Basic combinatorics in Isabelle/HOL (and the Archive of Formal Proofs)

March 13, 2025

Contents

1 Transposition function

theory *Transposition*
imports *Main*
begin

definition *transpose* :: ‹$'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$›
  where ‹*transpose a b c* = (*if c* = *a then b else if c* = *b then a else c*)›

lemma *transpose_apply_first* [*simp*]:
  ‹*transpose a b a* = *b*›
  by (*simp add*: *transpose_def*)

lemma *transpose_apply_second* [*simp*]:
  ‹*transpose a b b* = *a*›
  by (*simp add*: *transpose_def*)

lemma *transpose_apply_other* [*simp*]:
  ‹*transpose a b c* = *c*› **if** ‹$c \neq a$› ‹$c \neq b$›
  using *that* **by** (*simp add*: *transpose_def*)

lemma *transpose_same* [*simp*]:
  ‹*transpose a a* = *id*›
  by (*simp add*: *fun_eq_iff transpose_def*)

lemma *transpose_eq_iff*:
  ‹*transpose a b c* = *d* $\longleftrightarrow$ ($c \neq a \wedge c \neq b \wedge d = c$) $\vee$ ($c = a \wedge d = b$) $\vee$ ($c = b$
$\wedge$ $d = a$)›
  by (*auto simp add*: *transpose_def*)

lemma *transpose_eq_imp_eq*:
  ‹*c* = *d*› **if** ‹*transpose a b c* = *transpose a b d*›
  using *that* **by** (*auto simp add*: *transpose_eq_iff*)

lemma *transpose_commute* [*ac_simps*]:
  ‹*transpose b a* = *transpose a b*›
  by (*auto simp add*: *fun_eq_iff transpose_eq_iff*)

lemma *transpose_involutory* [*simp*]:
  ‹*transpose a b* (*transpose a b c*) = *c*›
  by (*auto simp add*: *transpose_eq_iff*)

lemma *transpose_comp_involutory* [*simp*]:
  ‹*transpose a b* ∘ *transpose a b* = *id*›
  by (*rule ext*) *simp*

lemma *transpose_triple*:
  ‹*transpose a b* (*transpose b c* (*transpose a b d*)) = *transpose a c d*›
  if ‹$a \neq c$› **and** ‹$b \neq c$›
  using *that* **by** (*simp add*: *transpose_def*)

lemma *transpose_comp_triple*:
  ‹*transpose a b* ∘ *transpose b c* ∘ *transpose a b* = *transpose a c*›
  if ‹$a \neq c$› **and** ‹$b \neq c$›
  using *that* **by** (*simp add*: *fun_eq_iff transpose_triple*)

lemma *transpose_image_eq* [*simp*]:
‹*transpose a b* ‘ *A* = *A*› **if** ‹*a* ∈ *A* ⟷ *b* ∈ *A*›
using *that* **by** (*auto simp add*: *transpose_def* [*abs_def*])

lemma *inj_on_transpose* [*simp*]:
‹*inj_on* (*transpose a b*) *A*›
by *rule* (*drule transpose_eq_imp_eq*)

lemma *inj_transpose*:
‹*inj* (*transpose a b*)›
by (*fact inj_on_transpose*)

lemma *surj_transpose*:
‹*surj* (*transpose a b*)›
by *simp*

lemma *bij_betw_transpose_iff* [*simp*]:
‹*bij_betw* (*transpose a b*) *A A*› **if** ‹*a* ∈ *A* ⟷ *b* ∈ *A*›
using *that* **by** (*auto simp*: *bij_betw_def*)

lemma *bij_transpose* [*simp*]:
‹*bij* (*transpose a b*)›
by (*rule bij_betw_transpose_iff*) *simp*

lemma *bijection_transpose*:
‹*bijection* (*transpose a b*)›
by *standard* (*fact bij_transpose*)

lemma *inv_transpose_eq* [*simp*]:
‹*inv* (*transpose a b*) = *transpose a b*›
by (*rule inv_unique_comp*) *simp_all*

lemma *transpose_apply_commute*:
‹*transpose a b* (*f c*) = *f* (*transpose* (*inv f a*) (*inv f b*) *c*)›
if ‹*bij f*›
proof –
from *that* **have** ‹*surj f*›
by (*rule bij_is_surj*)
with *that* **show** *?thesis*
by (*simp add*: *transpose_def bij_inv_eq_iff surj_f_inv_f*)
qed

lemma *transpose_comp_eq*:
‹*transpose a b* ∘ *f* = *f* ∘ *transpose* (*inv f a*) (*inv f b*)›
if ‹*bij f*›
using *that* **by** (*simp add*: *fun_eq_iff transpose_apply_commute*)

lemma *in_transpose_image_iff*:

$\langle x \in transpose\ a\ b\ `\ S \longleftrightarrow transpose\ a\ b\ x \in S\rangle$
by (*auto intro*!: *image_eqI*)

Legacy input alias

setup ‹*Context.theory_map* (*Name_Space.map_naming* (*Name_Space.qualified_path true* ***binding*** ‹*Fun*›))›

abbreviation (*input*) *swap* :: $\langle 'a \Rightarrow 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b\rangle$
where $\langle swap\ a\ b\ f \equiv f \circ transpose\ a\ b\rangle$

lemma *swap_def*:
$\langle Fun.swap\ a\ b\ f = f\ (a := f\ b,\ b := f\ a)\rangle$
by (*simp add*: *fun_eq_iff*)

setup ‹*Context.theory_map* (*Name_Space.map_naming* (*Name_Space.parent_path*))›

lemma *swap_apply*:
$Fun.swap\ a\ b\ f\ a = f\ b$
$Fun.swap\ a\ b\ f\ b = f\ a$
$c \neq a \Longrightarrow c \neq b \Longrightarrow Fun.swap\ a\ b\ f\ c = f\ c$
by *simp_all*

lemma *swap_self*: $Fun.swap\ a\ a\ f = f$
by *simp*

lemma *swap_commute*: $Fun.swap\ a\ b\ f = Fun.swap\ b\ a\ f$
by (*simp add*: *ac_simps*)

lemma *swap_nilpotent*: $Fun.swap\ a\ b\ (Fun.swap\ a\ b\ f) = f$
by (*simp add*: *comp_assoc*)

lemma *swap_comp_involutory*: $Fun.swap\ a\ b \circ Fun.swap\ a\ b = id$
by (*simp add*: *fun_eq_iff*)

lemma *swap_triple*:
assumes $a \neq c$ **and** $b \neq c$
shows $Fun.swap\ a\ b\ (Fun.swap\ b\ c\ (Fun.swap\ a\ b\ f)) = Fun.swap\ a\ c\ f$
using *assms transpose_comp_triple* [*of a c b*]
by (*simp add*: *comp_assoc*)

lemma *comp_swap*: $f \circ Fun.swap\ a\ b\ g = Fun.swap\ a\ b\ (f \circ g)$
by (*simp add*: *comp_assoc*)

lemma *swap_image_eq*:
assumes $a \in A\ b \in A$
shows $Fun.swap\ a\ b\ f\ `\ A = f\ `\ A$
using *assms* **by** (*metis image_comp transpose_image_eq*)

lemma *inj_on_imp_inj_on_swap*: $inj\_on\ f\ A \Longrightarrow a \in A \Longrightarrow b \in A \Longrightarrow inj\_on$

$(Fun.swap\ a\ b\ f)\ A$
  by $(simp\ add: comp\_inj\_on)$

lemma *inj_on_swap_iff*:
  assumes A: $a \in A$ $b \in A$
  shows $inj\_on\ (Fun.swap\ a\ b\ f)\ A \longleftrightarrow inj\_on\ f\ A$
  using *assms* **by** $(metis\ inj\_on\_imageI\ inj\_on\_imp\_inj\_on\_swap\ transpose\_image\_eq)$

lemma *surj_imp_surj_swap*: $surj\ f \Longrightarrow surj\ (Fun.swap\ a\ b\ f)$
  by $(meson\ comp\_surj\ surj\_transpose)$

lemma *surj_swap_iff*: $surj\ (Fun.swap\ a\ b\ f) \longleftrightarrow surj\ f$
  by $(metis\ fun.set\_map\ surj\_transpose)$

lemma *bij_betw_swap_iff*: $x \in A \Longrightarrow y \in A \Longrightarrow bij\_betw\ (Fun.swap\ x\ y\ f)\ A\ B \longleftrightarrow bij\_betw\ f\ A\ B$
  by $(meson\ bij\_betw\_comp\_iff\ bij\_betw\_transpose\_iff)$

lemma *bij_swap_iff*: $bij\ (Fun.swap\ a\ b\ f) \longleftrightarrow bij\ f$
  by $(simp\ add: bij\_betw\_swap\_iff)$

lemma *swap_image*:
  ‹$Fun.swap\ i\ j\ f$ ‘ $A = f$ ‘ $(A - \{i, j\}$
    $\cup\ (if\ i \in A\ then\ \{j\}\ else\ \{\}) \cup (if\ j \in A\ then\ \{i\}\ else\ \{\}))$›
  by $(auto\ simp\ add: Fun.swap\_def)$

lemma *inv_swap_id*: $inv\ (Fun.swap\ a\ b\ id) = Fun.swap\ a\ b\ id$
  by *simp*

lemma *bij_swap_comp*:
  assumes $bij\ p$
  shows $Fun.swap\ a\ b\ id \circ p = Fun.swap\ (inv\ p\ a)\ (inv\ p\ b)\ p$
  using *assms* **by** $(simp\ add: transpose\_comp\_eq)$

lemma *swap_id_eq*: $Fun.swap\ a\ b\ id\ x = (if\ x = a\ then\ b\ else\ if\ x = b\ then\ a\ else\ x)$
  by $(simp\ add: Fun.swap\_def)$

lemma *swap_unfold*:
  ‹$Fun.swap\ a\ b\ p = p \circ Fun.swap\ a\ b\ id$›
  by *simp*

lemma *swap_id_idempotent*: $Fun.swap\ a\ b\ id \circ Fun.swap\ a\ b\ id = id$
  by *simp*

lemma *bij_swap_compose_bij*:
  ‹$bij\ (Fun.swap\ a\ b\ id \circ p)$› **if** ‹$bij\ p$›
  using *that* **by** $(rule\ bij\_comp)$ *simp*

end

2 Stirling numbers of first and second kind

```
theory Stirling
imports Main
begin
```

2.1 Stirling numbers of the second kind

```
fun Stirling :: nat ⇒ nat ⇒ nat
  where
    Stirling 0 0 = 1
  | Stirling 0 (Suc k) = 0
  | Stirling (Suc n) 0 = 0
  | Stirling (Suc n) (Suc k) = Suc k * Stirling n (Suc k) + Stirling n k

lemma Stirling_1 [simp]: Stirling (Suc n) (Suc 0) = 1
  by (induct n) simp_all

lemma Stirling_less [simp]: n < k ⟹ Stirling n k = 0
  by (induct n k rule: Stirling.induct) simp_all

lemma Stirling_same [simp]: Stirling n n = 1
  by (induct n) simp_all

lemma Stirling_2_2: Stirling (Suc (Suc n)) (Suc (Suc 0)) = 2 ^ Suc n − 1
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have Stirling (Suc (Suc (Suc n))) (Suc (Suc 0)) =
      2 * Stirling (Suc (Suc n)) (Suc (Suc 0)) + Stirling (Suc (Suc n)) (Suc 0)
    by simp
  also have ... = 2 * (2 ^ Suc n − 1) + 1
    by (simp only: Suc Stirling_1)
  also have ... = 2 ^ Suc (Suc n) − 1
  proof −
    have (2::nat) ^ Suc n − 1 > 0
      by (induct n) simp_all
    then have 2 * ((2::nat) ^ Suc n − 1) > 0
      by simp
    then have 2 ≤ 2 * ((2::nat) ^ Suc n)
      by simp
    with add_diff_assoc2 [of 2 2 * 2 ^ Suc n 1]
    have 2 * 2 ^ Suc n − 2 + (1::nat) = 2 * 2 ^ Suc n + 1 − 2 .
    then show ?thesis
      by (simp add: nat_distrib)
```

```
  qed
  finally show ?case by simp
qed

lemma Stirling_2: Stirling (Suc n) (Suc (Suc 0)) = 2 ^ n - 1
  using Stirling_2_2 by (cases n) simp_all
```

2.2 Stirling numbers of the first kind

```
fun stirling :: nat ⇒ nat ⇒ nat
  where
    stirling 0 0 = 1
  | stirling 0 (Suc k) = 0
  | stirling (Suc n) 0 = 0
  | stirling (Suc n) (Suc k) = n * stirling n (Suc k) + stirling n k

lemma stirling_0 [simp]: n > 0 ⟹ stirling n 0 = 0
  by (cases n) simp_all

lemma stirling_less [simp]: n < k ⟹ stirling n k = 0
  by (induct n k rule: stirling.induct) simp_all

lemma stirling_same [simp]: stirling n n = 1
  by (induct n) simp_all

lemma stirling_Suc_n_1: stirling (Suc n) (Suc 0) = fact n
  by (induct n) auto

lemma stirling_Suc_n_n: stirling (Suc n) n = Suc n choose 2
  by (induct n) (auto simp add: numerals(2))

lemma stirling_Suc_n_2:
  assumes n ≥ Suc 0
  shows stirling (Suc n) 2 = (∑ k=1..n. fact n div k)
  using assms
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  show ?case
  proof (cases n)
    case 0
    then show ?thesis
      by (simp add: numerals(2))
  next
    case Suc
    then have geq1: Suc 0 ≤ n
      by simp
```

have *stirling* (*Suc* (*Suc n*)) *2* = *Suc n* ∗ *stirling* (*Suc n*) *2* + *stirling* (*Suc n*) (*Suc 0*)
by (*simp only*: *stirling.simps*(*4*)[*of Suc n*] *numerals*(*2*))
also have . . . = *Suc n* ∗ (∑ *k*=*1*..*n*. *fact n div k*) + *fact n*
using *Suc.hyps*[*OF geq1*]
by (*simp only*: *stirling_Suc_n_1 of_nat_fact of_nat_add of_nat_mult*)
also have . . . = *Suc n* ∗ (∑ *k*=*1*..*n*. *fact n div k*) + *Suc n* ∗ *fact n div Suc n*
by (*metis nat.distinct*(*1*) *nonzero_mult_div_cancel_left*)
also have . . . = (∑ *k*=*1*..*n*. *fact* (*Suc n*) *div k*) + *fact* (*Suc n*) *div Suc n*
by (*simp add*: *sum_distrib_left div_mult_swap dvd_fact*)
also have . . . = (∑ *k*=*1*..*Suc n*. *fact* (*Suc n*) *div k*)
by *simp*
finally show *?thesis* **.**
qed
qed

lemma *of_nat_stirling_Suc_n_2*:
assumes *n* ≥ *Suc 0*
shows (*of_nat* (*stirling* (*Suc n*) *2*)::*'a*::*field_char_0*) = *fact n* ∗ (∑ *k*=*1*..*n*. (*1* / *of_nat k*))
using *assms*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
show *?case*
proof (*cases n*)
case *0*
then show *?thesis*
by (*auto simp add*: *numerals*(*2*))
next
case *Suc*
then have *geq1*: *Suc 0* ≤ *n*
by *simp*
have (*of_nat* (*stirling* (*Suc* (*Suc n*)) *2*)::*'a*) =
of_nat (*Suc n* ∗ *stirling* (*Suc n*) *2* + *stirling* (*Suc n*) (*Suc 0*))
by (*simp only*: *stirling.simps*(*4*)[*of Suc n*] *numerals*(*2*))
also have . . . = *of_nat* (*Suc n*) ∗ (*fact n* ∗ (∑ *k* = *1*..*n*. *1* / *of_nat k*)) + *fact n*
using *Suc.hyps*[*OF geq1*]
by (*simp only*: *stirling_Suc_n_1 of_nat_fact of_nat_add of_nat_mult*)
also have . . . = *fact* (*Suc n*) ∗ (∑ *k* = *1*..*n*. *1* / *of_nat k*) + *fact* (*Suc n*) ∗ (*1* / *of_nat* (*Suc n*))
using *of_nat_neq_0* **by** *auto*
also have . . . = *fact* (*Suc n*) ∗ (∑ *k* = *1*..*Suc n*. *1* / *of_nat k*)
by (*simp add*: *distrib_left*)
finally show *?thesis* **.**
qed

```
qed

lemma sum_stirling: (∑ k≤n. stirling n k) = fact n
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have (∑ k≤Suc n. stirling (Suc n) k) = stirling (Suc n) 0 + (∑ k≤n. stirling
(Suc n) (Suc k))
    by (simp only: sum.atMost_Suc_shift)
  also have . . . = (∑ k≤n. stirling (Suc n) (Suc k))
    by simp
  also have . . . = (∑ k≤n. n ∗ stirling n (Suc k) + stirling n k)
    by simp
  also have . . . = n ∗ (∑ k≤n. stirling n (Suc k)) + (∑ k≤n. stirling n k)
    by (simp add: sum.distrib sum_distrib_left)
  also have . . . = n ∗ fact n + fact n
  proof −
    have n ∗ (∑ k≤n. stirling n (Suc k)) = n ∗ ((∑ k≤Suc n. stirling n k) −
stirling n 0)
      by (metis add_diff_cancel_left' sum.atMost_Suc_shift)
    also have . . . = n ∗ (∑ k≤n. stirling n k)
      by (cases n) simp_all
    also have . . . = n ∗ fact n
      using Suc.hyps by simp
    finally have n ∗ (∑ k≤n. stirling n (Suc k)) = n ∗ fact n .
    moreover have (∑ k≤n. stirling n k) = fact n
      using Suc.hyps .
    ultimately show ?thesis by simp
  qed
  also have . . . = fact (Suc n) by simp
  finally show ?case .
qed

lemma stirling_pochhammer:
  (∑ k≤n. of_nat (stirling n k) ∗ x ^ k) = (pochhammer x n :: 'a::comm_semiring_1)
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have of_nat (n ∗ stirling n 0) = (0 :: 'a) by (cases n) simp_all
  then have (∑ k≤Suc n. of_nat (stirling (Suc n) k) ∗ x ^ k) =
      (of_nat (n ∗ stirling n 0) ∗ x ^ 0 +
      (∑ i≤n. of_nat (n ∗ stirling n (Suc i)) ∗ (x ^ Suc i)) +
      (∑ i≤n. of_nat (stirling n i) ∗ (x ^ Suc i))
    by (subst sum.atMost_Suc_shift) (simp add: sum.distrib ring_distribs)
  also have . . . = pochhammer x (Suc n)
```

    by (*subst sum.atMost_Suc_shift* [*symmetric*])
     (*simp add*: *algebra_simps sum.distrib sum_distrib_left pochhammer_Suc flip*: *Suc*)
  finally show *?case* **.**
qed

A row of the Stirling number triangle

definition *stirling_row* :: *nat* $\Rightarrow$ *nat list*
  where *stirling_row n* = [*stirling n k*. *k* $\leftarrow$ [*0*..<*Suc n*]]

lemma *nth_stirling_row*: $k \le n \Longrightarrow$ *stirling_row n* ! *k* = *stirling n k*
  by (*simp add*: *stirling_row_def del*: *upt_Suc*)

lemma *length_stirling_row* [*simp*]: *length* (*stirling_row n*) = *Suc n*
  by (*simp add*: *stirling_row_def*)

lemma *stirling_row_nonempty* [*simp*]: *stirling_row n* $\neq$ []
  using *length_stirling_row*[*of n*] **by** (*auto simp del*: *length_stirling_row*)

2.2.1 Efficient code

Naively using the defining equations of the Stirling numbers of the first kind to compute them leads to exponential run time due to repeated computations. We can use memoisation to compute them row by row without repeating computations, at the cost of computing a few unneeded values.

As a bonus, this is very efficient for applications where an entire row of Stirling numbers is needed.

definition *zip_with_prev* :: (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *'b*) $\Rightarrow$ *'a* $\Rightarrow$ *'a list* $\Rightarrow$ *'b list*
  where *zip_with_prev f x xs* = *map2 f* (*x* # *xs*) *xs*

lemma *zip_with_prev_altdef*:
  zip_with_prev f x xs =
    (*if xs* = [] *then* [] *else f x* (*hd xs*) # [*f* (*xs*!*i*) (*xs*!(*i*+*1*)). *i* $\leftarrow$ [*0*..<*length xs* − *1*]])
proof (*cases xs*)
  case *Nil*
  then show *?thesis*
    by (*simp add*: *zip_with_prev_def*)
next
  case (*Cons y ys*)
  then have *zip_with_prev f x xs* = *f x* (*hd xs*) # *zip_with_prev f y ys*
    by (*simp add*: *zip_with_prev_def*)
  also have *zip_with_prev f y ys* = *map* (λ*i*. *f* (*xs* ! *i*) (*xs* ! (*i* + *1*))) [*0*..<*length xs* − *1*]
    unfolding *Cons*
    by (*induct ys arbitrary*: *y*)
      (*simp_all add*: *zip_with_prev_def upt_conv_Cons flip*: *map_Suc_upt del*: *upt_Suc*)

```
  finally show ?thesis
    using Cons by simp
qed


primrec stirling_row_aux
  where
    stirling_row_aux n y [] = [1]
  | stirling_row_aux n y (x#xs) = (y + n * x) # stirling_row_aux n x xs

lemma stirling_row_aux_correct:
  stirling_row_aux n y xs = zip_with_prev (λa b. a + n * b) y xs @ [1]
  by (induct xs arbitrary: y) (simp_all add: zip_with_prev_def)

lemma stirling_row_code [code]:
  stirling_row 0 = [1]
  stirling_row (Suc n) = stirling_row_aux n 0 (stirling_row n)
proof goal_cases
  case 1
  show ?case by (simp add: stirling_row_def)
next
  case 2
  have stirling_row (Suc n) =
    0 # [stirling_row n ! i + stirling_row n ! (i+1) * n. i ← [0..<n]] @ [1]
  proof (rule nth_equalityI, goal_cases length nth)
    case (nth i)
    from nth have i ≤ Suc n
      by simp
    then consider i = 0 ∨ i = Suc n | i > 0 i ≤ n
      by linarith
    then show ?case
    proof cases
      case 1
      then show ?thesis
        by (auto simp: nth_stirling_row nth_append)
    next
      case 2
      then show ?thesis
        by (cases i) (simp_all add: nth_append nth_stirling_row)
    qed
  next
    case length
    then show ?case by simp
  qed
  also have 0 # [stirling_row n ! i + stirling_row n ! (i+1) * n. i ← [0..<n]] @
[1] =
      zip_with_prev (λa b. a + n * b) 0 (stirling_row n) @ [1]
    by (cases n) (auto simp add: zip_with_prev_altdef stirling_row_def hd_map
simp del: upt_Suc)
```

```
  also have ... = stirling_row_aux n 0 (stirling_row n)
    by (simp add: stirling_row_aux_correct)
  finally show ?case .
qed

lemma stirling_code [code]:
  stirling n k =
    (if k = 0 then (if n = 0 then 1 else 0)
     else if k > n then 0
     else if k = n then 1
     else stirling_row n ! k)
  by (simp add: nth_stirling_row)

end
```

3 Permutations, both general and specifically on finite sets.

```
theory Permutations
  imports
    HOL−Library.Multiset
    HOL−Library.Disjoint_Sets
    Transposition
begin
```

3.1 Auxiliary

```
abbreviation (input) fixpoints :: ‹('a ⇒ 'a) ⇒ 'a set›
  where ‹fixpoints f ≡ {x. f x = x}›

lemma inj_on_fixpoints:
  ‹inj_on f (fixpoints f)›
  by (rule inj_onI) simp

lemma bij_betw_fixpoints:
  ‹bij_betw f (fixpoints f) (fixpoints f)›
  using inj_on_fixpoints by (auto simp add: bij_betw_def)
```

