

Two-Way Deterministic Finite Automata

Felipe Escallón and Tobias Nipkow

February 6, 2026

Abstract

This theory presents a proof that two-way DFAs are as powerful as DFAs, i.e. they accept exactly the regular languages. The formalization follows Kozen [1].

```
theory Two_Way_DFA_HF  
  imports Finite_Automata_HF.Finite_Automata_HF  
begin
```

A formalization of two-way deterministic finite automata (2DFA), based on Paulson's theory of DFAs using hereditarily finite sets [2]. Both the definition of 2DFAs and the proof follow Kozen [1].

1 Definition of Two-Way Deterministic Finite Automata

1.1 Basic Definitions

```
type_synonym state = hf
```

```
datatype dir = Left | Right
```

Left and right markers to prevent the 2DFA from reading out of bounds. The input for a 2DFA with alphabet Σ is $\vdash w \dashv$ for some $w \in \Sigma^*$. Note that $\vdash, \dashv \notin \Sigma$:

```
datatype 'a symbol = Letter 'a | Marker dir
```

```
abbreviation left_marker :: 'a symbol ( $\vdash$ ) where  
   $\vdash \equiv$  Marker Left
```

```
abbreviation right_marker :: 'a symbol ( $\dashv$ ) where  
   $\dashv \equiv$  Marker Right
```

```
record 'a dfa2 = states :: state set
```

```

init  :: state
acc   :: state
rej   :: state
nxt   :: state ⇒ 'a symbol ⇒ state × dir

```

2DFA configurations. A 2DFA M is in a configuration (u, q, v) if M is in state q , reading $hd\ v$, the substring $rev\ u$ is to the left and $tl\ v$ is to the right:

type_synonym 'a config = 'a symbol list × state × 'a symbol list

Some abbreviations to guarantee the validity of the input:

abbreviation $\Sigma :: 'a\ list \Rightarrow 'a\ symbol\ list$ **where**
 $\Sigma\ w \equiv map\ Letter\ w$

abbreviation $marker_map :: 'a\ list \Rightarrow 'a\ symbol\ list\ (\langle_ \rangle\ 70)$ **where**
 $\langle w \rangle \equiv \vdash \# (\Sigma\ w) @ [-]$

abbreviation $marker_mapl :: 'a\ list \Rightarrow 'a\ symbol\ list\ (\langle_ \langle \rangle\ 70)$ **where**
 $\langle w \langle \equiv \vdash \# (\Sigma\ w)$

abbreviation $marker_mapr :: 'a\ list \Rightarrow 'a\ symbol\ list\ (\rangle_ \rangle\ 70)$ **where**
 $\rangle w \rangle \equiv (\Sigma\ w) @ [-]$

lemma $mapl_app_mapr_eq_map:$
 $\langle u \langle @ \rangle v \rangle = \langle u @ v \rangle$ **by** *simp*

1.2 Steps and Reachability

```

locale dfa2 =
  fixes M :: 'a dfa2
  assumes init:      init M ∈ states M
  and accept:       acc M ∈ states M
  and reject:       rej M ∈ states M
  and neq_final:    acc M ≠ rej M
  and finite:       finite (states M)
  and nxt:          [[q ∈ states M; nxt M q x = (p, d)] ⇒ p ∈ states M
  and left_nxt:     [[q ∈ states M; nxt M q ⊢ = (p, d)] ⇒ d = Right
  and right_nxt:    [[q ∈ states M; nxt M q ⊣ = (p, d)] ⇒ d = Left
  and final_nxt_r:  [[x ≠ ⊣; q = acc M ∨ q = rej M] ⇒ nxt M q x = (q,
Right)
  and final_nxt_l:  q = acc M ∨ q = rej M ⇒ nxt M q ⊣ = (q, Left)
begin

```

A single

inductive $step :: 'a\ config \Rightarrow 'a\ config \Rightarrow bool$ (**infix** $\langle \rightarrow \rangle$ 55) **where**
 $step_left[intro]:\ nxt\ M\ p\ a = (q, Left) \Longrightarrow (x \# xs, p, a \# ys) \rightarrow (xs, q, x \# a \# ys) |$
 $step_right[intro]:\ nxt\ M\ p\ a = (q, Right) \Longrightarrow (xs, p, a \# ys) \rightarrow (a \# xs, q, ys)$

inductive_cases *step_foldedE*[*elim*]: $a \rightarrow b$
inductive_cases *step_leftE*[*elim*]: $(a\#u, q, v) \rightarrow (u, p, a\#v)$
inductive_cases *step_rightE*[*elim*]: $(u, q, a\#v) \rightarrow (a\#u, p, v)$

The reflexive transitive closure of \rightarrow :

abbreviation *steps* :: 'a config \Rightarrow 'a config \Rightarrow bool (**infix** $\langle \rightarrow^* \rangle$ 55) **where**
steps \equiv *step***

And the nth power of \rightarrow :

abbreviation *stepn* :: 'a config \Rightarrow nat \Rightarrow 'a config \Rightarrow bool ($_ \rightarrow^!(_) _$ 55) **where**
stepn *c* *n* \equiv (*step* $\overset{\sim}{\sim}$ *n*) *c*

lemma *rtranclp_induct3*[*consumes 1, case_names refl step*]:

$\llbracket r^{**} (ax, ay, az) (bx, by, bz); P ax ay az;$
 $\bigwedge u p v x q z.$
 $\llbracket r^{**} (ax, ay, az) (u, p, v); r (u, p, v) (x, q, z); P u p v \rrbracket$
 $\implies P x q z \rrbracket$
 $\implies P bx by bz$

by (*rule rtranclp_induct*[*of* $_ (ax, ay, az) (bx, by, bz), split_rule$])

The initial configuration of M on input word $w \in \Sigma^*$ is $([], \text{init } M, \vdash w \dashv)$. A configuration c is reachable by w if the initial configuration of M on input w reaches c :

abbreviation *reachable* :: 'a list \Rightarrow 'a config \Rightarrow bool (**infix** $\langle \rightarrow^{**} \rangle$ 55) **where**
 $w \rightarrow^{**} c \equiv ([], \text{init } M, \langle w \rangle) \rightarrow^* c$

abbreviation *nreachable* :: 'a list \Rightarrow nat \Rightarrow 'a config \Rightarrow bool ($_ \rightarrow^{*!}(_) _$ 55)
where
nreachable *w* *n* *c* $\equiv ([], \text{init } M, \langle w \rangle) \rightarrow^{(n)} c$

lemma *step_unique*:

$\llbracket c0 \rightarrow c1; c0 \rightarrow c2 \rrbracket \implies c1 = c2$

by *fastforce*

lemma *steps_impl_in_states*:

assumes $p \in \text{states } M$

shows $(u, p, v) \rightarrow^* (u', q, v') \implies q \in \text{states } M$

by (*induction rule: rtranclp_induct3*) (*use assms next in auto*)

corollary *reachable_impl_in_states*:

$w \rightarrow^{**} (u, q, v) \implies q \in \text{states } M$

using *init_steps_impl_in_states* **by** *blast*

1.3 Basic Properties of 2DFAs

The language accepted by M :

definition *Lang* :: 'a list set **where**

$\text{Lang} \equiv \{w. \exists u v. w \rightarrow^{**} (u, \text{acc } M, v)\}$

lemma *unchanged_substrings*:
 $(u, p, v) \rightarrow^* (u', q, v') \implies \text{rev } u @ v = \text{rev } u' @ v'$
proof (*induction rule: rtranclp_induct3*)
case (*step a q' b c q'' d*)
then obtain $p' d'$ **where** $qd_def: \text{nxt } M q' (\text{hd } b) = (p', d')$ **by** *fastforce*
then show *?case*
proof (*cases d'*)
case *Left*
hence $(c, q'', d) = (\text{tl } a, p', \text{hd } a \# b)$
using *step(2) qd_def step.simps* **by** *force*
then show *?thesis*
using *step.IH step.hyps(2)* **by** *fastforce*
next
case *Right*
hence $(c, q'', d) = (\text{hd } b \# a, p', \text{tl } b)$ **using** *step(2) qd_def step.simps* **by**
fastforce
then show *?thesis* **using** *step step.cases* **by** *fastforce*
qed
qed *simp*

lemma *unchanged_final*:
assumes $p = \text{acc } M \vee p = \text{rej } M$
shows $(u, p, v) \rightarrow^* (u', q, v') \implies p = q$
proof (*induction rule: rtranclp_induct3*)
case (*step a q' b c q'' d*)
then show *?case*
by (*smt (verit) assms final_nxt_l final_nxt_r prod.inject step.cases*)
qed *simp*

corollary *unchanged_word*:
 $([], p, w) \rightarrow^* (u, q, v) \implies w = \text{rev } u @ v$
using *unchanged_substrings* **by** *fastforce*

lemma *step_tl_indep*:
assumes $(u, p, w @ v) \rightarrow (y, q, z @ v)$
 $w \neq []$
shows $(u, p, w @ v') \rightarrow (y, q, z @ v')$
using *assms(1)* **proof** *cases*
case (*step_left a x ys*)
with *assms* **obtain** *as* **where** $w = a \# as$ **by** (*meson append_eq_Cons_conv*)
moreover with *step_left* **have** $(u, p, w @ v') \rightarrow (y, q, x \# w @ v')$ **by** *auto*
ultimately show *?thesis* **using** *step_left* **by** *auto*
next
case (*step_right a*)
with *assms* **obtain** *as* **where** $w = a \# as$ **by** (*meson append_eq_Cons_conv*)
then show *?thesis* **using** *step_right* **by** *auto*
qed

lemma *steps_app* [*simp, intro*]:
 $(u, p, v) \rightarrow^* (u', q, v') \implies (u, p, v @ xs) \rightarrow^* (u', q, v' @ xs)$

proof (*induction rule: rtranclp_induct3*)
case (*step w q' x y p' z*)
from *step(2)* **have** $(w, q', x @ xs) \rightarrow (y, p', z @ xs)$ **by** *fastforce*
then show *?case* **using** *step(3)* **by** *simp*
qed *simp*

lemma *left_to_right_impl_substring*:
assumes $(u, p, v) \rightarrow^* (w, q, y)$
 $length\ u \leq length\ w$
obtains *us* **where** $us @ u = w$

using *assms* **proof** (*induction arbitrary: thesis rule: rtranclp_induct3*)
case (*step u' p' v' x q z*)
then consider $(len_u_lt_x)\ length\ u < length\ x \mid (len_u_eq_x)\ length\ u =$
 $length\ x$ **by** *linarith*
then show *?case*
proof *cases*
case *len_u_lt_x*
then have $length\ u \leq length\ u'$ **using** *step* **by** *fastforce*
with *step(3)* **obtain** *us* **where** $us_app_u: us @ u = u'$ **by** *blast*
then show *thesis* **by** (*cases us*) (*use step us_app_u in auto*)
next
case *len_u_eq_x*
with *step(1,2)* *unchanged_substrings* **have** $u = x$
by (*metis rev_rev_ident rev_append[of rev x z] rev_append[of rev u v]*
append_eq_append_conv r_into_rtranclp)
then show *thesis* **using** *step(4)* **by** *simp*
qed
qed *simp*

lemma *acc_impl_reachable_substring*:
assumes $w \rightarrow^{**} (u, acc\ M, v)$
 $xs \neq []$
 $ys \neq []$
shows $v = xs @ ys \implies (u, acc\ M, v) \rightarrow^* (rev\ xs @ u, acc\ M, ys)$

using *assms*
proof (*induction v arbitrary: u xs ys*)
case (*Cons a v*)
consider (*not_right_marker*) *b* **where** $a = Letter\ b \vee a = \vdash \mid (right_marker)$
 $a = \dashv$
by (*metis dir.exhaust symbol.exhaust*)
then show *?case*
proof *cases*
case *not_right_marker*
hence *step*: $(u, acc\ M, a \# v) \rightarrow (a \# u, acc\ M, v)$ **using** *final_nxt_r* **by**
auto
with *Cons(3)* **have** *reach*: $w \rightarrow^{**} (a \# u, acc\ M, v)$ **by** *simp*
from *this* **obtain** *xs'* **where** *xs'_def*: $v = xs' @ ys$

```

    by (metis Cons.prem1,3) append_eq_Cons_conv)
  from xs'_def Cons(2) have a # xs' = xs by simp
  then show ?thesis using Cons Cons(1)[of xs' ys a # u, OF xs'_def reach]
step by fastforce
next
case right_marker
note unchanged = unchanged_word[OF Cons(3)]
have v = []
proof -
  have length u = length (⟨w⟩)
  proof (rule ccontr)
    assume length u ≠ length (⟨w⟩)
    with unchanged obtain n where n_len: n < length (⟨w⟩)
      and n_idx: (rev u @ a # v) ! n = ⊥
      using right_marker
  by (metis append_assoc length_Cons length_append length_append_singleton
length_rev
length_tl linorder_neqE_nat list.sel(3) not_add_less1 nth_append_length)
  have (⟨w⟩) ! n ≠ ⊥
  proof (cases n)
    case 0
    then show ?thesis by simp
  next
  case (Suc k)
  hence n_gt_0: n > 0 by simp
  hence (⟨w⟩) ! n = (⟨w⟩) ! n using n_len
  by (simp add: nth_append_left)
  also have ... = Σ w ! (n - 1) using Suc by simp
  finally show ?thesis
  by (metis n_gt_0 One_nat_def Suc_less_eq Suc_pred length_Cons
length_map n_len nth_map
symbol.distinct(1))
  qed
  with n_idx unchanged show False by argo
  qed
  with unchanged have length (a # v) = Suc 0
  by (metis add_left_cancel length_Cons length_append length_rev list.size(3,4))
  thus ?thesis by simp
  qed
  then show ?thesis using Cons right_marker by (metis append_assoc snoc_eq_iff_butlast)
  qed
qed simp

```

lemma all_geq_left_impl_left_indep:

```

assumes upv_reachable: w →** (u, p, v)
and (u, p, v) →(n) (vs @ u, q, x)
  ∀ i ≤ n. ∀ u' p' v'. ((u, p, v) →(i) (u', p', v')) → length u' ≥ length u
shows ((u', p, v) →(n) (vs @ u', q, x))
  ∧ (∀ i ≤ n. ∀ y p' v'. ((u', p, v) →(i) (y, p', v')) → length y ≥ length u')

