Analysis of List Update Algorithms

Maximilian P.L. Haslbeck and Tobias Nipkow

March 17, 2025

Abstract

These theories formalize the quantitative analysis of a number of classical algorithms for the list update problem: 2-competitiveness of move-to-front, the lower bound of 2 for the competitive- ness of deterministic list update algorithms and 1.6-competitiveness of the randomized COMB algorithm, the best randomized list update algorithm known to date.

An informal description is found in an accompanying report [HN16]. The material is based on the first two chapters of the book by Borodin and El-Yaniv [BEY98].

Contents

1 List Inversion

theory *Inversion*
imports *List−Index.List_Index*
begin

abbreviation *dist_perm xs ys* ≡ *distinct xs* ∧ *distinct ys* ∧ *set xs* = *set ys*

definition *before_in* :: ′*a* ⇒ ′*a* ⇒ ′*a list* ⇒ *bool*
 (‹(_ </ _/ in _)› [55,55,55] 55) **where**
x < *y in xs* = (*index xs x* < *index xs y* ∧ *y* ∈ *set xs*)

definition *Inv* :: ′*a list* ⇒ ′*a list* ⇒ (′*a* ∗ ′*a*) *set* **where**
Inv xs ys = {(*x*,*y*). *x* < *y in xs* ∧ *y* < *x in ys*}

lemma *before_in_setD1*: *x* < *y in xs* ⟹ *x* : *set xs*
by (*metis index_conv_size_if_notin index_less before_in_def less_asym order_refl*)

lemma *before_in_setD2*: *x* < *y in xs* ⟹ *y* : *set xs*
by (*simp add*: *before_in_def*)

lemma *not_before_in*:
 x : *set xs* ⟹ *y* : *set xs* ⟹ ¬ *x* < *y in xs* ⟷ *y* < *x in xs* ∨ *x*=*y*
by (*metis index_eq_index_conv before_in_def less_asym linorder_neqE_nat*)

lemma *before_in_irefl*: *x* < *x in xs* = *False*
by (*meson before_in_setD2 not_before_in*)

lemma *no_before_inI*[*simp*]: *x* < *y in xs* ⟹ (¬ *y* < *x in xs*) = *True*
by (*metis before_in_setD1 not_before_in*)

lemma *finite_Invs*[*simp*]: *finite*(*Inv xs ys*)
apply(*rule finite_subset*[**where** *B* = *set xs* × *set ys*])
apply(*auto simp add*: *Inv_def before_in_def*)
apply(*metis index_conv_size_if_notin index_less_size_conv less_asym*)+
done

lemma *Inv_id*[*simp*]: *Inv xs xs* = {}
by(*auto simp add*: *Inv_def before_in_def*)

lemma *card_Inv_sym*: *card*(*Inv xs ys*) = *card*(*Inv ys xs*)
proof −

have *Inv xs ys* = (λ(*x*,*y*). (*y*,*x*)) ‘ *Inv ys xs* **by**(*auto simp*: *Inv_def*)
thus *?thesis* **by** (*metis card_image swap_inj_on*)
qed

lemma *Inv_tri_ineq*:
dist_perm xs ys ⟹ *dist_perm ys zs* ⟹
Inv xs zs ⊆ *Inv xs ys Un Inv ys zs*
by(*auto simp*: *Inv_def*) (*metis before_in_setD1 not_before_in*)

lemma *card_Inv_tri_ineq*:
dist_perm xs ys ⟹ *dist_perm ys zs* ⟹
card (*Inv xs zs*) ≤ *card*(*Inv xs ys*) + *card* (*Inv ys zs*)
using *card_mono*[*OF _ Inv_tri_ineq*[*of xs ys zs*]]
by *auto* (*metis card_Un_Int finite_Invs trans_le_add1*)

end

2 Swapping Adjacent Elements in a List

theory *Swaps*
imports *Inversion*
begin

Swap elements at index *n* and *Suc n*:

definition *swap n xs* =
(*if Suc n* < *size xs then xs*[*n* := *xs*!*Suc n*, *Suc n* := *xs*!*n*] *else xs*)

lemma *length_swap*[*simp*]: *length*(*swap i xs*) = *length xs*
by(*simp add*: *swap_def*)

lemma *swap_id*[*simp*]: *Suc n* ≥ *size xs* ⟹ *swap n xs* = *xs*
by(*simp add*: *swap_def*)

lemma *distinct_swap*[*simp*]:
distinct(*swap i xs*) = *distinct xs*
by(*simp add*: *swap_def*)

lemma *swap_Suc*[*simp*]: *swap* (*Suc n*) (*a* # *xs*) = *a* # *swap n xs*
by(*induction xs*) (*auto simp*: *swap_def*)

lemma *index_swap_distinct*:
distinct xs ⟹ *Suc n* < *length xs* ⟹
index (*swap n xs*) *x* =
(*if x* = *xs*!*n then Suc n else if x* = *xs*!*Suc n then n else index xs x*)

by(*auto simp add*: *swap_def index_swap_if_distinct*)

lemma *set_swap*[*simp*]: *set*(*swap n xs*) = *set xs*
by(*auto simp add*: *swap_def set_conv_nth nth_list_update*) *metis*

lemma *nth_swap_id*[*simp*]: *Suc i* < *length xs* ⟹ *swap i xs* ! *i* = *xs*!(*i*+*1*)
by(*simp add*: *swap_def*)

lemma *before_in_swap*:
 dist_perm xs ys ⟹ *Suc n* < *size xs* ⟹
  x < *y in* (*swap n xs*) ⟷
  x < *y in xs* ∧ ¬ (*x* = *xs*!*n* ∧ *y* = *xs*!*Suc n*) ∨ *x* = *xs*!*Suc n* ∧ *y* = *xs*!*n*
by(*simp add*:*before_in_def index_swap_distinct*)
 (*metis Suc_lessD Suc_lessI index_less_size_conv index_nth_id less_Suc_eq n_not_Suc_n nth_index*)

lemma *Inv_swap*: **assumes** *dist_perm xs ys*
shows *Inv xs* (*swap n ys*) =
  (*if Suc n* < *size xs*
   then if ys!*n* < *ys*!*Suc n in xs*
        then Inv xs ys ∪ {(*ys*!*n*, *ys*!*Suc n*)}
        else Inv xs ys − {(*ys*!*Suc n*, *ys*!*n*)}
   else Inv xs ys)
proof−
  have *length xs* = *length ys* **using** *assms* **by** (*metis distinct_card*)
  with *assms* **show** *?thesis*
    by(*simp add*: *Inv_def set_eq_iff*)
      (*metis before_in_def not_before_in before_in_swap*)
qed

Perform a list of swaps, from right to left:

abbreviation *swaps* **where** *swaps* == *foldr swap*

lemma *swaps_inv*[*simp*]:
  set (*swaps sws xs*) = *set xs* ∧
  size(*swaps sws xs*) = *size xs* ∧
  distinct(*swaps sws xs*) = *distinct xs*
by (*induct sws arbitrary*: *xs*) (*simp_all add*: *swap_def*)

lemma *swaps_eq_Nil_iff*[*simp*]: *swaps acts xs* = [] ⟷ *xs* = []
by(*induction acts*)(*auto simp*: *swap_def*)

lemma *swaps_map_Suc*[*simp*]:
  swaps (*map Suc sws*) (*a* # *xs*) = *a* # *swaps sws xs*

by(*induction sws arbitrary*: *xs*) *auto*

lemma *card_Inv_swaps_le*:
  distinct xs ⟹ *card* (*Inv xs* (*swaps sws xs*)) ≤ *length sws*
by(*induction sws*) (*auto simp*: *Inv_swap card_insert_if card_Diff_singleton_if*)

lemma *nth_swaps*: ∀ *i*∈*set is*. *j* < *i* ⟹ *swaps is xs* ! *j* = *xs* ! *j*
by(*induction is*)(*simp_all add*: *swap_def*)

lemma *not_before0*[*simp*]: ~ *x* < *xs* ! *0 in xs*
apply(*cases xs* = [])
by(*auto simp*: *before_in_def neq_Nil_conv*)

lemma *before_id*[*simp*]: ⟦ *distinct xs*; *i* < *size xs*; *j* < *size xs* ⟧ ⟹
  xs ! *i* < *xs* ! *j in xs* ⟷ *i* < *j*
by(*simp add*: *before_in_def index_nth_id*)

lemma *before_swaps*:
  ⟦ *distinct is*; ∀ *i*∈*set is*. *Suc i* < *size xs*; *distinct xs*; *i* ∉ *set is*; *i* < *j*; *j* <
size xs ⟧ ⟹
  swaps is xs ! *i* < *swaps is xs* ! *j in xs*
apply(*induction is arbitrary*: *i j*)
 apply *simp*
apply(*auto simp*: *swap_def nth_list_update*)
done

lemma *card_Inv_swaps*:
  ⟦ *distinct is*; ∀ *i*∈*set is*. *Suc i* < *size xs*; *distinct xs* ⟧ ⟹
  card(*Inv xs* (*swaps is xs*)) = *length is*
apply(*induction is*)
 apply *simp*
apply(*simp add*: *Inv_swap before_swaps card_insert_if*)
apply(*simp add*: *Inv_def*)
done

lemma *swaps_eq_nth_take_drop*: *i* < *length xs* ⟹
    swaps [*0*..<*i*] *xs* = *xs*!*i* # *take i xs* @ *drop* (*Suc i*) *xs*
apply(*induction i arbitrary*: *xs*)
apply (*auto simp add*: *neq_Nil_conv swap_def drop_update_swap*
  take_Suc_conv_app_nth Cons_nth_drop_Suc[*symmetric*])
done

lemma *index_swaps_size*: *distinct s* ⟹
  index s q ≤ *index* (*swaps sws s*) *q* + *length sws*

apply(*induction sws arbitrary*: *s*)
apply *simp*
 apply (*fastforce simp*: *swap_def index_swap_if_distinct index_nth_id*)
done

lemma *index_swaps_last_size*: *distinct s* $\Longrightarrow$
  size s $\leq$ *index* (*swaps sws s*) (*last s*) + *length sws* + *1*
apply(*cases s* = [])
 apply *simp*
using *index_swaps_size*[*of s last s sws*] **by** *simp*

end

3 Deterministic Online and Offline Algorithms

theory *On_Off*
imports *Complex_Main*
begin

type_synonym (*'s*,*'r*,*'a*) *alg_off* = *'s* ⇒ *'r list* ⇒ *'a list*
type_synonym (*'s*,*'is*,*'r*,*'a*) *alg_on* = (*'s* ⇒ *'is*) ∗ (*'s* ∗ *'is* ⇒ *'r* ⇒ *'a* ∗ *'is*)

locale *On_Off* =
fixes *step* :: *'state* ⇒ *'request* ⇒ *'answer* ⇒ *'state*
fixes *t* :: *'state* ⇒ *'request* ⇒ *'answer* ⇒ *nat*
fixes *wf* :: *'state* ⇒ *'request list* ⇒ *bool*
begin

fun *T* :: *'state* ⇒ *'request list* ⇒ *'answer list* ⇒ *nat* **where**
T s [] [] = *0* |
T s (*r*#*rs*) (*a*#*as*) = *t s r a* + *T* (*step s r a*) *rs as*

definition *Step* ::
  (*'state* , *'istate*, *'request*, *'answer*)*alg_on*
   ⇒ *'state* ∗ *'istate* ⇒ *'request* ⇒ *'state* ∗ *'istate*
where
Step A s r = (*let* (*a*,*is'*) = *snd A s r in* (*step* (*fst s*) *r a*, *is'*))

fun *config'* :: (*'state*,*'is*,*'request*,*'answer*) *alg_on* ⇒ (*'state*∗*'is*) ⇒ *'request list*
    ⇒ (*'state* ∗ *'is*) **where**
config' A s [] = *s* |

config′ A s (r#rs) = config′ A (Step A s r) rs

lemma *config′_snoc*: *config′ A s (rs@[r]) = Step A (config′ A s rs) r*
apply(*induct rs arbitrary*: *s*) **by** *simp_all*

lemma *config′_append2*: *config′ A s (xs@ys) = config′ A (config′ A s xs) ys*
apply(*induct xs arbitrary*: *s*) **by** *simp_all*

lemma *config′_induct*: P (*fst init*) $\Longrightarrow$ ($\bigwedge$*s q a*. P *s* $\Longrightarrow$ P (*step s q a*)) $\Longrightarrow$ P (*fst* (*config′ A init rs*))
apply (*induct rs arbitrary*: *init*) **by**(*simp_all add*: *Step_def split*: *prod.split*)

abbreviation *config* **where**
config A s0 rs == config′ A (s0, fst A s0) rs

lemma *config_snoc*: *config A s (rs@[r]) = Step A (config A s rs) r*
using *config′_snoc* **by** *metis*

lemma *config_append*: *config A s (xs@ys) = config′ A (config A s xs) ys*
using *config′_append2* **by** *metis*

lemma *config_induct*: P *s0* $\Longrightarrow$ ($\bigwedge$*s q a*. P *s* $\Longrightarrow$ P (*step s q a*)) $\Longrightarrow$ P (*fst* (*config A s0 qs*))
using *config′_induct*[*of P* (*s0*, *fst A s0*)] **by** *auto*

fun *T_on′* :: (*′state*,*′is*,*′request*,*′answer*) *alg_on* ⇒ (*′state*∗*′is*) ⇒ *′request list* ⇒ *nat* **where**
T_on′ A s [] = *0* |
T_on′ A s (r#rs) = (t (fst s) r (fst (snd A s r))) + T_on′ A (Step A s r) rs

lemma *T_on′_append*: *T_on′ A s (xs@ys) = T_on′ A s xs + T_on′ A (config′ A s xs) ys*
apply(*induct xs arbitrary*: *s*) **by** *simp_all*

abbreviation *T_on′′* :: (*′state*,*′is*,*′request*,*′answer*) *alg_on* ⇒ *′state* ⇒ *′request list* ⇒ *nat* **where**
T_on′′ A s rs == T_on′ A (s,fst A s) rs

lemma *T_on_append*: *T_on′′ A s (xs@ys) = T_on′′ A s xs + T_on′ A (config A s xs) ys*
by(*rule T_on′_append*)

abbreviation *T_on_n A s0 xs n == T_on′ A (config A s0 (take n xs))* [*xs*!*n*]

lemma *T_on___as_sum*: *T_on′′ A s0 rs = sum (T_on_n A s0 rs)* {..<*length rs*}
apply(*induct rs rule*: *rev_induct*)
  by(*simp_all add*: *T_on′_append nth_append*)

fun *off2* :: (′*state*,′*is*,′*request*,′*answer*) *alg_on* ⇒ (′*state* ∗ ′*is*,′*request*,′*answer*) *alg_off* **where**
off2 A s [] = [] |
off2 A s (*r*#*rs*) = *fst* (*snd A s r*) # *off2 A* (*Step A s r*) *rs*

abbreviation *off* :: (′*state*,′*is*,′*request*,′*answer*) *alg_on* ⇒ (′*state*,′*request*,′*answer*) *alg_off* **where**
off A s0 ≡ *off2 A* (*s0*, *fst A s0*)

abbreviation *T_off* :: (′*state*,′*request*,′*answer*) *alg_off* ⇒ ′*state* ⇒ ′*request list* ⇒ *nat* **where**
T_off A s0 rs == *T s0 rs* (*A s0 rs*)

abbreviation *T_on* :: (′*state*,′*is*,′*request*,′*answer*) *alg_on* ⇒ ′*state* ⇒ ′*request list* ⇒ *nat* **where**
T_on A == *T_off* (*off A*)

lemma *T_on_on*′: *T_off* (λ*s0*. (*off2 A* (*s0*, *x*))) *s0 qs* = *T_on*′ *A* (*s0*,*x*) *qs*
apply(*induct qs arbitrary*: *s0 x*)
  by(*simp_all add*: *Step_def split*: *prod.split*)

lemma *T_on_on*′′: *T_on A s0 qs* = *T_on*′′ *A s0 qs*
using *T_on_on*′[**where** *x*=*fst A s0*, *of s0 qs A*] **by**(*auto*)

lemma *T_on_as_sum*: *T_on A s0 rs = sum (T_on_n A s0 rs)* {..<*length rs*}

using *T_on___as_sum T_on_on''* **by** *metis*

definition *T_opt* :: *'state* ⇒ *'request list* ⇒ *nat* **where**
T_opt s rs = Inf {T s rs as | as. size as = size rs}

definition *compet* :: (*'state,'is,'request,'answer*) *alg_on* ⇒ *real* ⇒ *'state set* ⇒ *bool* **where**
compet A c S = (∀ *s*∈*S*. ∃ *b* ≥ *0*. ∀ *rs*. *wf s rs* ⟶ *real*(*T_on A s rs*) ≤ *c* ∗ *T_opt s rs* + *b*)

lemma *length_off*[*simp*]: *length*(*off2 A s rs*) = *length rs*
by (*induction rs arbitrary*: *s*) (*auto split*: *prod.split*)

lemma *compet_mono*: **assumes** *compet A c S0* **and** *c* ≤ *c'*
shows *compet A c' S0*
proof (*unfold compet_def*, *auto*)
  let *?compt* = λ*s0 rs b* (*c*::*real*). *T_on A s0 rs* ≤ *c* ∗ *T_opt s0 rs* + *b*
  fix *s0* **assume** *s0* ∈ *S0*
  with *assms*(*1*) **obtain** *b* **where** *b* ≥ *0* **and** *1*: ∀ *rs*. *wf s0 rs* ⟶ *?compt s0 rs b c*
    by(*auto simp*: *compet_def*)
  have ∀ *rs*. *wf s0 rs* ⟶ *?compt s0 rs b c'*
  proof *safe*
    fix *rs*
    assume *wf*: *wf s0 rs*
    from *1 wf* **have** *?compt s0 rs b c* **by** *blast*
    thus *?compt s0 rs b c'*
     using *1 mult_right_mono*[*OF assms*(*2*) *of_nat_0_le_iff*[*of T_opt s0 rs*]]
      by *arith*
  qed
  thus ∃ *b*≥*0*. ∀ *rs*. *wf s0 rs* ⟶ *?compt s0 rs b c'* **using** ‹*b*≥*0*› **by**(*auto*)
qed

lemma *competE*: **fixes** *c* :: *real*
assumes *compet A c S0 c* ≥ *0* ∀ *s0 rs*. *size*(*aoff s0 rs*) = *length rs s0*∈*S0*
shows ∃ *b*≥*0*. ∀ *rs*. *wf s0 rs* ⟶ *T_on A s0 rs* ≤ *c* ∗ *T_off aoff s0 rs* + *b*
proof −
  from *assms*(*1*,*4*) **obtain** *b* **where** *b*≥*0* **and**
    1: ∀ *rs*. *wf s0 rs* ⟶ *T_on A s0 rs* ≤ *c* ∗ *T_opt s0 rs* + *b*
    by(*auto simp add*: *compet_def*)
  { **fix** *rs*

assume *wf s0 rs*
then have *2*: *real*(*T_on A s0 rs*) ≤ *c* ∗ *Inf* {*T s0 rs as* | *as*. *size as* = *size rs*} + *b*
(**is** _ ≤ _ ∗ *real*(*Inf ?T*) + _)
using *1* **by**(*auto simp add*: *T_opt_def*)
have *Inf ?T* ≤ *T_off aoff s0 rs*
using *assms*(*3*) **by** (*intro cInf_lower*) *auto*
from *mult_left_mono*[*OF of_nat_le_iff*[*THEN iffD2*, *OF this*] *assms*(*2*)]
have *T_on A s0 rs* ≤ *c* ∗ *T_off aoff s0 rs* + *b* **using** *2* **by** *arith*
}
thus *?thesis* **using** ‹*b*≥*0*› **by**(*auto simp*: *compet_def*)
qed

end

end

4 Probability Theory

theory *Prob_Theory*
imports *HOL*−*Probability*.*Probability*
begin

lemma *integral_map_pmf*[*simp*]:
fixes *f*::*real* ⇒ *real*
shows (∫ *x*. *f x* ∂(*map_pmf g M*)) = (∫ *x*. *f* (*g x*) ∂*M*)
unfolding *map_pmf_rep_eq*
using *integral_distr*[*of g* (*measure_pmf M*) (*count_space UNIV*) *f*] **by** *auto*

4.1 function *E*

definition *E* :: *real pmf* ⇒ *real* **where**
E M = (∫ *x*. *x* ∂ *measure_pmf M*)

translations
∫ *x*. *f* ∂*M* <= *CONST lebesgue_integral M* (λ*x*. *f*)

notation (*latex* **output**) *E* (‹*E*[_]› [*1*] *100*)

lemma *E_const*[*simp*]: *E* (*return_pmf a*) = *a*
unfolding *E_def*
unfolding *return_pmf*.*rep_eq*
by (*simp add*: *integral_return*)

lemma *E_null*[*simp*]: *E* (*return_pmf 0*) = *0*
by *auto*

lemma *E_finite_sum*: *finite* (*set_pmf X*) $\Longrightarrow$ *E X* = ($\sum$ *x*∈(*set_pmf X*). *pmf X x* ∗ *x*)
unfolding *E_def* **by** (*subst integral_measure_pmf*) *simp_all*

lemma *E_of_const*: *E*(*map_pmf* (λ*x*. *y*) (*X*::*real pmf*)) = *y* **by** *auto*

lemma *E_nonneg*:
shows (∀ *x*∈*set_pmf X*. *0*≤ *x*) $\Longrightarrow$ *0* ≤ *E X*
unfolding *E_def*
using *integral_nonneg* **by** (*simp add*: *AE_measure_pmf_iff integral_nonneg_AE*)

lemma *E_nonneg_fun*: **fixes** *f*::′*a*⇒*real*
shows (∀ *x*∈*set_pmf X*. *0*≤*f x*) $\Longrightarrow$ *0* ≤ *E* (*map_pmf f X*)
using *E_nonneg* **by** *auto*

lemma *E_cong*:
fixes *f*::′*a* ⇒ *real*
shows *finite* (*set_pmf X*) $\Longrightarrow$ (∀ *x*∈ *set_pmf X*. (*f x*) = (*u x*)) $\Longrightarrow$ *E* (*map_pmf f X*) = *E* (*map_pmf u X*)
unfolding *E_def integral_map_pmf* **apply**(*rule integral_cong_AE*)
apply(*simp add*: *integrable_measure_pmf_finite*)+
by (*simp add*: *AE_measure_pmf_iff*)

lemma *E_mono3*:
fixes *f*::′*a* ⇒ *real*
shows *integrable* (*measure_pmf X*) *f* $\Longrightarrow$ *integrable* (*measure_pmf X*) *u* $\Longrightarrow$ (∀ *x*∈ *set_pmf X*. (*f x*) ≤ (*u x*)) $\Longrightarrow$ *E* (*map_pmf f X*) ≤ *E* (*map_pmf u X*)
unfolding *E_def integral_map_pmf* **apply**(*rule integral_mono_AE*)
by (*auto simp add*: *AE_measure_pmf_iff*)

lemma *E_mono2*:
fixes *f*::′*a* ⇒ *real*
shows *finite* (*set_pmf X*) $\Longrightarrow$ (∀ *x*∈ *set_pmf X*. (*f x*) ≤ (*u x*)) $\Longrightarrow$ *E* (*map_pmf f X*) ≤ *E* (*map_pmf u X*)
unfolding *E_def integral_map_pmf* **apply**(*rule integral_mono_AE*)
apply(*simp add*: *integrable_measure_pmf_finite*)+
by (*simp add*: *AE_measure_pmf_iff*)

lemma *E_linear_diff2*: *finite* (*set_pmf A*) $\Longrightarrow$ *E* (*map_pmf f A*) − *E*

(*map_pmf g A*) = *E* (*map_pmf* (λ*x*. (*f x*) − (*g x*)) *A*)
unfolding *E_def integral_map_pmf* **apply**(*rule Bochner_Integration.integral_diff*[*of measure_pmf A f g*, *symmetric*])
by (*simp_all add*: *integrable_measure_pmf_finite*)

lemma *E_linear_plus2*: *finite* (*set_pmf A*) ⟹ *E* (*map_pmf f A*) + *E* (*map_pmf g A*) = *E* (*map_pmf* (λ*x*. (*f x*) + (*g x*)) *A*)
unfolding *E_def integral_map_pmf* **apply**(*rule Bochner_Integration.integral_add*[*of measure_pmf A f g*, *symmetric*])
by (*simp_all add*: *integrable_measure_pmf_finite*)

lemma *E_linear_sum2*: *finite* (*set_pmf D*) ⟹ *E*(*map_pmf* (λ*x*. (∑ *i*<*up*. *f i x*)) *D*)
= (∑ *i*<(*up*::*nat*). *E*(*map_pmf* (*f i*) *D*))
unfolding *E_def integral_map_pmf* **apply**(*rule Bochner_Integration.integral_sum*)
by (*simp add*: *integrable_measure_pmf_finite*)

lemma *E_linear_sum_allg*: *finite* (*set_pmf D*) ⟹ *E*(*map_pmf* (λ*x*. (∑ *i*∈ *A*. *f i x*)) *D*)
= (∑ *i*∈ (*A*::′*a set*). *E*(*map_pmf* (*f i*) *D*))
unfolding *E_def integral_map_pmf* **apply**(*rule Bochner_Integration.integral_sum*)
by (*simp add*: *integrable_measure_pmf_finite*)

lemma *E_finite_sum_fun*: *finite* (*set_pmf X*) ⟹
E (*map_pmf f X*) = (∑ *x*∈*set_pmf X*. *pmf X x* ∗ *f x*)
proof −
assume *finite*: *finite* (*set_pmf X*)
have *E* (*map_pmf f X*) = (∫ *x*. *f x* ∂*measure_pmf X*)
unfolding *E_def* **by** *auto*
also have . . . = (∑ *x*∈*set_pmf X*. *pmf X x* ∗ *f x*)
by (*subst integral_measure_pmf*) (*auto simp add*: *finite*)
finally show *?thesis* **.**
qed

lemma *E_bernoulli*: *0*≤*p* ⟹ *p*≤*1* ⟹
E (*map_pmf f* (*bernoulli_pmf p*)) = *p*∗(*f True*) + (*1*−*p*)∗(*f False*)
unfolding *E_def* **by** (*auto*)

4.2 function *bv*

fun *bv*:: *nat* ⇒ *bool list pmf* **where**
bv 0 = *return_pmf* []
| *bv* (*Suc n*) = *do* {
(*xs*::*bool list*) ← *bv n*;

```
                    (x::bool) ← (bernoulli_pmf 0.5);
                    return_pmf (x#xs)
                  }

lemma bv_finite: finite (bv n)
by (induct n) auto

lemma len_bv_n: ∀ xs ∈ set_pmf (bv n). length xs = n
apply(induct n) by auto

lemma bv_set: set_pmf (bv n) = {x::bool list. length x = n}
proof (induct n)
  case (Suc n)
  then have set_pmf (bv (Suc n)) = (⋃x∈{x. length x = n}. {True # x,
False # x})
    by(simp add: set_pmf_bernoulli UNIV_bool)
  also have ... = {x#xs| x xs. length xs = n} by auto
   also have ... = {x. length x = Suc n}  using Suc_length_conv by
fastforce
  finally show ?case .
qed (simp)

lemma len_not_in_bv: length xs ≠ n ⟹ xs ∉ set_pmf (bv n)
by(auto simp: len_bv_n)

lemma not_n_bv_0: length xs ≠ n ⟹ pmf (bv n) xs = 0
by (simp add: len_not_in_bv pmf_eq_0_set_pmf)

lemma bv_comp_bernoulli: n < l
        ⟹ map_pmf (λy. y!n) (bv l) = bernoulli_pmf (5 / 10)
proof (induct n arbitrary: l)
  case 0
  then obtain m where l = Suc m by (metis Suc_pred)
 then show map_pmf (λy. y!0) (bv l) = bernoulli_pmf (5 / 10) by (auto
simp: map_pmf_def bind_return_pmf bind_assoc_pmf bind_return_pmf')
next
  case (Suc n)
  then have 0 < l by auto
  then obtain m where lsm: l = Suc m by (metis Suc_pred)
  with Suc(2) have nltm: n < m by auto

  from lsm have map_pmf (λy. y ! Suc n) (bv l)
       = map_pmf (λx. x!n) (bind_pmf (bv m) (λt. (return_pmf t))) by
(auto simp: map_bind_pmf)
```

also
 have ... = *map_pmf* (λ*x*. *x*!*n*) (*bv m*) **by** (*auto simp*: *bind_return_pmf*′)
also
 have ... = *bernoulli_pmf* (*5* / *10*) **by** (*auto simp add*: *Suc*(*1*)[*of m*, *OF nltm*])
finally
 show *?case* **.**
qed

lemma *pmf_2elemlist*: *pmf* (*bv* (*Suc 0*)) ([*x*]) = *pmf* (*bv 0*) [] ∗ *pmf* (*bernoulli_pmf* (*5* / *10*)) *x*
 unfolding *bv.simps*(*2*)[**where** *n*=*0*] *pmf_bind pmf_return*
 apply (*subst integral_measure_pmf*[**where** *A*={[]}])
 apply (*auto*) **by** (*cases x*) *auto*

lemma *pmf_moreelemlist*: *pmf* (*bv* (*Suc n*)) (*x*#*xs*) = *pmf* (*bv n*) *xs* ∗ *pmf* (*bernoulli_pmf* (*5* / *10*)) *x*
 unfolding *bv.simps*(*2*) *pmf_bind pmf_return*
 apply (*subst integral_measure_pmf*[**where** *A*={*xs*}])
 apply *auto* **apply** (*cases x*) **apply**(*auto*)
 apply (*meson indicator_simps*(*2*) *list.inject singletonD*)
 apply (*meson indicator_simps*(*2*) *list.inject singletonD*)
 apply (*cases x*) **by**(*auto*)

lemma *list_pmf*: *length xs* = *n* ⟹ *pmf* (*bv n*) *xs* = (*1* / *2*)^*n*
proof(*induct n arbitrary*: *xs*)
 case *0*
 then have *xs* = [] **by** *auto*
 then show *pmf* (*bv 0*) *xs* = (*1* / *2*) ^ *0* **by**(*auto*)
next
 case (*Suc n xs*)
 then obtain *a as* **where** *split*: *xs* = *a*#*as* **by** (*metis Suc_length_conv*)
 have *length as* = *n* **using** *Suc*(*2*) *split* **by** *auto*
 with *Suc*(*1*) **have** *1*: *pmf* (*bv n*) *as* = (*1* / *2*) ^ *n* **by** *auto*

 from *split pmf_moreelemlist*[**where** *n*=*n* **and** *x*=*a* **and** *xs*=*as*] **have**
  pmf (*bv* (*Suc n*)) *xs* = *pmf* (*bv n*) *as* ∗ *pmf* (*bernoulli_pmf* (*5* / *10*)) *a* **by** *auto*
 then have *pmf* (*bv* (*Suc n*)) *xs* = (*1* / *2*) ^ *n* ∗ *1* / *2* **using** *1* **by** *auto*
 then show *pmf* (*bv* (*Suc n*)) *xs* = (*1* / *2*) ^ *Suc n* **by** *auto*
qed

lemma *bv_0_notlen*: *pmf* (*bv n*) *xs* = *0* ⟹ *length xs* ≠ *n*
by(*auto simp*: *list_pmf*)

lemma *length xs* > *n* ⟹ *pmf* (*bv n*) *xs* = *0*
proof (*induct n arbitrary*: *xs*)
case (*Suc n xs*)
then obtain *a as* **where** *split*: *xs* = *a*#*as* **by** (*metis Suc_length_conv Suc_lessE*)
have *length as* > *n* **using** *Suc*(*2*) *split* **by** *auto*
with *Suc*(*1*) **have** *1*: *pmf* (*bv n*) *as* = *0* **by** *auto*
from *split pmf_moreelemlist*[**where** *n*=*n* **and** *x*=*a* **and** *xs*=*as*] **have**
pmf (*bv* (*Suc n*)) *xs* = *pmf* (*bv n*) *as* ∗ *pmf* (*bernoulli_pmf* (*5* / *10*)) *a* **by** *auto*
then have *pmf* (*bv* (*Suc n*)) *xs* = *0* ∗ *1* / *2* **using** *1* **by** *auto*
then show *pmf* (*bv* (*Suc n*)) *xs* = *0* **by** *auto*
qed *simp*

lemma *map_hd_list_pmf*: *map_pmf hd* (*bv* (*Suc n*)) = *bernoulli_pmf* (*5* / *10*)
by (*simp add*: *map_pmf_def bind_assoc_pmf bind_return_pmf bind_return_pmf'*)

lemma *map_tl_list_pmf*: *map_pmf tl* (*bv* (*Suc n*)) = *bv n*
by (*simp add*: *map_pmf_def bind_assoc_pmf bind_return_pmf bind_return_pmf'*)

4.3 function *flip*

fun *flip* :: *nat* ⇒ *bool list* ⇒ *bool list* **where**
flip _ [] = []
| *flip 0* (*x*#*xs*) = (¬*x*)#*xs*
| *flip* (*Suc n*) (*x*#*xs*) = *x*#(*flip n xs*)

lemma *flip_length*[*simp*]: *length* (*flip i xs*) = *length xs*
apply(*induct xs arbitrary*: *i*) **apply**(*simp*) **apply**(*case_tac i*) **by**(*simp_all*)

lemma *flip_out_of_bounds*: *y* ≥ *length X* ⟹ *flip y X* = *X*
apply(*induct X arbitrary*: *y*)
proof −
case (*Cons X Xs*)
hence *y* > *0* **by** *auto*
with *Cons* **obtain** *y'* **where** *y1*: *y* = *Suc y'* **and** *y2*: *y'* ≥ *length Xs* **by** (*metis Suc_pred' length_Cons not_less_eq_eq*)
then have *flip y* (*X* # *Xs*) = *X*#(*flip y' Xs*) **by** *auto*
moreover from *Cons y2* **have** *flip y' Xs* = *Xs* **by** *auto*
ultimately show *?case* **by** *auto*
qed *simp*

```
lemma flip_other: y < length X ⟹ z < length X ⟹ z ≠ y ⟹ flip z X
! y = X ! y
apply(induct y arbitrary: X z)
apply(simp) apply (metis flip.elims neq0_conv nth_Cons_0)
proof (case_tac z, goal_cases)
  case (1 y X z)
   then obtain a as where X=a#as using length_greater_0_conv by
(metis (full_types) flip.elims)
  with 1(5) show ?case by(simp)
next
  case (2 y X z z')
  from 2 have 3: z' ≠ y by auto
  from 2(2) have length X > 0 by auto
 then obtain a as where aas: X = a#as by (metis (full_types) flip.elims
length_greater_0_conv)
  then have a: flip (Suc z') X ! Suc y = flip z' as ! y
    and b : (X ! Suc y) = (as ! y) by auto
  from 2(2) aas have 1: y < length as by auto
  from 2(3,5) aas have f2: z' < length as by auto
  note c=2(1)[OF 1 f2 3]

  have flip z X ! Suc y = flip (Suc z') X ! Suc y using 2 by auto
  also have ... = flip z' as ! y by (rule a)
  also have ... = as ! y by (rule c)
  also have ... = (X ! Suc y) by (rule b[symmetric])
  finally show flip z X ! Suc y = (X ! Suc y) .
qed

lemma flip_itself: y < length X ⟹ flip y X ! y = (¬ X ! y)
apply(induct y arbitrary: X)
apply(simp) apply (metis flip.elims nth_Cons_0 old.nat.distinct(2))
proof -
  fix y
  fix X::bool list
  assume iH: (⋀X. y < length X ⟹ flip y X ! y = (¬ X ! y))
  assume len: Suc y < length X
  from len have y < length X by auto
  from len have length X > 0 by auto
  then obtain z zs where zzs: X = z#zs by (metis (full_types) flip.elims
length_greater_0_conv)
  then have a: flip (Suc y) X ! Suc y = flip y zs ! y
    and b : (¬ X ! Suc y) = (¬ zs ! y) by auto
  from len zzs have y < length zs by auto
```

  note *c=iH[OF this]*
  from *a b c* **show** *flip (Suc y) X ! Suc y = (¬ X ! Suc y)* **by** *auto*
qed

lemma *flip_twice*: *flip i (flip i b) = b*
proof *(cases i < length b)*
  case *True*
  then have *A*: *i < length (flip i b)* **by** *simp*
  show *?thesis* **apply**(*simp add*: *list_eq_iff_nth_eq*) **apply**(*clarify*)
  proof (*goal_cases*)
    case (*1 j*)
    then show *?case*
      apply(*cases i=j*)
        using *flip_itself[OF A] flip_itself[OF True]* **apply**(*simp*)
        using *flip_other True 1* **by** *auto*
  qed
qed (*simp add*: *flip_out_of_bounds*)

lemma *flipidiflip*: *y < length X ⟹ e < length X ⟹ flip e X ! y = (if e=y then ~ (X ! y) else X ! y)*
apply(*cases e=y*)
apply(*simp add*: *flip_itself*)
by(*simp add*: *flip_other*)

lemma *bernoulli_Not*: *map_pmf Not (bernoulli_pmf (1 / 2)) = (bernoulli_pmf (1 / 2))*
apply(*rule pmf_eqI*)
proof (*case_tac i*, *goal_cases*)
  case (*1 i*)
  then have *pmf (map_pmf Not (bernoulli_pmf (1 / 2))) i =*
    pmf (map_pmf Not (bernoulli_pmf (1 / 2))) (Not False) **by** *auto*
  also have ... = *pmf (bernoulli_pmf (1 / 2)) False* **apply** (*rule pmf_map_inj'*)
apply(*rule injI*) **by** *auto*
  also have ... = *pmf (bernoulli_pmf (1 / 2)) i* **by** *auto*
  finally show *?case* **.**
next
  case (*2 i*)
  then have *pmf (map_pmf Not (bernoulli_pmf (1 / 2))) i =*
    pmf (map_pmf Not (bernoulli_pmf (1 / 2))) (Not True) **by** *auto*
  also have ... = *pmf (bernoulli_pmf (1 / 2)) True* **apply** (*rule pmf_map_inj'*)
apply(*rule injI*) **by** *auto*
  also have ... = *pmf (bernoulli_pmf (1 / 2)) i* **by** *auto*
  finally show *?case* **.**
qed

lemma *inv_flip_bv*: *map_pmf* (*flip i*) (*bv n*) = (*bv n*)
proof(*induct n arbitrary*: *i*)
case (*Suc n i*)
note *iH*=*this*
have *bind_pmf* (*bv n*) (λ*x*. *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *map_pmf* (*flip i*) (*return_pmf* (*xa* # *x*))))
= *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa* .*bind_pmf* (*bv n*) (λ*x*. *map_pmf* (*flip i*) (*return_pmf* (*xa* # *x*))))
by(*rule bind_commute_pmf*)
also have . . . = *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa* . *bind_pmf* (*bv n*) (λ*x*. *return_pmf* (*xa* # *x*)))
proof (*cases i*)
case *0*
then have *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *bind_pmf* (*bv n*) (λ*x*. *map_pmf* (*flip i*) (*return_pmf* (*xa* # *x*))))
= *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *bind_pmf* (*bv n*) (λ*x*. *return_pmf* ((¬ *xa*) # *x*))) **by** *auto*
also have . . . = *bind_pmf* (*bv n*) (λ*x*. *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *return_pmf* ((¬ *xa*) # *x*)))
by(*rule bind_commute_pmf*)
also have . . .
= *bind_pmf* (*bv n*) (λ*x*. *bind_pmf* (*map_pmf Not* (*bernoulli_pmf* (*1* / *2*))) (λ*xa*. *return_pmf* (*xa* # *x*)))
by(*auto simp add*: *bind_map_pmf*)
also have . . . = *bind_pmf* (*bv n*) (λ*x*. *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *return_pmf* (*xa* # *x*))) **by** (*simp only*: *bernoulli_Not*)
also have . . . = *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *bind_pmf* (*bv n*) (λ*x*. *return_pmf* (*xa* # *x*)))
by(*rule bind_commute_pmf*)
finally show *?thesis* **.**
next
case (*Suc i′*)
have *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *bind_pmf* (*bv n*) (λ*x*. *map_pmf* (*flip i*) (*return_pmf* (*xa* # *x*))))
= *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *bind_pmf* (*bv n*) (λ*x*. *return_pmf* (*xa* # *flip i′ x*))) **unfolding** *Suc* **by**(*simp*)
also have . . . = *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *bind_pmf* (*map_pmf* (*flip i′*) (*bv n*)) (λ*x*. *return_pmf* (*xa* # *x*)))
by(*auto simp add*: *bind_map_pmf*)
also have . . . = *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *bind_pmf* (*bv n*) (λ*x*. *return_pmf* (*xa* # *x*)))
using *iH*[*of i′*] **by** *simp*
finally show *?thesis* **.**

qed
also have ... = *bind_pmf* (*bv n*) (λ*x*. *bind_pmf* (*bernoulli_pmf* (*1* / *2*)) (λ*xa*. *return_pmf* (*xa* # *x*)))
by(*rule bind_commute_pmf*)
finally show *?case* **by**(*simp add*: *map_pmf_def bind_assoc_pmf*)
qed *simp*

4.4 Example for pmf

definition *twocoins* =
do {
x ← (*bernoulli_pmf 0.4*);
y ← (*bernoulli_pmf 0.5*);
return_pmf (*x* ∨ *y*)
}

lemma *experiment0_7*: *pmf twocoins True* = *0.7*
unfolding *twocoins_def*
unfolding *pmf_bind pmf_return*
apply (*subst integral_measure_pmf*[**where** *A*={*True*, *False*}])
by *auto*

4.5 Sum Distribution

definition *Sum_pmf p Da Db* = (*bernoulli_pmf p*) ≫= (%*b*. *if b then map_pmf Inl Da else map_pmf Inr Db*)

lemma *b0*: *bernoulli_pmf 0* = *return_pmf False*
apply(*rule pmf_eqI*) **apply**(*case_tac i*)
by(*simp_all*)
lemma *b1*: *bernoulli_pmf 1* = *return_pmf True*
apply(*rule pmf_eqI*) **apply**(*case_tac i*)
by(*simp_all*)

lemma *Sum_pmf_0*: *Sum_pmf 0 Da Db* = *map_pmf Inr Db*
unfolding *Sum_pmf_def*
apply(*rule pmf_eqI*)
by(*simp add*: *b0 bind_return_pmf*)

lemma *Sum_pmf_1*: *Sum_pmf 1 Da Db* = *map_pmf Inl Da*
unfolding *Sum_pmf_def*
apply(*rule pmf_eqI*)
by(*simp add*: *b1 bind_return_pmf*)

```
definition Proj1_pmf D = map_pmf (%a. case a of Inl e ⇒ e) (cond_pmf
D {f. (∃ e. Inl e = f)})


lemma A: (case_sum (λe. e) (λa. undefined)) (Inl e) = e
  by(simp)

lemma B: inj (case_sum (λe. e) (λa. undefined))
  oops

lemma none: p >0 ⟹ p < 1 ⟹ (set_pmf (bernoulli_pmf p ≫=
          (λb. if b then map_pmf Inl Da else map_pmf Inr Db))
          ∩ {f. (∃ e. Inl e = f)}) ≠ {}
    apply(simp add: UNIV_bool)
      using set_pmf_not_empty by fast
lemma none2: p >0 ⟹ p < 1 ⟹  (set_pmf (bernoulli_pmf p ≫=
          (λb. if b then map_pmf Inl Da else map_pmf Inr Db))
          ∩ {f. (∃ e. Inr e = f)}) ≠ {}
    apply(simp add: UNIV_bool)
      using set_pmf_not_empty by fast

lemma C: set_pmf (Proj1_pmf (Sum_pmf 0.5 Da Db)) = set_pmf Da
proof –
  show ?thesis
    unfolding Sum_pmf_def Proj1_pmf_def
    apply simp
    using none[of 0.5 Da Db] apply(simp add: set_cond_pmf UNIV_bool)
      by force
qed

thm integral_measure_pmf

thm pmf_cond pmf_cond[OF none]

lemma proj1_pmf: assumes p>0 p<1 shows Proj1_pmf (Sum_pmf p
Da Db) =  Da
proof –

  have kl: ⋀e. pmf (map_pmf Inr Db) (Inl e) = 0
    apply(simp only: pmf_eq_0_set_pmf)
    apply(simp) by blast
```

```
  have ll: measure_pmf.prob
          (bernoulli_pmf p ≫=
           (λb. if b then map_pmf Inl Da else map_pmf Inr Db))
          {f. ∃ e. Inl e = f} = p
       using assms
     apply(simp add: integral_pmf[symmetric] pmf_bind)
     apply(subst Bochner_Integration.integral_add)
      using integrable_pmf apply fast
      using integrable_pmf apply fast
        by(simp add: integral_pmf)

  have E: (cond_pmf
      (bernoulli_pmf p ≫=
       (λb. if b then map_pmf Inl Da else map_pmf Inr Db))
      {f. ∃ e. Inl e = f}) =
    map_pmf Inl Da
    apply(rule pmf_eqI)
      apply(subst pmf_cond)
      using none[of p Da Db] assms apply (simp)
       using assms apply(auto)
          apply(subst pmf_bind)
          apply(simp add: kl ll )
          apply(simp only: pmf_eq_0_set_pmf) by auto

  have ID: case_sum (λe. e) (λa. undefined) ∘ Inl = id
    by fastforce
  show ?thesis
    unfolding Sum_pmf_def Proj1_pmf_def
    apply(simp only: E)
    apply(simp add: pmf.map_comp ID)
  done

qed


definition Proj2_pmf D = map_pmf (%a. case a of Inr e ⇒ e) (cond_pmf
D {f. (∃ e. Inr e = f)})

lemma proj2_pmf: assumes p>0 p<1 shows Proj2_pmf (Sum_pmf p
Da Db) =  Db
proof −

  have kl: ⋀e. pmf (map_pmf Inl Da) (Inr e) = 0
    apply(simp only: pmf_eq_0_set_pmf)
```

```
    apply(simp) by blast

  have ll: measure_pmf.prob
           (bernoulli_pmf p ≫=
            (λb. if b then map_pmf Inl Da else map_pmf Inr Db))
           {f. ∃ e. Inr e = f} = 1−p
      using assms
     apply(simp add: integral_pmf[symmetric] pmf_bind)
     apply(subst Bochner_Integration.integral_add)
      using integrable_pmf apply fast
      using integrable_pmf apply fast
        by(simp add: integral_pmf)

  have E: (cond_pmf
       (bernoulli_pmf p ≫=
        (λb. if b then map_pmf Inl Da else map_pmf Inr Db))
       {f. ∃ e. Inr e = f}) =
    map_pmf Inr Db
    apply(rule pmf_eqI)
      apply(subst pmf_cond)
      using none2[of p Da Db] assms apply (simp)
       using assms apply(auto)
          apply(subst pmf_bind)
          apply(simp add: kl ll )
          apply(simp only: pmf_eq_0_set_pmf) by auto

  have ID: case_sum (λe. undefined) (λa. a) ∘ Inr = id
    by fastforce
  show ?thesis
    unfolding Sum_pmf_def Proj2_pmf_def
    apply(simp only: E)
    apply(simp add: pmf.map_comp ID)
  done

qed
```

```
definition invSum invA invB D x i == invA (Proj1_pmf D) x i ∧ invB
(Proj2_pmf D) x i
```

```
lemma invSum_split: p>0 ⟹ p<1 ⟹ invA Da x i ⟹ invB Db x i ⟹
```

invSum invA invB (*Sum_pmf p Da Db*) *x i*
by(*simp add*: *invSum_def proj1_pmf proj2_pmf*)

term (*%a. case a of Inl e* ⇒ *Inl* (*fa e*) | *Inr e* ⇒ *Inr* (*fb e*))
definition *f_on2 fa fb* = (*%a. case a of Inl e* ⇒ *map_pmf Inl* (*fa e*) | *Inr e* ⇒ *map_pmf Inr* (*fb e*))

term *bind_pmf*

lemma *Sum_bind_pmf*: **assumes** *a*: *bind_pmf Da fa* = *Da′* **and** *b*: *bind_pmf Db fb* = *Db′*
shows *bind_pmf* (*Sum_pmf p Da Db*) (*f_on2 fa fb*)
= *Sum_pmf p Da′ Db′*
proof −
{ **fix** *x*
have (*if x then map_pmf Inl Da else map_pmf Inr Db*) ≫=
case_sum (*λe. map_pmf Inl* (*fa e*))
(*λe. map_pmf Inr* (*fb e*))
=
(*if x then map_pmf Inl Da* ≫= *case_sum* (*λe. map_pmf Inl* (*fa e*))
(*λe. map_pmf Inr* (*fb e*))
else map_pmf Inr Db ≫= *case_sum* (*λe. map_pmf Inl* (*fa e*))
(*λe. map_pmf Inr* (*fb e*)))
apply(*simp*) **done**
also
have . . . = (*if x then map_pmf Inl* (*bind_pmf Da fa*) *else map_pmf Inr* (*bind_pmf Db fb*))
by(*auto simp add*: *map_pmf_def bind_assoc_pmf bind_return_pmf*)
also
have . . . = (*if x then map_pmf Inl Da′ else map_pmf Inr Db′*)
using *a b* **by** *simp*
finally
have (*if x then map_pmf Inl Da else map_pmf Inr Db*) ≫=
case_sum (*λe. map_pmf Inl* (*fa e*))
(*λe. map_pmf Inr* (*fb e*)) = (*if x then map_pmf Inl Da′ else map_pmf Inr Db′*) **.**
} **note** *gr=this*

show *?thesis*
unfolding *Sum_pmf_def f_on2_def*
apply(*rule pmf_eqI*)

```
  apply(case_tac i)
  by(simp_all add: bind_return_pmf bind_assoc_pmf gr)
qed

definition sum_map_pmf fa fb = (%a. case a of Inl e ⇒ Inl (fa e) | Inr
e ⇒ Inr (fb e))

lemma Sum_map_pmf: assumes a: map_pmf fa Da = Da' and b: map_pmf
fb Db = Db'
 shows map_pmf (sum_map_pmf fa fb) (Sum_pmf p Da Db)
        = Sum_pmf p Da' Db'
proof -
 have map_pmf (sum_map_pmf fa fb) (Sum_pmf p Da Db)
      = bind_pmf (Sum_pmf p Da Db) (f_on2 (λx. return_pmf (fa x))
(λx. return_pmf (fb x)))
     using a b
 unfolding map_pmf_def sum_map_pmf_def f_on2_def
  by(auto simp add: bind_return_pmf sum.case_distrib)
also
 have ... = Sum_pmf p Da' Db'
 using assms[unfolded map_pmf_def]
 by(rule Sum_bind_pmf )
finally
 show ?thesis .
qed



end
```

5 Randomized Online and Offline Algorithms

```
theory Competitive_Analysis
imports
 Prob_Theory
 On_Off
begin
```

5.1 Competitive Analysis Formalized

```
type_synonym ('s,'is,'r,'a)alg_on_step = ('s * 'is ⇒ 'r ⇒ ('a * 'is)
pmf)
type_synonym ('s,'is)alg_on_init = ('s ⇒ 'is pmf)
type_synonym ('s,'is,'q,'a)alg_on_rand = ('s,'is)alg_on_init * ('s,'is,'q,'a)alg_on_step
```

5.1.1 classes of algorithms

definition *deterministic_init* :: ('s,'is)*alg_on_init* ⇒ *bool* **where**
deterministic_init I ⟷ (∀ *init*. *card*(*set_pmf* (*I init*)) = *1*)

definition *deterministic_step* :: ('s,'is,'q,'a)*alg_on_step* ⇒ *bool* **where**
deterministic_step S ⟷ (∀ *i is q*. *card*(*set_pmf* (*S* (*i*, *is*) *q*)) = *1*)

definition *random_step* :: ('s,'is,'q,'a)*alg_on_step* ⇒ *bool* **where**
random_step S ⟷ ~ *deterministic_step S*

5.1.2 Randomized Online and Offline Algorithms

context *On_Off*
begin

fun *steps* **where**
steps s [] [] = *s*
| *steps s* (*q*#*qs*) (*a*#*as*) = *steps* (*step s q a*) *qs as*

lemma *steps_append*: *length qs* = *length as* ⟹ *steps s* (*qs*@*qs'*) (*as*@*as'*) = *steps* (*steps s qs as*) *qs' as'*
apply(*induct qs as arbitrary*: *s rule*: *list_induct2*)
by *simp_all*

lemma *T_append*: *length qs* = *length as* ⟹ *T s* (*qs*@[*q*]) (*as*@[*a*]) = *T s qs as* + *t* (*steps s qs as*) *q a*
apply(*induct qs as arbitrary*: *s rule*: *list_induct2*)
by *simp_all*

lemma *T_append2*: *length qs* = *length as* ⟹ *T s* (*qs*@*qs'*) (*as*@*as'*) = *T s qs as* + *T* (*steps s qs as*) *qs' as'*
apply(*induct qs as arbitrary*: *s rule*: *list_induct2*)
by *simp_all*

abbreviation *Step_rand* :: ('state,'is,'request,'answer) *alg_on_rand* ⇒ 'request ⇒ 'state ∗ 'is ⇒ ('state ∗ 'is) *pmf* **where**
Step_rand A r s ≡ *bind_pmf* ((*snd A*) *s r*) (λ(*a*,*is'*). *return_pmf* (*step* (*fst s*) *r a*, *is'*))

fun *config'_rand* :: (*'state*,*'is*,*'request*,*'answer*) *alg_on_rand* ⇒ (*'state*∗*'is*) *pmf* ⇒ *'request list*
⇒ (*'state* ∗ *'is*) *pmf* **where**
config'_rand A s [] = *s* |
config'_rand A s (*r*#*rs*) = *config'_rand A* (*s* ≫= *Step_rand A r*) *rs*

lemma *config'_rand_snoc*: *config'_rand A s* (*rs*@[*r*]) = *config'_rand A s rs* ≫= *Step_rand A r*
apply(*induct rs arbitrary*: *s*) **by**(*simp_all*)

lemma *config'_rand_append*: *config'_rand A s* (*xs*@*ys*) = *config'_rand A* (*config'_rand A s xs*) *ys*
apply(*induct xs arbitrary*: *s*) **by**(*simp_all*)

abbreviation *config_rand* **where**
config_rand A s0 rs == *config'_rand A* ((*fst A s0*) ≫= (λ*is*. *return_pmf* (*s0*, *is*))) *rs*

lemma *config'_rand_induct*: (∀ *x* ∈ *set_pmf init*. *P* (*fst x*)) ⟹ (⋀*s q a*. *P s* ⟹ *P* (*step s q a*))
⟹ ∀ *x*∈*set_pmf* (*config'_rand A init qs*). *P* (*fst x*)
proof (*induct qs arbitrary*: *init*)
case (*Cons r rs*)
show *?case* **apply**(*simp*)
apply(*rule Cons*(*1*))
apply(*subst Set.ball_simps*(*9*)[**where** *P*=*P*, *symmetric*])
apply(*subst set_map_pmf*[*symmetric*])
apply(*simp only*: *map_bind_pmf*)
apply(*simp add*: *bind_assoc_pmf bind_return_pmf split_def*)
using *Cons*(*2*,*3*) **apply** *blast*
by *fact*
qed (*simp*)

lemma *config_rand_induct*: *P s0* ⟹ (⋀*s q a*. *P s* ⟹ *P* (*step s q a*)) ⟹ ∀ *x*∈*set_pmf* (*config_rand A s0 qs*). *P* (*fst x*)
using *config'_rand_induct*[*of* ((*fst A s0*) ≫= (λ*is*. *return_pmf* (*s0*, *is*))) *P*] **by** *auto*

fun *T_on_rand'* :: (*'state*,*'is*,*'request*,*'answer*) *alg_on_rand* ⇒ (*'state*∗*'is*) *pmf* ⇒ *'request list* ⇒ *real* **where**
T_on_rand' A s [] = *0* |
T_on_rand' A s (*r*#*rs*) = *E* (*s* ≫= (λ*s*. *bind_pmf* (*snd A s r*) (λ(*a*,*is'*).

return_pmf (*real* (*t* (*fst s*) *r a*)))))
+ *T_on_rand' A* (*s* ≫= *Step_rand A r*) *rs*

lemma *T_on_rand'_append*: *T_on_rand' A s* (*xs*@*ys*) = *T_on_rand' A s xs* + *T_on_rand' A* (*config'_rand A s xs*) *ys*
apply(*induct xs arbitrary*: *s*) **by** *simp_all*

abbreviation *T_on_rand* :: (*'state*,*'is*,*'request*,*'answer*) *alg_on_rand* ⇒ *'state* ⇒ *'request list* ⇒ *real* **where**
T_on_rand A s rs == *T_on_rand' A* (*fst A s* ≫= (*λis*. *return_pmf* (*s*,*is*))) *rs*

lemma *T_on_rand_append*: *T_on_rand A s* (*xs*@*ys*) = *T_on_rand A s xs* + *T_on_rand' A* (*config_rand A s xs*) *ys*
by(*rule T_on_rand'_append*)

abbreviation *T_on_rand'_n A s0 xs n* == *T_on_rand' A* (*config'_rand A s0* (*take n xs*)) [*xs*!*n*]

lemma *T_on_rand'_as_sum*: *T_on_rand' A s0 rs* = *sum* (*T_on_rand'_n A s0 rs*) {..<*length rs*}
apply(*induct rs rule*: *rev_induct*)
by(*simp_all add*: *T_on_rand'_append nth_append*)

abbreviation *T_on_rand_n A s0 xs n* == *T_on_rand' A* (*config_rand A s0* (*take n xs*)) [*xs*!*n*]

lemma *T_on_rand_as_sum*: *T_on_rand A s0 rs* = *sum* (*T_on_rand_n A s0 rs*) {..<*length rs*}
apply(*induct rs rule*: *rev_induct*)
by(*simp_all add*: *T_on_rand'_append nth_append*)

lemma *T_on_rand'_nn*: *T_on_rand' A s qs* ≥ *0*
apply(*induct qs arbitrary*: *s*)
apply(*simp_all add*: *bind_return_pmf*)
apply(*rule add_nonneg_nonneg*)
apply(*rule E_nonneg*)
by(*simp_all add*: *split_def*)

lemma *T_on_rand_nn*: *T_on_rand* (*I*,*S*) *s0 qs* ≥ *0*

by (*rule T_on_rand′_nn*)

definition *compet_rand* :: (′*state*,′*is*,′*request*,′*answer*) *alg_on_rand* ⇒ *real* ⇒ ′*state set* ⇒ *bool* **where**
compet_rand A c S0 = (∀ *s*∈*S0*. ∃ *b* ≥ *0*. ∀ *rs*. *wf s rs* ⟶ *T_on_rand A s rs* ≤ *c* ∗ *T_opt s rs* + *b*)

5.2 embeding of deterministic into randomized algorithms

fun *embed* :: (′*state*,′*is*,′*request*,′*answer*) *alg_on* ⇒ (′*state*,′*is*,′*request*,′*answer*) *alg_on_rand* **where**
embed A = ((λ*s*. *return_pmf* (*fst A s*)) ,
(λ*s r*. *return_pmf* (*snd A s r*)))

lemma *T_deter_rand*: *T_off* (λ*s0*. (*off2 A* (*s0*, *x*))) *s0 qs* = *T_on_rand′* (*embed A*) (*return_pmf* (*s0*,*x*)) *qs*
apply(*induct qs arbitrary*: *s0 x*)
by(*simp_all add*: *Step_def bind_return_pmf split*: *prod.split*)

lemma *config′_embed*: *config′_rand* (*embed A*) (*return_pmf s0*) *qs* = *return_pmf* (*config′ A s0 qs*)
apply(*induct qs arbitrary*: *s0*)
apply(*simp_all add*: *Step_def split_def bind_return_pmf*) **by** *metis*

lemma *config_embed*: *config_rand* (*embed A*) *s0 qs* = *return_pmf* (*config A s0 qs*)
apply(*simp add*: *bind_return_pmf*)
apply(*subst config′_embed*[*unfolded embed.simps*])
by *simp*

lemma *T_on_embed*: *T_on A s0 qs* = *T_on_rand* (*embed A*) *s0 qs*
using *T_deter_rand*[**where** *x*=*fst A s0*, *of s0 qs A*] **by**(*auto simp*: *bind_return_pmf*)

lemma *T_on′_embed*: *T_on′ A* (*s0*,*x*) *qs* = *T_on_rand′* (*embed A*) (*return_pmf* (*s0*,*x*)) *qs*
using *T_deter_rand T_on_on′* **by** *metis*

lemma *compet_embed*: *compet A c S0* = *compet_rand* (*embed A*) *c S0*
unfolding *compet_def compet_rand_def* **using** *T_on_embed* **by** *metis*

end

end

6 Deterministic List Update

theory *Move_to_Front*
imports
  Swaps
  On_Off
  Competitive_Analysis
begin

declare *Let_def*[*simp*]

6.1 Function *mtf*

definition *mtf* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
mtf x xs =
 (*if x* ∈ *set xs then x* # (*take* (*index xs x*) *xs*) @ *drop* (*index xs x* + *1*) *xs*
  else xs)

lemma *mtf_id*[*simp*]: *x* ∉ *set xs* ⟹ *mtf x xs* = *xs*
by(*simp add*: *mtf_def*)

lemma *mtf0*[*simp*]: *x* ∈ *set xs* ⟹ *mtf x xs* ! *0* = *x*
by(*auto simp*: *mtf_def*)

lemma *before_in_mtf*: **assumes** *z* ∈ *set xs*
shows *x* < *y in mtf z xs* ⟷
      (*y* ≠ *z* ∧ (*if x*=*z then y* ∈ *set xs else x* < *y in xs*))
proof−
  have *0*: *index xs z* < *size xs* **by** (*metis assms index_less_size_conv*)
  let *?xs* = *take* (*index xs z*) *xs* @ *xs* ! *index xs z* # *drop* (*Suc* (*index xs z*))
xs
  have *x* < *y in mtf z xs* = (*y* ≠ *z* ∧ (*if x*=*z then y* ∈ *set ?xs else x* < *y in*
?xs))
    using *assms*
    by (*auto simp add*: *mtf_def before_in_def index_append*)
    (*metis index_take index_take_if_set le_add1 le_trans less_imp_le_nat*)
  with *id_take_nth_drop*[*OF 0*, *symmetric*] **show** *?thesis* **by**(*simp*)

qed

lemma *Inv_mtf*: *set xs* = *set ys* ⟹ *z* : *set ys* ⟹ *Inv xs* (*mtf z ys*) =
Inv xs ys ∪ {(*x*,*z*)|*x*. *x* < *z in xs* ∧ *x* < *z in ys*}
− {(*z*,*x*)|*x*. *z* < *x in xs* ∧ *x* < *z in ys*}
by(*auto simp add*: *Inv_def before_in_mtf not_before_in dest*: *before_in_setD1*)

lemma *set_mtf*[*simp*]: *set*(*mtf x xs*) = *set xs*
by(*simp add*: *mtf_def*)
(*metis append_take_drop_id Cons_nth_drop_Suc index_less le_refl Un_insert_right nth_index set_append set_simps*(*2*))

lemma *length_mtf*[*simp*]: *size* (*mtf x xs*) = *size xs*
by (*auto simp add*: *mtf_def min_def*) (*metis index_less_size_conv leD*)

lemma *distinct_mtf*[*simp*]: *distinct* (*mtf x xs*) = *distinct xs*
by (*metis length_mtf set_mtf card_distinct distinct_card*)

6.2 Function *mtf2*

definition *mtf2* :: *nat* ⇒ ′*a* ⇒ ′*a list* ⇒ ′*a list* **where**
mtf2 n x xs =
(*if x* : *set xs then swaps* [*index xs x* − *n*..<*index xs x*] *xs else xs*)

lemma *mtf_eq_mtf2*: *mtf x xs* = *mtf2* (*length xs* − *1*) *x xs*
proof −
  have *x* : *set xs* ⟹ *index xs x* − (*size xs* − *Suc 0*) = *0*
    by (*auto simp*: *less_Suc_eq_le*[*symmetric*])
  thus *?thesis*
    by(*auto simp*: *mtf_def mtf2_def swaps_eq_nth_take_drop*)
qed

lemma *mtf20*[*simp*]: *mtf2 0 x xs* = *xs*
by(*auto simp add*: *mtf2_def*)

lemma *length_mtf2*[*simp*]: *length* (*mtf2 n x xs*) = *length xs*
by (*auto simp*: *mtf2_def index_less_size_conv*[*symmetric*]
  simp del:*index_conv_size_if_notin*)

lemma *set_mtf2*[*simp*]: *set*(*mtf2 n x xs*) = *set xs*
by (*auto simp*: *mtf2_def index_less_size_conv*[*symmetric*]
  simp del:*index_conv_size_if_notin*)

lemma *distinct_mtf2*[*simp*]: *distinct* (*mtf2 n x xs*) = *distinct xs*

```
by (metis length_mtf2 set_mtf2 card_distinct distinct_card)

lemma card_Inv_mtf2: xs!j = ys!0 ⟹ j < length xs ⟹ dist_perm xs ys
⟹
   card (Inv (swaps [i..<j] xs) ys) = card (Inv xs ys) − int(j−i)
proof(induction j arbitrary: xs)
  case (Suc j)
  show ?case
  proof cases
    assume i > j thus ?thesis by simp
  next
    assume [arith]: ¬ i > j
    have 0: Suc j < length ys by (metis Suc.prems(2,3) distinct_card)
    have 1: (ys ! 0, xs ! j) : Inv ys xs
    proof (auto simp: Inv_def)
      show ys ! 0 < xs ! j in ys using Suc.prems
           by (metis Suc_lessD n_not_Suc_n not_before0 not_before_in
nth_eq_iff_index_eq nth_mem)
      show xs ! j < ys ! 0 in xs using Suc.prems
        by (metis Suc_lessD before_id lessI)
    qed
    have 2: card(Inv ys xs) ≠ 0 using 1 by auto
    have int(card (Inv (swaps [i..<Suc j] xs) ys)) =
          card (Inv (swap j xs) ys) − int (j−i) using Suc by simp
    also have ... = card (Inv ys (swap j xs)) − int (j−i)
      by(simp add: card_Inv_sym)
    also have ... = card (Inv ys xs − {(ys ! 0, xs ! j)}) − int (j − i)
      using Suc.prems 0 by(simp add: Inv_swap)
    also have ... = int(card (Inv ys xs) − 1) − (j − i)
      using 1 by(simp add: card_Diff_singleton)
    also have ... = card (Inv ys xs) − int (Suc j − i) using 2 by arith
     also have ... = card (Inv xs ys) − int (Suc j − i) by(simp add:
card_Inv_sym)
    finally show ?thesis .
  qed
qed simp
```

6.3 Function Lxy

```
definition Lxy :: 'a list ⇒ 'a set ⇒ 'a list where
  Lxy xs S = filter (λz. z∈S) xs
thm inter_set_filter

lemma Lxy_length_cons: length (Lxy xs S) ≤ length (Lxy (x#xs) S)
```

unfolding *Lxy_def* **by**(*simp*)

lemma *Lxy_empty*[*simp*]: *Lxy* [] *S* = []
unfolding *Lxy_def* **by** *simp*

lemma *Lxy_set_filter*: *set* (*Lxy xs S*) = *S* ∩ *set xs*
by (*simp add*: *Lxy_def inter_set_filter*)

lemma *Lxy_distinct*: *distinct xs* ⟹ *distinct* (*Lxy xs S*)
by (*simp add*: *Lxy_def*)

lemma *Lxy_append*: *Lxy* (*xs*@*ys*) *S* = *Lxy xs S* @ *Lxy ys S*
by(*simp add*: *Lxy_def*)

lemma *Lxy_snoc*: *Lxy* (*xs*@[*x*]) *S* = (*if x*∈*S then Lxy xs S* @ [*x*] *else Lxy xs S*)
by(*simp add*: *Lxy_def*)

lemma *Lxy_not*: *S* ∩ *set xs* = {} ⟹ *Lxy xs S* = []
unfolding *Lxy_def* **apply**(*induct xs*) **by** *simp_all*

lemma *Lxy_notin*: *set xs* ∩ *S* = {} ⟹ *Lxy xs S* = []
apply(*induct xs*) **by**(*simp_all add*: *Lxy_def*)

lemma *Lxy_in*: *x*∈*S* ⟹ *Lxy* [*x*] *S* = [*x*]
by(*simp add*: *Lxy_def*)

lemma *Lxy_project*:
  assumes *x*≠*y x* ∈ *set xs* *y*∈*set xs distinct xs*
    and *x* < *y in xs*
  shows *Lxy xs* {*x*,*y*} = [*x*,*y*]
proof −
  from *assms* **have** *ij*: *index xs x* < *index xs y*
      and *xinxs*: *index xs x* < *length xs*
      and *yinxs*: *index xs y* < *length xs* **unfolding** *before_in_def* **by** *auto*

  from *xinxs* **obtain** *a as* **where** *dec1*: *a* @ [*xs*!*index xs x*] @ *as* = *xs*
      and *a* = *take* (*index xs x*) *xs* **and** *as* = *drop* (*Suc* (*index xs x*)) *xs*
      and *length_a*: *length a* = *index xs x* **and** *length_as*: *length as* = *length xs* − *index xs x*− *1*

using *id_take_nth_drop* **by** *fastforce*
have *index xs y≥length* (*a* @ [*xs*!*index xs x*]) **using** *length_a ij* **by** *auto*
then have ((*a* @ [*xs*!*index xs x*]) @ *as*) ! *index xs y* = *as* ! (*index xs y−length* (*a* @ [*xs* ! *index xs x*])) **using** *nth_append*[**where** *xs*=*a* @ [*xs*!*index xs x*] **and** *ys*=*as*]
by(*simp*)
then have *xsj*: *xs* ! *index xs y* = *as* ! (*index xs y−index xs x−1*) **using** *dec1 length_a* **by** *auto*
have *las*: (*index xs y−index xs x−1*) < *length as* **using** *length_as yinxs ij* **by** *simp*
obtain *b c* **where** *dec2*: *b* @ [*xs*!*index xs y*] @ *c* = *as*
and *b* = *take* (*index xs y−index xs x−1*) *as c*=*drop* (*Suc* (*index xs y−index xs x−1*)) *as*
and *length_b*: *length b* = *index xs y−index xs x−1* **using** *id_take_nth_drop*[*OF las*] *xsj* **by** *force*
have *xs_dec*: *a* @ [*xs*!*index xs x*] @ *b* @ [*xs*!*index xs y*] @ *c* = *xs* **using** *dec1 dec2* **by** *auto*

from *xs_dec assms*(*4*) **have** *distinct* ((*a* @ [*xs*!*index xs x*] @ *b* @ [*xs*!*index xs y*]) @ *c*) **by** *simp*
then have *c_empty*: *set c* ∩ {*x*,*y*} = {}
and *b_empty*: *set b* ∩ {*x*,*y*} = {}**and** *a_empty*: *set a* ∩ {*x*,*y*} = {} **by**(*auto simp add*: *assms*(*2*,*3*))

have *Lxy* (*a* @ [*xs*!*index xs x*] @ *b* @ [*xs*!*index xs y*] @ *c*) {*x*,*y*} = [*x*,*y*]
apply(*simp only*: *Lxy_append*)
apply(*simp add*: *assms*(*2*,*3*))
using *a_empty b_empty c_empty* **by**(*simp add*: *Lxy_notin Lxy_in*)

with *xs_dec* **show** *?thesis* **by** *auto*
qed

lemma *Lxy_mono*: {*x*,*y*} ⊆ *set xs* ⟹ *distinct xs* ⟹ *x* < *y in xs* = *x* < *y in Lxy xs* {*x*,*y*}
apply(*cases x*=*y*)
apply(*simp add*: *before_in_irefl*)
proof −
assume *xyset*: {*x*,*y*} ⊆ *set xs*
assume *dxs*: *distinct xs*
assume *xy*: *x*≠*y*
{
fix *x y*
assume *1*: {*x*,*y*} ⊆ *set xs*

```
    assume xny: x≠y
    assume 3: x < y in xs
    have Lxy xs {x,y} = [x,y] apply(rule Lxy_project)
          using xny 1 3 dxs by(auto)
   then have x < y in Lxy xs {x,y} using xny by(simp add: before_in_def)
  } note aha=this
  have a: x < y in xs ⟹ x < y in Lxy xs {x,y}
    apply(subst Lxy_project)
      using xy xyset dxs by(simp_all add: before_in_def)
  have t: {x,y}={y,x} by(auto)
  have f: ~ x < y in xs ⟹ y < x in Lxy xs {x,y}
    unfolding t
    apply(rule aha)
      using xyset apply(simp)
      using xy apply(simp)
      using xy xyset by(simp add: not_before_in)
  have b: ~ x < y in xs ⟹ ~ x < y in Lxy xs {x,y}
  proof -
    assume ~ x < y in xs
    then have y < x in Lxy xs {x,y} using f by auto
   then have ~ x < y in Lxy xs {x,y} using xy by(simp add: not_before_in)
    then show ?thesis .
  qed
  from a b
  show ?thesis by metis
qed
```

6.4 List Update as Online/Offline Algorithm

```
type_synonym 'a state = 'a list
type_synonym answer = nat * nat list

definition step :: 'a state ⇒ 'a ⇒ answer ⇒ 'a state where
step s r a =
  (let (k,sws) = a in mtf2 k r (swaps sws s))

definition t :: 'a state ⇒ 'a ⇒ answer ⇒ nat where
t s r a = (let (mf,sws) = a in index (swaps sws s) r + 1 + size sws)

definition static where static s rs = (set rs ⊆ set s)

interpretation On_Off step t static .

type_synonym 'a alg_off = 'a state ⇒ 'a list ⇒ answer list
```

type_synonym ('*a*,'*is*) *alg_on* = ('*a state*,'*is*,'*a*,*answer*) *alg_on*

lemma *T_ge_len*: *length as* = *length rs* $\Longrightarrow$ *T s rs as* $\geq$ *length rs*
by(*induction arbitrary*: *s rule*: *list_induct2*)
  (*auto simp*: *t_def trans_le_add2*)

lemma *T_off_neq0*: ($\bigwedge$*rs s0*. *size*(*alg s0 rs*) = *length rs*) $\Longrightarrow$
  rs $\neq$ [] $\Longrightarrow$ *T_off alg s0 rs* $\neq$ *0*
apply(*erule_tac x*=*rs* **in** *meta_allE*)
apply(*erule_tac x*=*s0* **in** *meta_allE*)
apply (*auto simp*: *neq_Nil_conv length_Suc_conv t_def*)
done

lemma *length_step*[*simp*]: *length* (*step s r as*) = *length s*
by(*simp add*: *step_def split_def*)

lemma *step_Nil_iff*[*simp*]: *step xs r act* = [] $\longleftrightarrow$ *xs* = []
by(*auto simp add*: *step_def mtf2_def split*: *prod.splits*)

lemma *set_step2*: *set*(*step s r* (*mf*,*sws*)) = *set s*
by(*auto simp add*: *step_def*)

lemma *set_step*: *set*(*step s r act*) = *set s*
by(*cases act*)(*simp add*: *set_step2*)

lemma *distinct_step*: *distinct*(*step s r as*) = *distinct s*
by (*auto simp*: *step_def split_def*)

6.5 Online Algorithm Move-to-Front is 2-Competitive

definition *MTF* :: ('*a*,*unit*) *alg_on* **where**
MTF = (λ_. (), λ*s r*. ((*size* (*fst s*) − *1*,[]), ()))

It was first proved by Sleator and Tarjan [ST85] that the Move-to-Front algorithm is 2-competitive.

lemma *potential*:
fixes *t* :: *nat* $\Rightarrow$ '*a*::*linordered_ab_group_add* **and** *p* :: *nat* $\Rightarrow$ '*a*
assumes *p0*: *p 0* = *0* **and** *ppos*: $\bigwedge$*n*. *p n* $\geq$ *0*
and *ub*: $\bigwedge$*n*. *t n* + *p*(*n*+*1*) − *p n* $\leq$ *u n*
shows ($\sum$ *i*<*n*. *t i*) $\leq$ ($\sum$ *i*<*n*. *u i*)
proof−
  let *?a* = λ*n*. *t n* + *p*(*n*+*1*) − *p n*
  have *1*: ($\sum$ *i*<*n*. *t i*) = ($\sum$ *i*<*n*. *?a i*) − *p*(*n*)
    by(*induction n*) (*simp_all add*: *p0*)

thus *?thesis*
by (*metis* (*erased*, *lifting*) *add.commute diff_add_cancel le_add_same_cancel2 order.trans ppos sum_mono ub*)
qed

lemma *potential2*:
fixes t :: *nat* ⇒ *'a*::*linordered_ab_group_add* **and** p :: *nat* ⇒ *'a*
assumes *p0*: $p\ 0 = 0$ **and** *ppos*: $\bigwedge n.\ p\ n \geq 0$
and *ub*: $\bigwedge m.\ m<n \Longrightarrow t\ m + p(m+1) - p\ m \leq u\ m$
shows $(\sum i<n.\ t\ i) \leq (\sum i<n.\ u\ i)$
proof−
let $?a = \lambda n.\ t\ n + p(n+1) - p\ n$
have $(\sum i<n.\ t\ i) = (\sum i<n.\ ?a\ i) - p(n)$ **by**(*induction n*) (*simp_all add*: *p0*)
also have $\ldots \leq (\sum i<n.\ ?a\ i)$ **using** *ppos* **by** *auto*
also have $\ldots \leq (\sum i<n.\ u\ i)$ **apply**(*rule sum_mono*) **apply**(*rule ub*) **by** *auto*
finally show *?thesis* **.**
qed

abbreviation *before x xs* ≡ $\{y.\ y < x$ *in xs*$\}$
abbreviation *after x xs* ≡ $\{y.\ x < y$ *in xs*$\}$

lemma *finite_before*[*simp*]: *finite* (*before x xs*)
apply(*rule finite_subset*[**where** B = *set xs*])
apply (*auto dest*: *before_in_setD1*)
done

lemma *finite_after*[*simp*]: *finite* (*after x xs*)
apply(*rule finite_subset*[**where** B = *set xs*])
apply (*auto dest*: *before_in_setD2*)
done

lemma *before_conv_take*:
x : *set xs* ⟹ *before x xs* = *set*(*take* (*index xs x*) *xs*)
by (*auto simp add*: *before_in_def set_take_if_index index_le_size*) (*metis index_take leI*)

lemma *card_before*: *distinct xs* ⟹ x : *set xs* ⟹ *card* (*before x xs*) = *index xs x*
using *index_le_size*[*of xs x*]
by(*simp add*: *before_conv_take distinct_card*[*OF distinct_take*] *min_def*)

lemma *before_Un*: *set xs* = *set ys* ⟹ *x* : *set xs* ⟹
  before x ys = *before x xs* ∩ *before x ys Un after x xs* ∩ *before x ys*
by(*auto*)(*metis before_in_setD1 not_before_in*)

lemma *phi_diff_aux*:
  card (*Inv xs ys* ∪
    {(*y*, *x*) |*y*. *y* < *x in xs* ∧ *y* < *x in ys*} −
    {(*x*, *y*) |*y*. *x* < *y in xs* ∧ *y* < *x in ys*}) =
  card(*Inv xs ys*) + *card*(*before x xs* ∩ *before x ys*)
  − *int*(*card*(*after x xs* ∩ *before x ys*))
  (**is** *card*(*?I* ∪ *?B* − *?A*) = *card ?I* + *card ?b* − *int*(*card ?a*))
proof−
  have *1*: *?I* ∩ *?B* = {} **by**(*auto simp*: *Inv_def*) (*metis no_before_inI*)
  have *2*: *?A* ⊆ *?I* ∪ *?B* **by**(*auto simp*: *Inv_def*)
  have *3*: *?A* ⊆ *?I* **by**(*auto simp*: *Inv_def*)
  have *int*(*card*(*?I* ∪ *?B* − *?A*)) = *int*(*card ?I* + *card ?B*) − *int*(*card ?A*)
    using *card_mono*[*OF _ 3*]
    by(*simp add*: *card_Un_disjoint*[*OF _ _ 1*] *card_Diff_subset*[*OF _ 2*])
  also have *card ?B* = *card* (*fst ‘ ?B*) **by**(*auto simp*: *card_image inj_on_def*)
  also have *fst ‘ ?B* = *?b* **by** *force*
  also have *card ?A* = *card* (*snd ‘ ?A*) **by**(*auto simp*: *card_image inj_on_def*)
  also have *snd ‘ ?A* = *?a* **by** *force*
  finally show *?thesis* **.**
qed

lemma *not_before_Cons*[*simp*]: ¬ *x* < *y in y* # *xs*
by (*simp add*: *before_in_def*)

lemma *before_Cons*[*simp*]:
  y ∈ *set xs* ⟹ *y* ≠ *x* ⟹ *before y* (*x*#*xs*) = *insert x* (*before y xs*)
by(*auto simp*: *before_in_def*)

lemma *card_before_le_index*: *card* (*before x xs*) ≤ *index xs x*
apply(*cases x* ∈ *set xs*)
 prefer *2* **apply** (*simp add*: *before_in_def*)
apply(*induction xs*)
 apply (*simp add*: *before_in_def*)
apply (*auto simp*: *card_insert_if*)
done

lemma *config_config_length*: *length* (*fst* (*config A init qs*)) = *length init*
apply (*induct rule*: *config_induct*) **by** (*simp_all*)

lemma *config_config_distinct*:

shows *distinct* (*fst* (*config A init qs*)) = *distinct init*
apply (*induct rule*: *config_induct*) **by** (*simp_all add*: *distinct_step*)

lemma *config_config_set*:
shows *set* (*fst* (*config A init qs*)) = *set init*
apply(*induct rule*: *config_induct*) **by**(*simp_all add*: *set_step*)

lemma *config_config*:
set (*fst* (*config A init qs*)) = *set init*
∧ *distinct* (*fst* (*config A init qs*)) = *distinct init*
∧ *length* (*fst* (*config A init qs*)) = *length init*
using *config_config_distinct config_config_set config_config_length* **by** *metis*

lemma *config_dist_perm*:
distinct init ⟹ *dist_perm* (*fst* (*config A init qs*)) *init*
using *config_config_distinct config_config_set* **by** *metis*

lemma *config_rand_length*: ∀ *x*∈*set_pmf* (*config_rand A init qs*). *length* (*fst x*) = *length init*
apply (*induct rule*: *config_rand_induct*) **by** (*simp_all*)

lemma *config_rand_distinct*:
shows ∀ *x* ∈ (*config_rand A init qs*). *distinct* (*fst x*) = *distinct init*
apply (*induct rule*: *config_rand_induct*) **by** (*simp_all add*: *distinct_step*)

lemma *config_rand_set*:
shows ∀ *x* ∈ (*config_rand A init qs*). *set* (*fst x*) = *set init*
apply(*induct rule*: *config_rand_induct*) **by**(*simp_all add*: *set_step*)

lemma *config_rand*:
∀ *x* ∈ (*config_rand A init qs*). *set* (*fst x*) = *set init*
∧ *distinct* (*fst x*) = *distinct init* ∧ *length* (*fst x*) = *length init*
using *config_rand_distinct config_rand_set config_rand_length* **by** *metis*

lemma *config_rand_dist_perm*:
distinct init ⟹ ∀ *x* ∈ (*config_rand A init qs*). *dist_perm* (*fst x*) *init*
using *config_rand_distinct config_rand_set* **by** *metis*

lemma *amor_mtf_ub*: **assumes** *x* : *set ys set xs* = *set ys*
shows *int*(*card*(*before x xs Int before x ys*)) − *card*(*after x xs Int before x ys*)
≤ *2* ∗ *int*(*index xs x*) − *card* (*before x ys*) (**is** *?m* − *?n* ≤ *2* ∗ *?j* − *?k*)
proof−
  have *xxs*: *x* ∈ *set xs* **using** *assms*(*1*,*2*) **by** *simp*
  let *?bxxs* = *before x xs* **let** *?bxys* = *before x ys* **let** *?axxs* = *after x xs*
  have *0*: *?bxxs* ∩ *?axxs* = {} **by** (*auto simp*: *before_in_def*)
  hence *1*: (*?bxxs* ∩ *?bxys*) ∩ (*?axxs* ∩ *?bxys*) = {} **by** *blast*
  have (*?bxxs* ∩ *?bxys*) ∪ (*?axxs* ∩ *?bxys*) = *?bxys*
    using *assms*(*2*) *before_Un xxs* **by** *fastforce*
  hence *?m* + *?n* = *?k*
    using *card_Un_disjoint*[*OF _ _ 1*] **by** *simp*
  hence *?m* − *?n* = *2* ∗ *?m* − *?k* **by** *arith*
  also have *?m* ≤ *?j*
   using *card_before_le_index*[*of x xs*] *card_mono*[*of ?bxxs*, *OF _ Int_lower1*]
    by(*auto intro*: *order_trans*)
  finally show *?thesis* **by** *auto*
qed

locale *MTF_Off* =
fixes *as* :: *answer list*
fixes *rs* :: *'a list*
fixes *s0* :: *'a list*
assumes *dist_s0*[*simp*]: *distinct s0*
assumes *len_as*: *length as* = *length rs*
begin

definition *mtf_A* :: *nat list* **where**
mtf_A = *map fst as*

definition *sw_A* :: *nat list list* **where**
sw_A = *map snd as*

fun *s_A* :: *nat* ⇒ *'a list* **where**
s_A 0 = *s0* |
s_A(*Suc n*) = *step* (*s_A n*) (*rs*!*n*) (*mtf_A*!*n*, *sw_A*!*n*)

lemma *length_s_A*[*simp*]: *length*(*s_A n*) = *length s0*
by (*induction n*) *simp_all*

lemma *dist_s_A*[*simp*]: *distinct*(*s_A n*)
by(*induction n*) (*simp_all add*: *step_def*)

lemma *set_s_A*[*simp*]: *set*(*s_A n*) = *set s0*
by(*induction n*) (*simp_all add*: *step_def*)

fun *s_mtf* :: *nat* ⇒ *'a list* **where**
s_mtf 0 = *s0* |
s_mtf (*Suc n*) = *mtf* (*rs*!*n*) (*s_mtf n*)

definition *t_mtf* :: *nat* ⇒ *int* **where**
t_mtf n = *index* (*s_mtf n*) (*rs*!*n*) + *1*

definition *T_mtf* :: *nat* ⇒ *int* **where**
T_mtf n = (∑ *i*<*n*. *t_mtf i*)

definition *c_A* :: *nat* ⇒ *int* **where**
c_A n = *index* (*swaps* (*sw_A*!*n*) (*s_A n*)) (*rs*!*n*) + *1*

definition *f_A* :: *nat* ⇒ *int* **where**
f_A n = *min* (*mtf_A*!*n*) (*index* (*swaps* (*sw_A*!*n*) (*s_A n*)) (*rs*!*n*))

definition *p_A* :: *nat* ⇒ *int* **where**
p_A n = *size*(*sw_A*!*n*)

definition *t_A* :: *nat* ⇒ *int* **where**
t_A n = *c_A n* + *p_A n*

definition *T_A* :: *nat* ⇒ *int* **where**
T_A n = (∑ *i*<*n*. *t_A i*)

lemma *length_s_mtf*[*simp*]: *length*(*s_mtf n*) = *length s0*
by (*induction n*) *simp_all*

lemma *dist_s_mtf*[*simp*]: *distinct*(*s_mtf n*)
apply(*induction n*)
 apply (*simp*)
apply (*auto simp*: *mtf_def index_take set_drop_if_index*)
apply (*metis set_drop_if_index index_take less_Suc_eq_le linear*)
done

lemma *set_s_mtf*[*simp*]: *set* (*s_mtf n*) = *set s0*
by (*induction n*) (*simp_all*)

lemma *dperm_inv*: *dist_perm* (*s_A n*) (*s_mtf n*)

by (*metis dist_s_mtf dist_s_A set_s_mtf set_s_A*)

definition *Phi* :: *nat* ⇒ *int* (‹Φ›) **where**
Phi n = *card*(*Inv* (*s_A n*) (*s_mtf n*))

lemma *phi0*: *Phi 0* = *0*
by(*simp add*: *Phi_def*)

lemma *phi_pos*: *Phi n* ≥ *0*
by(*simp add*: *Phi_def*)

lemma *mtf_ub*: *t_mtf n* + *Phi* (*n*+*1*) − *Phi n* ≤ *2* ∗ *c_A n* − *1* + *p_A n* − *f_A n*
proof −
  let *?xs* = *s_A n* **let** *?ys* = *s_mtf n* **let** *?x* = *rs*!*n*
  let *?xs′* = *swaps* (*sw_A*!*n*) *?xs* **let** *?ys′* = *mtf ?x ?ys*
  show *?thesis*
  proof *cases*
  assume *xin*: *?x* ∈ *set ?ys*
  let *?bb* = *before ?x ?xs* ∩ *before ?x ?ys*
  let *?ab* = *after ?x ?xs* ∩ *before ?x ?ys*
  have *phi_mtf*:
    card(*Inv ?xs′ ?ys′*) − *int*(*card* (*Inv ?xs′ ?ys*))
   ≤ *2* ∗ *int*(*index ?xs′ ?x*) − *int*(*card* (*before ?x ?ys*))
      using *xin* **by**(*simp add*: *Inv_mtf phi_diff_aux amor_mtf_ub*)
  have *phi_sw*: *card*(*Inv ?xs′ ?ys*) ≤ *Phi n* + *length*(*sw_A*!*n*)
  proof −
    have *int*(*card* (*Inv ?xs′ ?ys*)) ≤ *card*(*Inv ?xs′ ?xs*) + *int*(*card*(*Inv ?xs ?ys*))
      using *card_Inv_tri_ineq*[*of ?xs′ ?xs ?ys*] *xin* **by** (*simp*)
    also have *card*(*Inv ?xs′ ?xs*) = *card*(*Inv ?xs ?xs′*)
      by (*rule card_Inv_sym*)
    also have *card*(*Inv ?xs ?xs′*) ≤ *size*(*sw_A*!*n*)
      by (*metis card_Inv_swaps_le dist_s_A*)
    finally show *?thesis* **by**(*fastforce simp*: *Phi_def*)
  qed
  have *phi_free*: *card*(*Inv ?xs′ ?ys′*) − *Phi* (*Suc n*) = *f_A n* **using** *xin*
   by(*simp add*: *Phi_def mtf2_def step_def card_Inv_mtf2 index_less_size_conv f_A_def*)
  show *?thesis* **using** *xin phi_sw phi_mtf phi_free card_before*[*of s_mtf n*]
    by(*simp add*: *t_mtf_def c_A_def p_A_def*)
  next
    assume *notin*: *?x* ∉ *set ?ys*
    have *int* (*card* (*Inv ?xs′ ?ys*)) − *card* (*Inv ?xs ?ys*) ≤ *card*(*Inv ?xs ?xs′*)

```
      using card_Inv_tri_ineq[OF _ dperm_inv, of ?xs' n]
            swaps_inv[of sw_A!n s_A n]
      by(simp add: card_Inv_sym)
    also have ... ≤ size(sw_A!n)
      by(simp add: card_Inv_swaps_le dperm_inv)
    finally show ?thesis using notin
     by(simp add: t_mtf_def step_def c_A_def p_A_def f_A_def Phi_def
mtf2_def)
  qed
qed

theorem Sleator_Tarjan: T_mtf n ≤ (∑ i<n. 2*c_A i + p_A i − f_A
i) − n
proof−
  have (∑ i<n. t_mtf i) ≤ (∑ i<n. 2*c_A i − 1 + p_A i − f_A i)
    by(rule potential[where p=Phi,OF phi0 phi_pos mtf_ub])
  also have ... = (∑ i<n. (2*c_A i + p_A i − f_A i) − 1)
    by (simp add: algebra_simps)
  also have ... = (∑ i<n. 2*c_A i + p_A i − f_A i) − n
    by(simp add: sumr_diff_mult_const2[symmetric])
  finally show ?thesis by(simp add: T_mtf_def)
qed

corollary Sleator_Tarjan': T_mtf n ≤ 2*T_A n − n
proof −
   have T_mtf n ≤ (∑ i<n. 2*c_A i + p_A i − f_A i) − n by (fact
Sleator_Tarjan)
  also have (∑ i<n. 2*c_A i + p_A i − f_A i) ≤ (∑ i<n. 2*(c_A i +
p_A i))
    by(intro sum_mono) (simp add: p_A_def f_A_def)
   also have ... ≤ 2* T_A n by (simp add: sum_distrib_left T_A_def
t_A_def)
  finally show T_mtf n ≤ 2* T_A n − n by auto
qed

lemma T_A_nneg: 0 ≤ T_A n
by(auto simp add: sum_nonneg T_A_def t_A_def c_A_def p_A_def)

lemma T_mtf_ub: ∀ i<n. rs!i ∈ set s0 ⟹ T_mtf n ≤ n * size s0
proof(induction n)
  case 0 show ?case by(simp add: T_mtf_def)
next
  case (Suc n) thus ?case
    using index_less_size_conv[of s_mtf n rs!n]
```

by(*simp add*: *T_mtf_def t_mtf_def less_Suc_eq del*: *index_less*)
qed

corollary *T_mtf_competitive*: **assumes** $s0 \neq []$ **and** $\forall\, i<n.\ rs!i \in set\ s0$
shows $T\_mtf\ n \leq (2 - 1/(size\ s0)) * T\_A\ n$
proof *cases*
assume *0*: $real\_of\_int(T\_A\ n) \leq n * (size\ s0)$
have $T\_mtf\ n \leq 2 * T\_A\ n - n$
proof $-$
have $T\_mtf\ n \leq (\sum i<n.\ 2{*}c\_A\ i + p\_A\ i - f\_A\ i) - n$ **by**(*rule Sleator_Tarjan*)
also have $(\sum i<n.\ 2{*}c\_A\ i + p\_A\ i - f\_A\ i) \leq (\sum i<n.\ 2{*}(c\_A\ i + p\_A\ i))$
by(*intro sum_mono*) (*simp add*: *p_A_def f_A_def*)
also have $\ldots \leq 2 * T\_A\ n$ **by** (*simp add*: *sum_distrib_left T_A_def t_A_def*)
finally show *?thesis* **by** *simp*
qed
hence $real\_of\_int(T\_mtf\ n) \leq 2 * of\_int(T\_A\ n) - n$ **by** *simp*
also have $\ldots = 2 * of\_int(T\_A\ n) - (n * size\ s0) / size\ s0$
using *assms*(*1*) **by** *simp*
also have $\ldots \leq 2 * real\_of\_int(T\_A\ n) - T\_A\ n / size\ s0$
by(*rule diff_left_mono*[*OF divide_right_mono*[*OF 0*]]) *simp*
also have $\ldots = (2 - 1 / size\ s0) * T\_A\ n$ **by** *algebra*
finally show *?thesis* **.**
next
assume *0*: $\neg\ real\_of\_int(T\_A\ n) \leq n * (size\ s0)$
have $2 - 1 / size\ s0 \geq 1$ **using** *assms*(*1*)
by (*auto simp add*: *field_simps neq_Nil_conv*)
have $real\_of\_int\ (T\_mtf\ n) \leq n * size\ s0$ **using** *T_mtf_ub*[*OF assms*(*2*)] **by** *linarith*
also have $\ldots < of\_int(T\_A\ n)$ **using** *0* **by** *simp*
also have $\ldots \leq (2 - 1 / size\ s0) * T\_A\ n$ **using** *assms*(*1*) *T_A_nneg*[*of n*]
by(*auto simp add*: *mult_le_cancel_right1 field_simps neq_Nil_conv*)
finally show *?thesis* **by** *linarith*
qed

lemma *t_A_t*: $n < length\ rs \Longrightarrow t\_A\ n = int\ (t\ (s\_A\ n)\ (rs\ !\ n)\ (as\ !\ n))$
by(*simp add*: *t_A_def t_def c_A_def p_A_def sw_A_def len_as split*: *prod.split*)

lemma *T_A_eq_lem*: $(\sum i=0..<length\ rs.\ t\_A\ i) =$

T (*s_A 0*) (*drop 0 rs*) (*drop 0 as*)
proof(*induction rule*: *zero_induct*[*of _ size rs*])
case *1* **thus** *?case* **by** (*simp add*: *len_as*)
next
case (*2 n*)
show *?case*
proof *cases*
assume *n* < *length rs*
thus *?case* **using** *2*
by(*simp add*: *Cons_nth_drop_Suc*[*symmetric*,**where** *i*=*n*] *len_as sum.atLeast_Suc_lessThan* *t_A_t mtf_A_def sw_A_def*)
next
assume ¬ *n* < *length rs* **thus** *?case* **by** (*simp add*: *len_as*)
qed
qed

lemma *T_A_eq*: *T_A* (*length rs*) = *T s0 rs as*
using *T_A_eq_lem* **by**(*simp add*: *T_A_def atLeast0LessThan*)

lemma *nth_off_MTF*: *n* < *length rs* ⟹ *off2 MTF s rs* ! *n* = (*size*(*fst s*) − *1*,[])
by(*induction rs arbitrary*: *s n*)(*auto simp add*: *MTF_def nth_Cons' Step_def*)

lemma *t_mtf_MTF*: *n* < *length rs* ⟹
t_mtf n = *int* (*t* (*s_mtf n*) (*rs* ! *n*) (*off MTF s rs* ! *n*))
by(*simp add*: *t_mtf_def t_def nth_off_MTF split*: *prod.split*)

lemma *mtf_MTF*: *n* < *length rs* ⟹ *length s* = *length s0* ⟹ *mtf* (*rs* ! *n*) *s* =
step s (*rs* ! *n*) (*off MTF s0 rs* ! *n*)
by(*auto simp add*: *nth_off_MTF step_def mtf_eq_mtf2*)

lemma *T_mtf_eq_lem*: (∑ *i*=*0*..<*length rs*. *t_mtf i*) =
T (*s_mtf 0*) (*drop 0 rs*) (*drop 0* (*off MTF s0 rs*))
proof(*induction rule*: *zero_induct*[*of _ size rs*])
case *1* **thus** *?case* **by** (*simp add*: *len_as*)
next
case (*2 n*)
show *?case*
proof *cases*
assume *n* < *length rs*
thus *?case* **using** *2*
by(*simp add*: *Cons_nth_drop_Suc*[*symmetric*,**where** *i*=*n*] *len_as* *sum.atLeast_Suc_lessThan*

t_mtf_MTF[**where** *s=s0*] *mtf_A_def sw_A_def mtf_MTF*)
next
assume ¬ *n* < *length rs* **thus** *?case* **by** (*simp add*: *len_as*)
qed
qed

lemma *T_mtf_eq*: *T_mtf* (*length rs*) = *T_on MTF s0 rs*
using *T_mtf_eq_lem* **by**(*simp add*: *T_mtf_def atLeast0LessThan*)

corollary *MTF_competitive2*: *s0* ≠ [] ⟹ ∀ *i*<*length rs*. *rs*!*i* ∈ *set s0* ⟹
T_on MTF s0 rs ≤ (*2* − *1*/(*size s0*)) ∗ *T s0 rs as*
by (*metis T_mtf_competitive T_A_eq T_mtf_eq of_int_of_nat_eq*)

corollary *MTF_competitive'*: *T_on MTF s0 rs* ≤ *2* ∗ *T s0 rs as*
using *Sleator_Tarjan'*[*of length rs*] *T_A_eq T_mtf_eq*
by *auto*

end

theorem *compet_MTF*: **assumes** *s0* ≠ [] *distinct s0 set rs* ⊆ *set s0*
shows *T_on MTF s0 rs* ≤ (*2* − *1*/(*size s0*)) ∗ *T_opt s0 rs*
proof−
from *assms*(*3*) **have** *1*: ∀ *i* < *length rs*. *rs*!*i* ∈ *set s0* **by** *auto*
{ fix *as* :: *answer list* **assume** *len*: *length as* = *length rs*
interpret *MTF_Off as rs s0* **proof qed** (*auto simp*: *assms*(*2*) *len*)
from *MTF_competitive2*[*OF assms*(*1*) *1*] *assms*(*1*)
have *T_on MTF s0 rs* / (*2* − *1* / (*length s0*)) ≤ *of_int*(*T s0 rs as*)
by(*simp add*: *field_simps length_greater_0_conv*[*symmetric*]
del: *length_greater_0_conv*) **}**
hence *T_on MTF s0 rs* / (*2* − *1*/(*size s0*)) ≤ *T_opt s0 rs*
apply(*simp add*: *T_opt_def Inf_nat_def*)
apply(*rule LeastI2_wellorder*)
using *length_replicate*[*of length rs undefined*] **apply** *fastforce*
apply *auto*
done
thus *?thesis* **using** *assms* **by**(*simp add*: *field_simps*
length_greater_0_conv[*symmetric*] *del*: *length_greater_0_conv*)
qed

theorem *compet_MTF'*: **assumes** *distinct s0*
shows *T_on MTF s0 rs* ≤ (*2*::*real*) ∗ *T_opt s0 rs*
proof−
{ fix *as* :: *answer list* **assume** *len*: *length as* = *length rs*
interpret *MTF_Off as rs s0* **proof qed** (*auto simp*: *assms*(*1*) *len*)

```
    from MTF_competitive′
    have T_on MTF s0 rs / 2 ≤ of_int(T s0 rs as)
      by(simp add: field_simps length_greater_0_conv[symmetric]
        del: length_greater_0_conv) }
  hence T_on MTF s0 rs / 2 ≤ T_opt s0 rs
    apply(simp add: T_opt_def Inf_nat_def)
    apply(rule LeastI2_wellorder)
    using length_replicate[of length rs undefined] apply fastforce
    apply auto
    done
  thus ?thesis using assms by(simp add: field_simps
    length_greater_0_conv[symmetric] del: length_greater_0_conv)
qed

theorem MTF_is_2_competitive: compet MTF 2 {s . distinct s}
unfolding compet_def using compet_MTF′ by fastforce
```

6.6 Lower Bound for Competitiveness

This result is independent of MTF but is based on the list update problem defined in this theory.

```
lemma rat_fun_lem:
   fixes l c :: real
   assumes [simp]: F ≠ bot
   assumes 0 < l
   assumes ev:
     eventually (λn. l ≤ f n / g n) F
     eventually (λn. (f n + c) / (g n + d) ≤ u) F
   and
     g: LIM n F. g n :> at_top
   shows l ≤ u
proof (rule dense_le_bounded[OF ‹0 < l›])
   fix x assume x: 0 < x x < l

   define m where m = (x − l) / 2
   define k where k = l / (x − m)
   have x = l / k + m 1 < k m < 0
     unfolding k_def m_def using x by (auto simp: divide_simps)

   from ‹1 < k› have LIM n F. (k − 1) ∗ g n :> at_top
     by (intro filterlim_tendsto_pos_mult_at_top[OF tendsto_const _ g])
(simp add: field_simps)
   then have eventually (λn. d ≤ (k − 1) ∗ g n) F
     by (simp add: filterlim_at_top)
```

moreover have *eventually* (λ*n*. *1* ≤ *g n*) *F eventually* (λ*n*. *1* − *d* ≤ *g n*) *F eventually* (λ*n*. *c* / *m* − *d* ≤ *g n*) *F*
  using *g* **by** (*auto simp add*: *filterlim_at_top*)
ultimately have *eventually* (λ*n*. *x* ≤ *u*) *F*
  using *ev*
proof *eventually_elim*
  fix *n* **assume** *d*: *d* ≤ (*k* − *1*) ∗ *g n 1* ≤ *g n 1* − *d* ≤ *g n c* / *m* − *d* ≤ *g n*
    and *l*: *l* ≤ *f n* / *g n* **and** *u*: (*f n* + *c*) / (*g n* + *d*) ≤ *u*
  from *d* **have** *g n* + *d* ≤ *k* ∗ *g n*
    by (*simp add*: *field_simps*)
  from *d* **have** *g*: *0* < *g n 0* < *g n* + *d*
    by (*auto simp*: *field_simps*)
  with ‹*0* < *l*› *l* **have** *0* < *f n*
    by (*auto simp*: *field_simps intro*: *mult_pos_pos less_le_trans*)

  note ‹*x* = *l* / *k* + *m*›
  also have *l* / *k* ≤ *f n* / (*k* ∗ *g n*)
    using *l* ‹*1* < *k*› **by** (*simp add*: *field_simps*)
  also have . . . ≤ *f n* / (*g n* + *d*)
    using *d* ‹*1* < *k*› ‹*0* < *f n*› **by** (*intro divide_left_mono mult_pos_pos*) (*auto simp*: *field_simps*)
  also have *m* ≤ *c* / (*g n* + *d*)
    using ‹*c* / *m* − *d* ≤ *g n*› ‹*0* < *g n*› ‹*0* < *g n* + *d*› ‹*m* < *0*› **by** (*simp add*: *field_simps*)
  also have *f n* / (*g n* + *d*) + *c* / (*g n* + *d*) = (*f n* + *c*) / (*g n* + *d*)
    using ‹*0* < *g n* + *d*› **by** (*auto simp*: *add_divide_distrib*)
  also note *u*
  finally show *x* ≤ *u* **by** *simp*
qed
then show *x* ≤ *u* **by** *auto*
qed

lemma *compet_lb0*:
fixes *a Aon Aoff cruel*
defines *f s0 rs* == *real*(*T_on Aon s0 rs*)
defines *g s0 rs* == *real*(*T_off Aoff s0 rs*)
assumes ⋀*rs s0*. *size*(*Aoff s0 rs*) = *length rs* **and** ⋀*n*. *cruel n* ≠ []
assumes *compet Aon c S0* **and** *c*≥*0* **and** *s0* ∈ *S0*
and *l*: *eventually* (λ*n*. *f s0* (*cruel n*) / (*g s0* (*cruel n*) + *a*) ≥ *l*) *sequentially*
and *g*: *LIM n sequentially*. *g s0* (*cruel n*) :> *at_top*
and *l* > *0* **and** ⋀*n*. *static s0* (*cruel n*)
shows *l* ≤ *c*

proof−
  let *?h* = λ*b s0 rs*. (*f s0 rs* − *b*) / *g s0 rs*
  have *g'*: *LIM n sequentially. g s0* (*cruel n*) + *a* :> *at_top*
    using *filterlim_tendsto_add_at_top*[*OF tendsto_const g*]
    by (*simp add*: *ac_simps*)
   from *competE*[*OF assms*(*5*) ‹*c*≥*0*› _ ‹*s0* ∈ *S0*›] *assms*(*3*) **obtain** *b* **where**
    ∀ *rs*. *static s0 rs* ∧ *rs* ≠ [] ⟶ *?h b s0 rs* ≤ *c*
    by (*fastforce simp del*: *neq0_conv simp*: *neq0_conv*[*symmetric*]
        field_simps f_def g_def T_off_neq0[*of Aoff*, *OF assms*(*3*)])
  hence ∀ *n*. (*?h b s0 o cruel*) *n* ≤ *c* **using** *assms*(*4*,*11*) **by** *simp*
  with *rat_fun_lem*[*OF sequentially_bot* ‹*l*>*0*› _ _ *g'*, *of f s0 o cruel* −*b* − *a c*] *assms*(*7*) *l*
  show *l* ≤ *c* **by** (*auto*)
qed

Sorting

fun *ins_sws* **where**
ins_sws k x [] = [] |
ins_sws k x (*y#ys*) = (*if k x* ≤ *k y then* [] *else map Suc* (*ins_sws k x ys*) @ [*0*])

fun *sort_sws* **where**
sort_sws k [] = [] |
sort_sws k (*x#xs*) =
  ins_sws k x (*sort_key k xs*) @ *map Suc* (*sort_sws k xs*)

lemma *length_ins_sws*: *length*(*ins_sws k x xs*) ≤ *length xs*
by(*induction xs*) *auto*

lemma *length_sort_sws_le*: *length*(*sort_sws k xs*) ≤ *length xs* ^ *2*
proof(*induction xs*)
  case (*Cons x xs*) **thus** *?case*
   using *length_ins_sws*[*of k x sort_key k xs*] **by** (*simp add*: *numeral_eq_Suc*)
qed *simp*

lemma *swaps_ins_sws*:
  swaps (*ins_sws k x xs*) (*x#xs*) = *insort_key k x xs*
by(*induction xs*)(*auto simp*: *swap_def*[*of 0*])

lemma *swaps_sort_sws*[*simp*]:
  swaps (*sort_sws k xs*) *xs* = *sort_key k xs*
by(*induction xs*)(*auto simp*: *swaps_ins_sws*)

The cruel adversary:

```
fun cruel :: ('a,'is) alg_on ⇒ 'a state * 'is ⇒ nat ⇒ 'a list where
cruel A s 0 = [] |
cruel A s (Suc n) = last (fst s) # cruel A (Step A s (last (fst s))) n

definition adv :: ('a,'is) alg_on ⇒ ('a::linorder) alg_off where
adv A s rs = (if rs=[] then [] else
  let crs = cruel A (Step A (s, fst A s) (last s)) (size rs − 1)
  in (0,sort_sws (λx. size rs − 1 − count_list crs x) s) # replicate (size
rs − 1) (0,[]))

lemma set_cruel: s ≠ [] ⟹ set(cruel A (s,is) n) ⊆ set s
apply(induction n arbitrary: s is)
apply(auto simp: step_def Step_def split: prod.split)
by (metis empty_iff swaps_inv last_in_set list.set(1) rev_subsetD set_mtf2)

lemma static_cruel: s ≠ [] ⟹ static s (cruel A (s,is) n)
by(simp add: set_cruel static_def)


lemma T_cruel:
  s ≠ [] ⟹ distinct s ⟹
  T s (cruel A (s,is) n) (off2 A (s,is) (cruel A (s,is) n)) ≥ n*(length s)
apply(induction n arbitrary: s is)
 apply(simp)
apply(erule_tac x = fst(Step A (s, is) (last s)) in meta_allE)
apply(erule_tac x = snd(Step A (s, is) (last s)) in meta_allE)
apply(frule_tac sws = snd(fst(snd A (s,is) (last s))) in index_swaps_last_size)
apply(simp add: distinct_step t_def split_def Step_def
        length_greater_0_conv[symmetric] del: length_greater_0_conv)
done

lemma length_cruel[simp]: length (cruel A s n) = n
by (induction n arbitrary: s) (auto)

lemma t_sort_sws: t s r (mf, sort_sws k s) ≤ size s ^ 2 + size s + 1
using length_sort_sws_le[of k s] index_le_size[of sort_key k s r]
by (simp add: t_def add_mono index_le_size algebra_simps)

lemma T_noop:
  n = length rs ⟹ T s rs (replicate n (0, [])) = (∑r←rs. index s r + 1)
by(induction rs arbitrary: s n)(auto simp: t_def step_def)


lemma sorted_asc: j≤i ⟹ i<size ss ⟹ ∀x ∈ set ss. ∀y ∈ set ss. k(x)
```

