

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

Kaan Taskin and Tobias Nipkow

October 6, 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	1
2	DCFLs are Closed Under Complementation	2
2.1	Setup and Auxiliary Lemmas	2
2.2	Scan Construction	4
2.2.1	Definition	4
2.2.2	Determinism	5
2.2.3	Equivalence	6
2.2.4	Scan Property	8
2.3	Complement Construction	9
2.3.1	Definition	9
2.3.2	Determinism	10
2.3.3	Complementation	10

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  $dpda\_step$ :  $step \ (q, w, \alpha) = \{\} \vee (\exists p \ u \ \gamma. \ step \ (q, w, \alpha) = \{(p, u, \gamma)\})$ 
 $\langle proof \rangle$ 

  A step of the automaton is indeed deterministic:

lemma  $dpda\_step1\_det$ :
assumes  $(q, w, \alpha) \rightsquigarrow (p, u, \gamma)$ 
and  $(q, w, \alpha) \rightsquigarrow (p', u', \gamma')$ 
shows  $p = p' \wedge u = u' \wedge \gamma = \gamma'$ 
 $\langle proof \rangle$ 

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

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
   $steps1 \equiv step_1 \hat{+}+$ 

```

abbreviation *no_step1* ($(_ \rightsquigarrow |)$ [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 $\bigwedge q w \alpha p u \gamma. (q, w, \alpha) \rightsquigarrow (p, u, \gamma) \implies P (q, w, \alpha) (p, u, \gamma)$
and $\bigwedge q w \alpha r v \beta p u \gamma. (q, w, \alpha) \rightsquigarrow (r, v, \beta) \implies (r, v, \beta) \rightsquigarrow + (p, u, \gamma)$
 $\implies$
 $P (r, v, \beta) (p, u, \gamma) \implies P (q, w, \alpha) (p, u, \gamma)$
shows $P x1 x2$
 $\langle proof \rangle$

lemma *steps1_converse_induct*[*consumes 1, case_names base step*]:

assumes $x1 \rightsquigarrow + x2$
and $\bigwedge q w \alpha p u \gamma. (q, w, \alpha) \rightsquigarrow (p, u, \gamma) \implies P (q, w, \alpha) (p, u, \gamma)$
and $\bigwedge q w \alpha r v \beta p u \gamma. (q, w, \alpha) \rightsquigarrow + (r, v, \beta) \implies (r, v, \beta) \rightsquigarrow (p, u, \gamma)$
 $\implies$
 $P (q, w, \alpha) (r, v, \beta) \implies P (q, w, \alpha) (p, u, \gamma)$
shows $P x1 x2$
 $\langle proof \rangle$

lemma *steps1_steps*:

$(q, w, \alpha) \rightsquigarrow + (p, u, \gamma) \implies (q, w, \alpha) \rightsquigarrow * (p, u, \gamma)$
 $\langle proof \rangle$

lemma *steps1_step*:

$(q, w, \alpha) \rightsquigarrow (p, u, \gamma) \implies (q, w, \alpha) \rightsquigarrow + (p, u, \gamma)$
 $\langle proof \rangle$

lemma *steps1_trans*:

$(q, w, \alpha) \rightsquigarrow + (p, u, \gamma) \implies (p, u, \gamma) \rightsquigarrow + (r, v, \beta) \implies (q, w, \alpha) \rightsquigarrow + (r, v, \beta)$
 $\langle proof \rangle$

lemma *steps_steps1*:

assumes $(q, w, \alpha) \rightsquigarrow * (p, u, \gamma)$
and $(q, w, \alpha) \neq (p, u, \gamma)$
shows $(q, w, \alpha) \rightsquigarrow + (p, u, \gamma)$
 $\langle proof \rangle$

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

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

lemma *split_path*:

assumes $(q, w, \alpha) \rightsquigarrow (n) (p, [], \gamma)$
and $m \leq n$
shows $\exists r u \beta. (q, w, \alpha) \rightsquigarrow (m) (r, u, \beta) \wedge (r, u, \beta) \rightsquigarrow (n-m) (p, [], \gamma)$
 $\langle proof \rangle$

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 $m \leq n$

$\langle proof \rangle$

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 $'q \text{ st_scan} = St \ 'q \mid Q0' \mid D \mid F$

datatype $'s \text{ sym_scan} = Sym \ 's \mid X0$

lemma *inj_Sym*: *inj Sym*

$\langle proof \rangle$

instance *st_scan* :: (*finite*) *finite*

$\langle proof \rangle$

instance *sym_scan* :: (*finite*) *finite*

$\langle proof \rangle$

