Two-Way Deterministic Finite Automata

Felipe Escallón and Tobias Nipkow

August 8, 2025

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 Σ :: *'a list* ⇒*'a symbol list* **where**
Σ *w* ≡ *map Letter w*

abbreviation *marker_map* :: *'a list* ⇒ *'a symbol list* (⟨_⟩ *70*) **where**
⟨*w*⟩ ≡ ⊢ # (Σ *w*) @ [⊣]

abbreviation *marker_mapl* :: *'a list* ⇒ *'a symbol list* (⟨_⟨ *70*) **where**
⟨*w*⟨ ≡ ⊢ # (Σ *w*)

abbreviation *marker_mapr* :: *'a list* ⇒ *'a symbol list* (⟩_⟩ *70*) **where**
⟩*w*⟩ ≡ (Σ *w*) @ [⊣]

lemma *mapl_app_mapr_eq_map*:
⟨*u*⟨ @ ⟩*v*⟩ = ⟨*u* @ *v*⟩ **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* ⇒ *'a config* ⇒ *bool* (**infix** ‹→› *55*) **where**
step_left[*intro*]: *nxt M p a* = (*q*, *Left*) ⟹ (*x* # *xs*, *p*, *a* # *ys*) → (*xs*, *q*, *x* # *a* # *ys*) |
step_right[*intro*]: *nxt M p a* = (*q*, *Right*) ⟹ (*xs*, *p*, *a* # *ys*) → (*a* # *xs*, *q*, *ys*)

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

The reflexive transitive closure of $\to$:

abbreviation *steps* :: *'a config* ⇒ *'a config* ⇒ *bool* (**infix** ‹→*› 55) **where**
  steps ≡ *step***

And the nth power of $\to$:

abbreviation *stepn* :: *'a config* ⇒ *nat* ⇒ *'a config* ⇒ *bool* (_ →'(_') _ 55) **where**
  stepn c n ≡ (*step* ^^ *n*) *c*

lemma *rtranclp_induct3*[*consumes 1*, *case_names refl step*]:
  ⟦*r*** (*ax*, *ay*, *az*) (*bx*, *by*, *bz*); *P ax ay az*;
    ⋀*u p v x q z*.
      ⟦*r*** (*ax*, *ay*, *az*) (*u*, *p*, *v*); *r* (*u*, *p*, *v*) (*x*, *q*, *z*); *P u p v*⟧
      ⟹ *P x q z*⟧
   ⟹ *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 $([], \mathit{init}\ M, \vdash w \dashv)$. A configuration c is reachable by w if the initial configuration of M on input w reaches c:

abbreviation *reachable* :: *'a list* ⇒ *'a config* ⇒ *bool* (**infix** ‹→**› 55) **where**
  w →** *c* ≡ ([], *init M*, ⟨*w*⟩) →* *c*

abbreviation *nreachable* :: *'a list* ⇒ *nat* ⇒ *'a config* ⇒ *bool* (_ →*'(_') _ 55)
where
  nreachable w n c ≡ ([], *init M*, ⟨*w*⟩) →(*n*) *c*

lemma *step_unique*:
  ⟦ *c0* → *c1*; *c0* → *c2* ⟧ ⟹ *c1* = *c2*
by *fastforce*

lemma *steps_impl_in_states*:
  assumes *p* ∈ *states M*
  shows (*u*, *p*, *v*) →* (*u'*, *q*, *v'*) ⟹ *q* ∈ *states M*
by (*induction rule*: *rtranclp_induct3*) (*use assms nxt* **in** *auto*)

corollary *reachable_impl_in_states*:
  w →** (*u*, *q*, *v*) ⟹ *q* ∈ *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**
  Lang ≡ {*w*. ∃ *u v*. *w* →** (*u*, *acc M*, *v*)}

lemma *unchanged_substrings*:
  $(u,\ p,\ v) \rightarrow* (u',\ q,\ v') \Longrightarrow rev\ u\ @\ v = rev\ u'\ @\ v'$
proof (*induction rule*: *rtranclp_induct3*)
  case (*step a q' b c q'' d*)
  then obtain *p' d'* **where** *qd_def*: *nxt M q' (hd b) = (p', d')* **by** *fastforce*
  then show *?case*
  proof (*cases d'*)
    case *Left*
    hence $(c,\ q'',\ d) = (tl\ a,\ p',\ hd\ a\ \#\ b)$
      using *step(2) qd_def step.simps* **by** *force*
    then show *?thesis*
      using *step.IH step.hyps(2)* **by** *fastforce*
  next
    case *Right*
    hence $(c,\ q'',\ d) = (hd\ b\ \#\ a,\ p',\ tl\ b)$ **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 $p = acc\ M \vee p = rej\ M$
  shows $(u,\ p,\ v) \rightarrow* (u',\ q,\ v') \Longrightarrow p = q$
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*:
  $([],\ p,\ w) \rightarrow* (u,\ q,\ v) \Longrightarrow w = rev\ u\ @\ v$
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*]:
  $(u,\ p,\ v) \rightarrow* (u',\ q,\ v') \Longrightarrow (u,\ p,\ v\ @\ xs) \rightarrow* (u',\ q,\ v'\ @\ xs)$
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* | (*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 $v = xs\ @\ ys \Longrightarrow (u,\ acc\ M,\ v) \rightarrow* (rev\ xs\ @\ u,\ acc\ M,\ ys)$
using *assms*
proof (*induction v arbitrary*: *u xs ys*)
  case (*Cons a v*)
  consider (*not_right_marker*) *b* **where** $a = Letter\ b \vee a = \vdash$ | (*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.prems(1,3) 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.prems*(*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
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_rtranclpE)
      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*:
$\langle x \langle = x\_init \mathbin{@} [x\_end]$ **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* ⇒ *bool* **where**
$left\_config\ c \equiv \exists\, u\ q\ v.\ c = (u,\ q,\ v) \wedge length\ u < length\ (\langle x \langle)$

definition *right_config* :: *'a config* ⇒ *bool* **where**
$right\_config\ c \equiv \exists\, u\ q\ v.\ c = (u,\ q,\ v) \wedge length\ u \geq length\ (\langle x \langle)$

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*:
$[\![\ left\_config\ (u,\ p,\ v);\ right\_config\ (w,\ q,\ y)\ ]\!] \Longrightarrow length\ u < length\ w$
using *left_config_def right_config_def* **by** *simp*

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

inductive *left_step* :: *'a config* ⇒ *'a config* ⇒ *bool* (**infix** $\langle \to^L \rangle$ *55*) **where**
lstep [*intro*]: $[\![c0 \to c1;\ left\_config\ c0;\ left\_config\ c1]\!] \Longrightarrow c0 \to^L c1$

inductive_cases *lstepE* [*elim*]: $c0 \to^L c1$

abbreviation *left_steps* :: *'a config* ⇒ *'a config* ⇒ *bool* (**infix** $\langle \to^L * \rangle$ *55*) **where**
$left\_steps \equiv left\_step^{**}$

abbreviation *left_stepn* :: *'a config* ⇒ *nat* ⇒ *'a config* ⇒ *bool* (_ $\to^L$ '(_') _ *55*) **where**
$left\_stepn\ c\ n \equiv (left\_step \mathbin{\frown\!\frown} n)\ c$

