Labeled Transition Systems

Anders Schlichtkrull, Morten Konggaard Schou, Jiří Srba and Dmitriy Traytel

Abstract

Labeled transition systems are ubiquitous in computer science. They are used e.g. for automata and for program graphs in program analysis. We formalize labeled transition systems with and without epsilon transitions. The main difference between formalizations of labeled transition systems is in their choice of how to represent the transition system. In the present formalization the set of nodes is a type, and a labeled transition system is represented as a locale fixing a set of transitions where each transition is a triple of respectively a start node, a label and an end node. Wimmer [Wim20] provides an overview of formalizations of graphs and transition systems.

Contents

theory *LTS* **imports** *Main "HOL−Library.Multiset_Order"* **begin**

1 LTS

1.1 Transitions

type-synonym ('*state*, '*label*) *transition* = "'*state* × '*label* × '*state*"

1.2 LTS functions

fun *trans_hd* :: "('*state*, '*label*) *transition* ⇒ '*state*" **where**
"*trans_hd* (*s1*,γ,*s2*) = *s1*"

fun *trans_tl* :: "('*state*, '*label*) *transition* ⇒ '*state*" **where**
"*trans_tl* (*s1*,γ,*s2*) = *s2*"

fun *transitions_of* :: "'*state list* ∗ '*label list* ⇒ ('*state*, '*label*) *transition multiset*" **where**
"*transitions_of* (*s1*#*s2*#*ss*, γ#*w*) = {# (*s1*, γ, *s2*) #} + *transitions_of* (*s2*#*ss*, *w*)"
| "*transitions_of* ([*s1*],_) = {#}"
| "*transitions_of* ([],_) = {#}"
| "*transitions_of* (_,[]) = {#}"

fun *transition_list* :: "'*state list* ∗ '*label list* ⇒ ('*state*, '*label*) *transition list*" **where**
"*transition_list* (*s1*#*s2*#*ss*, γ#*w*) = (*s1*, γ, *s2*) # (*transition_list* (*s2*#*ss*, *w*))"
| "*transition_list* ([*s1*],_) = []"
| "*transition_list* ([],_) = []"
| "*transition_list* (_,[]) = []"

fun *transition_list'* :: "'*state* ∗ '*label list* ∗ '*state list* ∗ '*state* ⇒ ('*state*, '*label*) *transition list*" **where**
"*transition_list'* (*p*,*w*,*ss*,*q*) = *transition_list* (*ss*, *w*)"

fun *transitions_of'* :: "'*state* ∗ '*label list* ∗ '*state list* ∗ '*state* ⇒ ('*state*, '*label*) *transition multiset*" **where**
"*transitions_of'* (*p*,*w*,*ss*,*q*) = *transitions_of* (*ss*, *w*)"

fun *transition_list_of'* **where**
"*transition_list_of'* (*p*,γ#*w*,*p'*#*p''*#*ss*,*q*) = (*p*, γ, *p''*)#(*transition_list_of'* (*p''*,*w*,*p''*#*ss*,*q*))"
| "*transition_list_of'* (*p*, [], _, *p''*) = []"
| "*transition_list_of'* (*p*, _, [], *p''*) = []"
| "*transition_list_of'* (*v*, *va* # *vc*, [*vf*], *ve*) = []"

fun *append_path_with_word* :: "('*a list* × '*b list*) ⇒ ('*a list* × '*b list*) ⇒ ('*a list* × '*b list*)" (**infix** "@´" *65*) **where**
"(*ss1*,*w1*) @´ (*ss2*,*w2*) = (*ss1*@(*tl ss2*), *w1* @ *w2*)"

fun *append_path_with_word_γ* :: "(('*a list* × '*b list*) ∗ '*b*) ⇒ ('*a list* × '*b list*) ⇒ ('*a list* × '*b list*)" (**infix** "@$^{\gamma}$" *65*) **where**
"((*ss1*,*w1*),γ) @$^{\gamma}$ (*ss2*,*w2*) = (*ss1*@*ss2*, *w1* @ [γ] @ *w2*)"

fun *append_trans_star_states* :: "('*a* × '*b list* × '*a list* × '*a*) ⇒ ('*a* × '*b list* × '*a list* × '*a*) ⇒ ('*a* × '*b list* × '*a list* × '*a*)" (**infix** "@@´" *65*) **where**
"(*p1*,*w1*,*ss1*,*q1*) @@´ (*p2*,*w2*,*ss2*,*q2*) = (*p1*, *w1* @ *w2*, *ss1*@(*tl ss2*), *q2*)"

fun *append_trans_star_states_γ* :: "(('*a* × '*b list* × '*a list* × '*a*) ∗ '*b*) ⇒ ('*a* × '*b list* × '*a list* × '*a*) ⇒ ('*a* × '*b list* × '*a list* × '*a*)" (**infix** "@@$^{\gamma}$" *65*) **where**
"((*p1*,*w1*,*ss1*,*q1*),γ) @@$^{\gamma}$ (*p2*,*w2*,*ss2*,*q2*) = (*p1*, *w1* @ [γ] @ *w2*, *ss1*@*ss2*, *q2*)"

definition *inters* :: "('*state*, '*label*) *transition set* ⇒ ('*state*, '*label*) *transition set* ⇒ (('*state* ∗ '*state*), '*label*) *transition set*" **where**
"*inters ts1 ts2* = {((*p1*, *q1*), α, (*p2*, *q2*)). (*p1*, α, *p2*) ∈ *ts1* ∧ (*q1*, α, *q2*) ∈ *ts2*}"

definition *inters_finals* :: "'*state set* ⇒ '*state set* ⇒ ('*state* ∗ '*state*) *set*" **where**
"*inters_finals finals1 finals2* = *finals1* × *finals2*"

lemma *inters_code*[*code*]:

"inters ts1 ts2 = (⋃(p1, α, p2) ∈ ts1. ⋃(q1, α', q2) ∈ ts2. if α = α' then {((p1, q1), α, (p2, q2))} else {})"
unfolding *inters_def* **by** (*force split*: *if_splits*)

1.3 LTS locale

locale *LTS* =
fixes *transition_relation* :: *"('state, 'label) transition set"*
begin

More definitions.

definition *step_relp* :: *"'state ⇒ 'state ⇒ bool"* (**infix** *"⇒"* *80*) **where**
"c ⇒ c' ⟷ (∃ l. (c, l, c') ∈ transition_relation)"

abbreviation *step_starp* :: *"'state ⇒ 'state ⇒ bool"* (**infix** *"⇒*"* *80*) **where**
"c ⇒ c' ≡ step_relp** c c'"*

definition *step_rel* :: *"'state rel"* **where**
"step_rel = {(c, c'). step_relp c c'}"

definition *step_star* :: *"'state rel"* **where**
"step_star = {(c, c'). step_starp c c'}"

definition *post_star* :: *"'state set ⇒ 'state set"* **where**
"post_star C = {c'. ∃ c ∈ C. c ⇒ c'}"*

definition *pre_star* :: *"'state set ⇒ 'state set"* **where**
"pre_star C = {c'. ∃ c ∈ C. c' ⇒ c}"*

inductive-set *path* :: *"'state list set"* **where**
"[s] ∈ path"
| *"(s'#ss) ∈ path ⟹ (s,l,s') ∈ transition_relation ⟹ s#s'#ss ∈ path"*

inductive-set *trans_star* :: *"('state * 'label list * 'state) set"* **where**
trans_star_refl[*iff*]:
"(p, [], p) ∈ trans_star"
| *trans_star_step*:
"(p,γ,q') ∈ transition_relation ⟹
(q',w,q) ∈ trans_star ⟹
(p, γ#w, q) ∈ trans_star"

inductive-cases *trans_star_empty* [*elim*]: *"(p, [], q) ∈ trans_star"*
inductive-cases *trans_star_cons*: *"(p, γ#w, q) ∈ trans_star"*

inductive-set *trans_star_states* :: *"('state * 'label list * 'state list * 'state) set"* **where**
trans_star_states_refl[*iff*]:
"(p,[],[p],p) ∈ trans_star_states"
| *trans_star_states_step*:
"(p,γ,q') ∈ transition_relation ⟹
(q',w,ss,q) ∈ trans_star_states ⟹
(p, γ#w, p#ss, q) ∈ trans_star_states"

inductive-set *path_with_word* :: *"('state list * 'label list) set"* **where**
path_with_word_refl[*iff*]:
"([s],[]) ∈ path_with_word"
| *path_with_word_step*:
"(s'#ss, w) ∈ path_with_word ⟹
(s,l,s') ∈ transition_relation ⟹
(s#s'#ss,l#w) ∈ path_with_word"

definition *start_of* :: *"('state list × 'label list) ⇒ 'state"* **where**
"start_of π = hd (fst π)"

definition *end_of* :: "('*state list* × '*label list*) ⇒ '*state*" **where**
"*end_of* π = *last* (*fst* π)"

abbreviation *path_with_word_from* :: "'*state* ⇒ ('*state list* ∗ '*label list*) *set*" **where**
"*path_with_word_from q* == {π. π ∈ *path_with_word* ∧ *start_of* π = *q*}"

abbreviation *path_with_word_to* :: "'*state* ⇒ ('*state list* ∗ '*label list*) *set*" **where**
"*path_with_word_to q* == {π. π ∈ *path_with_word* ∧ *end_of* π = *q*}"

abbreviation *path_with_word_from_to* :: "'*state* ⇒ '*state* ⇒ ('*state list* ∗ '*label list*) *set*" **where**
"*path_with_word_from_to start end* == {π. π ∈ *path_with_word* ∧ *start_of* π = *start* ∧ *end_of* π = *end*}"

inductive-set *transition_list_path* :: "('*state*, '*label*) *transition list set*" **where**
"(*q*, *l*, *q'*) ∈ *transition_relation* ⟹
[(*q*, *l*, *q'*)] ∈ *transition_list_path*"
| "(*q*, *l*, *q'*) ∈ *transition_relation* ⟹
(*q'*, *l'*, *q''*) # *ts* ∈ *transition_list_path* ⟹
(*q*, *l*, *q'*) # (*q'*, *l'*, *q''*) # *ts* ∈ *transition_list_path*"

lemma *singleton_path_start_end*:
assumes "([*s*], []) ∈ *LTS.path_with_word pg*"
shows "*start_of* ([*s*], []) = *end_of* ([*s*], [])"
using *assms*
by (*simp add*: *end_of_def start_of_def*)

