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 $\delta\_nonempty$: $\delta\ M\ q\ a\ X \neq \{\} \longrightarrow \delta\varepsilon\ M\ q\ X = \{\}$
and $\delta\_singleton$: $\delta\ M\ q\ a\ X = \{\} \vee (\exists\, p\ \gamma.\ \delta\ M\ q\ a\ X = \{(p, \gamma)\})$
and $\delta\varepsilon\_singleton$: $\delta\varepsilon\ M\ q\ X = \{\} \vee (\exists\, p\ \gamma.\ \delta\varepsilon\ M\ q\ X = \{(p, \gamma)\})$
begin

In every configuration,

- $\delta\_nonempty$ enforces that not both an epsilon step and a true step can be enabled.
- $\delta\_singleton$ allows for at most one true step.
- $\delta\varepsilon\_singleton$ allows for at most one epsilon step.

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

lemma $dpda\_step$: $step\ (q,\ w,\ \alpha) = \{\} \vee (\exists\, p\ u\ \gamma.\ step\ (q,\ w,\ \alpha) = \{(p,\ u,\ \gamma)\})$
⟨*proof*⟩

A step of the automaton is indeed deterministic:

lemma $dpda\_step_1\_det$:
assumes $(q,\ w,\ \alpha) \rightsquigarrow (p,\ u,\ \gamma)$
and $(q,\ w,\ \alpha) \rightsquigarrow (p',\ u',\ \gamma')$
shows $p = p' \wedge u = u' \wedge \gamma = \gamma'$
⟨*proof*⟩

lemma $dpda\_stepn\_det$:
assumes $(q,\ w,\ \alpha) \rightsquigarrow(n)\ (p,\ u,\ \gamma)$
and $(q,\ w,\ \alpha) \rightsquigarrow(n)\ (p',\ u',\ \gamma')$
shows $p = p' \wedge u = u' \wedge \gamma = \gamma'$
⟨*proof*⟩

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 $steps1 :: 'q \times 'a\ list \times 's\ list \Rightarrow 'q \times 'a\ list \times 's\ list \Rightarrow bool$ (**infix** $\rightsquigarrow+$ *55*) **where**
$steps1 \equiv step_1\ \widehat{}++$

abbreviation *no_step*$_1$ ((_ $\rightsquigarrow|$) [*1000*] *999*) **where**
  $cf \rightsquigarrow| \equiv (\forall\, cf'.\ \neg cf \rightsquigarrow cf')$

lemma *steps1_induct*[*consumes 1*, *case_names base step*]:
  assumes *x1* $\rightsquigarrow+$ *x2*
      and $\bigwedge q\ w\ \alpha\ p\ u\ \gamma.\ (q,\ w,\ \alpha) \rightsquigarrow (p,\ u,\ \gamma) \Longrightarrow P\ (q,\ w,\ \alpha)\ (p,\ u,\ \gamma)$
      and $\bigwedge q\ w\ \alpha\ r\ v\ \beta\ p\ u\ \gamma.\ (q,\ w,\ \alpha) \rightsquigarrow (r,\ v,\ \beta) \Longrightarrow (r,\ v,\ \beta) \rightsquigarrow+ (p,\ u,\ \gamma)$
$\Longrightarrow$
            $P\ (r,\ v,\ \beta)\ (p,\ u,\ \gamma) \Longrightarrow P\ (q,\ w,\ \alpha)\ (p,\ u,\ \gamma)$
    shows *P x1 x2*
⟨*proof*⟩

lemma *steps1_converse_induct*[*consumes 1*, *case_names base step*]:
  assumes *x1* $\rightsquigarrow+$ *x2*
      and $\bigwedge q\ w\ \alpha\ p\ u\ \gamma.\ (q,\ w,\ \alpha) \rightsquigarrow (p,\ u,\ \gamma) \Longrightarrow P\ (q,\ w,\ \alpha)\ (p,\ u,\ \gamma)$
      and $\bigwedge q\ w\ \alpha\ r\ v\ \beta\ p\ u\ \gamma.\ (q,\ w,\ \alpha) \rightsquigarrow+ (r,\ v,\ \beta) \Longrightarrow (r,\ v,\ \beta) \rightsquigarrow (p,\ u,\ \gamma)$
$\Longrightarrow$
            $P\ (q,\ w,\ \alpha)\ (r,\ v,\ \beta) \Longrightarrow P\ (q,\ w,\ \alpha)\ (p,\ u,\ \gamma)$
    shows *P x1 x2*
  ⟨*proof*⟩