```

using *assms(2,3)* **proof** (*induction n arbitrary: vs q x*)
case (*Suc n*)
then obtain *vs' q' x'* **where**
nsteps: (u, p, v) →(n) (vs' @ u, q', x') (vs' @ u, q', x') → (vs @ u, q, x)
proof –
from *Suc(2)* **obtain** *y q' x'* **where** *nstep:*
(u, p, v) →(n) (y, q', x') (y, q', x') → (vs @ u, q, x) **by** *auto*
moreover with *Suc(3)* **have** *y_gt_u: length y ≥ length u* **by** (*meson Suc_leD order_refl*)
ultimately obtain *vs'* **where** *y = vs' @ u* **using** *left_to_right_impl_substring*

by (*metis relpowp_imp_rtranclp*)
then show *thesis* **using** *nstep* **that** **by** *blast*
qed
with *Suc(1)* **have** *u'_stepn: (u', p, v) →(n) (vs' @ u', q', x')*
and *u'_n_geq: ∀ i ≤ n. ∀ y p' v'. ((u', p, v) →(i) (y, p', v'))*
→ length u' ≤ length y
using *Suc.IH Suc.prem(2) le_SucI le_Suc_eq nsteps(1)* **by** *blast+*
moreover from *u'_stepn nsteps(2)* **have** *Sucn_steps: (u', p, v) →(Suc n) (vs*
@ u', q, x) **by** *force*
moreover have *∀ i ≤ Suc n. ∀ y p' v'. ((u', p, v) →(i) (y, p', v')) → length y ≥*
length u'
proof (*(rule allI)+, (rule impI)+*)
fix *i y p' v'*
assume *i_lt_Suc: i ≤ Suc n*
and *stepi: (u', p, v) →(i) (y, p', v')*
then consider (*Suc*) *i = Suc n | (lt_Suc) i < Suc n* **by** *force*
then show *length u' ≤ length y*
proof *cases*
case *Suc*
with *stepi Sucn_steps* **show** *?thesis*
by (*metis leI length_append not_add_less2 prod.inject relpowp_right_unique*
step_unique)
next
case *lt_Suc*
then show *?thesis* **using** *u'_n_geq stepi* **using** *less_Suc_eq_le* **by** *auto*
qed
ultimately show *?case* **by** *auto*
qed *simp*

lemma *reachable_configs_impl_reachable:*

assumes *c0 →* c1*

c0 → c2*

shows *c1 →* c2 ∨ c2 →* c1*

proof –

from *assms* **obtain** *n m* **where** *c0 →(n) c1 c0 →(m) c2*

by (*metis rtranclp_power*)

from *this(1)* **show** *?thesis*

```

proof (induction n arbitrary: c1)
  case 0
  hence c0 = c1 by simp
  then show ?case using assms(2) by auto
next
  case (Suc n)
  then obtain c' where c'_defs: c0 →(n) c' c' → c1 by auto
  with Suc.IH have c' →* c2 ∨ c2 →* c' by simp
  then consider (c'_eq_c2) c' = c2 c' →* c2 |
    (c'_reaches_c2) c' ≠ c2 c' →* c2 |
    (c2_reaches_c') c2 →* c' by blast
  then show ?case
proof cases
  case c'_eq_c2
  then show ?thesis using c'_defs by blast
next
  case c'_reaches_c2
  then obtain c'' where c' → c'' c'' →* c2 by (metis converse_rtranclE)
  then show ?thesis using c'_defs step_unique by metis
next
  case c2_reaches_c'
  then show ?thesis using c'_defs(2) by (meson rtranclp.rtrancl_into_rtrancl)
qed
qed
qed
end

```

2 Boundary Crossings

2.1 Basic Definitions

In order to describe boundary crossings in general, we describe the behavior of M for a fixed, non-empty string x and input word xz , where z is an arbitrary string:

```

locale dfa2_transition = dfa2 +
  fixes x :: 'a list
  assumes x ≠ []
begin

definition x_init :: 'a symbol list where
  x_init ≡ butlast (<x>)

definition x_end :: 'a symbol where
  x_end ≡ last (<x>)

lemmas x_defs = x_init_def x_end_def

```

lemma *x_is_init_app_end*:

$\langle x \rangle = x_init \ @ \ [x_end]$ **unfolding** *x_defs* **by** *simp*

2.1.1 Left steps, right steps, and their reachabilities

A 2DFA is in a left configuration for input xz if it is currently reading x . Otherwise, it is in a right configuration:

definition *left_config* :: 'a config \Rightarrow bool **where**

$left_config \ c \equiv \exists u \ q \ v. \ c = (u, q, v) \wedge length \ u < length \ (\langle x \rangle)$

definition *right_config* :: 'a config \Rightarrow bool **where**

$right_config \ c \equiv \exists u \ q \ v. \ c = (u, q, v) \wedge length \ u \geq length \ (\langle x \rangle)$

lemma *left_config_is_not_right_config*:

$left_config \ c \longleftrightarrow \neg right_config \ c$

unfolding *left_config_def* *right_config_def*

by (*metis linorder_not_less prod.inject prod_cases3*)

lemma *left_config_lt_right_config*:

$\llbracket left_config \ (u, p, v); \ right_config \ (w, q, y) \rrbracket \Longrightarrow length \ u < length \ w$

using *left_config_def* *right_config_def* **by** *simp*

For configurations c_0 and c_1 , a step $c_0 \rightarrow c_1$ is a left step $c_0 \rightarrow^L c_1$ if both c_0 and c_1 are in x :

inductive *left_step* :: 'a config \Rightarrow 'a config \Rightarrow bool (**infix** $\langle \rightarrow^L \rangle$ 55) **where**

lstep [*intro*]: $\llbracket c_0 \rightarrow c_1; \ left_config \ c_0; \ left_config \ c_1 \rrbracket \Longrightarrow c_0 \rightarrow^L c_1$

inductive_cases *lstepE* [*elim*]: $c_0 \rightarrow^L c_1$

abbreviation *left_steps* :: 'a config \Rightarrow 'a config \Rightarrow bool (**infix** $\langle \rightarrow^{L*} \rangle$ 55) **where**

$left_steps \equiv left_step^{**}$

abbreviation *left_stepn* :: 'a config \Rightarrow nat \Rightarrow 'a config \Rightarrow bool ($_ \rightarrow^{L'} (_) _$ 55)

where

$left_stepn \ c \ n \equiv (left_step \ \overset{\sim}{\sim} \ n) \ c$

c is left reachable by a word w if $w \rightarrow^{**} c$ and M does not cross the boundary before reaching c :

abbreviation *left_reachable* :: 'a list \Rightarrow 'a config \Rightarrow bool (**infix** $\langle \rightarrow^{L**} \rangle$ 55)

where

$w \rightarrow^{L**} c \equiv (\llbracket, \ init \ M, \ \langle w \rangle \rrbracket \rightarrow^{L*} c$

abbreviation *left_nreachable* :: 'a list \Rightarrow nat \Rightarrow 'a config \Rightarrow bool ($_ \rightarrow^{L*'} (_) _$

55) **where**

$w \rightarrow^{L*(n)} c \equiv (\llbracket, \ init \ M, \ \langle w \rangle \rrbracket \rightarrow^L(n) \ c$

Right steps are defined analogously:

inductive *right_step* :: 'a config \Rightarrow 'a config \Rightarrow bool (**infix** $\langle \rightarrow^R \rangle$ 55) **where**

rstep [*intro*]: $\llbracket c0 \rightarrow c1; \text{right_config } c0; \text{right_config } c1 \rrbracket \Longrightarrow c0 \rightarrow^R c1$

inductive_cases *rstepE* [*elim*]: $c0 \rightarrow^R c1$

abbreviation *right_steps* :: 'a config \Rightarrow 'a config \Rightarrow bool (**infix** $\langle \rightarrow^{R*} \rangle$ 55) **where**
right_steps \equiv *right_step***

abbreviation *right_stepn* :: 'a config \Rightarrow nat \Rightarrow 'a config \Rightarrow bool ($_ \rightarrow^{R'}(_) _$ 55) **where**

right_stepn c $n \equiv$ (*right_step* $\overset{\sim}{\sim}$ n) c

2.1.2 Properties of left and right steps

lemma *left_steps_impl_steps* [*dest*]:

$c0 \rightarrow^{L*} c1 \Longrightarrow c0 \rightarrow^* c1$

by (*induction rule*: *rtranclp_induct*) *auto*

lemma *right_steps_impl_steps* [*dest*]:

$c0 \rightarrow^{R*} c1 \Longrightarrow c0 \rightarrow^* c1$

by (*induction rule*: *rtranclp_induct*) *auto*

lemma *left_steps_impl_left_config*[*dest*]:

$\llbracket c0 \rightarrow^{L*} c1; \text{left_config } c0 \rrbracket \Longrightarrow \text{left_config } c1$

by (*induction rule*: *rtranclp_induct*) *auto*

lemma *left_steps_impl_left_config_conv*[*dest*]:

$\llbracket c0 \rightarrow^{L*} c1; \text{left_config } c1 \rrbracket \Longrightarrow \text{left_config } c0$

by (*induction rule*: *rtranclp_induct*) *auto*

lemma *left_reachable_impl_left_config*:

$w \rightarrow^{L**} c \Longrightarrow \text{left_config } c$

using *left_config_def left_steps_impl_left_config* **by** *auto*

lemma *right_steps_impl_right_config*[*dest*]:

$\llbracket c0 \rightarrow^{R*} c1; \text{right_config } c0 \rrbracket \Longrightarrow \text{right_config } c1$

by (*induction rule*: *rtranclp_induct*) *auto*

lemma *left_step_tl_indep*:

$\llbracket (u, p, w @ v) \rightarrow^L (y, q, z @ v); w \neq [] \rrbracket \Longrightarrow (u, p, w @ v') \rightarrow^L (y, q, z @ v')$

using *step_tl_indep left_config_is_not_right_config left_config_lt_right_config left_step.cases lstep*

by (*meson order.irrefl*)

lemma *right_stepn_impl_interm_right_stepn*:

assumes $c0 \rightarrow^{R(n)} c2$

$c0 \rightarrow^{(m)} c1$

$m \leq n$

shows $c0 \rightarrow^{R(m)} c1$

proof –

```

from assms(1) obtain f where f_def:
  f 0 = c0
  f n = c2
   $\forall i < n. f\ i \rightarrow^R f\ (Suc\ i)$ 
  by (metis relpowp_fun_conv)
from assms(2) obtain g where g_def:
  g 0 = c0
  g m = c1
   $\forall i < m. g\ i \rightarrow g\ (Suc\ i)$ 
  by (metis relpowp_fun_conv)
have  $i < m \implies g\ i = f\ i$  for i
proof (induction i)
  case 0
  then show ?case using f_def g_def by blast
next
  case (Suc n)
  then have g n = f n by linarith
  with Suc f_def g_def show ?case using right_step.cases step_unique
  by (metis Suc_lessD assms(3) order_less_le_trans)
qed
with f_def g_def have  $\forall i < m. g\ i \rightarrow^R g\ (Suc\ i)$  using right_step.cases step_unique
  by (metis assms(3) order_less_le_trans)
  with g_def show ?thesis by (metis relpowp_fun_conv)
qed

```

These *list* lemmas are necessary for the two following *substring* lemmas:

```

lemma list_deconstruct1:
  assumes  $m \leq \text{length}\ xs$ 
  obtains ys zs where  $\text{length}\ ys = m$   $ys\ @\ zs = xs$  using assms
by (metis append_take_drop_id dual_order.eq_iff length_take min_def)

```

```

lemma list_deconstruct2:
  assumes  $m \leq \text{length}\ xs$ 
  obtains ys zs where  $\text{length}\ zs = m$   $ys\ @\ zs = xs$ 
proof –
  from assms have  $m \leq \text{length}\ (\text{rev}\ xs)$  by simp
  then obtain ys zs where  $\text{length}\ ys = m$   $ys\ @\ zs = \text{rev}\ xs$ 
  using list_deconstruct1 by blast
  then show thesis using list_deconstruct1 that by (auto simp: append_eq_rev_conv)
qed

```

```

lemma lstar_impl_substring_x:
  assumes app_eq:  $\text{rev}\ u\ @\ v = \langle x\ @\ z \rangle$ 
  and in_x:  $\text{length}\ u < \text{length}\ (\langle x \rangle)$ 
  and lsteps:  $(u, p, v) \rightarrow^{L*} (u', q, v')$ 
  obtains y where  $\text{rev}\ u'\ @\ y = \langle x\ \langle y\ @\ \rangle z \rangle = v'$ 
proof –
  have leftconfig: left_config (u, p, v) unfolding left_config_def using in_x by
blast