lemma *path_with_word_length*:
assumes "(*ss*, *w*) ∈ *path_with_word*"
shows "*length ss* = *length w* + *1*"
using *assms*
proof (*induction rule*: *path_with_word.induct*)
case (*path_with_word_refl s*)
then show *?case* **by** *auto*
next
case (*path_with_word_step s' ss w s l*)
then show *?case* **by** *auto*
qed

lemma *path_with_word_lengths*:
assumes "(*qs* @ [*qnminus1*], *w*) ∈ *path_with_word*"
shows "*length qs* = *length w*"
using *assms*
by (*metis LTS.path_with_word_length Suc_eq_plus1 Suc_inject length_Cons length_append*
list.size(*3*,*4*))

lemma *path_with_word_butlast*:
assumes "(*ss*, *w*) ∈ *path_with_word*"
assumes "*length ss* ≥ *2*"
shows "(*butlast ss*, *butlast w*) ∈ *path_with_word*"
using *assms*
proof (*induction rule*: *path_with_word.induct*)
case (*path_with_word_refl s*)
then show *?case*
by *force*
next
case (*path_with_word_step s' ss w s l*)
then show *?case*
by (*metis* (*no_types*) *LTS.path_with_word.path_with_word_refl*
LTS.path_with_word.path_with_word_step LTS.path_with_word_length One_nat_def Suc_1
Suc_inject Suc_leI Suc_le_mono butlast.simps(*2*) *length_0_conv length_Cons list.distinct*(*1*)
list.size(*4*) *not_gr0*)
qed

```
lemma transition_butlast:
  assumes "(ss, w) ∈ path_with_word"
  assumes "length ss ≥ 2"
  shows "(last (butlast ss), last w, last ss) ∈ transition_relation"
  using assms
proof (induction rule: path_with_word.induct)
  case (path_with_word_refl s)
  then show ?case
    by force
next
  case (path_with_word_step s' ss w s l)
  then show ?case
    by (metis (no_types) LTS.path_with_word_length One_nat_def Suc_1 Suc_inject Suc_leI Suc_le_mono
        butlast.simps(2) last.simps length_Cons length_greater_0_conv list.distinct(1) list.size(4))
qed

lemma path_with_word_induct_reverse [consumes 1, case_names path_with_word_refl path_with_word_step_rev]:
  "(ss, w) ∈ path_with_word ⟹
   (⋀s. P [s] []) ⟹
   (⋀ss s w l s'. (ss @ [s], w) ∈ path_with_word ⟹
                  P (ss @ [s]) w ⟹
                  (s, l, s') ∈ transition_relation ⟹
                  P (ss @ [s, s']) (w @ [l]))
   ⟹ P ss w"
proof (induction "length ss" arbitrary: ss w)
  case 0
  then show ?case
    by (metis LTS.path_with_word_length Suc_eq_plus1 Zero_not_Suc)
next
  case (Suc n)

  show ?case
  proof (cases "n = 0")
    case True
    then show ?thesis
      by (metis LTS.path_with_word_length Suc.hyps(2) Suc.prems(1) Suc.prems(2) Suc_eq_plus1 Suc_inject
Suc_length_conv length_0_conv)
  next
    case False
    define ss' where "ss' = butlast (butlast ss)"
    define s where "s = last (butlast ss)"
    define s' where "s' = last ss"
    define w' where "w' = butlast w"
    define l where "l = last w"

    have "length ss ≥ 2"
      using False Suc.hyps(2) by linarith

    then have s_split: "ss' @ [s, s'] = ss"
      by (metis One_nat_def Suc_1 Suc_le_mono Zero_not_Suc append.assoc append.simps(1) append_Cons
          append_butlast_last_id le_less length_append_singleton list.size(3) s'_def s_def ss'_def
          zero_order(3))

    have w_split: "w' @ [l] = w"
      by (metis LTS.path_with_word_length Suc.prems(1) add.commute butlast.simps(2) butlast_append
          l_def length_0_conv length_Suc_conv list.simps(3) plus_1_eq_Suc s_split
          snoc_eq_iff_butlast w'_def)

    have ss'w'_path: "(ss' @ [s], w') ∈ path_with_word"
      using Suc(3) path_with_word_butlast
      by (metis (no_types, lifting) ‹2 ≤ length ss› butlast.simps(2) butlast_append list.simps(3)
          s_split w'_def)
```

```
    have tr: "(s, l, s') ∈ transition_relation"
      using Suc(3) s'_def s_def l_def transition_butlast ‹2 ≤ length ss› by presburger

    have nl: "n = length (ss' @ [s])"
      by (metis LTS.path_with_word_length Suc.hyps(2) Suc.prems(1) Suc_eq_plus1
          length_append_singleton nat.inject ss'w'_path w_split)

    have "P (ss' @ [s]) w'"
      using Suc(1)[of "ss' @ [s]" w', OF nl ss'w'_path Suc(4)] Suc(5) by metis

    then have "P (ss' @ [s, s']) (w' @ [l])"
      using Suc(5)[of ss' s w' l s'] ss'w'_path tr by auto
    then show ?thesis
      using s_split w_split by auto
  qed
qed

lemma path_with_word_from_induct_reverse:
  "(ss, w) ∈ path_with_word_from start ⟹
   (⋀s. P [s] []) ⟹
   (⋀ss s w l s'. (ss @ [s], w) ∈ path_with_word_from start ⟹
                  P (ss @ [s]) w ⟹
                  (s, l, s') ∈ transition_relation ⟹
                  P (ss @ [s, s']) (w @ [l]))
   ⟹ P ss w"
proof (induction "length ss" arbitrary: ss w)
  case 0
  then show ?case
    by (metis (no_types, lifting) Suc_eq_plus1 mem_Collect_eq nat.simps(3) path_with_word_length)
next
  case (Suc n)

  show ?case
  proof (cases "n = 0")
    case True
    then show ?thesis
      using  Suc.prems(1,2) length_0_conv list.distinct(1) path_with_word.cases
      by (metis (no_types, lifting) Suc.hyps(2) length_Suc_conv list.inject mem_Collect_eq)
  next
    case False
    define ss' where "ss' = butlast (butlast ss)"
    define s where "s = last (butlast ss)"
    define s' where "s' = last ss"
    define w' where "w' = butlast w"
    define l where "l = last w"

    have len_ss: "length ss ≥ 2"
      using False Suc.hyps(2) by linarith

    then have s_split: "ss' @ [s, s'] = ss"
      by (metis One_nat_def Suc_1 Suc_le_mono Zero_not_Suc append.assoc append.simps(1) append_Cons
          append_butlast_last_id le_less length_append_singleton list.size(3) s'_def s_def ss'_def
          zero_order(3))

    have w_split: "w' @ [l] = w"
      by (metis (no_types, lifting) False LTS.path_with_word_length One_nat_def Suc.hyps(2)
          Suc.prems(1) Suc_inject add.right_neutral add_Suc_right l_def list.size(3) mem_Collect_eq
          snoc_eq_iff_butlast w'_def)

    have ss'w'_path: "(ss' @ [s], w') ∈ path_with_word"
      using Suc(3) path_with_word_butlast len_ss
      by (metis (no_types, lifting) butlast.simps(2) butlast_append list.discI mem_Collect_eq
          not_Cons_self2 s_split w'_def)
```

```
    have ss'w'_path_from: "(ss' @ [s], w') ∈ path_with_word_from start"
      using Suc(3) butlast.simps(2) start_of_def list.sel(1) list.simps(3) mem_Collect_eq
        path_with_word.simps prod.sel(1) s_def snoc_eq_iff_butlast ss'_def ss'w'_path w_split
      by (metis (no_types, lifting) hd_append)

    have tr: "(s, l, s') ∈ transition_relation"
      using Suc(3) s'_def s_def l_def transition_butlast len_ss by blast

    have nl: "n = length (ss' @ [s])"
      using False Suc.hyps(2) ss'_def by force

    have "P (ss' @ [s]) w'"
      using Suc(1)[of "ss' @ [s]" w', OF nl ss'w'_path_from Suc(4) ] Suc(5) by fastforce

    then have "P (ss' @ [s, s']) (w' @ [l])"
      using Suc(5)[of ss' s w' l s'] tr ss'w'_path_from by blast
    then show ?thesis
      using s_split w_split by auto
  qed
qed

inductive transition_of :: "('state, 'label) transition ⇒ 'state list * 'label list ⇒ bool" where
  "transition_of (s1,γ,s2) (s1#s2#ss, γ#w)"
| "transition_of (s1,γ,s2) (ss, w) ⟹
   transition_of (s1,γ,s2) (s#ss, μ#w)"

lemma path_with_word_not_empty[simp]: "¬([],w) ∈ path_with_word"
  using LTS.path_with_word.cases by blast

lemma trans_star_path_with_word:
  assumes "(p, w, q) ∈ trans_star"
  shows "∃ ss. hd ss = p ∧ last ss = q ∧ (ss, w) ∈ path_with_word"
  using assms
proof (induction rule: trans_star.inducts)
  case (trans_star_refl p)
  then show ?case
    by (meson LTS.path_with_word.path_with_word_refl last.simps list.sel(1))
next
  case (trans_star_step p γ q' w q)
  then show ?case
    by (metis LTS.path_with_word.simps hd_Cons_tl last_ConsR list.discI list.sel(1))
qed

lemma trans_star_trans_star_states:
  assumes "(p, w, q) ∈ trans_star"
  shows "∃ ss. (p, w, ss, q) ∈ trans_star_states"
  using assms
proof (induction rule: trans_star.induct)
  case (trans_star_refl p)
  then show ?case by auto
next
  case (trans_star_step p γ q' w q)
  then show ?case
    by (meson LTS.trans_star_states_step)
qed

lemma trans_star_states_trans_star:
  assumes "(p, w, ss, q) ∈ trans_star_states"
  shows "(p, w, q) ∈ trans_star"
  using assms
proof (induction rule: trans_star_states.induct)
  case (trans_star_states_refl p)
```

```
  then show ?case by auto
next
  case (trans_star_states_step p γ q' w q)
  then show ?case
    by (meson LTS.trans_star.trans_star_step)
qed

lemma path_with_word_trans_star:
  assumes "(ss, w) ∈ path_with_word"
  assumes "length ss ≠ 0"
  shows "(hd ss, w, last ss) ∈ trans_star"
  using assms
proof (induction rule: path_with_word.inducts)
  case (path_with_word_refl s)
  show ?case
    by simp
next
  case (path_with_word_step s' ss w s l)
  then show ?case
    using LTS.trans_star.trans_star_step by fastforce
qed

lemma path_with_word_trans_star_Cons:
  assumes "(s1#ss@[s2], w) ∈ path_with_word"
  shows "(s1, w, s2) ∈ trans_star"
  using assms path_with_word_trans_star by force

lemma path_with_word_trans_star_Singleton:
  assumes "([s2], w) ∈ path_with_word"
  shows "(s2, [], s2) ∈ trans_star"
  using assms path_with_word_trans_star by force

lemma trans_star_split:
  assumes "(p'', u1 @ w1, q) ∈ trans_star"
  shows "∃ q1. (p'', u1, q1) ∈ trans_star ∧ (q1, w1, q) ∈ trans_star"
  using assms
proof(induction u1 arbitrary: p'')
  case Nil
  then show ?case by auto
next
  case (Cons a u1)
  then show ?case
    by (metis LTS.trans_star.trans_star_step LTS.trans_star_cons append_Cons)
qed

lemma trans_star_states_append:
  assumes "(p2, w2, w2_ss, q') ∈ trans_star_states"
  assumes "(q', v, v_ss, q) ∈ trans_star_states"
  shows "(p2, w2 @ v, w2_ss @ tl v_ss, q) ∈ trans_star_states"
  using assms
proof (induction rule: trans_star_states.induct)
  case (trans_star_states_refl p)
  then show ?case
    by (metis append_Cons append_Nil list.sel(3) trans_star_states.simps)
next
  case (trans_star_states_step p γ q' w ss q)
  then show ?case
    using LTS.trans_star_states.trans_star_states_step by fastforce
qed

lemma trans_star_states_length:
  assumes "(p, u, u_ss, p1) ∈ trans_star_states"
  shows "length u_ss = Suc (length u)"
```

```
  using assms
proof (induction rule: trans_star_states.induct)
  case (trans_star_states_refl p)
  then show ?case
    by simp
next
  case (trans_star_states_step p γ q' w ss q)
  then show ?case
    by simp
qed

lemma trans_star_states_last:
  assumes "(p, u, u_ss, p1) ∈ trans_star_states"
  shows "p1 = last u_ss"
  using assms
proof (induction rule: trans_star_states.induct)
  case (trans_star_states_refl p)
  then show ?case
    by simp
next
  case (trans_star_states_step p γ q' w ss q)
  then show ?case
    using LTS.trans_star_states.cases by force
qed

lemma trans_star_states_hd:
  assumes "(q', v, v_ss, q) ∈ trans_star_states"
  shows "q' = hd v_ss"
  using assms
proof (induction rule: trans_star_states.induct)
  case (trans_star_states_refl p)
  then show ?case
    by simp
next
  case (trans_star_states_step p γ q' w ss q)
  then show ?case
    by force
qed

lemma trans_star_states_transition_relation:
  assumes "(p, γ#w_rest, ss, q) ∈ trans_star_states"
  shows "∃ s γ'. (s, γ', q) ∈ transition_relation"
  using assms
proof (induction w_rest arbitrary: ss p γ)
  case Nil
  then show ?case
    by (metis LTS.trans_star_empty LTS.trans_star_states_trans_star trans_star_cons)
next
  case (Cons a w_rest)
  then show ?case
    by (meson LTS.trans_star_cons LTS.trans_star_states_trans_star trans_star_trans_star_states)
qed

lemma trans_star_states_path_with_word:
  assumes "(p, w, ss, q) ∈ trans_star_states"
  shows "(ss,w) ∈ path_with_word"
  using assms
proof (induction rule: trans_star_states.induct)
  case (trans_star_states_refl p)
  then show ?case by auto
next
  case (trans_star_states_step p γ q' w ss q)
  then show ?case
```