3.2 Basic definition and consequences

```
definition permutes :: ‹('a ⇒ 'a) ⇒ 'a set ⇒ bool›  (infixr ‹permutes› 41)
  where ‹p permutes S ⟷ (∀x. x ∉ S ⟶ p x = x) ∧ (∀y. ∃!x. p x = y)›

lemma bij_imp_permutes:
  ‹p permutes S› if ‹bij_betw p S S› and stable: ‹⋀x. x ∉ S ⟹ p x = x›
proof −
  note ‹bij_betw p S S›
  moreover have ‹bij_betw p (− S) (− S)›
    by (auto simp add: stable intro!: bij_betw_imageI inj_onI)
```

```
  ultimately have ‹bij_betw p (S ∪ − S) (S ∪ − S)›
    by (rule bij_betw_combine) simp
  then have ‹∃!x. p x = y› for y
    by (simp add: bij_iff)
  with stable show ?thesis
    by (simp add: permutes_def)
qed

context
  fixes p :: ‹'a ⇒ 'a› and S :: ‹'a set›
  assumes perm: ‹p permutes S›
begin

lemma permutes_inj:
  ‹inj p›
  using perm by (auto simp: permutes_def inj_on_def)

lemma permutes_image:
  ‹p ‘ S = S›
proof (rule set_eqI)
  fix x
  show ‹x ∈ p ‘ S ⟷ x ∈ S›
  proof
    assume ‹x ∈ p ‘ S›
    then obtain y where ‹y ∈ S› ‹p y = x›
      by blast
    with perm show ‹x ∈ S›
      by (cases ‹y = x›) (auto simp add: permutes_def)
  next
    assume ‹x ∈ S›
    with perm obtain y where ‹y ∈ S› ‹p y = x›
      by (metis permutes_def)
    then show ‹x ∈ p ‘ S›
      by blast
  qed
qed

lemma permutes_not_in:
  ‹x ∉ S ⟹ p x = x›
  using perm by (auto simp: permutes_def)

lemma permutes_image_complement:
  ‹p ‘ (− S) = − S›
  by (auto simp add: permutes_not_in)

lemma permutes_in_image:
  ‹p x ∈ S ⟷ x ∈ S›
  using permutes_image permutes_inj by (auto dest: inj_image_mem_iff)
```

```
lemma permutes_surj:
  ‹surj p›
proof -
  have ‹p ` (S ∪ - S) = p ` S ∪ p ` (- S)›
    by (rule image_Un)
  then show ?thesis
    by (simp add: permutes_image permutes_image_complement)
qed

lemma permutes_inv_o:
  shows p ∘ inv p = id
    and inv p ∘ p = id
  using permutes_inj permutes_surj
  unfolding inj_iff [symmetric] surj_iff [symmetric] by auto

lemma permutes_inverses:
  shows p (inv p x) = x
    and inv p (p x) = x
  using permutes_inv_o [unfolded fun_eq_iff o_def] by auto

lemma permutes_inv_eq:
  ‹inv p y = x ⟷ p x = y›
  by (auto simp add: permutes_inverses)

lemma permutes_inj_on:
  ‹inj_on p A›
  by (rule inj_on_subset [of _ UNIV]) (auto intro: permutes_inj)

lemma permutes_bij:
  ‹bij p›
  unfolding bij_def by (metis permutes_inj permutes_surj)

lemma permutes_imp_bij:
  ‹bij_betw p S S›
  by (simp add: bij_betw_def permutes_image permutes_inj_on)

lemma permutes_subset:
  ‹p permutes T› if ‹S ⊆ T›
proof (rule bij_imp_permutes)
  define R where ‹R = T - S›
  with that have ‹T = R ∪ S› ‹R ∩ S = {}›
    by auto
  then have ‹p x = x› if ‹x ∈ R› for x
    using that by (auto intro: permutes_not_in)
  then have ‹p ` R = R›
    by simp
  with ‹T = R ∪ S› show ‹bij_betw p T T›
    by (simp add: bij_betw_def permutes_inj_on image_Un permutes_image)
  fix x
```

```
  assume ‹x ∉ T›
  with ‹T = R ∪ S› show ‹p x = x›
    by (simp add: permutes_not_in)
qed

lemma permutes_imp_permutes_insert:
  ‹p permutes insert x S›
  by (rule permutes_subset) auto

end

lemma permutes_id [simp]:
  ‹id permutes S›
  by (auto intro: bij_imp_permutes)

lemma permutes_empty [simp]:
  ‹p permutes {} ⟷ p = id›
proof
  assume ‹p permutes {}›
  then show ‹p = id›
    by (auto simp add: fun_eq_iff permutes_not_in)
next
  assume ‹p = id›
  then show ‹p permutes {}›
    by simp
qed

lemma permutes_sing [simp]:
  ‹p permutes {a} ⟷ p = id›
proof
  assume perm: ‹p permutes {a}›
  show ‹p = id›
  proof
    fix x
    from perm have ‹p ` {a} = {a}›
      by (rule permutes_image)
    with perm show ‹p x = id x›
      by (cases ‹x = a›) (auto simp add: permutes_not_in)
  qed
next
  assume ‹p = id›
  then show ‹p permutes {a}›
    by simp
qed

lemma permutes_univ: p permutes UNIV ⟷ (∀ y. ∃!x. p x = y)
  by (simp add: permutes_def)

lemma permutes_swap_id: a ∈ S ⟹ b ∈ S ⟹ transpose a b permutes S
```

```
  by (rule bij_imp_permutes) (auto intro: transpose_apply_other)

lemma permutes_superset:
  ‹p permutes T› if ‹p permutes S› ‹⋀x. x ∈ S − T ⟹ p x = x›
proof −
  define R U where ‹R = T ∩ S› and ‹U = S − T›
  then have ‹T = R ∪ (T − S)› ‹S = R ∪ U› ‹R ∩ U = {}›
    by auto
  from that ‹U = S − T› have ‹p ‘ U = U›
    by simp
  from ‹p permutes S› have ‹bij_betw p (R ∪ U) (R ∪ U)›
    by (simp add: permutes_imp_bij ‹S = R ∪ U›)
  moreover have ‹bij_betw p U U›
    using that ‹U = S − T› by (simp add: bij_betw_def permutes_inj_on)
  ultimately have ‹bij_betw p R R›
    using ‹R ∩ U = {}› ‹R ∩ U = {}› by (rule bij_betw_partition)
  then have ‹p permutes R›
  proof (rule bij_imp_permutes)
    fix x
    assume ‹x ∉ R›
    with ‹R = T ∩ S› ‹p permutes S› show ‹p x = x›
      by (cases ‹x ∈ S›) (auto simp add: permutes_not_in that(2))
  qed
  then have ‹p permutes R ∪ (T − S)›
    by (rule permutes_subset) simp
  with ‹T = R ∪ (T − S)› show ?thesis
    by simp
qed

lemma permutes_bij_inv_into:
  fixes A :: 'a set
    and B :: 'b set
  assumes p permutes A
    and bij_betw f A B
  shows (λx. if x ∈ B then f (p (inv_into A f x)) else x) permutes B
proof (rule bij_imp_permutes)
  from assms have bij_betw p A A bij_betw f A B bij_betw (inv_into A f) B A
    by (auto simp add: permutes_imp_bij bij_betw_inv_into)
  then have bij_betw (f ∘ p ∘ inv_into A f) B B
    by (simp add: bij_betw_trans)
  then show bij_betw (λx. if x ∈ B then f (p (inv_into A f x)) else x) B B
    by (subst bij_betw_cong[where g=f ∘ p ∘ inv_into A f]) auto
next
  fix x
  assume x ∉ B
  then show (if x ∈ B then f (p (inv_into A f x)) else x) = x by auto
qed

lemma permutes_image_mset:
```

assumes *p permutes A*
shows *image_mset p* (*mset_set A*) = *mset_set A*
using *assms* **by** (*metis image_mset_mset_set bij_betw_imp_inj_on permutes_imp_bij permutes_image*)

lemma *permutes_implies_image_mset_eq*:
assumes *p permutes A* $\bigwedge x.\ x \in A \Longrightarrow f\ x = f'\ (p\ x)$
shows *image_mset f'* (*mset_set A*) = *image_mset f* (*mset_set A*)
proof −
have $f\ x = f'\ (p\ x)$ **if** $x \in\#$ *mset_set A* **for** *x*
using *assms*(*2*)[*of x*] *that* **by** (*cases finite A*) *auto*
with *assms* **have** *image_mset f* (*mset_set A*) = *image_mset* ($f' \circ p$) (*mset_set A*)
by (*auto intro*!: *image_mset_cong*)
also have … = *image_mset f'* (*image_mset p* (*mset_set A*))
by (*simp add*: *image_mset.compositionality*)
also have … = *image_mset f'* (*mset_set A*)
proof −
from *assms permutes_image_mset* **have** *image_mset p* (*mset_set A*) = *mset_set A*
by *blast*
then show *?thesis* **by** *simp*
qed
finally show *?thesis* **..**
qed

3.3 Group properties

lemma *permutes_compose*: *p permutes S* $\Longrightarrow$ *q permutes S* $\Longrightarrow$ $q \circ p$ *permutes S*
unfolding *permutes_def o_def* **by** *metis*

lemma *permutes_inv*:
assumes *p permutes S*
shows *inv p permutes S*
using *assms* **unfolding** *permutes_def permutes_inv_eq*[*OF assms*] **by** *metis*

lemma *permutes_inv_inv*:
assumes *p permutes S*
shows *inv* (*inv p*) = *p*
unfolding *fun_eq_iff permutes_inv_eq*[*OF assms*] *permutes_inv_eq*[*OF permutes_inv*[*OF assms*]]
by *blast*

lemma *permutes_invI*:
assumes *perm*: *p permutes S*
and *inv*: $\bigwedge x.\ x \in S \Longrightarrow p'\ (p\ x) = x$
and *outside*: $\bigwedge x.\ x \notin S \Longrightarrow p'\ x = x$
shows *inv p* = p'
proof

```
  show inv p x = p' x for x
  proof (cases x ∈ S)
    case True
    from assms have p' x = p' (p (inv p x))
      by (simp add: permutes_inverses)
    also from permutes_inv[OF perm] True have ... = inv p x
      by (subst inv) (simp_all add: permutes_in_image)
    finally show ?thesis ..
  next
    case False
    with permutes_inv[OF perm] show ?thesis
      by (simp_all add: outside permutes_not_in)
  qed
qed

lemma permutes_vimage: f permutes A ⟹ f -` A = A
  by (simp add: bij_vimage_eq_inv_image permutes_bij permutes_image[OF permutes_inv])
```

3.4 Mapping permutations with bijections

```
lemma bij_betw_permutations:
  assumes bij_betw f A B
  shows   bij_betw (λπ x. if x ∈ B then f (π (inv_into A f x)) else x)
            {π. π permutes A} {π. π permutes B} (is bij_betw ?f _ _)
proof -
  let ?g = (λπ x. if x ∈ A then inv_into A f (π (f x)) else x)
  show ?thesis
  proof (rule bij_betw_byWitness [of _ ?g], goal_cases)
    case 3
    show ?case using permutes_bij_inv_into[OF _ assms] by auto
  next
    case 4
   have bij_inv: bij_betw (inv_into A f) B A by (intro bij_betw_inv_into assms)
    {
      fix π assume π permutes B
      from permutes_bij_inv_into[OF this bij_inv] and assms
        have (λx. if x ∈ A then inv_into A f (π (f x)) else x) permutes A
        by (simp add: inv_into_inv_into_eq cong: if_cong)
    }
    from this show ?case by (auto simp: permutes_inv)
  next
    case 1
    thus ?case using assms
     by (auto simp: fun_eq_iff permutes_not_in permutes_in_image bij_betw_inv_into_left
             dest: bij_betwE)
  next
    case 2
    moreover have bij_betw (inv_into A f) B A
```

by (*intro bij_betw_inv_into assms*)
ultimately show *?case* **using** *assms*
by (*auto simp*: *fun_eq_iff permutes_not_in permutes_in_image bij_betw_inv_into_right*

dest: *bij_betwE*)
qed
qed

lemma *bij_betw_derangements*:
assumes *bij_betw f A B*
shows *bij_betw* ($\lambda\pi$ *x*. *if* $x \in B$ *then f* (π (*inv_into A f x*)) *else x*)
$\{\pi.\ \pi$ *permutes* $A \wedge (\forall x{\in}A.\ \pi\ x \neq x)\}$ $\{\pi.\ \pi$ *permutes* $B \wedge (\forall x{\in}B.\ \pi\ x$
$\neq x)\}$
(**is** *bij_betw ?f _ _*)
proof −
let $?g = (\lambda\pi\ x.$ *if* $x \in A$ *then inv_into A f* (π (*f x*)) *else x*)
show *?thesis*
proof (*rule bij_betw_byWitness* [*of _ ?g*], *goal_cases*)
case *3*
have *?f* $\pi\ x \neq x$ **if** π *permutes* $A\ \bigwedge x.\ x \in A \Longrightarrow \pi\ x \neq x\ x \in B$ **for** $\pi\ x$
using *that* **and** *assms* **by** (*metis bij_betwE bij_betw_imp_inj_on bij_betw_imp_surj_on inv_into_f_f inv_into_into permutes_imp_bij*)
with *permutes_bij_inv_into*[*OF _ assms*] **show** *?case* **by** *auto*
next
case *4*
have *bij_inv*: *bij_betw* (*inv_into A f*) *B A* **by** (*intro bij_betw_inv_into assms*)
have *?g* π *permutes A* **if** π *permutes B* **for** π
using *permutes_bij_inv_into*[*OF that bij_inv*] **and** *assms*
by (*simp add*: *inv_into_inv_into_eq cong*: *if_cong*)
moreover have *?g* $\pi\ x \neq x$ **if** π *permutes* $B\ \bigwedge x.\ x \in B \Longrightarrow \pi\ x \neq x\ x \in A$
for $\pi\ x$
using *that* **and** *assms* **by** (*metis bij_betwE bij_betw_imp_surj_on f_inv_into_f*
permutes_imp_bij)
ultimately show *?case* **by** *auto*
next
case *1*
thus *?case* **using** *assms*
by (*force simp*: *fun_eq_iff permutes_not_in permutes_in_image bij_betw_inv_into_left*
dest: *bij_betwE*)
next
case *2*
moreover have *bij_betw* (*inv_into A f*) *B A*
by (*intro bij_betw_inv_into assms*)
ultimately show *?case* **using** *assms*
by (*force simp*: *fun_eq_iff permutes_not_in permutes_in_image bij_betw_inv_into_right*

dest: *bij_betwE*)
qed
qed

3.5 The number of permutations on a finite set

lemma *permutes_insert_lemma*:
  assumes *p permutes* (*insert a S*)
  shows *transpose a* (*p a*) $\circ$ *p permutes S*
proof (*rule permutes_superset*[**where** *S* = *insert a S*])
  show *Transposition.transpose a* (*p a*) $\circ$ *p permutes insert a S*
   by (*meson assms insertI1 permutes_compose permutes_in_image permutes_swap_id*)
qed *auto*

lemma *permutes_insert*: {*p*. *p permutes* (*insert a S*)} =
  (λ(*b*, *p*). *transpose a b* $\circ$ *p*) ‘ {(*b*, *p*). *b* $\in$ *insert a S* $\wedge$ *p* $\in$ {*p*. *p permutes S*}}
proof −
  have *p permutes insert a S* $\longleftrightarrow$
    ($\exists$ *b q*. *p* = *transpose a b* $\circ$ *q* $\wedge$ *b* $\in$ *insert a S* $\wedge$ *q permutes S*) **for** *p*
  proof −
    have $\exists$ *b q*. *p* = *transpose a b* $\circ$ *q* $\wedge$ *b* $\in$ *insert a S* $\wedge$ *q permutes S*
      if *p*: *p permutes insert a S*
    proof −
      let *?b* = *p a*
      let *?q* = *transpose a* (*p a*) $\circ$ *p*
      have $*$: *p* = *transpose a ?b* $\circ$ *?q*
        by (*simp add*: *fun_eq_iff o_assoc*)
      have $**$: *?b* $\in$ *insert a S*
        unfolding *permutes_in_image*[*OF p*] **by** *simp*
      from *permutes_insert_lemma*[*OF p*] $*$ $**$ **show** *?thesis*
        by *blast*
    qed
    moreover have *p permutes insert a S*
      if *bq*: *p* = *transpose a b* $\circ$ *q b* $\in$ *insert a S q permutes S* **for** *b q*
    proof −
      from *permutes_subset*[*OF bq*(*3*), *of insert a S*] **have** *q*: *q permutes insert a S*
        by *auto*
      have *a*: *a* $\in$ *insert a S*
        by *simp*
      from *bq*(*1*) *permutes_compose*[*OF q permutes_swap_id*[*OF a bq*(*2*)]] **show**
?thesis
        by *simp*
    qed
    ultimately show *?thesis* **by** *blast*
  qed
  then show *?thesis* **by** *auto*
qed

lemma *card_permutations*:
  assumes *card S* = *n*
    and *finite S*
  shows *card* {*p*. *p permutes S*} = *fact n*
  using *assms*(*2*,*1*)
proof (*induct arbitrary*: *n*)

```
  case empty
  then show ?case by simp
next
  case (insert x F)
  {
    fix n
    assume card_insert: card (insert x F) = n
    let ?xF = {p. p permutes insert x F}
    let ?pF = {p. p permutes F}
    let ?pF' = {(b, p). b ∈ insert x F ∧ p ∈ ?pF}
    let ?g = (λ(b, p). transpose x b ∘ p)
    have xfgpF': ?xF = ?g ` ?pF'
      by (rule permutes_insert[of x F])
    from ‹x ∉ F› ‹finite F› card_insert have Fs: card F = n − 1
      by auto
    from ‹finite F› insert.hyps Fs have pFs: card ?pF = fact (n − 1)
      by auto
    then have finite ?pF
      by (auto intro: card_ge_0_finite)
    with ‹finite F› card.insert_remove have pF'f: finite ?pF'
      by simp
    have ginj: inj_on ?g ?pF'
    proof −
      {
        fix b p c q
        assume bp: (b, p) ∈ ?pF'
        assume cq: (c, q) ∈ ?pF'
        assume eq: ?g (b, p) = ?g (c, q)
        from bp cq have pF: p permutes F and qF: q permutes F
          by auto
        from pF ‹x ∉ F› eq have b = ?g (b, p) x
          by (auto simp: permutes_def fun_upd_def fun_eq_iff)
        also from qF ‹x ∉ F› eq have ... = ?g (c, q) x
          by (auto simp: fun_upd_def fun_eq_iff)
        also from qF ‹x ∉ F› have ... = c
          by (auto simp: permutes_def fun_upd_def fun_eq_iff)
        finally have b = c .
        then have transpose x b = transpose x c
          by simp
        with eq have transpose x b ∘ p = transpose x b ∘ q
          by simp
        then have transpose x b ∘ (transpose x b ∘ p) = transpose x b ∘ (transpose
x b ∘ q)
          by simp
        then have p = q
          by (simp add: o_assoc)
        with ‹b = c› have (b, p) = (c, q)
          by simp
      }
```

then show *?thesis*
unfolding *inj_on_def* **by** *blast*
qed
from ‹$x \notin F$› ‹*finite F*› *card_insert* **have** $n \neq 0$
by *auto*
then have $\exists\, m.\ n = Suc\ m$
by *presburger*
then obtain *m* **where** *n*: $n = Suc\ m$
by *blast*
from *pFs card_insert* **have** ∗: *card ?xF = fact n*
unfolding *xfgpF′ card_image*[*OF ginj*]
using ‹*finite F*› ‹*finite ?pF*›
by (*simp only*: *Collect_case_prod Collect_mem_eq card_cartesian_product*) (*simp add*: *n*)
from *finite_imageI*[*OF pF′f, of ?g*] **have** *xFf*: *finite ?xF*
by (*simp add*: *xfgpF′ n*)
from ∗ **have** *card ?xF = fact n*
unfolding *xFf* **by** *blast*
}
with *insert* **show** *?case* **by** *simp*
qed

lemma *finite_permutations*:
assumes *finite S*
shows *finite* {*p. p permutes S*}
using *card_permutations*[*OF refl assms*] **by** (*auto intro*: *card_ge_0_finite*)

3.6 Hence a sort of induction principle composing by swaps

lemma *permutes_induct* [*consumes 2, case_names id swap*]:
‹*P p*› **if** ‹*p permutes S*› ‹*finite S*›
and *id*: ‹*P id*›
and *swap*: ‹$\bigwedge a\ b\ p.\ a \in S \Longrightarrow b \in S \Longrightarrow p$ *permutes* $S \Longrightarrow P\ p \Longrightarrow P$ (*transpose* $a\ b \circ p$)›
using ‹*finite S*› ‹*p permutes S*› *swap* **proof** (*induction S arbitrary*: *p*)
case *empty*
with *id* **show** *?case*
by (*simp only*: *permutes_empty*)
next
case (*insert x S p*)
define *q* **where** ‹$q =$ *transpose* x ($p\ x$) $\circ\ p$›
then have *swap_q*: ‹*transpose* x ($p\ x$) $\circ\ q = p$›
by (*simp add*: *o_assoc*)
from ‹*p permutes insert x S*› **have** ‹*q permutes S*›
by (*simp add*: *q_def permutes_insert_lemma*)
then have ‹*q permutes insert x S*›
by (*simp add*: *permutes_imp_permutes_insert*)
from ‹*q permutes S*› **have** ‹*P q*›
by (*auto intro*: *insert.IH insert.prems*(*2*) *permutes_imp_permutes_insert*)

have ‹$x \in$ *insert* x S›
  by *simp*
moreover from ‹p *permutes insert* x S› **have** ‹p $x \in$ *insert* x S›
  using *permutes_in_image* [*of* p ‹*insert* x S› x] **by** *simp*
ultimately have ‹P (*transpose* x (p x) $\circ$ q)›
  using ‹q *permutes insert* x S› ‹P q›
  by (*rule insert.prems*(*2*))
then show *?case*
  by (*simp add*: *swap_q*)
qed

lemma *permutes_rev_induct* [*consumes 2*, *case_names id swap*]:
‹P p› **if** ‹p *permutes* S› ‹*finite* S›
and id': ‹P *id*›
and $swap'$: ‹$\bigwedge a\ b\ p.\ a \in S \Longrightarrow b \in S \Longrightarrow p$ *permutes* $S \Longrightarrow P\ p \Longrightarrow P\ (p \circ$ *transpose* a b)›
using ‹p *permutes* S› ‹*finite* S› **proof** (*induction rule*: *permutes_induct*)
  case *id*
  from id' **show** *?case* **.**
next
  case (*swap* a b p)
  then have ‹*bij* p›
    using *permutes_bij* **by** *blast*
  have ‹P ($p \circ$ *transpose* (*inv* p a) (*inv* p b))›
    by (*rule* $swap'$) (*auto simp add*: *swap permutes_in_image permutes_inv*)
  also have ‹$p \circ$ *transpose* (*inv* p a) (*inv* p b) = *transpose* a b $\circ$ p›
    using ‹*bij* p› **by** (*rule transpose_comp_eq* [*symmetric*])
  finally show *?case* **.**
qed

3.7 Permutations of index set for iterated operations

lemma (**in** *comm_monoid_set*) *permute*:
  assumes p *permutes* S
  shows F g S = F ($g \circ p$) S
proof −
  from ‹p *permutes* S› **have** *inj* p
    by (*rule permutes_inj*)
  then have *inj_on* p S
    by (*auto intro*: *subset_inj_on*)
  then have F g (p ‘ S) = F ($g \circ p$) S
    by (*rule reindex*)
  moreover from ‹p *permutes* S› **have** p ‘ S = S
    by (*rule permutes_image*)
  ultimately show *?thesis*
    by *simp*
qed

3.8 Permutations as transposition sequences

inductive *swapidseq* :: *nat* ⇒ (*'a* ⇒ *'a*) ⇒ *bool*
  where
    id[*simp*]: *swapidseq 0 id*
  | *comp_Suc*: *swapidseq n p* ⟹ *a* ≠ *b* ⟹ *swapidseq* (*Suc n*) (*transpose a b* ∘ *p*)

declare *id*[*unfolded id_def*, *simp*]

definition *permutation p* ⟷ (∃ *n*. *swapidseq n p*)

