Deterministic Context-Free Languages are Closed Under Complementation (Hopcroft and Ullman)

Kaan Taskin and Tobias Nipkow

September 24, 2026

Abstract

A deterministic context-free language (DCFL) is a language accepted by a deterministic pushdown automaton (DPDA). This entry proves that the deterministic context-free languages are closed under complementation. The proof follows the one in the book by Hopcroft and Ullman (1979) [1].

Contents

1 Deterministic Pushdown Automata

theory *Det_Pushdown_Automata*
imports *Pushdown_Automata.Pushdown_Automata*
begin

A deterministic pushdown automaton, following Hopcroft and Ullman [1]:

locale *dpda* = *pda M* **for** *M* :: ('*q* :: *finite*, '*a* :: *finite*, '*s* :: *finite*) *pda* +

```
  assumes δ_nonempty: δ M q a X ≠ {} ⟶ δε M q X = {}
      and δ_singleton: δ M q a X = {} ∨ (∃ p γ. δ M q a X = {(p, γ)})
      and δε_singleton: δε M q X = {} ∨ (∃ p γ. δε M q X = {(p, γ)})
begin
```

In every configuration,

- *δ_nonempty* enforces that not both an epsilon step and a true step can be enabled.
- *δ_singleton* allows for at most one true step.
- *δε_singleton* allows for at most one epsilon step.

The automaton can take at most one step from any configuration:

```
lemma dpda_step: step (q, w, α) = {} ∨ (∃ p u γ. step (q, w, α) = {(p, u, γ)})
proof (cases α)
  case [simp]: (Cons X α')
  show ?thesis proof (cases w)
    case Nil
    then show ?thesis
      using δε_singleton[of q X] by auto
  next
    case [simp]: (Cons a w')
    consider (a) δ M q a X = {} | (b) δ M q a X ≠ {} by satx
    then show ?thesis proof cases
      case a
      then show ?thesis
        using δε_singleton[of q X] by auto
    next
      case b
      then show ?thesis
        using δ_nonempty[of q a X] δ_singleton[of q a X] by auto
    qed
  qed
qed auto
```

A step of the automaton is indeed deterministic:

```
lemma dpda_step₁_det:
  assumes (q, w, α) ⇝ (p, u, γ)
      and (q, w, α) ⇝ (p', u', γ')
    shows p = p' ∧ u = u' ∧ γ = γ'
using assms dpda_step by fastforce

lemma dpda_stepn_det:
  assumes (q, w, α) ⇝(n) (p, u, γ)
      and (q, w, α) ⇝(n) (p', u', γ')
    shows p = p' ∧ u = u' ∧ γ = γ'
using assms proof (induction (q, w, α) (p, u, γ) arbitrary: q w α rule: stepn_induct)
```

case (*stepn n q w α r u β*)
from *stepn*(*4*) **have** ∗: (*r*, *u*, *β*) ⇝(*n*) (*p′*, *u′*, *γ′*)
using *stepn_split_first*[*of q w α n p′ u′ γ′*] *dpda_step*$_1$*_det*[*OF stepn*(*1*)] **by** *auto*
from *stepn*(*3*)[*OF* ∗] **show** *?case* **.**
qed *auto*

end

end

2 DCFLs are Closed Under Complementation

theory *DPDA_Complement_HU*
imports *Det_Pushdown_Automata*
begin

2.1 Setup and Auxiliary Lemmas

context *pda* **begin**

definition *steps1* :: *′q* × *′a list* × *′s list* ⇒ *′q* × *′a list* × *′s list* ⇒ *bool* (**infix** ⇝+ *55*) **where**
steps1 ≡ *step*$_1$ ^++

abbreviation *no_step*$_1$ ((_ ⇝|) [*1000*] *999*) **where**
cf ⇝| ≡ (∀ *cf′*. ¬*cf* ⇝ *cf′*)

lemma *steps1_induct*[*consumes 1*, *case_names base step*]:
assumes *x1* ⇝+ *x2*
and ⋀*q w α p u γ*. (*q*, *w*, *α*) ⇝ (*p*, *u*, *γ*) ⟹ *P* (*q*, *w*, *α*) (*p*, *u*, *γ*)
and ⋀*q w α r v β p u γ*. (*q*, *w*, *α*) ⇝ (*r*, *v*, *β*) ⟹ (*r*, *v*, *β*) ⇝+ (*p*, *u*, *γ*) ⟹
P (*r*, *v*, *β*) (*p*, *u*, *γ*) ⟹ *P* (*q*, *w*, *α*) (*p*, *u*, *γ*)
shows *P x1 x2*
using *assms*[*unfolded steps1_def*]
proof(*induction rule*: *converse_tranclp_induct*)
case *base* **thus** *?case* **by** (*metis prod_cases3*)
next
case *step* **thus** *?case* **by** *simp* (*metis prod_cases3 step*$_1$*.simps*)
qed

lemma *steps1_converse_induct*[*consumes 1*, *case_names base step*]:
assumes *x1* ⇝+ *x2*
and ⋀*q w α p u γ*. (*q*, *w*, *α*) ⇝ (*p*, *u*, *γ*) ⟹ *P* (*q*, *w*, *α*) (*p*, *u*, *γ*)
and ⋀*q w α r v β p u γ*. (*q*, *w*, *α*) ⇝+ (*r*, *v*, *β*) ⟹ (*r*, *v*, *β*) ⇝ (*p*, *u*, *γ*) ⟹
P (*q*, *w*, *α*) (*r*, *v*, *β*) ⟹ *P* (*q*, *w*, *α*) (*p*, *u*, *γ*)
shows *P x1 x2*

using *assms*[*unfolded steps1_def*]
proof(*induction rule*: *tranclp_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

lemma *steps1_steps*:
$(q,\ w,\ \alpha) \rightsquigarrow\!+ (p,\ u,\ \gamma) \Longrightarrow (q,\ w,\ \alpha) \rightsquigarrow\!* (p,\ u,\ \gamma)$
by (*simp add*: *steps_def steps1_def*)

lemma *steps1_step*:
$(q,\ w,\ \alpha) \rightsquigarrow (p,\ u,\ \gamma) \Longrightarrow (q,\ w,\ \alpha) \rightsquigarrow\!+ (p,\ u,\ \gamma)$
by (*simp add*: *steps1_def tranclp.r_into_trancl*)

lemma *steps1_trans*:
$(q,\ w,\ \alpha) \rightsquigarrow\!+ (p,\ u,\ \gamma) \Longrightarrow (p,\ u,\ \gamma) \rightsquigarrow\!+ (r,\ v,\ \beta) \Longrightarrow (q,\ w,\ \alpha) \rightsquigarrow\!+ (r,\ v,\ \beta)$
using *steps1_def* **by** *force*

lemma *steps_steps1*:
assumes $(q,\ w,\ \alpha) \rightsquigarrow\!* (p,\ u,\ \gamma)$
and $(q,\ w,\ \alpha) \neq (p,\ u,\ \gamma)$
shows $(q,\ w,\ \alpha) \rightsquigarrow\!+ (p,\ u,\ \gamma)$
using *assms* **unfolding** *steps_def steps1_def* **by** (*meson rtranclpD*)

lemma *steps1_split_last*: $(\exists\, r\ v\ \beta.\ (q,\ w,\ \alpha) \rightsquigarrow\!* (r,\ v,\ \beta) \wedge (r,\ v,\ \beta) \rightsquigarrow (p,\ u,\ \gamma))$

$$\longleftrightarrow (q,\ w,\ \alpha) \rightsquigarrow\!+ (p,\ u,\ \gamma)$$

by (*smt* (*verit*, *ccfv_threshold*) *rtranclp_into_tranclp1 step*$_1$.*elims*(*2*) *steps1_def steps_def steps_refl tranclp.simps*
tranclp_into_rtranclp)

lemma *split_path*:
assumes $(q,\ w,\ \alpha) \rightsquigarrow(n)\ (p,\ [],\ \gamma)$
and $m \leq n$
shows $\exists\, r\ u\ \beta.\ (q,\ w,\ \alpha) \rightsquigarrow(m)\ (r,\ u,\ \beta) \wedge (r,\ u,\ \beta) \rightsquigarrow(n-m)\ (p,\ [],\ \gamma)$
using *assms* **proof** (*induction n*−*m arbitrary*: *m*)
case (*Suc x*)
from *Suc.prems*(*2*) **consider** (*a*) $m{=}n$ | (*b*) $m{<}n$ **by** *linarith*
then show *?case*
proof *cases*
case *a*
with *Suc.prems*(*1*) **show** *?thesis* **by** *auto*
next
case *b*
with *Suc.hyps*(*1*)[*of Suc m*] *Suc.hyps*(*2*) *Suc.prems*(*1*) **obtain** *r u β*
where *p1*: $(q,\ w,\ \alpha) \rightsquigarrow(Suc\ m)\ (r,\ u,\ \beta)$ **and** *p2*: $(r,\ u,\ \beta) \rightsquigarrow(n - Suc\ m)$

```
(p, [], γ) by fastforce
    from p1 obtain s v ζ where *: (q, w, α) ⇝(m) (s, v, ζ) and s: (s, v, ζ) ⇝
(r, u, β)
      using stepn_split_last[of m q w α r u β] by auto
    from s p2 b have **: (s, v, ζ) ⇝(n − m) (p, [], γ)
      using stepn_split_first[of s v ζ n − Suc m p [] γ] Suc_diff_Suc by force
    from * ** show ?thesis by blast
  qed
qed auto

end

context dpda begin

lemma max_eps_steps:
  assumes (q, w, α) ⇝(n) (p, u, γ)
      and (p, [], γ) ⇝|
      and (q, w, α) ⇝(m) (r, u, β)
    shows m ≤ n
proof (rule ccontr)
  assume ¬ m ≤ n
  then have n_less_m: Suc n ≤ m by simp
  from assms(1) obtain v where w_def: w = v@u
    using stepn_steps[of q w α p u γ] decreasing_word[of q w α p u γ] by blast
  from assms(3)[unfolded w_def] have p: (q, v, α) ⇝(m) (r, [], β)
    using stepn_word_app[of m q v α r [] β u] by simp
  obtain s y ζ where *: (q, v, α) ⇝(Suc n) (s, y, ζ)
    using split_path[OF p n_less_m] by fastforce
  from assms(1)[unfolded w_def] have **: (q, v, α) ⇝(n) (p, [], γ)
    using stepn_word_app[of n q v α p [] γ u] by simp
  from * ** have (p, [], γ) ⇝ (s, y, ζ)
    using stepn_split_last[of n q v α s y ζ] dpda_stepn_det[of n q v α p [] γ] by
metis
  with assms(2) show False by simp
qed

end
```

In what follows, we show that the complement of a deterministic context-free language is also a deterministic context-free language. For this purpose, we construct a deterministic pushdown automaton that recognizes the complement language of a given deterministic pushdown automaton. The proof follows that of Hopcroft and Ullman [1]. The construction is divided into two parts: First, construct an equivalent deterministic pushdown automaton that scans the entire input for all given input words, and then construct the automaton that recognizes the complement language out of this automaton.

2.2 Scan Construction

To scan the entire input, we address two complications: ensuring that every configuration has a possible step and that no configuration allows infinite epsilon steps. For the first problem, we introduce a new stack symbol to prevent the stack from becoming empty and a dead state that the automaton moves to when there are no possible steps. For the second problem, we introduce a new final state that the automaton moves to if an infinite number of epsilon steps pass through a final state; otherwise, the automaton moves to the dead state.

2.2.1 Definition

datatype *'q st_scan = St 'q | Q0' | D | F*
datatype *'s sym_scan = Sym 's | X0*

lemma *inj_Sym*: *inj Sym*
  by (*simp add*: *inj_def*)

instance *st_scan* :: (*finite*) *finite*
proof
  have ∗: *UNIV* = {*t*. ∃ *q*. *t* = *St q*} ∪ {*Q0'*, *D*, *F*}
    by *auto* (*metis st_scan.exhaust*)
  show *finite* (*UNIV* :: *'a st_scan set*)
    by (*simp add*: ∗ *full_SetCompr_eq*)
qed

instance *sym_scan* :: (*finite*) *finite*
proof
  have ∗: *UNIV* = {*t*. ∃ *q*. *t* = *Sym q*} ∪ {*X0*}
    by *auto* (*metis sym_scan.exhaust*)
  show *finite* (*UNIV* :: *'a sym_scan set*)
    by (*simp add*: ∗ *full_SetCompr_eq*)
qed

locale *dpda_scan* = *dpda M* **for** *M* :: (*'q* :: *finite*, *'a* :: *finite*, *'s* :: *finite*) *pda*
begin

definition *scan_dpda_final_states* :: *'q st_scan set* **where**
  scan_dpda_final_states ≡ *St* ‘ *final_states M* ∪ {*F*}

fun *scan_dpda_delta* :: *'q st_scan* ⇒ *'a* ⇒ *'s sym_scan* ⇒ (*'q st_scan* × *'s sym_scan list*) *set* **where**
  scan_dpda_delta (*St q*) *a* (*Sym X*) = (*if* δ *M q a X* = {} ∧ δε *M q X* = {} *then* {(*D*, [*Sym X*])}
                                else (λ(*q*, α). (*St q*, *map Sym* α)) ‘ δ *M q a X*)
| *scan_dpda_delta* (*St q*) _ *X0* = {(*D*, [*X0*])}
| *scan_dpda_delta D* _ *X* = {(*D*, [*X*])}
| *scan_dpda_delta* _ _ _ = {}

definition *eps_nonfinal* :: *'q* ⇒ *'s* ⇒ *bool* **where**
eps_nonfinal q X ≡ (∀ *i*. ∃ *p α*. (*q*, [], [*X*]) ⇝(*i*) (*p*, [], *α*) ∧ *p* ∉ *final_states M*)

definition *eps_final* :: *'q* ⇒ *'s* ⇒ *bool* **where**
eps_final q X ≡ (∀ *i*. ∃ *p α*. (*q*, [], [*X*]) ⇝(*i*) (*p*, [], *α*)) ∧ (∃ *i p α*. (*q*, [], [*X*]) ⇝(*i*) (*p*, [], *α*) ∧ *p* ∈ *final_states M*)

fun *scan_dpda_delta_eps* :: *'q st_scan* ⇒ *'s sym_scan* ⇒ (*'q st_scan* × *'s sym_scan list*) *set* **where**
scan_dpda_delta_eps Q0' X0 = {(*St* (*init_state M*), [*Sym* (*init_symbol M*), *X0*])}
| *scan_dpda_delta_eps* (*St q*) (*Sym X*) = (*if eps_nonfinal q X then* {(*D*, [*Sym X*])} *else*
if eps_final q X then {(*F*, [*Sym X*])} *else*
(λ(*q*, *α*). (*St q*, *map Sym α*)) ‘ *δε M q X*)
| *scan_dpda_delta_eps F X* = {(*D*, [*X*])}
| *scan_dpda_delta_eps* _ _ = {}

definition *scan_dpda* :: (*'q st_scan*, *'a*, *'s sym_scan*) *pda* **where**
scan_dpda =
(‖ *init_state* = *Q0'*,
init_symbol = *X0*,
final_states = *scan_dpda_final_states*,
delta = *scan_dpda_delta*, *delta_eps* = *scan_dpda_delta_eps* ‖)