by (*metis LTS.trans_star_states.simps path_with_word.path_with_word_step*)
qed

lemma *path_with_word_trans_star_states*:
assumes "(*ss*,*w*) ∈ *path_with_word*"
assumes "*p* = *hd ss*"
assumes "*q* = *last ss*"
shows "(*p*, *w*, *ss*, *q*) ∈ *trans_star_states*"
using *assms*
proof (*induction arbitrary*: *p q rule*: *path_with_word.induct*)
case (*path_with_word_refl s*)
then show *?case*
by *simp*
next
case (*path_with_word_step s' ss w s l*)
then show *?case*
using *trans_star_states.trans_star_states_step* **by** *auto*
qed

lemma *append_path_with_word_path_with_word*:
assumes "*last γ2ss* = *hd v_ss*"
assumes "(*γ2ss*, *γ2ε*) ∈ *path_with_word*"
assumes "(*v_ss*, *v*) ∈ *path_with_word*"
shows "(*γ2ss*, *γ2ε*) @′ (*v_ss*, *v*) ∈ *path_with_word*"
by (*metis LTS.trans_star_states_path_with_word append_path_with_word.simps*
path_with_word_trans_star_states assms(*1*,*2*,*3*) *trans_star_states_append*)

lemma *hd_is_hd*:
assumes "(*p*, *w*, *ss*, *q*) ∈ *trans_star_states*"
assumes "(*p1*, *γ*, *q1*) = *hd* (*transition_list*′ (*p*, *w*, *ss*, *q*))"
assumes "*transition_list*′ (*p*, *w*, *ss*, *q*) ≠ []"
shows "*p* = *p1*"
using *assms*
proof (*induction rule*: *trans_star_states.inducts*)
case (*trans_star_states_refl p*)
then show *?case*
by *auto*
next
case (*trans_star_states_step p γ q' w ss q*)
then show *?case*
by (*metis LTS.trans_star_states.simps Pair_inject list.sel*(*1*) *transition_list'.simps*
transition_list.simps(*1*))
qed

definition *srcs* :: "′*state set*" **where**
"*srcs* = {*p*. ∄ *q γ*. (*q*, *γ*, *p*) ∈ *transition_relation*}"

definition *sinks* :: "′*state set*" **where**
"*sinks* = {*p*. ∄ *q γ*. (*p*, *γ*, *q*) ∈ *transition_relation*}"

definition *isolated* :: "′*state set*" **where**
"*isolated* = *srcs* ∩ *sinks*"

lemma *srcs_def2*:
"*q* ∈ *srcs* ⟷ (∄ *q*′ *γ*. (*q*′, *γ*, *q*) ∈ *transition_relation*)"
by (*simp add*: *LTS.srcs_def*)

lemma *sinks_def2*:
"*q* ∈ *sinks* ⟷ (∄ *q*′ *γ*. (*q*, *γ*, *q*′) ∈ *transition_relation*)"
by (*simp add*: *LTS.sinks_def*)

lemma *isolated_no_edges*:
assumes "(*p*, *γ*, *q*) ∈ *transition_relation*"

```
  shows "p ∉ isolated ∧ q ∉ isolated"
  using assms isolated_def srcs_def2 sinks_def2 by fastforce

lemma source_never_or_hd:
  assumes "(ss, w) ∈ path_with_word"
  assumes "p1 ∈ srcs"
  assumes "t = (p1, γ, q1)"
  shows "count (transitions_of (ss, w)) t = 0 ∨
         ((hd (transition_list (ss, w)) = t ∧ count (transitions_of (ss, w)) t = 1))"
  using assms
proof (induction rule: path_with_word.induct)
  case (path_with_word_refl s)
  then show ?case
    by simp
next
  case (path_with_word_step s' ss w s l)
  then have "count (transitions_of (s' # ss, w)) t = 0 ∨
    (hd (transition_list (s' # ss, w)) = t ∧ count (transitions_of (s' # ss, w)) t = 1)"
    by auto
  then show ?case
  proof
    assume asm: "count (transitions_of (s' # ss, w)) t = 0"
    show ?case
    proof (cases "s = p1 ∧ l = γ ∧ q1 = s'")
      case True
      then have "hd (transition_list (s # s' # ss, l # w)) = t ∧
                 count (transitions_of (s # s' # ss, l # w)) t = 1"
        using path_with_word_step asm by simp
      then show ?thesis
        by auto
    next
      case False
      then have "count (transitions_of (s # s' # ss, l # w)) t = 0"
        using path_with_word_step asm by auto
      then show ?thesis
        by auto
    qed
  next
    assume "hd (transition_list (s' # ss, w)) = t ∧ count (transitions_of (s' # ss, w)) t = 1"
    moreover
    have "(∄ q γ. (q, γ, p1) ∈ transition_relation)"
      by (meson LTS.srcs_def2 assms(2))
    ultimately
    have False
      using path_with_word_step by (auto elim: path_with_word.cases)
    then show ?case
      by auto
  qed
qed

lemma source_only_hd:
  assumes "(ss, w) ∈ path_with_word"
  assumes "p1 ∈ srcs"
  assumes "count (transitions_of (ss, w)) t > 0"
  assumes "t = (p1, γ, q1)"
  shows "hd (transition_list (ss, w)) = t ∧ count (transitions_of (ss, w)) t = 1"
  using source_never_or_hd assms not_gr_zero
  by metis

lemma no_end_in_source:
  assumes "(p, w, qq) ∈ trans_star"
  assumes "w ≠ []"
  shows "qq ∉ srcs"
```

```
  using assms
proof (induction rule: trans_star.induct)
  case (trans_star_refl p)
  then show ?case
    by blast
next
  case (trans_star_step p γ q' w q)
  then show ?case
    by (metis LTS.srcs_def2 LTS.trans_star_empty)
qed

lemma transition_list_length_Cons:
  assumes "length ss = Suc (length w)"
  assumes "hd (transition_list (ss, w)) = (p, γ, q)"
  assumes "transition_list (ss, w) ≠ []"
  shows "∃ w' ss'. w = γ # w' ∧ ss = p # q # ss'"
proof (cases ss)
  case Nil
  note Nil_outer = Nil
  show ?thesis
  proof (cases w)
    case Nil
    then show ?thesis
      using assms Nil_outer by auto
  next
    case (Cons a list)
    then show ?thesis
      using assms Nil_outer by auto
  qed
next
  case (Cons a list)
  note Cons_outer = Cons
  then show ?thesis
  proof (cases w)
    case Nil
    then show ?thesis
      using assms Cons_outer by auto
  next
    case (Cons aa llist)
    with Cons_outer assms show ?thesis
      by (cases list) auto
  qed
qed

lemma transition_list_Cons:
  assumes "(p, w, ss, q) ∈ trans_star_states"
  assumes "hd (transition_list (ss, w)) = (p, γ, q1)"
  assumes "transition_list (ss, w) ≠ []"
  shows "∃ w' ss'. w = γ # w' ∧ ss = p # q1 # ss'"
  using assms transition_list_length_Cons by (metis LTS.trans_star_states_length)

lemma nothing_after_sink:
  assumes "([q, q']@ss, γ1#w) ∈ path_with_word"
  assumes "q' ∈ sinks"
  shows "ss = [] ∧ w = []"
  using assms
proof (induction rule: path_with_word.induct)
  case (path_with_word_refl s)
  then have "∄ q'' γ. (q', γ, q'') ∈ transition_relation"
    using sinks_def2[of "q'"]
    by auto
  with assms(1) show ?case
    by (auto elim: path_with_word.cases)
```