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

abbreviation *left_reachable* :: *'a list* ⇒ *'a config* ⇒ *bool* (**infix** $\langle \to^L ** \rangle$ *55*) **where**
$w \to^L ** c \equiv ([], init\ M, \langle w \rangle) \to^L * c$

abbreviation *left_nreachable* :: *'a list* ⇒ *nat* ⇒ *'a config* ⇒ *bool* (_ $\to^L$*'(_') _ *55*) **where**
$w \to^L (n)\ c \equiv ([], init\ M, \langle w \rangle) \to^L (n)\ c$

Right steps are defined analogously:

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

rstep [*intro*]: ⟦*c0* → *c1*; *right_config c0*; *right_config c1*⟧ ⟹ *c0* →^{R} *c1*

inductive_cases *rstepE* [*elim*]: *c0* →^{R} *c1*

abbreviation *right_steps* :: *'a config* ⇒ *'a config* ⇒ *bool* (**infix** ‹→^{R}*›* *55*) **where**
right_steps ≡ *right_step***

abbreviation *right_stepn* :: *'a config* ⇒ *nat* ⇒ *'a config* ⇒ *bool* (_ →^{R}'(_') _ *55*) **where**
right_stepn c n ≡ (*right_step* ^^ *n*) *c*

2.1.2 Properties of left and right steps

lemma *left_steps_impl_steps* [*dest*]:
c0 →^{L}* *c1* ⟹ *c0* →* *c1*
by (*induction rule*: *rtranclp_induct*) *auto*

lemma *right_steps_impl_steps* [*dest*]:
c0 →^{R}* *c1* ⟹ *c0* →* *c1*
by (*induction rule*: *rtranclp_induct*) *auto*

lemma *left_steps_impl_left_config*[*dest*]:
⟦*c0* →^{L}* *c1*; *left_config c0*⟧ ⟹ *left_config c1*
by (*induction rule*: *rtranclp_induct*) *auto*

lemma *left_steps_impl_left_config_conv*[*dest*]:
⟦*c0* →^{L}* *c1*; *left_config c1*⟧ ⟹ *left_config c0*
by (*induction rule*: *rtranclp_induct*) *auto*

lemma *left_reachable_impl_left_config*:
w →^{L}** *c* ⟹ *left_config c*
using *left_config_def left_steps_impl_left_config* **by** *auto*

lemma *right_steps_impl_right_config*[*dest*]:
⟦*c0* →^{R}* *c1*; *right_config c0*⟧ ⟹ *right_config c1*
by (*induction rule*: *rtranclp_induct*) *auto*

