Pushdown Automata

Kaan Taskin and Tobias Nipkow

October 17, 2025

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)

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 $\delta_1 : Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$ and $\delta_2 : Q \times \Gamma \to Q \times \Gamma^*$, 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* (δ)
notation *delta_eps* ($\delta\varepsilon$)

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

fun *step*$_1$ :: $'q \times 'a\ list \times 's\ list \Rightarrow 'q \times 'a\ list \times 's\ list \Rightarrow bool$
((_ $\rightsquigarrow$ _) [50, 50] 50) **where**
(p_1, w_1, α_1) $\rightsquigarrow$ (p_2, w_2, α_2) $\longleftrightarrow$ (p_2, w_2, α_2) $\in$ *step* (p_1, w_1, α_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 $\equiv$ *step*$_1$ $\widehat{}*$

inductive *stepn* :: *nat* $\Rightarrow$ $'q \times 'a\ list \times 's\ list \Rightarrow 'q \times 'a\ list \times 's\ list \Rightarrow bool$
where
refl$_n$: *stepn* 0 (p, w, α) (p, w, α) |
step$_n$: *stepn* n (p_1, w_1, α_1) (p_2, w_2, α_2) $\Longrightarrow$ *step*$_1$ (p_2, w_2, α_2) (p_3, w_3, α_3) $\Longrightarrow$
stepn (*Suc n*) (p_1, w_1, α_1) (p_3, w_3, α_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* (δ 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* $\{(q, w, \beta\ @\ \alpha)\ |q\ \beta.\ (q, \beta) \in \delta$ M p a $Z\}$ (**is** *finite ?A*)
using *bij_betw_finite*[*of* $\lambda(q,\beta).\ (q, w, \beta@\alpha)$ δ M p a Z *?A*] **by** (*fastforce simp add*: *bij_betw_def inj_on_def finite_delta*)

lemma *finite_delta_eps_step*: *finite* $\{(q, w, \beta\ @\ \alpha)\ |q\ \beta.\ (q, \beta) \in \delta\varepsilon$ 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* (δ 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 $step_1$*_nonempty_stack*: $(p_1, w_1, \alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \Longrightarrow \exists\, Z'\ \alpha'.\ \alpha_1 = Z'\#\alpha'$
by (*cases* α_1) *auto*

lemma $step_1$*_empty_stack*: $\neg\ (p_1, w_1, []) \rightsquigarrow (p_2, w_2, \alpha_2)$
by *simp*

lemma $step_1$*_rule*: $(p_1, w_1, Z\#\alpha_1) \rightsquigarrow (p_2, w_2, \alpha_2) \longleftrightarrow (\exists\, \beta.\ w_2 = w_1 \wedge \alpha_2 =$

$\beta@\alpha_1 \wedge (p_2,\ \beta) \in \delta\varepsilon\ M\ p_1\ Z)$
$\vee\ (\exists\, a\ \beta.\ w_1 = a\ \#\ w_2 \wedge \alpha_2 = \beta@\alpha_1 \wedge$
$(p_2,\ \beta) \in \delta\ M\ p_1\ a\ Z)$
by (*cases* w_1) *auto*

lemma *step*$_1$*_rule_ext*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p_2,\ w_2,\ \alpha_2) \longleftrightarrow (\exists\, Z'\ \alpha'.\ \alpha_1 = Z'\#\alpha' \wedge$
$((\exists\, \beta.\ w_2 = w_1 \wedge \alpha_2 = \beta@\alpha' \wedge (p_2,\ \beta) \in \delta\varepsilon\ M\ p_1\ Z')$
$\vee\ (\exists\, a\ \beta.\ w_1 = a\ \#\ w_2 \wedge \alpha_2 = \beta@\alpha' \wedge$
$(p_2,\ \beta) \in \delta\ M\ p_1\ a\ Z')))$ (**is** *?l* $\longleftrightarrow$ *?r*)
apply (*rule iffI*)
apply (*metis step*$_1$*_nonempty_stack step*$_1$*_rule*)
apply (*use step*$_1$*_rule* **in** *force*)
done

lemma *step*$_1$*_stack_app*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_1,\ w_1,\ \alpha_1\ @\ \gamma) \rightsquigarrow$
$(p_2,\ w_2,\ \alpha_2\ @\ \gamma)$
using *step*$_1$*_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*: $[\![\ (p_1,\ w_1,\ \alpha_1) \rightsquigarrow\!* (p_2,\ w_2,\ \alpha_2);\ (p_2,\ w_2,\ \alpha_2) \rightsquigarrow\!* (p_3,\ w_3,\ \alpha_3)\ ]\!]$
$\Longrightarrow (p_1,\ w_1,\ \alpha_1) \rightsquigarrow\!* (p_3,\ w_3,\ \alpha_3)$
unfolding *steps_def* **using** *rtranclp_trans*[**where** *?r* = *step*$_1$] **by** *blast*

lemma *step*$_1$*_steps*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_1,\ w_1,\ \alpha_1) \rightsquigarrow\!* (p_2,\ w_2,\ \alpha_2)$
by (*simp add*: *steps_def r_into_rtranclp*)

lemma *steps_empty_stack*: $(p_1,\ w_1,\ []) \rightsquigarrow\!* (p_2,\ w_2,\ \alpha_2) \Longrightarrow p_1 = p_2 \wedge w_1 = w_2 \wedge \alpha_2 = []$
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 $\bigwedge p_1\ w_1\ \alpha_1\ p_2\ w_2\ \alpha_2\ p_3\ w_3\ \alpha_3.\ (p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_2,\ w_2,\ \alpha_2) \rightsquigarrow\!* (p_3,\ w_3,\ \alpha_3) \Longrightarrow$
$P\ (p_2,\ w_2,\ \alpha_2)\ (p_3,\ w_3,\ \alpha_3) \Longrightarrow P\ (p_1,\ w_1,\ \alpha_1)\ (p_3,\ w_3,\ \alpha_3)$
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 step*$_1$*.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 $\bigwedge p_1\ w_1\ \alpha_1\ p_2\ w_2\ \alpha_2\ p_3\ w_3\ \alpha_3.\ (p_1,\ w_1,\ \alpha_1) \rightsquigarrow* (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_2,\ w_2,\ \alpha_2) \rightsquigarrow (p_3,\ w_3,\ \alpha_3) \Longrightarrow$
$P\ (p_1,\ w_1,\ \alpha_1)\ (p_2,\ w_2,\ \alpha_2) \Longrightarrow P\ (p_1,\ w_1,\ \alpha_1)\ (p_3,\ w_3,\ \alpha_3)$
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 step*$_1$*.simps*)
qed

lemmas *converse_rtranclp_induct3_aux* =
converse_rtranclp_induct [*of step*$_1$ (*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 *step*$_1$*_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 *step*$_1$*_rule_ext* **by** *simp*

lemma *decreasing_word*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow* (p_2,\ w_2,\ \alpha_2) \Longrightarrow \exists\, w.\ w_1 = w @ w_2$
by (*induction rule*: *steps_induct*) (*use step*$_1$*_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 *step*$_1$*_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*: $(\exists\, p'\ w'\ \alpha'.\ (p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p',\ w',\ \alpha') \wedge (p',\ w',\ \alpha')$

$\rightsquigarrow(n)$ $(p_2,\ w_2,\ \alpha_2)$)
$\longleftrightarrow$ $(p_1,\ w_1,\ \alpha_1)$ $\rightsquigarrow(Suc\ n)$ $(p_2,\ w_2,\ \alpha_2)$ (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l*
then obtain p' w' α' **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* $\Longrightarrow$ *?l*
by (*induction Suc n* $(p_1,\ w_1,\ \alpha_1)$ $(p_2,\ w_2,\ \alpha_2)$ *arbitrary*: *n* p_2 w_2 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 $\bigwedge n\ p_1\ w_1\ \alpha_1\ p_2\ w_2\ \alpha_2\ p_3\ w_3\ \alpha_3$. $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_2,\ w_2,\ \alpha_2) \rightsquigarrow(n)$ $(p_3,\ w_3,\ \alpha_3) \Longrightarrow$
P n $(p_2,\ w_2,\ \alpha_2)$ $(p_3,\ w_3,\ \alpha_3) \Longrightarrow$ *P* (*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.prems*(*1*) **have** *x1* = *x2* **by** *auto*
with *0.prems*(*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.prems*(*1*) **obtain** p' w' α' **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* $p_1\ w_1\ \alpha_1$ *n* $p_2\ w_2\ \alpha_2$] **by** *auto*
have *P n* $(p',\ w',\ \alpha')$ $(p_2,\ w_2,\ \alpha_2)$
using *Suc.IH*[*unfolded x2_def*, *OF rN Suc.prems*(*2*,*3*)] **by** *simp*
then show *?case*
using *Suc.prems*(*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*: $step_1$*_steps steps_trans*)
    done
next
  show *?r* $\Longrightarrow$ *?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* $\Longrightarrow$ *?r*
   by (*induction n* $(p_1,\ w_1,\ \alpha_1)\ (p_2,\ w_2,\ \alpha_2)$ *arbitrary*: $p_2\ w_2\ \alpha_2$ *rule*: *stepn.induct*) (*use* $step_1$*_word_app* **in** *auto*)
next
  show *?r* $\Longrightarrow$ *?l*
  proof (*induction n* $(p_1,\ w_1 @ w,\ \alpha_1)\ (p_2,\ w_2 @ w,\ \alpha_2)$ *arbitrary*: $p_2\ w_2\ \alpha_2$ *rule*: *stepn.induct*)
    case ($step_n$ $n\ p_2\ w_2\ \alpha_2\ p_3\ \alpha_3\ w_3$)
    obtain w' **where** w_2*_def*: $w_2 = w' @ w_3 @ w$
      using *decreasing_word*[*OF* $step_1$*_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 $step_1$*_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 $\exists\, n'\ p'\ w'\ \alpha'.\ n = Suc\ n' \wedge (p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p',\ w',\ \alpha') \wedge (p',\ w',\ \alpha') \rightsquigarrow(n')\ (p_2,\ w_2,\ \alpha_2)$
proof −
  from *assms* **have** $n > 0$ **by** *fast*

then obtain n' **where** $n = Suc\ n'$
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 $\exists\, n'\ p'\ w'\ \alpha'.\ n = Suc\ n' \wedge (p_1,\ w_1,\ \alpha_1) \rightsquigarrow(n')\ (p',\ w',\ \alpha') \wedge (p',\ w',\ \alpha') \rightsquigarrow (p_2,\ w_2,\ \alpha_2)$
proof −
from *assms* **have** $n > 0$ **by** *fast*
then obtain n' **where** $n = Suc\ n'$
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*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow(n)\ (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_1,\ w_1,\ \alpha_1\ @\ \beta) \rightsquigarrow(n)\ (p_2,\ w_2,\ \alpha_2\ @\ \beta)$
by (*induction n* $(p_1,\ w_1,\ \alpha_1)\ (p_2,\ w_2,\ \alpha_2)$ *arbitrary*: $p_2\ w_2\ \alpha_2$ *rule*: *stepn.induct*) (*fastforce intro*: *step*$_1$*_stack_app*)+