2.2.2 Determinism

The automaton *scan_dpda* is deterministic:

lemma *dpda_scan_dpda*: *dpda scan_dpda*
proof (*standard*, *goal_cases*)
case (*1 p a Z*)
have *finite* (*scan_dpda_delta p a Z*)
by (*induction p a Z rule*: *scan_dpda_delta.induct*) (*auto simp*: *finite_delta*)
then show *?case* **by** (*simp add*: *scan_dpda_def*)
next
case (*2 p Z*)
have *finite* (*scan_dpda_delta_eps p Z*)
by (*induction p Z rule*: *scan_dpda_delta_eps.induct*) (*auto simp*: *finite_delta_eps*)
then show *?case* **by** (*simp add*: *scan_dpda_def*)
next
case (*3 q a X*)
have *scan_dpda_delta q a X* ≠ {} ⟶ *scan_dpda_delta_eps q X* = {}
proof (*induction q a X rule*: *scan_dpda_delta.induct*)
case (*1 q a X*)
then show *?case* **proof**
assume *a*: *scan_dpda_delta* (*St q*) *a* (*Sym X*) ≠ {}
have ∗: *δε M q X* = {} **proof** (*rule ccontr*)
assume *c*: *δε M q X* ≠ {}

from *a c* **have** $\delta\ M\ q\ a\ X \neq \{\}$ **by** *simp*
hence $\delta\varepsilon\ M\ q\ X = \{\}$
using *δ_nonempty*[*of q a X*] **by** *satx*
with *c* **show** *False* **by** *satx*
qed
hence $**$: $(q, [], [X]) \rightsquigarrow|$ **by** *simp*
from $**$ **have** $***$: $\neg$*eps_final q X*
by (*force simp*: *eps_final_def*)
from $**$ **have** $****$: $\neg$*eps_nonfinal q X*
by (*force simp*: *eps_nonfinal_def*)
from $*$ $***$ $****$ **show** *scan_dpda_delta_eps* (*St q*) (*Sym X*) $= \{\}$ **by** *simp*
qed
qed *simp_all*
then show *?case* **by** (*simp add*: *scan_dpda_def*)
next
case (*4 q a X*)
have *scan_dpda_delta q a X* $= \{\} \vee (\exists\, p\ \alpha.$ *scan_dpda_delta q a X* $= \{(p, \alpha)\})$
by (*induction q a X rule*: *scan_dpda_delta.induct*, *auto*) (*use δ_singleton* **in** *force*)+
then show *?case* **by** (*simp add*: *scan_dpda_def*)
next
case (*5 q X*)
have *scan_dpda_delta_eps q X* $= \{\} \vee (\exists\, p\ \alpha.$ *scan_dpda_delta_eps q X* $= \{(p, \alpha)\})$
by (*induction q X rule*: *scan_dpda_delta_eps.induct*, *auto*) (*use δε_singleton* **in** *force*)+
then show *?case* **by** (*simp add*: *scan_dpda_def*)
qed

2.2.3 Equivalence

sublocale *scan*: *dpda scan_dpda*
using *dpda_scan_dpda* **.**

We abbreviate the definitions of *scan_dpda* with index *s*:

notation $scan.step_1$ (**infix** $\rightsquigarrow_s$ *55*)
notation *scan.steps* (**infix** $\rightsquigarrow_s*$ *55*)

abbreviation *stack_with_X0* :: *'s list* $\Rightarrow$ *'s sym_scan list* **where**
stack_with_X0 α $\equiv$ *map Sym α* @ [*X0*]

lemma *scan_dpda_first_step*:
assumes $(Q0', w, [X0]) \rightsquigarrow_s (q, u, \alpha)$
shows $q = St\ (init\_state\ M) \wedge u = w \wedge \alpha = [Sym\ (init\_symbol\ M), X0]$
using *assms scan.step*$_1$*_rule* **by** (*simp add*: *scan_dpda_def*)

lemma *scan_dpda_step_from_St*:
assumes $(St\ q, w, \alpha) \rightsquigarrow_s (p, u, \gamma)$
shows $(\exists\, p'.\ p = St\ p') \vee p = F \vee p = D$
proof −

from *assms* **obtain** X **where**
$(\exists\,\beta.\ (p,\ \beta) \in scan\_dpda\_delta\_eps\ (St\ q)\ X) \vee (\exists\,a\ \beta.\ (p,\beta) \in scan\_dpda\_delta\ (St\ q)\ a\ X)$ (**is** *?a* $\vee$ *?b*)
using $scan.step_1\_rule\_ext$ *scan_dpda_def* **by** *fastforce*
then consider (*a*) *?a* | (*b*) *?b* **by** *blast*
thus *?thesis*
proof *cases*
case *a*
then show *?thesis* **by** (*induction St q X rule*: *scan_dpda_delta_eps.induct*) (*auto split*: *if_splits*)
next
case *b*
then obtain *a* **where** $\exists\,\beta.\ (p,\ \beta) \in scan\_dpda\_delta\ (St\ q)\ a\ X$ **by** *blast*
then show *?thesis* **by** (*induction St q a X rule*: *scan_dpda_delta.induct*) (*auto split*: *if_splits*)
qed
qed

lemma *scan_dpda_step_from_F*:
assumes $(F,\ w,\ \alpha) \rightsquigarrow_s (q,\ u,\ \gamma)$
shows $q = D$
using *assms* $scan.step_1\_rule\_ext$[*of F w* α *q u* γ] *scan_dpda_def* **by** *auto*

lemma *scan_dpda_step_to_F*:
assumes $(q,\ w,\ X\#\alpha) \rightsquigarrow_s (F,\ u,\ \gamma)$
shows $u = w \wedge (\exists\,q'\ X'.\ q = St\ q' \wedge X = Sym\ X' \wedge eps\_final\ q'\ X')$
proof −
from *assms* **have** *cases*: $(\exists\,\beta.\ u = w \wedge \gamma = \beta\ @\ \alpha \wedge (F,\ \beta) \in scan\_dpda\_delta\_eps\ q\ X)$
$\vee\ (\exists\,a\ \beta.\ w = a\ \#\ u \wedge \gamma = \beta\ @\ \alpha \wedge (F,\ \beta) \in scan\_dpda\_delta\ q\ a\ X)$ (**is** *?a* $\vee$ *?b*)
using $scan.step_1\_rule$[*of q w X* α *F u* γ] *scan_dpda_def* **by** *simp*
from *cases* **consider** (*a*) *?a* | (*b*) *?b* **by** *blast*
then show *?thesis*
proof *cases*
case *a*
then show *?thesis* **by** (*induction q X rule*: *scan_dpda_delta_eps.induct*) (*auto split*: *if_splits*)
next
case *b*
then obtain *a* **where** $\exists\,\beta.\ w = a\ \#\ u \wedge \gamma = \beta\ @\ \alpha \wedge (F,\ \beta) \in scan\_dpda\_delta\ q\ a\ X$ **by** *blast*
then show *?thesis* **by** (*induction q a X rule*: *scan_dpda_delta.induct*) (*auto split*: *if_splits*)
qed
qed

lemma *scan_dpda_step_from_D*:
assumes $(D,\ w,\ \alpha) \rightsquigarrow_s (q,\ u,\ \gamma)$

shows $q = D$
using *assms scan.step*$_1$*_rule_ext*[*of D w α q u γ*] *scan_dpda_def* **by** *auto*

lemma *scan_dpda_steps_from_St*:
assumes (*St q*, *w*, *α*) $\rightsquigarrow_s*$ (*p*, *u*, *γ*)
shows $(\exists\, p'.\ p = St\ p') \vee p = F \vee p = D$
using *assms* **by** (*induction* (*St q*, *w*, *α*) (*p*, *u*, *γ*) *arbitrary*: *p u γ rule*: *scan.steps_induct2_bw*, *simp*)
(*use scan_dpda_step_from_St scan_dpda_step_from_F scan_dpda_step_from_D*
in *blast*)

lemma *scan_dpda_step_to_St*:
assumes (*q*, *w*, *α*) $\rightsquigarrow_s$ (*St p*, *u*, *γ*)
shows $q = Q0' \vee (\exists\, q'.\ q = St\ q')$
using *assms scan_dpda_step_from_F*[*of w α St p u γ*] *scan_dpda_step_from_D*[*of w α St p u γ*]
by (*metis st_scan.exhaust st_scan.simps*(*5*))

lemma *scan_dpda_stack_with_X0*:
assumes (*St q*, *w*, *stack_with_X0 α*) $\rightsquigarrow_s*$ (*St p*, *u*, *γ*)
shows $\exists\gamma'.\ \gamma = stack\_with\_X0\ \gamma'$
using *assms* **proof** (*induction* (*St q*, *w*, *stack_with_X0 α*) (*St p*, *u*, *γ*) *arbitrary*: *p u γ rule*: *scan.steps_induct2_bw*)
case (*step r u γ v β*)
from *step*(*1*,*2*) **obtain** r' **where** *r_St*[*simp*]: $r = St\ r'$
using *scan_dpda_steps_from_St*[*of q w stack_with_X0 α r u γ*] *scan_dpda_step_to_St*[*of r u γ p v β*] **by** *blast*
from *step*(*3*)[*OF r_St*] **obtain** γ'' **where** *γ_X0*: $\gamma = stack\_with\_X0\ \gamma''$ **by** *blast*
from *step*(*2*) **obtain** *X* γ' β' **where** *γ_def*: $\gamma = X\#\gamma'$ **and** *β_def*: $\beta = \beta' @ \gamma'$ **and**
cases: $(St\ p, \beta') \in scan\_dpda\_delta\_eps\ (St\ r')\ X \vee (\exists\, a.\ (St\ p, \beta') \in scan\_dpda\_delta\ (St\ r')\ a\ X)$ (**is** *?a* ∨ *?b*)
using *scan.step*$_1$*_rule_ext*[*of r u γ St p v β*] *scan_dpda_def* **by** *auto*
from *cases* **consider** (*a*) *?a* | (*b*) *?b* **by** *blast*
then have *X_and_β'_def*: $(\exists\, X'.\ X = Sym\ X') \wedge (\exists\, \beta''.\ \beta' = map\ Sym\ \beta'')$
proof *cases*
case *a*
then show *?thesis* **by** (*induction St r' X rule*: *scan_dpda_delta_eps.induct*) (*auto split*: *if_splits*)
next
case *b*
then obtain *a* **where** $(St\ p, \beta') \in scan\_dpda\_delta\ (St\ r')\ a\ X$ **by** *blast*
then show *?thesis* **by** (*induction St r' a X rule*: *scan_dpda_delta.induct*) (*auto split*: *if_splits*)
qed
from *X_and_β'_def*[*THEN conjunct1*] *γ_X0 γ_def* **have** $\exists\gamma'''.\ \gamma' = stack\_with\_X0\ \gamma'''$
by (*metis hd_append list.sel*(*1*,*3*) *map_tl sym_scan.distinct*(*1*) *tl_append_if*)

with *X_and_β'_def*[*THEN conjunct2*] *β_def* **show** *?case*
by (*metis append.assoc map_append*)
qed *blast*

lemma *scan_dpda_trans*:
assumes (*St q*, *map Sym α*) ∈ *δ scan_dpda* (*St p*) *a* (*Sym X*)
shows (*q*, *α*) ∈ *δ M p a X*
using *assms* **by** (*auto simp*: *scan_dpda_def inj_map_eq_map*[*OF inj_Sym*] *split*: *if_splits*)

lemma *scan_dpda_eps*:
assumes (*St q*, *map Sym α*) ∈ *δε scan_dpda* (*St p*) (*Sym X*)
shows (*q*, *α*) ∈ *δε M p X*
using *assms* **by** (*auto simp*: *scan_dpda_def inj_map_eq_map*[*OF inj_Sym*] *split*: *if_splits*)

lemma *scan_dpda_step*:
assumes (*St q*, *w*, *map Sym α*) ⇝$_s$ (*St p*, *u*, *map Sym γ*)
shows (*q*, *w*, *α*) ⇝ (*p*, *u*, *γ*)
proof −
from *assms* **obtain** *X α'* **where** *αSym_def*: *map Sym α* = *Sym X* # *map Sym α'* **and** *rule*:
(∃ *β*. *u* = *w* ∧ *map Sym γ* = *β* @ *map Sym α'* ∧ (*St p*, *β*) ∈ *δε scan_dpda* (*St q*) (*Sym X*)) ∨
(∃ *a β*. *w* = *a* # *u* ∧ *map Sym γ* = *β* @ *map Sym α'* ∧ (*St p*, *β*) ∈ *δ scan_dpda* (*St q*) *a* (*Sym X*))
using *scan.step*$_1$*_rule_ext*[*of St q w map Sym α St p u map Sym γ*] **by** *auto*
from *αSym_def* **have** *α_def*: *α* = *X*#*α'*
using *inj_map_eq_map*[*OF inj_Sym*] **by** *auto*
from *rule* **have** (∃ *β*. *u* = *w* ∧ *map Sym γ* = *map Sym β* @ *map Sym α'* ∧ (*St p*, *map Sym β*) ∈ *δε scan_dpda* (*St q*) (*Sym X*)) ∨
(∃ *a β*. *w* = *a* # *u* ∧ *map Sym γ* = *map Sym β* @ *map Sym α'* ∧ (*St p*, *map Sym β*) ∈ *δ scan_dpda* (*St q*) *a* (*Sym X*))
using *append_eq_map_conv*[**where** *?f* = *Sym*] **by** *metis*
then have (∃ *β*. *u* = *w* ∧ *γ* = *β*@*α'* ∧ (*p*, *β*) ∈ *δε M q X*) ∨ (∃ *a β*. *w* = *a*#*u* ∧ *γ* = *β*@*α'* ∧ (*p*, *β*) ∈ *δ M q a X*)
using *scan_dpda_trans scan_dpda_eps* **by** (*metis inj_Sym inj_map_eq_map map_append*)
with *α_def* **show** *?thesis*
using *step*$_1$*_rule* **by** *simp*
qed

lemma *scan_dpda_stepX0*:
assumes (*St q*, *w*, *stack_with_X0 α*) ⇝$_s$ (*St p*, *u*, *stack_with_X0 γ*)
shows (*q*, *w*, *α*) ⇝ (*p*, *u*, *γ*)
proof −
from *assms* **obtain** *X α'* **where** *α_def*: *stack_with_X0 α* = *X*#*α'* **and**
cases: (∃ *β*. (*St p*, *β*) ∈ *scan_dpda_delta_eps* (*St q*) *X*) ∨ (∃ *a β*. (*St p*, *β*) ∈ *scan_dpda_delta* (*St q*) *a X*) (**is** *?a* ∨ *?b*)

using *scan.step*$_1$*_rule_ext*[*of St q w stack_with_X0 α St p u stack_with_X0 γ*] *scan_dpda_def* **by** *force*
from *cases* **consider** (*a*) *?a* | (*b*) *?b* **by** *blast*
then have ∃ *X'*. *X* = *Sym X'*
proof *cases*
case *a*
then show *?thesis* **by** (*induction St q X rule*: *scan_dpda_delta_eps.induct*) *auto*
next
case *b*
then obtain *a* **where** ∃ *β*. (*St p*, *β*) ∈ *scan_dpda_delta* (*St q*) *a X* **by** *blast*
then show *?thesis* **by** (*induction St q a X rule*: *scan_dpda_delta.induct*) *auto*
qed
with *α_def* **have** *map Sym α* ≠ [] **by** *auto*
then have (*St q*, *w*, *map Sym α*) ⇝$_s$ (*St p*, *u*, *map Sym γ*)
using *scan.step*$_1$*_stack_drop*[*OF assms*] **by** *simp*
then show *?thesis*
using *scan_dpda_step* **by** *presburger*
qed