lemma *left_step_tl_indep*:
⟦ (*u*, *p*, *w* @ *v*) →^{L} (*y*, *q*, *z* @ *v*); *w* ≠ [] ⟧ ⟹ (*u*, *p*, *w* @ *v'*) →^{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* →^{R}(*n*) *c2*
c0 →(*m*) *c1*
m ≤ *n*
shows *c0* →^{R}(*m*) *c1*
proof −

```
  from assms(1) obtain f where f_def:
    f 0 = c0
    f n = c2
    ∀ i<n. f i →^R f (Suc i)
    by (metis relpowp_fun_conv)
  from assms(2) obtain g where g_def:
    g 0 = c0
    g m = c1
    ∀ i<m. g i → g (Suc i)
    by (metis relpowp_fun_conv)
  have i<m ⟹ 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 ∀ i<m. g i →^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 ≤ length xs
  obtains ys zs where 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 ≤ length xs
  obtains ys zs where length zs = m ys @ zs = xs
proof −
  from assms have m ≤ length (rev xs) by simp
  then obtain ys zs where length ys = m ys @ zs = 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: rev u @ v = ⟨x @ z⟩
     and in_x:   length u < length (⟨x⟨)
     and lsteps: (u, p, v) →^L* (u′, q, v′)
  obtains y where  rev u′ @ y = ⟨x⟨ y @ ⟩z⟩ = v′
proof −
  have leftconfig: left_config (u, p, v) unfolding left_config_def using in_x by
blast
```

```
  hence u'_lt_x: length u' < length (⟨x⟨) using lsteps left_config_def by force
  from lsteps show thesis
  proof (induction arbitrary: u p v rule: rtranclp_induct3)
    case refl
    from unchanged_word app_eq lsteps have app: ⟨x @ z⟩ = rev u' @ v'
      by (metis left_steps_impl_steps unchanged_substrings)
    moreover with u'_lt_x
    obtain y where rev u' @ y = ⟨x⟨ y @ ⟩z⟩ = v'
    proof −
      from u'_lt_x list_deconstruct1
      obtain xs ys where length xs = length u' and xapp: xs @ ys = ⟨x⟨
        using Nat.less_imp_le_nat by metis
      moreover from this have  length (ys @ ⟩z⟩) = length v' using app
        by (smt (verit) append_assoc append_eq_append_conv length_rev
            mapl_app_mapr_eq_map)
      ultimately have xs_is_rev: xs = rev u'
      by (metis (no_types) app append_Cons append_assoc append_eq_append_conv
map_append)
      then have ys @ ⟩z⟩ = v' using xapp app
        by (metis (no_types) append_assoc same_append_eq[of xs v' ys @ Σ z @
[⊣]] mapl_app_mapr_eq_map)
      thus thesis using that xs_is_rev xapp by presburger
    qed
    ultimately show ?case using that by simp
  qed blast
qed

corollary left_reachable_impl_substring_x:
  assumes x @ z →^L** (u, q, v)
  obtains y where  rev u @ y = ⟨x⟨ y @ ⟩z⟩ = v
using lstar_impl_substring_x assms left_config_def left_reachable_impl_left_config
by blast

corollary reachable_lconfig_impl_substring_x:
  assumes x @ z →** (u, p, v)
      and length u < length (⟨x⟨)
      and (u, p, v) →^L* (u', q, v')
  obtains y where  rev u' @ y = ⟨x⟨ y @ ⟩z⟩ = v'
using unchanged_word[OF assms(1)] lstar_impl_substring_x assms by metis

lemma star_rconfig_impl_substring_z:
  assumes app_eq: x @ z →** (u, p, v)
      and reach: (u, p, v) →* (u', q, v')
      and rconf: right_config (u', q, v')
  obtains y where  rev (⟨x⟨ @ y) = u' y @ v' = ⟩z⟩
proof −
  from right_config_def have u'_ge_x: length (⟨x⟨) ≤ 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 app: ⟨x @ z⟩ = rev u' @ v'
      by (metis reach unchanged_substrings)
    moreover with u'_ge_x
    obtain x' where rev (⟨x⟨ @ x') = u' x' @ v' = ⟩z⟩
    proof −
      have length v' ≤ length (⟩z⟩)
      proof (rule ccontr)
        assume ¬?thesis
        hence length v' > length (⟩z⟩) by simp
        with u'_ge_x
        have length (rev u' @ v') > length (⟨x @ z⟩) by simp
        thus False using app by (metis nat_less_le)
      qed
      from list_deconstruct2[OF this]
      obtain xs ys where length ys = length v' and zapp: xs @ ys = ⟩z⟩
        by metis
      moreover from this have length (⟨x⟨ @ xs) = 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': ⟨x⟨ @ xs = rev u' using zapp app
        by (metis (no_types, lifting) append_assoc append_eq_append_conv
            mapl_app_mapr_eq_map)
      hence rev (⟨x⟨ @ xs) = 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 →** (u, q, v)
          right_config (u, q, v)
  obtains y where rev (⟨x⟨ @ y) = u y @ v = ⟩z⟩
using assms star_rconfig_impl_substring_z right_config_def by blast

lemma lsteps_indep:
  assumes (u, p, v @ ⟩z⟩) →L* (w, q, y @ ⟩z⟩)
          rev u @ v @ ⟩z⟩ = ⟨x @ z⟩
  shows (u, p, v @ ⟩z'⟩) →L* (w, q, y @ ⟩z'⟩)
proof −
  from assms obtain n where nsteps: (u, p, v @ ⟩z⟩) →L(n) (w, q, y @ ⟩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 @ ⟩z⟩) →^L(n) (w', q', y' @ ⟩z⟩)
                      and lstep:  (w', q', y' @ ⟩z⟩) →^L (w, q, y @ ⟩z⟩)
    proof −
      from Suc.prems obtain w' q' y' where lstepn: (u, p, v @ ⟩z⟩) →^L(n) (w',
q', y')
                                      and lstep: (w', q', y') →^L (w, q, y @ ⟩z⟩) by auto
      hence w'_left: left_config (w', q', y') by blast
      then obtain xs where xs @ ⟩z⟩ = y'
      proof −
        from lstepn have (u, p, v @ ⟩z⟩) →^L* (w', q', y')
          by (simp add: relpowp_imp_rtranclp)
        moreover have length u < length (⟨x⟨)
       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.prems(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 @ ⟩z'⟩) →^L* (w', q', y' @ ⟩z'⟩)
      by (simp add: Suc.prems(2))
    moreover have (w', q', y' @ ⟩z'⟩) →^L (w, q, y @ ⟩z'⟩)
    proof −
      have y'_not_empty: y' ≠ []
      proof
        assume y' = []
        hence (u, p, v @ ⟩z'⟩) →^L* (w', q', ⟩z'⟩) using calculation by auto
        moreover have left_config (u, p, v @ ⟩z⟩)
          by (meson Suc.prems(1) ‹(w', q', y' @ Σ z @ [⊣]) →^L (w, q, y @ Σ z @
[⊣])›
          dfa2_transition.left_steps_impl_left_config_conv dfa2_transition_axioms
left_step.cases
              relpowp_imp_rtranclp)
        ultimately have lc: left_config (w', q', ⟩z'⟩)
            using left_config_is_not_right_config left_config_lt_right_config by
blast
        have rev u @ v @ ⟩z'⟩ = ⟨x @ z'⟩
          using assms(2) by force
        hence rev w' @ ⟩z'⟩ = ...
          by (metis ‹(u, p, v @ Σ z' @ [⊣]) →^L* (w', q', y' @ Σ z' @ [⊣])› ‹y' = []›
left_steps_impl_steps
              self_append_conv2 unchanged_substrings)
        hence length w' = length (⟨x⟨) 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 @ $\rangle y\langle$)
shows x @ z $\rightarrow^L{**}$ (u, q, v @ $\rangle z\langle$)
proof −
from *assms* **obtain** n **where** ([], *init* M, $\langle x$ @ $y\rangle$) $\rightarrow^L(n)$ (u, q, v @ $\rangle y\langle$)
by (*meson rtranclp_power*)
hence ([], *init* M, $\langle x$ @ $z\rangle$) $\rightarrow^L(n)$ (u, q, v @ $\rangle z\langle$)
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' @ $\rangle y\langle$)
and (u', p, v' @ $\rangle y\langle$) $\rightarrow^L(1)$ (u, q, v @ $\rangle y\langle$)
proof −
from *Suc*(*2*) **obtain** u' p v''
where x @ y $\rightarrow^L{*}(n)$ (u', p, v'')
(u', p, v'') $\rightarrow^L(1)$ (u, q, v @ $\rangle y\langle$) **by** *auto*
moreover with *left_reachable_impl_substring_x* **obtain** v' **where** $v'' = v'$ @ $\rangle y\langle$
using *rtranclp_power* **by** *metis*
ultimately show *thesis* **using** *that* **by** *blast*
qed
from *this* **have** *y_lstep*: (u', p, v' @ $\rangle y\langle$) $\rightarrow^L$ (u, q, v @ $\rangle y\langle$)
by *fastforce*
hence (u', p, v' @ $\rangle z\langle$) $\rightarrow^L$ (u, q, v @ $\rangle z\langle$)
proof −
from *y_lstep* **have** *left_configs*:
left_config (u', p, v' @ $\rangle y\langle$)
left_config (u, q, v @ $\rangle y\langle$) **by** *blast+*
hence *left_config* (u', p, v' @ $\rangle z\langle$)
left_config (u, q, v @ $\rangle z\langle$)
unfolding *left_config_def* **by** *auto*
moreover have (u', p, v' @ $\rangle z\langle$) → (u, q, v @ $\rangle z\langle$)
proof −
from *y_lstep* **have** *y_step*: (u', p, v' @ $\rangle y\langle$) → (u, q, v @ $\rangle y\langle$) **by** *blast*
obtain c *vs* **where** *v'_def*: $v' = c$ # *vs*
proof −
from *unchanged_word* **have** *rev* u' @ v' @ $\rangle y\langle = \langle x$ @ $y\rangle$
by (*metis left_steps_impl_steps relpowp_imp_rtranclp stepn*)
hence *rev_u'_app_v'*: *rev* u' @ $v' = \langle x\langle$ **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* @ $\rangle y\langle$) → (u, q, v @ $\rangle y\langle$) **by** *simp*
hence (u', p, c # *vs* @ $\rangle z\langle$) → (u, q, v @ $\rangle z\langle$) **by** *fastforce*
with *v'_def* **show** *?thesis* **by** *simp*
qed

ultimately show *?thesis* **by** *blast*
qed
moreover from *Suc(1)[OF stepn]* **have** $x \mathbin{@} z \rightarrow^L *(n)\ (u',\ p,\ v' \mathbin{@} \rangle z\langle)$ **.**
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* (_ $\rightarrow^X$ '(_') _ *55*) **where**
no_crossl: ⟦*left_config c0*; *c0* $\rightarrow^L *$ *c1*⟧ ⟹ *c0* $\rightarrow^X (0)$ *c1* |
no_crossr: ⟦*right_config c0*; *c0* $\rightarrow^R *$ *c1*⟧ ⟹ *c0* $\rightarrow^X (0)$ *c1* |
crossn_rtol: ⟦*c0* $\rightarrow^X (n)$ (*rev* (⟨*x*⟨, *p*, ⟩*z*⟩);
(*rev* (⟨*x*⟨, *p*, ⟩*z*⟩) → (*rev x_init*, *q*, *x_end* # ⟩*z*⟩);
(*rev x_init*, *q*, *x_end* # ⟩*z*⟩) $\rightarrow^L *$ *c1*⟧ ⟹ *c0* $\rightarrow^X$ (*Suc n*) *c1* |
crossn_ltor: ⟦ *c0* $\rightarrow^X (n)$ (*rev x_init*, *p*, *x_end* # ⟩*z*⟩);
(*rev x_init*, *p*, *x_end* # ⟩*z*⟩) → (*rev* (⟨*x*⟨, *q*, ⟩*z*⟩);
(*rev* (⟨*x*⟨, *q*, ⟩*z*⟩) $\rightarrow^R *$ *c1*⟧ ⟹ *c0* $\rightarrow^X$ (*Suc n*) *c1*

declare *crossn.intros*[*intro*]

inductive_cases *no_crossE*[*elim*]: *c0* $\rightarrow^X (0)$ *c1*
inductive_cases *crossE*[*elim*]: *c0* $\rightarrow^X$ (*Suc n*) *c1*

abbreviation *word_crossn* :: *'a list* ⇒ *nat* ⇒ *'a config* ⇒ *bool* (_ $\rightarrow^X *$'(_') _ *55*) **where**
word_crossn w n c ≡ ([], *init M*, ⟨*w*⟩) $\rightarrow^X (n)$ *c*

lemma *self_nocross*[*simp*]:
c $\rightarrow^X (0)$ *c* **using** *left_config_is_not_right_config* **by** *blast*

lemma *no_cross_impl_same_side*:
c0 $\rightarrow^X (0)$ *c1* ⟹ *left_config c0* = *left_config c1*
using *left_config_is_not_right_config* **by** *blast*

lemma *left_config_impl_rtol_cross*:
assumes *c0* $\rightarrow^X$ (*Suc n*) *c1*
left_config c1
obtains *p q z* **where** *c0* $\rightarrow^X (n)$ (*rev* (⟨*x*⟨, *p*, ⟩*z*⟩)
(*rev* (⟨*x*⟨, *p*, ⟩*z*⟩) → (*rev x_init*, *q*, *x_end* # ⟩*z*⟩)
(*rev x_init*, *q*, *x_end* # ⟩*z*⟩) $\rightarrow^L *$ *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 →^X (Suc n) c1
          right_config c1
  obtains p q z where 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
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 →^X (Suc n) c2
  obtains c1 where c0 →^X (n) c1 c1 →^X (Suc 0) c2
using assms proof cases
  case (crossn_rtol p z q)
  moreover have (rev (⟨x⟨, p, ⟩z⟩)) →^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 # ⟩z⟩) →^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 → c1
          c0 = (u, p, v)
          rev u @ v = ⟨x @ z⟩
  shows (c0 →^X (0) c1) ∨ (c0 →^X (Suc 0) c1)
proof (cases left_config c0 = left_config c1)
  case True
 consider left_config c0 | 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 = (rev (⟨x⟨(), p, ⟩z⟩))
                              c1 = (rev x_init, q, x_end # ⟩z⟩))
    proof −
      obtain y q w where c1_def: c1 = (y, q, w) using prod_cases3 by blast
      have length u = length (⟨x⟨() length y = length (⟨x⟨() − 1
      proof −
        from right assms(2) have length u ≥ length (⟨x⟨() using right_config_def
by auto
       moreover from right False left_config_def c1_def have length y < length
(⟨x⟨()
         using left_config_is_not_right_config by blast
         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⟨() length y = length (⟨x⟨() − 1
         by force+
      qed
      with assms(2,3) have u = rev (⟨x⟨()
          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 = ⟩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_init
        using ‹length y = length (⊢ # Σ x) − 1› ‹u = rev (⟨x⟨()› ‹v = Σ z @ [⊣]›
assms(1,2) c1_def
          x_is_init_app_end by auto
      moreover from this have w = x_end # ⟩z⟩
       by (smt (verit) ‹length u = length (⊢ # Σ x)› ‹length y = length (⊢ # Σ x)
− 1›
            ‹u = rev (⊢ # Σ x)› ‹v = Σ z @ [⊣]› assms(1,2) c1_def diff_le_self
impossible_Cons last_snoc
           prod.inject rev_eq_Cons_iff step_foldedE x_end_def)
     ultimately have c1 = (rev x_init, q, x_end # ⟩z⟩) using c1_def by simp
      moreover have c0 = (rev (⟨x⟨(), p, ⟩z⟩))
        by (simp add: ‹u = rev (⟨x⟨()› ‹v = 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
  qed
qed

lemma crossn_no_cross_eq_crossn:
  assumes c0 →^X(n) c1
          c1 →^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 $c1 \rightarrow^X (m)\ c2 \Longrightarrow c0 \rightarrow^X (n+m)\ c2$
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_is_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 →^R* c2 using left_config_is_not_right_config by blast
    with crossn_ltor(4) have (rev (⟨x⟨, q, ⟩z⟩)) →^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 →^X(m) c'
                                    and c'_cross: c' →^X(Suc 0) c2 by blast
  with Suc(1) have c0_nm_cross: c0 →^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 →^X(n+m) (rev (⟨x⟨, r, ⟩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 →^X(n+m) (rev x_init, r, x_end # ⟩w⟩)) by blast
    ultimately show ?thesis by auto
  qed
qed

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

lemma reachable_xz_impl_crossn:
  assumes c0 →* c1
          c0 = (u, p, v)
          rev u @ v = ⟨x @ z⟩
  obtains n where c0 →^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 →^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 = ⟨x @ z⟩
```