lemma *steps_stack_app*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow*\ (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_1,\ w_1,\ \alpha_1\ @\ \beta) \rightsquigarrow* (p_2,\ w_2,\ \alpha_2\ @\ \beta)$
using *stepn_steps stepn_stack_app* **by** *metis*

lemma *step*$_1$*_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** α_1*_*γ*_def*: $\alpha_1\ @\ \gamma = Z' \#\ \alpha'$ **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 *step*$_1$*_rule_ext* **by** *auto*

from $\alpha_1\_\gamma\_def\ assms(2)$ **obtain** α'' **where** α_1_def: $\alpha_1 = Z' \# \alpha''$ **and** α'_def: $\alpha' = \alpha'' @ \gamma$
using *Cons_eq_append_conv*[*of* $Z'\ \alpha'\ \alpha_1\ \gamma$] **by** *auto*
from *rule* α'_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 α_1_def **show** *?thesis*
using $step_1\_rule$ **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*: $p_1\ \alpha_1$ *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 $step_1\_rule\_ext$ **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*: $(q_1,\ a \# w,\ \gamma_1) \rightsquigarrow (q_2,\ w,\ \gamma_2)$ **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*:
$(p_1,\ w\ @\ w',\ \alpha_1) \rightsquigarrow(n)\ (p_2,\ [],\ \alpha_2) \Longrightarrow \exists k\ q\ \gamma.\ k \leq n \wedge (p_1,\ w,\ \alpha_1) \rightsquigarrow(k)\ (q,\ [],\ \gamma) \wedge (q,\ w',\ \gamma) \rightsquigarrow(n-k)\ (p_2,\ [],\ \alpha_2)$
proof (*induction w arbitrary*: $n\ p_1\ \alpha_1$)
case (*Cons a w*)
from *Cons*(*2*) **obtain** $n'\ k\ q_1\ q_2\ \gamma_1\ \gamma_2$ **where** *n_def*: $n = Suc\ n'$ **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* $p_1\ a\ w\ @\ w'\ \alpha_1\ p_2\ \alpha_2$] **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* $(Suc\ k)\ (p_1,\ a\ \#\ w,\ \alpha_1)\ (q_2,\ w,\ \gamma_2)$
using *stepn_word_app*[*of Suc k* $p_1\ a\ \#\ w\ \alpha_1\ q_2\ w\ \gamma_2\ w'$] **by** *simp*

have $(p_1,\ a\ \#\ w,\ \alpha_1) \rightsquigarrow(Suc\ k + k'')\ (q,\ [],\ \gamma)$
using *stepn_trans*[*OF stepSuck stepk''*] **.**

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

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

ultimately show *?case* **by** *blast*
qed *fastforce*

lemma *split_stack*:
stepn $n\ (p_1,\ w_1,\ \alpha_1\ @\ \beta_1)\ (p_2,\ [],\ []) \Longrightarrow \exists p'\ m_1\ m_2\ y\ y'.\ w_1 = y\ @\ y' \wedge m_1 + m_2 = n$
$\wedge\ (p_1,\ y,\ \alpha_1) \rightsquigarrow(m_1)\ (p',\ [],\ []) \wedge (p',\ y',\ \beta_1) \rightsquigarrow(m_2)\ (p_2,\ [],\ [])$
proof (*induction n arbitrary*: $p_1\ w_1\ \alpha_1$)
case (*Suc n*)
show *?case* **proof** (*cases* α_1)
case *Nil*

from *Nil* **have** *stepn 0* $(p_1,\ [],\ \alpha_1)\ (p_1,\ [],\ [])$ **by** *simp*

moreover from *Suc.prems Nil* **have** *stepn* $(Suc\ n)\ (p_1,\ w_1,\ \beta_1)\ (p_2,\ [],\ [])$ **by** *simp*

ultimately show *?thesis* **by** *force*
next
case (*Cons Z* α)
with *Suc.prems* **obtain** $p'\ w'\ \alpha'$ **where** *r1*: $step_1\ (p_1,\ w_1,\ Z\ \#\ \alpha\ @\ \beta_1)\ (p',\ w',\ \alpha')$ **and** *rN*: *stepn* $n\ (p',\ w',\ \alpha')\ (p_2,\ [],\ [])$
using *stepn_split_first*[*of* $p_1\ w_1\ Z\ \#\ \alpha\ @\ \beta_1\ n\ p_2\ []\ []$] **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 $step_1\_rule$ **by** *blast*
show *?thesis* **proof** (*rule disjE*[*OF rule*])
assume *?l*
then obtain β **where** *w1_def*: $w_1 = w'$ **and** α'_def: $\alpha' = \beta\ @\ \alpha\ @\ \beta_1$ **and** *e*: $(p',\beta) \in \delta\varepsilon\ M\ p_1\ Z$ **by** *blast*
from *rN* α'_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*: $step_1\ (p_1,\ y,\ Z\#\alpha)\ (p',\ y,\ \beta@\alpha)$
using $step_1\_rule$ **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* β **where** *w1_def*: $w_1 = a\ \#\ w'$ **and** α'_def: $\alpha' = \beta\ @\ \alpha\ @\ \beta_1$ **and** *tr*: $(p',\beta) \in \delta\ M\ p_1\ a\ Z$ **by** *blast*
from *rN* α'_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*: $step_1\ (p_1,\ a\#y,\ Z\#\alpha)\ (p',\ y,\ \beta@\alpha)$ **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. ∃ q. t = Old_st q} ∪ {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. ∃ s. t = Old_sym s} ∪ {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 ⇒ 'a ⇒ 's sym_extended ⇒ ('q st_extended
× 's sym_extended list) set where
  final_of_stack_delta (Old_st q) a (Old_sym Z) = (λ(p, α). (Old_st p, map
Old_sym α)) ‘ (δ 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* ⇒ *'s sym_extended* ⇒ (*'q st_extended* × *'s sym_extended list*) *set* **where**
  final_of_stack_delta_eps (*Old_st q*) (*Old_sym Z*) = (λ(*p*, *α*). (*Old_st p*, *map Old_sym α*)) ‘ (*δε 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 ≡ ⦇ *init_state* = *New_init*, *init_symbol* = *New_sym*, *final_states* = {*New_final*},
    delta = *final_of_stack_delta*, *delta_eps* = *final_of_stack_delta_eps* ⦈

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*, *β*) ∈ *δ M q a Z* ⟷
      (*Old_st p*, *map Old_sym β*) ∈ *δ 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*, *β*) ∈ *δε M q Z* ⟷ (*Old_st p*, *map Old_sym β*) ∈ *δε 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*$_1$, *α*$_1$) ⇝ (*p*$_2$, *w*$_2$, *α*$_2$) ⟷
     pda.step$_1$ *final_of_stack_pda* (*Old_st p*$_1$, *w*$_1$, *map Old_sym α*$_1$) (*Old_st p*$_2$, *w*$_2$, *map Old_sym α*$_2$) (**is** *?l* ⟷ *?r*)
proof
  assume *?l*
  then obtain *Z α* **where** *α*$_1$_*def*: *α*$_1$ = *Z* # *α* **and** *rule*: (∃ *β*. *w*$_2$ = *w*$_1$ ∧ *α*$_2$ =

$\beta@\alpha \wedge (p_2, \beta) \in \delta\varepsilon\ M\ p_1\ Z$)
$\vee$ ($\exists\, a\ \beta.\ w_1 = a \#\ w_2 \wedge \alpha_2 = \beta@\alpha \wedge (p_2,\beta) \in \delta\ M\ p_1\ a\ Z$)
using *step*$_1$*_rule_ext* **by** *auto*
from *rule* **have** ($\exists\, \beta.\ w_2 = w_1 \wedge$ *map Old_sym* α_2 = *map Old_sym* β @ *map Old_sym* $\alpha \wedge$ (*Old_st* p_2, *map Old_sym* β) $\in \delta\varepsilon$ *final_of_stack_pda* (*Old_st* p_1) (*Old_sym* Z))
$\vee$ ($\exists\, a\ \beta.\ w_1 = a \#\ w_2 \wedge$ *map Old_sym* α_2 = *map Old_sym* β @ *map Old_sym* $\alpha \wedge$
(*Old_st* p_2, *map Old_sym* β) $\in \delta$ *final_of_stack_pda* (*Old_st* p_1) a (*Old_sym* Z))
using *final_of_stack_pda_trans final_of_stack_pda_eps* **by** *auto*
hence ($\exists\, \beta.\ w_2 = w_1 \wedge$ *map Old_sym* α_2 = β @ *map Old_sym* $\alpha \wedge$ (*Old_st* p_2, β) $\in \delta\varepsilon$ *final_of_stack_pda* (*Old_st* p_1) (*Old_sym* Z))
$\vee$ ($\exists\, a\ \beta.\ w_1 = a \#\ w_2 \wedge$ *map Old_sym* α_2 = β @ *map Old_sym* $\alpha \wedge$ (*Old_st* p_2, β) $\in \delta$ *final_of_stack_pda* (*Old_st* p_1) a (*Old_sym* Z))
by *blast*
with α_1*_def* **show** *?r*
using *pda.step*$_1$*_rule*[*OF pda_final_of_stack*] **by** *simp*
next
assume *?r*
then obtain $Z\ \alpha$ **where** *map_*α_1*_def*: *map Old_sym* α_1 = *Old_sym* Z # *map Old_sym* α **and**
rule: ($\exists\, \beta.\ w_2 = w_1 \wedge$ *map Old_sym* α_2 = β @ *map Old_sym* $\alpha \wedge$ (*Old_st* p_2, β) $\in \delta\varepsilon$ *final_of_stack_pda* (*Old_st* p_1) (*Old_sym* Z))
$\vee$ ($\exists\, a\ \beta.\ w_1 = a \#\ w_2 \wedge$ *map Old_sym* α_2 = β@ *map Old_sym* $\alpha \wedge$ (*Old_st* p_2,β) $\in \delta$ *final_of_stack_pda* (*Old_st* p_1) a (*Old_sym* Z))
using *pda.step*$_1$*_rule_ext*[*OF pda_final_of_stack*] **by** *auto*
from *map_*α_1*_def* **have** α_1*_def*: $\alpha_1 = Z \#\ \alpha$
by (*metis list.inj_map_strong list.simps*(9) *sym_extended.inject*)
from *rule* **have** ($\exists\, \beta.\ w_2 = w_1 \wedge$ *map Old_sym* α_2 = *map Old_sym* β@ *map Old_sym* $\alpha \wedge$ (*Old_st* p_2, *map Old_sym* β) $\in \delta\varepsilon$ *final_of_stack_pda* (*Old_st* p_1) (*Old_sym* Z))
$\vee$ ($\exists\, a\ \beta.\ w_1 = a \#\ w_2 \wedge$ *map Old_sym* α_2 = *map Old_sym* β@ *map Old_sym* $\alpha \wedge$ (*Old_st* p_2, *map Old_sym* β) $\in \delta$ *final_of_stack_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 *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 *step*$_1$*_rule* **by** *simp*
qed