The original automaton can mimic the steps of the automaton *scan_dpda* in the old states:

lemma *scan_dpda_stepsX0*:
assumes (*St q*, *w*, *stack_with_X0 α*) ⇝$_s$* (*St p*, *u*, *stack_with_X0 γ*)
shows (*q*, *w*, *α*) ⇝* (*p*, *u*, *γ*)
using *assms* **proof** (*induction* (*St q*, *w*, *stack_with_X0 α*) (*St p*, *u*, *stack_with_X0 γ*) *arbitrary*: *p u γ rule*: *scan.steps_induct2_bw*)
case *base*
then show *?case*
by (*simp add*: *inj_Sym steps_refl*)
next
case (*step r u β v*)
obtain *r'* **where** *r_def*: *r* = *St r'*
using *scan_dpda_steps_from_St*[*OF step(1)*] *scan_dpda_step_to_St*[*OF step(2)*]
by *blast*
obtain *β'* **where** *β_def*: *β* = *stack_with_X0 β'*
using *scan_dpda_stack_with_X0*[*OF step(1)*[*simplified r_def*]] **by** *blast*
from *step(3)*[*OF r_def β_def*] **have** ∗: (*q*, *w*, *α*) ⇝* (*r'*, *u*, *β'*) **.**
have ∗∗: (*r'*, *u*, *β'*) ⇝ (*p*, *v*, *γ*)
using *scan_dpda_stepX0*[*OF step(2)*[*simplified r_def β_def*]] **.**
show *?case*
using *steps_trans*[*OF* ∗ *step*$_1$*_steps*[*OF* ∗∗]] **.**
qed

If a pair of a state and a stack symbol allows infinite epsilon steps then the rest of the stack content stays untouched:

lemma *stack_cycle_drop*:
assumes ∀ *i*. ∃ *p α*. (*q*, [], [*X*]) ⇝(*i*) (*p*, [], *α*)
and (*q*, [], *X*#*γ*) ⇝* (*r*, [], *β*)

shows $\exists\, Y\ \beta'.\ \beta = Y\ \#\ \beta' \ @\ \gamma \wedge (q, [], [X]) \rightsquigarrow* (r, [], Y\#\beta')$
using *assms(2)* **proof** (*induction* $(q, [] :: 'a\ list, X\#\gamma)$ $(r, [] :: 'a\ list, \beta)$ *arbitrary*: $r\ \beta$ *rule*: *steps_induct2_bw*)
case *base*
then show *?case*
by (*simp add*: *steps_refl*)
next
case (*step p w α r β*)
have *w_def*: $w = []$
using *decreasing_word*[*OF step(1)*] **by** *simp*
from *step(3)*[*OF w_def*] **obtain** $Y\ \beta'$ **where** *α_def*: $\alpha = Y\ \#\ \beta' \ @\ \gamma$ **and** *p1*: $(q, [], [X]) \rightsquigarrow* (p, [], Y\ \#\ \beta')$ **by** *blast*
from *step(2)* *α_def* **obtain** β'' **where** *β_def*: $\beta = \beta'' \ @\ \beta' \ @\ \gamma$
using *step₁_rule*[*of p w Y β' @ γ r [] β*] **by** *blast*
from *step(2)*[*unfolded w_def α_def β_def*] **have** *p2*: $(p, [], Y\ \#\ \beta') \rightsquigarrow (r, [], \beta'' \ @\ \beta')$
using *step₁_stack_drop*[*of p [] Y # β' γ r [] β'' @ β'*] **by** *simp*
have $*$: $(q, [], [X]) \rightsquigarrow* (r, [], \beta'' \ @\ \beta')$
using *steps_trans*[*OF p1 step₁_steps*[*OF p2*]] **.**
from $*$ **obtain** n **where** *p3*: $(q, [], [X]) \rightsquigarrow(n) (r, [], \beta'' \ @\ \beta')$
using *stepn_steps*[*of q [] [X] r [] β'' @ β'*] **by** *presburger*
from *assms(1)* **obtain** $s\ \zeta$ **where** *p4*: $(q, [], [X]) \rightsquigarrow(Suc\ n) (s, [], \zeta)$ **by** *presburger*
from *p4* **have** $**$: $(r, [], \beta'' \ @\ \beta') \rightsquigarrow (s, [], \zeta)$
using *stepn_split_last*[*of n q [] [X] s [] ζ*] *dpda_stepn_det*[*OF p3*] **by** *auto*
from *β_def* $*$ **show** *?case*
using *step₁_nonempty_stack*[*OF* $**$] **by** *auto*
qed

The automaton *scan_dpda* can either mimic the steps of the original automaton or detect a cycle:

lemma *scan_dpda_steps*:
assumes $(q, w, \alpha) \rightsquigarrow* (p, u, \gamma)$
shows $(St\ q, w, map\ Sym\ \alpha) \rightsquigarrow_s* (St\ p, u, map\ Sym\ \gamma) \vee$
$(\exists\, r\ X\ \beta.\ (St\ q, w, map\ Sym\ \alpha) \rightsquigarrow_s* (St\ r, u, Sym\ X\ \#\ map\ Sym\ \beta) \wedge (r, [], X\#\beta) \rightsquigarrow* (p, [], \gamma) \wedge (\forall\, i.\ \exists\, s\ \Delta.\ (r, [], [X]) \rightsquigarrow(i) (s, [], \Delta)))$
using *assms* **proof** (*induction* (q, w, α) (p, u, γ) *arbitrary*: $p\ u\ \gamma$ *rule*: *steps_induct2_bw*)
case *base*
then show *?case*
by (*simp add*: *scan.steps_refl*)
next
case (*step p u γ r v β*)
from *step(2)* **obtain** $X\ \gamma'$ **where** *γ_def*: $\gamma = X\#\gamma'$ **and**
cases: $(\exists\, \zeta.\ v = u \wedge \beta = \zeta \ @\ \gamma' \wedge (r, \zeta) \in \delta\varepsilon\ M\ p\ X) \vee (\exists\, a\ \zeta.\ u = a\ \#\ v \wedge \beta = \zeta \ @\ \gamma' \wedge (r, \zeta) \in \delta\ M\ p\ a\ X)$ (**is** *?a* $\vee$ *?b*)
using *step₁_rule_ext*[*of p u γ r v β*] **by** *blast*
from *cases* **consider** (*a*) *?a* | (*b*) *?b* **by** *blast*
then show *?case*
proof *cases*
case *a*

then obtain ζ **where** *v_def*: $u = v$ **and** β*_def*: $\beta = \zeta$ @ γ' **and** *elem*: $(r, \zeta) \in \delta\varepsilon\ M\ p\ X$ **by** *blast*
from *step*(*3*) **consider** (*a1*) (*St q*, *w*, *map Sym* α) $\rightsquigarrow_s*$ (*St p*, *u*, *map Sym* γ) |
(*a2*) $\exists\, r\ X\ \beta$. (*St q*, *w*, *map Sym* α) $\rightsquigarrow_s*$ (*St r*, *u*, *Sym X* # *map Sym* β) $\wedge$ (r, [], X # β) $\rightsquigarrow*$ (p, [], γ) $\wedge$ ($\forall\, i.\ \exists\, s\ \Delta.$ (r, [], [X]) $\rightsquigarrow(i)$ (s, [], Δ)) **by** *blast*
then show *?thesis*
proof *cases*
case *a1*
consider (*a11*) $\forall\, i.\ \exists\, s\ \Delta.$ (p, [], [X]) $\rightsquigarrow(i)$ (s, [], Δ) | (*a12*) $\neg(\forall\, i.\ \exists\, s\ \Delta.$ (p, [], [X]) $\rightsquigarrow(i)$ (s, [], Δ)) **by** *blast*
then show *?thesis*
proof *cases*
case *a11*
from *a1*[*unfolded v_def* γ*_def*] **have** $*$: (*St q*, *w*, *map Sym* α) $\rightsquigarrow_s*$ (*St p*, *v*, *Sym X* # *map Sym* γ') **by** *simp*
from *elem* β*_def* **have** $**$: (p, [], X # γ') $\rightsquigarrow*$ (r, [], β)
using *step*$_1$*_rule*[*of p* [] *X* γ' *r* [] β] *step*$_1$*_steps* **by** *blast*
from $*$ $**$ *a11* **show** *?thesis* **by** *blast*
next
case *a12*
then have $\neg$*eps_nonfinal p X* $\wedge$ $\neg$*eps_final p X*
by (*auto simp*: *eps_nonfinal_def eps_final_def*)
with *elem* **have** (*St r*, *map Sym* ζ) $\in \delta\varepsilon$ *scan_dpda* (*St p*) (*Sym X*)
by (*auto simp*: *scan_dpda_def*)
with *v_def* γ*_def* β*_def* **have** $*$: (*St p*, *u*, *map Sym* γ) $\rightsquigarrow_s$ (*St r*, *v*, *map Sym* β)
using *scan.step*$_1$*_rule*[*of St p u Sym X map Sym* γ' *St r u map Sym* ζ @ *map Sym* γ'] **by** *simp*
show *?thesis*
using *scan.steps_trans*[*OF a1 scan.step*$_1$*_steps*[*OF* $*$]] **by** *satx*
qed
next
case *a2*
then obtain s Y μ **where** $*$: (*St q*, *w*, *map Sym* α) $\rightsquigarrow_s*$ (*St s*, *u*, *Sym Y* # *map Sym* μ) **and** *spath*: (s, [], Y # μ) $\rightsquigarrow*$ (p, [], γ)
and $**$: $\forall\, i.\ \exists\, t\ \Delta.$ (s, [], [Y]) $\rightsquigarrow(i)$ (t, [], Δ) **by** *blast*
from *elem* γ*_def* β*_def* **have** $***$:(p, [], γ) $\rightsquigarrow$ (r, [], β)
using *step*$_1$*_rule*[*of p* [] *X* γ' *r* [] β] **by** *simp*
have $****$: (s, [], Y # μ) $\rightsquigarrow*$ (r, [], β)
using *steps_trans*[*OF spath step*$_1$*_steps*[*OF* $***$]] **.**
from $*$[*unfolded v_def*] $**$ $****$ **show** *?thesis* **by** *blast*
qed
next
case *b*
then obtain a ζ **where** *u_def*: $u = a\#v$ **and** β*_def*: $\beta = \zeta$ @ γ' **and** *elem*: $(r, \zeta) \in \delta\ M\ p\ a\ X$ **by** *blast*
from *elem* **have** *eps_empty*: $\delta\varepsilon\ M\ p\ X = \{\}$

```
    using δ_nonempty[of p a X] by blast
  from step(3) consider (t) (St q, w, map Sym α) ⇝s* (St p, u, map Sym γ)
    | (f) ∃ r X β. (St q, w, map Sym α) ⇝s* (St r, u, Sym X # map Sym β) ∧
(r, [], X # β) ⇝* (p, [], γ) ∧ (∀ i. ∃ s Δ. (r, [], [X]) ⇝(i) (s, [], Δ)) by blast
  then have *: (St q, w, map Sym α) ⇝s* (St p, u, map Sym γ)
  proof cases
    case t
    then show ?thesis .
  next
    case f
    then obtain s Y μ where f1: (s, [], Y # μ) ⇝* (p, [], γ) and f2: ∀ i. ∃ s'
μ'. (s, [], [Y]) ⇝(i) (s', [], μ') by blast
    with γ_def obtain γ'' where (s, [], [Y]) ⇝* (p, [], X#γ'')
      using stack_cycle_drop[OF f2 f1] by blast
    then obtain n where f3: (s, [], [Y]) ⇝(n) (p, [], X#γ'')
      using stepn_steps[of s [] [Y] p [] X#γ''] by presburger
    from eps_empty have (p, [], X#γ'') ⇝| by simp
    with f2 f3 max_eps_steps have False by (meson Suc_n_not_le_n)
    then show ?thesis ..
  qed
  from eps_empty have (p, [], [X]) ⇝| by auto
  then have ¬eps_nonfinal p X ∧ ¬eps_final p X
    unfolding eps_nonfinal_def eps_final_def using step1_stepn_one by blast
  with elem have (St r, map Sym ζ) ∈ δ scan_dpda (St p) a (Sym X)
    by (auto simp: scan_dpda_def)
  with u_def γ_def β_def have **: (St p, u, map Sym γ) ⇝s (St r, v, map
Sym β)
    using scan.step1_rule[of St p u Sym X map Sym γ' St r v map Sym β] by
simp
  show ?thesis
    using scan.steps_trans[OF * scan.step1_steps[OF **]] by simp
 qed
qed
```