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* ⇒ *state option* ⇒ *bool* **where**
init_tr: ⟦ x @ $z \rightarrow^L{**}$ (*rev x_init*, p, *x_end* # ⟩z⟩);
(*rev x_init*, p, *x_end* # ⟩z⟩) → (*rev* (⟨x⟨, q, ⟩z⟩)⟧ ⟹ T *None* (*Some q*) |

init_no_tr: ∄ $q\ z$. x @ z →** (*rev* (⟨x⟨, q, ⟩z⟩) ⟹ T *None None* |

some_tr: ⟦ $p' \in$ *states M*; (*rev* (⟨x⟨, p', ⟩z⟩) → (*rev x_init*, p, *x_end* # ⟩z⟩);
(*rev x_init*, p, *x_end* # ⟩z⟩) $\rightarrow^L{*}$ (*rev x_init*, q', *x_end* # ⟩z⟩);
(*rev x_init*, q', *x_end* # ⟩z⟩) → (*rev* (⟨x⟨, q, ⟩z⟩)⟧ ⟹ T (*Some p*) (*Some q*) |

no_tr: ⟦ $p' \in$ *states M*; (*rev* (⟨x⟨, p', ⟩z⟩) → (*rev x_init*, p, *x_end* # ⟩z⟩);
∄ $q'\ q''\ z$. (*rev x_init*, p, *x_end* # ⟩z⟩) $\rightarrow^L{*}$ (*rev x_init*, q', *x_end* # ⟩z⟩) ∧
(*rev x_init*, q', *x_end* # ⟩z⟩) → (*rev* (⟨x⟨, q'', ⟩z⟩)⟧ ⟹ 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* # ⟩z⟩)
(*rev x_init*, p, *x_end* # ⟩z⟩) → (*rev* (⟨x⟨, q, ⟩z⟩)
proof −

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

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

lemma *some_tr_indep*:
assumes $T\ (Some\ p)\ (Some\ q)$
obtains q' **where** $(rev\ x\_init,\ p,\ x\_end\ \#\ \rangle z\langle) \rightarrow^L{*}\ (rev\ x\_init,\ q',\ x\_end\ \#\ \rangle z\langle)$
$(rev\ x\_init,\ q',\ x\_end\ \#\ \rangle z\langle) \rightarrow (rev\ (\langle x\langle),\ q,\ \rangle z\langle)$
proof −
from *assms* **obtain** $p'\ q'\ z'$ **where** *prems*:
$(rev\ (\langle x\langle),\ p',\ \rangle z'\langle) \rightarrow (rev\ x\_init,\ p,\ x\_end\ \#\ \rangle z'\langle)$
$(rev\ x\_init,\ p,\ x\_end\ \#\ \rangle z'\langle) \rightarrow^L{*}\ (rev\ x\_init,\ q',\ x\_end\ \#\ \rangle z'\langle)$
$(rev\ x\_init,\ q',\ x\_end\ \#\ \rangle z'\langle) \rightarrow (rev\ (\langle x\langle),\ q,\ \rangle z'\langle)$ **by** *auto*
with *lsteps_indep*[*of rev x_init p* [*x_end*] *z' rev x_init q'* [*x_end*] *z*] **have**
$(rev\ x\_init,\ p,\ x\_end\ \#\ \rangle z\langle) \rightarrow^L{*}\ (rev\ x\_init,\ q',\ x\_end\ \#\ \rangle z\langle)$
$(rev\ x\_init,\ q',\ x\_end\ \#\ \rangle z\langle) \rightarrow (rev\ (\langle x\langle),\ q,\ \rangle z\langle)$
using *x_is_init_app_end* **by** *auto*
thus *thesis* **using** *that* **by** *simp*
qed

