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*⟩ ⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *steps_impl_in_states*:
  assumes *p* ∈ *states M*
  shows (*u*, *p*, *v*) →* (*u'*, *q*, *v'*) ⟹ *q* ∈ *states M*
⟨*proof*⟩

corollary *reachable_impl_in_states*:
  w →** (*u*, *q*, *v*) ⟹ *q* ∈ *states M*
⟨*proof*⟩

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*⟩

lemma *unchanged_final*:
assumes $p = acc\ M \vee p = rej\ M$
shows $(u,\ p,\ v) \rightarrow\!* (u',\ q,\ v') \Longrightarrow p = q$
⟨*proof*⟩

corollary *unchanged_word*:
$([],\ p,\ w) \rightarrow\!* (u,\ q,\ v) \Longrightarrow w = rev\ u\ @\ v$
⟨*proof*⟩

lemma *step_tl_indep*:
assumes $(u,\ p,\ w\ @\ v) \rightarrow (y,\ q,\ z\ @\ v)$
$w \neq []$
shows $(u,\ p,\ w\ @\ v') \rightarrow (y,\ q,\ z\ @\ v')$
⟨*proof*⟩

lemma *steps_app* [*simp*, *intro*]:
$(u,\ p,\ v) \rightarrow\!* (u',\ q,\ v') \Longrightarrow (u,\ p,\ v\ @\ xs) \rightarrow\!* (u',\ q,\ v'\ @\ xs)$
⟨*proof*⟩

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

lemma *acc_impl_reachable_substring*:
assumes $w \rightarrow\!\!* (u,\ acc\ M,\ v)$
$xs \neq []$
$ys \neq []$
shows $v = xs\ @\ ys \Longrightarrow (u,\ acc\ M,\ v) \rightarrow\!* (rev\ xs\ @\ u,\ acc\ M,\ ys)$
⟨*proof*⟩

lemma *all_geq_left_impl_left_indep*:
assumes *upv_reachable*: $w \rightarrow\!\!* (u,\ p,\ v)$
and $(u,\ p,\ v) \rightarrow(n)\ (vs\ @\ u,\ q,\ x)$
$\forall\, i{\leq}n.\ \forall\, u'\ p'\ v'.\ ((u,\ p,\ v) \rightarrow(i)\ (u',\ p',\ v')) \longrightarrow length\ u' \geq length\ u$
shows $((u',\ p,\ v) \rightarrow(n)\ (vs\ @\ u',\ q,\ x))$
$\wedge\ (\forall\, i{\leq}n.\ \forall\, y\ p'\ v'.\ ((u',\ p,\ v) \rightarrow(i)\ (y,\ p',\ v')) \longrightarrow length\ y \geq length\ u')$
⟨*proof*⟩

lemma *reachable_configs_impl_reachable*:
assumes $c0 \rightarrow\!* c1$
$c0 \rightarrow\!* c2$
shows $c1 \rightarrow\!* c2 \vee c2 \rightarrow\!* c1$

⟨*proof*⟩

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*:
  ⟨*x*⟨ = *x_init* @ [*x_end*] ⟨*proof*⟩

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 ≡ ∃ *u q v*. *c* = (*u*, *q*, *v*) ∧ *length u* < *length* (⟨*x*⟨)

definition *right_config* :: *'a config* ⇒ *bool* **where**
  right_config c ≡ ∃ *u q v*. *c* = (*u*, *q*, *v*) ∧ *length u* ≥ *length* (⟨*x*⟨)

lemma *left_config_is_not_right_config*:
  left_config c ⟷ ¬*right_config c*
⟨*proof*⟩

lemma *left_config_lt_right_config*:
  ⟦ *left_config* (*u*, *p*, *v*); *right_config* (*w*, *q*, *y*) ⟧ ⟹ *length u* < *length w*
⟨*proof*⟩

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** ‹$\rightarrow^L$› *55*) **where**
lstep [*intro*]: ⟦*c0* → *c1*; *left_config c0*; *left_config c1*⟧ ⟹ *c0* $\rightarrow^L$ *c1*