The language of the automaton *scan_dpda* and of the original automaton are the same, i.e. they are equivalent:

```
lemma lang_scan_dpda:
scan.accept_final = accept_final
proof
 show scan.accept_final ⊆ accept_final
 proof
  fix w
  assume w ∈ scan.accept_final
  then obtain q γ where q_final: q ∈ scan_dpda_final_states and scan_path:
(Q0', w, [X0]) ⇝s* (q, [], γ)
    unfolding scan.accept_final_def using scan_dpda_def by auto
  from q_final have cases: (∃ q' ∈ final_states M. q = St q') ∨ q = F (is ?a ∨
?b)
    unfolding scan_dpda_final_states_def by auto
```

then have $\exists\, p\ u\ \alpha.\ (Q0',\ w,\ [X0]) \rightsquigarrow_s (p,\ u,\ \alpha) \wedge (p,\ u,\ \alpha) \rightsquigarrow_s* (q,\ [],\ \gamma)$
using *scan.steps_not_refl_split_first*[*OF scan_path*] **by** *blast*
then have *p*: $(St\ (init\_state\ M),\ w,\ [Sym\ (init\_symbol\ M),\ X0]) \rightsquigarrow_s* (q,\ [],\ \gamma)$
using *scan_dpda_first_step* **by** *blast*
from *cases* **consider** (*a*) *?a* | (*b*) *?b* **by** *blast*
then show $w \in accept\_final$
proof *cases*
case *a*
then obtain q' **where** *q_def*: $q = St\ q'$ **and** q'*_final*: $q' \in final\_states\ M$ **by** *blast*
from *p*[*simplified q_def*] **obtain** γ' **where** γ*_def*: $\gamma = stack\_with\_X0\ \gamma'$
using *scan_dpda_stack_with_X0*[*of init_state M w* [*init_symbol M*] q' [] γ] **by** *auto*
from *p*[*simplified q_def* γ*_def*] **have** $(init\_state\ M,\ w,\ [init\_symbol\ M]) \rightsquigarrow* (q',\ [],\ \gamma')$
using *scan_dpda_stepsX0*[*of init_state M w* [*init_symbol M*] q' [] γ'] **by** *simp*
with q'*_final* **show** *?thesis*
unfolding *accept_final_def* **by** *blast*
next
case *b*
obtain $p\ u\ \alpha$ **where** *fss*: $(St\ (init\_state\ M),\ w,\ [Sym\ (init\_symbol\ M),\ X0]) \rightsquigarrow_s* (p,\ u,\ \alpha)$
and *ls*: $(p,\ u,\ \alpha) \rightsquigarrow_s (F,\ [],\ \gamma)$
using *scan.steps_not_refl_split_last*[*OF p*[*simplified b*]] **by** *blast*
obtain $X\ \alpha'$ **where** α*_def*: $\alpha = X\#\alpha'$
using *scan.step*$_1$*_nonempty_stack*[*OF ls*] **by** *blast*
obtain $p'\ X'$ **where** *u_def*: $u = []$ **and** *p_def*: $p = St\ p'$ **and** *X_def*: $X = Sym\ X'$ **and** *epath*: *eps_final* $p'\ X'$
using *scan_dpda_step_to_F*[*OF ls*[*simplified* α*_def*]] **by** *blast*
from *fss*[*simplified p_def*] **obtain** α'' **where** α*_with_X0*: $\alpha = stack\_with\_X0\ \alpha''$
using *scan_dpda_stack_with_X0*[*of init_state M w* [*init_symbol M*] p' *u* α] **by** *auto*
from *fss*[*simplified p_def u_def* α*_with_X0*] **have** $*$: $(init\_state\ M,\ w,\ [init\_symbol\ M]) \rightsquigarrow* (p',\ [],\ \alpha'')$
using *scan_dpda_stepsX0*[*of init_state M w* [*init_symbol M*] p' [] α''] **by** *simp*
from α*_def* α*_with_X0 X_def* **obtain** α''' **where** α''*_def*: $\alpha'' = X'\#\alpha'''$
by (*metis Nil_is_map_conv append_Cons append_Nil hd_Cons_tl list.map_sel*(*1*) *list.sel*(*1*) *sym_scan.distinct*(*1*) *sym_scan.inject*)
from *epath*[*unfolded eps_final_def*] **obtain** *i r* β **where** *path*: $(p',\ [],\ [X'])\ \rightsquigarrow(i)\ (r,\ [],\ \beta)$ **and** *r_final*: $r \in final\_states\ M$ **by** *blast*
from *path* **have** $(p',\ [],\ [X']) \rightsquigarrow* (r,\ [],\ \beta)$
using *stepn_steps*[*of* p' [] $[X']$ r [] β] **by** *auto*
with α''*_def* **have** $**$: $(p',\ [],\ \alpha'') \rightsquigarrow* (r,\ [],\ \beta@\alpha''')$
using *steps_stack_app*[*of* p' [] $[X']$ r [] β α'''] **by** *simp*
from *r_final* **show** *?thesis*

```
      unfolding accept_final_def using steps_trans[OF * **] by blast
    qed
  qed
next
  show accept_final ⊆ scan.accept_final
  proof
    fix w
    assume w ∈ accept_final
    then obtain q γ where q_final: q ∈ final_states M and p: (init_state M, w,
[init_symbol M]) ⇝* (q, [], γ)
      unfolding accept_final_def by blast
    have fs: (Q0', w, [X0]) ⇝s (St (init_state M), w, [Sym (init_symbol M), X0])
      using scan_dpda_def scan.step1_rule by auto
    from scan_dpda_steps[OF p] consider (a) (St (init_state M), w, [Sym (init_symbol
M)]) ⇝s* (St q, [], map Sym γ)
      | (b) ∃ r X β. (St (init_state M), w, [Sym (init_symbol M)]) ⇝s* (St r, [],
Sym X # map Sym β) ∧
                                (r, [], X # β) ⇝* (q, [], γ) ∧ (∀ i. ∃ s Δ. (r, [], [X]) ⇝(i)
(s, [], Δ)) by auto
    then show w ∈ scan.accept_final
    proof cases
      case a
        have (St (init_state M), w, [Sym (init_symbol M), X0]) ⇝s* (St q, [],
stack_with_X0 γ)
        using scan.steps_stack_app[OF a] by simp
      then have (Q0', w, [X0]) ⇝s* (St q, [], stack_with_X0 γ)
       using scan.step1_steps[OF fs] scan.steps_trans[of Q0' w [X0] St (init_state
M) w [Sym (init_symbol M), X0] St q [] stack_with_X0 γ] by simp
      with q_final show ?thesis
      unfolding scan.accept_final_def using scan_dpda_def scan_dpda_final_states_def
by auto
    next
      case b
      then obtain r X β where pscan: (St (init_state M), w, [Sym (init_symbol
M)]) ⇝s* (St r, [], Sym X # map Sym β) and
        pr: (r, [], X # β) ⇝* (q, [], γ) and cycle: ∀ i. ∃ s Δ. (r, [], [X]) ⇝(i) (s, [],
Δ) by blast
       have *: (St (init_state M), w, [Sym (init_symbol M), X0]) ⇝s* (St r, [],
stack_with_X0 (X # β))
        using scan.steps_stack_app[OF pscan] by simp
      have r_final: ∃γ'. (r, [], [X]) ⇝* (q, [], γ')
        using stack_cycle_drop[OF cycle pr] by auto
      from r_final q_final have e1: ¬eps_nonfinal r X
      unfolding eps_nonfinal_def using stepn_steps[of r [] [X] q []] dpda_stepn_det[of
_ r [] [X] q []] by blast
      from cycle r_final q_final have e2: eps_final r X
        unfolding eps_final_def using stepn_steps[of r [] [X] q []] by blast
      from e1 e2 have (St r, [], Sym X # map Sym β) ⇝s (F, [], Sym X # map
Sym β)
```

using *scan_dpda_def scan.step₁_rule*[*of St r* [] *Sym X map Sym β F* [] *Sym X # map Sym β*] **by** *simp*
then have ∗∗: (*St r*, [], *stack_with_X0* (*X#β*)) ⇝$_s$ (*F*, [], *stack_with_X0* (*X#β*))
using *scan.steps_stack_app*[*of St r* [] *Sym X # map Sym β F* [] *Sym X # map Sym β*] **by** *simp*
from *fs* ∗ ∗∗ **have** (*Q0'*, *w*, [*X0*]) ⇝$_s$∗ (*F*, [], *stack_with_X0* (*X # β*))
using *scan.step₁_steps scan.steps_trans* **by** *metis*
then show *?thesis*
unfolding *scan.accept_final_def* **using** *scan_dpda_def scan_dpda_final_states_def* **by** *auto*
qed
qed
qed

2.2.4 Scan Property

lemma *D_consumes*: (*D*, *w*, *X#α*) ⇝$_s$∗ (*D*, [], *X#α*)
proof (*induction w*)
case *Nil*
then show *?case*
by (*simp add*: *scan.steps_refl*)
next
case (*Cons a w*)
have (*D*, [*X*]) ∈ *scan_dpda_delta D a X* **by** *simp*
then have ∗: (*D*, *a#w*, *X#α*) ⇝$_s$ (*D*, *w*, *X#α*)
using *scan.step₁_rule*[*of D a#w X α D w X#α*] **by** (*simp add*: *scan_dpda_def*)
show *?case*
using *scan.steps_trans*[*OF scan.step₁_steps*[*OF* ∗] *Cons*] **.**
qed

definition *eps_inf* :: *'q* ⇒ *'s* ⇒ *bool* **where**
eps_inf q X ≡ ∀ *i*. ∃ *p α*. (*q*, [], [*X*]) ⇝(*i*) (*p*, [], *α*)

definition *eps_infs* :: *'q* ⇒ *'s list* ⇒ *bool* **where**
eps_infs q α ≡ ∀ *i*. ∃ *p γ*. (*q*, [], *α*) ⇝(*i*) (*p*, [], *γ*)

If a pair of a state and a stack symbol does not allow infinite epsilon steps, then the stack symbol will be consumed:

lemma *inf_consumes_sym*:
assumes *eps_infs q* (*X#α*)
and ¬*eps_inf q X*
shows ∃ *p*. (*q*, [], *X#α*) ⇝∗ (*p*, [], *α*)
proof (*rule ccontr*)
assume *asm*: ∄ *p*. (*q*, [], *X#α*) ⇝∗ (*p*, [], *α*)
have ∃ *p γ*. (*q*, [], *X#α*) ⇝(*i*) (*p*, [], *γ*@*α*) ∧ *γ* ≠ [] ∧ (*q*, [], [*X*]) ⇝(*i*) (*p*, [], *γ*)
for *i*
proof (*induct i*)
case *0*

have $(q, [], X \# \alpha) \rightsquigarrow(0) (q, [], [X] @ \alpha) \wedge [X] \neq [] \wedge (q, [], [X]) \rightsquigarrow(0) (q, [], [X])$ **by** *simp*
then show *?case* **by** *metis*
next
case (*Suc i*)
then obtain $p\ \gamma$ **where** p: $(q, [], X \# \alpha) \rightsquigarrow(i) (p, [], \gamma @ \alpha)$ **and** γ_def: $\gamma \neq []$ **and** *p1*: $(q, [], [X]) \rightsquigarrow(i) (p, [], \gamma)$ **by** *blast*
from *assms(1)*[*unfolded eps_infs_def*] **obtain** $r\ \beta$ **where** $*$: $(q, [], X\#\alpha) \rightsquigarrow(Suc\ i) (r, [], \beta)$ **by** *blast*
then have *st*: $(p, [], \gamma@\alpha) \rightsquigarrow (r, [], \beta)$
using *stepn_split_last*[*of i q* [] $X\#\alpha\ r$ [] β] *dpda_stepn_det*[*OF p*] **by** *auto*
from γ_def **obtain** $Y\ \gamma'$ **where** γ_def2: $\gamma = Y\#\gamma'$
using *list.exhaust* **by** *blast*
from *st*[*unfolded* γ_def2] **obtain** γ'' **where** β_def: $\beta = \gamma'' @ \gamma' @ \alpha$
using $step_1\_rule$[*of p* [] $Y\ \gamma'@\alpha\ r$ [] β] **by** *auto*
from $*$[*unfolded* β_def] *asm* **have** $**$: $\gamma'' @ \gamma' \neq []$
using *stepn_steps*[*of q* [] $X\#\alpha\ r$ [] $\gamma'' @ \gamma' @ \alpha$] **by** *force*
from *st*[*unfolded* β_def] γ_def **have** *st1*: $(p, [], \gamma) \rightsquigarrow (r, [], \gamma'' @ \gamma')$
using $step_1\_stack\_drop$[*of p* [] $\gamma\ \alpha\ r$ [] $\gamma'' @ \gamma'$] **by** *simp*
from *p1 st1* **have** $***$: $(q, [], [X]) \rightsquigarrow(Suc\ i) (r, [], \gamma'' @ \gamma')$ **by** *simp*
from $*$[*unfolded* β_def] $** ***$ **show** *?case*
by (*metis append.assoc*)
qed
then have *eps_inf q X*
by (*metis eps_inf_def*)
with *assms(2)* **show** *False* **by** *satx*
qed

lemma *some_pair_inf*:
assumes *eps_infs q* α
and $\bigwedge p\ X\ \gamma.\ (q, [], \alpha) \rightsquigarrow* (p, [], X\#\gamma) \longrightarrow \neg eps\_inf\ p\ X$
shows *False*
using *assms* **proof** (*induction* α *arbitrary*: q)
case *Nil*
then show *?case*
by (*force simp*: *eps_infs_def*)
next
case (*Cons X* α)
from *Cons(3)*[*of q X* α] **have** e: $\neg eps\_inf\ q\ X$
by (*simp add*: *steps_refl*)
obtain p **where** p: $(q, [], X\#\alpha) \rightsquigarrow* (p, [], \alpha)$
using *inf_consumes_sym*[*OF Cons(2) e*] **by** *blast*
then obtain i **where** *p1*: $(q, [], X\#\alpha) \rightsquigarrow(i) (p, [], \alpha)$
using *stepn_steps*[*of q* [] $X\#\alpha\ p$ [] α] **by** *blast*
have $*$: *eps_infs p* α **unfolding** *eps_infs_def* **proof**
fix j
from *Cons(2)*[*unfolded eps_infs_def*] **obtain** $r\ \beta$ **where** *p2*: $(q, [], X\#\alpha) \rightsquigarrow(i + j) (r, [], \beta)$ **by** *blast*
have $(p, [], \alpha) \rightsquigarrow(j) (r, [], \beta)$

```
    using split_path[OF p2, of i] dpda_stepn_det[OF p1] by auto
  then show ∃ r β. (p, [], α) ↝(j) (r, [], β) by blast
 qed
 from Cons(3) have **: ⋀r Y γ. (p, [], α) ↝* (r, [], Y # γ) ⟶ ¬ eps_inf r Y
  using steps_trans[OF p] by blast
 from Cons(1)[OF * **] show ?case .
qed
```