3.9 Some closure properties of the set of permutations, with lengths

lemma *permutation_id*[*simp*]: *permutation id*
  unfolding *permutation_def* **by** (*rule exI*[**where** *x=0*]) *simp*

declare *permutation_id*[*unfolded id_def*, *simp*]

lemma *swapidseq_swap*: *swapidseq* (*if a* = *b then 0 else 1*) (*transpose a b*)
  using *swapidseq.simps* **by** *fastforce*

lemma *permutation_swap_id*: *permutation* (*transpose a b*)
  by (*meson permutation_def swapidseq_swap*)

lemma *swapidseq_comp_add*: *swapidseq n p* ⟹ *swapidseq m q* ⟹ *swapidseq* (*n* + *m*) (*p* ∘ *q*)
proof (*induct n p arbitrary*: *m q rule*: *swapidseq.induct*)
  case (*id m q*)
  then show *?case* **by** *simp*
next
  case (*comp_Suc n p a b m q*)
  then show *?case*
    by (*metis add_Suc comp_assoc swapidseq.comp_Suc*)
qed

lemma *permutation_compose*: *permutation p* ⟹ *permutation q* ⟹ *permutation* (*p* ∘ *q*)
  unfolding *permutation_def* **using** *swapidseq_comp_add*[*of _ p _ q*] **by** *metis*

lemma *swapidseq_endswap*: *swapidseq n p* ⟹ *a* ≠ *b* ⟹ *swapidseq* (*Suc n*) (*p* ∘ *transpose a b*)
  by (*induct n p rule*: *swapidseq.induct*)
    (*use swapidseq_swap*[*of a b*] **in** ‹*auto simp add*: *comp_assoc intro*: *swapidseq.comp_Suc*›)

lemma *swapidseq_inverse_exists*: *swapidseq n p* ⟹ ∃ *q*. *swapidseq n q* ∧ *p* ∘ *q* = *id* ∧ *q* ∘ *p* = *id*
proof (*induct n p rule*: *swapidseq.induct*)

case *id*
then show *?case*
by (*rule exI*[**where** *x*=*id*]) *simp*
next
case (*comp_Suc n p a b*)
from *comp_Suc.hyps* **obtain** *q* **where** *q*: *swapidseq n q* $p \circ q = id$ $q \circ p = id$
by *blast*
let $?q = q \circ$ *transpose a b*
note H = *comp_Suc.hyps*
from *swapidseq_swap*[*of a b*] *H*(*3*) **have** ∗: *swapidseq 1* (*transpose a b*)
by *simp*
from *swapidseq_comp_add*[*OF q*(*1*) ∗] **have** ∗∗: *swapidseq* (*Suc n*) *?q*
by *simp*
have *transpose a b* $\circ\ p \circ\ ?q$ = *transpose a b* $\circ\ (p \circ q) \circ$ *transpose a b*
by (*simp add*: *o_assoc*)
also have ... = *id*
by (*simp add*: *q*(*2*))
finally have ∗∗∗: *transpose a b* $\circ\ p \circ\ ?q = id$ **.**
have $?q \circ$ (*transpose a b* $\circ\ p$) = $q \circ$ (*transpose a b* $\circ$ *transpose a b*) $\circ\ p$
by (*simp only*: *o_assoc*)
then have $?q \circ$ (*transpose a b* $\circ\ p$) = *id*
by (*simp add*: *q*(*3*))
with ∗∗ ∗∗∗ **show** *?case*
by *blast*
qed

lemma *swapidseq_inverse*:
assumes *swapidseq n p*
shows *swapidseq n* (*inv p*)
using *swapidseq_inverse_exists*[*OF assms*] *inv_unique_comp*[*of p*] **by** *auto*

lemma *permutation_inverse*: *permutation p* $\Longrightarrow$ *permutation* (*inv p*)
using *permutation_def swapidseq_inverse* **by** *blast*

3.10 Various combinations of transpositions with 2, 1 and 0 common elements

lemma *swap_id_common*: $a \neq c \Longrightarrow b \neq c \Longrightarrow$
transpose a b $\circ$ *transpose a c* = *transpose b c* $\circ$ *transpose a b*
by (*simp add*: *fun_eq_iff transpose_def*)

lemma *swap_id_common'*: $a \neq b \Longrightarrow a \neq c \Longrightarrow$
transpose a c $\circ$ *transpose b c* = *transpose b c* $\circ$ *transpose a b*
by (*simp add*: *fun_eq_iff transpose_def*)

lemma *swap_id_independent*: $a \neq c \Longrightarrow a \neq d \Longrightarrow b \neq c \Longrightarrow b \neq d \Longrightarrow$
transpose a b $\circ$ *transpose c d* = *transpose c d* $\circ$ *transpose a b*
by (*simp add*: *fun_eq_iff transpose_def*)

3.11 The identity map only has even transposition sequences

lemma *symmetry_lemma*:
  assumes $\bigwedge a\ b\ c\ d.\ P\ a\ b\ c\ d \Longrightarrow P\ a\ b\ d\ c$
    and $\bigwedge a\ b\ c\ d.\ a \neq b \Longrightarrow c \neq d \Longrightarrow$
      $a = c \wedge b = d \vee a = c \wedge b \neq d \vee a \neq c \wedge b = d \vee a \neq c \wedge a \neq d \wedge b \neq c$
$\wedge\ b \neq d \Longrightarrow$
      $P\ a\ b\ c\ d$
  shows $\bigwedge a\ b\ c\ d.\ a \neq b \longrightarrow c \neq d \longrightarrow\ P\ a\ b\ c\ d$
  using *assms* **by** *metis*

lemma *swap_general*:
  assumes $a \neq b\ c \neq d$
  shows *transpose* $a\ b \circ$ *transpose* $c\ d = id \vee$
  $(\exists x\ y\ z.\ x \neq a \wedge y \neq a \wedge z \neq a \wedge x \neq y \wedge$
    transpose $a\ b \circ$ *transpose* $c\ d =$ *transpose* $x\ y \circ$ *transpose* $a\ z)$
   by (*metis assms swap_id_common′ swap_id_independent transpose_commute transpose_comp_involutory*)

lemma *swapidseq_id_iff* [*simp*]: *swapidseq 0 p* $\longleftrightarrow p = id$
  using *swapidseq.cases*[*of 0 p p = id*] **by** *auto*

lemma *swapidseq_cases*: *swapidseq n p* $\longleftrightarrow$
    $n = 0 \wedge p = id \vee (\exists a\ b\ q\ m.\ n = Suc\ m \wedge p =$ *transpose* $a\ b \circ q \wedge$ *swapidseq*
$m\ q \wedge a \neq b)$
  by (*meson comp_Suc id swapidseq.cases*)

lemma *fixing_swapidseq_decrease*:
  assumes *swapidseq n p*
    and $a \neq b$
    and (*transpose* $a\ b \circ p$) $a = a$
  shows $n \neq 0 \wedge$ *swapidseq* $(n - 1)$ (*transpose* $a\ b \circ p$)
  using *assms*
proof (*induct n arbitrary*: *p a b*)
  case *0*
  then show *?case*
    by (*auto simp add*: *fun_upd_def*)
next
  case (*Suc n p a b*)
  from *Suc.prems*(*1*) *swapidseq_cases*[*of Suc n p*]
  obtain *c d q m* **where**
    cdqm: $Suc\ n = Suc\ m\ p =$ *transpose* $c\ d \circ q$ *swapidseq* $m\ q\ c \neq d\ n = m$
    by *auto*
  consider *transpose* $a\ b \circ$ *transpose* $c\ d = id$
    | $x\ y\ z$ **where** $x \neq a\ y \neq a\ z \neq a\ x \neq y$
      transpose $a\ b \circ$ *transpose* $c\ d =$ *transpose* $x\ y \circ$ *transpose* $a\ z$
    using *swap_general*[*OF Suc.prems*(*2*) *cdqm*(*4*)] **by** *metis*
  then show *?case*
  proof *cases*
    case *1*

```
    then show ?thesis
      by (simp only: cdqm o_assoc) (simp add: cdqm)
  next
    case 2
    then have az: a ≠ z
      by simp
    from 2 have ∗: (transpose x y ∘ h) a = a ⟷ h a = a for h
      by (simp add: transpose_def)
    from cdqm(2) have transpose a b ∘ p = transpose a b ∘ (transpose c d ∘ q)
      by simp
    then have §: transpose a b ∘ p = transpose x y ∘ (transpose a z ∘ q)
      by (simp add: o_assoc 2)
    obtain ∗∗: swapidseq (n − 1) (transpose a z ∘ q) and n≠0
      by (metis ∗ § Suc.hyps Suc.prems(3) az cdqm(3,5))
    then have Suc n − 1 = Suc (n − 1)
      by auto
    with 2 show ?thesis
      using ∗∗ § swapidseq.simps by blast
  qed
qed

lemma swapidseq_identity_even:
  assumes swapidseq n (id :: 'a ⇒ 'a)
  shows even n
  using ‹swapidseq n id›
proof (induct n rule: nat_less_induct)
  case H: (1 n)
  consider n = 0
    | a b :: 'a and q m where n = Suc m id = transpose a b ∘ q swapidseq m q a
≠ b
    using H(2)[unfolded swapidseq_cases[of n id]] by auto
  then show ?case
  proof cases
    case 1
    then show ?thesis by presburger
  next
    case h: 2
    from fixing_swapidseq_decrease[OF h(3,4), unfolded h(2)[symmetric]]
    have m: m ≠ 0 swapidseq (m − 1) (id :: 'a ⇒ 'a)
      by auto
    from h m have mn: m − 1 < n
      by arith
    from H(1)[rule_format, OF mn m(2)] h(1) m(1) show ?thesis
      by presburger
  qed
qed
```

3.12 Therefore we have a welldefined notion of parity

definition *evenperm p = even (SOME n. swapidseq n p)*

lemma *swapidseq_even_even*:
assumes *m*: *swapidseq m p*
and *n*: *swapidseq n p*
shows *even m ⟷ even n*
proof −
from *swapidseq_inverse_exists*[*OF n*] **obtain** *q* **where** *q*: *swapidseq n q p ∘ q = id q ∘ p = id*
by *blast*
from *swapidseq_identity_even*[*OF swapidseq_comp_add*[*OF m q*(*1*), *unfolded q*]] **show** *?thesis*
by *arith*
qed

lemma *evenperm_unique*:
assumes *swapidseq n p* **and***even n = b*
shows *evenperm p = b*
by (*metis evenperm_def assms someI swapidseq_even_even*)

3.13 And it has the expected composition properties

lemma *evenperm_id*[*simp*]: *evenperm id = True*
by (*rule evenperm_unique*[**where** *n = 0*]) *simp_all*

lemma *evenperm_identity* [*simp*]:
‹*evenperm* (*λx. x*)›
using *evenperm_id* **by** (*simp add*: *id_def* [*abs_def*])

lemma *evenperm_swap*: *evenperm* (*transpose a b*) = (*a = b*)
by (*rule evenperm_unique*[**where** *n=if a = b then 0 else 1*]) (*simp_all add*: *swapidseq_swap*)

lemma *evenperm_comp*:
assumes *permutation p permutation q*
shows *evenperm* (*p ∘ q*) ⟷ *evenperm p = evenperm q*
proof −
from *assms* **obtain** *n m* **where** *n*: *swapidseq n p* **and** *m*: *swapidseq m q*
unfolding *permutation_def* **by** *blast*
have *even* (*n + m*) ⟷ (*even n ⟷ even m*)
by *arith*
from *evenperm_unique*[*OF n refl*] *evenperm_unique*[*OF m refl*]
and *evenperm_unique*[*OF swapidseq_comp_add*[*OF n m*] *this*] **show** *?thesis*
by *blast*
qed

lemma *evenperm_inv*:
assumes *permutation p*

shows *evenperm* (*inv p*) = *evenperm p*
proof −
from *assms* **obtain** *n* **where** *n*: *swapidseq n p*
unfolding *permutation_def* **by** *blast*
show *?thesis*
by (*rule evenperm_unique*[*OF swapidseq_inverse*[*OF n*] *evenperm_unique*[*OF n refl*, *symmetric*]])
qed

3.14 A more abstract characterization of permutations

lemma *permutation_bijective*:
assumes *permutation p*
shows *bij p*
by (*meson assms o_bij permutation_def swapidseq_inverse_exists*)

lemma *permutation_finite_support*:
assumes *permutation p*
shows *finite* $\{x.\ p\ x \neq x\}$
proof −
from *assms* **obtain** *n* **where** *swapidseq n p*
unfolding *permutation_def* **by** *blast*
then show *?thesis*
proof (*induct n p rule*: *swapidseq.induct*)
case *id*
then show *?case* **by** *simp*
next
case (*comp_Suc n p a b*)
let *?S* = *insert a* (*insert b* $\{x.\ p\ x \neq x\}$)
from *comp_Suc.hyps*(*2*) **have** ∗: *finite ?S*
by *simp*
from ‹$a \neq b$› **have** ∗∗: $\{x.\ (transpose\ a\ b \circ p)\ x \neq x\} \subseteq$ *?S*
by *auto*
show *?case*
by (*rule finite_subset*[*OF* ∗∗ ∗])
qed
qed

lemma *permutation_lemma*:
assumes *finite S*
and *bij p*
and $\forall x.\ x \notin S \longrightarrow p\ x = x$
shows *permutation p*
using *assms*
proof (*induct S arbitrary*: *p rule*: *finite_induct*)
case *empty*
then show *?case*
by *simp*
next

case (*insert a F p*)
let *?r* = *transpose a* (*p a*) ∘ *p*
let *?q* = *transpose a* (*p a*) ∘ *?r*
have ∗: *?r a* = *a*
by *simp*
from *insert* ∗ **have** ∗∗: ∀ *x*. *x* ∉ *F* ⟶ *?r x* = *x*
by (*metis bij_pointE comp_apply id_apply insert_iff swap_apply*(*3*))
have *bij ?r*
using *insert* **by** (*simp add*: *bij_comp*)
have *permutation ?r*
by (*rule insert*(*3*)[*OF* ‹*bij ?r*› ∗∗])
then have *permutation ?q*
by (*simp add*: *permutation_compose permutation_swap_id*)
then show *?case*
by (*simp add*: *o_assoc*)
qed

lemma *permutation*: *permutation p* ⟷ *bij p* ∧ *finite* {*x*. *p x* ≠ *x*}
using *permutation_bijective permutation_finite_support permutation_lemma* **by** *auto*

lemma *permutation_inverse_works*:
assumes *permutation p*
shows *inv p* ∘ *p* = *id*
and *p* ∘ *inv p* = *id*
using *permutation_bijective* [*OF assms*] **by** (*auto simp*: *bij_def inj_iff surj_iff*)

lemma *permutation_inverse_compose*:
assumes *p*: *permutation p*
and *q*: *permutation q*
shows *inv* (*p* ∘ *q*) = *inv q* ∘ *inv p*
by (*simp add*: *o_inv_distrib p permutation_bijective q*)

3.15 Relation to *permutes*

lemma *permutes_imp_permutation*:
‹*permutation p*› **if** ‹*finite S*› ‹*p permutes S*›
proof −
from ‹*p permutes S*› **have** ‹{*x*. *p x* ≠ *x*} ⊆ *S*›
by (*auto dest*: *permutes_not_in*)
then have ‹*finite* {*x*. *p x* ≠ *x*}›
using ‹*finite S*› **by** (*rule finite_subset*)
moreover from ‹*p permutes S*› **have** ‹*bij p*›
by (*auto dest*: *permutes_bij*)
ultimately show *?thesis*
by (*simp add*: *permutation*)
qed

lemma *permutation_permutesE*:

assumes ‹*permutation p*›
obtains *S* **where** ‹*finite S*› ‹*p permutes S*›
proof −
from *assms* **have** *fin*: ‹*finite* {*x*. *p x* ≠ *x*}›
by (*simp add*: *permutation*)
from *assms* **have** ‹*bij p*›
by (*simp add*: *permutation*)
also have ‹*UNIV* = {*x*. *p x* ≠ *x*} ∪ {*x*. *p x* = *x*}›
by *auto*
finally have ‹*bij_betw p* {*x*. *p x* ≠ *x*} {*x*. *p x* ≠ *x*}›
by (*rule bij_betw_partition*) (*auto simp add*: *bij_betw_fixpoints*)
then have ‹*p permutes* {*x*. *p x* ≠ *x*}›
by (*auto intro*: *bij_imp_permutes*)
with *fin* **show** *thesis* **..**
qed

lemma *permutation_permutes*: *permutation p* ⟷ (∃ *S*. *finite S* ∧ *p permutes S*)
by (*auto elim*: *permutation_permutesE intro*: *permutes_imp_permutation*)

3.16 Sign of a permutation as a real number

definition *sign* :: ‹(*'a* ⇒ *'a*) ⇒ *int*› — TODO: prefer less generic name
where ‹*sign p* = (*if evenperm p then 1 else* − *1*)›

lemma *sign_cases* [*case_names even odd*]:
obtains ‹*sign p* = *1*› | ‹*sign p* = − *1*›
by (*cases* ‹*evenperm p*›) (*simp_all add*: *sign_def*)

lemma *sign_nz* [*simp*]: *sign p* ≠ *0*
by (*cases p rule*: *sign_cases*) *simp_all*

lemma *sign_id* [*simp*]: *sign id* = *1*
by (*simp add*: *sign_def*)

lemma *sign_identity* [*simp*]:
‹*sign* (λ*x*. *x*) = *1*›
by (*simp add*: *sign_def*)

lemma *sign_inverse*: *permutation p* ⟹ *sign* (*inv p*) = *sign p*
by (*simp add*: *sign_def evenperm_inv*)

lemma *sign_compose*: *permutation p* ⟹ *permutation q* ⟹ *sign* (*p* ∘ *q*) = *sign p* ∗ *sign q*
by (*simp add*: *sign_def evenperm_comp*)

lemma *sign_swap_id*: *sign* (*transpose a b*) = (*if a* = *b then 1 else* − *1*)
by (*simp add*: *sign_def evenperm_swap*)

lemma *sign_idempotent* [*simp*]: *sign p* ∗ *sign p* = *1*

by (*simp add*: *sign_def*)

lemma *sign_left_idempotent* [*simp*]:
‹*sign p* ∗ (*sign p* ∗ *sign q*) = *sign q*›
by (*simp add*: *sign_def*)

term (*bij*, *bij_betw*, *permutation*)

3.17 Permuting a list

This function permutes a list by applying a permutation to the indices.

definition *permute_list* :: (*nat* ⇒ *nat*) ⇒ *'a list* ⇒ *'a list*
where *permute_list f xs* = *map* (λ*i*. *xs* ! (*f i*)) [*0*..<*length xs*]

lemma *permute_list_map*:
assumes *f permutes* {..<*length xs*}
shows *permute_list f* (*map g xs*) = *map g* (*permute_list f xs*)
using *permutes_in_image*[*OF assms*] **by** (*auto simp*: *permute_list_def*)

lemma *permute_list_nth*:
assumes *f permutes* {..<*length xs*} *i* < *length xs*
shows *permute_list f xs* ! *i* = *xs* ! *f i*
using *permutes_in_image*[*OF assms*(*1*)] *assms*(*2*)
by (*simp add*: *permute_list_def*)

lemma *permute_list_Nil* [*simp*]: *permute_list f* [] = []
by (*simp add*: *permute_list_def*)

lemma *length_permute_list* [*simp*]: *length* (*permute_list f xs*) = *length xs*
by (*simp add*: *permute_list_def*)

lemma *permute_list_compose*:
assumes *g permutes* {..<*length xs*}
shows *permute_list* (*f* ∘ *g*) *xs* = *permute_list g* (*permute_list f xs*)
using *assms*[*THEN permutes_in_image*] **by** (*auto simp add*: *permute_list_def*)

lemma *permute_list_ident* [*simp*]: *permute_list* (λ*x*. *x*) *xs* = *xs*
by (*simp add*: *permute_list_def map_nth*)

lemma *permute_list_id* [*simp*]: *permute_list id xs* = *xs*
by (*simp add*: *id_def*)

lemma *mset_permute_list* [*simp*]:
fixes *xs* :: *'a list*
assumes *f permutes* {..<*length xs*}
shows *mset* (*permute_list f xs*) = *mset xs*
proof (*rule multiset_eqI*)
fix *y* :: *'a*
from *assms* **have** [*simp*]: *f x* < *length xs* ⟷ *x* < *length xs* **for** *x*

using *permutes_in_image*[*OF assms*] **by** *auto*
have *count* (*mset* (*permute_list f xs*)) y = *card* ((λi. *xs* ! *f i*) −‘ {y} ∩ {..<*length xs*})
by (*simp add*: *permute_list_def count_image_mset atLeast0LessThan*)
also have (λi. *xs* ! *f i*) −‘ {y} ∩ {..<*length xs*} = *f* −‘ {*i*. *i* < *length xs* ∧ *y* = *xs* ! *i*}
by *auto*
also from *assms* **have** *card* … = *card* {*i*. *i* < *length xs* ∧ *y* = *xs* ! *i*}
by (*intro card_vimage_inj*) (*auto simp*: *permutes_inj permutes_surj*)
also have … = *count* (*mset xs*) *y*
by (*simp add*: *count_mset length_filter_conv_card*)
finally show *count* (*mset* (*permute_list f xs*)) *y* = *count* (*mset xs*) *y*
by *simp*
qed

lemma *set_permute_list* [*simp*]:
assumes *f permutes* {..<*length xs*}
shows *set* (*permute_list f xs*) = *set xs*
by (*rule mset_eq_setD*[*OF mset_permute_list*]) *fact*

lemma *distinct_permute_list* [*simp*]:
assumes *f permutes* {..<*length xs*}
shows *distinct* (*permute_list f xs*) = *distinct xs*
by (*simp add*: *distinct_count_atmost_1 assms*)

lemma *permute_list_zip*:
assumes *f permutes A A* = {..<*length xs*}
assumes [*simp*]: *length xs* = *length ys*
shows *permute_list f* (*zip xs ys*) = *zip* (*permute_list f xs*) (*permute_list f ys*)
proof −
from *permutes_in_image*[*OF assms*(*1*)] *assms*(*2*) **have** ∗: *f i* < *length ys* ⟷ *i* < *length ys* **for** *i*
by *simp*
have *permute_list f* (*zip xs ys*) = *map* (λi. *zip xs ys* ! *f i*) [*0*..<*length ys*]
by (*simp_all add*: *permute_list_def zip_map_map*)
also have … = *map* (λ(*x*, *y*). (*xs* ! *f x*, *ys* ! *f y*)) (*zip* [*0*..<*length ys*] [*0*..<*length ys*])
by (*intro nth_equalityI*) (*simp_all add*: ∗)
also have … = *zip* (*permute_list f xs*) (*permute_list f ys*)
by (*simp_all add*: *permute_list_def zip_map_map*)
finally show *?thesis* **.**
qed

lemma *map_of_permute*:
assumes σ *permutes fst* ‘ *set xs*
shows *map_of xs* ∘ σ = *map_of* (*map* (λ(*x*,*y*). (*inv* σ *x*, *y*)) *xs*)
(**is** _ = *map_of* (*map ?f* _))
proof
from *assms* **have** *inj* σ *surj* σ

by (*simp_all add*: *permutes_inj permutes_surj*)
  then show $(map\_of\ xs \circ \sigma)\ x = map\_of\ (map\ ?f\ xs)\ x$ **for** x
    by (*induct xs*) (*auto simp*: *inv_f_f surj_f_inv_f*)
qed

lemma *list_all2_permute_list_iff*:
  ‹*list_all2 P* (*permute_list p xs*) (*permute_list p ys*) ⟷ *list_all2 P xs ys*›
  if ‹*p permutes* {..<*length xs*}›
  using *that* **by** (*auto simp add*: *list_all2_iff simp flip*: *permute_list_zip*)

3.18 More lemmas about permutations

lemma *permutes_in_funpow_image*:
  assumes *f permutes S* $x \in S$
  shows $(f \mathbin{\widehat{}\widehat{}} n)\ x \in S$
  using *assms* **by** (*induction n*) (*auto simp*: *permutes_in_image*)

lemma *permutation_self*:
  assumes ‹*permutation p*›
  obtains *n* **where** ‹$n > 0$› ‹$(p \mathbin{\widehat{}\widehat{}} n)\ x = x$›
proof (*cases* ‹$p\ x = x$›)
  case *True*
  with *that* [*of 1*] **show** *thesis* **by** *simp*
next
  case *False*
  from ‹*permutation p*› **have** ‹*inj p*›
    by (*intro permutation_bijective bij_is_inj*)
  moreover from ‹$p\ x \neq x$› **have** ‹$(p \mathbin{\widehat{}\widehat{}} Suc\ n)\ x \neq (p \mathbin{\widehat{}\widehat{}} n)\ x$› **for** *n*
  proof (*induction n arbitrary*: *x*)
    case *0* **then show** *?case* **by** *simp*
  next
    case (*Suc n*)
    have $p\ (p\ x) \neq p\ x$
    proof (*rule notI*)
      assume $p\ (p\ x) = p\ x$
      then show *False* **using** ‹$p\ x \neq x$› ‹*inj p*› **by** (*simp add*: *inj_eq*)
    qed
    have $(p \mathbin{\widehat{}\widehat{}} Suc\ (Suc\ n))\ x = (p \mathbin{\widehat{}\widehat{}} Suc\ n)\ (p\ x)$
      by (*simp add*: *funpow_swap1*)
    also have $\ldots \neq (p \mathbin{\widehat{}\widehat{}} n)\ (p\ x)$
      by (*rule Suc*) *fact*
    also have $(p \mathbin{\widehat{}\widehat{}} n)\ (p\ x) = (p \mathbin{\widehat{}\widehat{}} Suc\ n)\ x$
      by (*simp add*: *funpow_swap1*)
    finally show *?case* **by** *simp*
  qed
  then have $\{y.\ \exists n.\ y = (p \mathbin{\widehat{}\widehat{}} n)\ x\} \subseteq \{x.\ p\ x \neq x\}$
    by *auto*
  then have *finite* $\{y.\ \exists n.\ y = (p \mathbin{\widehat{}\widehat{}} n)\ x\}$
    using *permutation_finite_support*[*OF assms*] **by** (*rule finite_subset*)

ultimately obtain n **where** ‹$n > 0$› ‹$(p$ ^^ $n)$ $x = x$›
  by (*rule funpow_inj_finite*)
  with *that* [*of n*] **show** *thesis* **by** *blast*
qed