$\leq k(y) \longrightarrow f\ y \leq f\ x$
  $\Longrightarrow$ *sorted* (*map k ss*) $\Longrightarrow$ *f* (*ss* ! *i*) $\leq$ *f* (*ss* ! *j*)
by (*auto simp*: *sorted_iff_nth_mono*)

lemma *sorted_weighted_gauss_Ico_div2*:
  fixes *f* :: *nat* $\Rightarrow$ *nat*
  assumes $\bigwedge i\ j.\ i \leq j \Longrightarrow j < n \Longrightarrow f\ i \geq f\ j$
  shows ($\sum$ *i=0..<n.* (*i* + *1*) $*$ *f i*) $\leq$ (*n* + *1*) $*$ *sum f* {*0..<n*} *div 2*
proof (*cases n*)
  case *0*
  then show *?thesis*
    by *simp*
next
  case (*Suc n*)
  with *assms* **have** *Suc n* $*$ ($\sum$ *i=0..<Suc n. Suc i* $*$ *f i*) $\leq$ ($\sum$ *i=0..<Suc n. Suc i*) $*$ *sum f* {*0..<Suc n*}
    by (*intro Chebyshev_sum_upper_nat* [*of Suc n Suc f*]) *auto*
  then have *Suc n* $*$ (*2* $*$ ($\sum$ *i=0..n. Suc i* $*$ *f i*)) $\leq$ *2* $*$ ($\sum$ *i=0..n. Suc i*) $*$ *sum f* {*0..n*}
    by (*simp add*: *atLeastLessThanSuc_atLeastAtMost*)
  also have *2* $*$ ($\sum$ *i=0..n. Suc i*) = *Suc n* $*$ (*n* + *2*)
    using *arith_series_nat* [*of 1 1 n*] **by** *simp*
  finally have *2* $*$ ($\sum$ *i=0..n. Suc i* $*$ *f i*) $\leq$ (*n* + *2*) $*$ *sum f* {*0..n*}
    by (*simp only*: *ac_simps Suc_mult_le_cancel1*)
  with *Suc* **show** *?thesis*
    by (*simp only*: *atLeastLessThanSuc_atLeastAtMost*) *simp*
qed

lemma *T_adv*: **assumes** $l \neq 0$
shows *T_off* (*adv A*) [*0..<l*] (*cruel A* ([*0..<l*],*fst A* [*0..<l*]) (*Suc n*))
  $\leq l^2 + l + 1 + (l + 1) * n$ *div 2* (**is** *?l* $\leq$ *?r*)
proof−
  let *?s* = [*0..<l*]
  let *?r* = *last ?s*
  let *?S′* = *Step A* (*?s*,*fst A ?s*) *?r*
  let *?s′* = *fst ?S′*
  let *?cr* = *cruel A ?S′ n*
  let *?c* = *count_list ?cr*
  let *?k* = $\lambda x.$ *n* − *?c x*
  let *?sort* = *sort_key ?k ?s*
  have *1*: *set ?s′* = {*0..<l*}
    by(*simp add*: *set_step Step_def split*: *prod.split*)
  have *3*: $\bigwedge x.\ x < l \Longrightarrow$ *?c x* $\leq$ *n*

```
    by(simp) (metis count_le_length length_cruel)
  have ?l = t ?s (last ?s) (0, sort_sws ?k ?s) + (∑x∈set ?s'. ?c x * (index
?sort x + 1))
    using assms
    apply(simp add: adv_def T_noop sum_list_map_eq_sum_count2[OF
set_cruel] Step_def
      split: prod.split)
    apply(subst (3) step_def)
    apply(simp)
    done
  also have (∑x∈set ?s'. ?c x * (index ?sort x + 1)) = (∑x∈{0..<l}. ?c
x * (index ?sort x + 1))
    by (simp add: 1)
  also have ... = (∑x∈{0..<l}. ?c (?sort ! x) * (index ?sort (?sort ! x)
+ 1))
    by(rule sum.reindex_bij_betw[where ?h = nth ?sort, symmetric])
      (simp add: bij_betw_imageI inj_on_nth nth_image)
  also have ... = (∑x∈{0..<l}. ?c (?sort ! x) * (x+1))
    by(simp add: index_nth_id)
  also have ... ≤ (∑x∈{0..<l}. (x+1) * ?c (?sort ! x))
    by (simp add: algebra_simps)
  also(ord_eq_le_subst) have ... ≤ (l+1) * (∑x∈{0..<l}. ?c (?sort ! x))
div 2
    apply(rule sorted_weighted_gauss_Ico_div2)
    apply(erule sorted_asc[where k = λx. n − count_list (cruel A ?S' n)
x])
    apply(auto simp add: index_nth_id dest!: 3)
    using assms [[linarith_split_limit = 20]] by simp
  also have (∑x∈{0..<l}. ?c (?sort ! x)) = (∑x∈{0..<l}. ?c (?sort !
(index ?sort x)))
    by(rule sum.reindex_bij_betw[where ?h = index ?sort, symmetric])
      (simp add: bij_betw_imageI inj_on_index2 index_image)
  also have ... = (∑x∈{0..<l}. ?c x) by(simp)
  also have ... = length ?cr
    using set_cruel[of ?s' A _ n] assms 1
    by(auto simp add: sum_count_set Step_def split: prod.split)
  also have ... = n by simp
  also have t ?s (last ?s) (0, sort_sws ?k ?s) ≤ (length ?s)^2 + length ?s
+ 1
    by(rule t_sort_sws)
  also have ... = l^2 + l + 1 by simp
  finally show ?l ≤ l² + l + 1 + (l + 1) * n div 2 by auto
qed
```

The main theorem:

```
theorem compet_lb2:
assumes compet A c {xs::nat list. size xs = l} and l ≠ 0 and c ≥ 0
shows c ≥ 2*l/(l+1)
proof (rule compet_lb0[OF _ _ assms(1) ‹c≥0›])
  let ?S0 = {xs::nat list. size xs = l}
  let ?s0 = [0..<l]
  let ?cruel = cruel A (?s0,fst A ?s0) o Suc
  let ?on = λn. T_on A ?s0 (?cruel n)
  let ?off = λn. T_off (adv A) ?s0 (?cruel n)
  show ⋀s0 rs. length (adv A s0 rs) = length rs by(simp add: adv_def)
  show ⋀n. ?cruel n ≠ [] by auto
  show ?s0 ∈ ?S0 by simp
  { fix Z::real and n::nat assume n ≥ nat(ceiling Z)
   have ?off n ≥ length(?cruel n) by(rule T_ge_len) (simp add: adv_def)
    hence ?off n > n by simp
    hence Z ≤ ?off n using ‹n ≥ nat(ceiling Z)› by linarith }
  thus LIM n sequentially. real (?off n) :> at_top
    by(auto simp only: filterlim_at_top eventually_sequentially)
  let ?a = − (l^2 + l + 1)
  { fix n assume n ≥ l^2 + l + 1
    have 2*l/(l+1) = 2*l*(n+1) / ((l+1)*(n+1))
      by (simp del: One_nat_def)
    also have ... = 2*real(l*(n+1)) / ((l+1)*(n+1)) by simp
    also have l * (n+1) ≤ ?on n
      using T_cruel[of ?s0 Suc n] ‹l ≠ 0›
      by (simp add: ac_simps)
    also have 2*real(?on n) / ((l+1)*(n+1)) ≤ 2*real(?on n)/(2*(?off n
+ ?a))
    proof −
      have 0: 2*real(?on n) ≥ 0 by simp
      have 1: 0 < real ((l + 1) * (n + 1)) by (simp del: of_nat_Suc)
      have ?off n ≥ length(?cruel n)
        by(rule T_ge_len) (simp add: adv_def)
      hence ?off n > n by simp
      hence ?off n + ?a > 0 using ‹n ≥ l^2 + l + 1› by linarith
      hence 2: real_of_int(2*(?off n + ?a)) > 0
      by(simp only: of_int_0_less_iff zero_less_mult_iff zero_less_numeral
simp_thms)
      have ?off n + ?a ≤ (l+1)*(n) div 2
        using T_adv[OF ‹l≠0›, of A n]
        by (simp only: o_apply of_nat_add of_nat_le_iff)
      also have ... ≤ (l+1)*(n+1) div 2 by (simp)
```

finally have *2∗(?off n + ?a) ≤ (l+1)∗(n+1)*
by (*simp add*: *zdiv_int*)
hence *of_int(2∗(?off n + ?a)) ≤ real((l+1)∗(n+1))* **by** (*simp only*: *of_int_le_iff*)
from *divide_left_mono*[*OF this 0 mult_pos_pos*[*OF 1 2*]] **show** *?thesis* .
qed
also have . . . = *?on n* / (*?off n* + *?a*)
by (*simp del*: *distrib_left_numeral One_nat_def cruel.simps*)
finally have *2∗l/(l+1) ≤ ?on n* / (*real* (*?off n*) + *?a*)
by (*auto simp*: *divide_right_mono*)
}
thus *eventually* (λ*n*. (*2* ∗ *l*) / (*l* + *1*) ≤ *?on n* / (*real*(*?off n*) + *?a*)) *sequentially*
by(*auto simp add*: *filterlim_at_top eventually_sequentially*)
show *0* < *2∗l* / (*l+1*) **using** ‹*l* ≠ *0*› **by**(*simp*)
show ⋀*n*. *static ?s0* (*?cruel n*) **using** ‹*l* ≠ *0*› **by**(*simp add*: *static_cruel del*: *cruel.simps*)
qed

end

theory *Bit_Strings*
imports *Complex_Main*
begin

7 Lemmas about BitStrings and sets theirof

7.1 the set of bitstring of length m is finite

lemma *bitstrings_finite*: *finite* {*xs*::*bool list*. *length xs* = *m*}
using *finite_lists_length_eq*[**where** *A*=*UNIV*] **by** *force*

7.2 how to calculate the cardinality of the set of bitstrings with certain bits already set

lemma *fbool*: *finite* {*xs*. (∀ *i*∈*X*. *xs* ! *i*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *m* ∧ *f* (*xs*!*e*)}
by(*rule finite_subset*[**where** *B*={*xs*. *length xs* = *m*}])
(*auto simp*: *bitstrings_finite*)

fun *witness* :: *nat set* ⇒ *nat* ⇒ *bool list* **where**
witness X 0 = []

|*witness X* (*Suc n*) = (*witness X n*) @ [*n* ∈ *X*]

lemma *witness_length*: *length* (*witness X n*) = *n*
apply(*induct n*) **by** *auto*

lemma *iswitness*: *r*<*n* ⟹ ((*witness X n*)!*r*) = (*r*∈*X*)
proof (*induct n*)
  case (*Suc n*)

  have *witness X* (*Suc n*) ! *r* = ((*witness X n*) @ [*n* ∈ *X*]) ! *r* **by** *simp*
  also have ... = (*if r* < *length* (*witness X n*) *then* (*witness X n*) ! *r else* [*n* ∈ *X*] ! (*r* − *length* (*witness X n*))) **by**(*rule nth_append*)
  also have ... = (*if r* < *n then* (*witness X n*) ! *r else* [*n* ∈ *X*] ! (*r* − *n*)) **by** (*simp add*: *witness_length*)
  finally have *1*: *witness X* (*Suc n*) ! *r* = (*if r* < *n then* (*witness X n*) ! *r else* [*n* ∈ *X*] ! (*r* − *n*)) **.**

  show *?case*
  proof (*cases r* < *n*)
    case *True*
    with *1* **have** *a*: *witness X* (*Suc n*) ! *r* = (*witness X n*) ! *r* **by** *auto*
    from *Suc True* **have** *b*: *witness X n* ! *r* = (*r* ∈ *X*) **by** *auto*
    from *a b* **show** *?thesis* **by** *auto*
  next
    case *False*
    with *Suc* **have** *r* = *n* **by** *auto*
    with *1* **show** *witness X* (*Suc n*) ! *r* = (*r* ∈ *X*) **by** *auto*
  qed
qed *simp*

lemma *card1*: *finite S* ⟹ *finite X* ⟹ *finite Y* ⟹ *X* ∩ *Y* = {} ⟹ *S* ∩ (*X* ∪ *Y*) = {} ⟹ *S*∪*X*∪*Y*={*0*..<*m*} ⟹
    card {*xs*. (∀ *i*∈*X*. *xs* ! *i*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *m*} = *2*^(*m* − *card X* − *card Y*)
proof(*induct arbitrary*: *X Y rule*: *finite_induct*)
  case *empty*
  then have *x*: *X* ⊆ {*0*..<*m*} **and** *y*: *Y* ⊆ {*0*..<*m*} **and** *xy*: *X*∪ *Y* = {*0*..<*m*} **by** *auto*
  then have *card* (*X* ∪ *Y*) = *m* **by** *auto*
  with *empty*(*3*) **have** *cardXY*: *card X* + *card Y* = *m* **using** *card_Un_Int*[*OF empty*(*1*) *empty*(*2*)] **by** *auto*

  from *empty* **have** *ents*: ∀ *i*<*m*. (*i*∈*Y*) = (*i*∉*X*) **by** *auto*

have (∃! *w*. (∀ *i*∈*X*. *w* ! *i*) ∧ (∀ *i*∈*Y*. ¬ *w* ! *i*) ∧ *length w* = *m*)
proof (*rule ex1I*, *goal_cases*)
case *1*
show (∀ *i*∈*X*. (*witness X m*) ! *i*) ∧ (∀ *i*∈*Y*. ¬ (*witness X m*) ! *i*) ∧ *length* (*witness X m*) = *m*
proof (*safe*, *goal_cases*)
case (*2 i*)
with *y* **have** *a*: *i* < *m* **by** *auto*
with *iswitness* **have** *witness X m* ! *i* = (*i* ∈ *X*) **by** *auto*
with *a ents 2* **have** ~ *witness X m* ! *i* **by** *auto*
with *2*(*2*) **show** *False* **by** *auto*
next
case (*1 i*)
with *x* **have** *a*: *i* < *m* **by** *auto*
with *iswitness* **have** *witness X m* ! *i* = (*i* ∈ *X*) **by** *auto*
with *a ents 1* **show** *witness X m* ! *i* **by** *auto*
qed (*rule witness_length*)
next
case (*2 w*)
show *w* = *witness X m*
proof −
have (*length w* = *length* (*witness X m*) ∧ (∀ *i*<*length w*. *w* ! *i* = (*witness X m*) ! *i*))
using *2* **apply**(*simp add*: *witness_length*)
proof
fix *i*
assume *as*: (∀ *i*∈*X*. *w* ! *i*) ∧ (∀ *i*∈*Y*. ¬ *w* ! *i*) ∧ *length w* = *m*
have *i* < *m* ⟶ (*witness X m*) ! *i* = (*i* ∈ *X*) **using** *iswitness* **by** *auto*
then show *i* < *m* ⟶ *w* ! *i* = (*witness X m*) ! *i* **using** *ents as* **by** *auto*
qed
then show *?thesis* **using** *list_eq_iff_nth_eq* **by** *auto*
qed
qed
then obtain *w* **where** {*xs*. *Ball X* ((!) *xs*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *m*}
= { *w* } **using** *Nitpick.Ex1_unfold*[**where** *P*=(λ*xs*. *Ball X* ((!) *xs*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *m*)]
by *auto*

then have *card* {*xs*. *Ball X* ((!) *xs*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *m*} = *card* { *w* } **by** *auto*

also have ... = 1 **by** *auto*
also have ... = $2\hat{}(m - card\ X - card\ Y)$ **using** *cardXY* **by** *auto*
finally show *?case* **.**
next
case (*insert e S*)
then have *eX*: $e \notin X$ **and** *eY*: $e \notin Y$ **by** *auto*
from *insert*(*8*) **have** *insert e S* $\subseteq \{0..<m\}$ **by** *auto*
then have *ebetween0m*: $e \in \{0..<m\}$ **by** *auto*

have *fm*: *finite* $\{0..<m\}$ **by** *auto*
have *cardm*: *card* $\{0..<m\} = m$ **by** *auto*
from *insert*(*8*) *eX eY ebetween0m* **have** *sub*: $X \cup Y \subset \{0..<m\}$ **by** *auto*
from *insert* **have** *card* $(X \cap Y) = 0$ **by** *auto*
then have *cardXY*: *card* $(X \cup Y)$ = *card* X + *card* Y **using** *card_Un_Int*[*OF insert*(*4*) *insert*(*5*)] **by** *auto*

have $m >$ *card* X + *card* Y **using** *psubset_card_mono*[*OF fm sub*] *cardm cardXY* **by**(*auto*)
then have *carde*: $1 + (m - card\ X - card\ Y - 1) = m - card\ X - card\ Y$ **by** *auto*

have *is1*: $\{xs.\ Ball\ X\ ((!)\ xs) \wedge (\forall i \in Y.\ \neg\ xs\ !\ i) \wedge length\ xs = m \wedge xs!e\}$
$= \{xs.\ Ball\ (insert\ e\ X)\ ((!)\ xs) \wedge (\forall i \in Y.\ \neg\ xs\ !\ i) \wedge length\ xs = m\}$ **by** *auto*
have *is2*: $\{xs.\ Ball\ X\ ((!)\ xs) \wedge (\forall i \in Y.\ \neg\ xs\ !\ i) \wedge length\ xs = m \wedge {\sim}xs!e\}$
$= \{xs.\ Ball\ X\ ((!)\ xs) \wedge (\forall i \in (insert\ e\ Y).\ \neg\ xs\ !\ i) \wedge length\ xs = m\}$ **by** *auto*

have *2*: $\{xs.\ Ball\ X\ ((!)\ xs) \wedge (\forall i \in Y.\ \neg\ xs\ !\ i) \wedge length\ xs = m \wedge xs!e\}$
$\cup \{xs.\ Ball\ X\ ((!)\ xs) \wedge (\forall i \in Y.\ \neg\ xs\ !\ i) \wedge length\ xs = m \wedge {\sim}xs!e\}$
$= \{xs.\ Ball\ X\ ((!)\ xs) \wedge (\forall i \in Y.\ \neg\ xs\ !\ i) \wedge length\ xs = m\}$ **by** *auto*

have *3*: $\{xs.\ Ball\ X\ ((!)\ xs) \wedge (\forall i \in Y.\ \neg\ xs\ !\ i) \wedge length\ xs = m \wedge xs!e\}$
$\cap \{xs.\ Ball\ X\ ((!)\ xs) \wedge (\forall i \in Y.\ \neg\ xs\ !\ i) \wedge length\ xs = m \wedge {\sim}xs!e\} = \{\}$ **by** *auto*

have *fX*: *finite* (*insert e X*)
and *disjeXY*: *insert e* $X \cap Y = \{\}$
and *cutX*: $S \cap (insert\ e\ X \cup Y) = \{\}$
and *uniX*: $S \cup insert\ e\ X \cup Y = \{0..<m\}$ **using** *insert* **by** *auto*
have *fY*: *finite* (*insert e Y*)

and *disjXeY*: $X \cap$ (*insert e Y*) = {}
and *cutY*: $S \cap (X \cup$ *insert e Y*) = {}
and *uniY*: $S \cup\ X \cup$ *insert e Y* = {*0..<m*} **using** *insert* **by** *auto*

have *card* {*xs. Ball X* ((!) *xs*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *m*}
= *card* {*xs. Ball X* ((!) *xs*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *m* ∧ *xs*!*e*}
+ *card* {*xs. Ball X* ((!) *xs*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *m* ∧ ~*xs*!*e*}
apply(*subst card_Un_Int*)
apply(*rule fbool*) **apply**(*rule fbool*) **using** *2 3* **by** *auto*
also
have ... = *card* {*xs. Ball* (*insert e X*) ((!) *xs*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *m*}
+ *card* {*xs. Ball X* ((!) *xs*) ∧ (∀ *i*∈(*insert e Y*). ¬ *xs* ! *i*) ∧ *length xs* = *m*} **by** (*simp only*: *is1 is2*)

also
have ... = *2* ^(*m* − *card* (*insert e X*) − *card Y*)
+ *2* ^(*m* − *card X* − *card* (*insert e Y*))
apply(*simp only*: *insert*(*3*)[*of insert e X Y*, *OF fX insert*(*5*) *disjeXY cutX uniX*])
by(*simp only*: *insert*(*3*)[*of X insert e Y*, *OF insert*(*4*) *fY disjXeY cutY uniY*])
also
have ... = *2* ^(*m* − *card X* − *card Y* − *1*)
+ *2* ^(*m* − *card X* − *card Y* − *1*) **using** *insert*(*4*,*5*) *eX eY* **by** *auto*
also
have ... = *2* ∗ *2* ^(*m* − *card X* − *card Y* − *1*) **by** *auto*
also have ... = *2* ^(*1* + (*m* − *card X* − *card Y* − *1*)) **by** *auto*
also have ... = *2* ^(*m* − *card X* − *card Y*) **using** *carde* **by** *auto*
finally show *?case* **.**
qed

lemma *card2*: **assumes** *finite X* **and** *finite Y* **and** $X \cap Y$ = {} **and** *x*: $X \cup Y \subseteq$ {*0..<m*}
shows *card* {*xs*. (∀ *i*∈*X*. *xs* ! *i*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *m*} = *2* ^(*m* − *card X* − *card Y*)
proof −
let *?S* = {*0..<m*}−($X \cup Y$)
from *x* **have** *a*: *?S* $\cup X \cup Y$ = {*0..<m*} **by** *auto*
have *b*: *?S* $\cap (X \cup Y)$ = {} **by** *auto*
show *?thesis* **apply**(*rule card1*[**where** *?S*=*?S*]) **by**(*simp_all add*: *assms a b*)

qed

7.3 Average out the second sum for free-absch

lemma *Expactation2or1*: *finite S* $\Longrightarrow$ *finite Tr* $\Longrightarrow$ *finite Fa* $\Longrightarrow$ *card Tr* + *card Fa* + *card S* $\leq$ *l* $\Longrightarrow$
S $\cap$ (*Tr* $\cup$ *Fa*) = {} $\Longrightarrow$ *Tr* $\cap$ *Fa* = {} $\Longrightarrow$ *S* $\cup$ *Tr* $\cup$ *Fa* $\subseteq$ {*0..<l*} $\Longrightarrow$
($\sum$ *x*$\in${*xs*. ($\forall$ *i*$\in$*Tr*. *xs* ! *i*) $\wedge$ ($\forall$ *i*$\in$*Fa*. $\neg$ *xs* ! *i*) $\wedge$ *length xs* = *l*}. $\sum$*j*$\in$*S*. *if x* ! *j then 2 else 1*)
= *3* / *2* $*$ *real* (*card S*) $*$ *2* ^ (*l* − *card Tr* − *card Fa*)
proof (*induct arbitrary*: *Tr Fa rule*: *finite_induct*)
case (*insert e S*)

from *insert*(*7*) **have** *e* $\in$ (*insert e S*) **and** *eTr*: *e* $\notin$ *Tr* **and** *eFa*: *e* $\notin$ *Fa* **by** *auto*
from *insert*(*9*) **have** *tra*: *Tr* $\subseteq$ {*0..<l*} **and** *trb*: *Fa* $\subseteq$ {*0..<l*} **and** *trc*: *e* < *l* **by** *auto*

have *ntrFa*: *l* > (*card Tr* + *card Fa*) **using** *insert*(*6*) *card_insert_if* *insert*(*1*,*2*) **by** *auto*

have *myhelp2*: *1* + (*l* − *card Tr* − *card Fa* −*1*) = *l* − *card Tr* − *card Fa* **using** *ntrFa* **by** *auto*

have *juhuTr*: {*xs*. ($\forall$ *i*$\in$*Tr*. *xs* ! *i*) $\wedge$ ($\forall$ *i*$\in$*Fa*. $\neg$ *xs* ! *i*) $\wedge$ *length xs* = *l* $\wedge$ *xs*!*e*}
= {*xs*. ($\forall$ *i*$\in$(*insert e Tr*). *xs* ! *i*) $\wedge$ ($\forall$ *i*$\in$*Fa*. $\neg$ *xs* ! *i*) $\wedge$ *length xs* = *l*}
by *auto*
have *juhuFa*: {*xs*. ($\forall$ *i*$\in$*Tr*. *xs* ! *i*) $\wedge$ ($\forall$ *i*$\in$*Fa*. $\neg$ *xs* ! *i*) $\wedge$ *length xs* = *l* $\wedge$ ~*xs*!*e*}
= {*xs*. ($\forall$ *i*$\in$*Tr*. *xs* ! *i*) $\wedge$ ($\forall$ *i*$\in$(*insert e Fa*). $\neg$ *xs* ! *i*) $\wedge$ *length xs* = *l*}
by *auto*

let *?Tre* = {*xs*. ($\forall$ *i*$\in$(*insert e Tr*). *xs* ! *i*) $\wedge$ ($\forall$ *i*$\in$*Fa*. $\neg$ *xs* ! *i*) $\wedge$ *length xs* = *l*}

have *card ?Tre* = *2* ^ (*l* − *card* (*insert e Tr*) − *card Fa*)
apply(*rule card2*) **using** *insert* **by** *simp_all*
then have *resi*: *card ?Tre* = *2*^(*l* − *card Tr* − *card Fa* − *1*) **using** *insert*(*4*) *eTr* **by** *auto*
have *yabaTr*: ($\sum$ *x*$\in$*?Tre*. *2*::*real*) = *2* $*$ *2*^(*l* − *card Tr* − *card Fa* − *1*) **using** *resi* **by** (*simp add*: *power_commutes*)

let *?Fae* = {*xs*. (∀ *i*∈*Tr*. *xs* ! *i*) ∧ (∀ *i*∈(*insert e Fa*). ¬ *xs* ! *i*) ∧ *length xs* = *l*}

have *card ?Fae* = *2* ^ (*l* − *card Tr* − *card* (*insert e Fa*))
apply(*rule card2*) **using** *insert* **by** *simp_all*
then have *resi2*: *card ?Fae* = *2*^(*l* − *card Tr* − *card Fa* − *1*) **using** *insert*(*5*) *eFa* **by** *auto*
have *yabaFa*: (∑ *x*∈*?Fae*. *1*::*real*) = *1* ∗ *2* ^ (*l* − *card Tr* − *card Fa* − *1*) **using** *resi2* **by** (*simp add*: *power_commutes*)

{ fix *X Y*
have {*xs*. (∀ *i*∈*X*. *xs* ! *i*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *l* ∧ *xs*!*e*}
∩ {*xs*. (∀ *i*∈*X*. *xs* ! *i*) ∧ (∀ *i*∈*Y*. ¬ *xs* ! *i*) ∧ *length xs* = *l* ∧ ~*xs*!*e*} = {} **by** *auto*
} note *3*=*this*

have (∑ *x*∈{*xs*. (∀ *i*∈*Tr*. *xs* ! *i*) ∧ (∀ *i*∈*Fa*. ¬ *xs* ! *i*) ∧ *length xs* = *l*}. ∑ *j*∈(*insert e S*). *if x* ! *j then* (*2*::*real*) *else 1*)
= (∑ *x*∈{*xs*. (∀ *i*∈*Tr*. *xs* ! *i*) ∧ (∀ *i*∈*Fa*. ¬ *xs* ! *i*) ∧ *length xs* = *l* ∧ *xs*!*e*}. ∑ *j*∈(*insert e S*). *if x* ! *j then 2 else 1*)
+ (∑ *x*∈{*xs*. (∀ *i*∈*Tr*. *xs* ! *i*) ∧ (∀ *i*∈*Fa*. ¬ *xs* ! *i*) ∧ *length xs* = *l* ∧ ~*xs*!*e*}. ∑ *j*∈(*insert e S*). *if x* ! *j then 2 else 1*)
(**is** (∑ *x*∈*?all*. *?f x*) = (∑ *x*∈*?allT*. *?f x*) + (∑ *x*∈*?allF*. *?f x*))
proof −
have (∑ *x*∈*?all*. ∑ *j*∈(*insert e S*). *if x* ! *j then 2 else 1*)
= (∑ *x*∈(*?allT* ∪ *?allF*). ∑ *j*∈(*insert e S*). *if x* ! *j then 2 else 1*) **apply**(*rule sum.cong*) **by**(*auto*)
also have. . . = ((∑ *x*∈(*?allT*). ∑ *j*∈(*insert e S*). *if x* ! *j then* (*2*::*real*) *else 1*)
+ (∑ *x*∈(*?allF*). ∑ *j*∈(*insert e S*). *if x* ! *j then* (*2*::*real*) *else 1*))
− (∑ *x*∈(*?allT* ∩ *?allF*). ∑ *j*∈(*insert e S*). *if x* ! *j then 2 else 1*)
apply (*rule sum_Un*) **apply**(*rule fbool*)+ **done**
also have . . . = (∑ *x*∈(*?allT*). ∑ *j*∈(*insert e S*). *if x* ! *j then 2 else 1*)
+ (∑ *x*∈(*?allF*). ∑ *j*∈(*insert e S*). *if x* ! *j then 2 else 1*)
by(*simp add*: *3*)
finally show *?thesis* **.**
qed
also
have . . . = (∑ *x*∈*?Tre*. ∑ *j*∈(*insert e S*). *if x* ! *j then 2 else 1*)

+ ($\sum x \in ?Fae.\ \sum j \in (insert\ e\ S).\ if\ x\ !\ j\ then\ 2\ else\ 1$)
using *juhuTr juhuFa* **by** *auto*
also
have ... = ($\sum x \in ?Tre.\ (\lambda x.\ 2)\ x + (\lambda x.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1))\ x$)
+ ($\sum x \in ?Fae.\ (\lambda x.\ 1)\ x + (\lambda x.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1))\ x$)
using *insert(1,2)* **by** *auto*
also
have ... = ($\sum x \in ?Tre.\ 2$) + ($\sum x \in ?Tre.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1)$)
+ (($\sum x \in ?Fae.\ 1$) + ($\sum x \in ?Fae.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1)$))
by (*simp add*: *Groups_Big.comm_monoid_add_class.sum.distrib*)
also
have ... = $2 * 2\hat{}(l - card\ Tr - card\ Fa - 1) + (\sum x \in ?Tre.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1))$
+ ($1 * 2\hat{}(l - card\ Tr - card\ Fa - 1) + (\sum x \in ?Fae.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1))$)
by(*simp only*: *yabaTr yabaFa*)
also
have ... = $(2::real) * 2\hat{}(l - card\ Tr - card\ Fa - 1) + (\sum x \in ?Tre.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1))$
+ $(1::real) * 2\hat{}(l - card\ Tr - card\ Fa - 1) + (\sum x \in ?Fae.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1))$
by *auto*
also
have ... = $(3::real) * 2\hat{}(l - card\ Tr - card\ Fa - 1)$ +
$(\sum x \in ?Tre.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1)) + (\sum x \in ?Fae.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1))$
by *simp*
also
have ... = $3 * 2\hat{}(l - card\ Tr - card\ Fa - 1)$ +
$3\ /\ 2 * real\ (card\ S) * 2\ \hat{}\ (l - card\ (insert\ e\ Tr) - card\ Fa)$ +
$(\sum x \in ?Fae.\ (\sum j \in S.\ if\ x\ !\ j\ then\ 2\ else\ 1))$
apply(*subst insert(3)*) **using** *insert* **by** *simp_all*
also
have ... = $3 * 2\hat{}(l - card\ Tr - card\ Fa - 1)$ +
$3\ /\ 2 * real\ (card\ S) * 2\ \hat{}\ (l - card\ (insert\ e\ Tr) - card\ Fa)$ +
$3\ /\ 2 * real\ (card\ S) * 2\ \hat{}\ (l - card\ Tr - card\ (insert\ e\ Fa))$
apply(*subst insert(3)*) **using** *insert* **by** *simp_all*
also
have ... = $3 * 2\hat{}(l - card\ Tr - card\ Fa - 1)$ +
$3\ /\ 2 * real\ (card\ S) * 2\hat{}\ (l - (card\ Tr + 1) - card\ Fa)$ +
$3\ /\ 2 * real\ (card\ S) * 2\hat{}\ (l - card\ Tr - (card\ Fa + 1))$ **using** *card_insert_if insert(4,5) eTr eFa* **by** *auto*
also

have ... = *3* ∗ *2*^(*l* − *card Tr* − *card Fa* − *1*) +
3 / *2* ∗ *real* (*card S*) ∗ *2*^ (*l* − *card Tr* − *card Fa* − *1*) +
3 / *2* ∗ *real* (*card S*) ∗ *2*^ (*l* − *card Tr* − *card Fa* − *1*) **by** *auto*
also
have ... = (*3*/*2* ∗ *2* + *2* ∗ *3* / *2* ∗ *real* (*card S*)) ∗ *2*^ (*l* − *card Tr* − *card Fa* − *1*) **by** *algebra*
also
have ... = (*3* / *2* ∗ (*1* + *real* (*card S*))) ∗ *2* ∗ *2*^ (*l* − *card Tr* − *card Fa* − *1*) **by** *simp*
also
have ... = (*3* / *2* ∗ (*1* + *real* (*card S*))) ∗ *2*^ (*Suc* (*l* − *card Tr* − *card Fa* −*1*)) **by** *simp*
also
have ... = (*3* / *2* ∗ (*1* + *real* (*card S*))) ∗ *2*^ (*l* − *card Tr* − *card Fa*) **using** *myhelp2* **by** *auto*
also
have ... = (*3* / *2* ∗ (*real* (*1* + *card S*))) ∗ *2*^ (*l* − *card Tr* − *card Fa*) **by** *simp*
also
have ... = (*3* / *2* ∗ *real* (*card* (*insert e S*))) ∗ *2*^ (*l* − *card Tr* − *card Fa*) **using** *insert*(*1*,*2*) **by** *auto*
finally show *?case* **.**
qed *simp*

end

8 Effect of mtf2

theory *MTF2_Effects*
imports *Move_to_Front*
begin

lemma *difind_difelem*:
i < *length xs* ⟹ *distinct xs* ⟹ *xs* ! *j* = *a* ⟹ *j* < *length xs* ⟹ *i* ≠ *j*
⟹ ~ *a* = *xs* ! *i*
apply(*rule ccontr*) **by**(*metis index_nth_id*)

lemma *fullchar*: **assumes** *index xs q* < *length xs*
shows

$(i < \mathit{length}\ xs) =$
$(\mathit{index}\ xs\ q < i \wedge i < \mathit{length}\ xs$
$\vee\ \mathit{index}\ xs\ q = i$
$\vee\ \mathit{index}\ xs\ q - n \leq i \wedge i < \mathit{index}\ xs\ q$
$\vee\ i < \mathit{index}\ xs\ q - n)$
using *assms* **by** *auto*

lemma *mtf2_effect*:
$q \in \mathit{set}\ xs \Longrightarrow \mathit{distinct}\ xs \Longrightarrow (\mathit{index}\ xs\ q < i \wedge i < \mathit{length}\ xs \longrightarrow (\mathit{index}\ (\mathit{mtf2}\ n\ q\ xs)\ (xs!i) = \mathit{index}\ xs\ (xs!i) \wedge \mathit{index}\ xs\ q < \mathit{index}\ (\mathit{mtf2}\ n\ q\ xs)\ (xs!i) \wedge \mathit{index}\ (\mathit{mtf2}\ n\ q\ xs)\ (xs!i) < \mathit{length}\ xs))$
$\wedge\ (\mathit{index}\ xs\ q = i \longrightarrow (\mathit{index}\ (\mathit{mtf2}\ n\ q\ xs)\ (xs!i) = \mathit{index}\ xs\ q - n \wedge \mathit{index}\ (\mathit{mtf2}\ n\ q\ xs)\ (xs!i) = \mathit{index}\ xs\ q - n))$
$\wedge\ (\mathit{index}\ xs\ q - n \leq i \wedge i < \mathit{index}\ xs\ q \longrightarrow (\mathit{index}\ (\mathit{mtf2}\ n\ q\ xs)\ (xs!i) = \mathit{Suc}\ (\mathit{index}\ xs\ (xs!i)) \wedge \mathit{index}\ xs\ q - n < \mathit{index}\ (\mathit{mtf2}\ n\ q\ xs)\ (xs!i) \wedge \mathit{index}\ (\mathit{mtf2}\ n\ q\ xs)\ (xs!i) \leq \mathit{index}\ xs\ q))$
$\wedge\ (i < \mathit{index}\ xs\ q - n \longrightarrow (\mathit{index}\ (\mathit{mtf2}\ n\ q\ xs)\ (xs!i) = \mathit{index}\ xs\ (xs!i) \wedge \mathit{index}\ (\mathit{mtf2}\ n\ q\ xs)\ (xs!i) < \mathit{index}\ xs\ q - n))$
unfolding *mtf2_def*
apply (*induct n*)
proof −
case (*Suc n*)
note *indH=Suc(1)[OF Suc(2) Suc(3), simplified Suc(2) if_True]*
note *qinxs=Suc(2)[simp]*
note *distxs=Suc(3)[simp]*
show *?case* (**is** *?toshow*)
apply(*simp only*: *qinxs if_True*)
proof (*cases index xs q* $\geq$ *Suc n*)
case *True*
note *True1=this*
from *True* **have** *onemore*: $[\mathit{index}\ xs\ q - \mathit{Suc}\ n..<\mathit{index}\ xs\ q] = (\mathit{index}\ xs\ q - \mathit{Suc}\ n)\ \#\ [\mathit{index}\ xs\ q - n..<\mathit{index}\ xs\ q]$
using *Suc_diff_Suc upt_rec* **by** *auto*

from *onemore* **have** *yeah*: $\mathit{swaps}\ [\mathit{index}\ xs\ q - \mathit{Suc}\ n..<\mathit{index}\ xs\ q]\ xs$
$= \mathit{swap}\ (\mathit{index}\ xs\ q - \mathit{Suc}\ n)\ (\mathit{swaps}\ [\mathit{index}\ xs\ q - n..<\mathit{index}\ xs\ q]\ xs)$ **by** *auto*

have *sis*: $\mathit{Suc}\ (\mathit{index}\ xs\ q - \mathit{Suc}\ n) = \mathit{index}\ xs\ q - n$ **using** *True Suc_diff_Suc* **by** *auto*

have *indq*: $\mathit{index}\ xs\ q < \mathit{length}\ xs$
apply(*rule index_less*) **by** *auto*

let ?i′ = index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i)
let ?x = (xs!i) **and** ?xs=(swaps [index xs q − n..<index xs q] xs)
and ?n=(index xs q − Suc n)
have ?i′
= index (swap (index xs q − Suc n) (swaps [index xs q − n..<index xs q] xs)) (xs!i) **using** yeah **by** auto
also have ... = (if ?x = ?xs ! ?n then Suc ?n else if ?x = ?xs ! Suc ?n then ?n else index ?xs ?x)
apply(rule index_swap_distinct)
apply(simp)
apply(simp add: sis) **using** indq **by** linarith
finally have i′: ?i′ = (if ?x = ?xs ! ?n then Suc ?n else if ?x = ?xs ! Suc ?n then ?n else index ?xs ?x) **.**

let ?i′′=index (swaps [index xs q − n..<index xs q] xs) (xs ! i)

show (index xs q < i ∧ i < length xs ⟶
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) = index xs (xs ! i) ∧
index xs q < index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) ∧
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) < length xs) ∧
(index xs q = i ⟶
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) = index xs q − Suc n ∧
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) = index xs q − Suc n) ∧
(index xs q − Suc n ≤ i ∧ i < index xs q ⟶
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) = Suc (index xs (xs ! i)) ∧
index xs q − Suc n < index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) ∧
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) ≤ index xs q) ∧
(i < index xs q − Suc n ⟶
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) = index xs (xs ! i) ∧
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) < index xs q − Suc n)
apply(intro conjI)
apply(intro impI) **apply**(elim conjE) **prefer** 4 **apply**(intro impI) **pre-**

fer *4* **apply**(*intro impI*) **apply**(*elim conjE*)
prefer *4* **apply**(*intro impI*) **prefer** *4*
proof (*goal_cases*)
case *1*
have *indH1*: (*index xs q* < *i* ∧ *i* < *length xs* ⟶
?i'' = *index xs* (*xs* ! *i*)) **using** *indH* **by** *auto*
assume *ass*: *index xs q* < *i* **and** *ass2*:*i* < *length xs*
then have *a*: *?i''* = *index xs* (*xs* ! *i*) **using** *indH1* **by** *auto*
also have *a'*: ... = *i* **apply**(*rule index_nth_id*) **using** *ass2* **by**(*auto*)
finally have *ii*: *?i''* = *i* **.**
have *fstF*: ~ *?x* = *?xs* ! *?n* **apply**(*rule difind_difelem*[**where** *j*=*index* (*swaps* [*index xs q* − *n*..<*index xs q*] *xs*) (*xs*!*i*)])
using *indq* **apply** (*simp add*: *less_imp_diff_less*)
apply(*simp*)
apply(*rule nth_index*) **apply**(*simp*) **using** *ass2* **apply**(*simp*)
apply(*rule index_less*)
apply(*simp*) **using** *ass2* **apply**(*simp*)
apply(*simp*)
using *ii ass* **by** *auto*
have *sndF*: ~ *?x* = *?xs* ! *Suc ?n* **apply**(*rule difind_difelem*[**where** *j*=*index* (*swaps* [*index xs q* − *n*..<*index xs q*] *xs*) (*xs*!*i*)])
using *indq True* **apply** (*simp add*: *Suc_diff_Suc less_imp_diff_less*)
apply(*simp*)
apply(*rule nth_index*) **apply**(*simp*) **using** *ass2* **apply**(*simp*)
apply(*rule index_less*)
apply(*simp*) **using** *ass2* **apply**(*simp*)
apply(*simp*)
using *ii ass Suc_diff_Suc True* **by** *auto*

have *?i'* = *index xs* (*xs* ! *i*) **unfolding** *i'* **using** *fstF sndF a* **by** *simp*
then show *?case* **using** *a' ass ass2* **by** *auto*
next
case *2*
have *indH2*: *index xs q* = *i* ⟶ *?i''* = *index xs* (*xs* ! *i*) − *n* **using** *indH* **by** *auto*
assume *index xs q* = *i*
then have *ass*: *i* = *index xs q* **by** *auto*
with *indH2* **have** *a*: *i* − *n* = *?i''* **by** *auto*
from *ass* **have** *c*: *index xs* (*xs* ! *i*) = *i* **by** *auto*
have *Suc* (*index xs q* − *Suc n*) = *i* − *n* **using** *ass True Suc_diff_Suc* **by** *auto*
also have ... = *?i''* **using** *a* **by** *auto*
finally have *a*: *Suc ?n* = *?i''* **.**

```
      have sndTrue: ?x = ?xs ! Suc ?n apply(simp add: a)
            apply(rule nth_index[symmetric]) by (simp add: ass)
       have fstFalse: ~ ?x = ?xs ! ?n apply(rule difind_difelem[where
j=index (swaps [index xs q - n..<index xs q] xs) (xs!i)])
       using indq True apply (simp add: Suc_diff_Suc less_imp_diff_less)
        apply(simp)
        apply(rule nth_index) apply(simp) using ass  apply(simp)
        apply(rule index_less)
         apply(simp) using ass  apply(simp)
         apply(simp)
        using a by auto

      have ?i' = index xs (xs ! index xs q) - Suc n
         unfolding i' using sndTrue fstFalse by simp
      with ass show ?case by auto
    next
      case 3
      have indH3: index xs q - n ≤ i ∧ i < index xs q
            ⟶  ?i'' = Suc (index xs (xs ! i)) using indH by auto
      assume ass: index xs q - Suc n ≤ i and
            ass2: i < index xs q
     from ass2 have ilen: i < length xs using indq dual_order.strict_trans
by blast
      show ?case
      proof (cases index xs q - n ≤ i)
        case False
        then have i < index xs q - n by auto
         moreover have (i < index xs q - n ⟶ ?i'' = index xs (xs ! i))
using indH by auto
        ultimately have d: ?i'' = index xs (xs ! i) by simp
        from False ass have b: index xs q - Suc n = i by auto
        have index xs q < length xs apply(rule index_less) by (auto)
        have c: index xs (xs ! i) = i
          apply(rule index_nth_id) apply(simp) using indq ass2 using
less_trans by blast
        from b c d have f: ?i'' = index xs q - Suc n by auto
        have fstT: ?xs ! ?n = ?x
          apply(simp only: f[symmetric]) apply(rule nth_index)
          by (simp add: ilen)

        have ?i' = Suc (index xs q - Suc n)
         unfolding i' using fstT by simp
        also have ... = Suc (index xs (xs ! i)) by(simp only: b c)
        finally show ?thesis using c False ass by auto
```

next
case *True*
with *ass2 indH3* **have** *a*: *?i''* = *Suc* (*index xs* (*xs* ! *i*)) **by** *auto*
have *jo*: *index xs* (*xs* ! *i*) = *i* **apply**(*rule index_nth_id*) **using** *ilen* **by**(*auto*)
have *fstF*: ~ *?x* = *?xs* ! *?n* **apply**(*rule difind_difelem*[**where** *j*=*index* (*swaps* [*index xs q* − *n*..<*index xs q*] *xs*) (*xs*!*i*)])
using *indq* **apply** (*simp add*: *less_imp_diff_less*)
apply(*simp*)
apply(*rule nth_index*) **apply**(*simp*) **using** *ilen* **apply**(*simp*)
apply(*rule index_less*)
apply(*simp*) **using** *ilen* **apply**(*simp*)
apply(*simp*)
apply(*simp only*: *a jo*) **using** *True* **by** *auto*
have *sndF*: ~ *?x* = *?xs* ! *Suc ?n* **apply**(*rule difind_difelem*[**where** *j*=*index* (*swaps* [*index xs q* − *n*..<*index xs q*] *xs*) (*xs*!*i*)])
using *True1* **apply** (*simp add*: *Suc_diff_Suc less_imp_diff_less*)
apply(*simp*)
apply(*rule nth_index*) **apply**(*simp*) **using** *ilen* **apply**(*simp*)
apply(*rule index_less*)
apply(*simp*) **using** *ilen* **apply**(*simp*)
apply(*simp*)
apply(*simp only*: *a jo*) **using** *True1* **apply** (*simp add*: *Suc_diff_Suc less_imp_diff_less*)
using *True* **by** *auto*
have *?i'* = *Suc* (*index xs* (*xs* ! *i*)) **unfolding** *i'* **using** *fstF sndF a* **by** *simp*
then show *?thesis* **using** *ass ass2 jo* **by** *auto*
qed
next
case *4*
assume *ass*: *i* < *index xs q* − *Suc n*
then have *ass2*: *i* < *index xs q* − *n* **by** *auto*
moreover have (*i* < *index xs q* − *n* ⟶ *?i''* = *index xs* (*xs* ! *i*)) **using** *indH* **by** *auto*
ultimately have *a*: *?i''* = *index xs* (*xs* ! *i*) **by** *auto*
from *ass2* **have** *i* < *index xs q* **by** *auto*
then have *ilen*: *i* < *length xs* **using** *indq dual_order.strict_trans* **by** *blast*

have *jo*: *index xs* (*xs* ! *i*) = *i* **apply**(*rule index_nth_id*) **using** *ilen* **by**(*auto*)
have *fstF*: ~ *?x* = *?xs* ! *?n* **apply**(*rule difind_difelem*[**where** *j*=*index*

```
(swaps [index xs q − n..<index xs q] xs) (xs!i)])
        using indq apply (simp add: less_imp_diff_less)
        apply(simp)
        apply(rule nth_index) apply(simp) using ilen apply(simp)
        apply(rule index_less)
          apply(simp) using ilen apply(simp)
          apply(simp)
        apply(simp only: a jo) using ass by auto
       have sndF: ~ ?x = ?xs ! Suc ?n apply(rule difind_difelem[where
j=index (swaps [index xs q − n..<index xs q] xs) (xs!i)])
        using True1 apply (simp add: Suc_diff_Suc less_imp_diff_less)
        apply(simp)
        apply(rule nth_index) apply(simp) using ilen apply(simp)
        apply(rule index_less)
          apply(simp) using ilen apply(simp)
          apply(simp)
       apply(simp only: a jo) using True1 apply (simp add: Suc_diff_Suc
less_imp_diff_less)
        using ass by auto
     have ?i' = (index xs (xs ! i)) unfolding i' using fstF sndF a by simp
      then show ?case using jo ass by auto
    qed
  next
    case False

    then have smalla: index xs q − Suc n = index xs q − n by auto
    then have nomore: swaps [index xs q − Suc n..<index xs q] xs
           =swaps [index xs q − n..<index xs q] xs by auto
    show (index xs q < i ∧ i < length xs ⟶
     index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) = index xs
(xs ! i) ∧
     index xs q < index (swaps [index xs q − Suc n..<index xs q] xs) (xs !
i) ∧
     index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) < length xs)
∧
    (index xs q = i ⟶
     index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) = index xs
q − Suc n ∧
     index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) = index xs
q − Suc n) ∧
    (index xs q − Suc n ≤ i ∧ i < index xs q ⟶
     index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) = Suc (index
xs (xs ! i)) ∧
     index xs q − Suc n < index (swaps [index xs q − Suc n..<index xs q]
```

xs) (xs ! i) ∧
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) ≤ index xs q) ∧
(i < index xs q − Suc n ⟶
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) = index xs (xs ! i) ∧
index (swaps [index xs q − Suc n..<index xs q] xs) (xs ! i) < index xs q − Suc n)
unfolding *nomore smalla* **by** (*rule indH*)
qed
next
case *0*
then show *?case* **apply**(*simp*)
proof (*safe*, *goal_cases*)
case *1*
have *index xs* (*xs* ! *i*) = *i* **apply**(*rule index_nth_id*) **using** *1* **by** *auto*
with *1* **show** *?case* **by** *auto*
next
case *2*
have *xs* ! *index xs q* = *q* **using** *2* **by**(*auto*)
with *2* **show** *?case* **by** *auto*
next
case *3*
have *a*: *index xs q* < *length xs* **apply**(*rule index_less*) **using** *3* **by** *auto*
have *index xs* (*xs* ! *i*) = *i* **apply**(*rule index_nth_id*) **apply**(*fact 3*(*2*)) **using** *3*(*3*) *a* **by** *auto*
with *3* **show** *?case* **by** *auto*
qed
qed

lemma *mtf2_forward_effect1*:
q ∈ set xs ⟹ distinct xs ⟹ index xs q < i ∧ i < length xs
⟹ index (mtf2 n q xs) (xs ! i) = index xs (xs ! i) ∧ index xs q < index (mtf2 n q xs) (xs ! i) ∧ index (mtf2 n q xs) (xs ! i) < length xs **and**

mtf2_forward_effect2: q ∈ set xs ⟹ distinct xs ⟹ index xs q = i
⟹ index (mtf2 n q xs) (xs!i) = index xs q − n ∧ index xs q − n = index (mtf2 n q xs) (xs!i) **and**
mtf2_forward_effect3: q ∈ set xs ⟹ distinct xs ⟹ index xs q − n ≤ i ∧ i < index xs q
⟹ index (mtf2 n q xs) (xs!i) = Suc (index xs (xs!i)) ∧ index xs q − n < index (mtf2 n q xs) (xs!i) ∧ index (mtf2 n q xs) (xs!i) ≤ index xs q **and**

mtf2_forward_effect4: q ∈ set xs ⟹ distinct xs ⟹ i < index xs q − n

$\Longrightarrow$ *index* (*mtf2 n q xs*) (*xs*!*i*) = *index xs* (*xs*!*i*) ∧ *index* (*mtf2 n q xs*) (*xs*!*i*) < *index xs q* − *n*
apply(*safe*) **using** *mtf2_effect* **by** *metis*+

lemma *yes*[*simp*]: *index xs x* < *length xs*
$\Longrightarrow$ (*xs*!*index xs x*) = *x* **apply**(*rule nth_index*) **by** (*simp add*: *index_less_size_conv*)

lemma *mtf2_forward_effect1'*:
q ∈ *set xs* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ *index xs q* < *index xs x* ∧ *index xs x* < *length xs*
$\Longrightarrow$ *index* (*mtf2 n q xs*) *x* = *index xs x* ∧ *index xs q* < *index* (*mtf2 n q xs*) *x* ∧ *index* (*mtf2 n q xs*) *x* < *length xs*
using *mtf2_forward_effect1*[**where** *xs*=*xs* **and** *i*=*index xs x*] *yes*
by(*auto*)

lemma
mtf2_forward_effect2': *q* ∈ *set xs* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ *index xs q* = *index xs x*
$\Longrightarrow$ *index* (*mtf2 n q xs*) (*xs*!*index xs x*) = *index xs q* − *n* ∧ *index xs q* − *n* = *index* (*mtf2 n q xs*) (*xs*!*index xs x*)
using *mtf2_forward_effect2*[**where** *xs*=*xs* **and** *i*=*index xs x*]
by *fast*

lemma
mtf2_forward_effect3': *q* ∈ *set xs* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ *index xs q* − *n* ≤ *index xs x* $\Longrightarrow$ *index xs x* < *index xs q*
$\Longrightarrow$ *index* (*mtf2 n q xs*) (*xs*!*index xs x*) = *Suc* (*index xs* (*xs*!*index xs x*)) ∧ *index xs q* − *n* < *index* (*mtf2 n q xs*) (*xs*!*index xs x*) ∧ *index* (*mtf2 n q xs*) (*xs*!*index xs x*) ≤ *index xs q*
using *mtf2_forward_effect3*[**where** *xs*=*xs* **and** *i*=*index xs x*]
by *fast*

lemma
mtf2_forward_effect4': *q* ∈ *set xs* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ *index xs x* < *index xs q* − *n*
$\Longrightarrow$ *index* (*mtf2 n q xs*) (*xs*!*index xs x*) = *index xs* (*xs*!*index xs x*) ∧ *index* (*mtf2 n q xs*) (*xs*!*index xs x*) < *index xs q* − *n*
using *mtf2_forward_effect4*[**where** *xs*=*xs* **and** *i*=*index xs x*]
by *fast*

lemma *splitit*: (*index xs q* < *i* ∧ *i* < *length xs* $\Longrightarrow$ *P*)
$\Longrightarrow$ (*index xs q* = *i* $\Longrightarrow$ *P*)

$\Longrightarrow$ (*index xs q* − *n* ≤ *i* ∧ *i* < *index xs q* $\Longrightarrow$ *P*)
$\Longrightarrow$ (*i* < *index xs q* − *n* $\Longrightarrow$ *P*)
$\Longrightarrow$ (*i* < *length xs* $\Longrightarrow$ *P*)
by *force*

lemma *mtf2_forward_beforeq*: *q* ∈ *set xs* $\Longrightarrow$ *distinct xs* $\Longrightarrow$ *i* < *index xs q*
$\Longrightarrow$ *index* (*mtf2 n q xs*) (*xs*!*i*) ≤ *index xs q*
apply (*cases i* < *index xs q* − *n*)
using *mtf2_forward_effect4* **apply** *force*
using *mtf2_forward_effect3* **using** *leI* **by** *metis*

lemma *x_stays_before_y_if_y_not_moved_to_front*:
assumes *q* ∈ *set xs distinct xs x* ∈ *set xs y* ∈ *set xs y* ≠ *q*
and *x* < *y in xs*
shows *x* < *y in* (*mtf2 n q xs*)
proof −
from *assms*(*3*) **obtain** *i* **where** *i*: *i* = *index xs x* **and** *i2*: *i* < *length xs* **by** *auto*
from *assms*(*4*) **obtain** *j* **where** *j*: *j* = *index xs y* **and** *j2*: *j* < *length xs* **by** *auto*
have *x* < *y in xs* $\Longrightarrow$ *x* < *y in* (*mtf2 n q xs*)
apply(*cases i xs rule*: *splitit*[**where** *q*=*q* **and** *n*=*n*])
apply(*simp add*: *i assms*(*1*,*2*) *mtf2_forward_effect1'* *before_in_def*)
apply(*cases j xs rule*: *splitit*[**where** *q*=*q* **and** *n*=*n*])
apply (*metis before_in_def assms*(*1*−*3*) *i j less_imp_diff_less mtf2_effect nth_index set_mtf2*)
apply(*simp add*: *i j assms mtf2_forward_effect1' mtf2_forward_effect2' before_in_def*)
apply(*simp add*: *i j assms mtf2_forward_effect1' mtf2_forward_effect2' before_in_def*)
apply(*simp add*: *i j assms mtf2_forward_effect1' mtf2_forward_effect3' before_in_def*)
apply(*rule j2*)
apply(*cases j xs rule*: *splitit*[**where** *q*=*q* **and** *n*=*n*])
apply (*smt before_in_def assms*(*1*−*3*) *i j le_less_trans mtf2_forward_effect1 mtf2_forward_effect3 nth_index set_mtf2*)
using *assms*(*4*,*5*) *j* **apply** *simp*
apply (*smt Suc_leI before_in_def assms*(*1*−*3*) *i j le_less_trans lessI mtf2_forward_effect3 nth_index set_mtf2*)
apply (*simp add*: *before_in_def i j*)
apply(*rule j2*)

apply(*cases j xs rule*: *splitit*[**where** *q*=*q* **and** *n*=*n*])
apply (*smt before_in_def assms*(*1*−*3*) *i j le_less_trans mtf2_forward_effect1 mtf2_forward_effect4 nth_index set_mtf2*)
using *assms*(*4*−*5*) *j* **apply** *simp*
apply (*smt before_in_def assms*(*1*−*3*) *i j le_less_trans less_imp_le_nat mtf2_forward_effect3 mtf2_forward_effect4 nth_index set_mtf2*)
apply (*metis before_in_def assms*(*1*−*3*) *i j mtf2_forward_effect4 nth_index set_mtf2*)
apply(*rule j2*)
apply(*rule i2*) **done**
with *assms*(*6*) **show** *?thesis* **by** *auto*
qed

corollary *swapped_by_mtf2*: *q* ∈ *set xs* ⟹ *distinct xs* ⟹ *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹
x < *y in xs* ⟹ *y* < *x in* (*mtf2 n q xs*) ⟹ *y* = *q*
apply(*rule ccontr*) **using** *x_stays_before_y_if_y_not_moved_to_front not_before_in*
by (*metis before_in_setD1*)

lemma *x_stays_before_y_if_y_not_moved_to_front_2dir*: *q* ∈ *set xs* ⟹ *distinct xs* ⟹ *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *y* ≠ *q* ⟹
x < *y in xs* = *x* < *y in* (*mtf2 n q xs*)
oops

lemma *mtf2_backwards_effect1*:
assumes *index xs q* < *length xs q* ∈ *set xs distinct xs*
index xs q < *index* (*mtf2 n q xs*) (*xs* ! *i*) ∧ *index* (*mtf2 n q xs*) (*xs* ! *i*) < *length xs*
i < *length xs*
shows *index xs q* < *i* ∧ *i* < *length xs*
proof −
from *assms*(*4*) **have** ~ (*index xs q* − *n* = *index* (*mtf2 n q xs*) (*xs* ! *i*)) **by** *auto*
with *assms mtf2_forward_effect2* **have** *1*: ~ (*index xs q* = *i*) **by** *metis*
from *assms*(*4*) **have** ~ (*index xs q* − *n* < *index* (*mtf2 n q xs*) (*xs* ! *i*) ∧ *index* (*mtf2 n q xs*) (*xs* ! *i*) ≤ *index xs q*) **by** *auto*
with *assms mtf2_forward_effect3* **have** *2*: ~ (*index xs q* − *n* ≤ *i* ∧ *i* < *index xs q*) **by** *metis*
from *assms*(*4*) **have** ~ (*index* (*mtf2 n q xs*) (*xs* ! *i*) < *index xs q* − *n*) **by** *auto*
with *assms mtf2_forward_effect4* **have** *3*: ~ (*i* < *index xs q* − *n*) **by** *metis*

from *fullchar*[*OF assms*(*1*)] *assms*(*5*) *1 2 3* **show** *index xs q* < *i* ∧ *i* < *length xs* **by** *metis*
qed

lemma *mtf2_backwards_effect2*:
assumes *index xs q* < *length xs q* ∈ *set xs distinct xs index* (*mtf2 n q xs*) (*xs* ! *i*) = *index xs q* − *n*
i < *length xs*
shows *index xs q* = *i*
proof −
from *assms*(*4*) **have** ~ (*index xs q* < *index* (*mtf2 n q xs*) (*xs* ! *i*) ∧ *index* (*mtf2 n q xs*) (*xs* ! *i*) < *length xs*) **by** *auto*
with *assms mtf2_forward_effect1* **have** *1*: ~ (*index xs q* < *i* ∧ *i* < *length xs*) **by** *metis*
from *assms*(*4*) **have** ~ (*index xs q* − *n* < *index* (*mtf2 n q xs*) (*xs* ! *i*) ∧ *index* (*mtf2 n q xs*) (*xs* ! *i*) ≤ *index xs q*) **by** *auto*
with *assms mtf2_forward_effect3* **have** *2*: ~ (*index xs q* − *n* ≤ *i* ∧ *i* < *index xs q*) **by** *metis*
from *assms*(*4*) **have** ~ (*index* (*mtf2 n q xs*) (*xs* ! *i*) < *index xs q* − *n*) **by** *auto*
with *assms mtf2_forward_effect4* **have** *3*: ~ (*i* < *index xs q* − *n*) **by** *metis*

from *fullchar*[*OF assms*(*1*)] *assms*(*5*) *1 2 3* **show** *index xs q* = *i* **by** *metis*
qed

lemma *mtf2_backwards_effect3*:
assumes *index xs q* < *length xs q* ∈ *set xs distinct xs*
index xs q − *n* < *index* (*mtf2 n q xs*) (*xs* ! *i*) ∧ *index* (*mtf2 n q xs*) (*xs* ! *i*) ≤ *index xs q*
i < *length xs*
shows *index xs q* − *n* ≤ *i* ∧ *i* < *index xs q*
proof −
from *assms*(*4*) **have** ~ (*index xs q* < *index* (*mtf2 n q xs*) (*xs* ! *i*) ∧ *index* (*mtf2 n q xs*) (*xs* ! *i*) < *length xs*) **by** *auto*
with *assms mtf2_forward_effect1* **have** *2*: ~ (*index xs q* < *i* ∧ *i* < *length xs*) **by** *metis*
from *assms*(*4*) **have** ~ (*index xs q* − *n* = *index* (*mtf2 n q xs*) (*xs* ! *i*)) **by** *auto*
with *assms mtf2_forward_effect2* **have** *1*: ~ (*index xs q* = *i*) **by** *metis*
from *assms*(*4*) **have** ~ (*index* (*mtf2 n q xs*) (*xs* ! *i*) < *index xs q* − *n*) **by** *auto*
with *assms mtf2_forward_effect4* **have** *3*: ~ (*i* < *index xs q* − *n*) **by**

metis

from *fullchar*[*OF assms*(*1*)] *assms*(*5*) *1 2 3* **show** *index xs q* $-$ *n* $\leq$ *i* $\wedge$ *i* $<$ *index xs q* **by** *metis*
qed

lemma *mtf2_backwards_effect4*:
assumes *index xs q* $<$ *length xs q* $\in$ *set xs distinct xs*
index (*mtf2 n q xs*) (*xs* ! *i*) $<$ *index xs q* $-$ *n*
i $<$ *length xs*
shows *i* $<$ *index xs q* $-$ *n*
proof $-$
from *assms*(*4*) **have** ~ (*index xs q* $<$ *index* (*mtf2 n q xs*) (*xs* ! *i*) $\wedge$ *index* (*mtf2 n q xs*) (*xs* ! *i*) $<$ *length xs*) **by** *auto*
with *assms mtf2_forward_effect1* **have** *2*: ~ (*index xs q* $<$ *i* $\wedge$ *i* $<$ *length xs*) **by** *metis*
from *assms*(*4*) **have** ~ (*index xs q* $-$ *n* $=$ *index* (*mtf2 n q xs*) (*xs* ! *i*)) **by** *auto*
with *assms mtf2_forward_effect2* **have** *1*: ~ (*index xs q* $=$ *i*) **by** *metis*
from *assms*(*4*) **have** ~ (*index xs q* $-$ *n* $<$ *index* (*mtf2 n q xs*) (*xs* ! *i*) $\wedge$ *index* (*mtf2 n q xs*) (*xs* ! *i*) $\leq$ *index xs q*) **by** *auto*
with *assms mtf2_forward_effect3* **have** *3*: ~ (*index xs q* $-$ *n* $\leq$ *i* $\wedge$ *i* $<$ *index xs q*) **by** *metis*

from *fullchar*[*OF assms*(*1*)] *assms*(*5*) *1 2 3* **show** *i* $<$ *index xs q* $-$ *n* **by** *metis*
qed

lemma *mtf2_backwards_effect4'*:
assumes *index xs q* $<$ *length xs q* $\in$ *set xs distinct xs*
index (*mtf2 n q xs*) *x* $<$ *index xs q* $-$ *n*
x $\in$ *set xs*
shows (*index xs x*) $<$ *index xs q* $-$ *n*
using *assms mtf2_backwards_effect4*[**where** *xs*=*xs* **and** *i*=*index xs x*] *yes* **by** *auto*

lemma
assumes *distA*: *distinct A* **and**
asm: *q* $\in$ *set A*
shows
mtf2_mono: *q*$<$ *x in A* $\Longrightarrow$ *q* $<$ *x in* (*mtf2 n q A*) **and**
mtf2_q_after: *index* (*mtf2 n q A*) *q* $=$ *index A q* $-$ *n*
proof $-$

have *lele*: $(q < x \ in \ A \longrightarrow q < x \ in \ swaps \ [index \ A \ q - n..<index \ A \ q] \ A) \wedge (index \ (swaps \ [index \ A \ q - n..<index \ A \ q] \ A) \ q = \ index \ A \ q - n)$
apply(*induct n*) **apply**(*simp*)
proof –
fix *n*
assume *ind*: $(q < x \ in \ A \longrightarrow q < x \ in \ swaps \ [index \ A \ q - n..<index \ A \ q] \ A)$
$\wedge \ index \ (swaps \ [index \ A \ q - n..<index \ A \ q] \ A) \ q = \ index \ A \ q - n$
then have *iH*: $q < x \ in \ A \Longrightarrow q < x \ in \ swaps \ [index \ A \ q - n..<index \ A \ q] \ A$ **by** *auto*
from *ind* **have** *indH2*: $index \ (swaps \ [index \ A \ q - n..<index \ A \ q] \ A) \ q = \ index \ A \ q - n$ **by** *auto*

show $(q < x \ in \ A \longrightarrow q < x \ in \ swaps \ [index \ A \ q - Suc \ n..<index \ A \ q] \ A) \wedge$
$index \ (swaps \ [index \ A \ q - Suc \ n..<index \ A \ q] \ A) \ q = index \ A \ q - Suc \ n$ (**is** *?part1* $\wedge$ *?part2*)
proof (*cases index A q* $\geq$ *Suc n*)
case *True*
then have *onemore*: $[index \ A \ q - Suc \ n..<index \ A \ q] = (index \ A \ q - Suc \ n) \ \# \ [index \ A \ q - n..<index \ A \ q]$
using *Suc_diff_Suc upt_rec* **by** *auto*

from *onemore* **have** *yeah*: $swaps \ [index \ A \ q - Suc \ n..<index \ A \ q] \ A$
$= swap \ (index \ A \ q - Suc \ n) \ (swaps \ \ [index \ A \ q - n..<index \ A \ q] \ A)$ **by** *auto*

from *indH2* **have** *gr*: $index \ (swaps \ [index \ A \ q - n..<index \ A \ q] \ A) \ q = \ Suc(index \ A \ q - Suc \ n)$ **using** *Suc_diff_Suc True* **by** *auto*
have *whereisq*: $swaps \ [index \ A \ q - n..<index \ A \ q] \ A \ ! \ Suc \ (index \ A \ q - Suc \ n) = q$
unfolding *gr*[*symmetric*] **apply**(*rule nth_index*) **using** *asm* **by** *auto*

have *indSi*: $index \ A \ q < length \ A$ **using** *asm index_less* **by** *auto*
have *3*: $Suc \ (index \ A \ q - Suc \ n) < length \ (swaps \ [index \ A \ q - n..<index \ A \ q] \ A)$ **using** *True*
apply(*auto simp*: *Suc_diff_Suc asm*) **using** *indSi* **by** *auto*
have *1*: $q \neq swaps \ [index \ A \ q - n..<index \ A \ q] \ A \ ! \ (index \ A \ q - Suc \ n)$

proof
assume *as*: *q* = *swaps* [*index A q* − *n*..<*index A q*] *A* ! (*index A q* − *Suc n*)
{
fix *xs x*
have *Suc x* < *length xs* ⟹ *xs* ! *x* = *q* ⟹ *xs* ! *Suc x* = *q* ⟹ ¬ *distinct xs*
by (*metis Suc_lessD index_nth_id n_not_Suc_n*)
} **note** *cool*=*this*

have ¬ *distinct* (*swaps* [*index A q* − *n*..<*index A q*] *A*)
apply(*rule cool*[*of* (*index A q* − *Suc n*)])
apply(*simp only*: *3*)
apply(*simp only*: *as*[*symmetric*])
by(*simp only*: *whereisq*)
then show *False* **using** *distA* **by** *auto*
qed

have *part1*: *?part1*
proof
assume *qx*: *q* < *x in A*
{
fix *q x B i*
assume *a1*: *q* < *x in B*
assume *a2*: ~ *q* = *B* ! *i*
assume *a3*: *distinct B*
assume *a4*: *Suc i* < *length B*

have *dist_perm B B* **by**(*simp add*: *a3*)
moreover have *Suc i* < *length B* **using** *a4* **by** *auto*
moreover have *q* < *x in B* ∧ ¬ (*q* = *B* ! *i* ∧ *x* = *B* ! *Suc i*) **using** *a1 a2* **by** *auto*
ultimately have *q* < *x in swap i B*
using *before_in_swap*[*of B B*] **by** *simp*
} **note** *grr*=*this*

have *2*: *distinct* (*swaps* [*index A q* − *n*..<*index A q*] *A*) **using** *distA* **by** *auto*

show *q* < *x in swaps* [*index A q* − *Suc n*..<*index A q*] *A*
apply(*simp only*: *yeah*)
apply(*rule grr*[*OF iH*[*OF qx*]]) **using** *1 2 3* **by** *auto*
qed

let ?xs = (swaps [index A q − n..<index A q] A)
let ?n = (index A q − Suc n)
have ?xs ! Suc ?n = swaps [index A q − n..<index A q] A ! (index (swaps [index A q − n..<index A q] A) q)
using indH2 Suc_diff_Suc True **by** auto
also have ... = q **apply**(rule nth_index) **using** asm **by** auto
finally have sndTrue: ?xs ! Suc ?n = q **.**
have fstFalse: ~ q = ?xs ! ?n **by** (fact 1)

have index (swaps [index A q − Suc n..<index A q] A) q
= index (swap (index A q − Suc n) ?xs) q **by** (simp only: yeah)
also have ... = (if q = ?xs ! ?n then Suc ?n else if q = ?xs ! Suc ?n then ?n else index ?xs q)
apply(rule index_swap_distinct)
apply(simp add: distA)
by (fact 3)
also have ... = ?n **using** fstFalse sndTrue **by** auto
finally have part2: ?part2 **.**

from part1 part2 **show** ?part1 ∧ ?part2 **by** simp
next
case False
then have a: index A q − Suc n = index A q − n **by** auto
then have b: [index A q − Suc n..<index A q] = [index A q − n..<index A q] **by** auto
show ?thesis **apply**(simp only: b a) **by** (fact ind)
qed
qed

show q < x in A ⟹ q < x in (mtf2 n q A)
(index (mtf2 n q A) q) = index A q − n
unfolding mtf2_def
using asm lele **apply**(simp)
using asm lele **by**(simp)
qed

8.1 effect of mtf2 on index

lemma swapsthrough: distinct xs ⟹ q ∈ set xs ⟹ index (swaps [index xs q − entf..<index xs q] xs) q = index xs q − entf
proof (induct entf)

case (*Suc e*)
note *iH=this*
show *?case*
proof (*cases index xs q − e*)
case *0*
then have [*index xs q − Suc e..<index xs q*]
= [*index xs q − e..<index xs q*] **by** *force*
then have *index* (*swaps* [*index xs q − Suc e..<index xs q*] *xs*) *q*
= *index xs q − e* **using** *Suc* **by** *auto*
also have ... = *index xs q −* (*Suc e*) **using** *0* **by** *auto*
finally show *index* (*swaps* [*index xs q − Suc e..<index xs q*] *xs*) *q* = *index xs q − Suc e* **.**
next
case (*Suc f*)

have *gaa*: *Suc* (*index xs q − Suc e*) = *index xs q − e* **using** *Suc* **by** *auto*

from *Suc* **have** *index xs q − e ≤ index xs q* **by** *auto*
also have ... < *length xs* **by**(*simp add*: *index_less_size_conv iH*)
finally have *indle*: *index xs q − e < length xs***.**

have *arg*: *Suc* (*index xs q − Suc e*) < *length* (*swaps* [*index xs q − e..<index xs q*] *xs*)
apply(*auto*) **unfolding** *gaa* **using** *indle* **by** *simp*
then have *arg2*: *index xs q − Suc e < length* (*swaps* [*index xs q − e..<index xs q*] *xs*) **by** *auto*
from *Suc* **have** *nexter*: *index xs q − e = Suc* (*index xs q −* (*Suc e*)) **by** *auto*
then have *aaa*: [*index xs q − Suc e..<index xs q*]
= (*index xs q − Suc e*)#[*index xs q − e..<index xs q*] **using** *upt_rec* **by** *auto*

let *?i=index xs q − Suc e*
let *?rest=swaps* [*index xs q − e..<index xs q*] *xs*
from *iH nexter* **have** *indj*: *index ?rest q = Suc ?i* **by** *auto*

from *iH*(*2*) **have** *distinct ?rest* **by** *auto*

have *?rest* ! (*index ?rest q*) = *q* **apply**(*rule nth_index*) **by**(*simp add*: *iH*)
with *indj* **have** *whichcase*: *q = ?rest* ! *Suc ?i* **by** *auto*

with ‹distinct ?rest› **have** whichcase2: ~ q = ?rest ! ?i
by (metis Suc_lessD arg index_nth_id n_not_Suc_n)

from aaa **have** index (swaps [index xs q − Suc e..<index xs q] xs) q
= index (swap (index xs q − Suc e) (swaps [index xs q − e..<index xs q] xs)) q
by auto
also have ... = (if q = ?rest ! ?i then (Suc ?i) else if q = ?rest ! (Suc ?i) then ?i else index ?rest q)
apply(simp only: swap_def arg if_True)
apply(rule index_swap_if_distinct)
apply(simp add: iH)
apply(simp only: arg2)
by(simp only: arg)
also have ... = ?i **using** whichcase whichcase2 **by** simp
finally show index (swaps [index xs q − Suc e..<index xs q] xs) q =
index xs q − Suc e **.**
qed
next
case 0
show ?case **by** simp
qed

term mtf2
lemma mtf2_moves_to_front: distinct xs ⟹ q ∈ set xs ⟹ index (mtf2 (length xs) q xs) q = 0
unfolding mtf2_def
proof −
assume distxs: distinct xs
assume qinxs: q ∈ set xs
have index (if q ∈ set xs then swaps [index xs q − length xs..<index xs q] xs else xs) q
= index (swaps [index xs q − length xs..<index xs q] xs) q **using** qinxs **by** auto
also have ... = index xs q − (length xs) **apply**(rule swapsthrough) **using** distxs qinxs **by** auto
also have ... = 0 **using** index_less_size_conv qinxs **by** (simp add: index_le_size)
finally show index (if q ∈ set xs then swaps [index xs q − length xs..<index xs q] xs else xs) q = 0 **.**
qed

lemma *xy_relativorder_mtf2*:
  assumes
    q≠x q≠y distinct xs x∈set xs y∈set xs q∈set xs
  shows *x < y in mtf2 n q xs*
          = x < y in xs
using *assms*
by (*metis before_in_setD2 not_before_in x_stays_before_y_if_y_not_moved_to_front*)

lemma *mtf2_moves_to_frontm1*: *distinct xs ⟹ q ∈ set xs ⟹ index* (*mtf2* (*length xs −1*) *q xs*) *q = 0*
unfolding *mtf2_def*
proof −
  assume *distxs*: *distinct xs*
  assume *qinxs*: *q ∈ set xs*
  have *index* (*if q ∈ set xs then swaps* [*index xs q −* (*length xs −1*)..<*index xs q*] *xs else xs*) *q*
    = index (*swaps* [*index xs q −* (*length xs −1*)..<*index xs q*] *xs*) *q* **using** *qinxs* **by** *auto*
  also have ... *= index xs q −* (*length xs −1*) **apply**(*rule swapsthrough*) **using** *distxs qinxs* **by** *auto*
  also have ... *= 0* **using** *index_less_size_conv qinxs*
by (*metis Suc_pred' gr0I length_pos_if_in_set less_irrefl less_trans_Suc zero_less_diff*)
  finally show *index* (*if q ∈ set xs then swaps* [*index xs q −* (*length xs −1*)..<*index xs q*] *xs else xs*) *q = 0* **.**
qed

lemma *mtf2_moves_to_front'*: *distinct xs ⟹ y ∈ set xs ⟹ x ∈ set xs ⟹ x≠y ⟹ x < y in mtf2* (*length xs−1*) *x xs = True*
using *mtf2_moves_to_frontm1* **by** (*metis before_in_def gr0I index_eq_index_conv set_mtf2*)

lemma *mtf2_moves_to_front''*: *distinct xs ⟹ y ∈ set xs ⟹ x ∈ set xs ⟹ x≠y ⟹ x < y in mtf2* (*length xs*) *x xs = True*
using *mtf2_moves_to_front* **by** (*metis before_in_def gr0I index_eq_index_conv set_mtf2*)

end

9 BIT: an Online Algorithm for the List Update Problem

```
theory BIT
imports
  Bit_Strings
  MTF2_Effects
begin


abbreviation config'' A qs init n == config_rand A init (take n qs)


lemma sum_my: fixes f g::'b ⇒ 'a::ab_group_add
    assumes finite A finite B
  shows (∑x∈A. f x) − (∑x∈B. g x)
    = (∑x∈(A ∩ B). f x − g x) + (∑x∈A−B. f x) − (∑x∈B−A. g x)
proof −
  have finite (A−B) and finite (A∩B) and finite (B−A) and finite (B∩A)
using assms by auto
  note finites=this
  have (A−B) ∩ ( (A∩B)  ) = {} and (B−A) ∩ ( (B∩A)  ) = {}  by auto
  note inters=this
  have commute: A∩B=B∩A by auto
  have A = (A−B) ∪ (A∩B) and B = (B−A) ∪ ( (B∩A))  by auto
  then have (∑x∈A. f x) − (∑x∈B. g x) = (∑x∈(A−B) ∪ (A∩B). f x)
− (∑x∈(B−A) ∪ (B∩A). g x) by auto
 also have ... = ( (∑x∈(A−B). f x) + (∑x∈(A∩B). f x) − (∑x∈(A−B)∩(A∩B).
f x) )
      −( (∑x∈(B−A). g x) + (∑x∈(B∩A). g x) − (∑x∈(B−A)∩(B∩A).
g x))
        using sum_Un[where ?f=f,OF finites(1) finites(2)]
              sum_Un[where ?f=g,OF finites(3) finites(4)] by(simp)
  also have ... = ( (∑x∈(A−B). f x) + (∑x∈(A∩B). f x) )
      − (∑x∈(B−A). g x) − (∑x∈(B∩A). g x)  using inters by auto
 also have ... =  (∑x∈(A−B). f x) − (∑x∈(A∩B). g x) + (∑x∈(A∩B).
f x)
      − (∑x∈(B−A). g x)   using commute by auto
  also have ... = (∑x∈(A∩B). f x − g x) +(∑x∈(A−B). f x)
      − (∑x∈(B−A). g x) using sum_subtractf[of f g (A∩B)] by auto
  finally show ?thesis .
qed
```

lemma *sum_my2*: $(\forall x{\in}A.\ f\ x = g\ x) \Longrightarrow (\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ g\ x)$
by *auto*

9.1 Definition of BIT

definition *BIT_init* :: (*'a state*,*bool list* ∗ *'a list*)*alg_on_init* **where**
BIT_init init = *map_pmf* (*λl.* (*l*,*init*)) (*bv* (*length init*))

lemma ~ *deterministic_init BIT_init*
unfolding *deterministic_init_def BIT_init_def* **apply**(*auto*)
apply(*intro exI*[**where** *x*=[*a*]])
— comment in a proof
by(*auto simp*: *UNIV_bool set_pmf_bernoulli*)

definition *BIT_step* :: (*'a state*, *bool list* ∗ *'a list*, *'a*, *answer*)*alg_on_step* **where**
BIT_step s q = (*let a*=((*if* (*fst* (*snd s*))!(*index* (*snd* (*snd s*)) *q*) *then 0 else* (*length* (*fst s*))),[]) *in*
return_pmf (*a* , (*flip* (*index* (*snd* (*snd s*)) *q*) (*fst* (*snd s*)), *snd* (*snd s*))))

lemma *deterministic_step BIT_step*
unfolding *deterministic_step_def BIT_step_def*
by *simp*

abbreviation *BIT* :: (*'a state*, *bool list*∗*'a list*, *'a*, *answer*)*alg_on_rand* **where**
BIT == (*BIT_init*, *BIT_step*)

9.2 Properties of BIT's state distribution

lemma *BIT_no_paid*: ∀ ((*free*,*paid*),_) ∈ (*BIT_step s q*). *paid*=[]
unfolding *BIT_step_def*
by(*auto*)

9.2.1 About the Internal State

term (*config'_rand* (*BIT_init*, *BIT_step*) *s0 qs*)
lemma *config'_n_init*: **fixes** *qs init n*
shows *map_pmf* (*snd* ∘ *snd*) (*config'_rand* (*BIT_init*, *BIT_step*) *init qs*) = *map_pmf* (*snd* ∘ *snd*) *init*

apply (*induct qs arbitrary*: *init*)
by (*simp_all add*: *map_pmf_def bind_assoc_pmf BIT_step_def bind_return_pmf*)

lemma *config_n_init*: *map_pmf* (*snd* ∘ *snd*) (*config_rand* (*BIT_init*, *BIT_step*) *s0 qs*) = *return_pmf s0*
using *config′_n_init*[*of* ((*fst* (*BIT_init*, *BIT_step*) *s0*) ⤜ (*λis*. *return_pmf* (*s0*, *is*)))]
by (*simp_all add*: *map_pmf_def bind_assoc_pmf bind_return_pmf BIT_init_def*)

lemma *config_n_init2*: ∀ (_,(_,*x*)) ∈ *set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *init qs*). *x* = *init*
proof (*rule*, *goal_cases*)
case (*1 z*)
then have *1*: *snd*(*snd z*) ∈ (*snd* ∘ *snd*) ‘ *set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *init qs*)
by *force*
have (*snd* ∘ *snd*) ‘ *set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *init qs*)
= *set_pmf* (*map_pmf* (*snd* ∘ *snd*) (*config_rand* (*BIT_init*, *BIT_step*) *init qs*)) **by**(*simp*)
also have ... = {*init*} **apply**(*simp only*: *config_n_init*) **by** *simp*
finally have *snd*(*snd z*) = *init* **using** *1* **by** *auto*
then show *?case* **by** *auto*
qed
lemma *config_n_init3*: ∀ *x* ∈ *set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *init qs*). *snd* (*snd x*) = *init*
using *config_n_init2* **by**(*simp add*: *split_def*)

lemma *config′_n_bv*: **fixes** *qs init n*
shows *map_pmf* (*snd* ∘ *snd*) *init* = *return_pmf s0*
⟹ *map_pmf* (*fst* ∘ *snd*) *init* = *bv* (*length s0*)
⟹ *map_pmf* (*snd* ∘ *snd*) (*config′_rand* (*BIT_init*, *BIT_step*) *init qs*) = *return_pmf s0*
∧ *map_pmf* (*fst* ∘ *snd*) (*config′_rand* (*BIT_init*, *BIT_step*) *init qs*) = *bv* (*length s0*)
proof (*induct qs arbitrary*: *init*)
case (*Cons r rs*)
from *Cons*(*2*) **have** *a*: *map_pmf* (*snd* ∘ *snd*) (*init* ⤜ (*λs*. *snd* (*BIT_init*, *BIT_step*) *s r* ⤜
(*λ*(*a*, *is′*). *return_pmf* (*step* (*fst s*) *r a*, *is′*))))

= *return_pmf s0* **apply**(*simp add*: *BIT_step_def*)
by (*simp_all add*: *map_pmf_def bind_assoc_pmf BIT_step_def bind_return_pmf*)
then have *b*: ∀ *z*∈*set_pmf* (*init* ≫= (λ*s*. *snd* (*BIT_init*, *BIT_step*) *s r* ≫=
(λ(*a*, *is*′). *return_pmf* (*step* (*fst s*) *r a*, *is*′)))). *snd* (*snd z*) = *s0*
by (*metis* (*mono_tags*, *lifting*) *comp_eq_dest_lhs map_pmf_eq_return_pmf_iff*)

show *?case*
apply(*simp only*: *config′_rand.simps*)
proof (*rule Cons*(*1*), *goal_cases*)
case *2*
have *map_pmf* (*fst* ∘ *snd*)
(*init* ≫=
(λ*s*. *snd* (*BIT_init*, *BIT_step*) *s r* ≫=
(λ(*a*, *is*′).
return_pmf (*step* (*fst s*) *r a*, *is*′)))) = *map_pmf* (*flip* (*index s0 r*)) (*bv* (*length s0*))
using *b*
apply(*simp add*: *BIT_step_def Cons*(*3*)[*symmetric*] *bind_return_pmf map_pmf_def bind_assoc_pmf*)
apply(*rule bind_pmf_cong*)
apply(*simp*)
by(*simp add*: *inv_flip_bv*)
also have ... = *bv* (*length s0*) **using** *inv_flip_bv* **by** *auto*
finally show *?case* **.**
qed (*fact*)
qed *simp*

lemma *config_n_bv_2*: *map_pmf* (*snd* ∘ *snd*) (*config_rand* (*BIT_init*, *BIT_step*) *s0 qs*) = *return_pmf s0*
∧ *map_pmf* (*fst* ∘ *snd*) (*config_rand* (*BIT_init*, *BIT_step*) *s0 qs*) = *bv* (*length s0*)
apply(*rule config′_n_bv*)
by(*simp_all add*: *bind_return_pmf map_pmf_def bind_assoc_pmf bind_return_pmf′ BIT_init_def*)

lemma *config_n_bv*: *map_pmf* (*fst* ∘ *snd*) (*config_rand* (*BIT_init*, *BIT_step*) *s0 qs*) = *bv* (*length s0*)
using *config_n_bv_2* **by** *auto*

lemma *config_n_fst_init_length*: ∀ (_,(x,_)) ∈ *set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *s0 qs*). *length x* = *length s0*
proof
  fix *x*::(*'a list* × (*bool list* × *'a list*))
  assume *ass*:*x* ∈ *set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *s0 qs*)
  let *?a*=*fst* (*snd x*)
 from *ass* **have** (*fst x*,(*?a*,*snd* (*snd x*))) ∈ *set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *s0 qs*) **by** *auto*
   with *ass* **have** *?a* ∈ (*fst* ∘ *snd*) ‘ *set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *s0 qs*) **by** *force*
  then have *?a* ∈ *set_pmf* (*map_pmf* (*fst* ∘ *snd*) (*config_rand* (*BIT_init*, *BIT_step*) *s0 qs*)) **by** *auto*
  then have *?a* ∈ *bv* (*length s0*) **by**(*simp only*: *config_n_bv*)
  then have *length ?a* = *length s0* **by** (*auto simp*: *len_bv_n*)
  then show *case x of* (*uu_*, *xa*, *uua_*) ⇒ *length xa* = *length s0* **by**(*simp add*: *split_def*)
qed

lemma *config_n_fst_init_length2*: ∀ *x* ∈ *set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *s0 qs*). *length* (*fst* (*snd x*)) = *length s0*
using *config_n_fst_init_length* **by**(*simp add*: *split_def*)

lemma *fperms*: *finite* {*x*::*'a list*. *length x* = *length init* ∧ *distinct x* ∧ *set x* = *set init*}
apply(*rule finite_subset*[**where** *B*={*xs*. *set xs* ⊆ *set init* ∧ *length xs* ≤ *length init*}])
apply(*force*) **apply**(*rule finite_lists_length_le*) **by** *auto*

lemma *finite_config_BIT*: **assumes** [*simp*]: *distinct init*
  shows *finite* (*set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *init qs*)) (**is** *finite ?D*)
proof −
   have *a*: (*fst* ∘ *snd*) ‘ *?D* ⊆ {*x*. *length x* = *length init*} **using** *config_n_fst_init_length2* **by** *force*
  have *c*: (*snd* ∘ *snd*) ‘ *?D* = {*init*}
  proof −
    have (*snd* ∘ *snd*) ‘ *set_pmf* (*config_rand* (*BIT_init*, *BIT_step*) *init qs*)
              = *set_pmf* (*map_pmf* (*snd* ∘ *snd*) (*config_rand* (*BIT_init*, *BIT_step*) *init qs*)) **by**(*simp*)
    also have . . . = {*init*} **apply**(*subst config_n_init*) **by** *simp*

finally show *?thesis* **.**
qed
from *a c* **have** *d*: *snd ‘ ?D* ⊆ {*x. length x* = *length init*} × {*init*} **by** *force*
have *b*: *fst ‘ ?D* ⊆ {*x. length x* = *length init* ∧ *distinct x* ∧ *set x* = *set init*}
using *config_rand* **by** *fastforce*

from *b d* **have** *?D* ⊆ {*x. length x* = *length init* ∧ *distinct x* ∧ *set x* = *set init*} × ({*x. length x* = *length init*} × {*init*})
by *auto*
then show *?thesis*
apply (*rule finite_subset*)
apply(*rule finite_cartesian_product*)
apply(*rule fperms*)
apply(*rule finite_cartesian_product*)
apply (*rule bitstrings_finite*)
by(*simp*)
qed

9.3 BIT is 1.75-competitive (a combinatorial proof)

9.3.1 Definition of the Locale and Helper Functions

locale *BIT_Off* =
fixes *acts* :: *answer list*
fixes *qs* :: *'a list*
fixes *init* :: *'a list*
assumes *dist_init*[*simp*]: *distinct init*
assumes *len_acts*: *length acts* = *length qs*
begin

lemma *setinit*: (*index init*) *‘ set init* = {*0..<length init*}
using *dist_init*
proof(*induct init*)
case (*Cons a as*)
with *Cons* **have** *iH*: *index as ‘ set as* = {*0..<length as*} **by** *auto*
from *Cons* **have** *1*:(*set as* ∩ {*x.* (*a* ≠ *x*)}) = *set as* **by** *fastforce*
have *2*: (λ*a. Suc* (*index as a*)) *‘ set as* =
(λ*a. Suc a*) *‘* ((*index as*) *‘ set as*) **by** *auto*
show *?case*
apply(*simp add*: *1 2 iH*) **by** *auto*
qed *simp*

definition *free_A* :: *nat list* **where**
free_A = *map fst acts*

definition *paid_A'* :: *nat list list* **where**
paid_A' = *map snd acts*

definition *paid_A* :: *nat list list* **where**
paid_A = *map* (*filter* (λ*x*. *Suc x* < *length init*)) *paid_A'*

lemma *len_paid_A*[*simp*]: *length paid_A* = *length qs*
unfolding *paid_A_def paid_A'_def* **using** *len_acts* **by** *auto*
lemma *len_paid_A'*[*simp*]: *length paid_A'* = *length qs*
unfolding *paid_A'_def* **using** *len_acts* **by** *auto*

lemma *paidAnm_inbound*: *n* < *length paid_A* ⟹ *m* < *length*(*paid_A*!*n*)
⟹ (*Suc* ((*paid_A*!*n*)!(*length* (*paid_A* ! *n*) − *Suc m*))) < *length init*
proof −
  assume *n* < *length paid_A*
  then have *n* < *length paid_A'* **by** *auto*
  then have *a*: (*paid_A*!*n*)
    = *filter* (λ*x*. *Suc x* < *length init*) (*paid_A'* ! *n*) **unfolding** *paid_A_def*
by *auto*

  let *?filtered*=(*filter* (λ*x*. *Suc x* < *length init*) (*paid_A'* ! *n*))
  assume *mtt*: *m* < *length* (*paid_A*!*n*)
  with *a* **have** (*length* (*paid_A* ! *n*) − *Suc m*) < *length ?filtered* **by** *auto*
  with *nth_mem* **have** *b*: *Suc*(*?filtered* ! (*length* (*paid_A* ! *n*) − *Suc m*))
< *length init* **by** *force*

  show *Suc* (*paid_A* ! *n* ! (*length* (*paid_A* ! *n*) − *Suc m*)) < *length init*
using *a b* **by** *auto*
qed

fun *s_A'* :: *nat* ⇒ *'a list* **where**
s_A' 0 = *init* |
s_A'(*Suc n*) = *step* (*s_A' n*) (*qs*!*n*) (*free_A*!*n*, *paid_A'*!*n*)

lemma *length_s_A'*[*simp*]: *length*(*s_A' n*) = *length init*
by (*induction n*) *simp_all*

lemma *dist_s_A'*[*simp*]: *distinct*(*s_A' n*)
by(*induction n*) (*simp_all add*: *step_def*)

lemma *set_s_A'*[*simp*]: *set*(*s_A' n*) = *set init*
by(*induction n*) (*simp_all add*: *step_def*)

fun *s_A* :: *nat* ⇒ *'a list* **where**
s_A 0 = *init* |
s_A(*Suc n*) = *step* (*s_A n*) (*qs*!*n*) (*free_A*!*n*, *paid_A*!*n*)

lemma *length_s_A*[*simp*]: *length*(*s_A n*) = *length init*
by (*induction n*) *simp_all*

lemma *dist_s_A*[*simp*]: *distinct*(*s_A n*)
by(*induction n*) (*simp_all add*: *step_def*)

lemma *set_s_A*[*simp*]: *set*(*s_A n*) = *set init*
by(*induction n*) (*simp_all add*: *step_def*)

lemma *cost_paidAA'*: *n* < *length paid_A'* ⟹ *length* (*paid_A*!*n*) ≤ *length* (*paid_A'*!*n*)
  unfolding *paid_A_def* **by** *simp*

lemma *swaps_filtered*: *swaps* (*filter* (λ*x*. *Suc x* < *length xs*) *ys*) *xs* = *swaps* (*ys*) *xs*
apply (*induct ys*) **by** *auto*

lemma *sAsA'*: *n* < *length paid_A'* ⟹ *s_A' n* = *s_A n*
proof (*induct n*)
  case (*Suc m*)
  have *s_A'* (*Suc m*)
    = *mtf2* (*free_A*!*m*) (*qs*!*m*) (*swaps* (*paid_A'*!*m*) (*s_A' m*)) **by** (*simp add*: *step_def*)
  also from *Suc*(*2*) **have** ... = *mtf2* (*free_A*!*m*) (*qs*!*m*) (*swaps* (*paid_A*!*m*) (*s_A' m*))
    unfolding *paid_A_def*
    by (*simp only*: *nth_map swaps_filtered*[**where** *xs*=*s_A' m*, *simplified*])
  also have ... = *mtf2* (*free_A*!*m*) (*qs*!*m*) (*swaps* (*paid_A*!*m*) (*s_A m*)) **using** *Suc* **by** *auto*
  also have ... = *s_A* (*Suc m*) **by** (*simp add*: *step_def*)
  finally show *?case* **.**
qed *simp*

lemma *sAsA''*: *n* < *length qs* ⟹ *s_A n* = *s_A' n*
using *sAsA'* **by** *auto*

definition t_BIT :: $nat \Rightarrow real$ **where**
$t\_BIT\ n = T\_on\_rand\_n\ BIT\ init\ qs\ n$

definition T_BIT :: $nat \Rightarrow real$ **where**
$T\_BIT\ n = (\sum i{<}n.\ t\_BIT\ i)$

definition c_A :: $nat \Rightarrow int$ **where**
$c\_A\ n = index\ (swaps\ (paid\_A!n)\ (s\_A\ n))\ (qs!n) + 1$

definition f_A :: $nat \Rightarrow int$ **where**
$f\_A\ n = min\ (free\_A!n)\ (index\ (swaps\ (paid\_A!n)\ (s\_A\ n))\ (qs!n))$

definition p_A :: $nat \Rightarrow int$ **where**
$p\_A\ n = size(paid\_A!n)$

definition t_A :: $nat \Rightarrow int$ **where**
$t\_A\ n = c\_A\ n + p\_A\ n$

definition c_A' :: $nat \Rightarrow int$ **where**
$c\_A'\ n = index\ (swaps\ (paid\_A'!n)\ (s\_A'\ n))\ (qs!n) + 1$

definition p_A' :: $nat \Rightarrow int$ **where**
$p\_A'\ n = size(paid\_A'!n)$
definition t_A' :: $nat \Rightarrow int$ **where**
$t\_A'\ n = c\_A'\ n + p\_A'\ n$

lemma $t\_A\_A'\_leq$: $n < length\ paid\_A' \Longrightarrow t\_A\ n \leq t\_A'\ n$
unfolding *t_A_def t_A'_def c_A_def c_A'_def p_A_def p_A'_def*
  apply(*simp add*: *sAsA'*)
  unfolding *paid_A_def*
  by (*simp add*: *swaps_filtered*[**where** *xs*=(*s_A n*), *simplified*])

definition T_A' :: $nat \Rightarrow int$ **where**
$T\_A'\ n = (\sum i{<}n.\ t\_A'\ i)$

definition T_A :: $nat \Rightarrow int$ **where**
$T\_A\ n = (\sum i{<}n.\ t\_A\ i)$

lemma $T\_A\_A'\_leq$: $n \leq length\ paid\_A' \Longrightarrow T\_A\ n \leq T\_A'\ n$

unfolding *T_A′_def T_A_def* **apply**(*rule sum_mono*)
by (*simp add*: *t_A_A′_leq*)

lemma *T_A_A′_leq′*: $n \leq$ *length qs* $\Longrightarrow$ *T_A n* $\leq$ *T_A′ n*
using *T_A_A′_leq* **by** *auto*

fun *s′_A* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *′a list* **where**
s′_A n 0 = *s_A n*
| (*s′_A n* (*Suc m*)) = *swap* ((*paid_A* ! *n*)!(*length* (*paid_A* ! *n*) −(*Suc m*))
) (*s′_A n m*)

lemma *set_s′_A*[*simp*]: *set* (*s′_A n m*) = *set init*
apply(*induct m*) **by**(*auto*)

lemma *len_s′_A*[*simp*]: *length* (*s′_A n m*) = *length init*
apply(*induct m*) **by**(*auto*)

lemma *distperm_s′_A*[*simp*]: *dist_perm* (*s′_A n m*) *init*
apply(*induct m*) **by** *auto*

lemma *s′A_m_le*: $m \leq$ (*length* (*paid_A* ! *n*)) $\Longrightarrow$ *swaps* (*drop* (*length*
(*paid_A* ! *n*) − *m*) (*paid_A* ! *n*)) (*s_A n*) = *s′_A n m*
apply(*induct m*)
apply(*simp*)
proof −
  fix *m*
  assume *iH*: ($m \leq$ *length* (*paid_A* ! *n*) $\Longrightarrow$ *swaps* (*drop* (*length* (*paid_A*
! *n*) − *m*) (*paid_A* ! *n*)) (*s_A n*) = *s′_A n m*)
  assume *Suc*: *Suc m* $\leq$ *length* (*paid_A* ! *n*)
  then have $m \leq$ *length* (*paid_A* ! *n*) **by** *auto*
  with *iH* **have** *x*: *swaps* (*drop* (*length* (*paid_A* ! *n*) − *m*) (*paid_A* ! *n*))
(*s_A n*) = *s′_A n m* **by** *auto*

  from *Suc* **have** *mlen*: (*length* (*paid_A* ! *n*) − *Suc m*) < *length* (*paid_A*
! *n*) **by** *auto*

  let *?l*=*length* (*paid_A* ! *n*) − *Suc m*
  let *?Sucl*=*length* (*paid_A* ! *n*) − *m*
  have *Sucl*: *Suc ?l* = *?Sucl* **using** *Suc* **by** *auto*

  from *mlen* **have** *yu*: ((*paid_A* ! *n*)! *?l*) # (*drop* (*Suc ?l*) (*paid_A* ! *n*))
      = (*drop ?l* (*paid_A* ! *n*))
    by (*rule Cons_nth_drop_Suc*)

from *Suc* **have** *s′_A n* (*Suc m*)
= *swap* ((*paid_A* ! *n*)!(*length* (*paid_A* ! *n*) − (*Suc m*))) (*s′_A n m*)
by *auto*
also have ... = *swap* ((*paid_A* ! *n*)!(*length* (*paid_A* ! *n*) − (*Suc m*)))
(*swaps* (*drop* (*length* (*paid_A* ! *n*) − *m*) (*paid_A* ! *n*)) (*s_A* *n*))
by(*simp only*: *x*)
also have ... = (*swaps* (((*paid_A* ! *n*)!(*length* (*paid_A* ! *n*) − (*Suc m*))) # (*drop* (*length* (*paid_A* ! *n*) − *m*) (*paid_A* ! *n*))) (*s_A n*))
by *auto*
also have ... = (*swaps* (((*paid_A* ! *n*)! *?l*) # (*drop* (*Suc ?l*) (*paid_A* ! *n*))) (*s_A n*))
using *Sucl* **by** *auto*
also from *mlen* **have** ... = (*swaps* ((*drop ?l* (*paid_A* ! *n*))) (*s_A n*))
by (*simp only*: *yu*)
finally have *s′_A n* (*Suc m*) = *swaps* (*drop* (*length* (*paid_A* ! *n*) − *Suc m*) (*paid_A* ! *n*)) (*s_A n*) .
then show *swaps* (*drop* (*length* (*paid_A* ! *n*) − *Suc m*) (*paid_A* ! *n*)) (*s_A n*) = *s′_A n* (*Suc m*) **by** *auto*
qed

lemma *s′A_m*: *swaps* (*paid_A* ! *n*) (*s_A n*) = *s′_A n* (*length* (*paid_A* ! *n*))
using *s′A_m_le*[*of* (*length* (*paid_A* ! *n*)) *n*, *simplified*] **by** *auto*

definition *gebub* :: *nat* ⇒ *nat* ⇒ *nat* **where**
gebub n m = *index init* ((*s′_A n m*)!(*Suc* ((*paid_A*!*n*)!(*length* (*paid_A* ! *n*) − *Suc m*))))

lemma *gebub_inBound*: **assumes** *1*: *n* < *length paid_A* **and** *2*: *m* < *length* (*paid_A* ! *n*)
shows *gebub n m* < *length init*
proof −
have *Suc* (*paid_A* ! *n* ! (*length* (*paid_A* ! *n*) − *Suc m*)) < *length* (*s′_A n m*) **using** *paidAnm_inbound*[*OF 1 2*] **by** *auto*
then have *s′_A n m* ! *Suc* (*paid_A* ! *n* ! (*length* (*paid_A* ! *n*) − *Suc m*)) ∈ *set* (*s′_A n m*) **by** (*rule nth_mem*)
then show *?thesis*
unfolding *gebub_def* **using** *setinit* **by** *auto*
qed

9.3.2 The Potential Function

fun *phi* :: *nat* ⇒*'a list*× (*bool list* × *'a list*) ⇒ *real* (‹φ›) **where**
phi n (*c*,(*b*,_)) = ($\sum$ (*x*,*y*)∈(*Inv c* (*s_A n*)). (*if b*!(*index init y*) *then 2 else 1*))

lemma *phi'*: *phi n z* = ($\sum$ (*x*,*y*)∈(*Inv* (*fst z*) (*s_A n*)). (*if* (*fst* (*snd z*))!(*index init y*) *then 2 else 1*))
proof −
 have *phi n z* = *phi n* (*fst z*, (*fst*(*snd z*),*snd*(*snd z*))) **by** (*metis prod.collapse*)
 also have ... = ($\sum$ (*x*,*y*)∈(*Inv* (*fst z*) (*s_A n*)). (*if* (*fst* (*snd z*))!(*index init y*) *then 2 else 1*)) **by**(*simp del*: *prod.collapse*)
 finally show *?thesis* **.**
qed

lemma *Inv_empty2*: *length d* = *0* ⟹ *Inv c d* = {}
unfolding *Inv_def before_in_def* **by**(*auto*)

corollary *Inv_empty3*: *length init* = *0* ⟹ *Inv c* (*s_A n*) = {}
apply(*rule Inv_empty2*) **by** (*metis length_s_A*)

lemma *phi_empty2*: *length init* = *0* ⟹ *phi n* (*c*,(*b*,*i*)) = *0*
apply(*simp only*: *phi.simps Inv_empty3*) **by** *auto*

lemma *phi_nonzero*: *phi n* (*c*,(*b*,*i*)) ≥ *0*
by (*simp add*: *sum_nonneg split_def*)

definition *Phi* :: *nat* ⇒ *real* (‹Φ›) **where**
Phi n = *E*(*map_pmf* (φ *n*) (*config'' BIT qs init n*))

definition *PhiPlus* :: *nat* ⇒ *real* (‹Φ^+›) **where**
PhiPlus n = (*let*
  nextconfig = *bind_pmf* (*config'' BIT qs init n*)
   (λ(*s*,*is*). *bind_pmf* (*BIT_step* (*s*,*is*) (*qs*!*n*)) (λ(*a*,*nis*). *return_pmf* (*step s* (*qs*!*n*) *a*,*nis*)))
    in
  E(*map_pmf* (*phi* (*Suc n*)) *nextconfig*))

lemma *PhiPlus_is_Phi_Suc*: *n*<*length qs* ⟹ *PhiPlus n* = *Phi* (*Suc n*)
unfolding *PhiPlus_def Phi_def*
apply (*simp add*: *bind_return_pmf map_pmf_def bind_assoc_pmf split_def take_Suc_conv_app_nth*)
 apply(*simp add*: *config'_rand_snoc*)

by(*simp add*: *bind_assoc_pmf split_def bind_return_pmf*)

lemma *phi0*: *Phi 0 = 0* **unfolding** *Phi_def*
by (*simp add*: *bind_return_pmf map_pmf_def bind_assoc_pmf BIT_init_def*)

lemma *phi_pos*: *Phi n ≥ 0*
unfolding *Phi_def*
apply(*rule E_nonneg_fun*)
using *phi_nonzero* **by** *auto*

9.3.3 Helper lemmas

lemma *swap_subs*: *dist_perm X Y ⟹ Inv X (swap z Y) ⊆ Inv X Y ∪ {(Y ! z, Y ! Suc z)}*
proof –
assume *dist_perm X Y*
note *aj = Inv_swap*[*OF this*, *of z*]
show *Inv X (swap z Y) ⊆ Inv X Y ∪ {(Y ! z, Y ! Suc z)}*
proof *cases*
assume *c1*: *Suc z < length X*
show *Inv X (swap z Y) ⊆ Inv X Y ∪ {(Y ! z, Y ! Suc z)}*
proof *cases*
assume *Y ! z < Y ! Suc z in X*
with *c1* **have** *Inv X (swap z Y) = Inv X Y ∪ {(Y ! z, Y ! Suc z)}*
using *aj* **by** *auto*
then show *Inv X (swap z Y) ⊆ Inv X Y ∪ {(Y ! z, Y ! Suc z)}* **by**
auto
next
assume ~ *Y ! z < Y ! Suc z in X*
with *c1* **have** *Inv X (swap z Y) = Inv X Y − {(Y ! Suc z, Y ! z)}*
using *aj* **by** *auto*
then show *Inv X (swap z Y) ⊆ Inv X Y ∪ {(Y ! z, Y ! Suc z)}* **by**
auto
qed
next
assume ~ (*Suc z < length X*)
then have *Inv X (swap z Y) = Inv X Y* **using** *aj* **by** *auto*
then show *Inv X (swap z Y) ⊆ Inv X Y ∪ {(Y ! z, Y ! Suc z)}* **by**
auto
qed
qed

9.3.4 InvOf

term *Inv*
abbreviation *InvOf y bits as* ≡ {(*x*,*y*)|*x*. *x* < *y in bits* ∧ *y* < *x in as*}

lemma *InvOf y xs ys* = {(*x*,*y*)|*x*. (*x*,*y*)∈*Inv xs ys*}
unfolding *Inv_def* **by** *auto*

lemma *InvOf y xs ys* ⊆ *Inv xs ys* **unfolding** *Inv_def* **by** *auto*

```
lemma numberofIsbeschr: assumes
    distxsys: dist_perm xs ys and
    yinxs: y ∈ set xs
  shows index xs y ≤ index ys y + card (InvOf y xs ys)
    (is ?iBit ≤ ?iA + card ?I)
proof −
  from assms have distinctxs: distinct xs
      and distinctys: distinct ys
      and yinys: y ∈ set ys by auto

  let ?A=fst ‘ ?I
  have aha: card ?A = card ?I apply(rule card_image)
    unfolding inj_on_def by(auto)

  have ?A ⊆ (before y xs) by(auto)
  have ?A ⊆ (after y ys) by auto



  have finite (before y ys) by auto

  have bef: (before y xs) − ?A ⊆ before y ys apply(auto)
  proof −
    fix x
    assume a: x < y in xs
    assume  x ∉ fst ‘ {(x, y) |x. x < y in xs ∧ y < x in ys}
    then have ~ (x < y in xs ∧ y < x in ys) by force
    with a have d: ~ y < x in ys by auto
    from a have x ∈ set xs by (rule before_in_setD1)
    with distxsys have b: x ∈ set ys by auto
    from a have  y ∈ set xs by (rule before_in_setD2)
    with distxsys have c: y ∈ set ys by auto
    from a have e: ~ x = y unfolding before_in_def by auto
   have (¬ y < x in ys) = (x < y in ys ∨ y = x) apply(rule not_before_in)
```

using *b c* **by** *auto*
with *d e* **show** *x < y in ys* **by** *auto*
qed

have (*index xs y*) − *card* (*InvOf y xs ys*) = *card* (*before y xs*) − *card ?A*
by(*simp only*: *aha card_before*[*OF distinctxs yinxs*])
also have ... = *card* ((*before y xs*) − *?A*)
apply(*rule card_Diff_subset*[*symmetric*]) **by** *auto*
also have ... ≤ *card* (*before y ys*)
apply(*rule card_mono*)
apply(*simp*)
apply(*rule bef*)
done
also have ... = (*index ys y*) **by**(*simp only*: *card_before*[*OF distinctys yinys*])
finally have *index xs y* − *card ?I* ≤ *index ys y* **.**
then show *index xs y* ≤ *index ys y* + *card ?I* **by** *auto*
qed

lemma *length init* = *0* ⟹ *length xs* = *length init* ⟹ *t xs q* (*mf*, *sws*) = *1* + *length sws*
unfolding *t_def* **by**(*auto*)

lemma *integr_index*: *integrable* (*measure_pmf* (*config''* (*BIT_init*, *BIT_step*) *qs init n*))
(λ(*s*, *is*). *real* (*Suc* (*index s* (*qs* ! *n*))))
apply(*rule measure_pmf.integrable_const_bound*[**where** *B*=*Suc* (*length init*)])
apply(*simp add*: *split_def*) **apply** (*metis* (*mono_tags*) *index_le_size AE_measure_pmf_iff config_rand_length*)
by (*auto*)