If a configuration allows infinite epsilon steps, it will eventually reach a pair of a state and a stack symbol that allows infinite epsilon steps:

```
lemma inf_reaches:
 assumes eps_infs q α
 shows ∃ p X γ. (q, [], α) ↝* (p, [], X#γ) ∧ eps_inf p X
using assms some_pair_inf by blast

lemma eps_inf_to_D:
 assumes eps_inf q X
 shows (St q, w, Sym X # α) ↝s* (D, [], Sym X # α)
proof −
 from assms consider (a) eps_nonfinal q X | (b) ¬eps_nonfinal q X ∧ eps_final
q X
   unfolding eps_inf_def eps_nonfinal_def eps_final_def by blast
 then show ?thesis proof cases
   case a
   then have (D, [Sym X]) ∈ scan_dpda_delta_eps (St q) (Sym X) by simp
   then have st: (St q, w, Sym X # α) ↝s (D, w, Sym X # α)
     using scan.step1_rule[of St q w Sym X α D w Sym X # α] by (simp add:
scan_dpda_def)
   show ?thesis
     using scan.steps_trans[OF scan.step1_steps[OF st] D_consumes] .
 next
   case b
   then have (F, [Sym X]) ∈ scan_dpda_delta_eps (St q) (Sym X) by simp
   then have st1: (St q, w, Sym X # α) ↝s (F, w, Sym X # α)
     using scan.step1_rule[of St q w Sym X α F w Sym X # α] by (simp add:
scan_dpda_def)
   have (D, [Sym X]) ∈ scan_dpda_delta_eps F (Sym X) by simp
   then have st2: (F, w, Sym X # α) ↝s (D, w, Sym X # α)
     using scan.step1_rule[of F w  Sym X α D w Sym X # α] by (simp add:
scan_dpda_def)
   show ?thesis
     using scan.steps_trans[OF scan.step1_steps[OF st1] scan.steps_trans[OF
scan.step1_steps[OF st2] D_consumes]] .
 qed
qed

lemma scan_dpda_eps_inf:
 assumes (q, w, α) ↝* (p, w, X#γ)
    and eps_inf p X
```

```
    shows ∃ Y β. (St q, w, map Sym α) ⇝s* (D, [], Sym Y # β)
proof −
  consider (a) (St q, w, map Sym α) ⇝s* (St p, w, Sym X # map Sym γ) |
           (b) ∃ r Y β. (St q, w, map Sym α) ⇝s* (St r, w, Sym Y # map Sym β)
∧ (∀ i. ∃ s ζ. (r, [], [Y]) ⇝(i) (s, [], ζ))
    using scan_dpda_steps[OF assms(1)] by auto
  then show ?thesis proof cases
    case a
    have ∗: (St p, w, Sym X # map Sym γ) ⇝s* (D, [], Sym X # map Sym γ)
      using eps_inf_to_D[OF assms(2)] by simp
    show ?thesis
      using scan.steps_trans[OF a ∗] by blast
  next
    case b
    then obtain r Y β where p: (St q, w, map Sym α) ⇝s* (St r, w, Sym Y #
map Sym β) and r_inf: ∀ i. ∃ s ζ. (r, [], [Y]) ⇝(i) (s, [], ζ) by blast
    from r_inf have r_eps_inf: eps_inf r Y
      by (simp add: eps_inf_def)
    have ∗: (St r, w, Sym Y # map Sym β) ⇝s* (D, [], Sym Y # map Sym β)
      using eps_inf_to_D[OF r_eps_inf] by simp
    show ?thesis
      using scan.steps_trans[OF p ∗] by blast
  qed
qed

lemma scan_dpda_neps_infs:
  assumes ¬eps_infs q α
  shows ∃ p γ. (St q, w, map Sym α) ⇝s* (St p, w, map Sym γ) ∧ (∀ X γ'. γ =
X#γ' ⟶ δε M p X = {})
proof −
  from assms[unfolded eps_infs_def] obtain i where ∀ p γ. ¬(q, [], α) ⇝(i) (p,
[], γ) by blast
  then have ∃ j p γ. j < i ∧ (q, [], α) ⇝(j) (p, [], γ) ∧ (p, [], γ) ⇝| proof
(induction i)
    case 0
    then show ?case
      by (auto simp: steps_refl)
  next
    case (Suc i)
    consider (a) ∀ p γ. ¬ (q, [], α) ⇝(i) (p, [], γ) | (b) ∃ p γ. (q, [], α) ⇝(i) (p, [],
γ) by blast
    then show ?case proof cases
      case a
      from Suc(1)[OF a] show ?thesis
        by (auto simp: less_Suc_eq)
    next
      case b
      then obtain p γ where ∗: (q, [], α) ⇝(i) (p, [], γ) by blast
      with Suc(2) have ∗∗: (p, [], γ) ⇝|
```

```
      using decreasing_word step1_steps by fastforce
    from * ** show ?thesis by blast
  qed
 qed
 then obtain j p γ where j < i and p: (q, [], α) ⇝(j) (p, [], γ) and nst: (p, [], γ) ⇝| by blast
 from p have p1: (q, [], α) ⇝* (p, [], γ)
  using stepn_steps[of q [] α p [] γ] by auto
 consider (a) (St q, [], map Sym α) ⇝s* (St p, [], map Sym γ) |
          (b) ∃ r X β. (St q, [], map Sym α) ⇝s* (St r, [], Sym X # map Sym β)
∧ (∀ i. ∃ s Δ. (r, [], [X]) ⇝(i) (s, [], Δ))
  using scan_dpda_steps[OF p1] by blast
 then show ?thesis proof cases
  case a
  then have *: (St q, w, map Sym α) ⇝s* (St p, w, map Sym γ)
    using scan.steps_word_app[of St q [] map Sym α St p [] map Sym γ w] by
simp
  from nst have **: ∀ X γ'. γ = X # γ' ⟶ δε M p X = {} by auto
  from * ** show ?thesis by blast
 next
  case b
  then obtain r X β where p2: (St q, [], map Sym α) ⇝s* (St r, [], Sym X #
map Sym β) and cycle: ∀ i. ∃ s Δ. (r, [], [X]) ⇝(i) (s, [], Δ) by blast
  from p2 have p3: (St q, [], stack_with_X0 α) ⇝s* (St r, [], stack_with_X0
(X#β))
    using scan.steps_stack_app[where ?β = [X0]] by fastforce
  have (q, [], α) ⇝* (r, [], X#β)
    using scan_dpda_stepsX0[OF p3] .
  then obtain n where p4: (q, [], α) ⇝(n) (r, [], X#β)
    using stepn_steps[of q [] α r [] X#β] by blast
  have ∀ k. ∃ s Δ. (q, [], α) ⇝(k) (s, [], Δ) proof
    fix k
    show ∃ s Δ. (q, [], α) ⇝(k) (s, [], Δ) proof (cases k < n)
      case True
      then have k_leq: k ≤ n by simp
      then obtain s u ζ where *: (q, [], α) ⇝(k) (s, u, ζ)
        using split_path[OF p4 k_leq] by blast
      from * have u_def: u = []
       using stepn_steps[of q [] α s u ζ] decreasing_word[of q [] α s u ζ] by auto
      from *[unfolded u_def] show ?thesis by blast
    next
      case False
      then have k_beq: k ≥ n by simp
      with cycle obtain s ζ where (r, [], [X]) ⇝(k−n) (s, [], ζ) by presburger
      then have p5: (r, [], X#β) ⇝(k−n) (s, [], ζ@β)
        using stepn_stack_app[where ?β = β] by fastforce
      from k_beq show ?thesis
        using stepn_trans[OF p4 p5] by auto
    qed
```

```
    qed
    with assms show ?thesis
      by (simp add: eps_infs_def)
  qed
qed

lemma scan_dpda_scans_St:
∃ p X γ. (St q, w, stack_with_X0 α) ⇝s* (p, [], X#γ) ∧ δ scan_dpda p a X ≠
{}
proof (induction w arbitrary: q α)
  case Nil
  then show ?case proof (cases eps_infs q α)
    case True
    obtain p X γ where p: (q, [], α) ⇝* (p, [], X#γ) and eps_p: eps_inf p X
      using inf_reaches[OF True] by blast
    obtain Y β where (St q, [], map Sym α) ⇝s* (D, [], Sym Y # β)
      using scan_dpda_eps_inf[OF p eps_p] by blast
    then have *: (St q, [], stack_with_X0 α) ⇝s* (D, [], Sym Y # β @ [X0])
      using scan.steps_stack_app[where ?β = [X0]] by fastforce
    have **: δ scan_dpda D a (Sym Y) ≠ {}
      by (simp add: scan_dpda_def)
    from * ** show ?thesis by blast
  next
    case False
    obtain p γ where p: (St q, [], map Sym α) ⇝s* (St p, [], map Sym γ) and
γ_nempty: (∀ X γ'. γ = X#γ' ⟶ δε M p X = {})
      using scan_dpda_neps_infs[OF False] by blast
    from p have p1: (St q, [], stack_with_X0 α) ⇝s* (St p, [], stack_with_X0 γ)
      using scan.steps_stack_app[where ?β = [X0]] by simp
    show ?thesis proof (cases γ)
      case Nil
      with p1 show ?thesis
        by (force simp: scan_dpda_def)
    next
      case (Cons X γ')
      with γ_nempty have δε M p X = {} by simp
      then have δ scan_dpda (St p) a (Sym X) ≠ {}
        by (simp add: scan_dpda_def)
      with p1 Cons show ?thesis by auto
    qed
  qed
next
  case IH: (Cons b w)
  show ?case proof (cases eps_infs q α)
    case True
    obtain p X γ where p: (q, [], α) ⇝* (p, [], X#γ) and eps_p: eps_inf p X
      using inf_reaches[OF True] by blast
    from p have p1: (q, b#w, α) ⇝* (p, b#w, X#γ)
      using steps_word_app[of q [] α p [] X#γ b#w] by simp
```

obtain $Y\ \beta$ **where** $(St\ q,\ b\ \#\ w,\ map\ Sym\ \alpha) \rightsquigarrow_s* (D,\ [],\ Sym\ Y\ \#\ \beta)$
using *scan_dpda_eps_inf*[*OF p1 eps_p*] **by** *blast*
then have $*$: $(St\ q,\ b\ \#\ w,\ stack\_with\_X0\ \alpha) \rightsquigarrow_s* (D,\ [],\ Sym\ Y\ \#\ \beta\ @\ [X0])$
using *scan.steps_stack_app*[**where** $?\beta = [X0]$] **by** *fastforce*
have $**$: $\delta\ scan\_dpda\ D\ a\ (Sym\ Y) \neq \{\}$
by (*simp add*: *scan_dpda_def*)
from $*$ $**$ **show** *?thesis* **by** *blast*
next
case *False*
obtain $p\ \gamma$ **where** p: $(St\ q,\ b\#w,\ map\ Sym\ \alpha) \rightsquigarrow_s* (St\ p,\ b\#w,\ map\ Sym\ \gamma)$ **and** γ_nempty: $(\forall X\ \gamma'.\ \gamma = X\#\gamma' \longrightarrow \delta\varepsilon\ M\ p\ X = \{\})$
using *scan_dpda_neps_infs*[*OF False*] **by** *blast*
from p **have** $p1$: $(St\ q,\ b\#w,\ stack\_with\_X0\ \alpha) \rightsquigarrow_s* (St\ p,\ b\#w,\ stack\_with\_X0\ \gamma)$
using *scan.steps_stack_app*[**where** $?\beta = [X0]$] **by** *simp*
show *?thesis* **proof** (*cases* γ)
case *Nil*
have $(D,\ [X0]) \in scan\_dpda\_delta\ (St\ p)\ b\ X0$ **by** *simp*
with *Nil* **have** st: $(St\ p,\ b\#w,\ stack\_with\_X0\ \gamma) \rightsquigarrow_s (D,\ w,\ [X0])$
using $scan.step_1\_rule$[*of St p b#w X0 [] D w [X0]*] **by** (*simp add*: *scan_dpda_def*)
have $*$: $(St\ q,\ b\#w,\ stack\_with\_X0\ \alpha) \rightsquigarrow_s* (D,\ [],\ [X0])$
using *scan.steps_trans*[*OF p1 scan.steps_trans*[*OF* $scan.step_1\_steps$[*OF st*] *D_consumes*]] **.**
have $**$: $\delta\ scan\_dpda\ D\ a\ X0 \neq \{\}$
by (*simp add*: *scan_dpda_def*)
from $*$ $**$ **show** *?thesis* **by** *blast*
next
case $(Cons\ X\ \gamma')$
with γ_nempty **have** $\delta\varepsilon\ M\ p\ X = \{\}$ **by** *simp*
then consider (a) $(D,\ [Sym\ X]) \in scan\_dpda\_delta\ (St\ p)\ b\ (Sym\ X)$ | (b) $\exists r\ \beta.\ (St\ r,\ map\ Sym\ \beta) \in scan\_dpda\_delta\ (St\ p)\ b\ (Sym\ X)$
by (*cases* $\delta\ M\ p\ b\ X = \{\}$) *fastforce+*
then show *?thesis* **proof** *cases*
case *a*
with *Cons* **have** st: $(St\ p,\ b\#w,\ stack\_with\_X0\ \gamma) \rightsquigarrow_s (D,\ w,\ stack\_with\_X0\ \gamma)$
using $scan.step_1\_rule$[*of St p b#w Sym X stack_with_X0* γ' *D w stack_with_X0* γ] **by** (*simp add*: *scan_dpda_def*)
from *Cons* **have** $*$: $(St\ q,\ b\#w,\ stack\_with\_X0\ \alpha) \rightsquigarrow_s* (D,\ [],\ stack\_with\_X0\ \gamma)$
using *scan.steps_trans*[*OF p1 scan.steps_trans*[*OF* $scan.step_1\_steps$[*OF st*]]] *D_consumes*[*of w Sym X stack_with_X0* γ'] **by** *simp*
have $**$: $\delta\ scan\_dpda\ D\ a\ (Sym\ X) \neq \{\}$
by (*simp add*: *scan_dpda_def*)
from $*$ $**$ *Cons* **show** *?thesis* **by** *auto*
next
case *b*
then obtain $r\ \beta$ **where** $(St\ r,\ map\ Sym\ \beta) \in scan\_dpda\_delta\ (St\ p)\ b$

```
(Sym X) by blast
    with Cons have st: (St p, b#w, stack_with_X0 γ) ⇝s (St r, w, stack_with_X0
(β@γ'))
          using scan.step1_rule[of St p b#w Sym X stack_with_X0 γ' St r w
stack_with_X0 (β@γ')] by (simp add: scan_dpda_def)
       from IH have *: ∃p X γ. (St r, w, stack_with_X0 (β@γ')) ⇝s* (p, [], X
# γ) ∧ δ scan_dpda p a X ≠ {} by presburger
       from * show ?thesis
         using scan.steps_trans[OF scan.steps_trans[OF p1 scan.step1_steps[OF
st]]] by blast
     qed
   qed
 qed
qed
```