next
case (*path_with_word_step s' ss w s l*)
then show *?case*
by *metis*
qed

lemma *count_transitions_of'_tails*:
assumes "(*p*, *γ'*, *q'_add*) ≠ (*p1*, *γ*, *q'*)"
shows "*count* (*transitions_of'* (*p*, *γ'* # *w*, *p* # *q'_add* # *ss_rest*, *q*)) (*p1*, *γ*, *q'*) =
count (*transitions_of'* (*q'_add*, *w*, *q'_add* # *ss_rest*, *q*)) (*p1*, *γ*, *q'*)"
using *assms* **by** (*cases w*) *auto*

lemma *avoid_count_zero*:
assumes "(*p*, *w*, *ss*, *q*) ∈ *trans_star_states*"
assumes "(*p1*, *γ*, *q'*) ∉ *transition_relation*"
shows "*count* (*transitions_of'* (*p*, *w*, *ss*, *q*)) (*p1*, *γ*, *q'*) = *0*"
using *assms*
proof(*induction arbitrary*: *p rule*: *trans_star_states.induct*)
case (*trans_star_states_refl p*)
then show *?case*
by *auto*
next
case (*trans_star_states_step p γ q' w ss q*)
show *?case*
by (*metis trans_star_states_step trans_star_states.cases assms*(*2*)
count_transitions_of'_tails transitions_of'.simps)
qed

lemma *transition_list_append*:
assumes "(*ss*,*w*) ∈ *path_with_word*"
assumes "(*ss'*,*w'*) ∈ *path_with_word*"
assumes "*last ss* = *hd ss'*"
shows "*transition_list* ((*ss*,*w*) @´ (*ss'*,*w'*)) = *transition_list* (*ss*,*w*) @ *transition_list* (*ss'*,*w'*)"
using *assms*
proof (*induction rule*: *path_with_word.induct*)
case (*path_with_word_refl s*)
then have "*transition_list* (*hd ss'* # *tl ss'*, *w'*) = *transition_list* (*ss'*, *w'*)"
by (*metis LTS.path_with_word_not_empty list.exhaust_sel*)
then show *?case*
using *path_with_word_refl* **by** *auto*
next
case (*path_with_word_step s' ss w s l*)
then show *?case*
by *auto*
qed

lemma *split_path_with_word_beginning''*:
assumes "(*SS*, *WW*) ∈ *path_with_word*"
assumes "*SS* = (*ss* @ *ss'*)"
assumes "*length ss* = *Suc* (*length w*)"
assumes "*WW* = *w* @ *w'*"
shows "(*ss*,*w*) ∈ *path_with_word*"
using *assms*
proof (*induction arbitrary*: *ss ss' w w' rule*: *path_with_word.induct*)
case (*path_with_word_refl s*)
then show *?case*
by (*metis append.right_neutral append_is_Nil_conv list.sel*(*3*) *list.size*(*3*) *nat.discI*
path_with_word.path_with_word_refl tl_append2)
next
case (*path_with_word_step s'a ssa wa s l*)
then show *?case*
proof (*cases* "*w*")
case *Nil*

```
    then show ?thesis
      using path_with_word_step by (metis LTS.path_with_word.simps length_0_conv length_Suc_conv)
  next
    case (Cons)
    have "(s'a # ssa, wa) ∈ LTS.path_with_word transition_relation"
      by (simp add: "path_with_word_step.hyps"(1))
    moreover
    have "s'a # ssa = tl ss @ ss'"
      by (metis "path_with_word_step.prems"(1,2) Zero_not_Suc
          length_0_conv list.sel(3) tl_append2)
    moreover
    have "length (tl ss) = Suc (length (tl w))"
      using "path_with_word_step.prems" Cons by auto
    moreover
    have "wa = tl w @ w'"
      by (metis path_with_word_step(5,6) calculation(3) length_Suc_conv list.sel(3) list.size(3)
          nat.simps(3) tl_append2)
    ultimately
    have "(tl ss, tl w) ∈ LTS.path_with_word transition_relation"
      using path_with_word_step(3)[of "tl ss" ss' "tl w" w'] by auto
    then show ?thesis
      using path_with_word_step
      by (auto simp: Cons_eq_append_conv intro: path_with_word.path_with_word_step)
  qed
qed

lemma split_path_with_word_end':
  assumes "(SS,WW) ∈ path_with_word"
  assumes "SS = (ss @ ss')"
  assumes "length ss' = Suc (length w')"
  assumes "WW = w @ w'"
  shows "(ss',w') ∈ path_with_word"
  using assms(1) assms
proof (induction arbitrary: ss ss' w w' rule: path_with_word.induct)
  case (path_with_word_refl s)
  then show ?case
    by (metis Nil_is_append_conv Zero_not_Suc append_Nil list.sel(3) list.size(3) tl_append2)
next
  case (path_with_word_step s' ssa wa s l)
  show ?case
  proof (cases "ss")
    case Nil
    then show ?thesis
      using path_with_word_step(4,5,6,7) path_with_word_length
      by (auto simp: Cons_eq_append_conv)
  next
    case (Cons x xs)
    have "(s' # ssa, wa) ∈ LTS.path_with_word transition_relation"
      using "path_with_word_step.hyps"(1) by blast
    moreover
    have "s' # ssa = tl ss @ ss'"
      using path_with_word_step(5) using local.Cons by auto
    moreover
    have "length ss' = Suc (length w')"
      using "path_with_word_step.prems"(3) by blast
    moreover
    have "wa = tl w @ w'"
    proof (cases "wa = []")
      assume "wa ≠[]"
      then show ?thesis
        using path_with_word_step(4-7) Cons path_with_word_length
        by (fastforce simp: Cons_eq_append_conv)
    next
```

```
    assume wa_empty: "wa = []"
    have "tl ss @ ss' ≠ []"
      using calculation(2) by force
    then have "(butlast (tl ss @ ss') @ [last (s' # ssa)], []) = (s' # ssa, wa)"
      using wa_empty by (simp add: calculation(2))
    then have "(butlast (tl ss @ ss') @ [last (s' # ssa)], []) ∈ LTS.path_with_word transition_relation"
      using "path_with_word_step"(1) by metis
    then have "length (butlast (tl ss @ ss')) = length ([]::'v list)"
      using LTS.path_with_word_lengths by (metis list.size(3))
    then have "w' = []"
      by (simp add: calculation(3))
    then show ?thesis
      using "path_with_word_step.prems"(4) by force
  qed
  ultimately
  show ?thesis
    using path_with_word_step(3)[of "tl ss" ss' w' "tl w"] by auto
 qed
qed

lemma split_path_with_word_end:
 assumes "(ss @ ss',w @ w') ∈ path_with_word"
 assumes "length ss' = Suc (length w')"
 shows "(ss',w') ∈ path_with_word"
 using split_path_with_word_end' assms by blast

lemma split_path_with_word_beginning':
 assumes "(ss @ ss',w @ w') ∈ path_with_word"
 assumes "length ss = Suc (length w)"
 shows "(ss,w) ∈ path_with_word"
 using assms split_path_with_word_beginning'' by blast

lemma split_path_with_word_beginning:
 assumes "(ss, w) @´ (ss', w') ∈ path_with_word"
 assumes "length ss = Suc (length w)"
 shows "(ss,w) ∈ path_with_word"
 using assms split_path_with_word_beginning'' by (metis append_path_with_word.simps)

lemma path_with_word_remove_last':
 assumes "(SS, W) ∈ path_with_word"
 assumes "SS = ss @ [s, s']"
 assumes "W = w @ [l]"
 shows "(ss @ [s], w) ∈ path_with_word"
 using assms
proof (induction arbitrary: ss w rule: path_with_word_induct_reverse)
 case (path_with_word_refl s)
 then show ?case
   by auto
next
 case (path_with_word_step_rev ss s w l s')
 then show ?case
   by auto
qed

lemma path_with_word_remove_last:
 assumes "(ss @ [s, s'], w @ [l]) ∈ path_with_word"
 shows "(ss @ [s], w) ∈ path_with_word"
 using path_with_word_remove_last' assms by auto

lemma transition_list_append_edge:
 assumes "(ss @ [s, s'], w @ [l]) ∈ path_with_word"
 shows "transition_list (ss @ [s, s'], w @ [l]) = transition_list (ss @ [s], w) @ [(s, l, s')]"
proof −
```

```
  have "(ss @ [s], w) ∈ path_with_word"
    using assms path_with_word_remove_last by auto
  moreover
  have "([s, s'], [l]) ∈ path_with_word"
    using assms length_Cons list.size(3) by (metis split_path_with_word_end')
  moreover
  have "last (ss @ [s]) = hd [s, s']"
    by auto
  ultimately
  show ?thesis
    using transition_list_append[of "ss @ [s]" w "[s, s']" "[l]"] by auto
qed

end
```

