) (p_2, w_2 @ w, \alpha_2)$ 
  using step1_rule_ext by simp

```

```

lemma decreasing_word:  $(p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2, \alpha_2) \implies \exists w. w_1 = w @ w_2$ 
  by (induction rule: steps_induct) (use step1_rule_ext in auto)

```

1.2.3 stepn

```

inductive_cases stepn_zeroE[elim!]:  $(p_1, w_1, \alpha_1) \rightsquigarrow (0) (p_2, w_2, \alpha_2)$ 
thm stepn_zeroE
inductive_cases stepn_sucE[elim!]:  $(p_1, w_1, \alpha_1) \rightsquigarrow (Suc\ n) (p_2, w_2, \alpha_2)$ 
thm stepn_sucE

```

```

declare stepn.intros[simp, intro]

```

```

lemma step1_stepn_one:  $(p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \longleftrightarrow (p_1, w_1, \alpha_1) \rightsquigarrow (1) (p_2, w_2, \alpha_2)$ 
  by auto

```

```

lemma stepn_split_last:  $(\exists p' w' \alpha'. (p_1, w_1, \alpha_1) \rightsquigarrow (n) (p', w', \alpha') \wedge (p', w', \alpha') \rightsquigarrow (p_2, w_2, \alpha_2))$ 
   $\longleftrightarrow (p_1, w_1, \alpha_1) \rightsquigarrow (Suc\ n) (p_2, w_2, \alpha_2)$ 
  by auto

```

```

lemma stepn_split_first:  $(\exists p' w' \alpha'. (p_1, w_1, \alpha_1) \rightsquigarrow (p', w', \alpha') \wedge (p', w', \alpha') \rightsquigarrow (p_2, w_2, \alpha_2))$ 

```

$\rightsquigarrow(n) (p_2, w_2, \alpha_2)$ $\longleftrightarrow (p_1, w_1, \alpha_1) \rightsquigarrow(\text{Suc } n) (p_2, w_2, \alpha_2)$ (is ?l \longleftrightarrow ?r)

proof

assume ?l
 then obtain $p' w' \alpha'$ **where** $r1: (p_1, w_1, \alpha_1) \rightsquigarrow (p', w', \alpha')$ **and** $rN: (p', w', \alpha') \rightsquigarrow(n) (p_2, w_2, \alpha_2)$ **by** *blast*
 from rN $r1$ **show** ?r
 by (*induction rule: stepn.induct*) *auto*
next
 show ?r \implies ?l
 by (*induction Suc n (p₁, w₁, α_1) (p₂, w₂, α_2) arbitrary: n p₂ w₂ α_2 rule: stepn.induct*)
 (*metis old.nat.exhaust refl_n step_n stepn_zeroE*)
qed

lemma *stepn_induct*[*consumes 1, case_names basen stepn*]:

assumes $x1 \rightsquigarrow(n) x2$
 and $\bigwedge p w \alpha. P\ 0\ (p, w, \alpha)\ (p, w, \alpha)$
 and $\bigwedge n p_1 w_1 \alpha_1 p_2 w_2 \alpha_2 p_3 w_3 \alpha_3. (p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \implies (p_2, w_2, \alpha_2) \rightsquigarrow(n) (p_3, w_3, \alpha_3) \implies$
 $P\ n\ (p_2, w_2, \alpha_2)\ (p_3, w_3, \alpha_3) \implies P\ (\text{Suc } n)\ (p_1, w_1, \alpha_1)\ (p_3, w_3, \alpha_3)$
 shows $P\ n\ x1\ x2$
using *assms* **proof** (*induction n arbitrary: x1*)
 case 0
 obtain $p_1 w_1 \alpha_1 p_2 w_2 \alpha_2$ **where** [*simp*]: $x1 = (p_1, w_1, \alpha_1)$ **and** [*simp*]: $x2 = (p_2, w_2, \alpha_2)$
 by (*metis prod_cases3*)
 from 0.prem_s(1) **have** $x1 = x2$ **by** *auto*
 with 0.prem_s(2) **show** ?case **by** *simp*
next
 case (*Suc n*)
 obtain $p_1 w_1 \alpha_1 p_2 w_2 \alpha_2$ **where** [*simp*]: $x1 = (p_1, w_1, \alpha_1)$ **and** $x2_def$ [*simp*]: $x2 = (p_2, w_2, \alpha_2)$
 by (*metis prod_cases3*)
 from *Suc.prem_s*(1) **obtain** $p' w' \alpha'$ **where**
 $r1: (p_1, w_1, \alpha_1) \rightsquigarrow (p', w', \alpha')$ **and** $rN: (p', w', \alpha') \rightsquigarrow(n) (p_2, w_2, \alpha_2)$
 using *stepn_split_first*[*of p₁ w₁ α_1 n p₂ w₂ α_2*] **by** *auto*
 have $P\ n\ (p', w', \alpha')\ (p_2, w_2, \alpha_2)$
 using *Suc.IH*[*unfolded x2_def, OF rN Suc.prem_s(2,3)*] **by** *simp*
 then show ?case
 using *Suc.prem_s*(3)[*OF r1 rN*] **by** *simp*
qed

lemma *stepn_trans*:

assumes $(p_1, w_1, \alpha_1) \rightsquigarrow(n) (p_2, w_2, \alpha_2)$
 and $(p_2, w_2, \alpha_2) \rightsquigarrow(m) (p_3, w_3, \alpha_3)$
 shows $(p_1, w_1, \alpha_1) \rightsquigarrow(n+m) (p_3, w_3, \alpha_3)$
using *assms*(2,1) **by** (*induction rule: stepn.induct*) *auto*

lemma *stepn_steps*: $(\exists n. (p_1, w_1, \alpha_1) \rightsquigarrow(n) (p_2, w_2, \alpha_2)) \longleftrightarrow (p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2, \alpha_2)$ (**is** $?l \longleftrightarrow ?r$)

proof

assume $?l$

then obtain n **where** $(p_1, w_1, \alpha_1) \rightsquigarrow(n) (p_2, w_2, \alpha_2)$ **by** *blast*

thus $(p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2, \alpha_2)$

apply (*induction rule: stepn.induct*)

apply (*rule steps_refl*)

apply (*simp add: step1_steps steps_trans*)

done

next

show $?r \implies ?l$

by (*induction rule: steps_induct*) (*use stepn_split_first in blast*)+

qed

lemma *stepn_word_app*: $(p_1, w_1, \alpha_1) \rightsquigarrow(n) (p_2, w_2, \alpha_2) \longleftrightarrow (p_1, w_1 @ w, \alpha_1) \rightsquigarrow(n) (p_2, w_2 @ w, \alpha_2)$ (**is** $?l \longleftrightarrow ?r$)

proof

show $?l \implies ?r$

by (*induction n (p₁, w₁, α₁) (p₂, w₂, α₂) arbitrary: p₂ w₂ α₂ rule: stepn.induct*) (*use step1_word_app in auto*)

next

show $?r \implies ?l$

proof (*induction n (p₁, w₁ @ w, α₁) (p₂, w₂ @ w, α₂) arbitrary: p₂ w₂ α₂ rule: stepn.induct*)

case (*step_n n p₂ w₂ α₂ p₃ α₃ w₃*)

obtain w' **where** $w_2_def: w_2 = w' @ w_3 @ w$

using *decreasing_word[OF step1_steps[OF step_n(3)]]* **by** *blast*

with *step_n(2)* **have** $(p_1, w_1, \alpha_1) \rightsquigarrow(n) (p_2, w' @ w_3, \alpha_2)$ **by** *simp*

moreover from *step_n(3)* w_2_def **have** $(p_2, w' @ w_3, \alpha_2) \rightsquigarrow (p_3, w_3, \alpha_3)$

using *step1_word_app* **by** *force*

ultimately show $?case$ **by** *simp*

qed *simp*

qed

lemma *steps_word_app*: $(p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2, \alpha_2) \longleftrightarrow (p_1, w_1 @ w, \alpha_1) \rightsquigarrow^* (p_2, w_2 @ w, \alpha_2)$

using *stepn_steps stepn_word_app* **by** *metis*

lemma *stepn_not_refl_split_first*:

assumes $(p_1, w_1, \alpha_1) \rightsquigarrow(n) (p_2, w_2, \alpha_2)$

and $(p_1, w_1, \alpha_1) \neq (p_2, w_2, \alpha_2)$

shows $\exists n' p' w' \alpha'. n = \text{Suc } n' \wedge (p_1, w_1, \alpha_1) \rightsquigarrow (p', w', \alpha') \wedge (p', w', \alpha') \rightsquigarrow(n') (p_2, w_2, \alpha_2)$

proof –

from *assms* **have** $n > 0$ **by** *fast*

then obtain n' **where** $n = \text{Suc } n'$
using *not0_implies_Suc* **by** *blast*
with *assms(1)* **show** *?thesis*
using *stepn_split_first* **by** *simp*
qed

lemma *stepn_not_refl_split_last*:

assumes $(p_1, w_1, \alpha_1) \rightsquigarrow^{(n)} (p_2, w_2, \alpha_2)$
and $(p_1, w_1, \alpha_1) \neq (p_2, w_2, \alpha_2)$
shows $\exists n' p' w' \alpha'. n = \text{Suc } n' \wedge (p_1, w_1, \alpha_1) \rightsquigarrow^{(n')} (p', w', \alpha') \wedge (p', w', \alpha') \rightsquigarrow (p_2, w_2, \alpha_2)$

proof –

from *assms* **have** $n > 0$ **by** *fast*
then obtain n' **where** $n = \text{Suc } n'$
using *not0_implies_Suc* **by** *blast*
with *assms(1)* **show** *?thesis*
using *stepn_split_last* **by** *simp*
qed

lemma *steps_not_refl_split_first*:

assumes $(p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2, \alpha_2)$
and $(p_1, w_1, \alpha_1) \neq (p_2, w_2, \alpha_2)$
shows $\exists p' w' \alpha'. (p_1, w_1, \alpha_1) \rightsquigarrow (p', w', \alpha') \wedge (p', w', \alpha') \rightsquigarrow^* (p_2, w_2, \alpha_2)$
using *assms stepn_steps stepn_not_refl_split_first* **by** *metis*

lemma *steps_not_refl_split_last*:

assumes $(p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2, \alpha_2)$
and $(p_1, w_1, \alpha_1) \neq (p_2, w_2, \alpha_2)$
shows $\exists p' w' \alpha'. (p_1, w_1, \alpha_1) \rightsquigarrow^* (p', w', \alpha') \wedge (p', w', \alpha') \rightsquigarrow (p_2, w_2, \alpha_2)$
using *assms stepn_steps stepn_not_refl_split_last* **by** *metis*

lemma *stepn_stack_app*: $(p_1, w_1, \alpha_1) \rightsquigarrow^{(n)} (p_2, w_2, \alpha_2) \implies (p_1, w_1, \alpha_1 @ \beta) \rightsquigarrow^{(n)} (p_2, w_2, \alpha_2 @ \beta)$

by (*induction* n (p_1, w_1, α_1) (p_2, w_2, α_2) *arbitrary*: p_2 w_2 α_2 *rule*: *stepn.induct*)
(*fastforce* *intro*: *step1_stack_app*)**+**

lemma *steps_stack_app*: $(p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2, \alpha_2) \implies (p_1, w_1, \alpha_1 @ \beta) \rightsquigarrow^* (p_2, w_2, \alpha_2 @ \beta)$

using *stepn_steps stepn_stack_app* **by** *metis*

lemma *step1_stack_drop*:

assumes $(p_1, w_1, \alpha_1 @ \gamma) \rightsquigarrow (p_2, w_2, \alpha_2 @ \gamma)$

and $\alpha_1 \neq []$

shows $(p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2)$

proof –

from *assms(1)* **obtain** $Z' \alpha'$ **where** $\alpha_1 @ \gamma = Z' \# \alpha'$ **and**

rule: $(\exists \beta. w_2 = w_1 \wedge \alpha_2 @ \gamma = \beta @ \alpha' \wedge (p_2, \beta) \in \delta \varepsilon M p_1 Z')$

$\vee (\exists a \beta. w_1 = a \# w_2 \wedge \alpha_2 @ \gamma = \beta @ \alpha' \wedge (p_2, \beta) \in \delta M p_1 a Z')$

using *step1_rule_ext* **by** *auto*

from $\alpha_1_ \gamma_ def\ assms(2)$ **obtain** α'' **where** $\alpha_1_ def: \alpha_1 = Z' \# \alpha''$ **and** $\alpha'__ def:$
 $\alpha' = \alpha'' @ \gamma$
using $Cons_eq_append_conv$ [of $Z' \alpha' \alpha_1 \gamma$] **by** $auto$
from $rule\ \alpha'__ def$ **have** $(\exists \beta. w_2 = w_1 \wedge \alpha_2 = \beta @ \alpha'' \wedge (p_2, \beta) \in \delta \varepsilon M p_1 Z')$
 $\vee (\exists a \beta. w_1 = a \# w_2 \wedge \alpha_2 = \beta @ \alpha'' \wedge (p_2, \beta) \in \delta M p_1 a Z')$ **by** $auto$
with $\alpha_1_ def$ **show** $?thesis$
using $step1_rule$ **by** $simp$
qed

lemma $stepn_reads_input:$

assumes $(p_1, a \# w, \alpha_1) \rightsquigarrow(n) (p_2, [], \alpha_2)$
shows $\exists n' k q_1 q_2 \gamma_1 \gamma_2. n = Suc\ n' \wedge k \leq n' \wedge (p_1, a \# w, \alpha_1) \rightsquigarrow(k) (q_1, a$
 $\# w, \gamma_1) \wedge$
 $(q_1, a \# w, \gamma_1) \rightsquigarrow (q_2, w, \gamma_2) \wedge (q_2, w, \gamma_2) \rightsquigarrow(n'-k) (p_2, [], \alpha_2)$
using $assms$ **proof** ($induction\ n\ (p_1, a \# w, \alpha_1)\ (p_2, [] :: 'a\ list, \alpha_2)$ *arbitrary:* p_1
 α_1 *rule:* $stepn_induct$)
case ($stepn\ n\ p_1\ \alpha_1\ p'\ w'\ \alpha'$)
from $stepn(1)$ **have** $case_dist: w' = w \vee w' = a \# w$ (**is** $?l \vee ?r$)
using $step1_rule_ext$ **by** $auto$
show $?case$ **proof** ($rule\ disjE$ [$OF\ case_dist$])
assume $l: ?l$

from $l\ stepn(1)$ **have** $step_1\ (p_1, a \# w, \alpha_1)\ (p', w, \alpha')$ **by** $simp$

moreover from $l\ stepn(2)$ **have** $(p', w, \alpha') \rightsquigarrow(n) (p_2, [], \alpha_2)$ **by** $simp$

ultimately show $?case$ **by** $fastforce$

next

assume $r: ?r$
obtain $n' k q_1 q_2 \gamma_1 \gamma_2$ **where** $IH1: n = Suc\ n'$ **and** $IH2: k \leq n'$ **and**
 $IH3: (p', a \# w, \alpha') \rightsquigarrow(k) (q_1, a \# w, \gamma_1)$ **and** $IH4: (q_1, a \# w, \gamma_1) \rightsquigarrow$
 (q_2, w, γ_2) **and**
 $IH5: (q_2, w, \gamma_2) \rightsquigarrow(n'-k) (p_2, [], \alpha_2)$
using $stepn(3)$ [$OF\ r$] **by** $blast$

from $IH1\ IH2$ **have** $Suc\ k \leq n$ **by** $simp$

moreover from $stepn(1)\ r\ IH3$ **have** $(p_1, a \# w, \alpha_1) \rightsquigarrow(Suc\ k) (q_1, a \# w,$
 $\gamma_1)$
using $stepn_split_first$ **by** $blast$

moreover from $IH1\ IH5$ **have** $stepn\ (n - Suc\ k)\ (q_2, w, \gamma_2)\ (p_2, [], \alpha_2)$ **by**
 $simp$

ultimately show $?case$

using $IH4$ **by** $metis$

qed

qed

lemma *split_word*:

$(p_1, w @ w', \alpha_1) \rightsquigarrow(n) (p_2, [], \alpha_2) \implies \exists k q \gamma. k \leq n \wedge (p_1, w, \alpha_1) \rightsquigarrow(k) (q, [], \gamma) \wedge (q, w', \gamma) \rightsquigarrow(n-k) (p_2, [], \alpha_2)$

proof (*induction w arbitrary: n p1 alpha1*)

case (*Cons a w*)

from *Cons(2)* **obtain** $n' k q_1 q_2 \gamma_1 \gamma_2$ **where** $n_def: n = \text{Suc } n'$ **and** $k_lesseq_n': k \leq n'$ **and** $stepk: (p_1, a \# (w @ w'), \alpha_1) \rightsquigarrow(k) (q_1, a \# (w @ w'), \gamma_1)$ **and**
 $step1: (q_1, a \# (w @ w'), \gamma_1) \rightsquigarrow (q_2, w @ w', \gamma_2)$ **and** $stepnk: (q_2, w @ w', \gamma_2) \rightsquigarrow(n'-k) (p_2, [], \alpha_2)$

using *stepn_reads_input*[*of n p1 a w @ w' alpha1 p2 alpha2*] **by** *auto*

obtain $k'' q \gamma$ **where** $k''_lesseq_n'k: k'' \leq n'-k$ **and** $stepk'': (q_2, w, \gamma_2) \rightsquigarrow(k'')$
 $(q, [], \gamma)$ **and** $stepn'kk'': (q, w', \gamma) \rightsquigarrow(n'-k-k'')$ $(p_2, [], \alpha_2)$

using *Cons.IH*[*OF stepnk*] **by** *blast*

from *stepk step1* **have** *stepSuck: stepn (Suc k) (p1, a # w, alpha1) (q2, w, gamma2)*

using *stepn_word_app*[*of Suc k p1 a # w alpha1 q2 w gamma2 w'*] **by** *simp*

have $(p_1, a \# w, \alpha_1) \rightsquigarrow(\text{Suc } k + k'') (q, [], \gamma)$

using *stepn_trans*[*OF stepSuck stepk''*].

moreover from n_def *stepn'kk''* **have** $(q, w', \gamma) \rightsquigarrow(n - (\text{Suc } k + k'')) (p_2, [], \alpha_2)$ **by** *simp*

moreover from n_def k_lesseq_n' $k''_lesseq_n'k$ **have** $\text{Suc } k + k'' \leq n$ **by** *simp*

ultimately show *?case* **by** *blast*

qed *fastforce*

lemma *split_stack*:

$stepn\ n\ (p_1, w_1, \alpha_1 @ \beta_1) (p_2, [], []) \implies \exists p' m_1 m_2 y y'. w_1 = y @ y' \wedge m_1 + m_2 = n$

$\wedge (p_1, y, \alpha_1) \rightsquigarrow(m_1) (p', [], []) \wedge (p', y', \beta_1)$

$\rightsquigarrow(m_2) (p_2, [], [])$

proof (*induction n arbitrary: p1 w1 alpha1*)

case (*Suc n*)

show *?case* **proof** (*cases alpha1*)

case *Nil*

from *Nil* **have** *stepn 0 (p1, [], alpha1) (p1, [], [])* **by** *simp*

moreover from *Suc.prem Nil* **have** *stepn (Suc n) (p1, w1, beta1) (p2, [], [])* **by** *simp*

ultimately show *?thesis* **by** *force*

next

case (*Cons Z alpha*)

with *Suc.prem* **obtain** $p' w' \alpha'$ **where** $r1: step1 (p_1, w_1, Z \# \alpha @ \beta_1) (p', w', \alpha')$ **and** $rN: stepn\ n\ (p', w', \alpha') (p_2, [], [])$

using *stepn_split_first*[*of p1 w1 Z # alpha @ beta1 n p2 [] []*] **by** *auto*

from $r1$ **have** $rule: (\exists \beta. w' = w_1 \wedge \alpha' = \beta @ \alpha @ \beta_1 \wedge (p', \beta) \in \delta \varepsilon M p_1 Z) \vee (\exists a \beta. w_1 = a \# w' \wedge \alpha' = \beta @ \alpha @ \beta_1 \wedge (p', \beta) \in \delta M p_1 a Z)$ (**is ?l**
 $\vee ?r$)
using $step1_rule$ **by** $blast$
show $?thesis$ **proof** ($rule\ disjE[OF\ rule]$)
assume $?l$
then obtain β **where** $w1_def: w_1 = w'$ **and** $\alpha'_def: \alpha' = \beta @ \alpha @ \beta_1$ **and**
 $e: (p', \beta) \in \delta \varepsilon M p_1 Z$ **by** $blast$
from $rN\ \alpha'_def$ **have** $rN2: stepn\ n\ (p', w', (\beta @ \alpha) @ \beta_1)\ (p_2, [], [])$ **by**
 $simp$
obtain $p''\ m_1\ m_2\ y\ y'$ **where** $w'_def: w' = y @ y'$ **and** $m1_m2_n: m_1 +$
 $m_2 = n$
and $rm1: stepn\ m_1\ (p', y, \beta @ \alpha)\ (p'', [], [])$ **and** $rm2: stepn\ m_2\ (p'', y',$
 $\beta_1)\ (p_2, [], [])$
using $Suc.IH[OF\ rN2]$ **by** $blast$
from e **have** $s1: step1\ (p_1, y, Z\#\alpha)\ (p', y, \beta @ \alpha)$
using $step1_rule$ **by** $blast$

from $w1_def\ w'_def$ **have** $w_1 = y @ y'$ **by** $simp$

moreover from $m1_m2_n$ **have** $Suc\ m_1 + m_2 = Suc\ n$ **by** $simp$

moreover from $s1\ rm1\ Cons$ **have** $stepn\ (Suc\ m_1)\ (p_1, y, \alpha_1)\ (p'', [], [])$
using $stepn_split_first$ **by** $blast$

ultimately show $?thesis$
using $rm2$ **by** $metis$

next
assume $?r$
then obtain $a\ \beta$ **where** $w1_def: w_1 = a \# w'$ **and** $\alpha'_def: \alpha' = \beta @ \alpha @$
 β_1 **and** $tr: (p', \beta) \in \delta M p_1 a Z$ **by** $blast$
from $rN\ \alpha'_def$ **have** $rN2: stepn\ n\ (p', w', (\beta @ \alpha) @ \beta_1)\ (p_2, [], [])$ **by**
 $simp$
obtain $p''\ m_1\ m_2\ y\ y'$ **where** $w'_def: w' = y @ y'$ **and** $m1_m2_n: m_1 +$
 $m_2 = n$
and $rm1: stepn\ m_1\ (p', y, \beta @ \alpha)\ (p'', [], [])$ **and** $rm2: stepn\ m_2\ (p'', y',$
 $\beta_1)\ (p_2, [], [])$
using $Suc.IH[OF\ rN2]$ **by** $blast$
from tr **have** $s1: step1\ (p_1, a\#y, Z\#\alpha)\ (p', y, \beta @ \alpha)$ **by** $simp$

from $w1_def\ w'_def$ **have** $w_1 = (a \# y) @ y'$ **by** $simp$

moreover from $m1_m2_n$ **have** $Suc\ m_1 + m_2 = Suc\ n$ **by** $simp$

moreover from $s1\ rm1\ Cons$ **have** $stepn\ (Suc\ m_1)\ (p_1, a\#y, \alpha_1)\ (p'', [], [])$
using $stepn_split_first$ **by** $blast$

ultimately show $?thesis$
using $rm2$ **by** $metis$