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

lemma *T_impl_in_states*:

```
  assumes T p q
  shows p = Some p' ⟹ p' ∈ states M
        q = Some q' ⟹ q' ∈ states M
proof −
  assume somep: p = Some p'
  with assms obtain p'' z where
     p'' ∈ states M
     (rev (⟨x⟨, p'', ⟩z⟩) → (rev x_init, p', x_end # ⟩z⟩))
    by (cases q) auto
  then show p' ∈ states M using nxt by blast
next
  assume someq: q = Some q'
  then show q' ∈ 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' ∈ states M
      (rev (⟨x⟨, p', ⟩z⟩) → (rev x_init, a, x_end # ⟩z⟩))
      (rev x_init, a, x_end # ⟩z⟩) →∗ (rev (⟨x⟨, q', ⟩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:
  (∃ q. T None (Some q)) ⟷ ¬T None None
proof
  assume ∃ q. T None (Some q)
  then show ¬T None None
    by (metis T_None_Some_impl_reachable init_no_tr_indep)

next
  assume ¬ T None None
  then obtain q z where reach: x @ z →∗∗ (rev (⟨x⟨, q, ⟩z⟩)) by blast
  then obtain n where x @ z →^X∗(Suc n) (rev (⟨x⟨, q, ⟩z⟩))
    proof −
    from reach obtain n where ncross: x @ z →^X∗(n) (rev (⟨x⟨, q, ⟩z⟩))
      using reachable_xz_impl_crossn by blast
    moreover from this obtain m where n = Suc m
      proof −
        have ∃ m. n = Suc m
        proof (rule ccontr)
```

```
        assume ¬?thesis
        hence n = 0 by presburger
        with ncross have x @ z →^X*(0) (rev (⟨x⟨, q, ⟩z⟩)) by simp
       moreover have left_config ([], init M, ⟨x @ z⟩) unfolding left_config_def
by simp
       moreover have right_config (rev (⟨x⟨, q, ⟩z⟩)) unfolding right_config_def
by simp
        ultimately show False
          using 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 ∃ q. T None (Some q)
  proof (induction n arbitrary: q rule: less_induct)
    case (less n)
    then show ?case
    proof (cases n)
      case 0
      then obtain p p' where x @ z →^X*(0) (rev x_init, p, x_end # ⟩z⟩)
                        (rev x_init, p, x_end # ⟩z⟩) → (rev (⟨x⟨, p', ⟩z⟩))
                        (rev (⟨x⟨, p', ⟩z⟩)) →^R* (rev (⟨x⟨, q, ⟩z⟩))
      proof −
        have right_config (rev (⟨x⟨, q, ⟩z⟩)) unfolding right_config_def by simp
        from right_config_impl_ltor_cross[OF less(2) this]
        obtain p p' z' where x @ z →^X*(0) (rev x_init, p, x_end # ⟩z'⟩)
                          (rev x_init, p, x_end # ⟩z'⟩) → (rev (⟨x⟨, p', ⟩z'⟩))
                         (rev (⟨x⟨, p', ⟩z'⟩)) →^R* (rev (⟨x⟨, q, ⟩z⟩)) using 0 by metis
        moreover from this unchanged_word have ⟩z'⟩ = ⟩z⟩
       by (meson right_steps_impl_steps same_append_eq unchanged_substrings)
        ultimately show thesis using that by auto
      qed
      hence T None (Some p')
     by (metis Nil_is_append_conv init_tr left_config_def left_config_is_not_right_config
length_0_conv
         length_greater_0_conv no_crossE not_Cons_self2 x_is_init_app_end)
      then show ?thesis by blast
    next
      case (Suc k)
       then obtain p p' where Suc_k_cross: x @ z →^X*(Suc k) (rev x_init, p,
x_end # ⟩z⟩)
                        (rev x_init, p, x_end # ⟩z⟩) → (rev (⟨x⟨, p', ⟩z⟩))
                        (rev (⟨x⟨, p', ⟩z⟩)) →^R* (rev (⟨x⟨, q, ⟩z⟩))
      proof −
        have right_config (rev (⟨x⟨, q, ⟩z⟩)) unfolding right_config_def by simp
        from right_config_impl_ltor_cross[OF less(2) this]
        obtain p p' z' where x @ z →^X*(Suc k) (rev x_init, p, x_end # ⟩z'⟩)
```

```
                    (rev x_init, p, x_end # ⟩z'⟩) → (rev (⟨x⟨, p', ⟩z'⟩)
                      (rev (⟨x⟨, p', ⟩z'⟩) →^R* (rev (⟨x⟨, q, ⟩z⟩) using Suc by
metis
      moreover from this have ⟩z'⟩ = ⟩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 →^X*(Suc m) (rev (⟨x⟨, q', ⟩z⟩))
                           k = Suc m
    proof −
      from Suc_k_cross(1) obtain q' z' where k_steps: x @ z →^X*(k) (rev
(⟨x⟨, q', ⟩z'⟩))
        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 ⟩z⟩ = ⟩z'⟩
     by (metis crossn_impl_reachable reach same_append_eq unchanged_substrings)
      moreover obtain m where k = Suc m
      proof −
        have k ≠ 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 $T \equiv dfa2\_transition.T\ M$
abbreviation $left\_reachable \equiv dfa2\_transition.left\_reachable\ M$
abbreviation $left\_config \equiv dfa2\_transition.left\_config$
abbreviation $right\_config \equiv dfa2\_transition.right\_config$
abbreviation $pf\_init \equiv dfa2\_transition.x\_init$
abbreviation $pf\_end \equiv dfa2\_transition.x\_end$

abbreviation $left\_step' :: 'a\ config \Rightarrow 'a\ list \Rightarrow 'a\ config \Rightarrow bool\ (\_ \rightarrow^L (\!|\_|\!)\ \_\ 55)$ **where**
$c0 \rightarrow^L (\!|x|\!)\ c1 \equiv dfa2\_transition.left\_step\ M\ x\ c0\ c1$

abbreviation $left\_steps' :: 'a\ config \Rightarrow 'a\ list \Rightarrow 'a\ config \Rightarrow bool\ (\_ \rightarrow^L *(\!|\_|\!)\ \_\ 55)$ **where**
$c0 \rightarrow^L *(\!|x|\!)\ c1 \equiv dfa2\_transition.left\_steps\ M\ x\ c0\ c1$

abbreviation $right\_step' :: 'a\ config \Rightarrow 'a\ list \Rightarrow 'a\ config \Rightarrow bool\ (\_ \rightarrow^R (\!|\_|\!)\ \_\ 55)$ **where**
$c0 \rightarrow^R (\!|x|\!)\ c1 \equiv dfa2\_transition.right\_step\ M\ x\ c0\ c1$

abbreviation $right\_steps' :: 'a\ config \Rightarrow 'a\ list \Rightarrow 'a\ config \Rightarrow bool\ (\_ \rightarrow^R *(\!|\_|\!)\ \_\ 55)$ **where**
$c0 \rightarrow^R *(\!|x|\!)\ c1 \equiv dfa2\_transition.right\_steps\ M\ x\ c0\ c1$

abbreviation $right\_stepn' :: 'a\ config \Rightarrow 'a\ list \Rightarrow nat \Rightarrow 'a\ config \Rightarrow bool$
$(\_ \rightarrow^R '(\_,\_')\ \_\ 55)$ **where**
$c0 \rightarrow^R (x,n)\ c1 \equiv dfa2\_transition.right\_stepn\ M\ x\ c0\ n\ c1$

abbreviation $left\_reachable' :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ config \Rightarrow bool\ (\_ \rightarrow^L **(\!|\_|\!)\ \_\ 55)$ **where**
$w \rightarrow^L **(\!|x|\!)\ c \equiv dfa2\_transition.left\_reachable\ M\ x\ w\ c$

abbreviation $crossn' :: 'a\ config \Rightarrow 'a\ list \Rightarrow nat \Rightarrow 'a\ config \Rightarrow bool\ (\_ \rightarrow^X '(\_,\_')\ \_\ 55)$ **where**
$w \rightarrow^X (x,n)\ y \equiv dfa2\_transition.crossn\ M\ x\ w\ n\ y$

abbreviation $word\_crossn' :: 'a\ list \Rightarrow 'a\ list \Rightarrow nat \Rightarrow 'a\ config \Rightarrow bool\ (\_ \rightarrow^X *'(\_,\_')\ \_\ 55)$ **where**
$w \rightarrow^X *(x,\ n)\ c \equiv dfa2\_transition.word\_crossn\ M\ x\ w\ n\ c$

lemma *T_eq_impl_rconf_reachable*:
assumes *x_stepn*: $x\ @\ z \rightarrow^X *(x,n)\ (zs\ @\ rev\ (\langle x \langle),\ q,\ v)$
and *not_empty*: $x \neq []\ y \neq []$
and *T_eq*: $T\ x = T\ y$
shows $y\ @\ z \rightarrow^X *(y,n)\ (zs\ @\ rev\ (\langle y \langle),\ q,\ v)$
using *x_stepn* **proof** (*induction n arbitrary*: *zs q v rule*: *less_induct*)
case (*less n*)
have *T_axioms*: *dfa2_transition M x dfa2_transition M y* **using** *not_empty*
dfa2_axioms dfa2_transition_axioms_def dfa2_transition_def **by** *auto*