9.3.5 Upper Bound on the Cost of BIT

lemma *t_BIT_ub2*: (*qs*!*n*) ∉ *set init* ⟹ *t_BIT n* ≤ *Suc*(*size init*)
apply(*simp add*: *t_BIT_def t_def BIT_step_def*)
apply(*simp add*: *bind_return_pmf*)
proof (*goal_cases*)
case *1*
note *qs*=*this*
let *?D* = (*config''* (*BIT_init*, *BIT_step*) *qs init n*)

have *absch*: (∀ *x*∈ *set_pmf ?D*. ((λ(*s*,*is*). *real* (*Suc* (*index s* (*qs* ! *n*)))) *x*) ≤ ((λ(*is*,*s*). *Suc* (*length init*)) *x*))
proof (*rule ballI*, *goal_cases*)
case (*1 x*)
from *1 config_rand_length* **have** *f1*: *length* (*fst x*) = *length init* **by** *fastforce*
from *1 config_rand_set* **have** *2*: *set* (*fst x*) = *set init* **by** *fastforce*

from *qs 2* **have** (*qs*!*n*) ∉ *set* (*fst x*) **by** *auto*
then show *?case* **using** *f1* **by** (*simp add*: *split_def*)
qed

have *integrable* (*measure_pmf* (*config''* (*BIT_init*, *BIT_step*) *qs init n*)) (λ(*s*, *is*). *Suc* (*length init*)) **by**(*simp*)

have *E*(*bind_pmf ?D* (λ(*s*, *is*). *return_pmf* (*real* (*Suc* (*index s* (*qs* ! *n*))))))
= *E*(*map_pmf* (λ(*s*, *is*). *real* (*Suc* (*index s* (*qs* ! *n*)))) *?D*)
by(*simp add*: *split_def map_pmf_def*)
also have ... ≤ *E*(*map_pmf* (λ(*s*, *is*). *Suc* (*length init*)) *?D*)
apply (*rule E_mono3*)
apply(*fact integr_index*)
apply(*simp*)
using *absch* **by** *auto*
also have ... = *Suc* (*length init*)
by(*simp add*: *split_def*)
finally show *?case* **by**(*simp add*: *map_pmf_def bind_assoc_pmf bind_return_pmf split_def*)
qed

lemma *t_BIT_ub*: (*qs*!*n*) ∈ *set init* ⟹ *t_BIT n* ≤ *size init*
apply(*simp add*: *t_BIT_def t_def BIT_step_def*)
apply(*simp add*: *bind_return_pmf*)
proof (*goal_cases*)
case *1*
note *qs*=*this*
let *?D* = (*config''* (*BIT_init*, *BIT_step*) *qs init n*)

have *absch*: (∀ *x*∈ *set_pmf ?D*. ((λ(*s*, *is*). *real* (*Suc* (*index s* (*qs* ! *n*)))) *x*) ≤ ((λ(*s*, *is*). *length init*) *x*))
proof (*rule ballI*, *goal_cases*)
case (*1 x*)
from *1 config_rand_length* **have** *f1*: *length* (*fst x*) = *length init* **by** *fastforce*

```
    from 1 config_rand_set have 2: set (fst x) = set init by fastforce

    from qs 2 have (qs!n) ∈ set (fst x) by auto
    then have (index (fst x) (qs ! n)) < length init apply(rule index_less)
using f1 by auto
    then show ?case by (simp add: split_def)
  qed

   have E(bind_pmf ?D (λ(s, is). return_pmf (real (Suc (index s (qs !
n))))))
          = E(map_pmf (λ(s, is). real (Suc (index s (qs ! n)))) ?D)
          by(simp add: split_def map_pmf_def)
  also have ... ≤ E(map_pmf (λ(s, is). length init) ?D)
              apply(rule E_mono3)
                apply(fact integr_index)
                apply(simp)
                using absch by auto
  also have ... = length init
          by(simp add: split_def)
  finally show ?case by(simp add: map_pmf_def bind_assoc_pmf bind_return_pmf
split_def)
 qed

lemma T_BIT_ub: ∀ i<n. qs!i ∈ set init ⟹ T_BIT n ≤ n ∗ size init
proof(induction n)
  case 0 show ?case by(simp add: T_BIT_def)
next
  case (Suc n) thus ?case
    using t_BIT_ub[where n=n] by (simp add: T_BIT_def)
qed
```

9.3.6 Main Lemma

```
lemma myub: n < length qs ⟹ t_BIT n + Phi(n + 1) − Phi n ≤ (7 /
4) ∗ t_A n − 3/4
proof −
  assume nqs: n < length qs
  have t_BIT n + Phi (n+1) − Phi n ≤ (7 / 4) ∗ t_A n − 3/4
  proof (cases length init > 0)
    case False
    show ?thesis
    proof −
      from False have qsn: (qs!n) ∉ set init by auto
      from False have l0: length init = 0 by auto
```

then have *length* (*swaps* (*paid_A* ! *n*) (*s_A n*)) = *0* **using** *length_s_A* **by** *auto*

with *l0* **have** *4*: *t_A n* = *1* + *length* (*paid_A* ! *n*) **unfolding** *t_A_def c_A_def p_A_def* **by**(*simp*)

have *1*: *t_BIT n* ≤ *1* **using** *t_BIT_ub2*[*OF qsn*] *l0* **by** *auto*

{ fix *m*
have *phi m* = (λ(*b*,(*a*,*i*)). *phi m* (*b*,(*a*,*i*))) **by** *auto*
also have ... = (λ(*b*,(*a*,*i*)). *0*) **by**(*simp only*: *phi_empty2*[*OF l0*])
finally have *phi m*= (λ(*b*,(*a*,*i*)). *0*).
} note *phinull*=*this*

have *2*: *PhiPlus n* = *0* **unfolding** *PhiPlus_def* **apply**(*simp*) **apply**(*simp only*: *phinull*)
by (*auto simp*: *split_def*)
have *3*:*Phi n* = *0* **unfolding** *Phi_def* **apply**(*simp only*: *phinull*)
by (*auto simp*: *split_def*)

have *t_A n* ≥ *1* ⟹ *1* ≤ *7* / *4* ∗ (*t_A n*) − *3* / *4* **by**(*simp*)
with *4* **have** *5*: *1* ≤ *7* / *4* ∗ (*t_A n*) − *3* / *4* **by** *auto*

from *1 2 3* **have** *t_BIT n* + *PhiPlus n* − *Phi n* ≤ *1* **by** *auto*
also from *5* **have** ... ≤ *7* / *4* ∗ (*t_A n*) − *3* / *4* **by** *auto*

finally show *?thesis* **using** *PhiPlus_is_Phi_Suc nqs* **by** *auto*
qed
next
case *True*
let *?l* = *length init*
from *True* **obtain** *l′* **where** *lSuc*: *?l* = *Suc l′* **by** (*metis Suc_pred*)

have *31*: *n* < *length paid_A* **using** *nqs* **by** *auto*

define *q* **where** *q* = *qs*!*n*
define *D* **where** [*simp*]: *D* = (*config″* (*BIT_init*, *BIT_step*) *qs init n*)
define *cost* **where** [*simp*]: *cost* = (λ(*s*, *is*).(*t s q* (*if* (*fst is*) ! (*index* (*snd is*) *q*) *then 0 else length s*, [])))
define Φ_2 **where** [*simp*]: Φ_2 = (λ(*s*, *is*). ((*phi* (*Suc n*)) (*step s q* (*if* (*fst is*) ! (*index* (*snd is*) *q*) *then 0 else length s*, []),(*flip* (*index* (*snd is*) *q*) (*fst is*), *snd is*))))
define Φ_0 **where** [*simp*]: Φ_0 = *phi n*

have *inEreinziehn*: *t_BIT n* + *Phi* (*n*+*1*) − *Phi n* = *E* (*map_pmf* (λ*x*. (*cost x*) + (Φ_2 *x*) − (Φ_0 *x*)) *D*)
proof −
have *bind_pmf D*
(λ(*s*, *is*). *bind_pmf* (*BIT_step* (*s*, *is*) (*q*)) (λ(*a*,*nis*). *return_pmf* (*real*(*t s* (*q*) *a*))))
= *bind_pmf D*
(λ(*s*, *is*). *return_pmf* (*t s q* (*if* (*fst is*) ! (*index* (*snd is*) *q*) *then 0 else length s*, [])))
unfolding *BIT_step_def* **apply** (*auto simp*: *bind_return_pmf split_def*)
by (*metis prod.collapse*)
also have ... = *map_pmf cost D*
by (*auto simp*: *map_pmf_def split_def*)
finally have *rightform1*: *bind_pmf D*
(λ(*s*, *is*). *bind_pmf* (*BIT_step* (*s*, *is*) (*q*)) (λ(*a*,*nis*). *return_pmf* (*real*(*t s* (*q*) *a*))))
= *map_pmf cost D* **.**

have *rightform2*: *map_pmf* (*phi* (*Suc n*)) (*bind_pmf D*
(λ(*s*, *is*). *bind_pmf* (*BIT_step* (*s*, *is*) (*q*)) (λ(*a*, *nis*). *return_pmf* (*step s* (*q*) *a*, *nis*))))
= *map_pmf* Φ_2 *D* **apply**(*simp add*: *bind_return_pmf bind_assoc_pmf map_pmf_def split_def BIT_step_def*)
by (*metis prod.collapse*)
have *t_BIT n* + *Phi* (*n*+*1*) − *Phi n* =
t_BIT n + *PhiPlus n* − *Phi n* **using** *PhiPlus_is_Phi_Suc nqs* **by** *auto*
also have ... =
T_on_rand_n BIT init qs n
+ *E* (*map_pmf* (*phi* (*Suc n*)) (*bind_pmf D*
(λ(*s*, *is*). *bind_pmf* (*BIT_step* (*s*, *is*) (*q*)) (λ(*a*, *nis*). *return_pmf* (*step s* (*q*) *a*, *nis*)))))
− *E* (*map_pmf* (*phi n*) *D*)
unfolding *PhiPlus_def Phi_def t_BIT_def q_def* **by** *auto*
also have ... =
E (*bind_pmf D*
(λ(*s*, *is*). *bind_pmf* (*BIT_step* (*s*, *is*) (*q*)) (λ(*a*,*nis*). *return_pmf* (*t s* (*q*) *a*))))
+ *E* (*map_pmf* (*phi* (*Suc n*)) (*bind_pmf D*
(λ(*s*, *is*). *bind_pmf* (*BIT_step* (*s*, *is*) (*q*)) (λ(*a*, *nis*). *return_pmf* (*step s* (*q*) *a*, *nis*)))))
− *E* (*map_pmf* Φ_0 *D*) **by** (*auto simp*: *q_def split_def*)

also have ... = *E* (*map_pmf cost D*)
+ *E* (*map_pmf* Φ_2 *D*)
− *E* (*map_pmf* Φ_0 *D*) **using** *rightform1 rightform2 split_def*
by *auto*
also have ... = *E* (*map_pmf* (λ*x*. (*cost x*) + (Φ_2 *x*)) *D*) − *E* (*map_pmf* (λ*x*. (Φ_0 *x*)) *D*)
unfolding *D_def* **using** *E_linear_plus2*[*OF finite_config_BIT*[*OF dist_init*]] **by** *auto*
also have ... = *E* (*map_pmf* (λ*x*. (*cost x*) + (Φ_2 *x*) − (Φ_0 *x*)) *D*)
unfolding *D_def* **by**(*simp only*: *E_linear_diff2*[*OF finite_config_BIT*[*OF dist_init*]] *split_def*)
finally show *t_BIT n* + *Phi* (*n*+*1*) − *Phi n*
= *E* (*map_pmf* (λ*x*. (*cost x*) + (Φ_2 *x*) − (Φ_0 *x*)) *D*) **by** *auto*
qed

define *xs* **where** [*simp*]: *xs* = *s_A n*
define *xs'* **where** [*simp*]: *xs'* = *swaps* (*paid_A*!*n*) *xs*
define *xs''* **where** [*simp*]: *xs''* = *mtf2* (*free_A*!*n*) (*q*) *xs'*
define *k* **where** [*simp*]: *k* = *index xs' q*
define *k'* **where** [*simp*]: *k'* = *max 0* (*k*−*free_A*!*n*)

have [*simp*]: *length xs* = *length init* **by** *auto*

have *dp_xs_init*[*simp*]: *dist_perm xs init* **by** *auto*

The Transformation

have *ub_cost*: ∀*x*∈*set_pmf D*. (*real* (*cost x*)) + (Φ_2 *x*) − (Φ_0 *x*) ≤ *k* + *1* +
(*if* (*q*) ∈ *set init*
then (*if* (*fst* (*snd x*))!(*index init q*) *then k*−*k'*
else (∑*j*<*k'*. (*if* (*fst* (*snd x*))!(*index init* (*xs'*!*j*)) *then 2*::*real else 1*)))
else 0)
+ (∑*i*<(*length* (*paid_A*!*n*)). (*if* (*fst* (*snd x*))!(*gebub n i*) *then 2 else 1*))
proof (*rule*, *goal_cases*)
case (*1 x*)
note *xinD*=*1*
then have [*simp*]: *snd* (*snd x*) = *init* **using** *D_def config_n_init3* **by** *fast*

define *b* **where** *b* = *fst* (*snd x*)
define *ys* **where** *ys* = *fst x*
define *aBIT* **where** [*simp*]: *aBIT* = (*if b* ! (*index* (*snd* (*snd x*)) *q*)

then 0 else length ys, ([]::*nat list*))
define *ys′* **where** *ys′* = *step ys* (*q*) *aBIT*
define *b′* **where** *b′* = *flip* (*index init q*) *b*
define Φ_1 **where** Φ_1 = (λ*z*:: ′*a list*× (*bool list* × ′*a list*) . (∑ (*x*,*y*)∈(*Inv ys xs′*). (*if fst* (*snd z*)!(*index init y*) *then 2*::*real else 1*)))

have *xs′′_step*: *xs′′* = *step xs* (*q*) (*free_A*!*n*,*paid_A*!*n*)
unfolding *xs′_def xs′′_def xs_def step_def free_A_def paid_A_def*
by(*auto simp*: *split_def*)

have *gis2*: (Φ_2 (*ys*,(*b*,*init*))) = (∑ (*x*,*y*)∈(*Inv ys′ xs′′*). (*if b′*!(*index init y*) *then 2 else 1*))
apply(*simp only*: *split_def*)
apply(*simp only*: *xs′′_step*)
apply(*simp only*: Φ_2*_def phi.simps*)
unfolding *b′_def b_def ys′_def aBIT_def q_def*
unfolding *s_A.simps* **apply**(*simp only*: *split_def*) **by** *auto*
then have *gis*: Φ_2 *x* = (∑ (*x*,*y*)∈(*Inv ys′ xs′′*). (*if b′*!(*index init y*) *then 2 else 1*))
unfolding *ys_def b_def* **by** (*auto simp*: *split_def*)

have *his2*: (Φ_0 (*ys*,(*b*,*init*))) = (∑ (*x*,*y*)∈(*Inv ys xs*). (*if b*!(*index init y*) *then 2 else 1*))
apply(*simp only*: *split_def*)
apply(*simp only*: Φ_0*_def phi.simps*) **by**(*simp add*: *split_def*)
then have *his*: (Φ_0 *x*) = (∑ (*x*,*y*)∈(*Inv ys xs*). (*if b*!(*index init y*) *then 2 else 1*))
by(*auto simp*: *ys_def b_def split_def phi′*)

have *dis*: Φ_1 *x* = (∑ (*x*,*y*)∈(*Inv ys xs′*). (*if b*!(*index init y*) *then 2 else 1*))
unfolding Φ_1*_def b_def* **by** *auto*

have *ys′* = *mtf2* (*fst aBIT*) (*q*) *ys* **by** (*simp add*: *step_def ys′_def*)

from *config_rand_distinct*[*of BIT*] *config_rand_set*[*of BIT*] *xinD*
have *dp_ys_init*[*simp*]: *dist_perm ys init* **unfolding** *D_def ys_def* **by** *force*
have *dp_ys′_init*[*simp*]: *dist_perm ys′ init* **unfolding** *ys′_def step_def* **by** (*auto*)
then have *lenys′*[*simp*]: *length ys′* = *length init* **by** (*metis distinct_card*)
have *dp_xs′_init*[*simp*]: *dist_perm xs′ init* **by** *auto*
have *gra*: *dist_perm ys xs′* **by** *auto*

have *leninitb*[*simp*]: *length b* = *length init* **using** *b_def config_n_fst_init_length2 xinD*[*unfolded*] **by** *auto*
have *leninitys*[*simp*]: *length ys* = *length init* **using** *dp_ys_init* **by** (*metis distinct_card*)

{**fix** *m*
have *dist_perm ys* (*s'_A n m*) **using** *dp_ys_init* **by** *auto*
} **note** *dist*=*this*

Upper bound of the inversions created by paid exchanges of A

let *?paidUB*=($\sum$ *i*<(*length* (*paid_A*!*n*)). (*if b*!(*gebub n i*) *then 2*::*real else 1*))

have *paid_ub*: Φ_1 *x* ≤ Φ_0 *x* + *?paidUB*
proof −

have *a*: *length* (*paid_A* ! *n*) ≤ *length* (*paid_A* ! *n*) **by** *auto*
have *b*: *xs'* = (*s'_A n* (*length* (*paid_A* ! *n*))) **using** *s'A_m* **by** *auto*

{
fix *m*
have *m*≤*length* (*paid_A*!*n*) ⟹ ($\sum$ (*x*,*y*)∈(*Inv ys* (*s'_A n m*)). (*if b*!(*index init y*) *then 2*::*real else 1*)) ≤ ($\sum$ (*x*,*y*)∈(*Inv ys xs*). (*if b*!(*index init y*) *then 2 else 1*))
+ ($\sum$ *i*<*m*. (*if b*!(*gebub n i*) *then 2 else 1*))
proof (*induct m*)
case (*Suc m*)
then have *m_bd2*: *m* ≤ *length* (*paid_A* ! *n*)
and *m_bd*: *m* < *length* (*paid_A* ! *n*) **by** *auto*
note *yeah* = *Suc*(*1*)[*OF m_bd2*]

let *?revm*=(*length* (*paid_A* ! *n*) − *Suc m*)
note *ah*=*Inv_swap*[*of ys* (*s'_A n m*) (*paid_A* ! *n* ! *?revm*), *OF dist*]
have ($\sum$ (*xa*, *y*)∈*Inv ys* (*s'_A n* (*Suc m*)). *if b* ! (*index init y*) *then 2*::*real else 1*)
= ($\sum$ (*xa*, *y*)∈*Inv ys* (*swap* (*paid_A* ! *n* ! *?revm*) (*s'_A n m*)). *if b* ! (*index init y*) *then 2 else 1*) **using** *s'_A.simps*(*2*) **by** *auto*
also
have ... = ($\sum$ (*xa*, *y*)∈(*if Suc* (*paid_A* ! *n* ! *?revm*) < *length ys then if s'_A n m* ! (*paid_A* ! *n* ! *?revm*) < *s'_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*) *in ys*
then Inv ys (*s'_A n m*) ∪ {(*s'_A n m* ! (*paid_A* ! *n* ! *?revm*), *s'_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*))}

else Inv ys (*s′_A n m*) − {(*s′_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*), *s′_A n m* ! (*paid_A* ! *n* ! *?revm*))}
else Inv ys (*s′_A n m*)). *if b* ! (*index init y*) *then 2*::*real else 1*) **by** (*simp only*: *ah*)
also
have . . . ≤ (∑(*xa*, *y*)∈*Inv ys* (*s′_A n m*). *if b* ! (*index init y*) *then 2*::*real else 1*)
+ (*if* (*b*) ! (*index init* (*s′_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*))) *then 2*::*real else 1*) (**is** *?A* ≤ *?B*)
proof(*cases Suc* (*paid_A* ! *n* ! *?revm*) < *length ys*)
case *False*
then have *?A* = (∑(*xa*, *y*)∈(*Inv ys* (*s′_A n m*)). *if b* ! (*index init y*) *then 2 else 1*) **by** *auto*
also have . . . ≤ (∑(*xa*, *y*)∈(*Inv ys* (*s′_A n m*)). *if b* ! (*index init y*) *then 2 else 1*) +
(*if b* ! (*index init* (*s′_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*))) *then 2*::*real else 1*) **by** *auto*
finally show *?A* ≤ *?B* **.**
next
case *True*
then have *?A* = (∑(*xa*, *y*)∈(*if s′_A n m* ! (*paid_A* ! *n* ! *?revm*) < *s′_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*) *in ys*
then Inv ys (*s′_A n m*) ∪ {(*s′_A n m* ! (*paid_A* ! *n* ! *?revm*), *s′_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*))}
else Inv ys (*s′_A n m*) − {(*s′_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*), *s′_A n m* ! (*paid_A* ! *n* ! *?revm*))}
). *if b* ! (*index init y*) *then 2 else 1*) **by** *auto*
also have . . . ≤ *?B* (**is** *?A′* ≤ *?B*)
proof (*cases s′_A n m* ! (*paid_A* ! *n* ! *?revm*) < *s′_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*) *in ys*)
case *True*
let *?neurein*=(*s′_A n m* ! (*paid_A* ! *n* ! *?revm*), *s′_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*))
from *True* **have** *?A′* = (∑(*xa*, *y*)∈(*Inv ys* (*s′_A n m*) ∪ {*?neurein*}
). *if b* ! (*index init y*) *then 2 else 1*) **by** *auto*
also have . . . = (∑(*xa*, *y*)∈*insert ?neurein* (*Inv ys* (*s′_A n m*)
). *if b* ! (*index init y*) *then 2 else 1*) **by** *auto*
also have . . . ≤ (*if b* ! (*index init* (*snd ?neurein*)) *then 2 else 1*)
+ (∑(*xa*, *y*)∈(*Inv ys* (*s′_A n m*)). *if b* ! (*index init y*) *then 2 else 1*)
proof (*cases ?neurein* ∈ *Inv ys* (*s′_A n m*))

case *True*
then have *insert ?neurein* (*Inv ys* (*s'_A n m*)) = (*Inv ys* (*s'_A n m*)) **by** *auto*
then have (∑(*xa*, *y*)∈*insert ?neurein* (*Inv ys* (*s'_A n m*)). *if b* ! (*index init y*) *then 2 else 1*)
= (∑(*xa*, *y*)∈(*Inv ys* (*s'_A n m*)). *if b* ! (*index init y*) *then 2 else 1*) **by** *auto*
also have ... ≤ (*if b* ! (*index init* (*snd ?neurein*)) *then 2*::*real else 1*)
+ (∑(*xa*, *y*)∈(*Inv ys* (*s'_A n m*)). *if b* ! (*index init y*) *then 2 else 1*) **by** *auto*
finally show *?thesis* **.**
next
case *False*
have (∑(*xa*, *y*)∈*insert ?neurein* (*Inv ys* (*s'_A n m*)). *if b* ! (*index init y*) *then 2 else 1*)
= (∑*y*∈*insert ?neurein* (*Inv ys* (*s'_A n m*)). (λ*i*. *if b* ! (*index init* (*snd i*)) *then 2 else 1*) *y*) **by**(*auto simp*: *split_def*)
also have ... = (λ*i*. *if b* ! (*index init* (*snd i*)) *then 2 else 1*) *?neurein*
+ (∑*y*∈(*Inv ys* (*s'_A n m*)) − {*?neurein*}. (λ*i*. *if b* ! (*index init* (*snd i*)) *then 2 else 1*) *y*)
apply(*rule sum.insert_remove*) **by**(*auto*)
also have ... = (*if b* ! (*index init* (*snd ?neurein*)) *then 2 else 1*)
+ (∑*y*∈(*Inv ys* (*s'_A n m*)). (λ*i*. *if b* ! (*index init* (*snd i*)) *then 2*::*real else 1*) *y*) **using** *False* **by** *auto*
also have ... ≤ (*if b* ! (*index init* (*snd ?neurein*)) *then 2 else 1*)
+ (∑(*xa*, *y*)∈(*Inv ys* (*s'_A n m*)). *if b* ! (*index init y*) *then 2 else 1*) **by**(*simp only*: *split_def*)
finally show *?thesis* **.**
qed
also have ... = (∑(*xa*, *y*)∈*Inv ys* (*s'_A n m*). *if b* ! (*index init y*) *then 2 else 1*) +
(*if b* ! (*index init* (*s'_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*))) *then 2 else 1*) **by** *auto*
finally show *?thesis* **.**
next
case *False*
then have *?A'* = (∑(*xa*, *y*)∈(*Inv ys* (*s'_A n m*) − {(*s'_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*), *s'_A n m* ! (*paid_A* ! *n* ! *?revm*))}
). *if b* ! (*index init y*) *then 2 else 1*) **by** *auto*
also have ... ≤ (∑(*xa*, *y*)∈(*Inv ys* (*s'_A n m*)). *if b* ! (*index*

init y) then 2 else 1) (**is** ($\sum$(*xa, y*)∈*?X*−{*?x*}. *?g y*) ≤ ($\sum$(*xa, y*)∈*?X*. *?g y*))

proof (*cases ?x* ∈ *?X*)
case *True*
have ($\sum$(*xa, y*)∈*?X*−{*?x*}. *?g y*) ≤ (%(*xa*,*y*). *?g y*) *?x* + ($\sum$(*xa, y*)∈*?X*−{*?x*}. *?g y*)
by *simp*
also have ... = ($\sum$(*xa, y*)∈*?X*. *?g y*)
apply(*rule sum.remove*[*symmetric*])
apply *simp* **apply**(*fact*) **done**
finally show *?thesis* **.**
qed *simp*
also have ... ≤ *?B* **by** *auto*
finally show *?thesis* **.**
qed
finally show *?A* ≤ *?B* **.**
qed

also have ...
≤ ($\sum$(*xa, y*)∈*Inv ys* (*s_A n*). *if b* ! (*index init y*) *then 2*::*real else 1*) + ($\sum$ *i*<*m*. *if b* ! *gebub n i then 2*::*real else 1*)
+ (*if* (*b*) ! (*index init* (*s′_A n m* ! *Suc* (*paid_A* ! *n* ! *?revm*))) *then 2*::*real else 1*) **using** *yeah* **by** *simp*
also have ... = ($\sum$(*xa, y*)∈*Inv ys* (*s_A n*). *if b* ! (*index init y*) *then 2*::*real else 1*) + ($\sum$ *i*<*m*. *if b* ! *gebub n i then 2 else 1*)
+ (*if* (*b*) ! *gebub n m then 2 else 1*) **unfolding** *gebub_def* **by** *simp*
also have ... = ($\sum$(*xa, y*)∈*Inv ys* (*s_A n*). *if b* ! (*index init y*) *then 2*::*real else 1*) + ($\sum$ *i*<(*Suc m*). *if b* ! *gebub n i then 2 else 1*)
by *auto*
finally show *?case* **by** *simp*
qed (*simp add*: *split_def*)
} **note** *x* = *this*[*OF a*]

show *?thesis*
unfolding Φ_1*_def his* **apply**(*simp only*: *b*) **using** *x b_def* **by** *auto*
qed

Upper bound for the costs of BIT

define *inI* **where** [*simp*]: *inI* = *InvOf* (*q*) *ys xs′*
define *I* **where** [*simp*]: *I* = *card*(*InvOf* (*q*) *ys xs′*)

have *ub_cost_BIT*: (*cost x*) ≤ *k* + *1* + *I*

proof (*cases* (*q*) ∈ *set init*)
case *False*
from *False* **have** *4*: *I* = *0* **by**(*auto simp*: *before_in_def*)
have (*cost x*) = *1* + *index ys* (*q*) **by** (*auto simp*: *ys_def t_def split_def*)
also have ... = *1* + *length init* **using** *False* **by** *auto*
also have ... = *1* + *k* **using** *False* **by** *auto*
finally show *?thesis* **using** *4* **by** *auto*
next
case *True*
then have *gra2*: (*q*) ∈ *set ys* **using** *dp_ys_init* **by** *auto*
have (*cost x*) = *1* + *index ys* (*q*) **by**(*auto simp*: *ys_def t_def split_def*)
also have ... ≤ *k* + *1* + *I* **using** *numberofIsbeschr*[*OF gra gra2*] **by** *auto*
finally show(*cost x*) ≤ *k* + *1* + *I* **.**
qed

Upper bound for inversions generated by free exchanges

define *ub_free*
where *ub_free* =
(*if* (*q* ∈ *set init*)
then (*if b*!(*index init q*) *then* *k*−*k*′ *else* (∑ *j*<*k*′. (*if* (*b*)!(*index init* (*xs*′!*j*)) *then 2*::*real else 1*)))
else 0)
let *?ub2* = − *I* + *ub_free*
have *free_ub*: (∑ (*x*,*y*)∈(*Inv ys*′ *xs*′′). (*if b*′ !(*index init y*) *then 2 else 1*))
− (∑ (*x*,*y*)∈(*Inv ys xs*′). (*if b*!(*index init y*) *then 2 else 1*)) ≤ *?ub2*
proof (*cases* (*q*) ∈ *set init*)
case *False*

from *False* **have** *1*: *ys*′ = *ys* **unfolding** *ys*′*_def step_def mtf2_def* **by**(*simp*)
from *False* **have** *2*: *xs*′ = *xs*′′ **unfolding** *xs*′′*_def mtf2_def* **by**(*simp*)
from *False* **have** (*index init q*) ≥ *length b* **using** *setinit* **by** *auto*
then have *3*: *b*′ = *b* **unfolding** *b*′*_def* **using** *flip_out_of_bounds* **by** *auto*

from *False* **have** *4*: *I* = *0* **unfolding** *I_def before_in_def* **by**(*auto*)

note *ubnn*=*False*

have *nn*: $k-k'\geq 0$ **unfolding** *k_def k'_def* **by** *auto*

from *1 2 3 4* **have** $(\sum (x,y)\in(Inv\ ys'\ xs'').\ (if\ b'!(index\ init\ y)\ then$
$2::real\ else\ 1))$
$-\ (\sum (x,y)\in(Inv\ ys\ xs').\ (if\ b!(index\ init\ y)\ then\ 2\ else\ 1)) =$
$-I$ **by** *auto*
with *ubnn* **show** *?thesis* **unfolding** *ub_free_def* **by** *auto*
next
case *True*
note *queryinlist=this*

then have *gra2*: $q \in set\ ys$ **using** *dp_ys_init* **by** *auto*

have *k_inbounds*: $k < length\ init$
using *index_less_size_conv queryinlist*
by (*simp*)
{
fix *y e*
fix *X*::*bool list*
assume *rd*: $e < length\ X$
have $y < length\ X \Longrightarrow (if\ flip\ e\ X\ !\ y\ then\ 2::real\ else\ 1) - (if\ X\ !$
$y\ then\ 2\ else\ 1)$
$= (if\ e=y\ then\ (if\ X\ !\ y\ then\ -1\ else\ 1)\ else\ 0)$
proof *cases*
assume $y < length\ X$ **and** *ey*: $e=y$
then have $(if\ flip\ e\ X\ !\ y\ then\ 2::real\ else\ 1) - (if\ X\ !\ y\ then\ 2$
$else\ 1)$
$= (if\ X\ !\ y\ then\ 1::real\ else\ 2) - (if\ X\ !\ y\ then\ 2\ else\ 1)$
using *flip_itself* **by** *auto*
also have $\ldots = (if\ X\ !\ y\ then\ -1::real\ else\ 1)$ **by** *auto*
finally
show $(if\ flip\ e\ X\ !\ y\ then\ 2::real\ else\ 1) - (if\ X\ !\ y\ then\ 2\ else\ 1)$
$= (if\ e=y\ then\ (if\ X\ !\ y\ then\ -1\ else\ 1)\ else\ 0)$ **using** *ey* **by**
auto
next
assume *len*: $y < length\ X$ **and** *eny*: $e\neq y$
then have $(if\ flip\ e\ X\ !\ y\ then\ 2::real\ else\ 1) - (if\ X\ !\ y\ then\ 2$
$else\ 1)$
$= (if\ X\ !\ y\ then\ 2::real\ else\ 1) - (if\ X\ !\ y\ then\ 2\ else\ 1)$
using *flip_other*[*OF len rd eny*] **by** *auto*
also have $\ldots = 0$ **by** *auto*
finally
show $(if\ flip\ e\ X\ !\ y\ then\ 2::real\ else\ 1) - (if\ X\ !\ y\ then\ 2\ else\ 1)$

= (*if* $e=y$ *then* (*if* X ! y *then* -1 *else* 1) *else* 0) **using** *eny* **by** *auto*

qed

} **note** *flipstyle=this*

from *queryinlist setinit* **have** *qsfst*: (*index init q*) < *length b* **by** *simp*

have *fA*: *finite* (*Inv ys′ xs′′*) **by** *auto*

have *fB*: *finite* (*Inv ys xs′*) **by** *auto*

define Δ **where** [*simp*]: Δ = (∑ (*x*,*y*)∈(*Inv ys′ xs′′*). (*if b′*!(*index init y*) *then* 2::*real else* 1))

− (∑ (*x*,*y*)∈(*Inv ys xs′*). (*if b*!(*index init y*) *then* 2 *else* 1))

define *C* **where** [*simp*]: *C* = (∑ (*x*,*y*)∈(*Inv ys′ xs′′*) ∩ (*Inv ys xs′*). (*if b′*!(*index init y*) *then* 2::*real else* 1)

− (*if b*!(*index init y*) *then* 2 *else* 1))

define *A* **where** [*simp*]: *A* = (∑ (*x*,*y*)∈(*Inv ys′ xs′′*)−(*Inv ys xs′*). (*if b′*!(*index init y*) *then* 2::*real else* 1))

define *B* **where** [*simp*]: *B* = (∑ (*x*,*y*)∈(*Inv ys xs′*)−(*Inv ys′ xs′′*). (*if b*!(*index init y*) *then* 2::*real else* 1))

have *teilen*: Δ = *C* + *A* − *B*

unfolding Δ_*def A_def B_def C_def*

using *sum_my*[*OF fA fB*] **by** (*auto simp*: *split_def*)

then have Δ = *A* − *B* + *C* **by** *auto*

then have *teilen2*: Φ_2 *x* − Φ_1 *x* = *A* − *B* + *C* **unfolding** Δ_*def* **using** *dis gis* **by** *auto*

have *setys′*: (*index init*) ‘ (*set ys′*) = {0..<*length ys′*}

proof −

have (*index init*) ‘ (*set ys′*) = (*index init*) ‘ (*set init*) **by** *auto*

also have . . . = {0..<*length init*} **using** *setinit* **by** *auto*

also have . . . = {0..<*length ys′*} **using** *lenys′* **by** *auto*

finally show *?thesis* **.**

qed

have *BC_absch*: *C* − *B* ≤ −*I*

proof (*cases b*!(*index init q*))

case *True*

then have *samesame*: *ys′* = *ys* **unfolding** *ys′_def step_def* **by** *auto*

then have *puh*: (*Inv ys′ xs′*) = (*Inv ys xs′*) **by** *auto*

{
fix α β
assume $(\alpha,\beta)\in(Inv\ ys'\ xs'') \cap (Inv\ ys'\ xs')$
then have $(\alpha,\beta)\in(Inv\ ys'\ xs'')$ **by** *auto*
then have $(\alpha < \beta\ in\ ys')$ **unfolding** *Inv_def* **by** *auto*
then have *1*: $\beta \in set\ ys'$ **by** (*simp only*: *before_in_setD2*)
then have *index init* $\beta <$ *length ys'* **using** *setys'* **by** *auto*
then have *index init* $\beta <$ *length init* **using** *lenys'* **by** *auto*
then have *puzzel*: *index init* $\beta <$ *length b* **using** *leninitb* **by** *auto*

have *betainit*: $\beta \in$ *set init* **using** *1* **by** *auto*
have *aha*: $(q=\beta) = (index\ init\ q = index\ init\ \beta)$
using *betainit* **by** *simp*

have (*if b'*!(*index init* β) *then 2*::*real else 1*) $-$ (*if b*!(*index init* β) *then 2 else 1*)
= (*if* (*index init q*) = (*index init* β) *then if b* !(*index init* β) *then* $-$ *1 else 1 else 0*)
unfolding *b'_def* **apply**(*rule flipstyle*) **by**(*fact*)+
also have ... = (*if* (*index init q*) = (*index init* β) *then if b* !(*index init q*) *then* $-$ *1 else 1 else 0*) **by** *auto*
also have ... = (*if q* = β *then* $-$ *1 else 0*) **using** *aha True* **by** *auto*
finally have (*if b'*!(*index init* β) *then 2*::*real else 1*) $-$ (*if b*!(*index init* β) *then 2 else 1*)
= (*if* (*q*) = β *then* $-$*1*::*real else 0*) **by** *auto*
}
then have *grrreeeaa*: $\forall x\in(Inv\ ys'\ xs'') \cap (Inv\ ys'\ xs')$.
(λx. (*if b'*! (*index init* (*snd x*)) *then 2*::*real else 1*) $-$ (*if b*! (*index init* (*snd x*)) *then 2 else 1*)) *x*
= (λx. (*if* (*q*) = *snd x then* $-$*1*::*real else 0*)) *x* **by** *force*

let *?fin*=$(Inv\ ys'\ xs'') \cap (Inv\ ys'\ xs')$

have *ttt*: $\{(x,y).\ (x,y)\in(Inv\ ys'\ xs'') \cap (Inv\ ys'\ xs')$
$\wedge\ y = (q)\} \cup \{(x,y).\ (x,y)\in(Inv\ ys'\ xs'') \cap (Inv\ ys'\ xs')$
$\wedge\ y \neq (q)\} = (Inv\ ys'\ xs'') \cap (Inv\ ys'\ xs')$ (**is** *?split1* $\cup$ *?split2* = *?easy*) **by** *auto*
have *interem*: *?split1* $\cap$ *?split2* = {} **by** *auto*
have *split1subs*: *?split1* $\subseteq$ *?fin* **by** *auto*
have *split2subs*: *?split2* $\subseteq$ *?fin* **by** *auto*
have *fs1*: *finite ?split1* **apply**(*rule finite_subset*[**where** *B*=*?fin*])

apply(*rule split1subs*) **by**(*auto*)
have *fs2*: *finite ?split2* **apply**(*rule finite_subset*[**where** *B*=*?fin*])
apply(*rule split2subs*) **by**(*auto*)

have $k - k' \leq$ (*free_A*!*n*) **by** *auto*

have *g*: *InvOf* (*q*) *ys′ xs″* ⊇ *InvOf* (*q*) *ys′ xs′*
using *True* **apply**(*auto*) **apply**(*rule mtf2_mono*[*of swaps* (*paid_A* ! *n*) (*s_A n*)])
by (*auto simp*: *queryinlist*)
have *h*: *?split1* = (*InvOf* (*q*) *ys′ xs″*) ∩ (*InvOf* (*q*) *ys′ xs′*)
unfolding *Inv_def* **by** *auto*
also from *g* **have** ... = *InvOf* (*q*) *ys′ xs′* **by** *force*
also from *samesame* **have** ... = *InvOf* (*q*) *ys xs′* **by** *simp*
finally have *?split1* = *inI* **unfolding** *inI_def* **.**
then have *cardsp1isI*: *card ?split1* = *I* **by** *auto*

{
fix *a b*
assume (*a*,*b*)∈*?split1*
then have *b* = (*q*) **by** *auto*
then have (*if* (*q*) = *b then* (−*1*::*real*) *else 0*) = (−*1*::*real*) **by** *auto*
}
then have *split1easy*: ∀ *x*∈*?split1*.
(λ*x*. (*if* (*q*) = *snd x then* (−*1*::*real*) *else 0*)) *x* = (λ*x*. (−*1*::*real*)) *x* **by** *force*
{
fix *a b*
assume (*a*,*b*)∈*?split2*
then have ~ *b* = (*q*) **by** *auto*
then have (*if* (*q*) = *b then* (−*1*::*real*) *else 0*) = *0* **by** *auto*
}
then have *split2easy*: ∀ *x*∈*?split2*.
(λ*x*. (*if* (*q*) = *snd x then* (−*1*::*real*) *else 0*)) *x* = (λ*x*. *0*::*real*) *x* **by** *force*

have *E0*: *C* =
(∑ (*x*,*y*)∈(*Inv ys′ xs″*) ∩ (*Inv ys xs′*).
(*if b′*!(*index init y*) *then 2*::*real else 1*) − (*if b*!(*index init y*) *then 2 else 1*)) **by** *auto*
also from *puh* **have** *E1*: ... =
(∑ (*x*,*y*)∈(*Inv ys′ xs″*) ∩ (*Inv ys′ xs′*).

```
              (if b'!(index init y) then 2::real else 1) − (if b!(index init
y) then 2 else 1)) by auto
        also have E2: ... = (∑(x,y)∈?easy.
              (if (q) = y then (−1::real) else 0)) using sum_my2[OF
grreeeaa] by (auto simp: split_def)
        also have E3: ... = (∑(x,y)∈?split1 ∪ ?split2.
              (if (q) = y then (−1::real) else 0)) by(simp only: ttt)
        also have ... = (∑(x,y)∈?split1. (if (q) = y then (−1::real) else
0))
              + (∑(x,y)∈?split2. (if (q) = y then (−1::real) else 0))
              − (∑(x,y)∈?split1 ∩ ?split2. (if (q) = y then (−1::real)
else 0))
              by(rule sum_Un[OF fs1 fs2])
        also have ... = (∑(x,y)∈?split1. (if (q) = y then (−1::real) else
0))
              + (∑(x,y)∈?split2. (if (q) = y then (−1::real) else 0))
              apply(simp only: interem) by auto
        also have E4: ... = (∑(x,y)∈?split1. (−1::real) )
              + (∑(x,y)∈?split2. 0)
               using sum_my2[OF split1easy]sum_my2[OF split2easy]
by(simp only: split_def)
        also have ... = (∑(x,y)∈?split1. (−1::real) ) by auto
        also have E5: ... = − card ?split1  by auto
        also have E6: ... = − I  using cardsp1isI by auto
        finally have abschC: C = −I.

      have abschB: B ≥ (0::real) unfolding B_def apply(rule sum_nonneg)
by auto

        from abschB abschC show C − B ≤ −I by simp

      next
        case False
        from leninitys False have ya: ys' = mtf2 (length ys) q ys
            unfolding step_def ys'_def by(auto)
        have index ys' q = 0
          unfolding ya apply(rule mtf2_moves_to_front)
          using gra2 by simp_all
         then have nixbefore: before q ys' = {} unfolding before_in_def
by auto

        {
          fix α β
          assume (α,β)∈(Inv ys' xs'') ∩ (Inv ys xs')
```

then have $(\alpha,\beta)\in$(*Inv ys′ xs″*) **by** *auto*
then have ($\alpha <$ β *in ys′*) **unfolding** *Inv_def* **by** *auto*
then have *1*: $\beta \in$ *set ys′* **by** (*simp only*: *before_in_setD2*)
then have (*index init* β) $<$ *length ys′* **using** *setys′* **by** *auto*
then have (*index init* β) $<$ *length init* **using** *lenys′* **by** *auto*
then have *puzzel*: (*index init* β) $<$ *length b* **using** *leninitb* **by** *auto*

have *betainit*: $\beta \in$ *set init* **using** *1* **by** *auto*
have *aha*: ($q{=}\beta$) = (*index init q* = *index init* β)
using *betainit* **by** *simp*

have (*if b′*!(*index init* β) *then 2*::*real else 1*) − (*if b*!(*index init* β) *then 2 else 1*)
= (*if* (*index init q*) = (*index init* β) *then if b* ! (*index init* β) *then* − *1 else 1 else 0*)
unfolding *b′_def* **apply**(*rule flipstyle*) **by**(*fact*)+
also have . . . = (*if* (*index init q*) = (*index init* β) *then if b* ! (*index init q*) *then* − *1 else 1 else 0*) **by** *auto*
also have . . . = (*if* (*q*) = β *then 1 else 0*) **using** *False aha* **by** *auto*
finally have (*if b′*!(*index init* β) *then 2*::*real else 1*) − (*if b*!(*index init* β) *then 2 else 1*)
= (*if* (*q*) = β *then 1*::*real else 0*) **by** *auto*
}
then have *grreeeaa2*: $\forall\, x\in$(*Inv ys′ xs″*) $\cap$ (*Inv ys xs′*).
(λx. (*if b′*! (*index init* (*snd x*)) *then 2*::*real else 1*) − (*if b*! (*index init* (*snd x*)) *then 2 else 1*)) *x*
= (λx. (*if* (*q*) = *snd x then 1*::*real else 0*)) *x* **by** *force*

let *?fin*=(*Inv ys′ xs″*) $\cap$ (*Inv ys xs′*)

have *ttt*: {(*x*,*y*). (*x*,*y*)$\in$(*Inv ys′ xs″*) $\cap$ (*Inv ys xs′*)
$\wedge$ *y* = (*q*)} $\cup$ {(*x*,*y*). (*x*,*y*)$\in$(*Inv ys′ xs″*) $\cap$ (*Inv ys xs′*)
$\wedge$ *y* $\neq$ (*q*)} = (*Inv ys′ xs″*) $\cap$ (*Inv ys xs′*) (**is** *?split1* $\cup$ *?split2* = *?easy*) **by** *auto*
have *interem*: *?split1* $\cap$ *?split2* = {} **by** *auto*
have *split1subs*: *?split1* $\subseteq$ *?fin* **by** *auto*
have *split2subs*: *?split2* $\subseteq$ *?fin* **by** *auto*
have *fs1*: *finite ?split1* **apply**(*rule finite_subset*[**where** *B*=*?fin*])
apply(*rule split1subs*) **by**(*auto*)
have *fs2*: *finite ?split2* **apply**(*rule finite_subset*[**where** *B*=*?fin*])
apply(*rule split2subs*) **by**(*auto*)

have *split1easy* : ∀ *x*∈*?split1*.
(λ*x*. (*if* (*q*) = *snd x then* (*1*::*real*) *else 0*)) *x* = (λ*x*. (*1*::*real*)) *x*
by *force*

have *split2easy* : ∀ *x*∈*?split2*.
(λ*x*. (*if* (*q*) = *snd x then* (*1*::*real*) *else 0*)) *x* = (λ*x*. (*0*::*real*)) *x*
by *force*

from *nixbefore* **have** *InvOfempty*: *InvOf q ys′ xs″* = {} **unfolding** *Inv_def* **by** *auto*

have *?split1* = *InvOf q ys′ xs″* ∩ *InvOf q ys xs′*
unfolding *Inv_def* **by** *auto*
also from *InvOfempty* **have** ... = {} **by** *auto*
finally have *split1empty*: *?split1* = {} **.**

have *C* = (∑ (*x*,*y*)∈*?easy*.
(*if* (*q*) = *y then* (*1*::*real*) *else 0*)) **unfolding** *C_def*
by(*simp only*: *split_def sum_my2*[*OF grreeeaa2*])
also have ... = (∑ (*x*,*y*)∈*?split1* ∪ *?split2*.
(*if* (*q*) = *y then* (*1*::*real*) *else 0*)) **by**(*simp only*: *ttt*)
also have ... = (∑ (*x*,*y*)∈*?split1*. (*if* (*q*) = *y then* (*1*::*real*) *else 0*))
+ (∑ (*x*,*y*)∈*?split2*. (*if* (*q*) = *y then* (*1*::*real*) *else 0*))
− (∑ (*x*,*y*)∈*?split1* ∩ *?split2*. (*if* (*q*) = *y then* (*1*::*real*) *else 0*))
by(*rule sum_Un*[*OF fs1 fs2*])
also have ... = (∑ (*x*,*y*)∈*?split1*. (*if* (*q*) = *y then* (*1*::*real*) *else 0*))
+ (∑ (*x*,*y*)∈*?split2*. (*if* (*q*) = *y then* (*1*::*real*) *else 0*))
apply(*simp only*: *interem*) **by** *auto*
also have ... = (∑ (*x*,*y*)∈*?split1*. (*1*::*real*))
+ (∑ (*x*,*y*)∈*?split2*. *0*) **using** *sum_my2*[*OF split1easy*] *sum_my2*[*OF split2easy*] **by** (*simp only*: *split_def*)
also have ... = (∑ (*x*,*y*)∈*?split1*. (*1*::*real*)) **by** *auto*
also have ... = *card ?split1* **by** *auto*
also have ... = (*0*::*real*) **apply**(*simp only*: *split1empty*) **by** *auto*
finally have *abschC*: *C* = (*0*::*real*) **.**

have ttt2: {(x,y). (x,y)∈(Inv ys xs′) − (Inv ys′ xs′′)
∧ y = (q)} ∪ {(x,y). (x,y)∈(Inv ys xs′) − (Inv ys′ xs′′)
∧ y ≠ (q)} = (Inv ys xs′) − (Inv ys′ xs′′) (**is** ?split1
∪ ?split2 = ?easy2) **by** auto
have interem: ?split1 ∩ ?split2 = {} **by** auto
have split1subs: ?split1 ⊆ ?easy2 **by** auto
have split2subs: ?split2 ⊆ ?easy2 **by** auto
have fs1: finite ?split1 **apply**(rule finite_subset[**where** B=?easy2])
apply(rule split1subs) **by**(auto)
have fs2: finite ?split2 **apply**(rule finite_subset[**where** B=?easy2])
apply(rule split2subs) **by**(auto)

from False **have** split1easy2: ∀x∈?split1.
(λx. (if b! (index init (snd x)) then 2::real else 1)) x = (λx.
(1::real)) x **by** force

have ?split1 = (InvOf q ys xs′) − (InvOf q ys′ xs′′)
unfolding Inv_def **by** auto
also have ... = inI **unfolding** InvOfempty **by** auto
finally have splI: ?split1 = inI **.**

have abschaway: (∑(x,y)∈?split2. (if b!(index init y) then 2::real
else 1)) ≥ 0
apply(rule sum_nonneg) **by** auto

have B = (∑(x,y)∈?split1 ∪ ?split2.
(if b!(index init y) then 2::real else 1)) **unfolding** B_def
by(simp only: ttt2)
also have ... = (∑(x,y)∈?split1. (if b!(index init y) then 2::real
else 1))
+ (∑(x,y)∈?split2. (if b!(index init y) then 2::real else 1))
− (∑(x,y)∈?split1 ∩ ?split2. (if b!(index init y) then 2::real
else 1))
by(rule sum_Un[OF fs1 fs2])
also have ... = (∑(x,y)∈?split1. (if b!(index init y) then 2::real
else 1))
+ (∑(x,y)∈?split2. (if b!(index init y) then 2::real else 1))
apply(simp only: interem) **by** auto
also have ... = (∑(x,y)∈?split1. 1)
+ (∑(x,y)∈?split2. (if b!(index init y) then 2::real else 1))
using sum_my2[OF split1easy2] **by** (simp only: split_def)
also have ... = card ?split1
+ (∑(x,y)∈?split2. (if b!(index init y) then 2::real else 1))
by auto

also have ... = I
+ ($\sum$ $(x,y)\in$?*split2*. (*if* b!(*index init y*) *then* 2::*real else* 1))
using *splI* **by** *auto*
also have ... $\geq I$ **using** *abschaway* **by** *auto*
finally have *abschB*: $B \geq I$ **.**

from *abschB abschC* **show** $C - B \leq -I$ **by** *auto*
qed

have *A_absch*: A
$\leq$ (*if* b!(*index init q*) *then* $k-k'$ *else* ($\sum$ $j<k'$. (*if* b!(*index init* (xs'!j)) *then* 2::*real else* 1)))
proof (*cases* b!(*index init q*))
case *False*

from *leninitys False* **have** *ya*: $ys' = mtf2$ (*length ys*) *q ys*
unfolding *step_def ys'_def* **by**(*auto*)
have *index ys'* $q = 0$ **unfolding** *ya* **apply**(*rule mtf2_moves_to_front*)

using *gra2* **by**(*simp_all*)
then have *nixbefore*: *before q ys'* = {} **unfolding** *before_in_def*
by *auto*

have A = ($\sum$ $(x,y)\in$(*Inv ys' xs''*)$-$(*Inv ys xs'*). (*if* b'!(*index init y*) *then* 2::*real else* 1)) **by** *auto*

have *index* (*mtf2* (*free_A* ! *n*) (*q*) (*swaps* (*paid_A* ! *n*) (*s_A n*))) (*q*)
= (*index* (*swaps* (*paid_A* ! *n*) (*s_A n*)) (*q*) $-$ *free_A* ! *n*)
apply(*rule mtf2_q_after*) **using** *queryinlist* **by** *auto*
then have *whatisk'*: k' = *index xs'' q* **by** *auto*

have *ss*: *set ys'* = *set ys* **by** *auto*
have *ss2*: *set xs'* = *set xs''* **by** *auto*

have *di*: *distinct init* **by** *auto*
have *dys*: *distinct ys* **by** *auto*

have (*Inv ys' xs''*)$-$(*Inv ys xs'*)

$= \{(x,y).\ x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge (\sim x < y\ in\ ys \vee \sim y < x\ in\ xs')\}$
unfolding *Inv_def* **by** *auto*
also have ... =
$\{(x,y).\ y \neq q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge (\sim x < y\ in\ ys \vee \sim y < x\ in\ xs')\ \}$
using *nixbefore* **by** *blast*
also have ... =
$\{(x,y).\ x \neq y \wedge y \neq q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge (\sim x < y\ in\ ys \vee \sim y < x\ in\ xs')\ \}$
unfolding *before_in_def* **by** *auto*
also have ... =
$\{(x,y).\ x \neq y \wedge y \neq q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge \sim x < y\ in\ ys\ \}$
$\cup \{(x,y).\ x \neq y \wedge y \neq q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge\ \sim y < x\ in\ xs'\ \}$
by *force*
also have ... =
$\{(x,y).\ x \neq y \wedge y \neq q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge y < x\ in\ ys\ \}$
$\cup \{(x,y).\ x \neq y \wedge y \neq q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge\ \sim y < x\ in\ xs'\ \}$
using *before_in_setD1*[**where** *xs=ys'*] *before_in_setD2*[**where** *xs=ys'*] *not_before_in ss* **by** *metis*
also have ... =
$\{(x,y).\ x \neq y \wedge y \neq q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge y < x\ in\ ys\ \}$
$\cup \{(x,y).\ x \neq y \wedge y \neq q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge\ x < y\ in\ xs'\ \}$ (**is** $?S1 \cup ?S2 = ?S1 \cup ?S2'$)
proof −
have $?S2 = ?S2'$ **apply**(*safe*)
proof (*goal_cases*)
case (*2 a b*)
from *2*(*5*) **have** $\sim b < a\ in\ xs'$ **by** *auto*
with *2*(*6*) **show** *False* **by** *auto*
next
case (*1 a b*)
from *1*(*4*) **have** $a \in set\ xs'$ $b \in set\ xs'$
using *before_in_setD1*[**where** *xs=xs''*]
before_in_setD2[**where** *xs=xs''*] *ss2* **by** *auto*
with *not_before_in 1*(*5*) **have** $(a < b\ in\ xs' \vee a = b)$ **by** *metis*
with *1*(*1*) **show** $a < b\ in\ xs'$ **by** *auto*
qed
then show *?thesis* **by** *auto*
qed

also have ... =
$\{(x,y).\ x{\neq}y \wedge y{\neq}q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge y < x\ in\ ys\ \}$
$\cup \{(x,y).\ x{\neq}y \wedge y{\neq}q \wedge x < y\ in\ ys' \wedge \sim\ x < y\ in\ xs'' \wedge\ x < y\ in\ xs'\ \}$ (**is** $?S1 \cup ?S2 = ?S1 \cup ?S2'$)
proof −
have $?S2 = ?S2'$ **apply**(*safe*)
proof (*goal_cases*)
case (*1 a b*)
from *1*(*4*) **have** $\sim\ a < b\ in\ xs''$ **by** *auto*
with *1*(*6*) **show** *False* **by** *auto*
next
case (*2 a b*)
from *2*(*5*) **have** $a \in set\ xs''\ b \in set\ xs''$
using *before_in_setD1*[**where** *xs=xs'*]
before_in_setD2[**where** *xs=xs'*] *ss2* **by** *auto*
with *not_before_in 2*(*4*) **have** ($b < a\ in\ xs'' \vee a = b$) **by** *metis*
with *2*(*1*) **show** $b < a\ in\ xs''$ **by** *auto*
qed
then show *?thesis* **by** *auto*
qed
also have ... =
$\{(x,y).\ x{\neq}y \wedge y{\neq}q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge y < x\ in\ ys\ \}$
$\cup \{\}$
using *x_stays_before_y_if_y_not_moved_to_front*[**where** *xs=xs'* **and** *q=q*]
before_in_setD1[**where** *xs=xs'*] *before_in_setD2*[**where** *xs=xs'*] **by** (*auto simp*: *queryinlist*)
also have ... =
$\{(x,y).\ x{\neq}y \wedge x{=}q \wedge y{\neq}q \wedge x < y\ in\ ys' \wedge y < x\ in\ xs'' \wedge y < x\ in\ ys\ \}$
apply(*simp only*: *ya*) **using** *swapped_by_mtf2*[**where** *xs=ys* **and** *q=q* **and** *n=*(*length ys*)] *dys*
before_in_setD1[**where** *xs=ys*] *before_in_setD2*[**where** *xs=ys*] **by** (*auto simp*: *queryinlist*)
also have ... ⊆
$\{(x,y).\ x{=}q \wedge y{\neq}q \wedge q < y\ in\ ys' \wedge y < q\ in\ xs''\}$ **by** *force*
also have ... =
$\{(x,y).\ x{=}q \wedge y{\neq}q \wedge q < y\ in\ ys' \wedge y < q\ in\ xs'' \wedge y \in set\ xs''\}$
using *before_in_setD1* **by** *metis*
also have ... =
$\{(x,y).\ x{=}q \wedge y{\neq}q \wedge q < y\ in\ ys' \wedge index\ xs''\ y < index\ xs''\ q \wedge q \in set\ xs'' \wedge y \in set\ xs''\}$ **unfolding** *before_in_def* **by** *auto*

also have ... =
$\{(x,y).\ x=q \wedge y \neq q \wedge q < y\ in\ ys' \wedge index\ xs''\ y < index\ xs'\ q - (free\_A\ !\ n) \wedge q \in set\ xs'' \wedge y \in set\ xs''\}$
using *mtf2_q_after*[**where** $A=xs'$ **and** $q=q$] **by** *force*
also have ... $\subseteq$
$\{(x,y).\ x=q \wedge y \neq q \wedge index\ xs'\ y < index\ xs'\ q - (free\_A\ !\ n) \wedge y \in set\ xs''\}$
using *mtf2_backwards_effect4'*[**where** $xs=xs'$ **and** $q=q$ **and** $n=(free\_A\ !\ n)$, *simplified*]
by *auto*
also have ... $\subseteq$
$\{(x,y).\ x=q \wedge y \neq q \wedge index\ xs'\ y < k'\}$
using *mtf2_q_after*[**where** $A=xs'$ **and** $q=q$] **by** *auto*

finally have *subsa*: $(Inv\ ys'\ xs'') - (Inv\ ys\ xs')$
$\subseteq \{(x,y).\ x=q \wedge y \neq q \wedge index\ xs'\ y < k'\}$ **.**

have *k'xs'*: $k' < length\ xs''$ **unfolding** *whatisk'*
apply(*rule index_less*) **by** (*auto simp*: *queryinlist*)
then have *k'xs'*: $k' < length\ xs'$ **by** *auto*

have $\{(x,y).\ x=q \wedge index\ xs'\ y < k'\}$
$\subseteq \{(x,y).\ x=q \wedge index\ xs'\ y < length\ xs'\}$ **using** *k'xs'* **by** *auto*
also have ... $= \{(x,y).\ x=q \wedge y \in set\ xs'\}$
using *index_less_size_conv* **by** *fast*
finally have $\{(x,y).\ x=q \wedge index\ xs'\ y < k'\} \subseteq \{(x,y).\ x=q \wedge y \in set\ xs'\}$ **.**
then have *finia2*: *finite* $\{(x,y).\ x=q \wedge index\ xs'\ y < k'\}$
apply(*rule finite_subset*) **by**(*simp*)

have *lulae*: $\{(a,b).\ a=q \wedge index\ xs'\ b < k'\}$
$= \{(q,b)|b.\ index\ xs'\ b < k'\}$ **by** *auto*

have *k'b*: $k' < length\ b$ **using** *whatisk'* **by** (*auto simp*: *queryinlist*)
have *asdasd*: $\{(\alpha,\beta).\ \alpha=q \wedge \beta \neq q \wedge index\ xs'\ \beta < k'\}$
$= \{(\alpha,\beta).\ \alpha=q \wedge \beta \neq q \wedge index\ xs'\ \beta < k' \wedge (index\ init\ \beta) < length\ b\ \}$
proof (*auto*, *goal_cases*)
case (*1 b*)
from *1(2)* **have** $index\ xs'\ b < index\ xs'\ (q)$ **by** *auto*
also have ... $< length\ xs'$ **by** (*auto simp*: *queryinlist*)
finally have $b \in set\ xs'$ **using** *index_less_size_conv* **by** *metis*
then show *?case* **using** *setinit* **by** *auto*

qed

{ fix *β*
have $\beta \neq q \Longrightarrow$ (*index init* β)$\neq$(*index init* q)
using *queryinlist* **by** *auto*
} note *ij=this*
have *subsa2*: $\{(\alpha,\beta).\ \alpha=q \wedge \beta\neq q \wedge$ *index xs'* $\beta < k'\} \subseteq$
$\{(\alpha,\beta).\ \alpha=q \wedge$ *index xs'* $\beta < k'\}$ **by** *auto*
then have *finia*: *finite* $\{(x,y).\ x=q \wedge y\neq q \wedge$ *index xs'* $y < k'\}$
apply(*rule finite_subset*) **using** *finia2* **by** *auto*

have *E0*: $A = (\sum (x,y)\in($*Inv ys' xs''*$)-($*Inv ys xs'*$).$ (*if b'!*(*index init y*) *then 2::real else 1*)) **by** *auto*
also have *E1*: $\ldots \leq (\sum (x,y)\in\{(a,b).\ a=q \wedge b\neq q \wedge$ *index xs'* $b < k'\}.$ (*if b'!*(*index init y*) *then 2::real else 1*))
unfolding *A_def* **apply**(*rule sum_mono2*[*OF finia subsa*]) **by** *auto*
also have $\ldots = (\sum (x,y)\in\{(\alpha,\beta).\ \alpha=q \wedge \beta\neq q \wedge$ *index xs'* $\beta < k'$ $\wedge$ (*index init* β) $<$ *length b* $\}.$ (*if b'!*(*index init y*) *then 2::real else 1*))
using *asdasd* **by** *auto*
also have $\ldots = (\sum (x,y)\in\{(\alpha,\beta).\ \alpha=q \wedge \beta\neq q \wedge$ *index xs'* $\beta < k'$ $\wedge$ (*index init* β) $<$ *length b* $\}.$ (*if b!*(*index init y*) *then 2::real else 1*))
proof (*rule sum.cong*, *goal_cases*)
case (*2 z*)
then obtain α β **where** *zab*: $z=(\alpha,\ \beta)$ **and** $\alpha = q$ **and** *diff*: $\beta \neq q$ **and** *index xs'* $\beta < k'$ **and** *i*: *index init* $\beta <$ *length b* **by** *auto*
from *diff ij* **have** *index init* $\beta \neq$ *index init q* **by** *auto*
with *flip_other qsfst i* **have** *b'* ! *index init* $\beta =$ *b* ! *index init* β **unfolding** *b'_def* **by** *auto*
with *zab* **show** *?case* **by**(*auto simp add*: *split_def*)
qed *simp*
also have *E1a*: $\ldots = (\sum (x,y)\in\{(a,b).\ a=q \wedge b\neq q \wedge$ *index xs'* $b < k'\}.$ (*if b!*(*index init y*) *then 2::real else 1*))
using *asdasd* **by** *auto*
also have $\ldots \leq (\sum (x,y)\in\{(a,b).\ a=q \wedge$ *index xs'* $b < k'\}.$ (*if b!*(*index init y*) *then 2::real else 1*))
apply(*rule sum_mono2*[*OF finia2 subsa2*]) **by** *auto*
also have *E2*: $\ldots = (\sum (x,y)\in\{(q,b)|b.$ *index xs'* $b < k'\}.$ (*if b!*(*index init y*) *then 2::real else 1*))
by (*simp only*: *lulae*[*symmetric*])
finally have *aa*: $A \leq (\sum (x,y)\in\{(q,b)|b.$ *index xs'* $b < k'\}.$ (*if b!*(*index init y*) *then 2::real else 1*)) **.**

```
        have sameset: {y. index xs' y < k'} = {xs'!i | i. i < k'}
          proof (safe, goal_cases)
            case (1 z)
            show ?case
              proof
                from 1(1) have index xs' z < index (swaps (paid_A ! n) (s_A n)) (q)
                by auto
                also have ... < length xs' using index_less_size_conv by (auto simp: queryinlist)
              finally have index xs' z  < length xs' .
               then have zset: z ∈ set xs' using index_less_size_conv by metis
              have f1: xs' ! (index xs' z) = z
                apply(rule nth_index) using zset by auto
              show z = xs' ! (index xs' z) ∧ (index xs' z) < k'
              using f1 1(1)  by auto
            qed
          next
            case (2 k i)
            from 2(1) have i < index (swaps (paid_A ! n) (s_A n)) (q)
              by auto
          also have ... < length xs' using index_less_size_conv by (auto simp: queryinlist)
            finally have iset: i < length xs' .
            have index xs' (xs' ! i) = i apply(rule index_nth_id)
              using iset by(auto)
            with 2 show ?case by auto
          qed

        have aaa23: inj_on (λi. xs'!i) {i. i < k'}
          apply(rule inj_on_nth)
            apply(simp)
            apply(simp) proof (safe, goal_cases)
              case (1 i)
              then have i < index xs' (q) by auto
                also have ... < length xs' using index_less_size_conv by (auto simp: queryinlist)
              also have ... = length init by auto
              finally show  i < length init .
            qed
```

have *aa3*: $\{xs'!i \mid i.\ i < k'\} = (\lambda i.\ xs'!i)$ ‘ $\{i.\ i < k'\}$ **by** *auto*
have *aa4*: $\{(q,b)|b.\ index\ xs'\ b < k'\} = (\lambda b.\ (q,b))$ ‘ $\{b.\ index\ xs'\ b < k'\}$ **by** *auto*

have *unbelievable*: $\{i::nat.\ i < k'\} = \{..<k'\}$ **by** *auto*

have *aadad*: *inj_on* $(\lambda b.\ (q,b))$ $\{b.\ index\ xs'\ b < k'\}$
unfolding *inj_on_def* **by**(*simp*)

have $(\sum (x,y)\in\{(q,b)|b.\ index\ xs'\ b < k'\}.\ (if\ b!(index\ init\ y)\ then\ 2::real\ else\ 1))$
$= (\sum y\in\{y.\ index\ xs'\ y < k'\}.\ (if\ b!(index\ init\ y)\ then\ 2::real\ else\ 1))$
proof –
have $(\sum (x,y)\in\{(q,b)|b.\ index\ xs'\ b < k'\}.\ (if\ b!(index\ init\ y)\ then\ 2::real\ else\ 1))$
$= (\sum (x,y)\in (\lambda b.\ (q,b))$ ‘ $\{b.\ index\ xs'\ b < k'\}.\ (if\ b!(index\ init\ y)\ then\ 2::real\ else\ 1))$ **using** *aa4* **by** *simp*
also have $\ldots = (\sum z\in (\lambda b.\ (q,b))$ ‘ $\{b.\ index\ xs'\ b < k'\}.\ (if\ b!(index\ init\ (snd\ z))\ then\ 2::real\ else\ 1))$ **by** (*simp add*: *split_def*)
also have $\ldots = (\sum z\in\{b.\ index\ xs'\ b < k'\}.\ (if\ b!(index\ init\ (snd\ ((\lambda b.\ (q,b))\ z)))\ then\ 2::real\ else\ 1))$
apply(*simp only*: *sum.reindex*[*OF aadad*]) **by** *auto*
also have $\ldots = (\sum y\in\{y.\ index\ xs'\ y < k'\}.\ (if\ b!(index\ init\ y)\ then\ 2::real\ else\ 1))$ **by** *auto*
finally show *?thesis* **.**
qed
also have $\ldots = (\sum y\in\{xs'!i \mid i.\ i < k'\}.\ (if\ b!(index\ init\ y)\ then\ 2::real\ else\ 1))$ **using** *sameset* **by** *auto*
also have $\ldots = (\sum y\in(\lambda i.\ xs'!i)$ ‘ $\{i.\ i < k'\}.\ (if\ b!(index\ init\ y)\ then\ 2::real\ else\ 1))$ **using** *aa3* **by** *simp*
also have $\ldots = (\sum y\in\{i::nat.\ i < k'\}.\ (if\ b!(index\ init\ (xs'!y))\ then\ 2::real\ else\ 1))$
using *sum.reindex*[*OF aaa23*] **by** *simp*
also have *E3*: $\ldots = (\sum j::nat<k'.\ (if\ b!(index\ init\ (xs'!j))\ then\ 2::real\ else\ 1))$
using *unbelievable* **by** *auto*
finally have *bb*: $(\sum (x,y)\in\{(q,b)|b.\ index\ xs'\ b < k'\}.\ (if\ b!(index\ init\ y)\ then\ 2::real\ else\ 1))$
$= (\sum j<k'.\ (if\ b!(index\ init\ (xs'!j))\ then\ 2::real\ else\ 1))$ **.**

have $A \le (\sum j<k'.\ (if\ b!(index\ init\ (xs'!j))\ then\ 2::real\ else\ 1))$
using *aa bb* **by** *linarith*

then show *A*
$\leq$ (*if b*!(*index init q*) *then* $k-k'$ *else* ($\sum j<k'$. (*if b*!(*index init* (*xs*'!*j*)) *then 2*::*real else 1*)))
using *False* **by** *auto*

next
case *True*

then have *samesame*: *ys*' = *ys* **unfolding** *ys'_def step_def* **by** *auto*

have *setxsbleibt*: *set xs*'' = *set init* **by** *auto*

have *whatisk*': *k*' = *index xs*'' *q* **apply**(*simp*)
apply(*rule mtf2_q_after*[*symmetric*]) **using** *queryinlist* **by** *auto*

have (*Inv ys*' *xs*'')−(*Inv ys xs*')
= {(*x*,*y*). *x* < *y in ys* ∧ *y* < *x in xs*'' ∧ ~ *y* < *x in xs*'}
unfolding *Inv_def* **using** *samesame* **by** *auto*
also have
... ⊆ {(*xs*'!*i*,*q*)|*i*. *i*∈{*k*'..<*k*}}
apply(*clarify*)
proof
fix *a b*
assume *1*: *a* < *b in ys*
and *2*: *b* < *a in xs*''
and *3*: ¬ *b* < *a in xs*'
then have *anb*: *a* ≠ *b*
using *no_before_inI* **by**(*force*)
have *a*: *a* ∈ *set init*
and *b*: *b* ∈ *set init*
using *before_in_setD1*[*OF 1*] *before_in_setD2*[*OF 1*] **by** *auto*
with *anb 3* **have** *3*: *a* < *b in xs*'
by (*simp add*: *not_before_in*)
note *all*= *anb 1 2 3 a b*
have *bq*: *b*=*q* **apply**(*rule swapped_by_mtf2*[**where** *xs*=*xs*' **and** *x*=*a*])
using *queryinlist* **apply**(*simp_all add*: *all*)
using *all*(*4*) **apply**(*simp*)
using *all*(*3*) **apply**(*simp*) **done**

```
        note mine=mtf2_backwards_effect3[THEN conjunct1]

        from bq have q < a in xs'' using 2 by auto
        then have (k' < index xs'' a ∧ a ∈ set xs'')
          unfolding before_in_def
            using  whatisk' by auto
        then have low : k' ≤ index xs' a
          unfolding whatisk'
           unfolding xs''_def
           apply(subst mtf2_q_after)
             apply(simp)
            using queryinlist apply(simp)
           apply(rule mine)
              apply (simp add: queryinlist)
             using bq b apply(simp)
            apply(simp)
           apply(simp del: xs'_def)
           apply (metis 3 a before_in_def bq dp_xs'_init k'_def k_def
max_0L mtf2_forward_beforeq nth_index whatisk' xs''_def)
          using a by(simp)
        from bq have a < q in xs' using 3 by auto
        then have up: (index xs' a < k )
          unfolding before_in_def by auto

        from a have a ∈ set xs' by simp
        then have aa: a = xs'!index xs' a using nth_index by simp

        have inset: index xs' a ∈ {k'..<k}
          using low up by fastforce

        from bq aa show (a, b) = (xs' ! index xs' a, q) ∧ index xs' a ∈
{k'..<k}
          using inset by simp
      qed
    finally have a: (Inv ys' xs'')−(Inv ys xs') ⊆ {(xs'!i,q)|i. i∈{k'..<k}}
(is ?M ⊆ ?UB) .

      have card_of_UB: card {(xs'!i,q)|i. i∈{k'..<k}} = k−k'
      proof −
        have e: fst ‘ ?UB = (%i. xs' ! i) ‘ {k'..<k} by force
        have card ?UB = card (fst ‘ ?UB)
            apply(rule card_image[symmetric])
                using inj_on_def by fastforce
      also
```

```
      have ... = card ((%i. xs' ! i) ` {k'..<k})
        by (simp only: e)
    also
      have ... = card {k'..<k}
            apply(rule card_image)
            apply(rule inj_on_nth)
              using k_inbounds by simp_all
    also
      have ... = k-k' by auto
    finally
      show ?thesis .
    qed

     have flipit: flip (index init q) b ! (index init q) = (~ (b) ! (index
init q)) apply(rule flip_itself)
      using queryinlist setinit by auto


    have q: {x∈?UB. snd x=q} = ?UB by auto

   have E0: A = (∑(x,y)∈(Inv ys' xs'')-(Inv ys xs'). (if b'!(index init
y) then 2::real else 1)) by auto
   also have E1: ... ≤ (∑(z,y)∈?UB. if flip (index init q) (b) ! (index
init y) then 2::real else 1)
        unfolding b'_def apply(rule sum_mono2[OF _ a])
          by(simp_all add: split_def)
    also have ... = (∑(z,y)∈{x∈?UB. snd x=q}. if flip (index init q)
(b) ! (index init y) then 2::real else 1) by(simp only: q)
    also have ... = (∑z∈{x∈?UB. snd x=q}. if flip (index init q) (b)
! (index init (snd z)) then 2::real else 1) by(simp add: split_def)
    also have ... = (∑z∈{x∈?UB. snd x=q}. if flip (index init q) (b)
! (index init q) then 2::real else 1) by simp
     also have E2: ... = (∑z∈?UB. if flip (index init q) (b) ! (index
init q) then 2::real else 1) by(simp only: q)
    also have E3: ... = (∑y∈?UB. 1) using flipit True by simp
    also have E4: ... = k-k'
       by(simp only: real_of_card[symmetric] card_of_UB)
    finally have result: A ≤ k-k' .
    with True show ?thesis by auto
   qed


  show (∑(x,y)∈(Inv ys' xs''). (if b'!(index init y) then 2::real else 1))
- (∑(x,y)∈(Inv ys xs'). (if b!(index init y) then 2::real else 1)) ≤ ?ub2
```

unfolding *ub_free_def teilen*[*unfolded* Δ*_def A_def B_def C_def*] **using** *BC_absch A_absch* **using** *True*
by *auto*
qed
from *paid_ub* **have** *kl*: $\Phi_1\ x \leq \Phi_0\ x\ +$ *?paidUB* **by** *auto*
from *free_ub* **have** *kl2*: $\Phi_2\ x -$ *?ub2* $\leq \Phi_1\ x$ **using** *gis dis* **by** *auto*

have *iub_free*: *I* + *?ub2* = *ub_free* **by** *auto*

from *kl kl2* **have** $\Phi_2\ x - \Phi_0\ x \leq$ *?ub2* + *?paidUB* **by** *auto*

then have (*cost x*) + (Φ_2 *x*) − (Φ_0 *x*) ≤ *k* + *1* + *I* + *?ub2* + *?paidUB* **using** *ub_cost_BIT* **by** *auto*

then show *?case* **unfolding** *ub_free_def b_def* **by** *auto*
qed

Approximation of the Term for Free exchanges

have *free_absch*: *E*(*map_pmf* (λ*x*. (*if* (*q*) ∈ *set init then* (*if* (*fst* (*snd x*))!(*index init q*) *then k*−*k*′
else ($\sum$*j*<*k*′. (*if* (*fst* (*snd x*))!(*index init* (*xs*′!*j*)) *then 2*::*real else 1*))) *else 0*)) *D*)
≤ *3/4* ∗ *k* (**is** *?EA* ≤ *?absche*)
proof (*cases* (*q*) ∈ *set init*)
case *False*

then have *?EA* = *0* **by** *auto*
then show *?thesis* **by** *auto*
next
case *True*

note *queryinlist*=*this*

have *k*−*k*′ ≤ *k* **by** *auto*
have *k*′ ≤ *k* **by** *auto*

Transformation of the first term

have *qsn*: {*index init q*} ∪ {} ⊆ {*0*..<*?l*} **using** *setinit queryinlist* **by** *auto*

have {*l*::*bool list*. *length l* = *?l* ∧ *l*!(*index init q*)}
= {*xs*. *Ball* {(*index init q*)} ((!) *xs*) ∧ (∀ *i*∈{}. ¬ *xs* ! *i*) ∧ *length xs*

= *?l*} **by** *auto*
then have *card* {*l::bool list. length l* = *?l* ∧ *l*!(*index init q*)}
= *card* {*xs. Ball* {*index init q*} ((!) *xs*) ∧ (∀ *i*∈{}. ¬ *xs* ! *i*) ∧ *length xs* = *length init*} **by** *auto*
also have . . . = *2*^(*length init* − *card* {*index init q*} − *card* {})
apply(*subst card2*[*of* {(*index init q*)} {} *?l*]) **using** *qsn* **by** *auto*
finally have *lulu*: *card* {*l::bool list. length l* = *?l* ∧ *l*!(*index init q*)} = *2*^(*?l*−*1*) **by** *auto*

have (∑ *x*∈{*l::bool list. length l* = *?l* ∧ *l*!(*index init q*)}. *real*(*k*−*k′*))
= (∑ *x*∈{*l::bool list. length l* = *?l* ∧ *l*!(*index init q*)}. *k*−*k′*) **by** *auto*
also have . . . = (*k*−*k′*)∗*2*^(*?l*−*1*) **using** *lulu* **by** *simp*

finally have *absch1stterm*: (∑ *x*∈{*l::bool list. length l* = *?l* ∧ *l*!(*index init q*)}. *real*(*k*−*k′*))
= *real*((*k*−*k′*)∗*2*^(*?l*−*1*)) **.**

Transformation of the second term

let *?S*={(*xs′*!*j*)|*j. j*<*k′*}

from *queryinlist* **have** *q* ∈ *set* (*swaps* (*paid_A* ! *n*) (*s_A n*)) **by** *auto*
then have *index* (*swaps* (*paid_A* ! *n*) (*s_A n*)) *q* < *length xs′* **by** *auto*
then have *k′inbound*: *k′* < *length xs′* **by** *auto*

{ fix *x*
have *a*: {..<*k′*} = {*j. j*<*k′*} **by** *auto*
have *b*: *?S* = ((%*j. xs′*!*j*) ‘ {*j. j*<*k′*}) **by** *auto*

have (∑ *j*<*k′*. (λ*t.* (*if x*!(*index init t*) *then 2::real else 1*)) (*xs′*!*j*))
= *sum* ((λ*t.* (*if x*!(*index init t*) *then 2::real else 1*)) *o* (%*j. xs′*!*j*)) {..<*k′*}
by(*auto*)
also have . . . = *sum* ((λ*t.* (*if x*!(*index init t*) *then 2::real else 1*)) *o* (%*j. xs′*!*j*)) {*j. j*<*k′*}
by (*simp only*: *a*)
also have . . . = *sum* (λ*t.* (*if x*!(*index init t*) *then 2::real else 1*)) ((%*j. xs′*!*j*) ‘ {*j. j*<*k′*})
apply(*rule sum.reindex*[*symmetric*])
apply(*rule inj_on_nth*)
using *k′inbound* **by**(*simp_all*)

finally have ($\sum j<k'$. (λt. (*if* x!(*index init t*) *then 2::real else 1*)) (xs'!j))

= ($\sum j\in$*?S*. (λt. (*if* x!(*index init t*) *then 2 else 1*)) j) **using** b **by** *simp*

} **note** *reindex=this*

have *identS*: *?S* = *set* (*take* k' xs')

proof –

have *index* (*swaps* (*paid_A* ! *n*) (*s_A n*)) (q) $\leq$ *length* (*swaps* (*paid_A* ! *n*) (*s_A n*))

by (*rule index_le_size*)

then have kxs': $k' \leq$ *length* xs' **by** *simp*

have *?S* = (!) xs' ' {*0..<*k'} **by** *force*

also have ... = *set* (*take* k' xs') **apply**(*rule nth_image*) **by**(*rule* kxs')

finally show *?S* = *set* (*take* k' xs') **.**

qed

have *distinctS*: *distinct* (*take* k' xs') **using** *distinct_take identS* **by** *simp*

have *lengthS*: *length* (*take* k' xs') = k' **using** *length_take* k'*inbound* **by** *simp*

from *distinct_card*[*OF distinctS*] *lengthS* **have** *card* (*set* (*take* k' xs')) = k' **by** *simp*

then have *cardS*: *card ?S* = k' **using** *identS* **by** *simp*

have *a*: *?S* $\subseteq$ *set* xs' **using** *set_take_subset identS* **by** *metis*

then have *Ssubso*: (*index init*) ' *?S* $\subseteq$ {*0..<?l*} **using** *setinit* **by** *auto*

from *a* **have** *s_subst_init*: *?S* $\subseteq$ *set init* **by** *auto*

note *index_inj_on_S=subset_inj_on*[*OF inj_on_index*[*of init*] *s_subst_init*]

have *l*: xs'!k = q **unfolding** *k_def* **apply**(*rule nth_index*) **using** *queryinlist* **by**(*auto*)

have xs'!k $\notin$ *set* (*take* k' xs')

apply(*rule index_take*) **using** *l* **by** *simp*

then have *requestnotinS*: (q) $\notin$ *?S* **using** *l identS* **by** *simp*

then have *indexnotin*: *index init q* $\notin$ (*index init*) ' *?S*

using *index_inj_on_S s_subst_init* **by** *auto*

have *lua*: {*l*. *length l* = *?l* $\wedge$ ~*l*!(*index init q*)}

= {*xs*. ($\forall i\in${}. *xs* ! *i*) $\wedge$ ($\forall i\in${*index init q*}. $\neg$ *xs* ! *i*) $\wedge$ *length xs*

= *?l*} **by** *auto*

from *k′inbound* **have** *k′inbound2*: *Suc k′* ≤ *length init* **using** *Suc_le_eq* **by** *auto*

have (∑ *x*∈{*l*::*bool list*. *length l* = *?l* ∧ ~*l*!(*index init q*)}. (∑ *j*<*k′*. (*if x*!(*index init* (*xs′*!*j*)) *then 2*::*real else 1*)))

= (∑ *x*∈{*l*. *length l* = *?l* ∧ ~*l*!(*index init q*)}. (∑ *j*∈*?S*. (λ*t*. (*if x*!(*index init t*) *then 2 else 1*)) *j*))
using *reindex* **by** *auto*

also
have ... = (∑ *x*∈{*xs*. (∀ *i*∈{}. *xs* ! *i*) ∧ (∀ *i*∈{*index init q*}. ¬ *xs* ! *i*) ∧ *length xs* = *?l*}. (∑ *j*∈*?S*. (λ*t*. (*if x*!(*index init t*) *then 2 else 1*)) *j*))
using *lua* **by** *auto*
also
have ... = (∑ *x*∈{*xs*. (∀ *i*∈{}. *xs* ! *i*) ∧ (∀ *i*∈{*index init q*}. ¬ *xs* ! *i*) ∧ *length xs* = *?l*}. (∑ *j*∈(*index init*) ‘ *?S*. (λ*t*. (*if x*!*t then 2 else 1*)) *j*))
proof −
{ **fix** *x*
have (∑ *j*∈*?S*. (λ*t*. (*if x*!(*index init t*) *then 2 else 1*)) *j*)
= (∑ *j*∈(*index init*) ‘ *?S*. (λ*t*. (*if x*!*t then 2 else 1*)) *j*)
apply(*simp only*: *sum.reindex*[*OF index_inj_on_S*, **where** *g*=(%*j*. *if x* ! *j then 2 else 1*)])
by(*simp*)
} **note** *a*=*this*
show *?thesis* **by**(*simp only*: *a*)
qed

also
have ... = *3* / *2* ∗ *real* (*card ?S*) ∗ *2* ^ (*?l* − *card* {} − *card* {*q*})
apply(*subst Expactation2or1*)
apply(*simp*)
apply(*simp*)
apply(*simp*)
apply(*simp only*: *card_image index_inj_on_S cardS*) **apply**(*simp add*: *k′inbound2 del*: *k′_def*)
using *indexnotin* **apply** *simp*
apply(*simp*)

using *Ssubso queryinlist* **apply**(*simp*)
apply(*simp only*: *card_image*[*OF index_inj_on_S*]) **by** *simp*
finally have ($\sum x \in \{l.\ length\ l = ?l \wedge \neg\ l\ !\ (index\ init\ q)\}.\ \sum j<k'.$ *if* $x\ !\ (index\ init\ (xs'\ !\ j))$ *then 2 else 1*)
$= 3\ /\ 2 *\ real\ (card\ ?S) * 2\ \hat{}\ (?l - card\ \{\} - card\ \{q\})$ **.**

also
have $3\ /\ 2 *\ real\ (card\ ?S) *\ 2\ \hat{}\ (?l - card\ \{\} - card\ \{q\}) = (3/2) * (real\ (k')) *\ 2\ \hat{}\ (?l - 1)$ **using** *cardS* **by** *auto*

finally have *absch2ndterm*: ($\sum x \in \{l.\ length\ l = ?l \wedge\ \neg\ l\ !\ (index\ init\ q)\}.$
$\sum j<k'.$ *if* $x\ !(index\ init\ (xs'\ !\ j))$ *then 2 else 1*) $=$
$3\ /\ 2 * real\ (k') * 2\ \hat{}\ (?l - 1)$ **.**

Equational transformations to the goal

have *cardonebitset*: $card\ \{l::bool\ list.\ length\ l = ?l \wedge l!(index\ init\ q)\} = 2\hat{}(?l-1)$ **using** *lulu* **by** *auto*

have *splitie*: $\{l::bool\ list.\ length\ l = ?l\}$
$= \{l::bool\ list.\ length\ l = ?l \wedge l!(index\ init\ q)\} \cup \{l::bool\ list.\ length\ l = ?l \wedge \sim l!(index\ init\ q)\}$
by *auto*
have *interempty*: $\{l::bool\ list.\ length\ l = ?l \wedge l!(index\ init\ q)\} \cap \{l::bool\ list.\ length\ l = ?l \wedge \sim l!(index\ init\ q)\}$
$= \{\}$ **by** *auto*
have *fa*: *finite* $\{l::bool\ list.\ length\ l = ?l \wedge l!(index\ init\ q)\}$ **using** *bitstrings_finite* **by** *auto*
have *fb*: *finite* $\{l::bool\ list.\ length\ l = ?l \wedge \sim l!(index\ init\ q)\}$ **using** *bitstrings_finite* **by** *auto*

{ fix *f* :: *bool list* ⇒ *real*
have ($\sum x \in \{l::bool\ list.\ length\ l = ?l\}.\ f\ x$)
$= (\sum x \in \{l::bool\ list.\ length\ l = ?l \wedge l!(index\ init\ q)\} \cup \{l::bool\ list.\ length\ l = ?l \wedge \sim l!(index\ init\ q)\}.\ f\ x)$ **by**(*simp only*: *splitie*)
also have ...
$= (\sum x \in \{l::bool\ list.\ length\ l = ?l \wedge l!(index\ init\ q)\}.\ f\ x)$
$+ (\sum x \in \{l::bool\ list.\ length\ l = ?l \wedge \sim l!(index\ init\ q)\}.\ f\ x)$
$- (\sum x \in \{l::bool\ list.\ length\ l = ?l \wedge l!(index\ init\ q)\} \cap \{l::bool\ list.\ length\ l = ?l \wedge \sim l!(index\ init\ q)\}.\ f\ x)$
using *sum_Un*[*OF fa fb*, *of f*] **by** *simp*
also have ... $= (\sum x \in \{l::bool\ list.\ length\ l = ?l \wedge l!(index\ init\ q)\}.$

f x)
+ (∑*x*∈{*l*::*bool list*. *length l* = *?l* ∧ ~*l*!(*index init q*)}. *f x*) **by**(*simp add*: *interempty*)
finally have *sum f* {*l*. *length l* = *length init*} =
sum f {*l*. *length l* = *length init* ∧ *l* ! (*index init q*)} + *sum f* {*l*. *length l* = *length init* ∧ ¬ *l* ! (*index init q*)} **.**
} **note** *darfstsplitten*=*this*

have *E1*: *E*(*map_pmf* (λ*x*. (*if* (*fst* (*snd x*))!(*index init q*) *then real*(*k*−*k*′) *else* (∑*j*<*k*′. (*if* (*fst* (*snd x*))!(*index init* (*xs*′!*j*)) *then 2*::*real else 1*)))) *D*)
= *E*(*map_pmf* (λ*x*. (*if x*!(*index init q*) *then real*(*k*−*k*′) *else* (∑*j*<*k*′. (*if x*!(*index init* (*xs*′!*j*)) *then 2*::*real else 1*)))) (*map_pmf* (*fst* ∘ *snd*) *D*))
proof −
have *triv*: ⋀*x*. (*fst* ∘ *snd*) *x* = *fst* (*snd x*) **by** *simp*
have *E*((*map_pmf* (λ*x*. (*if* (*fst* (*snd x*))!(*index init q*) *then real*(*k*−*k*′) *else* (∑*j*<*k*′. (*if* (*fst* (*snd x*))!*index init* (*xs*′!*j*) *then 2*::*real else 1*))))) *D*)
= *E*(*map_pmf* (λ*x*. ((λ*y*. (*if y*!(*index init q*) *then real*(*k*−*k*′) *else* (∑*j*<*k*′. (*if y*!*index init* (*xs*′!*j*) *then 2*::*real else 1*)))) ∘ (*fst* ∘ *snd*)) *x*) *D*)
apply(*auto simp*: *comp_assoc*) **by** (*simp only*: *triv*)
also have ... = *E*((*map_pmf* (λ*x*. (*if x*!(*index init q*) *then real*(*k*−*k*′) *else* (∑*j*<*k*′. (*if x*!*index init* (*xs*′!*j*) *then 2*::*real else 1*)))) ∘ (*map_pmf* (*fst* ∘ *snd*))) *D*)
using *map_pmf_compose* **by** *metis*
also have ... = *E*(*map_pmf* (λ*x*. (*if x*!(*index init q*) *then real*(*k*−*k*′) *else* (∑*j*<*k*′. (*if x*!*index init* (*xs*′!*j*) *then 2*::*real else 1*)))) (*map_pmf* (*fst* ∘ *snd*) *D*)) **by** *auto*
finally show *?thesis* **.**
qed
also
have *E2*: ... = *E*(*map_pmf* (λ*x*. (*if x*!(*index init q*) *then real*(*k*−*k*′) *else* (∑*j*<*k*′. (*if x*!(*index init* (*xs*′!*j*)) *then 2*::*real else 1*)))) (*bv ?l*))
using *config_n_bv*[*of init* _] **by** *auto*
also
let *?insf*=(λ*x*. (*if x*!(*index init q*) *then k*−*k*′ *else* (∑*j*<*k*′. (*if x*!(*index init* (*xs*′!*j*)) *then 2*::*real else 1*))))
have *E3*: ... = (∑*x*∈(*set_pmf* (*bv ?l*)). (*?insf x*) ∗ *pmf* (*bv ?l*) *x*)
by (*subst E_finite_sum_fun*) (*auto simp*: *bv_finite mult_ac*)
also
have ... = (∑*x*∈{*l*::*bool list*. *length l* = *?l*}. (*?insf x*) ∗ *pmf* (*bv ?l*)

x)
    using *bv_set* **by** *auto*
    also
    have *E4*: ... = ($\sum$ *x*∈{*l*::*bool list*. *length l* = *?l*}. (*?insf x*) ∗ (*1/2*)^*?l*)
    by (*simp add*: *list_pmf*)
    also
    have ... = ($\sum$ *x*∈{*l*::*bool list*. *length l* = *?l*}. (*?insf x*)) ∗ ((*1/2*)^*?l*)
    by(*simp only*: *sum_distrib_right*[**where** *r*=(*1/2*)^*?l*])
    also
    have *E5*: ... = ((*1/2*)^*?l*) ∗($\sum$ *x*∈{*l*::*bool list*. *length l* = *?l*}. (*?insf x*))
    by(*auto*)
    also
    have *E6*: ... = ((*1/2*)^*?l*) ∗ (($\sum$ *x*∈{*l*::*bool list*. *length l* = *?l* ∧ *l*!(*index init q*)}. *?insf x*)
    + ($\sum$ *x*∈{*l*::*bool list*. *length l* = *?l* ∧ ~*l*!(*index init q*)}. *?insf x*)
) **using** *darfstsplitten* **by** *auto*
    also
    have *E7*: ... = ((*1/2*)^*?l*) ∗ (($\sum$ *x*∈{*l*::*bool list*. *length l* = *?l* ∧ *l*!(*index init q*)}. ((λ*x*. *real*(*k*−*k'*))) *x*)
    + ($\sum$ *x*∈{*l*::*bool list*. *length l* = *?l* ∧ ~*l*!(*index init q*)}. ((λ*x*. ($\sum$ *j*<*k'*. (*if x*!*index init* (*xs'*!*j*) *then 2*::*real else 1*)))) *x*)
) **by** *auto*
    finally have *E*(*map_pmf* (λ*x*. (*if* (*fst* (*snd x*))!(*index init q*) *then real*(*k*−*k'*) *else* ($\sum$ *j*<*k'*. (*if* (*fst* (*snd x*))!(*index init* (*xs'*!*j*)) *then 2*::*real else 1*)))) *D*)
    = ((*1/2*)^*?l*) ∗ (($\sum$ *x*∈{*l*::*bool list*. *length l* = *?l* ∧ *l*!(*index init q*)}. ((λ*x*. *real*(*k*−*k'*))) *x*)
    + ($\sum$ *x*∈{*l*::*bool list*. *length l* = *?l* ∧ ~*l*!(*index init q*)}. ((λ*x*. ($\sum$ *j*<*k'*. (*if x*!(*index init* (*xs'*!*j*)) *then 2*::*real else 1*)))) *x*)
) .
    also
    have ... = ((*1/2*)^*?l*) ∗ (($\sum$ *x*∈{*l*::*bool list*. *length l* = *?l* ∧ *l*!(*index init q*)}. *real*(*k*−*k'*))
    + (*3/2*)∗(*real* (*k'*))∗*2*^(*?l*−*1*)
) **by**(*simp only*: *absch2ndterm*)
    also
    have *E8*: ... = ((*1/2*)^*?l*) ∗ (*real*((*k*−*k'*)∗*2*^(*?l*−*1*)) + (*3/2*)∗(*real* (*k'*))∗*2*^(*?l*−*1*))
    by(*simp only*: *absch1stterm*)

    also have ... = ((*1/2*)^*?l*) ∗ (((*k*−*k'*) + (*k'*)∗(*3/2*)) ∗ *2*^(*?l*−*1*)) **apply**(*simp only*: *distrib_right*) **by** *simp*

also have ... = $((1/2)\hat{}?l) * 2\hat{}(?l-1) * (\quad (k-k') + (k')*(3/2) \quad)$ **by** *simp*

also have ... = $(((1::real)/2)\hat{}(Suc\ l')) * 2\hat{}(l') * (\quad real(k-k') + (k')*(3/2) \quad)$

using *lSuc* **by** *auto*

also have *E9*: ... = $(1/2) * \quad (\quad real(k-k') + (k')*(3/2) \quad)$

by (*simp add*: *field_simps*)

also have ... $\leq (3/4)*(k\ )$ **by** *auto*

finally show $E(map\_pmf\ (\lambda x.\ (if\ q \in set\ init\ then\ (if\ (fst\ (snd\ x))!(index\ init\ q)\ then\ real(\ k-k'\ )\ else\ (\sum j<k'.\ (if\ (fst\ (snd\ x))!index\ init\ (xs'!j)\ then\ 2::real\ else\ 1)))\ else\ 0\ ))\ D)$

$\leq 3/4 * k$

using *True* **by** *simp*

qed

Transformation of the Term for Paid Exchanges

have *paid_absch*: $E(map\_pmf\ (\lambda x.\ (\sum i<(length\ (paid\_A!n)).\ (if\ (fst\ (snd\ x))!(gebub\ n\ i)\ then\ 2::real\ else\ 1)\ ))\ D) = 3/2 * (length\ (paid\_A!n))$

proof –

{

fix *i*

assume *inbound*: $(index\ init\ i) < length\ init$

have $map\_pmf\ (\lambda xx.\ if\ fst\ (snd\ xx)\ !\ (index\ init\ i)\ then\ 2::real\ else\ 1)\ D =$

$bind\_pmf\ (map\_pmf\ (fst \circ snd)\ D)\ (\lambda b.\ return\_pmf\ (if\ b!\ index\ init\ i\ then\ 2::real\ else\ 1))$

unfolding *map_pmf_def* **by**(*simp add*: *bind_assoc_pmf bind_return_pmf*)

also have ... = $bind\_pmf\ (bv\ (length\ init))\ (\lambda b.\ return\_pmf\ (if\ b!\ index\ init\ i\ then\ 2::real\ else\ 1))$

using *config_n_bv*[*of init take n qs*] **by** *simp*

also have ... = $map\_pmf\ (\lambda yy.\ (if\ yy\ then\ 2\ else\ 1))\ (\ map\_pmf\ (\lambda y.\ y!(index\ init\ i))\ (bv\ (length\ init)))$

by (*simp add*: *map_pmf_def bind_return_pmf bind_assoc_pmf*)

also have ... = $map\_pmf\ (\lambda yy.\ (if\ yy\ then\ 2\ else\ 1))\ (bernoulli\_pmf\ (5\ /\ 10))$

by (*auto simp add*: *bv_comp_bernoulli*[*OF inbound*])

finally have $map\_pmf\ (\lambda xx.\ if\ fst\ (snd\ xx)\ !\ (index\ init\ i)\ then\ 2::real\ else\ 1)\ D =$

$map\_pmf\ (\lambda yy.\ if\ yy\ then\ 2::real\ else\ 1)\ (bernoulli\_pmf\ (5\ /\ 10))$ **.**

} **note** *umform* = *this*

have $E(map\_pmf\ (\lambda x.\ (\sum i<(length\ (paid\_A!n)).\ (if\ (fst\ (snd\ x))!(gebub\ n\ i)\ then\ 2::real\ else\ 1)))\ D) =$
$(\sum i<(length\ (paid\_A!n)).\ E(map\_pmf\ ((\lambda xx.\ (if\ (fst\ (snd\ xx))!(gebub\ n\ i)\ then\ 2::real\ else\ 1)))\ D))$
apply(*subst E_linear_sum2*)
using *finite_config_BIT*[*OF dist_init*] **by**(*simp_all*)
also have ... = $(\sum i<(length\ (paid\_A!n)).\ E(map\_pmf\ (\lambda y.\ if\ y\ then\ 2::real\ else\ 1)\ (bernoulli\_pmf\ (5\ /\ 10))))$ **using** *umform gebub_def gebub_inBound*[*OF 31*] **by** *simp*
also have ... = $3/2 * (length\ (paid\_A!n))$ **by**(*simp add*: *E_bernoulli*)
finally show $E(map\_pmf\ (\lambda x.\ (\sum i<(length\ (paid\_A!n)).\ (if\ (fst\ (snd\ x))!(gebub\ n\ i)\ then\ 2::real\ else\ 1)))\ D) = 3/2 * (length\ (paid\_A!n))$ **.**
qed

Combine the Results

have *costA_absch*: $k+(length\ (paid\_A!n)) + 1 = t\_A\ n$ **unfolding** *k_def q_def c_A_def p_A_def t_A_def* **by** (*auto*)

let $?yo= (\lambda x.\ (cost\ x) + (\Phi_2\ x) - (\Phi_0\ x))$
let $?yo2= (\lambda x.\ (k + 1) + (if\ (q)\in set\ init\ then\ (if\ (fst\ (snd\ x))!(index\ init\ q)\ then\ k-k'$
$else\ (\sum j<k'.\ (if\ (fst\ (snd\ x))!(index\ init\ (xs'!j))\ then\ 2::real\ else\ 1))\ )\ else\ 0)$
$+(\sum i<(length\ (paid\_A!n)).\ (if\ (fst\ (snd\ x))!(gebub\ n\ i)\ then\ 2\ else\ 1)))$

have *E0*: $t\_BIT\ n + Phi(n+1) - Phi\ n = E\ (map\_pmf\ ?yo\ D)$
using *inEreinziehn* **by** *auto*
also have ... $\leq E(map\_pmf\ ?yo2\ D)$
apply(*rule E_mono2*) **unfolding** *D_def*
apply(*fact finite_config_BIT*[*OF dist_init*])
apply(*fact ub_cost*[*unfolded D_def*])
done

also have *E2*: ... = $E(map\_pmf\ (\lambda x.\ k + 1::real)\ D)$
$+ (E(map\_pmf\ (\lambda x.\ (if\ (q)\in set\ init\ then\ (if\ (fst\ (snd\ x))!(index\ init\ q)\ then\ real(k-k')\ else\ (\sum j<k'.\ (if\ (fst\ (snd\ x))!(index\ init\ (xs'!j))\ then\ 2::real\ else\ 1)))else\ 0))\ D)$
$+ E(map\_pmf\ (\lambda x.\ (\sum i<(length\ (paid\_A!n)).\ (if\ (fst\ (snd\ x))!(gebub\ n\ i)\ then\ 2::real\ else\ 1)))\ D))$

unfolding *D_def* **apply**(*simp only*: *E_linear_plus2*[*OF finite_config_BIT*[*OF dist_init*]]) **by**(*auto simp*: *add.assoc*)

also have *E3*: ... ≤ *k + 1 +* (*3/4* ∗ (*real* (*k*)) + (*3/2* ∗ *real* (*length* (*paid_A*!*n*)))) **using** *paid_absch free_absch* **by** *auto*

also have ... = *k +* (*3/4* ∗ (*real k*)) + *1* + *3/2* ∗(*length* (*paid_A*!*n*)) **by** *auto*
also have ... = (*1+3/4*) ∗ (*real k*) + *1* + *3/2* ∗(*length* (*paid_A*!*n*)) **by** *auto*
also have *E4*: ... = *7/4*∗(*real k*) + *3/2* ∗(*length* (*paid_A*!*n*)) + *1* **by** *auto*
also have ... ≤ *7/4*∗(*real k*) + *7/4* ∗(*length* (*paid_A*!*n*)) + *1* **by** *auto*

also have *E5*:... = *7/4*∗(*k*+(*length* (*paid_A*!*n*))) + *1* **by** *auto*
also have *E6*:... = *7/4*∗(*t_A n* − (*1*::*real*)) + *1* **using** *costA_absch* **by** *auto*
also have ... = *7/4*∗(*t_A n*) − *7/4* + *1* **by** *algebra*
also have *E7*: ... = *7/4*∗(*t_A n*)− *3/4* **by** *auto*
finally show *t_BIT n + Phi*(*n+1*) − *Phi n* ≤ (*7* / *4*) ∗ *t_A n* − *3/4* **.**
qed
then show *t_BIT n + Phi*(*n + 1*) − *Phi n* ≤ (*7* / *4*) ∗ *t_A n* − *3/4* **.**
qed

9.3.7 Lift the Result to the Whole Request List

lemma *T_BIT_absch_le*: **assumes** *nqs*: *n* ≤ *length qs*
shows *T_BIT n* ≤ (*7* / *4*) ∗ *T_A n* − *3/4*∗*n*
unfolding *T_BIT_def T_A_def*
proof −
from *potential2*[*of Phi*, *OF phi0 phi_pos myub*] *nqs* **have**
sum t_BIT {..<*n*} ≤ (∑ *i*<*n*. *7* / *4* ∗ (*t_A i*) − *3* / *4*) **by** *auto*
also have ... = (∑ *i*<*n*. *7* / *4* ∗ *real_of_int* (*t_A i*)) − (∑ *i*<*n*. (*3/4*)) **by** (*rule sum_subtractf*)
also have ... = (∑ *i*<*n*. *7* / *4* ∗ *real_of_int* (*t_A i*)) − (*3/4*)∗(∑ *i*<*n*. *1*) **by** *simp*
also have ... = (∑ *i*<*n*. (*7* / *4*) ∗ *real_of_int* (*t_A i*)) − (*3/4*)∗*n* **by** *simp*
also have ... = (*7* / *4*) ∗ (∑ *i*<*n*. *real_of_int* (*t_A i*)) − (*3/4*)∗*n* **by** (*simp add*: *sum_distrib_left*)
also have ... = (*7* / *4*) ∗ *real_of_int* (∑ *i*<*n*.(*t_A i*)) − (*3/4*)∗*n* **by** *auto*
finally show *sum t_BIT* {..<*n*} ≤ *7* / *4* ∗ *real_of_int* (*sum t_A* {..<*n*})

− (*3*/*4*)∗*n* **by** *auto*
qed

lemma *T_BIT_absch*: **assumes** *nqs*: *n* ≤ *length qs*
  shows *T_BIT n* ≤ (*7* / *4*) ∗ *T_A′ n* − *3*/*4*∗*n*
using *nqs T_BIT_absch_le*[*of n*] *T_A_A′_leq*[*of n*] **by** *auto*

lemma *T_A_nneg*: *0* ≤ *T_A n*
by(*auto simp add*: *sum_nonneg T_A_def t_A_def c_A_def p_A_def*)

lemma *T_BIT_eq*: *T_BIT* (*length qs*) = *T_on_rand BIT init qs*
unfolding *T_BIT_def T_on_rand_as_sum* **using** *t_BIT_def* **by** *auto*

corollary *T_BIT_competitive*: **assumes** *n* ≤ *length qs* **and** *init* ≠ [] **and** ∀ *i*<*n*. *qs*!*i* ∈ *set init*
shows *T_BIT n* ≤ ((*7* / *4*) − *3*/(*4* ∗ *size init*)) ∗ *T_A′ n*
proof *cases*
  assume *0*: *real_of_int*(*T_A′ n*) ≤ *n* ∗ (*size init*)
  then have *1*: *3*/*4*∗*real_of_int*(*T_A′ n*) ≤ *3*/*4*∗(*n* ∗ (*size init*)) **by** *auto*
  have *T_BIT n* ≤ (*7* / *4*) ∗ *T_A′ n* − *3*/*4*∗*n* **using** *T_BIT_absch*[*OF assms*(*1*)] **by** *auto*
  also have . . . = ((*7* / *4*) ∗ *real_of_int*(*T_A′ n*)) − (*3*/*4*∗(*n* ∗ *size init*)) / *size init*
    using *assms*(*2*) **by** *simp*
  also have . . . ≤ ((*7* / *4*) ∗ *real_of_int*(*T_A′ n*)) − *3*/*4*∗*T_A′ n* / *size init*
    by(*rule diff_left_mono*[*OF divide_right_mono*[*OF 1*]]) *simp*
  also have . . . = ((*7* / *4*) − *3*/*4* / *size init*) ∗ *T_A′ n* **by** *algebra*
  also have . . . = ((*7* / *4*) − *3*/(*4* ∗ *size init*)) ∗ *T_A′ n* **by** *simp*
  finally show *?thesis* **.**
next
  assume *0*: ¬ *real_of_int*(*T_A′ n*) ≤ *n* ∗ (*size init*)

  have *T_A′_nneg*: *0* ≤ *T_A′ n* **using** *T_A_nneg*[*of n*] *T_A_A′_leq*[*of n*] *assms*(*1*) **by** *auto*

  have *2* − *1* / *size init* ≥ *1* **using** *assms*(*2*)
    by (*auto simp add*: *field_simps neq_Nil_conv*)
  have *T_BIT n* ≤ *n* ∗ *size init* **using** *T_BIT_ub*[*OF assms*(*3*)] **by**

```
linarith
  also have ... < real_of_int(T_A' n) using 0 by linarith
  also have ... ≤ ((7 / 4) − 3/4 / size init) ∗ T_A' n using assms(2)
T_A'_nneg
    by(auto simp add: mult_le_cancel_right1 field_simps neq_Nil_conv)
  finally show ?thesis by simp
qed


lemma t_A'_t: n < length qs ⟹ t_A' n = int (t (s_A' n) (qs!n) (acts !
n))
by (simp add: t_A'_def t_def c_A'_def p_A'_def paid_A'_def len_acts
split: prod.split)

lemma T_A'_eq_lem: (∑ i=0..<length qs. t_A' i) =
  T (s_A' 0) (drop 0 qs) (drop 0 acts)
proof(induction rule: zero_induct[of _ size qs])
  case 1 thus ?case by (simp add: len_acts)
next
  case (2 n)
  show ?case
  proof cases
    assume n < length qs
    thus ?case using 2
      by(simp add: Cons_nth_drop_Suc[symmetric,where i=n] len_acts
sum.atLeast_Suc_lessThan
       t_A'_t free_A_def paid_A'_def)
  next
    assume ¬ n < length qs thus ?case by (simp add: len_acts)
  qed
qed

lemma T_A'_eq: T_A' (length qs) = T init qs acts
using T_A'_eq_lem by(simp add: T_A'_def atLeast0LessThan)

corollary BIT_competitive3: init ≠ [] ⟹ ∀ i<length qs. qs!i ∈ set init
⟹
  T_BIT (length qs) ≤ ( (7/4) − 3 / (4 ∗ length init)) ∗ T init qs acts
using order.refl T_BIT_competitive[of length qs] T_A'_eq by (simp add:
of_int_of_nat_eq)

corollary BIT_competitive2: init ≠ [] ⟹ ∀ i<length qs. qs!i ∈ set init
⟹
  T_on_rand BIT init qs ≤ ( (7/4) − 3 / (4 ∗ length init)) ∗ T init qs
```

acts
using *BIT_competitive3 T_BIT_eq* **by** *auto*

corollary *BIT_absch_le*: *init* ≠ [] ⟹
  T_on_rand BIT init qs ≤ (*7* / *4*) ∗ (*T init qs acts*) − *3*/*4* ∗ *length qs*
using *T_BIT_absch*[*of length qs*, *unfolded T_A′_eq T_BIT_eq*] **by** *auto*

end

9.3.8 Generalize Competitivness of BIT

lemma *setdi*: *set xs* = {*0*..<*length xs*} ⟹ *distinct xs*
apply(*rule card_distinct*) **by** *auto*

theorem *compet_BIT*: **assumes** *init* ≠ [] *distinct init set qs* ⊆ *set init*
shows *T_on_rand BIT init qs* ≤ ((*7*/*4*) − *3* / (*4* ∗ *length init*)) ∗ *T_opt init qs*
proof−
  from *assms*(*3*) **have** *1*: ∀ *i* < *length qs*. *qs*!*i* ∈ *set init* **by** *auto*
  { **fix** *acts* :: *answer list*
    assume *len*: *length acts* = *length qs*
    interpret *BIT_Off acts qs init* **proof qed** (*auto simp*: *assms*(*2*) *len*)
    from *BIT_competitive2*[*OF assms*(*1*) *1*] *assms*(*1*)
     have *T_on_rand BIT init qs* / ((*7*/*4*) − *3* / (*4* ∗ *length init*)) ≤ *real*(*T init qs acts*)
      by(*simp add*: *field_simps length_greater_0_conv*[*symmetric*]
        del: *length_greater_0_conv*) }
    hence *T_on_rand BIT init qs* / ((*7*/*4*) − *3* / (*4* ∗ *length init*)) ≤ *T_opt init qs*
      apply(*simp add*: *T_opt_def Inf_nat_def*)
      apply(*rule LeastI2_wellorder*)
      using *length_replicate*[*of length qs undefined*] **apply** *fastforce*
      apply *auto*
      done
  thus *?thesis* **using** *assms* **by**(*simp add*: *field_simps*
    length_greater_0_conv[*symmetric*] *del*: *length_greater_0_conv*)
qed

theorem *compet_BIT4*: **assumes** *init* ≠ [] *distinct init*
shows *T_on_rand BIT init qs* ≤ *7*/*4* ∗ *T_opt init qs*
proof−
  { **fix** *acts* :: *answer list*
    assume *len*: *length acts* = *length qs*

interpret *BIT_Off acts qs init* **proof qed** (*auto simp*: *assms*(*2*) *len*)
from *BIT_absch_le*[*OF assms*(*1*)] *assms*(*1*)
have (*T_on_rand BIT init qs* + *3* / *4* ∗ *length qs*)/ (*7*/*4*) ≤ *real*(*T init qs acts*)
by(*simp add*: *field_simps length_greater_0_conv*[*symmetric*]
del: *length_greater_0_conv*) **}**
hence (*T_on_rand BIT init qs* + *3* / *4* ∗ *length qs*)/ (*7*/*4*) ≤ *T_opt init qs*
apply(*simp add*: *T_opt_def Inf_nat_def*)
apply(*rule LeastI2_wellorder*)
using *length_replicate*[*of length qs undefined*] **apply** *fastforce*
apply *auto*
done
thus *?thesis* **by**(*simp add*: *field_simps*
length_greater_0_conv[*symmetric*] *del*: *length_greater_0_conv*)
qed

theorem *compet_BIT_2*:
compet_rand BIT (*7*/*4*) {*init*. *init* ≠ [] ∧ *distinct init*}
unfolding *compet_rand_def*
proof
fix *init*
assume *init* ∈ {*init*. *init* ≠ [] ∧ *distinct init* }
then have *ne*: *init* ≠ [] **and** *a*: *distinct init* **by** *auto*
{
fix *qs*
assume *init* ≠ [] **and** *a*: *distinct init*
then have *T_on_rand BIT init qs* ≤ *7*/*4* ∗ *T_opt init qs*
using *compet_BIT4*[*of init qs*] **by** *simp*
}
with *a ne* **show** ∃ *b*≥*0*. ∀ *qs*. *static init qs* ⟶ *T_on_rand BIT init qs* ≤ (*7* / *4*) ∗ (*T_opt init qs*) + *b*
by *auto*
qed

end

10 Partial cost model

theory *Partial_Cost_Model*
imports *Move_to_Front*
begin

definition t_p :: *'a state* ⇒ *'a* ⇒ *answer* ⇒ *nat* **where**
t_p *s q a* = (*let* (*mf*,*sws*) = *a in index* (*swaps sws s*) *q* + *size sws*)

notation (*latex*) t_p (‹t^*›)

lemma $t_p t$: t_p *s q a* + *1* = *t s q a* **unfolding** t_p_*def t_def* **by**(*simp add*: *split_def*)

interpretation *On_Off step* t_p *static* **.**

abbreviation T_p == *T*
abbreviation T_p_*opt* == *T_opt*
abbreviation T_p_*on* == *T_on*
abbreviation T_p_*on_rand'* == *T_on_rand'*
abbreviation T_p_*on_n* == *T_on_n*
abbreviation T_p_*on_rand* == *T_on_rand*
abbreviation T_p_*on_rand_n* == *T_on_rand_n*
abbreviation $config_p$ == *config*
abbreviation $compet_p$ == *compet*

end

11 Equivalence of Regular Expression with Variables

theory *RExp_Var*
imports *Regular−Sets.Equivalence_Checking*
begin

fun *castdown* :: *nat rexp* ⇒ *nat rexp* **where**
castdown Zero = *Zero*
| *castdown One* = *One*
| *castdown* (*Plus a b*) = *Plus* (*castdown a*) (*castdown b*)
| *castdown* (*Times a b*) = *Times* (*castdown a*) (*castdown b*)
| *castdown* (*Star a*) = *Star* (*castdown a*)
| *castdown* (*Atom x*) = (*Atom* (*x div 2*))

fun *castup* :: *nat rexp* ⇒ *nat rexp* **where**
castup Zero = *Zero*
| *castup One* = *One*

```
| castup (Plus a b) = Plus (castup a) (castup b)
| castup (Times a b) = Times (castup a) (castup b)
| castup (Star a) = Star (castup a)
| castup (Atom x) = Atom (2*x)

lemma castdown (castup r) = r
apply(induct r) by(auto)


fun substvar :: nat ⇒ (nat ⇒ ((nat rexp) option)) ⇒ nat rexp where
  substvar i σ = (case σ i of Some x ⇒ x
                            | None ⇒ Atom (2*i+1))

fun w2rexp :: nat list ⇒ nat rexp where
  w2rexp [] = One
| w2rexp (a#as) = Times (Atom a) (w2rexp as)

lemma lang (w2rexp as) = { as }
apply(induct as)
  apply(simp)
  by(simp add: conc_def)



fun subst :: nat rexp ⇒ (nat ⇒ nat rexp option) ⇒ nat rexp where
  subst Zero _ = Zero
| subst One _ = One
| subst (Atom i) σ = (if i mod 2 = 0 then Atom i else substvar (i div 2) σ)
| subst (Plus a b) σ = Plus (subst a σ) (subst b σ)
| subst (Times a b) σ = Times (subst a σ) (subst b σ)
| subst (Star a) σ = Star (subst a σ)


lemma subst_w2rexp: lang (subst (w2rexp (xs @ ys)) σ) = lang (subst
(w2rexp xs) σ) @@ lang (subst (w2rexp ys) σ)
proof(induct xs)
  case (Cons x xs)
  have lang (subst (w2rexp ((x # xs) @ ys)) σ)
        = lang (subst (Times (Atom x) (w2rexp (xs @ ys))) σ) by simp
   also have ... = lang (Times (subst (Atom x) σ) (subst (w2rexp (xs @
ys)) σ)) by simp
  also have ... = lang (subst (Atom x) σ) @@ (lang (subst (w2rexp (xs @
ys)) σ)) by simp
  also have ... = lang (subst (Atom x) σ) @@ ( lang (subst (w2rexp xs) σ)
```