```

    qed
  qed
qed blast

end

end

```

2 Equivalence of Final and Stack Acceptance

2.1 Stack Acceptance to Final Acceptance

Starting from a PDA that accepts by empty stack we construct an equivalent PDA that accepts by final state, following Kozen [1].

```

theory Stack_To_Final_PDA
imports Pushdown_Automata
begin

```

```

datatype 'q st_extended = Old_st 'q | New_init | New_final
datatype 's sym_extended = Old_sym 's | New_sym

```

```

lemma inj_Old_sym: inj Old_sym
by (meson injI sym_extended.inject)

```

```

instance st_extended :: (finite) finite
proof

```

```

  have *: UNIV = {t.  $\exists q. t = \text{Old\_st } q$ }  $\cup$  {New_init, New_final}
    by auto (metis st_extended.exhaust)
  show finite (UNIV :: 'a st_extended set)
    by (simp add: * full_SetCompr_eq)

```

```

qed

```

```

instance sym_extended :: (finite) finite
proof

```

```

  have *: UNIV = {t.  $\exists s. t = \text{Old\_sym } s$ }  $\cup$  {New_sym}
    by auto (metis sym_extended.exhaust)
  show finite (UNIV :: 'a sym_extended set)
    by (simp add: * full_SetCompr_eq)

```

```

qed

```

```

context pda begin

```

```

fun final_of_stack_delta :: 'q st_extended  $\Rightarrow$  'a  $\Rightarrow$  's sym_extended  $\Rightarrow$  ('q st_extended
 $\times$  's sym_extended list) set where
  final_of_stack_delta (Old_st q) a (Old_sym Z) = ( $\lambda(p, \alpha). (\text{Old\_st } p, \text{map}$ 
  Old_sym  $\alpha)$ ) ' ( $\delta M q a Z$ )
| final_of_stack_delta _ _ _ = {}

```

We slight modify the transition function from Kozen's proof to simplify

the formalization (see *stack_to_final_pda_last_step*):

```
fun final_of_stack_delta_eps :: 'q st_extended  $\Rightarrow$  's sym_extended  $\Rightarrow$  ('q st_extended
 $\times$  's sym_extended list) set where
  final_of_stack_delta_eps (Old_st q) (Old_sym Z) = ( $\lambda(p, \alpha).$  (Old_st p, map
  Old_sym  $\alpha$ )) ' ( $\delta\varepsilon$  M q Z)
| final_of_stack_delta_eps New_init New_sym = {(Old_st (init_state M), [Old_sym
  (init_symbol M), New_sym])}
| final_of_stack_delta_eps (Old_st q) New_sym = {(New_final, [])}
| final_of_stack_delta_eps _ _ = {}
```