The following few lemmas were contributed by Lukas Bulwahn.

lemma *count_image_mset_eq_card_vimage*:
  assumes *finite A*
  shows *count* (*image_mset f* (*mset_set A*)) b = *card* $\{a \in A.\ f\ a = b\}$
  using *assms*
proof (*induct A*)
  case *empty*
  show *?case* **by** *simp*
next
  case (*insert x F*)
  show *?case*
  proof (*cases* $f\ x = b$)
    case *True*
    with *insert.hyps*
    have *count* (*image_mset f* (*mset_set* (*insert x F*))) b = *Suc* (*card* $\{a \in F.\ f$
$a = f\ x\}$)
      by *auto*
    also from *insert.hyps*(*1*,*2*) **have** ... = *card* (*insert x* $\{a \in F.\ f\ a = f\ x\}$)
      by *simp*
    also from ‹$f\ x = b$› **have** *card* (*insert x* $\{a \in F.\ f\ a = f\ x\}$) = *card* $\{a \in$ *insert*
x F. $f\ a = b\}$
      by (*auto intro*: *arg_cong*[**where** *f*=*card*])
    finally show *?thesis*
      using *insert* **by** *auto*
  next
    case *False*
    then have $\{a \in F.\ f\ a = b\} = \{a \in$ *insert x F*. $f\ a = b\}$
      by *auto*
    with *insert False* **show** *?thesis*
      by *simp*
  qed
qed

— Prove *image_mset_eq_implies_permutes* ...
lemma *image_mset_eq_implies_permutes*:
  fixes $f :: 'a \Rightarrow 'b$
  assumes *finite A*
    and *mset_eq*: *image_mset f* (*mset_set A*) = *image_mset f'* (*mset_set A*)
  obtains p **where** p *permutes A* **and** $\forall x \in A.\ f\ x = f'\ (p\ x)$
proof −
  from ‹*finite A*› **have** [*simp*]: *finite* $\{a \in A.\ f\ a = (b::'b)\}$ **for** $f\ b$ **by** *auto*
  have f ‘ $A = f'$ ‘ A
  proof −
    from ‹*finite A*› **have** f ‘ $A = f$ ‘ (*set_mset* (*mset_set A*))

by *simp*
also have ... = f' ‘ *set_mset* (*mset_set* A)
by (*metis mset_eq multiset.set_map*)
also from ‹*finite* A› **have** ... = f' ‘ A
by *simp*
finally show *?thesis* **.**
qed
have $\forall\, b \in (f$ ‘ $A)$. $\exists\, p$. *bij_betw* p $\{a \in A.\ f\ a = b\}$ $\{a \in A.\ f'\ a = b\}$
proof
fix b
from *mset_eq* **have** *count* (*image_mset* f (*mset_set* A)) b = *count* (*image_mset* f' (*mset_set* A)) b
by *simp*
with ‹*finite* A› **have** *card* $\{a \in A.\ f\ a = b\}$ = *card* $\{a \in A.\ f'\ a = b\}$
by (*simp add*: *count_image_mset_eq_card_vimage*)
then show $\exists\, p$. *bij_betw* p $\{a{\in}A.\ f\ a = b\}$ $\{a \in A.\ f'\ a = b\}$
by (*intro finite_same_card_bij*) *simp_all*
qed
then have $\exists\, p. \forall\, b{\in}f$ ‘ A. *bij_betw* (p b) $\{a \in A.\ f\ a = b\}$ $\{a \in A.\ f'\ a = b\}$
by (*rule bchoice*)
then obtain p **where** p: $\forall\, b{\in}f$ ‘ A. *bij_betw* (p b) $\{a \in A.\ f\ a = b\}$ $\{a \in A.\ f'\ a = b\}$ **..**
define p' **where** $p' = (\lambda a.$ *if* $a \in A$ *then* $p\ (f\ a)\ a$ *else* $a)$
have p' *permutes* A
proof (*rule bij_imp_permutes*)
have *disjoint_family_on* ($\lambda i.\ \{a \in A.\ f'\ a = i\}$) ($f$ ‘ A)
by (*auto simp*: *disjoint_family_on_def*)
moreover
have *bij_betw* ($\lambda a.\ p\ (f\ a)\ a$) $\{a \in A.\ f\ a = b\}$ $\{a \in A.\ f'\ a = b\}$ **if** $b \in f$ ‘ A **for** b
using p *that* **by** (*subst bij_betw_cong*[**where** $g{=}p\ b$]) *auto*
ultimately
have *bij_betw* ($\lambda a.\ p\ (f\ a)\ a$) ($\bigcup\, b{\in}f$ ‘ A. $\{a \in A.\ f\ a = b\}$) ($\bigcup\, b{\in}f$ ‘ A. $\{a \in A.\ f'\ a = b\}$)
by (*rule bij_betw_UNION_disjoint*)
moreover have ($\bigcup\, b{\in}f$ ‘ A. $\{a \in A.\ f\ a = b\}$) = A
by *auto*
moreover from ‹f ‘ A = f' ‘ A› **have** ($\bigcup\, b{\in}f$ ‘ A. $\{a \in A.\ f'\ a = b\}$) = A
by *auto*
ultimately show *bij_betw* p' A A
unfolding *p'_def* **by** (*subst bij_betw_cong*[**where** $g{=}(\lambda a.\ p\ (f\ a)\ a)$]) *auto*
next
show $\bigwedge x.\ x \notin A \Longrightarrow p'\ x = x$
by (*simp add*: *p'_def*)
qed
moreover from p **have** $\forall\, x{\in}A.\ f\ x = f'\ (p'\ x)$
unfolding *p'_def* **using** *bij_betwE* **by** *fastforce*
ultimately show *?thesis* **..**
qed

— ... and derive the existing property:

```
lemma mset_eq_permutation:
  fixes xs ys :: 'a list
  assumes mset_eq: mset xs = mset ys
  obtains p where p permutes {..<length ys} permute_list p ys = xs
proof –
  from mset_eq have length_eq: length xs = length ys
    by (rule mset_eq_length)
  have mset_set {..<length ys} = mset [0..<length ys]
    by (rule mset_set_upto_eq_mset_upto)
  with mset_eq length_eq have image_mset (λi. xs ! i) (mset_set {..<length ys})
=
    image_mset (λi. ys ! i) (mset_set {..<length ys})
    by (metis map_nth mset_map)
  from image_mset_eq_implies_permutes[OF _ this]
  obtain p where p: p permutes {..<length ys} and ∀ i∈{..<length ys}. xs ! i =
ys ! (p i)
    by auto
  with length_eq have permute_list p ys = xs
    by (auto intro!: nth_equalityI simp: permute_list_nth)
  with p show thesis ..
qed

lemma permutes_natset_le:
  fixes S :: 'a::wellorder set
  assumes p permutes S
    and ∀ i ∈ S. p i ≤ i
  shows p = id
proof –
  have p n = n for n
    using assms
  proof (induct n arbitrary: S rule: less_induct)
    case (less n)
    show ?case
    proof (cases n ∈ S)
      case False
      with less(2) show ?thesis
        unfolding permutes_def by metis
    next
      case True
      with less(3) have p n < n ∨ p n = n
        by auto
      then show ?thesis
      proof
        assume p n < n
        with less have p (p n) = p n
          by metis
        with permutes_inj[OF less(2)] have p n = n
```

```
          unfolding inj_def by blast
        with ‹p n < n› have False
          by simp
        then show ?thesis ..
      qed
    qed
  qed
  then show ?thesis by (auto simp: fun_eq_iff)
qed

lemma permutes_natset_ge:
  fixes S :: 'a::wellorder set
  assumes p: p permutes S
    and le: ∀ i ∈ S. p i ≥ i
  shows p = id
proof –
  have i ≥ inv p i if i ∈ S for i
  proof –
    from that permutes_in_image[OF permutes_inv[OF p]] have inv p i ∈ S
      by simp
    with le have p (inv p i) ≥ inv p i
      by blast
    with permutes_inverses[OF p] show ?thesis
      by simp
  qed
  then have ∀ i∈S. inv p i ≤ i
    by blast
  from permutes_natset_le[OF permutes_inv[OF p] this] have inv p = inv id
    by simp
  then show ?thesis
    using p permutes_inv_inv by fastforce
qed

lemma image_inverse_permutations: {inv p |p. p permutes S} = {p. p permutes
S}
  using permutes_inv permutes_inv_inv by force

lemma image_compose_permutations_left:
  assumes q permutes S
  shows {q ∘ p |p. p permutes S} = {p. p permutes S}
proof –
  have ⋀p. p permutes S ⟹ q ∘ p permutes S
    by (simp add: assms permutes_compose)
  moreover have ⋀x. x permutes S ⟹ ∃ p. x = q ∘ p ∧ p permutes S
   by (metis assms id_comp o_assoc permutes_compose permutes_inv permutes_inv_o(1))
  ultimately show ?thesis
    by auto
qed
```

lemma *image_compose_permutations_right*:
  assumes *q permutes S*
  shows $\{p \circ q \mid p.\ p\ permutes\ S\} = \{p\ .\ p\ permutes\ S\}$
  by (*metis* (*no_types*, *opaque_lifting*) *assms comp_id fun.map_comp permutes_compose permutes_inv permutes_inv_o*(*2*))

lemma *permutes_in_seg*: $p\ permutes\ \{1\ ..n\} \Longrightarrow i \in \{1..n\} \Longrightarrow 1 \leq p\ i \wedge p\ i \leq n$
  by (*simp add*: *permutes_def*) *metis*

lemma *sum_permutations_inverse*: $sum\ f\ \{p.\ p\ permutes\ S\} = sum\ (\lambda p.\ f(inv\ p))\ \{p.\ p\ permutes\ S\}$
  (**is** *?lhs* = *?rhs*)
proof −
  let $?S = \{p\ .\ p\ permutes\ S\}$
  have ∗: *inj_on inv ?S*
  proof (*auto simp add*: *inj_on_def*)
    fix *q r*
    assume *q*: *q permutes S*
      and *r*: *r permutes S*
      and *qr*: *inv q* = *inv r*
    then have *inv* (*inv q*) = *inv* (*inv r*)
      by *simp*
    with *permutes_inv_inv*[*OF q*] *permutes_inv_inv*[*OF r*] **show** *q* = *r*
      by *metis*
  qed
  have ∗∗: *inv* ‘ *?S* = *?S*
    using *image_inverse_permutations* **by** *blast*
  have ∗∗∗: $?rhs = sum\ (f \circ inv)\ ?S$
    by (*simp add*: *o_def*)
  from *sum.reindex*[*OF* ∗, *of f*] **show** *?thesis*
    by (*simp only*: ∗∗ ∗∗∗)
qed

lemma *setum_permutations_compose_left*:
  assumes *q*: *q permutes S*
  shows $sum\ f\ \{p.\ p\ permutes\ S\} = sum\ (\lambda p.\ f(q \circ p))\ \{p.\ p\ permutes\ S\}$
  (**is** *?lhs* = *?rhs*)
proof −
  let $?S = \{p.\ p\ permutes\ S\}$
  have ∗: $?rhs = sum\ (f \circ ((\circ)\ q))\ ?S$
    by (*simp add*: *o_def*)
  have ∗∗: $inj\_on\ ((\circ)\ q)\ ?S$
  proof (*auto simp add*: *inj_on_def*)
    fix *p r*
    assume *p permutes S*
      and *r*: *r permutes S*
      and *rp*: $q \circ p = q \circ r$
    then have $inv\ q \circ q \circ p = inv\ q \circ q \circ r$

by (*simp add*: *comp_assoc*)
with *permutes_inj*[*OF q*, *unfolded inj_iff*] **show** $p = r$
by *simp*
qed
have $((\circ)\ q)$ ` *?S* = *?S*
using *image_compose_permutations_left*[*OF q*] **by** *auto*
with $*$ *sum.reindex*[*OF* $**$, *of f*] **show** *?thesis*
by (*simp only*:)
qed

lemma *sum_permutations_compose_right*:
assumes *q*: *q permutes S*
shows *sum f* {*p*. *p permutes S*} = *sum* (λp. $f(p \circ q)$) {*p*. *p permutes S*}
(**is** *?lhs* = *?rhs*)
proof −
let *?S* = {*p*. *p permutes S*}
have $*$: *?rhs* = *sum* ($f \circ (\lambda p.\ p \circ q)$) *?S*
by (*simp add*: *o_def*)
have $**$: *inj_on* ($\lambda p.\ p \circ q$) *?S*
proof (*auto simp add*: *inj_on_def*)
fix *p r*
assume *p permutes S*
and *r*: *r permutes S*
and *rp*: $p \circ q = r \circ q$
then have $p \circ (q \circ inv\ q) = r \circ (q \circ inv\ q)$
by (*simp add*: *o_assoc*)
with *permutes_surj*[*OF q*, *unfolded surj_iff*] **show** $p = r$
by *simp*
qed
from *image_compose_permutations_right*[*OF q*] **have** ($\lambda p.\ p \circ q$) ` *?S* = *?S*
by *auto*
with $*$ *sum.reindex*[*OF* $**$, *of f*] **show** *?thesis*
by (*simp only*:)
qed

lemma *inv_inj_on_permutes*:
‹*inj_on inv* {*p*. *p permutes S*}›
proof (*intro inj_onI*, *unfold mem_Collect_eq*)
fix *p q*
assume *p*: *p permutes S* **and** *q*: *q permutes S* **and** *eq*: *inv p* = *inv q*
have *inv* (*inv p*) = *inv* (*inv q*) **using** *eq* **by** *simp*
thus $p = q$
using *inv_inv_eq*[*OF permutes_bij*] *p q* **by** *metis*
qed

lemma *permutes_pair_eq*:
‹{(*p s*, *s*) |*s*. $s \in S$} = {(*s*, *inv p s*) |*s*. $s \in S$}› (**is** ‹*?L* = *?R*›) **if** ‹*p permutes S*›
proof
show *?L* $\subseteq$ *?R*

```
  proof
    fix x assume x ∈ ?L
    then obtain s where x: x = (p s, s) and s: s ∈ S by auto
    note x
    also have (p s, s) = (p s, Hilbert_Choice.inv p (p s))
      using permutes_inj [OF that] inv_f_f by auto
    also have ... ∈ ?R using s permutes_in_image[OF that] by auto
    finally show x ∈ ?R.
  qed
  show ?R ⊆ ?L
  proof
    fix x assume x ∈ ?R
    then obtain s
      where x: x = (s, Hilbert_Choice.inv p s) (is _ = (s, ?ips))
        and s: s ∈ S by auto
    note x
    also have (s, ?ips) = (p ?ips, ?ips)
      using inv_f_f[OF permutes_inj[OF permutes_inv[OF that]]]
      using inv_inv_eq[OF permutes_bij[OF that]] by auto
    also have ... ∈ ?L
      using s permutes_in_image[OF permutes_inv[OF that]] by auto
    finally show x ∈ ?L.
  qed
qed

context
  fixes p and n i :: nat
  assumes p: ‹p permutes {0..<n}› and i: ‹i < n›
begin

lemma permutes_nat_less:
  ‹p i < n›
proof −
  have ‹?thesis ⟷ p i ∈ {0..<n}›
    by simp
  also from p have ‹p i ∈ {0..<n} ⟷ i ∈ {0..<n}›
    by (rule permutes_in_image)
  finally show ?thesis
    using i by simp
qed

lemma permutes_nat_inv_less:
  ‹inv p i < n›
proof −
  from p have ‹inv p permutes {0..<n}›
    by (rule permutes_inv)
  then show ?thesis
    using i by (rule Permutations.permutes_nat_less)
qed
```

end

context *comm_monoid_set*
begin

lemma *permutes_inv*:
  ‹F ($\lambda s.$ g (p s) s) S = F ($\lambda s.$ g s (inv p s)) S› (**is** ‹?l = ?r›)
  if ‹p *permutes* S›
proof −
  let ?g = $\lambda(x, y).$ g x y
  let ?ps = $\lambda s.$ (p s, s)
  let ?ips = $\lambda s.$ (s, inv p s)
  have *inj1*: *inj_on ?ps S* **by** (*rule inj_onI*) *auto*
  have *inj2*: *inj_on ?ips S* **by** (*rule inj_onI*) *auto*
  have ?l = F ?g (?ps ‘ S)
    using *reindex* [*OF inj1*, *of ?g*] **by** *simp*
  also have ?ps ‘ S = $\{(p\ s, s) \mid s.\ s \in S\}$ **by** *auto*
  also have ... = $\{(s, inv\ p\ s) \mid s.\ s \in S\}$
    unfolding *permutes_pair_eq* [*OF that*] **by** *simp*
  also have ... = ?ips ‘ S **by** *auto*
  also have F ?g ... = ?r
    using *reindex* [*OF inj2*, *of ?g*] **by** *simp*
  finally show *?thesis*.
qed

end

3.19 Sum over a set of permutations (could generalize to iteration)

lemma *sum_over_permutations_insert*:
  assumes *fS*: *finite S*
    and *aS*: $a \notin S$
  shows *sum f* $\{p.$ p *permutes* (*insert a S*)$\}$ =
    sum ($\lambda b.$ *sum* ($\lambda q.$ f (*transpose a b* $\circ$ q)) $\{p.$ p *permutes* $S\}$) (*insert a S*)
proof −
  have $*$: $\bigwedge f\ a\ b.\ (\lambda(b, p).\ f\ (transpose\ a\ b \circ p)) = f \circ (\lambda(b,p).\ transpose\ a\ b \circ p)$
    by (*simp add*: *fun_eq_iff*)
  have $**$: $\bigwedge P\ Q.\ \{(a, b).\ a \in P \land b \in Q\} = P \times Q$
    by *blast*
  show *?thesis*
    unfolding $*$ $**$ *sum.cartesian_product permutes_insert*
  proof (*rule sum.reindex*)
    let ?f = ($\lambda(b, y).$ *transpose a b* $\circ$ y)
    let ?P = $\{p.$ p *permutes* $S\}$
    {
      fix b c p q
      assume b: $b \in$ *insert a S*

assume *c*: $c \in$ *insert a S*
assume *p*: *p permutes S*
assume *q*: *q permutes S*
assume *eq*: *transpose a b* $\circ$ *p* = *transpose a c* $\circ$ *q*
from *p q aS* **have** *pa*: *p a* = *a* **and** *qa*: *q a* = *a*
unfolding *permutes_def* **by** *metis+*
from *eq* **have** (*transpose a b* $\circ$ *p*) *a* = (*transpose a c* $\circ$ *q*) *a*
by *simp*
then have *bc*: *b* = *c*
by (*simp add*: *permutes_def pa qa o_def fun_upd_def id_def*
cong del: *if_weak_cong split*: *if_split_asm*)
from *eq*[*unfolded bc*] **have** (λ*p*. *transpose a c* $\circ$ *p*) (*transpose a c* $\circ$ *p*) =
(λ*p*. *transpose a c* $\circ$ *p*) (*transpose a c* $\circ$ *q*) **by** *simp*
then have *p* = *q*
unfolding *o_assoc swap_id_idempotent* **by** *simp*
with *bc* **have** *b* = *c* $\wedge$ *p* = *q*
by *blast*
}
then show *inj_on ?f* (*insert a S* $\times$ *?P*)
unfolding *inj_on_def* **by** *clarify metis*
qed
qed

3.20 Constructing permutations from association lists

definition *list_permutes* :: ('*a* $\times$ '*a*) *list* $\Rightarrow$ '*a set* $\Rightarrow$ *bool*
where *list_permutes xs A* $\longleftrightarrow$
set (*map fst xs*) $\subseteq$ *A* $\wedge$
set (*map snd xs*) = *set* (*map fst xs*) $\wedge$
distinct (*map fst xs*) $\wedge$
distinct (*map snd xs*)

lemma *list_permutesI* [*simp*]:
assumes *set* (*map fst xs*) $\subseteq$ *A set* (*map snd xs*) = *set* (*map fst xs*) *distinct* (*map fst xs*)
shows *list_permutes xs A*
proof −
from *assms*(*2*,*3*) **have** *distinct* (*map snd xs*)
by (*intro card_distinct*) (*simp_all add*: *distinct_card del*: *set_map*)
with *assms* **show** *?thesis*
by (*simp add*: *list_permutes_def*)
qed

definition *permutation_of_list* :: ('*a* $\times$ '*a*) *list* $\Rightarrow$ '*a* $\Rightarrow$ '*a*
where *permutation_of_list xs x* = (*case map_of xs x of None* $\Rightarrow$ *x* | *Some y* $\Rightarrow$ *y*)

lemma *permutation_of_list_Cons*:
permutation_of_list ((*x*, *y*) # *xs*) *x'* = (*if x* = *x' then y else permutation_of_list*

```
xs x')
  by (simp add: permutation_of_list_def)

fun inverse_permutation_of_list :: ('a × 'a) list ⇒ 'a ⇒ 'a
  where
    inverse_permutation_of_list [] x = x
  | inverse_permutation_of_list ((y, x') # xs) x =
      (if x = x' then y else inverse_permutation_of_list xs x)

declare inverse_permutation_of_list.simps [simp del]

lemma inj_on_map_of:
  assumes distinct (map snd xs)
  shows inj_on (map_of xs) (set (map fst xs))
proof (rule inj_onI)
  fix x y
  assume xy: x ∈ set (map fst xs) y ∈ set (map fst xs)
  assume eq: map_of xs x = map_of xs y
  from xy obtain x' y' where x'y': map_of xs x = Some x' map_of xs y = Some
y'
   by (cases map_of xs x; cases map_of xs y) (simp_all add: map_of_eq_None_iff)
  moreover from x'y' have ∗: (x, x') ∈ set xs (y, y') ∈ set xs
    by (force dest: map_of_SomeD)+
  moreover from ∗ eq x'y' have x' = y'
    by simp
  ultimately show x = y
    using assms by (force simp: distinct_map dest: inj_onD[of _ _ (x,x') (y,y')])
qed

lemma inj_on_the: None ∉ A ⟹ inj_on the A
  by (auto simp: inj_on_def option.the_def split: option.splits)

lemma inj_on_map_of':
  assumes distinct (map snd xs)
  shows inj_on (the ∘ map_of xs) (set (map fst xs))
  by (intro comp_inj_on inj_on_map_of assms inj_on_the)
    (force simp: eq_commute[of None] map_of_eq_None_iff)

lemma image_map_of:
  assumes distinct (map fst xs)
  shows map_of xs ‘ set (map fst xs) = Some ‘ set (map snd xs)
  using assms by (auto simp: rev_image_eqI)

lemma the_Some_image [simp]: the ‘ Some ‘ A = A
  by (subst image_image) simp

lemma image_map_of':
  assumes distinct (map fst xs)
  shows (the ∘ map_of xs) ‘ set (map fst xs) = set (map snd xs)
```

by (*simp only*: *image_comp* [*symmetric*] *image_map_of assms the_Some_image*)

lemma *permutation_of_list_permutes* [*simp*]:
assumes *list_permutes xs A*
shows *permutation_of_list xs permutes A*
(**is** *?f permutes _*)
proof (*rule permutes_subset*[*OF bij_imp_permutes*])
from *assms* **show** *set* (*map fst xs*) $\subseteq$ *A*
by (*simp add*: *list_permutes_def*)
from *assms* **have** *inj_on* (*the* $\circ$ *map_of xs*) (*set* (*map fst xs*)) (**is** *?P*)
by (*intro inj_on_map_of'*) (*simp_all add*: *list_permutes_def*)
also have *?P* $\longleftrightarrow$ *inj_on ?f* (*set* (*map fst xs*))
by (*intro inj_on_cong*)
(*auto simp*: *permutation_of_list_def map_of_eq_None_iff split*: *option.splits*)
finally have *bij_betw ?f* (*set* (*map fst xs*)) (*?f* ` *set* (*map fst xs*))
by (*rule inj_on_imp_bij_betw*)
also from *assms* **have** *?f* ` *set* (*map fst xs*) = (*the* $\circ$ *map_of xs*) ` *set* (*map fst xs*)
by (*intro image_cong refl*)
(*auto simp*: *permutation_of_list_def map_of_eq_None_iff split*: *option.splits*)
also from *assms* **have** ... = *set* (*map fst xs*)
by (*subst image_map_of'*) (*simp_all add*: *list_permutes_def*)
finally show *bij_betw ?f* (*set* (*map fst xs*)) (*set* (*map fst xs*)) **.**
qed (*force simp*: *permutation_of_list_def dest*!: *map_of_SomeD split*: *option.splits*)+