@@ *lang* (*subst* (*w2rexp ys*) σ)) **by**(*simp only*: *Cons*)
also have ... = *lang* (*Times* (*subst* (*Atom x*) σ) (*subst* (*w2rexp xs*) σ)) @@ *lang* (*subst* (*w2rexp ys*) σ)
apply(*simp del*: *subst.simps*) **by**(*rule conc_assoc*[*symmetric*])
also have ... = *lang* (*subst* (*Times* (*Atom x*) (*w2rexp xs*)) σ) @@ *lang* (*subst* (*w2rexp ys*) σ) **by** *simp*
also have ... = *lang* (*subst* (*w2rexp* (*x* # *xs*)) σ) @@ *lang* (*subst* (*w2rexp ys*) σ) **by** *simp*
finally show *?case* **.**
qed *simp*

fun *substW* :: *nat list* ⇒ (*nat* ⇒ *nat rexp option*) ⇒ *nat rexp* **where**
substW as σ = *subst* (*w2rexp as*) σ

fun *substL* :: *nat lang* ⇒ (*nat* ⇒ *nat rexp option*) ⇒ *nat rexp set* **where**
substL S σ = {*substW a* σ|*a*. *a* ∈ *S*}

fun *L* :: *nat rexp set* ⇒ *nat lang* **where**
L S = (⋃*r*∈*S*. *lang r*)

lemma *L_mono*: *S1* ⊆ *S2* ⟹ *L S1* ⊆ *L S2*
apply(*simp*) **by** *blast*

definition *concS* :: *'b rexp set* ⇒ *'b rexp set* ⇒ *'b rexp set* **where**
concS S1 S2 = {*Times a b*|*a b*. *a*∈*S1* ∧ *b*∈*S2*}

lemma *substL_conc*: *L* (*substL* (*L1* @@ *L2*) σ) = *L* (*concS* (*substL L1* σ) (*substL L2* σ))
apply(*simp add*: *concS_def conc_def*)
apply(*auto*)
proof (*goal_cases*)
case (*1 x xs ys*)
show *?case*
apply(*rule exI*[**where** *x*=*Times* (*subst* (*w2rexp xs*) σ) (*subst* (*w2rexp ys*) σ)])
apply(*simp*)
apply(*safe*)
apply(*rule exI*[**where** *x*=*xs*]) **apply**(*simp add*: *1*(*2*))
apply(*rule exI*[**where** *x*=*ys*]) **apply**(*simp add*: *1*(*3*))
using *1*(*1*) *subst_w2rexp* **by** *auto*
next
case (*2 x xs ys*)
show *?case*
apply(*rule exI*[**where** *x*=*subst* (*w2rexp* (*xs* @ *ys*)) σ])

```
    apply(safe)
      apply(rule exI[where x=xs@ys]) apply(simp)
        apply(rule exI[where x=xs])
        apply(rule exI[where x=ys]) using 2(2,3) apply(simp)
      using 2(1) subst_w2rexp by(auto)
qed

lemma L_conc: L(concS M1 M2) = (L M1) @@ (L M2)
proof -
  have L(concS M1 M2) = (⋃x∈{Times a b |a b. a ∈ M1 ∧ b ∈ M2}. lang
x) unfolding concS_def by(simp)
  also have ... = (⋃{lang (Times a b) |a b. a ∈ M1 ∧ b ∈ M2} ) by blast
  also have ... = (⋃{lang a @@ lang b |a b. a ∈ M1 ∧ b ∈ M2} ) by simp
  also have ... = (⋃{{xs@ys | xs ys. xs ∈ lang a & ys ∈ lang b} |a b. a ∈
M1 ∧ b ∈ M2} ) unfolding conc_def by simp
  also have ... = {xs@ys | xs ys. xs∈ (⋃r∈M1. lang r) ∧ ys ∈ (⋃r∈M2.
lang r) } by blast
  also have ... = {xs@ys | xs ys. xs∈ L(M1) ∧ ys ∈ L(M2) } by simp
  also have ... = (L M1) @@ (L M2) unfolding conc_def by simp
  finally show ?thesis .
qed

lemma L(M1 ∪ M2) = (L M1) ∪ (L M2)
by simp

fun verund :: 'b rexp list ⇒ 'b rexp where
  verund [] = Zero
| verund [r] = r
| verund (r#rs) = Plus r (verund rs)

lemma lang_verund: r ∈ L (set rs) = (r ∈ lang (verund rs))
apply(induct rs)
  apply(simp)
  apply(case_tac rs) by auto

lemma obtainit:
  assumes r ∈ lang (verund rs)
  shows ∃x∈(set (rs::nat rexp list)). r ∈ lang x
proof -
  from assms have r ∈ L (set rs) by(simp only: lang_verund)
  then show ?thesis by(auto)
qed
```

```
lemma lang_verund4: L (set rs) = lang (verund rs)
apply(induct rs)
  apply(simp)
  apply(case_tac rs) by auto

lemma lang_verund1: r ∈ L (set rs) ⟹ r ∈ lang (verund rs)
apply(induct rs)
  apply(simp)
  apply(case_tac rs) by auto
lemma lang_verund2: r ∈ lang (verund rs) ⟹ r ∈ L (set rs)
apply(induct rs)
  apply(simp)
  apply(case_tac rs) by auto

definition starS :: 'b rexp set ⇒ 'b rexp set where
  starS S = {Star (verund xs)|xs. set xs ⊆ S}

lemma [] ∈ L (starS S)
unfolding starS_def apply(simp)
  apply(rule exI[where x=Star(verund [])])
  apply(simp)
    apply(rule exI[where x=[]])
    by (simp)

lemma power_mono: L1 ⊆ L2 ⟹ (L1::'a lang) ^^ n ⊆ L2 ^^ n
apply(auto) apply(induct n) by(auto simp: conc_def)

lemma star_mono: L1 ⊆ L2 ⟹ star L1 ⊆ star L2
  apply (simp add: star_def)
  apply (rule UN_mono)
  apply (auto simp: power_mono)
  done

lemma Lstar: L(starS M) = star ( L(M) )
unfolding starS_def apply(auto)
proof (goal_cases)
  case (1 x xs)
  from 1(2) have L (set xs) ⊆ L (M) by(rule L_mono)
  then have a: star (L (set xs)) ⊆ star (L (M)) by (rule star_mono)
   from 1(1) obtain n where x ∈ (lang (verund xs)) ^^ n unfolding
star_def by(auto)
  thm lang_verund4
  then have x ∈ (L (set xs)) ^^ n by(simp only: lang_verund4)
```

```
  then have x ∈ star (L (set xs)) unfolding star_def by auto
  with a have x ∈ star (L (M)) by auto
  then show x ∈ star (⋃x∈M. lang x) unfolding starS_def by auto
next
  case (2 x)
  then obtain n where x ∈ (⋃x∈M. lang x) ^^ n unfolding star_def
by auto
  then show ?case
  proof (induct n arbitrary: x)
    case 0
    then have t: x=[] by(simp)
    show ?case
      apply(rule exI[where x=Star Zero])
      apply(auto simp: t) apply(rule exI[where x=[]]) by(simp)
  next
    case (Suc n)
    from Suc(2) have t: x ∈ (⋃a∈M. lang a) @@ (⋃a∈M. lang a) ^^ n
by (simp)
    then obtain A B where x: x = A @ B and A: A ∈ (⋃a∈M. lang a)
and B: B ∈ (⋃a∈M. lang a) ^^ n by(auto simp: conc_def)
    then obtain m where am: A ∈ lang m and mM: m∈M by(auto)
    from Suc(1)[OF B] obtain b bs where b = Star (verund bs) and bsM:
set bs ⊆ M B ∈ lang b by auto
    then have Bin:  B ∈ lang (Star (verund bs)) by simp
    let ?c = Star (verund (m#bs))

    have ac: lang m ⊆ lang (Star (verund (m # bs)))
      apply(cases bs) by(auto)
    have ad: (lang (Star (verund bs))) ⊆ lang (Star (verund (m # bs)))
      apply (simp add: star_def)
      apply (rule UN_mono)
      apply simp_all
      proof -
        fix n
        have t: (lang (verund bs) ^^ n) ⊆ (lang m ∪ lang (verund bs)) ^^ n
          by (rule power_mono) simp
        then show lang (verund bs) ^^ n
          ⊆ lang (verund (m # bs)) ^^ n by (cases bs) simp_all
      qed

    from Bin am mM x have x ∈ lang m @@ (lang (Star (verund bs))) by
auto
    then have  x ∈ lang (Star (verund (m # bs))) @@ lang (Star (verund
(m # bs))) using ac ad by blast
```

then have *x_in*: $x \in$ *lang* (*Star* (*verund* (*m* # *bs*))) **by** (*auto*)

show *?case*
apply(*rule exI*[**where** *x*=*?c*])
apply(*safe*)
apply(*rule exI*[**where** *x*=*m*#*bs*]) **apply**(*simp add*: *bsM mM*)
by(*fact x_in*)
qed
qed

lemma *substL_star*: *L* (*substL* (*star L1*) σ) = *L* (*starS* (*substL L1* σ))
apply (*simp add*: *concS_def conc_def starS_def star_def*)
apply *auto* **unfolding** *star_def*
proof −
fix *x a n*
assume $x \in$ *lang* (*subst* (*w2rexp a*) σ)
moreover assume $a \in$ *L1* ^^ *n*
ultimately show ∃ *xa*. (∃ *xs*. *xa* = *Star* (*verund xs*) ∧ *set xs* ⊆ {*subst* (*w2rexp a*) σ | *a*. $a \in$ *L1*}) ∧ $x \in$ *lang xa*
proof(*induct n arbitrary*: *x a*)
case *0*
then have *a*=[] **by** *auto*
with *0* **show** *?case* **apply**(*simp*)
apply(*rule exI*[**where** *x*=*Star* (*Zero*)])
apply(*simp*)
apply(*rule exI*[**where** *x*=[]])
by(*simp*)
next
case (*Suc n*)
then have *a1*: $a \in$ *L1* @@ *L1* ^^ *n* **by** *auto*
then obtain *A B* **where** *a2*: *a* = *A* @ *B* **and** *A*: $A \in$ *L1* **and** *B*: $B \in$ *L1* ^^ *n* **by** *auto*

thm *subst_w2rexp*
from *Suc*(*2*) **have** $x \in$ *lang* (*subst* (*w2rexp A*) σ) @@ *lang* (*subst* (*w2rexp B*) σ) **unfolding** *a2*
by(*simp only*: *subst_w2rexp*)
then obtain *x1 x2* **where** *x*: *x* = *x1*@*x2* **and** *x1*: $x1 \in$ *lang* (*subst* (*w2rexp A*) σ)
and *x2*: $x2 \in$ *lang* (*subst* (*w2rexp B*) σ) **by** *auto*
from *Suc*(*1*)[*OF x2 B*] **obtain** *R li* **where**
R: *R* = *Star* (*verund li*) **and** *li*: *set li* ⊆ {*subst* (*w2rexp a*) σ |*a*. *a* ∈ *L1*}

```
          and x2R: x2 ∈ lang R by auto


    show ?case
      apply(rule exI[where x=Star (verund ((subst (w2rexp A) σ)#li))])
      apply(simp)
      apply(safe)
        apply(rule exI[where x=((subst (w2rexp A) σ)#li)])
        apply(simp add: li)
          apply(rule exI[where x=A]) apply(simp add: A)
        unfolding x
        proof (goal_cases)
          case 1
          let ?L = (lang (subst (w2rexp A) σ) ∪ lang (verund li))
          have t1: x1 ∈ ?L using x1 star_mono by blast
           have t2: x2 ∈ star ?L using x2R R star_mono apply(simp) by
blast
          have x1 @ x2 ∈ (?L @@ star ?L) using t1 t2 by auto
          then show ?case
          apply(cases li) by(auto)
        qed
    qed
next
  fix x and xs :: nat rexp list
  assume x ∈ (⋃n. lang (verund xs) ^^ n)
  then obtain n where x ∈ lang (verund xs) ^^ n by auto
  moreover assume set xs ⊆ {subst (w2rexp a) σ |a. a ∈ L1}
  ultimately show ∃xa. (∃a. xa = subst (w2rexp a) σ ∧
    (∃n. a ∈ L1 ^^ n)) ∧ x ∈ lang xa
  proof (induct n arbitrary: x)
    case 0
    then have xe: x=[] by auto
    show ?case
      apply(rule exI[where x=One])
      apply(simp add: xe)
        apply(rule exI[where x=[]])
        apply(simp)
          apply(rule exI[where x=0])
          by(simp)
  next
    case (Suc n)
    then have x ∈ lang (verund xs) @@ (lang (verund xs) ^^ n) by auto
    then obtain x1 x2 where x: x=x1@x2 and x1: x1∈lang (verund xs)
                    and x2: x2 ∈ (lang (verund xs) ^^ n) by auto
```

from *obtainit* [*OF x1*] **obtain** *el*
where *el* ∈ *set xs* **and** *x1* ∈ *lang el* **by** *auto*
with *Suc.prems* **obtain** *elem*
where *x1elem*: *x1* ∈ *lang* (*subst* (*w2rexp elem*) *σ*)
and *elemL1*: *elem* ∈ *L1* **by** *auto*
from *Suc.hyps* [*OF x2 Suc.prems*(*2*)] **obtain** *R word n* **where**
R: *R* = *subst* (*w2rexp word*) *σ* **and** *word*: *word* ∈ *L1* ^^ *n* **and** *x2*: *x2* ∈ *lang R* **by** *auto*

show *?case*
apply(*rule exI*[**where** *x*=*subst* (*w2rexp* (*elem*@*word*)) *σ*])
apply(*safe*)
apply(*rule exI*[**where** *x*=*elem*@*word*])
apply(*simp*)
apply(*rule exI*[**where** *x*=*Suc n*])
proof (*goal_cases*)
case *1*
have *elem* ∈ *L1* **by**(*fact elemL1*)
with *word*
show *elem* @ *word* ∈ *L1* ^^ *Suc n* **by** *simp*
next
case *2*
have *x1* ∈ *lang* (*subst* (*w2rexp elem*) *σ*) **by**(*fact x1elem*)
with *x2*[*unfolded R*] **show** *?case* **unfolding** *x* **apply**(*simp only*: *subst_w2rexp*) **by** *blast*
qed
qed
qed

lemma *substituitionslemma*:
fixes *E* :: *nat rexp*
shows *L* (*substL* (*lang*(*E*)) *σ*) = *lang* (*subst E σ*)
proof (*induct E*)
case (*Star e*)
have *L* (*substL* (*lang* (*Star e*)) *σ*) = *L* (*substL* (*star* (*lang e*)) *σ*) **by** *auto*
also have ... = *L* (*starS* (*substL* (*lang e*) *σ*)) **by**(*simp only*: *substL_star*)
also have ... = *star* (*L* (*substL* (*lang e*) *σ*)) **by**(*simp only*: *Lstar*)
also have ... = *star* (*lang* (*subst e σ*)) **by**(*simp only*: *Star*)
also have ... = *lang* ((*subst* (*Star e*) *σ*)) **by** *auto*
finally show *?case* **.**
next
case (*Plus e1 e2*)
have *L* (*substL* (*lang* (*Plus e1 e2*)) *σ*) = *L* (*substL* (*lang e1* ∪ *lang e2*)

σ) **by** *simp*
also have ... = *L* (*substL* (*lang e1*) σ ∪ *substL* (*lang e2*) σ) **by** *auto*
also have ... = *L* (*substL* (*lang e1*) σ) ∪ *L* (*substL* (*lang e2*) σ) **by** *auto*
also have ... = *lang* (*subst e1* σ) ∪ *lang* (*subst e2* σ) **by**(*simp only*: *Plus*)
also have ... = *lang* (*subst* (*Plus e1 e2*) σ) **by** *auto*
finally show *?case* **.**
next
case (*Times e1 e2*)
have *L* (*substL* (*lang* (*Times e1 e2*)) σ) = *L* (*substL* (*lang e1* @@ *lang e2*) σ) **by**(*simp*)
also have ... = *L* (*concS* (*substL* (*lang e1*) σ) (*substL* (*lang e2*) σ)) **by**(*simp only*: *substL_conc*)
thm *L_conc*
also have ... = *L* (*substL* (*lang e1*) σ) @@ *L* (*substL* (*lang e2*) σ) **by**(*simp only*: *L_conc*)
also have ... = *lang* (*subst e1* σ) @@ *lang* (*subst e2* σ) **by**(*simp only*: *Times*)
also have ... = *lang* (*Times* (*subst e1* σ) (*subst e2* σ)) **by** *auto*
also have ... = *lang* (*subst* (*Times e1 e2*) σ) **by** *auto*
finally show *?case* **.**
qed *simp_all*

corollary *lift*: *lang e1* = *lang e2* ⟹ *lang* (*subst e1* σ) = *lang* (*subst e2* σ)
proof −
assume *eq*: *lang e1* = *lang e2*
thm *substituitionslemma*
have *lang* (*subst e1* σ) = *L* (*substL* (*lang e1*) σ) **by**(*simp only*: *substituitionslemma*)
also have ... = *L* (*substL* (*lang e2*) σ) **using** *eq* **by** *simp*
also have ... = *lang* (*subst e2* σ) **by**(*simp only*: *substituitionslemma*)
finally show *?thesis* **.**
qed

11.1 Examples

lemma *lang* (*Plus* (*Atom* (*x::nat*)) (*Atom x*)) = *lang* (*Atom x*)
proof −
let *?σ* = (λ*i*. (*if i=0 then Some* (*Atom x*) *else None*))
let *?e1* = *Plus* (*Atom 1*) (*Atom 1*)
let *?e2* = *Atom 1*
have *lang* (*Plus* (*Atom x*) (*Atom x*)) = *lang* (*subst ?e1 ?σ*) **by** (*simp*)
thm *soundness*

also have ... = *lang* (*subst ?e2 ?σ*)
apply(*rule lift*)
apply(*rule soundness*)
by *eval*
also have ... = *lang* (*Atom x*) **by** *auto*
finally show *?thesis* **.**
qed

fun *seq* :: *'a rexp list* ⇒ *'a rexp* **where**
seq [] = *One* |
seq [*r*] = *r* |
seq (*r#rs*) = *Times r* (*seq rs*)

abbreviation *question* **where** *question x* == *Plus x One*

definition *L_4cases* (*x*::*nat*) *y*=
verund [*seq*[*question* (*Atom x*),(*Atom y*), (*Atom y*)],
seq[*question* (*Atom x*),(*Atom y*),(*Atom x*),*Star*(*Times* (*Atom y*)(*Atom x*)),(*Atom y*),(*Atom y*)],
seq[*question* (*Atom x*),(*Atom y*),(*Atom x*),*Star*(*Times* (*Atom y*)(*Atom x*)),(*Atom x*)],
seq[(*Atom x*),(*Atom x*)]]

definition *L_A x y* = *seq*[*question* (*Atom x*),(*Atom y*), (*Atom y*)]
definition *L_B x y* = *seq*[*question* (*Atom x*),(*Atom y*),(*Atom x*),*Star*(*Times* (*Atom y*)(*Atom x*)),(*Atom y*),(*Atom y*)]
definition *L_C x y* = *seq*[*question* (*Atom x*),(*Atom y*),(*Atom x*),*Star*(*Times* (*Atom y*)(*Atom x*)),(*Atom x*)]
definition *L_D x y* = *seq*[(*Atom x*),(*Atom x*)]

lemma *L_4cases x y* = *verund* [*L_A x y*, *L_B x y*, *L_C x y*, *L_D x y*]
unfolding *L_A_def L_B_def L_C_def L_D_def L_4cases_def* **by** *auto*

definition *L_lasthasxx x y* = (*Plus* (*seq*[*question* (*Atom x*), *Star*(*Times* (*Atom y*)(*Atom x*)),(*Atom y*),(*Atom y*)])
(*seq*[*question* (*Atom y*), *Star*(*Times*(*Atom x*) (*Atom y*)),(*Atom x*),(*Atom x*)]))

lemma *lastxx_com*: *lang* (*L_lasthasxx* (*x*::*nat*) *y*) = *lang* (*L_lasthasxx y*

x) (**is** *lang ?A* = *lang ?B*)
proof −
let *?σ* = (*λi*. (*if i=0 then Some* (*Atom x*) *else* (*if i=1 then Some* (*Atom y*) *else None*)))

let *?e1* = *Plus* (*seq*[*Plus* (*Atom 1*) *One*, *Star*(*Times* (*Atom 3*) (*Atom 1*)),(*Atom 3*),(*Atom 3*)])
(*seq*[*Plus* (*Atom 3*) *One*, *Star*(*Times* (*Atom 1*) (*Atom 3*)),(*Atom 1*),(*Atom 1*)])
let *?e2* = *Plus* (*seq*[*Plus* (*Atom 3*) *One*, *Star*(*Times* (*Atom 1*) (*Atom 3*)),(*Atom 1*),(*Atom 1*)])
(*seq*[*Plus* (*Atom 1*) *One*, *Star*(*Times* (*Atom 3*) (*Atom 1*)),(*Atom 3*),(*Atom 3*)])
have *lang ?A* = *lang* (*subst ?e1 ?σ*) **by**(*simp add*: *L_lasthasxx_def*)
thm *soundness*
also have ... = *lang* (*subst ?e2 ?σ*)
apply(*rule lift*)
apply(*rule soundness*)
by *eval*
also have ... = *lang ?B* **by** (*simp add*: *L_lasthasxx_def*)
finally show *?thesis* **.**
qed

lemma *lastxx_is_4cases*: *lang* (*L_4cases x y*) = *lang* (*L_lasthasxx x y*) (**is** *lang ?A* = *lang ?B*)
proof −
let *?σ* = (*λi*. (*if i=0 then Some* (*Atom x*) *else* (*if i=1 then Some* (*Atom y*) *else None*)))

let *?e1* = (*Plus* (*seq*[*Plus* (*Atom 1*) *One*,(*Atom 3*), (*Atom 3*)])
(*Plus* (*seq*[*Plus* (*Atom 1*) *One*,(*Atom 3*),(*Atom 1*),*Star*(*Times* (*Atom 3*) (*Atom 1*)),(*Atom 3*),(*Atom 3*)])
(*Plus* (*seq*[*Plus* (*Atom 1*) *One*,(*Atom 3*),(*Atom 1*),*Star*(*Times* (*Atom 3*) (*Atom 1*)),(*Atom 1*)])
(*seq*[(*Atom 1*),(*Atom 1*)]))))
let *?e2* = *Plus* (*seq*[*Plus* (*Atom 1*) *One*, *Star*(*Times* (*Atom 3*) (*Atom 1*)),(*Atom 3*),(*Atom 3*)])
(*seq*[*Plus* (*Atom 3*) *One*, *Star*(*Times* (*Atom 1*) (*Atom 3*)),(*Atom 1*),(*Atom 1*)])
have *lang ?A* = *lang* (*subst ?e1 ?σ*) **by** (*simp add*: *L_4cases_def*)
thm *soundness*
also have ... = *lang* (*subst ?e2 ?σ*)
apply(*rule lift*)

apply(*rule soundness*)
by *eval*
also have ... = *lang ?B* **by** (*simp add*: *L_lasthasxx_def*)
finally show *?thesis* **.**
qed

definition *myUNIV x y* = *Star* (*Plus* (*Atom x*) (*Atom y*))

lemma *myUNIV_alle*: *lang* (*myUNIV x y*) = {*xs*. *set xs* ⊆ {*x*,*y*}}
proof −
have *star* {[*y*], [*x*]} = {*concat ws* |*ws*. *set ws* ⊆ {[*y*], [*x*]}} **by**(*simp only*: *star_conv_concat*)
also have ... = {*xs*. *set xs* ⊆ {*x*, *y*}} **apply**(*auto*) **apply**(*cases x*=*y*) **apply**(*simp*)
apply(*case_tac ws*)
apply(*simp*)
apply(*auto*)
proof (*goal_cases*)
case (*1 as*)
then show *?case*
proof (*induct as*)
case (*Cons a as*)
then have *as*: *set as* ⊆ {*x*,*y*} **and** *axy*: *a* ∈ {*x*,*y*} **by** *auto*
from *Cons*(*1*)[*OF as*] **obtain** *ws* **where** *asco*: *as* = *concat ws* **and** *ws*: *set ws* ⊆ {[*y*],[*x*]} **by** *auto*
show *?case*
apply(*rule exI*[**where** *x*=[*a*]#*ws*])
using *axy* **by**(*auto simp add*: *asco ws*)
qed (*rule exI*[**where** *x*=[]], *simp*)
qed
finally show *?thesis* **by**(*simp add*: *myUNIV_def*)
qed

definition *nodouble x y* = (*Plus*
(*seq*[*question* (*Atom x*), *Star*(*Times*(*Atom y*)(*Atom x*)),(*Atom y*)])
(*seq*[*question* (*Atom y*), *Star*(*Times*(*Atom x*) (*Atom y*)),(*Atom x*)]))

lemma *myUNIV_char*: *lang* (*myUNIV* (*x*::*nat*) *y*) = *lang* (*Times* (*Star* (*L_lasthasxx x y*)) (*Plus One* (*nodouble x y*))) (**is** *lang ?A* = *lang ?B*)
proof −
let *?σ* = (*λi*. (*if i*=*0 then Some* (*Atom x*) *else* (*if i*=*1 then Some* (*Atom*

y) *else None*)))

let *?e1* = *Star* (*Plus* (*Atom 1*) (*Atom 3*))
let *?e2* = (*Times* (*Star* (*Plus* (*seq* [*Plus* (*Atom 1*) *One*, *Star* (*Times* (*Atom 3*) (*Atom 1*)), *Atom 3*, *Atom 3*])
(*seq* [*Plus* (*Atom 3*) *One*, *Star* (*Times* (*Atom 1*) (*Atom 3*)), *Atom 1*, *Atom 1*])))
(*Plus One*
(*Plus*
(*seq*
[*Plus* (*Atom 1*)
One,
Star
(*Times* (*Atom 3*)
(*Atom 1*)),
Atom 3])
(*seq*
[*Plus* (*Atom 3*)
One,
Star
(*Times* (*Atom 1*)
(*Atom 3*)),
Atom 1]))))
have *lang ?A* = *lang* (*subst ?e1 ?σ*) **by**(*simp add*: *myUNIV_def*)
thm *soundness*
also have ... = *lang* (*subst ?e2 ?σ*)
apply(*rule lift*)
apply(*rule soundness*)
by *eval*
also have ... = *lang ?B* **by** (*simp add*: *L_lasthasxx_def nodouble_def*)
finally show *?thesis* **.**
qed

definition *mycasexxy x y* = *Plus* (*seq*[*Star* (*Plus* (*Atom x*) (*Atom y*)), *Atom x*, *Atom x*, *Atom y*])
(*seq*[*Star* (*Plus* (*Atom x*) (*Atom y*)), *Atom y*, *Atom y*, *Atom x*])
definition *mycasexyx x y* = *Plus* (*seq*[*Star* (*Plus* (*Atom x*) (*Atom y*)), *Atom x*, *Atom y*, *Atom x*])
(*seq*[*Star* (*Plus* (*Atom x*) (*Atom y*)), *Atom y*, *Atom x*, *Atom y*])
definition *mycasexx x y* = *Plus* (*seq*[*Star* (*Plus* (*Atom x*) (*Atom y*)), *Atom x*, *Atom x*])
(*seq*[*Star* (*Plus* (*Atom x*) (*Atom y*)), *Atom y*, *Atom y*])
definition *mycasexy x y* = *Plus* (*seq*[*Atom x*, *Atom y*]) (*seq*[*Atom y*, *Atom*

x])
definition *mycasex x y* = *Plus* (*Atom y*) (*Atom x*)

definition *mycases x y* = *Plus*
    (*mycasexxy x y*)
  (*Plus* (*mycasexyx x y*)
  (*Plus* (*mycasexx x y*)
    (*Plus* (*mycasexy x y*) (*Plus* (*mycasex x y*) (*One*)))))

lemma *mycases_char*: *lang* (*myUNIV* (*x*::*nat*) *y*) = *lang* (*mycases x y*) (**is** *lang ?A* = *lang ?B*)
proof −
  let *?σ* = (*λi*. (*if i=0 then Some* (*Atom x*) *else* (*if i=1 then Some* (*Atom y*) *else None*)))

  let *?e1* = *Star* (*Plus* (*Atom 1*) (*Atom 3*))
  let *?e2* = *Plus* (*Plus* (*seq* [*Star* (*Plus* (*Atom 1*) (*Atom 3*)), *Atom 1*, *Atom 1*, *Atom 3*])
    (*seq* [*Star* (*Plus* (*Atom 1*) (*Atom 3*)), *Atom 3*, *Atom 3*, *Atom 1*]))
  (*Plus* (*Plus* (*seq* [*Star* (*Plus* (*Atom 1*) (*Atom 3*)), *Atom 1*, *Atom 3*, *Atom 1*])
    (*seq* [*Star* (*Plus* (*Atom 1*) (*Atom 3*)), *Atom 3*, *Atom 1*, *Atom 3*]))
  (*Plus* (*Plus* (*seq* [*Star* (*Plus* (*Atom 1*) (*Atom 3*)), *Atom 1*, *Atom 1*])
    (*seq* [*Star* (*Plus* (*Atom 1*) (*Atom 3*)), *Atom 3*, *Atom 3*]))
  (*Plus* (*Plus* (*seq* [*Atom 1*, *Atom 3*]) (*seq* [*Atom 3*, *Atom 1*])) (*Plus* (*Plus* (*Atom 3*) (*Atom 1*)) *One*))))

  have *lang ?A* = *lang* (*subst ?e1 ?σ*) **by**(*simp add*: *myUNIV_def*)
  thm *soundness*
  also have ... = *lang* (*subst ?e2 ?σ*)
    apply(*rule lift*)
    apply(*rule soundness*)
    by *eval*
  also have ... = *lang ?B* **by** (*simp add*: *mycases_def mycasexxy_def mycasexyx_def*
    mycasexx_def mycasex_def mycasexy_def)
  finally show *?thesis* **.**
qed

end

12 OPT2

```
theory OPT2
imports
Partial_Cost_Model
RExp_Var
begin
```

12.1 Definition

```
fun OPT2 :: 'a list ⇒ 'a list ⇒ (nat * nat list) list where
  OPT2 [] [x,y] = []
| OPT2 [a] [x,y] = [(0,[])]
| OPT2 (a#b#σ') [x,y] =  (if a=x then (0,[]) # (OPT2 (b#σ') [x,y])
                          else (if b=x then (0,[])# (OPT2 (b#σ') [x,y])
                                else (1,[])# (OPT2 (b#σ') [y,x])))


lemma OPT2_length: length (OPT2 σ [x, y]) = length σ
apply(induct σ arbitrary: x y)
  apply(simp)
  apply(case_tac σ) by(auto)

lemma OPT2x: OPT2 (x#σ') [x,y] = (0,[])#(OPT2 σ' [x,y])
apply(cases σ') by (simp_all)


lemma swapOpt: T_p_opt [x,y] σ ≤ 1 + T_p_opt [y,x] σ
proof -
  show ?thesis
  proof (cases length σ > 0)
    case True

    have T_p_opt [y,x] σ ∈ {T_p [y, x] σ as |as. length as = length σ}
    unfolding T_opt_def
      apply(rule Inf_nat_def1)
      apply(auto) by (rule Ex_list_of_length)

    then obtain asyx where costyx: T_p [y,x] σ asyx = T_p_opt [y,x] σ
                     and lenyx: length asyx = length σ
             unfolding T_opt_def by auto

    from True lenyx have length asyx > 0 by auto
    then obtain A asyx' where aa: asyx = A # asyx' using list.exhaust
```

by *blast*
then obtain *m1 a1* **where** *AA*: $A = (m1,a1)$ **by** *fastforce*

let *?asxy* = $(m1,a1@[0])$ # *asyx'*

from *True* **obtain** $q\ \sigma'$ **where** *qq*: $\sigma = q \# \sigma'$ **using** *list.exhaust* **by** *blast*

have *t*: t_p [x, y] q (m1, a1@[0]) = Suc (t_p [y, x] q (m1, a1))
unfolding t_p_def
apply(*simp*) **unfolding** *swap_def* **by** (*simp*)

have *s*: *step* [x, y] q (m1, a1 @ [0]) = *step* [y, x] q (m1, a1)
unfolding *step_def mtf2_def* **by**(*simp add*: *swap_def*)

have *T*: T_p [x,y] σ ?asxy = 1 + T_p [y,x] σ *asyx* **unfolding** *qq aa AA* **by**(*auto simp add*: *s t*)

have *l*: 1 + T_p_opt [y,x] σ = T_p [x, y] σ ?asxy **using** *T costyx* **by** *simp*
have *length ?asxy* = *length* σ **using** *lenyx aa* **by** *auto*
then have *inside*: ?asxy ∈ {as. size as = size σ} **by** *force*
then have *b*: T_p [x, y] σ ?asxy ∈ {T_p [x, y] σ as | as. size as = size σ} **by** *auto*

then show *?thesis* **unfolding** *l* **unfolding** *T_opt_def*
apply(*rule cInf_lower*) **by** *simp*
qed (*simp add*: *T_opt_def*)
qed

lemma *tt*: a ∈ {x,y} ⟹ *OPT2* (*rest1*) (*step* [x,y] a (*hd* (*OPT2* (a # *rest1*) [x, y])))
= *tl* (*OPT2* (a # *rest1*) [x, y])
apply(*cases rest1*) **by**(*auto simp add*: *step_def mtf2_def swap_def*)

lemma *splitqsallg*: *Strat* ≠ [] ⟹ a ∈ {x,y} ⟹
t_p [x, y] a (*hd* (*Strat*)) +
(*let L=step* [x,y] a (*hd* (*Strat*))
in T_p L (*rest1*) (*tl Strat*)) = T_p [x, y] (a # *rest1*) *Strat*
proof −
assume *ne*: *Strat* ≠ []
assume *axy*: a ∈ {x,y}
have T_p [x, y] (a # *rest1*) (*Strat*)

$= T_p$ [*x*, *y*] (*a* # *rest1*) ((*hd* (*Strat*)) # (*tl* (*Strat*)))
by(*simp only*: *List.list.collapse*[*OF ne*])
then show *?thesis* **by** *auto*
qed

lemma *splitqs*: *a* ∈ {*x*,*y*} ⟹ T_p [*x*, *y*] (*a* # *rest1*) (*OPT2* (*a* # *rest1*) [*x*, *y*])
= t_p [*x*, *y*] *a* (*hd* (*OPT2* (*a* # *rest1*) [*x*, *y*])) +
(*let L*=*step* [*x*,*y*] *a* (*hd* (*OPT2* (*a* # *rest1*) [*x*, *y*]))
in T_p *L* (*rest1*) (*OPT2* (*rest1*) *L*))
proof −
assume *axy*: *a* ∈ {*x*,*y*}
have *ne*: *OPT2* (*a* # *rest1*) [*x*, *y*] ≠ [] **apply**(*cases rest1*) **by**(*simp_all*)
have T_p [*x*, *y*] (*a* # *rest1*) (*OPT2* (*a* # *rest1*) [*x*, *y*])
= T_p [*x*, *y*] (*a* # *rest1*) ((*hd* (*OPT2* (*a* # *rest1*) [*x*, *y*])) # (*tl* (*OPT2* (*a* # *rest1*) [*x*, *y*])))
by(*simp only*: *List.list.collapse*[*OF ne*])
also have ... = T_p [*x*, *y*] (*a* # *rest1*) ((*hd* (*OPT2* (*a* # *rest1*) [*x*, *y*])) # (*OPT2* (*rest1*) (*step* [*x*,*y*] *a* (*hd* (*OPT2* (*a* # *rest1*) [*x*, *y*])))))
by(*simp only*: *tt*[*OF axy*])
also have ... = t_p [*x*, *y*] *a* (*hd* (*OPT2* (*a* # *rest1*) [*x*, *y*])) +
(*let L*=*step* [*x*,*y*] *a* (*hd* (*OPT2* (*a* # *rest1*) [*x*, *y*]))
in T_p *L* (*rest1*) (*OPT2* (*rest1*) *L*)) **by**(*simp*)
finally show *?thesis* **.**
qed

lemma *tpx*: t_p [*x*, *y*] *x* (*hd* (*OPT2* (*x* # *rest1*) [*x*, *y*])) = *0*
by (*simp add*: *OPT2x* t_p*_def*)

lemma *yup*: T_p [*x*, *y*] (*x* # *rest1*) (*OPT2* (*x* # *rest1*) [*x*, *y*])
= (*let L*=*step* [*x*,*y*] *x* (*hd* (*OPT2* (*x* # *rest1*) [*x*, *y*]))
in T_p *L* (*rest1*) (*OPT2* (*rest1*) *L*))
by (*simp add*: *splitqs tpx*)

lemma *swapsxy*: *A* ∈ { [*x*,*y*], [*y*,*x*]} ⟹ *swaps sws A* ∈ { [*x*,*y*], [*y*,*x*]}
apply(*induct sws*)
apply(*simp*)
apply(*simp*) **unfolding** *swap_def* **by** *auto*

lemma *mtf2xy*: *A* ∈ { [*x*,*y*], [*y*,*x*]} ⟹ *r*∈{*x*,*y*} ⟹ *mtf2 a r A* ∈ { [*x*,*y*], [*y*,*x*]}
by (*metis mtf2_def swapsxy*)

lemma *stepxy*: **assumes** $q \in \{x,y\}$ $A \in \{ [x,y], [y,x]\}$
shows *step A q a* $\in \{ [x,y], [y,x]\}$
unfolding *step_def* **apply**(*simp only*: *split_def Let_def*)
apply(*rule mtf2xy*)
apply(*rule swapsxy*) **by** *fact+*

12.2 Proof of Optimality

lemma *OPT2_is_lb*: *set* $\sigma \subseteq \{x,y\} \Longrightarrow x \neq y \Longrightarrow T_p$ $[x,y]$ σ (*OPT2* σ $[x,y]) \leq T_p$*_opt* $[x,y]$ σ
proof (*induct length* σ *arbitrary*: *x y* σ *rule*: *less_induct*)
case (*less*)
show *?case*
proof (*cases* σ)
case (*Cons a* σ')
note *Cons1=Cons*
show *?thesis* **unfolding** *Cons*
proof(*cases a=x*)
case *True*
from *True Cons* **have** *qsform*: $\sigma = x\#\sigma'$ **by** *auto*
have *up*: T_p $[x, y]$ $(x \# \sigma')$ (*OPT2* $(x \# \sigma')$ $[x, y]) \leq T_p$*_opt* $[x, y]$ $(x \# \sigma')$
unfolding *True*
unfolding *T_opt_def* **apply**(*rule cInf_greatest*)
apply(*simp add*: *Ex_list_of_length*)
proof –
fix *el*
assume $el \in \{T_p$ $[x, y]$ $(x \# \sigma')$ *as* |*as. length as = length* $(x \# \sigma')\}$
then obtain *Strat* **where** *lStrat*: *length Strat = length* $(x \# \sigma')$
and *el*: T_p $[x, y]$ $(x \# \sigma')$ *Strat = el* **by** *auto*
then have *ne*: *Strat* $\neq$ [] **by** *auto*
let *?LA=step* $[x,y]$ *x* (*hd* (*OPT2* $(x \# \sigma')$ $[x, y]$))

have *E0*: T_p $[x, y]$ $(x \# \sigma')$ (*OPT2* $(x \# \sigma')$ $[x, y]$)
$=T_p$ *?LA* (σ') (*OPT2* (σ') *?LA*) **using** *yup* **by** *auto*
also have *E1*: $\ldots = T_p$ $[x,y]$ (σ') (*OPT2* (σ') $[x,y]$) **by** (*simp add*: *OPT2x step_def*)
also have *E2*: $\ldots \leq$ T_p*_opt* $[x,y]$ σ' **apply**(*rule less*(*1*)) **using** *Cons less*(*2*,*3*) **by** *auto*
also have $\ldots \leq T_p$ $[x, y]$ $(x \# \sigma')$ *Strat*
proof (*cases* (*step* $[x, y]$ *x* (*hd Strat*)) = $[x,y]$)
case *True*
have *aha*: T_p*_opt* $[x, y]$ $\sigma' \leq T_p$ $[x, y]$ σ' (*tl Strat*)

unfolding *T_opt_def* **apply**(*rule cInf_lower*)
apply(*auto*) **apply**(*rule exI*[**where** *x*=*tl Strat*]) **using** *lStrat* **by** *auto*

also have *E4*: ... ≤ t_p [*x*, *y*] *x* (*hd Strat*) + T_p (*step* [*x*, *y*] *x* (*hd Strat*)) σ′ (*tl Strat*)
unfolding *True* **by**(*simp*)
also have *E5*: ... = T_p [*x*, *y*] (*x* # σ′) *Strat* **using** *splitqsallg*[*of Strat x x y* σ′, *OF ne*, *simplified*]
by (*auto*)
finally show *?thesis* **by** *auto*
next
case *False*
have *tp1*: t_p [*x*, *y*] *x* (*hd Strat*) ≥ *1*
proof (*rule ccontr*)
let *?a* = *hd Strat*
assume ¬ *1* ≤ t_p [*x*, *y*] *x ?a*
then have *tp0*: t_p [*x*, *y*] *x ?a* = *0* **by** *auto*
then have *size* (*snd ?a*) = *0* **unfolding** *t_p_def* **by**(*simp add*: *split_def*)
then have *nopaid*: (*snd ?a*) = [] **by** *auto*
have *step* [*x*, *y*] *x ?a* = [*x*, *y*]
unfolding *step_def* **apply**(*simp add*: *split_def nopaid*)
unfolding *mtf2_def* **by**(*simp*)
then show *False* **using** *False* **by** *auto*
qed

from *False* **have** *yx*: *step* [*x*, *y*] *x* (*hd Strat*) = [*y*, *x*]
using *stepxy*[**where** *x*=*x* **and** *y*=*y* **and** *a*=*hd Strat*] **by** *auto*

have *E3*: T_p*_opt* [*x*, *y*] σ′ ≤ *1* + T_p*_opt* [*y*, *x*] σ′ **using** *swapOpt* **by** *auto*
also have *E4*: ... ≤ *1* + T_p [*y*, *x*] σ′ (*tl Strat*)
apply(*simp*) **unfolding** *T_opt_def* **apply**(*rule cInf_lower*)
apply(*auto*) **apply**(*rule exI*[**where** *x*=*tl Strat*]) **using** *lStrat* **by** *auto*
also have *E5*: ... = *1* + T_p (*step* [*x*, *y*] *x* (*hd Strat*)) σ′ (*tl Strat*) **using** *yx* **by** *auto*
also have *E6*: ... ≤ t_p [*x*, *y*] *x* (*hd Strat*) + T_p (*step* [*x*, *y*] *x* (*hd Strat*)) σ′ (*tl Strat*) **using** *tp1* **by** *auto*

also have *E7*: ... = T_p [*x*, *y*] (*x* # σ') *Strat* **using** *splitqsallg*[*of Strat x x y* σ', *OF ne*, *simplified*]
by (*auto*)
finally show *?thesis* **by** *auto*
qed
also have ... = *el* **using** *True el* **by** *simp*
finally show T_p [*x*, *y*] (*x* # σ') (*OPT2* (*x* # σ') [*x*, *y*]) ≤ *el* **by** *auto*
qed
then show T_p [*x*, *y*] (*a* # σ') (*OPT2* (*a* # σ') [*x*, *y*]) ≤ T_{p}_*opt* [*x*, *y*] (*a* # σ')
using *True* **by** *simp*
next

case *False*
with *less Cons* **have** *ay*: *a*=*y* **by** *auto*
show T_p [*x*, *y*] (*a* # σ') (*OPT2* (*a* # σ') [*x*, *y*]) ≤ T_{p}_*opt* [*x*, *y*] (*a* # σ') **unfolding** *ay*
proof(*cases* σ')
case *Nil*
have *up*: T_{p}_*opt* [*x*, *y*] [*y*] ≥ *1*
unfolding *T_opt_def* **apply**(*rule cInf_greatest*)
apply(*simp add*: *Ex_list_of_length*)
proof −
fix *el*
assume *el* ∈ {T_p [*x*, *y*] [*y*] *as* |*as*. *length as* = *length* [*y*]}
then obtain *Strat* **where** *Strat*: *length Strat* = *length* [*y*] **and**
el: *el* = T_p [*x*, *y*] [*y*] *Strat* **by** *auto*
from *Strat* **obtain** *a* **where** *a*: *Strat* = [*a*] **by** (*metis Suc_length_conv length_0_conv*)
show *1* ≤ *el* **unfolding** *el a* **apply**(*simp*) **unfolding** t_{p}_*def* **apply**(*simp add*: *split_def*)
apply(*cases snd a*)
apply(*simp add*: *less*(*3*))
by(*simp*)
qed

show T_p [*x*, *y*] (*y* # σ') (*OPT2* (*y* # σ') [*x*, *y*]) ≤ T_{p}_*opt* [*x*, *y*] (*y* # σ') **unfolding** *Nil*
apply(*simp add*: t_{p}_*def*) **using** *less*(*3*) **apply**(*simp*)
using *up* **by**(*simp*)
next
case (*Cons b rest2*)

show *up*: T_p [*x*, *y*] (*y* # σ′) (*OPT2* (*y* # σ′) [*x*, *y*]) ≤ T_p_*opt* [*x*, *y*] (*y* # σ′)
unfolding *Cons*
proof (*cases b*=*x*)
case *True*

show T_p [*x*, *y*] (*y* # *b* # *rest2*) (*OPT2* (*y* # *b* # *rest2*) [*x*, *y*]) ≤ T_p_*opt* [*x*, *y*] (*y* # *b* # *rest2*)
unfolding *True*
unfolding *T_opt_def* **apply**(*rule cInf_greatest*)
apply(*simp add*: *Ex_list_of_length*)
proof −
fix *el*
assume *el* ∈ {T_p [*x*, *y*] (*y* # *x* # *rest2*) *as* |*as*. *length as* = *length* (*y* # *x* # *rest2*)}
then obtain *Strat* **where** *lenStrat*: *length Strat* = *length* (*y* # *x* # *rest2*) **and**
Strat: *el* = T_p [*x*, *y*] (*y* # *x* # *rest2*) *Strat* **by** *auto*
have *v*: *set rest2* ⊆ {*x*, *y*} **using** *less*(*2*)[*unfolded Cons1 Cons*] **by** *auto*

let *?L1* = (*step* [*x*, *y*] *y* (*hd Strat*))
let *?L2* = (*step ?L1 x* (*hd* (*tl Strat*)))

let *?a1* = *hd Strat*
let *?a2* = *hd* (*tl Strat*)
let *?r* = *tl* (*tl Strat*)

have *Strat* = *?a1* # *?a2* # *?r* **by** (*metis Nitpick.size_list_simp*(*2*) *Suc_length_conv lenStrat list.collapse list.discI list.inject*)

have *1*: T_p [*x*, *y*] (*y* # *x* # *rest2*) *Strat*
= t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1 x* (*hd* (*tl Strat*))
+ T_p *?L2 rest2* (*tl* (*tl Strat*))
proof −
have *a*: *Strat* ≠ [] **using** *lenStrat* **by** *auto*
have *b*: (*tl Strat*) ≠ [] **using** *lenStrat* **by** (*metis Nitpick.size_list_simp*(*2*) *Suc_length_conv list.discI list.inject*)

have *1*: T_p [*x*, *y*] (*y* # *x* # *rest2*) *Strat*
= t_p [*x*, *y*] *y* (*hd Strat*) + T_p *?L1* (*x* # *rest2*) (*tl Strat*)
using *splitqsallg*[*OF a*, **where** *a*=*y* **and** *x*=*x* **and** *y*=*y*, *simplified*] **by** (*simp*)
have *tt*: *step* [*x*, *y*] *y* (*hd Strat*) ≠ [*x*, *y*] ⟹ *step* [*x*, *y*] *y* (*hd Strat*) = [*y*,*x*]
using *stepxy*[**where** *A*=[*x*,*y*]] **by** *blast*

have *2*: T_p *?L1* (*x* # *rest2*) (*tl Strat*) = t_p *?L1 x* (*hd* (*tl Strat*)) + T_p *?L2* (*rest2*) (*tl* (*tl Strat*))
apply(*cases ?L1*=[*x*,*y*])
using *splitqsallg*[*OF b*, **where** *a*=*x* **and** *x*=*x* **and** *y*=*y*, *simplified*] **apply**(*auto*)
using *tt splitqsallg*[*OF b*, **where** *a*=*x* **and** *x*=*y* **and** *y*=*x*, *simplified*] **by** *auto*
from *1 2* **show** *?thesis* **by** *auto*
qed

have T_p [*x*, *y*] (*y* # *x* # *rest2*) (*OPT2* (*y* # *x* # *rest2*) [*x*, *y*])
= *1* + T_p [*x*, *y*] (*rest2*) (*OPT2* (*rest2*) [*x*, *y*])
unfolding *True*
using *less*(*3*) **by**(*simp add*: t_p_*def step*_*def OPT2x*)
also have ... ≤ *1* + T_p_*opt* [*x*, *y*] (*rest2*) **apply**(*simp*)
apply(*rule less*(*1*))
apply(*simp add*: *less*(*2*) *Cons1 Cons*)
apply(*fact*) **by** *fact*
also

have ... ≤ T_p [*x*, *y*] (*y* # *x* # *rest2*) *Strat*
proof (*cases ?L2* = [*x*,*y*])
case *True*
have *2*: t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1 x* (*hd* (*tl Strat*))
+ T_p [*x*,*y*] *rest2* (*tl* (*tl Strat*)) ≥ t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1 x* (*hd* (*tl Strat*))
+ T_p_*opt* [*x*,*y*] *rest2* **apply**(*simp*)
unfolding *T*_*opt*_*def* **apply**(*rule cInf*_*lower*)
apply(*simp*) **apply**(*rule exI*[**where** *x*=*tl* (*tl Strat*)]) **by** (*auto simp*: *lenStrat*)
have *3*: t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1 x* (*hd* (*tl Strat*))
+ T_p_*opt* [*x*,*y*] *rest2* ≥ *1* + T_p_*opt* [*x*,*y*] *rest2* **apply**(*simp*)
proof −

have t_p *[x, y] y (hd Strat)* ≥ *1*
unfolding t_p*_def* **apply**(*simp add*: *split_def*)
apply(*cases snd (hd Strat)*) **by** (*simp_all add*: *less(3)*)
then show *Suc 0* ≤ t_p *[x, y] y (hd Strat)* + t_p *?L1 x (hd (tl Strat))* **by** *auto*
qed
from *1 2 3 True* **show** *?thesis* **by** *auto*
next
case *False*
note *L2F=this*
have *L1*: *?L1* ∈ {[*x*, *y*], [*y*, *x*]} **apply**(*rule stepxy*) **by** *simp_all*
have *?L2* ∈ {[*x*, *y*], [*y*, *x*]} **apply**(*rule stepxy*) **using** *L1* **by** *simp_all*
with *False* **have** *2*: *?L2* = [*y*,*x*] **by** *auto*

have *k*: T_p *[x, y] (y # x # rest2) Strat*
= t_p *[x, y] y (hd Strat)* + t_p *?L1 x (hd (tl Strat))* +
T_p *[y,x] rest2 (tl (tl Strat))* **using** *1 2* **by** *auto*

have *l*: t_p *[x, y] y (hd Strat)* > *0*
using *less(3)* **unfolding** t_p*_def* **apply**(*cases snd (hd Strat) = []*)
by (*simp_all add*: *split_def*)

have *r*: T_p *[x, y] (y # x # rest2) Strat* ≥ *2* + T_p *[y,x] rest2 (tl (tl Strat))*
proof (*cases ?L1 = [x,y]*)
case *True*
note *T=this*
then have t_p *?L1 x (hd (tl Strat))* > *0* **unfolding** *True*
proof(*cases snd (hd (tl Strat)) = []*)
case *True*
have *?L2* = [*x*,*y*] **unfolding** *T* **apply**(*simp add*: *split_def step_def*)
unfolding *True mtf2_def* **by**(*simp*)
with *L2F* **have** *False* **by** *auto*
then show *0* < t_p *[x, y] x (hd (tl Strat))* **..**
next
case *False*
then show *0* < t_p *[x, y] x (hd (tl Strat))*
unfolding t_p*_def* **by**(*simp add*: *split_def*)
qed

```
                with l have  t_p [x, y] y (hd Strat) + t_p ?L1 x (hd (tl Strat)) ≥ 2 by auto
               with k show ?thesis by auto
             next
               case False
               from L1 False have 2: ?L1 = [y,x] by auto
               { fix k sws T
                 have T∈{[x,y],[y,x]} ⟹ mtf2 k x T = [y,x] ⟹ T = [y,x]
                   apply(rule ccontr) by (simp add: less(3) mtf2_def)
               }
                 have t1: t_p [x, y] y (hd Strat) ≥ 1 unfolding t_p_def apply(simp add: split_def)
                 apply(cases (snd (hd Strat))) using ‹x ≠ y› by auto
               have t2: t_p [y,x] x (hd (tl Strat)) ≥ 1 unfolding t_p_def apply(simp add: split_def)
               apply(cases (snd (hd (tl Strat)))) using ‹x ≠ y› by auto
               have T_p [x, y] (y # x # rest2) Strat
                   = t_p [x, y] y (hd Strat) + t_p (step [x, y] y (hd Strat)) x (hd (tl Strat)) + T_p [y, x] rest2 (tl (tl Strat))
                     by(rule k)
               with t1 t2 2 show ?thesis by auto
             qed
            have t: T_p [y, x] rest2 (tl (tl Strat)) ≥ T_p_opt [y, x] rest2
               unfolding T_opt_def apply(rule cInf_lower)
                 apply(auto) apply(rule exI[where x=(tl (tl Strat))]) by(simp add: lenStrat)
             show ?thesis
             proof −
              have 1 + T_p_opt [x, y] rest2 ≤ 2 + T_p_opt [y, x] rest2
               using  swapOpt by auto
               also have ... ≤ 2 + T_p [y, x] rest2 (tl (tl Strat)) using t by auto
               also have ... ≤ T_p [x, y] (y # x # rest2) Strat using r by auto
               finally show ?thesis .
             qed

           qed
           also have ... = el using Strat by auto
            finally show T_p [x, y] (y # x # rest2) (OPT2 (y # x # rest2) [x, y]) ≤ el .
         qed
```

next
case *False*
with *Cons1 Cons less*(*2*) **have** *bisy*: *b*=*y* **by** *auto*
with *less*(*3*) **have** *OPT2* (*y* # *b* # *rest2*) [*x*, *y*] = (*1*,[])# (*OPT2* (*b*#*rest2*) [*y*,*x*]) **by** *simp*
show T_p [*x*, *y*] (*y* # *b* # *rest2*) (*OPT2* (*y* # *b* # *rest2*) [*x*, *y*]) ≤ T_p*_opt* [*x*, *y*] (*y* # *b* # *rest2*)
unfolding *bisy*
unfolding *T_opt_def* **apply**(*rule cInf_greatest*)
apply(*simp add*: *Ex_list_of_length*)
proof −
fix *el*
assume *el* ∈ {T_p [*x*, *y*] (*y* # *y* # *rest2*) *as* |*as*. *length as* = *length* (*y* # *y* # *rest2*)}
then obtain *Strat* **where** *lenStrat*: *length Strat* = *length* (*y* # *y* # *rest2*) **and**
Strat: *el* = T_p [*x*, *y*] (*y* # *y* # *rest2*) *Strat* **by** *auto*
have *v*: *set rest2* ⊆ {*x*, *y*} **using** *less*(*2*)[*unfolded Cons1 Cons*] **by** *auto*

let *?L1* = (*step* [*x*, *y*] *y* (*hd Strat*))
let *?L2* = (*step ?L1 y* (*hd* (*tl Strat*)))

let *?a1* = *hd Strat*
let *?a2* = *hd* (*tl Strat*)
let *?r* = *tl* (*tl Strat*)

have *Strat* = *?a1* # *?a2* # *?r* **by** (*metis Nitpick.size_list_simp*(*2*) *Suc_length_conv lenStrat list.collapse list.discI list.inject*)

have *1*: T_p [*x*, *y*] (*y* # *y* # *rest2*) *Strat*
= t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1 y* (*hd* (*tl Strat*))
+ T_p *?L2 rest2* (*tl* (*tl Strat*))
proof −
have *a*: *Strat* ≠ [] **using** *lenStrat* **by** *auto*
have *b*: (*tl Strat*) ≠ [] **using** *lenStrat* **by** (*metis Nitpick.size_list_simp*(*2*) *Suc_length_conv list.discI list.inject*)

have *1*: T_p [*x*, *y*] (*y* # *y* # *rest2*) *Strat*

= t_p [*x*, *y*] *y* (*hd Strat*) + T_p *?L1* (*y* # *rest2*) (*tl Strat*)
using *splitqsallg*[*OF a*, **where** *a*=*y* **and** *x*=*x* **and** *y*=*y*, *simplified*] **by** (*simp*)
have *tt*: *step* [*x*, *y*] *y* (*hd Strat*) ≠ [*x*, *y*] ⟹ *step* [*x*, *y*] *y* (*hd Strat*) = [*y*,*x*]
using *stepxy*[**where** *A*=[*x*,*y*]] **by** *blast*

have *2*: T_p *?L1* (*y* # *rest2*) (*tl Strat*) = t_p *?L1* *y* (*hd* (*tl Strat*)) + T_p *?L2* (*rest2*) (*tl* (*tl Strat*))
apply(*cases ?L1*=[*x*,*y*])
using *splitqsallg*[*OF b*, **where** *a*=*y* **and** *x*=*x* **and** *y*=*y*, *simplified*] **apply**(*auto*)
using *tt splitqsallg*[*OF b*, **where** *a*=*y* **and** *x*=*y* **and** *y*=*x*, *simplified*] **by** *auto*
from *1 2* **show** *?thesis* **by** *auto*
qed

have T_p [*x*, *y*] (*y* # *y* # *rest2*) (*OPT2* (*y* # *y* # *rest2*) [*x*, *y*])
= *1* + T_p [*y*, *x*] (*rest2*) (*OPT2* (*rest2*) [*y*, *x*])
using *less*(*3*) **by**(*simp add*: *t_p_def step_def mtf2_def swap_def OPT2x*)
also have . . . ≤ *1* + *T_p_opt* [*y*, *x*] (*rest2*) **apply**(*simp*)
apply(*rule less*(*1*))
apply(*simp add*: *less*(*2*) *Cons1 Cons*)
using *v less*(*3*) **by**(*auto*)
also

have . . . ≤ T_p [*x*, *y*] (*y* # *y* # *rest2*) *Strat*
proof (*cases ?L2* = [*y*,*x*])
case *True*
have *2*: t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1* *y* (*hd* (*tl Strat*))
+ T_p [*y*,*x*] *rest2* (*tl* (*tl Strat*)) ≥ t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1* *y* (*hd* (*tl Strat*))
+ *T_p_opt* [*y*,*x*] *rest2* **apply**(*simp*)
unfolding *T_opt_def* **apply**(*rule cInf_lower*)
apply(*simp*) **apply**(*rule exI*[**where** *x*=*tl* (*tl Strat*)]) **by** (*auto simp*: *lenStrat*)
have *3*: t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1* *y* (*hd* (*tl Strat*))
+ *T_p_opt* [*y*,*x*] *rest2* ≥ *1* + *T_p_opt* [*y*,*x*] *rest2* **apply**(*simp*)
proof −
have t_p [*x*, *y*] *y* (*hd Strat*) ≥ *1*

unfolding t_p_*def* **apply**(*simp add*: *split_def*)
apply(*cases snd* (*hd Strat*)) **by** (*simp_all add*: *less*(*3*))
then show *Suc 0* ≤ t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1* *y* (*hd* (*tl Strat*)) **by** *auto*
qed
from *1 2 3 True* **show** *?thesis* **by** *auto*
next
case *False*
note *L2F*=*this*
have *L1*: *?L1* ∈ {[*x*, *y*], [*y*, *x*]} **apply**(*rule stepxy*) **by** *simp_all*
have *?L2* ∈ {[*x*, *y*], [*y*, *x*]} **apply**(*rule stepxy*) **using** *L1* **by** *simp_all*
with *False* **have** *2*: *?L2* = [*x*,*y*] **by** *auto*

have *k*: T_p [*x*, *y*] (*y* # *y* # *rest2*) *Strat*
= t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1* *y* (*hd* (*tl Strat*)) +
T_p [*x*,*y*] *rest2* (*tl* (*tl Strat*)) **using** *1 2* **by** *auto*

have *l*: t_p [*x*, *y*] *y* (*hd Strat*) > *0*
using *less*(*3*) **unfolding** t_p_*def* **apply**(*cases snd* (*hd Strat*) = [])
by (*simp_all add*: *split_def*)

have *r*: T_p [*x*, *y*] (*y* # *y* # *rest2*) *Strat* ≥ *2* + T_p [*x*,*y*] *rest2* (*tl* (*tl Strat*))
proof (*cases ?L1* = [*y*,*x*])
case *False*
from *L1 False* **have** *?L1* = [*x*,*y*] **by** *auto*
note *T*=*this*
then have t_p *?L1* *y* (*hd* (*tl Strat*)) > *0* **unfolding** *T*
unfolding t_p_*def* **apply**(*simp add*: *split_def*)
apply(*cases snd* (*hd* (*tl Strat*)) = [])
using ‹*x* ≠ *y*› **by** *auto*
with *l k* **show** *?thesis* **by** *auto*
next

case *True*
note *T*=*this*

have t_p *?L1* *y* (*hd* (*tl Strat*)) > *0* **unfolding** *T*
proof(*cases snd* (*hd* (*tl Strat*)) = [])
case *True*

have *?L2* = [*y*,*x*] **unfolding** *T* **apply**(*simp add*: *split_def step_def*)
unfolding *True mtf2_def* **by**(*simp*)
with *L2F* **have** *False* **by** *auto*
then show *0* < t_p [*y*, *x*] *y* (*hd* (*tl Strat*)) **..**
next
case *False*
then show *0* < t_p [*y*, *x*] *y* (*hd* (*tl Strat*))
unfolding t_p*_def* **by**(*simp add*: *split_def*)
qed
with *l* **have** t_p [*x*, *y*] *y* (*hd Strat*) + t_p *?L1 y* (*hd* (*tl Strat*)) ≥ *2* **by** *auto*
with *k* **show** *?thesis* **by** *auto*

qed
have *t*: T_p [*x*, *y*] *rest2* (*tl* (*tl Strat*)) ≥ T_p*_opt* [*x*, *y*] *rest2*
unfolding *T_opt_def* **apply**(*rule cInf_lower*)
apply(*auto*) **apply**(*rule exI*[**where** *x*=(*tl* (*tl Strat*))]) **by**(*simp add*: *lenStrat*)
show *?thesis*
proof −
have *1* + T_p*_opt* [*y*, *x*] *rest2* ≤ *2* + T_p*_opt* [*x*, *y*] *rest2*
using *swapOpt* **by** *auto*
also have ... ≤ *2* + T_p [*x*, *y*] *rest2* (*tl* (*tl Strat*)) **using** *t* **by** *auto*
also have ... ≤ T_p [*x*, *y*] (*y* # *y* # *rest2*) *Strat* **using** *r* **by** *auto*
finally show *?thesis* **.**
qed

qed
also have ... = *el* **using** *Strat* **by** *auto*
finally show T_p [*x*, *y*] (*y* # *y* # *rest2*) (*OPT2* (*y* # *y* # *rest2*) [*x*, *y*]) ≤ *el* **.**
qed
qed
qed
qed
qed (*simp add*: *T_opt_def*)
qed

lemma *OPT2_is_ub*: *set qs* ⊆ {*x*,*y*} ⟹ *x*≠*y* ⟹ T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) ≥ T_p*_opt* [*x*,*y*] *qs*

unfolding *T_opt_def* **apply**(*rule cInf_lower*)
apply(*simp*) **apply**(*rule exI*[**where** *x*=(*OPT2 qs* [*x*, *y*])])
by (*auto simp add*: *OPT2_length*)

lemma *OPT2_is_opt*: *set qs* ⊆ {*x*,*y*} ⟹ *x*≠*y* ⟹ T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) = T_p*_opt* [*x*,*y*] *qs*
by (*simp add*: *OPT2_is_lb OPT2_is_ub antisym*)

12.3 Performance on the four phase forms

lemma *OPT2_A*: **assumes** *x* ≠ *y qs* ∈ *lang* (*seq* [*Plus* (*Atom x*) *One*, *Atom y*, *Atom y*])
shows T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) = *1*
proof −
from *assms*(*2*) **obtain** *u v* **where** *qs*: *qs*=*u*@*v* **and** *u*: *u*=[*x*] ∨ *u*=[] **and** *v*: *v* = [*y*,*y*] **by** (*auto simp*: *conc_def*)
from *u* **have** *pref1*: T_p [*x*,*y*] (*u*@*v*) (*OPT2* (*u*@*v*) [*x*,*y*]) = T_p [*x*,*y*] *v* (*OPT2 v* [*x*,*y*])
apply(*cases u*=[])
apply(*simp*)
by(*simp add*: *OPT2x* t_p*_def step_def*)

have *ende*: T_p [*x*,*y*] *v* (*OPT2 v* [*x*,*y*]) = *1* **unfolding** *v* **using** *assms*(*1*) **by**(*simp add*: *mtf2_def swap_def* t_p*_def step_def*)

from *pref1 ende qs* **show** *?thesis* **by** *auto*
qed

lemma *OPT2_A'*: **assumes** *x* ≠ *y qs* ∈ *lang* (*seq* [*Plus* (*Atom x*) *One*, *Atom y*, *Atom y*])
shows *real* (T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*])) = *1*
using *OPT2_A*[*OF assms*] **by** *simp*

lemma *OPT2_B*: **assumes** *x* ≠ *y qs*=*u*@*v u*=[] ∨ *u*=[*x*] *v* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) = (*length v div 2*)
proof −
from *assms*(*3*) **have** *pref1*: T_p [*x*,*y*] (*u*@*v*) (*OPT2* (*u*@*v*) [*x*,*y*]) = T_p [*x*,*y*] *v* (*OPT2 v* [*x*,*y*])
apply(*cases u*=[])

apply(*simp*)
by(*simp add*: *OPT2x* t_p*_def step_def*)

from *assms*(*4*) **obtain** *a w* **where** *v*: *v*=*a*@*w* **and** *a*∈*lang* (*Times* (*Atom y*) (*Atom x*)) **and** *w*: *w*∈*lang* (*seq*[*Star*(*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*]) **by**(*auto*)
from *this*(*2*) **have** *aa*: *a*=[*y*,*x*] **by**(*simp add*: *conc_def*)

from *assms*(*1*) *this v* **have** *pref2*: T_p [*x*,*y*] *v* (*OPT2 v* [*x*,*y*]) = *1* + T_p [*x*,*y*] *w* (*OPT2 w* [*x*,*y*])
by(*simp add*: t_p*_def step_def OPT2x*)

from *w* **obtain** *c d* **where** *w2*: *w*=*c*@*d* **and** *c*: *c* ∈ *lang* (*Star* (*Times* (*Atom y*) (*Atom x*))) **and** *d*: *d* ∈ *lang* (*Times* (*Atom y*) (*Atom y*)) **by** *auto*
then have *dd*: *d*=[*y*,*y*] **by** *auto*

from *c*[*simplified*] **have** *star*: T_p [*x*,*y*] (*c*@*d*) (*OPT2* (*c*@*d*) [*x*,*y*]) = (*length c div 2*) + T_p [*x*,*y*] *d* (*OPT2 d* [*x*,*y*])
proof(*induct c rule*: *star_induct*)
case (*append r s*)
then have *r*: *r*=[*y*,*x*] **by** *auto*
then have T_p [*x*, *y*] ((*r* @ *s*) @ *d*) (*OPT2* ((*r* @ *s*) @ *d*) [*x*, *y*]) = T_p [*x*, *y*] ([*y*,*x*] @ (*s* @ *d*)) (*OPT2* ([*y*,*x*] @ (*s* @ *d*)) [*x*, *y*]) **by** *simp*
also have ... = *1* + T_p [*x*, *y*] (*s* @ *d*) (*OPT2* (*s* @ *d*) [*x*, *y*])
using *assms*(*1*) **by**(*simp add*: t_p*_def step_def OPT2x*)
also have ... = *1* + *length s div 2* + T_p [*x*, *y*] *d* (*OPT2 d* [*x*, *y*]) **using** *append* **by** *simp*
also have ... = *length* (*r* @ *s*) *div 2* + T_p [*x*, *y*] *d* (*OPT2 d* [*x*, *y*]) **using** *r* **by** *auto*
finally show *?case* **.**
qed *simp*

have *ende*: T_p [*x*,*y*] *d* (*OPT2 d* [*x*,*y*]) = *1* **unfolding** *dd* **using** *assms*(*1*) **by**(*simp add*: *mtf2_def swap_def* t_p*_def step_def*)

have *vv*: *v* = [*y*,*x*]@*c*@[*y*,*y*] **using** *w2 dd v aa* **by** *auto*

from *pref1 pref2 star w2 ende* **have**
T_p [*x*, *y*] *qs* (*OPT2 qs* [*x*, *y*]) = *1* + *length c div 2* + *1* **unfolding** *assms*(*2*) **by** *auto*
also have ... = (*length v div 2*) **using** *vv* **by** *auto*
finally show *?thesis* **.**
qed

lemma *OPT2_B1*: **assumes** $x \neq y$ *qs* ∈ *lang* (*seq*[*Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows *real* (T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*])) = *length qs* / *2*
proof −
from *assms*(*2*) **have** *qs*: *qs* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
by(*simp add*: *conc_assoc*)
have (*length qs*) *mod 2* = *0*
proof −
from *assms*(*2*) **have** *qs* ∈ ({[*y*]} @@ {[*x*]}) @@ *star*({[*y*]} @@ {[*x*]}) @@ {[*y*]} @@ {[*y*]} **by** (*simp add*: *conc_assoc*)
then obtain *p q r* **where** *pqr*: *qs*=*p*@*q*@*r* **and** *p*∈({[*y*]} @@ {[*x*]})
and *q*: *q* ∈ *star* ({[*y*]} @@ {[*x*]}) **and** *r* ∈{[*y*]} @@ {[*y*]} **by** (*metis concE*)
then have *rr*: *p* = [*y*,*x*] *r*=[*y*,*y*] **by** *auto*
with *pqr* **have** *a*: *length qs* = *4*+*length q* **by** *auto*
from *q* **have** *b*: *length q mod 2* = *0*
apply(*induct q rule*: *star_induct*) **by** (*auto*)
from *a b* **show** *?thesis* **by** *auto*
qed
with *OPT2_B*[**where** *u*=[], *OF assms*(*1*) _ _ *qs*] **show** *?thesis* **by** *auto*
qed

lemma *OPT2_B2*: **assumes** $x \neq y$ *qs* ∈ *lang* (*seq*[*Atom x*, *Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) = ((*length qs* − *1*) / *2*)
proof −
from *assms*(*2*) **obtain** *v* **where**
qsv: *qs* = [*x*]@*v* **and** *vv*: *v* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*]) **by** (*auto simp add*: *conc_def*)
have (*length v*) *mod 2* = *0*
proof −
from *vv* **have** *v* ∈ ({[*y*]} @@ {[*x*]}) @@ *star*({[*y*]} @@ {[*x*]}) @@ {[*y*]} @@ {[*y*]} **by** (*simp add*: *conc_assoc*)
then obtain *p q r* **where** *pqr*: *v*=*p*@*q*@*r* **and** *p*∈({[*y*]} @@ {[*x*]})
and *q*: *q* ∈ *star* ({[*y*]} @@ {[*x*]}) **and** *r* ∈{[*y*]} @@ {[*y*]} **by** (*metis concE*)
then have *rr*: *p* = [*y*,*x*] *r*=[*y*,*y*] **by**(*auto simp add*: *conc_def*)
with *pqr* **have** *a*: *length v* = *4*+*length q* **by** *auto*
from *q* **have** *b*: *length q mod 2* = *0*
apply(*induct q rule*: *star_induct*) **by** (*auto*)
from *a b* **show** *?thesis* **by** *auto*
qed

with *OPT2_B*[**where** *u*=[*x*], *OF assms*(*1*) *qsv* _ *vv*] *qsv* **show** *?thesis* **by**(*auto*)
qed

lemma *OPT2_C*: **assumes** $x \neq y$ *qs*=*u*@*v* *u*=[] ∨ *u*=[*x*]
and *v* ∈ *lang* (*seq*[*Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom x*])
shows T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) = (*length v div 2*)
proof −
from *assms*(*3*) **have** *pref1*: T_p [*x*,*y*] (*u*@*v*) (*OPT2* (*u*@*v*) [*x*,*y*]) = T_p [*x*,*y*] *v* (*OPT2 v* [*x*,*y*])
apply(*cases u*=[])
apply(*simp*)
by(*simp add*: *OPT2x* t_p_*def step_def*)

from *assms*(*4*) **obtain** *a w* **where** *v*: *v*=*a*@*w* **and** *aa*: *a*=[*y*,*x*] **and** *w*: *w*∈*lang* (*seq*[*Star*(*Times* (*Atom y*) (*Atom x*)), *Atom x*]) **by**(*auto simp*: *conc_def*)

from *assms*(*1*) *this v* **have** *pref2*: T_p [*x*,*y*] *v* (*OPT2 v* [*x*,*y*]) = *1* + T_p [*x*,*y*] *w* (*OPT2 w* [*x*,*y*])
by(*simp add*: t_p_*def step_def OPT2x*)

from *w* **obtain** *c d* **where** *w2*: *w*=*c*@*d* **and** *c*: *c* ∈ *lang* (*Star* (*Times* (*Atom y*) (*Atom x*))) **and** *d*: *d* ∈ *lang* (*Atom x*) **by** *auto*
then have *dd*: *d*=[*x*] **by** *auto*

from *c*[*simplified*] **have** *star*: T_p [*x*,*y*] (*c*@*d*) (*OPT2* (*c*@*d*) [*x*,*y*]) = (*length c div 2*) + T_p [*x*,*y*] *d* (*OPT2 d* [*x*,*y*]) ∧ (*length c*) *mod 2* = *0*
proof(*induct c rule*: *star_induct*)
case (*append r s*)
from *append* **have** *mod*: *length s mod 2* = *0* **by** *simp*
from *append* **have** *r*: *r*=[*y*,*x*] **by** *auto*
then have T_p [*x*, *y*] ((*r* @ *s*) @ *d*) (*OPT2* ((*r* @ *s*) @ *d*) [*x*, *y*]) = T_p [*x*, *y*] ([*y*,*x*] @ (*s* @ *d*)) (*OPT2* ([*y*,*x*] @ (*s* @ *d*)) [*x*, *y*]) **by** *simp*
also have ... = *1* + T_p [*x*, *y*] (*s* @ *d*) (*OPT2* (*s* @ *d*) [*x*, *y*])
using *assms*(*1*) **by**(*simp add*: t_p_*def step_def OPT2x*)
also have ... = *1* + *length s div 2* + T_p [*x*, *y*] *d* (*OPT2 d* [*x*, *y*]) **using** *append* **by** *simp*
also have ... = *length* (*r* @ *s*) *div 2* + T_p [*x*, *y*] *d* (*OPT2 d* [*x*, *y*]) **using** *r* **by** *auto*
finally show *?case* **by**(*simp add*: *mod r*)
qed *simp*

have *ende*: T_p [*x*,*y*] *d* (*OPT2 d* [*x*,*y*]) = *0* **unfolding** *dd* **using** *assms*(*1*)
by(*simp add*: *mtf2_def swap_def* t_p*_def step_def*)

have *vv*: *v* = [*y*,*x*]@*c*@[*x*] **using** *w2 dd v aa* **by** *auto*

from *pref1 pref2 star w2 ende* **have**
T_p [*x*, *y*] *qs* (*OPT2 qs* [*x*, *y*]) = *1* + *length c div 2* **unfolding** *assms*(*2*)
by *auto*
also have ... = (*length v div 2*) **using** *vv star* **by** *auto*
finally show *?thesis* **.**
qed

lemma *OPT2_C1*: **assumes** *x* ≠ *y qs* ∈ *lang* (*seq*[*Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom x*])
shows *real* (T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*])) = (*length qs* − *1*) / *2*
proof −
from *assms*(*2*) **have** *qs*: *qs* ∈ *lang* (*seq*[*Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom x*])
by(*simp add*: *conc_assoc*)
have (*length qs*) *mod 2* = *1*
proof −
from *assms*(*2*) **have** *qs* ∈ ({[*y*]} @@ {[*x*]}) @@ *star*({[*y*]} @@ {[*x*]}) @@ {[*x*]} **by** (*simp add*: *conc_assoc*)
then obtain *p q r* **where** *pqr*: *qs*=*p*@*q*@*r* **and** *p*∈({[*y*]} @@ {[*x*]})
and *q*: *q* ∈ *star* ({[*y*]} @@ {[*x*]}) **and** *r* ∈{[*x*]} **by** (*metis concE*)
then have *rr*: *p* = [*y*,*x*] *r*=[*x*] **by** *auto*
with *pqr* **have** *a*: *length qs* = *3*+*length q* **by** *auto*
from *q* **have** *b*: *length q mod 2* = *0*
apply(*induct q rule*: *star_induct*) **by** (*auto*)
from *a b* **show** *?thesis* **by** *auto*
qed
with *OPT2_C*[**where** *u*=[], *OF assms*(*1*) _ _ *qs*] **show** *?thesis*
by (*simp add*: *field_char_0_class.of_nat_div*)
qed

lemma *OPT2_C2*: **assumes** *x* ≠ *y qs* ∈ *lang* (*seq*[*Atom x*, *Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom x*])
shows T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) = ((*length qs* − *2*) / *2*)
proof −
from *assms*(*2*) **obtain** *v* **where**
qsv: *qs* = [*x*]@*v* **and** *vv*: *v* ∈ *lang* (*seq*[*Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom x*]) **by** (*auto simp add*: *conc_def*)
have (*length v*) *mod 2* = *1*
proof −

from *vv* **have** $v \in (\{[y]\}$ @@ $\{[x]\})$ @@ $star(\{[y]\}$ @@ $\{[x]\})$ @@ $\{[x]\}$
by (*simp add*: *conc_assoc*)
then obtain *p q r* **where** *pqr*: *v=p@q@r* **and** $p\in(\{[y]\}$ @@ $\{[x]\})$
and *q*: $q \in star$ $(\{[y]\}$ @@ $\{[x]\})$ **and** $r \in\{[x]\}$ **by** (*metis concE*)
then have *rr*: $p = [y,x]$ $r=[x]$ **by**(*auto simp add*: *conc_def*)
with *pqr* **have** *a*: *length v = 3+length q* **by** *auto*
from *q* **have** *b*: *length q mod 2 = 0*
apply(*induct q rule*: *star_induct*) **by** (*auto*)
from *a b* **show** *?thesis* **by** *auto*
qed
with *OPT2_C*[**where** *u=[x]*, *OF assms(1) qsv _ vv*] *qsv* **show** *?thesis*
by (*simp add*: *field_char_0_class.of_nat_div*)
qed

lemma *OPT2_ub*: *set qs* $\subseteq \{x,y\} \Longrightarrow T_p$ *[x,y] qs* (*OPT2 qs [x,y]*) $\leq$ *length qs*
proof(*induct qs arbitrary*: *x y*)
case (*Cons q qs*)
then have *set qs* $\subseteq \{x,y\}$ $q\in\{x,y\}$ **by** *auto*
note *Cons1=Cons this*
show *?case*
proof (*cases qs*)
case *Nil*
with *Cons1* **show** T_p *[x,y] (q # qs)* (*OPT2 (q # qs) [x,y]*) $\leq$ *length (q # qs)*
apply(*simp add*: t_p*_def*) **by** *blast*
next
case (*Cons q′ qs′*)
with *Cons1* **have** $q'\in\{x,y\}$ **by** *auto*
note *Cons=Cons this*

from *Cons1 Cons* **have** *T*: T_p *[x, y] qs* (*OPT2 qs [x, y]*) $\leq$ *length qs*
T_p *[y, x] qs* (*OPT2 qs [y, x]*) $\leq$ *length qs* **by** *auto*
show T_p *[x,y] (q # qs)* (*OPT2 (q # qs) [x,y]*) $\leq$ *length (q # qs)*
unfolding *Cons* **apply**(*simp only*: *OPT2.simps*)
apply(*split if_splits(1)*)
apply(*safe*)
proof (*goal_cases*)
case *1*
have T_p *[x, y] (x # q′ # qs′)* ((*0*, []) # *OPT2 (q′ # qs′) [x, y]*)
= t_p *[x, y] x (0,[])* + T_p *[x, y] qs* (*OPT2 qs [x, y]*)
by(*simp add*: *step_def Cons*)

also have ... ≤ t_p [*x*, *y*] *x* (*0*,[]) + *length qs* **using** *T* **by** *auto*
also have ... ≤ *length* (*x* # *q*′ # *qs*′) **using** *Cons* **by**(*simp add*: t_p_*def*)
finally show *?case* **.**
next
case *2*
with *Cons1 Cons* **show** *?case*
apply(*split if_splits*(*1*))
apply(*safe*)
proof (*goal_cases*)
case *1*
then have T_p [*x*, *y*] (*y* # *x* # *qs*′) ((*0*, []) # *OPT2* (*x* # *qs*′) [*x*, *y*])
= t_p [*x*, *y*] *y* (*0*,[]) + T_p [*x*, *y*] *qs* (*OPT2 qs* [*x*, *y*])
by(*simp add*: *step_def*)
also have ... ≤ t_p [*x*, *y*] *y* (*0*,[]) + *length qs* **using** *T* **by** *auto*
also have ... ≤ *length* (*y* # *x* # *qs*′) **using** *Cons* **by**(*simp add*: t_p_*def*)
finally show *?case* **.**
next
case *2*
then have T_p [*x*, *y*] (*y* # *y* # *qs*′) ((*1*, []) # *OPT2* (*y* # *qs*′) [*y*, *x*])
= t_p [*x*, *y*] *y* (*1*,[]) + T_p [*y*, *x*] *qs* (*OPT2 qs* [*y*, *x*])
by(*simp add*: *step_def mtf2_def swap_def*)
also have ... ≤ t_p [*x*, *y*] *y* (*1*,[]) + *length qs* **using** *T* **by** *auto*
also have ... ≤ *length* (*y* # *y* # *qs*′) **using** *Cons* **by**(*simp add*: t_p_*def*)
finally show *?case* **.**
qed
qed
qed
qed *simp*

lemma *OPT2_padded*: *R*∈{[*x*,*y*],[*y*,*x*]} ⟹ *set qs* ⊆ {*x*,*y*}
⟹ T_p *R* (*qs*@[*x*,*x*]) (*OPT2* (*qs*@[*x*,*x*]) *R*)
≤ T_p *R* (*qs*@[*x*]) (*OPT2* (*qs*@[*x*]) *R*) + *1*
apply(*induct qs arbitrary*: *R*)
apply(*simp*)
apply(*case_tac R*=[*x*,*y*])
apply(*simp add*: *step_def* t_p_*def*)
apply(*simp add*: *step_def mtf2_def swap_def* t_p_*def*)

```
  proof (goal_cases)
    case (1 a qs R)
    then have a: a ∈ {x,y} by auto
    with 1 show ?case
      apply(cases qs)
        apply(cases a=x)
          apply(cases R=[x,y])
            apply(simp add: step_def t_p_def)
            apply(simp add: step_def mtf2_def swap_def t_p_def)
          apply(cases R=[x,y])
            apply(simp add: step_def t_p_def)
            apply(simp add: step_def mtf2_def swap_def t_p_def)
      proof (goal_cases)
        case (1 p ps)
        show ?case
          apply(cases a=x)
            apply(cases R=[x,y])
              apply(simp add: OPT2x step_def) using 1 apply(simp)
              using 1(2) apply(simp)
                apply(cases qs)
                  apply(simp add: step_def mtf2_def swap_def t_p_def)
                  using 1 by(auto simp add: swap_def mtf2_def step_def)
       qed
qed


lemma OPT2_split11:
  assumes xy: x≠y
  shows R∈{[x,y],[y,x]} ⟹ set xs ⊆ {x,y} ⟹ set ys ⊆ {x,y} ⟹ OPT2
(xs@[x,x]@ys) R = OPT2 (xs@[x,x]) R @ OPT2 ys [x,y]
proof (induct xs arbitrary: R)
  case Nil
  then show ?case
  apply(simp)
  apply(cases ys)
    apply(simp)
    apply(cases R=[x,y])
      apply(simp)
      by(simp)
next
  case (Cons a as)
  note iH=this
  then have AS: set as ⊆ {x,y} and A: a ∈ {x,y} by auto
  note iH=Cons(1)[where R=[y,x], simplified, OF AS Cons(4)]
```

```
  note iH'=Cons(1)[where R=[x,y], simplified, OF AS Cons(4)]
  show ?case
  proof (cases R=[x,y])
    case True
    note R=this
    from iH iH' show ?thesis
    apply(cases a=x)
      apply(simp add: R OPT2x)
      using A apply(simp)
      apply(cases as)
        apply(simp add: R)
        using AS apply(simp)
        apply(case_tac aa=x)
          by(simp_all add: R)
  next
    case False
    with Cons(2) have R: R=[y,x] by auto
    from iH iH' show ?thesis
    apply(cases a=y)
      apply(simp add: R OPT2x)
      using A apply(simp)
      apply(cases as)
        apply(simp add: R)
        apply(case_tac aa=y)
          by (simp_all add: R)
   qed
qed
```

12.4 The function steps

```
lemma steps_append: length qs = length as ⟹ steps s (qs@[q]) (as@[a])
= step (steps s qs as) q a
  by (induct qs as arbitrary: s rule: list_induct2) simp_all

end
```

13 Phase Partitioning

```
theory Phase_Partitioning
imports OPT2
begin
```

13.1 Definition of Phases

definition *other a x y* = (*if a*=*x then y else x*)

definition *Lxx* **where**
Lxx (*x*::*nat*) *y* = *lang* (*L_lasthasxx x y*)

lemma *Lxx_not_nullable*: [] ∉ *Lxx x y*
unfolding *Lxx_def L_lasthasxx_def* **by** *simp*

lemma *Lxx_ends_in_two_equal*: *xs* ∈ *Lxx x y* ⟹ ∃ *pref e*. *xs* = *pref* @ [*e*,*e*]
by(*auto simp*: *conc_def Lxx_def L_lasthasxx_def*)

lemma *Lxx x y* = *Lxx y x* **unfolding** *Lxx_def* **by**(*rule lastxx_com*)

definition *hideit x y* = (*Plus rexp.One* (*nodouble x y*))

lemma *Lxx_othercase*: *set qs* ⊆ {*x*,*y*} ⟹ ¬ (∃ *xs ys*. *qs* = *xs* @ *ys* ∧ *xs* ∈ *Lxx x y*) ⟹ *qs* ∈ *lang* (*hideit x y*)
proof −
  assume *set qs* ⊆ {*x*,*y*}
  then have *qs* ∈ *lang* (*myUNIV x y*) **using** *myUNIV_alle*[*of x y*] **by** *blast*
  then have *qs* ∈ *star* (*lang* (*L_lasthasxx x y*)) @@ *lang* (*hideit x y*) **unfolding** *hideit_def*
    by(*auto simp add*: *myUNIV_char*)
  then have *qs*: *qs* ∈ *star* (*Lxx x y*) @@ *lang* (*hideit x y*) **by**(*simp add*: *Lxx_def*)
  assume *notpos*: ¬ (∃ *xs ys*. *qs* = *xs* @ *ys* ∧ *xs* ∈ *Lxx x y*)
  show *qs* ∈ *lang* (*hideit x y*)
  proof −
    from *qs* **obtain** *A B* **where** *qsAB*: *qs*=*A*@*B* **and** *A*: *A*∈*star* (*Lxx x y*) **and** *B*: *B*∈*lang* (*hideit x y*) **by** *auto*
    with *notpos* **have** *notin*: *A* ∉ (*Lxx x y*) **by** *blast*

    from *A* **have** *1*: *A* = [] ∨ *A* ∈ (*Lxx x y*) @@ *star* (*Lxx x y*) **using** *Regular_Set.star_unfold_left* **by** *auto*
    have *2*: *A* ∉ (*Lxx x y*) @@ *star* (*Lxx x y*)
    proof (*rule ccontr*)

```
      assume ¬ A ∉ Lxx x y @@ star (Lxx x y)
      then have  A ∈ Lxx x y @@ star (Lxx x y) by auto
      then obtain A1 A2 where A=A1@A2 and A1: A1∈(Lxx x y) and
A2∈ star (Lxx x y) by auto
      with qsAB have qs=A1@(A2@B) A1∈(Lxx x y) by auto
      with notpos have A1 ∉ (Lxx x y) by blast
      with A1 show False by auto
    qed
    from 1 2 have A=[] by auto
    with qsAB have qs=B by auto
    with B show ?thesis by simp
  qed
qed


fun pad where pad xs x y = (if xs=[] then [x,x] else
                                    (if last xs = x then xs @ [x] else xs @ [y]))

lemma pad_adds2: qs ≠ [] ⟹ set qs ⊆ {x,y} ⟹ pad qs x y = qs @ [last
qs]
apply(auto) by (metis insertE insert_absorb insert_not_empty last_in_set
subset_iff)


lemma nodouble_padded: qs ≠ [] ⟹ qs ∈ lang (nodouble x y) ⟹ pad qs
x y ∈ Lxx x y
proof -
  assume nn: qs ≠ []
  assume qs ∈ lang (nodouble x y)
  then have a: qs ∈ lang          (seq
          [Plus (Atom x) rexp.One,
           Star (Times (Atom y) (Atom x)),
           Atom y]) ∨ qs ∈ lang
        (seq
          [Plus (Atom y) rexp.One,
           Star (Times (Atom x) (Atom y)),
           Atom x])  unfolding nodouble_def by auto


  show ?thesis
  proof (cases qs ∈ lang (seq [Plus (Atom x) One, Star (Times (Atom y)
(Atom x)), Atom y]))
    case True
    then have qs ∈ lang (seq [Plus (Atom x) One, Star (Times (Atom y)
```

(*Atom x*))]) @@ {[*y*]}
by(*simp add*: *conc_assoc*)
then have *last qs* = *y* **by** *auto*
with *nn* **have** *p*: *pad qs x y* = *qs* @ [*y*] **by** *auto*
have *A*: *pad qs x y* ∈ *lang* (*seq* [*Plus* (*Atom x*) *One*, *Star* (*Times* (*Atom y*) (*Atom x*)),
Atom y]) @@ {[*y*]} **unfolding** *p*
apply(*simp*)
apply(*rule concI*)
using *True* **by** *auto*
have *B*: *lang* (*seq* [*Plus* (*Atom x*) *One*, *Star* (*Times* (*Atom y*) (*Atom x*)),
Atom y]) @@ {[*y*]} = *lang* (*seq* [*Plus* (*Atom x*) *One*, *Star* (*Times* (*Atom y*) (*Atom x*)),
Atom y, *Atom y*]) **by** (*simp add*: *conc_assoc*)
show *pad qs x y* ∈ *Lxx x y* **unfolding** *Lxx_def L_lasthasxx_def*
using *B A* **by** *auto*
next
case *False*
with *a* **have** *T*: *qs* ∈ *lang* (*seq* [*Plus* (*Atom y*) *One*, *Star* (*Times* (*Atom x*) (*Atom y*)), *Atom x*]) **by** *auto*

then have *qs* ∈ *lang* (*seq* [*Plus* (*Atom y*) *One*, *Star* (*Times* (*Atom x*) (*Atom y*))]) @@ {[*x*]}
by(*simp add*: *conc_assoc*)
then have *last qs* = *x* **by** *auto*
with *nn* **have** *p*: *pad qs x y* = *qs* @ [*x*] **by** *auto*
have *A*: *pad qs x y* ∈ *lang* (*seq* [*Plus* (*Atom y*) *One*, *Star* (*Times* (*Atom x*) (*Atom y*)),
Atom x]) @@ {[*x*]} **unfolding** *p*
apply(*simp*)
apply(*rule concI*)
using *T* **by** *auto*
have *B*: *lang* (*seq* [*Plus* (*Atom y*) *One*, *Star* (*Times* (*Atom x*) (*Atom y*)),
Atom x]) @@ {[*x*]} = *lang* (*seq* [*Plus* (*Atom y*) *One*, *Star* (*Times* (*Atom x*) (*Atom y*)),
Atom x, *Atom x*]) **by** (*simp add*: *conc_assoc*)
show *pad qs x y* ∈ *Lxx x y* **unfolding** *Lxx_def L_lasthasxx_def*
using *B A* **by** *auto*
qed
qed

thm *UnE*

lemma $c \in A \cup B \Longrightarrow P$
apply(*erule UnE*) **oops**

lemma *LxxE*: $qs \in Lxx\ x\ y$
$\Longrightarrow (qs \in lang\ (seq\ [Atom\ x,\ Atom\ x]) \Longrightarrow P\ x\ y\ qs)$
$\Longrightarrow (qs \in lang\ (seq\ [Plus\ (Atom\ x)\ rexp.One,\ Atom\ y,\ Atom\ x,\ Star\ (Times\ (Atom\ y)\ (Atom\ x)),\ Atom\ y,\ Atom\ y]) \Longrightarrow P\ x\ y\ qs)$
$\Longrightarrow (qs \in lang\ (seq\ [Plus\ (Atom\ x)\ rexp.One,\ Atom\ y,\ Atom\ x,\ Star\ (Times\ (Atom\ y)\ (Atom\ x)),\ Atom\ x]) \Longrightarrow P\ x\ y\ qs)$
$\Longrightarrow (qs \in lang\ (seq\ [Plus\ (Atom\ x)\ rexp.One,\ Atom\ y,\ Atom\ y]) \Longrightarrow P\ x\ y\ qs)$
$\Longrightarrow P\ x\ y\ qs$
unfolding *Lxx_def lastxx_is_4cases*[*symmetric*] *L_4cases_def* **apply**(*simp only*: *verund.simps lang.simps*)
using *UnE* **by** *blast*

thm *UnE LxxE*

lemma $qs \in Lxx\ x\ y \Longrightarrow P$
apply(*erule LxxE*) **oops**

lemma *LxxI*: $(qs \in lang\ (seq\ [Atom\ x,\ Atom\ x]) \Longrightarrow P\ x\ y\ qs)$
$\Longrightarrow (qs \in lang\ (seq\ [Plus\ (Atom\ x)\ rexp.One,\ Atom\ y,\ Atom\ x,\ Star\ (Times\ (Atom\ y)\ (Atom\ x)),\ Atom\ y,\ Atom\ y]) \Longrightarrow P\ x\ y\ qs)$
$\Longrightarrow (qs \in lang\ (seq\ [Plus\ (Atom\ x)\ rexp.One,\ Atom\ y,\ Atom\ x,\ Star\ (Times\ (Atom\ y)\ (Atom\ x)),\ Atom\ x]) \Longrightarrow P\ x\ y\ qs)$
$\Longrightarrow (qs \in lang\ (seq\ [Plus\ (Atom\ x)\ rexp.One,\ Atom\ y,\ Atom\ y]) \Longrightarrow P\ x\ y\ qs)$
$\Longrightarrow (qs \in Lxx\ x\ y \Longrightarrow P\ x\ y\ qs)$
unfolding *Lxx_def lastxx_is_4cases*[*symmetric*] *L_4cases_def* **apply**(*simp only*: *verund.simps lang.simps*)
by *blast*

lemma *Lxx1*: $xs \in Lxx\ x\ y \Longrightarrow length\ xs \geq 2$
apply(*rule LxxI*[**where** $P=(\lambda x\ y\ qs.\ length\ qs \geq 2)$])
apply(*auto*) **by**(*auto simp*: *conc_def*)

13.2 OPT2 Splitting

lemma *ayay*: $length\ qs = length\ as \Longrightarrow T_p\ s\ (qs@[q])\ (as@[a]) = T_p\ s\ qs\ as + t_p\ (steps\ s\ qs\ as)\ q\ a$
apply(*induct qs as arbitrary*: *s rule*: *list_induct2*) **by** *simp_all*

lemma *tlofOPT2*: *Q* ∈ {*x*,*y*} ⟹ *set QS* ⊆ {*x*,*y*} ⟹ *R* ∈ {[*x*, *y*], [*y*, *x*]}
⟹ *tl* (*OPT2* ((*Q* # *QS*) @ [*x*, *x*]) *R*) =
  OPT2 (*QS* @ [*x*, *x*]) (*step R Q* (*hd* (*OPT2* ((*Q* # *QS*) @ [*x*, *x*]) *R*)))
   apply(*cases Q*=*x*)
    apply(*cases R*=[*x*,*y*])
     apply(*simp add*: *OPT2x step_def*)
     apply(*simp*)
      apply(*cases QS*)
        apply(*simp add*: *step_def mtf2_def swap_def*)
        apply(*simp add*: *step_def mtf2_def swap_def*)
    apply(*cases R*=[*x*,*y*])
     apply(*simp*)
      apply(*cases QS*)
        apply(*simp add*: *step_def mtf2_def swap_def*)
        apply(*simp add*: *step_def mtf2_def swap_def*)
     by(*simp add*: *OPT2x step_def*)

lemma T_p*_split*: *length qs1*=*length as1* ⟹ T_p *s* (*qs1*@*qs2*) (*as1*@*as2*)
= T_p *s qs1 as1* + T_p (*steps s qs1 as1*) *qs2 as2*
apply(*induct qs1 as1 arbitrary*: *s rule*: *list_induct2*) **by**(*simp_all*)

lemma T_p*_spliting*: *x*≠*y* ⟹ *set xs* ⊆ {*x*,*y*} ⟹ *set ys* ⊆ {*x*,*y*} ⟹
   R ∈ {[*x*,*y*],[*y*,*x*]} ⟹
   T_p *R* (*xs*@[*x*,*x*]) (*OPT2* (*xs*@[*x*,*x*]) *R*) + T_p [*x*,*y*] *ys* (*OPT2 ys* [*x*,*y*])
   = T_p *R* (*xs*@[*x*,*x*]@*ys*) (*OPT2* (*xs*@[*x*,*x*]@*ys*) *R*)
proof −
 assume *nxy*: *x*≠*y*
 assume *XSxy*: *set xs* ⊆ {*x*,*y*}
 assume *YSxy*: *set ys* ⊆ {*x*,*y*}
 assume *R*: *R* ∈ {[*x*,*y*],[*y*,*x*]}
 {
  fix *R*
  assume *XSxy*: *set xs* ⊆ {*x*,*y*}
  have *R*∈{[*x*,*y*],[*y*,*x*]} ⟹ *set xs* ⊆ {*x*,*y*} ⟹ *steps R* (*xs*@[*x*,*x*]) (*OPT2*
(*xs*@[*x*,*x*]) *R*) = [*x*,*y*]
  proof(*induct xs arbitrary*: *R*)
   case *Nil*
   then show *?case*
    apply(*cases R*=[*x*,*y*])

     apply *simp_all* **apply**(*simp add*: *step_def*)
     by(*simp add*: *step_def mtf2_def swap_def*)
  next

case (*Cons Q QS*)
let *?R′*=(*step R Q* (*hd* (*OPT2* ((*Q* # *QS*) @ [*x*, *x*]) *R*)))

have *a*: *Q* ∈ {*x*,*y*} **and** *b*: *set QS* ⊆ {*x*,*y*} **using** *Cons* **by** *auto*
have *t*: *?R′* ∈ {[*x*,*y*],[*y*,*x*]}
apply(*rule stepxy*) **using** *nxy Cons* **by** *auto*
then have *length* (*OPT2* (*QS* @ [*x*, *x*]) *?R′*) > *0*
apply(*cases ?R′* = [*x*,*y*]) **by**(*simp_all add*: *OPT2_length*)
then have *OPT2* (*QS* @ [*x*, *x*]) *?R′* ≠ [] **by** *auto*
then have *hdtl*: *OPT2* (*QS* @ [*x*, *x*]) *?R′* = *hd* (*OPT2* (*QS* @ [*x*, *x*]) *?R′*) # *tl* (*OPT2* (*QS* @ [*x*, *x*]) *?R′*)
by *auto*

have *maa*: (*tl* (*OPT2* ((*Q* # *QS*) @ [*x*, *x*]) *R*)) = *OPT2* (*QS* @ [*x*, *x*]) *?R′*
using *tlofOPT2*[*OF a b Cons*(*2*)] **by** *auto*

from *Cons*(*2*) **have** *length* (*OPT2* ((*Q* # *QS*) @ [*x*, *x*]) *R*) > *0*
apply(*cases R* = [*x*,*y*]) **by**(*simp_all add*: *OPT2_length*)
then have *nempty*: *OPT2* ((*Q* # *QS*) @ [*x*, *x*]) *R* ≠ [] **by** *auto*
then have *steps R* ((*Q* # *QS*) @ [*x*, *x*]) (*OPT2* ((*Q* # *QS*) @ [*x*, *x*]) *R*)
= *steps R* ((*Q* # *QS*) @ [*x*, *x*]) (*hd*(*OPT2* ((*Q* # *QS*) @ [*x*, *x*]) *R*) # *tl*(*OPT2* ((*Q* # *QS*) @ [*x*, *x*]) *R*))
by(*simp*)
also have ...
= *steps ?R′* (*QS* @ [*x*,*x*]) (*tl* (*OPT2* ((*Q* # *QS*) @ [*x*, *x*]) *R*))
unfolding *maa* **by** *auto*
also have ... = *steps ?R′* (*QS* @ [*x*,*x*]) (*OPT2* (*QS* @ [*x*, *x*]) *?R′*) **using** *maa* **by** *auto*
also with *Cons*(*1*)[*OF t b*] **have** ... = [*x*,*y*] **by** *auto*

finally show *?case* **.**
qed
} **note** *aa*=*this*

from *aa XSxy R* **have** *ll*: *steps R* (*xs*@[*x*,*x*]) (*OPT2* (*xs*@[*x*,*x*]) *R*)
= [*x*,*y*] **by** *auto*

have *uer*: *length* (*xs* @ [*x*, *x*]) = *length* (*OPT2* (*xs* @ [*x*, *x*]) *R*)
using *R* **by** (*auto simp*: *OPT2_length*)

have *OPT2* (*xs* @ [*x*, *x*] @ *ys*) *R* = *OPT2* (*xs* @ [*x*, *x*]) *R* @ *OPT2* *ys* [*x*, *y*]
  apply(*rule OPT2_split11*)
    using *nxy XSxy YSxy R* **by** *auto*

then have T_p *R* (*xs*@[*x*,*x*]@*ys*) (*OPT2* (*xs*@[*x*,*x*]@*ys*) *R*)
  = T_p *R* ((*xs*@[*x*,*x*])@*ys*) (*OPT2* (*xs* @ [*x*, *x*]) *R* @ *OPT2* *ys* [*x*, *y*])
by *auto*
also have ... = T_p *R* (*xs*@[*x*,*x*]) (*OPT2* (*xs* @ [*x*, *x*]) *R*)
  + T_p [*x*,*y*] *ys* (*OPT2* *ys* [*x*, *y*])
  using T_p*_split*[*of xs*@[*x*,*x*] *OPT2* (*xs* @ [*x*, *x*]) *R R ys OPT2 ys* [*x*, *y*], *OF uer*, *unfolded ll*]
  by *auto*
finally show *?thesis* **by** *simp*
qed

lemma *OPTauseinander*: *x*≠*y* ⟹ *set xs* ⊆ {*x*,*y*} ⟹ *set ys* ⊆ {*x*,*y*} ⟹
  LTS ∈ {[*x*,*y*],[*y*,*x*]} ⟹ *hd LTS* = *last xs* ⟹
  xs = (*pref* @ [*hd LTS*, *hd LTS*]) ⟹
  T_p [*x*,*y*] *xs* (*OPT2 xs* [*x*,*y*]) + T_p *LTS ys* (*OPT2 ys LTS*)
  = T_p [*x*,*y*] (*xs*@*ys*) (*OPT2* (*xs*@*ys*) [*x*,*y*])
proof −
  assume *nxy*: *x*≠*y*
  assume *xsxy*: *set xs* ⊆ {*x*,*y*}
  assume *ysxy*: *set ys* ⊆ {*x*,*y*}
  assume *L*: *LTS* ∈ {[*x*,*y*],[*y*,*x*]}
  assume *hd LTS* = *last xs*
  assume *prefix*: *xs* = (*pref* @ [*hd LTS*, *hd LTS*])
  show *?thesis*
    proof (*cases LTS* = [*x*,*y*])
      case *True*
      show *?thesis* **unfolding** *True prefix*
        apply(*simp*)
        apply(*rule* T_p*_spliting*[*simplified*])
          using *nxy xsxy ysxy prefix* **by** *auto*
    next
      case *False*
      with *L* **have** *TT*: *LTS* = [*y*,*x*] **by** *auto*
      show *?thesis* **unfolding** *TT prefix*
        apply(*simp*)
        apply(*rule* T_p*_spliting*[*simplified*])
          using *nxy xsxy ysxy prefix* **by** *auto*

  qed
qed

13.3 Phase Partitioning lemma

theorem *Phase_partitioning_general*:
  fixes P :: (*nat state* $*$ *'is*) *pmf* $\Rightarrow$ *nat* $\Rightarrow$ *nat list* $\Rightarrow$ *bool*
    and A :: (*nat state*,*'is*,*nat*,*answer*) *alg_on_rand*
  assumes *xny*: (*x0*::*nat*) $\neq$ *y0*
    and *cpos*: (*c*::*real*)$\geq$*0*
    and *static*: *set* $\sigma \subseteq$ {*x0*,*y0*}
    and *initial*: *P* (*map_pmf* (%*is*. ([*x0*,*y0*],*is*)) (*fst A* [*x0*,*y0*])) *x0* [*x0*,*y0*]
    and *D*: $\bigwedge$*a b* σ *s*. $\sigma \in$ *Lxx a b* $\Longrightarrow$ *a*$\neq$*b* $\Longrightarrow$ {*a*,*b*}={*x0*,*y0*} $\Longrightarrow$ *P s a* [*x0*,*y0*] $\Longrightarrow$ *set* $\sigma \subseteq$ {*a*,*b*}
      $\Longrightarrow$ *T_on_rand' A s* $\sigma \leq c * T_p$ [*a*,*b*] σ (*OPT2* σ [*a*,*b*]) $\wedge$ *P* (*config'_rand A s* σ) (*last* σ) [*x0*,*y0*]
  shows T_p*_on_rand A* [*x0*,*y0*] σ $\leq c * T_p$*_opt* [*x0*,*y0*] $\sigma + c$
proof –