1.4 More LTS lemmas

```
lemma hd_transition_list_append_path_with_word:
  assumes "hd (transition_list (ss, w)) = (p1, γ, q1)"
  assumes "transition_list (ss, w) ≠ []"
  shows  "([p1, q1], [γ]) @´ (tl ss, tl w) = (ss, w)"
proof −
  have "p1 # q1 # tl (tl ss) = ss ∧ γ # tl w = w"
  proof (cases ss)
    case Nil
    note Nil_outer = Nil
    show ?thesis
    proof (cases w)
      case Nil
      then show ?thesis
        using assms Nil_outer by auto
    next
      case (Cons a list)
      then show ?thesis
        using assms Nil_outer by auto
    qed
  next
    case (Cons a list)
    note Cons_outer = Cons
    show ?thesis
    proof (cases w)
      case Nil
      then show ?thesis
        using assms Cons_outer using list.collapse by (metis transition_list.simps(2,4))
    next
      case (Cons aa llist)
      have "p1 = a"
        using assms Cons Cons_outer
        by (metis Pair_inject list.exhaust list.sel(1) transition_list.simps(1,2))
      moreover
      have "q1 # tl list = list"
        using assms Cons Cons_outer
        by (cases list) auto
      moreover
      have "γ = aa"
        by (metis Cons_outer Pair_inject assms(1) calculation(2) list.sel(1) local.Cons
            transition_list.simps(1))
      ultimately
      show ?thesis
        using assms Cons_outer Cons by auto
    qed
  qed
  then show ?thesis
    by auto
```

qed

lemma *counting*:
"count (transitions_of' ((hdss1,ww1,ss1,lastss1))) (s1, γ, s2) =
count (transitions_of ((ss1,ww1))) (s1, γ, s2)"
by *force*

lemma *count_append_path_with_word_γ*:
assumes *"length ss1 = Suc (length ww1)"*
assumes *"ss2 ≠ []"*
shows *"count (transitions_of (((ss1,ww1),γ') @^γ (ss2,ww2))) (s1, γ, s2) =*
count (transitions_of (ss1,ww1)) (s1, γ, s2) +
(if s1 = last ss1 ∧ s2 = hd ss2 ∧ γ = γ' then 1 else 0) +
count (transitions_of (ss2,ww2)) (s1, γ, s2)"
using *assms* **proof** (*induction ww1 arbitrary*: *ss1*)
case *Nil*
note *Nil_outer = Nil*
obtain *s* **where** *s_p*: *"ss1 = [s]"*
by (*metis Suc_length_conv length_0_conv local.Nil(1)*)
then show *?case*
proof (*cases ss2*)
case *Nil*
then show *?thesis*
using *assms* **by** *blast*
next
case (*Cons s2' ss2'*)
then show *?thesis*
proof (*cases "s1 = s2'"*)
case *True*
then show *?thesis*
by (*simp add*: *local.Cons s_p*)
next
case *False*
then show *?thesis*
using *s_p local.Cons* **by** *fastforce*
qed
qed
next
case (*Cons w ww11*)
obtain *s2' ss2'* **where** *s2'_ss2'_p*: *"ss2 = s2' # ss2'"*
by (*meson assms list.exhaust*)
obtain *s1' ss1'* **where** *s1'_ss1'_p*: *"ss1 = s1' # ss1'"*
by (*meson Cons.prems(1) length_Suc_conv*)
show *?case*
using *Cons(1)[of "ss1'"] Cons(2−) s2'_ss2'_p s1'_ss1'_p*
by (*auto simp*: *length_Suc_conv*)
qed

