Pushdown Automata

Kaan Taskin and Tobias Nipkow

February 6, 2026

Abstract

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

Contents

1 Pushdown Automata (PDA) 1
1.1 Definitions . . . 1
1.2 Basic Lemmas . . . 3
1.2.1 *step* and $step_1$ . . . 3
1.2.2 *steps* . . . 4
1.2.3 *stepn* . . . 5

2 Equivalence of Final and Stack Acceptance 7
2.1 Stack Acceptance to Final Acceptance . . . 7
2.2 Final Acceptance to Stack Acceptance . . . 10

3 Equivalence of CFG and PDA 14
3.1 CFG to PDA . . . 14
3.2 PDA to CFG . . . 16

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* $(\delta\ M\ p\ a\ Z) =$ *card* $\{(q,\ w,\ \beta@\alpha) \mid q\ \beta.\ (q,\ \beta) \in \delta\ M\ p\ a\ Z\}$
⟨*proof*⟩

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

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

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

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

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

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

lemma $step_1$*_nonempty_stack*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p_2,\ w_2,\ \alpha_2) \Longrightarrow \exists\, Z'\ \alpha'.\ \alpha_1 = Z'\#\alpha'$
⟨*proof*⟩

lemma $step_1$*_empty_stack*: $\neg\ (p_1,\ w_1,\ []) \rightsquigarrow (p_2,\ w_2,\ \alpha_2)$
⟨*proof*⟩

lemma $step_1$*_rule*: $(p_1,\ w_1,\ Z\#\alpha_1) \rightsquigarrow (p_2,\ w_2,\ \alpha_2) \longleftrightarrow (\exists\, \beta.\ w_2 = w_1 \wedge \alpha_2 = \beta@\alpha_1 \wedge (p_2,\ \beta) \in \delta\varepsilon\ M\ p_1\ Z)$
$\vee\ (\exists\, a\ \beta.\ w_1 = a\ \#\ w_2 \wedge \alpha_2 = \beta@\alpha_1 \wedge (p_2,\ \beta) \in \delta\ M\ p_1\ a\ Z)$
⟨*proof*⟩

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** $?l \longleftrightarrow\ ?r$)

⟨*proof*⟩

lemma *step$_1$_stack_app*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_1,\ w_1,\ \alpha_1\ @\ \gamma) \rightsquigarrow (p_2,\ w_2,\ \alpha_2\ @\ \gamma)$
⟨*proof*⟩

1.2.2 *steps*

lemma *steps_refl*: $(p,\ w,\ \alpha) \rightsquigarrow\!* (p,\ w,\ \alpha)$
⟨*proof*⟩

lemma *steps_trans*: $[\![(p_1,\ w_1,\ \alpha_1) \rightsquigarrow\!* (p_2,\ w_2,\ \alpha_2);\ (p_2,\ w_2,\ \alpha_2) \rightsquigarrow\!* (p_3,\ w_3,\ \alpha_3)]\!]$
$\Longrightarrow (p_1,\ w_1,\ \alpha_1) \rightsquigarrow\!* (p_3,\ w_3,\ \alpha_3)$
⟨*proof*⟩

lemma *step$_1$_steps*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_1,\ w_1,\ \alpha_1) \rightsquigarrow\!* (p_2,\ w_2,\ \alpha_2)$
⟨*proof*⟩

lemma *steps_empty_stack*: $(p_1,\ w_1,\ []) \rightsquigarrow\!* (p_2,\ w_2,\ \alpha_2) \Longrightarrow p_1 = p_2 \wedge w_1 = w_2 \wedge \alpha_2 = []$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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$_1$* $(p_1,\ w_1\ @\ w,$

$\alpha_1)$ $(p_2,\ w_2\ @\ w,\ \alpha_2)$
⟨*proof*⟩

lemma *decreasing_word*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow* (p_2,\ w_2,\ \alpha_2) \Longrightarrow \exists\, w.\ w_1 = w\ @\ w_2$
⟨*proof*⟩

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*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p_2,\ w_2,\ \alpha_2) \longleftrightarrow (p_1,\ w_1,\ \alpha_1) \rightsquigarrow(1)\ (p_2,\ w_2,\ \alpha_2)$
⟨*proof*⟩

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

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*⟩

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*
⟨*proof*⟩

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 $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow(n+m)\ (p_3,\ w_3,\ \alpha_3)$
⟨*proof*⟩