lemma *eval_permutation_of_list* [*simp*]:
permutation_of_list [] *x* = *x*
x = *x'* $\Longrightarrow$ *permutation_of_list* ((*x'*,*y*)#*xs*) *x* = *y*
x $\neq$ *x'* $\Longrightarrow$ *permutation_of_list* ((*x'*,*y'*)#*xs*) *x* = *permutation_of_list xs x*
by (*simp_all add*: *permutation_of_list_def*)

lemma *eval_inverse_permutation_of_list* [*simp*]:
inverse_permutation_of_list [] *x* = *x*
x = *x'* $\Longrightarrow$ *inverse_permutation_of_list* ((*y*,*x'*)#*xs*) *x* = *y*
x $\neq$ *x'* $\Longrightarrow$ *inverse_permutation_of_list* ((*y'*,*x'*)#*xs*) *x* = *inverse_permutation_of_list xs x*
by (*simp_all add*: *inverse_permutation_of_list.simps*)

lemma *permutation_of_list_id*: *x* $\notin$ *set* (*map fst xs*) $\Longrightarrow$ *permutation_of_list xs x* = *x*
by (*induct xs*) (*auto simp*: *permutation_of_list_Cons*)

lemma *permutation_of_list_unique'*:
distinct (*map fst xs*) $\Longrightarrow$ (*x*, *y*) $\in$ *set xs* $\Longrightarrow$ *permutation_of_list xs x* = *y*
by (*induct xs*) (*force simp*: *permutation_of_list_Cons*)+

lemma *permutation_of_list_unique*:
list_permutes xs A $\Longrightarrow$ (*x*, *y*) $\in$ *set xs* $\Longrightarrow$ *permutation_of_list xs x* = *y*
by (*intro permutation_of_list_unique'*) (*simp_all add*: *list_permutes_def*)

lemma *inverse_permutation_of_list_id*:
$x \notin$ *set* (*map snd xs*) $\Longrightarrow$ *inverse_permutation_of_list xs x* $= x$
by (*induct xs*) *auto*

lemma *inverse_permutation_of_list_unique'*:
distinct (*map snd xs*) $\Longrightarrow$ $(x, y) \in$ *set xs* $\Longrightarrow$ *inverse_permutation_of_list xs y* $= x$
by (*induct xs*) (*force simp*: *inverse_permutation_of_list.simps*(*2*))+

lemma *inverse_permutation_of_list_unique*:
list_permutes xs A $\Longrightarrow$ $(x,y) \in$ *set xs* $\Longrightarrow$ *inverse_permutation_of_list xs y* $= x$
by (*intro inverse_permutation_of_list_unique'*) (*simp_all add*: *list_permutes_def*)

lemma *inverse_permutation_of_list_correct*:
fixes A :: *'a set*
assumes *list_permutes xs A*
shows *inverse_permutation_of_list xs* = *inv* (*permutation_of_list xs*)
proof (*rule ext*, *rule sym*, *subst permutes_inv_eq*)
from *assms* **show** *permutation_of_list xs permutes A*
by *simp*
show *permutation_of_list xs* (*inverse_permutation_of_list xs x*) $= x$ **for** x
proof (*cases* $x \in$ *set* (*map snd xs*))
case *True*
then obtain y **where** $(y, x) \in$ *set xs* **by** *auto*
with *assms* **show** *?thesis*
by (*simp add*: *inverse_permutation_of_list_unique permutation_of_list_unique*)
next
case *False*
with *assms* **show** *?thesis*
by (*auto simp*: *list_permutes_def inverse_permutation_of_list_id permutation_of_list_id*)
qed
qed

end

4 Permuted Lists

theory *List_Permutation*
imports *Permutations*
begin

Note that multisets already provide the notion of permutated list and hence this theory mostly echoes material already logically present in theory *Permutations*; it should be seldom needed.

4.1 An existing notion

abbreviation (input) perm :: ‹'a list ⇒ 'a list ⇒ bool› (**infixr** ‹<~~>› 50)
  where ‹xs <~~> ys ≡ mset xs = mset ys›

4.2 Nontrivial conclusions

proposition perm_swap:
  ‹xs[i := xs ! j, j := xs ! i] <~~> xs›
  if ‹i < length xs› ‹j < length xs›
  using that **by** (simp add: mset_swap)

proposition mset_le_perm_append: mset xs ⊆# mset ys ⟷ (∃ zs. xs @ zs <~~> ys)
  by (auto simp add: mset_subset_eq_exists_conv ex_mset dest: sym)

proposition perm_set_eq: xs <~~> ys ⟹ set xs = set ys
  by (rule mset_eq_setD) simp

proposition perm_distinct_iff: xs <~~> ys ⟹ distinct xs ⟷ distinct ys
  by (rule mset_eq_imp_distinct_iff) simp

theorem eq_set_perm_remdups: set xs = set ys ⟹ remdups xs <~~> remdups ys
  by (simp add: set_eq_iff_mset_remdups_eq)

proposition perm_remdups_iff_eq_set: remdups x <~~> remdups y ⟷ set x = set y
  by (simp add: set_eq_iff_mset_remdups_eq)

theorem permutation_Ex_bij:
  assumes xs <~~> ys
  shows ∃ f. bij_betw f {..<length xs} {..<length ys} ∧ (∀ i<length xs. xs ! i = ys ! (f i))
proof −
  from assms **have** ‹mset xs = mset ys› ‹length xs = length ys›
    by (auto simp add: dest: mset_eq_length)
  from ‹mset xs = mset ys› **obtain** p **where** ‹p permutes {..<length ys}› ‹permute_list p ys = xs›
    by (rule mset_eq_permutation)
  then have ‹bij_betw p {..<length xs} {..<length ys}›
    by (simp add: ‹length xs = length ys› permutes_imp_bij)
  moreover have ‹∀ i<length xs. xs ! i = ys ! (p i)›
    using ‹permute_list p ys = xs› ‹length xs = length ys› ‹p permutes {..<length ys}› permute_list_nth
    by auto
  ultimately show ?thesis
    by blast
qed

proposition *perm_finite*: *finite* {*B*. *B* <~~> *A*}
  using *mset_eq_finite* **by** *auto*

4.3 Trivial conclusions:

proposition *perm_empty_imp*: [] <~~> *ys* ⟹ *ys* = []
  by *simp*

This more general theorem is easier to understand!

proposition *perm_length*: *xs* <~~> *ys* ⟹ *length xs* = *length ys*
  by (*rule mset_eq_length*) *simp*

proposition *perm_sym*: *xs* <~~> *ys* ⟹ *ys* <~~> *xs*
  by *simp*

We can insert the head anywhere in the list.

proposition *perm_append_Cons*: *a* # *xs* @ *ys* <~~> *xs* @ *a* # *ys*
  by *simp*

proposition *perm_append_swap*: *xs* @ *ys* <~~> *ys* @ *xs*
  by *simp*

proposition *perm_append_single*: *a* # *xs* <~~> *xs* @ [*a*]
  by *simp*

proposition *perm_rev*: *rev xs* <~~> *xs*
  by *simp*

proposition *perm_append1*: *xs* <~~> *ys* ⟹ *l* @ *xs* <~~> *l* @ *ys*
  by *simp*

proposition *perm_append2*: *xs* <~~> *ys* ⟹ *xs* @ *l* <~~> *ys* @ *l*
  by *simp*

proposition *perm_empty* [*iff*]: [] <~~> *xs* ⟷ *xs* = []
  by *simp*

proposition *perm_empty2* [*iff*]: *xs* <~~> [] ⟷ *xs* = []
  by *simp*

proposition *perm_sing_imp*: *ys* <~~> *xs* ⟹ *xs* = [*y*] ⟹ *ys* = [*y*]
  by *simp*

proposition *perm_sing_eq* [*iff*]: *ys* <~~> [*y*] ⟷ *ys* = [*y*]
  by *simp*

proposition *perm_sing_eq2* [*iff*]: [*y*] <~~> *ys* ⟷ *ys* = [*y*]
  by *simp*

proposition *perm_remove*: *x* ∈ *set ys* ⟹ *ys* <~~> *x* # *remove1 x ys*
  by *simp*

Congruence rule

proposition *perm_remove_perm*: *xs* <~~> *ys* ⟹ *remove1 z xs* <~~> *remove1 z ys*
  by *simp*

proposition *remove_hd* [*simp*]: *remove1 z* (*z* # *xs*) = *xs*
  by *simp*

proposition *cons_perm_imp_perm*: *z* # *xs* <~~> *z* # *ys* ⟹ *xs* <~~> *ys*
  by *simp*

proposition *cons_perm_eq* [*simp*]: *z*#*xs* <~~> *z*#*ys* ⟷ *xs* <~~> *ys*
  by *simp*

proposition *append_perm_imp_perm*: *zs* @ *xs* <~~> *zs* @ *ys* ⟹ *xs* <~~> *ys*
  by *simp*

proposition *perm_append1_eq* [*iff*]: *zs* @ *xs* <~~> *zs* @ *ys* ⟷ *xs* <~~> *ys*
  by *simp*

proposition *perm_append2_eq* [*iff*]: *xs* @ *zs* <~~> *ys* @ *zs* ⟷ *xs* <~~> *ys*
  by *simp*

end

5 Permutations of a Multiset

theory *Multiset_Permutations*
imports
  Complex_Main
  Permutations
begin

lemma *mset_tl*: *xs* ≠ [] ⟹ *mset* (*tl xs*) = *mset xs* − {#*hd xs*#}
  by (*cases xs*) *simp_all*

lemma *mset_set_image_inj*:
  assumes *inj_on f A*
  shows *mset_set* (*f* ‘ *A*) = *image_mset f* (*mset_set A*)
proof (*cases finite A*)
  case *True*
  from *this* **and** *assms* **show** *?thesis* **by** (*induction A*) *auto*
qed (*insert assms*, *simp add*: *finite_image_iff*)

lemma *multiset_remove_induct* [*case_names empty remove*]:
assumes P {#} $\bigwedge A.\ A \neq$ {#} $\Longrightarrow (\bigwedge x.\ x \in\#\ A \Longrightarrow P\ (A - \{\#x\#\})) \Longrightarrow P$ A
shows $P\ A$
proof (*induction A rule*: *full_multiset_induct*)
case (*less A*)
hence *IH*: $P\ B$ **if** $B \subset\#\ A$ **for** B **using** *that* **by** *blast*
show *?case*
proof (*cases* A = {#})
case *True*
thus *?thesis* **by** (*simp add*: *assms*)
next
case *False*
hence $P\ (A - \{\#x\#\})$ **if** $x \in\#\ A$ **for** x
using *that* **by** (*intro IH*) (*simp add*: *mset_subset_diff_self*)
from *False* **and** *this* **show** $P\ A$ **by** (*rule assms*)
qed
qed

lemma *map_list_bind*: *map g* (*List.bind xs f*) = *List.bind xs* (*map g* ∘ *f*)
by (*simp add*: *List.bind_def map_concat*)

lemma *mset_eq_mset_set_imp_distinct*:
finite A $\Longrightarrow$ *mset_set A* = *mset xs* $\Longrightarrow$ *distinct xs*
proof (*induction xs arbitrary*: *A*)
case (*Cons x xs A*)
from *Cons.prems*(*2*) **have** $x \in\#$ *mset_set A* **by** *simp*
with *Cons.prems*(*1*) **have** [*simp*]: $x \in A$ **by** *simp*
from *Cons.prems* **have** $x \notin\#$ *mset_set* $(A - \{x\})$ **by** *simp*
also from *Cons.prems* **have** *mset_set* $(A - \{x\})$ = *mset_set A* − {#*x*#}
by (*subst mset_set_Diff*) *simp_all*
also have *mset_set A* = *mset* (*x*#*xs*) **by** (*simp add*: *Cons.prems*)
also have . . . − {#*x*#} = *mset xs* **by** *simp*
finally have [*simp*]: $x \notin$ *set xs* **by** (*simp add*: *in_multiset_in_set*)
from *Cons.prems* **show** *?case* **by** (*auto intro*!: *Cons.IH*[*of A* − {*x*}] *simp*: *mset_set_Diff*)
qed *simp_all*

5.1 Permutations of a multiset

definition *permutations_of_multiset* :: *'a multiset* ⇒ *'a list set* **where**
permutations_of_multiset A = {*xs*. *mset xs* = *A*}

lemma *permutations_of_multisetI*: *mset xs* = *A* $\Longrightarrow$ *xs* ∈ *permutations_of_multiset* *A*
by (*simp add*: *permutations_of_multiset_def*)

lemma *permutations_of_multisetD*: *xs* ∈ *permutations_of_multiset A* $\Longrightarrow$ *mset* *xs* = *A*

by (*simp add*: *permutations_of_multiset_def*)

lemma *permutations_of_multiset_Cons_iff*:
x # *xs* ∈ *permutations_of_multiset* $A \longleftrightarrow x \in\# A \wedge$ *xs* ∈ *permutations_of_multiset* $(A - \{\#x\#\})$
by (*auto simp*: *permutations_of_multiset_def*)

lemma *permutations_of_multiset_empty* [*simp*]: *permutations_of_multiset* {#} = {[]}
unfolding *permutations_of_multiset_def* **by** *simp*

lemma *permutations_of_multiset_nonempty*:
assumes *nonempty*: $A \neq \{\#\}$
shows *permutations_of_multiset* A =
($\bigcup x \in$*set_mset* A. ((#) x) ‘ *permutations_of_multiset* $(A - \{\#x\#\})$)
(**is** _ = *?rhs*)
proof *safe*
fix *xs* **assume** *xs* ∈ *permutations_of_multiset* A
hence *mset_xs*: *mset xs* = A **by** (*simp add*: *permutations_of_multiset_def*)
hence *xs* ≠ [] **by** (*auto simp*: *nonempty*)
then obtain x *xs′* **where** *xs*: *xs* = x # *xs′* **by** (*cases xs*) *simp_all*
with *mset_xs* **have** $x \in$ *set_mset* A *xs′* ∈ *permutations_of_multiset* $(A - \{\#x\#\})$
by (*auto simp*: *permutations_of_multiset_def*)
with *xs* **show** *xs* ∈ *?rhs* **by** *auto*
qed (*auto simp*: *permutations_of_multiset_def*)

lemma *permutations_of_multiset_singleton* [*simp*]: *permutations_of_multiset* {#x#} = {[x]}
by (*simp add*: *permutations_of_multiset_nonempty*)

lemma *permutations_of_multiset_doubleton*:
permutations_of_multiset {#x,y#} = {[x,y], [y,x]}
by (*simp add*: *permutations_of_multiset_nonempty insert_commute*)

lemma *rev_permutations_of_multiset* [*simp*]:
rev ‘ *permutations_of_multiset* A = *permutations_of_multiset* A
proof
have *rev* ‘ *rev* ‘ *permutations_of_multiset* $A \subseteq$ *rev* ‘ *permutations_of_multiset* A
unfolding *permutations_of_multiset_def* **by** *auto*
also have *rev* ‘ *rev* ‘ *permutations_of_multiset* A = *permutations_of_multiset* A
by (*simp add*: *image_image*)
finally show *permutations_of_multiset* $A \subseteq$ *rev* ‘ *permutations_of_multiset* A
.
next
show *rev* ‘ *permutations_of_multiset* $A \subseteq$ *permutations_of_multiset* A
unfolding *permutations_of_multiset_def* **by** *auto*

qed

lemma *length_finite_permutations_of_multiset*:
$xs \in$ *permutations_of_multiset* $A \Longrightarrow$ *length* $xs =$ *size* A
by (*auto simp*: *permutations_of_multiset_def*)

lemma *permutations_of_multiset_lists*: *permutations_of_multiset* $A \subseteq$ *lists* (*set_mset* A)
by (*auto simp*: *permutations_of_multiset_def*)

lemma *finite_permutations_of_multiset* [*simp*]: *finite* (*permutations_of_multiset* A)
proof (*rule finite_subset*)
show *permutations_of_multiset* $A \subseteq \{xs.$ *set* $xs \subseteq$ *set_mset* $A \wedge$ *length* $xs =$ *size* $A\}$
by (*auto simp*: *permutations_of_multiset_def*)
show *finite* $\{xs.$ *set* $xs \subseteq$ *set_mset* $A \wedge$ *length* $xs =$ *size* $A\}$
by (*rule finite_lists_length_eq*) *simp_all*
qed

lemma *permutations_of_multiset_not_empty* [*simp*]: *permutations_of_multiset* $A \neq \{\}$
proof −
from *ex_mset*[*of* A] **obtain** xs **where** *mset* $xs = A$ **..**
thus *?thesis* **by** (*auto simp*: *permutations_of_multiset_def*)
qed

lemma *permutations_of_multiset_image*:
permutations_of_multiset (*image_mset* f A) = *map* f ‘ *permutations_of_multiset* A
proof *safe*
fix xs **assume** A: $xs \in$ *permutations_of_multiset* (*image_mset* f A)
from *ex_mset*[*of* A] **obtain** ys **where** ys: *mset* $ys = A$ **..**
with A **have** *mset* $xs =$ *mset* (*map* f ys)
by (*simp add*: *permutations_of_multiset_def*)
then obtain σ **where** σ: σ *permutes* $\{..<$*length* (*map* f ys)$\}$ *permute_list* σ (*map* f ys) $= xs$
by (*rule mset_eq_permutation*)
with ys **have** $xs =$ *map* f (*permute_list* σ ys)
by (*simp add*: *permute_list_map*)
moreover from σ ys **have** *permute_list* σ $ys \in$ *permutations_of_multiset* A
by (*simp add*: *permutations_of_multiset_def*)
ultimately show $xs \in$ *map* f ‘ *permutations_of_multiset* A **by** *blast*
qed (*auto simp*: *permutations_of_multiset_def*)

5.2 Cardinality of permutations

In this section, we prove some basic facts about the number of permutations of a multiset.

context
begin

private lemma *multiset_prod_fact_insert*:
$(\prod y{\in}set\_mset\ (A+\{\#x\#\}).\ fact\ (count\ (A+\{\#x\#\})\ y)) =$
$(count\ A\ x + 1) * (\prod y{\in}set\_mset\ A.\ fact\ (count\ A\ y))$
proof −
have $(\prod y{\in}set\_mset\ (A+\{\#x\#\}).\ fact\ (count\ (A+\{\#x\#\})\ y)) =$
$(\prod y{\in}set\_mset\ (A+\{\#x\#\}).\ (if\ y = x\ then\ count\ A\ x + 1\ else\ 1) * fact\ (count\ A\ y))$
by (*intro prod.cong*) *simp_all*
also have ... $= (count\ A\ x + 1) * (\prod y{\in}set\_mset\ (A+\{\#x\#\}).\ fact\ (count\ A\ y))$
by (*simp add*: *prod.distrib*)
also have $(\prod y{\in}set\_mset\ (A+\{\#x\#\}).\ fact\ (count\ A\ y)) = (\prod y{\in}set\_mset\ A.\ fact\ (count\ A\ y))$
by (*intro prod.mono_neutral_right*) (*auto simp*: *not_in_iff*)
finally show *?thesis* **.**
qed

private lemma *multiset_prod_fact_remove*:
$x \in\#\ A \Longrightarrow (\prod y{\in}set\_mset\ A.\ fact\ (count\ A\ y)) =$
$count\ A\ x * (\prod y{\in}set\_mset\ (A-\{\#x\#\}).\ fact\ (count\ (A-\{\#x\#\})\ y))$
using *multiset_prod_fact_insert*[*of* $A - \{\#x\#\}$ *x*] **by** *simp*

lemma *card_permutations_of_multiset_aux*:
$card\ (permutations\_of\_multiset\ A) * (\prod x{\in}set\_mset\ A.\ fact\ (count\ A\ x)) = fact\ (size\ A)$
proof (*induction A rule*: *multiset_remove_induct*)
case (*remove A*)
have $card\ (permutations\_of\_multiset\ A) =$
$card\ (\bigcup x{\in}set\_mset\ A.\ (\#)\ x\ `\ permutations\_of\_multiset\ (A - \{\#x\#\}))$
by (*simp add*: *permutations_of_multiset_nonempty remove.hyps*)
also have ... $= (\sum x{\in}set\_mset\ A.\ card\ (permutations\_of\_multiset\ (A - \{\#x\#\})))$
by (*subst card_UN_disjoint*) (*auto simp*: *card_image*)
also have ... $* (\prod x{\in}set\_mset\ A.\ fact\ (count\ A\ x)) =$
$(\sum x{\in}set\_mset\ A.\ card\ (permutations\_of\_multiset\ (A - \{\#x\#\})) *$
$(\prod y{\in}set\_mset\ A.\ fact\ (count\ A\ y)))$
by (*subst sum_distrib_right*) *simp_all*
also have ... $= (\sum x{\in}set\_mset\ A.\ count\ A\ x * fact\ (size\ A - 1))$
proof (*intro sum.cong refl*)
fix *x* **assume** *x*: $x \in\#\ A$
have $card\ (permutations\_of\_multiset\ (A - \{\#x\#\})) * (\prod y{\in}set\_mset\ A.\ fact\ (count\ A\ y)) =$
$count\ A\ x * (card\ (permutations\_of\_multiset\ (A - \{\#x\#\})) *$
$(\prod y{\in}set\_mset\ (A - \{\#x\#\}).\ fact\ (count\ (A - \{\#x\#\})\ y)))$ (**is** *?lhs* = _)
by (*subst multiset_prod_fact_remove*[*OF x*]) *simp_all*

also note *remove.IH*[*OF x*]
also from *x* **have** *size* (*A* − {#*x*#}) = *size A* − *1* **by** (*simp add*: *size_Diff_submset*)
finally show *?lhs* = *count A x* ∗ *fact* (*size A* − *1*) **.**
qed
also have (∑*x*∈*set_mset A*. *count A x* ∗ *fact* (*size A* − *1*)) =
size A ∗ *fact* (*size A* − *1*)
by (*simp add*: *sum_distrib_right size_multiset_overloaded_eq*)
also from *remove.hyps* **have** . . . = *fact* (*size A*)
by (*cases size A*) *auto*
finally show *?case* **.**
qed *simp_all*

theorem *card_permutations_of_multiset*:
card (*permutations_of_multiset A*) = *fact* (*size A*) *div* (∏*x*∈*set_mset A*. *fact* (*count A x*))
(∏*x*∈*set_mset A*. *fact* (*count A x*) :: *nat*) *dvd fact* (*size A*)
by (*simp_all flip*: *card_permutations_of_multiset_aux*[*of A*])

lemma *card_permutations_of_multiset_insert_aux*:
card (*permutations_of_multiset* (*A* + {#*x*#})) ∗ (*count A x* + *1*) =
(*size A* + *1*) ∗ *card* (*permutations_of_multiset A*)
proof −
note *card_permutations_of_multiset_aux*[*of A* + {#*x*#}]
also have *fact* (*size* (*A* + {#*x*#})) = (*size A* + *1*) ∗ *fact* (*size A*) **by** *simp*
also note *multiset_prod_fact_insert*[*of A x*]
also note *card_permutations_of_multiset_aux*[*of A*, *symmetric*]
finally have *card* (*permutations_of_multiset* (*A* + {#*x*#})) ∗ (*count A x* + *1*) ∗
(∏*y*∈*set_mset A*. *fact* (*count A y*)) =
(*size A* + *1*) ∗ *card* (*permutations_of_multiset A*) ∗
(∏*x*∈*set_mset A*. *fact* (*count A x*)) **by** (*simp only*: *mult_ac*)
thus *?thesis* **by** (*subst* (*asm*) *mult_right_cancel*) *simp_all*
qed

lemma *card_permutations_of_multiset_remove_aux*:
assumes *x* ∈# *A*
shows *card* (*permutations_of_multiset A*) ∗ *count A x* =
size A ∗ *card* (*permutations_of_multiset* (*A* − {#*x*#}))
proof −
from *assms* **have** *A*: *A* − {#*x*#} + {#*x*#} = *A* **by** *simp*
from *assms* **have** *B*: *size A* = *size* (*A* − {#*x*#}) + *1*
by (*subst A* [*symmetric*], *subst size_union*) *simp*
show *?thesis*
using *card_permutations_of_multiset_insert_aux*[*of A* − {#*x*#} *x*, *unfolded A*] *assms*
by (*simp add*: *B*)
qed

lemma *real_card_permutations_of_multiset_remove*:

assumes *x* ∈# *A*
shows *real* (*card* (*permutations_of_multiset* (*A* − {#*x*#}))) =
real (*card* (*permutations_of_multiset A*) ∗ *count A x*) / *real* (*size A*)
using *assms* **by** (*subst card_permutations_of_multiset_remove_aux*[*OF assms*]) *auto*