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

abbreviation *left_steps* :: *'a config* ⇒ *'a config* ⇒ *bool* (**infix** ‹$\rightarrow^L*$› *55*) **where**
left_steps ≡ *left_step***

abbreviation *left_stepn* :: *'a config* ⇒ *nat* ⇒ *'a config* ⇒ *bool* (_ $\rightarrow^L$'(_') _ *55*) **where**
left_stepn c n ≡ (*left_step* ^^ *n*) *c*

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

abbreviation *left_reachable* :: *'a list* ⇒ *'a config* ⇒ *bool* (**infix** ‹$\rightarrow^L$**› *55*) **where**
w $\rightarrow^L$** *c* ≡ ([], *init M*, ⟨*w*⟩) $\rightarrow^L$* *c*

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

Right steps are defined analogously:

inductive *right_step* :: *'a config* ⇒ *'a config* ⇒ *bool* (**infix** ‹$\rightarrow^R$› *55*) **where**
rstep [*intro*]: ⟦*c0* → *c1*; *right_config c0*; *right_config c1*⟧ ⟹ *c0* $\rightarrow^R$ *c1*

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

abbreviation *right_steps* :: *'a config* ⇒ *'a config* ⇒ *bool* (**infix** ‹$\rightarrow^R$*› *55*) **where**
right_steps ≡ *right_step***

abbreviation *right_stepn* :: *'a config* ⇒ *nat* ⇒ *'a config* ⇒ *bool* (_ $\rightarrow^R$'(_') _ *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 $\rightarrow^L$* *c1* ⟹ *c0* →* *c1*
⟨*proof*⟩

lemma *right_steps_impl_steps* [*dest*]:
c0 $\rightarrow^R$* *c1* ⟹ *c0* →* *c1*
⟨*proof*⟩

lemma *left_steps_impl_left_config*[*dest*]:
⟦*c0* $\rightarrow^L$* *c1*; *left_config c0*⟧ ⟹ *left_config c1*
⟨*proof*⟩

lemma *left_steps_impl_left_config_conv*[*dest*]:
$[\![c0 \rightarrow^L{*}\ c1;\ left\_config\ c1]\!] \Longrightarrow left\_config\ c0$
⟨*proof*⟩

lemma *left_reachable_impl_left_config*:
$w \rightarrow^L{**}\ c \Longrightarrow left\_config\ c$
⟨*proof*⟩

lemma *right_steps_impl_right_config*[*dest*]:
$[\![c0 \rightarrow^R{*}\ c1;\ right\_config\ c0]\!] \Longrightarrow right\_config\ c1$
⟨*proof*⟩

lemma *left_step_tl_indep*:
$[\![\ (u,\ p,\ w\ @\ v) \rightarrow^L (y,\ q,\ z\ @\ v);\ w \neq []\ ]\!] \Longrightarrow (u,\ p,\ w\ @\ v') \rightarrow^L (y,\ q,\ z\ @\ v')$
⟨*proof*⟩

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*⟩

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

lemma *list_deconstruct1*:
assumes $m \leq length\ xs$
obtains *ys zs* **where** $length\ ys = m\ ys\ @\ zs = xs$ ⟨*proof*⟩

lemma *list_deconstruct2*:
assumes $m \leq length\ xs$
obtains *ys zs* **where** $length\ zs = m\ ys\ @\ zs = xs$
⟨*proof*⟩

lemma *lstar_impl_substring_x*:
assumes *app_eq*: $rev\ u\ @\ v = \langle x\ @\ z\rangle$
and *in_x*: $length\ u < length\ (\langle x\langle)$
and *lsteps*: $(u,\ p,\ v) \rightarrow^L{*}\ (u',\ q,\ v')$
obtains *y* **where** $rev\ u'\ @\ y = \langle x\langle\ y\ @\ \rangle z\rangle = v'$
⟨*proof*⟩

corollary *left_reachable_impl_substring_x*:
assumes $x\ @\ z \rightarrow^L{**}\ (u,\ q,\ v)$
obtains *y* **where** $rev\ u\ @\ y = \langle x\langle\ y\ @\ \rangle z\rangle = v$
⟨*proof*⟩