```

hence $u' \text{ lt } x$: $\text{length } u' < \text{length } (\langle x \rangle)$ **using** $\text{lsteps left_config_def}$ **by force**
from lsteps **show** thesis
proof (*induction arbitrary: u p v rule: rtranclp_induct3*)
 case refl
from $\text{unchanged_word app_eq lsteps}$ **have** $\text{app: } \langle x @ z \rangle = \text{rev } u' @ v'$
 by (*metis left_steps_impl_steps unchanged_substrings*)
moreover with $u' \text{ lt } x$
obtain y **where** $\text{rev } u' @ y = \langle x \langle y @ \rangle z \rangle = v'$
proof –
from $u' \text{ lt } x \text{ list_deconstruct1}$
obtain $xs \ ys$ **where** $\text{length } xs = \text{length } u'$ **and** $xapp: xs @ ys = \langle x \langle$
 using $\text{Nat.less_imp_le_nat}$ **by** metis
moreover from this **have** $\text{length } (ys @ \rangle z) = \text{length } v'$ **using** app
 by (*smt (verit) append_assoc append_eq_append_conv length_rev*
mapl_app_mapr_eq_map)
ultimately have $xs \text{ is_rev: } xs = \text{rev } u'$
by (*metis (no_types) app append_Cons append_assoc append_eq_append_conv*
map_append)
then have $ys @ \rangle z = v'$ **using** $xapp \text{ app}$
 by (*metis (no_types) append_assoc same_append_eq[of xs v' ys @ \Sigma z @*
 [-]] *mapl_app_mapr_eq_map*)
thus thesis **using** $\text{that } xs \text{ is_rev } xapp$ **by** presburger
qed
ultimately show $?case$ **using** that by simp
qed blast
qed

corollary $\text{left_reachable_impl_substring_x}$:
assumes $x @ z \rightarrow^{L**} (u, q, v)$
obtains y **where** $\text{rev } u @ y = \langle x \langle y @ \rangle z \rangle = v$
using $\text{lstar_impl_substring_x}$ assms left_config_def $\text{left_reachable_impl_left_config}$
by blast

corollary $\text{reachable_lconfig_impl_substring_x}$:
assumes $x @ z \rightarrow^{**} (u, p, v)$
and $\text{length } u < \text{length } (\langle x \rangle)$
and $(u, p, v) \rightarrow^{L*} (u', q, v')$
obtains y **where** $\text{rev } u' @ y = \langle x \langle y @ \rangle z \rangle = v'$
using unchanged_word [*OF assms(1)*] $\text{lstar_impl_substring_x}$ assms **by** metis

lemma $\text{star_rconfig_impl_substring_z}$:
assumes $\text{app_eq: } x @ z \rightarrow^{**} (u, p, v)$
and $\text{reach: } (u, p, v) \rightarrow^* (u', q, v')$
and $\text{rconf: right_config } (u', q, v')$
obtains y **where** $\text{rev } (\langle x @ y \rangle = u' y @ v' = \rangle z)$
proof –
from right_config_def **have** $u' \text{ ge } x$: $\text{length } (\langle x \rangle) \leq \text{length } u'$
using rconf **by force**
from reach **show** thesis

proof (*induction arbitrary: u p v rule: rtranclp_induct3*)
case *refl*
from *unchanged_word app_eq* **have** $\langle x @ z \rangle = \text{rev } u' @ v'$
by (*metis reach unchanged_substrings*)
moreover with *u'_ge_x*
obtain x' **where** $\text{rev } (\langle x' @ x' \rangle = u' x' @ v' = \rangle z)$
proof –
have $\text{length } v' \leq \text{length } (\rangle z)$
proof (*rule ccontr*)
assume $\neg ?thesis$
hence $\text{length } v' > \text{length } (\rangle z)$ **by** *simp*
with *u'_ge_x*
have $\text{length } (\text{rev } u' @ v') > \text{length } (\langle x @ z \rangle)$ **by** *simp*
thus *False* **using** *app* **by** (*metis nat_less_le*)
qed
from *list_deconstruct2* [*OF this*]
obtain $xs\ ys$ **where** $\text{length } ys = \text{length } v'$ **and** $zapp: xs @ ys = \rangle z$
by *metis*
moreover from *this* **have** $\text{length } (\langle x' @ xs \rangle = \text{length } u')$ **using** *app*
by (*metis (no_types, lifting) append_assoc append_eq_append_conv length_rev*
mapl_app_mapr_eq_map)
ultimately have ys_is_v' : $ys = v'$
by (*metis app append_assoc append_eq_append_conv mapl_app_mapr_eq_map*)
then have $x_app_xs_eq_rev_u'$: $\langle x' @ xs \rangle = \text{rev } u'$ **using** *zapp app*
by (*metis (no_types, lifting) append_assoc append_eq_append_conv*
mapl_app_mapr_eq_map)
hence $\text{rev } (\langle x' @ xs \rangle = u')$ **by** *simp*
thus *thesis* **using** ys_is_v' *zapp* **that** **by** *presburger*
qed
ultimately show *?case* **using** *that* **by** *simp*
qed *blast*
qed

corollary *reachable_right_conf_impl_substring_z*:
assumes $x @ z \rightarrow^{**} (u, q, v)$
right_config (u, q, v)
obtains y **where** $\text{rev } (\langle x' @ y \rangle = u y @ v = \rangle z)$
using *assms star_rconfig_impl_substring_z right_config_def* **by** *blast*

lemma *lsteps_indep*:
assumes $(u, p, v @ \rangle z) \rightarrow^{L*} (w, q, y @ \rangle z)$
 $\text{rev } u @ v @ \rangle z = \langle x @ z \rangle$
shows $(u, p, v @ \rangle z') \rightarrow^{L*} (w, q, y @ \rangle z')$
proof –
from *assms* **obtain** n **where** $nsteps: (u, p, v @ \rangle z) \rightarrow^L(n) (w, q, y @ \rangle z)$
using *rtranclp_power* **by** *meson*
then show *?thesis* **using** *assms(2)*
proof (*induction n arbitrary: w q y*)
case (*Suc n*)

obtain $w' q' y'$ **where** $lstepn: (u, p, v @ \rangle z) \rightarrow^L(n) (w', q', y' @ \rangle z)$
and $lstep: (w', q', y' @ \rangle z) \rightarrow^L (w, q, y @ \rangle z)$

proof –

from *Suc.prem*s **obtain** $w' q' y'$ **where** $lstepn: (u, p, v @ \rangle z) \rightarrow^L(n) (w', q', y')$

and $lstep: (w', q', y') \rightarrow^L (w, q, y @ \rangle z)$ **by** *auto*

hence $w'_left: left_config (w', q', y')$ **by** *blast*

then obtain xs **where** $xs @ \rangle z = y'$

proof –

from $lstepn$ **have** $(u, p, v @ \rangle z) \rightarrow^{L*} (w', q', y')$
by (*simp add: relpowp_imp_rtranclp*)
moreover have $length\ u < length\ (\langle x \rangle)$
by (*meson calculation left_config_lt_right_config left_steps_impl_left_config_conv linorder_le_less_linear order_less_imp_not_eq2 right_config_def*)

w'_left

ultimately show thesis using *Suc.prem*s(2) **that by** (*meson lstar_impl_substring_x*)

qed

with $lstepn$ $lstep$ **show thesis using that by** *auto*

qed

with *Suc.IH* **have** $(u, p, v @ \rangle z) \rightarrow^{L*} (w', q', y' @ \rangle z)$
by (*simp add: Suc.prem*s(2))

moreover have $(w', q', y' @ \rangle z) \rightarrow^L (w, q, y @ \rangle z)$

proof –

have $y'_not_empty: y' \neq []$

proof

assume $y' = []$

hence $(u, p, v @ \rangle z) \rightarrow^{L*} (w', q', \rangle z)$ **using** *calculation by auto*

moreover have $left_config (u, p, v @ \rangle z)$
by (*meson Suc.prem*s(1) $\langle (w', q', y' @ \Sigma z @ [-]) \rightarrow^L (w, q, y @ \Sigma z @ [-]) \rangle$)

$[-])$

$dfa2_transition.left_steps_impl_left_config_conv\ dfa2_transition_axioms$

$left_step.cases$

$relpowp_imp_rtranclp$

ultimately have $lc: left_config (w', q', \rangle z)$
using $left_config_is_not_right_config\ left_config_lt_right_config$ **by**

$blast$

have $rev\ u\ @\ v\ @\ \rangle z = \langle x @ z \rangle$
using *assms*(2) **by** *force*

hence $rev\ w' @ \rangle z = \dots$
by (*metis* $\langle (u, p, v @ \Sigma z' @ [-]) \rightarrow^{L*} (w', q', y' @ \Sigma z' @ [-]) \rangle \langle y' = [] \rangle$)

$left_steps_impl_steps$

$self_append_conv2\ unchanged_substrings$

hence $length\ w' = length\ (\langle x \rangle)$ **by** *simp*

with lc **show** *False* **using** $left_config_def$ **by** *simp*

qed

with $left_step_tl_indep[OF\ lstep]$ **show** *?thesis* **by** *simp*

qed

ultimately show *?case* **by** *simp*

qed *simp*

qed

lemma *left_reachable_indep*:

assumes $x @ y \rightarrow^{L**} (u, q, v @ \rangle y)$

shows $x @ z \rightarrow^{L**} (u, q, v @ \rangle z)$

proof –

from *assms* obtain n where $([], \text{init } M, \langle x @ y \rangle) \rightarrow^L(n) (u, q, v @ \rangle y)$

by (*meson rtranclp_power*)

hence $([], \text{init } M, \langle x @ z \rangle) \rightarrow^L(n) (u, q, v @ \rangle z)$

proof (*induction n arbitrary: u q v*)

case (*Suc n*)

from *Suc(2)* obtain $u' p v'$

where *stepn*: $x @ y \rightarrow^{L*(n)} (u', p, v' @ \rangle y)$

and $(u', p, v' @ \rangle y) \rightarrow^L(1) (u, q, v @ \rangle y)$

proof –

from *Suc(2)* obtain $u' p v''$

where $x @ y \rightarrow^{L*(n)} (u', p, v'' @ \rangle y)$

$(u', p, v'' @ \rangle y) \rightarrow^L(1) (u, q, v @ \rangle y)$ by *auto*

moreover with *left_reachable_impl_substring_x* obtain v' where $v'' = v'$

$@ \rangle y)$

using *rtranclp_power* by *metis*

ultimately show *thesis* using *that* by *blast*

qed

from *this* have *y_lstep*: $(u', p, v' @ \rangle y) \rightarrow^L (u, q, v @ \rangle y)$

by *fastforce*

hence $(u', p, v' @ \rangle z) \rightarrow^L (u, q, v @ \rangle z)$

proof –

from *y_lstep* have *left_configs*:

left_config $(u', p, v' @ \rangle y)$

left_config $(u, q, v @ \rangle y)$ by *blast+*

hence *left_config* $(u', p, v' @ \rangle z)$

left_config $(u, q, v @ \rangle z)$

unfolding *left_config_def* by *auto*

moreover have $(u', p, v' @ \rangle z) \rightarrow (u, q, v @ \rangle z)$

proof –

from *y_lstep* have *y_step*: $(u', p, v' @ \rangle y) \rightarrow (u, q, v @ \rangle y)$ by *blast*

obtain c vs where *v'_def*: $v' = c \# vs$

proof –

from *unchanged_word* have $\text{rev } u' @ v' @ \rangle y = \langle x @ y \rangle$

by (*metis left_steps_impl_steps relpowp_imp_rtranclp stepn*)

hence $\text{rev } u' _ \text{app } v'$: $\text{rev } u' @ v' = \langle x \rangle$ by *simp*

have $v' \neq []$

by (*rule ccontr*) (*use rev_u'_app_v' left_config_def left_configs in auto*)

thus *thesis* using *that list.exhaust* by *blast*

qed

with *y_step* have $(u', p, c \# vs @ \rangle y) \rightarrow (u, q, v @ \rangle y)$ by *simp*

hence $(u', p, c \# vs @ \rangle z) \rightarrow (u, q, v @ \rangle z)$ by *fastforce*

with *v'_def* show *?thesis* by *simp*

qed

```

    ultimately show ?thesis by blast
qed
moreover from Suc(1)[OF stepn] have x @ z →L*(n) (u', p, v' @ >z) .
ultimately show ?case by auto
qed simp
then show ?thesis by (meson rtranclp_power)
qed

```

2.2 A Formal Definition of Boundary Crossings

$c_0 \rightarrow^X(n) c_1$ if c_0 reaches c_1 crossing the boundary n times:

```

inductive crossn :: 'a config ⇒ nat ⇒ 'a config ⇒ bool (⟦_ →X'(⟦_ 55) where
  no_crossl: [left_config c0; c0 →L* c1] ⇒ c0 →X(0) c1 |
  no_crossr: [right_config c0; c0 →R* c1] ⇒ c0 →X(0) c1 |
  crossn_rtol: [c0 →X(n) (rev (⟦x⟧, p, >z));
    (rev (⟦x⟧, p, >z)) → (rev x_init, q, x_end # >z);
    (rev x_init, q, x_end # >z) →L* c1] ⇒ c0 →X(Suc n) c1 |
  crossn_ltor: [c0 →X(n) (rev x_init, p, x_end # >z);
    (rev x_init, p, x_end # >z) → (rev (⟦x⟧, q, >z));
    (rev (⟦x⟧, q, >z)) →R* c1] ⇒ c0 →X(Suc n) c1

```

```

declare crossn.intros[intro]

```

```

inductive_cases no_crossE[elim]: c0 →X(0) c1

```

```

inductive_cases crossE[elim]: c0 →X(Suc n) c1

```

```

abbreviation word_crossn :: 'a list ⇒ nat ⇒ 'a config ⇒ bool (⟦_ →X*'(⟦_ 55) where

```

```

  word_crossn w n c ≡ ([, init M, <w>) →X(n) c

```

```

lemma self_nocross[simp]:

```

```

  c →X(0) c using left_config_is_not_right_config by blast

```

```

lemma no_cross_impl_same_side:

```

```

  c0 →X(0) c1 ⇒ left_config c0 = left_config c1

```

```

using left_config_is_not_right_config by blast

```

```

lemma left_config_impl_rtol_cross:

```

```

  assumes c0 →X(Suc n) c1

```

```

    left_config c1

```

```

  obtains p q z where c0 →X(n) (rev (⟦x⟧, p, >z))

```

```

    (rev (⟦x⟧, p, >z)) → (rev x_init, q, x_end # >z))

```

```

    (rev x_init, q, x_end # >z) →L* c1

```

```

using assms(1) proof cases

```

```

  case (crossn_rtol p z q)

```

```

  then show ?thesis using that by blast

```

```

next

```

```

  case (crossn_ltor p z q)