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

lemma *final_of_stack_pda_step*$_1$*_drop*:
assumes *pda.step*$_1$ *final_of_stack_pda* (*Old_st* p_1, w_1, *α_with_new* α_1)

```
                                        (Old_st p₂, w₂, α_with_new α₂)
    shows (p₁, w₁, α₁) ⇝ (p₂, w₂, α₂)
proof −
  from assms obtain Z α where α₁_with_new_def: α_with_new α₁ = Z # α
and
   rule: (∃β. w₂ = w₁ ∧ α_with_new α₂ = β@α ∧ (Old_st p₂, β) ∈ final_of_stack_delta_eps
(Old_st p₁) Z)
         ∨ (∃a β. w₁ = a # w₂ ∧ α_with_new α₂ = β@α ∧ (Old_st p₂,β) ∈
final_of_stack_delta (Old_st p₁) a Z)
   using pda.step₁_rule_ext[OF pda_final_of_stack] by (auto simp: final_of_stack_pda_def)
  from rule have Z ≠ New_sym
    by (induction Old_st p₁ Z rule: final_of_stack_delta_eps.induct) auto
  with α₁_with_new_def have map Old_sym α₁ ≠ [] by auto
  with assms have pda.step₁ final_of_stack_pda (Old_st p₁, w₁, map Old_sym
α₁)
                                        (Old_st p₂, w₂, map Old_sym α₂)
    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 p₁, w₁, α₁) (p₂, w₂, α₂)
    shows (∃p₂'. p₂ = Old_st p₂') ∨ p₂ = New_final
proof −
  from assms obtain Z α where
         (∃β. w₂ = w₁ ∧ α₂ = β@α ∧ (p₂, β) ∈ final_of_stack_delta_eps (Old_st
p₁) Z)
              ∨ (∃a β. w₁ = a # w₂ ∧ α₂ = β@α ∧ (p₂,β) ∈ final_of_stack_delta
(Old_st p₁) 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:
  ¬pda.step₁ final_of_stack_pda (New_final, w₁, α₁) (p, w₂, α₂)
  apply (cases α₁)
   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 p₁, w₁, α₁) (p₂, w₂, α₂)
  shows ∃q'. p₂ = Old_st q' ∨ p₂ = New_final
by (induction (Old_st p₁, w₁, α₁) (p₂, w₂, α₂) arbitrary: p₂ w₂ α₂ 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.step*$_1$ *final_of_stack_pda* (p_1, w_1, α_1) (*Old_st* p_2, w_2, α_2)
shows ($\exists\, q'.\ p_1 = Old\_st\ q'$) $\vee$ $p_1 = New\_init$
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, α*_with_new* α_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, α*_with_new* α_1) (*Old_st* p_2, w_2, γ) *arbitrary*: p_2 w_2 γ
rule: *pda.steps_induct2_bw*[*OF pda_final_of_stack*])
case (*3* p_2 w_2 α_2 w_3 α_3 p_3)
obtain p_2' **where** p_2*_def*: $p_2 = Old\_st\ p_2'$
using *final_of_stack_pda_from_oldn*[*OF 3*(*1*)] *final_of_stack_pda_to_old*[*OF 3*(*2*)] **by** *blast*
with *3*(*1*,*3*) **have** α_2*_def*: $\exists\, \alpha.\ \alpha_2 = \alpha\_with\_new\ \alpha$ **by** *simp*
from *3*(*2*)[*unfolded* p_2*_def*] **obtain** Z α **where** α_2*_split*: $\alpha_2 = Z\ \#\ \alpha$ **and** *rule*:
($\exists\, \beta.\ w_3 = w_2 \wedge \alpha_3 = \beta\ @\ \alpha \wedge (Old\_st\ p_3, \beta) \in final\_of\_stack\_delta\_eps\ (Old\_st\ p_2')\ Z$)
$\vee$ ($\exists\, a\ \beta.\ w_2 = a\ \#\ w_3 \wedge \alpha_3 = \beta\ @\ \alpha \wedge (Old\_st\ p_3, \beta) \in final\_of\_stack\_delta\ (Old\_st\ p_2')\ a\ Z$)
using *pda.step*$_1$*_rule_ext*[*OF pda_final_of_stack*] **by** (*auto simp*: *final_of_stack_pda_def*)
from *rule* **have** $\exists\, Z'.\ Z = Old\_sym\ Z'$
by (*induction Old_st* p_2' *Z rule*: *final_of_stack_delta_eps.induct*) *auto*
with α_2*_def* α_2*_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, α*_with_new* α_1) (*Old_st* p_2, w_2, α*_with_new* α_2) (**is** *?l* $\longleftrightarrow$ *?r*)
proof
show *?l* $\Longrightarrow$ *?r*
proof (*induction n* (p_1, w_1, α_1) (p_2, w_2, α_2) *arbitrary*: p_2 w_2 α_2 *rule*: *stepn.induct*)
case (*step*$_n$ *n* p_2 w_2 α_2 p_3 w_3 α_3)
from *step*$_n$(*3*) **have** *pda.step*$_1$ *final_of_stack_pda* (*Old_st* p_2, w_2, *map Old_sym* α_2) (*Old_st* p_3, w_3, *map Old_sym* α_3)
using *final_of_stack_pda_step* **by** *simp*
hence *pda.step*$_1$ *final_of_stack_pda* (*Old_st* p_2, w_2, α*_with_new* α_2) (*Old_st* p_3, w_3, α*_with_new* α_3)
using *pda.step*$_1$*_stack_app*[*OF pda_final_of_stack*] **by** *simp*
with *step*$_n$(*2*) **show** *?case*