lemma *real_card_permutations_of_multiset_remove′*:
assumes *x* ∈# *A*
shows *real* (*card* (*permutations_of_multiset A*)) =
real (*size A* ∗ *card* (*permutations_of_multiset* (*A* − {#*x*#}))) / *real* (*count A x*)
using *assms* **by** (*subst card_permutations_of_multiset_remove_aux*[*OF assms*, *symmetric*]) *simp*

end

5.3 Permutations of a set

definition *permutations_of_set* :: ′*a set* ⇒ ′*a list set* **where**
permutations_of_set A = {*xs*. *set xs* = *A* ∧ *distinct xs*}

lemma *permutations_of_set_altdef*:
finite A ⟹ *permutations_of_set A* = *permutations_of_multiset* (*mset_set A*)
by (*auto simp add*: *permutations_of_set_def permutations_of_multiset_def mset_set_set*
in_multiset_in_set [*symmetric*] *mset_eq_mset_set_imp_distinct*)

lemma *permutations_of_setI* [*intro*]:
assumes *set xs* = *A distinct xs*
shows *xs* ∈ *permutations_of_set A*
using *assms* **unfolding** *permutations_of_set_def* **by** *simp*

lemma *permutations_of_setD*:
assumes *xs* ∈ *permutations_of_set A*
shows *set xs* = *A distinct xs*
using *assms* **unfolding** *permutations_of_set_def* **by** *simp_all*

lemma *permutations_of_set_lists*: *permutations_of_set A* ⊆ *lists A*
unfolding *permutations_of_set_def* **by** *auto*

lemma *permutations_of_set_empty* [*simp*]: *permutations_of_set* {} = {[]}
by (*auto simp*: *permutations_of_set_def*)

lemma *UN_set_permutations_of_set* [*simp*]:
finite A ⟹ (⋃*xs*∈*permutations_of_set A*. *set xs*) = *A*
using *finite_distinct_list* **by** (*auto simp*: *permutations_of_set_def*)

lemma *permutations_of_set_infinite*:
¬*finite A* ⟹ *permutations_of_set A* = {}

by (*auto simp*: *permutations_of_set_def*)

lemma *permutations_of_set_nonempty*:
$A \neq \{\} \Longrightarrow$ *permutations_of_set* $A =$
$(\bigcup x{\in}A.\ (\lambda xs.\ x \ \#\ xs)$ ‘ *permutations_of_set* $(A - \{x\}))$
by (*cases finite A*)
(*simp_all add*: *permutations_of_multiset_nonempty mset_set_empty_iff mset_set_Diff*

permutations_of_set_altdef permutations_of_set_infinite)

lemma *permutations_of_set_singleton* [*simp*]: *permutations_of_set* $\{x\} = \{[x]\}$
by (*subst permutations_of_set_nonempty*) *auto*

lemma *permutations_of_set_doubleton*:
$x \neq y \Longrightarrow$ *permutations_of_set* $\{x,y\} = \{[x,y],\ [y,x]\}$
by (*subst permutations_of_set_nonempty*)
(*simp_all add*: *insert_Diff_if insert_commute*)

lemma *rev_permutations_of_set* [*simp*]:
rev ‘ *permutations_of_set A* = *permutations_of_set A*
by (*cases finite A*) (*simp_all add*: *permutations_of_set_altdef permutations_of_set_infinite*)

lemma *length_finite_permutations_of_set*:
$xs \in$ *permutations_of_set* $A \Longrightarrow$ *length* $xs =$ *card* A
by (*auto simp*: *permutations_of_set_def distinct_card*)

lemma *finite_permutations_of_set* [*simp*]: *finite* (*permutations_of_set A*)
by (*cases finite A*) (*simp_all add*: *permutations_of_set_infinite permutations_of_set_altdef*)

lemma *permutations_of_set_empty_iff* [*simp*]:
permutations_of_set $A = \{\} \longleftrightarrow \neg$*finite* A
unfolding *permutations_of_set_def* **using** *finite_distinct_list*[*of A*] **by** *auto*

lemma *card_permutations_of_set* [*simp*]:
finite $A \Longrightarrow$ *card* (*permutations_of_set A*) = *fact* (*card A*)
by (*simp add*: *permutations_of_set_altdef card_permutations_of_multiset del*:
One_nat_def)

lemma *permutations_of_set_image_inj*:
assumes *inj*: *inj_on f A*
shows *permutations_of_set* (*f* ‘ *A*) = *map f* ‘ *permutations_of_set A*
by (*cases finite A*)
(*simp_all add*: *permutations_of_set_infinite permutations_of_set_altdef*
permutations_of_multiset_image mset_set_image_inj inj
finite_image_iff)

lemma *permutations_of_set_image_permutes*:
σ *permutes* $A \Longrightarrow$ *map* σ ‘ *permutations_of_set A* = *permutations_of_set A*
by (*subst permutations_of_set_image_inj* [*symmetric*])

(*simp_all add*: *permutes_inj_on permutes_image*)

5.4 Code generation

First, we give code an implementation for permutations of lists.

declare *length_remove1* [*termination_simp*]

fun *permutations_of_list_impl* **where**
permutations_of_list_impl xs = (*if xs* = [] *then* [[]] *else*
List.bind (*remdups xs*) (λ*x*. *map* ((#) *x*) (*permutations_of_list_impl* (*remove1*
x xs))))

fun *permutations_of_list_impl_aux* **where**
permutations_of_list_impl_aux acc xs = (*if xs* = [] *then* [*acc*] *else*
List.bind (*remdups xs*) (λ*x*. *permutations_of_list_impl_aux* (*x*#*acc*) (*remove1*
x xs)))

declare *permutations_of_list_impl_aux.simps* [*simp del*]
declare *permutations_of_list_impl.simps* [*simp del*]

lemma *permutations_of_list_impl_Nil* [*simp*]:
permutations_of_list_impl [] = [[]]
by (*simp add*: *permutations_of_list_impl.simps*)

lemma *permutations_of_list_impl_nonempty*:
xs ≠ [] ⟹ *permutations_of_list_impl xs* =
List.bind (*remdups xs*) (λ*x*. *map* ((#) *x*) (*permutations_of_list_impl* (*remove1*
x xs)))
by (*subst permutations_of_list_impl.simps*) *simp_all*

lemma *set_permutations_of_list_impl*:
set (*permutations_of_list_impl xs*) = *permutations_of_multiset* (*mset xs*)
by (*induction xs rule*: *permutations_of_list_impl.induct*)
(*subst permutations_of_list_impl.simps*,
simp_all add: *permutations_of_multiset_nonempty set_list_bind*)

lemma *distinct_permutations_of_list_impl*:
distinct (*permutations_of_list_impl xs*)
by (*induction xs rule*: *permutations_of_list_impl.induct*,
subst permutations_of_list_impl.simps)
(*auto intro*!: *distinct_list_bind simp*: *distinct_map o_def disjoint_family_on_def*)

lemma *permutations_of_list_impl_aux_correct'*:
permutations_of_list_impl_aux acc xs =
map (λ*xs*. *rev xs* @ *acc*) (*permutations_of_list_impl xs*)
by (*induction acc xs rule*: *permutations_of_list_impl_aux.induct*,
subst permutations_of_list_impl_aux.simps, *subst permutations_of_list_impl.simps*)
(*auto simp*: *map_list_bind intro*!: *list_bind_cong*)

```
lemma permutations_of_list_impl_aux_correct:
  permutations_of_list_impl_aux [] xs = map rev (permutations_of_list_impl xs)
  by (simp add: permutations_of_list_impl_aux_correct')

lemma distinct_permutations_of_list_impl_aux:
  distinct (permutations_of_list_impl_aux acc xs)
  by (simp add: permutations_of_list_impl_aux_correct' distinct_map
        distinct_permutations_of_list_impl inj_on_def)

lemma set_permutations_of_list_impl_aux:
   set (permutations_of_list_impl_aux [] xs) = permutations_of_multiset (mset
xs)
 by (simp add: permutations_of_list_impl_aux_correct set_permutations_of_list_impl)

declare set_permutations_of_list_impl_aux [symmetric, code]

value [code] permutations_of_multiset {#1,2,3,4::int#}
```

Now we turn to permutations of sets. We define an auxiliary version with an accumulator to avoid having to map over the results.

```
function permutations_of_set_aux where
  permutations_of_set_aux acc A =
     (if ¬finite A then {} else if A = {} then {acc} else
        (⋃x∈A. permutations_of_set_aux (x#acc) (A − {x})))
by auto
termination by (relation Wellfounded.measure (card ∘ snd)) (simp_all add: card_gt_0_iff)

lemma permutations_of_set_aux_altdef:
  permutations_of_set_aux acc A = (λxs. rev xs @ acc) ` permutations_of_set A
proof (cases finite A)
  assume finite A
  thus ?thesis
  proof (induction A arbitrary: acc rule: finite_psubset_induct)
    case (psubset A acc)
    show ?case
    proof (cases A = {})
      case False
      note [simp del] = permutations_of_set_aux.simps
      from psubset.hyps False
        have permutations_of_set_aux acc A =
                (⋃y∈A. permutations_of_set_aux (y#acc) (A − {y}))
        by (subst permutations_of_set_aux.simps) simp_all
       also have . . . = (⋃y∈A. (λxs. rev xs @ acc) ` (λxs. y # xs) ` permuta-
tions_of_set (A − {y}))
        apply (rule arg_cong [of _ _ Union], rule image_cong)
         apply (simp_all add: image_image)
        apply (subst psubset)
         apply auto
        done
```

also from *False* **have** ... = (*λxs. rev xs @ acc*) ‘ *permutations_of_set A*
by (*subst* (*2*) *permutations_of_set_nonempty*) (*simp_all add*: *image_UN*)
finally show *?thesis* **.**
qed *simp_all*
qed
qed (*simp_all add*: *permutations_of_set_infinite*)

declare *permutations_of_set_aux.simps* [*simp del*]

lemma *permutations_of_set_aux_correct*:
permutations_of_set_aux [] *A* = *permutations_of_set A*
by (*simp add*: *permutations_of_set_aux_altdef*)

In another refinement step, we define a version on lists.

declare *length_remove1* [*termination_simp*]

fun *permutations_of_set_aux_list* **where**
permutations_of_set_aux_list acc xs =
(*if xs* = [] *then* [*acc*] *else*
List.bind xs (*λx. permutations_of_set_aux_list* (*x#acc*) (*List.remove1 x xs*)))

definition *permutations_of_set_list* **where**
permutations_of_set_list xs = *permutations_of_set_aux_list* [] *xs*

declare *permutations_of_set_aux_list.simps* [*simp del*]

lemma *permutations_of_set_aux_list_refine*:
assumes *distinct xs*
shows *set* (*permutations_of_set_aux_list acc xs*) = *permutations_of_set_aux acc* (*set xs*)
using *assms*
by (*induction acc xs rule*: *permutations_of_set_aux_list.induct*)
(*subst permutations_of_set_aux_list.simps*,
subst permutations_of_set_aux.simps,
simp_all add: *set_list_bind*)

The permutation lists contain no duplicates if the inputs contain no duplicates. Therefore, these functions can easily be used when working with a representation of sets by distinct lists. The same approach should generalise to any kind of set implementation that supports a monadic bind operation, and since the results are disjoint, merging should be cheap.

lemma *distinct_permutations_of_set_aux_list*:
distinct xs ⟹ *distinct* (*permutations_of_set_aux_list acc xs*)
by (*induction acc xs rule*: *permutations_of_set_aux_list.induct*)
(*subst permutations_of_set_aux_list.simps*,
auto intro!: *distinct_list_bind simp*: *disjoint_family_on_def*
permutations_of_set_aux_list_refine permutations_of_set_aux_altdef)

```
lemma distinct_permutations_of_set_list:
    distinct xs ⟹ distinct (permutations_of_set_list xs)
  by (simp add: permutations_of_set_list_def distinct_permutations_of_set_aux_list)

lemma permutations_of_list:
    permutations_of_set (set xs) = set (permutations_of_set_list (remdups xs))
  by (simp add: permutations_of_set_aux_correct [symmetric]
        permutations_of_set_aux_list_refine permutations_of_set_list_def)

lemma permutations_of_list_code [code]:
  permutations_of_set (set xs) = set (permutations_of_set_list (remdups xs))
  permutations_of_set (List.coset xs) =
     Code.abort (STR ''Permutation of set complement not supported'')
       (λ_. permutations_of_set (List.coset xs))
  by (simp_all add: permutations_of_list)

value [code] permutations_of_set (set ''abcd'')

end


theory Cycles
  imports
    HOL−Library.FuncSet
Permutations
begin
```

6 Cycles

6.1 Definitions

```
abbreviation cycle :: 'a list ⇒ bool
  where cycle cs ≡ distinct cs

fun cycle_of_list :: 'a list ⇒ 'a ⇒ 'a
  where
    cycle_of_list (i # j # cs) = transpose i j ∘ cycle_of_list (j # cs)
  | cycle_of_list cs = id
```

6.2 Basic Properties

We start proving that the function derived from a cycle rotates its support list.

```
lemma id_outside_supp:
  assumes x ∉ set cs shows (cycle_of_list cs) x = x
  using assms by (induct cs rule: cycle_of_list.induct) (simp_all)

lemma permutation_of_cycle: permutation (cycle_of_list cs)
```

```
proof (induct cs rule: cycle_of_list.induct)
  case 1 thus ?case
   using permutation_compose[OF permutation_swap_id] unfolding comp_apply
by simp
qed simp_all

lemma cycle_permutes: (cycle_of_list cs) permutes (set cs)
   using permutation_bijective[OF permutation_of_cycle] id_outside_supp[of _
cs]
  by (simp add: bij_iff permutes_def)

theorem cyclic_rotation:
  assumes cycle cs shows map ((cycle_of_list cs) ^^ n) cs = rotate n cs
proof –
  { have map (cycle_of_list cs) cs = rotate1 cs using assms(1)
    proof (induction cs rule: cycle_of_list.induct)
      case (1 i j cs)
      then have ‹i ∉ set cs› ‹j ∉ set cs›
        by auto
      then have ‹map (Transposition.transpose i j) cs = cs›
        by (auto intro: map_idI simp add: transpose_eq_iff)
      show ?case
      proof (cases)
        assume cs = Nil thus ?thesis by simp
      next
        assume cs ≠ Nil hence ge_two: length (j # cs) ≥ 2
          using not_less by auto
        have map (cycle_of_list (i # j # cs)) (i # j # cs) =
              map (transpose i j) (map (cycle_of_list (j # cs)) (i # j # cs)) by
simp
        also have ... = map (transpose i j) (i # (rotate1 (j # cs)))
          by (metis 1.IH 1.prems distinct.simps(2) id_outside_supp list.simps(9))
        also have ... = map (transpose i j) (i # (cs @ [j])) by simp
        also have ... = j # (map (transpose i j) cs) @ [i] by simp
        also have ... = j # cs @ [i]
          using ‹map (Transposition.transpose i j) cs = cs› by simp
        also have ... = rotate1 (i # j # cs) by simp
        finally show ?thesis .
      qed
    qed simp_all }
  note cyclic_rotation' = this

  show ?thesis
   using cyclic_rotation' by (induct n) (auto, metis map_map rotate1_rotate_swap
rotate_map)
qed

corollary cycle_is_surj:
  assumes cycle cs shows (cycle_of_list cs) ` (set cs) = (set cs)
```

```
  using cyclic_rotation[OF assms, of Suc 0] by (simp add: image_set)

corollary cycle_is_id_root:
  assumes cycle cs shows (cycle_of_list cs) ^^ (length cs) = id
proof -
  have map ((cycle_of_list cs) ^^ (length cs)) cs = cs
    unfolding cyclic_rotation[OF assms] by simp
  hence ((cycle_of_list cs) ^^ (length cs)) i = i if i ∈ set cs for i
    using that map_eq_conv by fastforce
  moreover have ((cycle_of_list cs) ^^ n) i = i if i ∉ set cs for i n
    using id_outside_supp[OF that] by (induct n) (simp_all)
  ultimately show ?thesis
    by fastforce
qed

corollary cycle_of_list_rotate_independent:
  assumes cycle cs shows (cycle_of_list cs) = (cycle_of_list (rotate n cs))
proof -
  { fix cs :: 'a list assume cs: cycle cs
    have (cycle_of_list cs) = (cycle_of_list (rotate1 cs))
    proof -
      from cs have rotate1_cs: cycle (rotate1 cs) by simp
      hence map (cycle_of_list (rotate1 cs)) (rotate1 cs) = (rotate 2 cs)
       using cyclic_rotation[OF rotate1_cs, of 1] by (simp add: numeral_2_eq_2)
      moreover have map (cycle_of_list cs) (rotate1 cs) = (rotate 2 cs)
        using cyclic_rotation[OF cs]
       by (metis One_nat_def Suc_1 funpow.simps(2) id_apply map_map rotate0
rotate_Suc)
      ultimately have (cycle_of_list cs) i = (cycle_of_list (rotate1 cs)) i if i ∈
set cs for i
       using that map_eq_conv unfolding sym[OF set_rotate1[of cs]] by fastforce

       moreover have (cycle_of_list cs) i = (cycle_of_list (rotate1 cs)) i if i ∉
set cs for i
        using that by (simp add: id_outside_supp)
      ultimately show (cycle_of_list cs) = (cycle_of_list (rotate1 cs))
        by blast
    qed } note rotate1_lemma = this

  show ?thesis
     using rotate1_lemma[of rotate n cs] by (induct n) (auto, metis assms dis-
tinct_rotate rotate1_lemma)
qed
```

6.3 Conjugation of cycles

```
lemma conjugation_of_cycle:
  assumes cycle cs and bij p
  shows p ∘ (cycle_of_list cs) ∘ (inv p) = cycle_of_list (map p cs)
```

using *assms*
proof (*induction cs rule*: *cycle_of_list.induct*)
case (*1 i j cs*)
have *p* ∘ *cycle_of_list* (*i* # *j* # *cs*) ∘ *inv p* =
(*p* ∘ (*transpose i j*) ∘ *inv p*) ∘ (*p* ∘ *cycle_of_list* (*j* # *cs*) ∘ *inv p*)
by (*simp add*: *assms*(*2*) *bij_is_inj fun.map_comp*)
also have ... = (*transpose* (*p i*) (*p j*)) ∘ (*p* ∘ *cycle_of_list* (*j* # *cs*) ∘ *inv p*)
using *1.prems*(*2*) **by** (*simp add*: *bij_inv_eq_iff transpose_apply_commute fun_eq_iff bij_betw_inv_into_left*)
finally have *p* ∘ *cycle_of_list* (*i* # *j* # *cs*) ∘ *inv p* =
(*transpose* (*p i*) (*p j*)) ∘ (*cycle_of_list* (*map p* (*j* # *cs*)))
using *1.IH 1.prems*(*1*) *assms*(*2*) **by** *fastforce*
thus *?case* **by** (*simp add*: *fun_eq_iff*)
next
case *2_1* **thus** *?case*
by (*metis bij_is_surj comp_id cycle_of_list.simps*(*2*) *list.simps*(*8*) *surj_iff*)
next
case *2_2* **thus** *?case*
by (*metis bij_is_surj comp_id cycle_of_list.simps*(*3*) *list.simps*(*8*) *list.simps*(*9*) *surj_iff*)
qed

6.4 When Cycles Commute

lemma *cycles_commute*:
assumes *cycle p cycle q* **and** *set p* ∩ *set q* = {}
shows (*cycle_of_list p*) ∘ (*cycle_of_list q*) = (*cycle_of_list q*) ∘ (*cycle_of_list p*)
proof
{ fix *p* :: *'a list* **and** *q* :: *'a list* **and** *i* :: *'a*
assume *A*: *cycle p cycle q set p* ∩ *set q* = {} *i* ∈ *set p i* ∉ *set q*
have ((*cycle_of_list p*) ∘ (*cycle_of_list q*)) *i* =
((*cycle_of_list q*) ∘ (*cycle_of_list p*)) *i*
proof –
have ((*cycle_of_list p*) ∘ (*cycle_of_list q*)) *i* = (*cycle_of_list p*) *i*
using *id_outside_supp*[*OF A*(*5*)] **by** *simp*
also have ... = ((*cycle_of_list q*) ∘ (*cycle_of_list p*)) *i*
using *id_outside_supp*[*of* (*cycle_of_list p*) *i*] *cycle_is_surj*[*OF A*(*1*)] *A*(*3*,*4*) **by** *fastforce*
finally show *?thesis* **.**
qed } **note** *aui_lemma* = *this*

fix *i* **consider** *i* ∈ *set p i* ∉ *set q* | *i* ∉ *set p i* ∈ *set q* | *i* ∉ *set p i* ∉ *set q*
using ‹*set p* ∩ *set q* = {}› **by** *blast*
thus ((*cycle_of_list p*) ∘ (*cycle_of_list q*)) *i* = ((*cycle_of_list q*) ∘ (*cycle_of_list p*)) *i*
proof *cases*
case *1* **thus** *?thesis*
using *aui_lemma*[*OF assms*] **by** *simp*

```
  next
    case 2 thus ?thesis
      using aui_lemma[OF assms(2,1)] assms(3) by (simp add: ac_simps)
  next
    case 3 thus ?thesis
      by (simp add: id_outside_supp)
  qed
qed
```

6.5 Cycles from Permutations

6.5.1 Exponentiation of permutations

Some important properties of permutations before defining how to extract its cycles.

```
lemma permutation_funpow:
  assumes permutation p shows permutation (p ^^ n)
  using assms by (induct n) (simp_all add: permutation_compose)

lemma permutes_funpow:
  assumes p permutes S shows (p ^^ n) permutes S
  using assms by (induct n) (simp add: permutes_def, metis funpow_Suc_right
permutes_compose)

lemma funpow_diff:
  assumes inj p and i ≤ j (p ^^ i) a = (p ^^ j) a shows (p ^^ (j − i)) a = a
proof −
  have (p ^^ i) ((p ^^ (j − i)) a) = (p ^^ i) a
    using assms(2−3) by (metis (no_types) add_diff_inverse_nat funpow_add
not_le o_def)
  thus ?thesis
    unfolding inj_eq[OF inj_fn[OF assms(1)], of i] .
qed

lemma permutation_is_nilpotent:
  assumes permutation p obtains n where (p ^^ n) = id and n > 0
proof −
  obtain S where finite S and p permutes S
    using assms unfolding permutation_permutes by blast
  hence ∃ n. (p ^^ n) = id ∧ n > 0
  proof (induct S arbitrary: p)
    case empty thus ?case
      using id_funpow[of 1] unfolding permutes_empty by blast
  next
    case (insert s S)
    have (λn. (p ^^ n) s) ‘ UNIV ⊆ (insert s S)
      using permutes_in_image[OF permutes_funpow[OF insert(4)], of _ s] by
auto
    hence ¬ inj_on (λn. (p ^^ n) s) UNIV
```

using *insert*(*1*) *infinite_iff_countable_subset* **unfolding** *sym*[*OF finite_insert*, *of S s*] **by** *metis*
then obtain *i j* **where** *ij*: $i < j$ $(p \mathbin{\hat{}\hat{}} i)\ s = (p \mathbin{\hat{}\hat{}} j)\ s$
unfolding *inj_on_def* **by** (*metis nat_neq_iff*)
hence $(p \mathbin{\hat{}\hat{}} (j - i))\ s = s$
using *funpow_diff*[*OF permutes_inj*[*OF insert*(*4*)]] *le_eq_less_or_eq* **by** *blast*
hence $p \mathbin{\hat{}\hat{}} (j - i)$ *permutes S*
using *permutes_superset*[*OF permutes_funpow*[*OF insert*(*4*), *of* $j - i$], *of S*] **by** *auto*
then obtain *n* **where** *n*: $((p \mathbin{\hat{}\hat{}} (j - i)) \mathbin{\hat{}\hat{}} n) = id$ $n > 0$
using *insert*(*3*) **by** *blast*
thus *?case*
using *ij*(*1*) *nat_0_less_mult_iff zero_less_diff* **unfolding** *funpow_mult* **by** *metis*
qed
thus *thesis*
using *that* **by** *blast*
qed

lemma *permutation_is_nilpotent'*:
assumes *permutation p* **obtains** *n* **where** $(p \mathbin{\hat{}\hat{}} n) = id$ **and** $n > m$
proof −
obtain *n* **where** $(p \mathbin{\hat{}\hat{}} n) = id$ **and** $n > 0$
using *permutation_is_nilpotent*[*OF assms*] **by** *blast*
then obtain *k* **where** $n * k > m$
by (*metis dividend_less_times_div mult_Suc_right*)
from ‹$(p \mathbin{\hat{}\hat{}} n) = id$› **have** $p \mathbin{\hat{}\hat{}} (n * k) = id$
by (*induct k*) (*simp*, *metis funpow_mult id_funpow*)
with ‹$n * k > m$› **show** *thesis*
using *that* **by** *blast*
qed