```

```

  from crossn_ltor(3) have right_config c1

```

using *right_config_def right_steps_impl_right_config* **by** *auto*
then show *?thesis using assms left_config_is_not_right_config* **by** *auto*
qed

lemma *right_config_impl_ltor_cross*:

assumes $c0 \rightarrow^X (Suc\ n)\ c1$

right_config $c1$

obtains $p\ q\ z$ **where** $c0 \rightarrow^X (n)\ (rev\ x_init,\ p,\ x_end\ \#\ \rangle z)$
 $(rev\ x_init,\ p,\ x_end\ \#\ \rangle z) \rightarrow (rev\ (\langle x(),\ q,\ \rangle z))$
 $(rev\ (\langle x(),\ q,\ \rangle z)) \rightarrow^{R*}\ c1$

using *assms(1)* **proof** *cases*

case $(crossn_rtol\ p\ z\ q)$

from $crossn_rtol(3)$ **have** *left_config* $c1$

using *left_config_def left_steps_impl_left_config*

by $(simp\ add:\ x_is_init_app_end)$

then show *?thesis using assms left_config_is_not_right_config* **by** *auto*

next

case $(crossn_ltor\ p\ z\ q)$

then show *?thesis using that* **by** *blast*

qed

lemma *crossn_decompose*:

assumes $c0 \rightarrow^X (Suc\ n)\ c2$

obtains $c1$ **where** $c0 \rightarrow^X (n)\ c1\ c1 \rightarrow^X (Suc\ 0)\ c2$

using *assms* **proof** *cases*

case $(crossn_rtol\ p\ z\ q)$

moreover have $(rev\ (\langle x(),\ p,\ \rangle z)) \rightarrow^X (Suc\ 0)\ c2$

by $(rule\ crossn.intros(3)[OF\ self_nocross])$

$(use\ crossn_rtol\ in\ auto)$

ultimately show *?thesis using that* **by** *blast*

next

case $(crossn_ltor\ p\ z\ q)$

moreover have $(rev\ x_init,\ p,\ x_end\ \#\ \rangle z) \rightarrow^X (Suc\ 0)\ c2$

by $(rule\ crossn.intros(4)[OF\ self_nocross])$

$(use\ crossn_ltor\ in\ auto)$

ultimately show *?thesis using that* **by** *blast*

qed

lemma *step_impl_crossn*:

assumes $c0 \rightarrow c1$

$c0 = (u,\ p,\ v)$

$rev\ u\ @\ v = \langle x\ @\ z \rangle$

shows $(c0 \rightarrow^X (0)\ c1) \vee (c0 \rightarrow^X (Suc\ 0)\ c1)$

proof $(cases\ left_config\ c0 = left_config\ c1)$

case *True*

consider $left_config\ c0 \mid right_config\ c0$ **using** *left_config_is_not_right_config*

by *blast*

then show *?thesis*

by *cases*

```

    ((use assms True in auto),
     (simp add: left_config_is_not_right_config no_crossr r_into_rtranclp rstep))
next
case False
consider (left) left_config c0 | (right) right_config c0
  using left_config_is_not_right_config by blast
then show ?thesis
proof cases
case left
  with False obtain q where c0 = (rev x_init, p, x_end # }z)
    c1 = (rev (<x>, q, }z))

  proof -
    obtain y q w where c1_def: c1 = (y, q, w) using prod_cases3 by blast
    have length u = length (<x> - 1 length y = length (<x>)
    proof -
      from left assms(2) have length u < length (<x>) using left_config_def by
auto
      moreover from left False right_config_def c1_def have length y ≥ length
(<x>)
        using left_config_is_not_right_config by simp
      moreover from assms(1,2) c1_def have length u = Suc (length y) ∨
length y = Suc (length u)
        by fastforce
      ultimately show length u = length (<x> - 1 length y = length (<x>)
        by force+
    qed
    with assms(2,3) have u = rev x_init
      by (smt (verit, ccfv_threshold) append_assoc append_eq_append_conv
mapl_app_mapr_eq_map
      length_butlast length_rev rev_swap x_init_def x_is_init_app_end)
    from this have v = x_end # }z)
      by (smt (verit) append_assoc append_eq_append_conv append_eq_rev_conv
assms(3) mapl_app_mapr_eq_map
      rev_simps(2) rev_rev_ident rev_singleton_conv x_is_init_app_end)
    have y = rev (<x>)
      using <length y = length (⊢ # Σ x) > <u = rev x_init> <v = x_end # Σ z
@ [-]> assms(1,2) c1_def
      x_is_init_app_end by auto
    moreover from this have w = }z)
      using <length u = length (⊢ # Σ x) - 1> <v = x_end # Σ z @ [-]>
assms(1,2) c1_def by auto
    ultimately have c1 = (rev (<x>, q, }z)) using c1_def by simp
    moreover have c0 = (rev x_init, p, x_end # }z)
      by (simp add: <u = rev x_init> <v = x_end # Σ z @ [-]> assms(2))
    ultimately show thesis using that by blast
  qed
then show ?thesis
  using assms(1) dfa2_transition.self_nocross dfa2_transition_axioms by blast
next

```

case *right*
with *False* **obtain** *q* **where** $c0 = (\text{rev } (\langle x \rangle), p, \rangle z)$
 $c1 = (\text{rev } x_init, q, x_end \# \rangle z)$
proof –
obtain *y q w* **where** $c1_def: c1 = (y, q, w)$ **using** *prod_cases3* **by** *blast*
have $\text{length } u = \text{length } (\langle x \rangle)$ $\text{length } y = \text{length } (\langle x \rangle) - 1$
proof –
from *right_assms(2)* **have** $\text{length } u \geq \text{length } (\langle x \rangle)$ **using** *right_config_def*
by *auto*
moreover from *right_False_left_config_def c1_def* **have** $\text{length } y < \text{length } (\langle x \rangle)$
using *left_config_is_not_right_config* **by** *blast*
moreover from *assms(1,2) c1_def* **have** $\text{length } u = \text{Suc } (\text{length } y) \vee$
 $\text{length } y = \text{Suc } (\text{length } u)$
by *fastforce*
ultimately show $\text{length } u = \text{length } (\langle x \rangle)$ $\text{length } y = \text{length } (\langle x \rangle) - 1$
by *force+*
qed
with *assms(2,3)* **have** $u = \text{rev } (\langle x \rangle)$
by (*smt (verit, ccfv_threshold) append_assoc append_eq_append_conv*
mapl_app_mapr_eq_map
 $\text{length_butlast length_rev rev_swap } x_init_def\ x_is_init_app_end$)
from this **have** $v = \rangle z$
by (*smt (verit) append_assoc append_eq_append_conv append_eq_rev_conv*
assms(3) mapl_app_mapr_eq_map
 $\text{rev.simps(2) rev_rev_ident rev_singleton_conv } x_is_init_app_end$)
have $y = \text{rev } x_init$
using $\langle \text{length } y = \text{length } (\vdash \# \Sigma x) - 1 \rangle$ $\langle u = \text{rev } (\langle x \rangle) \rangle$ $\langle v = \Sigma z @ [-] \rangle$
assms(1,2) c1_def
 $x_is_init_app_end$ **by** *auto*
moreover from this **have** $w = x_end \# \rangle z$
by (*smt (verit) \langle \text{length } u = \text{length } (\vdash \# \Sigma x) \rangle \langle \text{length } y = \text{length } (\vdash \# \Sigma x) - 1 \rangle*
 $\langle u = \text{rev } (\vdash \# \Sigma x) \rangle$ $\langle v = \Sigma z @ [-] \rangle$ *assms(1,2) c1_def diff_le_self*
impossible_Cons last_snoc
 $\text{prod.inject rev_eq_Cons_iff step_foldedE } x_end_def$)
ultimately have $c1 = (\text{rev } x_init, q, x_end \# \rangle z)$ **using** $c1_def$ **by** *simp*
moreover have $c0 = (\text{rev } (\langle x \rangle), p, \rangle z)$
by (*simp add: \langle u = \text{rev } (\langle x \rangle) \rangle \langle v = \Sigma z @ [-] \rangle* *assms(2)*)
ultimately show *thesis* **using** *that* **by** *blast*
qed
then show *?thesis*
using *assms(1) dfa2_transition.self_nocross dfa2_transition_axioms* **by** *blast*
qed
qed

lemma *crossn_no_cross_eq_crossn*:

assumes $c0 \rightarrow^X(n) c1$
 $c1 \rightarrow^X(0) c2$

```

shows  $c0 \rightarrow^X(n) c2$ 
using assms(1) proof cases
  case no_crossl
    then show ?thesis using left_steps_impl_left_config
      assms left_config_is_not_right_config by (meson crossn.no_crossl no_crossE
rtranclp_trans)
  next
    case no_crossr
      then show ?thesis using right_steps_impl_right_config
        assms left_config_is_not_right_config by (meson crossn.no_crossr no_crossE
rtranclp_trans)
  next
    case (crossn_rtol n p z q)
      from crossn_rtol(4) have left_config c1 using left_steps_impl_left_config
left_config_def
        x_is_init_app_end by auto
      with assms(2) have  $c1 \rightarrow^{L*} c2$  using left_config_is_not_right_config by blast
      then show ?thesis using crossn_rtol by auto
  next
    case (crossn_ltor n p z q)
      from crossn_ltor(4) have right_config c1 using right_steps_impl_right_config
right_config_def
        x_is_init_app_end by auto
      with assms(2) have  $c1 \rightarrow^{R*} c2$  using left_config_is_not_right_config by blast
      then show ?thesis using crossn_ltor by auto
qed

lemma crossn_trans:
  assumes  $c0 \rightarrow^X(n) c1$ 
  shows  $c1 \rightarrow^X(m) c2 \implies c0 \rightarrow^X(n+m) c2$ 
proof (induction m arbitrary: c2)
  case 0
    from assms show ?case
  proof cases
    case no_crossl
      then show ?thesis
      by (metis 0.prems(1) add_0 crossn.no_crossl left_config_is_not_right_config
no_crossE
      no_cross_impl_same_side rtranclp_trans)
    next
      case no_crossr
        then show ?thesis
        by (metis 0.prems(1) add_0_right crossn.no_crossr left_config_is_not_right_config
no_crossE
        right_steps_impl_right_config rtranclp_trans)
    next
      case (crossn_rtol n p z q)
        then show ?thesis
        by (smt (verit, best) 0.prems(1) Nat.add_0_right crossn.crossn_rtol left_config_def)

```

```

    left_config_is_not_right_config left_steps_impl_left_config length_append_singleton
length_rev
    lessI no_crossE rtranclp_trans x_is_init_app_end)
next
  case (crossn_ltor n p z q)
  from crossn_ltor(4) have right_config c1 using right_steps_impl_right_config

  by (simp add: right_config_def)
  with 0(1) have c1  $\rightarrow^{R*}$  c2 using left_config_is_not_right_config by blast
  with crossn_ltor(4) have (rev ( $\langle x \rangle$ , q,  $\rangle z$ ))  $\rightarrow^{R*}$  c2 by simp
  with crossn_ltor(1,2,3) show ?thesis by auto
qed
next
  case (Suc m)
  from Suc(2) crossn_decompose obtain c' where c1  $\rightarrow^X(m)$  c'
    and c'_cross: c'  $\rightarrow^X(Suc\ 0)$  c2 by blast
  with Suc(1) have c0_nm_cross: c0  $\rightarrow^X(n+m)$  c' by blast
  from c'_cross show ?case
  proof cases
    case (crossn_rtol r w s)
    moreover with c0_nm_cross crossn_no_cross_eq_crossn have
      c0  $\rightarrow^X(n+m)$  (rev ( $\langle x \rangle$ , r,  $\rangle w$ )) by blast
    ultimately show ?thesis by auto
  next
    case (crossn_ltor r w s)
    moreover with c0_nm_cross crossn_no_cross_eq_crossn have
      c0  $\rightarrow^X(n+m)$  (rev x_init, r, x_end #  $\rangle w$ ) by blast
    ultimately show ?thesis by auto
  qed
qed

lemma crossn_impl_reachable:
  assumes c0  $\rightarrow^X(n)$  c1
  shows c0  $\rightarrow^*$  c1
using assms by induction auto

lemma reachable_xz_impl_crossn:
  assumes c0  $\rightarrow^*$  c1
    c0 = (u, p, v)
    rev u @ v =  $\langle x @ z \rangle$ 
  obtains n where c0  $\rightarrow^X(n)$  c1
using assms proof (induction arbitrary: u p v thesis)
  case base
  then show ?case using self_nocross by blast
next
  case (step c1 c2)
  then obtain n where ncross: c0  $\rightarrow^X(n)$  c1 by blast
  obtain w q y where c1 = (w, q, y) using prod_cases3 by blast
  moreover from this have rev w @ y =  $\langle x @ z \rangle$ 

```

using *unchanged_substrings step(1,5,6)* **by** *simp*
ultimately obtain *m* **where** *c1* $\rightarrow^X(m)$ *c2* **using** *step(2) step_impl_crossn*
by *metis*
then show *?case* **using** *ncross crossn_trans step(4)* **by** *blast*
qed

2.3 The Transition Relation T_x

$T_x p q$ for a non-empty string x describes the behavior of a 2DFA M when it crosses the boundary between x and any string z for the input string xz . Intuitively, $T_x (Some p) (Some q)$ if whenever M enters x from the right in state p , when it re-enters z in the future, it will do so in state q . $T_x None (Some q)$ denotes the state in which M first enters z , while $T_x (Some p) None$ denotes that if M ever enters x in state p , it will never enter z in the future, and therefore does not terminate.

inductive $T :: state option \Rightarrow state option \Rightarrow bool$ **where**

$init_tr: \llbracket x @ z \rightarrow^{L**} (rev\ x_init, p, x_end \# \rangle z);$
 $(rev\ x_init, p, x_end \# \rangle z) \rightarrow (rev\ (\langle x \rangle, q, \rangle z)) \rrbracket \Longrightarrow T\ None\ (Some\ q) \mid$

$init_no_tr: \nexists q\ z. x @ z \rightarrow^{L**} (rev\ (\langle x \rangle, q, \rangle z)) \Longrightarrow T\ None\ None \mid$

$some_tr: \llbracket p' \in states\ M; (rev\ (\langle x \rangle, p', \rangle z) \rightarrow (rev\ x_init, p, x_end \# \rangle z));$
 $(rev\ x_init, p, x_end \# \rangle z) \rightarrow^{L*} (rev\ x_init, q', x_end \# \rangle z);$
 $(rev\ x_init, q', x_end \# \rangle z) \rightarrow (rev\ (\langle x \rangle, q, \rangle z)) \rrbracket \Longrightarrow T\ (Some\ p)$
 $(Some\ q) \mid$

$no_tr: \llbracket p' \in states\ M; (rev\ (\langle x \rangle, p', \rangle z) \rightarrow (rev\ x_init, p, x_end \# \rangle z));$
 $\nexists q' q'' z. (rev\ x_init, p, x_end \# \rangle z) \rightarrow^{L*} (rev\ x_init, q', x_end \# \rangle z)$
 $\rangle z) \wedge$
 $(rev\ x_init, q', x_end \# \rangle z) \rightarrow (rev\ (\langle x \rangle, q'', \rangle z)) \rrbracket \Longrightarrow T\ (Some\ p)$
 $None$

declare $T.intros[intro]$

inductive_cases $init_trNoneE[elim]: T\ None\ None$

inductive_cases $init_trSomeE[elim]: T\ None\ (Some\ q)$

inductive_cases $no_trE[elim]: T\ (Some\ q)\ None$

inductive_cases $some_trE[elim]: T\ (Some\ q)\ (Some\ p)$

Lemmas for the independence of T_x from z . This is a fundamental property to show the main theorem:

lemma $init_tr_indep:$

assumes $T\ None\ (Some\ q)$

obtains p **where** $x @ z \rightarrow^{L**} (rev\ x_init, p, x_end \# \rangle z)$