 {
  fix *x y s*
 have $x \neq y \Longrightarrow$ *P s x* [*x0*,*y0*] $\Longrightarrow$ *set* $\sigma \subseteq$ {*x*,*y*} $\Longrightarrow$ {*x*,*y*}={*x0*,*y0*} $\Longrightarrow$ *T_on_rand' A s* $\sigma \leq c * T_p$ [*x*,*y*] σ (*OPT2* σ [*x*,*y*]) + *c*
 proof (*induction length* σ *arbitrary*: σ *x y s rule*: *less_induct*)
  case (*less* σ)

  show *?case*
  proof (*cases* $\exists$*xs ys*. σ=*xs*@*ys* $\wedge$ *xs* $\in$ *Lxx x y*)
   case *True*

    then obtain *xs ys* **where** *qs*: σ=*xs*@*ys* **and** *xsLxx*: *xs* $\in$ *Lxx x y* **by** *auto*

   with *Lxx1* **have** *len*: *length ys* < *length* σ **by** *fastforce*
   from *qs*(*1*) *less*(*4*) **have** *ysxy*: *set ys* $\subseteq$ {*x*,*y*} **by** *auto*

   have *xsset*: *set xs* $\subseteq$ {*x*, *y*} **using** *less*(*4*) *qs* **by** *auto*
   from *xsLxx Lxx1* **have** *lxsgt1*: *length xs* $\geq$ *2* **by** *auto*
   then have *xs_not_Nil*: *xs* $\neq$ [] **by** *auto*

   from *D*[*OF xsLxx less*(*2*) *less*(*5*) *less*(*3*) *xsset*]
    have *D1*: *T_on_rand' A s xs* $\leq c * T_p$ [*x*, *y*] *xs* (*OPT2 xs* [*x*, *y*])
      and *inv*: *P* (*config'_rand A s xs*) (*last xs*) [*x0*, *y0*] **by** *auto*

from *xsLxx Lxx_ends_in_two_equal* **obtain** *pref e* **where** *xs* = *pref* @ [*e*,*e*] **by** *metis*
then have *endswithsame*: *xs* = *pref* @ [*last xs*, *last xs*] **by** *auto*

let *?c′* = [*last xs*, *other* (*last xs*) *x y*]

have *setys*: *set ys* ⊆ {*x*,*y*} **using** *qs less* **by** *auto*
have *setxs*: *set xs* ⊆ {*x*,*y*} **using** *qs less* **by** *auto*
have *lxs*: *last xs* ∈ *set xs* **using** *xs_not_Nil* **by** *auto*
from *lxs setxs* **have** *lxsxy*: *last xs* ∈ {*x*,*y*} **by** *auto*
from *lxs setxs* **have** *otherxy*: *other* (*last xs*) *x y* ∈ {*x*,*y*} **by** (*simp add*: *other_def*)
from *less*(*2*) **have** *other_diff*: *last xs* ≠ *other* (*last xs*) *x y* **by**(*simp add*: *other_def*)

have *lo*: {*last xs*, *other* (*last xs*) *x y*} = {*x0*, *y0*}
using *lxsxy otherxy other_diff less*(*5*) **by** *force*

have *nextstate*: {[*last xs*, *other* (*last xs*) *x y*], [*other* (*last xs*) *x y*, *last xs*]}
= { [*x*,*y*],[*y*,*x*]} **using** *lxsxy otherxy other_diff* **by** *fastforce*
have *setys′*: *set ys* ⊆ {*last xs*, *other* (*last xs*) *x y*}
using *setys lxsxy otherxy other_diff* **by** *fastforce*

have *c*: *T_on_rand′ A* (*config′_rand A s xs*) *ys*
≤ $c * T_p$ *?c′ ys* (*OPT2 ys ?c′*) + *c*
apply(*rule less*(*1*))
apply(*fact len*)
apply(*fact other_diff*)
apply(*fact inv*)
apply(*fact setys′*)
by(*fact lo*)

have *well*: T_p [*x*, *y*] *xs* (*OPT2 xs* [*x*, *y*]) + T_p *?c′ ys* (*OPT2 ys ?c′*)
= T_p [*x*, *y*] (*xs* @ *ys*) (*OPT2* (*xs* @ *ys*) [*x*, *y*])
apply(*rule OPTauseinander*[**where** *pref*=*pref*])
apply(*fact*)+
using *lxsxy other_diff otherxy* **apply**(*fastforce*)
apply(*simp*)
using *endswithsame* **by** *simp*

have *E0*: *T_on_rand′ A s σ*

= *T_on_rand' A s* (*xs*@*ys*) **using** *qs* **by** *auto*
also have *E1*: ... = *T_on_rand' A s xs* + *T_on_rand' A* (*config'_rand A s xs*) *ys*
by (*rule T_on_rand'_append*)
also have *E2*: ... ≤ *T_on_rand' A s xs* + $c * T_p$ *?c' ys* (*OPT2 ys ?c'*) + *c*
using *c* **by** *simp*
also have *E3*: ... ≤ $c * T_p$ [*x*, *y*] *xs* (*OPT2 xs* [*x*, *y*]) + $c * T_p$ *?c' ys* (*OPT2 ys ?c'*) + *c*
using *D1* **by** *simp*
also have ... = $c *$ (T_p [*x*,*y*] *xs* (*OPT2 xs* [*x*,*y*]) + T_p *?c' ys* (*OPT2 ys ?c'*)) + *c*
using *cpos* **apply**(*auto*) **by** *algebra*
also have ... = $c *$ (T_p [*x*,*y*] (*xs*@*ys*) (*OPT2* (*xs*@*ys*) [*x*,*y*])) + *c*
using *well* **by** *auto*
also have *E4*: ... = $c *$ (T_p [*x*,*y*] σ (*OPT2* σ [*x*,*y*])) + *c*
using *qs* **by** *auto*
finally show *?thesis* **.**
next
case *False*
note *f1*=*this*
from *Lxx_othercase*[*OF less*(*4*) *this*, *unfolded hideit_def*] **have**
nodouble: σ = [] ∨ σ ∈ *lang* (*nodouble x y*) **by** *auto*
show *?thesis*
proof (*cases* σ = [])
case *True*
then show *?thesis* **using** *cpos* **by** *simp*
next
case *False*

from *False nodouble* **have** *qsnodouble*: σ ∈ *lang* (*nodouble x y*) **by** *auto*
let *?padded* = *pad* σ *x y*

have *padset*: *set ?padded* ⊆ {*x*, *y*} **using** *less*(*4*) **by**(*simp*)

from *False pad_adds2*[*of* σ *x y*] *less*(*4*) **obtain** *addum* **where** *ui*: *pad* σ *x y* = σ @ [*last* σ] **by** *auto*
from *nodouble_padded*[*OF False qsnodouble*] **have** *pLxx*: *?padded* ∈ *Lxx x y* **.**

have *E0*: *T_on_rand' A s* σ ≤ *T_on_rand' A s ?padded*
proof −
have *T_on_rand' A s* σ = *sum* (*T_on_rand'_n A s* σ) {..<*length* σ}

by(*rule T_on_rand'_as_sum*)
also have ...
= *sum* (*T_on_rand'_n A s* (σ @ [*last* σ])) {..<*length* σ}
proof(*rule sum.cong*, *goal_cases*)
case (*2 t*)
then have *t* < *length* σ **by** *auto*
then show *?case* **by**(*simp add*: *nth_append*)
qed *simp*
also have ... ≤ *T_on_rand' A s ?padded*
unfolding *ui*
apply(*subst* (*2*) *T_on_rand'_as_sum*) **by**(*simp add*: *T_on_rand'_nn del*: *T_on_rand'.simps*)
finally show *?thesis* **by** *auto*
qed

also have *E1*: ... ≤ *c* ∗ T_p [*x*,*y*] *?padded* (*OPT2 ?padded* [*x*,*y*])
using *D*[*OF pLxx less*(*2*) *less*(*5*) *less*(*3*) *padset*] **by** *simp*
also have *E2*: ... ≤ *c* ∗ (T_p [*x*,*y*] σ (*OPT2* σ [*x*,*y*]) + *1*)
proof −
from *False less*(*2*) **obtain** σ' *x'* *y'* **where** *qs'*: σ = σ' @ [*x'*] **and** *x'*: *x'* = *last* σ *y'*≠*x'* *y'* ∈{*x*,*y*}
by (*metis append_butlast_last_id insert_iff*)
have *tla*: *last* σ ∈ {*x*,*y*} **using** *less*(*4*) *False last_in_set* **by** *blast*
with *x'* **have** *grgr*: {*x*,*y*} = {*x'*,*y'*} **by** *auto*
then have (*x* = *x'* ∧ *y* = *y'*) ∨ (*x* = *y'* ∧ *y* = *x'*) **using** *less*(*2*) **by** *auto*
then have *tts*: [*x*, *y*] ∈ {[*x'*, *y'*], [*y'*, *x'*]} **by** *blast*

from *qs' ui* **have** *pd*: *?padded* = σ' @ [*x'*, *x'*] **by** *auto*

have T_p [*x*,*y*] (*?padded*) (*OPT2* (*?padded*) [*x*,*y*])
= T_p [*x*,*y*] (σ' @ [*x'*, *x'*]) (*OPT2* (σ' @ [*x'*, *x'*]) [*x*,*y*])
unfolding *pd* **by** *simp*
also have *gr*: ...
≤ T_p [*x*,*y*] (σ' @ [*x'*]) (*OPT2* (σ' @ [*x'*]) [*x*,*y*]) + *1*
apply(*rule OPT2_padded*[**where** *x*=*x'* **and** *y*=*y'*])
apply(*fact*)
using *grgr qs' less*(*4*) **by** *auto*
also have ... ≤ T_p [*x*,*y*] (σ) (*OPT2* (σ) [*x*,*y*]) + *1*
unfolding *qs'* **by** *simp*
finally show *?thesis* **using** *cpos* **by** (*meson mult_left_mono of_nat_le_iff*)
qed
also have ... = *c* ∗ T_p [*x*,*y*] σ (*OPT2* σ [*x*,*y*]) + *c* **by** (*metis* (*no_types*, *lifting*) *mult.commute of_nat_1 of_nat_add semiring_normalization_rules*(*2*))

```
      finally show ?thesis .
    qed
  qed
qed
} note allg=this

 have T_on_rand A [x0,y0] σ ≤ c * real (T_p [x0, y0] σ (OPT2 σ [x0,
y0])) + c
  apply(rule allg)
    apply(fact)
    using initial apply(simp add: map_pmf_def)
    apply(fact assms(3))
    by simp
  also have ... = c * T_p_opt [x0, y0] σ + c
    using OPT2_is_opt[OF assms(3,1)] by(simp)
  finally show ?thesis .
qed

term A::(nat,'is) alg_on

theorem Phase_partitioning_general_det:
  fixes P :: (nat state * 'is) ⇒ nat ⇒ nat list ⇒ bool
      and A :: (nat,'is) alg_on
  assumes xny: (x0::nat) ≠ y0
    and cpos: (c::real)≥0
    and static: set σ ⊆ {x0,y0}
    and initial: P ([x0,y0],(fst A [x0,y0])) x0 [x0,y0]
    and D: ⋀a b σ s.  σ ∈ Lxx a b ⟹ a≠b ⟹ {a,b}={x0,y0} ⟹ P s a
[x0,y0]  ⟹ set σ ⊆ {a,b}
          ⟹ T_on' A s σ ≤ c * T_p [a,b] σ (OPT2 σ [a,b])  ∧ P (config' A
s σ) (last σ) [x0,y0]
  shows T_p_on A [x0,y0] σ  ≤ c * T_p_opt [x0,y0] σ + c
proof -
  thm Phase_partitioning_general

  thm T_deter_rand
  term T_on'
  term embed
  show ?thesis oops



end
```

14 List factoring technique

theory *List_Factoring*
imports
  Partial_Cost_Model
  MTF2_Effects
begin

hide_const *config compet*

14.1 Helper functions

14.1.1 Helper lemmas

lemma *befaf*: **assumes** *q∈set s distinct s*
shows *before q s* ∪ {*q*} ∪ *after q s* = *set s*
proof −
  have *before q s* ∪ {*y. index s y* = *index s q* ∧ *q* ∈ *set s*}
      = {*y. index s y* ≤ *index s q* ∧ *q* ∈ *set s*}
      unfolding *before_in_def* **apply**(*auto*) **by** (*simp add*: *le_neq_implies_less*)
  also have ... = {*y. index s y* ≤ *index s q* ∧ *y*∈ *set s* ∧ *q* ∈ *set s*}
   apply(*auto*) **by** (*metis index_conv_size_if_notin index_less_size_conv not_less*)
  also with ‹*q* ∈ *set s*› **have** ... = {*y. index s y* ≤ *index s q* ∧ *y*∈ *set s*} **by** *auto*
  finally have *before q s* ∪ {*y. index s y* = *index s q* ∧ *q* ∈ *set s*} ∪ *after q s*
      = {*y. index s y* ≤ *index s q* ∧ *y*∈ *set s*} ∪ {*y. index s y* > *index s q* ∧ *y* ∈ *set s*}
      unfolding *before_in_def* **by** *simp*
  also have ... = *set s* **by** *auto*
  finally show *?thesis* **using** *assms* **by** *simp*
qed

lemma *index_sum*: **assumes** *distinct s q∈set s*
shows *index s q* = (∑ *e*∈*set s. if e* < *q in s then 1 else 0*)
proof −
  from *assms* **have** *bia_empty*: *before q s* ∩ ({*q*} ∪ *after q s*) = {}
    by(*auto simp*: *before_in_def*)
  from *befaf*[*OF assms*(*2*) *assms*(*1*)] **have** (∑ *e*∈*set s. if e* < *q in s then 1*::*nat else 0*)
    = (∑ *e*∈(*before q s* ∪ {*q*} ∪ *after q s*). *if e* < *q in s then 1 else 0*) **by** *auto*
  also have ... = (∑ *e*∈*before q s. if e* < *q in s then 1 else 0*)
          + (∑ *e*∈{*q*}. *if e* < *q in s then 1 else 0*) + (∑ *e*∈*after q s. if e* <

q in s then 1 else 0)
  proof −
    have ($\sum$ *e*∈(*before q s* ∪ {*q*} ∪ *after q s*). *if e* < *q in s then 1*::*nat else 0*)
    = ($\sum$ *e*∈(*before q s* ∪ ({*q*} ∪ *after q s*)). *if e* < *q in s then 1*::*nat else 0*)
      by *simp*
    also have … = ($\sum$ *e*∈*before q s*. *if e* < *q in s then 1 else 0*)
      + ($\sum$ *e*∈({*q*} ∪ *after q s*). *if e* < *q in s then 1 else 0*)
      − ($\sum$ *e*∈(*before q s* ∩ ({*q*} ∪ *after q s*)). *if e* < *q in s then 1 else 0*)
      apply(*rule sum_Un_nat*) **by**(*simp_all*)
    also have … = ($\sum$ *e*∈*before q s*. *if e* < *q in s then 1 else 0*)
      + ($\sum$ *e*∈({*q*} ∪ *after q s*). *if e* < *q in s then 1 else 0*) **using** *bia_empty* **by** *auto*
    also have … = ($\sum$ *e*∈*before q s*. *if e* < *q in s then 1 else 0*)
      + ($\sum$ *e*∈{*q*}. *if e* < *q in s then 1 else 0*) + ($\sum$ *e*∈*after q s*. *if e* < *q in s then 1 else 0*)
      by (*simp add*: *before_in_def*)
    finally show *?thesis* **.**
  qed
  also have … = ($\sum$ *e*∈*before q s*. *1*) + ($\sum$ *e*∈({*q*} ∪ *after q s*). *0*) **apply**(*auto*)
    unfolding *before_in_def* **by** *auto*
  also have … = *card* (*before q s*) **by** *auto*
  also have … = *card* (*set* (*take* (*index s q*) *s*)) **using** *before_conv_take*[*OF assms*(*2*)] **by** *simp*
  also have … = *length* (*take* (*index s q*) *s*) **using** *distinct_card assms*(*1*) *distinct_take* **by** *metis*
  also have … = *min* (*length s*) (*index s q*) **by** *simp*
  also have … = *index s q* **using** *index_le_size*[*of s q*] **by**(*auto*)
  finally show *?thesis* **by** *simp*
qed

14.1.2 ALG

fun *ALG* :: *'a* ⇒ *'a list* ⇒ *nat* ⇒ (*'a list* ∗ *'is*) ⇒ *nat* **where**
  ALG x qs i s = (*if x* < (*qs*!*i*) *in fst s then 1*::*nat else 0*)

lemma t_p*_sumofALG*: *distinct* (*fst s*) ⟹ *snd a* = [] ⟹ (*qs*!*i*)∈*set* (*fst s*)
  ⟹ t_p (*fst s*) (*qs*!*i*) *a* = ($\sum$ *e*∈*set* (*fst s*). *ALG e qs i s*)
unfolding t_p*_def* **apply**(*simp add*: *split_def*)
  using *index_sum* **by** *metis*

lemma t_p*_sumofALGreal*: **assumes** *distinct* (*fst s*) *snd a* = [] *qs*!*i* ∈ *set*(*fst s*)
shows *real*(t_p (*fst s*) (*qs*!*i*) *a*) = (∑ *e*∈*set* (*fst s*). *real*(*ALG e qs i s*))
proof −
  from *assms* **have** *real*(t_p (*fst s*) (*qs*!*i*) *a*) = *real*(∑ *e*∈*set* (*fst s*). *ALG e qs i s*)
    using t_p*_sumofALG* **by** *metis*
  also have ... = (∑ *e*∈*set* (*fst s*). *real* (*ALG e qs i s*))
    by *auto*
  finally show *?thesis* **.**
qed

14.1.3 The function steps'

fun *steps'* **where**
  steps' s _ _ 0 = *s*
| *steps' s* [] [] (*Suc n*) = *s*
| *steps' s* (*q*#*qs*) (*a*#*as*) (*Suc n*) = *steps'* (*step s q a*) *qs as n*

lemma *steps'_steps*: *length as* = *length qs* ⟹ *steps' s as qs* (*length as*) = *steps s as qs*
by(*induct arbitrary*: *s rule*: *list_induct2*, *simp_all*)

lemma *steps'_length*: *length qs* = *length as* ⟹ *n* ≤ *length as*
  ⟹ *length* (*steps' s qs as n*) = *length s*
apply(*induct qs as arbitrary*: *s n rule*: *list_induct2*)
  apply(*simp*)
  apply(*case_tac n*)
    by (*auto*)

lemma *steps'_set*: *length qs* = *length as* ⟹ *n* ≤ *length as*
  ⟹ *set* (*steps' s qs as n*) = *set s*
apply(*induct qs as arbitrary*: *s n rule*: *list_induct2*)
  apply(*simp*)
  apply(*case_tac n*)
    by(*auto simp*: *set_step*)

lemma *steps'_distinct2*: *length qs* = *length as* ⟹ *n* ≤ *length as*
  ⟹ *distinct s* ⟹ *distinct* (*steps' s qs as n*)
apply(*induct qs as arbitrary*: *s n rule*: *list_induct2*)
  apply(*simp*)
  apply(*case_tac n*)

by(*auto simp*: *distinct_step*)

lemma *steps'_distinct*: *length qs* = *length as* ⟹ *length as* = *n*
⟹ *distinct* (*steps' s qs as n*) = *distinct s*
by (*induct qs as arbitrary*: *s n rule*: *list_induct2*) (*fastforce simp add*: *distinct_step*)+

lemma *steps'_dist_perm*: *length qs* = *length as* ⟹ *length as* = *n*
⟹ *dist_perm s s* ⟹ *dist_perm* (*steps' s qs as n*) (*steps' s qs as n*)
using *steps'_set steps'_distinct* **by** *blast*

lemma *steps'_rests*: *length qs* = *length as* ⟹ *n* ≤ *length as* ⟹ *steps' s qs as n* = *steps' s* (*qs*@*r1*) (*as*@*r2*) *n*
apply(*induct qs as arbitrary*: *s n rule*: *list_induct2*)
apply(*simp*) **apply**(*case_tac n*) **by** *auto*

lemma *steps'_append*: *length qs* = *length as* ⟹ *length qs* = *n* ⟹ *steps' s* (*qs*@[*q*]) (*as*@[*a*]) (*Suc n*) = *step* (*steps' s qs as n*) *q a*
apply(*induct qs as arbitrary*: *s n rule*: *list_induct2*) **by** *auto*

14.1.4 *ALG'_det*

definition *ALG'_det Strat qs init i x* = *ALG x qs i* (*swaps* (*snd* (*Strat*!*i*)) (*steps' init qs Strat i*),())

lemma *ALG'_det_append*: *n* < *length Strat* ⟹ *n* < *length qs* ⟹ *ALG'_det Strat* (*qs*@*a*) *init n x*
= *ALG'_det Strat qs init n x*
proof −
assume *qs*: *n* < *length qs*
assume *S*: *n* < *length Strat*

have *tt*: (*qs* @ *a*) ! *n* = *qs* ! *n*
using *qs* **by** (*simp add*: *nth_append*)

have *steps' init* (*take n qs*) (*take n Strat*) *n* = *steps' init* ((*take n qs*) @ *drop n qs*) ((*take n Strat*) @ (*drop n Strat*)) *n*
apply(*rule steps'_rests*)
using *S qs* **by** *auto*
then have *A*: *steps' init* (*take n qs*) (*take n Strat*) *n* = *steps' init qs Strat n* **by** *auto*
have *steps' init* (*take n qs*) (*take n Strat*) *n* = *steps' init* ((*take n qs*) @ ((*drop n qs*)@*a*)) ((*take n Strat*) @((*drop n Strat*)@[])) *n*

apply(*rule steps′_rests*)
using *S qs* **by** *auto*
then have *B*: *steps′ init* (*take n qs*) (*take n Strat*) *n* = *steps′ init* (*qs*@*a*) (*Strat*@[]) *n*
by (*metis append_assoc List.append_take_drop_id*)
from *A B* **have** *steps′ init qs Strat n* = *steps′ init* (*qs*@*a*) (*Strat*@[]) *n* **by** *auto*
then have *C*: *steps′ init qs Strat n* = *steps′ init* (*qs*@*a*) *Strat n* **by** *auto*

show *?thesis* **unfolding** *ALG′_det_def C*
unfolding *ALG.simps tt* **by** *auto*
qed

14.1.5 ALG'

abbreviation *config″ A qs init n* == *config_rand A init* (*take n qs*)

definition *ALG′ A qs init i x* = *E*(*map_pmf* (*ALG x qs i*) (*config″ A qs init i*))

lemma *ALG′_refl*: *qs*!*i* = *x* ⟹ *ALG′ A qs init i x* = *0*
unfolding *ALG′_def* **by**(*simp add*: *split_def before_in_def*)

14.1.6 *ALGxy_det*

definition *ALGxy_det* **where**
ALGxy_det A qs init x y = (∑ *i*∈{..<*length qs*}. (*if* (*qs*!*i* ∈ {*y*,*x*}) *then* *ALG′_det A qs init i y* + *ALG′_det A qs init i x*
else 0::*nat*))

lemma *ALGxy_det_alternativ*: *ALGxy_det A qs init x y*
= (∑ *i*∈{*i*. *i*<*length qs* ∧ (*qs*!*i* ∈ {*y*,*x*})}. *ALG′_det A qs init i y* + *ALG′_det A qs init i x*)
proof −
have *f*: {*i*. *i*<*length qs*} = {..<*length qs*} **by**(*auto*)

have *e*: {*i*. *i*<*length qs* ∧ (*qs*!*i* ∈ {*y*,*x*})} = {*i*. *i*<*length qs*} ∩ {*i*. (*qs*!*i* ∈ {*y*,*x*})}
by *auto*
have (∑ *i*∈{*i*. *i*<*length qs* ∧ (*qs*!*i* ∈ {*y*,*x*})}. *ALG′_det A qs init i y* + *ALG′_det A qs init i x*)
= (∑ *i*∈{*i*. *i*<*length qs*} ∩ {*i*. (*qs*!*i* ∈ {*y*,*x*})}. *ALG′_det A qs init i y* + *ALG′_det A qs init i x*)
unfolding *e* **by** *simp*

also have ... = ($\sum$ *i*∈{*i*. *i*<*length qs*}. (*if i* ∈ {*i*. (*qs*!*i* ∈ {*y*,*x*})} *then* *ALG'_det A qs init i y* + *ALG'_det A qs init i x*
else 0))
apply(*rule sum.inter_restrict*) **by** *auto*
also have ... = ($\sum$ *i*∈{..<*length qs*}. (*if i* ∈ {*i*. (*qs*!*i* ∈ {*y*,*x*})} *then* *ALG'_det A qs init i y* + *ALG'_det A qs init i x*
else 0))
unfolding *f* **by** *auto*
also have ... = *ALGxy_det A qs init x y*
unfolding *ALGxy_det_def* **by** *auto*
finally show *?thesis* **by** *simp*
qed

14.1.7 ALGxy

definition *ALGxy* **where**
ALGxy A qs init x y = ($\sum$ *i*∈{..<*length qs*} ∩ {*i*. (*qs*!*i* ∈ {*y*,*x*})}. *ALG'* *A qs init i y* + *ALG' A qs init i x*)

lemma *ALGxy_def2*:
ALGxy A qs init x y = ($\sum$ *i*∈{*i*. *i*<*length qs* ∧ (*qs*!*i* ∈ {*y*,*x*})}. *ALG' A* *qs init i y* + *ALG' A qs init i x*)
proof −
have *a*: {*i*. *i*<*length qs* ∧ (*qs*!*i* ∈ {*y*,*x*})} = {..<*length qs*} ∩ {*i*. (*qs*!*i* ∈ {*y*,*x*})} **by** *auto*
show *?thesis* **unfolding** *ALGxy_def a* **by** *simp*
qed
lemma *ALGxy_append*: *ALGxy A* (*rs*@[*r*]) *init x y* =
ALGxy A rs init x y + (*if* (*r* ∈ {*y*,*x*}) *then ALG' A* (*rs*@[*r*]) *init* (*length rs*) *y* + *ALG' A* (*rs*@[*r*]) *init* (*length rs*) *x else 0*)
proof −
have *ALGxy A* (*rs*@[*r*]) *init x y* = ($\sum$ *i*∈{..<(*Suc* (*length rs*))} ∩ {*i*. (*rs* @ [*r*]) ! *i* ∈ {*y*, *x*}}.
ALG' A (*rs* @ [*r*]) *init i y* +
ALG' A (*rs* @ [*r*]) *init i x*) **unfolding** *ALGxy_def* **by**(*simp*)
also have ... = ($\sum$ *i*∈{..<(*Suc* (*length rs*))}. (*if i*∈{*i*. (*rs* @ [*r*]) ! *i* ∈ {*y*, *x*}} *then*
ALG' A (*rs* @ [*r*]) *init i y* +
ALG' A (*rs* @ [*r*]) *init i x else 0*))
apply(*rule sum.inter_restrict*) **by** *simp*
also have ... = ($\sum$ *i*∈{..<*length rs*}. (*if i*∈{*i*. (*rs* @ [*r*]) ! *i* ∈ {*y*, *x*}} *then*
ALG' A (*rs* @ [*r*]) *init i y* +
ALG' A (*rs* @ [*r*]) *init i x else 0*)) + (*if length rs*∈{*i*. (*rs* @ [*r*]) ! *i*

$\in \{y, x\}\}$ *then*
ALG′ A (*rs* @ [*r*]) *init* (*length rs*) *y* +
ALG′ A (*rs* @ [*r*]) *init*(*length rs*) *x else 0*) **by** *simp*
also have ... = *ALGxy A rs init x y* + (*if r* $\in$ {*y*, *x*} *then*
ALG′ A (*rs* @ [*r*]) *init* (*length rs*) *y* +
ALG′ A (*rs* @ [*r*]) *init*(*length rs*) *x else 0*)
apply(*simp add*: *ALGxy_def sum.inter_restrict nth_append*)
unfolding *ALG′_def*
apply(*rule sum.cong*)
apply(*simp*) **by**(*auto simp*: *nth_append*)
finally show *?thesis* **.**
qed

lemma *ALGxy_wholerange*: *ALGxy A qs init x y*
= ($\sum$ *i*<(*length qs*). (*if qs* ! *i* $\in$ {*y*, *x*}
then ALG′ A qs init i y + *ALG′ A qs init i x*
else 0))
proof −
have *ALGxy A qs init x y*
= ($\sum$ *i*$\in$ {*i*. *i* < *length qs*} $\cap$ {*i*. *qs* ! *i* $\in$ {*y*, *x*}}.
ALG′ A qs init i y + *ALG′ A qs init i x*)
unfolding *ALGxy_def*
apply(*rule sum.cong*)
apply(*simp*) **apply**(*blast*)
by *simp*
also have ... = ($\sum$ *i*$\in${*i*. *i* < *length qs*}. *if i* $\in$ {*i*. *qs* ! *i* $\in$ {*y*, *x*}}
then ALG′ A qs init i y + *ALG′ A qs init i x*
else 0)
by(*rule sum.inter_restrict*) *simp*
also have ... = ($\sum$ *i*<(*length qs*). (*if qs* ! *i* $\in$ {*y*, *x*}
then ALG′ A qs init i y + *ALG′ A qs init i x*
else 0)) **apply**(*rule sum.cong*) **by**(*auto*)
finally show *?thesis* **.**
qed

14.2 Transformation to Blocking Cost

lemma *umformung*:
fixes *A* :: ((*′a*::*linorder*) *list*,*′is*,*′a*,(*nat* ∗ *nat list*)) *alg_on_rand*
assumes *no_paid*: $\bigwedge$*is s q*. $\forall$ ((*free*,*paid*),_) $\in$ (*snd A* (*s*,*is*) *q*). *paid*=[]
assumes *inlist*: *set qs* $\subseteq$ *set init*
assumes *dist*: *distinct init*
assumes $\bigwedge$*x*. *x* < *length qs* $\Longrightarrow$ *finite* (*set_pmf* (*config″ A qs init x*))
shows T_p*_on_rand A init qs* =

(∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ALGxy A qs init x y)
proof –
  have *config_dist*: ∀ n. ∀ xa ∈ set_pmf (config'' A qs init n). distinct (fst xa)
      using *dist config_rand_distinct* **by** *metis*

  have *E0*: T_p_on_rand A init qs =
      (∑ i∈{..<length qs}. T_p_on_rand_n A init qs i) **unfolding** *T_on_rand_as_sum* **by** *auto*
  also have ... =
  (∑ i<length qs. E (bind_pmf (config'' A qs init i)
                        (λs. bind_pmf (snd A s (qs ! i))
                            (λ(a, nis). return_pmf (real (∑ x∈set init. ALG x qs i s))))))
    apply(*rule sum.cong*)
      apply(*simp*)
      apply(*simp add*: *bind_return_pmf bind_assoc_pmf*)
      apply(*rule arg_cong*[**where** *f*=*E*])
        apply(*rule bind_pmf_cong*)
          apply(*simp*)
            apply(*rule bind_pmf_cong*)
              apply(*simp*)
              apply(*simp add*: *split_def*)
                apply(*subst* t_p_*sumofALGreal*)
                proof (*goal_cases*)
                  case *1*
                  then show *?case* **using** *config_dist* **by**(*metis*)
                next
                  case (*2 a b c*)
                    then show *?case* **using** *no_paid*[*of fst b snd b*] **by**(*auto simp add*: *split_def*)
                next
                  case (*3 a b c*)
                    with *config_rand_set* **have** *a*: *set* (*fst b*) = *set init* **by** *metis*
                  with *inlist* **have** *set qs* ⊆ *set* (*fst b*) **by** *auto*
                  with *3* **show** *?case* **by** *auto*
                next
                  case (*4 a b c*)
                    with *config_rand_set* **have** *a*: *set* (*fst b*) = *set init* **by** *metis*
                  then show *?case* **by**(*simp*)
                qed

also have ... = ($\sum$ *i*<*length qs*.
E (*map_pmf* (λ(*is*, *s*). (*real* ($\sum$ *x*∈*set init*. *ALG x qs i* (*is*,*s*))))
(*config''* *A qs init i*)))
apply(*simp only*: *map_pmf_def split_def*) **by** *simp*
also have *E1*: ... = ($\sum$ *i*<*length qs*. ($\sum$ *x*∈*set init*. *ALG' A qs init i x*))
apply(*rule sum.cong*)
apply(*simp*)
apply(*simp add*: *split_def ALG'_def*)
apply(*rule E_linear_sum_allg*)
by(*rule assms*(*4*))
also have *E2*: ... = ($\sum$ *x*∈*set init*.
($\sum$ *i*<*length qs*. *ALG' A qs init i x*))
by(*rule sum.swap*)
also have *E3*: ... = ($\sum$ *x*∈*set init*.
($\sum$ *y*∈*set init*.
($\sum$ *i*∈{*i*. *i*<*length qs* ∧ *qs*!*i*=*y*}. *ALG' A qs init i x*)))
proof (*rule sum.cong*, *goal_cases*)
case (*2 x*)
have ($\sum$ *i*<*length qs*. *ALG' A qs init i x*)
= *sum* (%*i*. *ALG' A qs init i x*) {*i*. *i*<*length qs*}
by (*metis lessThan_def*)
also have ... = *sum* (%*i*. *ALG' A qs init i x*)
($\bigcup$ *y*∈{*y*. *y* ∈ *set init*}. {*i*. *i* < *length qs* ∧ *qs* ! *i* = *y*})
apply(*rule sum.cong*)
apply(*auto*)
using *inlist* **by** *auto*
also have ... = *sum* (%*t*. *sum* (%*i*. *ALG' A qs init i x*) {*i*.
i<*length qs* ∧ *qs* ! *i* = *t*}) {*y*. *y*∈ *set init*}
apply(*rule sum.UNION_disjoint*)
apply(*simp_all*) **by** *force*
also have ... = ($\sum$ *y*∈*set init*. $\sum$ *i* | *i* < *length qs* ∧ *qs* ! *i* = *y*.
ALG' A qs init i x) **by** *auto*
finally show *?case* **.**
qed (*simp*)

also have ... = ($\sum$ (*x*,*y*)∈ (*set init* × *set init*).
($\sum$ *i*∈{*i*. *i*<*length qs* ∧ *qs*!*i*=*y*}. *ALG' A qs init i x*))
by (*rule sum.cartesian_product*)
also have ... = ($\sum$ (*x*,*y*)∈ {(*x*,*y*). *x*∈*set init* ∧ *y*∈ *set init*}.
($\sum$ *i*∈{*i*. *i*<*length qs* ∧ *qs*!*i*=*y*}. *ALG' A qs init i x*))
by *simp*

```
    also have E4: ... = (∑(x,y)∈{(x,y). x∈set init ∧ y∈ set init ∧ x≠y}.
          (∑i∈{i. i<length qs ∧ qs!i=y}. ALG' A qs init i x)) (is (∑(x,y)∈
?L. ?f x y) = (∑(x,y)∈ ?R. ?f x y))
      proof -
        let ?M = {(x,y). x∈set init ∧ y∈ set init ∧ x=y}
        have A: ?L = ?R ∪ ?M by auto
        have B: {} = ?R ∩ ?M by auto

        have (∑(x,y)∈ ?L. ?f x y) = (∑(x,y)∈ ?R ∪ ?M. ?f x y)
          by(simp only: A)
        also have ... = (∑(x,y)∈ ?R. ?f x y) + (∑(x,y)∈ ?M. ?f x y)
            apply(rule sum.union_disjoint)
              apply(rule finite_subset[where B=set init × set init])
                apply(auto)
              apply(rule finite_subset[where B=set init × set init])
                by(auto)
        also have (∑(x,y)∈ ?M. ?f x y) = 0
          apply(rule sum.neutral)
            by (auto simp add: ALG'_refl)
        finally show ?thesis by simp
      qed

    also have ... = (∑(x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
            (∑i∈{i. i<length qs ∧ qs!i=y}. ALG' A qs init i x)
          + (∑i∈{i. i<length qs ∧ qs!i=x}. ALG' A qs init i y) )
            (is (∑(x,y)∈ ?L. ?f x y) = (∑(x,y)∈ ?R. ?f x y +  ?f y x))
              proof -
                let ?R' = {(x,y). x ∈ set init ∧ y∈set init ∧ y<x}
                have A: ?L = ?R ∪ ?R' by auto
                have {} = ?R ∩ ?R' by auto
                have C: ?R' = (%(x,y). (y, x)) ` ?R by auto

                have D: (∑(x,y)∈ ?R'. ?f x y) = (∑(x,y)∈ ?R. ?f y x)
                proof -
                  have (∑(x,y)∈ ?R'. ?f x y) = (∑(x,y)∈ (%(x,y). (y, x)) `
?R. ?f x y)
                      by(simp only: C)
                  also have (∑z∈ (%(x,y). (y, x)) ` ?R. (%(x,y). ?f x y) z)
= (∑z∈?R. ((%(x,y). ?f x y) ∘ (%(x,y). (y, x))) z)
                    apply(rule sum.reindex)
                      by(fact swap_inj_on)
                  also have ... = (∑z∈?R. (%(x,y). ?f y x) z)
                    apply(rule sum.cong)
                      by(auto)
```

finally show *?thesis* **.**
qed

have $(\sum (x,y)\in\ ?L.\ ?f\ x\ y) = (\sum (x,y)\in\ ?R \cup\ ?R'.\ ?f\ x\ y)$
by(*simp only*: *A*)
also have ... $= (\sum (x,y)\in\ ?R.\ ?f\ x\ y) + (\sum (x,y)\in\ ?R'.\ ?f\ x$
$y)$
apply(*rule sum.union_disjoint*)
apply(*rule finite_subset*[**where** *B=set init* × *set init*])
apply(*auto*)
apply(*rule finite_subset*[**where** *B=set init* × *set init*])
by(*auto*)
also have ... $= (\sum (x,y)\in\ ?R.\ ?f\ x\ y) + (\sum (x,y)\in\ ?R.\ ?f\ y\ x)$
by(*simp only*: *D*)
also have ... $= (\sum (x,y)\in\ ?R.\ ?f\ x\ y +\ ?f\ y\ x)$
by(*simp add*: *split_def sum.distrib*[*symmetric*])
finally show *?thesis* **.**
qed

also have *E5*: ... $= (\sum (x,y)\in\{(x,y).\ x \in set\ init \wedge y\in set\ init \wedge x<y\}.$
$(\sum i\in\{i.\ i<length\ qs \wedge (qs!i=y \vee qs!i=x)\}.\ ALG'\ A\ qs\ init\ i\ y +$
$ALG'\ A\ qs\ init\ i\ x))$
apply(*rule sum.cong*)
apply(*simp*)
proof *goal_cases*
case (*1 x*)
then obtain *a b* **where** *x*: $x=(a,b)$ **and** *a*: $a \in set\ init\ b \in set\ init$
$a < b$ **by** *auto*
then have $a\neq b$ **by** *simp*
then have *disj*: $\{i.\ i < length\ qs \wedge qs\ !\ i = b\} \cap \{i.\ i < length\ qs$
$\wedge\ qs\ !\ i = a\} = \{\}$ **by** *auto*
have *unio*: $\{i.\ i < length\ qs \wedge (qs\ !\ i = b \vee qs\ !\ i = a)\}$
$= \{i.\ i < length\ qs \wedge qs\ !\ i = b\} \cup \{i.\ i < length\ qs \wedge qs\ !\ i =$
$a\}$ **by** *auto*
have $(\sum i\in\{i.\ i < length\ qs \wedge qs\ !\ i = b\} \cup$
$\{i.\ i < length\ qs \wedge qs\ !\ i = a\}.\ ALG'\ A\ qs\ init\ i\ b +$
$ALG'\ A\ qs\ init\ i\ a)$
$= (\sum i\in\{i.\ i < length\ qs \wedge qs\ !\ i = b\}.\ ALG'\ A\ qs\ init\ i\ b +$
$ALG'\ A\ qs\ init\ i\ a) + (\sum i\in$
$\{i.\ i < length\ qs \wedge qs\ !\ i = a\}.\ ALG'\ A\ qs\ init\ i\ b +$
$ALG'\ A\ qs\ init\ i\ a) - (\sum i\in\{i.\ i < length\ qs \wedge qs\ !\ i = b\} \cap$
$\{i.\ i < length\ qs \wedge qs\ !\ i = a\}.\ ALG'\ A\ qs\ init\ i\ b +$
$ALG'\ A\ qs\ init\ i\ a)$
apply(*rule sum_Un*)

by(*auto*)
also have ... = ($\sum$ *i*∈{*i*. *i* < *length qs* ∧ *qs* ! *i* = *b*}. *ALG′ A qs init i b* +
ALG′ A qs init i a) + ($\sum$ *i*∈
{*i*. *i* < *length qs* ∧ *qs* ! *i* = *a*}. *ALG′ A qs init i b* +
ALG′ A qs init i a) **using** *disj* **by** *auto*
also have ... = ($\sum$ *i*∈{*i*. *i* < *length qs* ∧ *qs* ! *i* = *b*}. *ALG′ A qs init i a*)
+ ($\sum$ *i*∈{*i*. *i* < *length qs* ∧ *qs* ! *i* = *a*}. *ALG′ A qs init i b*)
by (*auto simp*: *ALG′_refl*)
finally
show *?case* **unfolding** *x* **apply**(*simp add*: *split_def*)
unfolding *unio* **by** *simp*
qed
also have *E6*: ... = ($\sum$ (*x*,*y*)∈{(*x*,*y*). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*<*y*}.
ALGxy A qs init x y)
unfolding *ALGxy_def2* **by** *simp*
finally show *?thesis* **.**
qed

lemma *before_in_index1*:
fixes *l*
assumes *set l* = {*x*,*y*} **and** *length l* = *2* **and** *x*≠*y*
shows (*if* (*x* < *y in l*) *then 0 else 1*) = *index l x*
unfolding *before_in_def*
proof (*auto*, *goal_cases*)
case *1*
from *assms*(*1*) **have** *index l y* < *length l* **by** *simp*
with *assms*(*2*) *1*(*1*) **show** *index l x* = *0* **by** *auto*
next
case *2*
from *assms*(*1*) **have** *a*: *index l x* < *length l* **by** *simp*
from *assms*(*1*,*3*) **have** *index l y* ≠ *index l x* **by** *simp*
with *assms*(*2*) *2*(*1*) *a* **show** *Suc 0* = *index l x* **by** *simp*
qed (*simp add*: *assms*)

lemma *before_in_index2*:
fixes *l*
assumes *set l* = {*x*,*y*} **and** *length l* = *2* **and** *x*≠*y*
shows (*if* (*x* < *y in l*) *then 1 else 0*) = *index l y*
unfolding *before_in_def*
proof (*auto*, *goal_cases*)

case *2*
from *assms*(*1*,*3*) **have** *a*: *index l y* ≠ *index l x* **by** *simp*
from *assms*(*1*) **have** *index l x* < *length l* **by** *simp*
with *assms*(*2*) *a* *2*(*1*) **show** *index l y* = *0* **by** *auto*
next
case *1*
from *assms*(*1*) **have** *a*: *index l y* < *length l* **by** *simp*
from *assms*(*1*,*3*) **have** *index l y* ≠ *index l x* **by** *simp*
with *assms*(*2*) *1*(*1*) *a* **show** *Suc 0* = *index l y* **by** *simp*
qed (*simp add*: *assms*)

lemma *before_in_index*:
fixes *l*
assumes *set l* = {*x*,*y*} **and** *length l* = *2* **and** *x*≠*y*
shows (*x* < *y in l*) = (*index l x* = *0*)
unfolding *before_in_def*
proof (*safe*, *goal_cases*)
case *1*
from *assms*(*1*) **have** *index l y* < *length l* **by** *simp*
with *assms*(*2*) *1*(*1*) **show** *index l x* = *0* **by** *auto*
next
case *2*
from *assms*(*1*,*3*) **have** *index l y* ≠ *index l x* **by** *simp*
with *2*(*1*) **show** *index l x* < *index l y* **by** *simp*
qed (*simp add*: *assms*)

14.3 The pairwise property

definition *pairwise* **where**
pairwise A = (∀ *init*. *distinct init* ⟶ (∀ *qs*∈{*xs*. *set xs* ⊆ *set init*}. ∀ (*x*::('*a*::*linorder*),*y*)∈{(*x*,*y*). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*<*y*}. T_p*_on_rand A* (*Lxy init* {*x*,*y*}) (*Lxy qs* {*x*,*y*}) = *ALGxy A qs init x y*))

definition *Pbefore_in x y A qs init* = *map_pmf* (λ*p*. *x* < *y in fst p*) (*config_rand A init qs*)

lemma *T_on_n_no_paid*:
assumes
nopaid: ⋀*s n*. *map_pmf* (λ*x*. *snd* (*fst x*)) (*snd A s n*) = *return_pmf* []
shows *T_on_rand_n A init qs i* = *E* (*config''* *A qs init i* ≫= (λ*p*. *return_pmf* (*real*(*index* (*fst p*) (*qs* ! *i*)))))

proof −

have (*λs. snd A s* (*qs* ! *i*) ⤜
(*λ*(*a*, *is′*). *return_pmf* (*real* (t_p (*fst s*) (*qs* ! *i*) *a*))))
=
(*λs.* (*snd A s* (*qs* ! *i*) ⤜ (*λx. return_pmf* (*snd* (*fst x*))))
⤜ (*λp. return_pmf*
(*real* (*index* (*swaps p* (*fst s*)) (*qs* ! *i*)) +
real (*length p*))))
by(*simp add*: t_p*_def split_def bind_return_pmf bind_assoc_pmf*)
also
have ... = (*λp. return_pmf* (*real* (*index* (*fst p*) (*qs* ! *i*))))
using *nopaid*[*unfolded map_pmf_def*]
by(*simp add*: *split_def bind_return_pmf*)
finally
show *?thesis* **by** *simp*
qed

lemma *pairwise_property_lemma*:
assumes
relativeorder: (⋀*init qs. distinct init* ⟹ *qs* ∈ {*xs. set xs* ⊆ *set init*}
⟹ (⋀*x y.* (*x*,*y*)∈ {(*x*,*y*). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*≠*y*}
⟹ *x* ≠ *y*
⟹ *Pbefore_in x y A qs init* = *Pbefore_in x y A* (*Lxy qs* {*x*,*y*}) (*Lxy init* {*x*,*y*})
))
and *nopaid*: ⋀*xa r*. ∀ *z*∈ *set_pmf*(*snd A xa r*). *snd*(*fst z*) = []
shows *pairwise A*
unfolding *pairwise_def*
proof (*clarify*, *goal_cases*)
case (*1 init rs x y*)
then have *xny*: *x*≠*y* **by** *auto*

note *dinit=1*(*1*)
then have *dLyx*: *distinct* (*Lxy init* {*y*,*x*}) **by**(*rule Lxy_distinct*)
from *dinit* **have** *dLxy*: *distinct* (*Lxy init* {*x*,*y*}) **by**(*rule Lxy_distinct*)
have *setLxy*: *set* (*Lxy init* {*x*, *y*}) = {*x*,*y*} **apply**(*subst Lxy_set_filter*) **using** *1* **by** *auto*
have *setLyx*: *set* (*Lxy init* {*y*, *x*}) = {*x*,*y*} **apply**(*subst Lxy_set_filter*) **using** *1* **by** *auto*
have *lengthLyx*: *length* (*Lxy init* {*y*, *x*}) = *2* **using** *setLyx distinct_card*[*OF dLyx*] *xny* **by** *simp*
have *lengthLxy*: *length* (*Lxy init* {*x*, *y*}) = *2* **using** *setLxy distinct_card*[*OF dLxy*] *xny* **by** *simp*

have *aee*: {*x*,*y*} = {*y*,*x*} **by** *auto*

from *1*(*2*) **show** *?case*
proof(*induct rs rule*: *rev_induct*)
case (*snoc r rs*)

have *b*: *Pbefore_in x y A rs init* = *Pbefore_in x y A* (*Lxy rs* {*x*,*y*}) (*Lxy init* {*x*,*y*})
apply(*rule relativeorder*)
using *snoc 1 xny* **by**(*simp_all*)

show *?case* (**is** *?L* (*rs* @ [*r*]) = *?R* (*rs* @ [*r*]))
proof(*cases r*∈{*x*,*y*})
case *True*
note *xyrequest*=*this*
let *?expr* = *E* (*Partial_Cost_Model.config′_rand A*
(*fst A* (*Lxy init* {*x*, *y*}) ≫=
(*λis*. *return_pmf* (*Lxy init* {*x*, *y*}, *is*)))
(*Lxy rs* {*x*, *y*}) ≫=
(*λs*. *snd A s r* ≫=
(*λ*(*a*, *is′*).
return_pmf
(*real* (t_p (*fst s*) *r a*)))))
let *?expr2* = *ALG′ A* (*rs* @ [*r*]) *init* (*length rs*) *y* + *ALG′ A* (*rs* @ [*r*]) *init* (*length rs*) *x*

from *xyrequest* **have** *?L* (*rs* @ [*r*]) = *?L rs* + *?expr*
by(*simp add*: *Lxy_snoc T_on_rand′_append*)
also have . . . = *?L rs* + *?expr2*
proof(*cases r*=*x*)
case *True*
let *?projS* =*config′_rand A* (*fst A* (*Lxy init* {*x*, *y*}) ≫= (*λis*. *return_pmf* (*Lxy init* {*x*, *y*}, *is*))) (*Lxy rs* {*x*, *y*})
let *?S* = (*config′_rand A* (*fst A init* ≫= (*λis*. *return_pmf* (*init*, *is*))) *rs*)

have *?projS* ≫= (*λs*. *snd A s r*
≫= (*λ*(*a*, *is′*). *return_pmf* (*real* (t_p (*fst s*) *r a*))))
= *?projS* ≫= (*λs*. *return_pmf* (*real* (*index* (*fst s*) *r*)))
proof (*rule bind_pmf_cong*, *goal_cases*)
case (*2 z*)
have *snd A z r* ≫= (*λ*(*a*, *is′*). *return_pmf* (*real* (t_p (*fst*

z) r a))) = snd A z r ≫= (λx. return_pmf (real (index (fst z) r)))
apply(rule bind_pmf_cong)
apply(simp)
using nopaid[of z r] by(simp add: split_def t_p_def)
then show ?case by(simp add: bind_return_pmf)
qed simp
also have ... = map_pmf (%b. (if b then 0::real else 1)) (Pbefore_in x y A (Lxy rs {x,y}) (Lxy init {x,y}))
unfolding Pbefore_in_def map_pmf_def
apply(simp add: bind_return_pmf bind_assoc_pmf)
apply(rule bind_pmf_cong)
apply(simp add: aee)
proof goal_cases
case (1 z)
have (if x < y in fst z then 0 else 1) = (index (fst z) x)
apply(rule before_in_index1)
using 1 config_rand_set setLxy apply fast
using 1 config_rand_length lengthLxy apply metis

using xny by simp
with True show ?case
by(auto)
qed
also have ... = map_pmf (%b. (if b then 0::real else 1)) (Pbefore_in x y A rs init) by(simp add: b)

also have ... = map_pmf (λxa. real (if y < x in fst xa then 1 else 0)) ?S
apply(simp add: Pbefore_in_def map_pmf_comp)
proof (rule map_pmf_cong, goal_cases)
case (2 z)
then have set_z: set (fst z) = set init
using config_rand_set by fast
have (¬ x < y in fst z) = y < x in fst z
apply(subst not_before_in)
using set_z 1(3,4) xny by(simp_all)
then show ?case by simp
qed simp
finally have a: ?projS ≫= (λs. snd A s x
≫= (λ(a, is'). return_pmf (real (t_p (fst s) x a))))
= map_pmf (λxa. real (if y < x in fst xa then 1 else 0)) ?S
using True by simp
from True show ?thesis
apply(simp add: ALG'_refl nth_append)

unfolding *ALG′_def*
by(*simp add*: *a*)
next
case *False*
with *xyrequest* **have** *request*: *r*=*y* **by** *blast*

let *?projS* =*config′_rand A* (*fst A* (*Lxy init* {*x*, *y*}) ≫= (λ*is*. *return_pmf* (*Lxy init* {*x*, *y*}, *is*))) (*Lxy rs* {*x*, *y*})
let *?S* = (*config′_rand A* (*fst A init* ≫= (λ*is*. *return_pmf* (*init*, *is*))) *rs*)

have *?projS* ≫= (λ*s*. *snd A s r*
≫= (λ(*a*, *is′*). *return_pmf* (*real* (t_p (*fst s*) *r a*))))
= *?projS* ≫= (λ*s*. *return_pmf* (*real* (*index* (*fst s*) *r*)))
proof (*rule bind_pmf_cong*, *goal_cases*)
case (*2 z*)
have *snd A z r* ≫= (λ(*a*, *is′*). *return_pmf* (*real* (t_p (*fst z*) *r a*))) = *snd A z r* ≫= (λ*x*. *return_pmf* (*real* (*index* (*fst z*) *r*)))
apply(*rule bind_pmf_cong*)
apply(*simp*)
using *nopaid*[*of z r*] **by**(*simp add*: *split_def* t_p*_def*)
then show *?case* **by**(*simp add*: *bind_return_pmf*)
qed *simp*
also have ... = *map_pmf* (%*b*. (*if b then 1*::*real else 0*)) (*Pbefore_in x y A* (*Lxy rs* {*x*,*y*}) (*Lxy init* {*x*,*y*}))
unfolding *Pbefore_in_def map_pmf_def*
apply(*simp add*: *bind_return_pmf bind_assoc_pmf*)
apply(*rule bind_pmf_cong*)
apply(*simp add*: *aee*)
proof *goal_cases*
case (*1 z*)
have (*if x* < *y in fst z then 1 else 0*) = (*index* (*fst z*) *y*)
apply(*rule before_in_index2*)
using *1 config_rand_set setLxy* **apply** *fast*
using *1 config_rand_length lengthLxy* **apply** *metis*

using *xny* **by** *simp*
with *request* **show** *?case*
by(*auto*)
qed
also have ... = *map_pmf* (%*b*. (*if b then 1*::*real else 0*)) (*Pbefore_in x y A rs init*) **by**(*simp add*: *b*)

also have ... = *map_pmf* (*λxa. real* (*if x < y in fst xa then 1 else 0*)) *?S*
apply(*simp add*: *Pbefore_in_def map_pmf_comp*)
apply (*rule map_pmf_cong*) **by** *simp_all*
finally have *a*: *?projS* ⤜ (*λs. snd A s y*
⤜ (*λ*(*a*, *is′*). *return_pmf* (*real* (t_p (*fst s*) *y a*))))
= *map_pmf* (*λxa. real* (*if x < y in fst xa then 1 else 0*)) *?S*
using *request* **by** *simp*
from *request* **show** *?thesis*
apply(*simp add*: *ALG′_refl nth_append*)
unfolding *ALG′_def*
by(*simp add*: *a*)
qed
also have ... = *?R rs* + *?expr2* **using** *snoc* **by** *simp*
also from *True* **have** ... = *?R* (*rs*@[*r*])
apply(*subst ALGxy_append*) **by**(*auto*)
finally show *?thesis* **.**
next
case *False*
then have *?L* (*rs* @ [*r*]) = *?L rs* **apply**(*subst Lxy_snoc*) **by** *simp*
also have ... = *?R rs* **using** *snoc* **by**(*simp*)
also have ... = *?R* (*rs* @ [*r*])
apply(*subst ALGxy_append*) **using** *False* **by**(*simp*)
finally show *?thesis* **.**
qed
qed (*simp add*: *ALGxy_def*)
qed

lemma *umf_pair*: **assumes**
0: *pairwise A*
assumes *1*: ⋀*is s q*. ∀ ((*free*,*paid*),_) ∈ (*snd A* (*s*, *is*) *q*). *paid*=[]
assumes *2*: *set qs* ⊆ *set init*
assumes *3*: *distinct init*
assumes *4*: ⋀*x*. *x*<*length qs* ⟹ *finite* (*set_pmf* (*config″ A qs init x*))
shows T_p*_on_rand A init qs*
= (∑ (*x*,*y*)∈{(*x*, *y*). *x* ∈ *set init* ∧ *y* ∈ *set init* ∧ *x* < *y*}. T_p*_on_rand A* (*Lxy init* {*x*,*y*}) (*Lxy qs* {*x*,*y*}))
proof −
have T_p*_on_rand A init qs* = (∑ (*x*,*y*)∈{(*x*, *y*). *x* ∈ *set init* ∧ *y* ∈ *set init* ∧ *x* < *y*}. *ALGxy A qs init x y*)
by(*simp only*: *umformung*[*OF 1 2 3 4*])
also have ... = (∑ (*x*,*y*)∈{(*x*, *y*). *x* ∈ *set init* ∧ *y* ∈ *set init* ∧ *x* < *y*}. T_p*_on_rand A* (*Lxy init* {*x*,*y*}) (*Lxy qs* {*x*,*y*}))

```
    apply(rule sum.cong)
      apply(simp)
      using 0[unfolded pairwise_def] 2 3 by auto
  finally show ?thesis .
qed
```

14.4 List Factoring for OPT

```
fun ALG_P :: nat list ⇒ 'a ⇒ 'a ⇒ 'a list ⇒ nat where
  ALG_P [] x y xs = (0::nat)
| ALG_P (s#ss) x y xs = (if Suc s < length (swaps ss xs)
                        then (if ((swaps ss xs)!s=x ∧ (swaps ss xs)!(Suc s)=y)
∨ ((swaps ss xs)!s=y ∧ (swaps ss xs)!(Suc s)=x)
                              then 1
                              else 0)
                         else 0) + ALG_P ss x y xs


lemma ALG_P_erwischt_alle:
  assumes dinit: distinct init
  shows
  ∀ l< length sws. Suc (sws!l) < length init ⟹ length sws
        = (∑ (x,y)∈{(x,y). x ∈ set (init::('a::linorder) list) ∧ y∈set init ∧
x<y}. ALG_P sws x y init)
proof (induct sws)
  case (Cons s ss)
  then have isininit: Suc s < length init by auto
  from Cons have ∀ l<length ss. Suc (ss ! l)  < length init by auto
  note iH=Cons(1)[OF this]

  let ?expr = (λx y. (if Suc s < length (swaps ss init)
                        then (if ((swaps ss init)!s=x ∧ (swaps ss init)!(Suc
s)=y) ∨ ((swaps ss init)!s=y ∧ (swaps ss init)!(Suc s)=x)
                              then 1::nat
                              else 0)
                         else 0))

  let ?expr2 = (λx y. (if ((swaps ss init)!s=x ∧ (swaps ss init)!(Suc s)=y)
∨ ((swaps ss init)!s=y ∧ (swaps ss init)!(Suc s)=x)
                              then 1
                              else 0))

  let ?expr3 = (%x y.  ((swaps ss init)!s=x ∧ (swaps ss init)!(Suc s)=y)
                ∨ ((swaps ss init)!s=y ∧ (swaps ss init)!(Suc s)=x))
```

```
let ?co' = swaps ss init

from dinit have dco: distinct ?co' by auto

let ?expr4 = (λz. (if z∈{(x,y). ?expr3 x y}
                    then 1
                    else 0))

have scoinit: set ?co' = set init by auto
from isininit have isT: Suc s < length ?co' by auto
then have isT2: Suc s < length init by auto
then have isT3: s < length init by auto
then have isT6: s < length ?co' by auto
from isT2 have isT7: Suc s < length ?co' by auto
from isT6 have a: ?co'!s ∈ set ?co' by (rule nth_mem)
then have a: ?co'!s ∈ set init by auto
from isT7 have ?co'! (Suc s) ∈ set ?co' by (rule nth_mem)
then have b: ?co'!(Suc s) ∈ set init by auto

have {(x,y). x ∈ set init ∧ y∈set init ∧ x<y}
                ∩ {(x,y). ?expr3 x y}
  = {(x,y). x ∈ set init ∧ y∈set init ∧ x<y
                   ∧ (?co'!s=x ∧ ?co'!(Suc s)=y
                      ∨ ?co'!s=y ∧ ?co'!(Suc s)=x)} by auto
also have ... = {(x,y). x ∈ set init ∧ y∈set init ∧ x<y
                   ∧ ?co'!s=x ∧ ?co'!(Suc s)=y }
                ∪
           {(x,y). x ∈ set init ∧ y∈set init ∧ x<y
                   ∧  ?co'!s=y ∧ ?co'!(Suc s)=x} by auto
also have ... = {(x,y). x<y ∧ ?co'!s=x ∧ ?co'!(Suc s)=y}
                ∪
           {(x,y). x<y ∧ ?co'!s=y ∧ ?co'!(Suc s)=x}
        using a b by(auto)
finally have c1: {(x,y). x ∈ set init ∧ y∈set init ∧ x<y} ∩ {(x,y). ?expr3
x y}
   = {(x,y). x<y ∧ ?co'!s=x ∧ ?co'!(Suc s)=y}
                ∪
           {(x,y). x<y ∧ ?co'!s=y ∧ ?co'!(Suc s)=x} .

have c2: card ({(x,y). x<y ∧ ?co'!s=x ∧ ?co'!(Suc s)=y}
                ∪
           {(x,y). x<y ∧ ?co'!s=y ∧ ?co'!(Suc s)=x}) = 1 (is card (?A
∪ ?B) = 1)
proof (cases ?co'!s<?co'!(Suc s))
```

```
    case True
    then have a: ?A = { (?co'!s, ?co'!(Suc s)) }
         and b: ?B = {}  by auto
    have c: ?A ∪ ?B = { (?co'!s, ?co'!(Suc s)) } apply(simp only: a b) by
simp
    have card (?A ∪ ?B) = 1 unfolding c by auto
    then show ?thesis .
  next
    case False
    then have a: ?A = {} by auto
    have b: ?B = { (?co'!(Suc s), ?co'!s) }
    proof -
     from dco distinct_conv_nth[of ?co']
     have swaps ss init ! s ≠ swaps ss init ! (Suc s)
      using isT2 isT3 by simp
     with False show ?thesis by auto
    qed

    have c: ?A ∪ ?B = { (?co'!(Suc s), ?co'!s) } apply(simp only: a b) by
simp
    have card (?A ∪ ?B) = 1 unfolding c by auto
    then show ?thesis .
  qed



  have yeah: (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ?expr x y)
= (1::nat)
  proof -
    have (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ?expr x y)
       = (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ?expr2 x y)
         using isT by auto
    also have ... = (∑ z∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ?expr2
(fst z) (snd z))
       by(simp add: split_def)
    also have ... = (∑ z∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ?expr4
z)
       by(simp add: split_def)
    also have ... = (∑ z∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}
                    ∩{(x,y). ?expr3 x y} . 1)
       apply(rule sum.inter_restrict[symmetric])
             apply(rule finite_subset[where B=set init × set init])
               by(auto)
    also have ... = card ({(x,y). x ∈ set init ∧ y∈set init ∧ x<y}
```

∩ {(x,y). ?expr3 x y}) **by** *auto*
also have ... = *card* ({(x,y). x<y ∧ ?co'!s=x ∧ ?co'!(Suc s)=y}
∪
{(x,y). x<y ∧ ?co'!s=y ∧ ?co'!(Suc s)=x}) **by**(*simp only*: *c1*)
also have ... = (*1*::*nat*) **using** *c2* **by** *auto*
finally show *?thesis* **.**
qed

have *length* (*s* # *ss*) = *1* + *length ss*
by *auto*
also have ... = *1* + (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ALG_P ss x y init)
using *iH* **by** *auto*
also have ... = (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ?expr x y)
+ (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ALG_P ss x y init)
by(*simp only*: *yeah*)
also have ... = (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ?expr x y + ALG_P ss x y init)
(**is** *?A* + *?B* = *?C*)
by (*simp add*: *sum.distrib split_def*)
also have ... = (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ALG_P (s#ss) x y init)
by *auto*
finally show *?case* **.**
qed (*simp*)

lemma t_p*_sumofALGALGP*:
assumes *distinct s* (*qs*!*i*)∈*set s*
and ∀ l< *length* (*snd a*). *Suc* ((*snd a*)!*l*) < *length s*
shows t_p *s* (*qs*!*i*) *a* = (∑ e∈set s. ALG e qs i (swaps (snd a) s,()))
+ (∑ (x,y)∈{(x::('a::linorder),y). x ∈ set s ∧ y∈set s ∧ x<y}. ALG_P (snd a) x y s)
proof −

have *pe*: *length* (*snd a*)
= (∑ (x,y)∈{(x,y). x ∈ set s ∧ y∈set s ∧ x<y}. ALG_P (snd a) x y s)
apply(*rule ALG_P_erwischt_alle*)

by(*fact*)+

have *ac*: *index* (*swaps* (*snd a*) *s*) (*qs* ! *i*) = ($\sum$ *e*∈*set s*. *ALG e qs i* (*swaps* (*snd a*) *s*,()))
proof −
have *index* (*swaps* (*snd a*) *s*) (*qs* ! *i*)
= ($\sum$ *e*∈*set* (*swaps* (*snd a*) *s*). *if e* < (*qs* ! *i*) *in* (*swaps* (*snd a*) *s*) *then 1 else 0*)
apply(*rule index_sum*)
using *assms* **by**(*simp_all*)
also have ... = ($\sum$ *e*∈*set s*. *ALG e qs i* (*swaps* (*snd a*) *s*,())) **by** *auto*
finally show *?thesis* **.**
qed

show *?thesis*
unfolding t_p*_def* **apply** (*simp add*: *split_def*)
unfolding *ac pe* **by** (*simp add*: *split_def*)
qed

definition *ALG_P′ Strat qs init i x y* = *ALG_P* (*snd* (*Strat*!*i*)) *x y* (*steps′ init qs Strat i*)

lemma *ALG_P′_rest*: *n* < *length qs* ⟹ *n* < *length Strat* ⟹
ALG_P′ Strat (*take n qs* @ [*qs* ! *n*]) *init n x y* =
ALG_P′ (*take n Strat* @ [*Strat* ! *n*]) (*take n qs* @ [*qs* ! *n*]) *init n x y*
proof −
assume *qs*: *n* < *length qs*
assume *S*: *n* < *length Strat*

then have *lS*: *length* (*take n Strat*) = *n* **by** *auto*
have (*take n Strat* @ [*Strat* ! *n*]) ! *n* =
(*take n Strat* @ (*Strat* ! *n*) # []) ! *length* (*take n Strat*) **using** *lS* **by** *auto*
also have ... = *Strat* ! *n* **by**(*rule nth_append_length*)
finally have *tt*: (*take n Strat* @ [*Strat* ! *n*]) ! *n* = *Strat* ! *n* **.**

obtain *rest* **where** *rest*: *Strat* = (*take n Strat* @ [*Strat* ! *n*] @ *rest*)
using *S* **apply**(*auto*) **using** *id_take_nth_drop* **by** *blast*

have *steps′ init* (*take n qs* @ [*qs* ! *n*])

(*take n Strat* @ [*Strat* ! *n*]) *n*
= *steps′ init* (*take n qs*)
(*take n Strat*) *n*
apply(*rule steps′_rests*[*symmetric*])
using *S qs* **by** *auto*
also have . . . =
steps′ init (*take n qs* @ [*qs* ! *n*])
(*take n Strat* @ ([*Strat* ! *n*] @ *rest*)) *n*
apply(*rule steps′_rests*)
using *S qs* **by** *auto*
finally show *?thesis* **unfolding** *ALG_P′_def tt* **using** *rest* **by** *auto*
qed

lemma *ALG_P′_rest2*: *n* < *length qs* ⟹ *n* < *length Strat* ⟹
ALG_P′ Strat qs init n x y =
ALG_P′ (*Strat*@*r1*) (*qs*@*r2*) *init n x y*
proof −
assume *qs*: *n* < *length qs*
assume *S*: *n* < *length Strat*

have *tt*: *Strat* ! *n* = (*Strat* @ *r1*) ! *n*
using *S* **by** (*simp add*: *nth_append*)

have *steps′ init* (*take n qs*) (*take n Strat*) *n* = *steps′ init* ((*take n qs*) @ *drop n qs*) ((*take n Strat*) @ (*drop n Strat*)) *n*
apply(*rule steps′_rests*)
using *S qs* **by** *auto*
then have *A*: *steps′ init* (*take n qs*) (*take n Strat*) *n* = *steps′ init qs Strat n* **by** *auto*
have *steps′ init* (*take n qs*) (*take n Strat*) *n* = *steps′ init* ((*take n qs*) @ ((*drop n qs*)@*r2*)) ((*take n Strat*) @((*drop n Strat*)@*r1*)) *n*
apply(*rule steps′_rests*)
using *S qs* **by** *auto*
then have *B*: *steps′ init* (*take n qs*) (*take n Strat*) *n* = *steps′ init* (*qs*@*r2*) (*Strat*@*r1*) *n*
by (*metis append_assoc List.append_take_drop_id*)
from *A B* **have** *C*: *steps′ init qs Strat n* = *steps′ init* (*qs*@*r2*) (*Strat*@*r1*) *n* **by** *auto*
show *?thesis* **unfolding** *ALG_P′_def tt* **using** *C* **by** *auto*

qed

definition *ALG_Pxy* **where**
ALG_Pxy Strat qs init x y = ($\sum$ *i*<*length qs*. *ALG_P′ Strat qs init i x y*)

lemma *wegdamit*: *length A* < *length Strat* $\Longrightarrow$ *b* $\notin$ {*x*,*y*} $\Longrightarrow$ *ALGxy_det Strat* (*A* @ [*b*]) *init x y*
= *ALGxy_det Strat A init x y*
proof −
assume *bn*: *b* $\notin$ {*x*,*y*}
have (*A* @ [*b*]) ! (*length A*) = *b* **by** *auto*
assume *l*: *length A* < *length Strat*

term %*i*. *ALG′_det Strat* (*A* @ [*b*]) *init i y*

have *e*: $\bigwedge$*i*. *i*<*length A* $\Longrightarrow$ (*A* @ [*b*]) ! *i* = *A* ! *i* **by**(*auto simp*: *nth_append*)
have ($\sum$ *i*$\in$ {..< *length* (*A* @ [*b*])}.
if (*A* @ [*b*]) ! *i* $\in$ {*y*, *x*}
then ALG′_det Strat (*A* @ [*b*]) *init i y* +
ALG′_det Strat (*A* @ [*b*]) *init i x*
else 0) = ($\sum$ *i*$\in$ {..< *Suc*(*length* (*A*))}.
if (*A* @ [*b*]) ! *i* $\in$ {*y*, *x*}
then ALG′_det Strat (*A* @ [*b*]) *init i y* +
ALG′_det Strat (*A* @ [*b*]) *init i x*
else 0) **by** *auto*
also have ... = ($\sum$ *i*$\in$ {..< (*length* (*A*))}.
if (*A* @ [*b*]) ! *i* $\in$ {*y*, *x*}
then ALG′_det Strat (*A* @ [*b*]) *init i y* +
ALG′_det Strat (*A* @ [*b*]) *init i x*
else 0) + (*if* (*A* @ [*b*]) ! (*length A*) $\in$ {*y*, *x*}
then ALG′_det Strat (*A* @ [*b*]) *init* (*length A*) *y* +
ALG′_det Strat (*A* @ [*b*]) *init* (*length A*) *x*
else 0) **by** *simp*
also have ... = ($\sum$ *i*$\in$ {..< (*length* (*A*))}.
if (*A* @ [*b*]) ! *i* $\in$ {*y*, *x*}
then ALG′_det Strat (*A* @ [*b*]) *init i y* +
ALG′_det Strat (*A* @ [*b*]) *init i x*
else 0) **using** *bn* **by** *auto*
also have ... = ($\sum$ *i*$\in$ {..< (*length* (*A*))}.
if A ! *i* $\in$ {*y*, *x*}
then ALG′_det Strat A init i y +
ALG′_det Strat A init i x
else 0)
apply(*rule sum.cong*)

```
          apply(simp)
          using l ALG'_det_append[where qs=A] e by(simp)
    finally show ?thesis unfolding ALGxy_det_def by simp
qed

lemma ALG_P_split: length qs < length Strat ⟹ ALG_Pxy Strat (qs@[q])
init x y = ALG_Pxy Strat qs init x y
          + ALG_P' Strat (qs@[q]) init (length qs) x y
unfolding ALG_Pxy_def apply(auto)
  apply(rule sum.cong)
    apply(simp)
    using ALG_P'_rest2[symmetric, of _ qs Strat [] [q]] by(simp)


lemma swap0in2: assumes set l = {x,y} x≠y length l = 2 dist_perm l l
  shows
    x < y in (swap 0) l = (~ x < y in l)
proof (cases x < y in l)
  case True
  then have a: index l x < index l y unfolding before_in_def by simp
  from assms(1) have drin: x∈set l y∈set l by auto
  from assms(1,3) have b: index l y < 2 by simp
  from a b have k: index l x = 0 index l y = 1 by auto

  have g: x = l ! 0 y = l ! 1
    using k nth_index assms(1) by force+

      have x < y in swap 0 l
      = (x < y in l ∧ ¬ (x = l ! 0 ∧ y = l ! Suc 0)
          ∨ x = l ! Suc 0 ∧ y = l ! 0)
          apply(rule before_in_swap)
            apply(fact assms(4))
            using assms(3) by simp
  also have ... = (¬ (x = l ! 0 ∧ y = l ! Suc 0)
          ∨ x = l ! Suc 0 ∧ y = l ! 0) using True by simp
  also have ... = False using g assms(2) by auto
  finally have ~ x < y in (swap 0) l by simp
  then show ?thesis using True by auto
next
  case False
  from assms(1,2) have index l y ≠ index l x by simp
  with False assms(1,2) have a: index l y < index l x
    by (metis before_in_def insert_iff linorder_neqE_nat)
  from assms(1) have drin: x∈set l y∈set l by auto
```

from *assms*(*1*,*3*) **have** *b*: *index l x* < *2* **by** *simp*
from *a b* **have** *k*: *index l x* = *1 index l y* = *0* **by** *auto*
then have *g*: *x* = *l* ! *1 y* = *l* ! *0*
using *k nth_index assms*(*1*) **by** *force*+
have *x* < *y in swap 0 l*
= (*x* < *y in l* ∧ ¬ (*x* = *l* ! *0* ∧ *y* = *l* ! *Suc 0*)
∨ *x* = *l* ! *Suc 0* ∧ *y* = *l* ! *0*)
apply(*rule before_in_swap*)
apply(*fact assms*(*4*))
using *assms*(*3*) **by** *simp*
also have ... = (*x* = *l* ! *Suc 0* ∧ *y* = *l* ! *0*) **using** *False* **by** *simp*
also have ... = *True* **using** *g* **by** *auto*
finally have *x* < *y in* (*swap 0*) *l* **by** *simp*
then show *?thesis* **using** *False* **by** *auto*
qed

lemma *before_in_swap2*:
dist_perm xs ys ⟹ *Suc n* < *size xs* ⟹ *x*≠*y* ⟹
x < *y in* (*swap n xs*) ⟷
(~ *x* < *y in xs* ∧ (*y* = *xs*!*n* ∧ *x* = *xs*!*Suc n*)
∨ *x* < *y in xs* ∧ ~(*y* = *xs*!*Suc n* ∧ *x* = *xs*!*n*))
apply(*simp add*:*before_in_def index_swap_distinct*)
by (*metis Suc_lessD Suc_lessI index_nth_id less_Suc_eq nth_mem yes*)

lemma *projected_paid_same_effect*:
assumes
d: *dist_perm s1 s1*
and *ee*: *x*≠*y*
and *f*: *set s2* = {*x*, *y*}
and *g*: *length s2* = *2*
and *h*: *dist_perm s2 s2*
shows *x* < *y in s1* = *x* < *y in s2* ⟹
x < *y in swaps acs s1* = *x* < *y in* (*swap 0* ^^ *ALG_P acs x y s1*) *s2*
proof (*induct acs*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons s ss*)
from *d* **have** *dd*: *dist_perm* (*swaps ss s1*) (*swaps ss s1*) **by** *simp*
from *f* **have** *ff*: *set* ((*swap 0* ^^ *ALG_P ss x y s1*) *s2*) = {*x*, *y*} **by** (*metis foldr_replicate swaps_inv*)
from *g* **have** *gg*: *length* ((*swap 0* ^^ *ALG_P ss x y s1*) *s2*) = *2* **by** (*metis*

foldr_replicate swaps_inv)
  from *h* **have** *hh*: *dist_perm* ((*swap 0* ^^ *ALG_P ss x y s1*) *s2*) ((*swap 0* ^^ *ALG_P ss x y s1*) *s2*) **by** (*metis foldr_replicate swaps_inv*)
  show *?case* (**is** *?LHS* = *?RHS*)
  proof (*cases Suc s* < *length* (*swaps ss s1*) ∧ (((*swaps ss s1*)!*s*=*x* ∧ (*swaps ss s1*)!(*Suc s*)=*y*) ∨ ((*swaps ss s1*)!*s*=*y* ∧ (*swaps ss s1*)!(*Suc s*)=*x*)))
    case *True*
    from *True* **have** *1*: *Suc s* < *length* (*swaps ss s1*)
        and *2*: (*swaps ss s1* ! *s* = *x* ∧ *swaps ss s1* ! *Suc s* = *y*
          ∨ *swaps ss s1* ! *s* = *y* ∧ *swaps ss s1* ! *Suc s* = *x*) **by** *auto*
    from *True* **have** *ALG_P* (*s* # *ss*) *x y s1* = 1 + *ALG_P ss x y s1* **by** *auto*
    then have *?RHS* = *x* < *y in* (*swap 0*) ((*swap 0* ^^ *ALG_P ss x y s1*) *s2*)
      by *auto*
    also have ... = (~ *x* < *y in* ((*swap 0* ^^ *ALG_P ss x y s1*) *s2*))
      apply(*rule swap0in2*)
        by(*fact*)+
    also have ... = (~ *x* < *y in swaps ss s1*)
      using *Cons* **by** *auto*
    also have ... = *x* < *y in* (*swap s*) (*swaps ss s1*)
      using *1 2 before_in_swap*
      by (*metis Suc_lessD before_id dd lessI no_before_inI*)
    also have ... = *?LHS* **by** *auto*
    finally show *?thesis* **by** *simp*
  next
    case *False*
    note *F*=*this*
    then have *ALG_P* (*s* # *ss*) *x y s1* = *ALG_P ss x y s1* **by** *auto*
    then have *?RHS* = *x* < *y in* ((*swap 0* ^^ *ALG_P ss x y s1*) *s2*)
      by *auto*
    also have ... = *x* < *y in swaps ss s1*
      using *Cons* **by** *auto*
    also have ... = *x* < *y in* (*swap s*) (*swaps ss s1*)
    proof (*cases Suc s* < *length* (*swaps ss s1*))
      case *True*
      with *F* **have** *g*: *swaps ss s1* ! *s* ≠ *x* ∨
         swaps ss s1 ! *Suc s* ≠ *y* **and**
        h: *swaps ss s1* ! *s* ≠ *y* ∨
         swaps ss s1 ! *Suc s* ≠ *x* **by** *auto*
         show *?thesis*
          unfolding *before_in_swap*[*OF dd True*, *of x y*] **apply**(*simp*)
            using *g h* **by** *auto*
    next

    case *False*
    then show *?thesis* **unfolding** *swap_def* **by**(*simp*)
  qed
  also have ... = *?LHS* **by** *auto*
  finally show *?thesis* **by** *simp*
 qed
qed

lemma *steps_steps'*:
 length qs = length as ⟹ steps s qs as = steps' s qs as (length as)
by (*induct qs as arbitrary*: *s rule*: *list_induct2*) (*auto*)

lemma *T1_7'*: T_p *init qs Strat* = T_p*_opt init qs ⟹ length Strat = length qs*
    ⟹ n≤length qs ⟹
    x≠(y::('a::linorder)) ⟹
    x∈ set init ⟹ y ∈ set init ⟹ distinct init ⟹
    set qs ⊆ set init ⟹
    (∃ Strat2 sws.
     ~~T_p_opt (Lxy init {x,y}) (Lxy (take n qs) {x,y}) ≤ T_p (Lxy init {x,y}) (Lxy (take n qs) {x,y}) Strat2 ∧~~ *length Strat2 = length (Lxy (take n qs) {x,y})*
        ∧   (x < y in (steps' init (take n qs) (take n Strat) n))
          = (x < y in (swaps sws (steps' (Lxy init {x,y}) (Lxy (take n qs) {x,y}) Strat2 (length Strat2))))
        ∧ T_p *(Lxy init {x,y}) (Lxy (take n qs) {x,y}) Strat2 + length sws =*
       ALGxy_det Strat (take n qs) init x y + ALG_Pxy Strat (take n qs) init x y)
proof(*induct n*)
 case (*Suc n*)
 from *Suc*(*3*,*4*) **have** *ns*: *n < length qs* **by** *simp*
 then have *n*: *n ≤ length qs* **by** *simp*
 from *Suc*(*1*)[*OF Suc*(*2*) *Suc*(*3*) *n Suc*(*5*) *Suc*(*6*) *Suc*(*7*) *Suc*(*8*) *Suc*(*9*)] **obtain** *Strat2 sws* **where**
  len: *length Strat2 = length (Lxy (take n qs) {x, y})*
   and *iff*:
    x < y in steps' init (take n qs) (take n Strat) n
      =
     x < y in swaps sws (steps' (Lxy init {x,y}) (Lxy (take n qs) {x, y}) Strat2 (length Strat2))

and *T_Strat2*: T_p (*Lxy init* {*x*,*y*}) (*Lxy* (*take n qs*) {*x*, *y*}) *Strat2* + *length sws* =
ALGxy_det Strat (*take n qs*) *init x y* +
ALG_Pxy Strat (*take n qs*) *init x y* **by** (*auto*)

from *Suc*(*3*−*4*) **have** *nStrat*: *n* < *length Strat* **by** *auto*
from *take_Suc_conv_app_nth*[*OF this*] **have** *tak2*: *take* (*Suc n*) *Strat* = *take n Strat* @ [*Strat* ! *n*] **by** *auto*

from *take_Suc_conv_app_nth*[*OF ns*] **have** *tak*: *take* (*Suc n*) *qs* = *take n qs* @ [*qs* ! *n*] **by** *auto*

have *aS*: *length* (*take n Strat*) = *n* **using** *Suc*(*3*,*4*) **by** *auto*
have *aQ*: *length* (*take n qs*) = *n* **using** *Suc*(*4*) **by** *auto*
from *aS aQ* **have** *qQS*: *length* (*take n qs*) = *length* (*take n Strat*) **by** *auto*

have *xyininit*: *x*∈ *set init y* : *set init* **by** *fact*+
then have *xysubs*: {*x*,*y*} ⊆ *set init* **by** *auto*
have *dI*: *distinct init* **by** *fact*
have *set qs* ⊆ *set init* **by** *fact*
then have *qsnset*: *qs* ! *n* ∈ *set init* **using** *ns* **by** *auto*

from *xyininit* **have** *ahjer*: *set* (*Lxy init* {*x*, *y*}) = {*x*,*y*}
using *xysubs* **by** (*simp add*: *Lxy_set_filter*)
with *Suc*(*5*) **have** *ah*: *card* (*set* (*Lxy init* {*x*, *y*})) = *2* **by** *simp*
have *ahjer3*: *distinct* (*Lxy init* {*x*,*y*})
apply(*rule Lxy_distinct*) **by** *fact*
from *ah* **have** *ahjer2*: *length* (*Lxy init* {*x*,*y*}) = *2*
using *distinct_card*[*OF ahjer3*] **by** *simp*

show *?case*
proof (*cases qs* ! *n* ∈ {*x*,*y*})
case *False*
with *tak* **have** *nixzutun*: *Lxy* (*take* (*Suc n*) *qs*) {*x*,*y*} = *Lxy* (*take n qs*) {*x*,*y*}
unfolding *Lxy_def* **by** *simp*
let *?m*=*ALG_P′* (*take n Strat* @ [*Strat* ! *n*]) (*take n qs* @ [*qs* ! *n*]) *init n x y*
let *?L*=*replicate ?m 0* @ *sws*

{
fix *xs*::(′*a*::*linorder*) *list*
fix *m*::*nat*

fix *q*::′*a*
assume $q \notin \{x,y\}$
then have *5*: $y \neq q$ **by** *auto*
assume *1*: $q \in$ *set xs*
assume *2*: *distinct xs*
assume *3*: $x \in$ *set xs*
assume *4*: $y \in$ *set xs*
have (*x* < *y in xs*) = (*x* < *y in* (*mtf2 m q xs*))
by (*metis 1 2 3 4* ‹$q \notin \{x, y\}$› *insertCI not_before_in set_mtf2 swapped_by_mtf2*)
} **note** *f*=*this*

have (*x* < *y in steps′ init* (*take* (*Suc n*) *qs*) (*take* (*Suc n*) *Strat*) (*Suc n*))
= (*x* < *y in mtf2* (*fst* (*Strat* ! *n*)) (*qs* ! *n*)
(*swaps* (*snd* (*Strat* ! *n*)) (*steps′ init* (*take n qs*) (*take n Strat*) *n*)))
unfolding *tak2 tak* **apply**(*simp only*: *steps′_append*[*OF qQS aQ*])
by (*simp add*: *step_def split_def*)
also have ... = (*x* < *y in* (*swaps* (*snd* (*Strat* ! *n*)) (*steps′ init* (*take n qs*) (*take n Strat*) *n*)))
apply(*rule f*[*symmetric*])
apply(*fact*)
using *qsnset steps′_set*[*OF qQS*] *aS* **apply**(*simp*)
using *steps′_distinct*[*OF qQS*] *aS dI* **apply**(*simp*)
using *steps′_set*[*OF qQS*] *aS xyininit* **by** *simp_all*
also have ... = *x* < *y in* (*swap 0* ^^ *ALG_P* (*snd* (*Strat* ! *n*)) *x y* (*steps′ init* (*take n qs*) (*take n Strat*) *n*))
(*swaps sws* (*steps′* (*Lxy init* {*x*, *y*}) (*Lxy* (*take n qs*) {*x*, *y*}) *Strat2* (*length Strat2*)))
apply(*rule projected_paid_same_effect*)
apply(*rule steps′_dist_perm*)
apply(*fact qQS*)
apply(*fact aS*)
using *dI* **apply**(*simp*)
apply(*fact Suc*(*5*))
apply(*simp*)
apply(*rule steps′_set*[**where** *s*=*Lxy init* {*x*,*y*}, *unfolded ahjer*])
using *len* **apply**(*simp*)
apply(*simp*)
apply(*simp*)
apply(*rule steps′_length*[**where** *s*=*Lxy init* {*x*,*y*}, *unfolded ahjer2*])
using *len* **apply**(*simp*)

```
         apply(simp)
        apply(simp)
        apply(rule steps'_distinct2[where s=Lxy init {x,y}])
          using len apply(simp)
         apply(simp)
        apply(fact)
       using iff by auto

    finally have umfa: x < y in steps' init (take (Suc n) qs) (take (Suc n) Strat) (Suc n) =
  x < y
  in (swap 0 ^^ ALG_P (snd (Strat ! n)) x y (steps' init (take n qs) (take n Strat) n))
      (swaps sws (steps' (Lxy init {x, y}) (Lxy (take n qs) {x, y}) Strat2 (length Strat2))) .

    from Suc(3,4) have lS: length (take n Strat) = n by auto
    have (take n Strat @ [Strat ! n]) ! n =
            (take n Strat @ (Strat ! n) # []) ! length (take n Strat) using lS by auto
    also have ... = Strat ! n by(rule nth_append_length)
    finally have tt: (take n Strat @ [Strat ! n]) ! n = Strat ! n .

    show ?thesis
      apply(rule exI[where x=Strat2])
      apply(rule exI[where x=?L])
      unfolding nixzutun
      apply(safe)
         apply(fact)
        proof goal_cases
          case 1
          show ?case
          unfolding tak2 tak
          apply(simp add: step_def split_def)
          unfolding ALG_P'_def
          unfolding tt
            using aS apply(simp only: steps'_rests[OF qQS, symmetric])
           using 1(1) umfa by auto
        next
          case 2
          then show ?case
          apply(simp add: step_def split_def)
          unfolding ALG_P'_def
          unfolding tt
```

using *aS* **apply**(*simp only*: *steps′_rests*[*OF qQS*, *symmetric*])
using *umfa*[*symmetric*] **by** *auto*
next
case *3*
have *ns2*: *n* < *length* (*take n qs* @ [*qs* ! *n*])
using *ns* **by** *auto*

have *er*: *length* (*take n qs*) < *length Strat*
using *Suc.prems*(*2*) *aQ ns* **by** *linarith*

have T_p (*Lxy init* {*x*,*y*}) (*Lxy* (*take n qs*) {*x*, *y*}) *Strat2*
+ *length* (*replicate* (*ALG_P′ Strat* (*take n qs* @ [*qs* ! *n*]) *init n x y*) *0*
@ *sws*)
= (T_p (*Lxy init* {*x*,*y*}) (*Lxy* (*take n qs*) {*x*, *y*}) *Strat2* + *length sws*)
+ *ALG_P′ Strat* (*take n qs* @ [*qs* ! *n*]) *init n x y* **by** *simp*

also have ... = *ALGxy_det Strat* (*take n qs*) *init x y* +
ALG_Pxy Strat (*take n qs*) *init x y* +
ALG_P′ Strat (*take n qs* @ [*qs* ! *n*]) *init n x y*
unfolding *T_Strat2* **by** *simp*
also
have ... = *ALGxy_det Strat* (*take* (*Suc n*) *qs*) *init x y*
+ *ALG_Pxy Strat* (*take* (*Suc n*) *qs*) *init x y*
unfolding *tak* **unfolding** *wegdamit*[*OF er False*] **apply**(*simp*)
unfolding *ALG_P_split*[*of take n qs Strat qs* ! *n init x y*, *unfolded*
aQ, *OF nStrat*]
by(*simp*)
finally show *?case* **unfolding** *tak* **using** *ALG_P′_rest*[*OF ns*
nStrat] **by** *auto*
qed
next
case *True*
note *qsinxy*=*this*
then have *yeh*: *Lxy* (*take* (*Suc n*) *qs*) {*x*, *y*} = *Lxy* (*take n qs*) {*x*,*y*}
@ [*qs*!*n*]
unfolding *tak Lxy_def* **by** *auto*

from *True* **have** *garar*: (*take n qs* @ [*qs* ! *n*]) ! *n* ∈ {*y*, *x*}
using *tak*[*symmetric*] **by**(*auto*)
have *aer*: ∀ *i*<*n*.
((*take n qs* @ [*qs* ! *n*]) ! *i* ∈ {*y*, *x*})
= (*take n qs* ! *i* ∈ {*y*, *x*}) **using** *ns* **by** (*metis less_SucI nth_take*
tak)

let *?Strat_mft* = *fst* (*Strat* ! *n*)
let *?Strat_sws* = *snd* (*Strat* ! *n*)

let *?xs* = *steps' init* (*take n qs*) (*take n Strat*) *n*

let *?xs'* = (*swaps* (*snd* (*Strat*!*n*)) *?xs*)
let *?xs''* = *steps' init* (*take* (*Suc n*) *qs*) (*take* (*Suc n*) *Strat*) (*Suc n*)
let *?xs''2* = *mtf2 ?Strat_mft* (*qs*!*n*) *?xs'*

let *?no_swap_occurs* = (*x* < *y in ?xs'*) = (*x* < *y in ?xs''2*)

let *?mtf*=(*if ?no_swap_occurs then 0 else 1*::*nat*)
let *?m*=*ALG_P' Strat* (*take n qs* @ [*qs* ! *n*]) *init n x y*
let *?L*=*replicate ?m 0* @ *sws*

let *?newStrat*=*Strat2*@[(*?mtf*,*?L*)]

have *?xs''* = *step ?xs* (*qs*!*n*) (*Strat*!*n*)
unfolding *tak tak2*
apply(*rule steps'_append*) **by** *fact*+
also have ... = *mtf2* (*fst* (*Strat*!*n*)) (*qs*!*n*) (*swaps* (*snd* (*Strat*!*n*)) *?xs*)
unfolding *step_def*
by (*auto simp*: *split_def*)
finally have *A*: *?xs''* = *mtf2* (*fst* (*Strat*!*n*)) (*qs*!*n*) *?xs'* **.**

let *?ys* = (*steps'* (*Lxy init* {*x*, *y*})
(*Lxy* (*take n qs*) {*x*, *y*}) *Strat2* (*length Strat2*))
let *?ys'* = (*swaps sws* (*steps'* (*Lxy init* {*x*, *y*})
(*Lxy* (*take n qs*) {*x*, *y*}) *Strat2* (*length Strat2*)))
let *?ys''* = (*swap 0* ^^ *ALG_P* (*snd* (*Strat*!*n*)) *x y ?xs*) *?ys'*
let *?ys'''* = (*steps'* (*Lxy init* {*x*, *y*}) (*Lxy* (*take* (*Suc n*) *qs*) {*x*, *y*}) *?newStrat* (*length ?newStrat*))

have *gr*: *Lxy* (*take n qs* @ [*qs* ! *n*]) {*x*, *y*} =
Lxy (*take n qs*) {*x*, *y*} @ [*qs* ! *n*] **unfolding** *Lxy_def* **using** *True* **by**(*simp*)

have *steps' init* (*take n qs* @ [*qs* ! *n*]) *Strat n*
= *steps' init* (*take n qs* @ [*qs* ! *n*]) (*take n Strat* @ *drop n Strat*) *n* **by** *simp*
also have ... = *steps' init* (*take n qs*) (*take n Strat*) *n*

```
      apply(subst steps'_rests[symmetric]) using aS qQS by(simp_all)
  finally have t: steps' init (take n qs @ [qs ! n]) Strat n
    = steps' init (take n qs) (take n Strat) n .
  have gge: swaps (replicate ?m 0) ?ys'
    =  (swap 0 ^^ ALG_P (snd (Strat!n)) x y ?xs) ?ys'
      unfolding ALG_P'_def t by simp

  have gg: length ?newStrat = Suc (length Strat2) by auto
  have ?ys''' =  step ?ys (qs!n) (?mtf,?L)
      unfolding tak gr unfolding gg
      apply(rule steps'_append)
        using len by auto
  also have ... = mtf2 ?mtf (qs!n) (swaps ?L ?ys)
      unfolding step_def by (simp add: split_def)
  also have ... = mtf2 ?mtf (qs!n) (swaps (replicate ?m 0) ?ys')
    by (simp)
  also have ... = mtf2 ?mtf (qs!n) ?ys''
    using gge by (simp)
  finally have B: ?ys''' = mtf2 ?mtf (qs!n) ?ys'' .

  have 3: set ?ys' = {x,y}
    apply(simp add: swaps_inv) apply(subst steps'_set) using ahjer len
by(simp_all)
   have k: ?ys'' = swaps (replicate (ALG_P (snd (Strat!n)) x y ?xs) 0)
?ys'
    by (auto)
  have 6: set ?ys'' = {x,y} unfolding k using 3 swaps_inv by metis
   have 7: set ?ys''' = {x,y} unfolding B using set_mtf2 6 by metis

  have 22: x ∈ set ?ys'' y ∈ set ?ys'' using 6 by auto
  have 23: x ∈ set ?ys''' y ∈ set ?ys''' using 7 by auto

  have 26: (qs!n) ∈ set ?ys'' using 6 True by auto

  have distinct ?ys apply(rule steps'_distinct2)
    using len ahjer3 by(simp)+
  then have 9: distinct ?ys' using swaps_inv by metis
  then have 27: distinct ?ys'' unfolding k  using swaps_inv by metis

  from 3 Suc(5) have card (set ?ys') = 2 by auto
  then have 4: length ?ys' = 2 using distinct_card[OF 9] by simp
  have length ?ys'' = 2 unfolding k using 4 swaps_inv by metis
  have 5: dist_perm ?ys' ?ys' using 9 by auto
```

have *sxs*: *set ?xs* = *set init* **apply**(*rule steps'_set*) **using** *qQS n Suc*(*3*) **by**(*auto*)
have *sxs'*: *set ?xs'* = *set ?xs* **using** *swaps_inv* **by** *metis*
have *sxs''*: *set ?xs''* = *set ?xs'* **unfolding** *A* **using** *set_mtf2* **by** *metis*
have *24*: *x* ∈ *set ?xs' y*∈*set ?xs'* (*qs*!*n*) ∈ *set ?xs'*
  using *xysubs True sxs sxs'* **by** *auto*
have *28*: *x* ∈ *set ?xs'' y*∈*set ?xs''* (*qs*!*n*) ∈ *set ?xs''*
  using *xysubs True sxs sxs' sxs''* **by** *auto*

have *0*: *dist_perm init init* **using** *dI* **by** *auto*
have *1*: *dist_perm ?xs ?xs* **apply**(*rule steps'_dist_perm*)
 by *fact*+
then have *25*: *distinct ?xs'* **using** *swaps_inv* **by** *metis*

from *projected_paid_same_effect*[*OF 1 Suc*(*5*) *3 4 5*, *OF iff*, **where** *acs*=*snd* (*Strat* ! *n*)]
have *aaa*: *x* < *y in ?xs'* = *x* < *y in ?ys''* **.**

have *t*: *?mtf* = (*if* (*x*<*y in ?xs'*) = (*x*<*y in ?xs''*) *then 0 else 1*)
 by (*simp add*: *A*)

have *central*: *x* < *y in ?xs''* = *x* < *y in ?ys'''*
proof (*cases* (*x*<*y in ?xs'*) = (*x*<*y in ?xs''*))
 case *True*
 then have *?mtf* = *0* **using** *t* **by** *auto*
 with *B* **have** *?ys'''* = *?ys''* **by** *auto*
 with *aaa True* **show** *?thesis* **by** *auto*
next
 case *False*
 then have *k*: *?mtf* = *1* **using** *t* **by** *auto*
 from *False* **have** *i*: (*x*<*y in ?xs'*) = (~*x*<*y in ?xs''*) **by** *auto*

 have *gn*: ⋀*a b*. *a*∈{*x*,*y*} ⟹ *b*∈{*x*,*y*} ⟹ *set ?ys''* = {*x*,*y*} ⟹
   a≠*b* ⟹ *distinct ?ys''* ⟹
   a<*b in ?ys''* ⟹ ~*a*<*b in mtf2 1 b ?ys''*
 proof *goal_cases*
  case (*1 a b*)
  from *1* **have** *f*: *set ?ys''* = {*a*,*b*} **by** *auto*
  with *1* **have** *i*: *card* (*set ?ys''*) = *2* **by** *auto*

```
  from 1(5) have dist_perm ?ys'' ?ys'' by auto
  from i distinct_card 1(5) have g: length ?ys'' = 2 by metis
  with 1(6) have d: index ?ys'' b = 1
    using before_in_index2 f 1(4) by fastforce
  from 1(2,3) have e: b ∈ set ?ys'' by auto

  from d e have p: mtf2 1 b ?ys'' = swap 0 ?ys''
    unfolding mtf2_def by auto
  have q: a < b in swap 0 ?ys'' = (¬ a < b in ?ys'')
    apply(rule swap0in2) by(fact)+
  from 1(6) p q show ?case by metis
qed

show ?thesis
proof (cases x<y in ?xs')
  case True
  with aaa have st: x < y in ?ys'' by auto
  from True False have ~ x<y in ?xs'' by auto
  with Suc(5) 28 not_before_in A have y < x in ?xs'' by metis
  with A have y < x in mtf2 (fst (Strat!n)) (qs!n) ?xs' by auto

  have itisy: y = (qs!n)
    apply(rule swapped_by_mtf2[where xs= ?xs'])
         apply(fact)
        apply(fact)
       apply(fact 24)
      apply(fact 24)
     by(fact)+
  have ~x<y in mtf2 1 y ?ys''
    apply(rule gn)
         apply(simp)
        apply(simp)
       apply(simp add: 6)
      by(fact)+
  then have ts: ~x<y in ?ys''' using B itisy k by auto
  have ii: (x<y in ?ys'') = (~x<y in ?ys''') using st ts by auto
  from i ii aaa show ?thesis by metis
next
  case False
  with aaa have st: ~ x < y in ?ys'' by auto
  with Suc(5) 22 not_before_in have st: y < x in ?ys'' by metis
  from i False have kl: x<y in ?xs'' by auto
  with A have x < y in mtf2 (fst (Strat!n)) (qs!n) ?xs' by auto
  from False Suc(5) 24 not_before_in have y < x in ?xs' by metis
```

```
    have itisx: x = (qs!n)
      apply(rule swapped_by_mtf2[where xs= ?xs'])
           apply(fact)
          apply(fact)
         apply(fact 24(2))
        apply(fact 24)
       by(fact)+
    have ~y<x in mtf2 1 x ?ys''
      apply(rule gn)
           apply(simp)
          apply(simp)
         apply(simp add: 6)
        apply(metis Suc(5))
       by(fact)+
    then have ~y<x in ?ys''' using itisx k B by auto
    with Suc(5) not_before_in 23 have x<y in ?ys''' by metis
    with st have (x<y in ?ys'') = (~x<y in ?ys''') using  B k by auto
    with i aaa show ?thesis by metis
  qed
qed

show ?thesis
  apply(rule exI[where x=?newStrat])
  apply(rule exI[where x=[]])
  proof (standard, goal_cases)
    case 1
    show ?case unfolding yeh using len by(simp)
  next
    case 2
    show ?case
    proof (standard, goal_cases)
      case 1

      from central show ?case by auto
    next
      case 2

      have j: ALGxy_det Strat (take (Suc n) qs) init x y =
        ALGxy_det Strat (take n qs) init x y
            + (ALG'_det Strat qs init n y + ALG'_det Strat qs init n x)
      proof -
        have ALGxy_det Strat (take (Suc n) qs) init x y =
          (∑ i∈{..<length (take n qs @ [qs ! n])}.
          if (take n qs @ [qs ! n]) ! i ∈ {y, x}
```

then ALG′_det Strat (*take n qs* @ [*qs* ! *n*]) *init i y*
+ *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init i x*
else 0) **unfolding** *ALGxy_det_def tak* **by** *auto*
also have . . .
= (∑ *i*∈{..<*Suc n*}.
if (*take n qs* @ [*qs* ! *n*]) ! *i* ∈ {*y*, *x*}
then ALG′_det Strat (*take n qs* @ [*qs* ! *n*]) *init i y*
+ *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init i x*
else 0) **using** *ns* **by** *simp*
also have . . . = (∑ *i*∈{..<*n*}.
if (*take n qs* @ [*qs* ! *n*]) ! *i* ∈ {*y*, *x*}
then ALG′_det Strat (*take n qs* @ [*qs* ! *n*]) *init i y*
+ *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init i x*
else 0)
+ (*if* (*take n qs* @ [*qs* ! *n*]) ! *n* ∈ {*y*, *x*}
then ALG′_det Strat (*take n qs* @ [*qs* ! *n*]) *init n y*
+ *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init n x*
else 0) **by** *simp*
also have . . . = (∑ *i*∈{..< *n*}.
if take n qs ! *i* ∈ {*y*, *x*}
then ALG′_det Strat (*take n qs* @ [*qs* ! *n*]) *init i y*
+ *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init i x*
else 0)
+ *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init n y*
+ *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init n x*
using *aer* **using** *garar* **by** *simp*
also have . . . = (∑ *i*∈{..< *n*}.
if take n qs ! *i* ∈ {*y*, *x*}
then ALG′_det Strat (*take n qs* @ [*qs* ! *n*]) *init i y*
+ *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init i x*
else 0)
+ *ALG′_det Strat qs init n y* + *ALG′_det Strat qs init n x*
proof −
have *ALG′_det Strat qs init n y*
= *ALG′_det Strat* ((*take n qs* @ [*qs* ! *n*]) @ *drop* (*Suc n*) *qs*)
init n y
unfolding *tak*[*symmetric*] **by** *auto*
also have . . . = *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init n y*
apply(*rule ALG′_det_append*) **using** *nStrat ns* **by**(*auto*)
finally have *1*: *ALG′_det Strat qs init n y* = *ALG′_det Strat*
(*take n qs* @ [*qs* ! *n*]) *init n y* **.**
have *ALG′_det Strat qs init n x*
= *ALG′_det Strat* ((*take n qs* @ [*qs* ! *n*]) @ *drop* (*Suc n*) *qs*)
init n x

unfolding *tak*[*symmetric*] **by** *auto*
also have ... = *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init n x*
apply(*rule ALG′_det_append*) **using** *nStrat ns* **by**(*auto*)
finally have *2*: *ALG′_det Strat qs init n x* = *ALG′_det Strat* (*take n qs* @ [*qs* ! *n*]) *init n x* **.**
from *1 2* **show** *?thesis* **by** *auto*
qed
also have ... = ($\sum$ *i*∈{..< *n*}.
if take n qs ! *i* ∈ {*y*, *x*}
then ALG′_det Strat (*take n qs*) *init i y*
+ *ALG′_det Strat* (*take n qs*) *init i x*
else 0)
+ *ALG′_det Strat qs init n y* + *ALG′_det Strat qs init n x*
apply(*simp*)
apply(*rule sum.cong*)
apply(*simp*)
apply(*simp*)
using *ALG′_det_append*[**where** *qs*=*take n qs*] *Suc.prems*(*2*) *ns* **by** *auto*
also have ... = ($\sum$ *i*∈{..< *length*(*take n qs*)}.
if take n qs ! *i* ∈ {*y*, *x*}
then ALG′_det Strat (*take n qs*) *init i y*
+ *ALG′_det Strat* (*take n qs*) *init i x*
else 0)
+ *ALG′_det Strat qs init n y* + *ALG′_det Strat qs init n x*
using *aQ* **by** *auto*
also have ... = *ALGxy_det Strat* (*take n qs*) *init x y*
+ (*ALG′_det Strat qs init n y* + *ALG′_det Strat qs init n x*)
unfolding *ALGxy_det_def* **by**(*simp*)
finally show *?thesis* **.**
qed

have *list*: *?ys′* = *swaps sws* (*steps* (*Lxy init* {*x*, *y*}) (*Lxy* (*take n qs*) {*x*, *y*}) *Strat2*)
unfolding *steps_steps′*[*OF len*[*symmetric*], *of* (*Lxy init* {*x*, *y*})] **by** *simp*

have *j2*: *steps′ init* (*take n qs* @ [*qs* ! *n*]) *Strat n*
= *steps′ init* (*take n qs*) (*take n Strat*) *n*
proof −
have *steps′ init* (*take n qs* @ [*qs* ! *n*]) *Strat n*
= *steps′ init* (*take n qs* @ [*qs* ! *n*]) (*take n Strat* @ *drop n Strat*)

n
by *auto*
also have ... = *steps′ init* (*take n qs*) (*take n Strat*) *n*
apply(*rule steps′_rests*[*symmetric*]) apply *fact* using *aS* by *simp*
finally show *?thesis* .
qed

have *arghschonwieder*: *steps′ init* (*take n qs*) (*take n Strat*) *n*
= *steps′ init qs Strat n*
proof −
have *steps′ init qs Strat n*
= *steps′ init* (*take n qs* @ *drop n qs*) (*take n Strat* @ *drop n Strat*) *n*
by *auto*
also have ... = *steps′ init* (*take n qs*) (*take n Strat*) *n*
apply(*rule steps′_rests*[*symmetric*]) apply *fact* using *aS* by *simp*
finally show *?thesis* by *simp*
qed

have *indexe*: ((*swap 0* ^^ *?m*) (*swaps sws*
(*steps* (*Lxy init* {*x*,*y*}) (*Lxy* (*take n qs*) {*x*, *y*}) *Strat2*)))
= *?ys″* unfolding *ALG_P′_def* unfolding *list* using *j2* by *auto*

have *blocky*: *ALG′_det Strat qs init n y*
= (*if y* < *qs* ! *n in ?xs′ then 1 else 0*)
unfolding *ALG′_det_def ALG.simps* by(*auto simp*: *arghschonwieder split_def*)
have *blockx*: *ALG′_det Strat qs init n x*
= (*if x* < *qs* ! *n in ?xs′ then 1 else 0*)
unfolding *ALG′_det_def ALG.simps* by(*auto simp*: *arghschonwieder split_def*)

have *index_is_blocking_cost*: *index* ((*swap 0* ^^ *?m*) (*swaps sws*
(*steps* (*Lxy init* {*x*,*y*}) (*Lxy* (*take n qs*) {*x*, *y*}) *Strat2*))) (*qs* ! *n*)
= *ALG′_det Strat qs init n y* + *ALG′_det Strat qs init n x*
proof (*cases x*= *qs*!*n*)
case *True*
then have *ALG′_det Strat qs init n x* = *0*
unfolding *blockx* apply(*simp*) using *before_in_irefl* by *metis*

then have *ALG′_det Strat qs init n y* + *ALG′_det Strat qs init n x*
= (*if y* < *x in ?xs′ then 1 else 0*) **unfolding** *blocky* **using** *True* **by** *simp*
also have ... = (*if* ~*y* < *x in ?xs′ then 0 else 1*) **by** *auto*
also have ... = (*if x* < *y in ?xs′ then 0 else 1*)
apply(*simp*) **by** (*meson 24 Suc.prems*(*4*) *not_before_in*)
also have ... = (*if x* < *y in ?ys″ then 0 else 1*) **using** *aaa* **by** *simp*
also have ... = *index ?ys″ x*
apply(*rule before_in_index1*) **by**(*fact*)+
finally show *?thesis* **unfolding** *indexe* **using** *True* **by** *auto*
next
case *False*
then have *q*: *y* = *qs*!*n* **using** *qsinxy* **by** *auto*
then have *ALG′_det Strat qs init n y* = *0*
unfolding *blocky* **apply**(*simp*) **using** *before_in_irefl* **by** *metis*
then have *ALG′_det Strat qs init n y* + *ALG′_det Strat qs init n x*
= (*if x* < *y in ?xs′ then 1 else 0*) **unfolding** *blockx* **using** *q* **by** *simp*
also have ... = (*if x* < *y in ?ys″ then 1 else 0*) **using** *aaa* **by** *simp*
also have ... = *index ?ys″ y*
apply(*rule before_in_index2*) **by**(*fact*)+
finally show *?thesis* **unfolding** *indexe* **using** *q* **by** *auto*
qed

have *jj*: *ALG_Pxy Strat* (*take* (*Suc n*) *qs*) *init x y* =
ALG_Pxy Strat (*take n qs*) *init x y*
+ *ALG_P′ Strat* (*take n qs* @ [*qs* ! *n*]) *init n x y*
proof −
have *ALG_Pxy Strat* (*take* (*Suc n*) *qs*) *init x y*
= ($\sum$ *i*<*length* (*take* (*Suc n*) *qs*). *ALG_P′ Strat* (*take* (*Suc n*) *qs*) *init i x y*)
unfolding *ALG_Pxy_def* **by** *simp*
also have ... = ($\sum$ *i*< *Suc n*. *ALG_P′ Strat* (*take* (*Suc n*) *qs*) *init i x y*)
unfolding *tak* **using** *ns* **by** *simp*
also have ... = ($\sum$ *i*<*n*. *ALG_P′ Strat* (*take* (*Suc n*) *qs*) *init i x y*)
+ *ALG_P′ Strat* (*take* (*Suc n*) *qs*) *init n x y*
by *simp*
also have ... = ($\sum$ *i*<*length* (*take n qs*). *ALG_P′ Strat* (*take n*

qs @ [qs ! n]) init i x y)
+ ALG_P' Strat (take n qs @ [qs ! n]) init n x y
unfolding *tak* **using** *ns* **by** *auto*
also have ... = ($\sum$ i<length (take n qs). ALG_P' Strat (take n qs) init i x y)
+ ALG_P' Strat (take n qs @ [qs ! n]) init n x y (**is** ?A + ?B = ?A' + ?B)
proof −
have ?A = ?A'
apply(*rule sum.cong*)
apply(*simp*)
proof *goal_cases*
case *1*
show ?case
apply(*rule ALG_P'_rest2*[*symmetric*, **where** ?r1.0=[], *simplified*])
using *1* **apply**(*simp*)
using *1 nStrat* **by**(*simp*)
qed
then show ?thesis **by** *auto*
qed
also have ... = ALG_Pxy Strat (take n qs) init x y
+ ALG_P' Strat (take n qs @ [qs ! n]) init n x y
unfolding *ALG_Pxy_def* **by** *auto*
finally show ?thesis **.**
qed