```
definition final_of_stack_pda :: ('q st_extended, 'a, 's sym_extended) pda where
  final_of_stack_pda  $\equiv$  ( $\lfloor$  init_state = New_init, init_symbol = New_sym, fi-
  nal_states = {New_final},
  delta = final_of_stack_delta, delta_eps = final_of_stack_delta_eps
   $\rfloor$ )
```

lemma pda_final_of_stack: pda final_of_stack_pda

proof (standard, goal_cases)

case (1 p x z)

have finite (final_of_stack_delta p x z)

by (induction p x z rule: final_of_stack_delta.induct) (auto simp: finite_delta)

then show ?case

by (simp add: final_of_stack_pda_def)

next

case (2 p z)

have finite (final_of_stack_delta_eps p z)

by (induction p z rule: final_of_stack_delta_eps.induct) (auto simp: finite_delta_eps)

then show ?case

by (simp add: final_of_stack_pda_def)

qed

lemma final_of_stack_pda_trans:

$(p, \beta) \in \delta$ M q a Z \longleftrightarrow

(Old_st p, map Old_sym β) $\in \delta$ final_of_stack_pda (Old_st q) a (Old_sym Z)

by (auto simp: final_of_stack_pda_def inj_map_eq_map[OF inj_Old_sym])

lemma final_of_stack_pda_eps:

$(p, \beta) \in \delta\varepsilon$ M q Z \longleftrightarrow (Old_st p, map Old_sym β) $\in \delta\varepsilon$ final_of_stack_pda (Old_st q) (Old_sym Z)

by (auto simp: final_of_stack_pda_def inj_map_eq_map[OF inj_Old_sym])

lemma final_of_stack_pda_step:

$(p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \longleftrightarrow$

pda.step1 final_of_stack_pda (Old_st p₁, w₁, map Old_sym α_1) (Old_st p₂, w₂, map Old_sym α_2) (is ?l \longleftrightarrow ?r)

proof

assume ?l

then obtain Z α where α_1 _def: $\alpha_1 = Z \# \alpha$ and rule: $(\exists \beta. w_2 = w_1 \wedge \alpha_2 =$

$\beta @ \alpha \wedge (p_2, \beta) \in \delta \varepsilon M p_1 Z$
 $\vee (\exists a \beta. w_1 = a \# w_2 \wedge \alpha_2 = \beta @ \alpha \wedge (p_2, \beta) \in \delta M p_1 a Z)$
using *step1_rule_ext* **by** *auto*
from *rule* **have** $(\exists \beta. w_2 = w_1 \wedge \text{map Old_sym } \alpha_2 = \text{map Old_sym } \beta @ \text{map Old_sym } \alpha \wedge (\text{Old_st } p_2, \text{map Old_sym } \beta) \in \delta \varepsilon \text{final_of_stack_pda } (\text{Old_st } p_1) (\text{Old_sym } Z))$
 $\vee (\exists a \beta. w_1 = a \# w_2 \wedge \text{map Old_sym } \alpha_2 = \text{map Old_sym } \beta @ \text{map Old_sym } \alpha \wedge (\text{Old_st } p_2, \text{map Old_sym } \beta) \in \delta \text{final_of_stack_pda } (\text{Old_st } p_1) a (\text{Old_sym } Z))$
using *final_of_stack_pda_trans final_of_stack_pda_eps* **by** *auto*
hence $(\exists \beta. w_2 = w_1 \wedge \text{map Old_sym } \alpha_2 = \beta @ \text{map Old_sym } \alpha \wedge (\text{Old_st } p_2, \beta) \in \delta \varepsilon \text{final_of_stack_pda } (\text{Old_st } p_1) (\text{Old_sym } Z))$
 $\vee (\exists a \beta. w_1 = a \# w_2 \wedge \text{map Old_sym } \alpha_2 = \beta @ \text{map Old_sym } \alpha \wedge (\text{Old_st } p_2, \beta) \in \delta \text{final_of_stack_pda } (\text{Old_st } p_1) a (\text{Old_sym } Z))$
by *blast*
with α_1_def **show** *?r*
using *pda.step1_rule[OF pda_final_of_stack]* **by** *simp*
next
assume *?r*
then obtain $Z \alpha$ **where** *map_alpha1_def*: $\text{map Old_sym } \alpha_1 = \text{Old_sym } Z \# \text{map Old_sym } \alpha$ **and**
rule: $(\exists \beta. w_2 = w_1 \wedge \text{map Old_sym } \alpha_2 = \beta @ \text{map Old_sym } \alpha \wedge (\text{Old_st } p_2, \beta) \in \delta \varepsilon \text{final_of_stack_pda } (\text{Old_st } p_1) (\text{Old_sym } Z))$
 $\vee (\exists a \beta. w_1 = a \# w_2 \wedge \text{map Old_sym } \alpha_2 = \beta @ \text{map Old_sym } \alpha \wedge (\text{Old_st } p_2, \beta) \in \delta \text{final_of_stack_pda } (\text{Old_st } p_1) a (\text{Old_sym } Z))$
using *pda.step1_rule_ext[OF pda_final_of_stack]* **by** *auto*
from *map_alpha1_def* **have** α_1_def : $\alpha_1 = Z \# \alpha$
by (*metis list.inj_map_strong list.simps(9) sym_extended.inject*)
from *rule* **have** $(\exists \beta. w_2 = w_1 \wedge \text{map Old_sym } \alpha_2 = \text{map Old_sym } \beta @ \text{map Old_sym } \alpha \wedge (\text{Old_st } p_2, \text{map Old_sym } \beta) \in \delta \varepsilon \text{final_of_stack_pda } (\text{Old_st } p_1) (\text{Old_sym } Z))$
 $\vee (\exists a \beta. w_1 = a \# w_2 \wedge \text{map Old_sym } \alpha_2 = \text{map Old_sym } \beta @ \text{map Old_sym } \alpha \wedge (\text{Old_st } p_2, \text{map Old_sym } \beta) \in \delta \text{final_of_stack_pda } (\text{Old_st } p_1) a (\text{Old_sym } Z))$
using *append_eq_map_conv[where ?f = Old_sym]* **by** *metis*
hence $(\exists \beta. w_2 = w_1 \wedge \alpha_2 = \beta @ \alpha \wedge (p_2, \beta) \in \delta \varepsilon M p_1 Z)$
 $\vee (\exists a \beta. w_1 = a \# w_2 \wedge \alpha_2 = \beta @ \alpha \wedge (p_2, \beta) \in \delta M p_1 a Z)$
using *final_of_stack_pda_trans final_of_stack_pda_eps* **by** (*metis list.inj_map_strong sym_extended.inject map_append*)
with α_1_def **show** *?l*
using *step1_rule* **by** *simp*
qed

abbreviation $\alpha_with_new :: 's \text{ list} \Rightarrow 's \text{ sym_extended list}$ **where**
 $\alpha_with_new \alpha \equiv \text{map Old_sym } \alpha @ [\text{New_sym}]$

lemma *final_of_stack_pda_step1_drop*:
assumes *pda.step1 final_of_stack_pda* $(\text{Old_st } p_1, w_1, \alpha_with_new \alpha_1)$

(*Old_st* p_2 , w_2 , α_with_new α_2)

shows $(p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2)$

proof –

from *assms* **obtain** Z α **where** $\alpha_1_with_new_def: \alpha_with_new$ $\alpha_1 = Z \# \alpha$

and

rule: $(\exists \beta. w_2 = w_1 \wedge \alpha_with_new$ $\alpha_2 = \beta@ \alpha \wedge (Old_st$ $p_2, \beta) \in final_of_stack_delta_eps$ $(Old_st$ $p_1) Z)$

$\vee (\exists a \beta. w_1 = a \# w_2 \wedge \alpha_with_new$ $\alpha_2 = \beta@ \alpha \wedge (Old_st$ $p_2, \beta) \in$ $final_of_stack_delta$ $(Old_st$ $p_1) a Z)$

using *pda.step1_rule_ext*[*OF pda_final_of_stack*] **by** (*auto simp: final_of_stack_pda_def*)

from *rule* **have** $Z \neq New_sym$

by (*induction Old_st p1 Z rule: final_of_stack_delta_eps.induct*) *auto*

with $\alpha_1_with_new_def$ **have** *map Old_sym* $\alpha_1 \neq []$ **by** *auto*

with *assms* **have** *pda.step1 final_of_stack_pda* $(Old_st$ p_1, w_1, map *Old_sym* $\alpha_1)$

(*Old_st* p_2 , w_2 , *map Old_sym* α_2)

using *pda.step1_stack_drop*[*OF pda_final_of_stack*] **by** *blast*

thus *?thesis*

using *final_of_stack_pda_step* **by** *simp*

qed

lemma *final_of_stack_pda_from_old*:

assumes *pda.step1 final_of_stack_pda* $(Old_st$ $p_1, w_1, \alpha_1) (p_2, w_2, \alpha_2)$

shows $(\exists p_2'. p_2 = Old_st$ $p_2') \vee p_2 = New_final$

proof –

from *assms* **obtain** Z α **where**

$(\exists \beta. w_2 = w_1 \wedge \alpha_2 = \beta@ \alpha \wedge (p_2, \beta) \in final_of_stack_delta_eps$ $(Old_st$ $p_1) Z)$

$\vee (\exists a \beta. w_1 = a \# w_2 \wedge \alpha_2 = \beta@ \alpha \wedge (p_2, \beta) \in final_of_stack_delta$ $(Old_st$ $p_1) a Z)$

using *pda.step1_rule_ext*[*OF pda_final_of_stack*] **by** (*auto simp: final_of_stack_pda_def*) +

thus *?thesis*

by (*induction Old_st p1 Z rule: final_of_stack_delta_eps.induct*) *auto*

qed

lemma *final_of_stack_pda_no_step_final*:

$\neg pda.step1$ *final_of_stack_pda* $(New_final, w_1, \alpha_1) (p, w_2, \alpha_2)$

apply (*cases* α_1)

apply (*simp add: pda.step1_empty_stack*[*OF pda_final_of_stack*])

apply (*use pda.step1_rule*[*OF pda_final_of_stack*] *final_of_stack_pda_def* **in** *simp*)

done

lemma *final_of_stack_pda_from_oldn*:

assumes *pda.steps final_of_stack_pda* $(Old_st$ $p_1, w_1, \alpha_1) (p_2, w_2, \alpha_2)$

shows $\exists q'. p_2 = Old_st$ $q' \vee p_2 = New_final$

by (*induction* $(Old_st$ $p_1, w_1, \alpha_1) (p_2, w_2, \alpha_2)$ *arbitrary: p2 w2 alpha2 rule: pda.steps_induct2_bw*[*OF pda_final_of_stack*])

(use *assms final_of_stack_pda_from_old final_of_stack_pda_no_step_final* **in**

blast)+

lemma *final_of_stack_pda_to_old*:

assumes *pda.step1 final_of_stack_pda* (p_1, w_1, α_1) (*Old_st* p_2, w_2, α_2)

shows $(\exists q'. p_1 = \text{Old_st } q') \vee p_1 = \text{New_init}$

using *assms final_of_stack_pda_no_step_final* **by** (*metis st_extended.exhaust*)

lemma *final_of_stack_pda_bottom_elem*:

assumes *pda.steps final_of_stack_pda* (*Old_st* $p_1, w_1, \alpha_with_new \alpha_1$)

(*Old_st* p_2, w_2, γ)

shows $\exists \alpha. \gamma = \alpha_with_new \alpha$

using *assms proof* (*induction* (*Old_st* $p_1, w_1, \alpha_with_new \alpha_1$) (*Old_st* $p_2, w_2,$

γ) *arbitrary*: $p_2 w_2 \gamma$

rule: *pda.steps_induct2_bw*[*OF pda_final_of_stack*])

case ($\exists p_2 w_2 \alpha_2 w_3 \alpha_3 p_3$)

obtain p_2' **where** *p2_def*: $p_2 = \text{Old_st } p_2'$

using *final_of_stack_pda_from_oldn*[*OF* $\exists(1)$] *final_of_stack_pda_to_old*[*OF* $\exists(2)$] **by** *blast*

with $\exists(1, \exists)$ **have** *alpha2_def*: $\exists \alpha. \alpha_2 = \alpha_with_new \alpha$ **by** *simp*

from $\exists(2)$ [*unfolded p2_def*] **obtain** $Z \alpha$ **where** *alpha2_split*: $\alpha_2 = Z \# \alpha$ **and** *rule*:

($\exists \beta. w_3 = w_2 \wedge \alpha_3 = \beta @ \alpha \wedge (\text{Old_st } p_3, \beta) \in \text{final_of_stack_delta_eps}$ (*Old_st* p_2') Z)

$\vee (\exists a \beta. w_2 = a \# w_3 \wedge \alpha_3 = \beta @ \alpha \wedge (\text{Old_st } p_3, \beta) \in \text{final_of_stack_delta}$ (*Old_st* p_2') $a Z$)

using *pda.step1_rule_ext*[*OF pda_final_of_stack*] **by** (*auto simp: final_of_stack_pda_def*)

from *rule* **have** $\exists Z'. Z = \text{Old_sym } Z'$

by (*induction* *Old_st* $p_2' Z$ *rule: final_of_stack_delta_eps.induct*) *auto*

with *alpha2_def alpha2_split* **have** $\exists \gamma. \alpha = \alpha_with_new \gamma$

by (*metis hd_append list.sel(1,3) map_tl sym_extended.simps(3) tl_append_if*)

with *rule* **show** *?case*

by (*induction* *Old_st* $p_2' Z$ *rule: final_of_stack_delta_eps.induct, auto*) (*metis map_append*)+

qed (*rule assms, blast*)

lemma *final_of_stack_pda_stepn*:

(p_1, w_1, α_1) $\rightsquigarrow(n)$ (p_2, w_2, α_2) \longleftrightarrow

pda.stepn final_of_stack_pda n (*Old_st* $p_1, w_1, \alpha_with_new \alpha_1$) (*Old_st* $p_2, w_2, \alpha_with_new \alpha_2$) (**is** $?l \longleftrightarrow ?r$)

proof

show $?l \implies ?r$

proof (*induction* n (p_1, w_1, α_1) (p_2, w_2, α_2) *arbitrary*: $p_2 w_2 \alpha_2$ *rule: stepn.induct*)

case (*stepn* $n p_2 w_2 \alpha_2 p_3 w_3 \alpha_3$)

from *stepn(3)* **have** *pda.step1 final_of_stack_pda* (*Old_st* $p_2, w_2, \text{map Old_sym}$ α_2) (*Old_st* $p_3, w_3, \text{map Old_sym}$ α_3)

using *final_of_stack_pda_step* **by** *simp*

hence *pda.step1 final_of_stack_pda* (*Old_st* $p_2, w_2, \alpha_with_new \alpha_2$) (*Old_st* $p_3, w_3, \alpha_with_new \alpha_3$)

using *pda.step1_stack_app*[*OF pda_final_of_stack*] **by** *simp*

with *stepn(2)* **show** *?case*

by (simp add: pda.stepn[OF pda_final_of_stack])
 qed (simp add: pda.refl_n[OF pda_final_of_stack])
 next
 assume r: ?r thus ?l
 proof (induction n (Old_st p₁, w₁, α_with_new α₁) (Old_st p₂, w₂, α_with_new α₂)
 arbitrary: p₂ w₂ α₂ rule: pda.stepn.induct[OF pda_final_of_stack])
 case (3 n p₂ w₂ α₂ w₃ p₃ α₃)
 from 3(1) have steps_3_1: pda.steps final_of_stack_pda (Old_st p₁, w₁,
 α_with_new α₁) (p₂, w₂, α₂)
 using pda.stepn_steps[OF pda_final_of_stack] by blast
 obtain p₂' where p₂_def: p₂ = Old_st p₂'
 using final_of_stack_pda_from_oldn[OF steps_3_1] final_of_stack_pda_to_old[OF
 3(3)] by blast
 with steps_3_1 obtain γ where α₂_def: α₂ = map Old_sym γ @ [New_sym]
 using final_of_stack_pda_bottom_elem by blast

 with p₂_def 3(1,2) have pda.stepn M n (p₁, w₁, α₁) (p₂', w₂, γ) by simp

 moreover from p₂_def α₂_def 3(3) have pda.step1 M (p₂', w₂, γ) (p₃, w₃,
 α₃)
 using final_of_stack_pda_step1_drop by simp

 ultimately show ?case by simp
 qed (rule r, metis refl_n list.inj_map_strong_sym_extended.inject)
 qed

lemma final_of_stack_pda_steps:
 (p₁, w₁, α₁) \rightsquigarrow^* (p₂, w₂, α₂) \longleftrightarrow
 pda.steps final_of_stack_pda (Old_st p₁, w₁, α_with_new α₁) (Old_st
 p₂, w₂, α_with_new α₂)
 using final_of_stack_pda_stepn pda.stepn_steps[OF pda_final_of_stack] stepn_steps
 by simp

lemma final_of_stack_pda_first_step:
 assumes pda.step1 final_of_stack_pda (New_init, w₁, [New_sym]) (p₂, w₂, α)
 shows p₂ = Old_st (init_state M) ∧ w₂ = w₁ ∧ α = [Old_sym (init_symbol
 M), New_sym]
 using assms pda.step1_rule[OF pda_final_of_stack] by (simp add: final_of_stack_pda_def)

By not allowing any moves from the new final state, we obtain a distinct
 last step, which simplifies the argument about splitting the path that the
 constructed automaton takes upon accepting a word:

lemma final_of_stack_pda_last_step:
 assumes pda.step1 final_of_stack_pda (p₁, w₁, α₁) (New_final, w₂, α₂)
 shows $\exists q. p_1 = \text{Old_st } q \wedge w_1 = w_2 \wedge \alpha_1 = \text{New_sym } \# \alpha_2$
proof –
 from assms obtain Z α where α₁_def: α₁ = Z # α and rule:
 ($\exists \beta. w_2 = w_1 \wedge \alpha_2 = \beta @ \alpha \wedge (\text{New_final}, \beta) \in \text{final_of_stack_delta_eps}$)

$p_1 Z$)
 $\vee (\exists a \beta. w_1 = a \# w_2 \wedge \alpha_2 = \beta @ \alpha \wedge (New_final, \beta) \in final_of_stack_delta\ p_1\ a\ Z)$
using $pda.step_1_rule_ext[OF\ pda_final_of_stack]$ **by** (*auto simp: final_of_stack_pda_def*)
from rule **have** $w_2 = w_1$ **and** $\alpha_2 = \alpha$ **and** $\exists q. p_1 = Old_st\ q \wedge Z = New_sym$
by (*induction p_1 Z rule: final_of_stack_delta_eps.induct*) *auto*
with α_1_def **show** *?thesis* **by** *simp*
qed

lemma *final_of_stack_pda_split_path*:
assumes $pda.stepn\ final_of_stack_pda\ (Suc\ (Suc\ n))\ (New_init, w_1, [New_sym])$
 (New_final, w_2, γ)
shows $\exists q. pda.step_1\ final_of_stack_pda\ (New_init, w_1, [New_sym])$
 $(Old_st\ (init_state\ M), w_1, [Old_sym$
 $(init_symbol\ M), New_sym]) \wedge$
 $pda.stepn\ final_of_stack_pda\ n\ (Old_st\ (init_state\ M), w_1, [Old_sym$
 $(init_symbol\ M), New_sym])$
 $(Old_st\ q, w_2, [New_sym]) \wedge$
 $pda.step_1\ final_of_stack_pda\ (Old_st\ q, w_2, [New_sym])$
 $(New_final, w_2, \gamma) \wedge \gamma = []$

proof –
from *assms* **have** $fstep: pda.step_1\ final_of_stack_pda\ (New_init, w_1, [New_sym])$
 $(Old_st\ (init_state\ M), w_1, [Old_sym$
 $(init_symbol\ M), New_sym])$
and $stepn: pda.stepn\ final_of_stack_pda\ (Suc\ n)$
 $(Old_st\ (init_state\ M), w_1, [Old_sym\ (init_symbol\ M),$
 $New_sym])$
 (New_final, w_2, γ)
using $pda.stepn_split_first[OF\ pda_final_of_stack]$ $final_of_stack_pda_first_step$
by *blast+*
from $stepn$ **obtain** q **where** $lstep: pda.step_1\ final_of_stack_pda\ (Old_st\ q, w_2,$
 $New_sym\ \# \gamma)\ (New_final, w_2, \gamma)$
and $stepn': pda.stepn\ final_of_stack_pda\ n$
 $(Old_st\ (init_state\ M), w_1, [Old_sym\ (init_symbol\ M),$
 $New_sym])$
 $(Old_st\ q, w_2, New_sym\ \# \gamma)$
using $pda.stepn_split_last[OF\ pda_final_of_stack]$ $final_of_stack_pda_last_step$
by *blast*
from $stepn'$ **have** $\exists \alpha. New_sym\ \# \gamma = \alpha_with_new\ \alpha$
using $final_of_stack_pda_bottom_elem\ pda.stepn_steps[OF\ pda_final_of_stack]$
by (*metis (no_types, opaque_lifting) Cons_eq_appendI append_Nil list.map_disc_iff list.simps(9)*)
hence $\gamma = []$
by (*metis Nil_is_map_conv hd_append2 hd_map list.sel(1,3) sym_extended.simps(3) tl_append_if*)
with $fstep\ lstep\ stepn'$ **show** *?thesis* **by** *auto*
qed

lemma *final_of_stack_pda_path_length*:

assumes *pda.stepn* *final_of_stack_pda* *n* (*New_init*, *w*₁, [*New_sym*]) (*New_final*, *w*₂, γ)

shows $\exists n'. n = \text{Suc} (\text{Suc} (\text{Suc } n'))$

proof –

from *assms* **obtain** *n'* **where** *n_def*: $n = \text{Suc } n'$ **and** *fstep*: *pda.step*₁ *final_of_stack_pda* (*New_init*, *w*₁, [*New_sym*])

(*init_symbol* *M*), [*New_sym*])

(*Old_st* (*init_state* *M*), *w*₁, [*Old_sym*])

and *stepn*: *pda.stepn* *final_of_stack_pda* *n'* (*Old_st* (*init_state* *M*), *w*₁, [*Old_sym*])

(*init_symbol* *M*), [*New_sym*])

(*New_final*, *w*₂, γ)

using *pda.stepn_not_refl_split_first*[*OF pda_final_of_stack*] *final_of_stack_pda_first_step* **by** *blast*

from *stepn* **obtain** *n''* **where** *n'_def*: $n' = \text{Suc } n''$

using *pda.stepn_not_refl_split_last*[*OF pda_final_of_stack*] **by** *blast*

with *n_def* *assms* **have** $\exists q. \text{pda.stepn } \text{final_of_stack_pda } n''$

(*Old_st* (*init_state* *M*), *w*₁, [*Old_sym* (*init_symbol* *M*), *New_sym*]) (*Old_st* *q*, *w*₂, [*New_sym*])

using *final_of_stack_pda_split_path* **by** *blast*

then obtain *n'''* **where** $n'' = \text{Suc } n'''$

using *pda.stepn_not_refl_split_last*[*OF pda_final_of_stack*] **by** *blast*

with *n_def* *n'_def* **show** *?thesis* **by** *simp*

qed

lemma *accepted_final_of_stack*:

$(\exists q. (\text{init_state } M, w, [\text{init_symbol } M]) \rightsquigarrow^* (q, [], [])) \longleftrightarrow (\exists q \gamma. q \in \text{final_states } \text{final_of_stack_pda} \wedge$

$\text{pda.steps } \text{final_of_stack_pda } (\text{init_state } \text{final_of_stack_pda}, w, [\text{init_symbol } \text{final_of_stack_pda}]) (q, [], \gamma))$ **(is** *?l* \longleftrightarrow *?r***)**

proof

assume *?l*

then obtain *q* **where** $(\text{init_state } M, w, [\text{init_symbol } M]) \rightsquigarrow^* (q, [], [])$ **by** *blast*

hence *pda.steps* *final_of_stack_pda* (*Old_st* (*init_state* *M*), *w*, [*Old_sym* (*init_symbol* *M*), *New_sym*]) (*Old_st* *q*, [], [*New_sym*])

using *final_of_stack_pda_steps* **by** *simp*

moreover have *pda.step*₁ *final_of_stack_pda* (*init_state* *final_of_stack_pda*, *w*, [*init_symbol* *final_of_stack_pda*])

(*Old_st* (*init_state* *M*), *w*, [*Old_sym* (*init_symbol* *M*), *New_sym*])

using *pda.step1_rule*[*OF pda_final_of_stack*] **by** (*simp add*: *final_of_stack_pda_def*)

moreover have *pda.step*₁ *final_of_stack_pda* (*Old_st* *q*, [], [*New_sym*]) (*New_final*, [], [])

using *pda.step1_rule*[*OF pda_final_of_stack*] **by** (*simp add*: *final_of_stack_pda_def*)

ultimately have *a1*:

```

    pda.steps final_of_stack_pda (init_state final_of_stack_pda, w, [init_symbol
final_of_stack_pda ]) (New_final, [], [])
    using pda.step1_steps[OF pda_final_of_stack] pda.steps_trans[OF pda_final_of_stack]
by metis

```

```

moreover have New_final ∈ final_states final_of_stack_pda
by (simp add: final_of_stack_pda_def)

```

```

ultimately show ?r by blast

```

```

next

```

```

assume ?r

```

```

then obtain q γ where q_final: q ∈ final_states final_of_stack_pda and

```

```

    steps: pda.steps final_of_stack_pda (init_state final_of_stack_pda,
w, [init_symbol final_of_stack_pda]) (q, [], γ) by blast

```

```

from q_final have q_def: q = New_final

```

```

by (simp add: final_of_stack_pda_def)

```

```

with steps obtain n where stepn: pda.stepn final_of_stack_pda n (New_init,
w, [New_sym]) (New_final, [], γ)

```

```

using pda.stepn_steps[OF pda_final_of_stack] by (fastforce simp add: fi-
nal_of_stack_pda_def)

```

```

then obtain n' where n = Suc (Suc n')

```

```

using final_of_stack_pda_path_length by blast

```

```

with stepn obtain p where pda.stepn final_of_stack_pda n' (Old_st (init_state
M), w, [Old_sym (init_symbol M), New_sym])

```

```

(Old_st p, [], [New_sym])

```

```

using final_of_stack_pda_split_path by blast

```

```

hence (init_state M, w, [(init_symbol M)])  $\rightsquigarrow$ (n') (p, [], [])

```

```

using final_of_stack_pda_stepn by simp

```

```

thus ?l

```

```

using stepn_steps by blast

```

```

qed

```

```

lemma final_of_stack: pda.accept_stack M = pda.accept_final final_of_stack_pda

```

```

unfolding accept_stack_def pda.accept_final_def[OF pda_final_of_stack] us-
ing accepted_final_of_stack by blast

```

```

end

```

```

end

```

2.2 Final Acceptance to Stack Acceptance

Starting from a PDA that accepts by final state we construct an equivalent PDA that accepts by empty stack, following Kozen [1].