corollary *reachable_lconfig_impl_substring_x*:
assumes $x\ @\ z \rightarrow{**}\ (u,\ p,\ v)$
and $length\ u < length\ (\langle x\langle)$
and $(u,\ p,\ v) \rightarrow^L{*}\ (u',\ q,\ v')$

obtains y **where** $rev\ u' \mathbin{@} y = \langle x\langle\ y \mathbin{@} \rangle z\rangle = v'$
⟨*proof*⟩

lemma *star_rconfig_impl_substring_z*:
assumes *app_eq*: $x \mathbin{@} z \rightarrow** (u,\ p,\ v)$
and *reach*: $(u,\ p,\ v) \rightarrow* (u',\ q,\ v')$
and *rconf*: *right_config* $(u',\ q,\ v')$
obtains y **where** $rev\ (\langle x\langle \mathbin{@} y) = u'\ y \mathbin{@} v' = \rangle z\rangle$
⟨*proof*⟩

corollary *reachable_right_conf_impl_substring_z*:
assumes $x \mathbin{@} z \rightarrow** (u,\ q,\ v)$
right_config $(u,\ q,\ v)$
obtains y **where** $rev\ (\langle x\langle \mathbin{@} y) = u\ y \mathbin{@} v = \rangle z\rangle$
⟨*proof*⟩

lemma *lsteps_indep*:
assumes $(u,\ p,\ v \mathbin{@} \rangle z\rangle) \rightarrow^L* (w,\ q,\ y \mathbin{@} \rangle z\rangle)$
$rev\ u \mathbin{@} v \mathbin{@} \rangle z\rangle = \langle x \mathbin{@} z\rangle$
shows $(u,\ p,\ v \mathbin{@} \rangle z'\rangle) \rightarrow^L* (w,\ q,\ y \mathbin{@} \rangle z'\rangle)$
⟨*proof*⟩

lemma *left_reachable_indep*:
assumes $x \mathbin{@} y \rightarrow^L** (u,\ q,\ v \mathbin{@} \rangle y\rangle)$
shows $x \mathbin{@} z \rightarrow^L** (u,\ q,\ v \mathbin{@} \rangle z\rangle)$
⟨*proof*⟩

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: $[\![left\_config\ c0;\ c0 \rightarrow^L* c1]\!] \Longrightarrow c0 \rightarrow^X(0)\ c1$ |
no_crossr: $[\![right\_config\ c0;\ c0 \rightarrow^R* c1]\!] \Longrightarrow c0 \rightarrow^X(0)\ c1$ |
crossn_rtol: $[\![c0 \rightarrow^X(n)\ (rev\ (\langle x\langle, p, \rangle z\rangle);$
$(rev\ (\langle x\langle, p, \rangle z\rangle) \rightarrow (rev\ x\_init,\ q,\ x\_end\ \#\ \rangle z\rangle);$
$(rev\ x\_init,\ q,\ x\_end\ \#\ \rangle z\rangle) \rightarrow^L* c1]\!] \Longrightarrow c0 \rightarrow^X(Suc\ n)\ c1$ |
crossn_ltor: $[\![\ c0 \rightarrow^X(n)\ (rev\ x\_init,\ p,\ x\_end\ \#\ \rangle z\rangle);$
$(rev\ x\_init,\ p,\ x\_end\ \#\ \rangle z\rangle) \rightarrow (rev\ (\langle x\langle, q, \rangle z\rangle);$
$(rev\ (\langle x\langle, q, \rangle z\rangle) \rightarrow^R* c1]\!] \Longrightarrow c0 \rightarrow^X(Suc\ n)\ c1$