6.5.2 Extraction of cycles from permutations

definition *least_power* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow nat$
where *least_power f x* = (*LEAST n*. $(f \mathbin{\hat{}\hat{}} n)\ x = x \wedge n > 0$)

abbreviation *support* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ list$
where *support p x* ≡ *map* ($\lambda i.\ (p \mathbin{\hat{}\hat{}} i)\ x$) [*0*..< (*least_power p x*)]

lemma *least_powerI*:
assumes $(f \mathbin{\hat{}\hat{}} n)\ x = x$ **and** $n > 0$
shows $(f \mathbin{\hat{}\hat{}} (least\_power\ f\ x))\ x = x$ **and** *least_power f x* > *0*
using *assms* **unfolding** *least_power_def* **by** (*metis* (*mono_tags*, *lifting*) *LeastI*)+

lemma *least_power_le*:
assumes $(f \mathbin{\hat{}\hat{}} n)\ x = x$ **and** $n > 0$ **shows** *least_power f x* ≤ *n*

using *assms* **unfolding** *least_power_def* **by** (*simp add*: *Least_le*)

lemma *least_power_of_permutation*:
assumes *permutation p* **shows** $(p \frown\frown (least\_power\ p\ a))\ a = a$ **and** *least_power p a* > 0
using *permutation_is_nilpotent*[*OF assms*] *least_powerI* **by** (*metis id_apply*)+

lemma *least_power_gt_one*:
assumes *permutation p* **and** $p\ a \neq a$ **shows** *least_power p a* $>$ *Suc 0*
using *least_power_of_permutation*[*OF assms*(*1*)] *assms*(*2*)
by (*metis Suc_lessI funpow.simps*(*2*) *funpow_simps_right*(*1*) *o_id*)

lemma *least_power_minimal*:
assumes $(p \frown\frown n)\ a = a$ **shows** (*least_power p a*) *dvd n*
proof (*cases* $n = 0$, *simp*)
let *?lpow* = *least_power p*

assume $n \neq 0$ **then have** $n > 0$ **by** *simp*
hence $(p \frown\frown (?lpow\ a))\ a = a$ **and** *least_power p a* > 0
using *assms* **unfolding** *least_power_def* **by** (*metis* (*mono_tags*, *lifting*) *LeastI*)+
hence *aux_lemma*: $(p \frown\frown ((?lpow\ a) * k))\ a = a$ **for** k :: *nat*
by (*induct k*) (*simp_all add*: *funpow_add*)

have $(p \frown\frown (n\ mod\ ?lpow\ a))\ ((p \frown\frown (n - (n\ mod\ ?lpow\ a)))\ a) = (p \frown\frown n)\ a$
by (*metis add_diff_inverse_nat funpow_add mod_less_eq_dividend not_less o_apply*)
with ‹$(p \frown\frown n)\ a = a$› **have** $(p \frown\frown (n\ mod\ ?lpow\ a))\ a = a$
using *aux_lemma* **by** (*simp add*: *minus_mod_eq_mult_div*)
hence $?lpow\ a \leq n\ mod\ ?lpow\ a$ **if** $n\ mod\ ?lpow\ a > 0$
using *least_power_le*[*OF _ that*, *of p a*] **by** *simp*
with ‹*least_power p a* > 0› **show** (*least_power p a*) *dvd n*
using *mod_less_divisor not_le* **by** *blast*
qed

lemma *least_power_dvd*:
assumes *permutation p* **shows** (*least_power p a*) *dvd n* $\longleftrightarrow (p \frown\frown n)\ a = a$
proof
show $(p \frown\frown n)\ a = a \Longrightarrow$ (*least_power p a*) *dvd n*
using *least_power_minimal*[*of _ p*] **by** *simp*
next
have $(p \frown\frown ((least\_power\ p\ a) * k))\ a = a$ **for** k :: *nat*
using *least_power_of_permutation*(*1*)[*OF assms*(*1*)] **by** (*induct k*) (*simp_all add*: *funpow_add*)
thus (*least_power p a*) *dvd n* $\Longrightarrow (p \frown\frown n)\ a = a$ **by** *blast*
qed

theorem *cycle_of_permutation*:
assumes *permutation p* **shows** *cycle* (*support p a*)
proof −

have (*least_power p a*) *dvd* (*j* − *i*) **if** *i* ≤ *j j* < *least_power p a* **and** (*p* ^^ *i*) *a* = (*p* ^^ *j*) *a* **for** *i j*
  using *funpow_diff*[*OF bij_is_inj that*(*1*,*3*)] *assms* **by** (*simp add*: *permutation least_power_dvd*)
moreover have *i* = *j* **if** *i* ≤ *j j* < *least_power p a* **and** (*least_power p a*) *dvd* (*j* − *i*) **for** *i j*
  using *that le_eq_less_or_eq nat_dvd_not_less* **by** *auto*
ultimately have *inj_on* (λ*i*. (*p* ^^ *i*) *a*) {..< (*least_power p a*)}
  unfolding *inj_on_def* **by** (*metis le_cases lessThan_iff*)
thus *?thesis*
  by (*simp add*: *atLeast_upt distinct_map*)
qed

6.6 Decomposition on Cycles

We show that a permutation can be decomposed on cycles

6.6.1 Preliminaries

lemma *support_set*:
assumes *permutation p* **shows** *set* (*support p a*) = *range* (λ*i*. (*p* ^^ *i*) *a*)
proof
show *set* (*support p a*) ⊆ *range* (λ*i*. (*p* ^^ *i*) *a*)
  by *auto*
next
show *range* (λ*i*. (*p* ^^ *i*) *a*) ⊆ *set* (*support p a*)
proof (*auto*)
  fix *i*
  have (*p* ^^ *i*) *a* = (*p* ^^ (*i mod* (*least_power p a*))) ((*p* ^^ (*i* − (*i mod* (*least_power p a*)))) *a*)
    by (*metis add_diff_inverse_nat funpow_add mod_less_eq_dividend not_le o_apply*)
  also have ... = (*p* ^^ (*i mod* (*least_power p a*))) *a*
    using *least_power_dvd*[*OF assms*] **by** (*metis dvd_minus_mod*)
  also have ... ∈ (λ*i*. (*p* ^^ *i*) *a*) ‘ {*0*..< (*least_power p a*)}
    using *least_power_of_permutation*(*2*)[*OF assms*] **by** *fastforce*
  finally show (*p* ^^ *i*) *a* ∈ (λ*i*. (*p* ^^ *i*) *a*) ‘ {*0*..< (*least_power p a*)} **.**
qed
qed

lemma *disjoint_support*:
assumes *permutation p* **shows** *disjoint* (*range* (λ*a*. *set* (*support p a*))) (**is** *disjoint ?A*)
proof (*rule disjointI*)
{ fix *i j a b*
  assume *set* (*support p a*) ∩ *set* (*support p b*) ≠ {} **have** *set* (*support p a*) ⊆ *set* (*support p b*)
    unfolding *support_set*[*OF assms*]
  proof (*auto*)

from ‹*set* (*support p a*) ∩ *set* (*support p b*) ≠ {}›
obtain *i j* **where** *ij*: (*p* ^^ *i*) *a* = (*p* ^^ *j*) *b*
by *auto*

fix *k*
have (*p* ^^ *k*) *a* = (*p* ^^ (*k* + (*least_power p a*) ∗ *l*)) *a* **for** *l*
using *least_power_dvd*[*OF assms*] **by** (*induct l*) (*simp*, *metis dvd_triv_left funpow_add o_def*)
then obtain *m* **where** *m* ≥ *i* **and** (*p* ^^ *m*) *a* = (*p* ^^ *k*) *a*
using *least_power_of_permutation*(*2*)[*OF assms*]
by (*metis dividend_less_times_div le_eq_less_or_eq mult_Suc_right trans_less_add2*)
hence (*p* ^^ *m*) *a* = (*p* ^^ (*m* − *i*)) ((*p* ^^ *i*) *a*)
by (*metis Nat.le_imp_diff_is_add funpow_add o_apply*)
with ‹(*p* ^^ *m*) *a* = (*p* ^^ *k*) *a*› **have** (*p* ^^ *k*) *a* = (*p* ^^ ((*m* − *i*) + *j*)) *b*
unfolding *ij* **by** (*simp add*: *funpow_add*)
thus (*p* ^^ *k*) *a* ∈ *range* (λ*i*. (*p* ^^ *i*) *b*)
by *blast*
qed **}** **note** *aux_lemma* = *this*

fix *supp_a supp_b*
assume *supp_a* ∈ *?A* **and** *supp_b* ∈ *?A*
then obtain *a b* **where** *a*: *supp_a* = *set* (*support p a*) **and** *b*: *supp_b* = *set* (*support p b*)
by *auto*
assume *supp_a* ≠ *supp_b* **thus** *supp_a* ∩ *supp_b* = {}
using *aux_lemma* **unfolding** *a b* **by** *blast*
qed

lemma *disjoint_support'*:
assumes *permutation p*
shows *set* (*support p a*) ∩ *set* (*support p b*) = {} ⟷ *a* ∉ *set* (*support p b*)
proof −
have *a* ∈ *set* (*support p a*)
using *least_power_of_permutation*(*2*)[*OF assms*] **by** *force*
show *?thesis*
proof
assume *set* (*support p a*) ∩ *set* (*support p b*) = {}
with ‹*a* ∈ *set* (*support p a*)› **show** *a* ∉ *set* (*support p b*)
by *blast*
next
assume *a* ∉ *set* (*support p b*) **show** *set* (*support p a*) ∩ *set* (*support p b*) = {}
proof (*rule ccontr*)
assume *set* (*support p a*) ∩ *set* (*support p b*) ≠ {}
hence *set* (*support p a*) = *set* (*support p b*)
using *disjoint_support*[*OF assms*] **by** (*meson UNIV_I disjoint_def image_iff*)
with ‹*a* ∈ *set* (*support p a*)› **and** ‹*a* ∉ *set* (*support p b*)› **show** *False*
by *simp*

qed
qed
qed

lemma *support_coverture*:
assumes *permutation p* **shows** $\bigcup$ { *set* (*support p a*) | *a*. *p a* $\neq$ *a* } = { *a*. *p a* $\neq$ *a* }
proof
show { *a*. *p a* $\neq$ *a* } $\subseteq$ $\bigcup$ { *set* (*support p a*) | *a*. *p a* $\neq$ *a* }
proof
fix *a* **assume** *a* $\in$ { *a*. *p a* $\neq$ *a* }
have *a* $\in$ *set* (*support p a*)
using *least_power_of_permutation*(*2*)[*OF assms*, *of a*] **by** *force*
with ‹*a* $\in$ { *a*. *p a* $\neq$ *a* }› **show** *a* $\in$ $\bigcup$ { *set* (*support p a*) | *a*. *p a* $\neq$ *a* }
by *blast*
qed
next
show $\bigcup$ { *set* (*support p a*) | *a*. *p a* $\neq$ *a* } $\subseteq$ { *a*. *p a* $\neq$ *a* }
proof
fix *b* **assume** *b* $\in$ $\bigcup$ { *set* (*support p a*) | *a*. *p a* $\neq$ *a* }
then obtain *a i* **where** *p a* $\neq$ *a* **and** (*p* ^^ *i*) *a* = *b*
by *auto*
have *p a* = *a* **if** (*p* ^^ *i*) *a* = (*p* ^^ *Suc i*) *a*
using *funpow_diff*[*OF bij_is_inj _ that*] *assms* **unfolding** *permutation* **by** *simp*
with ‹*p a* $\neq$ *a*› **and** ‹(*p* ^^ *i*) *a* = *b*› **show** *b* $\in$ { *a*. *p a* $\neq$ *a* }
by *auto*
qed
qed

theorem *cycle_restrict*:
assumes *permutation p* **and** *b* $\in$ *set* (*support p a*) **shows** *p b* = (*cycle_of_list* (*support p a*)) *b*
proof –
note *least_power_props* [*simp*] = *least_power_of_permutation*[*OF assms*(*1*)]

have *map* (*cycle_of_list* (*support p a*)) (*support p a*) = *rotate1* (*support p a*)
using *cyclic_rotation*[*OF cycle_of_permutation*[*OF assms*(*1*)], *of 1 a*] **by** *simp*
hence *map* (*cycle_of_list* (*support p a*)) (*support p a*) = *tl* (*support p a*) @ [*a*]
by (*simp add*: *hd_map rotate1_hd_tl*)
also have ... = *map p* (*support p a*)
proof (*rule nth_equalityI*, *auto*)
fix *i* **assume** *i* < *least_power p a* **show** (*tl* (*support p a*) @ [*a*]) ! *i* = *p* ((*p* ^^ *i*) *a*)
proof (*cases*)
assume *i*: *i* = *least_power p a* – *1*
hence (*tl* (*support p a*) @ [*a*]) ! *i* = *a*
by (*metis* (*no_types*, *lifting*) *diff_zero length_map length_tl length_upt nth_append_length*)

also have ... = *p* ((*p* ^^ *i*) *a*)
by (*metis* (*mono_tags*, *opaque_lifting*) *least_power_props i Suc_diff_1 funpow_simps_right*(*2*) *funpow_swap1 o_apply*)
finally show *?thesis* **.**
next
assume $i \neq$ *least_power p a* − *1*
with ‹*i* < *least_power p a*› **have** *i* < *least_power p a* − *1*
by *simp*
hence (*tl* (*support p a*) @ [*a*]) ! *i* = (*p* ^^ (*Suc i*)) *a*
by (*metis One_nat_def Suc_eq_plus1 add.commute length_map length_upt map_tl nth_append nth_map_upt tl_upt*)
thus *?thesis*
by *simp*
qed
qed
finally have *map* (*cycle_of_list* (*support p a*)) (*support p a*) = *map p* (*support p a*) **.**
thus *?thesis*
using *assms*(*2*) **by** *auto*
qed

6.6.2 Decomposition

inductive *cycle_decomp* :: *'a set* ⇒ (*'a* ⇒ *'a*) ⇒ *bool*
where
empty: *cycle_decomp* {} *id*
| *comp*: ⟦ *cycle_decomp I p*; *cycle cs*; *set cs* ∩ *I* = {} ⟧ ⟹
cycle_decomp (*set cs* ∪ *I*) ((*cycle_of_list cs*) ∘ *p*)

lemma *semidecomposition*:
assumes *p permutes S* **and** *finite S*
shows (λ*y*. *if y* ∈ (*S* − *set* (*support p a*)) *then p y else y*) *permutes* (*S* − *set* (*support p a*))
proof (*rule bij_imp_permutes*)
show (*if b* ∈ (*S* − *set* (*support p a*)) *then p b else b*) = *b* **if** *b* ∉ *S* − *set* (*support p a*) **for** *b*
using *that* **by** *auto*
next
have *is_permutation*: *permutation p*
using *assms* **unfolding** *permutation_permutes* **by** *blast*

let *?q* = λ*y*. *if y* ∈ (*S* − *set* (*support p a*)) *then p y else y*
show *bij_betw ?q* (*S* − *set* (*support p a*)) (*S* − *set* (*support p a*))
proof (*rule bij_betw_imageI*)
show *inj_on ?q* (*S* − *set* (*support p a*))
using *permutes_inj*[*OF assms*(*1*)] **unfolding** *inj_on_def* **by** *auto*
next
have *aux_lemma*: *set* (*support p s*) ⊆ (*S* − *set* (*support p a*)) **if** *s* ∈ *S* − *set*

```
(support p a) for s
    proof –
      have (p ^^ i) s ∈ S for i
      using that unfolding permutes_in_image[OF permutes_funpow[OF assms(1)]]
by simp
      thus ?thesis
        using that disjoint_support'[OF is_permutation, of s a] by auto
    qed
    have (p ^^ 1) s ∈ set (support p s) for s
      unfolding support_set[OF is_permutation] by blast
    hence p s ∈ set (support p s) for s
      by simp
    hence p ‘ (S − set (support p a)) ⊆ S − set (support p a)
      using aux_lemma by blast
    moreover have (p ^^ ((least_power p s) − 1)) s ∈ set (support p s) for s
      unfolding support_set[OF is_permutation] by blast
    hence ∃ s' ∈ set (support p s). p s' = s for s
     using least_power_of_permutation[OF is_permutation] by (metis Suc_diff_1
funpow.simps(2) o_apply)
    hence S − set (support p a) ⊆ p ‘ (S − set (support p a))
      using aux_lemma
      by (clarsimp simp add: image_iff) (metis image_subset_iff)
    ultimately show ?q ‘ (S − set (support p a)) = (S − set (support p a))
      by auto
  qed
qed

theorem cycle_decomposition:
  assumes p permutes S and finite S shows cycle_decomp S p
  using assms
proof(induct card S arbitrary: S p rule: less_induct)
  case less show ?case
  proof (cases)
    assume S = {} thus ?thesis
      using empty less(2) by auto
  next
    have is_permutation: permutation p
      using less(2−3) unfolding permutation_permutes by blast

    assume S ≠ {} then obtain s where s ∈ S
      by blast
    define q where q = (λy. if y ∈ (S − set (support p s)) then p y else y)
    have (cycle_of_list (support p s) ∘ q) = p
    proof
      fix a
       consider a ∈ S − set (support p s) | a ∈ set (support p s) | a ∉ S a ∉ set
(support p s)
        by blast
      thus ((cycle_of_list (support p s) ∘ q)) a = p a
```

proof *cases*
case *1*
have $(p \mathbin{\widehat{\ }\widehat{\ }} 1)\ a \in set\ (support\ p\ a)$
unfolding *support_set*[*OF is_permutation*] **by** *blast*
with ‹$a \in S - set\ (support\ p\ s)$› **have** $p\ a \notin set\ (support\ p\ s)$
using *disjoint_support'*[*OF is_permutation*, *of a s*] **by** *auto*
with ‹$a \in S - set\ (support\ p\ s)$› **show** *?thesis*
using *id_outside_supp*[*of _ support p s*] **unfolding** *q_def* **by** *simp*
next
case *2* **thus** *?thesis*
using *cycle_restrict*[*OF is_permutation*] **unfolding** *q_def* **by** *simp*
next
case *3* **thus** *?thesis*
using *id_outside_supp*[*OF 3*(*2*)] *less*(*2*) *permutes_not_in* **unfolding** *q_def* **by** *fastforce*
qed
qed

moreover from ‹$s \in S$› **have** $(p \mathbin{\widehat{\ }\widehat{\ }} i)\ s \in S$ **for** i
unfolding *permutes_in_image*[*OF permutes_funpow*[*OF less*(*2*)]] **.**
hence $set\ (support\ p\ s) \cup (S - set\ (support\ p\ s)) = S$
by *auto*

moreover have $s \in set\ (support\ p\ s)$
using *least_power_of_permutation*[*OF is_permutation*] **by** *force*
with ‹$s \in S$› **have** $card\ (S - set\ (support\ p\ s)) < card\ S$
using *less*(*3*) **by** (*metis DiffE card_seteq linorder_not_le subsetI*)
hence $cycle\_decomp\ (S - set\ (support\ p\ s))\ q$
using *less*(*1*)[*OF _ semidecomposition*[*OF less*(*2*−*3*)], *of s*] *less*(*3*) **unfolding** *q_def* **by** *blast*

moreover show *?thesis*
using *comp*[*OF calculation*(*3*) *cycle_of_permutation*[*OF is_permutation*], *of s*]
unfolding *calculation*(*1*−*2*) **by** *blast*
qed
qed

end

7 Permutations as abstract type

theory *Perm*
imports
Transposition
begin

This theory introduces basics about permutations, i.e. almost everywhere fix bijections. But it is by no means complete. Grieviously missing are cycles since these would require more elaboration, e.g. the concept of distinct lists equivalent under rotation, which maybe would also deserve its own theory. But see theory *src/HOL/ex/Perm_Fragments.thy* for fragments on that.

7.1 Abstract type of permutations

typedef *'a perm* = $\{f :: 'a \Rightarrow 'a.\ bij\ f \wedge finite\ \{a.\ f\ a \neq a\}\}$
  morphisms *apply Perm*
proof
  show *id* ∈ *?perm* **by** *simp*
qed

setup_lifting *type_definition_perm*

notation *apply* (**infixl** ‹⟨$⟩› *999*)

lemma *bij_apply* [*simp*]:
  bij (*apply f*)
  using *apply* [*of f*] **by** *simp*

lemma *perm_eqI*:
  assumes $\bigwedge a.\ f\ \langle\$\rangle\ a = g\ \langle\$\rangle\ a$
  shows $f = g$
  using *assms* **by** *transfer* (*simp add*: *fun_eq_iff*)

lemma *perm_eq_iff*:
  $f = g \longleftrightarrow (\forall\, a.\ f\ \langle\$\rangle\ a = g\ \langle\$\rangle\ a)$
  by (*auto intro*: *perm_eqI*)

lemma *apply_inj*:
  $f\ \langle\$\rangle\ a = f\ \langle\$\rangle\ b \longleftrightarrow a = b$
  by (*rule inj_eq*) (*rule bij_is_inj*, *simp*)

lift_definition *affected* :: *'a perm* ⇒ *'a set*
  is $\lambda f.\ \{a.\ f\ a \neq a\}$ **.**

lemma *in_affected*:
  $a \in affected\ f \longleftrightarrow f\ \langle\$\rangle\ a \neq a$
  by *transfer simp*

lemma *finite_affected* [*simp*]:
  finite (*affected f*)
  by *transfer simp*

lemma *apply_affected* [*simp*]:
  $f\ \langle\$\rangle\ a \in affected\ f \longleftrightarrow a \in affected\ f$
proof *transfer*
  fix $f :: 'a \Rightarrow 'a$ **and** $a :: 'a$
  assume $bij\ f \wedge finite\ \{b.\ f\ b \neq b\}$

then have *bij f* **by** *simp*
interpret *bijection f* **by** *standard* (*rule* ‹*bij f*›)
have $f\ a \in \{a.\ f\ a = a\} \longleftrightarrow a \in \{a.\ f\ a = a\}$ (**is** *?P* $\longleftrightarrow$ *?Q*)
by *auto*
then show $f\ a \in \{a.\ f\ a \neq a\} \longleftrightarrow a \in \{a.\ f\ a \neq a\}$
by *simp*
qed

lemma *card_affected_not_one*:
card (*affected f*) $\neq$ *1*
proof
interpret *bijection apply f*
by *standard* (*rule bij_apply*)
assume *card* (*affected f*) = *1*
then obtain *a* **where** ∗: *affected f* = {*a*}
by (*rule card_1_singletonE*)
then have ∗∗: $f\ \langle\$\rangle\ a \neq a$
by (*simp flip*: *in_affected*)
with ∗ **have** $f\ \langle\$\rangle\ a \notin$ *affected f*
by *simp*
then have $f\ \langle\$\rangle\ (f\ \langle\$\rangle\ a) = f\ \langle\$\rangle\ a$
by (*simp add*: *in_affected*)
then have *inv* (*apply f*) $(f\ \langle\$\rangle\ (f\ \langle\$\rangle\ a))$ = *inv* (*apply f*) $(f\ \langle\$\rangle\ a)$
by *simp*
with ∗∗ **show** *False* **by** *simp*
qed