lemma *steps1_steps*:
  $(q,\ w,\ \alpha) \rightsquigarrow+ (p,\ u,\ \gamma) \Longrightarrow (q,\ w,\ \alpha) \rightsquigarrow* (p,\ u,\ \gamma)$
  ⟨*proof*⟩

lemma *steps1_step*:
  $(q,\ w,\ \alpha) \rightsquigarrow (p,\ u,\ \gamma) \Longrightarrow (q,\ w,\ \alpha) \rightsquigarrow+ (p,\ u,\ \gamma)$
  ⟨*proof*⟩

lemma *steps1_trans*:
  $(q,\ w,\ \alpha) \rightsquigarrow+ (p,\ u,\ \gamma) \Longrightarrow (p,\ u,\ \gamma) \rightsquigarrow+ (r,\ v,\ \beta) \Longrightarrow (q,\ w,\ \alpha) \rightsquigarrow+ (r,\ v,\ \beta)$
⟨*proof*⟩

lemma *steps_steps1*:
  assumes $(q,\ w,\ \alpha) \rightsquigarrow* (p,\ u,\ \gamma)$
      and $(q,\ w,\ \alpha) \neq (p,\ u,\ \gamma)$
    shows $(q,\ w,\ \alpha) \rightsquigarrow+ (p,\ u,\ \gamma)$
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *split_path*:
  assumes $(q,\ w,\ \alpha) \rightsquigarrow(n)\ (p,\ [],\ \gamma)$
      and $m \leq n$
    shows $\exists\, r\ u\ \beta.\ (q,\ w,\ \alpha) \rightsquigarrow(m)\ (r,\ u,\ \beta) \wedge (r,\ u,\ \beta) \rightsquigarrow(n-m)\ (p,\ [],\ \gamma)$
⟨*proof*⟩

end

context *dpda* **begin**

lemma *max_eps_steps*:
  assumes $(q,\ w,\ \alpha) \rightsquigarrow (n)\ (p,\ u,\ \gamma)$
    and $(p,\ [],\ \gamma) \rightsquigarrow |$
    and $(q,\ w,\ \alpha) \rightsquigarrow (m)\ (r,\ u,\ \beta)$
  shows $m \leq n$
⟨*proof*⟩

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*
  ⟨*proof*⟩

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

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

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*⟩

2.2.3 Equivalence

sublocale *scan*: *dpda scan_dpda*
⟨*proof*⟩

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* ⇒ *'s sym_scan list* **where**
stack_with_X0 $\alpha \equiv$ *map Sym* α @ [*X0*]

lemma *scan_dpda_first_step*:
assumes (*Q0'*, *w*, [*X0*]) $\rightsquigarrow_s$ (*q*, *u*, α)
shows q = *St* (*init_state M*) ∧ $u = w \wedge \alpha$ = [*Sym* (*init_symbol M*), *X0*]
⟨*proof*⟩

lemma *scan_dpda_step_from_St*:
assumes (*St q*, *w*, α) $\rightsquigarrow_s$ (*p*, *u*, γ)
shows (∃ *p'*. p = *St p'*) ∨ $p = F \vee p = D$
⟨*proof*⟩

lemma *scan_dpda_step_from_F*:
assumes (*F*, *w*, α) $\rightsquigarrow_s$ (*q*, *u*, γ)
shows $q = D$
⟨*proof*⟩

lemma *scan_dpda_step_to_F*:
assumes (*q*, *w*, *X*#α) $\rightsquigarrow_s$ (*F*, *u*, γ)
shows $u = w \wedge$ (∃ *q' X'*. q = *St q'* ∧ X = *Sym X'* ∧ *eps_final q' X'*)
⟨*proof*⟩

lemma *scan_dpda_step_from_D*:
assumes (*D*, *w*, α) $\rightsquigarrow_s$ (*q*, *u*, γ)
shows $q = D$
⟨*proof*⟩