```

theory Final_To_Stack_PDA

```

```

imports Pushdown_Automata

```

```

begin

```

```

datatype 'q st_extended = Old_st 'q | New_init | New_final

```

```

datatype 's sym_extended = Old_sym 's | New_sym

```

lemma *inj_Old_sym*: *inj Old_sym*
by (*meson injI sym_extended.inject*)

instance *st_extended* :: (*finite*) *finite*

proof

have *: $UNIV = \{t. \exists q. t = Old_st\ q\} \cup \{New_init, New_final\}$

by *auto* (*metis st_extended.exhaust*)

show *finite* (*UNIV* :: 'a *st_extended* *set*)

by (*simp add: * full_SetCompr_eq*)

qed

instance *sym_extended* :: (*finite*) *finite*

proof

have *: $UNIV = \{t. \exists s. t = Old_sym\ s\} \cup \{New_sym\}$

by *auto* (*metis sym_extended.exhaust*)

show *finite* (*UNIV* :: 'a *sym_extended* *set*)

by (*simp add: * full_SetCompr_eq*)

qed

context *pda* **begin**

fun *stack_of_final_delta* :: 'q *st_extended* \Rightarrow 'a \Rightarrow 's *sym_extended* \Rightarrow ('q *st_extended* \times 's *sym_extended* *list*) *set* **where**

stack_of_final_delta (*Old_st* q) a (*Old_sym* Z) = ($\lambda(p, \alpha). (Old_st\ p, map\ Old_sym\ \alpha)$) ' ($\delta\ M\ q\ a\ Z$)

| *stack_of_final_delta* _ _ _ = {}

fun *stack_of_final_delta_eps* :: 'q *st_extended* \Rightarrow 's *sym_extended* \Rightarrow ('q *st_extended* \times 's *sym_extended* *list*) *set* **where**

stack_of_final_delta_eps (*Old_st* q) (*Old_sym* Z) = (*if* q \in *final_states* M *then* {(New_final, [Old_sym Z])} *else* {}) \cup

($\lambda(p, \alpha). (Old_st\ p, map\ Old_sym$

$\alpha)$) ' ($\delta\ \epsilon\ M\ q\ Z$)

| *stack_of_final_delta_eps* (*Old_st* q) *New_sym* = (*if* q \in *final_states* M *then* {(New_final, [New_sym])} *else* {})

| *stack_of_final_delta_eps* *New_init* *New_sym* = {(Old_st (*init_state* M), [Old_sym (*init_symbol* M), *New_sym*])}

| *stack_of_final_delta_eps* *New_final* _ = {(New_final, [])}

| *stack_of_final_delta_eps* _ _ = {}

definition *stack_of_final_pda* :: ('q *st_extended*, 'a, 's *sym_extended*) *pda* **where**

stack_of_final_pda \equiv ($\lambda(). (init_state = New_init, init_symbol = New_sym, final_states = \{New_final\},$

$\delta = stack_of_final_delta, \delta_eps = stack_of_final_delta_eps)$)

lemma *pda_final_to_stack*:

pda *stack_of_final_pda*

proof (*standard, goal_cases*)

```

case (1 p x z)
have finite (stack_of_final_delta p x z)
  by (induction p x z rule: stack_of_final_delta.induct) (auto simp: finite_delta)
then show ?case
  by (simp add: stack_of_final_pda_def)
next
case (2 p z)
have finite (stack_of_final_delta_eps p z)
  by (induction p z rule: stack_of_final_delta_eps.induct) (auto simp: finite_delta_eps)
then show ?case
  by (simp add: stack_of_final_pda_def)
qed

```

lemma stack_of_final_pda_trans:

```

(p, β) ∈ δ M q a Z ↔
  (Old_st p, map Old_sym β) ∈ δ stack_of_final_pda (Old_st q) a (Old_sym
Z)
by (auto simp: stack_of_final_pda_def inj_map_eq_map[OF inj_Old_sym])

```

lemma stack_of_final_pda_eps:

```

(p, β) ∈ δε M q Z ↔ (Old_st p, map Old_sym β) ∈ δε stack_of_final_pda
(Old_st q) (Old_sym Z)
by (auto simp: stack_of_final_pda_def inj_map_eq_map[OF inj_Old_sym] split:
if_splits)

```

lemma stack_of_final_pda_step:

```

(p1, w1, α1) ↔ (p2, w2, α2) ↔
  pda.step1 stack_of_final_pda (Old_st p1, w1, map Old_sym α1) (Old_st
p2, w2, map Old_sym α2) (is ?l ↔ ?r)

```

proof

assume ?l

then obtain Z α **where** α1_def: α1 = Z#α **and** rule:

```

(∃ β. w2 = w1 ∧ α2 = β@α ∧ (p2, β) ∈ δε M p1 Z)
∨ (∃ a β. w1 = a # w2 ∧ α2 = β@α ∧ (p2,β) ∈ δ M p1 a Z)

```

using step1_rule_ext **by** auto

from rule **have** (∃ β. w2 = w1 ∧ map Old_sym α2 = map Old_sym β @ map
Old_sym α ∧

```

  (Old_st p2, map Old_sym β) ∈ δε stack_of_final_pda (Old_st
p1) (Old_sym Z))

```

```

  ∨ (∃ a β. w1 = a # w2 ∧ map Old_sym α2 = map Old_sym β @
map Old_sym α ∧

```

```

  (Old_st p2, map Old_sym β) ∈ δ stack_of_final_pda (Old_st p1)
a (Old_sym Z))

```

using stack_of_final_pda_trans stack_of_final_pda_eps **by** fastforce

hence (∃ β. w2 = w1 ∧ map Old_sym α2 = β @ map Old_sym α ∧ (Old_st p2,
β) ∈ δε stack_of_final_pda (Old_st p1) (Old_sym Z))