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*⟩

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*⟩

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

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*⟩

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*⟩

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 $\exists\, p'\ w'\ \alpha'.\ (p_1,\ w_1,\ \alpha_1) \rightsquigarrow (p',\ w',\ \alpha') \wedge (p',\ w',\ \alpha') \rightsquigarrow* (p_2,\ w_2,\ \alpha_2)$
⟨*proof*⟩

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 $\exists\, p'\ w'\ \alpha'.\ (p_1,\ w_1,\ \alpha_1) \rightsquigarrow* (p',\ w',\ \alpha') \wedge (p',\ w',\ \alpha') \rightsquigarrow (p_2,\ w_2,\ \alpha_2)$
⟨*proof*⟩

lemma *stepn_stack_app*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow(n)\ (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_1,\ w_1,\ \alpha_1\ @\ \beta) \rightsquigarrow(n)\ (p_2,\ w_2,\ \alpha_2\ @\ \beta)$
⟨*proof*⟩

lemma *steps_stack_app*: $(p_1,\ w_1,\ \alpha_1) \rightsquigarrow* (p_2,\ w_2,\ \alpha_2) \Longrightarrow (p_1,\ w_1,\ \alpha_1\ @\ \beta) \rightsquigarrow* (p_2,\ w_2,\ \alpha_2\ @\ \beta)$
⟨*proof*⟩

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*⟩

lemma *stepn_reads_input*:
assumes $(p_1,\ a\ \#\ w,\ \alpha_1) \rightsquigarrow(n)\ (p_2,\ [],\ \alpha_2)$

shows $\exists\, n'\ k\ q_1\ q_2\ \gamma_1\ \gamma_2.\ n = Suc\ n' \wedge k \leq n' \wedge (p_1,\ a\ \#\ w,\ \alpha_1) \rightsquigarrow(k)\ (q_1,\ a\ \#\ w,\ \gamma_1) \wedge$
$(q_1,\ a\ \#\ w,\ \gamma_1) \rightsquigarrow (q_2,\ w,\ \gamma_2) \wedge (q_2,\ w,\ \gamma_2) \rightsquigarrow(n'-k)\ (p_2,\ [],\ \alpha_2)$
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

instance *st_extended* :: (*finite*) *finite*
⟨*proof*⟩

instance *sym_extended* :: (*finite*) *finite*
⟨*proof*⟩

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)) ‘ ($\delta\varepsilon$ *M q Z*)
| *final_of_stack_delta_eps New_init New_sym* = {(*Old_st* (*init_state M*), [*Old_sym* (*init_symbol M*), *New_sym*])}
| *final_of_stack_delta_eps* (*Old_st q*) *New_sym* = {(*New_final*, [])}
| *final_of_stack_delta_eps* _ _ = {}

definition *final_of_stack_pda* :: (*'q st_extended*, *'a*, *'s sym_extended*) *pda* **where**
final_of_stack_pda ≡ ⦇ *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*⟩

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*)
⟨*proof*⟩

lemma *final_of_stack_pda_eps*:
(p, β) ∈ $\delta\varepsilon$ *M q Z* ⟷ (*Old_st p*, *map Old_sym* β) ∈ $\delta\varepsilon$ *final_of_stack_pda* (*Old_st q*) (*Old_sym Z*)
⟨*proof*⟩

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*⟩

abbreviation α*_with_new* :: *'s list* ⇒ *'s sym_extended list* **where**
α*_with_new* α ≡ *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* p_2, w_2, α*_with_new* α_2)
shows (p_1, w_1, α_1) ⇝ (p_2, w_2, α_2)
⟨*proof*⟩

lemma *final_of_stack_pda_from_old*:
assumes $pda.step_1\ final\_of\_stack\_pda\ (Old\_st\ p_1,\ w_1,\ \alpha_1)\ (p_2,\ w_2,\ \alpha_2)$
shows $(\exists\, p_2'.\ p_2 = Old\_st\ p_2') \vee p_2 = New\_final$
⟨*proof*⟩

lemma *final_of_stack_pda_no_step_final*:
$\neg pda.step_1\ final\_of\_stack\_pda\ (New\_final,\ w_1,\ \alpha_1)\ (p,\ w_2,\ \alpha_2)$
⟨*proof*⟩

lemma *final_of_stack_pda_from_oldn*:
assumes $pda.steps\ final\_of\_stack\_pda\ (Old\_st\ p_1,\ w_1,\ \alpha_1)\ (p_2,\ w_2,\ \alpha_2)$
shows $\exists\, q'.\ p_2 = Old\_st\ q' \vee p_2 = New\_final$
⟨*proof*⟩