lemma *count_append_path_with_word*:
assumes *"length ss1 = Suc (length ww1)"*
assumes *"ss2 ≠ []"*
assumes *"last ss1 = hd ss2"*
shows *"count (transitions_of (((ss1, ww1)) @´ (ss2, ww2))) (s1, γ, s2) =*
count (transitions_of (ss1, ww1)) (s1, γ, s2) +
count (transitions_of (ss2, ww2)) (s1, γ, s2)"
using *assms* **proof** (*induction ww1 arbitrary*: *ss1*)
case *Nil*
note *Nil_outer = Nil*
obtain *s* **where** *s_p*: *"ss1 = [s]"*
by (*metis Suc_length_conv length_0_conv local.Nil(1)*)
then show *?case*
proof (*cases ss2*)
case *Nil*

```
    then show ?thesis
      using assms by blast
  next
    case (Cons s2' ss2')
    then show ?thesis
    proof (cases "s1 = s2'")
      case True
      then show ?thesis
        using local.Cons s_p
        using Nil_outer(3) by auto
    next
      case False
      then show ?thesis
        using s_p local.Cons
        using Nil_outer(3) by fastforce
    qed
  qed
next
  case (Cons w ww11)
  show ?case
    using Cons by (fastforce simp: length_Suc_conv split: if_splits)
qed

lemma count_append_trans_star_states_γ_length:
  assumes "length (ss1) = Suc (length (ww1))"
  assumes "ss2 ≠ []"
  shows "count (transitions_of' (((hdss1,ww1,ss1,lastss1),γ') @@^γ (hdss2,ww2,ss2,lastss2))) (s1, γ, s2) =
         count (transitions_of' (hdss1,ww1,ss1,lastss1)) (s1, γ, s2) +
         (if s1 = last ss1 ∧ s2 = hd ss2 ∧ γ = γ' then 1 else 0) +
         count (transitions_of' (hdss2,ww2,ss2,lastss2)) (s1, γ, s2)"
  using assms count_append_path_with_word_γ by force

lemma count_append_trans_star_states_γ:
  assumes "(hdss1,ww1,ss1,lastss1) ∈ LTS.trans_star_states A"
  assumes "(hdss2,ww2,ss2,lastss2) ∈ LTS.trans_star_states A"
  shows "count (transitions_of' (((hdss1,ww1,ss1,lastss1),γ') @@^γ (hdss2,ww2,ss2,lastss2))) (s1, γ, s2) =
         count (transitions_of' (hdss1,ww1,ss1,lastss1)) (s1, γ, s2) +
         (if s1 = last ss1 ∧ s2 = hd ss2 ∧ γ = γ' then 1 else 0) +
         count (transitions_of' (hdss2,ww2,ss2,lastss2)) (s1, γ, s2)"
proof −
  have "length (ss1) = Suc (length (ww1))"
    by (meson LTS.trans_star_states_length assms(1))
  moreover
  have "ss2 ≠ []"
    by (metis LTS.trans_star_states.simps assms(2) list.discI)
  ultimately
  show ?thesis
    using count_append_trans_star_states_γ_length by metis
qed

lemma count_append_trans_star_states_length:
  assumes "length (ss1) = Suc (length (ww1))"
  assumes "ss2 ≠ []"
  assumes "last ss1 = hd ss2"
  shows "count (transitions_of' (((hdss1,ww1,ss1,lastss1)) @@' (hdss2,ww2,ss2,lastss2))) (s1, γ, s2) =
         count (transitions_of' (hdss1,ww1,ss1,lastss1)) (s1, γ, s2) +
         count (transitions_of' (hdss2,ww2,ss2,lastss2)) (s1, γ, s2)"
  using count_append_path_with_word[OF assms(1) assms(2) assms(3), of ww2 s1 γ s2] by auto

lemma count_append_trans_star_states:
  assumes "(hdss1,ww1,ss1,lastss1) ∈ LTS.trans_star_states A"
  assumes "(lastss1,ww2,ss2,lastss2) ∈ LTS.trans_star_states A"
  shows "count (transitions_of' (((hdss1,ww1,ss1,lastss1)) @@' (lastss1,ww2,ss2,lastss2))) (s1, γ, s2) =
```

```
        count (transitions_of' (hdss1,ww1,ss1,lastss1)) (s1, γ, s2) +
        count (transitions_of' (lastss1,ww2,ss2,lastss2)) (s1, γ, s2)"
proof −
  have "length (ss1) = Suc (length (ww1))"
    by (meson LTS.trans_star_states_length assms(1))
  moreover
  have "last ss1 = hd ss2"
    by (metis LTS.trans_star_states_hd LTS.trans_star_states_last assms(1) assms(2))
  moreover
  have "ss2 ≠ []"
    by (metis LTS.trans_star_states_length Zero_not_Suc assms(2) list.size(3))
  ultimately
  show ?thesis
    using count_append_trans_star_states_length assms by auto
qed


context fixes Δ :: "('state, 'label) transition set" begin
fun reach where
  "reach p [] = {p}"
| "reach p (γ#w) =
    (⋃ q' ∈ (⋃ (p',γ',q') ∈ Δ. if p' = p ∧ γ' = γ then {q'} else {}).
      reach q' w)"
end
lemma trans_star_imp_exec: "(p,w,q) ∈ LTS.trans_star Δ ⟹ q ∈ reach Δ p w"
  by (induct p w q rule: LTS.trans_star.induct[of _ _ _ Δ, consumes 1]) force+
lemma reach_imp: "q ∈ reach Δ p w ⟹ (p,w,q) ∈ LTS.trans_star Δ"
  by (induct p w rule: reach.induct)
    (auto intro!: LTS.trans_star_refl[of _ Δ] LTS.trans_star_step[of _ _ _ Δ] split: if_splits)
lemma trans_star_code[code_unfold]: "(p,w,q) ∈ LTS.trans_star Δ ⟷ q ∈ reach Δ p w"
  by (meson reach_imp trans_star_imp_exec)

lemma subset_srcs_code[code_unfold]:
  "X ⊆ LTS.srcs A ⟷ (∀ q ∈ X. q ∉ snd ` snd ` A)"
  by (auto simp add: LTS.srcs_def image_iff)


lemma LTS_trans_star_mono:
  "mono LTS.trans_star"
proof (rule, rule)
  fix pwq :: "'a × 'b list × 'a"
  fix ts ts' :: "('a, 'b) transition set"
  assume sub: "ts ⊆ ts'"
  assume pwq_ts: "pwq ∈ LTS.trans_star ts"
  then obtain p w q where pwq_p: "pwq = (p, w, q)"
    using prod_cases3 by blast
  then have "(p, w, q) ∈ LTS.trans_star ts"
    using pwq_ts by auto
  then have "(p, w, q) ∈  LTS.trans_star ts'"
  proof(induction w arbitrary: p)
    case Nil
    then show ?case
      by (metis LTS.trans_star.trans_star_refl LTS.trans_star_empty)
  next
    case (Cons γ w)
    then show ?case
      by (meson LTS.trans_star.simps LTS.trans_star_cons sub subsetD)
  qed
  then show "pwq ∈ LTS.trans_star ts'"
    unfolding pwq_p .
qed

lemma count_next_0:
```

```
  assumes "count (transitions_of (s # s' # ss, l # w)) (p1, γ, q') = 0"
  shows "count (transitions_of (s' # ss, w)) (p1, γ, q') = 0"
  using assms by (cases "s = p1 ∧ l = γ ∧ s' = q'") auto

lemma count_next_hd:
  assumes "count (transitions_of (s # s' # ss, l # w)) (p1, γ, q') = 0"
  shows "(s, l, s') ≠ (p1, γ, q')"
  using assms by auto

lemma count_empty_zero: "count (transitions_of' (p, [], [p_add], p_add)) (p1, γ, q') = 0"
  by simp

lemma count_zero_remove_path_with_word:
  assumes "(ss, w) ∈ LTS.path_with_word Ai"
  assumes "0 = count (transitions_of (ss, w)) (p1, γ, q')"
  assumes "Ai = Aiminus1 ∪ {(p1, γ, q')}"
  shows "(ss, w) ∈ LTS.path_with_word Aiminus1"
  using assms
proof (induction rule: LTS.path_with_word.induct[OF assms(1)])
  case (1 s)
  then show ?case
    by (simp add: LTS.path_with_word.path_with_word_refl)
next
  case (2 s' ss w s l)
  from 2(5) have "0 = count (transitions_of (s' # ss, w)) (p1, γ, q')"
    using count_next_0 by auto
  then have s'_ss_w_Aiminus1: "(s' # ss, w) ∈ LTS.path_with_word Aiminus1"
    using 2 by auto
  have "(s, l, s') ∈ Aiminus1"
    using 2(2,5) assms(3) by force
  then show ?case
    using s'_ss_w_Aiminus1 by (simp add: LTS.path_with_word.path_with_word_step)
qed

lemma count_zero_remove_path_with_word_trans_star_states:
  assumes "(p, w, ss ,q) ∈ LTS.trans_star_states Ai"
  assumes "0 = count (transitions_of' (p, w, ss, q)) (p1, γ, q')"
  assumes "Ai = Aiminus1 ∪ {(p1, γ, q')}"
  shows "(p, w, ss, q) ∈ LTS.trans_star_states Aiminus1"
  using assms
proof (induction arbitrary: p rule: LTS.trans_star_states.induct[OF assms(1)])
  case (1 p)
  then show ?case
    by (metis LTS.trans_star_states.simps list.distinct(1))
next
  case (2 p' γ' q'' w ss q)
  have p_is_p': "p' = p"
    by (meson "2.prems"(1) LTS.trans_star_states.cases list.inject)
  {
    assume len: "length ss > 0"
    have not_found: "(p, γ', hd ss) ≠ (p1, γ, q')"
      using LTS.trans_star_states.cases count_next_hd list.sel(1) transitions_of'.simps
      using 2(4) 2(5) by (metis len hd_Cons_tl length_greater_0_conv)
    have hdAI: "(p, γ', hd ss) ∈ Ai"
      by (metis "2.hyps"(1) "2.hyps"(2) LTS.trans_star_states.cases list.sel(1) p_is_p')
    have t_Aiminus1: "(p, γ', hd ss) ∈ Aiminus1"
      using 2 hdAI not_found by force
    have "(p, γ' # w, p' # ss, q) ∈ LTS.trans_star_states (Aiminus1 ∪ {(p1, γ, q')})"
      using "2.prems"(1) assms(3) by fastforce
    have ss_hd_tl: "hd ss # tl ss = ss"
      using len hd_Cons_tl by blast
    moreover
    have "(hd ss, w, ss, q) ∈ LTS.trans_star_states Ai"
```

```
    using ss_hd_tl "2.hyps"(2) using LTS.trans_star_states.cases
    by (metis list.sel(1))
   ultimately have "(hd ss, w, ss, q) ∈ LTS.trans_star_states Aiminus1"
    using ss_hd_tl using "2.IH" "2.prems"(2) not_found assms(3) p_is_p'
     LTS.count_transitions_of'_tails by (metis)
   from this t_Aiminus1 have ?case
    using LTS.trans_star_states.intros(2)[of p γ' "hd ss" Aiminus1 w ss q] using p_is_p' by auto
  }
  moreover
  {
   assume "length ss = 0"
   then have ?case
    using "2.hyps"(2) LTS.trans_star_states.cases by force
  }
  ultimately show ?case
   by auto
qed

lemma count_zero_remove_trans_star_states_trans_star:
  assumes "(p, w, ss ,q) ∈ LTS.trans_star_states Ai"
  assumes "0 = count (transitions_of' (p, w, ss, q)) (p1, γ, q')"
  assumes "Ai = Aiminus1 ∪ {(p1, γ, q')}"
  shows "(p, w, q) ∈ LTS.trans_star Aiminus1"
 using assms count_zero_remove_path_with_word_trans_star_states by (metis LTS.trans_star_states_trans_star)

lemma split_at_first_t:
  assumes "(p, w, ss, q) ∈ LTS.trans_star_states Ai"
  assumes "Suc j' = count (transitions_of' (p, w, ss, q)) (p1, γ, q')"
  assumes "(p1, γ, q') ∉ Aiminus1"
  assumes "Ai = Aiminus1 ∪ {(p1, γ, q')}"
  shows "∃ u v u_ss v_ss.
         ss = u_ss @ v_ss ∧
         w = u @ [γ] @ v ∧
         (p, u, u_ss, p1) ∈ LTS.trans_star_states Aiminus1 ∧
         (p1, [γ], q') ∈ LTS.trans_star Ai ∧
         (q', v, v_ss, q) ∈ LTS.trans_star_states Ai ∧
         (p, w, ss, q) = ((p, u, u_ss, p1),γ) @@^γ (q', v, v_ss,q)"
  using assms
proof(induction arbitrary: p rule: LTS.trans_star_states.induct[OF assms(1)])
  case (1 p_add p)
  from 1(2) have "False"
   using count_empty_zero by auto
  then show ?case
   by auto
next
  case (2 p_add γ' q'_add w ss q p)
  then have p_add_p: "p_add = p"
   by (meson LTS.trans_star_states.cases list.inject)
  from p_add_p have p_Ai: "(p, γ', q'_add) ∈ Ai"
   using 2(1) by auto
  from p_add_p have p_γ'_w_ss_Ai: "(p, γ' # w, p # ss, q) ∈ LTS.trans_star_states Ai"
   using 2(4) by auto
  from p_add_p have count_p_γ'_w_ss: "Suc j' = count (transitions_of' (p, γ' # w, p # ss, q)) (p1, γ, q')"
   using 2(5) by auto
  show ?case
  proof(cases "(p, γ', q'_add) = (p1, γ, q')")
   case True
   define u :: "'b list" where "u = []"
   define u_ss :: "'a list" where "u_ss = [p]"
   define v where "v = w"
   define v_ss where "v_ss = ss"
   have "(p, u, u_ss, p1) ∈ LTS.trans_star_states Aiminus1"
    unfolding u_def u_ss_def using LTS.trans_star_states.intros
```

```
    using True by fastforce
  have "(p1, [γ], q') ∈ LTS.trans_star Ai"
    using p_Ai by (metis LTS.trans_star.trans_star_refl LTS.trans_star.trans_star_step True)
  have "(q', v, v_ss, q) ∈ LTS.trans_star_states Ai"
    using 2(2) True v_def v_ss_def by blast
  show ?thesis
    using Pair_inject True ‹(p, u, u_ss, p1) ∈ LTS.trans_star_states Aiminus1›
      ‹(p1, [γ], q') ∈ LTS.trans_star Ai› ‹(q', v, v_ss, q) ∈ LTS.trans_star_states Ai›
      append_Cons p_add_p self_append_conv2 u_def u_ss_def v_def v_ss_def
    by (metis (no_types) append_trans_star_states_γ.simps)
next
  case False
  have "hd ss = q'_add"
    by (metis LTS.trans_star_states.cases 2(2) list.sel(1))
  from this False have g: "Suc j' = count (transitions_of' (q'_add, w, ss, q)) (p1, γ, q')"
    using count_p_γ'_w_ss by (cases ss) auto
  have "∃ u_ih v_ih u_ss_ih v_ss_ih.
        ss = u_ss_ih @ v_ss_ih ∧
        w = u_ih @ [γ] @ v_ih ∧
        (q'_add, u_ih, u_ss_ih, p1) ∈ LTS.trans_star_states Aiminus1 ∧
        (p1, [γ], q') ∈ LTS.trans_star Ai ∧
        (q', v_ih, v_ss_ih, q) ∈ LTS.trans_star_states Ai"
    using 2(3)[of q'_add, OF 2(2) g 2(6) 2(7)] by auto
  then obtain u_ih v_ih u_ss_ih v_ss_ih where splitting_p:
    "ss = u_ss_ih @ v_ss_ih"
    "w = u_ih @ [γ] @ v_ih"
    "(q'_add, u_ih, u_ss_ih, p1) ∈ LTS.trans_star_states Aiminus1"
    "(p1, [γ], q') ∈ LTS.trans_star Ai"
    "(q', v_ih, v_ss_ih, q) ∈ LTS.trans_star_states Ai"
    by metis
  define v where "v = v_ih"
  define v_ss where "v_ss = v_ss_ih"
  define u where "u = γ' # u_ih"
  define u_ss where "u_ss = p # u_ss_ih"
  have "p_add # ss = u_ss @ v_ss"
    by (simp add: p_add_p splitting_p(1) u_ss_def v_ss_def)
  have "γ' # w = u @ [γ] @ v"
    using splitting_p(2) u_def v_def by auto
  have "(p, u, u_ss, p1) ∈ LTS.trans_star_states Aiminus1"
    using False LTS.trans_star_states.trans_star_states_step 2(7) p_Ai splitting_p(3) u_def
      u_ss_def by fastforce
  have "(p1, [γ], q') ∈ LTS.trans_star Ai"
    by (simp add: splitting_p(4))
  have "(q', v, v_ss, q) ∈ LTS.trans_star_states Ai"
    by (simp add: splitting_p(5) v_def v_ss_def)
  show ?thesis
    using ‹(p, u, u_ss, p1) ∈ LTS.trans_star_states Aiminus1›
      ‹(q', v, v_ss, q) ∈ LTS.trans_star_states Ai› ‹γ' # w = u @ [γ] @ v›
      ‹p_add # ss = u_ss @ v_ss› splitting_p(4)
    by auto
qed
qed

lemma trans_star_states_mono:
  assumes "(p, w, ss, q) ∈ LTS.trans_star_states A1"
  assumes "A1 ⊆ A2"
  shows "(p, w, ss, q) ∈ LTS.trans_star_states A2"
  using assms
proof (induction rule: LTS.trans_star_states.induct[OF assms(1)])
  case (1 p)
  then show ?case
    by (simp add: LTS.trans_star_states.trans_star_states_refl)
next
```

```
  case (2 p γ q' w ss q)
  then show ?case
    by (meson LTS.trans_star_states.trans_star_states_step in_mono)
qed

lemma count_combine_trans_star_states_append:
  assumes "ss = u_ss @ v_ss ∧ w = u @ [γ] @ v"
  assumes "t = (p1, γ, q')"
  assumes "(p, u, u_ss, p1) ∈ LTS.trans_star_states A"
  assumes "(q', v, v_ss, q) ∈ LTS.trans_star_states B"
  shows "count (transitions_of' (p, w, ss, q)) t =
         count (transitions_of' (p, u, u_ss, p1)) t +
         1 +
         count (transitions_of' (q', v, v_ss, q)) t"
proof −
  have v_ss_non_empt: "v_ss ≠ []"
    using LTS.trans_star_states.cases assms by force

  have u_ss_l: "length u_ss = Suc (length u)"
    using assms LTS.trans_star_states_length by metis

  have p1_u_ss: "p1 = last u_ss"
    using assms LTS.trans_star_states_last by metis

  have q'_v_ss: "q' = hd v_ss"
    using assms LTS.trans_star_states_hd by metis

  have one: "(if p1 = last u_ss ∧ q' = hd v_ss then 1 else 0) = 1"
    using p1_u_ss q'_v_ss by auto

  from count_append_trans_star_states_γ_length[of u_ss u v_ss p q γ q' v q p1 ] show ?thesis
    using assms(1) assms(2) assms(3) by (auto simp add: assms(3) one u_ss_l v_ss_non_empt)
qed

lemma count_combine_trans_star_states:
  assumes "t = (p1, γ, q')"
  assumes "(p, u, u_ss, p1) ∈ LTS.trans_star_states A"
  assumes "(q', v, v_ss, q) ∈ LTS.trans_star_states B"
  shows "count (transitions_of' (((p, u, u_ss, p1),γ) @@^γ (q', v, v_ss, q))) t =
       count (transitions_of' (p, u, u_ss, p1)) t + 1 + count (transitions_of' (q', v, v_ss, q)) t"
  by (metis append_trans_star_states_γ.simps assms count_combine_trans_star_states_append)

lemma transition_list_reversed_simp:
  assumes "length ss = length w"
  shows "transition_list (ss @ [s, s'], w @ [l]) = (transition_list (ss@[s],w)) @ [(s,l,s')]"
  using assms
proof (induction ss arbitrary: w)
  case Nil
  then show ?case
    by auto
next
  case (Cons a ss)
  define w' where "w' = tl w"
  define l' where "l' = hd w"
  have w_split: "l' # w' = w"
    by (metis Cons.prems l'_def length_0_conv list.distinct(1) list.exhaust_sel w'_def)
  then have "length ss = length w'"
    using Cons.prems by force
  then have "transition_list (ss @ [s, s'], w' @ [l]) = transition_list (ss @ [s], w') @ [(s, l, s')]"
    using Cons(1)[of w'] by auto
  then have "transition_list (a # ss @ [s, s'], l' # w' @ [l]) = transition_list (a # ss @ [s], l' # w') @ [(s, l, s')]"
    by (cases ss) auto
  then show ?case
```