```
    by (simp add: pda.step_n[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) ⇝* (p₂, w₂, α₂) ⟷
          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 ∃ q. p₁ = Old_st q ∧ w₁ = w₂ ∧ α₁ = New_sym # α₂
proof −
  from assms obtain Z α where α₁_def: α₁ = Z # α and rule:
       (∃ β. w₂ = w₁ ∧ α₂ = β @ α ∧ (New_final, β) ∈ final_of_stack_delta_eps
```

p_1 Z)
∨ (∃ a β. $w_1 = a \# w_2 \wedge \alpha_2 = \beta$ @ $\alpha \wedge$ (*New_final*, β) ∈ *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** ∃ 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 ∃ 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*]) ∧
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*]) ∧
pda.step$_1$ *final_of_stack_pda* (*Old_st* q, w_2, [*New_sym*])
(*New_final*, w_2, γ) ∧ γ = []
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* # γ) (*New_final*, w_2, γ)
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* # γ)
using *pda.stepn_split_last*[*OF pda_final_of_stack*] *final_of_stack_pda_last_step* **by** *blast*
from *stepn'* **have** ∃ α. *New_sym* # γ = α*_with_new* α
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 γ = []
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 $\exists\, n'.\ n = Suc\ (Suc\ (Suc\ n'))$
proof −
from *assms* **obtain** n' **where** *n_def*: $n = Suc\ n'$ **and** *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 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*: $n' = Suc\ n''$
using *pda.stepn_not_refl_split_last*[*OF pda_final_of_stack*] **by** *blast*
with *n_def assms* **have** $\exists\, q.$ *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*])
using *final_of_stack_pda_split_path* **by** *blast*
then obtain n''' **where** $n'' = Suc\ n'''$
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*:
$(\exists\, q.\ (init\_state\ M,\ w,\ [init\_symbol\ M]) \rightsquigarrow* (q,\ [],\ [])) \longleftrightarrow (\exists\, q\ \gamma.\ q \in final\_states\ final\_of\_stack\_pda \wedge$
pda.steps final_of_stack_pda (*init_state final_of_stack_pda*, *w*, [*init_symbol final_of_stack_pda*]) (*q*, [], γ)) (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l*
then obtain *q* **where** (*init_state M*, *w*, [*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.step*$_1$*_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.step*$_1$*_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.step*$_1$*_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*: *final_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*)]) ⇝(*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*] **using** *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*. ∃ *q*. *t* = *Old_st q*} ∪ {*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*. ∃ *s*. *t* = *Old_sym s*} ∪ {*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* ⇒ *'a* ⇒ *'s sym_extended* ⇒ (*'q st_extended* × *'s sym_extended list*) *set* **where**
  stack_of_final_delta (*Old_st q*) *a* (*Old_sym Z*) = (λ(*p*, *α*). (*Old_st p*, *map Old_sym α*)) ‘ (*δ M q a Z*)
| *stack_of_final_delta* _ _ _ = {}

fun *stack_of_final_delta_eps* :: *'q st_extended* ⇒ *'s sym_extended* ⇒ (*'q st_extended* × *'s sym_extended list*) *set* **where**
  stack_of_final_delta_eps (*Old_st q*) (*Old_sym Z*) = (*if q* ∈ *final_states M then* {(*New_final*, [*Old_sym Z*])} *else* {}) ∪
                                                (λ(*p*, *α*). (*Old_st p*, *map Old_sym α*)) ‘ (*δε M q Z*)
| *stack_of_final_delta_eps* (*Old_st q*) *New_sym* = (*if q* ∈ *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 ≡ (| *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*:
(*p*$_1$, *w*$_1$, *α*$_1$) ⇝ (*p*$_2$, *w*$_2$, *α*$_2$) ⟷
pda.step$_1$ *stack_of_final_pda* (*Old_st p*$_1$, *w*$_1$, *map Old_sym α*$_1$) (*Old_st p*$_2$, *w*$_2$, *map Old_sym α*$_2$) (**is** *?l* ⟷ *?r*)
proof
assume *?l*
then obtain *Z α* **where** *α*$_1$*_def*: *α*$_1$ = *Z*#*α* **and** *rule*:
(∃ *β*. *w*$_2$ = *w*$_1$ ∧ *α*$_2$ = *β*@*α* ∧ (*p*$_2$, *β*) ∈ *δε M p*$_1$ *Z*)
∨ (∃ *a β*. *w*$_1$ = *a* # *w*$_2$ ∧ *α*$_2$ = *β*@*α* ∧ (*p*$_2$,*β*) ∈ *δ M p*$_1$ *a Z*)
using *step*$_1$*_rule_ext* **by** *auto*
from *rule* **have** (∃ *β*. *w*$_2$ = *w*$_1$ ∧ *map Old_sym α*$_2$ = *map Old_sym β* @ *map Old_sym α* ∧
(*Old_st p*$_2$, *map Old_sym β*) ∈ *δε stack_of_final_pda* (*Old_st p*$_1$) (*Old_sym Z*))
∨ (∃ *a β*. *w*$_1$ = *a* # *w*$_2$ ∧ *map Old_sym α*$_2$ = *map Old_sym β* @ *map Old_sym α* ∧
(*Old_st p*$_2$, *map Old_sym β*) ∈ *δ stack_of_final_pda* (*Old_st p*$_1$) *a* (*Old_sym Z*))
using *stack_of_final_pda_trans stack_of_final_pda_eps* **by** *fastforce*
hence (∃ *β*. *w*$_2$ = *w*$_1$ ∧ *map Old_sym α*$_2$ = *β* @ *map Old_sym α* ∧ (*Old_st p*$_2$, *β*) ∈ *δε stack_of_final_pda* (*Old_st p*$_1$) (*Old_sym Z*))
∨ (∃ *a β*. *w*$_1$ = *a* # *w*$_2$ ∧ *map Old_sym α*$_2$ = *β* @ *map Old_sym α* ∧ (*Old_st p*$_2$, *β*) ∈ *δ stack_of_final_pda* (*Old_st p*$_1$) *a* (*Old_sym Z*)) **by** *blast*
with *α*$_1$*_def* **show** *?r*

using *pda.step*$_1$*_rule*[*OF pda_final_to_stack*] **by** *simp*
next
assume *?r*
then obtain Z α **where** *map_*α_1*_def*: *map Old_sym* α_1 = *Old_sym* Z # *map Old_sym* α **and** *rule*:
$(\exists \beta.\ w_2 = w_1 \wedge$ *map Old_sym* $\alpha_2 = \beta$@ *map Old_sym* $\alpha \wedge$
(*Old_st* p_2, β) $\in \delta\varepsilon$ *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,β) $\in \delta$ *stack_of_final_pda* (*Old_st* p_1) a (*Old_sym* Z))
using *pda.step*$_1$*_rule_ext*[*OF pda_final_to_stack*] **by** *auto*
from *map_*α_1*_def* **have** α_1*_def*: $\alpha_1 = Z$ # α
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* α_2 = *map Old_sym* β@ *map Old_sym* $\alpha \wedge$
(*Old_st* p_2, *map Old_sym* β) $\in \delta\varepsilon$ *stack_of_final_pda* (*Old_st* p_1) (*Old_sym* Z))
$\vee$ $(\exists a\ \beta.\ w_1 = a$ # $w_2 \wedge$ *map Old_sym* α_2 = *map Old_sym* β@ *map Old_sym* $\alpha \wedge$
(*Old_st* p_2, *map Old_sym* β) $\in \delta$ *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 α_1*_def* **show** *?l*
using *step*$_1$*_rule* **by** *simp*
qed

abbreviation *α_with_new* :: *'s list* ⇒ *'s sym_extended list* **where**
α_with_new $\alpha \equiv$ *map Old_sym* α @ [*New_sym*]

lemma *stack_of_final_pda_step*$_1$*_drop*:
assumes *pda.step*$_1$ *stack_of_final_pda* (*Old_st* p_1, w_1, *α_with_new* α_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** α_1*_with_new_def*: *α_with_new* $\alpha_1 = Z$ # α **and** *rule*:
$(\exists \beta.\ w_2 = w_1 \wedge \alpha$*_with_new* $\alpha_2 = \beta$@$\alpha \wedge$ (*Old_st* p_2, β) $\in$ *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,β) $\in$ *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*)
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 α_1*_with_new_def* **have** *map Old_sym* $\alpha_1 \neq$ [] **by** *auto*
with *assms* **have** *pda.step*$_1$ *stack_of_final_pda* (*Old_st* p_1, w_1, *map Old_sym*

α_1)
(*Old_st* p_2, w_2, *map Old_sym* α_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, α_1) (p_2, w_2, α_2)
shows ($\exists\, p_2'.\ p_2 =$ *Old_st* p_2') $\vee$ $p_2 =$ *New_final*
proof −
from *assms* **obtain** Z α **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, α_1) (p_2, w_2, α_2)
shows $\exists\, Z'.\ p_2 =$ *New_final* $\wedge$ $w_2 = w_1 \wedge \alpha_1 = Z'\#\alpha_2$
proof −
from *assms* **obtain** Z α **where** α_1*_def*: $\alpha_1 = Z\#\alpha$ **and** *rule*:
($\exists\, \beta.\ w_2 = w_1 \wedge \alpha_2 = \beta@\alpha \wedge (p_2,\beta) \in$ *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, α_1) (p_2, w_2, α_2)
shows $\exists\, q'.\ p_2 =$ *Old_st* $q' \vee p_2 =$ *New_final*
proof (*induction* (*Old_st* p_1, w_1, α_1) (p_2, w_2, α_2) *arbitrary*: p_2 w_2 α_2 *rule*: *pda.steps_induct2_bw*[*OF pda_final_to_stack*])
case (*3* p_2 w_2 α_2 p_3 w_3 α_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, α_1) (*Old_st* p_2, w_2, α_2)

shows $(\exists\, q'.\ p_1 = Old\_st\ q') \vee p_1 = New\_init$
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, α*_with_new* α_1) (*Old_st* p_2, w_2, γ)
shows $\exists\, \alpha.\ \gamma = \alpha\_with\_new\ \alpha$
proof (*induction* (*Old_st* p_1, w_1, α*_with_new* α_1) (*Old_st* p_2, w_2, γ) *arbitrary*: p_2 w_2 γ *rule*: *pda.steps_induct2_bw*[*OF pda_final_to_stack*])
case (*3* p_2 w_2 α_2 w_3 α_3 p_3)
obtain p_2' **where** *p*$_2$*_def*: $p_2 = Old\_st\ p_2'$
using *stack_of_final_pda_from_oldn*[*OF 3*(*1*)] *stack_of_final_pda_to_old*[*OF 3*(*2*)] **by** *blast*
with *3*(*3*) **have** α_2*_def*: $\exists\, \alpha.\ \alpha_2 = \alpha\_with\_new\ \alpha$ **by** *simp*
from *p*$_2$*_def 3*(*2*) **obtain** Z α **where** α_2*_split*: $\alpha_2 = Z\ \#\ \alpha$ **and** *rule*:
$(\exists\, \beta.\ w_3 = w_2 \wedge \alpha_3 = \beta\ @\ \alpha \wedge (Old\_st\ p_3, \beta) \in stack\_of\_final\_delta\_eps\ (Old\_st\ p_2')\ Z)$
$\vee\ (\exists\, a\ \beta.\ w_2 = a\ \#\ w_3 \wedge \alpha_3 = \beta\ @\ \alpha \wedge (Old\_st\ p_3, \beta) \in stack\_of\_final\_delta\ (Old\_st\ p_2')\ a\ Z)$
using *pda.step*$_1$*_rule_ext*[*OF pda_final_to_stack*] **by** (*auto simp*: *stack_of_final_pda_def*)
hence $\exists\, Z'.\ Z = Old\_sym\ Z'$
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*(*3*))
with α_2*_def* α_2*_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*: *stack_of_final_delta_eps.induct*, *auto*)
(*metis empty_iff fst_conv singleton_iff st_extended.distinct*(*3*), *metis map_append*,
metis map_append, *metis empty_iff fst_conv singleton_iff st_extended.distinct*(*3*))
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, α*_with_new* α_1) (*Old_st* p_2, w_2, α*_with_new* α_2) (**is** *?l* $\longleftrightarrow$ *?r*)
proof
show *?l* $\Longrightarrow$ *?r*
proof (*induction n* (p_1, w_1, α_1) (p_2, w_2, α_2) *arbitrary*: p_2 w_2 α_2 *rule*: *stepn.induct*)
case (*step*$_n$ *n* p_2 w_2 α_2 p_3 w_3 α_3)
from *step*$_n$*(3)* **have** *pda.step*$_1$ *stack_of_final_pda* (*Old_st* p_2, w_2, *map Old_sym* α_2) (*Old_st* p_3, w_3, *map Old_sym* α_3)
using *stack_of_final_pda_step* **by** *simp*
hence *pda.step*$_1$ *stack_of_final_pda* (*Old_st* p_2, w_2, α*_with_new* α_2) (*Old_st* p_3, w_3, α*_with_new* α_3)
using *pda.step*$_1$*_stack_app*[*OF pda_final_to_stack*] **by** *simp*
with *step*$_n$*(2)* **show** *?case*
by (*simp add*: *pda.step*$_n$[*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, α_*with_new* α_1) (*Old_st* p_2, w_2, α_*with_new* α_2)
 arbitrary: p_2 w_2 α_2 *rule*: *pda.stepn.induct*[*OF pda_final_to_stack*])
 case (*3 n* p_2 w_2 α_2 w_3 p_3 α_3)
 from *3*(*1*) **have** *steps_3_1*: *pda.steps stack_of_final_pda* (*Old_st* p_1, w_1, α_*with_new* α_1) (p_2, w_2, α_2)
 using *pda.stepn_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 3*(*3*)] **by** *blast*
 with *steps_3_1* **obtain** γ **where** α_2_*def*: α_2 = α_*with_new* γ
 using *stack_of_final_pda_bottom_elem* **by** *blast*

 with *p*$_2$_*def* *3*(*1*,*2*) **have** *pda.stepn M n* (p_1, w_1, α_1) ($p_2{}'$, w_2, γ) **by** *simp*

 moreover from *p*$_2$_*def* α_2_*def* *3*(*3*) **have** *pda.step*$_1$ *M* ($p_2{}'$, w_2, γ) (p_3, w_3, α_3)
 using *stack_of_final_pda_step*$_1$_*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, α_1) $\rightsquigarrow*$ (p_2, w_2, α_2) $\longleftrightarrow$
 pda.steps stack_of_final_pda (*Old_st* p_1, w_1, α_*with_new* α_1) (*Old_st* p_2, w_2, α_*with_new* α_2)
using *stack_of_final_pda_stepn pda.stepn_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* γ)
 have (*New_final*, []) $\in$ *stack_of_final_delta_eps New_final Z* **by** *simp*
 hence *pda.step*$_1$ *stack_of_final_pda* (*New_final*, *w*, *Z* # γ) (*New_final*, *w*, γ)
 using *pda.step*$_1$_*rule*[*OF pda_final_to_stack*] **by** (*simp add*: *stack_of_final_pda_def*)
 with *Cons.IH* **show** *?case*
 using *pda.step*$_1$_*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.step*$_1$ *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 $\wedge$ α = [*Old_sym* (*init_symbol M*), *New_sym*]
using *assms pda.step*$_1$_*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 $\exists\, q\ k\ \gamma.\ k \leq n \wedge q \in$ *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 (*3 n* p_2 w_2 α_2 w_3 α_3)
then show *?case* **proof** (*cases n*)
case *0*
with *3*(*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 3*(*3*) **obtain** Z α **where** α_2*_def*: $\alpha_2 = Z \# \alpha$ **and** *rule*:
($\exists\,\beta.\ w_3 = w_2 \wedge \alpha_3 = \beta@\alpha \wedge$ (*New_final*,β) $\in$ *stack_of_final_delta_eps* (*Old_st* p_1) Z)
$\vee$ ($\exists\, a\ \beta.\ w_2 = a \# w_3 \wedge \alpha_3 = \beta@\alpha \wedge$ (*New_final*,β) $\in$ *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*)
from α_2*_def rule* **have** p_1*_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 *3*(*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, α_3) (*New_final*, w_3, α_3)
using $pda.refl_n$[*OF pda_final_to_stack*] **by** *simp*

ultimately show *?thesis*
using p_1*_final* **by** *blast*
next
case (*Suc n'*)
then show *?thesis* **proof** (*cases* p_2 = *New_final*)
case *True*
with *Suc 3*(*1*,*2*) **obtain** *q k* γ **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, α_1) (*Old_st q*, w_2, γ)
and
step1: $pda.step_1$ *stack_of_final_pda* (*Old_st q*, w_2, γ) (*New_final*, w_2, γ)
and
stepn': *pda.stepn stack_of_final_pda* $(n' - k)$ (*New_final*, w_2, γ) (*New_final*, w_2, α_2) **by** *blast*
from *True 3*(*3*) **obtain** Z' α' **where** $\alpha_2 = Z' \# \alpha'$ **and** *rule*:
$(\exists \beta.\ w_3 = w_2 \wedge \alpha_3 = \beta @ \alpha' \wedge$ (*New_final*,β) $\in$ *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*,β) $\in$ *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*)
from *rule* **have** *w23*: $w_3 = w_2$
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 w23* **have** *pda.stepn stack_of_final_pda k* (*Old_st* p_1, w_1, α_1) (*Old_st q*, w_3, γ) **by** *simp*

moreover from *step1 w23* **have** $pda.step_1$ *stack_of_final_pda* (*Old_st q*, w_3, γ) (*New_final*, w_3, γ) **by** *simp*

moreover from *stepn' 3*(*3*) *True w23 Suc k_less* **have** *pda.stepn stack_of_final_pda* $(n - k)$ (*New_final*, w_3, γ) (*New_final*, w_3, α_3)
using $pda.step_n$[*OF pda_final_to_stack*] **by** (*simp add*: *Suc_diff_le*)

ultimately show *?thesis*
using *q_final* **by** *blast*
next
case *False*
with *3*(*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 3*(*3*) **obtain** Z' α' **where** $\alpha_2 = Z' \# \alpha'$ **and**