have *tw*: length (Lxy (take n qs) {x, y}) = length Strat2
using *len* **by** *auto*
have T_p (Lxy init {x,y}) (Lxy (take (Suc n) qs) {x, y}) ?newStrat + length []
= T_p (Lxy init {x,y}) (Lxy (take n qs) {x, y}) Strat2
+ t_p (steps (Lxy init {x, y}) (Lxy (take n qs) {x, y}) Strat2) (qs ! n) (?mtf,?L)
unfolding *yeh*
by(*simp add*: *T_append*[*OF tw*, *of* (*Lxy init*) {*x*,*y*}])
also have ... =
T_p (Lxy init {x,y}) (Lxy (take n qs) {x, y}) Strat2
+ length sws
+ index ((swap 0 ^^ ?m) (swaps sws
(steps (Lxy init {x,y}) (Lxy (take n qs) {x, y}) Strat2))) (qs ! n)
+ ALG_P' Strat (take n qs @ [qs ! n]) init n x y
by(*simp add*: t_p*_def*)

also have ... = (*ALGxy_det Strat* (*take n qs*) *init x y*
+ *index* ((*swap 0* ^^ *?m*) (*swaps sws*
(*steps* (*Lxy init* {*x*,*y*}) (*Lxy* (*take n qs*) {*x*, *y*}) *Strat2*)))
(*qs* ! *n*))
+ (*ALG_Pxy Strat* (*take n qs*) *init x y*
+ *ALG_P′ Strat* (*take n qs* @ [*qs* ! *n*]) *init n x y*)
by (*simp only*: *T_Strat2*)

also from *index_is_blocking_cost* **have** ... = (*ALGxy_det Strat*
(*take n qs*) *init x y*
+ *ALG′_det Strat qs init n y* + *ALG′_det Strat qs init n x*)
+ (*ALG_Pxy Strat* (*take n qs*) *init x y*
+ *ALG_P′ Strat* (*take n qs* @ [*qs* ! *n*]) *init n x y*) **by** *auto*
also have ... = *ALGxy_det Strat* (*take* (*Suc n*) *qs*) *init x y*
+ (*ALG_Pxy Strat* (*take n qs*) *init x y*
+ *ALG_P′ Strat* (*take n qs* @ [*qs* ! *n*]) *init n x y*) **using** *j*
by *auto*
also have ... = *ALGxy_det Strat* (*take* (*Suc n*) *qs*) *init x y*
+ *ALG_Pxy Strat* (*take* (*Suc n*) *qs*) *init x y* **using** *jj* **by** *auto*
finally show *?case* **.**
qed
qed
qed
next
case *0*
then show *?case*
apply (*simp add*: *Lxy_def ALGxy_det_def ALG_Pxy_def T_opt_def*)
proof *goal_cases*
case *1*
show *?case* **apply**(*rule Lxy_mono*[*unfolded Lxy_def*, *simplified*])
using *1* **by** *auto*
qed
qed

lemma *T1_7*:
assumes T_p *init qs Strat* = T_p*_opt init qs length Strat* = *length qs*
x ≠ (*y*::(*′a*::*linorder*)) *x*∈ *set init y* ∈ *set init distinct init*
set qs ⊆ *set init*
shows T_p*_opt* (*Lxy init* {*x*,*y*}) (*Lxy qs* {*x*,*y*})
≤ *ALGxy_det Strat qs init x y* + *ALG_Pxy Strat qs init x y*
proof −
have *A*:*length qs* ≤ *length qs* **by** *auto*

```
  have B:  x ≠ y  using assms by auto

  from T1_7'[OF assms(1,2), of length qs x y, OF A B assms(4−7)]
  obtain Strat2 sws where
      len: length Strat2 = length (Lxy qs {x, y})
    and x < y in steps' init qs (take (length qs) Strat)
        (length qs) = x < y in swaps sws (steps' (Lxy init {x,y})
          (Lxy qs {x, y}) Strat2 (length Strat2))
    and Tp: T_p (Lxy init {x,y}) (Lxy qs {x, y}) Strat2 + length sws
       =  ALGxy_det Strat qs init x y
        + ALG_Pxy Strat qs init x y by auto

  have T_p_opt (Lxy init {x,y}) (Lxy qs {x,y}) ≤ T_p (Lxy init {x,y}) (Lxy
qs {x, y}) Strat2
    unfolding T_opt_def
    apply(rule cInf_lower)
      using len by auto
  also have ... ≤ ALGxy_det Strat qs init x y
        + ALG_Pxy Strat qs init x y using Tp by auto
  finally show ?thesis .
qed

lemma T_snoc: length rs = length as
       ⟹  T init (rs@[r]) (as@[a])
        = T init rs as + t_p (steps' init rs as (length rs)) r a
apply(induct rs as arbitrary: init rule: list_induct2) by simp_all

lemma steps'_snoc: length rs = length as ⟹ n = (length as)
       ⟹ steps' init (rs@[r]) (as@[a]) (Suc n) = step (steps' init rs as n) r
a
apply(induct rs as arbitrary: init n r a rule: list_induct2)
  by (simp_all)

lemma steps'_take:
  assumes n<length qs length qs = length Strat
  shows steps' init (take n qs) (take n Strat) n
      = steps' init qs Strat n
proof −
  have steps' init qs Strat n =
     steps' init (take n qs @ drop n qs) (take n Strat @ drop n Strat) n  by
simp
  also have ... = steps' init (take n qs) (take n Strat) n
      apply(subst steps'_rests[symmetric]) using assms  by auto
  finally show ?thesis by simp
```

qed

lemma *Tp_darstellung*: *length qs* = *length Strat*
$\Longrightarrow$ T_p *init qs Strat* =
($\sum$ *i*∈{..<*length qs*}. t_p (*steps' init qs Strat i*) (*qs*!*i*) (*Strat*!*i*))
proof –
assume *a*[*simp*]: *length qs* = *length Strat*
{fix *n*
have *n*≤*length qs*
$\Longrightarrow$ T_p *init* (*take n qs*) (*take n Strat*) =
($\sum$ *i*∈{..<*n*}. t_p (*steps' init qs Strat i*) (*qs*!*i*) (*Strat*!*i*))
apply(*induct n*)
apply(*simp*)
apply(*simp add*: *take_Suc_conv_app_nth*)
apply(*subst T_snoc*)
apply(*simp*)
by(*simp add*: *min_def steps'_take*)
}
from *a this*[*of length qs*] **show** *?thesis* **by** *auto*
qed

lemma *umformung_OPT'*:
assumes *inlist*: *set qs* ⊆ *set init*
assumes *dist*: *distinct init*
assumes *qsStrat*: *length qs* = *length Strat*
assumes *noStupid*: ⋀*x l*. *x*<*length Strat* $\Longrightarrow$ *l*< *length* (*snd* (*Strat* ! *x*))
$\Longrightarrow$ *Suc* ((*snd* (*Strat* ! *x*))!*l*) < *length init*
shows T_p *init qs Strat* =
($\sum$ (*x*,*y*)∈{(*x*,*y*::('*a*::*linorder*)). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*<*y*}.
ALGxy_det Strat qs init x y + *ALG_Pxy Strat qs init x y*)
proof –

have ($\sum$ *i*∈{..<*length qs*}.
($\sum$ (*x*,*y*)∈{(*x*,*y*). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*<*y*}. *ALG_P* (*snd* (*Strat*!*i*)) *x y* (*steps' init qs Strat i*)))
= ($\sum$ *i*∈{..<*length qs*}.
($\sum$ *z*∈{(*x*,*y*). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*<*y*}. *ALG_P* (*snd* (*Strat*!*i*)) (*fst z*) (*snd z*) (*steps' init qs Strat i*)))
by(*auto simp*: *split_def*)
also have . . .

```
      = (∑ z∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
              (∑ i∈{..<length qs}. ALG_P (snd (Strat!i)) (fst z) (snd z)
(steps' init qs Strat i)) )
         by(rule sum.swap)
  also have ... = (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
             (∑ i∈{..<length qs}. ALG_P (snd (Strat!i)) x y (steps' init qs
Strat i)) )
         by(auto simp: split_def)
  also have ... = (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
             ALG_Pxy Strat qs init x y)
         unfolding ALG_P'_def ALG_Pxy_def by auto
  finally have paid_part: (∑ i∈{..<length qs}.
      (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ALG_P (snd
(Strat!i)) x y (steps' init qs Strat i)) )
      = (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
             ALG_Pxy Strat qs init x y) .


  let ?config = (%i. swaps (snd (Strat!i)) (steps' init qs Strat i))

  have (∑ i∈{..<length qs}.
             (∑ e∈set init. ALG e qs i (?config i, ())))
        = (∑ e∈set init.
            (∑ i∈{..<length qs}. ALG e qs i (?config i, ())))
            by(rule sum.swap)
  also have ... = (∑ e∈set init.
          (∑ y∈set init.
            (∑ i∈{i. i<length qs ∧ qs!i=y}. ALG e qs i (?config i,()))))
            proof (rule sum.cong, goal_cases)
              case (2 x)
              have (∑ i<length qs. ALG x qs i (?config i, ()))
                = sum (%i. ALG x qs i (?config i, ())) {i. i<length qs}
                by (simp add: lessThan_def)
              also have ... = sum (%i. ALG x qs i (?config i, ()))
                         (⋃ y∈{y. y ∈ set init}. {i. i < length qs ∧ qs ! i = y})
                         apply(rule sum.cong)
                         proof goal_cases
                          case 1
                          show ?case apply(auto) using inlist by auto
                         qed simp
             also have ... = sum (%t. sum (%i. ALG x qs i (?config i, ()))
{i. i<length qs ∧ qs ! i = t}) {y. y∈ set init}
                apply(rule sum.UNION_disjoint)
```

apply(*simp_all*) **by** *force*
also have ... = ($\sum$ *y*∈*set init*. $\sum$ *i* | *i* < *length qs* ∧ *qs* ! *i* = *y*. *ALG x qs i* (*?config i*, ())) **by** *auto*
finally show *?case* **.**
qed (*simp*)
also have ... = ($\sum$ (*x*,*y*)∈ (*set init* × *set init*). ($\sum$ *i*∈{*i*. *i*<*length qs* ∧ *qs*!*i*=*y*}. *ALG x qs i* (*?config i*, ())))
by (*rule sum.cartesian_product*)
also have ... = ($\sum$ (*x*,*y*)∈ {(*x*,*y*). *x*∈*set init* ∧ *y*∈ *set init*}. ($\sum$ *i*∈{*i*. *i*<*length qs* ∧ *qs*!*i*=*y*}. *ALG x qs i* (*?config i*, ())))
by *simp*
also have *E4*: ... = ($\sum$ (*x*,*y*)∈{(*x*,*y*). *x*∈*set init* ∧ *y*∈ *set init* ∧ *x*≠*y*}. ($\sum$ *i*∈{*i*. *i*<*length qs* ∧ *qs*!*i*=*y*}. *ALG x qs i* (*?config i*, ()))) (**is** ($\sum$ (*x*,*y*)∈ *?L*. *?f x y*) = ($\sum$ (*x*,*y*)∈ *?R*. *?f x y*))
proof *goal_cases*
case *1*
let *?M* = {(*x*,*y*). *x*∈*set init* ∧ *y*∈ *set init* ∧ *x*=*y*}
have *A*: *?L* = *?R* ∪ *?M* **by** *auto*
have *B*: {} = *?R* ∩ *?M* **by** *auto*
have ($\sum$ (*x*,*y*)∈ *?L*. *?f x y*) = ($\sum$ (*x*,*y*)∈ *?R* ∪ *?M*. *?f x y*)
by(*simp only*: *A*)
also have ... = ($\sum$ (*x*,*y*)∈ *?R*. *?f x y*) + ($\sum$ (*x*,*y*)∈ *?M*. *?f x y*)
apply(*rule sum.union_disjoint*)
apply(*rule finite_subset*[**where** *B*=*set init* × *set init*])
apply(*auto*)
apply(*rule finite_subset*[**where** *B*=*set init* × *set init*])
by(*auto*)
also have ($\sum$ (*x*,*y*)∈ *?M*. *?f x y*) = *0*
apply(*rule sum.neutral*)
by (*auto simp add*: *split_def before_in_def*)
finally show *?case* **by** *simp*
qed

also have ... = ($\sum$ (*x*,*y*)∈{(*x*,*y*). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*<*y*}. ($\sum$ *i*∈{*i*. *i*<*length qs* ∧ *qs*!*i*=*y*}. *ALG x qs i* (*?config i*, ()))
+ ($\sum$ *i*∈{*i*. *i*<*length qs* ∧ *qs*!*i*=*x*}. *ALG y qs i* (*?config i*, ())))
(**is** ($\sum$ (*x*,*y*)∈ *?L*. *?f x y*) = ($\sum$ (*x*,*y*)∈ *?R*. *?f x y* + *?f y x*))
proof −
let *?R′* = {(*x*,*y*). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *y*<*x*}
have *A*: *?L* = *?R* ∪ *?R′* **by** *auto*
have {} = *?R* ∩ *?R′* **by** *auto*
have *C*: *?R′* = (%(*x*,*y*). (*y*, *x*)) ‘ *?R* **by** *auto*

have *D*: ($\sum$ (*x*,*y*)∈ *?R′*. *?f x y*) = ($\sum$ (*x*,*y*)∈ *?R*. *?f y x*)

proof –
have (∑ (*x*,*y*)∈ *?R'*. *?f x y*) = (∑ (*x*,*y*)∈ (%(*x*,*y*). (*y*, *x*)) ‘ *?R*. *?f x y*)
by(*simp only*: *C*)
also have (∑ *z*∈ (%(*x*,*y*). (*y*, *x*)) ‘ *?R*. (%(*x*,*y*). *?f x y*) *z*) = (∑ *z*∈*?R*. ((%(*x*,*y*). *?f x y*) ∘ (%(*x*,*y*). (*y*, *x*))) *z*)
apply(*rule sum.reindex*)
by(*fact swap_inj_on*)
also have ... = (∑ *z*∈*?R*. (%(*x*,*y*). *?f y x*) *z*)
apply(*rule sum.cong*)
by(*auto*)
finally show *?thesis* **.**
qed

have (∑ (*x*,*y*)∈ *?L*. *?f x y*) = (∑ (*x*,*y*)∈ *?R* ∪ *?R'*. *?f x y*)
by(*simp only*: *A*)
also have ... = (∑ (*x*,*y*)∈ *?R*. *?f x y*) + (∑ (*x*,*y*)∈ *?R'*. *?f x y*)
apply(*rule sum.union_disjoint*)
apply(*rule finite_subset*[**where** *B*=*set init* × *set init*])
apply(*auto*)
apply(*rule finite_subset*[**where** *B*=*set init* × *set init*])
by(*auto*)
also have ... = (∑ (*x*,*y*)∈ *?R*. *?f x y*) + (∑ (*x*,*y*)∈ *?R*. *?f y x*)
by(*simp only*: *D*)
also have ... = (∑ (*x*,*y*)∈ *?R*. *?f x y* + *?f y x*)
by(*simp add*: *split_def sum.distrib*[*symmetric*])
finally show *?thesis* **.**
qed

also have *E5*: ... = (∑ (*x*,*y*)∈{(*x*,*y*). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*<*y*}. (∑ *i*∈{*i*. *i*<*length qs* ∧ (*qs*!*i*=*y* ∨ *qs*!*i*=*x*)}. *ALG y qs i* (*?config i*, ()) + *ALG x qs i* (*?config i*, ())))
apply(*rule sum.cong*)
apply(*simp*)
proof *goal_cases*
case (*1 x*)
then obtain *a b* **where** *x*: *x*=(*a*,*b*) **and** *a*: *a* ∈ *set init b* ∈ *set init a* < *b* **by** *auto*
then have *a*≠*b* **by** *simp*
then have *disj*: {*i*. *i* < *length qs* ∧ *qs* ! *i* = *b*} ∩ {*i*. *i* < *length qs* ∧ *qs* ! *i* = *a*} = {} **by** *auto*
have *unio*: {*i*. *i* < *length qs* ∧ (*qs* ! *i* = *b* ∨ *qs* ! *i* = *a*)}
= {*i*. *i* < *length qs* ∧ *qs* ! *i* = *b*} ∪ {*i*. *i* < *length qs* ∧ *qs* ! *i* =

a} **by** *auto*
let *?f*=%*i*. *ALG b qs i* (*?config i*, ()) +
ALG a qs i (*?config i*, ())
let *?B*={*i*. *i* < *length qs* ∧ *qs* ! *i* = *b*}
let *?A*={*i*. *i* < *length qs* ∧ *qs* ! *i* = *a*}
have (∑ *i*∈*?B* ∪ *?A*. *?f i*)
= (∑ *i*∈*?B*. *?f i*) + (∑ *i*∈*?A*. *?f i*) − (∑ *i*∈*?B* ∩ *?A*. *?f i*)
apply(*rule sum_Un_nat*) **by** *auto*
also have ... = (∑ *i*∈*?B*. *ALG b qs i* (*?config i*, ()) + *ALG a qs i* (*?config i*, ()))
+ (∑ *i*∈*?A*. *ALG b qs i* (*?config i*, ()) + *ALG a qs i* (*?config i*, ()))
using *disj* **by** *auto*
also have ... = (∑ *i*∈*?B*. *ALG a qs i* (*?config i*, ()))
+ (∑ *i*∈*?A*. *ALG b qs i* (*?config i*, ()))
by (*auto simp*: *split_def before_in_def*)
finally
show *?case* **unfolding** *x* **apply**(*simp add*: *split_def*)
unfolding *unio* **by** *simp*
qed
also have *E6*: ... = (∑ (*x*,*y*)∈{(*x*,*y*). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*<*y*}.
ALGxy_det Strat qs init x y)
apply(*rule sum.cong*)
unfolding *ALGxy_det_alternativ* **unfolding** *ALG′_det_def* **by** *auto*
finally have *blockingpart*: (∑ *i*<*length qs*.
∑ *e*∈*set init*.
ALG e qs i (*?config i*, ()))
= (∑ (*x*,*y*)∈{(*x*,*y*). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*<*y*}.
ALGxy_det Strat qs init x y) **.**
from *Tp_darstellung*[*OF qsStrat*] **have** *E0*: T_p *init qs Strat* =
(∑ *i*∈{..<*length qs*}. t_p (*steps′ init qs Strat i*) (*qs*!*i*) (*Strat*!*i*))
by *auto*
also have ... = (∑ *i*∈{..<*length qs*}.
(∑ *e*∈*set* (*steps′ init qs Strat i*). *ALG e qs i* (*swaps* (*snd* (*Strat*!*i*)) (*steps′ init qs Strat i*),()))
+ (∑ (*x*,*y*)∈{(*x*,(*y*::(′*a*::*linorder*))). *x* ∈ *set* (*steps′ init qs Strat i*) ∧ *y*∈*set* (*steps′ init qs Strat i*) ∧ *x*<*y*}. *ALG_P* (*snd* (*Strat*!*i*)) *x y* (*steps′ init qs Strat i*)))
apply(*rule sum.cong*)
apply(*simp*)
apply (*rule* t_p*_sumofALGALGP*)
apply(*rule steps′_distinct2*)
using *dist qsStrat* **apply**(*simp_all*)

```
        apply(subst steps'_set)
          using dist qsStrat inlist apply(simp_all)
       apply fastforce
      apply(subst steps'_length)
        apply(simp_all)
          using noStupid by auto
  also have ... = (∑ i∈{..<length qs}.
              (∑ e∈set init. ALG e qs i (swaps (snd (Strat!i)) (steps' init qs
Strat i),()))
+ (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ALG_P (snd (Strat!i))
x y (steps' init qs Strat i)) )
               apply(rule sum.cong)
                 apply(simp)
                 proof goal_cases
                   case (1 x)
                   then have set (steps' init qs Strat x) = set init
                     apply(subst steps'_set)
                     using dist qsStrat 1 by(simp_all)
                   then show ?case by simp
                 qed
  also have ... = (∑ i∈{..<length qs}.
              (∑ e∈set init. ALG e qs i (swaps (snd (Strat!i)) (steps' init qs
Strat i), ())))
              + (∑ i∈{..<length qs}.
                (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ALG_P
(snd (Strat!i)) x y (steps' init qs Strat i)) )
    by (simp add: sum.distrib split_def)
  also have ... = (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
                    ALGxy_det Strat qs init x y)
              + (∑ i∈{..<length qs}.
                (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}. ALG_P
(snd (Strat!i)) x y (steps' init qs Strat i)) )
               by(simp only: blockingpart)
  also have ... = (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
                    ALGxy_det Strat qs init x y)
              + (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
               ALG_Pxy Strat qs init x y)
               by(simp only: paid_part)
  also have ... = (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
                    ALGxy_det Strat qs init x y
              +    ALG_Pxy Strat qs init x y)
    by (simp add: sum.distrib split_def)
  finally show ?thesis by auto
qed
```

lemma *nn_contains_Inf*:
  fixes *S* :: *nat set*
  assumes *nn*: $S \neq \{\}$
  shows *Inf S* $\in$ *S*
using *assms Inf_nat_def LeastI* **by** *force*

lemma *steps_length*: *length qs* = *length as* $\Longrightarrow$ *length* (*steps s qs as*) = *length s*
apply(*induct qs as arbitrary*: *s rule*: *list_induct2*)
  by *simp_all*

lemma *OPT_noStupid*:
  fixes *Strat*
  assumes [*simp*]: *length Strat* = *length qs*
  assumes *opt*: T_p *init qs Strat* = T_p*_opt init qs*
  assumes *init_nempty*: *init*$\neq$[]
  shows $\bigwedge$*x l*. *x* < *length Strat* $\Longrightarrow$
      l < *length* (*snd* (*Strat* ! *x*)) $\Longrightarrow$
      Suc ((*snd* (*Strat* ! *x*))!*l*) < *length init*
proof (*rule ccontr*, *goal_cases*)
  case (*1 x l*)

  let *?sws'* = *take l* (*snd* (*Strat*!*x*)) @ *drop* (*Suc l*) (*snd* (*Strat*!*x*))
  let *?Strat'* = *take x Strat* @ (*fst* (*Strat*!*x*),*?sws'*) # *drop* (*Suc x*) *Strat*

  from *1*(*1*) **have** *valid*: *length ?Strat'* = *length qs* **by** *simp*
  from *valid* **have** *isin*: T_p *init qs ?Strat'* $\in$ {T_p *init qs as* |*as*. *length as* = *length qs*} **by** *blast*

  from *1*(*1*,*2*) **have** *lsws'*: *length* (*snd* (*Strat*!*x*)) = *length ?sws'* + *1*
   by (*simp*)

  have *a*: (*take x ?Strat'*) = (*take x Strat*)
   using *1*(*1*) **by**(*auto simp add*: *min_def take_Suc_conv_app_nth*)
  have *b*: (*drop* (*Suc x*) *Strat*) = (*drop* (*Suc x*) *?Strat'*)
   using *1*(*1*) **by**(*auto simp add*: *min_def take_Suc_conv_app_nth*)

have *aa*: (*take l* (*snd* (*Strat*!*x*))) = (*take l* (*snd* (*?Strat′*!*x*)))
 using *1*(*1*,*2*) **by**(*auto simp add*: *min_def take_Suc_conv_app_nth nth_append*)
have *bb*: (*drop* (*Suc l*) (*snd* (*Strat*!*x*))) = (*drop l* (*snd* (*?Strat′*!*x*)))
 using *1*(*1*,*2*) **by**(*auto simp add*: *min_def take_Suc_conv_app_nth nth_append*)

have (*swaps* (*snd* (*Strat* ! *x*)) (*steps init* (*take x qs*) (*take x Strat*)))
 = (*swaps* (*take l* (*snd* (*Strat* ! *x*)) @ (*snd* (*Strat* ! *x*))!*l* # *drop* (*Suc l*) (*snd* (*Strat* ! *x*))) (*steps init* (*take x qs*) (*take x Strat*)))
 unfolding *id_take_nth_drop*[*OF 1*(*2*), *symmetric*] **by** *simp*
also have ...
 = (*swaps* (*take l* (*snd* (*Strat* ! *x*)) @ *drop* (*Suc l*) (*snd* (*Strat* ! *x*))) (*steps init* (*take x qs*) (*take x Strat*)))
 using *1*(*3*) **by**(*simp add*: *swap_def steps_length*)
finally have *noeffect*: (*swaps* (*snd* (*Strat* ! *x*)) (*steps init* (*take x qs*) (*take x Strat*)))
 = (*swaps* (*take l* (*snd* (*Strat* ! *x*)) @ *drop* (*Suc l*) (*snd* (*Strat* ! *x*))) (*steps init* (*take x qs*) (*take x Strat*)))
 .

have *c*: t_p (*steps init* (*take x qs*) (*take x Strat*)) (*qs* ! *x*) (*Strat* ! *x*) =
 t_p (*steps init* (*take x qs*) (*take x ?Strat′*)) (*qs* ! *x*) (*?Strat′* ! *x*) + *1*
 unfolding *a* t_p*_def* **using** *1*(*1*,*2*)
 apply(*simp add*: *min_def split_def nth_append*) **unfolding** *noeffect*
 by(*simp*)

have T_p *init* (*take* (*Suc x*) *qs*) (*take* (*Suc x*) *Strat*)
 = T_p *init* (*take x qs*) (*take x ?Strat′*) +
 t_p (*steps init* (*take x qs*) (*take x Strat*)) (*qs* ! *x*) (*Strat* ! *x*)
 using *1*(*1*) *a* **by**(*simp add*: *take_Suc_conv_app_nth T_append*)
also have ... = T_p *init* (*take x qs*) (*take x ?Strat′*) +
 t_p (*steps init* (*take x qs*) (*take x ?Strat′*)) (*qs* ! *x*) (*?Strat′* ! *x*) + *1*
 unfolding *c* **by**(*simp*)
also have ... = T_p *init* (*take* (*Suc x*) *qs*) (*take* (*Suc x*) *?Strat′*) + *1*
 using *1*(*1*) *a* **by**(*simp add*: *min_def take_Suc_conv_app_nth T_append nth_append*)
finally have *bef*: T_p *init* (*take* (*Suc x*) *qs*) (*take* (*Suc x*) *Strat*)
 = T_p *init* (*take* (*Suc x*) *qs*) (*take* (*Suc x*) *?Strat′*) + *1* **.**

let *?interstate* = (*steps init* (*take* (*Suc x*) *qs*) (*take* (*Suc x*) *Strat*))
let *?interstate′* = (*steps init* (*take* (*Suc x*) *qs*) (*take* (*Suc x*) *?Strat′*))

have *state*: *?interstate'* = *?interstate*
  using *1(1)* **apply**(*simp add*: *take_Suc_conv_app_nth min_def*)
  apply(*simp add*: *steps_append step_def split_def*) **using** *noeffect* **by** *simp*

have T_p *init qs Strat*
  = T_p *init* (*take* (*Suc x*) *qs* @ *drop* (*Suc x*) *qs*) (*take* (*Suc x*) *Strat* @ *drop* (*Suc x*) *Strat*)
  by *simp*
also have ... = T_p *init* (*take* (*Suc x*) *qs*) (*take* (*Suc x*) *Strat*)
  + T_p *?interstate* (*drop* (*Suc x*) *qs*) (*drop* (*Suc x*) *Strat*)
  apply(*subst T_append2*) **by**(*simp_all*)
also have ... = T_p *init* (*take* (*Suc x*) *qs*) (*take* (*Suc x*) *?Strat'*)
  + T_p *?interstate'* (*drop* (*Suc x*) *qs*) (*drop* (*Suc x*) *?Strat'*) + *1*
  unfolding *bef state* **using** *1(1)* **by**(*simp add*: *min_def nth_append*)
also have ... = T_p *init* (*take* (*Suc x*) *qs* @ *drop* (*Suc x*) *qs*) (*take* (*Suc x*) *?Strat'* @ *drop* (*Suc x*) *?Strat'*) + *1*
  apply(*subst T_append2*) **using** *1(1)* **by**(*simp_all add*: *min_def*)

also have ... = T_p *init qs ?Strat'* + *1* **by** *simp*
finally have *better*: T_p *init qs ?Strat'* + *1* = T_p *init qs Strat* **by** *simp*

have T_p *init qs ?Strat'* + *1* = T_p *init qs Strat* **by** (*fact better*)
also have ... = T_p*_opt init qs* **by** (*fact opt*)
also from *cInf_lower*[*OF isin*] **have** ... ≤ T_p *init qs ?Strat'* **unfolding** *T_opt_def* **by** *simp*
finally show *False* **using** *init_nempty* **by** *auto*
qed

lemma *umformung_OPT*:
  assumes *inlist*: *set qs* ⊆ *set init*
  assumes *dist*: *distinct init*
  assumes *a*: T_p*_opt init qs* = T_p *init qs Strat*
  assumes *b*: *length qs* = *length Strat*
  assumes *c*: *init*≠[]
  shows T_p*_opt init qs* =
  (∑ (*x*,*y*)∈{(*x*,*y*::('*a*::*linorder*)). *x* ∈ *set init* ∧ *y*∈*set init* ∧ *x*<*y*}.
  ALGxy_det Strat qs init x y + *ALG_Pxy Strat qs init x y*)
proof −
  have T_p*_opt init qs* = T_p *init qs Strat* **by**(*fact a*)

also have ... =
(∑ (x,y)∈{(x,y::('a::linorder)). x ∈ set init ∧ y∈set init ∧ x<y}.
ALGxy_det Strat qs init x y + ALG_Pxy Strat qs init x y)
apply(*rule umformung_OPT'*)
apply(*fact*)+
using *OPT_noStupid*[*OF b*[*symmetric*] *a*[*symmetric*] *c*] **apply**(*simp*) **done**
finally show *?thesis* **.**
qed

corollary *OPT_zerlegen*:
assumes
dist: *distinct init*
and *c*: *init≠*[]
and *setqsinit*: *set qs ⊆ set init*
shows (∑ (x,y)∈{(x,y::('a::linorder)). x ∈ set init ∧ y∈set init ∧ x<y}. (T_p_opt (Lxy init {x,y}) (Lxy qs {x,y})))
≤ T_p_opt init qs
proof −

have T_p_opt init qs ∈ {T_p init qs as |as. length as = length qs}
unfolding *T_opt_def*
apply(*rule nn_contains_Inf*)
apply(*auto*) **by** (*rule Ex_list_of_length*)

then obtain *Strat* **where** *a*: T_p *init qs Strat* = T_p_opt init qs
and *b*: *length Strat = length qs*
unfolding *T_opt_def* **by** *auto*

have (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
T_p_opt (Lxy init {x,y}) (Lxy qs {x, y})) ≤ (∑ (x,y)∈{(x,y). x ∈ set init ∧ y∈set init ∧ x<y}.
ALGxy_det Strat qs init x y + ALG_Pxy Strat qs init x y)
apply (*rule sum_mono*)
apply(*auto*)
proof *goal_cases*
case (*1 a b*)
then have *a≠b* **by** *auto*
show *?case* **apply**(*rule T1_7*[*OF a b*]) **by**(*fact*)+
qed
also from *umformung_OPT*[*OF setqsinit dist*] *a b c* **have** ... = T_p *init qs Strat* **by** *auto*
also from *a* **have** ... = T_p_opt init qs **by** *simp*

```
  finally show ?thesis .
qed
```

14.5 Factoring Lemma

```
lemma cardofpairs: S ≠ [] ⟹ sorted S ⟹ distinct S ⟹ card {(x,y). x
∈ set S ∧ y∈set S ∧ x<y} = ((length S)∗(length S−1)) / 2
proof (induct S rule: list_nonempty_induct)
  case (cons s ss)
  then have sorted ss distinct ss by auto
  from cons(2)[OF this(1) this(2)] have iH: card {(x, y) . x ∈ set ss ∧ y
∈ set ss ∧ x < y}
    = (length ss ∗ (length ss−1)) / 2
    by auto

  from cons have sss: s ∉ set ss by auto

  from cons have tt: (∀ y∈set (s#ss). s ≤ y) by auto
  with cons  have tt': (∀ y∈set ss. s < y)
  proof −
    from sss have (∀ y∈set ss. s ≠ y) by auto
    with tt show ?thesis by fastforce
  qed

  then have {(x, y) . x = s ∧ y ∈ set ss ∧ x < y}
        = {(x, y) . x = s ∧ y ∈ set ss} by auto
  also have ... = {s}×(set ss) by auto
  finally have {(x, y) . x = s ∧ y ∈ set ss ∧ x < y} = {s}×(set ss) .
  then have card {(x, y) . x = s ∧ y ∈ set ss ∧ x < y}
        = card (set ss) by(auto)
  also from cons distinct_card have ... = length ss by auto
  finally have step: card {(x, y) . x = s ∧ y ∈ set ss ∧ x < y} =
          length ss .

  have uni: {(x, y) . x ∈ set (s # ss) ∧ y ∈ set (s # ss) ∧ x < y}
     = {(x, y) . x ∈ set ss ∧ y ∈ set ss ∧ x < y}
      ∪ {(x, y) . x = s ∧ y ∈ set ss ∧ x < y}
      using tt by auto

  have disj: {(x, y) . x ∈ set ss ∧ y ∈ set ss ∧ x < y}
      ∩ {(x, y) . x = s ∧ y ∈ set ss ∧ x < y} = {}
        using sss by(auto)
  have card {(x, y) . x ∈ set (s # ss) ∧ y ∈ set (s # ss) ∧ x < y}
    = card ({(x, y) . x ∈ set ss ∧ y ∈ set ss ∧ x < y}
```

∪ {(x, y) . x = s ∧ y ∈ set ss ∧ x < y}) **using** *uni* **by** *auto*
also have ... = *card* {(x, y) . x ∈ set ss ∧ y ∈ set ss ∧ x < y}
+ *card* {(x, y) . x = s ∧ y ∈ set ss ∧ x < y}
apply(*rule card_Un_disjoint*)
apply(*rule finite_subset*[**where** *B*=(*set ss*) × (*set ss*)])
apply(*force*)
apply(*simp*)
apply(*rule finite_subset*[**where** *B*={*s*} × (*set ss*)])
apply(*force*)
apply(*simp*)
using *disj* **apply**(*simp*) **done**
also have ... = (*length ss* ∗ (*length ss*−*1*)) / *2*
+ *length ss* **using** *iH step* **by** *auto*
also have ... = (*length ss* ∗ (*length ss*−*1*) + *2*∗*length ss*) / *2* **by** *auto*
also have ... = (*length ss* ∗ (*length ss*−*1*) + *length ss* ∗ *2*) / *2* **by** *auto*
also have ... = (*length ss* ∗ (*length ss*−*1*+*2*)) / *2*
by *simp*
also have ... = (*length ss* ∗ (*length ss*+*1*)) / *2*
using *cons*(*1*) **by** *simp*
also have ... = ((*length ss*+*1*) ∗ *length ss*) / *2* **by** *auto*
also have ... = (*length* (*s*#*ss*) ∗ (*length* (*s*#*ss*)−*1*)) / *2* **by** *auto*
finally show *?case* **by** *auto*
next
case *single* **thus** *?case* **by**(*simp cong*: *conj_cong*)
qed

lemma *factoringlemma_withconstant*:
fixes *A*
and *b*::*real*
and *c*::*real*
assumes *c*: $c \geq 1$
assumes *dist*: ∀ *e*∈*S0*. *distinct e*
assumes *notempty*: ∀ *e*∈*S0*. *length e* > *0*

assumes *pw*: *pairwise A*

assumes *on2*: ∀ *s0*∈*S0*. ∃ *b*≥*0*. ∀ *qs*∈{*x*. *set x* ⊆ *set s0*}. ∀ (*x*,*y*)∈{(*x*,*y*). *x* ∈ *set s0* ∧ *y*∈*set s0* ∧ *x*<*y*}. T_p_*on_rand A* (*Lxy s0* {*x*,*y*}) (*Lxy qs* {*x*,*y*}) ≤ *c* ∗ (T_p_*opt* (*Lxy s0* {*x*,*y*}) (*Lxy qs* {*x*,*y*})) + *b*
assumes *nopaid*: ⋀*is s q*. ∀ ((*free*,*paid*),_) ∈ (*snd A* (*s*, *is*) *q*). *paid*=[]
assumes *4*: ⋀*init qs*. *distinct init* ⟹ *set qs* ⊆ *set init* ⟹ (⋀*x*. *x*<*length qs* ⟹ *finite* (*set_pmf* (*config''* *A qs init x*)))

shows $\forall$ *s0*$\in$*S0*. $\exists$ *b*$\geq$*0*. $\forall$ *qs*$\in${*x*. *set x* $\subseteq$ *set s0*}.
T$_p$*_on_rand A s0 qs* $\leq$ *c* $*$ *real* (*T*$_p$*_opt s0 qs*) + *b*
proof (*standard*, *goal_cases*)
case (*1 init*)
have *d*: *distinct init* **using** *dist 1* **by** *auto*
have *d2*: *init* $\neq$ [] **using** *notempty 1* **by** *auto*

obtain *b* **where** *on3*: $\forall$ *qs*$\in${*x*. *set x* $\subseteq$ *set init*}. $\forall$ (*x*,*y*)$\in${(*x*,*y*). *x* $\in$ *set init* $\wedge$ *y*$\in$*set init* $\wedge$ *x*<*y*}. *T*$_p$*_on_rand A* (*Lxy init* {*x*,*y*}) (*Lxy qs* {*x*,*y*}) $\leq$ *c* $*$ (*T*$_p$*_opt* (*Lxy init* {*x*,*y*}) (*Lxy qs* {*x*,*y*})) + *b*
and *b*: *b*$\geq$*0*
using *on2 1* **by** *auto*

{

fix *qs*
assume *drin*: *set qs* $\subseteq$ *set init*

have *T*$_p$*_on_rand A init qs* =
($\sum$ (*x*,*y*)$\in${(*x*, *y*) . *x* $\in$ *set init* $\wedge$ *y* $\in$ *set init* $\wedge$ *x* < *y*}.
T$_p$*_on_rand A* (*Lxy init* {*x*,*y*}) (*Lxy qs* {*x*, *y*}))
apply(*rule umf_pair*)
apply(*fact*)+
using *4*[*of init qs*] *drin d* **by**(*simp add*: *split_def*)

also have ... $\leq$ ($\sum$ (*x*,*y*)$\in${(*x*,*y*). *x* $\in$ *set init* $\wedge$ *y*$\in$*set init* $\wedge$ *x*<*y*}. *c* $*$ (*T*$_p$*_opt* (*Lxy init* {*x*,*y*}) (*Lxy qs* {*x*,*y*})) + *b*)
apply(*rule sum_mono*)
using *on3 drin* **by**(*simp add*: *split_def*)
also have ... = *c* $*$ ($\sum$ (*x*,*y*)$\in${(*x*,*y*). *x* $\in$ *set init* $\wedge$ *y*$\in$*set init* $\wedge$ *x*<*y*}. *T*$_p$*_opt* (*Lxy init* {*x*,*y*}) (*Lxy qs* {*x*,*y*})) + *b*$*$(((*length init*)$*$(*length init*$-$*1*)) / *2*)
proof $-$

{
fix *S*::'*a list*
assume *dis*: *distinct S*
assume *d2*: *S* $\neq$ []
then have *d3*: *sort S* $\neq$ [] **by** (*metis length_0_conv length_sort*)
have *card* {(*x*,*y*). *x* $\in$ *set S* $\wedge$ *y*$\in$*set S* $\wedge$ *x*<*y*}
= *card* {(*x*,*y*). *x* $\in$ *set* (*sort S*) $\wedge$ *y*$\in$*set* (*sort S*) $\wedge$ *x*<*y*}
by *auto*

```
      also have ... = (length (sort S) ∗ (length (sort S) − 1)) / 2
        apply(rule cardofpairs) using dis d2 d3 by (simp_all)
      finally have card {(x, y) . x ∈ set S ∧ y ∈ set S ∧ x < y} =
            (length (sort S) ∗ (length (sort S) − 1)) / 2 .
    }
     with d d2 have e: card {(x,y). x ∈ set init ∧ y∈set init ∧ x<y} =
((length init)∗(length init−1)) / 2 by auto
    show ?thesis (is (∑(x,y)∈?S. c ∗ (?T x y) + b) = c ∗ ?R + b∗?T2)
    proof −
      have (∑(x,y)∈?S. c ∗ (?T x y) + b) =
            c ∗ (∑(x,y)∈?S. (?T x y)) + (∑(x,y)∈?S. b)
            by(simp add: split_def sum.distrib sum_distrib_left)
      also have ... = c ∗ (∑(x,y)∈?S. (?T x y)) + b∗?T2
        using e by(simp add: split_def)
      finally show ?thesis by(simp add: split_def)
    qed
  qed
  also have ... ≤ c ∗ T_p_opt init qs + (b∗((length init)∗(length init−1))
/ 2)
    proof −
      have (∑(x, y)∈{(x, y) . x ∈ set init ∧
           y ∈ set init ∧ x < y}. T_p_opt (Lxy init {x,y}) (Lxy qs {x, y}))
            ≤  T_p_opt init qs
            using OPT_zerlegen drin d d2 by auto
      then have   real (∑(x, y)∈{(x, y) . x ∈ set init ∧
           y ∈ set init ∧ x < y}. T_p_opt (Lxy init {x,y}) (Lxy qs {x, y}))
            ≤    (T_p_opt init qs)
               by linarith
      with c show ?thesis by(auto simp: split_def)
    qed
   finally have f: T_p_on_rand A init qs ≤ c ∗ real (T_p_opt init qs) +
(b∗((length init)∗(length init−1)) / 2) .
  } note all=this
  show ?case unfolding compet_def
    apply(auto)
      apply(rule exI[where x=(b∗((length init)∗(length init−1)) / 2)])
      apply(safe)
        using notempty 1 b apply simp
        using all b by simp
qed

lemma factoringlemma_withconstant':
   fixes A
        and b::real
```

and *c*::*real*
assumes *c*: $c \geq 1$
assumes *dist*: ∀ *e*∈*S0*. *distinct e*
assumes *notempty*: ∀ *e*∈*S0*. *length e* > *0*

assumes *pw*: *pairwise A*

assumes *on2*: ∀ *s0*∈*S0*. ∃ *b*≥*0*. ∀ *qs*∈{*x*. *set x* ⊆ *set s0*}. ∀ (*x*,*y*)∈{(*x*,*y*). *x* ∈ *set s0* ∧ *y*∈*set s0* ∧ *x*<*y*}. T_p_*on*_*rand A* (*Lxy s0* {*x*,*y*}) (*Lxy qs* {*x*,*y*}) ≤ *c* ∗ (T_p_*opt* (*Lxy s0* {*x*,*y*}) (*Lxy qs* {*x*,*y*})) + *b*
assumes *nopaid*: ⋀*is s q*. ∀ ((*free*,*paid*),_) ∈ (*snd A* (*s*, *is*) *q*). *paid*=[]
assumes *4*: ⋀*init qs*. *distinct init* ⟹ *set qs* ⊆ *set init* ⟹ (⋀*x*. *x*<*length qs* ⟹ *finite* (*set*_*pmf* (*config*′′ *A qs init x*)))

shows *compet*_*rand A c S0*
unfolding *compet*_*rand*_*def static*_*def* **using** *factoringlemma*_*withconstant*[*OF assms*] **by** *simp*

end

15 TS: another 2-competitive Algorithm

theory *TS*
imports
OPT2
*Phase*_*Partitioning*
*Move*_*to*_*Front*
*List*_*Factoring*
*RExp*_*Var*
begin

15.1 Definition of TS

definition *TS*_*step*_*d* **where**
*TS*_*step*_*d s q* = ((
(
let li = *index* (*snd s*) *q in*
(*if li* = *length* (*snd s*) *then 0* — requested for first time
else (*let sincelast* = *take li* (*snd s*)
in (*let S*={*x*. *x* < *q in* (*fst s*) ∧ *count*_*list sincelast x* ≤ *1*}
in
(*if S*={} *then 0*
else

```
            (index (fst s) q) − Min ( (index (fst s)) ‘ S)))
          )
        )
      )
      ,[]), q#(snd s))
```

definition *rTS* :: *nat list* ⇒ (*nat*,*nat list*) *alg_on* **where** *rTS h* = ((λ*s*. *h*), *TS_step_d*)

fun *TSstep* **where**

```
  TSstep qs n (is,s)
     = ((qs!n)#is,
        step s (qs!n) ((
          let li = index is (qs!n) in
          (if li = length is then 0 — requested for first time
          else (let sincelast = take li is
            in (let S={x. x < (qs!n) in s ∧ count_list sincelast x ≤ 1}
              in
               (if S={} then 0
               else
                (index s (qs!n)) − Min ( (index s) ‘ S)))
              )
          )
          ),[]))
```

lemma *TSnopaid*: (*snd* (*fst* (*snd* (*rTS initH*) *is q*))) = []
unfolding *rTS_def* **by**(*simp add*: *TS_step_d_def*)

abbreviation *TSdet* **where**
TSdet init initH qs n == *config* (*rTS initH*) *init* (*take n qs*)

lemma *TSdet_Suc*: *Suc n* ≤ *length qs* ⟹ *TSdet init initH qs* (*Suc n*) = *Step* (*rTS initH*) (*TSdet init initH qs n*) (*qs*!*n*)
by(*simp add*: *take_Suc_conv_app_nth config_snoc*)

definition *s_TS* **where** *s_TS init initH qs n* = *fst* (*TSdet init initH qs n*)

lemma *sndTSdet*: *n*≤*length xs* ⟹ *snd* (*TSdet init initH xs n*) = *rev* (*take*

n xs) @ initH
apply(induct n)
apply(simp add: rTS_def)
by(simp add: split_def TS_step_d_def take_Suc_conv_app_nth config′_snoc Step_def rTS_def)

15.2 Behaviour of TS on lists of length 2

lemma
fixes hs x y
assumes x≠y
shows oneTS_step : TS_step_d ([x, y], x#y#hs) y = ((1, []), y # x # y # hs)
and oneTS_stepyyy: TS_step_d ([x, y], y#x#hs) y = ((Suc 0, []), y#y#x#hs)
and oneTS_stepx: TS_step_d ([x, y], x#x#hs) y = ((0, []), y # x # x # hs)
and oneTS_stepy: TS_step_d ([x, y], []) y = ((0, []), [y])
and oneTS_stepxy: TS_step_d ([x, y], [x]) y = ((0, []), [y, x])
and oneTS_stepyy: TS_step_d ([x, y], [y]) y = ((Suc 0, []), [y, y])
and oneTS_stepyx: TS_step_d ([x, y], hs) x = ((0, []), x # hs)
using assms **by**(auto simp add: step_def mtf2_def swap_def TS_step_d_def before_in_def)

lemmas oneTS_steps = oneTS_stepx oneTS_stepxy oneTS_stepyx oneTS_stepy oneTS_stepyy oneTS_stepyyy oneTS_step

15.3 Analysis of the Phases

definition TS_inv c x i ≡ (∃ hs. c = return_pmf ((if x=hd i then i else rev i),[x,x]@hs))
∨ c = return_pmf ((if x=hd i then i else rev i),[])

lemma TS_inv_sym: a≠b ⟹ {a,b}={x,y} ⟹ z∈{x,y} ⟹ TS_inv c z [a,b] = TS_inv c z [x,y]
unfolding TS_inv_def **by** auto

abbreviation TS_inv′ s x i == TS_inv (return_pmf s) x i

lemma TS_inv′_det: TS_inv′ s x i = ((∃ hs. s = ((if x=hd i then i else rev i),[x,x]@hs))
∨ s = ((if x=hd i then i else rev i),[]))
unfolding TS_inv_def **by** auto

lemma *TS_inv'_det2*: *TS_inv'* (*s*,*h*) *x i* = (∃ *hs*. (*s*,*h*) = ((*if x*=*hd i then i else rev i*),[*x*,*x*]@*hs*))
∨ (*s*,*h*) = ((*if x*=*hd i then i else rev i*),[])
unfolding *TS_inv_def* **by** *auto*

15.3.1 (yx)*?

lemma *TS_yx'*: **assumes** *x* ≠ *y qs* ∈ *lang* (*Star*(*Times* (*Atom y*) (*Atom x*)))
∃ *hs*. *h*=[*x*,*y*]@*hs*
shows *T_on'* (*rTS h0*) ([*x*,*y*],*h*) (*qs*@*r*) = *length qs* + *T_on'* (*rTS h0*) ([*x*,*y*],((*rev qs*) @*h*)) *r*
∧ (∃ *hs*. ((*rev qs*) @*h*) = [*x*, *y*] @ *hs*)
∧ *config'* (*rTS h0*) ([*x*, *y*],*h*) *qs* = ([*x*,*y*],*rev qs* @ *h*)
proof −
from *assms* **have** *qs* ∈ *star* ({[*y*]} @@ {[*x*]}) **by** (*simp*)
from *this assms*(*3*) **show** *?thesis*
proof (*induct qs arbitrary*: *h rule*: *star_induct*)
case *Nil*
then show *?case* **by**(*simp add*: *rTS_def*)
next
case (*append u v*)
then have *uyx*: *u* = [*y*,*x*] **by** *auto*
from *append* **obtain** *hs* **where** *a*: *h* = [*x*,*y*]@*hs* **by** *blast*

have *T_on'* (*rTS h0*) ([*x*, *y*], (*rev u* @ *h*)) (*v* @ *r*) = *length v* + *T_on'* (*rTS h0*) ([*x*, *y*], *rev v* @ (*rev u* @ *h*)) *r*
∧ (∃ *hs*. *rev v* @ (*rev u* @ *h*) = [*x*, *y*] @ *hs*)
∧ *config'* (*rTS h0*) ([*x*, *y*], (*rev u* @ *h*)) *v* = ([*x*, *y*], *rev v* @ (*rev u* @ *h*))
apply(*simp only*: *uyx*) **apply**(*rule append*(*3*)) **by** *simp*
then have *yy*: *T_on'* (*rTS h0*) ([*x*, *y*], (*rev u* @ *h*)) (*v* @ *r*) = *length v* + *T_on'* (*rTS h0*) ([*x*, *y*], *rev v* @ (*rev u* @ *h*)) *r*
and *history*: (∃ *hs*. *rev v* @ (*rev u* @ *h*) = [*x*, *y*] @ *hs*)
and *state*: *config'* (*rTS h0*) ([*x*, *y*], (*rev u* @ *h*)) *v* = ([*x*, *y*], *rev v* @ (*rev u* @ *h*)) **by** *auto*

have *s0*: *s_TS* [*x*, *y*] *h* [*y*, *x*] *0* = [*x*,*y*] **unfolding** *s_TS_def* **by**(*simp*)

from *assms*(*1*) **have** *hahah*: {*xa*. *xa* < *y in* [*x*, *y*] ∧ *count_list* [*x*] *xa* ≤ *1*} = {*x*}
unfolding *before_in_def* **by** *auto*

have *config′* (*rTS h0*) ([*x*, *y*],*h*) *u* = ([*x*, *y*], *x* # *y* # *x* # *y* # *hs*)
apply(*simp add*: *split_def rTS_def uyx a*)
using *assms*(*1*) **by**(*auto simp add*: *Step_def oneTS_steps step_def mtf2_def swap_def*)

then have *s2*: *config′* (*rTS h0*) ([*x*, *y*],*h*) *u* = ([*x*, *y*], ((*rev u*) @ *h*))
unfolding *a uyx* **by** *simp*

have *config′* (*rTS h0*) ([*x*, *y*], *h*) (*u* @ *v*) =
config′ (*rTS h0*) (*Partial_Cost_Model.config′* (*rTS h0*) ([*x*, *y*], *h*) *u*) *v* **by** (*rule config′_append2*)
also
have ... = *config′* (*rTS h0*) ([*x*, *y*], ((*rev u*) @ *h*)) *v* **by**(*simp only*: *s2*)
also
have ... = ([*x*, *y*], *rev* (*u* @ *v*) @ *h*) **by** (*simp add*: *state*)
finally
have *alles*: *config′* (*rTS h0*) ([*x*, *y*], *h*) (*u* @ *v*) = ([*x*, *y*], *rev* (*u* @ *v*) @ *h*) **.**

have *ta*: *T_on′* (*rTS h0*) ([*x*,*y*],*h*) *u* = *2*
unfolding *rTS_def uyx a* **apply**(*simp only*: *T_on′.simps*(*2*))
using *assms*(*1*) **apply**(*auto simp add*: *Step_def step_def mtf2_def swap_def oneTS_steps*)
by(*simp add*: t_p*_def*)

have *T_on′* (*rTS h0*) ([*x*,*y*],*h*) ((*u* @ *v*) @ *r*)
= *T_on′* (*rTS h0*) ([*x*,*y*],*h*) (*u* @ (*v* @ *r*)) **by** *auto*
also have ...
= *T_on′* (*rTS h0*) ([*x*,*y*],*h*) *u*
+ *T_on′* (*rTS h0*) (*config′* (*rTS h0*) ([*x*, *y*],*h*) *u*) (*v* @ *r*)
by(*rule T_on′_append*)
also have ... = *T_on′* (*rTS h0*) ([*x*,*y*],*h*) *u*
+ *T_on′* (*rTS h0*) ([*x*, *y*],(*rev u* @ *h*)) (*v* @ *r*) **by**(*simp only*: *s2*)
also have ... = *T_on′* (*rTS h0*) ([*x*,*y*],*h*) *u* + *length v* + *T_on′* (*rTS h0*) ([*x*, *y*], *rev v* @ (*rev u* @ *h*)) *r* **by**(*simp only*: *yy*)
also have ... = *2* + *length v* + *T_on′* (*rTS h0*) ([*x*, *y*], *rev v* @ (*rev u* @ *h*)) *r* **by**(*simp only*: *ta*)
also have ... = *length* (*u* @ *v*) + *T_on′* (*rTS h0*) ([*x*, *y*], *rev v* @ (*rev*

u @ *h*)) *r* **using** *uyx* **by** *auto*
also have ... = *length* (*u* @ *v*) + *T_on′* (*rTS h0*) ([*x*, *y*], (*rev* (*u* @ *v*) @ *h*)) *r* **by** *auto*
finally show *?case* **using** *history alles* **by** *simp*
qed
qed

15.3.2 ?x

lemma *TS_x′*: *T_on′* (*rTS h0*) ([*x*,*y*],*h*) [*x*] = *0* ∧ *config′* (*rTS h0*) ([*x*, *y*],*h*) [*x*] = ([*x*,*y*], *rev* [*x*] @ *h*)
by(*auto simp add*: t_p*_def rTS_def TS_step_d_def Step_def step_def*)

15.3.3 ?yy

lemma *TS_yy′*: **assumes** *x* ≠ *y* ∃ *hs*. *h* = [*x*, *y*] @ *hs*
shows *T_on′* (*rTS h0*) ([*x*,*y*],*h*) [*y*, *y*] = *1* *config′* (*rTS h0*) ([*x*, *y*],*h*) [*y*,*y*] = ([*y*,*x*],*rev* [*y*,*y*] @ *h*)
proof −
from *assms* **obtain** *hs* **where** *a*: *h* = [*x*,*y*]@*hs* **by** *blast*

from *a* **show** *T_on′* (*rTS h0*) ([*x*,*y*],*h*) [*y*, *y*] = *1*
unfolding *rTS_def*
using *assms*(*1*) **apply**(*auto simp add*: *oneTS_steps Step_def step_def mtf2_def swap_def*)
by(*simp add*: t_p*_def*)

show *config′* (*rTS h0*) ([*x*, *y*],*h*) [*y*,*y*] = ([*y*,*x*],*rev* [*y*,*y*] @ *h*)
unfolding *rTS_def a* **using** *assms*(*1*)
by(*simp add*: *Step_def oneTS_steps step_def mtf2_def swap_def*)
qed

15.3.4 yx(yx)*?

lemma *TS_yxyx′*: **assumes** [*simp*]: *x* ≠ *y* **and** *qs* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
(∃ *hs*. *h*=[*x*,*x*]@*hs*) ∨ *index h y* = *length h*
shows *T_on′* (*rTS h0*) ([*x*,*y*],*h*) (*qs*@*r*) = *length qs* − *1* + *T_on′* (*rTS h0*) ([*x*,*y*],*rev qs* @ *h*) *r*
∧ (∃ *hs*. (*rev qs* @ *h*) = [*x*, *y*] @ *hs*)
∧ *config′* (*rTS h0*) ([*x*, *y*],*h*) *qs* = ([*x*,*y*], *rev qs* @ *h*)
proof −
obtain *u v* **where** *uu*: *u* ∈ *lang* (*Times* (*Atom y*) (*Atom x*))
and *vv*: *v* ∈ *lang* (*seq*[*Star*(*Times* (*Atom y*) (*Atom x*))])
and *qsuv*: *qs* = *u* @ *v*

using *assms*(*2*)
by (*auto simp*: *conc_def*)
from *uu* **have** *uyx*: *u* = [*y*,*x*] **by**(*auto*)

from *qsuv uyx* **have** *vqs*: *length v* = *length qs* − *2* **by** *auto*
from *qsuv uyx* **have** *vqs2*: *length v* + *1* = *length qs* − *1* **by** *auto*

have *firststep*: *TS_step_d* ([*x*, *y*], *h*) *y* = ((*0*, []), *y* # *h*)
proof (*cases index h y* = *length h*)
case *True*
then show *?thesis* **unfolding** *TS_step_d_def* **by**(*simp*)
next
case *False*
with *assms*(*3*) **obtain** *hs* **where** *a*: *h* = [*x*,*x*]@*hs* **by** *auto*
then show *?thesis* **by**(*simp add*: *oneTS_steps*)
qed

have *s2*: *config′* (*rTS h0*) ([*x*,*y*],*h*) *u* = ([*x*, *y*], *x* # *y* # *h*)
unfolding *rTS_def uyx* **apply** *simp*
unfolding *Step_def* **by**(*simp add*: *firststep step_def oneTS_steps*)

have *ta*: *T_on′* (*rTS h0*) ([*x*,*y*],*h*) *u* = *1*
unfolding *rTS_def uyx*
apply(*simp*)
apply(*simp add*: *firststep*)
unfolding *Step_def*
using *assms*(*1*) **by** (*simp add*: *firststep step_def oneTS_steps*
t_p_*def*)

have *ttt*:
T_on′ (*rTS h0*) ([*x*,*y*],*rev u* @ *h*) (*v*@*r*) = *length v* + *T_on′* (*rTS h0*)
([*x*,*y*],((*rev v*) @(*rev u* @ *h*))) *r*
∧ (∃ *hs*. ((*rev v*) @(*rev u* @ *h*)) = [*x*, *y*] @ *hs*)
∧ *config′* (*rTS h0*) ([*x*, *y*],(*rev u* @ *h*)) *v* = ([*x*,*y*],*rev v* @ (*rev u* @ *h*))
apply(*rule TS_yx′*)
apply(*fact*)
using *vv* **apply**(*simp*)
using *uyx* **by**(*simp*)
then have *tat*: *T_on′* (*rTS h0*) ([*x*,*y*], *x* # *y* # *h*) (*v*@*r*) =
length v + *T_on′* (*rTS h0*) ([*x*,*y*],*rev qs* @ *h*) *r*
and *history*: (∃ *hs*. (*rev qs* @ *h*) = [*x*, *y*] @ *hs*)
and *state*: *config′* (*rTS h0*) ([*x*, *y*], *x* # *y* # *h*) *v* = ([*x*,*y*],*rev qs* @
h) **using** *qsuv uyx*
by *auto*

have *config′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = *config′* (*rTS h0*) (*config′* (*rTS h0*) ([*x*, *y*], *h*) *u*) *v*
unfolding *qsuv* **by** (*rule config′_append2*)
also
have ... = ([*x*, *y*], *rev qs* @ *h*) **by**(*simp add*: *s2 state*)
finally
have *his*: *config′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = ([*x*, *y*], *rev qs* @ *h*) **.**

have *T_on′* (*rTS h0*) ([*x*,*y*],*h*) (*qs*@*r*) = *T_on′* (*rTS h0*) ([*x*,*y*],*h*) (*u* @ *v* @ *r*) **using** *qsuv* **by** *auto*
also have ...
= *T_on′* (*rTS h0*) ([*x*,*y*],*h*) *u* + *T_on′* (*rTS h0*) (*config′* (*rTS h0*) ([*x*,*y*],*h*) *u*) (*v* @ *r*)
by(*rule T_on′_append*)
also have ... = *T_on′* (*rTS h0*) ([*x*,*y*],*h*) *u* + *T_on′* (*rTS h0*) ([*x*, *y*], *x* # *y* # *h*) (*v* @ *r*) **by**(*simp only*: *s2*)
also have ... = *T_on′* (*rTS h0*) ([*x*,*y*],*h*) *u* + *length v* + *T_on′* (*rTS h0*) ([*x*,*y*],*rev qs* @ *h*) *r* **by** (*simp only*: *tat*)
also have ... = *1* + *length v* + *T_on′* (*rTS h0*) ([*x*,*y*],*rev qs* @ *h*) *r* **by**(*simp only*: *ta*)
also have ... = *length qs* − *1* + *T_on′* (*rTS h0*) ([*x*,*y*],*rev qs* @ *h*) *r* **using** *vqs2* **by** *auto*
finally show *?thesis*
apply(*safe*)
using *history* **apply**(*simp*)
using *his* **by** *auto*
qed

lemma *TS_xr′*: **assumes** *x* ≠ *y qs* ∈ *lang* (*Plus* (*Atom x*) *One*)
h = [] ∨ (∃ *hs*. *h* = [*x*, *x*] @ *hs*)
shows *T_on′* (*rTS h0*) ([*x*,*y*],*h*) (*qs*@*r*) = *T_on′* (*rTS h0*) ([*x*,*y*],*rev qs*@*h*) *r*
((∃ *hs*. (*rev qs* @ *h*) = [*x*, *x*] @ *hs*) ∨ (*rev qs* @ *h*) = [*x*] ∨ (*rev qs* @ *h*)=[])
config′ (*rTS h0*) ([*x*,*y*],*h*) (*qs*@*r*) = *config′* (*rTS h0*) ([*x*,*y*],*rev qs* @ *h*) *r*
using *assms*
by (*auto simp add*: *T_on′_append Step_def rTS_def TS_step_d_def step_def* t_p*_def*)

15.3.5 (x+1)yx(yx)*yy

lemma *ts_b'*: **assumes** $x \neq y$
$v \in$ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
($\exists$ *hs*. h = [x, x] @ *hs*) $\vee$ h = [x] $\vee$ h = []
shows *T_on'* (*rTS h0*) ([x, y], h) v = (*length* v − *2*)
$\wedge$ ($\exists$ *hs*. (*rev* v @ h) = [y,y]@*hs*) $\wedge$ *config'* (*rTS h0*) ([x,y], h) v = ([y,x], *rev* v @ h)
proof −
from *assms* **have** *lenvmod*: *length* v *mod 2* = *0* **apply**(*simp*)
proof −
assume $v \in$ ({[y]} @@ {[x]}) @@ *star* ({[y]} @@ {[x]}) @@ {[y]} @@ {[y]}
then obtain p q r **where** *pqr*: v=p@q@r **and** $p\in$({[y]} @@ {[x]})
and q: $q \in$ *star* ({[y]} @@ {[x]}) **and** $r \in$ {[y]} @@ {[y]} **by** (*metis concE*)
then have p = [y,x] r=[y,y] **by** *auto*
with *pqr* **have** a: *length* v = *4*+*length* q **by** *auto*

from q **have** b: *length* q *mod 2* = *0*
apply(*induct* q *rule*: *star_induct*) **by** (*auto*)
from a b **show** *?thesis* **by** *auto*
qed

with *assms*(*1*,*3*) **have** *fall*: ($\exists$ *hs*. h = [x, x] @ *hs*) $\vee$ *index* h y = *length* h
by(*auto*)

from *assms*(*2*) **have** $v \in$ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
@@ *lang* (*seq*[*Atom y*, *Atom y*]) **by** (*auto simp*: *conc_def*)
then obtain a b **where** *aa*: $a \in$ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
and $b \in$ *lang* (*seq*[*Atom y*, *Atom y*])
and *vab*: v = a @ b
by(*erule concE*)
then have *bb*: b=[y,y] **by** *auto*
from *aa* **have** *lena*: *length* a > *0* **by** *auto*

from *TS_yxyx'*[*OF assms*(*1*) *aa fall*] **have** *stars*: *T_on'* (*rTS h0*) ([x, y], h) (a @ b) =
length a − *1* + *T_on'* (*rTS h0*) ([x, y], *rev* a @ h) b

and *history*: (∃ *hs*. *rev a* @ *h* = [*x*, *y*] @ *hs*)
and *state*: *config′* (*rTS h0*) ([*x*, *y*], *h*) *a* = ([*x*,*y*],*rev a* @ *h*) **by** *auto*

have *suffix*: *T_on′* (*rTS h0*) ([*x*, *y*], *rev a* @ *h*) *b* = *1*
and *jajajaj*: *config′* (*rTS h0*) ([*x*, *y*],*rev a* @ *h*) *b* = ([*y*,*x*],*rev b* @ *rev a* @ *h*) **unfolding** *bb*
using *TS_yy′ history assms*(*1*) **by** *auto*

from *stars suffix* **have** *T_on′* (*rTS h0*) ([*x*, *y*], *h*) (*a* @ *b*) = *length a* **using** *lena* **by** *auto*
then have *whatineed*: *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *v* = (*length v* − *2*) **using** *vab bb* **by** *auto*

have *grgr*:*config′* (*rTS h0*) ([*x*, *y*], *h*) *v* = ([*y*, *x*], *rev v* @ *h*)
unfolding *vab*
apply(*simp only*: *config′_append2 state jajajaj*) **by** *simp*

from *history* **obtain** *hs′* **where** *rev a* @ *h* = [*x*, *y*] @ *hs′* **by** *auto*
then obtain *hs2* **where** *reva*: *rev a* @ *h* = *x* # *hs2* **by** *auto*

show *?thesis* **using** *whatineed grgr*
by(*auto simp add*: *reva vab bb*)
qed

lemma *TS_b′1*: **assumes** *x* ≠ *y h* = [] ∨ (∃ *hs*. *h* = [*x*, *x*] @ *hs*)
qs ∈ *lang* (*seq* [*Atom y*, *Atom x*, *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = (*length qs* − *2*)
∧ *TS_inv′* (*config′* (*rTS h0*) ([*x*, *y*], *h*) *qs*) (*last qs*) [*x*,*y*]
proof −
have *f*: *qs* ∈ *lang* (*seq* [*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
using *assms*(*3*) **by**(*simp add*: *conc_assoc*)

from *ts_b′*[*OF assms*(*1*) *f*] *assms*(*2*) **have**
T_star: *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = *length qs* − *2*
and *inv1*: *config′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = ([*y*, *x*], *rev qs* @ *h*)
and *inv2*: (∃ *hs*. *rev qs* @ *h* = [*y*, *y*] @ *hs*) **by** *auto*

from *T_star* **have** *TS*: *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = (*length qs* − *2*) **by** *metis*

have *lqs*: *last qs* = *y* **using** *assms*(*3*) **by** *force*

from *inv1* **have** *inv*: *TS_inv′* (*config′* (*rTS h0*) ([*x*, *y*], *h*) *qs*) (*last qs*) [*x*, *y*]
apply(*simp add*: *lqs*)
apply(*subst TS_inv′_det*)
using *assms*(*2*) *inv2* **by**(*simp*)

show *?thesis* **unfolding** *TS*
apply(*safe*)
by(*fact inv*)
qed

lemma *TS_b1′′*: **assumes**
$x \neq y$ $\{x, y\} = \{x0, y0\}$ *TS_inv s x* [*x0*, *y0*]
set qs $\subseteq \{x, y\}$
qs ∈ *lang* (*seq* [*Atom y*, *Atom x*, *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows *TS_inv* (*config′_rand* (*embed* (*rTS h0*)) *s qs*) (*last qs*) [*x0*, *y0*]
∧ *T_on_rand′* (*embed* (*rTS h0*)) *s qs* = (*length qs* − *2*)
proof −
from *assms*(*1*,*2*) **have** *kas*: (*x0*=*x* ∧ *y0*=*y*) ∨ (*y0*=*x* ∧ *x0*=*y*) **by**(*auto*)
then obtain *h* **where** *S*: *s* = *return_pmf* ([*x*,*y*],*h*) **and** *h*: *h* = [] ∨ (∃ *hs*. *h* = [*x*, *x*] @ *hs*)
apply(*rule disjE*) **using** *assms*(*1*,*3*) **unfolding** *TS_inv_def* **by**(*auto*)

have *l*: *qs* ≠ [] **using** *assms* **by** *auto*
{
fix *x y qs h0*
fix *h*:: *nat list*
assume *A*: *x* ≠ *y*
and *B*: *qs* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
and *C*: *h* = [] ∨ (∃ *hs*. *h* = [*x*, *x*] @ *hs*)

then have *C′*: (∃ *hs*. *h* = [*x*, *x*] @ *hs*) ∨ *h* = [*x*] ∨ *h* = [] **by** *blast*
from *B* **have** *lqs*: *last qs* = *y* **using** *assms*(*5*) **by**(*auto simp add*: *conc_def*)

have *TS_inv* (*config′_rand* (*embed* (*rTS h0*)) (*return_pmf* ([*x*, *y*], *h*)) *qs*) (*last qs*) [*x*, *y*] ∧

T_on_rand′ (*embed* (*rTS h0*)) (*return_pmf* ([*x*, *y*], *h*)) *qs* = *length qs* − *2*
apply(*simp only*: *T_on′_embed*[*symmetric*] *config′_embed*)
using *ts_b′*[*OF A B C′*] *A lqs* **unfolding** *TS_inv′_det* **by** *auto*
} **note** *b1*=*this*

show *?thesis* **unfolding** *S*
using *kas* **apply**(*rule disjE*)
apply(*simp only*:)
apply(*rule b1*)
using *assms* **apply**(*simp*)
using *assms* **apply**(*simp add*: *conc_assoc*)
using *h* **apply**(*simp*)
apply(*simp only*:)

apply(*subst TS_inv_sym*[*of y x x y*])
using *assms*(*1*) **apply**(*simp*)
apply(*blast*)
defer
apply(*rule b1*)
using *assms* **apply**(*simp*)
using *assms* **apply**(*simp add*: *conc_assoc*)
using *h* **apply**(*simp*)
using *last_in_set l assms*(*4*) **by** *blast*
qed

lemma *ts_b2′*: **assumes** $x \neq y$
$qs \in$ *lang* (*seq*[*Atom x*, *Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
($\exists$ *hs*. $h = [x, x]$ @ *hs*) $\vee$ $h = []$
shows *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = (*length qs* − *3*)
$\wedge$ *config′* (*rTS h0*) ([*x*,*y*], *h*) *qs* = ([*y*,*x*],*rev qs*@*h*) $\wedge$ ($\exists$ *hs*. (*rev qs* @ *h*) = [*y*,*y*]@*hs*)
proof −
from *assms*(*2*) **obtain** *v* **where** *qs*: *qs* = [*x*]@*v*
and *V*: *v*∈*lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
by(*auto simp add*: *conc_assoc*)

from *assms*(*3*) **have** *3*: ($\exists$ *hs*. *x#h* = [*x*, *x*] @ *hs*) $\vee$ *x#h* = [*x*] $\vee$ *x#h* = [] **by** *auto*

from *ts_b′*[*OF assms*(*1*) *V 3*]
have *T*: *T_on′* (*rTS h0*) ([*x*, *y*], *x#h*) *v* = *length v* − *2*
and *C*: *config′* (*rTS h0*) ([*x*, *y*], *x#h*) *v* = ([*y*, *x*], *rev v* @ *x#h*)
and *H*: (∃ *hs*. *rev v* @ *x#h* = [*y*, *y*] @ *hs*) **by** *auto*

have *t*: t_p [*x*, *y*] *x* (*fst* (*snd* (*rTS h0*) ([*x*, *y*], *h*) *x*)) = *0*
by (*simp add*: *step_def rTS_def TS_step_d_def* t_p*_def*)
have *c*: *Partial_Cost_Model.Step* (*rTS h0*) ([*x*, *y*], *h*) *x*
= ([*x*,*y*], *x#h*) **by** (*simp add*: *Step_def rTS_def TS_step_d_def step_def*)

show *?thesis*
unfolding *qs* **apply**(*safe*)
apply(*simp add*: *T_on′_append T c t*)
apply(*simp add*: *config′_rand_append C c*)
using *H* **by** *simp*
qed

lemma *TS_b2″*: **assumes**
x ≠ *y* {*x*, *y*} = {*x0*, *y0*} *TS_inv s x* [*x0*, *y0*]
set qs ⊆ {*x*, *y*}
qs ∈ *lang* (*seq* [*Atom x*, *Atom y*, *Atom x*, *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows *TS_inv* (*config′_rand* (*embed* (*rTS h0*)) *s qs*) (*last qs*) [*x0*, *y0*]
∧ *T_on_rand′* (*embed* (*rTS h0*)) *s qs* = (*length qs* − *3*)
proof −
from *assms*(*1*,*2*) **have** *kas*: (*x0*=*x* ∧ *y0*=*y*) ∨ (*y0*=*x* ∧ *x0*=*y*) **by**(*auto*)
then obtain *h* **where** *S*: *s* = *return_pmf* ([*x*,*y*],*h*) **and** *h*: *h* = [] ∨ (∃ *hs*. *h* = [*x*, *x*] @ *hs*)
apply(*rule disjE*) **using** *assms*(*1*,*3*) **unfolding** *TS_inv_def* **by**(*auto*)

have *l*: *qs* ≠ [] **using** *assms* **by** *auto*
{
fix *x y qs h0*
fix *h*:: *nat list*
assume *A*: *x* ≠ *y*
and *B*: *qs* ∈ *lang* (*seq*[*Atom x*, *Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
and *C*: *h* = [] ∨ (∃ *hs*. *h* = [*x*, *x*] @ *hs*)

from *B* **have** *lqs*: *last qs* = *y* **using** *assms*(*5*) **by**(*auto simp add*: *conc_def*)

from *C* **have** *C'*: (∃ *hs*. *h* = [*x*, *x*] @ *hs*) ∨ *h* = [] **by** *blast*

have *TS_inv* (*config'_rand* (*embed* (*rTS h0*)) (*return_pmf* ([*x*, *y*], *h*)) *qs*) (*last qs*) [*x*, *y*] ∧
T_on_rand' (*embed* (*rTS h0*)) (*return_pmf* ([*x*, *y*], *h*)) *qs* = *length qs* − *3*
apply(*simp only*: *T_on'_embed*[*symmetric*] *config'_embed*)
using *ts_b2'*[*OF A B C'*] *A lqs* **unfolding** *TS_inv'_det* **by** *auto*
} **note** *b2=this*

show *?thesis* **unfolding** *S*
using *kas* **apply**(*rule disjE*)
apply(*simp only*:)
apply(*rule b2*)
using *assms* **apply**(*simp*)
using *assms* **apply**(*simp add*: *conc_assoc*)
using *h* **apply**(*simp*)
apply(*simp only*:)

apply(*subst TS_inv_sym*[*of y x x y*])
using *assms*(*1*) **apply**(*simp*)
apply(*blast*)
defer
apply(*rule b2*)
using *assms* **apply**(*simp*)
using *assms* **apply**(*simp add*: *conc_assoc*)
using *h* **apply**(*simp*)
using *last_in_set l assms*(*4*) **by** *blast*
qed

lemma *TS_b'*: **assumes** *x* ≠ *y h* = [] ∨ (∃ *hs*. *h* = [*x*, *x*] @ *hs*)
qs ∈ *lang* (*seq* [*Plus* (*Atom x*) *rexp.One*, *Atom y*, *Atom x*, *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows *T_on'* (*rTS h0*) ([*x*, *y*], *h*) *qs*
≤ *2* ∗ T_p [*x*, *y*] *qs* (*OPT2 qs* [*x*, *y*]) ∧ *TS_inv'* (*config'* (*rTS h0*) ([*x*, *y*], *h*) *qs*) (*last qs*) [*x*,*y*]
proof −
obtain *u v* **where** *uu*: *u* ∈ *lang* (*Plus* (*Atom x*) *One*)
and *vv*: *v* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
and *qsuv*: *qs* = *u* @ *v*

using *assms*(*3*)
by (*auto simp*: *conc_def*)

from *TS_xr′*[*OF assms*(*1*) *uu assms*(*2*)] **have**
T_pre: *T_on′* (*rTS h0*) ([*x*, *y*], *h*) (*u* @ *v*) =
T_on′ (*rTS h0*) ([*x*, *y*], *rev u* @ *h*) *v*
and *fall′*: (∃ *hs*. (*rev u* @ *h*) = [*x*, *x*] @ *hs*) ∨ (*rev u* @ *h*) = [*x*] ∨ (*rev u* @ *h*)=[]
and *conf*: *config′* (*rTS h0*) ([*x*,*y*],*h*) (*u*@*v*) = *config′* (*rTS h0*) ([*x*,*y*],*rev u* @ *h*) *v*
by *auto*

with *assms uu* **have** *fall*: (∃ *hs*. (*rev u* @ *h*) = [*x*, *x*] @ *hs*) ∨ *index* (*rev u* @ *h*) *y* = *length* (*rev u* @ *h*)
by(*auto*)

from *ts_b′*[*OF assms*(*1*) *vv fall′*] **have**
T_star: *T_on′* (*rTS h0*) ([*x*, *y*], *rev u* @ *h*) *v* = *length v* − *2*
and *inv1*: *config′* (*rTS h0*) ([*x*, *y*], *rev u* @ *h*) *v* = ([*y*, *x*], *rev v* @ *rev u* @ *h*)
and *inv2*: (∃ *hs*. *rev v* @ *rev u* @ *h* = [*y*, *y*] @ *hs*) **by** *auto*

from *T_pre T_star qsuv* **have** *TS*: *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = (*length v* − *2*) **by** *metis*

from *uu* **have** *uuu*: *u*=[] ∨ *u*=[*x*] **by** *auto*
from *vv* **have** *vvv*: *v* ∈ *lang* (*seq*
[*Atom y*, *Atom x*, *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*]) **by**(*auto simp*: *conc_def*)
have *OPT*: T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) = (*length v*) *div 2* **apply**(*rule OPT2_B*) **by**(*fact*)+

have *lqs*: *last qs* = *y* **using** *assms*(*3*) **by** *force*

have *config′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = ([*y*, *x*], *rev qs* @ *h*)
unfolding *qsuv conf inv1* **by** *simp*

then have *inv*: *TS_inv′* (*config′* (*rTS h0*) ([*x*, *y*], *h*) *qs*) (*last qs*) [*x*, *y*]
apply(*simp add*: *lqs*)
apply(*subst TS_inv′_det*)
using *assms*(*2*) *inv2 qsuv* **by**(*simp*)

show *?thesis* **unfolding** *TS OPT*
  apply(*safe*)
    apply(*simp*)
    by(*fact inv*)
qed

15.3.6 (x+1)yy

lemma *ts_a'*: **assumes** $x \neq y$ $qs \in lang$ (*seq* [*Plus* (*Atom x*) *One*, *Atom y*, *Atom y*])
  $h = [] \vee (\exists hs.\ h = [x, x] @ hs)$
  shows *TS_inv'* (*config'* (*rTS h0*) ([*x*, *y*], *h*) *qs*) (*last qs*) [*x*,*y*]
    ∧ *T_on'* (*rTS h0*) ([*x*, *y*], *h*) *qs* = *2*
proof −
  obtain *u v* **where** *uu*: $u \in lang$ (*Plus* (*Atom x*) *One*)
    and *vv*: $v \in lang$ (*seq*[*Atom y*, *Atom y*])
    and *qsuv*: *qs* = *u* @ *v*
    using *assms*(*2*)
    by (*auto simp*: *conc_def*)

  from *vv* **have** *vv2*: *v* = [*y*,*y*] **by** *auto*

  from *uu* **have** *TS_prefix*: *T_on'* (*rTS h0*) ([*x*, *y*], *h*) *u* = *0*
  using *assms*(*1*) **by**(*auto simp add*: *rTS_def oneTS_steps* t_p*_def*)

  have *h_split*: *rev u* @ *h* = [] ∨ *rev u* @ *h* = [*x*] ∨ (∃ *hs*. *rev u* @ *h* = [*x*,*x*]@*hs*)
    using *assms*(*3*) *uu* **by**(*auto*)

  then have *e*: *T_on'* (*rTS h0*) ([*x*,*y*],*rev u* @ *h*) [*y*,*y*] = *2*
    using *assms*(*1*)
    apply(*auto simp add*: *rTS_def*
        oneTS_steps
        Step_def step_def t_p*_def*) **done**

  have *conf*: *config'* (*rTS h0*) ([*x*, *y*], *h*) *u* = ([*x*,*y*], *rev u* @ *h*)
  using *uu* **by**(*auto simp add*: *Step_def rTS_def TS_step_d_def step_def*)

  have *T_on'* (*rTS h0*) ([*x*, *y*], *h*) *qs* = *T_on'* (*rTS h0*) ([*x*, *y*], *h*) (*u* @ *v*) **using** *qsuv* **by** *auto*
  also have ...
    =*T_on'* (*rTS h0*) ([*x*, *y*], *h*) *u* + *T_on'* (*rTS h0*) (*config'* (*rTS h0*) ([*x*, *y*], *h*) *u*) *v*

by(*rule T_on′_append*)
also have ...
= *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *u* + *T_on′* (*rTS h0*) ([*x*,*y*],*rev u* @ *h*) [*y*,*y*]
by(*simp add*: *conf vv2*)
also have ... = *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *u* + *2* **by** (*simp only*: *e*)
also have ... = *2* **by** (*simp add*: *TS_prefix*)
finally have *TS*: *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *qs*= *2* **.**

have *lqs*: *last qs* = *y* **using** *assms*(*2*) **by** *force*

from *assms*(*1*) **have** *config′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = ([*y*,*x*], *rev qs* @ *h*)
unfolding *qsuv*
apply(*simp only*: *config′_append2 conf vv2*)
using *h_split*
apply(*auto simp add*: *Step_def rTS_def*
oneTS_steps
step_def)
by(*simp_all add*: *mtf2_def swap_def*)

with *assms*(*1*) **have** *TS_inv′* (*config′* (*rTS h0*) ([*x*, *y*], *h*) *qs*) (*last qs*) [*x*,*y*]
apply(*subst TS_inv′_det*)
by(*simp add*: *qsuv vv2 lqs*)

show *?thesis* **unfolding** *TS* **apply**(*auto*) **by** *fact*
qed

lemma *TS_a′*: **assumes** $x \neq y$
$h = [] \vee (\exists hs.\ h = [x, x] @ hs)$
and $qs \in lang\ (seq\ [Plus\ (Atom\ x)\ rexp.One,\ Atom\ y,\ Atom\ y])$
shows $T\_on'\ (rTS\ h0)\ ([x, y], h)\ qs \leq 2 * T_p\ [x, y]\ qs\ (OPT2\ qs\ [x, y])$
$\wedge$ *TS_inv′* (*config′* (*rTS h0*) ([*x*, *y*], *h*) *qs*) (*last qs*) [*x*, *y*]
$\wedge$ *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = *2*
proof −
have *OPT*: T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) = *1* **using** *OPT2_A*[*OF assms*(*1*,*3*)] **by** *auto*
show *?thesis* **using** *OPT ts_a′*[*OF assms*(*1*,*3*,*2*)] **by** *auto*
qed

lemma *TS_a″*: **assumes**
$x \neq y\ \{x, y\} = \{x0, y0\}$ *TS_inv s x* [*x0*, *y0*]

```
    set qs ⊆ {x, y} qs ∈ lang (seq [Plus (Atom x) One, Atom y, Atom y])
  shows
    TS_inv (config'_rand (embed (rTS h0)) s qs) (last qs) [x0, y0]
     ∧ T_p_on_rand' (embed (rTS h0)) s qs = 2
proof −
  from assms(1,2) have kas: (x0=x ∧ y0=y) ∨ (y0=x ∧ x0=y) by(auto)
  then obtain h where S: s = return_pmf ([x,y],h) and h: h = [] ∨ (∃hs.
h = [x, x] @ hs)
    apply(rule disjE) using assms(1,3) unfolding TS_inv_def by(auto)

  have l: qs ≠ [] using assms by auto

  {
    fix x y qs h0
    fix h:: nat list
    assume A: x ≠ y
        qs ∈ lang (seq [question (Atom x), Atom y, Atom y])
        h = [] ∨ (∃hs. h = [x, x] @ hs)

    have TS_inv (config'_rand (embed (rTS h0)) (return_pmf ([x, y], h))
qs) (last qs) [x, y] ∧
            T_on_rand' (embed (rTS h0)) (return_pmf ([x, y], h)) qs = 2
      apply(simp only: T_on'_embed[symmetric] config'_embed)
      using ts_a'[OF A] by auto
  } note b=this


  show ?thesis unfolding S
    using kas apply(rule disjE)
      apply(simp only:)
      apply(rule b)
        using assms apply(simp)
        using assms apply(simp)
        using h apply(simp)
      apply(simp only:)

      apply(subst TS_inv_sym[of y x x y])
        using assms(1) apply(simp)
        apply(blast)
        defer
        apply(rule b)
          using assms apply(simp)
          using assms apply(simp)
          using h apply(simp)
```

using *last_in_set l assms*(*4*) **by** *blast*
qed

15.3.7 x+yx(yx)*x

lemma *ts_c'*: **assumes** $x \neq y$
$v \in lang$ (*seq*[*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
($\exists$ *hs*. $h = [x, x]$ @ *hs*) $\vee$ $h = [x]$ $\vee$ $h = []$
shows *T_on'* (*rTS h0*) ([*x*, *y*], *h*) *v* = (*length v* − *2*)
$\wedge$ *config'* (*rTS h0*) ([*x*,*y*], *h*) *v* = ([*x*,*y*],*rev v*@*h*) $\wedge$ ($\exists$ *hs*. (*rev v* @ *h*) = [*x*,*x*]@*hs*)
proof −
from *assms* **have** *lenvmod*: *length v mod 2* = *1* **apply**(*simp*)
proof −
assume $v \in$ ({[*y*]} @@ {[*x*]}) @@ *star*({[*y*]} @@ {[*x*]}) @@ {[*x*]}
then obtain *p q r* **where** *pqr*: *v*=*p*@*q*@*r* **and** *p*∈({[*y*]} @@ {[*x*]})
and *q*: $q \in$ *star* ({[*y*]} @@ {[*x*]}) **and** $r \in$ {[*x*]} **by** (*metis concE*)
then have *p* = [*y*,*x*] *r*=[*x*] **by** *auto*
with *pqr* **have** *a*: *length v* = *3*+*length q* **by** *auto*

from *q* **have** *b*: *length q mod 2* = *0*
apply(*induct q rule*: *star_induct*) **by** (*auto*)
from *a b* **show** *length v mod 2* = *Suc 0* **by** *auto*
qed

with *assms*(*1*,*3*) **have** *fall*: ($\exists$ *hs*. *h* = [*x*, *x*] @ *hs*) $\vee$ *index h y* = *length h*
by(*auto*)

from *assms*(*2*) **have** $v \in lang$ (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
@@ *lang* (*seq*[*Atom x*]) **by** (*auto simp*: *conc_def*)
then obtain *a b* **where** *aa*: $a \in lang$ (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
and $b \in lang$ (*seq*[*Atom x*])
and *vab*: *v* = *a* @ *b*
by(*erule concE*)
then have *bb*: *b*=[*x*] **by** *auto*
from *aa* **have** *lena*: *length a* > *0* **by** *auto*

from *TS_yxyx'*[*OF assms*(*1*) *aa fall*] **have** *stars*: *T_on'* (*rTS h0*) ([*x*,

y], *h*) (*a* @ *b*) =
  length a − *1* + *T_on′* (*rTS h0*) ([*x*, *y*],*rev a* @ *h*) *b*
  and *history*: (∃*hs*. *rev a* @ *h* = [*x*, *y*] @ *hs*)
  and *state*: *config′* (*rTS h0*) ([*x*, *y*], *h*) *a* = ([*x*, *y*], *rev a* @ *h*) **by** *auto*

have *suffix*: *T_on′* (*rTS h0*) ([*x*, *y*],*rev a* @ *h*) *b* = *0*
    and *suState*: *config′* (*rTS h0*) ([*x*, *y*], *rev a* @ *h*) *b* = ([*x*,*y*], *rev b* @ (*rev a* @ *h*))
  unfolding *bb* **using** *TS_x′* **by** *auto*

from *stars suffix* **have** *T_on′* (*rTS h0*) ([*x*, *y*], *h*) (*a* @ *b*) = *length a* − *1* **by** *auto*
then have *whatineed*: *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *v* = (*length v* − *2*) **using** *vab bb* **by** *auto*

have *conf*: *config′* (*rTS h0*) ([*x*, *y*], *h*) *v* = ([*x*, *y*], *rev v* @ *h*)
  by(*simp add*: *vab config′_append2 state suState*)

from *history* **obtain** *hs′* **where** *rev a* @ *h* = [*x*, *y*] @ *hs′* **by** *auto*
then obtain *hs2* **where** *reva*: *rev a* @ *h* = *x* # *hs2* **by** *auto*

show *?thesis* **using** *whatineed*
  apply(*auto*)
    using *conf* **apply**(*simp*)
    by(*simp add*: *reva vab bb*)
qed

lemma *TS_c1″*: **assumes**
  x ≠ *y* {*x*, *y*} = {*x0*, *y0*} *TS_inv s x* [*x0*, *y0*]
  set qs ⊆ {*x*, *y*}
  qs ∈ *lang* (*seq* [*Atom y*, *Atom x*, *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
shows *TS_inv* (*config′_rand* (*embed* (*rTS h0*)) *s qs*) (*last qs*) [*x0*, *y0*]
    ∧ *T_on_rand′* (*embed* (*rTS h0*)) *s qs* = (*length qs* − *2*)
proof −
from *assms*(*1*,*2*) **have** *kas*: (*x0*=*x* ∧ *y0*=*y*) ∨ (*y0*=*x* ∧ *x0*=*y*) **by**(*auto*)
then obtain *h* **where** *S*: *s* = *return_pmf* ([*x*,*y*],*h*) **and** *h*: *h* = [] ∨ (∃*hs*. *h* = [*x*, *x*] @ *hs*)
  apply(*rule disjE*) **using** *assms*(*1*,*3*) **unfolding** *TS_inv_def* **by**(*auto*)

```
  have l: qs ≠ [] using assms by auto
  {
    fix x y qs h0
    fix h:: nat list
    assume A: x ≠ y
       and B: qs ∈ lang (seq[Times (Atom y) (Atom x), Star (Times (Atom
y) (Atom x)), Atom x])
        and C: h = [] ∨ (∃hs. h = [x, x] @ hs)

    then have C': (∃hs. h = [x, x] @ hs) ∨ h = [x] ∨ h = [] by blast
      from B have lqs: last qs = x using assms(5) by(auto simp add:
conc_def)

    have TS_inv (config'_rand (embed (rTS h0)) (return_pmf ([x, y], h))
qs) (last qs) [x, y] ∧
              T_on_rand' (embed (rTS h0)) (return_pmf ([x, y], h)) qs =
length qs − 2
      apply(simp only: T_on'_embed[symmetric] config'_embed)
      using ts_c'[OF A B C'] A lqs unfolding TS_inv'_det by auto
  } note c1=this


  show ?thesis unfolding S
    using kas apply(rule disjE)
      apply(simp only:)
      apply(rule c1)
        using assms apply(simp)
        using assms apply(simp add: conc_assoc)
        using h apply(simp)
      apply(simp only:)

      apply(subst TS_inv_sym[of y x x y])
        using assms(1) apply(simp)
        apply(blast)
        defer
        apply(rule c1)
          using assms apply(simp)
          using assms apply(simp add: conc_assoc)
          using h apply(simp)
        using last_in_set l assms(4) by blast
qed

lemma ts_c2': assumes x ≠ y
  qs ∈ lang (seq[Atom x, Times (Atom y) (Atom x), Star (Times (Atom y)
```

(*Atom x*)), *Atom x*])
(∃ *hs*. *h* = [*x*, *x*] @ *hs*) ∨ *h* = []
shows *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = (*length qs* − *3*)
∧ *config′* (*rTS h0*) ([*x*,*y*], *h*) *qs* = ([*x*,*y*],*rev qs*@*h*) ∧ (∃ *hs*. (*rev qs* @ *h*) = [*x*,*x*]@*hs*)
proof −
from *assms*(*2*) **obtain** *v* **where** *qs*: *qs* = [*x*]@*v*
and *V*: *v*∈*lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
by(*auto simp add*: *conc_assoc*)

from *assms*(*3*) **have** *3*: (∃ *hs*. *x*#*h* = [*x*, *x*] @ *hs*) ∨ *x*#*h* = [*x*] ∨ *x*#*h* = [] **by** *auto*

from *ts_c′*[*OF assms*(*1*) *V 3*]
have *T*: *T_on′* (*rTS h0*) ([*x*, *y*], *x*#*h*) *v* = *length v* − *2*
and *C*: *config′* (*rTS h0*) ([*x*, *y*], *x*#*h*) *v* = ([*x*, *y*], *rev v* @ *x*#*h*)
and *H*: (∃ *hs*. *rev v* @ *x*#*h* = [*x*, *x*] @ *hs*) **by** *auto*

have *t*: t_p [*x*, *y*] *x* (*fst* (*snd* (*rTS h0*) ([*x*, *y*], *h*) *x*)) = *0*
by (*simp add*: *step_def rTS_def TS_step_d_def* t_p*_def*)
have *c*: *Partial_Cost_Model.Step* (*rTS h0*) ([*x*, *y*], *h*) *x*
= ([*x*,*y*], *x*#*h*) **by** (*simp add*: *Step_def rTS_def TS_step_d_def step_def*)

show *?thesis*
unfolding *qs* **apply**(*safe*)
apply(*simp add*: *T_on′_append T c t*)
apply(*simp add*: *config′_rand_append C c*)
using *H* **by** *simp*
qed

lemma *TS_c2″*: **assumes**
x ≠ *y* {*x*, *y*} = {*x0*, *y0*} *TS_inv s x* [*x0*, *y0*]
set qs ⊆ {*x*, *y*}
qs ∈ *lang* (*seq* [*Atom x*, *Atom y*, *Atom x*, *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
shows *TS_inv* (*config′_rand* (*embed* (*rTS h0*)) *s qs*) (*last qs*) [*x0*, *y0*]
∧ *T_on_rand′* (*embed* (*rTS h0*)) *s qs* = (*length qs* − *3*)
proof −
from *assms*(*1*,*2*) **have** *kas*: (*x0*=*x* ∧ *y0*=*y*) ∨ (*y0*=*x* ∧ *x0*=*y*) **by**(*auto*)
then obtain *h* **where** *S*: *s* = *return_pmf* ([*x*,*y*],*h*) **and** *h*: *h* = [] ∨ (∃ *hs*. *h* = [*x*, *x*] @ *hs*)

apply(*rule disjE*) **using** *assms*(*1*,*3*) **unfolding** *TS_inv_def* **by**(*auto*)

have *l*: *qs* ≠ [] **using** *assms* **by** *auto*
{
fix *x y qs h0*
fix *h*:: *nat list*
assume *A*: *x* ≠ *y*
and *B*: *qs* ∈ *lang* (*seq*[*Atom x*, *Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
and *C*: *h* = [] ∨ (∃ *hs*. *h* = [*x*, *x*] @ *hs*)

from *B* **have** *lqs*: *last qs* = *x* **using** *assms*(*5*) **by**(*auto simp add*: *conc_def*)

from *C* **have** *C'*: (∃ *hs*. *h* = [*x*, *x*] @ *hs*) ∨ *h* = [] **by** *blast*

have *TS_inv* (*config'_rand* (*embed* (*rTS h0*)) (*return_pmf* ([*x*, *y*], *h*)) *qs*) (*last qs*) [*x*, *y*] ∧
T_on_rand' (*embed* (*rTS h0*)) (*return_pmf* ([*x*, *y*], *h*)) *qs* = *length qs* − *3*
apply(*simp only*: *T_on'_embed*[*symmetric*] *config'_embed*)
using *ts_c2'*[*OF A B C'*] *A lqs* **unfolding** *TS_inv'_det* **by** *auto*
} **note** *c2=this*

show *?thesis* **unfolding** *S*
using *kas* **apply**(*rule disjE*)
apply(*simp only*:)
apply(*rule c2*)
using *assms* **apply**(*simp*)
using *assms* **apply**(*simp add*: *conc_assoc*)
using *h* **apply**(*simp*)
apply(*simp only*:)

apply(*subst TS_inv_sym*[*of y x x y*])
using *assms*(*1*) **apply**(*simp*)
apply(*blast*)
defer
apply(*rule c2*)
using *assms* **apply**(*simp*)
using *assms* **apply**(*simp add*: *conc_assoc*)
using *h* **apply**(*simp*)
using *last_in_set l assms*(*4*) **by** *blast*
qed

lemma *TS_c′*: **assumes** $x \neq y$ $h = [] \vee (\exists hs.\ h = [x, x] @ hs)$
qs ∈ *lang* (*seq* [*Plus* (*Atom x*) *rexp.One*, *Atom y*, *Atom x*, *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
shows *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *qs*
≤ *2* ∗ T_p [*x*, *y*] *qs* (*OPT2 qs* [*x*, *y*]) ∧ *TS_inv′* (*config′* (*rTS h0*) ([*x*, *y*], *h*) *qs*) (*last qs*) [*x*,*y*]
proof −
obtain *u v* **where** *uu*: *u* ∈ *lang* (*Plus* (*Atom x*) *One*)
and *vv*: *v* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
and *qsuv*: *qs* = *u* @ *v*
using *assms*(*3*)
by (*auto simp*: *conc_def*)

from *TS_xr′*[*OF assms*(*1*) *uu assms*(*2*)] **have**
T_pre: *T_on′* (*rTS h0*) ([*x*, *y*], *h*) (*u*@*v*) = *T_on′* (*rTS h0*) ([*x*, *y*], *rev u* @ *h*) *v*
and *fall′*: (∃ *hs*. (*rev u* @ *h*) = [*x*, *x*] @ *hs*) ∨ (*rev u* @ *h*) = [*x*] ∨ (*rev u* @ *h*)=[]
and *conf′*: *config′* (*rTS h0*) ([*x*, *y*], *h*) (*u* @ *v*) =
config′ (*rTS h0*) ([*x*, *y*], *rev u* @ *h*) *v* **by** *auto*

with *assms uu* **have** *fall*: (∃ *hs*. (*rev u* @ *h*) = [*x*, *x*] @ *hs*) ∨ *index* (*rev u* @ *h*) *y* = *length* (*rev u* @ *h*)
by(*auto*)

from *ts_c′*[*OF assms*(*1*) *vv fall′*] **have**
T_star: *T_on′* (*rTS h0*) ([*x*, *y*], *rev u* @ *h*) *v* = (*length v* − *2*)
and *inv1*: *config′* (*rTS h0*) ([*x*, *y*], (*rev u* @ *h*)) *v* = ([*x*, *y*], *rev v* @ *rev u* @ *h*)
and *inv2*: (∃ *hs*. *rev v* @ *rev u* @ *h* = [*x*, *x*] @ *hs*) **by** *auto*

from *T_pre T_star qsuv* **have** *TS*: *T_on′* (*rTS h0*) ([*x*, *y*], *h*) *qs* = (*length v* − *2*) **by** *metis*

from *uu* **have** *uuu*: *u*=[] ∨ *u*=[*x*] **by** *auto*
from *vv* **have** *vvv*: *v* ∈ *lang* (*seq*
[*Atom y*, *Atom x*,
Star (*Times* (*Atom y*) (*Atom x*)),
Atom x]) **by**(*auto simp*: *conc_def*)

have *OPT*: T_p *[x,y] qs (OPT2 qs [x,y]) = (length v) div 2* **apply**(*rule OPT2_C*) **by**(*fact*)+

have *lqs*: *last qs = x* **using** *assms(3)* **by** *force*

have *conf*: *config′ (rTS h0) ([x, y], h) qs = ([x, y], rev qs @ h)*
by(*simp add*: *qsuv conf′ inv1*)
then have *conf*: *TS_inv′ (config′ (rTS h0) ([x, y], h) qs) (last qs) [x,y]*
apply(*simp add*: *lqs*)
apply(*subst TS_inv′_det*)
using *inv2 qsuv* **by**(*simp*)

show *?thesis* **unfolding** *TS OPT*
by (*auto simp add*: *conf*)
qed

15.3.8 xx

lemma *request_first*: *x≠y ⟹ Step (rTS h) ([x, y], is) x = ([x,y],x#is)*
unfolding *rTS_def Step_def* **by**(*simp add*: *split_def TS_step_d_def step_def*)

lemma *ts_d′*: *qs ∈ Lxx x y ⟹*
x ≠ y ⟹
h = [] ∨ (∃ hs. h = [x, x] @ hs) ⟹
qs ∈ lang (seq [Atom x, Atom x]) ⟹
T_on′ (rTS h0) ([x, y], h) qs = 0 ∧
TS_inv′ (config′ (rTS h0) ([x, y], h) qs) x [x,y]
proof −
assume *xny*: *x ≠ y*
assume *qs ∈ lang (seq [Atom x, Atom x])*
then have *xx*: *qs = [x,x]* **by** *auto*

from *xny* **have** *TS*: *T_on′ (rTS h0) ([x, y], h) qs = 0* **unfolding** *xx*
by(*auto simp add*: *Step_def step_def oneTS_steps rTS_def* t_p*_def*)

from *xny* **have** *config′ (rTS h0) ([x, y], h) qs = ([x, y], x # x # h)*
by(*auto simp add*: *xx Step_def rTS_def oneTS_steps step_def*)

then have *TS_inv′ (config′ (rTS h0) ([x, y], h) qs) x [x, y]*
by(*simp add*: *TS_inv′_det*)

with *TS* **show** *?thesis* **by** *simp*
qed

```
lemma TS_d': assumes xny: x ≠ y and h = [] ∨ (∃hs. h = [x, x] @ hs)
    and qsis: qs ∈ lang (seq [Atom x, Atom x])
    shows T_on' (rTS h0) ([x,y],h) qs ≤ 2 ∗ T_p [x, y] qs (OPT2 qs [x, y])

      and TS_inv' (config' (rTS h0) ([x,y],h) qs)  (last qs) [x, y]
      and T_on' (rTS h0) ([x,y],h) qs = 0
proof −
  from qsis have xx: qs = [x,x] by auto

  show TS: T_on' (rTS h0) ([x,y],h) qs = 0
   using assms(1) by (auto simp add: xx t_p_def rTS_def Step_def oneTS_steps
step_def)
  then show T_on' (rTS h0) ([x,y],h) qs ≤ 2 ∗ T_p [x, y] qs (OPT2 qs [x,
y]) by simp

  show TS_inv' (config' (rTS h0) ([x,y],h) qs)  (last qs) [x, y]
    unfolding TS_inv_def
      by(simp add: xx request_first[OF xny])
qed


lemma TS_d'': assumes
    x ≠ y {x, y} = {x0, y0} TS_inv s x [x0, y0]
    set qs ⊆ {x, y}
   qs ∈ lang (seq [Atom x, Atom x])
 shows TS_inv (config'_rand (embed (rTS h0)) s qs) (last qs) [x0, y0]
      ∧ T_on_rand' (embed (rTS h0)) s qs = 0
proof −
  from assms(1,2) have kas: (x0=x ∧ y0=y) ∨ (y0=x ∧ x0=y) by(auto)
  then obtain h where S: s = return_pmf ([x,y],h) and h: h = [] ∨ (∃hs.
h = [x, x] @ hs)
    apply(rule disjE) using assms(1,3) unfolding TS_inv_def by(auto)

  have l: qs ≠ [] using assms by auto
  {
    fix x y qs h0
    fix h:: nat list
    assume A: x ≠ y
        and B: qs ∈ lang (seq [Atom x, Atom x])
        and C: h = [] ∨ (∃hs. h = [x, x] @ hs)

      from B have lqs: last qs = x using assms(5) by(auto simp add:
```

conc_def)

have *TS_inv* (*config'_rand* (*embed* (*rTS h0*)) (*return_pmf* ([*x*, *y*], *h*)) *qs*) (*last qs*) [*x*, *y*] ∧
T_on_rand' (*embed* (*rTS h0*)) (*return_pmf* ([*x*, *y*], *h*)) *qs* = *0*
apply(*simp only*: *T_on'_embed*[*symmetric*] *config'_embed*)
using *TS_d'*[*OF A C B*] *A lqs* **unfolding** *TS_inv'_det* **by** *auto*
} **note** *d*=*this*

show *?thesis* **unfolding** *S*
using *kas* **apply**(*rule disjE*)
apply(*simp only*:)
apply(*rule d*)
using *assms* **apply**(*simp*)
using *assms* **apply**(*simp add*: *conc_assoc*)
using *h* **apply**(*simp*)
apply(*simp only*:)

apply(*subst TS_inv_sym*[*of y x x y*])
using *assms*(*1*) **apply**(*simp*)
apply(*blast*)
defer
apply(*rule d*)
using *assms* **apply**(*simp*)
using *assms* **apply**(*simp add*: *conc_assoc*)
using *h* **apply**(*simp*)
using *last_in_set l assms*(*4*) **by** *blast*
qed

15.4 Phase Partitioning

lemma *D'*: **assumes** $\sigma' \in$ *Lxx x y* **and** $x \neq y$ **and** *TS_inv'* ([*x*, *y*], *h*) *x* [*x*, *y*]
shows *T_on'* (*rTS h0*) ([*x*, *y*], *h*) $\sigma' \leq 2 * T_p$ [*x*, *y*] σ' (*OPT2* σ' [*x*, *y*])
∧ *TS_inv* (*config'_rand* (*embed* (*rTS h0*)) (*return_pmf* ([*x*, *y*], *h*)) σ') (*last* σ') [*x*, *y*]
proof −

from *config'_embed* **have** *config'_rand* (*embed* (*rTS h0*)) (*return_pmf* ([*x*, *y*], *h*)) σ'
= *return_pmf* (*Partial_Cost_Model.config'* (*rTS h0*) ([*x*, *y*], *h*) σ')

```
by blast

  then have L: TS_inv (config'_rand (embed (rTS h0)) (return_pmf ([x,
y], h)) σ') (last σ') [x, y]
      = TS_inv' (config' (rTS h0) ([x, y], h) σ')  (last σ') [x, y] by auto

  from assms(3) have
      h: h = [] ∨ (∃hs. h = [x, x] @ hs)
      by(auto simp add: TS_inv'_det)

  have T_on' (rTS h0) ([x, y], h) σ' ≤ 2 * T_p [x, y] σ' (OPT2 σ' [x, y])
      ∧  TS_inv' (config' (rTS h0) ([x, y], h) σ')  (last σ') [x, y]
  apply(rule LxxE[OF assms(1)])
    using TS_d'[OF assms(2) h, of σ'] apply(simp)
    using TS_b'[OF assms(2) h] apply(simp)
    using TS_c'[OF assms(2) h] apply(simp)
    using TS_a'[OF assms(2) h] apply fast
    done

  then show ?thesis using L by auto
qed

theorem TS_OPT2': (x::nat) ≠ y ⟹ set σ ⊆ {x,y}
     ⟹ T_p_on (rTS []) [x,y] σ  ≤ 2 * real (T_p_opt [x,y] σ) + 2
apply(subst T_on_embed)
  apply(rule Phase_partitioning_general[where P=TS_inv])
      apply(simp)
     apply(simp)
    apply(simp)
   apply(simp add: TS_inv_def rTS_def)
  proof (goal_cases)
    case (1 a b σ' s)
    from 1(6) obtain h hist' where s: s = return_pmf ([a, b], h)
           and h = [] ∨ h = [a,a]@hist'
      unfolding TS_inv_def apply(cases a=hd [x,y])
        apply(simp) using 1 apply fast
        apply(simp) using 1 by blast

    from 1 have xyab: TS_inv' ([a, b], h) a [x, y]
          = TS_inv' ([a, b], h) a [a, b]
           by(auto simp add: TS_inv'_det)

    with 1(6) s have inv: TS_inv' ([a, b], h) a [a, b] by simp
```

from ‹σ' ∈ *Lxx a b*› **have** σ' ≠ [] **using** *Lxx1* **by** *fastforce*
then have *l*: *last* σ' ∈ {*x*,*y*} **using** *1*(*5*,*7*) *last_in_set* **by** *blast*

show *?case* **unfolding** *s T_on'_embed*[*symmetric*]
using *D'*[*OF 1*(*3*,*4*) *inv*, *of* []]
apply(*safe*)
apply *linarith*
using *TS_inv_sym*[*OF 1*(*4*,*5*)] *l* **apply** *blast*
done
qed