$(rev\ x_init, p, x_end \# \rangle z) \rightarrow (rev\ (\langle x \rangle, q, \rangle z))$

proof –

from *assms* **obtain** $p \ z'$ **where** *prems*: $x \ @ \ z' \rightarrow^{L**} (\text{rev } x_init, p, x_end \# \rangle z')$
 $(\text{rev } x_init, p, x_end \# \rangle z') \rightarrow (\text{rev } (\langle x \rangle, q, \rangle z')$
by *auto*
with *left_reachable_indep*[of $_ _ _ [x_end]$] **have** $x \ @ \ z \rightarrow^{L**} (\text{rev } x_init, p, x_end \# \rangle z)$
by *auto*
moreover from *prems*(2) **have** $(\text{rev } x_init, p, x_end \# \rangle z) \rightarrow (\text{rev } (\langle x \rangle, q, \rangle z)$
by *fastforce*
ultimately show *thesis* **using** *that* **by** *simp*
qed

lemma *init_no_tr_indep*:
 $T \ \text{None} \ \text{None} \Longrightarrow \nexists q. x \ @ \ z \rightarrow^{**} (\text{rev } (\langle x \rangle, q, \rangle z)$
by *auto*

lemma *some_tr_indep*:
assumes $T \ (\text{Some } p) \ (\text{Some } q)$
obtains q' **where** $(\text{rev } x_init, p, x_end \# \rangle z) \rightarrow^{L*} (\text{rev } x_init, q', x_end \# \rangle z)$
 $(\text{rev } x_init, q', x_end \# \rangle z) \rightarrow (\text{rev } (\langle x \rangle, q, \rangle z)$

proof –
from *assms* **obtain** $p' \ q' \ z'$ **where** *prems*:
 $(\text{rev } (\langle x \rangle, p', \rangle z') \rightarrow (\text{rev } x_init, p, x_end \# \rangle z')$
 $(\text{rev } x_init, p, x_end \# \rangle z') \rightarrow^{L*} (\text{rev } x_init, q', x_end \# \rangle z')$
 $(\text{rev } x_init, q', x_end \# \rangle z') \rightarrow (\text{rev } (\langle x \rangle, q, \rangle z')$ **by** *auto*
with *lsteps_indep*[of $\text{rev } x_init \ p \ [x_end] \ z' \ \text{rev } x_init \ q' \ [x_end] \ z]$ **have**
 $(\text{rev } x_init, p, x_end \# \rangle z) \rightarrow^{L*} (\text{rev } x_init, q', x_end \# \rangle z)$
 $(\text{rev } x_init, q', x_end \# \rangle z) \rightarrow (\text{rev } (\langle x \rangle, q, \rangle z)$
using *x_is_init_app_end* **by** *auto*
thus *thesis* **using** *that* **by** *simp*
qed

lemma *T_None_Some_impl_reachable*:
assumes $T \ \text{None} \ (\text{Some } q)$
shows $x \ @ \ z \rightarrow^{**} (\text{rev } (\langle x \rangle, q, \rangle z)$
proof –
obtain $q' \ z'$ **where** $x \ @ \ z' \rightarrow^{L**} (\text{rev } x_init, q', x_end \# \rangle z')$
 $(\text{rev } x_init, q', x_end \# \rangle z') \rightarrow (\text{rev } (\langle x \rangle, q, \rangle z')$
using *assms* **by** *auto*
with *left_reachable_indep*[of $z' \ \text{rev } x_init \ q' \ [x_end]$] **have** $x \ @ \ z \rightarrow^{L**} (\text{rev } x_init, q', x_end \# \rangle z)$
 $(\text{rev } x_init, q', x_end \# \rangle z) \rightarrow (\text{rev } (\langle x \rangle, q, \rangle z)$
by *fastforce+*
thus $x \ @ \ z \rightarrow^{**} (\text{rev } (\langle x \rangle, q, \rangle z)$ **by** *auto*
qed

lemma *T_impl_in_states*:

```

assumes  $T\ p\ q$ 
shows  $p = \text{Some } p' \implies p' \in \text{states } M$ 
          $q = \text{Some } q' \implies q' \in \text{states } M$ 
proof -
assume somep:  $p = \text{Some } p'$ 
with assms obtain  $p''\ z$  where
   $p'' \in \text{states } M$ 
   $(\text{rev } (\langle x \rangle, p'', \rangle z)) \rightarrow (\text{rev } x\_init, p', x\_end \# \rangle z)$ 
  by (cases  $q$ ) auto
then show  $p' \in \text{states } M$  using next by blast
next
assume someq:  $q = \text{Some } q'$ 
then show  $q' \in \text{states } M$ 
proof (cases  $p$ )
  case None
    then show ?thesis using reachable_impl_in_states assms someq
      using  $T\_None\_Some\_impl\_reachable$  by blast
  next
  case (Some  $a$ )
    with someq assms obtain  $p'\ z$  where
       $p' \in \text{states } M$ 
       $(\text{rev } (\langle x \rangle, p', \rangle z)) \rightarrow (\text{rev } x\_init, a, x\_end \# \rangle z)$ 
       $(\text{rev } x\_init, a, x\_end \# \rangle z) \rightarrow^* (\text{rev } (\langle x \rangle, q', \rangle z))$ 
      by (smt (verit)  $T.cases$  left_steps_impl_steps option.discI option.inject
        rtranclp.rtrancl_into_rtrancl)
      then show ?thesis using steps_impl_in_states by blast
    qed
  qed

```

With *crossn* we show there is always a first boundary cross if a 2DFA ever crosses the boundary:

lemma $T_none_none_iff_not_some$:

$(\exists q. T\ None\ (Some\ q)) \longleftrightarrow \neg T\ None\ None$

proof

assume $\exists q. T\ None\ (Some\ q)$

then show $\neg T\ None\ None$

by (*metis* $T_None_Some_impl_reachable$ *init_no_tr_indep*)

next

assume $\neg T\ None\ None$

then obtain $q\ z$ **where** *reach*: $x @ z \rightarrow^{**} (\text{rev } (\langle x \rangle, q, \rangle z))$ **by** *blast*

then obtain n **where** $x @ z \rightarrow^X (Suc\ n) (\text{rev } (\langle x \rangle, q, \rangle z))$

proof -

from *reach* **obtain** n **where** *ncross*: $x @ z \rightarrow^X (n) (\text{rev } (\langle x \rangle, q, \rangle z))$

using *reachable_xz_impl_crossn* **by** *blast*

moreover from *this* **obtain** m **where** $n = Suc\ m$

proof -

have $\exists m. n = Suc\ m$

proof (*rule* *ccontr*)

```

    assume  $\neg ?thesis$ 
    hence  $n = 0$  by presburger
    with  $ncross$  have  $x @ z \rightarrow^{X*}(0) (\text{rev } (\langle x \rangle, q, \rangle z))$  by simp
    moreover have  $\text{left\_config } ([], \text{init } M, \langle x @ z \rangle)$  unfolding  $\text{left\_config\_def}$ 
  by simp
    moreover have  $\text{right\_config } (\text{rev } (\langle x \rangle, q, \rangle z))$  unfolding  $\text{right\_config\_def}$ 
  by simp
    ultimately show  $False$ 
      using  $\text{left\_config\_is\_not\_right\_config no\_cross\_impl\_same\_side}$  by
  auto
  qed
  thus  $thesis$  using that by blast
  qed
  ultimately show  $thesis$  using that by blast
  qed
  then show  $\exists q. T \text{ None } (Some\ q)$ 
  proof (induction  $n$  arbitrary:  $q$  rule:  $\text{less\_induct}$ )
  case ( $\text{less } n$ )
  then show  $?case$ 
  proof (cases  $n$ )
  case 0
  then obtain  $p\ p'$  where  $x @ z \rightarrow^{X*}(0) (\text{rev } x\_init, p, x\_end \# \rangle z)$ 
     $(\text{rev } x\_init, p, x\_end \# \rangle z) \rightarrow (\text{rev } (\langle x \rangle, p', \rangle z))$ 
     $(\text{rev } (\langle x \rangle, p', \rangle z) \rightarrow^{R*} (\text{rev } (\langle x \rangle, q, \rangle z))$ 
  proof -
  have  $\text{right\_config } (\text{rev } (\langle x \rangle, q, \rangle z))$  unfolding  $\text{right\_config\_def}$  by simp
  from  $\text{right\_config\_impl\_ltor\_cross}[OF\ \text{less}(2)\ \text{this}]$ 
  obtain  $p\ p'\ z'$  where  $x @ z \rightarrow^{X*}(0) (\text{rev } x\_init, p, x\_end \# \rangle z')$ 
     $(\text{rev } x\_init, p, x\_end \# \rangle z') \rightarrow (\text{rev } (\langle x \rangle, p', \rangle z')$ 
     $(\text{rev } (\langle x \rangle, p', \rangle z') \rightarrow^{R*} (\text{rev } (\langle x \rangle, q, \rangle z))$  using 0 by metis
  moreover from  $\text{this unchanged\_word}$  have  $\rangle z' = \rangle z$ 
  by ( $\text{meson right\_steps\_impl\_steps same\_append\_eq unchanged\_substrings}$ )
  ultimately show  $thesis$  using that by auto
  qed
  hence  $T \text{ None } (Some\ p')$ 
  by ( $\text{metis Nil\_is\_append\_conv init\_tr left\_config\_def left\_config\_is\_not\_right\_config}$ 
 $\text{length\_0\_conv}$ 
 $\text{length\_greater\_0\_conv no\_crossE not\_Cons\_self2 } x\_is\_init\_app\_end$ )
  then show  $?thesis$  by blast
  next
  case ( $\text{Suc } k$ )
  then obtain  $p\ p'$  where  $\text{Suc\_k\_cross}: x @ z \rightarrow^{X*}(\text{Suc } k) (\text{rev } x\_init, p,$ 
 $x\_end \# \rangle z)$ 
     $(\text{rev } x\_init, p, x\_end \# \rangle z) \rightarrow (\text{rev } (\langle x \rangle, p', \rangle z))$ 
     $(\text{rev } (\langle x \rangle, p', \rangle z) \rightarrow^{R*} (\text{rev } (\langle x \rangle, q, \rangle z))$ 
  proof -
  have  $\text{right\_config } (\text{rev } (\langle x \rangle, q, \rangle z))$  unfolding  $\text{right\_config\_def}$  by simp
  from  $\text{right\_config\_impl\_ltor\_cross}[OF\ \text{less}(2)\ \text{this}]$ 
  obtain  $p\ p'\ z'$  where  $x @ z \rightarrow^{X*}(\text{Suc } k) (\text{rev } x\_init, p, x\_end \# \rangle z')$ 

```

$(rev\ x_init, p, x_end \# \rangle z^\wedge) \rightarrow (rev\ (\langle x \rangle, p', \rangle z^\wedge)$
 $(rev\ (\langle x \rangle, p', \rangle z^\wedge) \rightarrow^{R*} (rev\ (\langle x \rangle, q, \rangle z))$ **using** *Suc* **by**

metis

moreover from this have $\rangle z^\wedge = \rangle z$
by (*meson right_steps_impl_steps same_append_eq unchanged_substrings*)
ultimately show thesis using that by auto
qed
then obtain $q' m$ **where** $x @ z \rightarrow^{X*} (Suc\ m) (rev\ (\langle x \rangle, q', \rangle z)$
 $k = Suc\ m$

proof –

from *Suc_k_cross(1)* **obtain** $q' z'$ **where** $k_steps: x @ z \rightarrow^{X*}(k) (rev$
 $(\langle x \rangle, q', \rangle z^\wedge)$
using *right_config_impl_ltor_cross*
by (*smt (verit, del_insts) append.assoc append_Cons append_Nil*
crossn_impl_reachable
left_config_impl_rtol_cross left_config_is_not_right_config list.distinct(1)
reachable_right_conf_impl_substring_z rev.simps(2) rev_append
rev_is_rev_conv rev_swap
self_append_conv x_is_init_app_end)
moreover from this have $\rangle z = \rangle z^\wedge$
by (*metis crossn_impl_reachable reach same_append_eq unchanged_substrings*)
moreover obtain m **where** $k = Suc\ m$

proof –

have $k \neq 0$
proof
assume $k = 0$
with k_steps *no_cross_impl_same_side* **show** *False*
using *left_config_def right_config_def left_config_is_not_right_config*
by (*metis (no_types, lifting) add_0 bot_nat_0.extremum le_add_same_cancel2*
length_0_conv
length_Suc_conv length_rev zero_less_Suc)
qed
thus thesis using that using *not0_implies_Suc* **by auto**
qed
ultimately show thesis using that by auto
qed
then show *?thesis* **using** *less(1) Suc* **using** *less_Suc_eq* **by auto**
qed
qed
qed
end