lemma *final_of_stack_pda_to_old*:
assumes $pda.step_1\ final\_of\_stack\_pda\ (p_1,\ w_1,\ \alpha_1)\ (Old\_st\ p_2,\ w_2,\ \alpha_2)$
shows $(\exists\, q'.\ p_1 = Old\_st\ q') \vee p_1 = New\_init$
⟨*proof*⟩

lemma *final_of_stack_pda_bottom_elem*:
assumes $pda.steps\ final\_of\_stack\_pda\ (Old\_st\ p_1,\ w_1,\ \alpha\_with\_new\ \alpha_1)$
$(Old\_st\ p_2,\ w_2,\ \gamma)$
shows $\exists\, \alpha.\ \gamma = \alpha\_with\_new\ \alpha$
⟨*proof*⟩

lemma *final_of_stack_pda_stepn*:
$(p_1,\ w_1,\ \alpha_1) \rightsquigarrow (n)\ (p_2,\ w_2,\ \alpha_2) \longleftrightarrow$
$pda.stepn\ final\_of\_stack\_pda\ n\ (Old\_st\ p_1,\ w_1,\ \alpha\_with\_new\ \alpha_1)\ (Old\_st\ p_2,\ w_2,\ \alpha\_with\_new\ \alpha_2)$ (**is** $?l \longleftrightarrow ?r$)
⟨*proof*⟩

lemma *final_of_stack_pda_steps*:
$(p_1,\ w_1,\ \alpha_1) \rightsquigarrow * (p_2,\ w_2,\ \alpha_2) \longleftrightarrow$
$pda.steps\ final\_of\_stack\_pda\ (Old\_st\ p_1,\ w_1,\ \alpha\_with\_new\ \alpha_1)\ (Old\_st\ p_2,\ w_2,\ \alpha\_with\_new\ \alpha_2)$
⟨*proof*⟩

lemma *final_of_stack_pda_first_step*:
assumes $pda.step_1\ final\_of\_stack\_pda\ (New\_init,\ w_1,\ [New\_sym])\ (p_2,\ w_2,\ \alpha)$
shows $p_2 = Old\_st\ (init\_state\ M) \wedge w_2 = w_1 \wedge \alpha = [Old\_sym\ (init\_symbol\ M),\ New\_sym]$
⟨*proof*⟩

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 $pda.step_1\ final\_of\_stack\_pda\ (p_1,\ w_1,\ \alpha_1)\ (New\_final,\ w_2,\ \alpha_2)$
shows $\exists\, q.\ p_1 = Old\_st\ q \wedge w_1 = w_2 \wedge \alpha_1 = New\_sym\ \#\ \alpha_2$

⟨*proof*⟩

lemma *final_of_stack_pda_split_path*:
assumes *pda.stepn final_of_stack_pda* (*Suc* (*Suc n*)) (*New_init*, w_1, [*New_sym*]) (*New_final*, w_2, γ)
shows $\exists\, q.$ *pda.step*$_1$ *final_of_stack_pda* (*New_init*, w_1, [*New_sym*]) (*Old_st* (*init_state M*), w_1, [*Old_sym* (*init_symbol M*), *New_sym*]) ∧
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*⟩

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*⟩

lemma *accepted_final_of_stack*:
($\exists\, q.$ (*init_state M*, w, [*init_symbol M*]) ⇝* (q, [], [])) ⟷ ($\exists\, q\ \gamma.$ q ∈ *final_states final_of_stack_pda* ∧
pda.steps final_of_stack_pda (*init_state final_of_stack_pda*, w, [*init_symbol final_of_stack_pda*]) (q, [], γ)) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma *final_of_stack*: *pda.accept_stack M* = *pda.accept_final final_of_stack_pda*
⟨*proof*⟩

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*
⟨*proof*⟩

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

⟨*proof*⟩

instance *sym_extended* :: (*finite*) *finite*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* $\longleftrightarrow$ *?r*)
⟨*proof*⟩

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_step₁_drop*:
assumes *pda.step₁ 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, α_1) $\rightsquigarrow$ (p_2, w_2, α_2)
⟨*proof*⟩

lemma *stack_of_final_pda_from_old*:
assumes *pda.step₁ 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*⟩

lemma *stack_of_final_pda_from_final*:
assumes *pda.step₁ 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*⟩