15.5 TS is pairwise

lemma *config'_distinct*[*simp*]:
shows *distinct* (*fst* (*config' A S qs*)) = *distinct* (*fst S*)
apply (*induct qs rule*: *rev_induct*) **by**(*simp_all add*: *config'_snoc Step_def split_def distinct_step*)

lemma *config'_set*[*simp*]:
shows *set* (*fst* (*config' A S qs*)) = *set* (*fst S*)
apply (*induct qs rule*: *rev_induct*) **by**(*simp_all add*: *config'_snoc Step_def split_def set_step*)

lemma *s_TS_append*: *i*≤*length as* ⟹*s_TS init h* (*as*@*bs*) *i* = *s_TS init h as i*
by (*simp add*: *s_TS_def*)

lemma *s_TS_distinct*: *distinct init* ⟹ *i*<*length qs* ⟹ *distinct* (*fst* (*TSdet init h qs i*))
by(*simp_all add*: *config_config_distinct*)

lemma *othersdontinterfere*: *distinct init* ⟹ *i* < *length qs* ⟹ *a*∈*set init* ⟹ *b*∈*set init*
⟹ *set qs* ⊆ *set init* ⟹ *qs*!*i*∉{*a*,*b*} ⟹ *a* < *b in s_TS init h qs i* ⟹ *a* < *b in s_TS init h qs* (*Suc i*)
apply(*simp add*: *s_TS_def split_def take_Suc_conv_app_nth config_append Step_def step_def*)
apply(*subst x_stays_before_y_if_y_not_moved_to_front*)
apply(*simp_all add*: *config_config_distinct config_config_set*)
by(*auto simp*: *rTS_def TS_step_d_def*)

lemma *TS_mono*:
fixes *l*::*nat*
assumes *1*: *x* < *y in s_TS init h xs* (*length xs*)

```
    and l_in_cs: l ≤ length cs
    and firstocc: ∀j<l. cs ! j ≠ y
    and x ∉ set cs
    and di: distinct init
    and inin: set (xs @ cs) ⊆ set init
  shows x < y in s_TS init h (xs@cs) (length (xs)+l)
proof -
  from before_in_setD2[OF 1] have y: y : set init unfolding s_TS_def
by(simp add: config_config_set)
  from before_in_setD1[OF 1] have x: x : set init unfolding s_TS_def
by(simp add: config_config_set)

  {
    fix n
    assume n≤l
    then have x < y in s_TS init h ((xs)@cs) (length (xs)+n)
      proof(induct n)
        case 0
        show ?case apply (simp only: s_TS_append ) using 1 by(simp)
      next
        case (Suc n)
        then have n_lt_l: n<l by auto
        show ?case apply(simp)
          apply(rule othersdontinterfere)
            apply(rule di)
            using n_lt_l l_in_cs apply(simp)
            apply(fact x)
            apply(fact y)
            apply(fact inin)
            apply(simp add: nth_append) apply(safe)
              using assms(4) n_lt_l l_in_cs apply fastforce
              using firstocc n_lt_l apply blast
              using Suc(1) n_lt_l by(simp)
      qed
  }
  — before the request to y, x is in front of y
  then show x < y in s_TS init h (xs@cs) (length (xs)+l)
    by blast
qed

lemma step_no_action: step s q (0,[]) = s
unfolding step_def mtf2_def by simp

lemma s_TS_set: i ≤ length qs ⟹ set (s_TS init h qs i) = set init
```

apply(*induct i*)
apply(*simp add*: *s_TS_def*)
apply(*simp add*: *s_TS_def TSdet_Suc*)
by(*simp add*: *split_def rTS_def Step_def step_def*)

lemma *count_notin2*: *count_list xs x* = *0* $\Longrightarrow$ *x* $\notin$ *set xs*
by (*simp add*: *count_list_0_iff*)

lemma *mtf2_q_passes*: **assumes** *q* $\in$ *set xs distinct xs*
and *index xs q* $-$ *n* $\leq$ *index xs x index xs x* $<$ *index xs q*
shows *q* $<$ *x in* (*mtf2 n q xs*)
proof $-$
from *assms* **have** *index xs q* $<$ *length xs* **by** *auto*
with *assms*(*4*) **have** *ind_x*: *index xs x* $<$ *length xs* **by** *auto*
then have *xinxs*: *x*$\in$*set xs* **using** *index_less_size_conv* **by** *metis*

have *B*: *index* (*mtf2 n q xs*) *q* = *index xs q* $-$ *n*
apply(*rule mtf2_q_after*)
by(*fact*)+
also from *ind_x mtf2_forward_effect3'*[*OF assms*]
have *A*: ... $<$ *index* (*mtf2 n q xs*) *x* **by** *auto*
finally show *?thesis* **unfolding** *before_in_def* **using** *xinxs* **by** *force*
qed

lemma *twotox*:
assumes *count_list bs y* $\leq$ *1*
and *distinct init*
and *x* $\in$ *set init*
and *y* : *set init*
and *x* $\notin$ *set bs*
and *x*$\neq$*y*
shows *x* $<$ *y in s_TS init h* (*as*@[*x*]@*bs*@[*x*]) (*length* (*as*@[*x*]@*bs*@[*x*]))
proof $-$
have *aa*: *snd* (*TSdet init h* ((*as* @ *x* # *bs*) @ [*x*]) (*Suc* (*length as* + *length bs*)))
= *rev* (*take* (*Suc* (*length as* + *length bs*)) ((*as* @ *x* # *bs*) @ [*x*])) @ *h*
apply(*rule sndTSdet*) **by**(*simp*)
then have *aa'*: *snd* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))
= *rev* (*take* (*Suc* (*length as* + *length bs*)) ((*as* @ *x* # *bs*) @ [*x*])) @ *h* **by** *auto*
have *lasocc_x*: *index* (*snd* (*TSdet init h* ((*as* @ *x* # *bs*) @ [*x*]) (*Suc* (*length as* + *length bs*)))) *x* = *length bs*
unfolding *aa*

apply(*simp add*: *del*: *config′.simps*)
using *assms*(*5*) **by**(*simp add*: *index_append*)
then have *lasocc_x′*: (*index* (*snd* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *x*) = *length bs* **by** *auto*

let *?sincelast* = *take* (*length bs*)
(*snd* (*TSdet init h* ((*as* @ *x* # *bs*) @ [*x*])
(*Suc* (*length as* + *length bs*))))
have *sl*: *?sincelast* = *rev bs* **unfolding** *aa* **by**(*simp*)
let *?S* = {*xa*. *xa* < *x in fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*])
(*Suc* (*length as* + *length bs*))) ∧
count_list ?sincelast xa ≤ *1*}

have *y*: *y* ∈ *?S* ∨ ~ *y* < *x* *in s_TS init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*))
unfolding *sl* **unfolding** *s_TS_def* **using** *assms*(*1*) **by**(*simp del*: *config′.simps*)

have *eklr*: *length* (*as*@[*x*]@*bs*@[*x*]) = *Suc* (*length* (*as*@[*x*]@*bs*)) **by** *simp*
have *1*: *s_TS init h* (*as*@[*x*]@*bs*@[*x*]) (*length* (*as*@[*x*]@*bs*@[*x*]))
= *fst* (*Partial_Cost_Model.Step* (*rTS h*)
(*TSdet init h* (*as* @ [*x*] @ *bs* @ [*x*])
(*length* (*as* @ [*x*] @ *bs*)))
((*as* @ [*x*] @ *bs* @ [*x*]) ! *length* (*as* @ [*x*] @ *bs*))) **unfolding** *s_TS_def*
unfolding *eklr* **apply**(*subst TSdet_Suc*)
by(*simp_all add*: *split_def*)

have *brrr*: *x*∈ *set* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*))))
apply(*subst s_TS_set*[*unfolded s_TS_def*])
apply(*simp*) **by** *fact*
have *ydrin*: *y*∈*set* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*))))
apply(*subst s_TS_set*[*unfolded s_TS_def*]) **apply**(*simp*) **by** *fact*
have *dbrrr*: *distinct* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*))))
apply(*subst s_TS_distinct*[*unfolded s_TS_def*]) **using** *assms*(*2*) **by**(*simp_all*)

show *?thesis*
proof (*cases y* < *x* *in s_TS init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))
case *True*
with *y* **have** *yS*: *y*∈*?S* **by** *auto*
then have *minsteps*: *Min* (*index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*])

(*Suc* (*length as* + *length bs*)))) ‘ *?S*)
≤ *index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *y*
by *auto*
let *?entf* = *index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *x* −
Min (*index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) ‘ *?S*)
from *minsteps* **have** *br*: *index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *x* − (*?entf*)
≤ *index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *y*
by *presburger*
have *brr*: *index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *y*
< *index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *x*
using *True* **unfolding** *before_in_def s_TS_def* **by** *auto*

from *br brr* **have** *klo*: *index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *x* − (*?entf*)
≤ *index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *y*
∧ *index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *y*
< *index* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))) *x* **by** *metis*

let *?result* =(*mtf2 ?entf x* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))))

have *whatsthat*: *s_TS init h* (*as* @ [*x*] @ *bs* @ [*x*]) (*length* (*as* @ [*x*] @ *bs* @ [*x*]))
= *?result*
unfolding *1* **apply**(*simp add*: *split_def step_def rTS_def Step_def TS_step_d_def del*: *config'.simps*)
apply(*simp add*: *nth_append del*: *config'.simps*)
using *lasocc_x'*[*unfolded rTS_def*] *aa'*[*unfolded rTS_def*]
apply(*simp add*: *del*: *config'.simps*)
using *yS*[*unfolded sl rTS_def*] **by** *auto*

have *ydrinee*: *y* ∈ *set* (*mtf2 ?entf x* (*fst* (*TSdet init h* (*as* @ *x* # *bs* @

[x]) (*Suc* (*length as* + *length bs*)))))
      apply(*subst set_mtf2*)
      apply(*subst s_TS_set*[*unfolded s_TS_def*]) **apply**(*simp*) **by** *fact*

   show *?thesis* **unfolding** *whatsthat* **apply**(*rule mtf2_q_passes*) **by**(*fact*)+

  next
   case *False*
   then have *2*: *x* < *y* *in s_TS init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*))
     using *brrr ydrin not_before_in assms*(*6*) **unfolding** *s_TS_def* **by** *metis*
   {
    fix *e*
    have *x* < *y in mtf2 e x* (*s_TS init h* (*as* @ *x* # *bs* @ [*x*]) (*Suc* (*length as* + *length bs*)))
      apply(*rule x_stays_before_y_if_y_not_moved_to_front*)
       unfolding *s_TS_def*
       apply(*fact*)+
       using *assms*(*6*) **apply**(*simp*)
       using *2* **unfolding** *s_TS_def* **by** *simp*
   } **note** *bratz=this*
   show *?thesis* **unfolding** *1* **apply**(*simp add*: *TSnopaid split_def Step_def s_TS_def TS_step_d_def step_def nth_append* *del*: *config'.simps*)
        using *bratz*[*unfolded s_TS_def*] **by** *simp*
  qed

qed

lemma *count_drop*: *count_list* (*drop n cs*) *x* ≤ *count_list cs x*
proof −
  have *count_list cs x* = *count_list* (*take n cs* @ *drop n cs*) *x* **by** *auto*
  also have ... = *count_list* (*take n cs*) *x* + *count_list* (*drop n cs*) *x* **by** (*rule count_list_append*)
  also have ... ≥ *count_list* (*drop n cs*) *x* **by** *auto*
  finally show *?thesis* **.**
qed

lemma *count_take_less*: **assumes** *n*≤*m*
  shows *count_list* (*take n cs*) *x* ≤ *count_list* (*take m cs*) *x*
proof −
   from *assms* **have** *count_list* (*take n cs*) *x* = *count_list* (*take n* (*take m cs*)) *x* **by** *auto*
   also have ... ≤ *count_list* (*take n* (*take m cs*) @ *drop n* (*take m cs*)) *x*

by (*simp*)
    also have ... = *count_list* (*take m cs*) *x*
        by(*simp only*: *append_take_drop_id*)
    finally show *?thesis* **.**
qed

lemma *count_take*: *count_list* (*take n cs*) *x* ≤ *count_list cs x*
proof −
  have *count_list cs x* = *count_list* (*take n cs* @ *drop n cs*) *x* **by** *auto*
  also have ... = *count_list* (*take n cs*) *x* + *count_list* (*drop n cs*) *x* **by** (*rule count_list_append*)
  also have ... ≥ *count_list* (*take n cs*) *x* **by** *auto*
  finally show *?thesis* **.**
qed

lemma *casexxy*: **assumes** *σ*=*as*@[*x*]@*bs*@[*x*]@*cs*
    and *x* ∉ *set cs*
    and *set cs* ⊆ *set init*
    and *x* ∈ *set init*
    and *distinct init*
    and *x* ∉ *set bs*
    and *set as* ⊆ *set init*
    and *set bs* ⊆ *set init*
  shows (%*i*. *i*<*length cs* ⟶ (∀ *j*<*i*. *cs*!*j*≠*cs*!*i*) ⟶ *cs*!*i*≠*x*
      ⟶ (*cs*!*i*) ∉ *set bs*
      ⟶ *x* < (*cs*!*i*) *in* (*s_TS init h σ* (*length* (*as*@[*x*]@*bs*@[*x*]) + *i*+*1*))) *i*
proof (*rule infinite_descent*[**where** *P*=(%*i*. *i*<*length cs* ⟶ (∀ *j*<*i*. *cs*!*j*≠*cs*!*i*) ⟶ *cs*!*i*≠*x*
      ⟶ (*cs*!*i*) ∉ *set bs*
      ⟶ *x* < (*cs*!*i*) *in* (*s_TS init h σ* (*length* (*as*@[*x*]@*bs*@[*x*]) + *i*+*1*)))], *goal_cases*)
  case (*1 i*)
  let *?y* = *cs*!*i*
  from *1* **have** *i_in_cs*: *i* < *length cs* **and**
      firstocc: (∀ *j*<*i*. *cs* ! *j* ≠ *cs* ! *i*)
      and *ynx*: *cs* ! *i* ≠ *x*
      and *ynotinbs*: *cs* ! *i* ∉ *set bs*
      and *y_before_x'*: ~*x* < *cs* ! *i in s_TS init h σ* (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*+*1*) **by** *auto*

  have *ss*: *set* (*s_TS init h σ* (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*+*1*)) = *set init* **using** *assms*(*1*) *i_in_cs* **by**(*simp add*: *s_TS_set*)
  then have *cs* ! *i* ∈ *set* (*s_TS init h σ* (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*+*1*))

unfolding *ss* **using** *assms(3) i_in_cs* **by** *fastforce*
moreover have *x* : *set* (*s_TS init h* σ (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*+*1*))
unfolding *ss* **using** *assms(4)* **by** *fastforce*

— after the request to y, y is in front of x
ultimately have *y_before_x_Suct3*: *?y* < *x in s_TS init h* σ (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*+*1*)
using *y_before_x'* *ynx not_before_in* **by** *metis*

from *ynotinbs* **have** *yatmostonceinbs*: *count_list bs* (*cs*!*i*) ≤ *1* **by** *simp*

let *?y* = *cs*!*i*
have *yininit*: *?y* ∈ *set init* **using** *assms(3) i_in_cs* **by** *fastforce*
{
fix *y*
assume *y* ∈ *set init*
assume *x*≠*y*
assume *count_list bs y* ≤ *1*
then have *x* < *y in s_TS init h* (*as*@[*x*]@*bs*@[*x*]) (*length* (*as*@[*x*]@*bs*@[*x*]))
apply(*rule twotox*) **by**(*fact*)+
} **note** *xgoestofront*=*this*
with *yatmostonceinbs ynx yininit* **have** *zeitpunktt2*: *x* < *?y in s_TS init h* (*as*@[*x*]@*bs*@[*x*]) (*length* (*as*@[*x*]@*bs*@[*x*])) **by** *blast*

have *i* ≤ *length cs* **using** *i_in_cs* **by** *auto*
have *x_before_y_t3*: *x* < *?y in s_TS init h* ((*as*@[*x*]@*bs*@[*x*])@*cs*) (*length* (*as*@[*x*]@*bs*@[*x*])+*i*)
apply(*rule TS_mono*)
apply(*fact*)+
using *assms* **by** *simp*
— so x and y swap positions when y is requested, that means that y was inserted infront of some elment z (which cannot be x, has only been requested at most once since last request of y but is in front of x)

— first show that y must have been requested in as

have *snd* (*TSdet init h* (*as* @ [*x*] @ *bs* @ [*x*] @ *cs*) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*)) =
rev (*take* (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*) (*as* @ [*x*] @ *bs* @ [*x*] @ *cs*)) @ *h*
apply(*rule sndTSdet*) **using** *i_in_cs* **by** *simp*
also have ... = (*rev* (*take i cs*)) @ [*x*] @ (*rev bs*) @ [*x*] @ (*rev as*) @ *h*

by *simp*
finally have *fstTS_t3*: *snd* (*TSdet init h* (*as* @ [*x*] @ *bs* @ [*x*] @ *cs*) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*)) =
(*rev* (*take i cs*)) @ [*x*] @ (*rev bs*) @ [*x*] @ (*rev as*) @ *h* **.**
then have *fstTS_t3'*: (*snd* (*TSdet init h σ* (*Suc* (*Suc* (*length as* + *length bs* + *i*))))) =
(*rev* (*take i cs*)) @ [*x*] @ (*rev bs*) @ [*x*] @ (*rev as*) @ *h* **using** *assms*(*1*) **by** *auto*

let *?is* = *snd* (*TSdet init h* (*as* @ [*x*] @ *bs* @ [*x*] @ *cs*) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*))
let *?is'* = *snd* (*config* (*rTS h*) *init* (*as* @ [*x*] @ *bs* @ [*x*] @ (*take i cs*)))
let *?s* = *fst* (*TSdet init h* (*as* @ [*x*] @ *bs* @ [*x*] @ *cs*) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*))
let *?s'* = *fst* (*config* (*rTS h*) *init* (*as* @ [*x*] @ *bs* @ [*x*] @ (*take i cs*)))
let *?s_Suct3*=*s_TS init h* (*as* @ [*x*] @ *bs* @ [*x*] @ *cs*) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*+*1*)

let *?S* = {*xa*. (*xa* < (*as* @ [*x*] @ *bs* @ [*x*] @ *cs*) ! (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*) *in ?s* ∧
count_list (*take* (*index ?is* ((*as* @ [*x*] @ *bs* @ [*x*] @ *cs*) ! (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*))) *?is*) *xa* ≤ *1*) }
let *?S'* = {*xa*. (*xa* < (*as* @ [*x*] @ *bs* @ [*x*] @ *cs*) ! (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*) *in ?s'* ∧
count_list (*take* (*index ?is'* ((*cs*!*i*))) *?is'*) *xa* ≤ *1*) }

have *isis'*: *?is* = *?is'* **by**(*simp*)
have *ss'*: *?s* = *?s'* **by**(*simp*)
then have *SS'*: *?S* = *?S'* **using** *i_in_cs* **by**(*simp add*: *nth_append*)

have *once*: *TSdet init h* (*as* @ *x* # *bs* @ *x* # *cs*) (*Suc* (*Suc* (*Suc* (*length as* + *length bs* + *i*))))
= *Step* (*rTS h*) ($config_p$ (*rTS h*) *init* (*as* @ *x* # *bs* @ *x* # *take i cs*)) (*cs* ! *i*)
apply(*subst TSdet_Suc*)
using *i_in_cs* **apply**(*simp*)
by(*simp add*: *nth_append*)

have *aha*: (*index ?is* (*cs* ! *i*) ≠ *length ?is*)
∧ *?S* ≠ {}
proof (*rule ccontr*, *goal_cases*)
case *1*

then have (*index ?is* (*cs* ! *i*) = *length ?is*) ∨ *?S* = {} **by**(*simp*)
then have *alters*: (*index ?is′* (*cs* ! *i*) = *length ?is′*) ∨ *?S′* = {}
apply(*simp only*: *SS′*) **by**(*simp only*: *isis′*)
— wenn (cs ! i) noch nie requested wurde, dann kann es gar nicht nach vorne gebracht werden! also widerspruch mit *y_before_x′*
have *?s_Suct3* = *fst* (*config* (*rTS h*) *init* ((*as* @ [*x*] @ *bs* @ [*x*]) @ (*take* (*i*+*1*) *cs*)))
unfolding *s_TS_def*
apply(*simp only*: *length_append*)
apply(*subst take_append*)
apply(*subst take_append*)
apply(*subst take_append*)
apply(*subst take_append*)
by(*simp*)
also have ... = *fst* (*config* (*rTS h*) *init* (((*as* @ [*x*] @ *bs* @ [*x*]) @ (*take* *i cs*)) @ [*cs*!*i*]))
using *i_in_cs* **by**(*simp add*: *take_Suc_conv_app_nth*)
also have ... = *step ?s′ ?y* (*0*, [])
proof (*cases index ?is′* (*cs* ! *i*) = *length ?is′*)
case *True*
show *?thesis*
apply(*subst config_append*)
using *i_in_cs* **apply**(*simp add*: *rTS_def Step_def split_def nth_append*)
apply(*subst TS_step_d_def*)
apply(*simp only*: *True*[*unfolded rTS_def*,*simplified*])
by(*simp*)
next
case *False*
with *alters* **have** *S′*: *?S′* = {} **by** *simp*

have *1* : {*xa*. *xa* < *cs* ! *i*
in fst (*Partial_Cost_Model.config′* (λ*s*. *h*, *TS_step_d*) (*init*, *h*)
(*as* @ *x* # *bs* @ *x* # *take i cs*)) ∧
count_list (*take* (*index*
(*snd*
(*Partial_Cost_Model.config′*
(λ*s*. *h*, *TS_step_d*) (*init*, *h*)
(*as* @ *x* # *bs* @ *x* # *take i cs*)))
(*cs* ! *i*))
(*snd*
(*Partial_Cost_Model.config′*
(λ*s*. *h*, *TS_step_d*) (*init*, *h*)

(*as* @ *x* # *bs* @ *x* # *take i cs*)))) *xa* ≤ *1*} = {} **using** *S′* **by**(*simp add*: *rTS_def nth_append*)

show *?thesis*
apply(*subst config_append*)
using *i_in_cs* **apply**(*simp add*: *rTS_def Step_def split_def nth_append*)
apply(*subst TS_step_d_def*)
apply(*simp only*: *1 Let_def*)
by(*simp*)
qed
finally have *?s_Suct3* = *step ?s ?y* (*0*, []) **using** *ss′* **by** *simp*
then have *e*: *?s_Suct3* = *?s* **by**(*simp only*: *step_no_action*)
from *x_before_y_t3* **have** *x* < *cs* ! *i in ?s_Suct3* **unfolding** *e* **unfolding** *s_TS_def* **by** *simp*
with *y_before_x′* **show** *False* **unfolding** *assms*(*1*) **by** *auto*
qed
then have *aha′*: *index* (*snd* (*TSdet init h* (*as* @ *x* # *bs* @ *x* # *cs*) (*Suc* (*Suc* (*length as* + *length bs* + *i*)))))
(*cs* ! *i*) ≠
length (*snd* (*TSdet init h* (*as* @ *x* # *bs* @ *x* # *cs*) (*Suc* (*Suc* (*length as* + *length bs* + *i*)))))
and
aha2: *?S* ≠ {} **by** *auto*

from *fstTS_t3′ assms*(*1*) **have** *is_*: *?is* = (*rev* (*take i cs*)) @ [*x*] @ (*rev bs*) @ [*x*] @ (*rev as*) @ *h* **by** *auto*

have *minlencsi*: *min* (*length cs*) *i* = *i* **using** *i_in_cs* **by** *linarith*

let *?lastoccy*=(*index* (*rev* (*take i cs*) @ *x* # *rev bs* @ *x* # *rev as* @ *h*) (*cs* ! *i*))
have *?y* ∉ *set* (*rev* (*take i cs*)) **using** *firstocc* **by** (*simp add*: *in_set_conv_nth*)
then have *lastoccy*: *?lastoccy* ≥
i + *1* + *length bs* + *1* **using** *ynx ynotinbs minlencsi* **by**(*simp add*: *index_append*)

have *x_nin_S*: *x*∉*?S*
using *is_* **apply**(*simp add*: *split_def nth_append del*: *config′.simps*)
proof (*goal_cases*)
case *1*
have *count_list* (*take ?lastoccy* (*rev* (*take i cs*))) *x* ≤

count_list (*drop* (*length cs* − *i*) (*rev cs*)) *x* **by** (*simp add*: *count_take rev_take*)
also have ... ≤ *count_list* (*rev cs*) *x* **by** (*meson count_drop*)
also have ... = *0* **using** *assms*(*2*) **by**(*simp*)
finally have *count_list* (*take ?lastoccy* (*rev* (*take i cs*))) *x* = *0* **by** *auto*
have
2 ≤
count_list ([*x*] @ *rev bs* @ [*x*]) *x* **by**(*simp*)
also have ... = *count_list* (*take* (*1* + *length bs* + *1*) (*x* # *rev bs* @ *x* # *rev as* @ *h*)) *x* **by** *auto*
also have ... ≤ *count_list* (*take* (*?lastoccy* − *i*) (*x* # *rev bs* @ *x* # *rev as* @ *h*)) *x*
apply(*rule count_take_less*)
using *lastoccy* **by** *linarith*
also have ... ≤ *count_list* (*take ?lastoccy* (*rev* (*take i cs*))) *x*
\+ *count_list* (*take* (*?lastoccy* − *i*) (*x* # *rev bs* @ *x* # *rev as* @ *h*)) *x* **by** *auto*
finally show *?case* **by**(*simp add*: *minlencsi*)
qed

have *Min* (*index ?s* ‘ *?S*) ∈ (*index ?s* ‘ *?S*) **apply**(*rule Min_in*) **using** *aha2* **by** (*simp_all*)
then obtain *z* **where** *zminimal*: *index ?s z* = *Min* (*index ?s* ‘ *?S*)**and** *z_in_S*: *z* ∈ *?S* **by** *auto*
then have *bef*: *z* < (*as* @ [*x*] @ *bs* @ [*x*] @ *cs*) ! (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*) *in ?s*
and *count_list* (*take* (*index ?is* ((*as* @ [*x*] @ *bs* @ [*x*] @ *cs*) ! (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*))) *?is*) *z* ≤ *1* **by**(*blast*)+

with *zminimal* **have** *zbeforey*: *z* < *cs* ! *i in ?s*
and *zatmostonce*: *count_list* (*take* (*index ?is* (*cs* ! *i*)) *?is*) *z* ≤ *1*
and *isminimal*: *index ?s z* = *Min* (*index ?s* ‘ *?S*) **by**(*simp_all add*: *nth_append*)
have *elemins*: *z* ∈ *set ?s* **unfolding** *before_in_def* **by** (*meson zbeforey before_in_setD1*)
then have *zininit*: *z* ∈ *set init*
using *i_in_cs* **by**(*simp add*: *s_TS_set*[*unfolded s_TS_def*] *del*: *config′.simps*)

from *zbeforey* **have** *zbeforey_ind*: *index ?s z* < *index ?s ?y* **unfolding** *before_in_def* **by** *auto*
then have *el_n_y*: *z* ≠ *?y* **by** *auto*

have *el_n_x*: $z \neq x$ **using** *x_nin_S z_in_S* **by** *blast*

{ fix *s q*
have *TS_step_d2*: *TS_step_d s q* =
(*let* V_r={*x. x* < *q in fst s* ∧ *count_list* (*take* (*index* (*snd s*) *q*) (*snd s*)) *x* ≤ *1*}
in ((*if index* (*snd s*) *q* ≠ *length* (*snd s*) ∧ V_r ≠ {}
then index (*fst s*) *q* − *Min* ((*index* (*fst s*)) ' V_r)
else 0,[]),*q*#(*snd s*)))
unfolding *TS_step_d_def*
apply(*cases index* (*snd s*) *q* < *length* (*snd s*))
using *index_le_size* **apply**(*simp split*: *prod.split*) **apply** *blast*
by(*auto simp add*: *index_less_size_conv split*: *prod.split*)
} **note** *alt_chara*=*this*

have *iF*: (*index* (*snd* (*config'* (λ*s. h*, *TS_step_d*) (*init*, *h*) (*as* @ *x* # *bs* @ *x* # *take i cs*))) (*cs* ! *i*)
≠ *length* (*snd* (*config'* (λ*s. h*, *TS_step_d*) (*init*, *h*) (*as* @ *x* # *bs* @ *x* # *take i cs*))) ∧
{*xa. xa* < *cs* ! *i in fst* (*config'* (λ*s. h*, *TS_step_d*) (*init*, *h*) (*as* @ *x* # *bs* @ *x* # *take i cs*)) ∧
count_list
(*take* (*index* (*snd* (*config'* (λ*s. h*, *TS_step_d*) (*init*, *h*) (*as* @ *x* # *bs* @ *x* # *take i cs*))) (*cs* ! *i*))
(*snd* (*Partial_Cost_Model.config'* (λ*s. h*, *TS_step_d*) (*init*, *h*) (*as* @ *x* # *bs* @ *x* # *take i cs*))))
xa
≤ *1*} ≠
{}) = *True* **using** *aha*[*unfolded rTS_def*] *ss' SS'* **by**(*simp add*: *nth_append*)

have *?s_Suct3* = *fst* (*TSdet init h* (*as* @ *x* # *bs* @ *x* # *cs*) (*Suc* (*Suc* (*Suc* (*length as* + *length bs* + *i*)))))
by(*auto simp add*: *s_TS_def*)
also have ... = *step ?s ?y* (*index ?s ?y* − *Min* (*index ?s* ' *?S*), [])
apply(*simp only*: *once*[*unfolded assms*(*1*)])
apply(*simp add*: *Step_def split_def rTS_def del*: *config'.simps*)
apply(*subst alt_chara*)
apply(*simp only*: *Let_def*)
apply(*simp only*: *iF*)
by(*simp add*: *nth_append*)
finally have *?s_Suct3* = *step ?s ?y* (*index ?s ?y* − *Min* (*index ?s* ' *?S*), []) .

with *isminimal* **have** *state_dannach*: *?s_Suct3* = *step ?s ?y* (*index ?s ?y* − *index ?s z*, []) **by** *presburger*

— so y is moved in front of z, that means:
have *yinfrontofz*: *?y* < *z in s_TS init h* σ (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*+*1*)
unfolding *assms*(*1*) *state_dannach* **apply**(*simp add*: *step_def del*: *config'.simps*)
apply(*rule mtf2_q_passes*)
using *i_in_cs assms*(*5*) **apply**(*simp_all add*: *s_TS_distinct*[*unfolded s_TS_def*] *s_TS_set*[*unfolded s_TS_def*])
using *yininit* **apply**(*simp*)
using *zbeforey_ind* **by** *simp*

have *yins*: *?y* ∈ *set ?s*
using *i_in_cs assms*(*3*,*5*) **apply**(*simp_all add*: *s_TS_set*[*unfolded s_TS_def*] *del*: *config'.simps*)
by *fastforce*

have *index ?s_Suct3 ?y* = *index ?s z*
and *index ?s_Suct3 z* = *Suc* (*index ?s z*)
proof −
let *?xs* = (*fst* (*TSdet init h* (*as* @ *x* # *bs* @ *x* # *cs*) (*Suc* (*Suc* (*length as* + *length bs* + *i*)))))
have *setxs*: *set ?xs* = *set init*
apply(*rule s_TS_set*[*unfolded s_TS_def*])
using *i_in_cs* **by** *auto*
then have *yinxs*: *cs* ! *i* ∈ *set ?xs*
apply(*simp add*: *setxs del*: *config'.simps*)
using *assms*(*3*) *i_in_cs* **by** *fastforce*

have *distinctxs*: *distinct ?xs*
apply(*rule s_TS_distinct*[*unfolded s_TS_def*])
using *i_in_cs assms*(*5*) **by** *auto*

let *?n* = (*index*
(*fst* (*TSdet init h* (*as* @ *x* # *bs* @ *x* # *cs*)
(*Suc* (*Suc* (*length as* + *length bs* + *i*)))))
(*cs* ! *i*) −

index
(*fst* (*TSdet init h* (*as* @ *x* # *bs* @ *x* # *cs*)
(*Suc* (*Suc* (*length as* + *length bs* + *i*)))))
z)

have *index* (*mtf2 ?n ?y ?xs*) (*?xs* ! *index ?xs ?y*) = *index ?xs ?y* − *?n*∧
index ?xs ?y − *?n* = *index* (*mtf2 ?n ?y ?xs*) (*?xs* ! *index ?xs ?y*)
apply(*rule mtf2_forward_effect2*)
apply(*fact*)
apply(*fact*)
by *simp*

then have *index* (*mtf2 ?n ?y ?xs*) (*?xs* ! *index ?xs ?y*) = *index ?xs ?y* − *?n* **by** *metis*
also have ... = *index ?s z* **using** *zbeforey_ind* **by** *force*
finally have *A*: *index* (*mtf2 ?n ?y ?xs*) (*?xs* ! *index ?xs ?y*) = *index ?s z* **.**

have *aa*: *index ?xs ?y* − *?n* ≤ *index ?xs z index ?xs z* < *index ?xs ?y*
apply(*simp*)
using *zbeforey_ind* **by** *fastforce*

from *mtf2_forward_effect3'*[*OF yinxs distinctxs aa*]
have *B*: *index* (*mtf2 ?n ?y ?xs*) *z* = *Suc* (*index ?xs z*)
using *elemins yins* **by**(*simp add*: *nth_append split_def del*: *config'.simps*)

show *index ?s_Suct3 ?y* = *index ?s z*
unfolding *state_dannach* **apply**(*simp add*: *step_def nth_append del*: *config'.simps*)
using *A yins* **by**(*simp add*: *nth_append del*: *config'.simps*)

show *index ?s_Suct3 z* = *Suc* (*index ?s z*)
unfolding *state_dannach* **apply**(*simp add*: *step_def nth_append del*: *config'.simps*)
using *B yins* **by**(*simp add*: *nth_append del*: *config'.simps*)
qed

then have *are*: *Suc* (*index ?s_Suct3 ?y*) = *index ?s_Suct3 z* **by** *presburger*

from *are before_in_def y_before_x_Suct3 el_n_x assms*(*1*) **have** *z_before_x*: *z* < *x in ?s_Suct3*
by (*metis Suc_lessI not_before_in yinfrontofz*)

have *xSuct3*: *x*∈*set ?s_Suct3* **using** *assms*(*4*) *i_in_cs* **by**(*simp add*: *s_TS_set*)
have *elSuct3*: *z*∈*set ?s_Suct3* **using** *zininit i_in_cs* **by**(*simp add*: *s_TS_set*)

have *xt3*: *x*∈*set ?s* **apply**(*subst config_config_set*) **by** *fact*

note *elt3*=*elemins*

have *z_s*: *z* < *x in ?s*
proof(*rule ccontr*, *goal_cases*)
case *1*
then have *x* < *z in ?s* **using** *not_before_in*[*OF xt3 elt3*] *el_n_x* **unfolding** *s_TS_def* **by** *blast*
then have *x* < *z in ?s_Suct3*
apply (*simp only*: *state_dannach*)
apply (*simp only*: *step_def*)
apply(*simp add*: *nth_append del*: *config'.simps*)
apply(*rule x_stays_before_y_if_y_not_moved_to_front*)
apply(*subst config_config_set*) **using** *i_in_cs assms*(*3*) **apply** *fastforce*
apply(*subst config_config_distinct*) **using** *assms*(*5*) **apply** *fastforce*
apply(*subst config_config_set*) **using** *assms*(*4*) **apply** *fastforce*
apply(*subst config_config_set*) **using** *zininit* **apply** *fastforce*
using *el_n_y* **apply**(*simp*)
by(*simp*)

then show *False* **using** *z_before_x not_before_in*[*OF xSuct3 elSuct3*] **by** *blast*
qed

have *mind*: (*index ?is* (*cs* ! *i*)) ≥ *i* + *1* + *length bs* + *1* **using** *lastoccy*
using *i_in_cs fstTS_t3'*[*unfolded assms*(*1*)] **by**(*simp add*: *split_def nth_append del*: *config'.simps*)

have *count_list* (*rev* (*take i cs*) @ [*x*] @ *rev bs* @ [*x*]) *z*=
count_list (*take* (*i* + *1* + *length bs* + *1*) *?is*) *z* **unfolding** *is_*
using *el_n_x* **by**(*simp add*: *minlencsi*)

also from *mind* **have** ...
≤ *count_list* (*take* (*index ?is* (*cs* ! *i*)) *?is*) *z*
by(*rule count_take_less*)
also have ... ≤ *1* **using** *zatmostonce* **by** *metis*
finally have *aaa*: *count_list* (*rev* (*take i cs*) @ [*x*] @ *rev bs* @ [*x*]) *z* ≤ *1*
.
with *el_n_x* **have** *count_list bs z* + *count_list* (*take i cs*) *z* ≤ *1*
by(*simp*)
moreover have *count_list* (*take* (*Suc i*) *cs*) *z* = *count_list* (*take i cs*) *z*
using *i_in_cs el_n_y* **by**(*simp add*: *take_Suc_conv_app_nth*)
ultimately have *aaaa*: *count_list bs z* + *count_list* (*take* (*Suc i*) *cs*) *z* ≤ *1* **by** *simp*

have *z_occurs_once_in_cs*: *count_list* (*take* (*Suc i*) *cs*) *z* = *1*
proof (*rule ccontr*, *goal_cases*)
case *1*
with *aaaa* **have** *atmost1*: *count_list bs z* ≤ *1* **and** *count_list* (*take* (*Suc i*) *cs*) *z* = *0* **by** *force*+
have *yeah*: *z* ∉ *set* (*take* (*Suc i*) *cs*) **apply**(*rule count_notin2*) **by** *fact*

— now we know that x is in front of z after 2nd request to x, and that z is not requested any more, that means it stays behind x, which leads to a contradiction with *z_before_x*

have *xin123*: *x* ∈ *set* (*s_TS init h* ((*as* @ [*x*] @ *bs* @ [*x*]) @ (*take* (*i*+*1*) *cs*)) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + (*i*+*1*)))
using *i_in_cs assms*(*4*) **by**(*simp add*: *s_TS_set*)
have *zin123*: *z* ∈ *set* (*s_TS init h* ((*as* @ [*x*] @ *bs* @ [*x*]) @ (*take* (*i*+*1*) *cs*)) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + (*i*+*1*)))
using *i_in_cs elemins* **by**(*simp add*: *s_TS_set del*: *config'.simps*)

have *x* < *z in s_TS init h* ((*as* @ [*x*] @ *bs* @ [*x*]) @ (*take* (*i*+*1*) *cs*)) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + (*i* + *1*))
apply(*rule TS_mono*)
apply(*rule xgoestofront*)
apply(*fact*) **using** *el_n_x* **apply**(*simp*) **apply**(*fact*)
using *i_in_cs* **apply**(*simp*)
using *yeah i_in_cs length_take nth_mem*
apply (*metis Suc_eq_plus1 Suc_leI min_absorb2*)
using *set_take_subset assms*(*2*) **apply** *fast*
using *assms i_in_cs* **apply**(*simp_all*) **using** *set_take_subset* **by** *fast*
then have *ge*: ¬ *z* < *x in s_TS init h* ((*as* @ [*x*] @ *bs* @ [*x*]) @ (*take* (*i*+*1*) *cs*)) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + (*i*+*1*))

using *not_before_in*[*OF zin123 xin123*] *el_n_x* **by** *blast*

have *s_TS init h* ((*as* @ [*x*] @ *bs* @ [*x*]) @ *cs*) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + (*i*+*1*))
= *s_TS init h* ((*as* @ [*x*] @ *bs* @ [*x*] @ (*take* (*i*+*1*) *cs*)) @ (*drop* (*i*+*1*) *cs*)) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + (*i*+*1*)) **by** *auto*
also have ...
= *s_TS init h* (*as* @ [*x*] @ *bs* @ [*x*] @ (*take* (*i*+*1*) *cs*)) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + (*i*+*1*))
apply(*rule s_TS_append*)
using *i_in_cs* **by**(*simp*)
finally have *aaa*: *s_TS init h* ((*as* @ [*x*] @ *bs* @ [*x*]) @ *cs*) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + (*i*+*1*))
= *s_TS init h* (*as* @ [*x*] @ *bs* @ [*x*] @ (*take* (*i*+*1*) *cs*)) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + (*i*+*1*)) **.**

from *ge z_before_x* **show** *False* **unfolding** *assms*(*1*) **using** *aaa* **by** *auto*
qed
from *z_occurs_once_in_cs* **have** *kinSuci*: *z* ∈ *set* (*take* (*Suc i*) *cs*) **by** (*metis One_nat_def count_notin n_not_Suc_n*)
then have *zincs*: *z*∈*set cs* **using** *set_take_subset* **by** *fast*
from *z_occurs_once_in_cs* **obtain** *k* **where** *k_def*: *k*=*index* (*take* (*Suc i*) *cs*) *z* **by** *blast*

then have *k*=*index cs z* **using** *kinSuci* **by** (*simp add*: *index_take_if_set*)
then have *zcsk*: *z* = *cs*!*k* **using** *zincs* **by** *simp*

have *era*: *cs* ! *index* (*take* (*Suc i*) *cs*) *z* = *z* **using** *kinSuci in_set_takeD index_take_if_set* **by** *fastforce*
have *ki*: *k*<*i* **unfolding** *k_def* **using** *kinSuci el_n_y*
by (*metis i_in_cs index_take index_take_if_set le_neq_implies_less not_less_eq_eq yes*)
have *zmustbebeforex*: *cs*!*k* < *x in ?s*
unfolding *k_def era* **by** (*fact z_s*)

— before the request to z, x is in front of z, analog zu oben, vllt generell machen?

— element z does not occur between t1 and position k
have *z_notinbs*: *cs* ! *k* $\notin$ *set bs*
proof −
from *z_occurs_once_in_cs aaaa* **have** *count_list bs z* = *0* **by** *auto*
then show *?thesis* **using** *zcsk count_notin2* **by** *metis*
qed

have *count_list bs z* ≤ *1* **using** *aaaa* **by** *linarith*
with *xgoestofront*[*OF zininit el_n_x*[*symmetric*]] **have** *xbeforez*: *x* < *z in s_TS init h* (*as* @ [*x*] @ *bs* @ [*x*]) (*length* (*as* @ [*x*] @ *bs* @ [*x*])) **by** *auto*

obtain *cs1 cs2* **where** *v*: *cs1* @ *cs2* = *cs* **and** *cs1*: *cs1* = *take* (*Suc k*) *cs* **and** *cs2*: *cs2* = *drop* (*Suc k*) *cs* **by** *auto*

have *z_firstocc*: ∀ *j*<*k*. *cs* ! *j* ≠ *cs* ! *k*
and *z_lastocc*: ∀ *j*<*i*−*k*−*1*. *cs2* ! *j* ≠ *cs* ! *k*
proof (*safe*, *goal_cases*)
case (*1 j*)
with *ki i_in_cs* **have** *2*: *j* < *length* (*take k cs*) **by** *auto*
have *un1*: (*take* (*Suc i*) *cs*)!*k* = *cs*!*k* **apply**(*rule nth_take*) **using** *ki* **by** *auto*
have *un2*: (*take k cs*)!*j* = *cs*!*j* **apply**(*rule nth_take*) **using** *1*(*1*) *ki* **by** *auto*

from *i_in_cs ki* **have** *f1*: *k* < *length* (*take* (*Suc i*) *cs*) **by** *auto*
then have (*take* (*Suc i*) *cs*) = (*take k* (*take* (*Suc i*) *cs*)) @ (*take* (*Suc i*) *cs*)!*k* # (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
by(*rule id_take_nth_drop*)
also have (*take k* (*take* (*Suc i*) *cs*)) = *take k cs* **using** *i_in_cs ki* **by** (*simp add*: *min_def*)
also have ... = (*take j* (*take k cs*)) @ (*take k cs*)!*j* # (*drop* (*Suc j*) (*take k cs*))
using *2* **by**(*rule id_take_nth_drop*)
finally have *take* (*Suc i*) *cs*
= (*take j* (*take k cs*)) @ [(*take k cs*)!*j*] @ (*drop* (*Suc j*) (*take k cs*))
@ [(*take* (*Suc i*) *cs*)!*k*] @ (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
by(*simp*)
then have *A*: *take* (*Suc i*) *cs*
= (*take j* (*take k cs*)) @ [*cs*!*j*] @ (*drop* (*Suc j*) (*take k cs*))
@ [*cs*!*k*] @ (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
unfolding *un1 un2* **by** *simp*
have *count_list* ((*take j* (*take k cs*)) @ [*cs*!*j*] @ (*drop* (*Suc j*) (*take k*

cs))
@ [*cs*!*k*] @ (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))) *z* ≥ *2*
using *zcsk 1*(*2*) **by**(*simp*)
with *A* **have** *count_list* (*take* (*Suc i*) *cs*) *z* ≥ *2* **by** *auto*
with *z_occurs_once_in_cs* **show** *False* **by** *auto*
next
case (*2 j*)
then have *1*: *Suc k*+*j* < *i* **by** *auto*
then have *f2*: *j* < *length* (*drop* (*Suc k*) (*take* (*Suc i*) *cs*)) **using** *i_in_cs* **by** *simp*
have *3*: (*drop* (*Suc k*) (*take* (*Suc i*) *cs*)) = *take j* (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
@ (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))! *j*
# *drop* (*Suc j*) (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
using *f2* **by**(*rule id_take_nth_drop*)
have (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))! *j* = (*take* (*Suc i*) *cs*) ! (*Suc k*+*j*)
apply(*rule nth_drop*) **using** *i_in_cs 1* **by** *auto*
also have . . . = *cs* ! (*Suc k*+*j*) **apply**(*rule nth_take*) **using** *1* **by** *auto*
finally have *4*: (*drop* (*Suc k*) (*take* (*Suc i*) *cs*)) = *take j* (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
@ *cs*! (*Suc k* +*j*)
# *drop* (*Suc j*) (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
using *3* **by** *auto*
have *5*: *cs2* ! *j* = *cs*! (*Suc k* +*j*) **unfolding** *cs2*
apply(*rule nth_drop*) **using** *i_in_cs 1* **by** *auto*

from *4 5 2*(*2*) **have** *6*: (*drop* (*Suc k*) (*take* (*Suc i*) *cs*)) = *take j* (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
@ *cs*! *k*
# *drop* (*Suc j*) (*drop* (*Suc k*) (*take* (*Suc i*) *cs*)) **by** *auto*

from *i_in_cs ki* **have** *1*: *k* < *length* (*take* (*Suc i*) *cs*) **by** *auto*
then have *7*: (*take* (*Suc i*) *cs*) = (*take k* (*take* (*Suc i*) *cs*)) @ (*take* (*Suc i*) *cs*)!*k* # (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
by(*rule id_take_nth_drop*)
have *9*: (*take* (*Suc i*) *cs*)!*k* = *z* **unfolding** *zcsk* **apply**(*rule nth_take*) **using** *ki* **by** *auto*
from *6 7* **have** *A*: (*take* (*Suc i*) *cs*) = (*take k* (*take* (*Suc i*) *cs*)) @ *z* # *take j* (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
@ *z*
# *drop* (*Suc j*) (*drop* (*Suc k*) (*take* (*Suc

i) *cs*)) **using** *ki 9* **by** *auto*

have *count_list* ((*take k* (*take* (*Suc i*) *cs*)) @ *z* # *take j* (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))
@ *z*
# *drop* (*Suc j*) (*drop* (*Suc k*) (*take* (*Suc i*) *cs*))) *z*
≥ *2*
by(*simp*)
with *A* **have** *count_list* (*take* (*Suc i*) *cs*) *z* ≥ *2* **by** *auto*
with *z_occurs_once_in_cs* **show** *False* **by** *auto*
qed

have *k_in_cs*: *k* < *length cs* **using** *ki i_in_cs* **by** *auto*
with *cs1* **have** *lenkk*: *length cs1* = *k*+*1* **by** *auto*
from *k_in_cs* **have** *mincsk*: *min* (*length cs*) (*Suc k*) = *Suc k* **by** *auto*

have *s_TS init h* (((*as*@[*x*]@*bs*@[*x*])@*cs1*) @ *cs2*) (*length* (*as*@[*x*]@*bs*@[*x*])+*k*+*1*)
= *s_TS init h* ((*as*@[*x*]@*bs*@[*x*])@(*cs1*)) (*length* (*as*@[*x*]@*bs*@[*x*])+*k*+*1*)
apply(*rule s_TS_append*)
using *cs1 cs2 k_in_cs* **by**(*simp*)
then have *spliter*: *s_TS init h* ((*as*@[*x*]@*bs*@[*x*])@(*cs1*)) (*length* (*as*@[*x*]@*bs*@[*x*]@(*cs1*)))
= *s_TS init h* ((*as*@[*x*]@*bs*@[*x*])@*cs*) (*length* (*as*@[*x*]@*bs*@[*x*])+*k*+*1*)

using *lenkk v cs1* **apply**(*auto*) **by** (*simp add*: *add.commute add.left_commute*)

from *cs2* **have** *length cs2* = *length cs* − (*Suc k*) **by** *auto*

have *notxbeforez*: ~ *x* < *z in s_TS init h σ* (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *k* + *1*)
proof (*rule ccontr*, *goal_cases*)
case *1*
then have *a*: *x* < *z in s_TS init h* ((*as*@[*x*]@*bs*@[*x*])@(*cs1*)) (*length* (*as*@[*x*]@*bs*@[*x*]@(*cs1*)))
unfolding *spliter assms*(*1*) **by** *auto*

have *41*: *x* ∈ *set*(*s_TS init h* ((*as* @ [*x*] @ *bs* @ [*x*]) @ *cs*) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*))
using *i_in_cs assms*(*4*) **by**(*simp add*: *s_TS_set*)
have *42*: *z* ∈ *set*(*s_TS init h* ((*as* @ [*x*] @ *bs* @ [*x*]) @ *cs*) (*length* (*as* @ [*x*] @ *bs* @ [*x*]) + *i*))
using *i_in_cs zininit* **by**(*simp add*: *s_TS_set*)

```
  have rewr: s_TS init h ((as@[x]@bs@[x]@cs1)@cs2) (length (as@[x]@bs@[x]@cs1)+(i−k−1))
=
            s_TS init h (as@[x]@bs@[x]@cs) (length (as@[x]@bs@[x])+i)
                 using cs1 v ki  apply(simp add: mincsk) by (simp add:
add.commute add.left_commute)

  have x < z in s_TS init h ((as@[x]@bs@[x]@cs1)@cs2) (length (as@[x]@bs@[x]@cs1)+(i−k−1))
     apply(rule TS_mono)
       using a apply(simp)
       using cs2 i_in_cs ki v cs1 apply(simp)
       using z_lastocc zcsk apply(simp)
       using v assms(2) apply force
       using assms by(simp_all add: cs1 cs2)


  from zmustbebeforex this[unfolded rewr ] el_n_x zcsk 41 42 not_before_in
show False
     unfolding s_TS_def  by fastforce
  qed

  have 1: k < length cs
               (∀j<k. cs ! j ≠ cs ! k)
               cs ! k ≠ x cs ! k ∉ set bs
          ~ x < z in s_TS init h σ (length (as @ [x] @ bs @ [x]) + k + 1)
     apply(safe)
        using ki i_in_cs apply(simp)
        using z_firstocc apply(simp)
        using assms(2) ki i_in_cs apply(fastforce)
        using z_notinbs apply(simp)
        using notxbeforez by auto



  show ?case apply(simp only: ex_nat_less_eq)
     apply(rule bexI[where x=k])
       using 1 zcsk apply(simp)
       using ki by simp
qed

lemma nopaid: snd (fst (TS_step_d s q)) = [] unfolding TS_step_d_def
by simp

lemma staysuntouched:
```

assumes *d*[*simp*]: *distinct* (*fst S*)
and *x*: *x* ∈ *set* (*fst S*)
and *y*: *y* ∈ *set* (*fst S*)
shows *set qs* ⊆ *set* (*fst S*) ⟹ *x* ∉ *set qs* ⟹ *y* ∉ *set qs*
⟹ *x* < *y in fst* (*config′* (*rTS* []) *S qs*) = *x* < *y in fst S*
proof(*induct qs rule*: *rev_induct*)
case (*snoc q qs*)
have *x* < *y in fst* (*config′* (*rTS* []) *S* (*qs* @ [*q*])) =
x < *y in fst* (*config′* (*rTS* []) *S qs*)
apply(*simp add*: *config′_snoc Step_def split_def step_def rTS_def nopaid*)
apply(*rule xy_relativorder_mtf2*)
using *snoc* **by**(*simp_all add*: *x y*)
also have . . . = *x* < *y in fst S*
apply(*rule snoc*)
using *snoc* **by** *simp_all*
finally show *?case* **.**
qed *simp*

lemma *staysuntouched′*:
assumes *d*[*simp*]: *distinct init*
and *x*: *x* ∈ *set init*
and *y*: *y* ∈ *set init*
and *set qs* ⊆ *set init*
and *x* ∉ *set qs* **and** *y* ∉ *set qs*
shows *x* < *y in fst* (*config* (*rTS* []) *init qs*) = *x* < *y in init*
proof −
let *?S*=(*init*, *fst* (*rTS* []) *init*)
have *x* < *y in fst* (*config′* (*rTS* []) *?S qs*) = *x* < *y in fst ?S*
apply(*rule staysuntouched*)
using *assms* **by**(*simp_all*)
then show *?thesis* **by** *simp*
qed

lemma *projEmpty*: *Lxy qs S* = [] ⟹ *x* ∈ *S* ⟹ *x* ∉ *set qs*
unfolding *Lxy_def* **by** (*metis filter_empty_conv*)

lemma *Lxy_index_mono*:
assumes *x*∈*S y*∈*S*
and *index xs x* < *index xs y*
and *index xs y* < *length xs*
and *x*≠*y*
shows *index* (*Lxy xs S*) *x* < *index* (*Lxy xs S*) *y*
proof −

from *assms* **have** *ij*: *index xs x* < *index xs y*
and *xinxs*: *index xs x* < *length xs*
and *yinxs*: *index xs y* < *length xs* **by** *auto*
then have *inset*: *x*∈*set xs y*∈*set xs* **using** *index_less_size_conv* **by** *fast*+
from *xinxs* **obtain** *a as* **where** *dec1*: *a* @ [*xs*!*index xs x*] @ *as* = *xs*
and *a*: *a* = *take* (*index xs x*) *xs* **and** *as* = *drop* (*Suc* (*index xs x*)) *xs*
and *length_a*: *length a* = *index xs x* **and** *length_as*: *length as* = *length xs* − *index xs x*− *1*
using *id_take_nth_drop* **by** *fastforce*
have *index xs y*≥*length* (*a* @ [*xs*!*index xs x*]) **using** *length_a ij* **by** *auto*
then have ((*a* @ [*xs*!*index xs x*]) @ *as*) ! *index xs y* = *as* ! (*index xs y*−*length* (*a* @ [*xs* ! *index xs x*])) **using** *nth_append*[**where** *xs*=*a* @ [*xs*!*index xs x*] **and** *ys*=*as*]
by(*simp*)
then have *xsj*: *xs* ! *index xs y* = *as* ! (*index xs y*−*index xs x*−*1*) **using** *dec1 length_a* **by** *auto*
have *las*: (*index xs y*−*index xs x*−*1*) < *length as* **using** *length_as yinxs ij* **by** *simp*
obtain *b c* **where** *dec2*: *b* @ [*xs*!*index xs y*] @ *c* = *as*
and *b* = *take* (*index xs y*−*index xs x*−*1*) *as c*=*drop* (*Suc* (*index xs y*−*index xs x*−*1*)) *as*
and *length_b*: *length b* = *index xs y*−*index xs x*−*1* **using** *id_take_nth_drop*[*OF las*] *xsj* **by** *force*

have *xs_dec*: *a* @ [*xs*!*index xs x*] @ *b* @ [*xs*!*index xs y*] @ *c* = *xs* **using** *dec1 dec2* **by** *auto*

then have *Lxy xs S* = *Lxy* (*a* @ [*xs*!*index xs x*] @ *b* @ [*xs*!*index xs y*] @ *c*) *S*
by(*simp add*: *xs_dec*)
also have . . . = *Lxy a S* @ *Lxy* [*x*] *S* @ *Lxy b S* @ *Lxy* [*y*] *S* @ *Lxy c S*
by(*simp add*: *Lxy_append Lxy_def assms inset*)
finally have *gr*: *Lxy xs S* = *Lxy a S* @ [*x*] @ *Lxy b S* @ [*y*] @ *Lxy c S*
using *assms* **by**(*simp add*: *Lxy_def*)

have *y* ∉ *set* (*take* (*index xs x*) *xs*)
apply(*rule index_take*) **using** *assms* **by** *simp*
then have *y* ∉ *set* (*Lxy* (*take* (*index xs x*) *xs*) *S*)
apply(*subst Lxy_set_filter*) **by** *blast*
with *a* **have** *ynot*: *y* ∉ *set* (*Lxy a S*) **by** *simp*
have *index* (*Lxy xs S*) *y* =
index (*Lxy a S* @ [*x*] @ *Lxy b S* @ [*y*] @ *Lxy c S*) *y*
by(*simp add*: *gr*)

```
  also have ... ≥ length (Lxy a S) + 1
    using assms(5) ynot by(simp add: index_append)
  finally have 1: index (Lxy xs S) y ≥ length (Lxy a S) + 1 .

  have index (Lxy xs S) x = index (Lxy a S @ [x] @ Lxy b S @ [y] @ Lxy
c S) x
    by (simp add: gr)
  also have ... ≤ length (Lxy a S)
    apply(simp add: index_append)
    apply(subst index_less_size_conv[symmetric]) by simp
  finally have 2: index (Lxy xs S) x ≤ length (Lxy a S) .

  from 1 2 show ?thesis by linarith
qed

lemma proj_Cons:
  assumes filterd_cons: Lxy qs S = a#as
    and a_filter: a∈S
  obtains pre suf where qs = pre @ [a] @ suf and ⋀x. x ∈ S ⟹ x ∉ set
pre
                and Lxy suf S = as
proof -
  have set (Lxy qs S) ⊆ set qs using Lxy_set_filter by fast
  with filterd_cons have a_inq: a ∈ set qs by simp
  then have index qs a < length qs by(simp)
  { fix e
    assume eS:e∈S
    assume e≠a
    have index qs a ≤ index qs e
    proof (rule ccontr)
      assume ¬ index qs a ≤ index qs e
      then have 1: index qs e < index qs a by simp
      have 0: index (Lxy qs S) a = 0 unfolding filterd_cons by simp
      have 2: index (Lxy qs S) e < index (Lxy qs S) a
        apply(rule Lxy_index_mono)
          by(fact)+
      from 0 2 show False by linarith
    qed
  } note atfront=this


  let ?lastInd=index qs a
  have qs = take ?lastInd qs @ qs!?lastInd # drop (Suc ?lastInd) qs
    apply(rule id_take_nth_drop)
```

    using *a_inq* **by** *simp*
  also have . . . = *take ?lastInd qs* @ [*a*] @ *drop* (*Suc ?lastInd*) *qs*
   using *a_inq* **by** *simp*
  finally have *split*: *qs* = *take ?lastInd qs* @ [*a*] @ *drop* (*Suc ?lastInd*) *qs* **.**

  have *nothingin*: ⋀*s*. *s*∈*S* ⟹ *s* ∉ *set* (*take ?lastInd qs*)
   apply(*rule index_take*)
   apply(*case_tac a*=*s*)
    apply(*simp*)
   by (*rule atfront*) *simp_all*
  then have *set* (*Lxy* (*take ?lastInd qs*) *S*) = {}
   apply(*subst Lxy_set_filter*) **by** *blast*
  then have *emptyPre*: *Lxy* (*take ?lastInd qs*) *S* = [] **by** *blast*

  have *a*#*as* = *Lxy qs S*
   using *filterd_cons* **by** *simp*
  also have . . . = *Lxy* (*take ?lastInd qs* @ [*a*] @ *drop* (*Suc ?lastInd*) *qs*) *S*
   using *split* **by** *simp*
  also have . . . = *Lxy* (*take ?lastInd qs*) *S* @ (*Lxy* [*a*] *S*) @ *Lxy* (*drop* (*Suc ?lastInd*) *qs*) *S*
   by(*simp add*: *Lxy_append Lxy_def*)
  also have . . . = *a*#*Lxy* (*drop* (*Suc ?lastInd*) *qs*) *S*
   unfolding *emptyPre* **by**(*simp add*: *Lxy_def a_filter*)
  finally have *suf*: *Lxy* (*drop* (*Suc ?lastInd*) *qs*) *S* = *as* **by** *simp*

  from *split nothingin suf* **show** *?thesis* **..**
qed

lemma *Lxy_rev*: *rev* (*Lxy qs S*) = *Lxy* (*rev qs*) *S*
apply(*induct qs*)
  by(*simp_all add*: *Lxy_def*)

lemma *proj_Snoc*:
  assumes *filterd_cons*: *Lxy qs S* = *as*@[*a*]
   and *a_filter*: *a*∈*S*
  obtains *pre suf* **where** *qs* = *pre* @ [*a*] @ *suf* **and** ⋀*x*. *x* ∈ *S* ⟹ *x* ∉ *set suf*
         and *Lxy pre S* = *as*
proof −
  have *Lxy* (*rev qs*) *S* = *rev* (*Lxy qs S*) **by**(*simp add*: *Lxy_rev*)
  also have . . . = *a*#(*rev as*) **unfolding** *filterd_cons* **by** *simp*
  finally have *Lxy* (*rev qs*) *S* = *a* # (*rev as*) **.**
  with *a_filter*

obtain *pre′ suf′* **where** *1*: *rev qs* = *pre′* @[*a*] @ *suf′*
and *2*: ⋀*x*. *x* ∈ *S* ⟹ *x* ∉ *set pre′*
and *3*: *Lxy suf′ S* = *rev as*
using *proj_Cons* **by** *metis*
have *qs* = *rev* (*rev qs*) **by** *simp*
also have ... = *rev suf′* @ [*a*] @ *rev pre′* **using** *1* **by** *simp*
finally have *a1*: *qs* = *rev suf′* @ [*a*] @ *rev pre′* **.**

have *Lxy* (*rev suf′*) *S* = *rev* (*Lxy suf′ S*) **by**(*simp add*: *Lxy_rev*)
also have ... = *as* **using** *3* **by** *simp*
finally have *a3*: *Lxy* (*rev suf′*) *S* = *as* **.**

have *a2*: ⋀*x*. *x* ∈ *S* ⟹ *x* ∉ *set* (*rev pre′*) **using** *2* **by** *simp*

from *a1 a2 a3* **show** *?thesis* **..**
qed

lemma *sndTSconfig′*: *snd* (*config′* (*rTS initH*) (*init*,[]) *qs*) = *rev qs* @ []
apply(*induct qs rule*: *rev_induct*)
apply(*simp add*: *rTS_def*)
by(*simp add*: *split_def TS_step_d_def config′_snoc Step_def rTS_def*)

lemma *projxx*:
fixes *e a bs*
assumes *axy*: *a*∈{*x*,*y*}
assumes *ane*: *a*≠*e*
assumes *exy*: *e*∈{*x*,*y*}
assumes *add*: *f*∈{[],[*e*]}
assumes *bsaxy*: *set* (*bs* @ [*a*] @ *f*) ⊆ {*x*,*y*}
assumes *Lxyinitxy*: *Lxy init* {*x*, *y*} ∈ {[*x*,*y*],[*y*,*x*]}
shows *a* < *e in fst* ($config_p$ (*rTS* []) (*Lxy init* {*x*, *y*}) ((*bs* @ [*a*] @ *f*) @ [*a*]))
proof −
have *aexy*: {*a*,*e*}={*x*,*y*} **using** *exy axy ane* **by** *blast*

let *?h*=*snd* (*Partial_Cost_Model.config′* (λ*s*. [], *TS_step_d*)
(*Lxy init* {*x*, *y*}, []) (*bs* @ *a* # *f*))
have *history*: *?h* = (*rev f*)@*a*#(*rev bs*)
using *sndTSdet*[*of length* (*bs*@*a*#*f*) *bs*@*a*#*f*, *unfolded rTS_def*] **by**(*simp*)

{ **fix** *xs s*
assume *sinit*: *s*:{[*a*,*e*],[*e*,*a*]}
assume *set xs* ⊆ {*a*,*e*}

```
    then have fst (config' (λs. [], TS_step_d) (s, []) xs) ∈ {[a,e], [e,a]}
      apply (induct xs rule: rev_induct)
        using sinit apply(simp)
       apply(subst config'_append2)
       apply(simp only: Step_def config'.simps Let_def split_def fst_conv)
       apply(rule stepxy) by simp_all
   } note staysae=this

  have opt: fst (config' (λs. [], TS_step_d)
                                  (Lxy init {x, y}, []) (bs @ [a] @ f)) ∈ {[a,e],
[e,a]}
    apply(rule staysae)
      using Lxyinitxy exy axy ane apply fast
      unfolding aexy by(fact bsaxy)

  have contr: (∀x. 0 < (if e = x then 0 else index [a] x + 1)) = False
  proof (rule ccontr, goal_cases)
    case 1
    then have ⋀x. 0 < (if e = x then 0 else index [a] x + 1) by simp
    then have 0 < (if e = e then 0 else index [a] e + 1) by blast
    then have 0<0 by simp
    then show False by auto
  qed


  show a < e in fst (config_p (rTS []) (Lxy init {x, y}) ((bs @ [a] @ f) @
[a]))
      apply(subst config_append)
      apply(simp add: rTS_def Step_def split_def)
      apply(subst TS_step_d_def)
      apply(simp only: history)
      using opt ane add
      apply(auto simp: step_def)
           apply(simp add: before_in_def)
          apply(simp add: before_in_def)
         apply(simp add: before_in_def contr)
        apply(simp add: mtf2_def swap_def before_in_def)
       apply(auto simp add: before_in_def contr)
     apply (metis One_nat_def add_is_1 count_list.simps(1) le_Suc_eq)
      by(simp add: mtf2_def swap_def)
qed

lemma oneposs:
   assumes set xs = {x,y}
```

assumes *x≠y*
assumes *distinct xs*
assumes *True*: *x<y in xs*
shows *xs* = [*x*,*y*]
proof –
from *assms* **have** *len2*: *length xs* = *2* **using** *distinct_card*[*OF assms*(*3*)] **by** *fastforce*
from *True* **have** *index xs x* < *index xs y index xs y* < *length xs* **unfolding** *before_in_def* **using** *assms*
by *simp_all*
then have *f*: *index xs x* = *0* ∧ *index xs y* = *1* **using** *len2* **by** *linarith*
have *xs* = *take 1 xs* @ *xs*!*1* # *drop* (*Suc 1*) *xs*
apply(*rule id_take_nth_drop*) **using** *len2* **by** *simp*
also have ... = *take 1 xs* @ [*xs*!*1*] **using** *len2* **by** *simp*
also have *take 1 xs* = *take 0* (*take 1 xs*) @ (*take 1 xs*)!*0* # *drop* (*Suc 0*) (*take 1 xs*)
apply(*rule id_take_nth_drop*) **using** *len2* **by** *simp*
also have ... = [*xs*!*0*] **by**(*simp*)
finally have *xs* = [*xs*!*0*, *xs*!*1*] **by** *simp*
also have ... = [*xs*!(*index xs x*), *xs*!*index xs y*] **using** *f* **by** *simp*
also have ... = [*x*,*y*] **using** *assms* **by**(*simp*)
finally show *xs* = [*x*,*y*] **.**
qed

lemma *twoposs*:
assumes *set xs* = {*x*,*y*}
assumes *x≠y*
assumes *distinct xs*
shows *xs* ∈ {[*x*,*y*], [*y*,*x*]}
proof (*cases x<y in xs*)
case *True*
from *assms* **have** *len2*: *length xs* = *2* **using** *distinct_card*[*OF assms*(*3*)] **by** *fastforce*
from *True* **have** *index xs x* < *index xs y index xs y* < *length xs* **unfolding** *before_in_def* **using** *assms*
by *simp_all*
then have *f*: *index xs x* = *0* ∧ *index xs y* = *1* **using** *len2* **by** *linarith*
have *xs* = *take 1 xs* @ *xs*!*1* # *drop* (*Suc 1*) *xs*
apply(*rule id_take_nth_drop*) **using** *len2* **by** *simp*
also have ... = *take 1 xs* @ [*xs*!*1*] **using** *len2* **by** *simp*
also have *take 1 xs* = *take 0* (*take 1 xs*) @ (*take 1 xs*)!*0* # *drop* (*Suc 0*) (*take 1 xs*)
apply(*rule id_take_nth_drop*) **using** *len2* **by** *simp*
also have ... = [*xs*!*0*] **by**(*simp*)

finally have *xs* = [*xs*!*0*, *xs*!*1*] **by** *simp*
also have ... = [*xs*!(*index xs x*), *xs*!*index xs y*] **using** *f* **by** *simp*
also have ... = [*x*,*y*] **using** *assms* **by**(*simp*)
finally have *xs* = [*x*,*y*] **.**
then show *?thesis* **by** *simp*
next
case *False*
from *assms* **have** *len2*: *length xs* = *2* **using** *distinct_card*[*OF assms*(*3*)]
by *fastforce*
from *False* **have** *y*<*x in xs* **using** *not_before_in assms*(*1*,*2*) **by** *fastforce*
then have *index xs y* < *index xs x index xs x* < *length xs* **unfolding** *before_in_def* **using** *assms*
by *simp_all*
then have *f*: *index xs y* = *0* ∧ *index xs x* = *1* **using** *len2* **by** *linarith*
have *xs* = *take 1 xs* @ *xs*!*1* # *drop* (*Suc 1*) *xs*
apply(*rule id_take_nth_drop*) **using** *len2* **by** *simp*
also have ... = *take 1 xs* @ [*xs*!*1*] **using** *len2* **by** *simp*
also have *take 1 xs* = *take 0* (*take 1 xs*) @ (*take 1 xs*)!*0* # *drop* (*Suc 0*) (*take 1 xs*)
apply(*rule id_take_nth_drop*) **using** *len2* **by** *simp*
also have ... = [*xs*!*0*] **by**(*simp*)
finally have *xs* = [*xs*!*0*, *xs*!*1*] **by** *simp*
also have ... = [*xs*!(*index xs y*), *xs*!*index xs x*] **using** *f* **by** *simp*
also have ... = [*y*,*x*] **using** *assms* **by**(*simp*)
finally have *xs* = [*y*,*x*] **.**
then show *?thesis* **by** *simp*
qed

lemma *TS_pairwise'*: **assumes** *qs* ∈ {*xs*. *set xs* ⊆ *set init*}
(*x*, *y*) ∈ {(*x*, *y*). *x* ∈ *set init* ∧ *y* ∈ *set init* ∧ *x* ≠ *y*}
x ≠ *y distinct init*
shows *Pbefore_in x y* (*embed* (*rTS* [])) *qs init* =
Pbefore_in x y (*embed* (*rTS* [])) (*Lxy qs* {*x*, *y*}) (*Lxy init* {*x*, *y*})
proof −
from *assms* **have** *xyininit*: {*x*, *y*} ⊆ *set init*
and *qsininit*: *set qs* ⊆ *set init* **by** *auto*
note *dinit*=*assms*(*4*)
from *assms* **have** *xny*: *x*≠*y* **by** *simp*
have *Lxyinitxy*: *Lxy init* {*x*, *y*} ∈ {[*x*, *y*], [*y*, *x*]}
apply(*rule twoposs*)
apply(*subst Lxy_set_filter*) **using** *xyininit* **apply** *fast*
using *xny Lxy_distinct*[*OF dinit*] **by** *simp_all*

have *lq_s*: *set* (*Lxy qs* {*x*, *y*}) ⊆ {*x*,*y*} **by** (*simp add*: *Lxy_set_filter*)

let *?pH* = *snd* (*config*$_p$ (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}))
have *?pH* =*snd* (*TSdet* (*Lxy init* {*x*, *y*}) [] (*Lxy qs* {*x*, *y*}) (*length* (*Lxy qs* {*x*, *y*})))
by(*simp*)
also have ... = *rev* (*take* (*length* (*Lxy qs* {*x*, *y*})) (*Lxy qs* {*x*, *y*})) @ []
apply(*rule sndTSdet*) **by** *simp*
finally have *pH*: *?pH* = *rev* (*Lxy qs* {*x*, *y*}) **by** *simp*

let *?pQs* = (*Lxy qs* {*x*, *y*})

have *A*: *x* < *y in fst* (*config*$_p$ (*rTS* []) *init qs*)
= *x* < *y in fst* (*config*$_p$ (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}))
proof(*cases ?pQs rule*: *rev_cases*)
case *Nil*
then have *xqs*: *x* ∉ *set qs* **and** *yqs*: *y* ∉ *set qs* **by**(*simp_all add*: *projEmpty*)
have *x* < *y in fst* (*config*$_p$ (*rTS* []) *init qs*)
= *x* < *y in init* **apply**(*rule staysuntouched′*) **using** *assms xqs yqs* **by**(*simp_all*)
also have ... = *x* < *y in fst* (*config*$_p$ (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}))
unfolding *Nil* **apply**(*simp*) **apply**(*rule Lxy_mono*) **using** *xyininit dinit* **by**(*simp_all*)
finally show *?thesis* **.**
next
case (*snoc as a*)
then have *a*∈*set* (*Lxy qs* {*x*, *y*}) **by** (*simp*)
then have *axy*: *a*∈{*x*,*y*} **by**(*simp add*: *Lxy_set_filter*)
with *xyininit* **have** *ainit*: *a*∈*set init* **by** *auto*
note *a*=*snoc*
from *a axy* **obtain** *pre suf* **where** *qs*: *qs* = *pre* @ [*a*] @ *suf*
and *nosuf*: ⋀*e*. *e* ∈ {*x*,*y*} ⟹ *e* ∉ *set suf*
and *pre*: *Lxy pre* {*x*,*y*} = *as*
using *proj_Snoc* **by** *metis*
show *?thesis*
proof (*cases as rule*: *rev_cases*)
case *Nil*
from *pre Nil* **have** *xqs*: *x* ∉ *set pre* **and** *yqs*: *y* ∉ *set pre* **by**(*simp_all add*: *projEmpty*)
from *xqs yqs axy* **have** *a* ∉ *set pre* **by** *blast*
then have *noocc*: *index* (*rev pre*) *a* = *length* (*rev pre*) **by** *simp*
have *x* < *y in fst* (*config*$_p$ (*rTS* []) *init qs*)

= $x < y$ *in fst* (*config*$_p$ (*rTS* []) *init* ((*pre* @ [*a*]) @ *suf*)) **by**(*simp add*: *qs*)
also have ... = $x < y$ *in fst* (*config*$_p$ (*rTS* []) *init* (*pre* @ [*a*]))
apply(*subst config_append*)
apply(*rule staysuntouched*) **using** *assms xqs yqs qs nosuf* **by**(*simp_all*)
also have ... = $x < y$ *in fst* (*config*$_p$ (*rTS* []) *init pre*)
apply(*subst config_append*)
apply(*simp add*: *rTS_def Step_def split_def*)
apply(*simp only*: *TS_step_d_def*)
apply(*simp only*: *sndTSconfig'*[*unfolded rTS_def*])
by(*simp add*: *noocc step_def*)
also have ... = $x < y$ *in init*
apply(*rule staysuntouched'*) **using** *assms xqs yqs qs* **by**(*simp_all*)

also have ... = $x < y$ *in fst* (*config*$_p$ (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}))
unfolding *a Nil* **apply**(*simp add*: *Step_def split_def rTS_def TS_step_d_def step_def*)
apply(*rule Lxy_mono*) **using** *xyininit dinit* **by**(*simp_all*)
finally show *?thesis* **.**
next
case (*snoc bs b*)
note *b*=*this*
with *a* **have** *b*∈*set* (*Lxy qs* {*x*, *y*}) **by** (*simp*)
then have *bxy*: *b*∈{*x*,*y*} **by**(*simp add*: *Lxy_set_filter*)
with *xyininit* **have** *binit*: *b*∈*set init* **by** *auto*
from *b pre* **have** *Lxy pre* {*x*,*y*} = *bs* @ [*b*] **by** *simp*
with *bxy* **obtain** *pre2 suf2* **where** *bs*: *pre* = *pre2* @ [*b*] @ *suf2*
and *nosuf2*: ⋀*e*. *e* ∈ {*x*,*y*} ⟹ *e* ∉ *set suf2*
and *pre2*: *Lxy pre2* {*x*,*y*} = *bs*
using *proj_Snoc* **by** *metis*

from *bs qs* **have** *qs2*: *qs* = *pre2* @ [*b*] @ *suf2* @ [*a*] @ *suf* **by** *simp*

show *?thesis*
proof (*cases a*=*b*)
case *True*
note *ab*=*this*

let *?qs* =(*pre2* @ [*a*] @ *suf2* @ [*a*]) @ *suf*
{
fix *e*
assume *ane*: *a*≠*e*
assume *exy*: *e*∈{*x*,*y*}

have *a* < *e in fst* (*config$_p$* (*rTS* []) *init qs*)
= *a* < *e in fst* (*config$_p$* (*rTS* []) *init ?qs*) **using** *True qs2* **by**(*simp*)
also have ... = *a* < *e in fst* (*config$_p$* (*rTS* []) *init* (*pre2* @ [*a*] @ *suf2* @ [*a*]))
apply(*subst config_append*)
apply(*rule staysuntouched*) **using** *assms qs nosuf* **apply**(*simp_all*)
using *exy xyininit* **apply** *fast*
using *nosuf axy* **apply**(*simp*)
using *nosuf exy* **by** *simp*
also have ...
apply(*simp*)
apply(*rule twotox*[*unfolded s_TS_def*, *simplified*])
using *nosuf2 exy* **apply**(*simp*)
using *assms* **apply**(*simp_all*)
using *axy xyininit* **apply** *fast*
using *exy xyininit* **apply** *fast*
using *nosuf2 axy* **apply**(*simp*)
using *ane* **by** *simp*
finally have *a* < *e in fst* (*config$_p$* (*rTS* []) *init qs*) **by** *simp*
} **note** *full*=*this*

have *set* (*bs* @ [*a*]) ⊆ *set* (*Lxy qs* {*x*, *y*}) **using** *a b* **by** *auto*
also have ... = {*x*,*y*} ∩ *set qs* **by** (*rule Lxy_set_filter*)
also have ... ⊆ {*x*,*y*} **by** *simp*
finally have *bsaxy*: *set* (*bs* @ [*a*]) ⊆ {*x*,*y*} **.**

with *xny* **show** *?thesis*
proof(*cases x*=*a*)
case *True*
have *1*: *a* < *y in fst* (*config$_p$* (*rTS* []) *init qs*)
apply(*rule full*)
using *True xny* **apply** *blast*
by *simp*

have *a* < *y in fst* (*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}))
= *a* < *y in fst* (*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) ((*bs* @ [*a*] @ []) @ [*a*]))
using *a b ab* **by** *simp*
also have ...
apply(*rule projxx*[**where** *bs*=*bs* **and** *f*=[]])
using *True* **apply** *blast*
using *a b True ab xny Lxyinitxy bsaxy* **by**(*simp_all*)

finally show *?thesis* **using** *True 1* **by** *simp*
next
case *False*
with *axy* **have** *ay*: *a=y* **by** *blast*
have *1*: *a < x in fst (config$_p$ (rTS []) init qs)*
apply(*rule full*)
using *False xny* **apply** *blast*
by *simp*
have *a < x in fst (config$_p$ (rTS []) (Lxy init {x, y}) (Lxy qs {x, y}))*
= *a < x in fst (config$_p$ (rTS []) (Lxy init {x, y}) ((bs @ [a] @ []) @ [a]))*
using *a b ab* **by** *simp*
also have . . .
apply(*rule projxx*[**where** *bs=bs* **and** *f=*[]])
using *True axy* **apply** *blast*
using *a b True ab xny Lxyinitxy ay bsaxy* **by**(*simp_all*)
finally have *2*: *a < x in fst (config$_p$ (rTS []) (Lxy init {x, y}) (Lxy qs {x, y}))* **.**

have *x < y in fst (config$_p$ (rTS []) init qs) =*
(¬ *y < x in fst (config$_p$ (rTS []) init qs)*)
apply(*subst not_before_in*)
using *assms* **by**(*simp_all*)
also have . . . = *False* **using** *1 ay* **by** *simp*
also have . . . = (¬ *y < x in fst (config$_p$ (rTS []) (Lxy init {x, y}) (Lxy qs {x, y}))*)
using *2 ay* **by** *simp*
also have . . . = *x < y in fst (config$_p$ (rTS []) (Lxy init {x, y}) (Lxy qs {x, y}))*
apply(*subst not_before_in*)
using *assms* **by**(*simp_all add*: *Lxy_set_filter*)
finally show *?thesis* **.**
qed
next
case *False*
note *ab=this*

show *?thesis*
proof (*cases bs rule*: *rev_cases*)
case *Nil*
with *a b* **have** *Lxy qs {x, y} = [b,a]* **by** *simp*
from *pre2 Nil* **have** *xqs*: *x* ∉ *set pre2* **and** *yqs*: *y* ∉ *set pre2* **by**(*simp_all add*: *projEmpty*)

from *xqs yqs bxy* **have** *b* ∉ *set pre2* **by** *blast*
then have *noocc2*: *index* (*rev pre2*) *b* = *length* (*rev pre2*) **by** *simp*
from *axy nosuf2* **have** *a* ∉ *set suf2* **by** *blast*
with *xqs yqs axy False* **have** *a* ∉ *set* ((*pre2* @ *b* # *suf2*)) **by**(*auto*)
then have *noocc*: *index* (*rev* (*pre2* @ *b* # *suf2*) @ []) *a* = *length* (*rev* (*pre2* @ *b* # *suf2*)) **by** *simp*
have *x* < *y in fst* (*config*$_p$ (*rTS* []) *init qs*)
= *x* < *y in fst* (*config*$_p$ (*rTS* []) *init* ((((*pre2* @ [*b*]) @ *suf2*) @ [*a*]) @ *suf*)) **by**(*simp add*: *qs2*)
also have ... = *x* < *y in fst* (*config*$_p$ (*rTS* []) *init* (((*pre2* @ [*b*]) @ *suf2*) @ [*a*]))
apply(*subst config_append*)
apply(*rule staysuntouched*) **using** *assms xqs yqs qs nosuf* **by**(*simp_all*)
also have ... = *x* < *y in fst* (*config*$_p$ (*rTS* []) *init* ((*pre2* @ [*b*]) @ *suf2*))
apply(*subst config_append*)
apply(*simp add*: *rTS_def Step_def split_def*)
apply(*simp only*: *TS_step_d_def*)
apply(*simp only*: *sndTSconfig'*[*unfolded rTS_def*])
apply(*simp only*: *noocc*) **by** (*simp add*: *step_def*)
also have ... = *x* < *y in fst* (*config*$_p$ (*rTS* []) *init* (*pre2* @ [*b*]))
apply(*subst config_append*)
apply(*rule staysuntouched*) **using** *assms xqs yqs qs2 nosuf2* **by**(*simp_all*)
also have ... = *x* < *y in fst* (*config*$_p$ (*rTS* []) *init* (*pre2*))
apply(*subst config_append*)
apply(*simp add*: *rTS_def Step_def split_def*)
apply(*simp only*: *TS_step_d_def*)
apply(*simp only*: *sndTSconfig'*[*unfolded rTS_def*])
by(*simp add*: *noocc2 step_def*)
also have ... = *x* < *y in init*
apply(*rule staysuntouched'*) **using** *assms xqs yqs qs2* **by**(*simp_all*)

also have ... = *x* < *y in fst* (*config*$_p$ (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}))
unfolding *a b Nil*
using *False*
apply(*simp add*: *Step_def split_def rTS_def TS_step_d_def step_def*)
apply(*rule Lxy_mono*) **using** *xyininit dinit* **by**(*simp_all*)
finally show *?thesis* **.**
next
case (*snoc cs c*)

note *c=this*
with *a b* **have** *c∈set* (*Lxy qs* {*x*, *y*}) **by** (*simp*)
then have *cxy*: *c*∈{*x*,*y*} **by**(*simp add*: *Lxy_set_filter*)
from *c pre2* **have** *Lxy pre2* {*x*,*y*} = *cs* @ [*c*] **by** *simp*
with *cxy* **obtain** *pre3 suf3* **where** *cs*: *pre2* = *pre3* @ [*c*] @ *suf3*
and *nosuf3*: ⋀*e*. *e* ∈ {*x*,*y*} ⟹ *e* ∉ *set suf3*
and *pre3*: *Lxy pre3* {*x*,*y*} = *cs*
using *proj_Snoc* **by** *metis*

let *?qs*= *pre3* @ [*c*] @ *suf3* @ [*b*] @ *suf2* @ [*a*] @ *suf*
from *bs cs qs* **have** *qs2*: *qs* = *?qs* **by** *simp*

show *?thesis*
proof(*cases c*=*a*)
case *True*
note *ca*=*this*

have *a* < *b in fst* (*config*$_p$ (*rTS* []) *init qs*)
= *a* < *b in fst* (*config*$_p$ (*rTS* []) *init* ((*pre3* @ *a* # (*suf3* @ [*b*] @ *suf2*) @ [*a*]) @ *suf*))
using *qs2 True* **by** *simp*
also have ... = *a* < *b in fst* (*config*$_p$ (*rTS* []) *init* (*pre3* @ *a* # (*suf3* @ [*b*] @ *suf2*) @ [*a*]))
apply(*subst config_append*)
apply(*rule staysuntouched*) **using** *assms qs nosuf* **apply**(*simp_all*)
using *bxy xyininit* **apply**(*fast*)
using *nosuf axy bxy* **by**(*simp_all*)
also have ...
apply(*rule twotox*[*unfolded s_TS_def*, *simplified*])
using *nosuf2 nosuf3 bxy* **apply**(*simp*)
using *assms* **apply**(*simp_all*)
using *axy xyininit* **apply**(*fast*)
using *bxy xyininit* **apply**(*fast*)
using *ab nosuf2 nosuf3 axy* **apply**(*simp*)
using *ab* **by** *simp*
finally have *full*: *a* < *b in fst* (*config*$_p$ (*rTS* []) *init qs*) **by** *simp*

have *set* (*cs* @ [*a*] @ [*b*]) ⊆ *set* (*Lxy qs* {*x*, *y*}) **using** *a b c* **by** *auto*
also have ... = {*x*,*y*} ∩ *set qs* **by** (*rule Lxy_set_filter*)
also have ... ⊆ {*x*,*y*} **by** *simp*
finally have *csabxy*: *set* (*cs* @ [*a*] @ [*b*]) ⊆ {*x*,*y*} **.**

with *xny* **show** *?thesis*
proof(*cases x=a*)
case *True*
with *xny ab bxy* **have** *bisy*: *b=y* **by** *blast*
have *1*: *x < y in fst* (*config$_p$* (*rTS* []) *init qs*)
using *full True bisy* **by** *simp*

have *a < y in fst* (*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}))
= *a < y in fst* (*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) ((*cs* @ [*a*] @ [*b*]) @ [*a*]))
using *a b c ca ab* **by** *simp*
also have ...
apply(*rule projxx*)
using *True* **apply** *blast*
using *a b True ab xny Lxyinitxy csabxy* **by**(*simp_all*)
finally show *?thesis* **using** *1 True* **by** *simp*
next
case *False*
with *axy* **have** *ay*: *a=y* **by** *blast*
with *xny ab bxy* **have** *bisx*: *b=x* **by** *blast*
have *1*: *y < x in fst* (*config$_p$* (*rTS* []) *init qs*)
using *full ay bisx* **by** *simp*

have *a < x in fst* (*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}))
= *a < x in fst* (*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) ((*cs* @ [*a*] @ [*b*]) @ [*a*]))
using *a b c ca ab* **by** *simp*
also have ...
apply(*rule projxx*)
using *a b True ab xny Lxyinitxy csabxy False* **by**(*simp_all*)
finally have *2*: *a < x in fst* (*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*})) **.**

have *x < y in fst* (*config$_p$* (*rTS* []) *init qs*) =
(¬ *y < x in fst* (*config$_p$* (*rTS* []) *init qs*))
apply(*subst not_before_in*)
using *assms* **by**(*simp_all*)
also have ... = *False* **using** *1 ay* **by** *simp*
also have ... = (¬ *y < x in fst* (*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*})))
using *2 ay* **by** *simp*

also have ... = $x < y$ *in fst* (*config*$_p$ (*rTS* [])) (*Lxy init* {*x*, *y*})
(*Lxy qs* {*x*, *y*}))
apply(*subst not_before_in*)
using *assms* **by**(*simp_all add*: *Lxy_set_filter*)
finally show *?thesis* **.**
qed
next
case *False*
then have *cb*: *c*=*b* **using** *bxy cxy axy ab* **by** *blast*

let *?cs* = *suf2* @ [*a*] @ *suf*
let *?i* = *index ?cs a*

have *aed*: (∀ *j*<*index* (*suf2* @ *a* # *suf*) *a*. (*suf2* @ *a* # *suf*) ! *j*
≠ *a*)
by (*metis add.right_neutral axy index_Cons index_append*
nosuf2 nth_append nth_mem)

have *?i* < *length ?cs*
⟶ (∀ *j*<*?i*. *?cs* ! *j* ≠ *?cs* ! *?i*) ⟶ *?cs* ! *?i* ≠ *b*
⟶ *?cs* ! *?i* ∉ *set suf3*
⟶ *b* < *?cs* ! *?i in s_TS init* [] *qs* (*length* (*pre3* @ [*b*] @ *suf3*
@ [*b*]) + *?i* + *1*)
apply(*rule casexxy*)
using *cb qs2* **apply**(*simp*)
using *bxy ab nosuf2 nosuf* **apply**(*simp*)
using *bs qs qsininit* **apply**(*simp*)
using *bxy xyininit* **apply**(*blast*)
apply(*fact*)
using *nosuf3 bxy* **apply**(*simp*)
using *cs bs qs qsininit* **by**(*simp_all*)

then have *inner*: *b* < *a in s_TS init* [] *qs* (*length* (*pre3* @ [*b*] @
suf3 @ [*b*]) + *?i* + *1*)
using *ab nosuf3 axy bxy aed*
by(*simp*)
let *?n* = (*length* (*pre3* @ [*b*] @ *suf3* @ [*b*]) + *?i* + *1*)
let *?inner*=(*config*$_p$ (*rTS* []) *init* (*take* (*length* (*pre3* @ [*b*] @ *suf3*
@ [*b*]) + *?i* + *1*) *?qs*))

have *b* < *a in fst* (*config*$_p$ (*rTS* []) *init qs*)
= *b* < *a in fst* (*config*$_p$ (*rTS* []) *init* (*take ?n ?qs* @ *drop ?n ?qs*))
using *qs2* **by** *simp*

also have ... = *b* < *a in fst* (*config'* (*rTS* []) *?inner suf*)
apply(*simp only*: *config_append drop_append*)
using *nosuf2 axy* **by**(*simp add*: *index_append config_append*)
also have ... = *b* < *a in fst ?inner*
apply(*rule staysuntouched*) **using** *assms bxy xyininit qs nosuf*
apply(*simp_all*)
using *bxy xyininit* **apply**(*blast*)
using *axy xyininit* **by** (*blast*)
also have ... = *True* **using** *inner* **by**(*simp add*: *s_TS_def qs2*)
finally have *full*: *b* < *a in fst* (*config$_p$* (*rTS* []) *init qs*) **by** *simp*

have *set* (*cs* @ [*b*] @ []) ⊆ *set* (*Lxy qs* {*x*, *y*}) **using** *a b c* **by**
auto
also have ... = {*x*,*y*} ∩ *set qs* **by** (*rule Lxy_set_filter*)
also have ... ⊆ {*x*,*y*} **by** *simp*
finally have *csbxy*: *set* (*cs* @ [*b*] @ []) ⊆ {*x*,*y*} **.**

have *set* (*Lxy init* {*x*, *y*}) = {*x*,*y*} ∩ *set init*
by(*rule Lxy_set_filter*)
also have ... = {*x*,*y*} **using** *xyininit* **by** *fast*
also have ... = {*b*,*a*} **using** *axy bxy ab* **by** *fast*
finally have *r*: *set* (*Lxy init* {*x*, *y*}) = {*b*, *a*} **.**

let *?confbef*=(*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) ((*cs* @ [*b*] @ [])
@ [*b*]))
have *f1*: *b* < *a in fst ?confbef*
apply(*rule projxx*)
using *bxy ab axy a b c csbxy Lxyinitxy* **by**(*simp_all*)
have *1*: *fst ?confbef* = [*b*,*a*]
apply(*rule oneposs*)
using *ab axy bxy xyininit Lxy_distinct*[*OF dinit*] *f1 r*
by(*simp_all*)
have *2*: *snd* (*Partial_Cost_Model.config'*
(λ*s*. [], *TS_step_d*)
(*Lxy init* {*x*, *y*}, [])
(*cs* @ [*b*, *b*])) = [*b*,*b*]@(*rev cs*)
using *sndTSdet*[*of length* (*cs* @ [*b*, *b*]) (*cs* @ [*b*, *b*]), *unfolded*
rTS_def] **by**(*simp*)
have *b* < *a in fst* (*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*,
y}))
= *b* < *a in fst* (*config$_p$* (*rTS* []) (*Lxy init* {*x*, *y*}) (((*cs* @ [*b*] @
[]) @ [*b*])@[*a*]))
using *a b c cb* **by**(*simp*)
also have ...

apply(*subst config_append*)
using *1 2 ab* **apply**(*simp add*: *step_def Step_def split_def rTS_def TS_step_d_def*)
by(*simp add*: *before_in_def*)
finally have *projected*: $b < a$ *in fst* (*config*$_p$ (*rTS* [])) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*})) **.**

have *1*: {*x*,*y*} = {*a*,*b*} **using** *ab axy bxy* **by** *fast*
with *xny* **show** *?thesis*
proof(*cases x*=*a*)
case *True*
with *1 xny* **have** *y*: *y*=*b* **by** *fast*
have $a < b$ *in fst* (*config*$_p$ (*rTS* [])) *init qs*) =
(¬ $b < a$ *in fst* (*config*$_p$ (*rTS* [])) *init qs*))
apply(*subst not_before_in*)
using *binit ainit ab* **by**(*simp_all*)
also have ... = *False* **using** *full* **by** *simp*
also have ... = (¬ $b < a$ *in fst* (*config*$_p$ (*rTS* [])) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*})))
using *projected* **by** *simp*
also have ... = $a < b$ *in fst* (*config*$_p$ (*rTS* [])) (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}))
apply(*subst not_before_in*)
using *binit ainit ab axy bxy* **by**(*simp_all add*: *Lxy_set_filter*)
finally show *?thesis* **using** *True y* **by** *simp*
next
case *False*
with *1 xny* **have** *y*: *y*=*a x*=*b* **by** *fast*+
with *full projected* **show** *?thesis* **by** *fast*
qed
qed
qed
qed
qed
qed

show *?thesis* **unfolding** *Pbefore_in_def*
apply(*subst config_embed*)
apply(*subst config_embed*)
apply(*simp*) **by** (*rule A*)

qed

theorem *TS_pairwise*: *pairwise* (*embed* (*rTS* []))
apply(*rule pairwise_property_lemma*)
apply(*rule TS_pairwise′*) **by** (*simp_all add*: *rTS_def TS_step_d_def*)

15.6 TS is 2-compet

lemma *TS_compet′*: *pairwise* (*embed* (*rTS* [])) $\Longrightarrow$
$\forall$ *s0*$\in${*init*::(*nat list*). *distinct init* $\wedge$ *init*$\neq$[]}. $\exists$ *b*$\geq$*0*. $\forall$ *qs*$\in${*x*. *set x* $\subseteq$ *set s0*}. T_p*_on_rand* (*embed* (*rTS* [])) *s0 qs* $\leq$ (*2*::*real*) $*$ T_p*_opt s0 qs* + *b*
unfolding *rTS_def*
proof (*rule factoringlemma_withconstant*, *goal_cases*)
case *5*
show *?case*
proof (*safe*, *goal_cases*)
case (*1 init*)
note *out*=*this*
show *?case*
apply(*rule exI*[**where** *x*=*2*])
apply(*simp*)
proof (*safe*, *goal_cases*)
case (*1 qs a b*)
then have *a*: *a*$\neq$*b* **by** *simp*
have *twist*: {*a*,*b*}={*b*, *a*} **by** *auto*
have *b1*: *set* (*Lxy qs* {*a*, *b*}) $\subseteq$ {*a*, *b*} **unfolding** *Lxy_def* **by** *auto*
with *this*[*unfolded twist*] **have** *b2*: *set* (*Lxy qs* {*b*, *a*}) $\subseteq$ {*b*, *a*} **by**(*auto*)

have *set* (*Lxy init* {*a*, *b*}) = {*a*,*b*} $\cap$ (*set init*) **apply**(*induct init*)
unfolding *Lxy_def* **by**(*auto*)
with *1* **have** *A*: *set* (*Lxy init* {*a*, *b*}) = {*a*,*b*} **by** *auto*
have *finite* {*a*,*b*} **by** *auto*
from *out* **have** *B*: *distinct* (*Lxy init* {*a*, *b*}) **unfolding** *Lxy_def* **by** *auto*
have *C*: *length* (*Lxy init* {*a*, *b*}) = *2*
using *distinct_card*[*OF B*, *unfolded A*] **using** *a* **by** *auto*

have {*xs*. *set xs* = {*a*,*b*} $\wedge$ *distinct xs* $\wedge$ *length xs* =(*2*::*nat*)}
= { [*a*,*b*], [*b*,*a*] }
apply(*auto simp*: *a a*[*symmetric*])
proof (*goal_cases*)

case (*1 xs*)
from *1*(*4*) **obtain** *x xs′* **where** *r*:*xs*=*x*#*xs′* **by** (*metis Suc_length_conv add_2_eq_Suc′ append_Nil length_append*)
with *1*(*4*) **have** *length xs′* = *1* **by** *auto*
then obtain *y* **where** *s*: [*y*] = *xs′* **by** (*metis One_nat_def length_0_conv length_Suc_conv*)
from *r s* **have** *t*: [*x*,*y*] = *xs* **by** *auto*
moreover from *t 1*(*1*) **have** *x*=*b* **using** *doubleton_eq_iff 1*(*2*) **by** *fastforce*
moreover from *t 1*(*1*) **have** *y*=*a* **using** *doubleton_eq_iff 1*(*2*) **by** *fastforce*
ultimately show *?case* **by** *auto*
qed

with *A B C* **have** *pos*: (*Lxy init* {*a*, *b*}) = [*a*,*b*]
∨ (*Lxy init* {*a*, *b*}) = [*b*,*a*] **by** *auto*

{ fix *a*::*nat*
fix *b*::*nat*
fix *qs*
assume *as*: *a* ≠ *b set qs* ⊆ {*a*, *b*}
have *T_on_rand′* (*embed* (*rTS* [])) (*fst* (*embed* (*rTS* [])) [*a*,*b*] ≫= (λ*is*. *return_pmf* ([*a*,*b*], *is*))) *qs*
= T_p*_on* (*rTS* []) [*a*, *b*] *qs* **by** (*rule T_on_embed*[*symmetric*])
also from *as* **have** ... ≤ *2* ∗ T_p*_opt* [*a*, *b*] *qs* + *2* **using** *TS_OPT2′* **by** *fastforce*
finally have *T_on_rand′* (*embed* (*rTS* [])) (*fst* (*embed* (*rTS* [])) [*a*,*b*] ≫= (λ*is*. *return_pmf* ([*a*,*b*], *is*))) *qs*
≤ *2* ∗ T_p*_opt* [*a*, *b*] *qs* + *2* **.**
} **note** *ye*=*this*

show *?case*
apply(*cases* (*Lxy init* {*a*, *b*}) = [*a*,*b*])
using *ye*[*OF a b1*, *unfolded rTS_def*] **apply**(*simp*)
using *pos ye*[*OF a*[*symmetric*] *b2*, *unfolded rTS_def*] **by**(*simp add*: *twist*)
qed
qed
next
case *6*
show *?case* **unfolding** *TS_step_d_def* **by** (*simp add*: *split_def TS_step_d_def*)
next
case (*7 init qs x*)
then show *?case*

apply(*induct x*)
by (*simp_all add*: *rTS_def split_def take_Suc_conv_app_nth config′_rand_snoc*)
next
case *4* **then show** *?case* **by** *simp*
qed (*simp_all*)

lemma *TS_compet*: *compet_rand* (*embed* (*rTS* [])) *2* {*init. distinct init* ∧ *init* ≠ []}
unfolding *compet_rand_def static_def*
using *TS_compet′*[*OF TS_pairwise*] **by** *simp*

end

16 BIT is pairwise

theory *BIT_pairwise*
imports *List_Factoring BIT*
begin

lemma *L_nths*: *S* ⊆ {..<*length init*}
⟹ *map_pmf* (λ*l. nths l S*) (*Prob_Theory.bv* (*length init*))
= (*Prob_Theory.bv* (*length* (*nths init S*)))
proof(*induct init arbitrary*: *S*)
case (*Cons a as*)
then have *passt*: {*j. Suc j* ∈ *S*} ⊆ {..<*length as*} **by** *auto*

have *map_pmf* (λ*l. nths l S*) (*Prob_Theory.bv* (*length* (*a # as*))) =
Prob_Theory.bv (*length as*) ≫=
(λ*x. bernoulli_pmf* (*1* / *2*) ≫=
(λ*xa. return_pmf*
((*if 0* ∈ *S then* [*xa*] *else* []) @ *nths x* {*j. Suc j* ∈ *S*})))
by(*simp add*: *map_pmf_def bind_return_pmf bind_assoc_pmf nths_Cons*)

also have ... = (*bernoulli_pmf* (*1* / *2*)) ≫=
(λ*xa.* (*Prob_Theory.bv* (*length as*) ≫=
(λ*x. return_pmf* ((*if 0* ∈ *S then* [*xa*] *else* []) @ *nths x* {*j. Suc j* ∈ *S*}))))
by(*rule bind_commute_pmf*)
also have ... = (*bernoulli_pmf* (*1* / *2*)) ≫=
(λ*xa.* (*map_pmf* (λ*x.* (*nths x* {*j. Suc j* ∈ *S*})) (*Prob_Theory.bv* (*length as*)))
≫= (λ*xs. return_pmf* ((*if 0* ∈ *S then* [*xa*] *else* []) @ *xs*)))

```
    by(simp add: bind_return_pmf bind_assoc_pmf map_pmf_def)
  also have ... = (bernoulli_pmf (1 / 2)) ≫=
        (λxa. Prob_Theory.bv (length (nths as {j. Suc j ∈ S}))
           ≫= (λxs. return_pmf ((if 0 ∈ S then [xa] else []) @ xs)))
      using Cons(1)[OF passt] by auto
  also have ... = Prob_Theory.bv (length (nths (a # as) S))
    apply(auto simp add: nths_Cons bind_return_pmf')
    by(rule bind_commute_pmf)
  finally show ?case .
qed (simp)

lemma L_nths_Lxy:
 assumes x∈set init y∈set init x≠y distinct init
 shows map_pmf (λl. nths l {index init x,index init y}) (Prob_Theory.bv
(length init))
     = (Prob_Theory.bv (length (Lxy init {x,y})))
proof −
 from assms(4) have setinit: (index init) ‘ set init = {0..<length init}
 proof(induct init)
   case (Cons a as)
   with Cons have iH: index as ‘ set as = {0..<length as} by auto
   from Cons have 1:(set as ∩ {x. (a ≠ x)}) = set as by fastforce
   have 2: (λa. Suc (index as a)) ‘ set as =
         (λa. Suc a) ‘ ((index as) ‘ set as ) by auto
   show ?case
   apply(simp add: 1 2 iH) by auto
 qed simp

 have xy_le: index init x<length init index init y<length init using assms
by auto
 have map_pmf (λl. nths l {index init x,index init y}) (Prob_Theory.bv
(length init))
     = (Prob_Theory.bv (length (nths init {index init x,index init y})))
      apply(rule L_nths)
      using assms(1,2) by auto
 moreover have length (Lxy init {x,y}) = length (nths init {index init
x,index init y})
 proof −
   have set (Lxy init {x,y}) = {x,y}
     using assms(1,2) by(simp add: Lxy_set_filter)
   moreover have card {x,y} = 2 using assms(3) by auto
   moreover have distinct (Lxy init {x,y}) using assms(4) by(simp add:
Lxy_distinct)
    ultimately have 1: length (Lxy init {x,y}) = 2 by(simp add: dis-
```

tinct_card[*symmetric*])
have *set* (*nths init* {*index init x*,*index init y*}) = {(*init* ! *i*) | *i*. *i* < *length init* ∧ *i* ∈ {*index init x*,*index init y*}}
using *assms*(*1*,*2*) **by**(*simp add*: *set_nths*)
moreover have *card* {(*init* ! *i*) | *i*. *i* < *length init* ∧ *i* ∈ {*index init x*,*index init y*}} = *2*
proof −
have *1*: {(*init* ! *i*) | *i*. *i* < *length init* ∧ *i* ∈ {*index init x*,*index init y*}} = {*init* ! *index init x*, *init* ! *index init y*} **using** *xy_le* **by** *blast*
also have ... = {*x*,*y*} **using** *nth_index assms*(*1*,*2*) **by** *auto*
finally show *?thesis* **using** *assms*(*3*) **by** *auto*
qed
moreover have *distinct* (*nths init* {*index init x*,*index init y*}) **using** *assms*(*4*) **by**(*simp*)
ultimately have *2*: *length* (*nths init* {*index init x*,*index init y*}) = *2* **by**(*simp add*: *distinct_card*[*symmetric*])
show *?thesis* **using** *1 2* **by** *simp*
qed
ultimately show *?thesis* **by** *simp*
qed

lemma *nths_map*: *map f* (*nths xs S*) = *nths* (*map f xs*) *S*
apply(*induct xs arbitrary*: *S*) **by**(*simp_all add*: *nths_Cons*)

lemma *nths_empty*: (∀ *i*∈*S*. *i*≥*length xs*) ⟹ *nths xs S* = []
proof −
assume (∀ *i*∈*S*. *i*≥*length xs*)
then have *set* (*nths xs S*) = {} **apply**(*simp add*: *set_nths*) **by** *force*
then show *nths xs S* = [] **by** *simp*
qed

lemma *nths_project'*: *i* < *length xs* ⟹ *j* < *length xs* ⟹ *i*<*j*
⟹ *nths xs* {*i*,*j*} = [*xs*!*i*, *xs*!*j*]
proof −
assume *il*: *i* < *length xs* **and** *jl*: *j* < *length xs* **and** *ij*: *i*<*j*

from *il* **obtain** *a as* **where** *dec1*: *a* @ [*xs*!*i*] @ *as* = *xs*
and *a* = *take i xs as*=*drop* (*Suc i*) *xs*
and *length_a*: *length a* = *i* **and** *length_as*: *length as* = *length xs* − *i* − *1* **using** *id_take_nth_drop* **by** *fastforce*
have *j*≥*length* (*a* @ [*xs*!*i*]) **using** *length_a ij* **by** *auto*
then have ((*a* @ [*xs*!*i*]) @ *as*) ! *j* = *as* ! (*j*−*length* (*a* @ [*xs* ! *i*])) **using** *nth_append*[**where** *xs*=*a* @ [*xs*!*i*] **and** *ys*=*as*]

```
    by(simp)
  then have xsj: xs ! j = as ! (j−i−1) using dec1 length_a by auto
  have las: (j−i−1) < length as using length_as jl ij by simp
  obtain b c where dec2: b @ [xs!j] @ c = as
          and b = take (j−i−1) as c=drop (Suc (j−i−1)) as
            and length_b: length b = j−i−1 using id_take_nth_drop[OF
las] xsj by force
  have xs_dec: a @ [xs!i] @ b @ [xs!j] @ c = xs using dec1 dec2 by auto

  have s2: {k. (k + i ∈ {i, j})} = {0,j−i}  using ij by force
  have s3: {k. (k  + length [xs ! i] ∈ {0, j−i})} = {j−i−1}  using ij by
force
  have s4: {k. (k  + length b ∈ {j−i−1})} = {0}  using length_b by force
  have s5: {k. (k  + length [xs!j] ∈ {0})} = {} by force
  have l1: nths a {i,j} = []
    apply(rule nths_empty) using length_a ij by fastforce
  have l2: nths b {j − Suc i} = []
    apply(rule nths_empty) using length_b ij by fastforce
  have nths ( a @ [xs!i] @ b @ [xs!j] @ c) {i,j} = [xs!i, xs!j]
      apply(simp only: nths_append length_a s2 s3 s4 s5)
      by(simp add: l1 l2)
  then show nths xs {i,j} = [xs!i, xs!j] unfolding xs_dec .
qed

lemma nths_project:
  assumes  i < length xs j < length xs i<j
   shows nths xs {i,j} ! 0 = xs ! i ∧ nths xs {i,j} ! 1 = xs ! j
proof −
  from assms have nths xs {i,j} = [xs!i, xs!j] by(rule nths_project')
  then show ?thesis by simp
qed

lemma BIT_pairwise':
  assumes set qs ⊆ set init
    (x,y)∈ {(x,y). x ∈ set init ∧ y∈set init ∧ x≠y}
   and  xny:x ≠ y and dinit: distinct init
  shows Pbefore_in x y BIT qs init = Pbefore_in x y BIT (Lxy qs {x,y})
(Lxy init {x,y})
proof −
  from assms have xyininit: {x, y} ⊆ set init
        and qsininit: set qs ⊆ set init by auto

  have xyininit': {y,x} ⊆ set init using xyininit by auto
```

have *a*: *x* ∈ *set init* *y*∈*set init* **using** *assms* **by** *auto*

{ fix *n*
have *strong*: *set qs* ⊆ *set init* ⟹
map_pmf (λ(*l*,(*w*,*i*)). (*Lxy l* {*x*,*y*},(*nths w* {*index init x*,*index init y*},*Lxy init* {*x*,*y*}))) (*config_rand BIT init qs*) =
config_rand BIT (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}) (**is** *?inv* ⟹ *?L qs* = *?R qs*)
proof (*induct qs rule*: *rev_induct*)
case *Nil*

have *map_pmf* (λ(*l*,(*w*,*i*)). (*Lxy l* {*x*,*y*},(*nths w* {*index init x*,*index init y*},*Lxy init* {*x*,*y*}))) (*config_rand BIT init* [])
= *map_pmf* (λ*w*. (*Lxy init* {*x*,*y*}, (*w*, *Lxy init* {*x*,*y*}))) (*map_pmf* (λ*l*. *nths l* {*index init x*,*index init y*}) (*Prob_Theory.bv* (*length init*)))
by(*simp add*: *bind_return_pmf map_pmf_def bind_assoc_pmf split_def BIT_init_def*)
also have ... = *map_pmf* (λ*w*. (*Lxy init* {*x*,*y*}, (*w*, *Lxy init* {*x*,*y*}))) (*Prob_Theory.bv* (*length* (*Lxy init* {*x*, *y*})))
using *L_nths_Lxy*[*OF a xny dinit*] **by** *simp*
also have ... = *config_rand BIT* (*Lxy init* {*x*, *y*}) (*Lxy* [] {*x*, *y*})
by(*simp add*: *BIT_init_def bind_return_pmf bind_assoc_pmf map_pmf_def*)
finally show *?case* **.**
next
case (*snoc q qs*)
then have *qininit*: *q* ∈ *set init*
and *qsininit*: *set qs* ⊆ *set init* **using** *qsininit* **by** *auto*

from *snoc*(*1*)[*OF qsininit*] **have** *iH*: *?L qs* = *?R qs* **by** (*simp add*: *split_def*)

show *?case*
proof (*cases q* ∈ {*x*,*y*})
case *True*
note *whatisq*=*this*

have *?L* (*qs*@[*q*]) =
map_pmf (λ(*l*,(*w*,*i*)). (*Lxy l* {*x*,*y*},(*nths w* {*index init x*,*index init y*},*Lxy init* {*x*,*y*}))) (*config_rand BIT init qs* ⤜
(λ*s*. *BIT_step s q* ⤜ (λ(*a*, *nis*). *return_pmf* (*step* (*fst s*) *q a*, *nis*))))
by(*simp add*: *split_def config'_rand_snoc*)
also have ... =

map_pmf (λ(*l*,(*w*,*i*)). (*Lxy l* {*x*,*y*}, (*nths w* {*index init x*,*index init y*},*Lxy init* {*x*,*y*}))) (*config_rand BIT init qs*) ≫=
(λ*s*.
BIT_step s q ≫=
(λ(*a*, *nis*). *return_pmf* (*step* (*fst s*) *q a*, *nis*)))
apply(*simp add*: *map_pmf_def split_def bind_return_pmf bind_assoc_pmf*)
apply(*simp add*: *BIT_step_def bind_return_pmf*)
proof (*rule bind_pmf_cong*, *goal_cases*)
case (*2 z*)
let *?s* = *fst z*
let *?b* = *fst* (*snd z*)

from *2* **have** *z*: *set* (*?s*) = *set init* **using** *config_rand_set*[*of BIT*, *simplified*] **by** *metis*
with *True* **have** *qLxy*: *q* ∈ *set* (*Lxy* (*?s*) {*x*, *y*}) **using** *xyininit* **by** (*simp add*: *Lxy_set_filter*)
from *2* **have** *dz*: *distinct* (*?s*) **using** *dinit config_rand_distinct*[*of BIT*, *simplified*] **by** *metis*
then have *dLxy*: *distinct* (*Lxy* (*?s*) {*x*, *y*}) **using** *Lxy_distinct* **by** *auto*

from *2* **have** [*simp*]: *snd* (*snd z*) = *init* **using** *config_n_init3*[*simplified*] **by** *metis*

from *2* **have** [*simp*]: *length* (*fst* (*snd z*)) = *length init* **using** *config_n_fst_init_length2*[*simplified*] **by** *metis*

have *indexinbounds*: *index init x* < *length init index init y* < *length init* **using** *a* **by** *auto*
from *a xny* **have** *indnot*: *index init x* ≠ *index init y* **by** *auto*

have *f1*: *index init x* < *length* (*fst* (*snd z*)) **using** *xyininit* **by** *auto*
have *f2*: *index init y* < *length* (*fst* (*snd z*)) **using** *xyininit* **by** *auto*
have *3*: *index init x* ≠ *index init y* **using** *xny xyininit* **by** *auto*

from *dinit* **have** *dfil*: *distinct* (*Lxy init* {*x*,*y*}) **by**(*rule Lxy_distinct*)
have *Lxy_set*: *set* (*Lxy init* {*x*, *y*}) = {*x*,*y*} **apply**(*simp add*: *Lxy_set_filter*) **using** *xyininit* **by** *fast*
then have *xLxy*: *x*∈*set* (*Lxy init* {*x*, *y*}) **by** *auto*
have *Lxy_length*: *length* (*Lxy init* {*x*, *y*}) = *2* **using** *dfil Lxy_set*

xny distinct_card **by** *fastforce*
have *31*: *index* (*Lxy init* {*x*, *y*}) *x* < *2*
and *32*: *index* (*Lxy init* {*x*, *y*}) *y* < *2* **using** *Lxy_set xyininit Lxy_length* **by** *auto*
have *33*: *index* (*Lxy init* {*x*, *y*}) *x* ≠ *index* (*Lxy init* {*x*,*y*}) *y*
using *xny xLxy* **by** *auto*

have *a1*: *nths* (*flip* (*index init* (*q*)) (*fst* (*snd z*))) {*index init x*,*index init y*}
= *flip* (*index* (*Lxy init* {*x*,*y*}) (*q*)) (*nths* (*fst* (*snd z*)) {*index init x*,*index init y*}) (**is** *?A*=*?B*)
proof (*simp only*: *list_eq_iff_nth_eq*, *goal_cases*)
case *1*