3 2DFAs and Regular Languages

3.1 Every Language Accepted by 2DFAs is Regular

context *dfa2*
begin

abbreviation $T \equiv \text{dfa2_transition}.T\ M$
abbreviation $\text{left_reachable} \equiv \text{dfa2_transition}.\text{left_reachable}\ M$
abbreviation $\text{left_config} \equiv \text{dfa2_transition}.\text{left_config}$
abbreviation $\text{right_config} \equiv \text{dfa2_transition}.\text{right_config}$
abbreviation $\text{pf_init} \equiv \text{dfa2_transition}.x_init$
abbreviation $\text{pf_end} \equiv \text{dfa2_transition}.x_end$

abbreviation $\text{left_step}' :: 'a\ \text{config} \Rightarrow 'a\ \text{list} \Rightarrow 'a\ \text{config} \Rightarrow \text{bool}\ (_ \rightarrow^L(_) _ \ 55)$ **where**
 $c0 \rightarrow^L(_) c1 \equiv \text{dfa2_transition}.\text{left_step}\ M\ x\ c0\ c1$

abbreviation $\text{left_steps}' :: 'a\ \text{config} \Rightarrow 'a\ \text{list} \Rightarrow 'a\ \text{config} \Rightarrow \text{bool}\ (_ \rightarrow^{L*}(_) _ \ 55)$ **where**
 $c0 \rightarrow^{L*}(_) c1 \equiv \text{dfa2_transition}.\text{left_steps}\ M\ x\ c0\ c1$

abbreviation $\text{right_step}' :: 'a\ \text{config} \Rightarrow 'a\ \text{list} \Rightarrow 'a\ \text{config} \Rightarrow \text{bool}\ (_ \rightarrow^R(_) _ \ 55)$ **where**
 $c0 \rightarrow^R(_) c1 \equiv \text{dfa2_transition}.\text{right_step}\ M\ x\ c0\ c1$

abbreviation $\text{right_steps}' :: 'a\ \text{config} \Rightarrow 'a\ \text{list} \Rightarrow 'a\ \text{config} \Rightarrow \text{bool}\ (_ \rightarrow^{R*}(_) _ \ 55)$ **where**
 $c0 \rightarrow^{R*}(_) c1 \equiv \text{dfa2_transition}.\text{right_steps}\ M\ x\ c0\ c1$

abbreviation $\text{right_stepn}' :: 'a\ \text{config} \Rightarrow 'a\ \text{list} \Rightarrow \text{nat} \Rightarrow 'a\ \text{config} \Rightarrow \text{bool}\ (_ \rightarrow^{R'}(_, _) _ \ 55)$ **where**
 $c0 \rightarrow^{R'}(x, n) c1 \equiv \text{dfa2_transition}.\text{right_stepn}\ M\ x\ c0\ n\ c1$

abbreviation $\text{left_reachable}' :: 'a\ \text{list} \Rightarrow 'a\ \text{list} \Rightarrow 'a\ \text{config} \Rightarrow \text{bool}\ (_ \rightarrow^{L**}(_) _ \ 55)$ **where**
 $w \rightarrow^{L**}(_) c \equiv \text{dfa2_transition}.\text{left_reachable}\ M\ x\ w\ c$

abbreviation $\text{crossn}' :: 'a\ \text{config} \Rightarrow 'a\ \text{list} \Rightarrow \text{nat} \Rightarrow 'a\ \text{config} \Rightarrow \text{bool}\ (_ \rightarrow^X(_, _) _ \ 55)$ **where**
 $w \rightarrow^X(x, n) y \equiv \text{dfa2_transition}.\text{crossn}\ M\ x\ w\ n\ y$

abbreviation $\text{word_crossn}' :: 'a\ \text{list} \Rightarrow 'a\ \text{list} \Rightarrow \text{nat} \Rightarrow 'a\ \text{config} \Rightarrow \text{bool}\ (_ \rightarrow^{X*}(_, _) _ \ 55)$ **where**
 $w \rightarrow^{X*}(x, n) c \equiv \text{dfa2_transition}.\text{word_crossn}\ M\ x\ w\ n\ c$

lemma $T_eq_impl_rconf_reachable$:
assumes x_stepn : $x @ z \rightarrow^{X*}(x, n) (zs @ \text{rev} (\langle x \rangle), q, v)$
and not_empty : $x \neq [] \ \& \ y \neq []$
and T_eq : $T\ x = T\ y$
shows $y @ z \rightarrow^{X*}(y, n) (zs @ \text{rev} (\langle y \rangle), q, v)$

using x_stepn **proof** (*induction* n *arbitrary*: $zs\ q\ v$ *rule*: *less_induct*)
case (*less* n)
have T_axioms : $\text{dfa2_transition}\ M\ x\ \text{dfa2_transition}\ M\ y$ **using** not_empty
 $\text{dfa2_axioms}\ \text{dfa2_transition_axioms_def}\ \text{dfa2_transition_def}$ **by** *auto*

have $n_gt_0: n > 0$
proof (rule ccontr)
 assume $\neg ?thesis$
 hence $n = 0$ **by** blast
 with less(2) **have** $(x @ z \rightarrow^{L**} \langle x \rangle) (zs @ rev (\langle x \rangle, q, v))$
 $\vee (([], init M, \langle x @ z \rangle) \rightarrow^{R*} \langle x \rangle) (zs @ rev (\langle x \rangle, q, v))$
 using $T_axioms(1)$ dfa2_transition.no_crossE **by** blast
 moreover **have** left_config x ($[], init M, \langle x @ z \rangle$)
 unfolding dfa2_transition.left_config_def[OF $T_axioms(1)$] **by** simp
 moreover **have** right_config x ($zs @ rev (\langle x \rangle, q, v)$)
 unfolding dfa2_transition.right_config_def[OF $T_axioms(1)$] **by** simp
 ultimately **show** False
 using $T_axioms(1)$ $\langle n = 0 \rangle$ dfa2_transition.left_config_is_not_right_config
 dfa2_transition.no_cross_impl_same_side less.premis **by** blast
qed
then **obtain** m **where** $m_def: n = Suc\ m$ **using** not0_implies_Suc **by** auto
obtain $p\ q'$ **where** ltor: $x @ z \rightarrow^{X*} \langle x, m \rangle (rev (pf_init\ x), p, pf_end\ x \# \rangle z)$
 $(rev (pf_init\ x), p, pf_end\ x \# \rangle z) \rightarrow (rev (\langle x \rangle, q', \rangle z))$
 $(rev (\langle x \rangle, q', \rangle z)) \rightarrow^{R*} \langle x \rangle (zs @ rev (\langle x \rangle, q, v))$
proof –
 have right_config x ($zs @ rev (\langle x \rangle, q, v)$)
 using dfa2_transition.right_config_def[OF $T_axioms(1)$] **by** auto
 from m_def less(2) dfa2_transition.right_config_impl_ltor_cross[OF $T_axioms(1)$]
 obtain $p\ q'\ z'$ **where** ltor': $x @ z \rightarrow^{X*} \langle x, m \rangle (rev (pf_init\ x), p, pf_end\ x \#$
 $\rangle z^\wedge)$
 $(rev (pf_init\ x), p, pf_end\ x \# \rangle z^\wedge) \rightarrow (rev (\langle x \rangle, q', \rangle z^\wedge))$
 $(rev (\langle x \rangle, q', \rangle z^\wedge)) \rightarrow^{R*} \langle x \rangle (zs @ rev (\langle x \rangle, q, v))$
 by (metis $\langle dfa2_transition.right_config\ x\ (zs @ rev (\vdash \# \Sigma\ x), q, v) \rangle$)
 moreover **have** $\rangle z) = \rangle z^\wedge$
proof –
 from ltor' **have** $x @ z \rightarrow^{**} (rev (\langle x \rangle, q', \rangle z^\wedge))$
 using dfa2_transition.crossn_impl_reachable[OF $T_axioms(1)$]
 by (meson rtranclp.rtrancl_into_rtrancl)
 with unchanged_word **show** ?thesis **by** force
qed
 ultimately **show** thesis **using** that **by** presburger
qed
have $y_rsteps_zs: (rev (\langle y \rangle, q', \rangle z)) \rightarrow^{R*} \langle y \rangle (zs @ rev (\langle y \rangle, q, v))$
proof –
 from ltor **have** $xq'z_reach: x @ z \rightarrow^{**} (rev (\langle x \rangle, q', \rangle z))$
 by (meson $T_axioms(1)$ dfa2_transition.crossn_impl_reachable rtranclp.rtrancl_into_rtrancl)
 from ltor(3) **obtain** i **where** $rstepi:$
 $(rev (\langle x \rangle, q', \rangle z)) \rightarrow^{R*} \langle x, i \rangle (zs @ rev (\langle x \rangle, q, v))$
 by (meson rtranclp_imp_relpowp)
 hence $stepi: (rev (\langle x \rangle, q', \rangle z)) \rightarrow \langle i \rangle (zs @ rev (\langle x \rangle, q, v))$
 by (metis $T_axioms(1)$ dfa2_transition.right_step.simps relpowp_mono)
 moreover **have**
 $x_all_geq: \forall j \leq i. \forall u' p' v'. ((rev (\langle x \rangle, q', \rangle z)) \rightarrow \langle j \rangle (u', p', v'))$

$\longrightarrow \text{length } (\text{rev } (\langle x \rangle)) \leq \text{length } u'$

proof ((rule allI)+, rule impI)+
fix $j \ u' \ p' \ v'$
assume $j_lt_i: j \leq i$
and $stepj: (\text{rev } (\langle x \rangle, q', \rangle z)) \rightarrow (j) (u', p', v')$
with $dfa2_transition.right_stepn_impl_interm_right_stepn[OF \ T_axioms(1)]$
have $(\text{rev } (\langle x \rangle, q', \rangle z)) \rightarrow^R(x,j) (u', p', v')$
using $rstepi$ **by** $blast$
hence $right_config \ x \ (u', p', v')$ **using** $dfa2_transition.right_config_def[OF \ T_axioms(1)]$
by ($metis \ T_axioms(1) \ dfa2_transition.rstepE \ length_rev \ nat_le_linear \ relpowp_E$)
thus $\text{length } (\text{rev } (\langle x \rangle)) \leq \text{length } u'$ **using** $dfa2_transition.right_config_def[OF \ T_axioms(1)]$
by $auto$
qed
ultimately have $stepi_y: (\text{rev } (\langle y \rangle, q', \rangle z)) \rightarrow (i) (zs \ @ \ \text{rev } (\langle y \rangle, q, v))$
and $y_all_geq:$
 $\forall j \leq i. \forall u' \ p' \ v'. ((\text{rev } (\langle y \rangle, q', \rangle z)) \rightarrow (j) (u', p', v'))$
 $\longrightarrow \text{length } (\text{rev } (\langle y \rangle)) \leq \text{length } u'$
using $all_geq_left_impl_left_indep[OF \ xq'z_reach \ stepi \ x_all_geq]$ **by**
 $blast+$
thus $?thesis$
proof –
note $rconf_def_y = dfa2_transition.right_config_def[OF \ T_axioms(2)]$
from $stepi_y$ **obtain** f **where** $f_def:$
 $f \ 0 = (\text{rev } (\langle y \rangle, q', \rangle z))$
 $f \ i = (zs \ @ \ \text{rev } (\langle y \rangle, q, v))$
 $\forall n < i. f \ n \rightarrow f \ (Suc \ n)$
by ($metis \ relpowp_fun_conv[of \ i \ (\rightarrow) \ (\text{rev } (\langle y \rangle, q', \rangle z)) \ (zs \ @ \ \text{rev } (\langle y \rangle, q, v))]$)
hence $\forall n < i. (f \ 0 \ \rightarrow(n) \ f \ n)$
by ($metis \ Suc_lessD \ less_trans \ Suc \ relpowp_fun_conv$)
have $rstepn_fn: \forall n < i. ((\text{rev } (\langle y \rangle, q', \rangle z)) \rightarrow^R(y,n) \ f \ n)$
proof ($rule \ allI, \ rule \ impI$)
fix n
assume $n_lt_i: n < i$
then show $((\text{rev } (\langle y \rangle, q', \rangle z)) \rightarrow^R(y,n) \ f \ n)$
proof ($induction \ n$)
case ($Suc \ n$)
hence $rstepn: ((\text{rev } (\langle y \rangle, q', \rangle z)) \rightarrow^R(y,n) \ f \ n)$ **by** $simp$
moreover from this have $fn_rconf: right_config \ y \ (f \ n)$
by ($metis \ T_axioms(2) \ dfa2_transition.rstepE \ le_refl \ length_rev \ rconf_def_y \ relpowp_E$)
moreover have $f \ n \ \rightarrow^R(|y|) \ f \ (Suc \ n)$
proof –
from $f_def \ Suc$ **have** $f \ n \ \rightarrow \ f \ (Suc \ n)$ **by** $simp$
moreover from this rstepn have $(\text{rev } (\langle y \rangle, q', \rangle z)) \rightarrow (Suc \ n) \ f \ (Suc \ n)$

```

    using Suc.premis <∀ n<i. ((→) ~ n) (f 0) (f n)> f_def(1) by auto
  moreover have right_config y (f (Suc n))
  proof -
    obtain u p v where f (Suc n) = (u, p, v) using prod_cases3 by
blast
    moreover from this y_all_geq have right_config y (u, p, v)
    using rstepn
    by (metis Suc.premis <((→) ~ Suc n) (rev (⊢ # Σ y), q', Σ z @
[-]) (f (Suc n))>
        length_rev_nat_less_le_rconf_def_y)
    ultimately show ?thesis by simp
  qed
  ultimately show ?thesis
  by (simp add: T_axioms(2) dfa2_transition.right_step.simps fn_rconf)
  qed
  ultimately show ?case by auto
  qed (use f_def in simp)
  qed
  thus ?thesis
  proof (cases i)
    case 0
    then show ?thesis using f_def by simp
  next
  case (Suc k)
  with rstepn_fn have stepk: (rev (<y>, q', >z)) →R(y,k) f k by blast
  moreover have ... →R(y) (zs @ rev (<y>, q, v))
  proof -
    from Suc_f_def have f k → (zs @ rev (<y>, q, v)) by auto
    moreover have right_config y (f k)
    using stepk
  by (metis T_axioms(2) dfa2_transition.right_config_def dfa2_transition.right_step.simps

        length_rev_less_or_eq_imp_le_relpowp_E)
    moreover have right_config y (zs @ rev (<y>, q, v))
    unfolding rconf_def_y by simp
    ultimately show ?thesis
    by (simp add: T_axioms(2) dfa2_transition.right_step.simps)
  qed
  ultimately show ?thesis
  by (meson relpowp_imp_rtranclp rtranclp.rtranclp_into_rtrancl)
  qed
  qed
  show ?case
  proof (cases m)
    case 0
    with ltor have x @ z →L**(<x>) (rev (pf_init x), p, pf_end x # >z))
    by (metis T_axioms(1) dfa2_transition.left_config_lt_right_config
        dfa2_transition.left_reachable_impl_left_config dfa2_transition.no_crossE)