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*⟩

lemma *stack_of_final_pda_to_old*:
assumes *pda.step₁ 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*
⟨*proof*⟩

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 $\exists\, \alpha.\ \gamma$ = *α_with_new* α
⟨*proof*⟩

lemma *stack_of_final_pda_stepn*:
(p_1, w_1, α_1) $\rightsquigarrow(n)$ (p_2, w_2, α_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*⟩

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)
⟨*proof*⟩

lemma *stack_of_final_pda_final_dump*:
pda.steps stack_of_final_pda (*New_final*, *w*, γ) (*New_final*, *w*, [])
⟨*proof*⟩

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*) ∧ $w_2 = w_1 \wedge \alpha$ = [*Old_sym* (*init_symbol M*), *New_sym*]
⟨*proof*⟩

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*⟩

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 ∃ *q k* γ. $k \leq n \wedge q \in$ *final_states M* ∧
pda.stepn stack_of_final_pda k (*Old_st* p_1, w_1, α_1) (*Old_st q*, w_2, γ) ∧
pda.step$_1$ *stack_of_final_pda* (*Old_st q*, w_2, γ) (*New_final*, w_2, γ) ∧
pda.stepn stack_of_final_pda ($n-k$) (*New_final*, w_2, γ) (*New_final*, w_2, α_2)
⟨*proof*⟩

lemma *stack_of_final_pda_split_path*:
assumes *pda.stepn stack_of_final_pda* (*Suc* (*Suc n*)) (*New_init*, w_1, [*New_sym*]) (*New_final*, w_2, γ)
shows ∃ *q k* α. $k \leq n \wedge q \in$ *final_states M* ∧ *pda.step*$_1$ *stack_of_final_pda* (*New_init*, w_1, [*New_sym*])
(*Old_st* (*init_state M*), w_1, [*Old_sym* (*init_symbol M*), *New_sym*]) ∧
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) ∧
pda.step$_1$ *stack_of_final_pda* (*Old_st q*, w_2, α) (*New_final*, w_2, α) ∧
pda.stepn stack_of_final_pda ($n-k$) (*New_final*, w_2, α) (*New_final*, w_2, γ)
⟨*proof*⟩

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 ∃ n'. *n* = *Suc* (*Suc* n')
⟨*proof*⟩

lemma *accepted_final_to_stack*:
(∃ *q* γ. *q* ∈ *final_states M* ∧ (*init_state M*, *w*, [*init_symbol M*]) ⇝* (*q*, [], γ))

$\longleftrightarrow$
(∃ *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*⟩

lemma *final_to_stack*:
pda.accept_final M = *pda.accept_stack stack_of_final_pda*
⟨*proof*⟩

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*⟩

instance *sym* :: (*finite*, *finite*) *finite*
⟨*proof*⟩

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*⟩

lemma *cfg_to_pda_cons_tm*:
pda.step$_1$ *cfg_to_pda_pda* (*Q_loop*, *a*#*w*, *Tm a*#γ) (*Q_loop*, *w*, γ)
⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩

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*⟩

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*′ ∧ β = β′
⟨*proof*⟩

lemma *cfg_to_pda_tm_stack_path*:
assumes *pda.steps cfg_to_pda_pda* (*Q_loop*, *w*, *Tm a* # α) (*Q_loop*, [], [])
shows ∃ *w*′. *w* = *a*#*w*′ ∧ *pda.steps cfg_to_pda_pda* (*Q_loop*, *w*′, α) (*Q_loop*, [], [])
⟨*proof*⟩

lemma *cfg_to_pda_tms_stack_path*:
assumes *pda.steps cfg_to_pda_pda* (*Q_loop*, *w*, *map Tm v* @ α) (*Q_loop*, [], [])
shows ∃ *w*′. *w* = *v* @ *w*′ ∧ *pda.steps cfg_to_pda_pda* (*Q_loop*, *w*′, α) (*Q_loop*, [], [])
⟨*proof*⟩

lemma *cfg_to_pda_accepts_if_G_derives*:
assumes *Prods G* ⊢ α ⇒*l*∗ *map Tm w*
shows *pda.steps cfg_to_pda_pda* (*Q_loop*, *w*, α) (*Q_loop*, [], [])
⟨*proof*⟩

lemma *G_derives_if_cfg_to_pda_accepts*:
assumes *pda.steps cfg_to_pda_pda* (*Q_loop*, *w*, α) (*Q_loop*, [], [])

shows *Prods G* $\vdash$ α $\Rightarrow*$ *map Tm w*
⟨*proof*⟩

lemma *cfg_to_pda*: *LangS G* = *pda.accept_stack cfg_to_pda_pda* (**is** *?L* = *?P*)
⟨*proof*⟩