```

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

$$scan_dpda_final_states \equiv St \text{ ' } final_states \text{ } M \cup \{F\}$$
$$\text{fun } \text{scan_dpda_delta} :: 'q \text{ st_scan} \Rightarrow 'a \Rightarrow 's \text{ sym_scan} \Rightarrow ('q \text{ st_scan} \times 's \text{ sym_scan list}) \text{ set where}$$
$$scan_dpda_delta (St\ q)\ a\ (Sym\ X) = (if\ \delta\ M\ q\ a\ X = \{\} \wedge \delta\varepsilon\ M\ q\ X = \{\} \text{ then } \{(D, [Sym\ X])\})$$
$$\text{else } (\lambda(q, \alpha). (St\ q, \text{map Sym } \alpha)) \text{ ' } \delta\ M\ q\ a\ X)$$
$$| \text{scan_dpda_delta} (St\ q) _ X0 = \{(D, [X0])\}$$
$$| \text{scan_dpda_delta } D _ X = \{(D, [X])\}$$
$$| \textit{scan_dpda_delta} _ _ _ = \{ \}$$

definition $eps_nonfinal :: 'q \Rightarrow 's \Rightarrow bool$ **where**

$$eps_nonfinal\ q\ X \equiv (\forall i. \exists p\ \alpha. (q, [], [X]) \rightsquigarrow(i) (p, [], \alpha) \wedge p \notin final_states\ M)$$

definition $eps_final :: 'q \Rightarrow 's \Rightarrow bool$ **where**

$$eps_final\ q\ X \equiv (\forall i. \exists p\ \alpha. (q, [], [X]) \rightsquigarrow (i) (p, [], \alpha)) \wedge (\exists i\ p\ \alpha. (q, [], [X])$$
$$\rightsquigarrow (i) \ (p, [], \alpha) \wedge p \in final_states \ M)$$
$$\text{fun } \text{scan_dpda_delta_eps} :: 'q \text{ st_scan} \Rightarrow 's \text{ sym_scan} \Rightarrow ('q \text{ st_scan} \times 's \text{ sym_scan} \text{ list}) \text{ set where}$$
$$scan_dpda_delta_eps \ Q0' \ X0 = \{(St \ (init_state \ M), [Sym \ (init_symbol \ M), X0])\}$$
$$| \text{scan_dpda_delta_eps } (St \ q) \ (Sym \ X) = (if \ \text{eps_nonfinal } q \ X \ \text{then } \{(D, [Sym \ X])\} \ \text{else}$$

if $eps_final \ q \ X$ then $\{(F, [Sym \ X])\}$ else

$$(\lambda(q, \alpha). (St\ q, map\ Sym\ \alpha)) \cdot \delta\varepsilon\ M\ q\ X)$$
$$\mid scan_dpda_delta_eps \ F \ X = \{(D, [X])\}$$
$$| \textit{scan_dpda_delta_eps} _ _ = \{ \}$$

definition $scan_dpda :: ('q\ st_scan, 'a, 's\ sym_scan)\ pda$ **where**

$$scan_dpda =$$
$$(\mid init_state = Q0',$$
$$init_symbol = X0,$$
$$final_states = scan_dpda_final_states,$$
$$\text{delta} = \text{scan_dpda_delta}, \text{delta_eps} = \text{scan_dpda_delta_eps} \mid$$

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*
 $\langle proof \rangle$

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

notation *scan.step₁* (**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 $\alpha \equiv \text{map } \text{Sym } \alpha @ [X0]$

lemma *scan_dpda_first_step*:
assumes (*Q0'*, *w*, [*X0*]) $\rightsquigarrow_s$ (*q*, *u*, α)
shows $q = \text{St } (\text{init_state } M) \wedge u = w \wedge \alpha = [\text{Sym } (\text{init_symbol } M), X0]$
 $\langle proof \rangle$

lemma *scan_dpda_step_from_St*:
assumes (*St q*, *w*, α) $\rightsquigarrow_s$ (*p*, *u*, γ)
shows $(\exists p'. p = \text{St } p') \vee p = F \vee p = D$
 $\langle proof \rangle$

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

lemma *scan_dpda_step_to_F*:
assumes (*q*, *w*, $X \# \alpha$) $\rightsquigarrow_s$ (*F*, *u*, γ)
shows $u = w \wedge (\exists q' X'. q = \text{St } q' \wedge X = \text{Sym } X' \wedge \text{eps_final } q' X')$
 $\langle proof \rangle$

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

lemma *scan_dpda_steps_from_St*:
assumes (*St q*, *w*, α) $\rightsquigarrow_{s*}$ (*p*, *u*, γ)
shows $(\exists p'. p = \text{St } p') \vee p = F \vee p = D$
 $\langle proof \rangle$

lemma *scan_dpda_step_to_St*:
assumes (*q*, *w*, α) $\rightsquigarrow_s$ (*St p*, *u*, γ)
shows $q = Q0' \vee (\exists q'. q = \text{St } q')$
 $\langle proof \rangle$

lemma *scan_dpda_stack_with_X0*:
assumes (*St q*, *w*, *stack_with_X0* α) $\rightsquigarrow_{s*}$ (*St p*, *u*, γ)
shows $\exists \gamma'. \gamma = \text{stack_with_X0 } \gamma'$