For every input word, the automaton *scan_dpda* scans the entire input and, moreover, ends in a configuration where no epsilon step is possible:

```
lemma scan_dpda_scans:
∃q X α. (init_state scan_dpda, w, [init_symbol scan_dpda]) ⇝s* (q, [], X#α) ∧
δ scan_dpda q a X ≠ {}
proof −
 have (St (init_state M), stack_with_X0 [init_symbol M]) ∈ scan_dpda_delta_eps
Q0' X0 by simp
 then have *: (init_state scan_dpda, w, [init_symbol scan_dpda]) ⇝s (St (init_state
M), w, stack_with_X0 [init_symbol M])
    using scan.step1_rule[of Q0' w X0 [] St (init_state M) w stack_with_X0
[init_symbol M]] by (simp add: scan_dpda_def)
  obtain p X γ where **: (St (init_state M), w, stack_with_X0 [init_symbol
M]) ⇝s* (p, [], X#γ) and d: δ scan_dpda p a X ≠ {}
    using scan_dpda_scans_St[of init_state M w [init_symbol M]] by blast
  have (init_state scan_dpda, w, [init_symbol scan_dpda]) ⇝s* (p, [], X#γ)
    using scan.steps_trans[OF scan.step1_steps[OF *] **] .
  with d show ?thesis by blast
qed

end
```

2.3 Complement Construction

After ensuring that the automaton scans its entire input word, we are left with constructing the deterministic pushdown automaton that recognizes the complement language. The main idea is to keep track of whether a final state has been visited since the last true step using a second component for states. If no final state has been visited, the complement automaton enters a final state of its own just before the next true step.

2.3.1 Definition

An S1-state indicates that a final state has been visited since the last true step, whereas an S2-state indicates that no final state has been visited since the last true step. S3-states are the final states of the complement automaton:

datatype *'q s123 = S1 'q | S2 'q | S3 'q*

instance *s123* :: (*finite*) *finite*
proof
  have ∗: *UNIV* = {*t*. ∃ *q*. *t* = *S1 q*} ∪ {*t*. ∃ *q*. *t* = *S2 q*} ∪ {*t*. ∃ *q*. *t* = *S3 q*}
    by *auto* (*metis s123.exhaust*)
  show *finite* (*UNIV* :: *'a s123 set*)
    by (*simp add*: ∗ *full_SetCompr_eq*)
qed

lemma *inj_S1*: *inj S1*
  by (*simp add*: *inj_def*)

lemma *inj_S2*: *inj S2*
  by (*simp add*: *inj_def*)

We can now assume the scan property proved in the last subsection:

locale *complement_dpda* = *dpda M* **for** *M* :: (*'q* :: *finite*, *'a* :: *finite*, *'s* :: *finite*) *pda* +
  assumes *M_path*: ∃ *q X α*. (*init_state M*, *w*, [*init_symbol M*]) ⇝∗ (*q*, [], *X*#*α*) ∧ *δ M q a X* ≠ {}
begin

definition *comp_dpda_init_state* :: *'q s123* **where**
  comp_dpda_init_state ≡ *if init_state M* ∈ *final_states M then S1* (*init_state M*) *else S2* (*init_state M*)

definition *comp_dpda_final_states* :: *'q s123 set* **where**
  comp_dpda_final_states ≡ *range S3*

fun *comp_dpda_delta* :: *'q s123* ⇒ *'a* ⇒ *'s* ⇒ (*'q s123* × *'s list*) *set* **where**
  comp_dpda_delta (*S1 q*) *a X* = (λ(*p*, *α*). *if p* ∈ *final_states M then* (*S1 p*, *α*) *else* (*S2 p*, *α*)) ‘ *δ M q a X*
| *comp_dpda_delta* (*S3 q*) *a X* = (λ(*p*, *α*). *if p* ∈ *final_states M then* (*S1 p*, *α*) *else* (*S2 p*, *α*)) ‘ *δ M q a X*
| *comp_dpda_delta* _ _ _ = {}

fun *comp_dpda_delta_eps* :: *'q s123* ⇒ *'s* ⇒ (*'q s123* × *'s list*) *set* **where**
  comp_dpda_delta_eps (*S1 q*) *X* = (λ(*p*, *α*). (*S1 p*, *α*)) ‘ *δε M q X*
| *comp_dpda_delta_eps* (*S2 q*) *X* = (λ(*p*, *α*). *if p* ∈ *final_states M then* (*S1 p*, *α*) *else* (*S2 p*, *α*)) ‘ *δε M q X*
    ∪ (*if* ∃ *a*. *δ M q a X* ≠ {} *then* {(*S3 q*, [*X*])} *else* {})
| *comp_dpda_delta_eps* _ _ = {}

definition *comp_dpda* :: (*'q s123*, *'a*, *'s*) *pda* **where**
comp_dpda =
(| *init_state* = *comp_dpda_init_state*,
init_symbol = *init_symbol M*,
final_states = *comp_dpda_final_states*,
delta = *comp_dpda_delta*, *delta_eps* = *comp_dpda_delta_eps* |)

2.3.2 Determinism

lemma *image_singleton_if_inj*:
assumes *inj f*
shows (∃ *x*. *A* = {*x*}) ⟷ (∃ *x*. *f* ‘ *A* = {*x*})
using *assms* **by** (*metis image_empty image_insert image_inv_f_f*)

The automaton *comp_dpda* is deterministic:

lemma *dpda_comp_dpda*: *dpda comp_dpda*
proof (*standard*, *goal_cases*)
case (*1 p a Z*)
have *finite* (*comp_dpda_delta p a Z*)
by (*induction p a Z rule*: *comp_dpda_delta.induct*) (*auto simp*: *finite_delta*)
then show *?case*
by (*simp add*: *comp_dpda_def*)
next
case (*2 p Z*)
have *finite* (*comp_dpda_delta_eps p Z*)
by (*induction p Z rule*: *comp_dpda_delta_eps.induct*) (*auto simp*: *finite_delta_eps*)
then show *?case*
by (*simp add*: *comp_dpda_def*)
next
case (*3 q a X*)
then show *?case*
proof
assume *δ comp_dpda q a X* ≠ {}
then have *comp_dpda_delta q a X* ≠ {}
by (*simp add*: *comp_dpda_def*)
then have *comp_dpda_delta_eps q X* = {}
by (*induction q a X rule*: *comp_dpda_delta.induct*) (*auto simp*: *δ_nonempty*)
then show *δε comp_dpda q X* = {}
by (*simp add*: *comp_dpda_def*)
qed
next
case (*4 q a X*)
let *?f* = (λ(*p*, *α*). *if p* ∈ *final_states M then* (*S1 p*, *α*) *else* (*S2 p*, *α*))
have ∗: *inj ?f*
by (*simp add*: *inj_def*)
have *comp_dpda_delta q a X* = {} ∨ (∃ *p γ*. *comp_dpda_delta q a X* = {(*p*, *γ*)})
proof (*induction q a X rule*: *comp_dpda_delta.induct*)
case (*1 q a X*)

```
    then show ?case
     using δ_singleton[of q a X] image_singleton_if_inj[OF ∗, of δ M q a X] by
simp
  next
    case (2 q a X)
    then show ?case
     using δ_singleton[of q a X] image_singleton_if_inj[OF ∗, of δ M q a X] by
simp
  qed simp
  then show ?case
    by (simp add: comp_dpda_def)
next
  case (5 q X)
  have comp_dpda_delta_eps q X = {} ∨ (∃ p γ. comp_dpda_delta_eps q X =
{(p, γ)})
  proof (induction q X rule: comp_dpda_delta_eps.induct)
    case (1 q X)
    then show ?case
     using δε_singleton[of q X] by auto
  next
    case (2 q X)
    consider (a) ∃ a. δ M q a X ≠ {} | (b) ¬(∃ a. δ M q a X ≠ {}) by blast
    then show ?case
    proof cases
      case a
      then have δε M q X = {}
       using δ_nonempty[of q _ X] by blast
      then show ?thesis by simp
    next
      case b
      let ?f = (λ(p, α). if p ∈ final_states M then (S1 p, α) else (S2 p, α))
      have ∗: inj ?f
       by (simp add: inj_def)
      from b show ?thesis
       using δε_singleton[of q X] image_singleton_if_inj[OF ∗, of δε M q X] by
simp
    qed
  qed simp
  then show ?case
    by (simp add: comp_dpda_def)
qed
```

2.3.3 Complementation

```
sublocale comp: dpda comp_dpda
  using dpda_comp_dpda .
```

We abbreviate the definitions of *comp_dpda* with index c:

```
notation comp.step1 (infix ⇝c 55)
notation comp.steps (infix ⇝c∗ 55)
```

notation *comp.stepsn* ((_ /$\rightsquigarrow_c$'(_')/ _) [*55*, *0*, *55*] *55*)
notation *comp.steps1* (**infix** $\rightsquigarrow_c+$ *55*)

lemma *comp_dpda_step_from_S1*:
  assumes (*S1 q*, [], α) $\rightsquigarrow_c$ (*p*, [], γ)
  shows $\exists\, p'.\ p = S1\ p'$
using *assms comp.step*$_1$*_rule_ext*[*of S1 q* [] α *p* [] γ] **by** (*auto simp*: *comp_dpda_def*)

lemma *comp_dpda_steps_from_S1*:
  assumes (*S1 q*, [], α) $\rightsquigarrow_c*$ (*p*, [], γ)
  shows $\exists\, p'.\ p = S1\ p'$
using *assms comp_dpda_step_from_S1 comp.decreasing_word*
  by (*induction* (*S1 q*, [] :: *'a list*, α) (*p*, [] :: *'a list*, γ) *arbitrary*: *p* γ *rule*: *comp.steps_induct2_bw*) *fastforce+*

lemma *comp_dpda_nonfinal_stepsS2*:
  assumes (*q*, [], α) $\rightsquigarrow*$ (*p*, [], γ)
    and $\bigwedge r\ \beta.$ (*q*, [], α) $\rightsquigarrow*$ (*r*, [], β) $\longrightarrow$ *r* $\notin$ *final_states M*
   shows (*S2 q*, [], α) $\rightsquigarrow_c*$ (*S2 p*, [], γ)
using *assms* **proof** (*induction* (*q*, [] :: *'a list*, α) (*p*, [] :: *'a list*, γ) *arbitrary*: *q* α *rule*: *steps_induct2*)
  case *1*
  then show *?case*
    by (*simp add*: *comp.steps_refl*)
next
  case (*2 q* α *r u* β)
  from *2*(*1*) **obtain** *X* α' ζ **where** α*_def*: $\alpha = X\#\alpha'$ **and** *u_def*: *u* = [] **and** $*$: $\beta = \zeta$ @ α' **and** *elem*: (*r*, ζ) $\in$ $\delta\varepsilon$ *M q X*
    using *step*$_1$*_rule_ext*[*of q* [] α *r u* β] **by** *blast*
  from *2*(*4*)[*of r* β] **have** *r* $\notin$ *final_states M*
    using *step*$_1$*_steps*[*OF 2*(*1*)[*unfolded u_def*]] **by** *simp*
  with *elem* **have** (*S2 r*, ζ) $\in$ *comp_dpda_delta_eps* (*S2 q*) *X* **by** *force*
  with α*_def* $*$ **have** $**$: (*S2 q*, [], α) $\rightsquigarrow_c$ (*S2 r*, [], β)
   using *comp.step*$_1$*_rule*[*of S2 q* [] *X* α' *S2 r* [] β] **by** (*simp add*: *comp_dpda_def*)
  from *2*(*4*) **have** *nr*: $\bigwedge s\ \mu.$ (*r*, [], β) $\rightsquigarrow*$ (*s*, [], μ) $\longrightarrow$ *s* $\notin$ *final_states M*
    using *steps_trans*[*OF step*$_1$*_steps*[*OF 2*(*1*)], *unfolded u_def*] **by** *blast*
  from *2*(*3*)[*OF u_def nr*] **have** $***$: (*S2 r*, [], β) $\rightsquigarrow_c*$ (*S2 p*, [], γ) .
  show *?case*
    using *comp.steps_trans*[*OF comp.step*$_1$*_steps*[*OF* $**$] $***$] .
qed