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) ∈ $\delta\varepsilon$ *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)])) ‘ $\delta\varepsilon$ *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*

$p_0\ Z\ q)$, $Nt\ (List\_sym\ q\ \alpha\ p_1)])\}$
$\mid$ *split_rule* _ _ $= \{\}$

abbreviation *start_rule* :: $'q \Rightarrow ((' q, 's)\ pda\_nt, 'a)\ Prods$ **where**
start_rule $q \equiv \{(Start\_sym, [Nt\ (List\_sym\ (init\_state\ M)\ [init\_symbol\ M]\ q)])\}$

abbreviation *rule_set* :: $((' q, 's)\ pda\_nt, 'a)\ Prods$ **where**
rule_set $\equiv (\bigcup q.\ empty\_rule\ q) \cup (\bigcup q\ p\ a\ Z.\ trans\_rule\ q\ p\ a\ Z) \cup (\bigcup q\ p\ Z.$ *eps_rule* $q\ p\ Z) \cup$
$\bigcup \{split\_rule\ q\ nt \mid q\ nt.\ nt \in is\_allowed\_nt\} \cup (\bigcup q.\ start\_rule\ q)$

definition $G :: ((' q, 's)\ pda\_nt, 'a)\ Cfg$ **where**
$G \equiv Cfg\ rule\_set\ Start\_sym$

lemma *finite_is_allowed_nt*: *finite* (*is_allowed_nt*)
⟨*proof*⟩

lemma *finite_split_rule*: *finite* (*split_rule q nt*)
⟨*proof*⟩

lemma *finite* (*Prods G*)
⟨*proof*⟩

lemma *split_rule_simp*:
$(A, w) \in split\_rule\ q\ nt \longleftrightarrow$
$(\exists p_0\ Z\ \alpha\ p_1.\ nt = (List\_sym\ p_0\ (Z\#\alpha)\ p_1) \wedge$
$A = List\_sym\ p_0\ (Z\#\alpha)\ p_1 \wedge w = [Nt\ (Single\_sym\ p_0\ Z\ q), Nt\ (List\_sym\ q\ \alpha\ p_1)])$
⟨*proof*⟩

lemma *pda_to_cfg_derive_empty*:
$Prods\ G \vdash [Nt\ (List\_sym\ p_1\ []\ p_2)] \Rightarrow x \longleftrightarrow p_2 = p_1 \wedge x = []$
⟨*proof*⟩

lemma *finite_all_pushes*: *finite all_pushes*
⟨*proof*⟩

lemma *push_trans_leq_max*:
$(p, \alpha) \in \delta\ M\ q\ a\ Z \Longrightarrow length\ \alpha \leq max\_push$
⟨*proof*⟩

lemma *push_eps_leq_max*:
$(p, \alpha) \in \delta\varepsilon\ M\ q\ Z \Longrightarrow length\ \alpha \leq max\_push$
⟨*proof*⟩

lemma *pda_to_cfg_derive_split*:
$Prods\ G \vdash [Nt\ (List\_sym\ p_1\ (Z\#\alpha)\ p_2)] \Rightarrow w \longleftrightarrow$
$(\exists q.\ length\ (Z\#\alpha) \leq max\_push \wedge w = [Nt\ (Single\_sym\ p_1\ Z\ q), Nt\ (List\_sym\ q\ \alpha\ p_2)])$

(**is** *?l* $\longleftrightarrow$ *?r*)
⟨*proof*⟩

lemma *pda_to_cfg_derive_single*:
Prods G ⊢ [*Nt* (*Single_sym q*$_0$ *Z q*$_1$)] ⇒ *w* $\longleftrightarrow$
(∃ *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*$_1$)])
⟨*proof*⟩

lemma *pda_to_cfg_derive_start*:
Prods G ⊢ [*Nt Start_sym*] ⇒ *w* $\longleftrightarrow$ (∃ *q*. *w* = [*Nt* (*List_sym* (*init_state M*) [*init_symbol M*] *q*)])
⟨*proof*⟩

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*
⟨*proof*⟩

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*, [], [])
⟨*proof*⟩

lemma *pda_to_cfg*: *LangS G* = *accept_stack* (**is** *?L* = *?P*)
⟨*proof*⟩

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.