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**
$word\_crossn\ w\ n\ c \equiv ([],\ init\ M,\ \langle w\rangle) \rightarrow^X(n)\ c$

lemma *self_nocross*[*simp*]:
$c \rightarrow^X (0)\ c$ ⟨*proof*⟩

lemma *no_cross_impl_same_side*:
$c0 \rightarrow^X (0)\ c1 \Longrightarrow$ *left_config c0* = *left_config c1*
⟨*proof*⟩

lemma *left_config_impl_rtol_cross*:
assumes $c0 \rightarrow^X (Suc\ n)\ c1$
left_config c1
obtains *p q z* **where** $c0 \rightarrow^X (n)$ $(rev\ (\langle x\langle, p, \rangle z\rangle))$
$(rev\ (\langle x\langle, p, \rangle z\rangle)) \rightarrow (rev\ x\_init, q, x\_end\ \#\ \rangle z\rangle)$
$(rev\ x\_init, q, x\_end\ \#\ \rangle z\rangle) \rightarrow^L * c1$
⟨*proof*⟩

lemma *right_config_impl_ltor_cross*:
assumes $c0 \rightarrow^X (Suc\ n)\ c1$
right_config c1
obtains *p q z* **where** $c0 \rightarrow^X (n)$ $(rev\ x\_init, p, x\_end\ \#\ \rangle z\rangle)$
$(rev\ x\_init, p, x\_end\ \#\ \rangle z\rangle) \rightarrow (rev\ (\langle x\langle, q, \rangle z\rangle))$
$(rev\ (\langle x\langle, q, \rangle z\rangle)) \rightarrow^R * c1$
⟨*proof*⟩

lemma *crossn_decompose*:
assumes $c0 \rightarrow^X (Suc\ n)\ c2$
obtains *c1* **where** $c0 \rightarrow^X (n)\ c1\ c1 \rightarrow^X (Suc\ 0)\ c2$
⟨*proof*⟩

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*⟩

lemma *crossn_no_cross_eq_crossn*:
assumes $c0 \rightarrow^X (n)\ c1$
$c1 \rightarrow^X (0)\ c2$
shows $c0 \rightarrow^X (n)\ c2$
⟨*proof*⟩

lemma *crossn_trans*:
assumes $c0 \rightarrow^X (n)\ c1$
shows $c1 \rightarrow^X (m)\ c2 \Longrightarrow c0 \rightarrow^X (n+m)\ c2$
⟨*proof*⟩

lemma *crossn_impl_reachable*:
assumes $c0 \rightarrow^X (n)\ c1$
shows $c0 \rightarrow *\ c1$

⟨*proof*⟩

lemma *reachable_xz_impl_crossn*:
assumes *c0* $\rightarrow*$ *c1*
c0 = (*u*, *p*, *v*)
rev u @ *v* = ⟨*x* @ *z*⟩
obtains *n* **where** *c0* $\rightarrow^{X}(n)$ *c1*
⟨*proof*⟩

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* $\rightarrow**$ (*rev* (⟨*x*⟨, *q*, ⟩*z*⟩) ⟹ *T None None* |

some_tr: ⟦ *p′* ∈ *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′* ∈ *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*⟩)

$(rev\ x\_init,\ p,\ x\_end\ \#\ \rangle z\rangle) \rightarrow (rev\ (\langle x\langle),\ q,\ \rangle z\rangle)$
⟨*proof*⟩

lemma *init_no_tr_indep*:
$T\ None\ None \Longrightarrow \nexists\, q.\ x\ @\ z \rightarrow** (rev\ (\langle x\langle),\ q,\ \rangle z\rangle)$
⟨*proof*⟩

lemma *some_tr_indep*:
assumes $T\ (Some\ p)\ (Some\ q)$
obtains q' **where** $(rev\ x\_init,\ p,\ x\_end\ \#\ \rangle z\rangle) \rightarrow^L* (rev\ x\_init,\ q',\ x\_end\ \#\ \rangle z\rangle)$
$(rev\ x\_init,\ q',\ x\_end\ \#\ \rangle z\rangle) \rightarrow (rev\ (\langle x\langle),\ q,\ \rangle z\rangle)$
⟨*proof*⟩

lemma *T_None_Some_impl_reachable*:
assumes $T\ None\ (Some\ q)$
shows $x\ @\ z \rightarrow** (rev\ (\langle x\langle),\ q,\ \rangle z\rangle)$
⟨*proof*⟩

lemma *T_impl_in_states*:
assumes $T\ p\ q$
shows $p = Some\ p' \Longrightarrow p' \in states\ M$
$q = Some\ q' \Longrightarrow q' \in states\ M$
⟨*proof*⟩