The automaton *comp_dpda* mimics the steps of the original automaton in S2-states, provided that the word read so far is not accepted:

lemma *comp_dpda_nonfinal_steps*:
  assumes (*q'*, *w*, α) $\rightsquigarrow*$ (*p*, *u*, γ)
    and *w* $\neq$ *u*
    and $\bigwedge r\ \beta.$ (*q'*, *w*, α) $\rightsquigarrow*$ (*r*, *u*, β) $\longrightarrow$ *r* $\notin$ *final_states M*
    and *q* = *S1 q'* $\vee$ *q* = *S2 q'*
   shows (*q*, *w*, α) $\rightsquigarrow_c*$ (*S2 p*, *u*, γ)

using *assms* **proof** (*induction* $(q',\ w,\ \alpha)$ $(p,\ u,\ \gamma)$ *arbitrary*: $q\ q'\ w\ \alpha$ *rule*: *steps_induct2*)
case $(2\ q'\ w\ \alpha\ r\ v\ \beta)$
from *2(1)* **obtain** $X\ \alpha'$ **where** α_def: $\alpha = X\#\alpha'$ **and** *cases*:
$(\exists\,\zeta.\ v = w \wedge \beta = \zeta\ @\ \alpha' \wedge (r,\ \zeta) \in \delta\varepsilon\ M\ q'\ X) \vee (\exists\,a\ \zeta.\ w = a\ \#\ v \wedge \beta = \zeta\ @\ \alpha' \wedge (r,\ \zeta) \in \delta\ M\ q'\ a\ X)$ (**is** *?a* $\vee$ *?b*)
using $step_1\_rule\_ext$[*of* $q'\ w\ \alpha\ r\ v\ \beta$] **by** *blast*
from *2(5)* **have** *rp_nonfinal*: $\bigwedge ra\ \beta'.\ (r,\ v,\ \beta) \rightsquigarrow* (ra,\ u,\ \beta') \longrightarrow ra \notin$ *final_states M*
using $steps\_trans$[*OF* $step_1\_steps$[*OF* *2(1)*]] **by** *blast*
from *cases* **consider** (a) *?a* | (b) *?b* **by** *blast*
then show *?case*
proof *cases*
case *a*
then obtain ζ **where** $*$: $v = w$ **and** $**$: $\beta = \zeta\ @\ \alpha'$ **and** *elem*: $(r,\ \zeta) \in \delta\varepsilon\ M\ q'\ X$ **by** *blast*
from *elem* *2(6)* **have** $(S1\ r,\ \zeta) \in$ *comp_dpda_delta_eps* $q\ X \vee (S2\ r,\ \zeta) \in$ *comp_dpda_delta_eps* $q\ X$ **by** *force*
with $*\ **\ \alpha\_def$ **have** $***$: $(q,\ w,\ \alpha) \rightsquigarrow_c (S1\ r,\ v,\ \beta) \vee (q,\ w,\ \alpha) \rightsquigarrow_c (S2\ r,\ v,\ \beta)$
using $comp.step_1\_rule$[*of* $q\ w\ X\ \alpha'\ \_\ v\ \beta$] **by** (*simp add*: *comp_dpda_def*)
from *2(4)* $*$ **have** *v_neq*: $v \neq u$ **by** *simp*
from *2(3)*[*OF v_neq rp_nonfinal*] **have** $****$: $(S1\ r,\ v,\ \beta) \rightsquigarrow_c* (S2\ p,\ u,\ \gamma)$ **by** *simp*
from *2(3)*[*OF v_neq rp_nonfinal*] **have** $*****$: $(S2\ r,\ v,\ \beta) \rightsquigarrow_c* (S2\ p,\ u,\ \gamma)$ **by** *simp*
from $***$ $****$ $*****$ **show** *?thesis*
using $comp.step_1\_steps$ *comp.steps_trans* **by** *blast*
next
case *b*
then obtain $a\ \zeta$ **where** $*$: $w = a\ \#\ v$ **and** $**$: $\beta = \zeta\ @\ \alpha'$ **and** *elem*: $(r,\ \zeta) \in \delta\ M\ q'\ a\ X$ **by** *blast*
show *?thesis*
proof (*cases* $v = u$)
case *True*
from *2(5)* **have** *r_nonfinal*: $r \notin$ *final_states M*
using $step_1\_steps$[*OF* *2(1)*[*unfolded True*]] **by** *simp*
from *2(6)* **consider** $(s1)\ q = S1\ q'$ | $(s2)\ q = S2\ q'$ **by** *blast*
then have *p1*: $(q,\ w,\ \alpha) \rightsquigarrow_c* (S2\ r,\ v,\ \beta)$
proof *cases*
case *s1*
with *elem r_nonfinal* **have** $(S2\ r,\ \zeta) \in$ *comp_dpda_delta* $q\ a\ X$ **by** *force*
with $*\ **\ \alpha\_def$ **have** $***$: $(q,\ w,\ \alpha) \rightsquigarrow_c (S2\ r,\ v,\ \beta)$
using $comp.step_1\_rule$[*of* $q\ w\ X\ \alpha'\ S2\ r\ v\ \beta$] **by** (*simp add*: *comp_dpda_def*)
show *?thesis*
using $comp.step_1\_steps$[*OF* $***$] **.**
next
case *s2*
with *elem* **have** $(S3\ q',\ [X]) \in$ *comp_dpda_delta_eps* $q\ X$ **by** *auto*

with α_def **have** $***$: $(q,\ w,\ \alpha) \rightsquigarrow_c (S3\ q',\ w,\ \alpha)$
using $comp.step_1\_rule[of\ q\ w\ X\ \alpha'\ S3\ q'\ w\ \alpha]$ **by** $(simp\ add: comp\_dpda\_def)$
from $elem\ r\_nonfinal$ **have** $(S2\ r,\ \zeta) \in comp\_dpda\_delta\ (S3\ q')\ a\ X$ **by** *force*
with $* ** \alpha\_def$ **have** $****$: $(S3\ q',\ w,\ \alpha) \rightsquigarrow_c (S2\ r,\ v,\ \beta)$
using $comp.step_1\_rule[of\ S3\ q'\ w\ X\ \alpha'\ S2\ r\ v\ \beta]$ **by** $(simp\ add: comp\_dpda\_def)$
show *?thesis*
using $comp.steps\_trans[OF\ comp.step_1\_steps[OF\ ***]\ comp.step_1\_steps[OF\ ****]]$ **.**
qed
from $2(2)[unfolded\ True]$ **have** $a1$: $(r,\ [],\ \beta) \rightsquigarrow* (p,\ [],\ \gamma)$
using $steps\_word\_app[of\ r\ []\ \beta\ p\ []\ \gamma\ u]$ **by** *simp*
from $rp\_nonfinal[unfolded\ True]$ **have** $a2$: $\bigwedge ra\ \beta'.\ (r,\ [],\ \beta) \rightsquigarrow* (ra,\ [],\ \beta') \longrightarrow ra \notin final\_states\ M$
using $steps\_word\_app[of\ r\ []\ \beta\ \_\ []\ \_\ u]$ **by** *simp*
have $(S2\ r,\ [],\ \beta) \rightsquigarrow_c* (S2\ p,\ [],\ \gamma)$
using $comp\_dpda\_nonfinal\_stepsS2[OF\ a1\ a2]$ **.**
with $True$ **have** $p2$: $(S2\ r,\ v,\ \beta) \rightsquigarrow_c* (S2\ p,\ u,\ \gamma)$
using $comp.steps\_word\_app[of\ S2\ r\ []\ \beta\ S2\ p\ []\ \gamma\ u]$ **by** *simp*
show *?thesis*
using $comp.steps\_trans[OF\ p1\ p2]$ **.**
next
case *False*
from $2(6)$ **consider** $(s1)\ q = S1\ q' \mid (s2)\ q = S2\ q'$ **by** *blast*
then have p: $(q,\ w,\ \alpha) \rightsquigarrow_c* (S1\ r,\ v,\ \beta) \vee (q,\ w,\ \alpha) \rightsquigarrow_c* (S2\ r,\ v,\ \beta)$
proof $(cases)$
case *s1*
with $elem$ **have** $(S1\ r,\ \zeta) \in comp\_dpda\_delta\ q\ a\ X \vee (S2\ r,\ \zeta) \in comp\_dpda\_delta\ q\ a\ X$ **by** *force*
with $* ** \alpha\_def$ **have** $(q,\ w,\ \alpha) \rightsquigarrow_c (S1\ r,\ v,\ \beta) \vee (q,\ w,\ \alpha) \rightsquigarrow_c (S2\ r,\ v,\ \beta)$
using $comp.step_1\_rule[of\ q\ w\ X\ \alpha'\ \_\ v\ \beta]$ **by** $(simp\ add: comp\_dpda\_def)$
then show *?thesis*
using $comp.step_1\_steps$ **by** *blast*
next
case *s2*
with $elem$ **have** $(S3\ q',\ [X]) \in comp\_dpda\_delta\_eps\ q\ X$ **by** *auto*
with α_def **have** $***$: $(q,\ w,\ \alpha) \rightsquigarrow_c (S3\ q',\ w,\ \alpha)$
using $comp.step_1\_rule[of\ q\ w\ X\ \alpha'\ S3\ q'\ w\ \alpha]$ **by** $(simp\ add: comp\_dpda\_def)$
from $elem$ **have** $(S1\ r,\ \zeta) \in comp\_dpda\_delta\ (S3\ q')\ a\ X \vee (S2\ r,\ \zeta) \in comp\_dpda\_delta\ (S3\ q')\ a\ X$ **by** *force*
with $* ** \alpha\_def$ **have** $****$: $(S3\ q',\ w,\ \alpha) \rightsquigarrow_c (S1\ r,\ v,\ \beta) \vee (S3\ q',\ w,\ \alpha) \rightsquigarrow_c (S2\ r,\ v,\ \beta)$
using $comp.step_1\_rule[of\ S3\ q'\ w\ X\ \alpha'\ \_\ v\ \beta]$ **by** $(simp\ add: comp\_dpda\_def)$
from $***\ ****$ **show** *?thesis*
using $comp.step_1\_steps\ comp.steps\_trans$ **by** *metis*
qed
from $2(3)[OF\ False\ rp\_nonfinal]$ **have** $p1$: $(S1\ r,\ v,\ \beta) \rightsquigarrow_c* (S2\ p,\ u,\ \gamma)$ **by**

simp
from *2(3)*[*OF False rp_nonfinal*] **have** *p2*: (*S2 r*, *v*, β) $\rightsquigarrow_c*$ (*S2 p*, *u*, γ) **by** *simp*
from *p p1 p2* **show** *?thesis*
using *comp.steps_trans* **by** *blast*
qed
qed
qed *simp*

The automaton *comp_dpda* transitions to an S1-state if the target state is a final state:

lemma *comp_dpda_final_steps*:
assumes (q', *w*, α) $\rightsquigarrow+$ (*p*, *u*, γ)
and *p* $\in$ *final_states M*
and *q* = *S1 q'* $\vee$ *q* = *S2 q'*
shows (*q*, *w*, α) $\rightsquigarrow_c+$ (*S1 p*, *u*, γ)
using *assms* **proof** (*induction* (q', *w*, α) (*p*, *u*, γ) *arbitrary*: *q q' w* α *rule*: *steps1_induct*)
case (*base q' w* α)
from *base(1)* **obtain** *X* α' **where** α*_def*: α = *X*#α' **and** *cases*:
(∃ β. *u* = *w* ∧ γ = β @ α' ∧ (*p*, β) ∈ $\delta\varepsilon$ *M q' X*) ∨ (∃ *a* β. *w* = *a* # *u* ∧ γ = β @ α' ∧ (*p*, β) ∈ δ *M q' a X*) (**is** *?a* ∨ *?b*)
using *step*$_1$*_rule_ext*[*of q' w* α *p u* γ] **by** *blast*
from *cases* **consider** (*a*) *?a* | (*b*) *?b* **by** *blast*
then show *?case*
proof *cases*
case *a*
then obtain β **where** ∗: *u* = *w* **and** ∗∗: γ = β @ α' **and** *elem*: (*p*, β) ∈ $\delta\varepsilon$ *M q' X* **by** *blast*
from *base(2,3) elem* **have** (*S1 p*, β) ∈ *comp_dpda_delta_eps q X* **by** *force*
with ∗ ∗∗ α*_def* **have** ∗∗∗: (*q*, *w*, α) $\rightsquigarrow_c$ (*S1 p*, *u*, γ)
using *comp.step*$_1$*_rule*[*of q w X* α' *S1 p u* γ] **by** (*simp add*: *comp_dpda_def*)
show *?thesis*
using *comp.steps1_step*[*OF* ∗∗∗] **.**
next
case *b*
then obtain *a* β **where** ∗: *w* = *a* # *u* **and** ∗∗: γ = β @ α' **and** *elem*: (*p*, β) ∈ δ *M q' a X* **by** *blast*
from *base(3)* **consider** (*c*) *q* = *S1 q'* | (*d*) *q* = *S2 q'* **by** *blast*
then show *?thesis*
proof *cases*
case *c*
with *elem base(2)* **have** (*S1 p*, β) ∈ *comp_dpda_delta q a X* **by** *force*
with ∗ ∗∗ α*_def* **have** ∗∗∗: (*q*, *w*, α) $\rightsquigarrow_c$ (*S1 p*, *u*, γ)
using *comp.step*$_1$*_rule*[*of q w X* α' *S1 p u* γ] **by** (*simp add*: *comp_dpda_def*)
show *?thesis*
using *comp.steps1_step*[*OF* ∗∗∗] **.**
next
case *d*