```

  ∨ (∃ a β. w1 = a # w2 ∧ map Old_sym α2 = β @ map Old_sym α ∧
(Old_st p2, β) ∈ δ stack_of_final_pda (Old_st p1) a (Old_sym Z)) by blast

```

with α1_def **show** ?r

using $pda.step1_rule[OF\ pda_final_to_stack]$ **by** $simp$
next
assume $?r$
then obtain $Z\ \alpha$ **where** $map_alpha_def: map\ Old_sym\ \alpha_1 = Old_sym\ Z\ \# map\ Old_sym\ \alpha$ **and** **rule:**
 $(\exists\ \beta. w_2 = w_1 \wedge map\ Old_sym\ \alpha_2 = \beta @ map\ Old_sym\ \alpha \wedge$
 $(Old_st\ p_2, \beta) \in \delta\varepsilon\ stack_of_final_pda\ (Old_st\ p_1)\ (Old_sym\ Z))$
 $\vee (\exists\ a\ \beta. w_1 = a\ \# w_2 \wedge map\ Old_sym\ \alpha_2 = \beta @ map\ Old_sym\ \alpha \wedge$
 $(Old_st\ p_2, \beta) \in \delta\ stack_of_final_pda\ (Old_st\ p_1)\ a\ (Old_sym\ Z))$
using $pda.step1_rule_ext[OF\ pda_final_to_stack]$ **by** $auto$
from map_alpha_def **have** $alpha_def: \alpha_1 = Z\ \# \alpha$
by $(metis\ list.inj_map_strong\ list.simps(9)\ sym_extended.inject)$
from **rule** **have** $(\exists\ \beta. w_2 = w_1 \wedge map\ Old_sym\ \alpha_2 = map\ Old_sym\ \beta @ map\ Old_sym\ \alpha \wedge$
 $(Old_st\ p_2, map\ Old_sym\ \beta) \in \delta\varepsilon\ stack_of_final_pda\ (Old_st\ p_1)\ (Old_sym\ Z))$
 $\vee (\exists\ a\ \beta. w_1 = a\ \# w_2 \wedge map\ Old_sym\ \alpha_2 = map\ Old_sym\ \beta @ map\ Old_sym\ \alpha \wedge$
 $(Old_st\ p_2, map\ Old_sym\ \beta) \in \delta\ stack_of_final_pda\ (Old_st\ p_1)\ a\ (Old_sym\ Z))$
using $append_eq_map_conv[where\ ?f = Old_sym]$ **by** $metis$
hence $(\exists\ \beta. w_2 = w_1 \wedge \alpha_2 = \beta @ \alpha \wedge (p_2, \beta) \in \delta\varepsilon\ M\ p_1\ Z)$
 $\vee (\exists\ a\ \beta. w_1 = a\ \# w_2 \wedge \alpha_2 = \beta @ \alpha \wedge (p_2, \beta) \in \delta\ M\ p_1\ a\ Z)$
using $stack_of_final_pda_trans\ stack_of_final_pda_eps$ **by** $(metis\ list.inj_map_strong\ sym_extended.inject\ map_append)$
with $alpha_def$ **show** $?l$
using $step1_rule$ **by** $simp$
qed

abbreviation $\alpha_with_new :: 's\ list \Rightarrow 's\ sym_extended\ list$ **where**
 $\alpha_with_new\ \alpha \equiv map\ Old_sym\ \alpha\ @\ [New_sym]$

lemma $stack_of_final_pda_step1_drop:$

assumes $pda.step1\ stack_of_final_pda\ (Old_st\ p_1, w_1, \alpha_with_new\ \alpha_1)$
 $(Old_st\ p_2, w_2, \alpha_with_new\ \alpha_2)$

shows $(p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2)$

proof –

from $assms$ **obtain** $Z\ \alpha$ **where** $alpha_with_new_def: \alpha_with_new\ \alpha_1 = Z\ \# \alpha$
and **rule:**

$(\exists\ \beta. w_2 = w_1 \wedge \alpha_with_new\ \alpha_2 = \beta @ \alpha \wedge (Old_st\ p_2, \beta) \in stack_of_final_delta_eps$
 $(Old_st\ p_1)\ Z)$

$\vee (\exists\ a\ \beta. w_1 = a\ \# w_2 \wedge \alpha_with_new\ \alpha_2 = \beta @ \alpha \wedge (Old_st\ p_2, \beta) \in$
 $stack_of_final_delta\ (Old_st\ p_1)\ a\ Z)$

using $pda.step1_rule_ext[OF\ pda_final_to_stack]$ **by** $(auto\ simp: stack_of_final_pda_def)$

from **rule** **have** $Z \neq New_sym$

by $(induction\ Old_st\ p_1\ Z\ rule: stack_of_final_delta_eps.induct)\ (auto, metis\ empty_iff_fst_conv\ singletonD\ st_extended.simps(5))$

with $alpha_with_new_def$ **have** $map\ Old_sym\ \alpha_1 \neq []$ **by** $auto$

with $assms$ **have** $pda.step1\ stack_of_final_pda\ (Old_st\ p_1, w_1, map\ Old_sym\ \alpha_1)$

α_1)
 $(Old_st\ p_2, w_2, map\ Old_sym\ \alpha_2)$
using $pda.step_1_stack_drop[OF\ pda_final_to_stack]$ **by** $blast$
thus $?thesis$
using $stack_of_final_pda_step$ **by** $simp$
qed

lemma $stack_of_final_pda_from_old$:
assumes $pda.step_1\ stack_of_final_pda\ (Old_st\ p_1, w_1, \alpha_1)\ (p_2, w_2, \alpha_2)$
shows $(\exists\ p_2'.\ p_2 = Old_st\ p_2') \vee p_2 = New_final$
proof –
from $assms$ **obtain** $Z\ \alpha$ **where** $rule$:
 $(\exists\ \beta.\ w_2 = w_1 \wedge \alpha_2 = \beta @ \alpha \wedge (p_2, \beta) \in stack_of_final_delta_eps\ (Old_st\ p_1)\ Z)$
 $\vee (\exists\ a\ \beta.\ w_1 = a \# w_2 \wedge \alpha_2 = \beta @ \alpha \wedge (p_2, \beta) \in stack_of_final_delta\ (Old_st\ p_1)\ a\ Z)$
using $pda.step_1_rule_ext[OF\ pda_final_to_stack]$ **by** $(auto\ simp:\ stack_of_final_pda_def)+$
thus $?thesis$
by $(induction\ Old_st\ p_1\ Z\ rule:\ stack_of_final_delta_eps.induct,\ auto)\ (metis\ empty_iff_fst_conv\ singletonD)+$
qed

lemma $stack_of_final_pda_from_final$:
assumes $pda.step_1\ stack_of_final_pda\ (New_final, w_1, \alpha_1)\ (p_2, w_2, \alpha_2)$
shows $\exists\ Z'.\ p_2 = New_final \wedge w_2 = w_1 \wedge \alpha_1 = Z' \# \alpha_2$
proof –
from $assms$ **obtain** $Z\ \alpha$ **where** $\alpha_1_def:\ \alpha_1 = Z \# \alpha$ **and** $rule$:
 $(\exists\ \beta.\ w_2 = w_1 \wedge \alpha_2 = \beta @ \alpha \wedge (p_2, \beta) \in stack_of_final_delta_eps\ New_final\ Z)$
 $\vee (\exists\ a\ \beta.\ w_1 = a \# w_2 \wedge \alpha_2 = \beta @ \alpha \wedge (p_2, \beta) \in stack_of_final_delta\ New_final\ a\ Z)$
using $pda.step_1_rule_ext[OF\ pda_final_to_stack]$ **by** $(auto\ simp:\ stack_of_final_pda_def)$
thus $?thesis$
by $(induction\ New_final::\ 'q\ st_extended\ Z\ rule:\ stack_of_final_delta_eps.induct)\ auto$
qed

lemma $stack_of_final_pda_from_oldn$:
assumes $pda.steps\ stack_of_final_pda\ (Old_st\ p_1, w_1, \alpha_1)\ (p_2, w_2, \alpha_2)$
shows $\exists\ q'.\ p_2 = Old_st\ q' \vee p_2 = New_final$
proof $(induction\ (Old_st\ p_1, w_1, \alpha_1)\ (p_2, w_2, \alpha_2)\ arbitrary:\ p_2\ w_2\ \alpha_2\ rule:\ pda.steps_induct2_bw[OF\ pda_final_to_stack])$
case $(\exists\ p_2\ w_2\ \alpha_2\ p_3\ w_3\ \alpha_3)$
then **show** $?case$
using $stack_of_final_pda_from_final\ stack_of_final_pda_from_old$ **by** $blast$
qed $(auto\ simp:\ assms)$

lemma $stack_of_final_pda_to_old$:
assumes $pda.step_1\ stack_of_final_pda\ (p_1, w_1, \alpha_1)\ (Old_st\ p_2, w_2, \alpha_2)$

shows $(\exists q'. p_1 = \text{Old_st } q') \vee p_1 = \text{New_init}$
using *assms stack_of_final_pda_from_final* **by** (*metis st_extended.exhaust*)

lemma *stack_of_final_pda_bottom_elem*:

assumes *pda.steps stack_of_final_pda* (*Old_st* $p_1, w_1, \alpha_with_new \alpha_1$) (*Old_st* p_2, w_2, γ)

shows $\exists \alpha. \gamma = \alpha_with_new \alpha$

proof (*induction* (*Old_st* $p_1, w_1, \alpha_with_new \alpha_1$) (*Old_st* p_2, w_2, γ) *arbitrary*):
 $p_2 w_2 \gamma$ *rule*: *pda.steps_induct2_bw*[*OF pda_final_to_stack*])

case ($\exists p_2 w_2 \alpha_2 w_3 \alpha_3 p_3$)

obtain p_2' **where** p_2_def : $p_2 = \text{Old_st } p_2'$

using *stack_of_final_pda_from_oldn*[*OF* $\exists(1)$] *stack_of_final_pda_to_old*[*OF* $\exists(2)$] **by** *blast*

with $\exists(3)$ **have** α_2_def : $\exists \alpha. \alpha_2 = \alpha_with_new \alpha$ **by** *simp*

from $p_2_def \exists(2)$ **obtain** $Z \alpha$ **where** α_2_split : $\alpha_2 = Z \# \alpha$ **and** *rule*:

$(\exists \beta. w_3 = w_2 \wedge \alpha_3 = \beta @ \alpha \wedge (\text{Old_st } p_3, \beta) \in \text{stack_of_final_delta_eps}$
 $(\text{Old_st } p_2') Z)$

$\vee (\exists a \beta. w_2 = a \# w_3 \wedge \alpha_3 = \beta @ \alpha \wedge (\text{Old_st } p_3, \beta) \in \text{stack_of_final_delta}$
 $(\text{Old_st } p_2') a Z)$

using *pda.step1_rule_ext*[*OF pda_final_to_stack*] **by** (*auto simp: stack_of_final_pda_def*)

hence $\exists Z'. Z = \text{Old_sym } Z'$

by (*induction* *Old_st* $p_2' Z$ *rule*: *stack_of_final_delta_eps.induct*)

(*auto, meson empty_iff_insert_iff prod.inject st_extended.distinct*(\exists))

with $\alpha_2_def \alpha_2_split$ **have** $\exists \gamma. \alpha = \alpha_with_new \gamma$

by (*metis hd_append list.sel*(1,3) *map_tl sym_extended.simps*(\exists) *tl_append_if*)

with *rule show ?case*

by (*induction* *Old_st* $p_2' Z$ *rule*: *stack_of_final_delta_eps.induct, auto*)

(*metis empty_iff fst_conv singleton_iff st_extended.distinct*(\exists), *metis*

map_append,

metis map_append, metis empty_iff fst_conv singleton_iff st_extended.distinct(\exists))

qed (*auto simp: assms*)

lemma *stack_of_final_pda_stepn*:

$(p_1, w_1, \alpha_1) \rightsquigarrow(n) (p_2, w_2, \alpha_2) \longleftrightarrow$

pda.stepn stack_of_final_pda n (*Old_st* $p_1, w_1, \alpha_with_new \alpha_1$) (*Old_st* $p_2,$
 $w_2, \alpha_with_new \alpha_2$) (**is** $?l \longleftrightarrow ?r$)

proof

show $?l \implies ?r$

proof (*induction* n (p_1, w_1, α_1) (p_2, w_2, α_2) *arbitrary*: $p_2 w_2 \alpha_2$ *rule*: *stepn.induct*)

case (*stepn* $n p_2 w_2 \alpha_2 p_3 w_3 \alpha_3$)

from *stepn*(\exists) **have** *pda.step1 stack_of_final_pda* (*Old_st* $p_2, w_2, \text{map Old_sym}$
 α_2) (*Old_st* $p_3, w_3, \text{map Old_sym } \alpha_3$)

using *stack_of_final_pda_step* **by** *simp*

hence *pda.step1 stack_of_final_pda* (*Old_st* $p_2, w_2, \alpha_with_new \alpha_2$) (*Old_st* $p_3,$
 $w_3, \alpha_with_new \alpha_3$)

using *pda.step1_stack_app*[*OF pda_final_to_stack*] **by** *simp*

with *stepn*(2) **show** *?case*

by (*simp add: pda.stepn*[*OF pda_final_to_stack*])

qed (*simp add: pda.refl_n*[*OF pda_final_to_stack*])

next
assume $r: ?r$ **thus** $?l$
proof (*induction* n (*Old_st* $p_1, w_1, \alpha_with_new$ α_1) (*Old_st* $p_2, w_2, \alpha_with_new$ α_2))
arbitrary: $p_2 w_2 \alpha_2$ *rule:* $pda.stepsn.induct[OF pda_final_to_stack]$
case ($\exists n p_2 w_2 \alpha_2 w_3 p_3 \alpha_3$)
from $\exists(1)$ **have** $steps_3_1: pda.steps$ *stack_of_final_pda* (*Old_st* $p_1, w_1, \alpha_with_new$ α_1) (p_2, w_2, α_2)
using $pda.stepsn_steps[OF pda_final_to_stack]$ **by** *blast*
obtain p_2' **where** $p_2_def: p_2 = Old_st p_2'$
using $stack_of_final_pda_from_oldn[OF steps_3_1]$ $stack_of_final_pda_to_old[OF \exists(3)]$ **by** *blast*
with $steps_3_1$ **obtain** γ **where** $\alpha_2_def: \alpha_2 = \alpha_with_new \gamma$
using $stack_of_final_pda_bottom_elem$ **by** *blast*

with $p_2_def \exists(1,2)$ **have** $pda.stepsn$ M n (p_1, w_1, α_1) (p_2', w_2, γ) **by** *simp*

moreover from $p_2_def \alpha_2_def \exists(3)$ **have** $pda.step1$ M (p_2', w_2, γ) (p_3, w_3, α_3)
using $stack_of_final_pda_step1_drop$ **by** *simp*

ultimately show $?case$ **by** *simp*
qed (*rule* $r,metis refl_n list.inj_map_strong sym_extended.inject$)
qed

lemma *stack_of_final_pda_steps:*
 $(p_1, w_1, \alpha_1) \rightsquigarrow^* (p_2, w_2, \alpha_2) \longleftrightarrow$
 $pda.steps$ *stack_of_final_pda* (*Old_st* $p_1, w_1, \alpha_with_new$ α_1) (*Old_st* $p_2, w_2, \alpha_with_new$ α_2)
using $stack_of_final_pda_stepn$ $pda.stepsn_steps[OF pda_final_to_stack]$ $stepn_steps$ **by** *simp*

lemma *stack_of_final_pda_final_dump:*
 $pda.steps$ *stack_of_final_pda* (*New_final*, w, γ) (*New_final*, $w, []$)
proof (*induction* γ)
case (*Cons* $Z \gamma$)
have (*New_final*, $[]$) \in *stack_of_final_delta_eps* *New_final* Z **by** *simp*
hence $pda.step1$ *stack_of_final_pda* (*New_final*, $w, Z \# \gamma$) (*New_final*, w, γ)
using $pda.step1_rule[OF pda_final_to_stack]$ **by** (*simp add:* $stack_of_final_pda_def$)
with *Cons.IH* **show** $?case$
using $pda.step1_steps[OF pda_final_to_stack]$ $pda.steps_trans[OF pda_final_to_stack]$ **by** *blast*
qed (*simp add:* $pda.steps_refl[OF pda_final_to_stack]$)

lemma *stack_of_final_pda_first_step:*
assumes $pda.step1$ *stack_of_final_pda* (*New_init*, $w_1, [New_sym]$) (p_2, w_2, α)
shows $p_2 = Old_st$ (*init_state* M) $\wedge w_2 = w_1 \wedge \alpha = [Old_sym$ (*init_symbol* M), *New_sym*]
using $assms$ $pda.step1_rule[OF pda_final_to_stack]$ **by** (*simp add:* $stack_of_final_pda_def$)

lemma *stack_of_final_pda_empty_only_final*:

assumes *pda.steps stack_of_final_pda* (*New_init*, *w*₁, [*New_sym*]) (*q*, *w*₂, [])
shows *q = New_final*

proof –

from *assms* **have** *pda.steps stack_of_final_pda* (*Old_st* (*init_state M*), *w*₁,
[*Old_sym* (*init_symbol M*), *New_sym*]) (*q*, *w*₂, [])

using *pda.steps_not_refl_split_first*[*OF pda_final_to_stack*] *stack_of_final_pda_first_step*
by *blast*

thus *?thesis*

using *stack_of_final_pda_bottom_elem*[*of init_state M w*₁ [*init_symbol M*]
_ *w*₂ []] *stack_of_final_pda_from_oldn* **by** *fastforce*

qed

lemma *stack_of_final_pda_split_old_final*:

assumes *pda.stepn stack_of_final_pda* (*Suc n*) (*Old_st p*₁, *w*₁, α ₁) (*New_final*
:: '*q st_extended*, *w*₂, α ₂)

shows $\exists q k \gamma. k \leq n \wedge q \in \text{final_states } M \wedge$

pda.stepn stack_of_final_pda *k* (*Old_st p*₁, *w*₁, α ₁) (*Old_st q*, *w*₂, γ) \wedge

*pda.step*₁ *stack_of_final_pda* (*Old_st q*, *w*₂, γ) (*New_final*, *w*₂, γ) \wedge

pda.stepn stack_of_final_pda (*n-k*) (*New_final*, *w*₂, γ) (*New_final*, *w*₂,

α ₂)

using *assms* **proof** (*induction Suc n* (*Old_st p*₁, *w*₁, α ₁) (*New_final* :: '*q st_extended*,
*w*₂, α ₂)

*arbitrary: n w*₂ α ₂ *rule: pda.stepn.induct*[*OF pda_final_to_stack*])

case ($\exists n p_2 w_2 \alpha_2 w_3 \alpha_3$)

then show *?case* **proof** (*cases n*)

case 0

with $\exists(1)$ **have** *p2_def*: *Old_st p*₁ = *p*₂ **and** *w12*: *w*₁ = *w*₂ **and** *a12*: α ₁ =

α ₂

using *pda.stepn_zeroE*[*OF pda_final_to_stack*] **by** *blast+*

from *p2_def* $\exists(3)$ **obtain** *Z* α **where** *alpha_def*: $\alpha_2 = Z \# \alpha$ **and** *rule*:

$(\exists \beta. w_3 = w_2 \wedge \alpha_3 = \beta @ \alpha \wedge (\text{New_final}, \beta) \in \text{stack_of_final_delta_eps}$

(*Old_st p*₁) *Z*)

$\vee (\exists a \beta. w_2 = a \# w_3 \wedge \alpha_3 = \beta @ \alpha \wedge (\text{New_final}, \beta) \in \text{stack_of_final_delta}$

(*Old_st p*₁) *a Z*)

using *pda.step1_rule_ext*[*OF pda_final_to_stack*] **by** (*auto simp: stack_of_final_pda_def*)

from *alpha_def* *rule* **have** *p1_final*: *p*₁ \in *final_states M* **and** *w23*: *w*₃ = *w*₂ **and**

a23: $\alpha_3 = \alpha_2$

by (*induction Old_st p*₁ *Z* *rule: stack_of_final_delta_eps.induct, auto*)

(*meson empty_iff, meson empty_iff prod.inject singletonD, meson empty_iff,*
meson empty_iff prod.inject singletonD)

from *w12 w23 a12 a23* **have** *pda.stepn stack_of_final_pda* 0 (*Old_st p*₁, *w*₁,
 α ₁) (*Old_st p*₁, *w*₃, α ₃)

using *pda.refl_n*[*OF pda_final_to_stack*] **by** *simp*

moreover from $\exists(3)$ *p2_def w23 a23* **have** *pda.step*₁ *stack_of_final_pda*
(*Old_st p*₁, *w*₃, α ₃) (*New_final*, *w*₃, α ₃) **by** *simp*

moreover from 0 have $pda.stepn\ stack_of_final_pda\ (n - 0)\ (New_final,$
 $w_3, \alpha_3)\ (New_final, w_3, \alpha_3)$
using $pda.refl_n[OF\ pda_final_to_stack]$ **by** *simp*

ultimately show *?thesis*
using $p1_final$ **by** *blast*

next

case $(Suc\ n')$

then show *?thesis* **proof** (*cases* $p_2 = New_final$)

case *True*

with $Suc\ \exists(1,2)$ **obtain** $q\ k\ \gamma$ **where** $k_less: k \leq n'$ **and** $q_final: q \in$
 $final_states\ M$ **and**

$stepn: pda.stepn\ stack_of_final_pda\ k\ (Old_st\ p_1, w_1, \alpha_1)\ (Old_st\ q, w_2, \gamma)$

and

$step1: pda.step1\ stack_of_final_pda\ (Old_st\ q, w_2, \gamma)\ (New_final, w_2, \gamma)$

and

$stepn': pda.stepn\ stack_of_final_pda\ (n' - k)\ (New_final, w_2, \gamma)\ (New_final,$
 $w_2, \alpha_2)$ **by** *blast*

from *True* $\exists(\beta)$ **obtain** $Z'\ \alpha'$ **where** $\alpha_2 = Z' \# \alpha'$ **and** *rule*:

$(\exists \beta. w_3 = w_2 \wedge \alpha_3 = \beta @ \alpha' \wedge (New_final, \beta) \in stack_of_final_delta_eps$
 $New_final\ Z')$

$\vee (\exists a\ \beta. w_2 = a \# w_3 \wedge \alpha_3 = \beta @ \alpha' \wedge (New_final, \beta) \in stack_of_final_delta$
 $New_final\ a\ Z')$

using $pda.step1_rule_ext[OF\ pda_final_to_stack]$ **by** (*auto* *simp*: $stack_of_final_pda_def$)

from *rule* **have** $w2\beta: w_3 = w_2$

by (*induction* $New_final :: 'q\ st_extended\ Z'$ *rule*: $stack_of_final_delta_eps.induct$)

auto

moreover from $k_less\ Suc$ **have** $k \leq n$ **by** *simp*

moreover from $stepn\ w2\beta$ **have** $pda.stepn\ stack_of_final_pda\ k\ (Old_st\ p_1,$
 $w_1, \alpha_1)\ (Old_st\ q, w_3, \gamma)$ **by** *simp*

moreover from $step1\ w2\beta$ **have** $pda.step1\ stack_of_final_pda\ (Old_st\ q,$
 $w_3, \gamma)\ (New_final, w_3, \gamma)$ **by** *simp*

moreover from $stepn'\ \exists(\beta)\ True\ w2\beta\ Suc\ k_less$ **have** $pda.stepn\ stack_of_final_pda$
 $(n - k)\ (New_final, w_3, \gamma)\ (New_final, w_3, \alpha_3)$
using $pda.stepn[OF\ pda_final_to_stack]$ **by** (*simp* *add*: Suc_diff_le)

ultimately show *?thesis*
using q_final **by** *blast*

next

case *False*

with $\exists(1)$ **obtain** p_2' **where** $p_2_def: p_2 = Old_st\ p_2'$

using $stack_of_final_pda_from_oldn\ pda.stepn_steps[OF\ pda_final_to_stack]$

by *blast*

from $p_2_def\ \exists(\beta)$ **obtain** $Z'\ \alpha'$ **where** $\alpha_2 = Z' \# \alpha'$ **and**

$(\exists \beta. w_3 = w_2 \wedge \alpha_3 = \beta @ \alpha' \wedge (New_final, \beta) \in stack_of_final_delta_eps$
 $(Old_st\ p_2')\ Z')$
 $\vee (\exists a\ \beta. w_2 = a \# w_3 \wedge \alpha_3 = \beta @ \alpha' \wedge (New_final, \beta) \in$
 $stack_of_final_delta\ (Old_st\ p_2')\ a\ Z')$
using $pda.step1_rule_ext[OF\ pda_final_to_stack]$ **by** $(auto\ simp:\ stack_of_final_pda_def)$
hence $p_2_final: p_2' \in final_states\ M$ **and** $w23: w_3 = w_2$ **and** $a23: \alpha_3 = \alpha_2$
by $(induction\ Old_st\ p_2'\ Z'\ rule:\ stack_of_final_delta_eps.induct,\ auto)$
 $(meson\ empty_iff,\ meson\ empty_iff\ prod.inject\ singletonD,\ meson$
 $empty_iff,\ meson\ empty_iff\ prod.inject\ singletonD)$
from $\exists(1)\ p_2_def\ w23\ a23$ **have** $pda.stepn\ stack_of_final_pda\ n\ (Old_st$
 $p_1,\ w_1,\ \alpha_1)\ (Old_st\ p_2',\ w_3,\ \alpha_3)$ **by** $simp$
moreover from $\exists(3)\ p_2_def\ w23\ a23$ **have** $pda.step1\ stack_of_final_pda$
 $(Old_st\ p_2',\ w_3,\ \alpha_3)\ (New_final,\ w_3,\ \alpha_3)$ **by** $simp$
moreover have $pda.stepn\ stack_of_final_pda\ 0\ (New_final,\ w_3,\ \alpha_3)\ (New_final,$
 $w_3,\ \alpha_3)$
using $pda.refl_n[OF\ pda_final_to_stack]$ **by** $simp$
ultimately show $?thesis$
using p_2_final **by** $force$
qed
qed
qed $(simp\ add:\ assms)$

lemma $stack_of_final_pda_split_path:$
assumes $pda.stepn\ stack_of_final_pda\ (Suc\ (Suc\ n))\ (New_init,\ w_1,\ [New_sym])$
 $(New_final,\ w_2,\ \gamma)$
shows $\exists q\ k\ \alpha. k \leq n \wedge q \in final_states\ M \wedge pda.step1\ stack_of_final_pda$
 $(New_init,\ w_1,\ [New_sym])$
 $(Old_st\ (init_state\ M),\ w_1,\ [Old_sym$
 $(init_symbol\ M),\ New_sym]) \wedge$
 $pda.stepn\ stack_of_final_pda\ k\ (Old_st\ (init_state\ M),\ w_1,\ [Old_sym$
 $(init_symbol\ M),\ New_sym])$
 $(Old_st\ q,\ w_2,\ \alpha) \wedge$
 $pda.step1\ stack_of_final_pda\ (Old_st\ q,\ w_2,\ \alpha)\ (New_final,\ w_2,\ \alpha) \wedge$
 $pda.stepn\ stack_of_final_pda\ (n-k)\ (New_final,\ w_2,\ \alpha)\ (New_final,\ w_2,$
 $\gamma)$
proof –
from $assms$ **have** $fstep: pda.step1\ stack_of_final_pda\ (New_init,\ w_1,\ [New_sym])$
 $(Old_st\ (init_state\ M),\ w_1,\ [Old_sym\ (init_symbol$
 $M),\ New_sym])$
and $stepn: pda.stepn\ stack_of_final_pda\ (Suc\ n)$
 $(Old_st\ (init_state\ M),\ w_1,\ [Old_sym\ (init_symbol\ M),$
 $New_sym])$
 $(New_final,\ w_2,\ \gamma)$
using $pda.stepn_split_first[OF\ pda_final_to_stack]$ $stack_of_final_pda_first_step$

by *blast+*
from *stepn* **have** $\exists q k \alpha. k \leq n \wedge q \in \text{final_states } M \wedge$
 $\text{pda.stepn stack_of_final_pda } k \text{ (Old_st (init_state } M), w_1, [\text{Old_sym}$
 $(\text{init_symbol } M), \text{New_sym}])$
 $(\text{Old_st } q, w_2, \alpha) \wedge$
 $\text{pda.step}_1 \text{ stack_of_final_pda (Old_st } q, w_2, \alpha) (\text{New_final}, w_2, \alpha) \wedge$
 $\text{pda.stepn stack_of_final_pda } (n-k) (\text{New_final}, w_2, \alpha) (\text{New_final}, w_2,$
 $\gamma)$
using *stack_of_final_pda_split_old_final* **by** *blast*
with *fstep* **show** *?thesis* **by** *blast*
qed

lemma *stack_of_final_pda_path_length*:
assumes $\text{pda.stepn stack_of_final_pda } n \text{ (New_init}, w_1, [\text{New_sym}]) (\text{New_final},$
 $w_2, \gamma)$
shows $\exists n'. n = \text{Suc } (\text{Suc } n')$
proof –
from *assms* **obtain** n' **where** $n_def: n = \text{Suc } n'$ **and**
 $\text{stepn}' : \text{pda.stepn stack_of_final_pda } n' \text{ (Old_st (init_state } M), w_1,$
 $[\text{Old_sym } (\text{init_symbol } M), \text{New_sym}])$
 $(\text{New_final}, w_2, \gamma)$
using *pda.stepn_not_refl_split_first[OF pda_final_to_stack]* *stack_of_final_pda_first_step*
by *blast*
from stepn' **obtain** n'' **where** $n' = \text{Suc } n''$
using *pda.stepn_not_refl_split_first[OF pda_final_to_stack]* **by** *blast*
with n_def **show** *?thesis* **by** *simp*
qed

lemma *accepted_final_to_stack*:
 $(\exists q \gamma. q \in \text{final_states } M \wedge (\text{init_state } M, w, [\text{init_symbol } M]) \rightsquigarrow^* (q, [], \gamma))$
 \longleftrightarrow
 $(\exists q. \text{pda.steps stack_of_final_pda } (\text{init_state } \text{stack_of_final_pda}, w, [\text{init_symbol}$
 $\text{stack_of_final_pda}]) (q, [], [])) \text{ (is } ?l \longleftrightarrow ?r)$
proof
assume $?l$
then obtain $q \gamma$ **where** $q_final: q \in \text{final_states } M$ **and** $\text{steps: } (\text{init_state } M,$
 $w, [\text{init_symbol } M]) \rightsquigarrow^* (q, [], \gamma)$ **by** *blast*
obtain $Z \alpha$ **where** $\text{map_}\gamma_def: \alpha_with_new \gamma = Z \# \alpha$
by *(auto intro: list.exhaust_sel)*
from q_final **have** $(\text{New_final}, [Z]) \in \text{stack_of_final_delta_eps } (\text{Old_st } q) Z$
by *(induction Old_st q Z rule: stack_of_final_delta_eps.induct) auto*

with $\text{map_}\gamma_def$ **have** $\text{pda.step}_1 \text{ stack_of_final_pda } (\text{Old_st } q, [], \alpha_with_new$
 $\gamma) (\text{New_final}, [], Z \# \alpha)$
using *pda.step1_rule[OF pda_final_to_stack]* **by** *(simp add: stack_of_final_pda_def)*

moreover from steps **have** $\text{pda.steps stack_of_final_pda } (\text{Old_st } (\text{init_state}$
 $M), w, [\text{Old_sym } (\text{init_symbol } M), \text{New_sym}])$
 $(\text{Old_st } q, [], \alpha_with_new \gamma)$