using *w_split* **by** *auto*
qed

lemma *LTS_trans_star_mono'*:
"mono LTS.trans_star_states"
by (*auto simp*: *mono_def trans_star_states_mono*)

lemma *path_with_word_mono'*:
assumes *"(ss, w) ∈ LTS.path_with_word A1"*
assumes *"A1 ⊆ A2"*
shows *"(ss, w) ∈ LTS.path_with_word A2"*
by (*meson LTS.trans_star_states_path_with_word LTS.path_with_word_trans_star_states assms(1,2) trans_star_states_mono*)

lemma *LTS_path_with_word_mono*:
"mono LTS.path_with_word"
by (*auto simp*: *mono_def path_with_word_mono'*)

1.5 Reverse transition system

fun *rev_edge* :: *"('n,'v) transition ⇒ ('n,'v) transition"* **where**
"rev_edge (q_s,α,q_o) = (q_o, α, q_s)"

lemma *rev_edge_rev_edge_id*[*simp*]: *"rev_edge (rev_edge x) = x"*
by (*cases x*) *auto*

fun *rev_path_with_word* :: *"'n list * 'v list ⇒ 'n list * 'v list"* **where**
"rev_path_with_word (es,ls) = (rev es, rev ls)"

definition *rev_edge_list* :: *"('n,'v) transition list ⇒ ('n,'v) transition list"* **where**
"rev_edge_list ts = rev (map rev_edge ts)"

context *LTS* **begin**

interpretation *rev_LTS*: *LTS "(rev_edge ‘ transition_relation)"*
.

lemma *rev_path_in_rev_pg*:
assumes *"(ss, w) ∈ path_with_word"*
shows *"(rev ss, rev w) ∈ rev_LTS.path_with_word"*
using *assms(1) assms*
proof (*induction rule*: *path_with_word_induct_reverse*)
case (*path_with_word_refl s*)
then show *?case*
by (*simp add*: *LTS.path_with_word.path_with_word_refl*)
next
case (*path_with_word_step_rev ss s w l s'*)
have *"(s', l, s) ∈ rev_edge ‘ transition_relation"*
using *path_with_word_step_rev* **by** (*simp add*: *rev_image_eqI*)
moreover
have *"(rev (ss @ [s]), rev w) ∈ LTS.path_with_word (rev_edge ‘ transition_relation)"*
using *"path_with_word_step_rev.IH" "path_with_word_step_rev.hyps"(1)* **by** *blast*
then have *"(s # rev ss, rev w) ∈ LTS.path_with_word (rev_edge ‘ transition_relation)"*
by *auto*
ultimately
have *"(s' # s # rev ss, l # rev w) ∈ LTS.path_with_word (rev_edge ‘ transition_relation)"*
by (*simp add*: *LTS.path_with_word.path_with_word_step*)
then show *?case*
by *auto*
qed

lemma *transition_list_rev_edge_list*:
assumes *"(ss,w) ∈ path_with_word"*
shows *"transition_list (rev ss, rev w) = rev_edge_list (transition_list (ss, w))"*

using *assms(1) assms*
proof (*induction rule*: *path_with_word.induct*)
case (*path_with_word_refl s*)
then show *?case*
by (*simp add*: *rev_edge_list_def*)
next
case (*path_with_word_step s′ ss w s l*)
have "*transition_list* (*rev* (*s* # *s′* # *ss*), *rev* (*l* # *w*)) = *transition_list* (*rev ss* @ [*s′*, *s*], *rev w* @ [*l*])"
by *auto*
moreover
have "... = *transition_list* (*rev ss* @ [*s′*], *rev w*) @ [(*s′*, *l*, *s*)]"
using *transition_list_reversed_simp*[*of* "*rev ss*" "*rev w*" *s′ s l*]
using "*path_with_word_step.hyps*"(*1*) *LTS.path_with_word_lengths rev_path_in_rev_pg* **by** *fastforce*
moreover
have "... = *rev_edge_list* (*transition_list* (*s′* # *ss*, *w*)) @ [(*s′*, *l*, *s*)]"
using *path_with_word_step* **by** *auto*
moreover
have "... = *rev_edge_list* ((*s*, *l*, *s′*) # *transition_list* (*s′* # *ss*, *w*))"
unfolding *rev_edge_list_def* **by** *auto*
moreover
have "... = *rev_edge_list* (*transition_list* (*s* # *s′* # *ss*, *l* # *w*))"
by *auto*
ultimately
show *?case*
by *metis*
qed

end

2 LTS with epsilon

2.1 LTS functions

context begin

private abbreviation *ε* :: "'*label option*" **where**
"*ε* == *None*"

definition *inters_ε* :: "('*state*, '*label option*) *transition set* ⇒ ('*state*, '*label option*) *transition set* ⇒ (('*state* ∗ '*state*), '*label option*) *transition set*" **where**
"*inters_ε ts1 ts2* =
{((*p1*, *q1*), *α*, (*p2*, *q2*)) | *p1 q1 α p2 q2*. (*p1*, *α*, *p2*) ∈ *ts1* ∧ (*q1*, *α*, *q2*) ∈ *ts2*} ∪
{((*p1*, *q1*), *ε*, (*p2*, *q1*)) | *p1 p2 q1*. (*p1*, *ε*, *p2*) ∈ *ts1*} ∪
{((*p1*, *q1*), *ε*, (*p1*, *q2*)) | *p1 q1 q2*. (*q1*, *ε*, *q2*) ∈ *ts2*}"

end

2.2 LTS with epsilon locale

locale *LTS_ε* = *LTS transition_relation* **for** *transition_relation* :: "('*state*, '*label option*) *transition set*"
begin

abbreviation *ε* :: "'*label option*" **where**
"*ε* == *None*"

inductive-set *trans_star_ε* :: "('*state* ∗ '*label list* ∗ '*state*) *set*" **where**
trans_star_ε_refl[*iff*]: "(*p*, [], *p*) ∈ *trans_star_ε*"
| *trans_star_ε_step_γ*: "(*p*, *Some γ*, *q′*) ∈ *transition_relation* ⟹ (*q′*,*w*,*q*) ∈ *trans_star_ε*
⟹ (*p*, *γ*#*w*, *q*) ∈ *trans_star_ε*"
| *trans_star_ε_step_ε*: "(*p*, *ε*, *q′*) ∈ *transition_relation* ⟹ (*q′*,*w*,*q*) ∈ *trans_star_ε*
⟹ (*p*, *w*, *q*) ∈ *trans_star_ε*"

inductive-cases *trans_star_ε_empty* [*elim*]: "(*p*, [], *q*) ∈ *trans_star_ε*"

inductive-cases *trans_star_cons_ε*: "(p, γ#w, q) ∈ *trans_star*"

definition *remove_ε* :: "'*label option list* ⇒ '*label list*" **where**
"*remove_ε w* = *map the* (*removeAll ε w*)"

definition *ε_exp* :: "'*label option list* ⇒ '*label list* ⇒ *bool*" **where**
"*ε_exp w' w* ⟷ *map the* (*removeAll ε w'*) = *w*"

lemma *trans_star_trans_star_ε*:
assumes "(p, w, q) ∈ *trans_star*"
shows "(p, *map the* (*removeAll ε w*), q) ∈ *trans_star_ε*"
using *assms*
proof (*induction rule*: *trans_star.induct*)
case (*trans_star_refl p*)
then show *?case*
by *simp*
next
case (*trans_star_step p γ q' w q*)
show *?case*
proof (*cases γ*)
case *None*
then show *?thesis*
using *trans_star_step* **by** (*simp add*: *trans_star_ε.trans_star_ε_step_ε*)
next
case (*Some γ'*)
then show *?thesis*
using *trans_star_step* **by** (*simp add*: *trans_star_ε.trans_star_ε_step_γ*)
qed
qed

lemma *trans_star_ε_ε_exp_trans_star*:
assumes "(p, w, q) ∈ *trans_star_ε*"
shows "∃ w'. *ε_exp w' w* ∧ (p, w', q) ∈ *trans_star*"
using *assms*
proof (*induction rule*: *trans_star_ε.induct*)
case (*trans_star_ε_refl p*)
then show *?case*
by (*metis LTS.trans_star.trans_star_refl ε_exp_def list.simps*(*8*) *removeAll.simps*(*1*))
next
case (*trans_star_ε_step_γ p γ q' w q*)
obtain *wε* :: "'*label option list*" **where**
f1: "(q', *wε*, q) ∈ *trans_star* ∧ *ε_exp wε w*"
using *trans_star_ε_step_γ.IH* **by** *blast*
then have "*ε_exp* (*Some γ* # *wε*) (*γ* # *w*)"
by (*simp add*: *LTS_ε.ε_exp_def*)
then show *?case*
using *f1* **by** (*meson trans_star.simps trans_star_ε_step_γ.hyps*(*1*))
next
case (*trans_star_ε_step_ε p q' w q*)
then show *?case*
by (*metis trans_starp.trans_star_step trans_starp_trans_star_eq ε_exp_def removeAll.simps*(*2*))
qed