{**assume** *ass*: *index init x* < *index init y*
then have *index* (*Lxy init* {*x*,*y*}) *x* < *index* (*Lxy init* {*x*,*y*}) *y*
using *Lxy_mono*[*OF xyininit dinit*] *before_in_def a*(*2*) **by** *force*
with *31 32* **have** *ix*: *index* (*Lxy init* {*x*,*y*}) *x* = *0*
and *iy*: *index* (*Lxy init* {*x*,*y*}) *y* = *1* **by** *auto*

have *g1*: (*nths* (*fst* (*snd z*)) {*index init x*,*index init y*})
= [(*fst* (*snd z*)) ! *index init x*, (*fst* (*snd z*)) ! *index init y*]
apply(*rule nths_project'*)
using *xyininit* **apply**(*simp*)
using *xyininit* **apply**(*simp*)
by *fact*

have *nths* (*flip* (*index init* (*q*)) (*fst* (*snd z*))) {*index init x*,*index init y*}
= [*flip* (*index init* (*q*)) (*fst* (*snd z*))!*index init x*,
flip (*index init* (*q*)) (*fst* (*snd z*))!*index init y*]
apply(*rule nths_project'*)
using *xyininit* **apply**(*simp*)
using *xyininit* **apply**(*simp*)
by *fact*
also have ... = *flip* (*index* (*Lxy init* {*x*,*y*}) (*q*)) [(*fst* (*snd z*)) ! *index init x*, (*fst* (*snd z*)) ! *index init y*]
apply(*cases q*=*x*)
apply(*simp add*: *ix*) **using** *flip_other*[*OF f2 f1 3*] *flip_itself*[*OF f1*] **apply**(*simp*)
using *whatisq* **apply**(*simp add*: *iy*) **using** *flip_other*[*OF f1 f2*

3[symmetric]] flip_itself[OF f2] **by**(simp)
finally have nths (flip (index init (q)) (fst (snd z))) {index init x,index init y}
= flip (index (Lxy init {x,y}) (q)) (nths (fst (snd z)) {index init x,index init y})
by(simp add: g1)

}**note** cas1=this
have man: {x,y} = {y,x} **by** auto
{**assume** ~ index init x < index init y
then have ass: index init y < index init x **using** 3 **by** auto
then have index (Lxy init {x,y}) y < index (Lxy init {x,y}) x
using Lxy_mono[OF xyininit' dinit] xyininit a(1) man **by**(simp add: before_in_def)
with 31 32 **have** ix: index (Lxy init {x,y}) x = 1
and iy: index (Lxy init {x,y}) y = 0 **by** auto

have g1: (nths (fst (snd z)) {index init y,index init x})
= [(fst (snd z)) ! index init y, (fst (snd z)) ! index init x]
apply(rule nths_project')
using xyininit **apply**(simp)
using xyininit **apply**(simp)
by fact

have man2: {index init x,index init y} = {index init y,index init x} **by** auto
have nths (flip (index init (q)) (fst (snd z))) {index init y,index init x}
= [flip (index init (q)) (fst (snd z))!index init y,
flip (index init (q)) (fst (snd z))!index init x]
apply(rule nths_project')
using xyininit **apply**(simp)
using xyininit **apply**(simp)
by fact
also have ... = flip (index (Lxy init {x,y}) (q)) [(fst (snd z)) ! index init y, (fst (snd z)) ! index init x]
apply(cases q=x)
apply(simp add: ix) **using** flip_other[OF f2 f1 3] flip_itself[OF f1] **apply**(simp)
using whatisq **apply**(simp add: iy) **using** flip_other[OF f1 f2 3[symmetric]] flip_itself[OF f2] **by**(simp)
finally have nths (flip (index init (q)) (fst (snd z))) {index init y,index init x}

= *flip* (*index* (*Lxy init* {*x*,*y*}) (*q*)) (*nths* (*fst* (*snd z*)) {*index init y*,*index init x*})
by(*simp add*: *g1*)
then have *nths* (*flip* (*index init* (*q*)) (*fst* (*snd z*))) {*index init x*,*index init y*}
= *flip* (*index* (*Lxy init* {*x*,*y*}) (*q*)) (*nths* (*fst* (*snd z*)) {*index init x*,*index init y*})
using *man2* **by** *auto*
} **note** *cas2*=*this*

from *cas1 cas2 3* **show** *?case* **by** *metis*
qed

have *a*: *nths* (*fst* (*snd z*)) {*index init x*, *index init y*} ! (*index* (*Lxy init* {*x*,*y*}) (*q*))
= *fst* (*snd z*) ! (*index init* (*q*))
proof −
from *31 32 33***have** *ca*: (*index* (*Lxy init* {*x*,*y*}) *x* = *0* ∧ *index* (*Lxy init* {*x*,*y*}) *y* = *1*)
∨ (*index* (*Lxy init* {*x*,*y*}) *x* = *1* ∧ *index* (*Lxy init* {*x*,*y*}) *y* = *0*) **by** *force*
show *?thesis*
proof (*cases index* (*Lxy init* {*x*,*y*}) *x* = *0*)
case *True*

from *True ca* **have** *y1*: *index* (*Lxy init* {*x*,*y*}) *y* = *1* **by** *auto*
with *True* **have** *index* (*Lxy init* {*x*,*y*}) *x* < *index* (*Lxy init* {*x*,*y*}) *y* **by** *auto*
then have *xy*: *index init x* < *index init y* **using** *dinit dfil Lxy_mono*
using *32 before_in_def Lxy_length xyininit* **by** *fastforce*

have *4*: {*index init y*, *index init x*} = {*index init x*, *index init y*} **by** *auto*

have *nths* (*fst* (*snd z*)) {*index init x*, *index init y*} ! *index* (*Lxy init* {*x*,*y*}) *x* = (*fst* (*snd z*)) ! *index init x*
nths (*fst* (*snd z*)) {*index init x*, *index init y*} ! *index* (*Lxy init* {*x*,*y*}) *y* = (*fst* (*snd z*)) ! *index init y*
unfolding *True y1*
by (*simp_all only*: *nths_project*[*OF f1 f2 xy*])
with *whatisq* **show** *?thesis* **by** *auto*
next

case *False*
with *ca* **have** *x1*: *index* (*Lxy init* {*x*,*y*}) *x* = *1* **by** *auto*
from *dinit* **have** *dfil*: *distinct* (*Lxy init* {*x*,*y*}) **by**(*rule Lxy_distinct*)

from *x1 ca* **have** *y1*: *index* (*Lxy init* {*x*,*y*}) *y* = *0* **by** *auto*
with *x1* **have** *index* (*Lxy init* {*x*,*y*}) *y* < *index* (*Lxy init* {*x*,*y*}) *x* **by** *auto*
then have *xy*: *index init y* < *index init x* **using** *dinit dfil Lxy_mono*
using *32 before_in_def Lxy_length xyininit* **by** (*metis a*(*2*) *indnot linorder_neqE_nat not_less0 y1*)

have *4*: {*index init y*, *index init x*} = {*index init x*, *index init y*} **by** *auto*

have *nths* (*?b*) {*index init x*, *index init y*} ! *index* (*Lxy init* {*x*,*y*}) *x* = (*?b*) ! *index init x*
nths (*?b*) {*index init x*, *index init y*} ! *index* (*Lxy init* {*x*,*y*}) *y* = (*?b*) ! *index init y*
unfolding *x1 y1*
using *4 nths_project*[*OF f2 f1 xy*]
by *simp_all*
with *whatisq* **show** *?thesis* **by** *auto*
qed
qed

have *b*: *Lxy* (*mtf2* (*length ?s*) (*q*) *?s*) {*x*, *y*}
= *mtf2* (*length* (*Lxy ?s* {*x*, *y*})) (*q*) (*Lxy ?s* {*x*, *y*}) (**is** *?A* = *?B*)
proof −
have *sA*: *set ?A* = {*x*,*y*} **using** *z xyininit* **by**(*simp add*: *Lxy_set_filter*)
then have *xlxymA*: *x* ∈ *set ?A*
and *ylxymA*: *y* ∈ *set ?A* **by** *auto*
have *dA*: *distinct ?A* **apply**(*rule Lxy_distinct*) **by**(*simp add*: *dz*)
have *lA*: *length ?A* = *2* **using** *xny sA dA distinct_card* **by** *fastforce*
from *lA ylxymA* **have** *yindA*: *index ?A y* < *2* **by** *auto*
from *lA xlxymA* **have** *xindA*: *index ?A x* < *2* **by** *auto*
have *geA*: {*x*,*y*} ⊆ *set* (*mtf2* (*length ?s*) (*q*) *?s*) **using** *xyininit*

z **by** *auto*
have *geA′*: {*y*,*x*} ⊆ *set* (*mtf2* (*length ?s*) (*q*) (*?s*)) **using** *xyininit z* **by** *auto*
have *man*: {*y*,*x*} = {*x*,*y*} **by** *auto*

have *sB*: *set ?B* = {*x*,*y*} **using** *z xyininit* **by**(*simp add*: *Lxy_set_filter*)
then have *xlxymB*: *x* ∈ *set ?B*
and *ylxymB*: *y* ∈ *set ?B* **by** *auto*
have *dB*: *distinct ?B* **apply**(*simp*) **apply**(*rule Lxy_distinct*) **by**(*simp add*: *dz*)
have *lB*: *length ?B* = *2* **using** *xny sB dB distinct_card* **by** *fastforce*
from *lB ylxymB* **have** *yindB*: *index ?B y* < *2* **by** *auto*
from *lB xlxymB* **have** *xindB*: *index ?B x* < *2* **by** *auto*

show *?thesis*
proof (*cases q* = *x*)
case *True*
then have *xby*: *x* < *y in* (*mtf2* (*length* (*?s*)) (*q*) (*?s*))
apply(*simp*)
apply(*rule mtf2_moves_to_front′′*[*simplified*])
using *z xyininit xny* **by**(*simp_all add*: *dz*)
then have *x* < *y in ?A* **using** *Lxy_mono*[*OF geA*] *dz* **by**(*auto*)
then have *index ?A x* < *index ?A y* **unfolding** *before_in_def* **by** *auto*
then have *in1*: *index ?A x* = *0*
and *in2*: *index ?A y* = *1* **using** *yindA* **by** *auto*
have *?A* = [*?A*!*0*,*?A*!*1*]
apply(*simp only*: *list_eq_iff_nth_eq*)
apply(*auto simp*: *lA*) **apply**(*case_tac i*) **by**(*auto*)
also have ... = [*?A*!*index ?A x*, *?A*!*index ?A y*] **by**(*simp only*: *in1 in2*)
also have ... = [*x*,*y*] **using** *xlxymA ylxymA* **by** *simp*
finally have *end1*: *?A* = [*x*,*y*] **.**

have *x* < *y in ?B*
using *True* **apply**(*simp*)
apply(*rule mtf2_moves_to_front′′*[*simplified*])
using *z xyininit xny* **by**(*simp_all add*: *Lxy_distinct dz Lxy_set_filter*)
then have *index ?B x* < *index ?B y*
unfolding *before_in_def* **by** *auto*

```
        then have in1: index ?B x = 0
              and in2: index ?B y = 1
                using yindB by auto

        have ?B = [?B!0, ?B!1]
              apply(simp only: list_eq_iff_nth_eq)
                apply(simp only: lB)
                apply(auto) apply(case_tac i) by(auto)
        also have ... = [?B!index ?B x,  ?B!index ?B y]
                  by(simp only: in1 in2)
        also have ... = [x,y] using xlxymB ylxymB  by simp
        finally have end2: ?B = [x,y] .

        then show ?A = ?B  using end1 end2 by simp
      next
        case False
        with whatisq have qsy: q=y by simp
        then have xby: y < x in (mtf2 (length (?s)) (q) (?s))
          apply(simp)
              apply(rule mtf2_moves_to_front''[simplified])
                using z xyininit xny by(simp_all add: dz)
        then have y < x in ?A using Lxy_mono[OF geA'] man dz
by auto
         then have index ?A y < index ?A x unfolding before_in_def
by auto
        then have in1: index ?A y = 0
              and in2: index ?A x = 1  using xindA by auto
        have ?A = [?A!0,?A!1]
              apply(simp only: list_eq_iff_nth_eq)
                apply(auto simp: lA) apply(case_tac i) by(auto)
          also have ... = [?A!index ?A y, ?A!index ?A x] by(simp
only: in1 in2)
        also have ... = [y,x] using xlxymA ylxymA  by simp
        finally have end1: ?A  = [y,x] .

        have y < x in ?B
          using qsy apply(simp)
              apply(rule mtf2_moves_to_front''[simplified])
                using z xyininit xny by(simp_all add: Lxy_distinct
dz Lxy_set_filter)
        then have index ?B y < index ?B x
                unfolding before_in_def by auto
        then have in1: index ?B y = 0
              and in2: index ?B x = 1
```

using *xindB* **by** *auto*

have *?B* = [*?B*!*0*, *?B*!*1*]
apply(*simp only*: *list_eq_iff_nth_eq*)
apply(*simp only*: *lB*)
apply(*auto*) **apply**(*case_tac i*) **by**(*auto*)
also have ... = [*?B*!*index ?B y*, *?B*!*index ?B x*]
by(*simp only*: *in1 in2*)
also have ... = [*y*,*x*] **using** *xlxymB ylxymB* **by**(*simp*)
finally have *end2*: *?B* = [*y*,*x*] **.**

then show *?A* = *?B* **using** *end1 end2* **by** *simp*
qed
qed

have *a2*: *Lxy* (*step* (*?s*) (*q*) (*if ?b* ! (*index init* (*q*)) *then 0 else length* (*?s*), [])) {*x*, *y*}
= *step* (*Lxy* (*?s*) {*x*, *y*}) (*q*) (*if nths* (*?b*) {*index init x*, *index init y*} ! (*index* (*Lxy init* {*x*,*y*}) (*q*))
then 0
else length (*Lxy* (*?s*) {*x*, *y*}), [])
apply(*auto simp add*: *a step_def*) **by**(*simp add*: *b*)

show *?case* **using** *a1 a2* **by**(*simp*)
qed *simp*
also have ... = *?R* (*qs*@[*q*])
using *True* **apply**(*simp add*: *Lxy_snoc take_Suc_conv_app_nth config′_rand_snoc*)
using *iH* **by**(*simp add*: *split_def*)
finally show *?thesis* **.**
next
case *False*
then have *qnx*: (*q*) ≠ *x* **and** *qny*: (*q*) ≠ *y* **by** *auto*

let *?proj*=(λ*a*. (*Lxy* (*fst a*) {*x*, *y*}, (*nths* (*fst* (*snd a*)) {*index init x*, *index init y*}, *Lxy init* {*x*, *y*})))

have *map_pmf ?proj* (*config_rand BIT init* (*qs*@[*q*]))
= *map_pmf ?proj* (*config_rand* (*BIT_init*, *BIT_step*) *init qs*
≫= (λ*p*. *BIT_step p* (*q*) ≫= (λ*pa*. *return_pmf* (*step* (*fst p*) (*q*) (*fst pa*), *snd pa*))))
by (*simp add*: *split_def take_Suc_conv_app_nth config′_rand_snoc*)

also have . . . = *map_pmf ?proj* (*config_rand* (*BIT_init*, *BIT_step*) *init qs*)
apply(*simp add*: *map_pmf_def bind_assoc_pmf bind_return_pmf BIT_step_def*)
proof (*rule bind_pmf_cong*, *goal_cases*)
case (*2 z*)
let *?s* = *fst z*
let *?m* = *snd* (*snd z*)
let *?b* = *fst* (*snd z*)

from *2* **have** *sf_init*: *?m* = *init* **using** *config_n_init3* **by** *auto*

from *2* **have** *ff_len*: *length* (*?b*) = *length init* **using** *config_n_fst_init_length2* **by** *auto*

have *ff_ix*: *index init x* < *length* (*?b*) **unfolding** *ff_len* **using** *a*(*1*) **by** *auto*
have *ff_iy*: *index init y* < *length* (*?b*) **unfolding** *ff_len* **using** *a*(*2*) **by** *auto*
have *ff_q*: *index init* (*q*) < *length* (*?b*) **unfolding** *ff_len* **using** *qininit* **by** *auto*
have *iq_ix*: *index init* (*q*) ≠ *index init x* **using** *a*(*1*) *qnx* **by** *auto*
have *iq_iy*: *index init* (*q*) ≠ *index init y* **using** *a*(*2*) *qny* **by** *auto*
have *ix_iy*: *index init x* ≠ *index init y* **using** *a*(*2*) *xny* **by** *auto*

from *2* **have** *s_set*[*simp*]: *set* (*?s*) = *set init* **using** *config_rand_set* **by** *force*
have *s_xin*: *x*∈*set* (*?s*) **using** *a*(*1*) **by** *simp*
have *s_yin*: *y*∈*set* (*?s*) **using** *a*(*2*) **by** *simp*
from *2* **have** *s_dist*: *distinct* (*?s*) **using** *config_rand_distinct dinit* **by** *force*
have *s_qin*: *q* ∈ *set* (*?s*) **using** *qininit* **by** *simp*

have *fstfst*: *nths* (*flip* (*index ?m* (*q*)) (*?b*)) {*index init x*, *index init y*}
= *nths* (*?b*) {*index init x*, *index init y*} (**is** *nths ?A ?I* = *nths ?B ?I*)
proof (*cases index init x* < *index init y*)
case *True*
have *nths ?A ?I* = [*?A*!*index init x*, *?A*!*index init y*]
apply(*rule nths_project*′)

```
          by(simp_all add: ff_ix ff_iy True)
      also have ... = [?B!index init x, ?B!index init y]
        unfolding sf_init using flip_other ff_ix ff_iy ff_q iq_ix
iq_iy by auto
      also have ... = nths ?B ?I
          apply(rule nths_project'[symmetric])
          by(simp_all add: ff_ix ff_iy True)
      finally show ?thesis .
    next
      case False
     then have yx: index init y < index init x using ix_iy by auto
      have man: ?I =  {index init y, index init x} by auto
      have nths ?A {index init y, index init x}  = [?A!index init y,
?A!index init x]
          apply(rule nths_project')
          by(simp_all add: ff_ix ff_iy yx)
      also have ... = [?B!index init y, ?B!index init x]
        unfolding sf_init using flip_other ff_ix ff_iy ff_q iq_ix
iq_iy by auto
      also have ... = nths ?B {index init y, index init x}
          apply(rule nths_project'[symmetric])
          by(simp_all add: ff_ix ff_iy yx)
      finally show ?thesis by(simp add: man)
    qed


    have snd: Lxy (step (?s) (q)
        (if ?b ! index ?m (q) then 0 else length (?s),
          [])) {x, y} = Lxy (?s) {x, y} (is Lxy ?A {x,y} = Lxy ?B
{x,y})
    proof (cases x < y in ?B)
      case True
      note B=this
         then have A: x<y in ?A apply(auto simp add: step_def
split_def)
        apply(rule x_stays_before_y_if_y_not_moved_to_front)
          by(simp_all add: a s_dist qny[symmetric] qininit)

      have Lxy ?A {x,y} = [x,y]
        apply(rule Lxy_project)
          by(simp_all add: xny set_step distinct_step A s_dist a)
      also have ... = Lxy ?B {x,y}
        apply(rule Lxy_project[symmetric])
          by(simp_all add: xny B s_dist a)
```

finally show *?thesis* **.**
next
case *False*
then have *B*: *y* < *x in ?B* **using** *not_before_in*[*OF s_xin s_yin*] *xny* **by** *simp*
then have *A*: *y* < *x in ?A* **apply**(*auto simp add*: *step_def split_def*)
apply(*rule x_stays_before_y_if_y_not_moved_to_front*)
by(*simp_all add*: *a s_dist qnx*[*symmetric*] *qininit*)
have *man*: {*x*,*y*} = {*y*,*x*} **by** *auto*
have *Lxy ?A* {*y*,*x*} = [*y*,*x*]
apply(*rule Lxy_project*)
by(*simp_all add*: *xny*[*symmetric*] *set_step distinct_step A s_dist a*)
also have ... = *Lxy ?B* {*y*,*x*}
apply(*rule Lxy_project*[*symmetric*])
by(*simp_all add*: *xny*[*symmetric*] *B s_dist a*)
finally show *?thesis* **using** *man* **by** *auto*
qed

show *?case* **by**(*simp add*: *fstfst snd*)
qed *simp*
also have . . . = *config_rand BIT* (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*})
using *iH* **by** (*auto simp*: *split_def*)
also have . . . = *?R* (*qs*@[*q*])
using *False* **by**(*simp add*: *Lxy_snoc*)
finally show *?thesis* **by** (*simp add*: *split_def*)
qed
qed
} **note** *strong*=*this*

{
fix *n*::*nat*
have *Pbefore_in x y BIT qs init* =
map_pmf (λ*p*. *x* < *y in fst p*)
(*map_pmf* (λ(*l*, (*w*, *i*)). (*Lxy l* {*x*, *y*}, (*nths w* {*index init x*, *index init y*}, *Lxy init* {*x*, *y*})))
(*config_rand BIT init qs*))
unfolding *Pbefore_in_def* **apply**(*simp add*: *map_pmf_def bind_return_pmf bind_assoc_pmf split_def*)
apply(*rule bind_pmf_cong*)
apply(*simp*)
proof (*goal_cases*)
case (*1 z*)

let *?s* = *fst z*
from *1* **have** *u*: *set* (*?s*) = *set init* **using** *config_rand*[*of BIT*, *simplified*] **by** *metis*
from *1* **have** *v*: *distinct* (*?s*) **using** *dinit config_rand*[*of BIT*, *simplified*] **by** *metis*
have (*x* < *y in ?s*) = (*x* < *y in Lxy* (*?s*) {*x*, *y*})
apply(*rule Lxy_mono*)
using *u xyininit* **apply**(*simp*)
using *v* **by** *simp*
then show *?case* **by** *simp*
qed
also have ... = *map_pmf* (λ*p*. *x* < *y in fst p*) (*config_rand BIT* (*Lxy init* {*x*, *y*}) (*Lxy qs* {*x*, *y*}))
apply(*subst strong*) **using** *assms* **by** *simp_all*
also have ... = *Pbefore_in x y BIT* (*Lxy qs* {*x*, *y*}) (*Lxy init* {*x*, *y*}) **unfolding** *Pbefore_in_def* **by** *simp*
finally have *Pbefore_in x y BIT qs init* =
Pbefore_in x y BIT (*Lxy qs* {*x*, *y*}) (*Lxy init* {*x*, *y*}) **.**

} **note** *fine=this*

with *assms* **show** *?thesis* **by** *simp*
qed

theorem *BIT_pairwise*: *pairwise BIT*
apply(*rule pairwise_property_lemma*)
apply(*rule BIT_pairwise′*)
by(*simp_all add*: *BIT_step_def*)

end

17 BIT is 1.75 competitive on lists of length 2

theory *BIT_2comp_on2*
imports *BIT Phase_Partitioning*
begin

17.1 auxliary lemmas

17.1.1 *E_bernoulli3*

lemma *E_bernoulli3*: **assumes** *0<p*

and *p*<*1*
and *finite* (*set_pmf* (*bind_pmf* (*bernoulli_pmf p*) *f*))
shows *E* (*bind_pmf* (*bernoulli_pmf p*) *f*) = *E*(*f True*)∗*p* + *E*(*f False*)∗(*1*−*p*)
(**is** *?L* = *?R*)
proof −

have *T*: (∑ *a*∈(⋃ *x*. *set_pmf* (*f x*)). (*a* ∗ *pmf* (*f True*) *a*))
= *E*(*f True*)
unfolding *E_def*
apply(*subst integral_measure_pmf*[*of bind_pmf* (*bernoulli_pmf p*) *f*])
using *assms* **apply**(*simp*)
using *assms* **apply**(*simp add*: *set_pmf_bernoulli*) **apply** *blast*
using *assms* **by**(*simp add*: *set_pmf_bernoulli mult_ac*)
have *F*: (∑ *a*∈(⋃ *x*. *set_pmf* (*f x*)). (*a* ∗ *pmf* (*f False*) *a*))
= *E*(*f False*)
unfolding *E_def*
apply(*subst integral_measure_pmf*[*of bind_pmf* (*bernoulli_pmf p*) *f*])
using *assms* **apply**(*simp*)
using *assms* **apply**(*simp add*: *set_pmf_bernoulli*) **apply** *blast*
using *assms* **by**(*simp add*: *set_pmf_bernoulli mult_ac*)

have *?L* = (∑ *a*∈(⋃ *x*. *set_pmf* (*f x*)).
a ∗
(*pmf* (*f True*) *a* ∗ *p* +
pmf (*f False*) *a* ∗ (*1* − *p*)))
unfolding *E_def*
apply(*subst integral_measure_pmf*[*of bind_pmf* (*bernoulli_pmf p*) *f*])
using *assms* **apply**(*simp*)
apply(*simp*)
using *assms* **apply**(*simp add*: *set_pmf_bernoulli*)
by(*simp add*: *pmf_bind mult_ac*)
also have . . . = (∑ *a*∈(⋃ *x*. *set_pmf* (*f x*)). (*a* ∗ *pmf* (*f True*) *a* ∗ *p*)
+ (*a* ∗ *pmf* (*f False*) *a* ∗ (*1* − *p*)))
apply(*rule sum.cong*) **apply**(*simp*) **by** *algebra*
also have . . . = (∑ *a*∈(⋃ *x*. *set_pmf* (*f x*)). (*a* ∗ *pmf* (*f True*) *a* ∗ *p*))
+ (∑ *a*∈(⋃ *x*. *set_pmf* (*f x*)). (*a* ∗ *pmf* (*f False*) *a* ∗ (*1* −
p)))
by (*simp add*: *sum.distrib*)
also have . . . = (∑ *a*∈(⋃ *x*. *set_pmf* (*f x*)). (*a* ∗ *pmf* (*f True*) *a*)) ∗ *p*
+ (∑ *a*∈(⋃ *x*. *set_pmf* (*f x*)). (*a* ∗ *pmf* (*f False*) *a*)) ∗ (*1* −
p)
by (*simp add*: *sum_distrib_right*)
also have . . . = *?R* **unfolding** *T F* **by** *simp*
finally show *?thesis* **.**

qed

17.1.2 types of configurations

definition *type0 init x y* = *do* {
  (*a*::*bool*) ← (*bernoulli_pmf 0.5*);
  (*b*::*bool*) ← (*bernoulli_pmf 0.5*);
  return_pmf ([*x*,*y*], ([*a*,*b*],*init*))
}

definition *type1 init x y* = *do* {
  (*a*::*bool*) ← (*bernoulli_pmf 0.5*);
  (*b*::*bool*) ← (*bernoulli_pmf 0.5*);
  return_pmf (*if* ~[*a*,*b*]!(*index init x*)∧[*a*,*b*]!(*index init y*) *then* ([*y*,*x*], ([*a*,*b*],*init*))
    else ([*x*,*y*], ([*a*,*b*],*init*)))
}

definition *type3 init x y* = *do* {
  (*a*::*bool*) ← (*bernoulli_pmf 0.5*);
  (*b*::*bool*) ← (*bernoulli_pmf 0.5*);
  return_pmf (*if* [*a*,*b*]!(*index init x*)∧~[*a*,*b*]!(*index init y*) *then* ([*x*,*y*], ([*a*,*b*],*init*))
    else ([*y*,*x*], ([*a*,*b*],*init*)))
}

definition *type4 init x y* = *do* {
  (*a*::*bool*) ← (*bernoulli_pmf 0.5*);
  (*b*::*bool*) ← (*bernoulli_pmf 0.5*);
  return_pmf (*if* ~[*a*,*b*]!(*index init y*) *then* ([*x*,*y*], ([*a*,*b*],*init*))
    else ([*y*,*x*], ([*a*,*b*],*init*)))
}

definition *BIT_inv s x i* == (*s* = (*type0 i x* (*hd* (*filter* (λ*y*. *y*≠*x*) *i*))))

lemma *BIT_inv2*: *x*≠*y* ⟹ *z*∈{*x*,*y*} ⟹ *BIT_inv s z* [*x*,*y*] = (*s*= *type0* [*x*,*y*] *z* (*other z x y*))
  unfolding *BIT_inv_def* **by**(*auto simp add*: *other_def*)

17.1.3 cost of BIT

lemma *costBIT_0x*:
  assumes *x*≠*y x* : {*x0*,*y0*} *y*∈{*x0*,*y0*}
  shows

E (*type0* [*x0*, *y0*] *x* *y* ≫=
(λ*s*. *BIT_step* *s* *x* ≫=
(λ(*a*, *is'*). *return_pmf* (*real* (t_p (*fst* *s*) *x* *a*))))) = *0*
using *assms* **apply**(*auto*)
apply(*simp_all add*: *type0_def BIT_step_def bind_assoc_pmf bind_return_pmf*
)
apply(*simp_all add*: *E_bernoulli3* t_p*_def*)
done

lemma *costBIT_0y*:
assumes *x*≠*y* *x* : {*x0*,*y0*} *y*∈{*x0*,*y0*}
shows
E (*type0* [*x0*, *y0*] *x* *y* ≫=
(λ*s*. *BIT_step* *s* *y* ≫=
(λ(*a*, *is'*). *return_pmf* (*real* (t_p (*fst* *s*) *y* *a*))))) = *1*
using *assms* **apply**(*auto*)
apply(*simp_all add*: *type0_def BIT_step_def bind_assoc_pmf bind_return_pmf*
)
apply(*simp_all add*: *E_bernoulli3* t_p*_def*)
done

lemma *costBIT_1x*:
assumes *x*≠*y* *x* : {*x0*,*y0*} *y*∈{*x0*,*y0*}
shows
E (*type1* [*x0*, *y0*] *x* *y* ≫=
(λ*s*. *BIT_step* *s* *x* ≫=
(λ(*a*, *is'*). *return_pmf* (*real* (t_p (*fst* *s*) *x* *a*))))) = *1/4*
using *assms* **apply**(*auto*)
apply(*simp_all add*: *type1_def BIT_step_def bind_assoc_pmf bind_return_pmf*
)
apply(*simp_all add*: *E_bernoulli3* t_p*_def*)
done

lemma *costBIT_1y*:
assumes *x*≠*y* *x* : {*x0*,*y0*} *y*∈{*x0*,*y0*}
shows
E (*type1* [*x0*, *y0*] *x* *y* ≫=
(λ*s*. *BIT_step* *s* *y* ≫=
(λ(*a*, *is'*). *return_pmf* (*real* (t_p (*fst* *s*) *y* *a*))))) = *3/4*
using *assms* **apply**(*auto*)
apply(*simp_all add*: *type1_def BIT_step_def bind_assoc_pmf bind_return_pmf*
)
apply(*simp_all add*: *E_bernoulli3* t_p*_def*)
done

lemma *costBIT_3x*:
assumes $x \neq y$ $x : \{x0, y0\}$ $y \in \{x0, y0\}$
shows
E (*type3* [*x0*, *y0*] *x y* ≫=
(λ*s*. *BIT_step s x* ≫=
(λ(*a*, *is'*). *return_pmf* (*real* (t_p (*fst s*) *x a*))))) = *3/4*
using *assms* **apply**(*auto*)
apply(*simp_all add*: *type3_def BIT_step_def bind_assoc_pmf bind_return_pmf*
)
apply(*simp_all add*: *E_bernoulli3* t_p*_def*)
done

lemma *costBIT_3y*:
assumes $x \neq y$ $x : \{x0, y0\}$ $y \in \{x0, y0\}$
shows
E (*type3* [*x0*, *y0*] *x y* ≫=
(λ*s*. *BIT_step s y* ≫=
(λ(*a*, *is'*). *return_pmf* (*real* (t_p (*fst s*) *y a*))))) = *1/4*
using *assms* **apply**(*auto*)
apply(*simp_all add*: *type3_def BIT_step_def bind_assoc_pmf bind_return_pmf*
)
apply(*simp_all add*: *E_bernoulli3* t_p*_def*)
done

lemma *costBIT_4x*:
assumes $x \neq y$ $x : \{x0, y0\}$ $y \in \{x0, y0\}$
shows
E (*type4* [*x0*, *y0*] *x y* ≫=
(λ*s*. *BIT_step s x* ≫=
(λ(*a*, *is'*). *return_pmf* (*real* (t_p (*fst s*) *x a*))))) = *0.5*
using *assms* **apply**(*auto*)
apply(*simp_all add*: *type4_def BIT_step_def bind_assoc_pmf bind_return_pmf*
)
apply(*simp_all add*: *E_bernoulli3* t_p*_def*)
done

lemma *costBIT_4y*:
assumes $x \neq y$ $x : \{x0, y0\}$ $y \in \{x0, y0\}$
shows
E (*type4* [*x0*, *y0*] *x y* ≫=
(λ*s*. *BIT_step s y* ≫=
(λ(*a*, *is'*). *return_pmf* (*real* (t_p (*fst s*) *y a*))))) = *0.5*
using *assms* **apply**(*auto*)

apply(*simp_all add*: *type4_def BIT_step_def bind_assoc_pmf bind_return_pmf*
)
apply(*simp_all add*: *E_bernoulli3* t_p*_def*)
done

lemmas *costBIT* = *costBIT_0x costBIT_0y costBIT_1x costBIT_1y costBIT_3x costBIT_3y costBIT_4x costBIT_4y*

17.1.4 state transformation of BIT

abbreviation *BIT_Step s x* == (*s* ≫= (λ*s*. *BIT_step s x* ≫= (λ(*a*, *is'*). *return_pmf* (*step* (*fst s*) *x a*, *is'*))))

lemma *oneBIT_step0x*:
assumes *x*≠*y x* : {*x0*,*y0*} *y*∈{*x0*,*y0*}
shows *BIT_Step* (*type0* [*x0*, *y0*] *x y*) *x* = *type0* [*x0*, *y0*] *x y*
using *assms*
apply(*simp add*: *type0_def BIT_step_def bind_assoc_pmf bind_return_pmf step_def mtf2_def*)
apply(*safe*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *pmf_bind swap_def type0_def*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *add.commute pmf_bind swap_def type0_def*)
done

lemma *oneBIT_step0y*:
assumes *x*≠*y x* : {*x0*,*y0*} *y*∈{*x0*,*y0*}
shows *BIT_Step* (*type0* [*x0*, *y0*] *x y*) *y* = *type4* [*x0*, *y0*] *x y*
using *assms*
apply(*simp add*: *type0_def BIT_step_def bind_assoc_pmf bind_return_pmf step_def mtf2_def*)
apply(*safe*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *add.commute pmf_bind swap_def type4_def*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *pmf_bind swap_def type4_def*)

done

lemma *oneBIT_step1x*:
assumes *x*≠*y x* : {*x0*,*y0*} *y*∈{*x0*,*y0*}
shows *BIT_Step* (*type1* [*x0*, *y0*] *x y*) *x* = *type0* [*x0*, *y0*] *x y*
using *assms*
apply(*simp add*: *type1_def BIT_step_def bind_assoc_pmf bind_return_pmf step_def mtf2_def*)

apply(*safe*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *pmf_bind swap_def type0_def*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *add.commute pmf_bind swap_def type0_def*)
done

lemma *oneBIT_step1y*:
assumes *x≠y x* : {*x0*,*y0*} *y*∈{*x0*,*y0*}
shows *BIT_Step* (*type1* [*x0*, *y0*] *x y*) *y* = *type3* [*x0*, *y0*] *x y*
using *assms*
apply(*simp add*: *type1_def BIT_step_def bind_assoc_pmf bind_return_pmf step_def mtf2_def*)
apply(*safe*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *add.commute pmf_bind swap_def type3_def*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *pmf_bind swap_def type3_def*)

done

lemma *oneBIT_step3x*:
assumes *x≠y x*:{*x0*,*y0*} *y*:{*x0*,*y0*}
shows *BIT_Step* (*type3* [*x0*, *y0*] *x y*) *x* = *type1* [*x0*, *y0*] *x y*
using *assms*
apply(*simp add*: *type3_def BIT_step_def bind_assoc_pmf bind_return_pmf step_def mtf2_def*)
apply(*safe*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *pmf_bind swap_def type1_def*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *add.commute pmf_bind swap_def type1_def*)
done

lemma *oneBIT_step3y*:
assumes *x≠y x* : {*x0*,*y0*} *y*∈{*x0*,*y0*}
shows *BIT_Step* (*type3* [*x0*, *y0*] *x y*) *y* = *type0* [*x0*, *y0*] *y x*
using *assms*
apply(*simp add*: *type3_def BIT_step_def bind_assoc_pmf bind_return_pmf step_def mtf2_def*)
apply(*safe*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *add.commute pmf_bind swap_def type0_def*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *pmf_bind swap_def type0_def*)

done

lemma *oneBIT_step4x*:
assumes $x \neq y$ $x : \{x0, y0\}$ $y \in \{x0, y0\}$
shows *BIT_Step* (*type4* [*x0*, *y0*] *x y*) *x* = *type1* [*x0*, *y0*] *x y*
using *assms*
apply(*simp add*: *type4_def BIT_step_def bind_assoc_pmf bind_return_pmf step_def mtf2_def*)
apply(*safe*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *pmf_bind swap_def type1_def*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *add.commute pmf_bind swap_def type1_def*)
done

lemma *oneBIT_step4y*:
assumes $x \neq y$ $x : \{x0, y0\}$ $y \in \{x0, y0\}$
shows *BIT_Step* (*type4* [*x0*, *y0*] *x y*) *y* = *type0* [*x0*, *y0*] *y x*
using *assms*
apply(*simp add*: *type4_def BIT_step_def bind_assoc_pmf bind_return_pmf step_def mtf2_def*)
apply(*safe*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *add.commute pmf_bind swap_def type0_def*)
apply(*rule pmf_eqI*) **apply**(*simp add*: *pmf_bind swap_def type0_def*)

done

lemmas *oneBIT_step* = *oneBIT_step0x oneBIT_step0y oneBIT_step1x oneBIT_step1y oneBIT_step3x oneBIT_step3y oneBIT_step4x oneBIT_step4y*

17.2 Analysis of the four phase forms

17.2.1 yx

lemma *bit_yx*: **assumes** $x \neq y$
and *kas*: *init* $\in$ {[*x*,*y*],[*y*,*x*]}
and *qs* $\in$ *lang* (*Star*(*Times* (*Atom y*) (*Atom x*)))
shows T_p*_on_rand′ BIT* (*type1 init x y*) (*qs*@*r*) = *0.75* ∗ *length qs* + T_p*_on_rand′ BIT* (*type1 init x y*) *r*
∧ *config′_rand BIT* (*type1 init x y*) *qs* = (*type1 init x y*)
proof −
from *assms* **have** *qs* $\in$ *star* ({[*y*]} @@ {[*x*]}) **by** (*simp*)
from *this assms* **show** *?thesis*
proof (*induct qs rule*: *star_induct*)
case (*append u v*)
then have *uyx*: *u* = [*y*,*x*] **by** *auto*

have *yy*: T_p*_on_rand′ BIT* (*type1 init x y*) (*v* @ *r*) = *0.75*∗*length v* + T_p*_on_rand′ BIT* (*type1 init x y*) *r*
∧ *config′_rand BIT* (*type1 init x y*) *v* = (*type1 init x y*)
apply(*rule append*(*3*))
apply(*fact*)+
using *append*(*2*,*6*) **by**(*simp_all*)

have *s2*: *config′_rand BIT* (*type1 init x y*) [*y*,*x*] = (*type1 init x y*)
using *kas assms*(*1*) **by** (*auto simp add*: *oneBIT_step*)

have *ta*: T_p*_on_rand′ BIT* (*type1 init x y*) *u* = *1.5*
using *kas assms*(*1*)
by(*auto simp add*: *uyx oneBIT_step costBIT_1y costBIT_3x*)

have *config*: *config′_rand BIT* (*type1 init x y*) (*u* @ *v*)
= *type1 init x y* **by** (*simp only*: *config′_rand_append s2 uyx yy*)

have T_p*_on_rand′ BIT* (*type1 init x y*) (*u* @ (*v* @ *r*))
= T_p*_on_rand′ BIT* (*type1 init x y*) *u* + T_p*_on_rand′ BIT* (*config′_rand BIT* (*type1 init x y*) *u*) (*v*@*r*)
by (*simp only*: *T_on_rand′_append*)
also have . . . = T_p*_on_rand′ BIT* (*type1 init x y*) *u* + T_p*_on_rand′ BIT* (*type1 init x y*) (*v*@*r*)
unfolding *uyx* **by**(*simp only*: *s2*)
also have . . . = T_p*_on_rand′ BIT* (*type1 init x y*) *u* + *0.75*∗*length v* + T_p*_on_rand′ BIT* (*type1 init x y*) *r*
by(*simp only*: *yy*)
also have . . . = *2*∗*0.75* + *0.75*∗*length v* + T_p*_on_rand′ BIT* (*type1 init x y*) *r* **by**(*simp add*: *ta*)
also have . . . = *0.75* ∗ (*2*+*length v*) + T_p*_on_rand′ BIT* (*type1 init x y*) *r*
by (*simp add*: *ring_distribs del*: *add_2_eq_Suc′ add_2_eq_Suc*)
also have . . . = *0.75* ∗ *length* (*u* @ *v*) + T_p*_on_rand′ BIT* (*type1 init x y*) *r*
using *uyx* **by** *simp*
finally show *?case* **using** *config* **by** *simp*
qed *simp*
qed

17.2.2 (yx)*yx

lemma *bit_yxyx*: **assumes** $x \neq y$ **and** *kas*: *init* ∈ {[*x*,*y*],[*y*,*x*]} **and**

qs ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
shows T_p*_on_rand′ BIT* (*type0 init x y*) (*qs*@*r*) = *0.75* ∗ *length qs* + T_p*_on_rand′ BIT* (*type1 init x y*) *r*
∧ *config′_rand BIT* (*type0 init x y*) *qs* = (*type1 init x y*)
proof −
obtain *u v* **where** *uu*: *u* ∈ *lang* (*Times* (*Atom y*) (*Atom x*))
and *vv*: *v* ∈ *lang* (*seq*[*Star*(*Times* (*Atom y*) (*Atom x*))])
and *qsuv*: *qs* = *u* @ *v*
using *assms*(*3*) **by** (*auto simp*: *conc_def*)

from *uu* **have** *uyx*: *u* = [*y*,*x*] **by**(*auto*)

from *qsuv uyx* **have** *vqs*: *length v* = *length qs* − *2* **by** *auto*
from *qsuv uyx* **have** *vqs2*: *length v* + *2* = *length qs* **by** *auto*

have *s2*: *config′_rand BIT* (*type0 init x y*) [*y*,*x*] = (*type1 init x y*)
using *kas assms*(*1*) **by**(*auto simp add*: *oneBIT_step*)

have *ta*: T_p*_on_rand′ BIT* (*type0 init x y*) *u* = *1.5*
using *kas assms*(*1*) **by** (*auto simp add*: *uyx oneBIT_step costBIT*)

have *tat*: T_p*_on_rand′ BIT* (*type1 init x y*) (*v* @ *r*) = *0.75*∗*length v* + T_p*_on_rand′ BIT* (*type1 init x y*) *r*
∧ *config′_rand BIT* (*type1 init x y*) *v* = (*type1 init x y*)
apply(*rule bit_yx*)
apply(*fact*)+
using *vv* **by**(*simp_all*)

have *config*: *config′_rand BIT* (*type0 init x y*) (*u* @ *v*) = *type1 init x y*
by(*simp only*: *config′_rand_append s2 uyx tat*)

have T_p*_on_rand′ BIT* (*type0 init x y*) (*u* @ (*v* @ *r*))
= T_p*_on_rand′ BIT* (*type0 init x y*) *u* + T_p*_on_rand′ BIT* (*config′_rand BIT* (*type0 init x y*) *u*) (*v*@*r*) **by** (*simp only*: *T_on_rand′_append*)
also
have . . . = T_p*_on_rand′ BIT* (*type0 init x y*) *u* + T_p*_on_rand′ BIT* (*type1 init x y*) (*v*@*r*) **by**(*simp only*: *uyx s2*)
also
have . . . = T_p*_on_rand′ BIT* (*type0 init x y*) *u* + *0.75*∗*length v* + T_p*_on_rand′ BIT* (*type1 init x y*) *r* **by**(*simp only*: *tat*)
also
have . . . = *2*∗*0.75* + *0.75*∗*length v* + T_p*_on_rand′ BIT* (*type1 init x*

y) *r* **by**(*simp add*: *ta*)
also
  have ... = *0.75* ∗ (*2*+*length v*) + T_p*_on_rand′ BIT* (*type1 init x y*) *r*
by (*simp add*: *ring_distribs del*: *add_2_eq_Suc′ add_2_eq_Suc*)
also
  have ... = *0.75* ∗ *length* (*u* @ *v*) + T_p*_on_rand′ BIT* (*type1 init x y*)
r **using** *uyx* **by** *simp*
finally
  show *?thesis* **using** *qsuv config* **by** *simp*
qed

17.2.3 $x\hat{}+..$

lemma *BIT_x*: **assumes** *x*≠*y*
    init ∈ {[*x*,*y*],[*y*,*x*]} *qs* ∈ *lang* (*Plus* (*Atom x*) *One*)
 shows T_p*_on_rand′ BIT* (*type0 init x y*) (*qs*@*r*) = T_p*_on_rand′ BIT*
(*type0 init x y*) *r*
   ∧ *config′_rand BIT* (*type0 init x y*) *qs* = (*type0 init x y*)
proof −
  have *s*: *config′_rand BIT* (*type0 init x y*) *qs* = *type0 init x y*
    using *assms* **by** (*auto simp add*: *oneBIT_step*)

  have *t*: T_p*_on_rand′ BIT* (*type0 init x y*) *qs* = *0*
    using *assms* **by** (*auto simp add*: *costBIT*)

  show *?thesis* **using** *s t* **by**(*simp add*: *T_on_rand′_append*)
qed

17.2.4 Phase Form A

lemma *BIT_a*: **assumes** *x* ≠ *y*
   init ∈ {[*x*,*y*],[*y*,*x*]}
  qs ∈ *lang* (*seq* [*Plus* (*Atom x*) *One*, *Atom y*, *Atom y*])
  shows *config′_rand BIT* (*type0 init x y*) *qs* = (*type0 init y x*) (**is** *?C*)
    and *b*: T_p*_on_rand′ BIT* (*type0 init x y*) *qs* = *1.5* (**is** *?T*)
proof −
  from *assms*(*3*) **have** *alt*: *qs* = [*x*,*y*,*y*] ∨ *qs* = [*y*,*y*] **apply**(*simp*) **by** *fast-force*
  show *?C*
    using *assms*(*1*,*2*) *alt* **by** (*auto simp add*: *oneBIT_step*)
  show *?T*
    using *assms*(*1*,*2*) *alt* **by**(*auto simp add*: *oneBIT_step costBIT*)
qed

```
lemma bit_a: assumes
    x ≠ y {x, y} = {x0, y0} BIT_inv s x [x0, y0]
    set qs ⊆ {x, y} qs ∈ lang (seq [Plus (Atom x) One, Atom y, Atom y])
 shows
    T_p_on_rand' BIT s qs  ≤ 1.75 ∗ T_p [x,y] qs (OPT2 qs [x,y])
     ∧  BIT_inv (config'_rand BIT s qs) (last qs) [x0, y0]
     ∧ T_p_on_rand' BIT s qs = 1.5
proof −
  from assms have f: x0≠y0 by auto
  from assms(1,3) assms(2)[symmetric] have s: s = type0 [x0,y0] x y
    apply(simp add: BIT_inv2[OF f] other_def) by fast

  from assms(1,2) have kas: [x,y] = [x0,y0] ∨ [x,y] = [y0,x0] by auto

  from assms have lqs: last qs = y by fastforce
  from assms(1,2) kas have p: T_p_on_rand' BIT s qs = 1.5
    unfolding s
    apply(safe)
      apply(rule BIT_a)
        apply(simp) apply(simp) using assms(5) apply(simp)
      apply(rule BIT_a)
        apply(simp) apply(simp) using assms(5) apply(simp)
  done
   with OPT2_A[OF assms(1,5)] have BIT: T_p_on_rand' BIT s qs ≤
1.75 ∗ T_p [x, y] qs (OPT2 qs [x, y]) by auto


  from assms(1,2) kas have config'_rand BIT s qs = type0 [x0, y0] y x
    unfolding s
    apply(safe)
      apply(rule BIT_a)
        apply(simp) apply(simp) using assms(5) apply(simp)
      apply(rule BIT_a)
        apply(simp) apply(simp) using assms(5) apply(simp)
  done

  then have BIT_inv (config'_rand BIT s qs) (last qs) [x0, y0]
    apply(simp)
    using assms(1) kas f lqs by(auto simp add: BIT_inv2 other_def)

  then show ?thesis using BIT s p by simp
qed

lemma bit_a'': a ≠ b =⇒
```

{a, b} = {x, y} ⟹
BIT_inv s a [x, y] ⟹
set qs ⊆ {a, b} ⟹
qs ∈ lang (seq [question (Atom a), Atom b, Atom b]) ⟹
BIT_inv (Partial_Cost_Model.config'_rand BIT s qs) (last qs) [x, y] ∧ T_p_on_rand' BIT s qs = 1.5
using *bit_a[of a b x y]* **by** *blast*

17.2.5 Phase Form B

lemma *BIT_b*: **assumes** *A*: $x \neq y$
init ∈ {[x,y],[y,x]}
v ∈ *lang* (*seq* [*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows T_p_on_rand' BIT (type0 init x y) v = 0.75 ∗ length v − 0.5 (**is** *?T*)
and *config'_rand BIT* (*type0 init x y*) *v* = (*type0 init y x*) (**is** *?C*)
proof −
have *lenvmod*: *length v mod 2 = 0*
proof −
from *assms(3)* **have** *v* ∈ ({[y]} @@ {[x]}) @@ *star*({[y]} @@ {[x]}) @@ {[y]} @@ {[y]} **by**(*simp add*: *conc_assoc*)
then obtain *p q r* **where** *pqr*: *v=p@q@r* **and** *p*∈({[y]} @@ {[x]})
and *q*: *q* ∈ *star* ({[y]} @@ {[x]}) **and** *r* ∈{[y]} @@ {[y]} **by** (*metis concE*)
then have *p* = [y,x] *r*=[y,y] **by** *auto*
with *pqr* **have** *a*: *length v = 4+length q* **by** *auto*

from *q* **have** *b*: *length q mod 2 = 0*
apply(*induct q rule*: *star_induct*) **by** (*auto*)
from *a b* **show** *length v mod 2 = 0* **by** *auto*
qed

from *assms(3)* **have** *v* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
@@ *lang* (*seq*[*Atom y*, *Atom y*]) **by** (*auto simp*: *conc_def*)
then obtain *a b* **where** *aa*: *a* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
and *b* ∈ *lang* (*seq*[*Atom y*, *Atom y*])
and *vab*: *v = a @ b*
by(*erule concE*)
then have *bb*: *b*=[y,y] **by** *auto*
from *vab bb* **have** *lenv*: *length v = length a + 2* **by** *auto*

from *bit_yxyx*[*OF assms*(*1*,*2*) *aa*] **have** *stars*: T_p*_on_rand′ BIT* (*type0 init x y*) (*a* @ *b*) = *0.75* ∗ *length a* + T_p*_on_rand′ BIT* (*type1 init x y*) *b*
and *s2*: *config′_rand BIT* (*type0 init x y*) *a* = *type1 init x y* **by** *fast+*

have *t*: T_p*_on_rand′ BIT* (*type1 init x y*) *b* = *1*
using *assms*(*1*,*2*) **by** (*auto simp add*: *oneBIT_step costBIT bb*)

have *s*: *config′_rand BIT* (*type1 init x y*) [*y*, *y*] = *type0 init y x*
using *assms*(*1*,*2*) **by** (*auto simp add*: *oneBIT_step*)

have *config*: *config′_rand BIT* (*type0 init x y*) (*a* @ *b*) = *type0 init y x*
by (*simp only*: *config′_rand_append s2 vab bb s*)

have *calc*: *3* ∗ *Suc* (*Suc* (*length a*)) / *4* − *1* / *2* = *3* ∗ (*2*+*length a*) / *4* − *1* / *2* **by** *simp*

from *t stars* **have** T_p*_on_rand′ BIT* (*type0 init x y*) (*a* @ *b*) = *0.75* ∗ *length a* + *1* **by** *auto*
then show T_p*_on_rand′ BIT* (*type0 init x y*) *v* = *0.75* ∗ *length v* − *0.5*
unfolding *lenv* **by**(*simp add*: *vab ring_distribs del*: *add_2_eq_Suc′*)
from *config vab* **show** *?C* **by** *simp*
qed

lemma *bit_b′′1*: **assumes**
x ≠ *y* {*x*, *y*} = {*x0*, *y0*} *BIT_inv s x* [*x0*, *y0*]
set qs ⊆ {*x*, *y*}
qs ∈ *lang* (*seq*[*Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows *BIT_inv* (*config′_rand BIT s qs*) (*last qs*) [*x0*, *y0*] ∧
T_p*_on_rand′ BIT s qs* = *0.75* ∗ *length qs* − *0.5*
proof −
from *assms* **have** *f*: *x0*≠*y0* **by** *auto*
from *assms*(*1*,*3*) *assms*(*2*)[*symmetric*] **have** *s*: *s* = *type0* [*x0*,*y0*] *x y*
apply(*simp add*: *BIT_inv2*[*OF f*] *other_def*) **by** *fast*

from *assms*(*1*,*2*) **have** *kas*: [*x*,*y*] = [*x0*,*y0*] ∨ [*x*,*y*] = [*y0*,*x0*] **by** *auto*

from *assms*(*5*) **have** *lqs*: *last qs* = *y* **by** *fastforce*
from *assms*(*1*,*2*) *kas* **have** *BIT*: T_p*_on_rand′ BIT s qs* = *0.75* ∗ *length*

qs − *0.5*
unfolding *s*
apply(*safe*)
apply(*rule BIT_b*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp add*: *conc_assoc*)
apply(*rule BIT_b*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp add*: *conc_assoc*)
done

from *assms*(*1*,*2*) *kas* **have** *config'_rand BIT s qs* = *type0* [*x0*, *y0*] *y x*
unfolding *s*
apply(*safe*)
apply(*rule BIT_b*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp add*: *conc_assoc*)
apply(*rule BIT_b*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp add*: *conc_assoc*)
done

then have *config*: *BIT_inv* (*config'_rand BIT s qs*) (*last qs*) [*x0*, *y0*]
apply(*simp*)
using *assms*(*1*) *kas lqs* **by**(*auto simp add*: *BIT_inv2 other_def*)

show *?thesis* **using** *BIT config* **by** *simp*
qed

lemma *BIT_b2*: **assumes** *A*: *x* ≠ *y*
init ∈ {[*x*,*y*],[*y*,*x*]}
v ∈ *lang* (*seq* [*Atom x*, *Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows T_p*_on_rand' BIT* (*type0 init x y*) *v* = *0.75* ∗ (*length v* − *1*) − *0.5* (**is** *?T*)
and *config'_rand BIT* (*type0 init x y*) *v* = (*type0 init y x*) (**is** *?C*)
proof −
from *assms*(*3*) **obtain** *w* **where** *vw*: *v* = [*x*]@*w* **and**
w: *w* ∈ *lang* (*seq* [*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
by(*auto*)
have *c1*: *config'_rand BIT* (*type0 init x y*) [*x*] = *type0 init x y*
using *assms* **by**(*auto simp add*: *oneBIT_step*)
have *t1*: T_p*_on_rand' BIT* (*type0 init x y*) [*x*] = *0*
using *assms* **by**(*auto simp add*: *costBIT*)
show T_p*_on_rand' BIT* (*type0 init x y*) *v*
= *0.75* ∗ (*length v* − *1*) − *0.5*

unfolding *vw* **apply**(*simp only*: *T_on_rand'_append c1 BIT_b*[*OF assms*(*1*,*2*) *w*] *t1*)
by (*simp*)
show *config'_rand BIT* (*type0 init x y*) *v* = (*type0 init y x*)
unfolding *vw* **by**(*simp only*: *config'_rand_append c1 BIT_b*[*OF assms*(*1*,*2*) *w*])
qed

lemma *bit_b''2*: **assumes**
$x \neq y$ $\{x, y\} = \{x0, y0\}$ *BIT_inv s x* [*x0*, *y0*]
set qs $\subseteq \{x, y\}$
qs $\in$ *lang* (*seq*[*Atom x*, *Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows *BIT_inv* (*config'_rand BIT s qs*) (*last qs*) [*x0*, *y0*] $\wedge$
T_p*_on_rand' BIT s qs* = *0.75* $*$ (*length qs* − *1*) − *0.5*
proof −
from *assms* **have** *f*: *x0*$\neq$*y0* **by** *auto*
from *assms*(*1*,*3*) *assms*(*2*)[*symmetric*] **have** *s*: *s* = *type0* [*x0*,*y0*] *x y*
apply(*simp add*: *BIT_inv2*[*OF f*] *other_def*) **by** *fast*

from *assms*(*1*,*2*) **have** *kas*: [*x*,*y*] = [*x0*,*y0*] $\vee$ [*x*,*y*] = [*y0*,*x0*] **by** *auto*

from *assms*(*5*) **have** *lqs*: *last qs* = *y* **by** *fastforce*
from *assms*(*1*,*2*) *kas* **have** *BIT*: T_p*_on_rand' BIT s qs* = *0.75* $*$ (*length qs*−*1*) − *0.5*
unfolding *s*
apply(*safe*)
apply(*rule BIT_b2*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp add*: *conc_assoc*)
apply(*rule BIT_b2*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp add*: *conc_assoc*)
done

from *assms*(*1*,*2*) *kas* **have** *config'_rand BIT s qs* = *type0* [*x0*, *y0*] *y x*
unfolding *s*
apply(*safe*)
apply(*rule BIT_b2*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp add*: *conc_assoc*)
apply(*rule BIT_b2*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp add*: *conc_assoc*)
done

then have *config*: *BIT_inv* (*config'_rand BIT s qs*) (*last qs*) [*x0*, *y0*]
apply(*simp*)

using *assms*(*1*) *kas lqs* **by**(*auto simp add*: *BIT_inv2 other_def*)

show *?thesis* **using** *BIT config* **by** *simp*
qed

lemma *bit_b*: **assumes** $x \neq y$
init ∈ {[*x*,*y*],[*y*,*x*]}
qs ∈ *lang* (*seq*[*Plus* (*Atom x*) *One*, *Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows T_p*_on_rand' BIT* (*type0 init x y*) *qs* ≤ *1.75* ∗ T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*])
and *config'_rand BIT* (*type0 init x y*) *qs* = *type0 init y x*
proof −
obtain *u v* **where** *uu*: *u* ∈ *lang* (*Plus* (*Atom x*) *One*)
and *vv*: *v* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
and *qsuv*: *qs* = *u* @ *v*
using *assms*
by (*auto simp*: *conc_def*)
have *lenv*: *length v mod 2* = *0* ∧ *last v* = *y* ∧ *v*≠[]
proof −
from *vv* **have** *v* ∈ ({[*y*]} @@ {[*x*]}) @@ *star*({[*y*]} @@ {[*x*]}) @@ {[*y*]} @@ {[*y*]} **by** *simp*
then obtain *p q r* **where** *pqr*: *v*=*p*@*q*@*r* **and** *p*∈({[*y*]} @@ {[*x*]})
and *q*: *q* ∈ *star* ({[*y*]} @@ {[*x*]}) **and** *r* ∈{[*y*]} @@ {[*y*]} **by** (*metis concE*)
then have *rr*: *p* = [*y*,*x*] *r*=[*y*,*y*] **by** *auto*
with *pqr* **have** *a*: *length v* = *4*+*length q* **by** *auto*

have *last v* = *last r* **using** *pqr rr* **by** *auto*
then have *c*: *last v* = *y* **using** *rr* **by** *auto*

from *q* **have** *b*: *length q mod 2* = *0*
apply(*induct q rule*: *star_induct*) **by** (*auto*)
from *a b c* **show** *?thesis* **by** *auto*
qed

from *vv* **have** *v* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
@@ *lang* (*seq*[*Atom y*, *Atom y*]) **by** (*auto simp*: *conc_def*)
then obtain *a b* **where** *aa*: *a* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
and *b* ∈ *lang* (*seq*[*Atom y*, *Atom y*])

and *vab*: $v = a @ b$
by(*erule concE*)

from *BIT_x*[*OF assms*(*1*,*2*) *uu*] **have** *u_t*: T_p*_on_rand′ BIT* (*type0 init x y*) (*u @ v*) = T_p*_on_rand′ BIT* (*type0 init x y*) *v*
and *u_c*: *config′_rand BIT* (*type0 init x y*) *u* = *type0 init x y* **by** *auto*
from *BIT_b*[*OF assms*(*1*,*2*) *vv*] **have** *b_t*: T_p*_on_rand′ BIT* (*type0 init x y*) *v* = *0.75* ∗ *length v* − *0.5*
and *b_c*: *config′_rand BIT* (*type0 init x y*) *v* = (*type0 init y x*) **by** *auto*

have *BIT*: T_p*_on_rand′ BIT* (*type0 init x y*) *qs* = *0.75* ∗ *length v* − *0.5*
by(*simp add*: *qsuv u_t b_t*)

from *uu* **have** *uuu*: *u*=[] ∨ *u*=[*x*] **by** *auto*
have *OPT*: T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) = (*length v*) *div 2* **apply**(*rule OPT2_B*) **by**(*fact*)+

from *lenv* **have** *v* ≠ [] *last v* = *y* **by** *auto*
then have *1*: *last qs* = *y* **using** *last_appendR qsuv* **by** *simp*
then have *2*: *other* (*last qs*) *x y* = *x* **unfolding** *other_def* **by** *simp*

show T_p*_on_rand′ BIT* (*type0 init x y*) *qs* ≤ *1.75* ∗ T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*])
using *BIT OPT lenv* **by** *auto*

show *config′_rand BIT* (*type0 init x y*) *qs* = *type0 init y x*
by (*auto simp add*: *config′_rand_append qsuv u_c b_c*)
qed

lemma *bit_b′′*: **assumes**
x ≠ *y* {*x*, *y*} = {*x0*, *y0*} *BIT_inv s x* [*x0*, *y0*]
set qs ⊆ {*x*, *y*}
qs ∈ *lang* (*seq*[*Plus* (*Atom x*) *One*, *Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom y*, *Atom y*])
shows
T_p*_on_rand′ BIT s qs* ≤ *1.75* ∗ T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*])

```
    ∧  BIT_inv (config'_rand BIT s qs) (last qs) [x0, y0]
proof -
  from assms have f: x0≠y0 by auto
  from assms(1,3) assms(2)[symmetric] have s: s = type0 [x0,y0] x y
    apply(simp add: BIT_inv2[OF f] other_def) by fast

  from assms(1,2) have kas: [x,y] = [x0,y0] ∨ [x,y] = [y0,x0] by auto

  from assms(5) have lqs: last qs = y by fastforce
  from assms(1,2) kas have BIT: T_p_on_rand' BIT s qs ≤ 1.75 * T_p
[x, y] qs (OPT2 qs [x, y])
    unfolding s
    apply(safe)
      apply(rule bit_b)
        apply(simp) apply(simp) using assms(5) apply(simp)
      apply(rule bit_b)
        apply(simp) apply(simp) using assms(5) apply(simp)
  done

  from assms(1,2) kas have config'_rand BIT s qs = type0 [x0, y0] y x
    unfolding s
    apply(safe)
      apply(rule bit_b)
        apply(simp) apply(simp) using assms(5) apply(simp)
      apply(rule bit_b)
        apply(simp) apply(simp) using assms(5) apply(simp)
  done

  then have BIT_inv (config'_rand BIT s qs) (last qs) [x0, y0]
    apply(simp)
    using assms(1) kas lqs by(auto simp add: BIT_inv2 other_def)

  then show ?thesis using BIT s by simp
qed

lemma bit_b''': a ≠ b ⟹
         {a, b} = {x, y} ⟹
         BIT_inv s a [x, y] ⟹
         set qs ⊆ {a, b} ⟹
           qs ∈ lang (seq[Plus (Atom x) One, Atom y, Atom x, Star(Times
(Atom y) (Atom x)), Atom y, Atom y]) ⟹
         BIT_inv (Partial_Cost_Model.config'_rand BIT s qs) (last qs) [x,
y] ∧ T_p_on_rand' BIT s qs = 1.5
using bit_a[of a b x y] oops
```

17.2.6 Phase Form C

lemma *BIT_c*: **assumes** $x \neq y$
init ∈ {[*x*,*y*],[*y*,*x*]}
v ∈ *lang* (*seq* [*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
shows T_p*_on_rand′ BIT* (*type0 init x y*) *v* = *0.75* ∗ *length v* − *0.5*
and *config′_rand BIT* (*type0 init x y*) *v* = (*type0 init x y*) (**is** *?C*)
proof −
have *A*: *x*≠*y* **using** *assms* **by** *auto*

from *assms*(*3*) **have** *v* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
@@ *lang* (*seq*[*Atom x*]) **by** (*auto simp*: *conc_def*)
then obtain *a b* **where** *aa*: *a* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
and *b* ∈ *lang* (*seq*[*Atom x*])
and *vab*: *v* = *a* @ *b*
by(*erule concE*)
then have *bb*: *b*=[*x*] **by** *auto*
from *aa* **have** *lena*: *length a* > *0* **by** *auto*
from *vab bb* **have** *lenv*: *length v* = *length a* + *1* **by** *auto*

from *bit_yxyx assms*(*1*,*2*) *aa* **have** *stars*: T_p*_on_rand′ BIT* (*type0 init x y*) (*a* @ *b*) = *0.75* ∗ *length a* + T_p*_on_rand′ BIT* (*type1 init x y*) *b*
and *s2*: *config′_rand BIT* (*type0 init x y*) *a* = *type1 init x y* **by** *fast*+

have *t*: T_p*_on_rand′ BIT* (*type1 init x y*) *b* = *1/4*
using *assms*(*1*,*2*) **by** (*auto simp add*: *bb costBIT*)

have *s*: *config′_rand BIT* (*type1 init x y*) *b* = *type0 init x y*
using *assms*(*1*,*2*) **by** (*auto simp add*: *bb oneBIT_step1x*)

have *config*: *config′_rand BIT* (*type0 init x y*) (*a* @ *b*) = *type0 init x y*
by (*simp only*: *config′_rand_append s2 vab s*)

have *calc*: *3* ∗ *Suc* (*Suc* (*length a*)) / *4* − *1* / *2* = *3* ∗ (*2*+*length a*) / *4* − *1* / *2* **by** *simp*

from *t stars* **have** T_p*_on_rand′ BIT* (*type0 init x y*) (*a* @ *b*) = *0.75* ∗ *length a* + *1/4* **by** *auto*

then show T_p*_on_rand' BIT (type0 init x y) v = 0.75 * length v − 0.5* **unfolding** *lenv*
by(*simp add*: *vab ring_distribs del*: *add_2_eq_Suc'*)
from *config vab* **show** *?C* **by** *simp*
qed

lemma *bit_c''1*: **assumes**
x ≠ y {x, y} = {x0, y0} BIT_inv s x [x0, y0]
set qs ⊆ {x, y}
qs ∈ lang (seq[Atom y, Atom x, Star(Times (Atom y) (Atom x)), Atom x])
shows *BIT_inv (config'_rand BIT s qs) (last qs) [x0, y0] ∧*
T_p*_on_rand' BIT s qs = 0.75 * length qs − 0.5*
proof −
from *assms* **have** *f*: *x0≠y0* **by** *auto*
from *assms(1,3) assms(2)[symmetric]* **have** *s*: *s = type0 [x0,y0] x y*
apply(*simp add*: *BIT_inv2[OF f] other_def*) **by** *fast*

from *assms(1,2)* **have** *kas*: *[x,y] = [x0,y0] ∨ [x,y] = [y0,x0]* **by** *auto*

from *assms(5)* **have** *lqs*: *last qs = x* **by** *fastforce*
from *assms(1,2) kas* **have** *BIT*: T_p*_on_rand' BIT s qs = 0.75 * length qs − 0.5*
unfolding *s*
apply(*safe*)
apply(*rule BIT_c*)
apply(*simp*) **apply**(*simp*) **using** *assms(5)* **apply**(*simp add*: *conc_assoc*)
apply(*rule BIT_c*)
apply(*simp*) **apply**(*simp*) **using** *assms(5)* **apply**(*simp add*: *conc_assoc*)
done

from *assms(1,2) kas* **have** *config'_rand BIT s qs = type0 [x0, y0] x y*
unfolding *s*
apply(*safe*)
apply(*rule BIT_c*)
apply(*simp*) **apply**(*simp*) **using** *assms(5)* **apply**(*simp add*: *conc_assoc*)
apply(*rule BIT_c*)
apply(*simp*) **apply**(*simp*) **using** *assms(5)* **apply**(*simp add*: *conc_assoc*)
done

then have *config*: *BIT_inv (config'_rand BIT s qs) (last qs) [x0, y0]*
apply(*simp*)
using *assms(1) kas lqs* **by**(*auto simp add*: *BIT_inv2 other_def*)

show *?thesis* **using** *BIT config* **by** *simp*
qed

lemma *bit_c*: **assumes** $x \neq y$
init ∈ {[*x*,*y*],[*y*,*x*]}
qs ∈ *lang* (*seq*[*Plus* (*Atom x*) *One*, *Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom x*])
shows $T_p\_on\_rand'$ *BIT* (*type0 init x y*) *qs* ≤ *1.75* ∗ T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*])
and *config'_rand BIT* (*type0 init x y*) *qs* = *type0 init x y*
proof −
obtain *u v* **where** *uu*: *u* ∈ *lang* (*Plus* (*Atom x*) *One*)
and *vv*: *v* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
and *qsuv*: *qs* = *u* @ *v*
using *assms*
by (*auto simp*: *conc_def*)
have *lenv*: *length v mod 2* = *1* ∧ *length v* ≥ *3* ∧ *last v* = *x*
proof −
from *vv* **have** *v* ∈ ({[*y*]} @@ {[*x*]}) @@ *star*({[*y*]} @@ {[*x*]}) @@ {[*x*]} **by** *auto*
then obtain *p q r* **where** *pqr*: *v*=*p*@*q*@*r* **and** *p*∈({[*y*]} @@ {[*x*]})
and *q*: *q* ∈ *star* ({[*y*]} @@ {[*x*]}) **and** *r* ∈ {[*x*]} **by** (*metis concE*)
then have *rr*: *p* = [*y*,*x*] *r*=[*x*] **by** *auto*
with *pqr* **have** *a*: *length v* = *3*+*length q* **by** *auto*

have *last v* = *last r* **using** *pqr rr* **by** *auto*
then have *c*: *last v* = *x* **using** *rr* **by** *auto*

from *q* **have** *b*: *length q mod 2* = *0*
apply(*induct q rule*: *star_induct*) **by** (*auto*)
from *a b c* **show** *length v mod 2* = *1* ∧ *length v* ≥ *3* ∧ *last v* = *x* **by** *auto*
qed

from *vv* **have** *v* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
@@ *lang* (*seq*[*Atom x*]) **by** (*auto simp*: *conc_def*)
then obtain *a b* **where** *aa*: *a* ∈ *lang* (*seq*[*Times* (*Atom y*) (*Atom x*), *Star*(*Times* (*Atom y*) (*Atom x*))])
and *b* ∈ *lang* (*seq*[*Atom x*])
and *vab*: *v* = *a* @ *b*
by(*erule concE*)

from *BIT_x*[*OF assms*(*1*,*2*) *uu*] **have** *u_t*: T_p*_on_rand′ BIT* (*type0 init x y*) (*u* @ *v*) = T_p*_on_rand′ BIT* (*type0 init x y*) *v*
and *u_c*: *config′_rand BIT* (*type0 init x y*) *u* = *type0 init x y* **by** *auto*
from *BIT_c*[*OF assms*(*1*,*2*) *vv*] **have** *b_t*: T_p*_on_rand′ BIT* (*type0 init x y*) *v* = *0.75* ∗ *length v* − *0.5*
and *b_c*: *config′_rand BIT* (*type0 init x y*) *v* = (*type0 init x y*) **by** *auto*

have *BIT*: T_p*_on_rand′ BIT* (*type0 init x y*) *qs* = *0.75* ∗ *length v* − *0.5*
by(*simp add*: *qsuv u_t b_t*)

from *uu* **have** *uuu*: *u*=[] ∨ *u*=[*x*] **by** *auto*
from *vv* **have** *vvv*: *v* ∈ *lang* (*seq*
[*Atom y*, *Atom x*,
Star (*Times* (*Atom y*) (*Atom x*)),
Atom x]) **by**(*auto simp*: *conc_def*)
have *OPT*: T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*]) = (*length v*) *div 2* **apply**(*rule OPT2_C*) **by**(*fact*)+

from *lenv* **have** *v* ≠ [] *last v* = *x* **by** *auto*
then have *1*: *last qs* = *x* **using** *last_appendR qsuv* **by** *simp*
then have *2*: *other* (*last qs*) *x y* = *y* **unfolding** *other_def* **by** *simp*

have *vgt3*: *length v* ≥ *3* **using** *lenv* **by** *simp*
have T_p*_on_rand′ BIT* (*type0 init x y*) *qs* = *0.75* ∗ *length v* − *0.5* **using** *BIT* **by** *simp*
also
have . . . ≤ *1.75* ∗ (*length v* − *1*)/*2*
proof −
have *10* + *6* ∗ *length v* ≤ *7* ∗ *Suc* (*length v*)
⟷ *10* + *6* ∗ *length v* ≤ *7* ∗ *length v* + *7* **by** *auto*
also have . . . ⟷ *3* ≤ *length v* **by** *auto*
also have . . . ⟷ *True* **using** *vgt3* **by** *auto*
finally have *A*: *6* ∗ *length v* − *4* ≤ *7* ∗ (*length v* − *1*) **by** *simp*
show *?thesis* **apply**(*simp*) **using** *A* **by** *linarith*
qed
also
have . . . = *1.75* ∗ (*length v div 2*)

proof −
from *div_mult_mod_eq* **have** *length v* = *length v div 2* ∗ *2* + *length v mod 2* **by** *auto*
with *lenv* **have** *length v* = *length v div 2* ∗ *2* + *1* **by** *auto*
then have (*length v* − *1*) / *2* = *length v div 2* **by** *simp*
then show *?thesis* **by** *simp*
qed
also
have ... = *1.75* ∗ T_p [*x*, *y*] *qs* (*OPT2 qs* [*x*, *y*]) **using** *OPT* **by** *auto*
finally
show T_p*_on_rand′ BIT* (*type0 init x y*) *qs* ≤ *1.75* ∗ T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*])
using *BIT OPT lenv 1 2* **by** *auto*

show *config′_rand BIT* (*type0 init x y*) *qs* = *type0 init x y*
by (*auto simp add*: *config′_rand_append qsuv u_c b_c*)
qed

lemma *bit_c′′*: **assumes**
x ≠ *y* {*x*, *y*} = {*x0*, *y0*} *BIT_inv s x* [*x0*, *y0*]
set qs ⊆ {*x*, *y*}
qs ∈ *lang* (*seq*[*Plus* (*Atom x*) *One*, *Atom y*, *Atom x*, *Star*(*Times* (*Atom y*) (*Atom x*)), *Atom x*])
shows
T_p*_on_rand′ BIT s qs* ≤ *1.75* ∗ T_p [*x*,*y*] *qs* (*OPT2 qs* [*x*,*y*])
∧ *BIT_inv* (*config′_rand BIT s qs*) (*last qs*) [*x0*, *y0*]
proof −
from *assms* **have** *f*: *x0*≠*y0* **by** *auto*
from *assms*(*1*,*3*) *assms*(*2*)[*symmetric*] **have** *s*: *s* = *type0* [*x0*,*y0*] *x y*
apply(*simp add*: *BIT_inv2*[*OF f*] *other_def*) **by** *fast*

from *assms*(*1*,*2*) **have** *kas*: [*x*,*y*] = [*x0*,*y0*] ∨ [*x*,*y*] = [*y0*,*x0*] **by** *auto*

from *assms* **have** *lqs*: *last qs* = *x* **by** *fastforce*
from *assms*(*1*,*2*) *kas* **have** *BIT*: T_p*_on_rand′ BIT s qs* ≤ *1.75* ∗ T_p [*x*, *y*] *qs* (*OPT2 qs* [*x*, *y*])
unfolding *s*
apply(*safe*)
apply(*rule bit_c*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp*)
apply(*rule bit_c*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp*)

done

from *assms*(*1*,*2*) *kas* **have** *config′_rand BIT s qs* = *type0* [*x0*, *y0*] *x y*
unfolding *s*
apply(*safe*)
apply(*rule bit_c*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp*)
apply(*rule bit_c*)
apply(*simp*) **apply**(*simp*) **using** *assms*(*5*) **apply**(*simp*)
done

then have *BIT_inv* (*config′_rand BIT s qs*) (*last qs*) [*x0*, *y0*]
apply(*simp*)
using *assms*(*1*) *kas f lqs* **by**(*auto simp add*: *BIT_inv2 other_def*)

then show *?thesis* **using** *BIT s* **by** *simp*
qed

lemma *BIT_c2*: **assumes** *A*: $x \neq y$
init ∈ {[*x*,*y*],[*y*,*x*]}
v ∈ *lang* (*seq* [*Atom x*, *Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
shows T_p*_on_rand′ BIT* (*type0 init x y*) *v* = *0.75* ∗ (*length v* − *1*) − *0.5* (**is** *?T*)
and *config′_rand BIT* (*type0 init x y*) *v* = (*type0 init x y*) (**is** *?C*)
proof −
from *assms*(*3*) **obtain** *w* **where** *vw*: *v* = [*x*]@*w* **and**
w: *w* ∈ *lang* (*seq* [*Times* (*Atom y*) (*Atom x*), *Star* (*Times* (*Atom y*) (*Atom x*)), *Atom x*])
by(*auto*)
have *c1*: *config′_rand BIT* (*type0 init x y*) [*x*] = *type0 init x y*
using *assms* **by**(*auto simp add*: *oneBIT_step*)
have *t1*: T_p*_on_rand′ BIT* (*type0 init x y*) [*x*] = *0*
using *assms* **by**(*auto simp add*: *costBIT*)
show T_p*_on_rand′ BIT* (*type0 init x y*) *v*
= *0.75* ∗ (*length v* − *1*) − *0.5*
unfolding *vw* **apply**(*simp only*: *T_on_rand′_append c1 BIT_c*[*OF assms*(*1*,*2*) *w*] *t1*)
by (*simp*)
show *config′_rand BIT* (*type0 init x y*) *v* = (*type0 init x y*)
unfolding *vw* **by**(*simp only*: *config′_rand_append c1 BIT_c*[*OF assms*(*1*,*2*)

w])
qed

lemma *bit_c''2*: **assumes**
x ≠ y {x, y} = {x0, y0} BIT_inv s x [x0, y0]
set qs ⊆ {x, y}
qs ∈ lang (seq[Atom x, Atom y, Atom x, Star(Times (Atom y) (Atom x)), Atom x])
shows *BIT_inv (config'_rand BIT s qs) (last qs) [x0, y0] ∧*
T_p_on_rand' BIT s qs = 0.75 ∗ (length qs − 1) − 0.5
proof −
from *assms* **have** *f*: *x0≠y0* **by** *auto*
from *assms(1,3) assms(2)[symmetric]* **have** *s*: *s = type0 [x0,y0] x y*
apply(*simp add*: *BIT_inv2[OF f] other_def*) **by** *fast*

from *assms(1,2)* **have** *kas*: *[x,y] = [x0,y0] ∨ [x,y] = [y0,x0]* **by** *auto*

from *assms(5)* **have** *lqs*: *last qs = x* **by** *fastforce*
from *assms(1,2) kas* **have** *BIT*: *T_p_on_rand' BIT s qs = 0.75 ∗ (length qs−1) − 0.5*
unfolding *s*
apply(*safe*)
apply(*rule BIT_c2*)
apply(*simp*) **apply**(*simp*) **using** *assms(5)* **apply**(*simp add*: *conc_assoc*)
apply(*rule BIT_c2*)
apply(*simp*) **apply**(*simp*) **using** *assms(5)* **apply**(*simp add*: *conc_assoc*)
done

from *assms(1,2) kas* **have** *config'_rand BIT s qs = type0 [x0, y0] x y*
unfolding *s*
apply(*safe*)
apply(*rule BIT_c2*)
apply(*simp*) **apply**(*simp*) **using** *assms(5)* **apply**(*simp add*: *conc_assoc*)
apply(*rule BIT_c2*)
apply(*simp*) **apply**(*simp*) **using** *assms(5)* **apply**(*simp add*: *conc_assoc*)
done

then have *config*: *BIT_inv (config'_rand BIT s qs) (last qs) [x0, y0]*
apply(*simp*)
using *assms(1) kas lqs* **by**(*auto simp add*: *BIT_inv2 other_def*)

show *?thesis* **using** *BIT config* **by** *simp*
qed

17.2.7 Phase Form D

lemma *bit_d*: **assumes**
  $x \neq y$ $\{x, y\} = \{x0, y0\}$ *BIT_inv s x* $[x0, y0]$
  set qs $\subseteq \{x, y\}$ *qs* $\in$ *lang* (*seq* [*Atom x*, *Atom x*])
 shows T_p*_on_rand′ BIT s qs* $\leq 175 / 10^2 *$ *real* (T_p [*x*, *y*] *qs* (*OPT2 qs* [*x*, *y*])) ∧
  BIT_inv (*config′_rand BIT s qs*) (*last qs*) [*x0*, *y0*] ∧
  T_p*_on_rand′ BIT s qs* = *0*
proof −
 from *assms* **have** *qs*: *qs* = [*x*,*x*] **by** *auto*
 then have *OPT*: T_p [*x*, *y*] *qs* (*OPT2 qs* [*x*, *y*]) = *0* **by** (*simp add*: t_p*_def step_def*)

 from *assms* **have** *f*: *x0*≠*y0* **by** *auto*
 from *assms*(*1*,*3*) *assms*(*2*)[*symmetric*] **have** *s*: *s* = *type0* [*x0*,*y0*] *x y*
  apply(*simp add*: *BIT_inv2*[*OF f*] *other_def*) **by** *fast*

 from *assms*(*1*,*2*) **have** *kas*: [*x*,*y*] = [*x0*,*y0*] ∨ [*x*,*y*] = [*y0*,*x0*] **by** *auto*

 have *BIT*: T_p*_on_rand′ BIT* (*type0* [*x0*,*y0*] *x y*) *qs* = *0*
  using *kas assms*(*1*,*2*) **by** (*auto simp add*: *qs oneBIT_step costBIT*)

 have *lqs*: *last qs* = *x last qs* ∈ {*x0*,*y0*} **using** *assms*(*2*,*4*) *qs* **by** *auto*

 have *inv*: *config′_rand BIT s qs* = *type0* [*x0*, *y0*] *x y*
  using *kas assms*(*1*,*2*) **by** (*auto simp add*: *qs s oneBIT_step0x*)

 then have *BIT_inv* (*config′_rand BIT s qs*) (*last qs*) [*x0*, *y0*]
  apply(*simp*)
  using *assms*(*1*) *kas f lqs* **by**(*auto simp add*: *BIT_inv2 other_def*)

 then show *?thesis* **using** *BIT s* **by**(*auto*)
qed

lemma *bit_d′*: **assumes**
  $x \neq y$ $\{x, y\} = \{x0, y0\}$ *BIT_inv s x* $[x0, y0]$
  set qs $\subseteq \{x, y\}$ *qs* $\in$ *lang* (*seq* [*Atom x*, *Atom x*])
 shows *BIT_inv* (*config′_rand BIT s qs*) (*last qs*) [*x0*, *y0*] ∧
  T_p*_on_rand′ BIT s qs* = *0*
using *bit_d*[*OF assms*] **by** *blast*

17.3 Phase Partitioning

lemma *BIT_inv_initial*: **assumes** $(x::nat) \neq y$
shows *BIT_inv* (*map_pmf* (*Pair* [*x*, *y*]) (*fst BIT* [*x*, *y*])) *x* [*x*, *y*]
using *assms*(*1*) **apply**(*simp add*: *BIT_inv2 BIT_init_def type0_def*)
apply(*simp add*: *map_pmf_def other_def bind_return_pmf bind_assoc_pmf*)
using *bind_commute_pmf* **by** *fast*

lemma D'': **assumes** $qs \in Lxx\ a\ b$
$a \neq b$ $\{a, b\} = \{x, y\}$ *BIT_inv s a* [*x*, *y*]
$set\ qs \subseteq \{a, b\}$
shows T_p*_on_rand*$'$ *BIT s qs* $\leq 175\ /\ 10^2 * real$ (T_p [*a*, *b*] *qs* (*OPT2 qs* [*a*, *b*])) $\wedge$
BIT_inv (*Partial_Cost_Model.config*$'$*_rand BIT s qs*) (*last qs*) [*x*, *y*]
apply(*rule LxxE*[*OF assms*(*1*)])
using *bit_d*[*OF assms*(*2*−*5*)] **apply**(*simp*)
apply(*rule bit_b*$''$[*OF assms*(*2*−*5*)]) **apply**(*simp*)
apply(*rule bit_c*$''$[*OF assms*(*2*−*5*)]) **apply**(*simp*)
using *bit_a*[*OF assms*(*2*−*5*)] **apply**(*simp*)
done

theorem *BIT_175comp_on_2*:
assumes $(x::nat) \neq y$ $set\ \sigma \subseteq \{x,y\}$
shows T_p*_on_rand BIT* [*x*,*y*] σ $\leq 1.75 * real$ (T_p*_opt* [*x*,*y*] σ) + *1.75*
proof (*rule Phase_partitioning_general*[**where** *P*=*BIT_inv*], *goal_cases*)
case *4*
show *BIT_inv* (*map_pmf* (*Pair* [*x*, *y*]) (*fst BIT* [*x*, *y*])) *x* [*x*, *y*]
by (*rule BIT_inv_initial*[*OF assms*(*1*)])
next
case (*5 a b qs s*)
then show *?case* **by**(*rule* D'')
qed (*simp_all add*: *assms*)

end

18 COMB

theory *Comb*
imports *TS BIT_2comp_on2 BIT_pairwise*
begin

18.1 Definition of COMB

type_synonym *CombState* = (*bool list* ∗ *nat list*) + (*nat list*)

definition *COMB_init* :: *nat list* ⇒ (*nat state*, *CombState*) *alg_on_init*
where
COMB_init h init =
Sum_pmf 0.8 (*fst BIT init*) (*fst* (*embed* (*rTS h*)) *init*)

lemma *COMB_init*[*simp*]: *COMB_init h init* =
do {
(*b*::*bool*) ← (*bernoulli_pmf 0.8*);
(*xs*::*bool list*) ← *Prob_Theory.bv* (*length init*);
return_pmf (*if b then Inl* (*xs*, *init*) *else Inr h*)
}
apply(*simp add*: *bind_return_pmf COMB_init_def BIT_init_def rTS_def bind_assoc_pmf*)
unfolding *map_pmf_def Sum_pmf_def*
apply(*simp add*: *if_distrib bind_return_pmf bind_assoc_pmf*)
apply(*rule bind_pmf_cong*)
by(*auto simp add*: *bind_return_pmf bind_assoc_pmf*)

definition *COMB_step* :: (*nat state*, *CombState*, *nat*, *answer*) *alg_on_step*
where
COMB_step s q = (*case snd s of Inl b* ⇒ *map_pmf* (λ((*a*,*b*),*c*). ((*a*,*b*),*Inl c*)) (*BIT_step* (*fst s*, *b*) *q*)
| *Inr b* ⇒ *map_pmf* (λ((*a*,*b*),*c*). ((*a*,*b*),*Inr c*)) (*return_pmf* (*TS_step_d* (*fst s*, *b*) *q*)))

definition *COMB h* = (*COMB_init h*, *COMB_step*)

18.2 Comb 1.6-competitive on 2 elements

abbreviation *noc* == (%*x*. *case x of Inl* (*s*,*is*) ⇒ (*s*,*Inl is*) | *Inr* (*s*,*is*) ⇒ (*s*,*Inr is*))
abbreviation *con* == (%(*s*,*is*). *case is of Inl is* ⇒ *Inl* (*s*,*is*) | *Inr is* ⇒ *Inr* (*s*,*is*))

definition *inv_COMB s x i* == (∃ *Da Db*. *finite* (*set_pmf Da*) ∧ *finite* (*set_pmf Db*) ∧
(*map_pmf con s*) = *Sum_pmf 0.8 Da Db* ∧ *BIT_inv Da x i* ∧ *TS_inv Db x i*)

lemma *noccon*: *noc o con* = *id*

```
  apply(rule ext)
  apply(case_tac x) by(auto simp add: sum.case_eq_if)

lemma connoc: con o noc = id
  apply(rule ext)
  apply(case_tac x) by(auto simp add: sum.case_eq_if)

lemma obligation1': assumes map_pmf con s = Sum_pmf (8 / 10) Da
Db
    shows config'_rand (COMB h) s qs =
    map_pmf noc (Sum_pmf (8 / 10) (config'_rand BIT Da qs)
                  (config'_rand (embed (rTS h)) Db qs))
proof (induct qs rule: rev_induct)
  case Nil
  have s = map_pmf noc (map_pmf con s)
    by(simp add: pmf.map_comp noccon)
also
  from assms have ... = map_pmf noc (Sum_pmf (8 / 10) Da Db)
    by simp
finally
  show ?case by simp
next
  case (snoc q qs)
  show ?case apply(simp)
    apply(subst config'_rand_append)
    apply(subst snoc)
    apply(simp)
      unfolding Sum_pmf_def
      apply(simp add:
          bind_assoc_pmf bind_return_pmf COMB_def COMB_step_def)
      apply(subst config'_rand_append)
      apply(subst config'_rand_append)
      apply(simp only: map_pmf_def[where f=noc])
        apply(simp add: bind_return_pmf bind_assoc_pmf)
        apply(rule bind_pmf_cong)
          apply(simp)
         apply(simp only: set_pmf_bernoulli UNIV_bool)
          apply(auto)
            apply(simp only: map_pmf_def[where f=Inl])
            apply(simp add: bind_return_pmf bind_assoc_pmf)
            apply(rule bind_pmf_cong)
            apply(simp add: bind_return_pmf bind_assoc_pmf )
            apply(simp add:  split_def)
        apply(simp add: bind_return_pmf bind_assoc_pmf map_pmf_def)
```

apply(*simp only*: *map_pmf_def*[**where** *f*=*Inr*])
apply(*simp add*: *bind_return_pmf bind_assoc_pmf*)
apply(*rule bind_pmf_cong*)
apply(*simp add*: *bind_return_pmf bind_assoc_pmf*)
apply(*simp add*: *split_def*)
apply(*simp add*: *bind_return_pmf bind_assoc_pmf map_pmf_def rTS_def*)
done
qed

lemma *obligation1''*:
shows *config_rand* (*COMB h*) *init qs* =
map_pmf noc (*Sum_pmf* (*8* / *10*) (*config_rand BIT init qs*)
(*config_rand* (*embed* (*rTS h*)) *init qs*))
apply(*rule obligation1'*)
apply(*simp add*: *Sum_pmf_def COMB_def map_pmf_def bind_assoc_pmf bind_return_pmf split_def COMB_init_def del*: *COMB_init*)
apply(*rule bind_pmf_cong*)
by(*auto simp add*: *split_def map_pmf_def bind_return_pmf bind_assoc_pmf*)

lemma *obligation1*: **assumes** *map_pmf con s* = *Sum_pmf* (*8* / *10*) *Da Db*
shows *map_pmf con* (*config'_rand* (*COMB* []) *s qs*) =
Sum_pmf (*8* / *10*) (*config'_rand BIT Da qs*)
(*config'_rand* (*embed* (*rTS* [])) *Db qs*)
proof −
from *obligation1'*[*OF assms*] **have** *map_pmf con* (*config'_rand* (*COMB* []) *s qs*)
= *map_pmf con* (*map_pmf noc* (*Sum_pmf* (*8* / *10*) (*config'_rand BIT Da qs*)
(*config'_rand* (*embed* (*rTS* [])) *Db qs*)))
by *simp*
also
have . . . = *Sum_pmf* (*8* / *10*) (*config'_rand BIT Da qs*)
(*config'_rand* (*embed* (*rTS* [])) *Db qs*)
apply(*simp only*: *pmf.map_comp connoc*) **by** *simp*
finally
show *?thesis* **.**
qed

lemma *BIT_config'_fin*: *finite* (*set_pmf s*) ⟹ *finite* (*set_pmf* (*config'_rand BIT s qs*))
apply(*induct qs rule*: *rev_induct*)
apply(*simp*)

by(*simp add*: *config′_rand_append BIT_step_def*)

lemma *TS_config′_fin*: *finite* (*set_pmf s*) ⟹ *finite* (*set_pmf* (*config′_rand* (*embed* (*rTS h*)) *s qs*))
apply(*induct qs rule*: *rev_induct*)
apply(*simp*)
by(*simp add*: *config′_rand_append rTS_def TS_step_d_def*)

lemma *obligation2*: **assumes** *map_pmf con s* = *Sum_pmf* (*8* / *10*) *Da Db*
and *finite* (*set_pmf Da*)
and *finite* (*set_pmf Db*)
shows T_p*_on_rand′* (*COMB* []) *s qs* =
2 / *10* ∗ T_p*_on_rand′* (*embed* (*rTS* [])) *Db qs* +
8 / *10* ∗ T_p*_on_rand′ BIT Da qs*
proof (*induct qs rule*: *rev_induct*)
case (*snoc q qs*)
have *P*: T_p*_on_rand′* (*COMB* []) (*config′_rand* (*COMB* []) *s qs*) [*q*]
= *2* / *10* ∗ T_p*_on_rand′* (*embed* (*rTS* [])) (*config′_rand* (*embed* (*rTS* [])) *Db qs*) [*q*] +
8 / *10* ∗ T_p*_on_rand′ BIT* (*config′_rand BIT Da qs*) [*q*]
apply(*subst obligation1′*[*OF assms*(*1*)])
unfolding *Sum_pmf_def*
apply(*simp*)
apply(*simp only*: *map_pmf_def*[**where** *f*=*noc*])
apply(*simp add*: *bind_assoc_pmf*)
apply(*subst E_bernoulli3*)
apply(*simp*) **apply**(*simp*)
apply(*simp add*: *set_pmf_bernoulli*)
apply(*simp add*: *BIT_step_def COMB_def COMB_step_def split_def*)
apply(*safe*)
using *BIT_config′_fin*[*OF assms*(*2*)] **apply**(*simp*)
using *TS_config′_fin*[*OF assms*(*3*)] **apply**(*simp*)
apply(*simp*)
apply(*simp only*: *map_pmf_def*[**where** *f*=*Inl*])
apply(*simp only*: *map_pmf_def*[**where** *f*=*Inr*])
apply(*simp add*: *bind_return_pmf bind_assoc_pmf COMB_def COMB_step_def*)
apply(*simp add*: *split_def*)
apply(*simp add*: *rTS_def map_pmf_def bind_return_pmf bind_assoc_pmf COMB_def COMB_step_def*)

done

show *?case*
apply(*simp only*: *T_on_rand′_append*)
apply(*subst snoc*)
apply(*subst P*) **by** *algebra*

qed *simp*

lemma *Combination*:
fixes *bit*
assumes $qs \in pattern\ a \neq b\ \{a,\ b\} = \{x,\ y\}\ set\ qs \subseteq \{a,\ b\}$
and *inv_COMB s a* [*x*,*y*]
and *TS*: $\bigwedge s\ h.\ a \neq b \Longrightarrow \{a,\ b\} = \{x,\ y\} \Longrightarrow TS\_inv\ s\ a\ [x,\ y] \Longrightarrow set\ qs \subseteq \{a,\ b\}$
$\Longrightarrow qs \in pattern \Longrightarrow$
TS_inv (*config′_rand* (*embed* (*rTS h*)) *s qs*) (*last qs*) [*x*, *y*]
$\wedge\ T_p\_on\_rand'$ (*embed* (*rTS h*)) *s qs* = *ts*
and *BIT*: $\bigwedge s.\ a \neq b \Longrightarrow \{a,\ b\} = \{x,\ y\} \Longrightarrow BIT\_inv\ s\ a\ [x,\ y] \Longrightarrow set\ qs \subseteq \{a,\ b\}$
$\Longrightarrow qs \in pattern \Longrightarrow$
BIT_inv (*config′_rand BIT s qs*) (*last qs*) [*x*, *y*]
$\wedge\ T_p\_on\_rand'$ *BIT s qs* = *bit*
and *OPT_cost*: $a \neq b \Longrightarrow qs \in pattern \Longrightarrow real\ (T_p\ [a,\ b]\ qs\ (OPT2\ qs\ [a,\ b])) = opt$
and *absch*: $qs \in pattern \Longrightarrow 0.2 * ts + 0.8 * bit \leq 1.6 * opt$
shows $T_p\_on\_rand'\ (COMB\ [])\ s\ qs \leq 16\ /\ 10 * real\ (T_p\ [a,\ b]\ qs\ (OPT2\ qs\ [a,\ b])) \wedge$
inv_COMB (*Partial_Cost_Model.config′_rand* (*COMB* []) *s qs*) (*last qs*) [*x*, *y*]
proof −
let *?D* = *map_pmf con s*
from *assms*(*5*) **obtain** *Da Db* **where** *Daf*: *finite* (*set_pmf Da*)
and *Dbf*: *finite* (*set_pmf Db*)
and *D*: *?D* = *Sum_pmf 0.8 Da Db*
and *B*: *BIT_inv Da a* [*x*,*y*] **and** *T*: *TS_inv Db a* [*x*,*y*]
unfolding *inv_COMB_def* **by** *auto*

let *?Da′* = *config′_rand BIT Da qs*
from *BIT*[*OF assms*(*2*,*3*) *B assms*(*4*,*1*)]
have *B′*: *BIT_inv ?Da′* (*last qs*) [*x*, *y*]
and *B_cost*: $T_p\_on\_rand'$ *BIT Da qs* = *bit* **by** *auto*

let *?Db′* = *config′_rand* (*embed* (*rTS* [])) *Db qs*

```
  from TS[OF assms(2,3) T assms(4,1)]
    have T': TS_inv ?Db' (last qs) [x, y]
    and T_cost: T_p_on_rand' (embed (rTS [])) Db qs = ts by auto

  have T_p_on_rand' (COMB []) s qs
       = 0.2 * T_p_on_rand' (embed (rTS [])) Db qs
          + 0.8 * T_p_on_rand' BIT Da qs
        using D apply(rule obligation2) apply(fact Daf) apply(fact Dbf)
done
also
  have ... ≤ 1.6 * opt
    by (simp only: B_cost T_cost absch[OF assms(1)])
also
 have ... = 1.6 * T_p [a, b] qs (OPT2 qs [a, b]) by (simp add: OPT_cost[OF
assms(2,1)])
finally
  have Comb_cost: T_p_on_rand' (COMB []) s qs ≤ 1.6 * T_p [a, b] qs
(OPT2 qs [a, b]) .

  have Comb_inv: inv_COMB (config'_rand (COMB []) s qs) (last qs) [x,
y]
      unfolding inv_COMB_def
      apply(rule exI[where x=?Da'])
      apply(rule exI[where x=?Db'])
        apply(safe)
          apply(rule BIT_config'_fin[OF Daf])
          apply(rule TS_config'_fin[OF Dbf])
          apply(rule obligation1)
          apply(fact D)
          apply(fact B')
          apply(fact T') done

  from Comb_cost Comb_inv show ?thesis by simp
qed

theorem COMB_OPT2': (x::nat) ≠ y ⟹ set σ ⊆ {x,y}
    ⟹ T_p_on_rand (COMB []) [x,y] σ  ≤ 1.6 * real (T_p_opt [x,y] σ) +
1.6
proof (rule Phase_partitioning_general[where P=inv_COMB], goal_cases)
  case 4
  let ?initBIT =(map_pmf (Pair [x, y]) (fst BIT [x, y]))
  let ?initTS =(map_pmf (Pair [x, y]) (fst (embed (rTS [])) [x, y]))
  show  inv_COMB (map_pmf (Pair [x, y]) (fst (COMB []) [x, y])) x [x,
```

```
y]
    unfolding inv_COMB_def
    apply(rule exI[where x=?initBIT])
    apply(rule exI[where x=?initTS])
      apply(simp only: BIT_inv_initial[OF 4(1)] )
        apply(simp add: map_pmf_def bind_return_pmf bind_assoc_pmf
COMB_def)
      apply(simp add: Sum_pmf_def)
     apply(safe)
         apply(simp add: BIT_init_def)
        apply(rule bind_pmf_cong)
          apply(simp)
             apply(simp add: bind_return_pmf bind_assoc_pmf rTS_def
map_pmf_def BIT_init_def)
       apply(simp add: TS_inv_def rTS_def)
      done
next
  case (5 a b qs s)
  from 5(3)
  show ?case
    proof (rule LxxE, goal_cases)
      case 4
      then show ?thesis apply(rule Combination)
        apply(fact)+
        using TS_a'' apply(simp)
        apply(fact bit_a'')
        apply(fact OPT2_A')
        apply(simp)
      done
    next
      case 1
      then show ?case
        apply(rule Combination)
        apply(fact)+
        apply(fact TS_d'')
        apply(fact bit_d')
        by auto
    next
      case 2
       then have qs ∈ lang (seq [Atom b, Atom a, Star (Times (Atom b)
(Atom a)), Atom b, Atom b])
             ∨ qs ∈ lang (seq [Atom a, Atom b, Atom a, Star (Times (Atom
b) (Atom a)), Atom b, Atom b]) by auto
      then show ?case
```

```
      apply(rule disjE)
        apply(erule Combination)
          apply(fact)+
          apply(fact TS_b1'')
          apply(fact bit_b''1)
          apply(fact OPT2_B1)
          apply(simp add: ring_distribs)
       apply(erule Combination)
         apply(fact)+
         apply(fact TS_b2'')
         apply(fact bit_b''2)
         apply(fact OPT2_B2)
         apply(simp add: ring_distribs)
      done
  next
    case 3
    then have len: length qs ≥ 2 by(auto simp add: conc_def)
     have len2: qs ∈ lang (seq [Atom a, Atom b, Atom a, Star (Times
(Atom b) (Atom a)), Atom a])
             ⟹ length qs ≥ 3 by (auto simp add: conc_def)
    from 3 have qs ∈ lang (seq [Atom b, Atom a, Star (Times (Atom b)
(Atom a)), Atom a])
           ∨ qs ∈ lang (seq [Atom a, Atom b, Atom a, Star (Times (Atom
b) (Atom a)), Atom a]) by auto
    then show ?case
      apply(rule disjE)
        apply(erule Combination)
          apply(fact)+
          apply(fact TS_c1'')
          apply(fact bit_c''1)
          apply(fact OPT2_C1)
          using len apply(simp add: ring_distribs)
       apply(erule Combination)
         apply(fact)+
         apply(fact TS_c2'')
         apply(fact bit_c''2)
         apply(fact OPT2_C2)
         using len2 apply(simp add: ring_distribs conc_def)
      done
  qed
qed (simp_all)
```

18.3 COMB pairwise

lemma *config_rand_COMB*: *config_rand* (*COMB h*) *init qs* = *do* {
(*b*::*bool*) ← (*bernoulli_pmf 0.8*);
(*b1*,*b2*) ← (*config_rand BIT init qs*);
(*t1*,*t2*) ← (*config_rand* (*embed* (*rTS h*)) *init qs*);
return_pmf (*if b then* (*b1*, *Inl b2*) *else* (*t1*, *Inr t2*))
} (**is** *?LHS* = *?RHS*)
proof (*induct qs rule*: *rev_induct*)
case *Nil*
show *?case*
apply(*simp add*: *BIT_init_def COMB_def rTS_def map_pmf_def bind_return_pmf bind_assoc_pmf*)
apply(*rule bind_pmf_cong*)
apply(*simp*)
apply(*simp only*: *set_pmf_bernoulli*)
apply(*case_tac x*)
by(*simp_all*)
next
case (*snoc q qs*)
show *?case* **apply**(*simp add*: *take_Suc_conv_app_nth*)
apply(*subst config'_rand_append*)
apply(*subst snoc*)
apply(*simp*)
apply(*simp add*: *bind_return_pmf bind_assoc_pmf split_def config'_rand_append*)
apply(*rule bind_pmf_cong*)
apply(*simp*)
apply(*simp only*: *set_pmf_bernoulli*)
apply(*case_tac x*)
by(*simp_all add*: *COMB_def COMB_step_def rTS_def map_pmf_def split_def bind_return_pmf bind_assoc_pmf*)
qed

lemma *COMB_no_paid*: ∀((*free*, *paid*), *t*)∈*set_pmf* (*snd* (*COMB* []) (*s*, *is*) *q*). *paid* = []
apply(*simp add*: *COMB_def COMB_step_def split_def BIT_step_def TS_step_d_def*)
apply(*case_tac is*)
by(*simp_all add*: *BIT_step_def TS_step_d_def*)

lemma *COMB_pairwise*: *pairwise* (*COMB* [])
proof(*rule pairwise_property_lemma*, *goal_cases*)

case (*1 init qs x y*)
then have *qsininit*: *set qs* ⊆ *set init* **by** *simp*

show *Pbefore_in x y* (*COMB* []) *qs init*
= *Pbefore_in x y* (*COMB* []) (*Lxy qs* {*x*, *y*}) (*Lxy init* {*x*, *y*})
unfolding *Pbefore_in_def*
apply(*subst config_rand_COMB*)
apply(*subst config_rand_COMB*)
apply(*simp only*: *map_pmf_def bind_assoc_pmf*)
apply(*rule bind_pmf_cong*)
apply(*simp*)
apply(*simp only*: *set_pmf_bernoulli*)
apply(*case_tac xa*)
apply(*simp add*: *split_def*)
using *BIT_pairwise'*[*OF qsininit 1*(*3*,*4*,*1*), *unfolded Pbefore_in_def map_pmf_def*]
apply(*simp add*: *bind_return_pmf bind_assoc_pmf*)
apply(*simp add*: *split_def*)
using *TS_pairwise'*[*OF 1*(*2*,*3*,*4*,*1*), *unfolded Pbefore_in_def map_pmf_def*]
by(*simp add*: *bind_return_pmf bind_assoc_pmf*)
next
case (*2 xa r*)
show *?case*
using *COMB_no_paid*
by (*metis* (*mono_tags*) *case_prod_unfold surj_pair*)
qed

18.4 COMB 1.6-competitive

lemma *finite_config_TS*: *finite* (*set_pmf* (*config''* (*embed* (*rTS h*)) *qs init n*)) (**is** *finite ?D*)
apply(*subst config_embed*)
by(*simp*)

lemma *COMB_has_finite_config_set*: **assumes** [*simp*]: *distinct init*
and *set qs* ⊆ *set init*
shows *finite* (*set_pmf* (*config_rand* (*COMB h*) *init qs*))
proof −
from *finite_config_TS*[**where** *n*=*length qs* **and** *qs*=*qs*]
finite_config_BIT[*OF assms*(*1*)]
show *?thesis*
apply(*subst obligation1''*)
by(*simp add*: *Sum_pmf_def*)

qed

theorem *COMB_competitive*: ∀ *s0*∈{*x*::*nat list*. *distinct x* ∧ *x*≠[]}.
∃ *b*≥*0*. ∀ *qs*∈{*x*. *set x* ⊆ *set s0*}.
T_p_*on_rand* (*COMB* []) *s0 qs* ≤ ((*8*::*nat*)/(*5*::*nat*)) ∗ T_p_*opt s0 qs* + *b*
proof(*rule factoringlemma_withconstant*, *goal_cases*)
case *5*
show *?case*
proof (*safe*, *goal_cases*)
case (*1 init*)
note *out*=*this*
show *?case*
apply(*rule exI*[**where** *x*=*2*])
apply(*simp*)
proof (*safe*, *goal_cases*)
case (*1 qs a b*)
then have *a*: *a*≠*b* **by** *simp*
have *twist*: {*a*,*b*}={*b*, *a*} **by** *auto*
have *b1*: *set* (*Lxy qs* {*a*, *b*}) ⊆ {*a*, *b*} **unfolding** *Lxy_def* **by** *auto*
with *this*[*unfolded twist*] **have** *b2*: *set* (*Lxy qs* {*b*, *a*}) ⊆ {*b*, *a*} **by**(*auto*)

have *set* (*Lxy init* {*a*, *b*}) = {*a*,*b*} ∩ (*set init*) **apply**(*induct init*)
unfolding *Lxy_def* **by**(*auto*)
with *1* **have** *A*: *set* (*Lxy init* {*a*, *b*}) = {*a*,*b*} **by** *auto*
have *finite* {*a*,*b*} **by** *auto*
from *out* **have** *B*: *distinct* (*Lxy init* {*a*, *b*}) **unfolding** *Lxy_def* **by** *auto*
have *C*: *length* (*Lxy init* {*a*, *b*}) = *2*
using *distinct_card*[*OF B*, *unfolded A*] **using** *a* **by** *auto*

have {*xs*. *set xs* = {*a*,*b*} ∧ *distinct xs* ∧ *length xs* =(*2*::*nat*)}
= { [*a*,*b*], [*b*,*a*] }
apply(*auto simp*: *a a*[*symmetric*])
proof (*goal_cases*)
case (*1 xs*)
from *1*(*4*) **obtain** *x xs'* **where** *r*:*xs*=*x*#*xs'* **by** (*metis Suc_length_conv add_2_eq_Suc' append_Nil length_append*)
with *1*(*4*) **have** *length xs'* = *1* **by** *auto*
then obtain *y* **where** *s*: [*y*] = *xs'* **by** (*metis One_nat_def length_0_conv length_Suc_conv*)
from *r s* **have** *t*: [*x*,*y*] = *xs* **by** *auto*

```
                moreover from t 1(1) have x=b using doubleton_eq_iff
1(2) by fastforce
                moreover from t 1(1) have y=a using doubleton_eq_iff
1(2) by fastforce
                 ultimately show ?case by auto
               qed

          with A B C have pos: (Lxy init {a, b}) = [a,b]
               ∨ (Lxy init {a, b}) = [b,a] by auto

          show ?case
            apply(cases (Lxy init {a, b}) = [a,b])
              apply(simp) using COMB_OPT2'[OF a b1] a apply(simp)
              using pos apply(simp) unfolding twist
            using COMB_OPT2'[OF a[symmetric] b2] by simp
        qed
    qed
next
  case 4  then show ?case using COMB_pairwise by simp
next
  case 7 then show ?case apply(subst COMB_has_finite_config_set[OF
7(1)])
        using set_take_subset apply fast by simp
qed (simp_all add: COMB_no_paid)



theorem COMB_competitive_nice: compet_rand (COMB []) ((8::nat)/(5::nat))
{x::nat list. distinct x ∧ x≠[]}
  unfolding compet_rand_def static_def using COMB_competitive by
simp



end
```

References

[BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[HN16] Maximilian P.L. Haslbeck and Tobias Nipkow. Verified analysis of list update algorithms. http://www.in.tum.de/~nipkow/pubs/list_update.pdf, 2016.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.