with *elem* **have** $(S3\ q', [X]) \in comp\_dpda\_delta\_eps\ q\ X$ **by** *auto*
with α*_def* **have** $***$: $(q, w, \alpha) \rightsquigarrow_c (S3\ q', w, \alpha)$
using $comp.step_1\_rule[of\ q\ w\ X\ \alpha'\ S3\ q'\ w\ \alpha]$ **by** (*simp add*: *comp_dpda_def*)
from *elem base*(2) **have** $(S1\ p, \beta) \in comp\_dpda\_delta\ (S3\ q')\ a\ X$ **by** *force*
with $*$ $**$ α*_def* **have** $****$: $(S3\ q', w, \alpha) \rightsquigarrow_c (S1\ p, u, \gamma)$
using $comp.step_1\_rule[of\ S3\ q'\ w\ X\ \alpha'\ S1\ p\ u\ \gamma]$ **by** (*simp add*: *comp_dpda_def*)
show *?thesis*
using $comp.steps1\_trans[OF\ comp.steps1\_step[OF\ ***]\ comp.steps1\_step[OF\ ****]]$ **.**
qed
qed
next
case $(step\ q'\ w\ \alpha\ r\ v\ \beta)$
from *step*(1) **obtain** $X\ \alpha'$ **where** α*_def*: $\alpha = X\#\alpha'$ **and** *cases*:
$(\exists \beta'.\ v = w \wedge \beta = \beta' @ \alpha' \wedge (r, \beta') \in \delta\varepsilon\ M\ q'\ X) \vee (\exists a\ \beta'.\ w = a \#\ v \wedge \beta = \beta' @ \alpha' \wedge (r, \beta') \in \delta\ M\ q'\ a\ X)$ (**is** *?c* $\vee$ *?d*)
using $step_1\_rule\_ext[of\ q'\ w\ \alpha\ r\ v\ \beta]$ **by** *blast*
from *step*(5) **consider** (*a*) $q = S1\ q'$ | (*b*) $q = S2\ q'$ **by** *blast*
then show *?case*
proof *cases*
case *a*
from *cases* **consider** (*c*) *?c* | (*d*) *?d* **by** *blast*
then show *?thesis*
proof *cases*
case *c*
then obtain ζ **where** $*$: $v = w$ **and** $**$: $\beta = \zeta @ \alpha'$ **and** *elem*: $(r, \zeta) \in \delta\varepsilon\ M\ q'\ X$ **by** *blast*
from *elem a* **have** $(S1\ r, \zeta) \in comp\_dpda\_delta\_eps\ q\ X$ **by** *auto*
with $*$ $**$ α*_def* **have** $***$: $(q, w, \alpha) \rightsquigarrow_c (S1\ r, v, \beta)$
using $comp.step_1\_rule[of\ q\ w\ X\ \alpha'\ S1\ r\ v\ \beta]$ **by** (*simp add*: *comp_dpda_def*)
from $step(3)[OF\ step(4)]$ **have** $****$: $(S1\ r, v, \beta) \rightsquigarrow_c+ (S1\ p, u, \gamma)$ **by** *simp*
show *?thesis*
using $comp.steps1\_trans[OF\ comp.steps1\_step[OF\ ***]\ ****]$ **.**
next
case *d*
then obtain $a\ \zeta$ **where** $*$: $w = a \#\ v$ **and** $**$: $\beta = \zeta @ \alpha'$ **and** *elem*: $(r, \zeta) \in \delta\ M\ q'\ a\ X$ **by** *blast*
from *elem a* **have** $(S1\ r, \zeta) \in comp\_dpda\_delta\ q\ a\ X \vee (S2\ r, \zeta) \in comp\_dpda\_delta\ q\ a\ X$ **by** *force*
with $*$ $**$ α*_def* **have** $***$: $(q, w, \alpha) \rightsquigarrow_c (S1\ r, v, \beta) \vee (q, w, \alpha) \rightsquigarrow_c (S2\ r, v, \beta)$
using $comp.step_1\_rule[of\ q\ w\ X\ \alpha'\ \_\ v\ \beta]$ **by** (*simp add*: *comp_dpda_def*)
from $step(3)[OF\ step(4)]$ **have** $****$: $(S1\ r, v, \beta) \rightsquigarrow_c+ (S1\ p, u, \gamma)$ **by** *simp*
from $step(3)[OF\ step(4)]$ **have** $*****$: $(S2\ r, v, \beta) \rightsquigarrow_c+ (S1\ p, u, \gamma)$ **by** *simp*
from $***$ $****$ $*****$ **show** *?thesis*
using *comp.steps1_step comp.steps1_trans* **by** *blast*
qed
next
case *b*

```
    from cases consider (c) ?c | (d) ?d by blast
    then show ?thesis
    proof cases
      case c
      then obtain ζ where ∗: v = w and ∗∗: β = ζ @ α' and elem: (r, ζ) ∈ δε
M q' X by blast
       from elem b have (S1 r, ζ) ∈ comp_dpda_delta_eps q X ∨ (S2 r, ζ) ∈
comp_dpda_delta_eps q X by force
      with ∗ ∗∗ α_def have ∗∗∗: (q, w, α) ⇝c (S1 r, v, β) ∨ (q, w, α) ⇝c (S2 r,
v, β)
       using comp.step1_rule[of q w X α' _ v β] by (simp add: comp_dpda_def)
      from step(3)[OF step(4)] have ∗∗∗∗: (S1 r, v, β) ⇝c+ (S1 p, u, γ) by simp
      from step(3)[OF step(4)] have ∗∗∗∗∗: (S2 r, v, β) ⇝c+ (S1 p, u, γ) by simp
       from ∗∗∗ ∗∗∗∗ ∗∗∗∗∗ show ?thesis
        using comp.steps1_step comp.steps1_trans by blast
    next
      case d
      then obtain a ζ where ∗: w = a # v and ∗∗: β = ζ @ α' and elem: (r, ζ)
∈ δ M q' a X by blast
      from elem b have (S3 q', [X]) ∈ comp_dpda_delta_eps q X by auto
      with α_def have ∗∗∗: (q, w, α) ⇝c (S3 q', w, α)
      using comp.step1_rule[of q w X α' S3 q' w α] by (simp add: comp_dpda_def)
       from elem have (S1 r, ζ) ∈ comp_dpda_delta (S3 q') a X ∨ (S2 r, ζ) ∈
comp_dpda_delta (S3 q') a X by force
       with ∗ ∗∗ α_def have ∗∗∗∗: (S3 q', w, α) ⇝c (S1 r, v, β) ∨ (S3 q', w, α)
⇝c (S2 r, v, β)
      using comp.step1_rule[of S3 q' w X α' _ v β] by (simp add: comp_dpda_def)
      from step(3)[OF step(4)] have ∗∗∗∗∗: (S1 r, v, β) ⇝c+ (S1 p, u, γ) by simp
       from step(3)[OF step(4)] have ∗∗∗∗∗∗: (S2 r, v, β) ⇝c+ (S1 p, u, γ) by
simp
      from ∗∗∗ ∗∗∗∗ ∗∗∗∗∗ ∗∗∗∗∗∗ show ?thesis
        using comp.steps1_step comp.steps1_trans by metis
    qed
  qed
qed
```

If the automaton *comp_dpda* ends up in an S3-state while reading a word, then no state reached while reading that same word is final:

```
lemma comp_dpda_steps_nonfinalS1:
  assumes (S1 q, w, α) ⇝c∗ (S3 p, [], γ)
      and (q, w, α) ⇝∗ (r, [], β)
    shows r ∉ final_states M
proof
  assume r_final: r ∈ final_states M
  from assms(1) obtain n where p1: (S1 q, w, α) ⇝c(n) (S3 p, [], γ)
    using comp.stepn_steps[of S1 q w α S3 p [] γ] by blast
  have np: comp.no_step1 (S3 p, [], γ)
    using comp.step1_rule_ext[of S3 p [] γ] by (simp add: comp_dpda_def)
  have (S1 q, w, α) ⇝c∗ (S1 r, [], β)
```

```
  proof (cases (q, w, α) = (r, [], β))
    case True
    then show ?thesis
      by (simp add: comp.steps_refl)
  next
    case False
    have q1: (q, w, α) ⇝+ (r, [], β)
      using steps_steps1[OF assms(2) False] .
    have q2: (S1 q, w, α) ⇝c+ (S1 r, [], β)
      using comp_dpda_final_steps[OF q1 r_final] by simp
    show ?thesis
      using comp.steps1_steps[OF q2] .
  qed
  then obtain m where p2: (S1 q, w, α) ⇝c(m) (S1 r, [], β)
    using comp.stepn_steps[of S1 q w α S1 r [] β] by blast
  have m_leq_n: m ≤ n
    using comp.max_eps_steps[OF p1 np p2] .
  have (S1 r, [], β) ⇝c(n − m) (S3 p, [], γ)
   using comp.split_path[OF p1 m_leq_n] comp.dpda_stepn_det[OF p2] by blast
  then have ∗: (S1 r, [], β) ⇝c∗ (S3 p, [], γ)
    using comp.stepn_steps[of S1 r [] β S3 p [] γ] by blast
  show False
    using comp_dpda_steps_from_S1[OF ∗] by simp
qed

lemma comp_dpda_steps_nonfinalS2:
  assumes (S2 q, w, α) ⇝c∗ (S3 p, [], γ)
      and (q, w, α) ⇝+ (r, [], β)
    shows r ∉ final_states M
proof
  assume r_final: r ∈ final_states M
  from assms(1) obtain n where p1: (S2 q, w, α) ⇝c(n) (S3 p, [], γ)
    using comp.stepn_steps[of S2 q w α S3 p [] γ] by blast
  have np: comp.no_step1 (S3 p, [], γ)
    using comp.step1_rule_ext[of S3 p [] γ] by (simp add: comp_dpda_def)
  obtain m where p2: (S2 q, w, α) ⇝c(m) (S1 r, [], β)
    using comp.steps1_steps[OF comp_dpda_final_steps[OF assms(2) r_final, of
S2 q, simplified]] comp.stepn_steps[of S2 q w α S1 r [] β] by blast
  have m_leq_n: m ≤ n
    using comp.max_eps_steps[OF p1 np p2] .
  have (S1 r, [], β) ⇝c(n − m) (S3 p, [], γ)
   using comp.split_path[OF p1 m_leq_n] comp.dpda_stepn_det[OF p2] by blast
  then have ∗: (S1 r, [], β) ⇝c∗ (S3 p, [], γ)
    using comp.stepn_steps[of S1 r [] β S3 p [] γ] by blast
  show False
    using comp_dpda_steps_from_S1[OF ∗] by simp
qed
```

The language of the automaton *comp_dpda* is the complement language:

```
lemma lang_comp_dpda:
  comp.accept_final = - accept_final
proof
  show comp.accept_final ⊆ - accept_final
  proof
    fix w
    assume w ∈ comp.accept_final
   then obtain q γ where p: (init_state comp_dpda, w, [init_symbol comp_dpda])
↝c* (q, [], γ) and q_final: q ∈ final_states comp_dpda
      unfolding comp.accept_final_def by blast
    from q_final obtain q' where q_def: q = S3 q'
      by (auto simp: comp_dpda_def comp_dpda_final_states_def)
   have ⋀r β. (init_state M, w, [init_symbol M]) ↝* (r, [], β) ⟶ r ∉ final_states
M
    proof
      fix r β
      assume asm: (init_state M, w, [init_symbol M]) ↝* (r, [], β)
     consider (a) init_state M ∈ final_states M | (b) init_state M ∉ final_states
M by satx
      then show r ∉ final_states M
      proof cases
        case a
       with p q_def have *: (S1 (init_state M), w, [init_symbol M]) ↝c* (S3 q',
[], γ)
          by (simp add: comp_dpda_def comp_dpda_init_state_def)
        show ?thesis
          using comp_dpda_steps_nonfinalS1[OF * asm] .
      next
        case b
        consider (c) r = init_state M | (d) r ≠ init_state M by satx
        then show ?thesis
        proof cases
          case c
          with b show ?thesis by simp
        next
          case d
          then have *: (init_state M, w, [init_symbol M]) ↝+ (r, [], β)
            using steps_steps1[OF asm] by simp
           from b p q_def have **: (S2 (init_state M), w, [init_symbol M]) ↝c*
(S3 q', [], γ)
            by (simp add: comp_dpda_def comp_dpda_init_state_def)
          show ?thesis
            using comp_dpda_steps_nonfinalS2[OF ** *] .
        qed
      qed
    qed
    then show w ∈ - accept_final
      by (auto simp: accept_final_def)
  qed
```

```
next
  show - accept_final ⊆ comp.accept_final
  proof
    fix w
    assume w ∈ - accept_final
    then have nonfinal: ⋀r β. (init_state M, w, [init_symbol M]) ⇝* (r, [], β)
⟶ r ∉ final_states M
      by (auto simp: accept_final_def)
    from M_path[of w] obtain q X α a where p: (init_state M, w, [init_symbol
M]) ⇝* (q, [], X # α) and delta_M: δ M q a X ≠ {} by blast
    consider (a) w = [] | (b) w ≠ [] by satx
    then have *: (init_state comp_dpda, w, [init_symbol comp_dpda]) ⇝c* (S2
q, [], X#α)
    proof cases
      case a
      from nonfinal[unfolded a] have init_state M ∉ final_states M
        using steps_refl[of init_state M [] [init_symbol M]] by simp
      with a show ?thesis
        using comp_dpda_nonfinal_stepsS2[OF p[unfolded a] nonfinal[unfolded a]]
by (simp add: comp_dpda_def comp_dpda_init_state_def)
    next
      case b
      with p nonfinal show ?thesis
        using comp_dpda_nonfinal_steps[of init_state M w [init_symbol M] q []
X#α] by (simp add: comp_dpda_def comp_dpda_init_state_def)
    qed
    from delta_M have **: (S2 q, [], X#α) ⇝c (S3 q, [], X#α)
      using comp.step1_rule[of S2 q [] X α S3 q [] X#α] comp_dpda_def by auto
    have (init_state comp_dpda, w, [init_symbol comp_dpda]) ⇝c* (S3 q, [], X #
α)
      using comp.steps_trans[OF * comp.step1_steps[OF **]] .
    then show w ∈ comp.accept_final
      using comp.accept_final_def comp_dpda_def comp_dpda_final_states_def
by force
  qed
qed

end
```

By constructing the two automata one after another, we get the final lemma stating that deterministic context-free languages are closed under complementation:

```
lemma complement_dpda:
  assumes dpda (M :: ('q :: finite, 'a :: finite, 's :: finite) pda)
  shows ∃ M' :: ('q st_scan s123, 'a, 's sym_scan) pda.
    dpda M' ∧ pda.accept_final M' = - pda.accept_final M
proof -
  let ?SM = dpda_scan.scan_dpda M :: ('q st_scan, 'a, 's sym_scan) pda
  have dpda_sm: dpda ?SM
```

using *assms dpda_scan.dpda_scan_dpda dpda_scan_def* **by** *blast*
have $*$: $\bigwedge a\ w.\ \exists\, q\ X\ \alpha.$ *pda.steps ?SM* (*init_state ?SM*, *w*, [*init_symbol ?SM*])
(q, [], $X\#\alpha$) $\wedge$ *delta ?SM q a X* $\neq$ {}
using *assms dpda_scan.scan_dpda_scans dpda_scan_def* **by** *blast*
have *L1*: *pda.accept_final ?SM* = *pda.accept_final M*
using *assms dpda_scan.lang_scan_dpda dpda_scan.intro* **by** *auto*
let *?CM* = *complement_dpda.comp_dpda ?SM* :: (*'q st_scan s123*, *'a*, *'s sym_scan*)
pda
from *dpda_sm* $*$ **have** *dpda_cm*: *dpda ?CM*
using *complement_dpda.dpda_comp_dpda complement_dpda.intro complement_dpda_axioms_def*
by *blast*
from *dpda_sm* $*$ **have** *L2*: *pda.accept_final ?CM* = *UNIV* $-$ *pda.accept_final*
?SM
using *complement_dpda.lang_comp_dpda complement_dpda_axioms_def com-*
plement_dpda_def **by** *blast*
from *dpda_cm L1 L2* **show** *?thesis* **by** *blast*
qed

end

References

[1] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.