```

using stack_of_final_pda_steps by simp

moreover have pda.step1_stack_of_final_pda (init_state_stack_of_final_pda,
w, [init_symbol_stack_of_final_pda])
  (Old_st (init_state M), w, [Old_sym (init_symbol
M), New_sym])
using pda.step1_rule[OF pda_final_to_stack] by (simp add: stack_of_final_pda_def)

moreover have pda.steps_stack_of_final_pda (New_final, [], Z#α) (New_final,
[], [])
using stack_of_final_pda_final_dump by simp

ultimately show ?r
using pda.step1_steps[OF pda_final_to_stack] pda.steps_trans[OF pda_final_to_stack]
by metis
next
assume ?r
then obtain q where steps: pda.steps_stack_of_final_pda (New_init, w, [New_sym])
(q, [], [])
by (auto simp: stack_of_final_pda_def)
hence q_def: q = New_final
using stack_of_final_pda_empty_only_final by simp
with steps obtain n where stepn: pda.stepn_stack_of_final_pda n (New_init,
w, [New_sym]) (New_final, [], [])
using pda.stepn_steps[OF pda_final_to_stack] by blast
then obtain n' where n = Suc (Suc n')
using stack_of_final_pda_path_length by blast
with stepn obtain p k α where p_final: p ∈ final_states M and stepn':
pda.stepn_stack_of_final_pda k
  (Old_st (init_state M), w, [Old_sym (init_symbol M), New_sym])
(Old_st p, [], α)
using stack_of_final_pda_split_path by blast
from stepn' obtain α' where α = α_with_new α'
using stack_of_final_pda_bottom_elem pda.stepn_steps[OF pda_final_to_stack]
by (metis (no_types, opaque_lifting) append_Cons append_Nil list.simps(8,9))
with stepn' have pda.stepn M k (init_state M, w, [init_symbol M]) (p, [], α')
using stack_of_final_pda_stepn by simp
with p_final show ?l
using stepn_steps by blast
qed

lemma final_to_stack:
  pda.accept_final M = pda.accept_stack_stack_of_final_pda
unfolding accept_final_def pda.accept_stack_def[OF pda_final_to_stack] us-
ing accepted_final_to_stack by blast

end
end

```

3 Equivalence of CFG and PDA

3.1 CFG to PDA

Starting from a CFG, we construct an equivalent single-state PDA. The formalization is based on the Lean formalization by Leichtfried[2].

```
theory CFG_To_PDA
imports
  Pushdown_Automata
  Context_Free_Grammar.Context_Free_Grammar
begin

datatype sing_st = Q_loop

instance sing_st :: finite
proof
  have *: UNIV = {Q_loop}
    by (auto intro: sing_st.exhaust)
  show finite (UNIV :: sing_st set)
    by (simp add: *)
qed

instance sym :: (finite, finite) finite
proof
  have *: UNIV = {t. ∃ s. t = Nt s} ∪ {t. ∃ s. t = Tm s}
    by (auto intro: sym.exhaust)
  show finite (UNIV :: ('a, 'b) sym set)
    by (simp add: * full_SetCompr_eq)
qed

locale cfg_to_pda =
  fixes G :: ('n :: finite, 't :: finite) Cfg
  assumes finite_G: finite (Prods G)
begin

fun pda_of_cfg :: sing_st ⇒ 't ⇒ ('n,'t) sym ⇒ (sing_st × ('n,'t) syms) set
where
  pda_of_cfg Q_loop a (Tm b) = (if a = b then {(Q_loop, [])} else {})
| pda_of_cfg _ _ _ = {}

fun pda_eps_of_cfg :: sing_st ⇒ ('n,'t) sym ⇒ (sing_st × ('n,'t) syms) set
where
  pda_eps_of_cfg Q_loop (Nt A) = {(Q_loop, α) | α. (A, α) ∈ Prods G}
| pda_eps_of_cfg _ _ = {}

definition cfg_to_pda_pda :: (sing_st, 't, ('n,'t) sym) pda where
  cfg_to_pda_pda ≡ (| init_state = Q_loop, init_symbol = Nt (Start G), final_states = {},
    delta = pda_of_cfg, delta_eps = pda_eps_of_cfg |)
```

```

lemma pda_cfg_to_pda: pda_cfg_to_pda_pda
proof (standard, goal_cases)
  case (1 p x z)
  have finite (pda_of_cfg p x z)
    by (induction p x z rule: pda_of_cfg.induct) auto
  then show ?case
    by (simp add: cfg_to_pda_pda_def)
next
  case (2 p z)
  let ?h =  $\lambda(A, \alpha). (Q\_loop, \alpha)$ 
  have *:  $\bigwedge A. \{(Q\_loop, \alpha) \mid \alpha. (A, \alpha) \in Prods\ G\} \subseteq (?h \text{ ' } Prods\ G)$  by auto
  have **: finite (?h ' Prods G)
    by (simp add: finite_G)
  have  $\bigwedge A. finite \{(Q\_loop, \alpha) \mid \alpha. (A, \alpha) \in Prods\ G\}$ 
    using finite_subset[OF * **] by simp
  hence finite (pda_eps_of_cfg p z)
    by (induction p z rule: pda_eps_of_cfg.induct) auto
  then show ?case
    by (simp add: cfg_to_pda_pda_def)
qed

lemma cfg_to_pda_cons_tm:
  pda.step1_cfg_to_pda_pda (Q_loop, a#w, Tm a# $\gamma$ ) (Q_loop, w,  $\gamma$ )
using pda.step1_rule[OF pda_cfg_to_pda] by (simp add: cfg_to_pda_pda_def)

lemma cfg_to_pda_cons_nt:
  assumes  $(A, \alpha) \in Prods\ G$ 
  shows pda.step1_cfg_to_pda_pda (Q_loop, w, Nt A# $\gamma$ ) (Q_loop, w,  $\alpha @ \gamma$ )
using assms pda.step1_rule[OF pda_cfg_to_pda] by (simp add: cfg_to_pda_pda_def)

lemma cfg_to_pda_cons_tms:
  pda.steps_cfg_to_pda_pda (Q_loop, w@w', map Tm w @  $\gamma$ ) (Q_loop, w',  $\gamma$ )
proof (induction w)
  case (Cons a w)
  have pda.step1_cfg_to_pda_pda (Q_loop, (a # w) @ w', map Tm (a # w) @  $\gamma$ )
  (Q_loop, w @ w', map Tm w @  $\gamma$ )
    using cfg_to_pda_cons_tm by simp
  with Cons.IH show ?case
    using pda.step1_steps[OF pda_cfg_to_pda] pda.steps_trans[OF pda_cfg_to_pda]
by blast
qed (simp add: pda.steps_refl[OF pda_cfg_to_pda])

lemma cfg_to_pda_nt_cons:
  assumes pda.step1_cfg_to_pda_pda (Q_loop, w, Nt A# $\gamma$ ) (Q_loop, w',  $\beta$ )
  shows  $\exists \alpha. (A, \alpha) \in Prods\ G \wedge \beta = \alpha @ \gamma \wedge w' = w$ 
proof -
  from assms have  $(\exists \beta_0. w' = w \wedge \beta = \beta_0 @ \gamma \wedge (Q\_loop, \beta_0) \in pda\_eps\_of\_cfg$ 
  Q_loop (Nt A))

```

$\vee (\exists a \beta_0. w = a \# w' \wedge \beta = \beta_0 @ \gamma \wedge (Q_loop, \beta_0) \in pda_of_cfg$
 $Q_loop a (Nt A))$
using $pda.step1_rule[OF pda_cfg_to_pda]$ **by** ($simp$ $add: cfg_to_pda_pda_def$)
thus $?thesis$
by ($induction Q_loop Nt A :: ('n, 't) sym rule: pda_eps_of_cfg.induct$) $auto$
qed

lemma $cfg_to_pda_tm_stack_cons$:
assumes $pda.step1_cfg_to_pda_pda (Q_loop, w, Tm a \# \beta) (Q_loop, w', \beta')$
shows $w = a \# w' \wedge \beta = \beta'$
proof –
from $assms$ **have** $(\exists \beta_0. w' = w \wedge \beta' = \beta_0 @ \beta \wedge (Q_loop, \beta_0) \in pda_eps_of_cfg$
 $Q_loop (Tm a))$
 $\vee (\exists a_0 \beta_0. w = a_0 \# w' \wedge \beta' = \beta_0 @ \beta \wedge (Q_loop, \beta_0) \in pda_of_cfg$
 $Q_loop a_0 (Tm a))$
using $pda.step1_rule[OF pda_cfg_to_pda]$ **by** ($simp$ $add: cfg_to_pda_pda_def$)
thus $?thesis$
by ($induction Q_loop Tm a :: ('n, 't) sym rule: pda_eps_of_cfg.induct, auto$)
 $(metis emptyE, metis empty_iff prod.inject singletonD)$
qed

lemma $cfg_to_pda_tm_stack_path$:
assumes $pda.steps_cfg_to_pda_pda (Q_loop, w, Tm a \# \alpha) (Q_loop, [], [])$
shows $\exists w'. w = a \# w' \wedge pda.steps_cfg_to_pda_pda (Q_loop, w', \alpha) (Q_loop,$
 $[], [])$
proof –
from $assms$ **obtain** $q' w' \alpha'$ **where** $step1: pda.step1_cfg_to_pda_pda (Q_loop,$
 $w, Tm a \# \alpha) (q', w', \alpha')$ **and**
 $steps: pda.steps_cfg_to_pda_pda (q', w', \alpha') (Q_loop,$
 $[], [])$
using $pda.steps_not_refl_split_first[OF pda_cfg_to_pda]$ **by** $blast$
have $q'_def: q' = Q_loop$
using $sing_st.exhaust$ **by** $blast$
from $step1[unfolded q'_def]$ **have** $(\exists \beta_0. w' = w \wedge \alpha' = \beta_0 @ \alpha \wedge (Q_loop, \beta_0)$
 $\in pda_eps_of_cfg Q_loop (Tm a))$
 $\vee (\exists a_0 \beta_0. w = a_0 \# w' \wedge \alpha' = \beta_0 @ \alpha \wedge (Q_loop, \beta_0) \in pda_of_cfg$
 $Q_loop a_0 (Tm a))$
using $pda.step1_rule[OF pda_cfg_to_pda]$ **by** ($simp$ $add: cfg_to_pda_pda_def$)
hence $w = a \# w' \wedge \alpha' = \alpha$
by ($induction Q_loop Tm a :: ('n, 't) sym rule: pda_eps_of_cfg.induct, auto$)
 $(metis empty_iff, metis empty_iff prod.inject singletonD)$
with $steps q'_def$ **show** $?thesis$ **by** $simp$
qed

lemma $cfg_to_pda_tms_stack_path$:
assumes $pda.steps_cfg_to_pda_pda (Q_loop, w, map Tm v @ \alpha) (Q_loop, [], [])$
shows $\exists w'. w = v @ w' \wedge pda.steps_cfg_to_pda_pda (Q_loop, w', \alpha) (Q_loop,$
 $[], [])$
using $assms$ $cfg_to_pda_tm_stack_path$ **by** ($induction v arbitrary: w$) $fastforce+$

```

lemma cfg_to_pda_accepts_if_G_derives:
  assumes Prods G  $\vdash \alpha \Rightarrow l^* \text{map } Tm \ w$ 
  shows pda.steps cfg_to_pda_pda (Q_loop, w,  $\alpha$ ) (Q_loop, [], [])
using assms proof (induction rule: converse_rtranclp_induct)
  case base
  then show ?case
    using cfg_to_pda_cons_tms[where ?w' = [] and ? $\gamma$  = []] by simp
next
  case (step y z)
  from step(1) obtain A  $\alpha$  u1 u2 where prod: (A,  $\alpha$ )  $\in$  Prods G and y_def: y =
map Tm u1 @ Nt A # u2 and z_def: z = map Tm u1 @  $\alpha$  @ u2
    using derivel_iff[of Prods G y z] by blast
  from step(3) z_def obtain w' where w_def: w = u1 @ w' and
  *: pda.steps cfg_to_pda_pda (Q_loop, w',  $\alpha$  @ u2)
(Q_loop, [], [])
  using cfg_to_pda_tms_stack_path by blast

  from w_def y_def have pda.steps cfg_to_pda_pda (Q_loop, w, y) (Q_loop, w',
Nt A # u2)
  using cfg_to_pda_cons_tms by simp

  moreover from prod have pda.steps cfg_to_pda_pda (Q_loop, w', Nt A # u2)
(Q_loop, w',  $\alpha$  @ u2)
  using cfg_to_pda_cons_nt pda.step1_steps[OF pda_cfg_to_pda] by simp

  ultimately show ?case
    using * pda.steps_trans[OF pda_cfg_to_pda] by blast
qed

lemma G_derives_if_cfg_to_pda_accepts:
  assumes pda.steps cfg_to_pda_pda (Q_loop, w,  $\alpha$ ) (Q_loop, [], [])
  shows Prods G  $\vdash \alpha \Rightarrow^* \text{map } Tm \ w$ 
using assms proof (induction (Q_loop, w,  $\alpha$ ) (Q_loop, [] :: 't list, [] :: ('n, 't)
syms)
  arbitrary: w  $\alpha$  rule: pda.steps_induct2[OF pda_cfg_to_pda])
  case ( $\exists w_1 \alpha_1 p_2 w_2 \alpha_2$ )
  then show ?case proof (cases  $\alpha_1$ )
    case (Cons Z'  $\alpha'$ )
    have p2_def: p2 = Q_loop
    using sing_st.exhaust by blast
    with  $\exists(2,3)$  have IH: Prods G  $\vdash \alpha_2 \Rightarrow^* \text{map } Tm \ w_2$  by simp
    show ?thesis proof (cases Z')
      case (Nt A)
      with Cons p2_def  $\exists(1)$  obtain  $\alpha$  where prod: (A,  $\alpha$ )  $\in$  Prods G and  $\alpha_2\_def:$ 
 $\alpha_2 = \alpha @ \alpha'$  and w2_def: w2 = w1
      using cfg_to_pda_nt_cons by blast
      from Cons Nt prod  $\alpha_2\_def$  have Prods G  $\vdash \alpha_1 \Rightarrow \alpha_2$ 
      using derive_iff by fast

```