```

```

length_greater_0_conv
  list.size(3) rtranclp.rtrancl_refl)
with ltor have T x None (Some q')
  using T_axioms(1) dfa2_transition.init_tr by blast
with T_eq obtain p' where y @ z →L**(|y|) (rev (pf_init y), p', pf_end y #
)z))
      (rev (pf_init y), p', pf_end y # )z)) → (rev (⟨y⟩, q', )z))
  using dfa2_transition.init_tr_indep[OF T_axioms(2)] by auto
hence y @ z →X*(y, Suc 0) (rev (⟨y⟩, q', )z))
by (meson T_axioms(2) dfa2_transition.crossn_ltor dfa2_transition.left_reachable_impl_left_config
  dfa2_transition.no_crossl rtranclp.rtrancl_refl)
then have y @ z →X*(y, Suc 0) (zs @ rev (⟨y⟩, q, v)
  using dfa2_transition.crossn_trans[OF T_axioms(2)] y_rsteps_zs
  by (meson T_axioms(2)
    ⟨∧thesis. (∧p'. [|y @ z →L**(|y|) (rev (pf_init y), p', pf_end y # )z));
    (rev (pf_init y), p', pf_end y # )z)) → (rev (⊢ # Σ y), q', Σ z @
[-])])
    ⇒ thesis) ⇒ thesis)
  dfa2_transition.crossn_ltor dfa2_transition.left_reachable_impl_left_config
  dfa2_transition.left_steps_impl_left_config_conv dfa2_transition.no_crossl)
then show ?thesis using 0 m_def by blast
next
case (Suc k)
with ltor(1) obtain p' q'' where rtol:
  x @ z →X*(x, k) (rev (⟨x⟩, p', )z))
  (rev (⟨x⟩, p', )z)) → (rev (pf_init x), q'', pf_end x # )z))
  (rev (pf_init x), q'', pf_end x # )z)) →L*(|x|) (rev (pf_init x), p, pf_end x
# )z))
proof -
  note lconf_def_x = dfa2_transition.left_config_def[OF T_axioms(1)]
  have left_config_x (rev (pf_init x), p, pf_end x # )z))
    using lconf_def_x dfa2_transition.x_defs[OF T_axioms(1)] by auto
with ltor(1) Suc dfa2_transition.left_config_impl_rtol_cross[OF T_axioms(1)]
  obtain p' q'' z' where
    x @ z →X*(x, k) (rev (⟨x⟩, p', )z^))
    (rev (⟨x⟩, p', )z^)) → (rev (pf_init x), q'', pf_end x # )z^))
    (rev (pf_init x), q'', pf_end x # )z^)) →L*(|x|) (rev (pf_init x), p, pf_end
x # )z))
  by metis
  moreover from this have )z^ = )z) using unchanged_word
  by (metis T_axioms(1) dfa2_transition.crossn_impl_reachable mapl_app_mapr_eq_map

      rev_rev_ident same_append_eq)
  ultimately show thesis using that by simp
qed
with ltor(2) have Tx_Some: T x (Some q'') (Some q')
  by (metis T_axioms(1) dfa2_transition.crossn_impl_reachable
    dfa2_transition.some_tr_init_steps_impl_in_states)
from Suc m_def have k < n by simp

```

```

with rtol(1) have y @ z →X*(y,k) (rev (<y>, p', >z))
  using less(1)[of _ []] by simp
moreover have ... → (rev (pf_init y), q'', pf_end y # >z)
proof -
  have z_empty: >z ≠ [] by simp
  with rtol(2) have p'_nxt: nxt M p' (hd (>z)) = (q'', Left) by auto
  have (rev (<y>, p', >z)) = (rev (pf_init y @ [pf_end y]), p', >z)
    using dfa2_transition.x_defs[OF T_axioms(2)] by simp
  also have ... = (pf_end y # rev (pf_init y), p', >z) by simp
  also have ... → (rev (pf_init y), q'', pf_end y # >z) using p'_nxt z_empty

  by (metis list.exhaust list.sel(1) step_left)
  finally show ?thesis .
qed
moreover from Tx_Some T_eq obtain r where
  ... →L*(y) (rev (pf_init y), r, pf_end y # >z)
  (rev (pf_init y), r, pf_end y # >z) → (rev (<y>, q', >z))
  using that dfa2_transition.some_tr_indep[OF T_axioms(2)] by metis
ultimately show ?thesis using y_rsteps_zs
using Suc T_axioms(2) dfa2_transition.crossn_ltor dfa2_transition.crossn_rtol
m_def
  by blast
qed
qed

```

The initial implication:

```

theorem T_eq_impl_eq_app_right:
  assumes not_empty: x ≠ [] y ≠ []
    and T_eq: T x = T y
    and xz_in_lang: x @ z ∈ Lang
  shows y @ z ∈ Lang
proof -
  from not_empty dfa2_axioms have T_axioms: dfa2_transition M x dfa2_transition
  M y
  using dfa2_transition_def unfolding dfa2_transition_axioms_def by auto
  with xz_in_lang obtain u v where x_acc_reachable: x @ z →** (u, acc M, v)

  unfolding Lang_def by blast
  with dfa2_transition.reachable_xz_impl_crossn[OF T_axioms(1)]
  obtain n where x @ z →X*(x,n) (u, acc M, v) by blast
  consider (left) left_config x (u, acc M, v) | (right) right_config x (u, acc M, v)
  unfolding dfa2_transition.left_config_def[OF T_axioms(1)]
    dfa2_transition.right_config_def[OF T_axioms(1)]
  by fastforce
  then show ?thesis
proof cases
  case left
  then obtain xs where rev u @ xs = <x< xs @ >z = v xs ≠ []
proof -

```

obtain xs **where** $rev\ u\ @\ xs = \langle x\langle xs\ @\ \rangle z \rangle = v$
by (*smt* (*verit*, *ccfv_threshold*) *left* $T_axioms(1)$ *dfa2_transition.left_config_def* *dfa2_transition.reachable_lconfig_impl_substring_x* *rtranclp.rtrancl_refl* *x_acc_reachable*)
moreover from *this left* **have** $xs \neq []$ **using** *dfa2_transition.left_config_def* [*OF* $T_axioms(1)$] **by** *auto*
ultimately show *thesis* **using** *that* **by** *blast*
qed
with *acc_impl_reachable_substring* **have** $revxsu_reach: x\ @\ z \rightarrow^{**} (rev\ xs\ @\ u, acc\ M, \rangle z)$
by (*smt* (*verit*, *ccfv_SIG*) *rtranclp_trans* *snoc_eq_iff_butlast* *x_acc_reachable*)
obtain m **where** $x\ @\ z \rightarrow^{X*}(x,m)$ ($rev\ (\langle x \rangle, acc\ M, \rangle z)$)
proof –
from *revxsu_reach* *dfa2_transition.reachable_xz_impl_crossn* [*OF* $T_axioms(1)$]

obtain m **where** $x\ @\ z \rightarrow^{X*}(x,m)$ ($rev\ xs\ @\ u, acc\ M, \rangle z$) **by** *blast*
moreover have $rev\ xs\ @\ u = rev\ (\langle x \rangle)$
by (*metis* $\langle rev\ u\ @\ xs = \vdash\ \# \Sigma\ x \rangle$ *rev_append* *rev_rev_ident*)
ultimately show *thesis* **using** *that* **by** *simp*
qed
with *T_eq_impl_rconf_reachable* [*OF* $_not_empty\ T_eq, of\ _ _ []$]
have $y\ @\ z \rightarrow^{X*}(y,m)$ ($rev\ (\langle y \rangle, acc\ M, \rangle z)$)
by *auto*
hence $y\ @\ z \rightarrow^{**} (rev\ (\langle y \rangle, acc\ M, \rangle z)$) **using** *dfa2_transition.crossn_impl_reachable* [*OF* $T_axioms(2)$]
by *blast*
then show *?thesis* **using** *Lang_def* **by** *blast*
next
case *right*
from *x_acc_reachable* *dfa2_transition.reachable_xz_impl_crossn* [*OF* $T_axioms(1)$]
obtain n **where** $x_crossn: x\ @\ z \rightarrow^{X*}(x,n)$ ($u, acc\ M, v$) **by** *blast*
from *right* **obtain** zs **where** $zs_defs: rev\ zs\ @\ rev\ (\langle x \rangle) = u\ zs\ @\ v = \rangle z$
by (*metis* $T_axioms(1)$ *dfa2_transition.reachable_right_conf_impl_substring_z* *rev_append* *x_acc_reachable*)
with *T_eq_impl_rconf_reachable* [*OF* $_not_empty\ T_eq$] **have**
 $y\ @\ z \rightarrow^{X*}(y,n)$ ($rev\ zs\ @\ rev\ (\langle y \rangle, acc\ M, v)$)
using x_crossn **by** *blast*
then show *?thesis* **using** *dfa2_transition.crossn_impl_reachable* [*OF* $T_axioms(2)$]

unfolding *Lang_def* **by** *blast*
qed
qed

There are finitely many transitions:

definition $\mathcal{T} :: 'a\ list \Rightarrow (state\ option \times state\ option)\ set$ **where**
 $\mathcal{T}\ x \equiv \{(q, p).\ dfa2_transition\ M\ x \wedge T\ x\ q\ p\}$

lemma $T_subset_states_none$:

```

shows  $\mathcal{T} x \subseteq (\{\text{Some } q \mid q. q \in \text{states } M\} \cup \{\text{None}\}) \times (\{\text{Some } q \mid q. q \in \text{states } M\} \cup \{\text{None}\})$ 
  (is  $\_ \subseteq ?S \times \_$ )
proof (cases x)
  case Nil
  then show ?thesis unfolding  $\mathcal{T\_def}$ 
    by (simp add: dfa2_transition_axioms_def dfa2_transition_def)
next
  case (Cons a as)
  show ?thesis
  proof
    from Cons have trans: dfa2_transition M x
      unfolding dfa2_transition_axioms_def dfa2_transition_def
      by (simp add: dfa2_axioms)
    fix pq :: state option  $\times$  state option
    assume pq  $\in \mathcal{T} x$ 
    then obtain p q where pq_def: pq = (p, q) (p, q)  $\in \mathcal{T} x$ 
      by (metis surj_pair)
    have (p, q)  $\in ?S \times ?S$ 
      using pq_def(2) dfa2_transition.T_impl_in_states[OF trans] unfolding
 $\mathcal{T\_def}$ 
      by fast
    thus pq  $\in ?S \times ?S$  using pq_def by simp
  qed
qed

```

```

lemma  $\mathcal{T\_Nil\_eq\_T\_Nil}$ :
  assumes  $\mathcal{T} x = \mathcal{T} []$ 
  shows  $x = []$ 
proof (rule ccontr)
  assume empty:  $x \neq []$ 
  then obtain p q where T x p q
    using dfa2_transition.T_none_none_iff_not_some
    by (metis dfa2_axioms dfa2_transition_axioms_def dfa2_transition_def)
  moreover from empty have dfa2_transition M x
    by (simp add: dfa2_axioms dfa2_transition.intro dfa2_transition_axioms_def)
  moreover from assms have  $\mathcal{T} x = \{\}$  unfolding  $\mathcal{T\_def}$ 
    by (simp add: dfa2_transition_axioms_def dfa2_transition_def)
  ultimately show False unfolding  $\mathcal{T\_def}$  by blast
qed

```

```

lemma  $T\_eq\_is\_T\_eq$ :
  assumes dfa2_transition M x
    dfa2_transition M y
  shows  $T x = T y \iff \mathcal{T} x = \mathcal{T} y$ 
  using assms  $\mathcal{T\_def}$  by fastforce

```

```

theorem  $\mathcal{T\_finite\_image}$ :
  finite ( $\mathcal{T} \text{ ' UNIV}$ )

```

```

proof –
  let ?S = {Some q | q. q ∈ states M} ∪ {None}
  have finite_state_options: finite ?S using finite by simp
  hence  $\mathcal{T} \text{ ' UNIV} \subseteq \text{Pow } (?S \times ?S)$  using  $\mathcal{T}$ _subset_states_none by blast
  moreover have finite (Pow (?S × ?S)) using finite_state_options by simp
  ultimately show finite ( $\mathcal{T} \text{ ' UNIV}$ ) by (simp add: finite_subset)
qed

lemma kern_  $\mathcal{T}$ _subset_eq_app_right:
  kernel  $\mathcal{T} \subseteq \text{eq\_app\_right Lang}$ 
proof
  fix xy
  assume xy ∈ kernel  $\mathcal{T}$ 
  then obtain x y where xy_def: xy = (x, y)  $\mathcal{T}$  x =  $\mathcal{T}$  y
    unfolding kernel_def by blast
  show xy ∈ eq_app_right Lang
  proof (cases x)
    case Nil
      with xy_def have y = [] using  $\mathcal{T}$ _Nil_eq_  $\mathcal{T}$ _Nil by simp
      with xy_def Nil have (x, y) = ([], []) by simp
      then show ?thesis using xy_def unfolding eq_app_right_def by simp
    next
      case (Cons a as)
      moreover from this  $\mathcal{T}$ _Nil_eq_  $\mathcal{T}$ _Nil xy_def have y ≠ [] by auto
      ultimately have T_axioms: dfa2_transition M x
        dfa2_transition M y
      by (simp add: dfa2_axioms dfa2_transition.intro dfa2_transition_axioms.intro)+
      then show ?thesis using T_eq_impl_eq_app_right
        unfolding eq_app_right_def
      by (smt (verit) T_eq_is_  $\mathcal{T}$ _eq_  $\mathcal{T}$ _Nil_eq_  $\mathcal{T}$ _Nil case_prod_conv mem_Collect_eq
xy_def(1,2))
  qed
qed

```

Lastly, eq_app_right is of finite index, from which the theorem follows by Myhill-Nerode:

```

theorem dfa2_Lang_regular:
  regular Lang
proof –
  from  $\mathcal{T}$ _finite_image have finite (UNIV // kernel  $\mathcal{T}$ )
    by (simp add: quotient_kernel_eq_image)
  then have finite (UNIV // eq_app_right Lang)
    using equiv_kernel equiv_eq_app_right finite_refines_finite kern_  $\mathcal{T}$ _subset_eq_app_right

    by blast
  then show regular Lang using L3_1 by auto
qed

end

```

3.2 Every Regular Language is Accepted by Some 2DFA

abbreviation $step' :: 'a\ config \Rightarrow 'a\ dfa2 \Rightarrow 'a\ config \Rightarrow bool\ (_ \rightarrow(_) _ 55)$

where

$$c0 \rightarrow(M) c1 \equiv dfa2.step\ M\ c0\ c1$$

abbreviation $steps' :: 'a\ config \Rightarrow 'a\ dfa2 \Rightarrow 'a\ config \Rightarrow bool\ (_ \rightarrow*(_) _ 55)$

where

$$c0 \rightarrow*(M) c1 \equiv dfa2.steps\ M\ c0\ c1$$

lemma $finite_arbitrarily_large_disj$:

$\llbracket infinite(UNIV::'a\ set); finite(A::'a\ set) \rrbracket \Longrightarrow \exists B. finite\ B \wedge card\ B = n \wedge A \cap B = \{\}$

using $infinite_arbitrarily_large[of\ UNIV - A]$

by $fastforce$

lemma $infinite_UNIV_state$: $infinite(UNIV :: state\ set)$

using $hmem_HF_iff$ **by** $blast$

Let $L \subseteq \Sigma^*$ be regular. Then there exists a DFA $M = (Q, q_0, F, \delta)^1$ that accepts L . Furthermore, let $q_0, q_a, q_r \notin Q$ be pairwise distinct states. We construct the 2DFA $M' = (Q \cup \{q_0', q_a, q_r\}, q_0', q_a, q_r, \delta')$ where

$$\delta'(q, a) = \begin{cases} (\delta(q, a), Right) & \text{if } q \in Q \text{ and } a \in \Sigma \\ (q_a, Right) & \text{if } q = q_0 \text{ and } a \in \Sigma \\ (q_r, Right) & \text{if } q \in \{q_0', q_r\} \text{ and } a \in \Sigma \\ (q_0, Right) & \text{if } (q, a) = (q_0', \vdash) \\ (q_a, Right) & \text{if } (q, a) = (q_a, \vdash) \\ (q_r, Right) & \text{if } q \in Q \cup \{q_r\} \text{ and } a = \vdash \\ (q_a, Left) & \text{if } q \in F \cup \{q_a\} \text{ and } a = \dashv \\ (q_r, Left) & \text{otherwise} \end{cases}$$

Intuitively, M' executes M on a word $w \in \Sigma^*$, and accepts it if and only if M does so:

Recall that the input of M' for w is $\vdash w \dashv$, and therefore, M' always reads \vdash in its initial configuration. The start state of M' , q_0' , moves the head of M' to the first character of w , and M' goes into state q_0 , the start state of M . Then, M' reads each character of w moving exclusively to the right, mimicking the behavior of a traditional DFA. Since M' computes its next state with δ , it behaves exactly like M until the entire word is read.