lemma *scan_dpda_steps_from_St*:
assumes (*St q*, *w*, α) $\rightsquigarrow_s*$ (*p*, *u*, γ)
shows (∃ *p'*. p = *St p'*) ∨ $p = F \vee p = D$
⟨*proof*⟩

lemma *scan_dpda_step_to_St*:
assumes (*q*, *w*, α) $\rightsquigarrow_s$ (*St p*, *u*, γ)
shows q = *Q0'* ∨ (∃ *q'*. q = *St q'*)
⟨*proof*⟩

lemma *scan_dpda_stack_with_X0*:
assumes (*St q*, *w*, *stack_with_X0* α) $\rightsquigarrow_s*$ (*St p*, *u*, γ)
shows ∃ γ'. γ = *stack_with_X0* γ'

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *scan_dpda_eps*:
assumes (*St q*, *map Sym α*) ∈ δε *scan_dpda* (*St p*) (*Sym X*)
shows (*q*, *α*) ∈ δε *M p X*
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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 ∃ *Y β′*. *β* = *Y* # *β′* @ *γ* ∧ (*q*, [], [*X*]) ⇝* (*r*, [], *Y*#*β′*)
⟨*proof*⟩

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

lemma *scan_dpda_steps*:
assumes (*q*, *w*, *α*) ⇝* (*p*, *u*, *γ*)
shows (*St q*, *w*, *map Sym α*) ⇝$_s$* (*St p*, *u*, *map Sym γ*) ∨
(∃ *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)))
⟨*proof*⟩

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*⟩

2.2.4 Scan Property

lemma *D_consumes*: $(D, w, X\#\alpha) \rightsquigarrow_s* (D, [], X\#\alpha)$
⟨*proof*⟩

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

definition *eps_infs* :: $'q \Rightarrow 's\ list \Rightarrow bool$ **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*⟩

lemma *some_pair_inf*:
assumes *eps_infs q* α
and ⋀*p X* γ. (*q*, [], α) ⇝* (*p*, [], *X*#γ) ⟶ ¬*eps_inf p X*
shows *False*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eps_inf_to_D*:
assumes *eps_inf q X*
shows (*St q*, *w*, *Sym X* # α) $\rightsquigarrow_s*$ (*D*, [], *Sym X* # α)
⟨*proof*⟩

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) $\rightsquigarrow_s*$ (*D*, [], *Sym Y* # β)
⟨*proof*⟩

lemma *scan_dpda_neps_infs*:
assumes ¬*eps_infs q* α
shows ∃ *p* γ. (*St q*, *w*, *map Sym* α) $\rightsquigarrow_s*$ (*St p*, *w*, *map Sym* γ) ∧ (∀ *X* γ′. γ = *X*#γ′ ⟶ δε *M p X* = {})

⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *inj_S1*: *inj S1*
⟨*proof*⟩

lemma *inj_S2*: *inj S2*
⟨*proof*⟩

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*})
⟨*proof*⟩

The automaton *comp_dpda* is deterministic:

lemma *dpda_comp_dpda*: *dpda comp_dpda*
⟨*proof*⟩

2.3.3 Complementation

sublocale *comp*: *dpda comp_dpda*
⟨*proof*⟩

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

notation *comp.step*$_1$ (**infix** $\rightsquigarrow_c$ *55*)
notation *comp.steps* (**infix** $\rightsquigarrow_c*$ *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'
⟨*proof*⟩

lemma *comp_dpda_steps_from_S1*:
assumes (*S1* q, [], α) $\rightsquigarrow_c*$ (p, [], γ)
shows $\exists\, p'.\ p$ = *S1* p'
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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) $\rightsquigarrow_c*$ (*S3* p, [], γ)
and (q, w, α) $\rightsquigarrow*$ (r, [], β)
shows $r \notin$ *final_states M*
⟨*proof*⟩

lemma *comp_dpda_steps_nonfinalS2*:
assumes (*S2* q, w, α) $\rightsquigarrow_c*$ (*S3* p, [], γ)
and (q, w, α) $\rightsquigarrow+$ (r, [], β)
shows $r \notin$ *final_states M*

⟨*proof*⟩

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

lemma *lang_comp_dpda*:
comp.accept_final = − *accept_final*
⟨*proof*⟩

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*⟩

end

References

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