```
  have n_gt_0: n > 0
  proof (rule ccontr)
    assume ¬?thesis
    hence n = 0 by blast
    with less(2) have (x @ z →^L**(|x|) (zs @ rev (⟨x⟨(), q, v))
                  ∨ (([], init M, ⟨x @ z⟩) →^R*(|x|) (zs @ rev (⟨x⟨(), q, v))
      using T_axioms(1) dfa2_transition.no_crossE by blast
    moreover have left_config x ([], init M, ⟨x @ z⟩)
      unfolding dfa2_transition.left_config_def[OF T_axioms(1)] by simp
    moreover have right_config x (zs @ rev (⟨x⟨(), q, v)
      unfolding dfa2_transition.right_config_def[OF T_axioms(1)] by simp
    ultimately show False
     using T_axioms(1) ‹n = 0› dfa2_transition.left_config_is_not_right_config
       dfa2_transition.no_cross_impl_same_side less.prems 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 →^X*(x,m) (rev (pf_init x), p, pf_end x # ⟩z⟩)
                  (rev (pf_init x), p, pf_end x # ⟩z⟩) → (rev (⟨x⟨(), q', ⟩z⟩)
                  (rev (⟨x⟨(), q', ⟩z⟩) →^R*(|x|) (zs @ rev (⟨x⟨(), q, v)
  proof −
    have right_config x (zs @ rev (⟨x⟨(), 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 →^X*(x,m) (rev (pf_init x), p, pf_end x #
⟩z'⟩)
                  (rev (pf_init x), p, pf_end x # ⟩z'⟩) → (rev (⟨x⟨(), q', ⟩z'⟩)
                  (rev (⟨x⟨(), q', ⟩z'⟩) →^R*(|x|) (zs @ rev (⟨x⟨(), q, v)
      by (metis ‹dfa2_transition.right_config x (zs @ rev (⊢ # Σ x), q, v)›)
    moreover have ⟩z⟩ = ⟩z'⟩
    proof −
      from ltor' have x @ z →** (rev (⟨x⟨(), q', ⟩z'⟩)
        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 (⟨y⟨(), q', ⟩z⟩) →^R*(|y|) (zs @ rev (⟨y⟨(), q, v)
  proof −
     from ltor have xq'z_reach: x @ z →** (rev (⟨x⟨(), q', ⟩z⟩)
         by (meson T_axioms(1) dfa2_transition.crossn_impl_reachable rtran-
clp.rtrancl_into_rtrancl)
     from ltor(3) obtain i where rstepi:
       (rev (⟨x⟨(), q', ⟩z⟩) →^R(x,i) (zs @ rev (⟨x⟨(), q, v)
       by (meson rtranclp_imp_relpowp)
     hence stepi: (rev (⟨x⟨(), q', ⟩z⟩) →(i) (zs @ rev (⟨x⟨(), q, v)
       by (metis T_axioms(1) dfa2_transition.right_step.simps relpowp_mono)
     moreover have
       x_all_geq: ∀j≤i. ∀u' p' v'. ((rev (⟨x⟨(), q', ⟩z⟩)→(j) (u', p', v'))
```

```
            ⟶ length (rev (⟨x⟨)) ≤ length u′
    proof ((rule allI)+, rule impI)+
      fix j u′ p′ v′
      assume j_lt_i: j ≤ i
        and stepj: (rev (⟨x⟨), q′, ⟩z⟩)→(j) (u′, p′, v′)
    with dfa2_transition.right_stepn_impl_interm_right_stepn[OF T_axioms(1)]
      have (rev (⟨x⟨), q′, ⟩z⟩)→^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 length (rev (⟨x⟨)) ≤ length u′ using dfa2_transition.right_config_def[OF
T_axioms(1)]
        by auto
    qed
    ultimately have stepi_y: (rev (⟨y⟨), q′, ⟩z⟩) →(i) (zs @ rev (⟨y⟨), q, v)
                  and y_all_geq:
                  ∀ j≤i. ∀ u′ p′ v′. ((rev (⟨y⟨), q′, ⟩z⟩)→(j) (u′, p′, v′))
                    ⟶ length (rev (⟨y⟨)) ≤ 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 = (rev (⟨y⟨), q′, ⟩z⟩)
        f i = (zs @ rev (⟨y⟨), q, v)
        ∀ n<i. f n → f (Suc n)
        by (metis relpowp_fun_conv[of i (→) (rev (⟨y⟨), q′, ⟩z⟩) (zs @ rev (⟨y⟨),
q, v)])
      hence ∀ n<i. (f 0 →(n) f n)
        by (metis Suc_lessD less_trans_Suc relpowp_fun_conv)
      have rstepn_fn: ∀ n<i. ((rev (⟨y⟨), q′, ⟩z⟩) →^R(y,n) f n)
      proof (rule allI, rule impI)
        fix n
        assume n_lt_i: n < i
        then show ((rev (⟨y⟨), q′, ⟩z⟩) →^R(y,n) f n)
        proof (induction n)
          case (Suc n)
          hence rstepn: ((rev (⟨y⟨), q′, ⟩z⟩) →^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 →^R(‖y‖) f (Suc n)
          proof −
            from f_def Suc have f n → f (Suc n) by simp
            moreover from this rstepn have (rev (⟨y⟨), q′, ⟩z⟩) →(Suc n) f (Suc
n)
```