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*⟩

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

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

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

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

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

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

abbreviation *word_crossn'* :: *'a list* ⇒ *'a list* ⇒ *nat* ⇒ *'a config* ⇒ *bool* (*_ →^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 $y\ @\ z \rightarrow^X\!*(y,n)\ (zs\ @\ rev\ (\langle y\langle),\ q,\ v)$
⟨*proof*⟩

The initial implication:

theorem *T_eq_impl_eq_app_right*:
assumes *not_empty*: $x \neq []\ y \neq []$
and *T_eq*: $T\ x = T\ y$
and *xz_in_lang*: $x\ @\ z \in Lang$
shows $y\ @\ z \in Lang$
⟨*proof*⟩

There are finitely many transitions:

definition $\mathcal{T}$ :: *'a list* ⇒ (*state option* × *state option*) *set* **where**
$\mathcal{T}\ x \equiv \{(q,\ p).\ dfa2\_transition\ M\ x \wedge T\ x\ q\ p\}$

lemma $\mathcal{T}\_subset\_states\_none$:
shows $\mathcal{T}\ x \subseteq (\{Some\ q \mid q.\ q \in states\ M\} \cup \{None\}) \times (\{Some\ q \mid q.\ q \in states\ M\} \cup \{None\})$
(**is** $\_ \subseteq ?S \times \_$)
⟨*proof*⟩

lemma $\mathcal{T}\_Nil\_eq\_\mathcal{T}\_Nil$:
assumes $\mathcal{T}\ x = \mathcal{T}\ []$
shows $x = []$
⟨*proof*⟩

lemma $T\_eq\_is\_\mathcal{T}\_eq$:
assumes *dfa2_transition M x*
dfa2_transition M y
shows $T\ x = T\ y \longleftrightarrow \mathcal{T}\ x = \mathcal{T}\ y$
⟨*proof*⟩

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

lemma *equiv_kern*:
equiv UNIV (*kern f*)
⟨*proof*⟩

lemma *inj_on_vimage_image*: *inj_on* ($\lambda b.\ f\ -`\ \{b\}$) ($f\ `\ A$)
⟨*proof*⟩

lemma *kern_Image*: *kern f* `` *A* = *f* −` (*f* ` *A*)
⟨*proof*⟩

lemma *quotient_kern_eq_image*: *A* // *kern f* = ($\lambda b.\ f-`\ \{b\}$) ` *f* ` *A*
⟨*proof*⟩

lemma *bij_betw_image_quotient_kern*:
bij_betw ($\lambda b.\ f\ -`\ \{b\}$) (*f* ` *A*) (*A* // *kern f*)
⟨*proof*⟩

lemma *finite_quotient_kern_iff_finite_image*:
finite (*A* // *kern* $\mathcal{T}$) = *finite* ($\mathcal{T}$ ` *A*)
⟨*proof*⟩

theorem $\mathcal{T}\_finite\_image$:
finite ($\mathcal{T}$ ` *UNIV*)
⟨*proof*⟩

lemma $kern\_\mathcal{T}\_subset\_eq\_app\_right$:

kern $\mathcal{T} \subseteq$ *eq_app_right Lang*
⟨*proof*⟩

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

theorem *dfa2_Lang_regular*:
regular Lang
⟨*proof*⟩

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* = {}
⟨*proof*⟩

lemma *infinite_UNIV_state*: *infinite*(*UNIV* :: *state set*)
⟨*proof*⟩

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

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

[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'.

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
          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*⟩

The equality follows trivially:

corollary *dfa2_accepts_regular_languages*:
  regular L = (∃ M. dfa2 M ∧ dfa2.Lang M = L)
⟨*proof*⟩

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.