$\langle proof \rangle$

lemma *scan_dpda_trans*:

assumes $(St\ q, map\ Sym\ \alpha) \in \delta\ scan_dpda\ (St\ p)\ a\ (Sym\ X)$

shows $(q, \alpha) \in \delta\ M\ p\ a\ X$

$\langle proof \rangle$

lemma *scan_dpda_eps*:

assumes $(St\ q, map\ Sym\ \alpha) \in \delta\epsilon\ scan_dpda\ (St\ p)\ (Sym\ X)$

shows $(q, \alpha) \in \delta\epsilon\ M\ p\ X$

$\langle proof \rangle$

lemma *scan_dpda_step*:

assumes $(St\ q, w, map\ Sym\ \alpha) \rightsquigarrow_s (St\ p, u, map\ Sym\ \gamma)$

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

$\langle proof \rangle$

lemma *scan_dpda_stepX0*:

assumes $(St\ q, w, stack_with_X0\ \alpha) \rightsquigarrow_s (St\ p, u, stack_with_X0\ \gamma)$

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

$\langle proof \rangle$

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

lemma *scan_dpda_stepsX0*:

assumes $(St\ q, w, stack_with_X0\ \alpha) \rightsquigarrow_{s*} (St\ p, u, stack_with_X0\ \gamma)$

shows $(q, w, \alpha) \rightsquigarrow_* (p, u, \gamma)$

$\langle proof \rangle$

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 $\forall i. \exists p\ \alpha. (q, [], [X]) \rightsquigarrow(i) (p, [], \alpha)$

and $(q, [], X\#\gamma) \rightsquigarrow_* (r, [], \beta)$

shows $\exists Y\ \beta'. \beta = Y\ \#\ \beta' @ \gamma \wedge (q, [], [X]) \rightsquigarrow_* (r, [], Y\#\beta')$

$\langle proof \rangle$

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 $(St\ q, w, map\ Sym\ \alpha) \rightsquigarrow_{s*} (St\ p, u, map\ Sym\ \gamma) \vee$

$(\exists r\ X\ \beta. (St\ q, w, map\ Sym\ \alpha) \rightsquigarrow_{s*} (St\ r, u, Sym\ X\ \#\ map\ Sym\ \beta) \wedge$

$(r, [], X\#\beta) \rightsquigarrow_* (p, [], \gamma) \wedge (\forall i. \exists s\ \Delta. (r, [], [X]) \rightsquigarrow(i) (s, [], \Delta)))$

$\langle proof \rangle$

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* ⇒ '*s* ⇒ *bool* **where**
eps_inf *q* *X* ≡ ∀ *i*. ∃ *p* *α*. (*q*, [], [*X*]) $\rightsquigarrow(i)$ (*p*, [], *α*)

definition *eps_infs* :: '*q* ⇒ '*s* *list* ⇒ *bool* **where**
eps_infs *q* *α* ≡ ∀ *i*. ∃ *p* *γ*. (*q*, [], *α*) $\rightsquigarrow(i)$ (*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*) $\rightsquigarrow^*$ (*p*, [], *α*)
 ⟨*proof*⟩

lemma *some_pair_inf*:
assumes *eps_infs* *q* *α*
and $\bigwedge p X \gamma. (q, [], \alpha) \rightsquigarrow^* (p, [], X\#\gamma) \longrightarrow \neg \text{eps_inf } p X$
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*) $\rightsquigarrow^*$ (*p*, [], *X* # *γ*) ∧ *eps_inf* *p* *X*
 ⟨*proof*⟩

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

lemma *scan_dpda_eps_inf*:
assumes (*q*, *w*, *α*) $\rightsquigarrow^*$ (*p*, *w*, *X* # *γ*)
and *eps_inf* *p* *X*
shows ∃ *Y* *β*. $(St\ q, w, map\ Sym\ \alpha) \rightsquigarrow_s^* (D, [], Sym\ Y\ \# \ \beta)$
 ⟨*proof*⟩

lemma *scan_dpda_neps_infs*:
assumes ¬*eps_infs* *q* *α*
shows ∃ *p* *γ*. $(St\ q, w, map\ Sym\ \alpha) \rightsquigarrow_s^* (St\ p, w, map\ Sym\ \gamma) \wedge (\forall X\ \gamma'. \gamma = X\ \# \ \gamma' \longrightarrow \delta\varepsilon\ M\ p\ X = \{\})$

$\langle proof \rangle$

lemma *scan_dpda_scans_St*:

$\exists p \ X \ \gamma. (St \ q, w, stack_with_X0 \ \alpha) \rightsquigarrow_s^* (p, [], X\#\gamma) \wedge \delta \ scan_dpda \ p \ a \ X \neq \{\}$

$\langle proof \rangle$

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*:

$\exists q \ X \ \alpha. (init_state \ scan_dpda, w, [init_symbol \ scan_dpda]) \rightsquigarrow_s^* (q, [], X\#\alpha) \wedge \delta \ scan_dpda \ q \ a \ X \neq \{\}$

$\langle proof \rangle$

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*

$\langle proof \rangle$

lemma *inj_S1*: *inj S1*

$\langle proof \rangle$

lemma *inj_S2*: *inj S2*

$\langle proof \rangle$

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*: $\exists q \ X \ \alpha. (init_state \ M, w, [init_symbol \ M]) \rightsquigarrow^* (q, [], X\#\alpha) \wedge \delta \ M \ q \ a \ X \neq \{\}$

begin

definition *comp_dpda_init_state* :: 'q s123 **where**

comp_dpda_init_state $\equiv$ if *init_state* *M* $\in$ *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 $\equiv$ range *S3*

fun *comp_dpda_delta* :: 'q s123 $\Rightarrow$ 'a $\Rightarrow$'s $\Rightarrow$ ('q s123 $\times$'s list) **set where**

comp_dpda_delta (*S1* *q*) *a* *X* = ($\lambda(p, \alpha).$ if *p* $\in$ *final_states* *M* then (*S1* *p*, α) else (*S2* *p*, α)) ' δ *M* *q* *a* *X*
| *comp_dpda_delta* (*S3* *q*) *a* *X* = ($\lambda(p, \alpha).$ if *p* $\in$ *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 $\Rightarrow$'s $\Rightarrow$ ('q s123 $\times$'s list) **set where**

comp_dpda_delta_eps (*S1* *q*) *X* = ($\lambda(p, \alpha).$ (*S1* *p*, α)) ' $\delta\epsilon$ *M* *q* *X*
| *comp_dpda_delta_eps* (*S2* *q*) *X* = ($\lambda(p, \alpha).$ if *p* $\in$ *final_states* *M* then (*S1* *p*, α) else (*S2* *p*, α)) ' $\delta\epsilon$ *M* *q* *X*
 $\cup$ (if $\exists a. \delta$ *M* *q* *a* *X* $\neq$ {} 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 ($\exists x. A = \{x\}$) $\longleftrightarrow$ ($\exists x. f$ ' *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.step1* (**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*, $\square$, α) $\rightsquigarrow_c$ (*p*, $\square$, γ)
shows $\exists p'. p = S1\ p'$
 $\langle proof \rangle$

lemma *comp_dpda_steps_from_S1*:
assumes (*S1* *q*, $\square$, α) $\rightsquigarrow_{c*}$ (*p*, $\square$, γ)
shows $\exists p'. p = S1\ p'$
 $\langle proof \rangle$

lemma *comp_dpda_nonfinal_stepsS2*:
assumes (*q*, $\square$, α) $\rightsquigarrow^*$ (*p*, $\square$, γ)
and $\bigwedge r\ \beta. (q, \square, \alpha) \rightsquigarrow^* (r, \square, \beta) \longrightarrow r \notin final_states\ M$
shows (*S2* *q*, $\square$, α) $\rightsquigarrow_{c*}$ (*S2* *p*, $\square$, γ)
 $\langle proof \rangle$

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 $\bigwedge r\ \beta. (q', w, \alpha) \rightsquigarrow^* (r, u, \beta) \longrightarrow r \notin final_states\ M$
and $q = S1\ q' \vee q = S2\ q'$
shows (*q*, *w*, α) $\rightsquigarrow_{c*}$ (*S2* *p*, *u*, γ)
 $\langle proof \rangle$

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)
 $\langle proof \rangle$

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*, $\square$, γ)
and (*q*, *w*, α) $\rightsquigarrow^*$ (*r*, $\square$, β)
shows $r \notin final_states\ M$
 $\langle proof \rangle$

lemma *comp_dpda_steps_nonfinalS2*:
assumes (*S2* *q*, *w*, α) $\rightsquigarrow_{c*}$ (*S3* *p*, $\square$, γ)
and (*q*, *w*, α) $\rightsquigarrow^+$ (*r*, $\square$, β)
shows $r \notin final_states\ M$

$\langle proof \rangle$

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

lemma $lang_comp_dpda$:
 $comp.accept_final = -\ accept_final$
 $\langle proof \rangle$

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 $\exists M' :: ('q\ st_scan\ s123, 'a, 's\ sym_scan)\ pda.$
 $dpda\ M' \wedge pda.accept_final\ M' = -\ pda.accept_final\ M$
 $\langle proof \rangle$

end

References

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