```
            using Suc.prems ‹∀ 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.prems ‹((→) ^^ 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.rtrancl_into_rtrancl)
  qed
 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* $\rightarrow^L$**(⦇*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* $\rightarrow^X$*(*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* $\rightarrow^X$*(*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* $\rightarrow^L$**(⦇*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* $\rightarrow^X$*(*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*⟩) $\rightarrow^L$*(⦇*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* $\rightarrow^X$*(*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'*⟩) $\rightarrow^L$*(⦇*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_nempty: ⟩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_nempty

        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* = ⟨*x*⟨ *xs @* ⟩*z*⟩ = *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* ≠ [] **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* →∗∗ (*rev xs @*
u, *acc M*, ⟩*z*⟩)
by (*smt* (*verit*, *ccfv_SIG*) *rtranclp_trans snoc_eq_iff_butlast x_acc_reachable*)
obtain *m* **where** *x @ z* →^{X}∗(*x*,*m*) (*rev* (⟨*x*⟨, *acc M*, ⟩*z*⟩)
proof −
from *revxsu_reach dfa2_transition.reachable_xz_impl_crossn*[*OF T_axioms*(*1*)]

obtain *m* **where** *x @ z* →^{X}∗(*x*,*m*) (*rev xs @ u*, *acc M*, ⟩*z*⟩) **by** *blast*
moreover have *rev xs @ u* = *rev* (⟨*x*⟨)
by (*metis* ‹*rev u @ xs* = ⊢ # Σ *x*› *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* →^{X}∗(*y*,*m*) (*rev* (⟨*y*⟨, *acc M*, ⟩*z*⟩)
by *auto*
hence *y @ z* →∗∗ (*rev* (⟨*y*⟨, *acc M*, ⟩*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* →^{X}∗(*x*,*n*) (*u*, *acc M*, *v*) **by** *blast*
from *right* **obtain** *zs* **where** *zs_defs*: *rev zs @ rev* (⟨*x*⟨ = *u zs @ v* = ⟩*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 →^{X}∗(*y*,*n*) (*rev zs @ rev* (⟨*y*⟨, *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* ⇒ (*state option* × *state option*) *set* **where**
$\mathcal{T}$ *x* ≡ {(*q*, *p*). *dfa2_transition M x* ∧ *T x q p*}

lemma $\mathcal{T}$*_subset_states_none*:

```
  shows 𝒯 x ⊆ ({Some q | q. q ∈ states M} ∪ {None}) × ({Some q | q. q ∈ states
M} ∪ {None})
    (is _ ⊆ ?S × _)
proof (cases x)
  case Nil
  then show ?thesis unfolding 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 × state option
    assume pq ∈ 𝒯 x
    then obtain p q where pq_def: pq = (p, q) (p, q) ∈ 𝒯 x
      by (metis surj_pair)
    have (p, q) ∈ ?S × ?S
       using pq_def(2) dfa2_transition.T_impl_in_states[OF trans] unfolding
𝒯_def
      by fast
    thus pq ∈ ?S × ?S using pq_def by simp
  qed
qed

lemma 𝒯_Nil_eq_𝒯_Nil:
  assumes 𝒯 x = 𝒯 []
  shows x = []
proof (rule ccontr)
  assume nempty: x ≠ []
  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 nempty have dfa2_transition M x
   by (simp add: dfa2_axioms dfa2_transition.intro dfa2_transition_axioms_def)
  moreover from assms have 𝒯 x = {} unfolding 𝒯_def
    by (simp add: dfa2_transition_axioms_def dfa2_transition_def)
  ultimately show False unfolding 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 ⟷ 𝒯 x = 𝒯 y
  using assms 𝒯_def by fastforce
```

definition *kern* :: $('b \Rightarrow 'c) \Rightarrow ('b \times 'b)$ *set* **where**
kern f $\equiv \{(x, y).\ f\ x = f\ y\}$

lemma *equiv_kern*:
equiv UNIV (*kern f*)
unfolding *kern_def* **by** (*simp add*: *equivI refl_on_def sym_def trans_def eq_app_right_def*)

lemma *inj_on_vimage_image*: *inj_on* ($\lambda b.\ f\ -`\ \{b\}$) (*f* ` *A*)
using *inj_on_def* **by** *fastforce*

lemma *kern_Image*: *kern f* `` *A* = *f* $-`$ (*f* ` *A*)
unfolding *kern_def* **by** (*auto simp*: *rev_image_eqI*)

lemma *quotient_kern_eq_image*: *A* // *kern f* = ($\lambda b.\ f-`\ \{b\}$) ` *f* ` *A*
by (*auto simp*: *quotient_def kern_Image*)

lemma *bij_betw_image_quotient_kern*:
bij_betw ($\lambda b.\ f\ -`\ \{b\}$) (*f* ` *A*) (*A* // *kern f*)
unfolding *bij_betw_def*
by (*simp add*: *inj_on_vimage_image quotient_kern_eq_image*)

lemma *finite_quotient_kern_iff_finite_image*:
finite (*A* // *kern* $\mathcal{T}$) = *finite* ($\mathcal{T}$ ` *A*)
by (*metis bij_betw_finite bij_betw_image_quotient_kern*)

theorem $\mathcal{T}$*_finite_image*:
finite ($\mathcal{T}$ ` *UNIV*)
proof −
let *?S* = {*Some q* | *q*. *q* ∈ *states M*} ∪ {*None*}
have *finite_state_options*: *finite ?S* **using** *finite* **by** *simp*
hence $\mathcal{T}$ ` *UNIV* ⊆ *Pow* (*?S* × *?S*) **using** $\mathcal{T}$*_subset_states_none* **by** *blast*
moreover have *finite* (*Pow* (*?S* × *?S*)) **using** *finite_state_options* **by** *simp*
ultimately show *finite* ($\mathcal{T}$ ` *UNIV*) **by** (*simp add*: *finite_subset*)
qed

lemma *kern_*$\mathcal{T}$*_subset_eq_app_right*:
kern $\mathcal{T}$ ⊆ *eq_app_right Lang*
proof
fix *xy*
assume *xy* ∈ *kern* $\mathcal{T}$
then obtain *x y* **where** *xy_def*: *xy* = (*x*, *y*) $\mathcal{T}$ *x* = $\mathcal{T}$ *y*
unfolding *kern_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 𝒯_Nil_eq_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_𝒯_eq 𝒯_Nil_eq_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 𝒯_finite_image have finite (UNIV // kern 𝒯)
    using finite_quotient_kern_iff_finite_image by blast
  then have finite (UNIV // eq_app_right Lang)
   using equiv_kern equiv_eq_app_right finite_refines_finite kern_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 ⇒ 'a dfa2 ⇒ 'a config ⇒ bool (_ →(⦇_⦈) _ 55)
where
  c0 →(⦇M⦈) c1 ≡ dfa2.step M c0 c1

abbreviation steps' :: 'a config ⇒ 'a dfa2 ⇒ 'a config ⇒ bool (_ →*(⦇_⦈) _ 55)
where
  c0 →*(⦇M⦈) c1 ≡ dfa2.steps M c0 c1


lemma finite_arbitrarily_large_disj:
  ⟦ infinite(UNIV::'a set); finite (A::'a set) ⟧ ⟹ ∃ B. finite B ∧ card B = n ∧ A
∩ 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