```
        (∃β. w3 = w2 ∧ α3 = β@α' ∧ (New_final,β) ∈ stack_of_final_delta_eps
(Old_st p2') Z')
                    ∨ (∃a β. w2 = a # w3 ∧ α3 = β@α' ∧ (New_final,β) ∈
stack_of_final_delta (Old_st p2') a Z')
    using pda.step1_rule_ext[OF pda_final_to_stack] by (auto simp: stack_of_final_pda_def)
    hence p2_final: p2' ∈ final_states M and w23: w3 = w2 and a23: α3 = α2
      by (induction Old_st p2' 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 3(1) p2_def w23 a23 have pda.stepn stack_of_final_pda n (Old_st
p1, w1, α1) (Old_st p2', w3, α3) by simp

     moreover from 3(3) p2_def w23 a23 have pda.step1 stack_of_final_pda
(Old_st p2', w3, α3) (New_final, w3, α3) by simp

   moreover have pda.stepn stack_of_final_pda 0 (New_final, w3, α3) (New_final,
w3, α3)
      using pda.refl_n[OF pda_final_to_stack] by simp

    ultimately show ?thesis
      using p2_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, w1, [New_sym])
(New_final, w2, γ)
   shows ∃q k α. k ≤ n ∧ q ∈ final_states M ∧ pda.step1 stack_of_final_pda
(New_init, w1, [New_sym])
                                    (Old_st (init_state M), w1, [Old_sym
(init_symbol M), New_sym]) ∧
         pda.stepn stack_of_final_pda k (Old_st (init_state M), w1, [Old_sym
(init_symbol M), New_sym])
                                    (Old_st q, w2, α) ∧
        pda.step1 stack_of_final_pda (Old_st q, w2, α) (New_final, w2, α) ∧
        pda.stepn stack_of_final_pda (n−k) (New_final, w2, α) (New_final, w2,
γ)
proof −
 from assms have fstep: pda.step1 stack_of_final_pda (New_init, w1, [New_sym])