```

    with IH w2_def show ?thesis by simp
  next
    case (Tm a)
    with Cons p2_def 3(1) have w_alpha_def: w1 = a # w2 ∧ α' = α2
      using cfg_to_pda_tm_stack_cons by simp
    from IH have Prods G ⊢ Tm a # α2 ⇒* Tm a # map Tm w2
      using derives_Cons by auto
    with Cons Tm w_alpha_def show ?thesis by simp
  qed
qed (simp add: 3(1) pda.step1_empty_stack[OF pda_cfg_to_pda])
qed (simp_all add: assms)

lemma cfg_to_pda: LangS G = pda.accept_stack cfg_to_pda_pda (is ?L = ?P)
proof
  show ?L ⊆ ?P
  proof
    fix x
    assume x ∈ ?L
    hence Prods G ⊢ [Nt (Start G)] ⇒* map Tm x
      by (simp add: Lang_def)
    hence Prods G ⊢ [Nt (Start G)] ⇒l* map Tm x
      using derives_iff_derives by auto
    hence pda.steps cfg_to_pda_pda (Q_loop, x, [Nt (Start G)]) (Q_loop, [], [])
      using cfg_to_pda_accepts_if_G_derives by simp
    thus x ∈ ?P
  unfolding pda.accept_stack_def[OF pda_cfg_to_pda] by (auto simp: cfg_to_pda_pda_def)
  qed
next
  show ?P ⊆ ?L
  proof
    fix x
    assume x ∈ ?P
    then obtain q where steps: pda.steps cfg_to_pda_pda (Q_loop, x, [Nt (Start
G)]) (q, [], [])
      unfolding pda.accept_stack_def[OF pda_cfg_to_pda] by (auto simp: cfg_to_pda_pda_def)
    have q = Q_loop
      using sing_st.exhaust by blast
    with steps have Prods G ⊢ [Nt (Start G)] ⇒* map Tm x
      using G_derives_if_cfg_to_pda_accepts by simp
    thus x ∈ ?L
      by (simp add: Lang_def)
  qed
qed
end
end

```

3.2 PDA to CFG

Starting from a PDA that accepts by empty stack, we construct an equivalent CFG. The formalization is based on the Lean formalization by Leichtfried[2].

theory *PDA_To_CFG*

imports

Pushdown_Automata

Context_Free_Grammar.Context_Free_Grammar

begin

datatype $(q, 's)$ *pda_nt* = *Start_sym* | *Single_sym* $'q$ $'s$ $'q$ | *List_sym* $'q$ $'s$ *list* $'q$

context *pda* **begin**

abbreviation *all_pushes* :: $'s$ *list* **set** **where**

$all_pushes \equiv \{\alpha. \exists p q a z. (p, \alpha) \in \delta M q a z\} \cup \{\alpha. \exists p q z. (p, \alpha) \in \delta \varepsilon M q z\}$

abbreviation *max_push* :: *nat* **where**

$max_push \equiv Suc (Max (length \text{ 'all_pushes}))$

abbreviation *is_allowed_nt* :: $(q, 's)$ *pda_nt* **set** **where**

$is_allowed_nt \equiv \{List_sym p \alpha q \mid p \alpha q. length \alpha \leq max_push\} \cup (\bigcup p Z q. \{Single_sym p Z q\}) \cup \{Start_sym\}$

abbreviation *empty_rule* :: $'q \Rightarrow ((q, 's)$ *pda_nt*, $'a)$ *Prods* **where**

$empty_rule q \equiv \{(List_sym q [] q, [])\}$

abbreviation *trans_rule* :: $'q \Rightarrow 'q \Rightarrow 'a \Rightarrow 's \Rightarrow ((q, 's)$ *pda_nt*, $'a)$ *Prods* **where**

$trans_rule q_0 q_1 a Z \equiv (\lambda(p, \alpha). (Single_sym q_0 Z q_1, [Tm a, Nt (List_sym p \alpha q_1)])) \text{ ' } \delta M q_0 a Z$

abbreviation *eps_rule* :: $'q \Rightarrow 'q \Rightarrow 's \Rightarrow ((q, 's)$ *pda_nt*, $'a)$ *Prods* **where**

$eps_rule q_0 q_1 Z \equiv (\lambda(p, \alpha). (Single_sym q_0 Z q_1, [Nt (List_sym p \alpha q_1)])) \text{ ' } \delta \varepsilon M q_0 Z$

fun *split_rule* :: $'q \Rightarrow (q, 's)$ *pda_nt* $\Rightarrow ((q, 's)$ *pda_nt*, $'a)$ *Prods* **where**

$split_rule q (List_sym p_0 (Z\#\alpha) p_1) = \{(List_sym p_0 (Z\#\alpha) p_1, [Nt (Single_sym p_0 Z q), Nt (List_sym q \alpha p_1)])\}$
| *split_rule* $_ _ = \{\}$

abbreviation *start_rule* :: $'q \Rightarrow ((q, 's)$ *pda_nt*, $'a)$ *Prods* **where**

$start_rule q \equiv \{(Start_sym, [Nt (List_sym (init_state M) [init_symbol M] q)])\}$

abbreviation *rule_set* :: $((q, 's)$ *pda_nt*, $'a)$ *Prods* **where**

$rule_set \equiv (\bigcup q. empty_rule q) \cup (\bigcup q p a Z. trans_rule q p a Z) \cup (\bigcup q p Z. eps_rule q p Z) \cup$

$$\bigcup \{ \text{split_rule } q \text{ nt} \mid q \text{ nt. nt} \in \text{is_allowed_nt} \} \cup (\bigcup q. \text{start_rule } q)$$

definition $G :: (('q, 's) \text{ pda_nt}, 'a) \text{ Cfg}$ **where**

$$G \equiv \text{Cfg rule_set Start_sym}$$

lemma $\text{finite_is_allowed_nt}$: $\text{finite (is_allowed_nt)}$

proof (intro finite_UnI)

show $\text{finite } \{ \text{List_sym } (p :: 'q) (\alpha :: 's \text{ list}) q \mid p \alpha q. \text{length } \alpha \leq \text{max_push} \}$

proof –

let $?A = \bigcup (\bigcup ((\lambda s. (\lambda f. f \text{ ' UNIV}) ' s) ' ((\lambda f. f \text{ ' } \{xs :: 's \text{ list. set } xs \subseteq \text{UNIV} \wedge \text{length } xs \leq \text{max_push}\}) ' (\text{List_sym } ' (\text{UNIV} :: 'q \text{ set}))))))$

have $\{ \text{List_sym } p \alpha q \mid p \alpha q. \text{length } \alpha \leq \text{max_push} \} = ?A$
by auto

moreover have $\text{finite } ?A$ (**is** $\text{finite } (\bigcup ?B)$)

proof (rule finite_Union)

show $\text{finite } ?B$ (**is** $\text{finite } (\bigcup ?C)$)

proof (rule finite_Union)

show $\text{finite } ?C$ **by** simp

next

show $\bigwedge M. M \in ?C \implies \text{finite } M$

using $\text{finite_lists_length_le}$ [of UNIV max_push] **by** force

qed

next

show $\bigwedge M. M \in ?B \implies \text{finite } M$ **by** fastforce

qed

ultimately show $?thesis$ **by** simp

qed

next

show $\text{finite } (\bigcup (p :: 'q) (Z :: 's) q. \{ \text{Single_sym } p Z q \})$

by (rule, simp) $+$

qed simp

lemma finite_split_rule : $\text{finite (split_rule } q \text{ nt)}$

by ($\text{induction } q \text{ nt rule: split_rule.induct}$) auto

lemma $\text{finite (Prods } G)$

proof –

have $\text{finite } (\bigcup q. \text{empty_rule } q)$ **by** simp

moreover have $\text{finite } (\bigcup q p a Z. \text{trans_rule } q p a Z)$

by ($\text{simp add: finite_delta}$)

moreover have $\text{finite } (\bigcup q p Z. \text{eps_rule } q p Z)$

by ($\text{simp add: finite_delta_eps}$)

moreover have $\text{finite } (\bigcup \{ \text{split_rule } q \text{ nt} \mid q \text{ nt. nt} \in \text{is_allowed_nt} \})$

proof –
have $\{split_rule\ q\ nt \mid q\ nt.\ nt \in is_allowed_nt\} = \bigcup ((\lambda f. f\ 'is_allowed_nt)$
 $'(split_rule\ 'UNIV))$
by *fastforce*

moreover have *finite* $(\bigcup (\bigcup ((\lambda f. f\ 'is_allowed_nt)\ ' (split_rule\ 'UNIV))))$
(is *finite* $(\bigcup ?A)$)
proof (*rule* *finite_Union*)
show *finite* $?A$ **(is** *finite* $(\bigcup ?B)$)
proof (*rule* *finite_Union*)
show *finite* $?B$ **by** *simp*
next
show $\bigwedge M. M \in ?B \implies finite\ M$
using *finite_is_allowed_nt* **by** *blast*
qed
next
show $\bigwedge M. M \in ?A \implies finite\ M$
by (*auto simp: finite_split_rule*)
qed

ultimately show $?thesis$ **by** *simp*
qed

moreover have *finite* $(\bigcup q. start_rule\ q)$ **by** *simp*

ultimately show $?thesis$
by (*simp add: G_def*)
qed

lemma *split_rule_simp*:
 $(A, w) \in split_rule\ q\ nt \iff$
 $(\exists p_0\ Z\ \alpha\ p_1. nt = (List_sym\ p_0\ (Z\#\alpha)\ p_1) \wedge$
 $A = List_sym\ p_0\ (Z\#\alpha)\ p_1 \wedge w = [Nt\ (Single_sym\ p_0\ Z\ q), Nt$
 $(List_sym\ q\ \alpha\ p_1)])$
by (*induction q nt rule: split_rule.induct*) *auto*

lemma *pda_to_cfg_derive_empty*:
 $Prods\ G \vdash [Nt\ (List_sym\ p_1\ []\ p_2)] \implies x \iff p_2 = p_1 \wedge x = []$
unfolding *G_def* **using** *derive_singleton[of rule_set]* *split_rule_simp* **by** *auto*

lemma *finite_all_pushes*: *finite all_pushes*
proof –
let $?A = (\lambda(p, \alpha). \alpha)\ '(\bigcup q\ a\ Z. \delta\ M\ q\ a\ Z \cup (\bigcup q\ Z. \delta\varepsilon\ M\ q\ Z))$
have *all_pushes* = $?A$ **by** *fast*

moreover have *finite* $?A$
by (*rule, simp add: finite_delta finite_delta_eps*) $+$

ultimately show $?thesis$ **by** *simp*

qed

lemma *push_trans_leq_max*:

$(p, \alpha) \in \delta M q a Z \implies \text{length } \alpha \leq \text{max_push}$

proof –

have $(p, \alpha) \in \delta M q a Z \implies \text{length } \alpha \leq \text{Max} (\text{length } \text{'all_pushes})$

by (rule *Max_ge*) (use *finite_all_pushes* in *blast*)+

thus $(p, \alpha) \in \delta M q a Z \implies \text{length } \alpha \leq \text{max_push}$ **by** *simp*

qed

lemma *push_eps_leq_max*:

$(p, \alpha) \in \delta\varepsilon M q Z \implies \text{length } \alpha \leq \text{max_push}$

proof –

have $(p, \alpha) \in \delta\varepsilon M q Z \implies \text{length } \alpha \leq \text{Max} (\text{length } \text{'all_pushes})$

by (rule *Max_ge*) (use *finite_all_pushes* in *blast*)+

thus $(p, \alpha) \in \delta\varepsilon M q Z \implies \text{length } \alpha \leq \text{max_push}$ **by** *simp*

qed

lemma *pda_to_cfg_derive_split*:

$\text{Prods } G \vdash [\text{Nt} (\text{List_sym } p_1 (Z\#\alpha) p_2)] \Rightarrow w \longleftrightarrow$
 $(\exists q. \text{length} (Z\#\alpha) \leq \text{max_push} \wedge w = [\text{Nt} (\text{Single_sym } p_1 Z q), \text{Nt} (\text{List_sym}$
 $q \alpha p_2)])$
(is ?l \longleftrightarrow ?r)

proof

assume ?l

hence $(\text{List_sym } p_1 (Z \# \alpha) p_2, w) \in \text{rule_set}$

using *derive_singleton*[of *Prods G Nt (List_sym p1 (Z # alpha) p2) w*] **by** (*simp*
add: G_def)

thus ?r

by (*auto simp: split_rule_simp*)

next

assume ?r

then obtain *q* **where** *len_alpha*: $\text{length} (Z\#\alpha) \leq \text{max_push}$ **and** *w_def*: $w = [\text{Nt}$
 $(\text{Single_sym } p_1 Z q), \text{Nt} (\text{List_sym } q \alpha p_2)]$ **by** *blast*

from *w_def* **have** $(\text{List_sym } p_1 (Z\#\alpha) p_2, w) \in \text{split_rule } q (\text{List_sym } p_1 (Z$
 $\# \alpha) p_2)$ **by** *simp*

with *len_alpha* **have** $(\text{List_sym } p_1 (Z\#\alpha) p_2, w) \in \bigcup \{\text{split_rule } q \text{ nt} \mid q \text{ nt. nt} \in$
is_allowed_nt\}

by (*subst Union_iff*) *fast*

hence $(\text{List_sym } p_1 (Z\#\alpha) p_2, w) \in \text{rule_set}$ **by** *simp*

thus ?l

using *derive_singleton*[of *Prods G Nt (List_sym p1 (Z # alpha) p2) w*] **by** (*simp*
add: G_def)

qed

lemma *pda_to_cfg_derive_single*:

$\text{Prods } G \vdash [\text{Nt} (\text{Single_sym } q_0 Z q_1)] \Rightarrow w \longleftrightarrow$
 $(\exists p \alpha a. (p, \alpha) \in \delta M q_0 a Z \wedge w = [\text{Tm } a, \text{Nt} (\text{List_sym } p \alpha q_1)]) \vee$
 $(\exists p \alpha. (p, \alpha) \in \delta\varepsilon M q_0 Z \wedge w = [\text{Nt} (\text{List_sym } p \alpha q_1)])$

unfolding G_def **using** $derive_singleton[of\ rule_set]$ $split_rule_simp$ **by** $fast_force$

lemma $pda_to_cfg_derive_start$:

$Prods\ G \vdash [Nt\ Start_sym] \Rightarrow w \longleftrightarrow (\exists q. w = [Nt\ (List_sym\ (init_state\ M)\ [init_symbol\ M]\ q)])$

unfolding G_def **using** $derive_singleton[of\ rule_set]$ $split_rule_simp$ **by** $auto$

lemma $pda_to_cfg_derives_if_stepn$:

assumes $(q, x, \gamma) \rightsquigarrow(n) (p, [], [])$

and $length\ \gamma \leq max_push$

shows $Prods\ G \vdash [Nt\ (List_sym\ q\ \gamma\ p)] \Rightarrow^* map\ Tm\ x$

using $assms$ **proof** ($induction\ n$ $arbitrary: x\ p\ q\ \gamma$ $rule: less_induct$)

case ($less\ n$)

then show $?case$ **proof** ($cases\ \gamma$)

case Nil

from $less(2)$ **have** $(q, x, \gamma) \rightsquigarrow^* (p, [], [])$

using $stepn_steps$ **by** $blast$

with Nil **have** $q = p \wedge x = []$

using $steps_empty_stack$ **by** $simp$

with Nil **show** $?thesis$

using $pda_to_cfg_derive_empty$ **by** $auto$

next

case ($Cons\ Z\ \alpha$)

with $less(2)$ **obtain** $n'\ q'\ x'\ \gamma'$ **where** $n_def: n = Suc\ n'$ **and**

$step1: (q, x, \gamma) \rightsquigarrow (q', x', \gamma')$ **and**

$stepn: (q', x', \gamma') \rightsquigarrow(n') (p, [], [])$

using $stepn_not_refl_split_first$ **by** $blast$

from $Cons\ step1$ **have** $rule: (\exists \beta. x' = x \wedge \gamma' = \beta @ \alpha \wedge (q', \beta) \in \delta \varepsilon\ M\ q\ Z)$

$\vee (\exists a\ \beta. x = a \# x' \wedge \gamma' = \beta @ \alpha \wedge (q', \beta) \in \delta\ M\ q\ a\ Z)$ (**is**

$?l \vee ?r$)

using $step1_rule$ **by** $simp$

show $?thesis$ **proof** ($rule\ disjE[OF\ rule]$)

assume $?l$

then obtain β **where** $x_def: x' = x$ **and** $\gamma'_split: \gamma' = \beta @ \alpha$ **and** $eps: (q', \beta) \in \delta \varepsilon\ M\ q\ Z$ **by** $blast$

from $stepn\ \gamma'_split$ **obtain** $p'\ m_1\ m_2\ y\ y'$ **where** $x'_def: x' = y @ y'$ **and** $m1_m2_n': m_1 + m_2 = n'$

and $stepm1: stepn\ m_1\ (q', y, \beta)\ (p', [], [])$ **and** $stepm2: stepn\ m_2\ (p', y', \alpha)\ (p, [], [])$

using $split_stack[of\ n'\ q'\ x'\ \beta\ \alpha\ p]$ **by** $blast$

from $n_def\ m1_m2_n'$ **have** $m1_less_n: m_1 < n$ **by** $simp$

from $n_def\ m1_m2_n'$ **have** $m2_less_n: m_2 < n$ **by** $simp$

from eps **have** $len_beta: length\ \beta \leq max_push$

using $push_eps_leq_max$ **by** $blast$

from $Cons\ less(3)$ **have** $Prods\ G \vdash [Nt\ (List_sym\ q\ \gamma\ p)] \Rightarrow [Nt\ (Single_sym\ q\ Z\ p'), Nt\ (List_sym\ p'\ \alpha\ p)]$

using $pda_to_cfg_derive_split$ **by** $simp$

moreover from *eps* **have** $\text{Prods } G \vdash [\text{Nt } (\text{Single_sym } q \ Z \ p'), \text{Nt } (\text{List_sym } p' \ \alpha \ p)] \Rightarrow$
 $[\text{Nt } (\text{List_sym } q' \ \beta \ p'), \text{Nt } (\text{List_sym } p' \ \alpha \ p)]$
using *pda_to_cfg_derive_single* *derive_append*[of *Prods G* [*Nt (Single_sym q Z p')*] [*Nt (List_sym q' beta p')*]] [*Nt (List_sym p' alpha p)*] **by** *simp*

moreover have $\text{Prods } G \vdash [\text{Nt } (\text{List_sym } q' \ \beta \ p'), \text{Nt } (\text{List_sym } p' \ \alpha \ p)]$
 $\Rightarrow^* \text{map } \text{Tm } y \ @ \ [\text{Nt } (\text{List_sym } p' \ \alpha \ p)]$
using *derives_append*[*OF less(1)* [*OF m1_less_n stepm1 len_beta*]] **by** *simp*

moreover from *x_def x'_def* *Cons less(3)* **have** $\text{Prods } G \vdash \text{map } \text{Tm } y \ @$
 $[\text{Nt } (\text{List_sym } p' \ \alpha \ p)] \Rightarrow^* \text{map } \text{Tm } x$
using *derives_prepend*[*OF less(1)* [*OF m2_less_n stepm2*]] **by** *auto*

ultimately show *?thesis* **by** *simp*
next
assume *?r*
then obtain *a beta* **where** *x_def: x = a # x'* **and** *gamma'_split: gamma' = beta @ alpha* **and**
trans: (q', beta) in delta M q a Z **by** *blast*
from *stepn gamma'_split* **obtain** *p' m1 m2 y y'* **where** *x'_def: x' = y @ y'* **and**
m1_m2_n': m1 + m2 = n'
and *stepm1: stepn m1 (q', y, beta) (p', [], [])* **and** *stepm2: stepn m2*
 $(p', y', \alpha) (p, [], [])$
using *split_stack*[of *n' q' x' beta alpha p*] **by** *blast*
from *n_def m1_m2_n'* **have** *m1_less_n: m1 < n* **by** *simp*
from *n_def m1_m2_n'* **have** *m2_less_n: m2 < n* **by** *simp*
from *trans* **have** *len_beta: length beta <= max_push*
using *push_trans_leq_max* **by** *blast*

from *Cons less(3)* **have** $\text{Prods } G \vdash [\text{Nt } (\text{List_sym } q \ \gamma \ p)] \Rightarrow [\text{Nt } (\text{Single_sym } q \ Z \ p'), \text{Nt } (\text{List_sym } p' \ \alpha \ p)]$
using *pda_to_cfg_derive_split* **by** *simp*

moreover from *trans* **have** $\text{Prods } G \vdash [\text{Nt } (\text{Single_sym } q \ Z \ p'), \text{Nt } (\text{List_sym } p' \ \alpha \ p)] \Rightarrow$
 $[\text{Tm } a, \text{Nt } (\text{List_sym } q' \ \beta \ p'), \text{Nt } (\text{List_sym } p' \ \alpha \ p)]$
using *pda_to_cfg_derive_single* *derive_append*[of *Prods G* [*Nt (Single_sym q Z p')*] [*Tm a, Nt (List_sym q' beta p')*]] [*Nt (List_sym p' alpha p)*] **by** *simp*

moreover have $\text{Prods } G \vdash [\text{Tm } a, \text{Nt } (\text{List_sym } q' \ \beta \ p'), \text{Nt } (\text{List_sym } p' \ \alpha \ p)] \Rightarrow^*$
 $\text{Tm } a \ # \ \text{map } \text{Tm } y \ @ \ [\text{Nt } (\text{List_sym } p' \ \alpha \ p)]$
using *derives_append*[*OF less(1)* [*OF m1_less_n stepm1 len_beta*]] **by** (*simp*
add: derives_Tm_Cons)

moreover from *x'_def x_def* *Cons less(3)* **have** $\text{Prods } G \vdash \text{Tm } a \ # \ \text{map}$

Tm y @ [*Nt (List_sym p' α p)*] ⇒* *map Tm x*
using *derives_prepend*[*OF less(1)*][*OF m2_less_n stepm2*], of *Tm a # map*
Tm y] **by** *simp*

ultimately show *?thesis* **by** *simp*
qed
qed
qed

lemma *pda_to_cfg_steps_if_derivel*:

assumes *Prods G* ⊢ [*Nt (List_sym q γ p)*] ⇒*l(n)* *map Tm x*

shows (*q, x, γ*) ⇝* (*p, [], []*)

using *assms* **proof** (*induction n arbitrary: x p q γ rule: less_induct*)

case (*less n*)

then show *?case* **proof** (*cases γ*)

case *Nil*

have *derives: Prods G* ⊢ [*Nt (List_sym q γ p)*] ⇒* *map Tm x*

using *derives_imp_derives*[*OF relpowp_imp_rtranclp*][*OF less(2)*]

have *p = q* ∧ *x = []*

proof –

from *derives_start1*[*OF derives*] **obtain** *α* **where** *d1: Prods G* ⊢ [*Nt*
(*List_sym q γ p*)] ⇒ *α* **and**

ds: Prods G ⊢ *α* ⇒* *map Tm x*

using *derive_singleton* **by** *blast*

from *Nil d1* **have** ***: *p = q* **and** *α_def: α = []*

using *pda_to_cfg_derive_empty* **by** *simp_all*

from *α_def ds* **have** ****: *x = []* **by** *simp*

from ** *** **show** *?thesis* **by** *simp*

qed

with *Nil* **show** *?thesis*

by (*simp add: steps_refl*)

next

case (*Cons Z α*)

from *less(2)* **have** *n > 0*

using *gr0I* **by** *fastforce*

then obtain *n'* **where** *n_def: n = Suc n'*

using *not0_implies_Suc* **by** *blast*

with *less(2)* **obtain** *γ'* **where** *l1: Prods G* ⊢ [*Nt (List_sym q γ p)*] ⇒*l* *γ'* **and**
ln': Prods G ⊢ *γ' ⇒l(n')* *map Tm x*

using *relpowp_Suc_E2*[of *n'* *derivel (Prods G) [Nt (List_sym q γ p)] map*
Tm x] **by** *blast*

from *Cons* **obtain** *q'* **where** *γ'_def: γ' = [Nt (Single_sym q Z q'), Nt*
(*List_sym q' α p*)]

using *pda_to_cfg_derive_split* *derivel_imp_derive*[*OF l1*] **by** *blast*

with *ln'* **have** *n' > 0*

using *gr0I* **by** *fastforce*

then obtain *n''* **where** *n'_def: n' = Suc n''*

using *not0_implies_Suc* **by** *blast*

with *ln' γ'_def* **obtain** *γ''* **where** *l2: Prods G* ⊢ [*Nt (Single_sym q Z q'), Nt*

$(List_sym\ q'\ \alpha\ p)] \Rightarrow l\ \gamma''$ **and**
 $ln'': Prods\ G \vdash \gamma'' \Rightarrow l(n'')\ map\ Tm\ x$
using $relpowp_Suc_E2$ [of $n''\ derivel\ (Prods\ G)\ [Nt\ (Single_sym\ q\ Z\ q')]$, $Nt\ (List_sym\ q'\ \alpha\ p)]\ map\ Tm\ x$ **by** $blast$
from $l2$ **obtain** γ''_2 **where** $l2': Prods\ G \vdash [Nt\ (Single_sym\ q\ Z\ q')] \Rightarrow l\ \gamma''_2$
and $\gamma''_split: \gamma'' = \gamma''_2\ @\ [Nt\ (List_sym\ q'\ \alpha\ p)]$
using $derivel_Nt_Cons$ **by** ($metis\ append.right_neutral$)
have $(\exists q''\ \alpha''\ a.\ (q'',\ \alpha'') \in \delta\ M\ q\ a\ Z \wedge \gamma''_2 = [Tm\ a,\ Nt\ (List_sym\ q''\ \alpha''\ q')]) \vee$
 $(\exists q''\ \alpha''.\ (q'',\ \alpha'') \in \delta\varepsilon\ M\ q\ Z \wedge \gamma''_2 = [Nt\ (List_sym\ q''\ \alpha''\ q')])$
using $pda_to_cfg_derive_single\ derivel_imp_derive$ [OF $l2'$] **by** $simp$
with γ''_split **have** $rule: (\exists q''\ \alpha''\ a.\ (q'',\ \alpha'') \in \delta\ M\ q\ a\ Z \wedge$
 $\gamma'' = [Tm\ a,\ Nt\ (List_sym\ q''\ \alpha''\ q')],\ Nt\ (List_sym\ q'$
 $\alpha\ p)) \vee$
 $(\exists q''\ \alpha''.\ (q'',\ \alpha'') \in \delta\varepsilon\ M\ q\ Z \wedge$
 $\gamma'' = [Nt\ (List_sym\ q''\ \alpha''\ q')],\ Nt\ (List_sym\ q'\ \alpha\ p))$
(is $?l \vee ?r$) **by** $simp$
show $?thesis$ **proof** ($rule\ disjE$ [OF $rule$])
assume $?l$
then **obtain** $q''\ \alpha''\ a$ **where** $trans: (q'',\ \alpha'') \in \delta\ M\ q\ a\ Z$ **and**
 $\gamma''_def: \gamma'' = [Tm\ a,\ Nt\ (List_sym\ q''\ \alpha''\ q')],\ Nt$
 $(List_sym\ q'\ \alpha\ p)]$ **by** $blast$
from $\gamma''_def\ ln''$ **obtain** x' **where** $x_def: x = a\ \#\ x'$ **and**
 $split: Prods\ G \vdash [Nt\ (List_sym\ q''\ \alpha''\ q')],\ Nt\ (List_sym$
 $q'\ \alpha\ p)] \Rightarrow l(n'')\ map\ Tm\ x'$
using $deriveln_Tm_Cons$ [of $n''\ Prods\ G\ a\ [Nt\ (List_sym\ q''\ \alpha''\ q')]$, $Nt\ (List_sym\ q'\ \alpha\ p)]\ map\ Tm\ x$ **by** $auto$
obtain $w_1\ w_2\ m_1\ m_2$ **where** $m1_m2_n''': m_1 + m_2 = n''$ **and** $x'_def: x'$
 $= w_1\ @\ w_2$
and $m1_path: Prods\ G \vdash [Nt\ (List_sym\ q''\ \alpha''\ q')]$
 $\Rightarrow l(m_1)\ map\ Tm\ w_1$
and $m2_path: Prods\ G \vdash [Nt\ (List_sym\ q'\ \alpha\ p)]$
 $\Rightarrow l(m_2)\ map\ Tm\ w_2$
using $deriveln_Cons_TmsD$ [OF $split$] **by** $blast$
from $m1_m2_n'''$ $n_def\ n'_def$ **have** $m1_lessn: m_1 < n$ **by** $simp$
from $m1_m2_n'''$ $n_def\ n'_def$ **have** $m2_lessn: m_2 < n$ **by** $simp$

from $trans\ x_def\ Cons$ **have** $(q,\ x,\ \gamma) \rightsquigarrow (q'',\ x',\ \alpha''\ @\ \alpha)$
using $step1_rule$ **by** $simp$

moreover **from** x'_def **have** $(q'',\ x',\ \alpha''\ @\ \alpha) \rightsquigarrow^* (q',\ w_2,\ \alpha)$
using $steps_stack_app$ [OF $less(1)$][OF $m1_lessn\ m1_path$], of α
 $steps_word_app$ [of $q''\ w_1\ \alpha''\ @\ \alpha\ q'\ []\ \alpha\ w_2$] **by** $simp$

moreover **have** $(q',\ w_2,\ \alpha) \rightsquigarrow^* (p,\ [],\ [])$
using $less(1)$ [OF $m2_lessn\ m2_path$].

ultimately **show** $?thesis$
unfolding $steps_def$

by (*meson converse_rtranclp_into_rtranclp rtranclp_trans*)
next
assume *?r*
then obtain $q'' \alpha''$ **where** $eps: (q'', \alpha'') \in \delta\varepsilon M q Z$ **and**
 $\gamma''_def: \gamma'' = [Nt (List_sym q'' \alpha'' q'), Nt (List_sym$
 $q' \alpha p)]$ **by** *blast*
from γ''_def **have** $split: Prods G \vdash [Nt (List_sym q'' \alpha'' q'), Nt (List_sym$
 $q' \alpha p)] \Rightarrow l(n'')$ **map** $Tm x$ **by** *simp*
obtain $w_1 w_2 m_1 m_2$ **where** $m1_m2_n''': m_1 + m_2 = n''$ **and** $x_def: x =$
 $w_1 @ w_2$
and $m1_path: Prods G \vdash [Nt (List_sym q'' \alpha'' q')] \Rightarrow l(m_1)$
 $map Tm w_1$
and $m2_path: Prods G \vdash [Nt (List_sym q' \alpha p)] \Rightarrow l(m_2)$
 $map Tm w_2$
using *deriveln_Cons_TmsD[OF split]* **by** *blast*
from $m1_m2_n''' n_def n'_def$ **have** $m1_lessn: m_1 < n$ **by** *simp*
from $m1_m2_n''' n_def n'_def$ **have** $m2_lessn: m_2 < n$ **by** *simp*

from $eps Cons$ **have** $(q, x, \gamma) \rightsquigarrow (q'', x, \alpha'' @ \alpha)$
using *step1_rule* **by** *simp*

moreover from x_def **have** $(q'', x, \alpha'' @ \alpha) \rightsquigarrow^* (q', w_2, \alpha)$
using *steps_stack_app[OF less(1)[OF m1_lessn m1_path], of \alpha]*
 $steps_word_app[of q'' w_1 \alpha'' @ \alpha q' [] \alpha w_2]$ **by** *simp*

moreover have $(q', w_2, \alpha) \rightsquigarrow^* (p, [], [])$
using *less(1)[OF m2_lessn m2_path]* .

ultimately show *?thesis*
using *step1_steps steps_trans* **by** *metis*
qed
qed
qed

lemma *pda_to_cfg: LangS G = accept_stack (is ?L = ?P)*
proof
show $?L \subseteq ?P$
proof
fix x
assume $x \in ?L$
hence derives: $Prods G \vdash [Nt Start_sym] \Rightarrow^* map Tm x$
by (*simp add: G_def Lang_def*)
then obtain γ **where** $fs: Prods G \vdash [Nt Start_sym] \Rightarrow \gamma$ **and** $ls: Prods G \vdash$
 $\gamma \Rightarrow^* map Tm x$
using *converse_rtranclpE[OF derives]* **by** *blast*
from fs **obtain** q **where** $\gamma = [Nt (List_sym (init_state M) [init_symbol M]$
 $q)]$
using *pda_to_cfg_derive_start[of \gamma]* **by** *blast*
with ls **obtain** n **where** $Prods G \vdash [Nt (List_sym (init_state M) [init_symbol$