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

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 same state M is in after reading w. It is worth noting that, since the markers aren't in Σ, M' will not reach $\dashv$ while simulating the execution of M. Hence, if the state of M' when reading $\dashv$ is in F, M accepts w, and M' goes into its accepting state, q_a. Otherwise, it goes into its rejecting state q_r. At this point, the simulation of M is over, and M' behaves in accordance to the formal definition of 2DFAs. In particular, it always remains in its current state, and it moves to the right for all symbols except for $\dashv$.
We now formally prove that $L(M') = L(M)$:

theorem *regular_language_impl_dfa2*:
  assumes *regular L*
  obtains *M M' q0 qa qr* **where**
    dfa M dfa.language M = L
    {q0, qa, qr} ∩ dfa.states M = {}
    qa ≠ qr
    qa ≠ q0
    qr ≠ q0
    dfa2 M' dfa2.Lang M' = L
    M' = (let δ = (λq a. case a of

[1] We define automata in accordance with the records *'a 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.nxt 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)))
        in (|dfa2.states = dfa.states M ∪ {q0, qa, qr},
            dfa2.init = q0,
            dfa2.acc = qa,
            dfa2.rej = qr,
            dfa2.nxt = δ|))
proof −
  from assms obtain M where M_def: dfa M dfa.language M = L
    unfolding regular_def by blast
  then obtain q0 qa qr where q_defs:
    {q0, qa, qr} ∩ dfa.states M = {}
    qa ≠ qr
    qa ≠ q0
    qr ≠ q0
  proof −
    from dfa.finite[OF ‹dfa M›] obtain Q where
      card Q = Suc (Suc (Suc 0))
      Q ∩ dfa.states M = {}
      using infinite_UNIV_state
      by (metis finite_arbitrarily_large_disj inf_commute)
    thus thesis using that distinct_def
      by (smt (verit, ccfv_SIG) card_Suc_eq insertCI)
  qed

  let ?δ = (λq a. case a of
          Letter a' ⇒ (if q ∈ dfa.states M then ((dfa.nxt 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.nxt = ?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.nxt 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.nxt[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 auto
  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 ∈ dfa.states M
      show (u, q, ⟩w⟩) →*⦇?M'⦈ (rev (Σ 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: ‹dfa M› dfa.nextl.simps(1))
        moreover from this have (u, q, ⟩[]⟩) = (rev (Σ []) @ u, dfa.nextl M q [],
[⊣]) by simp
        ultimately show ?case by simp
      next
        case (Cons x xs)
        from Cons(2) have step1: (u, q, ⟩x # xs⟩) →⦇?M'⦈ (Letter x # u, dfa.nxt
M q x, ⟩xs⟩)
                          dfa.nxt M q x ∈ dfa.states M
          using M.step.simps by (auto simp add: ‹dfa M› dfa.nxt)
        with Cons(1) have
          (Letter x # u, dfa.nxt M q x, ⟩xs⟩)
            →*⦇?M'⦈ (rev (Σ xs) @ (Letter x # u), dfa.nextl M (dfa.nxt M q x) xs,
[⊣])
```

```
      by simp
    moreover have ... = (rev (Σ (x # xs)) @ u, dfa.nextl M q (x # xs), [⊣])
      by (simp add: ‹dfa M› dfa.nextl.simps(2))
    ultimately show ?case using step1 by auto
   qed
  qed
  hence steps: ([⊢], dfa.init M, ⟩w⟩) →*⦇?M'⦈ (rev (⟨w⟨), (dfa.nextl M (dfa.init
M) w, [⊣]))
  proof −
   have dfa.init M ∈ dfa.states M using ‹dfa M› dfa.init by blast
   with reach show ?thesis by simp
  qed
  with step show ([], dfa2.init ?M', ⟨w⟩) →*⦇?M'⦈ (rev (⟨w⟨), (dfa.nextl M
(dfa.init M) w, [⊣]))
   by simp
 qed

 have M.Lang = dfa.language M
 proof
  show M.Lang ⊆ dfa.language M
  proof
   fix w
   assume w ∈ M.Lang
  then obtain u v where acc_reachable: ([], dfa2.init ?M', ⟨w⟩) →*⦇?M'⦈ (u,
dfa2.acc ?M', v)
    using M.Lang_def by blast
   from nextl_reachable obtain q where final_state:
    ([], dfa2.init ?M', ⟨w⟩) →*⦇?M'⦈ (rev (⟨w⟨), q, [⊣])
    q ∈ dfa.states M  dfa.nextl M (dfa.init M) w = q
    using ‹dfa M› dfa.nextl_init_state by blast
   with acc_reachable have acc_step:
    (rev (⟨w⟨), q, [⊣]) →⦇?M'⦈ (tl (rev (⟨w⟨)), dfa2.acc ?M', hd (rev (⟨w⟨)) #
[⊣])
   proof −
    have disj: ((rev (⟨w⟨), q, [⊣]) →⦇?M'⦈ (tl (rev (⟨w⟨)), dfa2.acc ?M', hd (rev
(⟨w⟨)) # [⊣]))
          ∨ ((rev (⟨w⟨), q, [⊣]) →⦇?M'⦈ (tl (rev (⟨w⟨)), dfa2.rej ?M', hd (rev
(⟨w⟨)) # [⊣]))
        (is ?acc_step ∨ ?rej_step)
     proof (cases q ∈ dfa.final M)
      case True
      hence nxt ?M' q ⊣ = (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 ≠ dfa2.acc ?M' by auto
      ultimately have nxt ?M' q ⊣ = (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 ¬?thesis
        with disj have ?rej_step by blast
        hence rej_reachable:
          ([], dfa2.init ?M', ⟨w⟩) →*⦇?M'⦈ (tl (rev (⟨w⟩), dfa2.rej ?M', hd (rev
(⟨w⟩)) # [⊣])
          using final_state by auto
        with acc_reachable consider
        ((tl (rev (⟨w⟩), dfa2.rej ?M', hd (rev (⟨w⟩)) # [⊣]) →*⦇?M'⦈ (u, dfa2.acc
?M', v)) |
          (u, dfa2.acc ?M', v) →*⦇?M'⦈ (tl (rev (⟨w⟩), dfa2.rej ?M', hd (rev
(⟨w⟩)) # [⊣])
          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 ∈ dfa.final M
    proof (rule ccontr)
      assume q ∉ dfa.final M
      with final_state(2) have dfa2.nxt ?M' q ⊣ = (dfa2.rej ?M', Left)
        using q_defs(1) by auto
      with acc_step show False using q_defs(2) by fastforce
    qed
    thus w ∈ dfa.language M
      using final_state(3) dfa.language_def[OF ‹dfa M›] by blast
  qed
 next
  show dfa.language M ⊆ M.Lang
  proof
    fix w
    assume w ∈ dfa.language M
     then obtain q where dfa.nextl M (dfa.init M) w = q and in_final: q ∈
dfa.final M
      by (simp add: ‹dfa M› dfa.language_def)
     with nextl_reachable have ([], dfa2.init ?M', ⟨w⟩) →*⦇?M'⦈ (rev (⟨w⟩), q,
[⊣])
      by blast
     also from this in_final have ... →⦇?M'⦈ (tl (rev (⟨w⟩)), dfa2.acc ?M', hd
(rev (⟨w⟩)) # [⊣])
      using M.step.simps by auto
    finally show w ∈ M.Lang unfolding M.Lang_def by blast
  qed
 qed
 then show thesis using that ‹dfa.language M = L› M.dfa2_axioms M_def
q_defs by presburger
```

qed

The equality follows trivially:

corollary *dfa2_accepts_regular_languages*:
  regular L = (∃ *M*. *dfa2 M* ∧ *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.