                                    (Old_st (init_state M), w1, [Old_sym (init_symbol
M), New_sym])
            and stepn: pda.stepn stack_of_final_pda (Suc n)
                                (Old_st (init_state M), w1, [Old_sym (init_symbol M),
New_sym])
                               (New_final, w2, γ)
  using pda.stepn_split_first[OF pda_final_to_stack] stack_of_final_pda_first_step
```

by *blast+*
from *stepn* **have** $\exists\, q\ k\ \alpha.\ k \leq n \wedge q \in \mathit{final\_states}\ M \wedge$
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, α) $\wedge$
pda.step$_1$ *stack_of_final_pda* (*Old_st q*, w_2, α) (*New_final*, w_2, α) $\wedge$
pda.stepn stack_of_final_pda ($n-k$) (*New_final*, w_2, α) (*New_final*, w_2, γ)
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 *pda.stepn stack_of_final_pda n* (*New_init*, w_1, [*New_sym*]) (*New_final*, w_2, γ)
shows $\exists\, n'.\ n = \mathit{Suc}\ (\mathit{Suc}\ n')$
proof −
from *assms* **obtain** n' **where** *n_def*: $n = \mathit{Suc}\ n'$ **and**
stepn': *pda.stepn stack_of_final_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_to_stack*] *stack_of_final_pda_first_step* **by** *blast*
from *stepn'* **obtain** n'' **where** $n' = \mathit{Suc}\ n''$
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*:
($\exists\, q\ \gamma.\ q \in \mathit{final\_states}\ M \wedge (\mathit{init\_state}\ M,\ w,\ [\mathit{init\_symbol}\ M]) \rightsquigarrow\!*\ (q,\ [],\ \gamma)$)
$\longleftrightarrow$
($\exists\, q.$ *pda.steps stack_of_final_pda* (*init_state stack_of_final_pda*, w, [*init_symbol stack_of_final_pda*]) (q, [], [])) (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l*
then obtain $q\ \gamma$ **where** *q_final*: $q \in \mathit{final\_states}\ M$ **and** *steps*: (*init_state M*, w, [*init_symbol M*]) $\rightsquigarrow\!*$ (q, [], γ) **by** *blast*
obtain $Z\ \alpha$ **where** *map_γ_def*: $\alpha\_\mathit{with\_new}\ \gamma = Z\#\alpha$
by (*auto intro*: *list.exhaust_sel*)
from *q_final* **have** (*New_final*, [Z]) $\in$ *stack_of_final_delta_eps* (*Old_st q*) Z
by (*induction Old_st q Z rule*: *stack_of_final_delta_eps.induct*) *auto*

with *map_γ_def* **have** *pda.step*$_1$ *stack_of_final_pda* (*Old_st q*, [], $\alpha\_\mathit{with\_new}$ γ) (*New_final*, [], $Z\#\alpha$)
using *pda.step*$_1$*_rule*[*OF pda_final_to_stack*] **by** (*simp add*: *stack_of_final_pda_def*)

moreover from *steps* **have** *pda.steps stack_of_final_pda* (*Old_st* (*init_state M*), w, [*Old_sym* (*init_symbol M*), *New_sym*])
(*Old_st q*, [], $\alpha\_\mathit{with\_new}\ \gamma$)

using *stack_of_final_pda_steps* **by** *simp*

moreover have *pda.step*$_1$ *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.step*$_1$*_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.step*$_1$*_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* $\in$ *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*] **using** *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* = λ(*A*,α). (*Q_loop*, α)
have ∗: ⋀*A*. {(*Q_loop*, α) |α. (*A*, α) ∈ *Prods G*} ⊆ (*?h* ‘ *Prods G*) **by** *auto*
have ∗∗: *finite* (*?h* ‘ *Prods G*)
by (*simp add*: *finite_G*)
have ⋀*A*. *finite* {(*Q_loop*, α) |α. (*A*, α) ∈ *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.step$_1$ *cfg_to_pda_pda* (*Q_loop*, *a*#*w*, *Tm a*#γ) (*Q_loop*, *w*, γ)
using *pda.step*$_1$*_rule*[*OF pda_cfg_to_pda*] **by** (*simp add*: *cfg_to_pda_pda_def*)

lemma *cfg_to_pda_cons_nt*:
assumes (*A*, α) ∈ *Prods G*
shows *pda.step*$_1$ *cfg_to_pda_pda* (*Q_loop*, *w*, *Nt A*#γ) (*Q_loop*, *w*, α@γ)
using *assms pda.step*$_1$*_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* @ γ) (*Q_loop*, *w*′, γ)
proof (*induction w*)
case (*Cons a w*)
have *pda.step*$_1$ *cfg_to_pda_pda* (*Q_loop*, (*a* # *w*) @ *w*′, *map Tm* (*a* # *w*) @ γ)
(*Q_loop*, *w* @ *w*′, *map Tm w* @ γ)
using *cfg_to_pda_cons_tm* **by** *simp*
with *Cons.IH* **show** *?case*
using *pda.step*$_1$*_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.step*$_1$ *cfg_to_pda_pda* (*Q_loop*, *w*, *Nt A*#γ) (*Q_loop*, *w*′, β)
shows ∃α. (*A*, α) ∈ *Prods G* ∧ β = α @ γ ∧ *w*′ = *w*
proof −
from *assms* **have** (∃β$_0$. *w*′ = *w* ∧ β = β$_0$@γ ∧ (*Q_loop*, β$_0$) ∈ *pda_eps_of_cfg*
Q_loop (*Nt A*))

$\vee$ ($\exists$ *a* β_0. *w* = *a* # *w'* $\wedge$ β = β_0@γ $\wedge$ (*Q_loop*,β_0) $\in$ *pda_of_cfg Q_loop a* (*Nt A*))
using *pda.step*$_1$*_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.step*$_1$ *cfg_to_pda_pda* (*Q_loop*, *w*, *Tm a* # β) (*Q_loop*, *w'*, β')
shows *w* = *a* # *w'* $\wedge$ β = β'
proof −
from *assms* **have** ($\exists$ β_0. *w'* = *w* $\wedge$ β' = β_0@β $\wedge$ (*Q_loop*, β_0) $\in$ *pda_eps_of_cfg Q_loop* (*Tm a*))
$\vee$ ($\exists$ a_0 β_0. *w* = a_0#*w'* $\wedge$ β' = β_0@β $\wedge$ (*Q_loop*, β_0) $\in$ *pda_of_cfg Q_loop* a_0 (*Tm a*))
using *pda.step*$_1$*_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* # α) (*Q_loop*, [], [])
shows $\exists$ *w'*. *w* = *a*#*w'* $\wedge$ *pda.steps cfg_to_pda_pda* (*Q_loop*, *w'*, α) (*Q_loop*, [], [])
proof −
from *assms* **obtain** *q'* *w'* α' **where** *step1*: *pda.step*$_1$ *cfg_to_pda_pda* (*Q_loop*, *w*, *Tm a* # α) (*q'*, *w'*, α') **and**
steps: *pda.steps cfg_to_pda_pda* (*q'*, *w'*, α') (*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** ($\exists$ β_0. *w'* = *w* $\wedge$ α' = β_0@α $\wedge$ (*Q_loop*, β_0) $\in$ *pda_eps_of_cfg Q_loop* (*Tm a*))
$\vee$ ($\exists$ a_0 β_0. *w* = a_0#*w'* $\wedge$ α' = β_0@α $\wedge$ (*Q_loop*, β_0) $\in$ *pda_of_cfg Q_loop* a_0 (*Tm a*))
using *pda.step*$_1$*_rule*[*OF pda_cfg_to_pda*] **by** (*simp add*: *cfg_to_pda_pda_def*)
hence *w* = *a* # *w'* $\wedge$ α' = α
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* @ α) (*Q_loop*, [], [])
shows $\exists$ *w'*. *w* = *v* @ *w'* $\wedge$ *pda.steps cfg_to_pda_pda* (*Q_loop*, *w'*, α) (*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* $\vdash \alpha \Rightarrow l*$ *map Tm w*
shows *pda.steps cfg_to_pda_pda* (*Q_loop*, *w*, α) (*Q_loop*, [], [])
using *assms* **proof** (*induction rule*: *converse_rtranclp_induct*)
case *base*
then show *?case*
using *cfg_to_pda_cons_tms*[**where** *?w'* = [] **and** *?*γ = []] **by** *simp*
next
case (*step y z*)
from *step*(*1*) **obtain** *A* α *u1 u2* **where** *prod*: (*A*,α) $\in$ *Prods G* **and** *y_def*: *y* = *map Tm u1* @ *Nt A* # *u2* **and** *z_def*: *z* = *map Tm u1* @ α @ *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'*, α @ *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'*, α @ *u2*)
using *cfg_to_pda_cons_nt pda.step*$_1$*_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*, α) (*Q_loop*, [], [])
shows *Prods G* $\vdash \alpha \Rightarrow*$ *map Tm w*
using *assms* **proof** (*induction* (*Q_loop*, *w*, α) (*Q_loop*, [] :: *'t list*, [] :: (*'n*, *'t*) *syms*)
arbitrary: *w* α *rule*: *pda.steps_induct2*[*OF pda_cfg_to_pda*])
case (*3* w_1 α_1 p_2 w_2 α_2)
then show *?case* **proof** (*cases* α_1)
case (*Cons Z' α'*)
have p_2*_def*: p_2 = *Q_loop*
using *sing_st.exhaust* **by** *blast*
with *3*(*2*,*3*) **have** *IH*: *Prods G* $\vdash \alpha_2 \Rightarrow*$ *map Tm* w_2 **by** *simp*
show *?thesis* **proof** (*cases Z'*)
case (*Nt A*)
with *Cons* p_2*_def 3*(*1*) **obtain** α **where** *prod*: (*A*, α) $\in$ *Prods G* **and** α_2*_def*: α_2 = α @ α' **and** w_2*_def*: w_2 = w_1
using *cfg_to_pda_nt_cons* **by** *blast*
from *Cons Nt prod* α_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_α_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_α_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 derivels_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 ≡ {α. ∃ p q a z. (p, α) ∈ δ M q a z} ∪ {α.∃p q z. (p, α) ∈ δε M q z}

abbreviation *max_push* :: *nat* **where**
max_push ≡ *Suc* (*Max* (*length* ‘ *all_pushes*))

abbreviation *is_allowed_nt* :: ('q, 's) *pda_nt set* **where**
is_allowed_nt ≡ {*List_sym* p α q| p α q. *length* α ≤ *max_push*} ∪ (⋃p Z q. {*Single_sym* p Z q}) ∪ {*Start_sym*}

abbreviation *empty_rule* :: 'q ⇒ (('q, 's) *pda_nt*, 'a) *Prods* **where**
empty_rule q ≡ {(*List_sym* q [] q, [])}

abbreviation *trans_rule* :: 'q ⇒ 'q ⇒ 'a ⇒ 's ⇒ (('q, 's) *pda_nt*, 'a) *Prods* **where**
trans_rule q_0 q_1 a Z ≡ (λ(p, α). (*Single_sym* q_0 Z q_1, [*Tm* a, *Nt* (*List_sym* p α q_1)])) ‘ δ M q_0 a Z

abbreviation *eps_rule* :: 'q ⇒ 'q ⇒ 's ⇒ (('q, 's) *pda_nt*, 'a) *Prods* **where**
eps_rule q_0 q_1 Z ≡ (λ(p, α). (*Single_sym* q_0 Z q_1, [*Nt* (*List_sym* p α q_1)])) ‘ δε M q_0 Z

fun *split_rule* :: 'q ⇒ ('q, 's) *pda_nt* ⇒ (('q, 's) *pda_nt*, 'a) *Prods* **where**
split_rule q (*List_sym* p_0 (Z#α) p_1) = {(*List_sym* p_0 (Z#α) p_1, [*Nt* (*Single_sym* p_0 Z q), *Nt* (*List_sym* q α p_1)])}
| *split_rule* _ _ = {}

abbreviation *start_rule* :: 'q ⇒ (('q, 's) *pda_nt*, 'a) *Prods* **where**
start_rule q ≡ {(*Start_sym*, [*Nt* (*List_sym* (*init_state* M) [*init_symbol* M] q)])}

abbreviation *rule_set* :: (('q, 's) *pda_nt*, 'a) *Prods* **where**
rule_set ≡ (⋃q. *empty_rule* q) ∪ (⋃q p a Z. *trans_rule* q p a Z) ∪ (⋃q p Z. *eps_rule* q p Z) ∪

$\bigcup$ {*split_rule q nt*| *q nt*. *nt* ∈ *is_allowed_nt*} ∪ ($\bigcup$ *q*. *start_rule q*)

definition *G* :: (('*q*, '*s*) *pda_nt*,'*a*) *Cfg* **where**
G ≡ *Cfg rule_set Start_sym*

lemma *finite_is_allowed_nt*: *finite* (*is_allowed_nt*)
proof (*intro finite_UnI*)
show *finite* {*List_sym* (*p* :: '*q*) (α :: '*s list*) *q*| *p* α *q*. *length* α ≤ *max_push*}
proof −
let *?A* = $\bigcup$($\bigcup$((λ*s*. (λ*f*. *f* ‘ *UNIV*) ‘ *s*) ‘ ((λ*f*. *f* ‘ {*xs* :: '*s list*. *set xs* ⊆ *UNIV*
∧ *length xs* ≤ *max_push*}) ‘ (*List_sym* ‘ (*UNIV* :: '*q set*)))))

have {*List_sym p* α *q*| *p* α *q*. *length* α ≤ *max_push*} = *?A*
by *auto*

moreover have *finite ?A* (**is** *finite* ($\bigcup$ *?B*))
proof (*rule finite_Union*)
show *finite ?B* (**is** *finite* ($\bigcup$ *?C*))
proof (*rule finite_Union*)
show *finite ?C* **by** *simp*
next
show $\bigwedge$*M*. *M* ∈ *?C* ⟹ *finite M*
using *finite_lists_length_le*[*of UNIV max_push*] **by** *force*
qed
next
show $\bigwedge$*M*. *M* ∈ *?B* ⟹ *finite M* **by** *fastforce*
qed

ultimately show *?thesis* **by** *simp*
qed
next
show *finite* ($\bigcup$(*p* :: '*q*) (*Z* :: '*s*) *q*. {*Single_sym p Z q*})
by (*rule*, *simp*)+
qed *simp*

lemma *finite_split_rule*: *finite* (*split_rule q nt*)
by (*induction q nt rule*: *split_rule.induct*) *auto*

lemma *finite* (*Prods G*)
proof −
have *finite* ($\bigcup$ *q*. *empty_rule q*) **by** *simp*

moreover have *finite* ($\bigcup$ *q p a Z*. *trans_rule q p a Z*)
by (*simp add*: *finite_delta*)

moreover have *finite* ($\bigcup$ *q p Z*. *eps_rule q p Z*)
by (*simp add*: *finite_delta_eps*)

moreover have *finite* ($\bigcup$ {*split_rule q nt*| *q nt*. *nt* ∈ *is_allowed_nt*})

```
  proof −
    have {split_rule q nt| q nt. nt ∈ is_allowed_nt} = ⋃ ((λf. f ‘ is_allowed_nt)
‘ (split_rule ‘ UNIV))
      by fastforce

    moreover have finite (⋃ (⋃ ((λf. f ‘ is_allowed_nt) ‘ (split_rule ‘ UNIV))))
(is finite (⋃ ?A))
    proof (rule finite_Union)
      show finite ?A (is finite (⋃ ?B))
      proof (rule finite_Union)
        show finite ?B by simp
      next
        show ⋀M. M ∈ ?B ⟹ finite M
          using finite_is_allowed_nt by blast
      qed
    next
      show ⋀M. M ∈ ?A ⟹ finite M
        by (auto simp: finite_split_rule)
    qed

    ultimately show ?thesis by simp
  qed

  moreover have finite (⋃ q. start_rule q) by simp

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

lemma split_rule_simp:
  (A, w) ∈ split_rule q nt ⟷
   (∃ p0 Z α p1. nt = (List_sym p0 (Z#α) p1) ∧
                 A = List_sym p0 (Z#α) p1 ∧ w = [Nt (Single_sym p0 Z q), Nt
(List_sym q α p1)])
by (induction q nt rule: split_rule.induct) auto

lemma pda_to_cfg_derive_empty:
  Prods G ⊢ [Nt (List_sym p1 [] p2)] ⇒ x ⟷ p2 = p1 ∧ 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 = (λ(p, α). α) ‘ (⋃ q a Z. δ M q a Z ∪ (⋃ q Z. δε 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*, *α*) ∈ *δ M q a Z* ⟹ *length α* ≤ *max_push*
proof −
have (*p*, *α*) ∈ *δ M q a Z* ⟹ *length α* ≤ *Max* (*length* ‘ *all_pushes*)
by (*rule Max_ge*) (*use finite_all_pushes* **in** *blast*)+
thus (*p*, *α*) ∈ *δ M q a Z* ⟹ *length α* ≤ *max_push* **by** *simp*
qed