```

M] q]]  $\Rightarrow$ l(n) map Tm x
  using derives_iff_derives[of Prods G  $\gamma$  x] rtranclp_power[of derivel (Prods
G)  $\gamma$  map Tm x] by blast
  hence steps (init_state M, x, [init_symbol M]) (q, [], [])
  using pda_to_cfg_steps_if_derivel by simp
  thus  $x \in ?P$ 
  by (auto simp: accept_stack_def)
qed
next
show  $?P \subseteq ?L$ 
proof
fix x
assume  $x \in ?P$ 
then obtain q where steps (init_state M, x, [init_symbol M]) (q, [], [])
  by (auto simp: accept_stack_def)
then obtain n where (init_state M, x, [init_symbol M])  $\rightsquigarrow$ (n) (q, [], [])
  using steppn_steps by blast

  hence Prods G  $\vdash$  [Nt (List_sym (init_state M) [init_symbol M] q)]  $\Rightarrow^*$  map
Tm x
  using pda_to_cfg_derives_if_steppn by simp

  moreover have Prods G  $\vdash$  [Nt Start_sym]  $\Rightarrow$  [Nt (List_sym (init_state M)
[init_symbol M] q)]
  using pda_to_cfg_derive_start by simp

  ultimately have Prods G  $\vdash$  [Nt (Start G)]  $\Rightarrow^*$  map Tm x
  by (simp add: G_def)

  thus  $x \in ?L$ 
  by (simp add: Lang_def)
qed
qed
end
end

```

References

- [1] D. C. Kozen. *Automata and Computability*. Springer, 2007.
- [2] T. Leichtfried. autth. <https://github.com/shetzl/autth/tree/PDA/autth>, 2025. Accessed: 2025-09-28.