When M' finishes reading w , the head is on \dashv , and its current state is the

¹We define automata in accordance with the records $(a, 's)\ dfa$ and $'a\ dfa2$, which do not define an alphabet explicitly. Hence, we implicitly set Σ as the input alphabet for M and M' .


```

Letter a' ⇒ (if q ∈ dfa.states M then ((dfa.next M) q a', Right)
             else if q = qa then (qa, Right) else (qr, Right)) |
Marker Left ⇒ (if q = q0 then (dfa.init M, Right)
               else if q = qa then (qa, Right) else (qr, Right)) |
Marker Right ⇒ (if q ∈ dfa.final M ∨ q = qa then (qa, Left) else (qr,
Left)))

let ?M' = (dfa2.states = dfa.states M ∪ {q0, qa, qr},
          dfa2.init = q0,
          dfa2.acc = qa,
          dfa2.rej = qr,
          dfa2.next = ?δ)

interpret M: dfa2 ?M'
proof (standard, goal_cases)
  case (6 p a q d)
  then show ?case
  proof (cases a)
    case (Letter a')
    with 6 have d_eq_ite: ?δ p a = (if p ∈ dfa.states M then (dfa.next M p a',
Right)
    else if p = qa then (qa, Right) else (qr, Right)) (is _ = ?ite) by simp
  then show ?thesis
  proof (cases p ∈ dfa.states M)
    case True
    then show ?thesis using Letter dfa.next[OF ‹dfa M› True] 6 by fastforce
  next
    case False
    then show ?thesis using 6 using Letter
  by (smt (verit) Un_def dfa2.select_convs(1,5) insert_compr mem_Collect_eq
old.prod.inject
    symbol.simps(5))
  qed
next
  case (Marker d')
  then show ?thesis using 6
  by (smt (verit) Un_iff ‹dfa M› dfa.init dfa2.select_convs(1,5) dir.exhaust
dir.simps(3,4)
    insertCI old.prod.inject symbol.simps(6))
  qed
next
  case (7 q p d)
  then show ?case
  by (smt (verit, best) dfa2.select_convs(5) dir.simps(3) prod.inject sym-
bol.simps(6))
next
  case (8 q p d)
  then show ?case
  by (smt (verit, best) dfa2.select_convs(5) dir.simps(4) prod.inject sym-

```

```

bol.simps(6)
next
  case (9 a q)
  then consider (acc) q = qa | (rej) q = qr by auto
  then show ?case
  proof cases
    case acc
    then show ?thesis
    proof (cases a)
      case (Letter a')
      then show ?thesis using acc q_defs by simp
    next
      case (Marker d)
      then show ?thesis
      by (smt (verit, ccfv_SIG) 9(1) acc dfa2.select_convs(5) dir.exhaust
dir.simps(3) q_defs(3)
symbol.simps(6))
    qed
  next
  case rej
  then show ?thesis
  proof (cases a)
    case (Letter a')
    then show ?thesis using rej q_defs by simp
  next
    case (Marker d)
    then show ?thesis
    by (smt (verit) 9(1) dfa2.select_convs(5) dir.exhaust dir.simps(3)
q_defs(4) rej
symbol.simps(6))
    qed
  qed
next
  case (10 q)
  then show ?case using ⟨dfa M⟩ dfa_def q_defs(1) by fastforce
qed (use q_defs dfa.finite[OF ⟨dfa M⟩] in simp)+

have nextl_reachable:
  ∀ w. (([], dfa2.init ?M', ⟨w⟩) →*(! ?M') (rev (⟨w⟩), (dfa.nextl M (dfa.init
M) w, [¬])))
proof
  fix w
  have step: ([], dfa2.init ?M', ⟨w⟩) →(! ?M') ([¬], dfa.init M, ⟨w⟩)
  using M.step_right by auto
  have reach:
    ∀ u. ∀ q ∈ dfa.states M. ((u, q, ⟨w⟩)
→*(! ?M') (rev (Σ w) @ u, dfa.nextl M q w, [¬]))
  proof (standard, standard)
    fix u q

```

```

assume  $in\_Q: q \in dfa.states\ M$ 
show  $(u, q, \rangle w) \rightarrow^*(\langle ?M' \rangle) (rev\ (\Sigma\ w) \ @\ u, (dfa.nextl\ M\ q\ w, [-]))$ 
  using  $in\_Q$  proof (induction  $w$  arbitrary:  $q\ u$ )
  case Nil
    moreover from this have  $dfa.nextl\ M\ q\ [] = q$ 
      by (simp add:  $\langle dfa\ M \rangle\ dfa.nextl.simps(1)$ )
    moreover from this have  $(u, q, \rangle []) = (rev\ (\Sigma\ []) \ @\ u, dfa.nextl\ M\ q\ [],$ 
 $[-])$  by simp
    ultimately show ?case by simp
  next
  case (Cons  $x\ xs$ )
  from  $Cons(2)$  have  $step1: (u, q, \rangle x \# xs) \rightarrow(\langle ?M' \rangle) (Letter\ x \# u, dfa.next$ 
 $M\ q\ x, \rangle xs)$ 
     $dfa.next\ M\ q\ x \in dfa.states\ M$ 
    using  $M.step.simps$  by (auto simp add:  $\langle dfa\ M \rangle\ dfa.next$ )
  with  $Cons(1)$  have
     $(Letter\ x \# u, dfa.next\ M\ q\ x, \rangle xs)$ 
     $\rightarrow^*(\langle ?M' \rangle) (rev\ (\Sigma\ xs) \ @\ (Letter\ x \# u), dfa.nextl\ M\ (dfa.next\ M\ q\ x)\ xs,$ 
 $[-])$ 
    by simp
    moreover have  $\dots = (rev\ (\Sigma\ (x \# xs)) \ @\ u, dfa.nextl\ M\ q\ (x \# xs), [-])$ 
      by (simp add:  $\langle dfa\ M \rangle\ dfa.nextl.simps(2)$ )
    ultimately show ?case using  $step1$  by auto
  qed
qed
hence  $steps: ([-], dfa.init\ M, \rangle w) \rightarrow^*(\langle ?M' \rangle) (rev\ (\langle w \rangle), (dfa.nextl\ M\ (dfa.init$ 
 $M)\ w, [-]))$ 
proof –
  have  $dfa.init\ M \in dfa.states\ M$  using  $\langle dfa\ M \rangle\ dfa.init$  by blast
  with reach show ?thesis by simp
qed
with step show  $([], dfa2.init\ ?M', \langle w \rangle) \rightarrow^*(\langle ?M' \rangle) (rev\ (\langle w \rangle), (dfa.nextl\ M$ 
 $(dfa.init\ M)\ w, [-]))$ 
  by simp
qed

have  $M.Lang = dfa.language\ M$ 
proof
  show  $M.Lang \subseteq dfa.language\ M$ 
  proof
    fix  $w$ 
    assume  $w \in M.Lang$ 
    then obtain  $u\ v$  where  $acc\_reachable: ([], dfa2.init\ ?M', \langle w \rangle) \rightarrow^*(\langle ?M' \rangle) (u,$ 
 $dfa2.acc\ ?M', v)$ 
    using  $M.Lang\_def$  by blast
    from  $nextl\_reachable$  obtain  $q$  where  $final\_state:$ 
 $([], dfa2.init\ ?M', \langle w \rangle) \rightarrow^*(\langle ?M' \rangle) (rev\ (\langle w \rangle), q, [-])$ 
 $q \in dfa.states\ M\ dfa.nextl\ M\ (dfa.init\ M)\ w = q$ 
    using  $\langle dfa\ M \rangle\ dfa.nextl\_init\_state$  by blast

```

```

with acc_reachable have acc_step:
  (rev ( $\langle w \rangle$ ), q, [-])  $\rightarrow$ ( $?M'$ ) (tl (rev ( $\langle w \rangle$ )), dfa2.acc  $?M'$ , hd (rev ( $\langle w \rangle$ )) #
[-])
proof -
  have disj: ((rev ( $\langle w \rangle$ ), q, [-])  $\rightarrow$ ( $?M'$ ) (tl (rev ( $\langle w \rangle$ )), dfa2.acc  $?M'$ , hd (rev
( $\langle w \rangle$ ) # [-]))
     $\vee$  ((rev ( $\langle w \rangle$ ), q, [-])  $\rightarrow$ ( $?M'$ ) (tl (rev ( $\langle w \rangle$ )), dfa2.rej  $?M'$ , hd (rev
( $\langle w \rangle$ ) # [-]))
    (is ?acc_step  $\vee$  ?rej_step)
  proof (cases q  $\in$  dfa.final M)
    case True
      hence next  $?M'$  q  $\dashv$  = (dfa2.acc  $?M'$ , Left) by auto
      hence ?acc_step using M.step.simps by fastforce
      then show ?thesis by simp
    next
      case False
        moreover from q_defs final_state have q  $\neq$  dfa2.acc  $?M'$  by auto
        ultimately have next  $?M'$  q  $\dashv$  = (dfa2.rej  $?M'$ , Left) by auto
        hence ?rej_step using M.step.simps by fastforce
        then show ?thesis by simp
      qed
    show ?thesis
    proof (rule ccontr)
      assume  $\neg ?thesis$ 
      with disj have ?rej_step by blast
      hence rej_reachable:
        ( $\square$ , dfa2.init  $?M'$ ,  $\langle w \rangle$ )  $\rightarrow^*$ ( $?M'$ ) (tl (rev ( $\langle w \rangle$ )), dfa2.rej  $?M'$ , hd (rev
( $\langle w \rangle$ ) # [-])
        using final_state by auto
        with acc_reachable consider
          ((tl (rev ( $\langle w \rangle$ )), dfa2.rej  $?M'$ , hd (rev ( $\langle w \rangle$ ) # [-])  $\rightarrow^*$ ( $?M'$ ) (u, dfa2.acc
 $?M'$ , v) |
            (u, dfa2.acc  $?M'$ , v)  $\rightarrow^*$ ( $?M'$ ) (tl (rev ( $\langle w \rangle$ )), dfa2.rej  $?M'$ , hd (rev
( $\langle w \rangle$ ) # [-])
            using M.reachable_configs_impl_reachable by blast
            then show False
            by cases (use M.unchanged_final M.neq_final in fastforce)+
            qed
          qed
        have q  $\in$  dfa.final M
        proof (rule ccontr)
          assume q  $\notin$  dfa.final M
          with final_state(2) have dfa2.next  $?M'$  q  $\dashv$  = (dfa2.rej  $?M'$ , Left)
          using q_defs(1) by auto
          with acc_step show False using q_defs(2) by fastforce
        qed
      thus w  $\in$  dfa.language M
      using final_state(3) dfa.language_def[OF  $\langle$ dfa M $\rangle$ ] by blast
      qed

```

```

next
  show  $dfa.language\ M \subseteq M.Lang$ 
  proof
    fix  $w$ 
    assume  $w \in dfa.language\ M$ 
    then obtain  $q$  where  $dfa.nextl\ M\ (dfa.init\ M)\ w = q$  and  $in\_final: q \in$ 
 $dfa.final\ M$ 
    by (simp add:  $\langle dfa\ M \rangle\ dfa.language\_def$ )
    with nextl_reachable have  $([],\ dfa2.init\ ?M',\ \langle w \rangle) \rightarrow^*(\ ?M')$   $(rev\ (\langle w \rangle),\ q,$ 
 $[-])$ 
    by blast
    also from this in_final have  $\dots \rightarrow(\ ?M')$   $(tl\ (rev\ (\langle w \rangle)),\ dfa2.acc\ ?M',\ hd$ 
 $(rev\ (\langle w \rangle))\ \# [-])$ 
    using  $M.step.simps$  by auto
    finally show  $w \in M.Lang$  unfolding  $M.Lang\_def$  by blast
  qed
qed
then show thesis using that  $\langle dfa.language\ M = L \rangle\ M.dfa2\_axioms\ M\_def$ 
 $q\_defs$  by presburger
qed

```

The equality follows trivially:

corollary $dfa2_accepts_regular_languages:$

$regular\ L = (\exists M. dfa2\ M \wedge dfa2.Lang\ M = L)$

using $dfa2.dfa2_Lang_regular\ regular_language_impl_dfa2$ **by** *fastforce*

end

References

- [1] D. Kozen. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997.
- [2] L. C. Paulson. Finite automata in hereditarily finite set theory. *Archive of Formal Proofs*, February 2015. https://isa-afp.org/entries/Finite_Automata_HF.html, Formal proof development.