lemma *push_eps_leq_max*:
(*p*, *α*) ∈ *δε M q Z* ⟹ *length α* ≤ *max_push*
proof −
have (*p*, *α*) ∈ *δε M q Z* ⟹ *length α* ≤ *Max* (*length* ‘ *all_pushes*)
by (*rule Max_ge*) (*use finite_all_pushes* **in** *blast*)+
thus (*p*, *α*) ∈ *δε M q Z* ⟹ *length α* ≤ *max_push* **by** *simp*
qed

lemma *pda_to_cfg_derive_split*:
Prods G ⊢ [*Nt* (*List_sym p*$_1$ (*Z*#*α*) *p*$_2$)] ⇒ *w* ⟷
(∃ *q*. *length* (*Z*#*α*) ≤ *max_push* ∧ *w* = [*Nt* (*Single_sym p*$_1$ *Z q*), *Nt* (*List_sym q α p*$_2$)])
(**is** *?l* ⟷ *?r*)
proof
assume *?l*
hence (*List_sym p*$_1$ (*Z* # *α*) *p*$_2$, *w*) ∈ *rule_set*
using *derive_singleton*[*of Prods G Nt* (*List_sym p*$_1$ (*Z* # *α*) *p*$_2$) *w*] **by** (*simp add*: *G_def*)
thus *?r*
by (*auto simp*: *split_rule_simp*)
next
assume *?r*
then obtain *q* **where** *len_α*: *length* (*Z*#*α*) ≤ *max_push* **and** *w_def*: *w* = [*Nt* (*Single_sym p*$_1$ *Z q*), *Nt* (*List_sym q α p*$_2$)] **by** *blast*
from *w_def* **have** (*List_sym p*$_1$ (*Z*#*α*) *p*$_2$, *w*) ∈ *split_rule q* (*List_sym p*$_1$ (*Z* # *α*) *p*$_2$) **by** *simp*
with *len_α* **have** (*List_sym p*$_1$ (*Z*#*α*) *p*$_2$, *w*) ∈ ⋃ {*split_rule q nt*| *q nt*. *nt* ∈ *is_allowed_nt*}
by (*subst Union_iff*) *fast*
hence (*List_sym p*$_1$ (*Z*#*α*) *p*$_2$, *w*) ∈ *rule_set* **by** *simp*
thus *?l*
using *derive_singleton*[*of Prods G Nt* (*List_sym p*$_1$ (*Z* # *α*) *p*$_2$) *w*] **by** (*simp add*: *G_def*)
qed

lemma *pda_to_cfg_derive_single*:
Prods G ⊢ [*Nt* (*Single_sym q*$_0$ *Z q*$_1$)] ⇒ *w* ⟷
(∃ *p α a*. (*p*, *α*) ∈ *δ M q*$_0$ *a Z* ∧ *w* = [*Tm a*, *Nt* (*List_sym p α q*$_1$)]) ∨
(∃ *p α*. (*p*, *α*) ∈ *δε M q*$_0$ *Z* ∧ *w* = [*Nt* (*List_sym p α q*$_1$)])

unfolding *G_def* **using** *derive_singleton*[*of rule_set*] *split_rule_simp* **by** *fastforce*

lemma *pda_to_cfg_derive_start*:
Prods G ⊢ [*Nt Start_sym*] ⇒ *w* ⟷ (∃ *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*, *γ*) ⇝(*n*) (*p*, [], [])
and *length γ* ≤ *max_push*
shows *Prods G* ⊢ [*Nt* (*List_sym q γ p*)] ⇒∗ *map Tm x*
using *assms* **proof** (*induction n arbitrary*: *x p q γ rule*: *less_induct*)
case (*less n*)
then show *?case* **proof** (*cases γ*)
case *Nil*
from *less*(*2*) **have** (*q*, *x*, *γ*) ⇝∗ (*p*, [], [])
using *stepn_steps* **by** *blast*
with *Nil* **have** *q* = *p* ∧ *x* = []
using *steps_empty_stack* **by** *simp*
with *Nil* **show** *?thesis*
using *pda_to_cfg_derive_empty* **by** *auto*
next
case (*Cons Z α*)
with *less*(*2*) **obtain** *n′ q′ x′ γ′* **where** *n_def*: *n* = *Suc n′* **and**
step1: (*q*, *x*, *γ*) ⇝ (*q′*, *x′*, *γ′*) **and**
stepn: (*q′*, *x′*, *γ′*) ⇝(*n′*) (*p*, [], [])
using *stepn_not_refl_split_first* **by** *blast*
from *Cons step1* **have** *rule*: (∃ *β*. *x′* = *x* ∧ *γ′* = *β*@*α* ∧ (*q′*, *β*) ∈ *δε M q Z*)
∨ (∃ *a β*. *x* = *a* # *x′* ∧ *γ′* = *β*@*α* ∧ (*q′*,*β*) ∈ *δ M q a Z*) (**is** *?l* ∨ *?r*)
using *step*$_1$*_rule* **by** *simp*
show *?thesis* **proof** (*rule disjE*[*OF rule*])
assume *?l*
then obtain *β* **where** *x_def*: *x′* = *x* **and** *γ′_split*: *γ′* = *β*@*α* **and** *eps*: (*q′*, *β*) ∈ *δε M q Z* **by** *blast*
from *stepn γ′_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*, *β*) (*p′*, [], []) **and** *stepm2*: *stepn m*$_2$ (*p′*, *y′*, *α*) (*p*, [], [])
using *split_stack*[*of n′ q′ x′ β α 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_β*: *length β* ≤ *max_push*
using *push_eps_leq_max* **by** *blast*

from *Cons less*(*3*) **have** *Prods G* ⊢ [*Nt* (*List_sym q γ p*)] ⇒ [*Nt* (*Single_sym q Z p′*), *Nt* (*List_sym p′ α p*)]
using *pda_to_cfg_derive_split* **by** *simp*

moreover from *eps* **have** *Prods G* ⊢ [*Nt* (*Single_sym q Z p′*), *Nt* (*List_sym p′ α p*)] ⇒
[*Nt* (*List_sym q′ β p′*), *Nt* (*List_sym p′ α p*)]
using *pda_to_cfg_derive_single derive_append*[*of Prods G* [*Nt* (*Single_sym q Z p′*)] [*Nt* (*List_sym q′ β p′*)]
[*Nt* (*List_sym p′ α p*)]] **by** *simp*

moreover have *Prods G* ⊢ [*Nt* (*List_sym q′ β p′*), *Nt* (*List_sym p′ α p*)] ⇒∗ *map Tm y* @ [*Nt* (*List_sym p′ α p*)]
using *derives_append*[*OF less*(*1*)[*OF m1_less_n stepm1 len_β*]] **by** *simp*

moreover from *x_def x′_def Cons less*(*3*) **have** *Prods G* ⊢ *map Tm y* @ [*Nt* (*List_sym p′ α p*)] ⇒∗ *map 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 β* **where** *x_def*: *x* = *a* # *x′* **and** *γ′_split*: *γ′* = *β*@*α* **and** *trans*: (*q′*, *β*) ∈ *δ M q a Z* **by** *blast*
from *stepn γ′_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*, *β*) (*p′*, [], []) **and** *stepm2*: *stepn m*2 (*p′*, *y′*, *α*) (*p*, [], [])
using *split_stack*[*of n′ q′ x′ β α 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 *trans* **have** *len_β*: *length β* ≤ *max_push*
using *push_trans_leq_max* **by** *blast*

from *Cons less*(*3*) **have** *Prods G* ⊢ [*Nt* (*List_sym q γ p*)] ⇒ [*Nt* (*Single_sym q Z p′*), *Nt* (*List_sym p′ α p*)]
using *pda_to_cfg_derive_split* **by** *simp*

moreover from *trans* **have** *Prods G* ⊢ [*Nt* (*Single_sym q Z p′*), *Nt* (*List_sym p′ α p*)] ⇒
[*Tm a*, *Nt* (*List_sym q′ β p′*), *Nt* (*List_sym p′ α p*)]
using *pda_to_cfg_derive_single derive_append*[*of Prods G* [*Nt* (*Single_sym q Z p′*)] [*Tm a*, *Nt* (*List_sym q′ β p′*)]
[*Nt* (*List_sym p′ α p*)]] **by** *simp*

moreover have *Prods G* ⊢ [*Tm a*, *Nt* (*List_sym q′ β p′*), *Nt* (*List_sym p′ α p*)] ⇒∗
Tm a # *map Tm y* @ [*Nt* (*List_sym p′ α p*)]
using *derives_append*[*OF less*(*1*)[*OF m1_less_n stepm1 len_β*]] **by** (*simp add*: *derives_Tm_Cons*)

moreover from *x′_def x_def Cons less*(*3*) **have** *Prods G* ⊢ *Tm a* # *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 derivel_append_decomp:
  P ⊢ u@v ⇒l w ⟷
  (∃ u'. w = u'@v ∧ P ⊢ u ⇒l u') ∨ (∃ u' v'. w = u@v' ∧ u = map Tm u' ∧ P ⊢
v ⇒l v')
(is ?l ⟷ ?r)
proof
  assume ?l
  then obtain A r u1 u2
    where Ar: (A,r) ∈ P
      and uv: u@v = map Tm u1 @ Nt A # u2
      and w: w = map Tm u1 @ r @ u2
    by (auto simp: derivel_iff)
  from uv have case_dist: (∃ s. u2 = s @ v ∧ u = map Tm u1 @ Nt A # s) ∨
  (∃ s. map Tm u1 = u @ s  ∧ v = s @ Nt A # u2) (is ?h1 ∨ ?h2)
    by (auto simp: append_eq_append_conv2 append_eq_Cons_conv)
  show ?r proof (rule disjE[OF case_dist])
    assume ?h1
    with Ar w show ?thesis by (fastforce simp: derivel_iff)
  next
    assume ?h2
    then obtain s where map_u1_def: map Tm u1 = u @ s and v_def: v = s
@ Nt A # u2 by blast
    from map_u1_def obtain u' s' where u_def: u = map Tm u' and s_def: s
= map Tm s'
      using append_eq_map_conv[of u s Tm u1] by auto

    from w map_u1_def s_def have w = u @ (map Tm s' @ r @ u2) by simp

    moreover from Ar v_def s_def have P ⊢ v ⇒l map Tm s' @ r @ u2
      using derivel_iff[of P] by blast