lemma *trans_star_ε_iff_ε_exp_trans_star*:
"(p, w, q) ∈ *trans_star_ε* ⟷ (∃ w'. *ε_exp w' w* ∧ (p, w', q) ∈ *trans_star*)"
proof
assume "(p, w, q) ∈ *trans_star_ε*"
then show "∃ w'. *ε_exp w' w* ∧ (p, w', q) ∈ *trans_star*"
using *trans_star_ε_ε_exp_trans_star trans_star_trans_star_ε* **by** *auto*
next
assume "∃ w'. *ε_exp w' w* ∧ (p, w', q) ∈ *trans_star*"
then show "(p, w, q) ∈ *trans_star_ε*"
using *trans_star_ε_ε_exp_trans_star trans_star_trans_star_ε ε_exp_def* **by** *auto*

qed

lemma *ε_exp_split′*:
assumes "*ε_exp u_ε (γ1 # u1)*"
shows "*∃γ1_ε u1_ε. ε_exp γ1_ε [γ1] ∧ ε_exp u1_ε u1 ∧ u_ε = γ1_ε @ u1_ε*"
using *assms*
proof (*induction u_ε arbitrary*: *u1 γ1*)
case *Nil*
then show *?case*
by (*metis LTS_ε.ε_exp_def list.distinct*(1) *list.simps*(8) *removeAll.simps*(1))
next
case (*Cons a u_ε*)
then show *?case*
proof (*induction a*)
case *None*
then have "*ε_exp u_ε (γ1 # u1)*"
using *ε_exp_def* **by** *force*
then have "*∃γ1_ε u1_ε. ε_exp γ1_ε [γ1] ∧ ε_exp u1_ε u1 ∧ u_ε = γ1_ε @ u1_ε*"
using *None*(1) **by** *auto*
then show *?case*
by (*metis LTS_ε.ε_exp_def append_Cons removeAll.simps*(2))
next
case (*Some γ1′*)
have "*γ1′ = γ1*"
using *Some.prems*(2) *ε_exp_def* **by** *auto*
have "*ε_exp u_ε u1*"
using *Some.prems*(2) *ε_exp_def* **by** *force*
show *?case*
proof (*cases u1*)
case *Nil*
then show *?thesis*
by (*metis Some.prems*(2) *ε_exp_def append_Nil2 list.simps*(8) *removeAll.simps*(1))
next
case (*Cons a list*)
then show *?thesis*
using *LTS_ε.ε_exp_def* ‹*ε_exp u_ε u1*› ‹*γ1′ = γ1*› **by** *force*
qed
qed
qed

lemma *remove_ε_append_dist*:
"*remove_ε (w @ w′) = remove_ε w @ remove_ε w′*"
proof (*induction w*)
case *Nil*
then show *?case*
by (*simp add*: *LTS_ε.remove_ε_def*)
next
case (*Cons a w*)
then show *?case*
by (*simp add*: *LTS_ε.remove_ε_def*)
qed

lemma *remove_ε_Cons_tl*:
assumes "*remove_ε w = remove_ε (Some γ′ # tl w)*"
shows "*γ′ # remove_ε (tl w) = remove_ε w*"
using *assms* **unfolding** *remove_ε_def* **by** *auto*

lemma *trans_star_states_trans_star_ε*:
assumes "(*p*, *w*, *ss*, *q*) ∈ *trans_star_states*"
shows "(*p*, *LTS_ε.remove_ε w*, *q*) ∈ *trans_star_ε*"
by (*metis LTS_ε.trans_star_trans_star_ε assms remove_ε_def trans_star_states_trans_star*)

lemma *no_edge_to_source_ε*:
assumes "(*p*, [*γ*], *qq*) ∈ *trans_star_ε*"
shows "*qq* ∉ *srcs*"
proof −
have "∃ *w*. *LTS_ε.ε_exp w* [*γ*] ∧ (*p*, *w*, *qq*) ∈ *trans_star* ∧ *w* ≠ []"
by (*metis* (*no_types*) *LTS_ε.ε_exp_def LTS_ε.ε_exp_split′ LTS_ε.trans_star_ε_iff_ε_exp_trans_star append_Cons append_Nil assms*(1) *list.distinct*(1) *list.exhaust*)
then obtain *w* **where** "*LTS_ε.ε_exp w* [*γ*] ∧ (*p*, *w*, *qq*) ∈ *trans_star* ∧ *w* ≠ []"
by *blast*
then show *?thesis*
using *LTS.no_end_in_source*[*of p w qq*] *assms* **by** *auto*
qed

lemma *trans_star_not_to_source_ε*:
assumes "(*p‴*, *w*, *q*) ∈ *trans_star_ε*"
assumes "*p‴* ≠ *q*"
assumes "*q′* ∈ *srcs*"
shows "*q′* ≠ *q*"
using *assms*
proof (*induction rule*: *trans_star_ε.induct*)
case (*trans_star_ε_refl p*)
then show *?case*
by *blast*
next
case (*trans_star_ε_step_γ p γ q′ w q*)
then show *?case*
using *srcs_def2* **by** *metis*
next
case (*trans_star_ε_step_ε p q′ w q*)
then show *?case*
using *srcs_def2* **by** *metis*
qed

lemma *append_edge_edge_trans_star_ε*:
assumes "(*p1*, *Some γ′*, *p2*) ∈ *transition_relation*"
assumes "(*p2*, *Some γ″*, *q1*) ∈ *transition_relation*"
assumes "(*q1*, *u1*, *q*) ∈ *trans_star_ε*"
shows "(*p1*, [*γ′*, *γ″*] @ *u1*, *q*) ∈ *trans_star_ε*"
using *assms* **by** (*metis trans_star_ε_step_γ append_Cons append_Nil*)

inductive-set *trans_star_states_ε* :: "('*state* ∗ '*label list* ∗ '*state list* ∗ '*state*) *set*" **where**
trans_star_states_ε_refl[*iff*]:
"(*p*,[],[*p*],*p*) ∈ *trans_star_states_ε*"
| *trans_star_states_ε_step_γ*:
"(*p*,*Some γ*,*q′*) ∈ *transition_relation* ⟹
(*q′*,*w*,*ss*,*q*) ∈ *trans_star_states_ε* ⟹
(*p*, *γ*#*w*, *p*#*ss*, *q*) ∈ *trans_star_states_ε*"
| *trans_star_states_ε_step_ε*:
"(*p*, *ε* ,*q′*) ∈ *transition_relation* ⟹
(*q′*,*w*,*ss*,*q*) ∈ *trans_star_states_ε* ⟹
(*p*, *w*, *p*#*ss*, *q*) ∈ *trans_star_states_ε*"

inductive-set *path_with_word_ε* :: "('*state list* ∗ '*label list*) *set*" **where**
path_with_word_ε_refl[*iff*]:
"([*s*],[]) ∈ *path_with_word_ε*"
| *path_with_word_ε_step_γ*:
"(*s′*#*ss*, *w*) ∈ *path_with_word_ε* ⟹
(*s*,*Some l*,*s′*) ∈ *transition_relation* ⟹
(*s*#*s′*#*ss*,*l*#*w*) ∈ *path_with_word_ε*"
| *path_with_word_ε_step_ε*:
"(*s′*#*ss*, *w*) ∈ *path_with_word_ε* ⟹
(*s*,*ε*,*s′*) ∈ *transition_relation* ⟹
(*s*#*s′*#*ss*,*w*) ∈ *path_with_word_ε*"

lemma *ε_exp_Some_length*:
assumes "*ε_exp* (*Some* α # *w1'*) *w*"
shows "*0* < *length w*"
using *assms*
by (*metis LTS_ε.ε_exp_def length_greater_0_conv list.map*(*2*) *neq_Nil_conv option.simps*(*3*) *removeAll.simps*(*2*))

lemma *ε_exp_Some_hd*:
assumes "*ε_exp* (*Some* α # *w1'*) *w*"
shows "*hd w* = α"
using *assms*
by (*metis LTS_ε.ε_exp_def list.sel*(*1*) *list.simps*(*9*) *option.sel option.simps*(*3*) *removeAll.simps*(*2*))

lemma *exp_empty_empty*:
assumes "*ε_exp* [] *w*"
shows "*w* = []"
using *assms* **by** (*metis LTS_ε.ε_exp_def list.simps*(*8*) *removeAll.simps*(*1*))

end

2.3 More LTS lemmas

lemma *LTS_ε_trans_star_ε_mono*:
"*mono LTS_ε.trans_star_ε*"
proof (*rule*, *rule*)
fix *pwq* :: "'*a* × '*b list* × '*a*"
fix *ts ts'* :: "('*a*, '*b option*) *transition set*"
assume *sub*: "*ts* ⊆ *ts'*"
assume *pwq_ts*: "*pwq* ∈ *LTS_ε.trans_star_ε ts*"
then obtain *p w q* **where** *pwq_p*: "*pwq* = (*p*, *w*, *q*)"
using *prod_cases3* **by** *blast*
then have *x*: "(*p*, *w*, *q*) ∈ *LTS_ε.trans_star_ε ts*"
using *pwq_ts* **by** *auto*
then have "(∃ *w'*. *LTS_ε.ε_exp w'* *w* ∧ (*p*, *w'*, *q*) ∈ *LTS.trans_star ts*)"
using *LTS_ε.trans_star_ε_iff_ε_exp_trans_star*[*of p w q ts*] **by** *auto*
then have "(∃ *w'*. *LTS_ε.ε_exp w'* *w* ∧ (*p*, *w'*, *q*) ∈ *LTS.trans_star ts'*)"
using *LTS_trans_star_mono sub*
using *monoD* **by** *blast*
then have "(*p*, *w*, *q*) ∈ *LTS_ε.trans_star_ε ts'*"
using *LTS_ε.trans_star_ε_iff_ε_exp_trans_star*[*of p w q ts'*] **by** *auto*
then show "*pwq* ∈ *LTS_ε.trans_star_ε ts'*"
unfolding *pwq_p* .
qed

definition *ε_edge_of_edge* **where**
"*ε_edge_of_edge* = (λ(*a*, *l*, *b*). (*a*, *Some l*, *b*))"

definition *LTS_ε_of_LTS* **where**
"*LTS_ε_of_LTS transition_relation* = *ε_edge_of_edge* ' *transition_relation*"

end

References

[Wim20] Simon Wimmer. Archive of graph formalizations. 2020. https://github.com/wimmers/archive-of-graph-formalizations.