    ultimately show ?thesis
      using u_def by blast
  qed
next
  show ?r ⟹ ?l
    by (auto simp add: derivel_append derivel_map_Tm_append)
qed
```

lemma *split_derivel'*:
assumes $P \vdash x\#v \Rightarrow l(n)\ u$
shows $(\exists\, u'.\ u = u' @ v \wedge P \vdash [x] \Rightarrow l(n)\ u') \vee (\exists\, w_1\ u_2\ m_1\ m_2.\ m_1 + m_2 = n \wedge u = map\ Tm\ w_1 @ u_2 \wedge P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1 \wedge P \vdash v \Rightarrow l(m_2)\ u_2)$
using *assms* **proof** (*induction n arbitrary*: *u*)
case (*Suc n*)
from *Suc(2)* **obtain** *w* **where** *x_v_deriveln_w*: $P \vdash x \#\ v \Rightarrow l(n)\ w$ **and** *w_derivel_u*: $P \vdash w \Rightarrow l\ u$ **by** *auto*
from *Suc(1)*[*OF x_v_deriveln_w*] **have** *IH*: $(\exists\, u'.\ w = u' @ v \wedge P \vdash [x] \Rightarrow l(n)\ u') \vee$
$(\exists\, w_1\ u_2\ m_1\ m_2.\ m_1 + m_2 = n \wedge w = map\ Tm\ w_1 @ u_2 \wedge P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1 \wedge P \vdash v \Rightarrow l(m_2)\ u_2)$ (**is** *?l* $\vee$ *?r*) **.**
show *?case* **proof** (*rule disjE*[*OF IH*])
assume *?l*
then obtain u' **where** *w_def*: $w = u' @ v$ **and** *x_deriveln_u'*: $P \vdash [x] \Rightarrow l(n)\ u'$ **by** *blast*
from *w_def w_derivel_u* **have** $P \vdash u' @ v \Rightarrow l\ u$ **by** *simp*
hence *case_dist*: $(\exists\, u_0.\ u = u_0 @ v \wedge P \vdash u' \Rightarrow l\ u_0) \vee$
$(\exists\, u_1\ u_2.\ u = u' @ u_2 \wedge u' = map\ Tm\ u_1 \wedge P \vdash v \Rightarrow l\ u_2)$ (**is** *?h1* $\vee$ *?h2*)
using *derivel_append_decomp*[*of P u' v u*] **by** *simp*
show *?thesis* **proof** (*rule disjE*[*OF case_dist*])
assume *?h1*
then obtain u_0 **where** *u_def*: $u = u_0 @ v$ **and** *u'_derivel_u0*: $P \vdash u' \Rightarrow l\ u_0$ **by** *blast*
from *x_deriveln_u' u'_derivel_u0* **have** $P \vdash [x] \Rightarrow l(Suc\ n)\ u_0$ **by** *auto*
with *u_def* **show** *?thesis* **by** *blast*
next
assume *?h2*
then obtain $u_1\ u_2$ **where** *u_def*: $u = u' @ u_2$ **and** *u'_def*: $u' = map\ Tm\ u_1$ **and** *v_derivel_u2*: $P \vdash v \Rightarrow l\ u_2$ **by** *blast*
from *x_deriveln_u' u'_def* **have** $P \vdash [x] \Rightarrow l(n)\ map\ Tm\ u_1$ **by** *simp*
with *u_def u'_def v_derivel_u2* **show** *?thesis* **by** *fastforce*
qed
next
assume *?r*
then obtain $w_1\ u_2\ m_1\ m_2$ **where** *m1_m2_n*: $m_1 + m_2 = n$ **and** *w_def*: $w = map\ Tm\ w_1 @ u_2$ **and**
x_derivelm1_w1: $P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1$ **and** *v_derivelm2_u2*: $P \vdash v \Rightarrow l(m_2)\ u_2$ **by** *blast*
from *w_def w_derivel_u* **have** $P \vdash map\ Tm\ w_1 @ u_2 \Rightarrow l\ u$ **by** *simp*
then obtain u' **where** *u_def*: $u = map\ Tm\ w_1 @ u'$ **and** *u2_derivel_u'*: $P \vdash u_2 \Rightarrow l\ u'$
using *derivel_map_Tm_append* **by** *blast*

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

moreover from *v_derivelm2_u2 u2_derivel_u'* **have** $P \vdash v \Rightarrow l(Suc\ m_2)\ u'$ **by** *auto*

ultimately show *?thesis*
using *u_def x_derivelm1_w1* **by** *blast*
qed
qed *simp*

lemma *split_derivel*:
assumes $P \vdash x\#v \Rightarrow l(n)\ map\ Tm\ w$
shows $\exists\, w_1\ w_2\ m_1\ m_2.\ m_1 + m_2 = n \wedge w = w_1 @ w_2 \wedge P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1 \wedge P \vdash v \Rightarrow l(m_2)\ map\ Tm\ w_2$
proof −
have *case_dist*: $(\exists\, u'.\ map\ Tm\ w = u' @ v \wedge P \vdash [x] \Rightarrow l(n)\ u') \vee (\exists\, w_1\ u_2\ m_1\ m_2.\ m_1 + m_2 = n \wedge map\ Tm\ w = map\ Tm\ w_1 @ u_2 \wedge P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1 \wedge P \vdash v \Rightarrow l(m_2)\ u_2)$ (**is** *?l* $\vee$ *?r*)
using *split_derivel'*[*OF assms*] **by** *simp*
show *?thesis* **proof** (*rule disjE*[*OF case_dist*])
assume *?l*
then obtain u' **where** *map_w_def*: $map\ Tm\ w = u' @ v$ **and** *x_derives_u'*: $P \vdash [x] \Rightarrow l(n)\ u'$ **by** *blast*
from *map_w_def* **obtain** $w_1\ w_2$ **where** $w = w_1 @ w_2$ **and** *map_w1_def*: $map\ Tm\ w_1 = u'$ **and** $map\ Tm\ w_2 = v$
using *map_eq_append_conv*[*of Tm w u' v*] **by** *blast*

moreover from *x_derives_u' map_w1_def* **have** $P \vdash [x] \Rightarrow l(n)\ map\ Tm\ w_1$ **by** *simp*

moreover have $P \vdash map\ Tm\ w_2 \Rightarrow l(0)\ map\ Tm\ w_2$ **by** *simp*

ultimately show *?thesis* **by** *force*
next
assume *?r*
then obtain $w_1\ u_2\ m_1\ m_2$ **where** *m1_m2_n*: $m_1 + m_2 = n$ **and** *map_w_def*: $map\ Tm\ w = map\ Tm\ w_1 @ u_2$ **and** *x_derivelm1_w1*: $P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1$ **and** *v_derivelm2_u2*: $P \vdash v \Rightarrow l(m_2)\ u_2$ **by** *blast*
from *map_w_def* **obtain** $w_1'\ u_2'$ **where** $w = w_1' @ u_2'$ **and** $map\ (Tm :: 'c \Rightarrow ('b, 'c)\ sym)\ w_1 = map\ Tm\ w_1'$ **and** $u_2 = map\ (Tm :: 'c \Rightarrow ('b, 'c)\ sym)\ u_2'$
using *map_eq_append_conv*[*of Tm* :: $'c \Rightarrow ('b, 'c)$ *sym w map Tm* $w_1\ u_2$] **by** *blast*
with *m1_m2_n x_derivelm1_w1 v_derivelm2_u2* **show** *?thesis* **by** *auto*
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 *derivels_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′ α p*)] ⇒*l γ″* **and**
ln″: *Prods G* ⊢ *γ″* ⇒*l*(*n″*) *map Tm x*
using *relpowp_Suc_E2*[*of n″ derivel* (*Prods G*) [*Nt* (*Single_sym q Z q′*), *Nt* (*List_sym q′ α p*)] *map Tm x*] **by** *blast*
from *l2* **obtain** *γ″*$_2$ **where** *l2′*: *Prods G* ⊢ [*Nt* (*Single_sym q Z q′*)] ⇒*l γ″*$_2$ **and** *γ″_split*: *γ″* = *γ″*$_2$ @ [*Nt* (*List_sym q′ α p*)]
using *derivel_Nt_Cons* **by** (*metis append.right_neutral*)
have (∃ *q″ α″ a*. (*q″*, *α″*) ∈ *δ M q a Z* ∧ *γ″*$_2$ = [*Tm a*, *Nt* (*List_sym q″ α″ q′*)]) ∨
(∃ *q″ α″*. (*q″*, *α″*) ∈ *δε M q Z* ∧ *γ″*$_2$ = [*Nt* (*List_sym q″ α″ 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**
γ''*_def*: $\gamma'' = [Tm\ a,\ Nt\ (List\_sym\ q''\ \alpha''\ q'),\ Nt\ (List\_sym\ q'\ \alpha\ p)]$ **by** *blast*
from γ''*_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'' α'' q'*), *Nt* (*List_sym q' α 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 *split_derivel*[*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 *step*$_1$*_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*$_1$ α'' @ α *q' [] α 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**
γ''*_def*: $\gamma'' = [Nt\ (List\_sym\ q''\ \alpha''\ q'),\ Nt\ (List\_sym\ q'\ \alpha\ p)]$ **by** *blast*
from γ''*_def ln''* **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 ⊢ [Nt (List_sym q'' α'' q')] ⇒l(m1)
map Tm w1
                    and m2_path: Prods G ⊢ [Nt (List_sym q' α p)] ⇒l(m2)
map Tm w2
      using split_derivel[OF split] by blast
    from m1_m2_n''' n_def n'_def have m1_lessn: m1 < n by simp
    from m1_m2_n''' n_def n'_def have m2_lessn: m2 < n by simp

    from eps Cons have (q, x, γ) ⇝ (q'', x, α'' @ α)
      using step1_rule by simp

    moreover from x_def have (q'', x, α'' @ α) ⇝* (q', w2, α)
      using steps_stack_app[OF less(1)[OF m1_lessn m1_path], of α]
          steps_word_app[of q'' w1 α'' @ α q' [] α w2] by simp

    moreover have (q', w2, α) ⇝* (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 ⊆ ?P
 proof
  fix x
  assume x ∈ ?L
  hence derives: Prods G ⊢ [Nt Start_sym] ⇒* map Tm x
    by (simp add: G_def Lang_def)
  then obtain γ where fs: Prods G ⊢ [Nt Start_sym] ⇒ γ and ls: Prods G ⊢
γ ⇒* map Tm x
    using converse_rtranclpE[OF derives] by blast
  from fs obtain q where γ = [Nt (List_sym (init_state M) [init_symbol M]
q)]
    using pda_to_cfg_derive_start[of γ] by blast
  with ls obtain n where Prods G ⊢ [Nt (List_sym (init_state M) [init_symbol
M] q)] ⇒l(n) map Tm x
    using derivels_iff_derives[of Prods G γ x] rtranclp_power[of derivel (Prods
G) γ 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 ∈ ?P
    by (auto simp: accept_stack_def)
 qed
next
 show ?P ⊆ ?L
```

```
  proof
    fix x
    assume x ∈ ?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]) ↝(n) (q, [], [])
      using stepn_steps by blast

    hence Prods G ⊢ [Nt (List_sym (init_state M) [init_symbol M] q)] ⇒* map
Tm x
      using pda_to_cfg_derives_if_stepn by simp

    moreover have Prods G ⊢ [Nt Start_sym] ⇒ [Nt (List_sym (init_state M)
[init_symbol M] q)]
      using pda_to_cfg_derive_start by simp

    ultimately have Prods G ⊢ [Nt (Start G)] ⇒* map Tm x
      by (simp add: G_def)

    thus x ∈ ?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.