7.2 Identity, composition and inversion

instantiation *Perm.perm* :: (*type*) {*monoid_mult*, *inverse*}
begin

lift_definition *one_perm* :: *'a perm*
is *id*
by *simp*

lemma *apply_one* [*simp*]:
apply 1 = *id*
by (*fact one_perm.rep_eq*)

lemma *affected_one* [*simp*]:
affected 1 = {}
by *transfer simp*

lemma *affected_empty_iff* [*simp*]:
affected f = {} $\longleftrightarrow$ *f* = *1*
by *transfer auto*

lift_definition *times_perm* :: *'a perm* ⇒ *'a perm* ⇒ *'a perm*

```
  is comp
proof
  fix f g :: 'a ⇒ 'a
  assume bij f ∧ finite {a. f a ≠ a}
    bij g ∧finite {a. g a ≠ a}
  then have finite ({a. f a ≠ a} ∪ {a. g a ≠ a})
    by simp
  moreover have {a. (f ∘ g) a ≠ a} ⊆ {a. f a ≠ a} ∪ {a. g a ≠ a}
    by auto
  ultimately show finite {a. (f ∘ g) a ≠ a}
    by (auto intro: finite_subset)
qed (auto intro: bij_comp)

lemma apply_times:
  apply (f ∗ g) = apply f ∘ apply g
  by (fact times_perm.rep_eq)

lemma apply_sequence:
  f ⟨$⟩ (g ⟨$⟩ a) = apply (f ∗ g) a
  by (simp add: apply_times)

lemma affected_times [simp]:
  affected (f ∗ g) ⊆ affected f ∪ affected g
  by transfer auto

lift_definition inverse_perm :: 'a perm ⇒ 'a perm
  is inv
proof transfer
  fix f :: 'a ⇒ 'a and a
  assume bij f ∧ finite {b. f b ≠ b}
  then have bij f and fin: finite {b. f b ≠ b}
    by auto
  interpret bijection f by standard (rule ‹bij f›)
  from fin show bij (inv f) ∧ finite {a. inv f a ≠ a}
    by (simp add: bij_inv)
qed

instance
  by standard (transfer; simp add: comp_assoc)+

end

lemma apply_inverse:
  apply (inverse f) = inv (apply f)
  by (fact inverse_perm.rep_eq)

lemma affected_inverse [simp]:
  affected (inverse f) = affected f
proof transfer
```

```
  fix f :: 'a ⇒ 'a and a
  assume bij f ∧ finite {b. f b ≠ b}
  then have bij f by simp
  interpret bijection f by standard (rule ‹bij f›)
  show {a. inv f a ≠ a} = {a. f a ≠ a}
    by simp
qed

global_interpretation perm: group times 1::'a perm inverse
proof
  fix f :: 'a perm
  show 1 * f = f
    by transfer simp
  show inverse f * f = 1
  proof transfer
    fix f :: 'a ⇒ 'a and a
    assume bij f ∧ finite {b. f b ≠ b}
    then have bij f by simp
    interpret bijection f by standard (rule ‹bij f›)
    show inv f ∘ f = id
      by simp
  qed
qed

declare perm.inverse_distrib_swap [simp]

lemma perm_mult_commute:
  assumes affected f ∩ affected g = {}
  shows g * f = f * g
proof (rule perm_eqI)
  fix a
  from assms have *: a ∈ affected f ⟹ a ∉ affected g
    a ∈ affected g ⟹ a ∉ affected f for a
    by auto
  consider a ∈ affected f ∧ a ∉ affected g
        ∧ f ⟨$⟩ a ∈ affected f
    | a ∉ affected f ∧ a ∈ affected g
        ∧ f ⟨$⟩ a ∉ affected f
    | a ∉ affected f ∧ a ∉ affected g
    using assms by auto
  then show (g * f) ⟨$⟩ a = (f * g) ⟨$⟩ a
  proof cases
    case 1
    with * have f ⟨$⟩ a ∉ affected g
      by auto
    with 1 show ?thesis by (simp add: in_affected apply_times)
  next
    case 2
    with * have g ⟨$⟩ a ∉ affected f
```

```
      by auto
    with 2 show ?thesis by (simp add: in_affected apply_times)
  next
    case 3
    then show ?thesis by (simp add: in_affected apply_times)
  qed
qed

lemma apply_power:
  apply (f ^ n) = apply f ^^ n
  by (induct n) (simp_all add: apply_times)

lemma perm_power_inverse:
  inverse f ^ n = inverse ((f :: 'a perm) ^ n)
proof (induct n)
  case 0 then show ?case by simp
next
  case (Suc n)
  then show ?case
    unfolding power_Suc2 [of f] by simp
qed
```

7.3 Orbit and order of elements

```
definition orbit :: 'a perm ⇒ 'a ⇒ 'a set
where
  orbit f a = range (λn. (f ^ n) ⟨$⟩ a)

lemma in_orbitI:
  assumes (f ^ n) ⟨$⟩ a = b
  shows b ∈ orbit f a
  using assms by (auto simp add: orbit_def)

lemma apply_power_self_in_orbit [simp]:
  (f ^ n) ⟨$⟩ a ∈ orbit f a
  by (rule in_orbitI) rule

lemma in_orbit_self [simp]:
  a ∈ orbit f a
  using apply_power_self_in_orbit [of _ 0] by simp

lemma apply_self_in_orbit [simp]:
  f ⟨$⟩ a ∈ orbit f a
  using apply_power_self_in_orbit [of _ 1] by simp

lemma orbit_not_empty [simp]:
  orbit f a ≠ {}
  using in_orbit_self [of a f] by blast
```

lemma *not_in_affected_iff_orbit_eq_singleton*:
  $a \notin$ *affected f* $\longleftrightarrow$ *orbit f a* $= \{a\}$ (**is** *?P* $\longleftrightarrow$ *?Q*)
proof
  assume *?P*
  then have $f \langle\$\rangle\ a = a$
    by (*simp add*: *in_affected*)
  then have $(f \frown n) \langle\$\rangle\ a = a$ **for** n
    by (*induct n*) (*simp_all add*: *apply_times*)
  then show *?Q*
    by (*auto simp add*: *orbit_def*)
next
  assume *?Q*
  then show *?P*
    by (*auto simp add*: *orbit_def in_affected dest*: *range_eq_singletonD* [*of* _ _
1])
qed

definition *order* :: *'a perm* $\Rightarrow$ *'a* $\Rightarrow$ *nat*
where
  order f $=$ *card* $\circ$ *orbit f*

lemma *orbit_subset_eq_affected*:
  assumes $a \in$ *affected f*
  shows *orbit f a* $\subseteq$ *affected f*
proof (*rule ccontr*)
  assume $\neg$ *orbit f a* $\subseteq$ *affected f*
  then obtain b **where** $b \in$ *orbit f a* **and** $b \notin$ *affected f*
    by *auto*
  then have $b \in$ *range* $(\lambda n.\ (f \frown n) \langle\$\rangle\ a)$
    by (*simp add*: *orbit_def*)
  then obtain n **where** $b = (f \frown n) \langle\$\rangle\ a$
    by *blast*
  with ‹$b \notin$ *affected f*›
  have $(f \frown n) \langle\$\rangle\ a \notin$ *affected f*
    by *simp*
  then have $f \langle\$\rangle\ a \notin$ *affected f*
    by (*induct n*) (*simp_all add*: *apply_times*)
  with *assms* **show** *False*
    by *simp*
qed

lemma *finite_orbit* [*simp*]:
  finite (*orbit f a*)
proof (*cases* $a \in$ *affected f*)
  case *False* **then show** *?thesis*
    by (*simp add*: *not_in_affected_iff_orbit_eq_singleton*)
next
  case *True* **then have** *orbit f a* $\subseteq$ *affected f*
    by (*rule orbit_subset_eq_affected*)

```
  then show ?thesis using finite_affected
    by (rule finite_subset)
qed

lemma orbit_1 [simp]:
  orbit 1 a = {a}
  by (auto simp add: orbit_def)

lemma order_1 [simp]:
  order 1 a = 1
  unfolding order_def by simp

lemma card_orbit_eq [simp]:
  card (orbit f a) = order f a
  by (simp add: order_def)

lemma order_greater_zero [simp]:
  order f a > 0
  by (simp only: card_gt_0_iff order_def comp_def) simp

lemma order_eq_one_iff:
  order f a = Suc 0 ⟷ a ∉ affected f (is ?P ⟷ ?Q)
proof
  assume ?P then have card (orbit f a) = 1
    by simp
  then obtain b where orbit f a = {b}
    by (rule card_1_singletonE)
  with in_orbit_self [of a f]
    have b = a by simp
  with ‹orbit f a = {b}› show ?Q
    by (simp add: not_in_affected_iff_orbit_eq_singleton)
next
  assume ?Q
  then have orbit f a = {a}
    by (simp add: not_in_affected_iff_orbit_eq_singleton)
  then have card (orbit f a) = 1
    by simp
  then show ?P
    by simp
qed

lemma order_greater_eq_two_iff:
  order f a ≥ 2 ⟷ a ∈ affected f
  using order_eq_one_iff [of f a]
  apply (auto simp add: neq_iff)
  using order_greater_zero [of f a]
  apply simp
  done
```

```
lemma order_less_eq_affected:
  assumes f ≠ 1
  shows order f a ≤ card (affected f)
proof (cases a ∈ affected f)
  from assms have affected f ≠ {}
    by simp
  then obtain B b where affected f = insert b B
    by blast
  with finite_affected [of f] have card (affected f) ≥ 1
    by (simp add: card.insert_remove)
  case False then have order f a = 1
    by (simp add: order_eq_one_iff)
  with ‹card (affected f) ≥ 1› show ?thesis
    by simp
next
  case True
  have card (orbit f a) ≤ card (affected f)
   by (rule card_mono) (simp_all add: True orbit_subset_eq_affected card_mono)
  then show ?thesis
    by simp
qed

lemma affected_order_greater_eq_two:
  assumes a ∈ affected f
  shows order f a ≥ 2
proof (rule ccontr)
  assume ¬ 2 ≤ order f a
  then have order f a < 2
    by (simp add: not_le)
  with order_greater_zero [of f a] have order f a = 1
    by arith
  with assms show False
    by (simp add: order_eq_one_iff)
qed

lemma order_witness_unfold:
  assumes n > 0 and (f ^ n) ⟨$⟩ a = a
  shows order f a = card ((λm. (f ^ m) ⟨$⟩ a) ‘ {0..<n})
proof –
  have orbit f a = (λm. (f ^ m) ⟨$⟩ a) ‘ {0..<n} (is _ = ?B)
  proof (rule set_eqI, rule)
    fix b
    assume b ∈ orbit f a
    then obtain m where (f ^ m) ⟨$⟩ a = b
      by (auto simp add: orbit_def)
    then have b = (f ^ (m mod n + n ∗ (m div n))) ⟨$⟩ a
      by simp
    also have . . . = (f ^ (m mod n)) ⟨$⟩ ((f ^ (n ∗ (m div n))) ⟨$⟩ a)
      by (simp only: power_add apply_times) simp
```

also have $(f\ \widehat{}\ (n * q))\ \langle\$\rangle\ a = a$ **for** q
by $(induct\ q)$
$(simp\_all\ add:\ power\_add\ apply\_times\ assms)$
finally have $b = (f\ \widehat{}\ (m\ mod\ n))\ \langle\$\rangle\ a$ **.**
moreover from ‹$n > 0$›
have $m\ mod\ n < n$
by *simp*
ultimately show $b \in ?B$
by *auto*
next
fix b
assume $b \in ?B$
then obtain m **where** $(f\ \widehat{}\ m)\ \langle\$\rangle\ a = b$
by *blast*
then show $b \in orbit\ f\ a$
by $(rule\ in\_orbitI)$
qed
then have $card\ (orbit\ f\ a) = card\ ?B$
by $(simp\ only:)$
then show *?thesis*
by *simp*
qed

lemma *inj_on_apply_range*:
$inj\_on\ (\lambda m.\ (f\ \widehat{}\ m)\ \langle\$\rangle\ a)\ \{..<order\ f\ a\}$
proof −
have $inj\_on\ (\lambda m.\ (f\ \widehat{}\ m)\ \langle\$\rangle\ a)\ \{..<n\}$
if $n \leq order\ f\ a$ **for** n
using *that* **proof** $(induct\ n)$
case *0* **then show** *?case* **by** *simp*
next
case $(Suc\ n)$
then have $prem:\ n < order\ f\ a$
by *simp*
with *Suc.hyps* **have** $hyp:\ inj\_on\ (\lambda m.\ (f\ \widehat{}\ m)\ \langle\$\rangle\ a)\ \{..<n\}$
by *simp*
have $(f\ \widehat{}\ n)\ \langle\$\rangle\ a \notin (\lambda m.\ (f\ \widehat{}\ m)\ \langle\$\rangle\ a)\ `\ \{..<n\}$
proof
assume $(f\ \widehat{}\ n)\ \langle\$\rangle\ a \in (\lambda m.\ (f\ \widehat{}\ m)\ \langle\$\rangle\ a)\ `\ \{..<n\}$
then obtain m **where** $*:\ (f\ \widehat{}\ m)\ \langle\$\rangle\ a = (f\ \widehat{}\ n)\ \langle\$\rangle\ a$ **and** $m < n$
by *auto*
interpret *bijection apply* $(f\ \widehat{}\ m)$
by *standard simp*
from ‹$m < n$› **have** $n = m + (n - m)$
and $nm:\ 0 < n - m\ n - m \leq n$
by *arith+*
with $*$ **have** $(f\ \widehat{}\ m)\ \langle\$\rangle\ a = (f\ \widehat{}\ (m + (n - m)))\ \langle\$\rangle\ a$
by *simp*
then have $(f\ \widehat{}\ m)\ \langle\$\rangle\ a = (f\ \widehat{}\ m)\ \langle\$\rangle\ ((f\ \widehat{}\ (n - m))\ \langle\$\rangle\ a)$

```
        by (simp add: power_add apply_times)
      then have (f ^ (n − m)) ⟨$⟩ a = a
        by simp
      with ‹n − m > 0›
      have order f a = card ((λm. (f ^ m) ⟨$⟩ a) ‘ {0..<n − m})
         by (rule order_witness_unfold)
      also have card ((λm. (f ^ m) ⟨$⟩ a) ‘ {0..<n − m}) ≤ card {0..<n − m}
        by (rule card_image_le) simp
      finally have order f a ≤ n − m
        by simp
      with prem show False by simp
    qed
    with hyp show ?case
      by (simp add: lessThan_Suc)
  qed
  then show ?thesis by simp
qed

lemma orbit_unfold_image:
  orbit f a = (λn. (f ^ n) ⟨$⟩ a) ‘ {..<order f a} (is _ = ?A)
proof (rule sym, rule card_subset_eq)
  show finite (orbit f a)
    by simp
  show ?A ⊆ orbit f a
    by (auto simp add: orbit_def)
  from inj_on_apply_range [of f a]
  have card ?A = order f a
    by (auto simp add: card_image)
  then show card ?A = card (orbit f a)
    by simp
qed

lemma in_orbitE:
  assumes b ∈ orbit f a
  obtains n where b = (f ^ n) ⟨$⟩ a and n < order f a
  using assms unfolding orbit_unfold_image by blast

lemma apply_power_order [simp]:
  (f ^ order f a) ⟨$⟩ a = a
proof −
  have (f ^ order f a) ⟨$⟩ a ∈ orbit f a
    by simp
  then obtain n where
    ∗: (f ^ order f a) ⟨$⟩ a = (f ^ n) ⟨$⟩ a
    and n < order f a
    by (rule in_orbitE)
  show ?thesis
  proof (cases n)
    case 0 with ∗ show ?thesis by simp
```

next
  case (*Suc m*)
  from *order_greater_zero* [*of f a*]
    have *Suc* (*order f a* − *1*) = *order f a*
    by *arith*
  from *Suc* ‹*n* < *order f a*›
    have *m* < *order f a*
    by *simp*
  with *Suc* ∗
  have (*inverse f*) ⟨$⟩ ((*f* ^ *Suc* (*order f a* − *1*)) ⟨$⟩ *a*) =
    (*inverse f*) ⟨$⟩ ((*f* ^ *Suc m*) ⟨$⟩ *a*)
    by *simp*
  then have (*f* ^ (*order f a* − *1*)) ⟨$⟩ *a* =
    (*f* ^ *m*) ⟨$⟩ *a*
    by (*simp only*: *power_Suc apply_times*)
      (*simp add*: *apply_sequence mult.assoc* [*symmetric*])
  with *inj_on_apply_range*
  have *order f a* − *1* = *m*
    by (*rule inj_onD*)
      (*simp_all add*: ‹*m* < *order f a*›)
  with *Suc* **have** *n* = *order f a*
    by *auto*
  with ‹*n* < *order f a*›
  show *?thesis* **by** *simp*
qed
qed

lemma *apply_power_left_mult_order* [*simp*]:
  (*f* ^ (*n* ∗ *order f a*)) ⟨$⟩ *a* = *a*
  by (*induct n*) (*simp_all add*: *power_add apply_times*)

lemma *apply_power_right_mult_order* [*simp*]:
  (*f* ^ (*order f a* ∗ *n*)) ⟨$⟩ *a* = *a*
  by (*simp add*: *ac_simps*)

lemma *apply_power_mod_order_eq* [*simp*]:
  (*f* ^ (*n mod order f a*)) ⟨$⟩ *a* = (*f* ^ *n*) ⟨$⟩ *a*
proof −
  have (*f* ^ *n*) ⟨$⟩ *a* = (*f* ^ (*n mod order f a* + *order f a* ∗ (*n div order f a*))) ⟨$⟩ *a*
    by *simp*
  also have … = (*f* ^ (*n mod order f a*) ∗ *f* ^ (*order f a* ∗ (*n div order f a*))) ⟨$⟩ *a*
    by (*simp flip*: *power_add*)
  finally show *?thesis*
    by (*simp add*: *apply_times*)
qed

lemma *apply_power_eq_iff*:
  (*f* ^ *m*) ⟨$⟩ *a* = (*f* ^ *n*) ⟨$⟩ *a* ⟷ *m mod order f a* = *n mod order f a* (**is** *?P* ⟷ *?Q*)

proof
  assume *?Q*
  then have $(f \frown (m\ mod\ order\ f\ a))\ \langle\$\rangle\ a = (f \frown (n\ mod\ order\ f\ a))\ \langle\$\rangle\ a$
    by *simp*
  then show *?P*
    by *simp*
next
  assume *?P*
  then have $(f \frown (m\ mod\ order\ f\ a))\ \langle\$\rangle\ a = (f \frown (n\ mod\ order\ f\ a))\ \langle\$\rangle\ a$
    by *simp*
  with *inj_on_apply_range*
  show *?Q*
    by (*rule inj_onD*) *simp_all*
qed

lemma *apply_inverse_eq_apply_power_order_minus_one*:
  $(inverse\ f)\ \langle\$\rangle\ a = (f \frown (order\ f\ a - 1))\ \langle\$\rangle\ a$
proof (*cases order f a*)
  case *0* **with** *order_greater_zero* [*of f a*] **show** *?thesis*
    by *simp*
next
  case (*Suc n*)
  moreover have $(f \frown order\ f\ a)\ \langle\$\rangle\ a = a$
    by *simp*
  then have ∗: $(inverse\ f)\ \langle\$\rangle\ ((f \frown order\ f\ a)\ \langle\$\rangle\ a) = (inverse\ f)\ \langle\$\rangle\ a$
    by *simp*
  ultimately show *?thesis*
    by (*simp add*: *apply_sequence mult.assoc* [*symmetric*])
qed

lemma *apply_inverse_self_in_orbit* [*simp*]:
  $(inverse\ f)\ \langle\$\rangle\ a \in orbit\ f\ a$
  using *apply_inverse_eq_apply_power_order_minus_one* [*symmetric*]
  by (*rule in_orbitI*)

lemma *apply_inverse_power_eq*:
  $(inverse\ (f \frown n))\ \langle\$\rangle\ a = (f \frown (order\ f\ a - n\ mod\ order\ f\ a))\ \langle\$\rangle\ a$
proof (*induct n*)
  case *0* **then show** *?case* **by** *simp*
next
  case (*Suc n*)
  define *m* **where** $m = order\ f\ a - n\ mod\ order\ f\ a - 1$
  moreover have $order\ f\ a - n\ mod\ order\ f\ a > 0$
    by *simp*
  ultimately have ∗: $order\ f\ a - n\ mod\ order\ f\ a = Suc\ m$
    by *arith*
  moreover from ∗ **have** *m2*: $order\ f\ a - Suc\ n\ mod\ order\ f\ a = (if\ m = 0\ then\ order\ f\ a\ else\ m)$
    by (*auto simp add*: *mod_Suc*)

```
  ultimately show ?case
    using Suc
      by (simp_all add: apply_times power_Suc2 [of _ n] power_Suc [of _ m] del:
power_Suc)
        (simp add: apply_sequence mult.assoc [symmetric])
qed

lemma apply_power_eq_self_iff:
  (f ^ n) ⟨$⟩ a = a ⟷ order f a dvd n
  using apply_power_eq_iff [of f n a 0]
    by (simp add: mod_eq_0_iff_dvd)

lemma orbit_equiv:
  assumes b ∈ orbit f a
  shows orbit f b = orbit f a (is ?B = ?A)
proof
  from assms obtain n where n < order f a and b: b = (f ^ n) ⟨$⟩ a
    by (rule in_orbitE)
  then show ?B ⊆ ?A
     by (auto simp add: apply_sequence power_add [symmetric] intro: in_orbitI
elim!: in_orbitE)
  from b have (inverse (f ^ n)) ⟨$⟩ b = (inverse (f ^ n)) ⟨$⟩ ((f ^ n) ⟨$⟩ a)
    by simp
  then have a: a = (inverse (f ^ n)) ⟨$⟩ b
    by (simp add: apply_sequence)
  then show ?A ⊆ ?B
   apply (auto simp add: apply_sequence power_add [symmetric] intro: in_orbitI
elim!: in_orbitE)
    unfolding apply_times comp_def apply_inverse_power_eq
    unfolding apply_sequence power_add [symmetric]
    apply (rule in_orbitI) apply rule
    done
qed

lemma orbit_apply [simp]:
  orbit f (f ⟨$⟩ a) = orbit f a
  by (rule orbit_equiv) simp

lemma order_apply [simp]:
  order f (f ⟨$⟩ a) = order f a
  by (simp only: order_def comp_def orbit_apply)

lemma orbit_apply_inverse [simp]:
  orbit f (inverse f ⟨$⟩ a) = orbit f a
  by (rule orbit_equiv) simp

lemma order_apply_inverse [simp]:
  order f (inverse f ⟨$⟩ a) = order f a
  by (simp only: order_def comp_def orbit_apply_inverse)
```

```
lemma orbit_apply_power [simp]:
  orbit f ((f ^ n) ⟨$⟩ a) = orbit f a
  by (rule orbit_equiv) simp

lemma order_apply_power [simp]:
  order f ((f ^ n) ⟨$⟩ a) = order f a
  by (simp only: order_def comp_def orbit_apply_power)

lemma orbit_inverse [simp]:
  orbit (inverse f) = orbit f
proof (rule ext, rule set_eqI, rule)
  fix b a
  assume b ∈ orbit f a
  then obtain n where b: b = (f ^ n) ⟨$⟩ a n < order f a
    by (rule in_orbitE)
  then have b = apply (inverse (inverse f) ^ n) a
    by simp
  then have b = apply (inverse (inverse f ^ n)) a
    by (simp add: perm_power_inverse)
  then have b = apply (inverse f ^ (n * (order (inverse f ^ n) a − 1))) a
   by (simp add: apply_inverse_eq_apply_power_order_minus_one power_mult)
  then show b ∈ orbit (inverse f) a
    by simp
next
  fix b a
  assume b ∈ orbit (inverse f) a
  then show b ∈ orbit f a
    by (rule in_orbitE)
      (simp add: apply_inverse_eq_apply_power_order_minus_one
      perm_power_inverse power_mult [symmetric])
qed

lemma order_inverse [simp]:
  order (inverse f) = order f
  by (simp add: order_def)

lemma orbit_disjoint:
  assumes orbit f a ≠ orbit f b
  shows orbit f a ∩ orbit f b = {}
proof (rule ccontr)
  assume orbit f a ∩ orbit f b ≠ {}
  then obtain c where c ∈ orbit f a ∩ orbit f b
    by blast
  then have c ∈ orbit f a and c ∈ orbit f b
    by auto
  then obtain m n where c = (f ^ m) ⟨$⟩ a
    and c = apply (f ^ n) b by (blast elim!: in_orbitE)
  then have (f ^ m) ⟨$⟩ a = apply (f ^ n) b
```

```
    by simp
  then have apply (inverse f ^ m) ((f ^ m) ⟨$⟩ a) =
    apply (inverse f ^ m) (apply (f ^ n) b)
    by simp
  then have *: apply (inverse f ^ m * f ^ n) b = a
    by (simp add: apply_sequence perm_power_inverse)
  have a ∈ orbit f b
  proof (cases n m rule: linorder_cases)
    case equal with * show ?thesis
      by (simp add: perm_power_inverse)
  next
    case less
    moreover define q where q = m − n
    ultimately have m = q + n by arith
    with * have apply (inverse f ^ q) b = a
      by (simp add: power_add mult.assoc perm_power_inverse)
    then have a ∈ orbit (inverse f) b
      by (rule in_orbitI)
    then show ?thesis
      by simp
  next
    case greater
    moreover define q where q = n − m
    ultimately have n = m + q by arith
    with * have apply (f ^ q) b = a
      by (simp add: power_add mult.assoc [symmetric] perm_power_inverse)
    then show ?thesis
      by (rule in_orbitI)
  qed
  with assms show False
    by (auto dest: orbit_equiv)
qed
```

7.4 Swaps

```
lift_definition swap :: 'a ⇒ 'a ⇒ 'a perm  (‹⟨_ ↔ _⟩›)
  is λa b. transpose a b
proof
  fix a b :: 'a
  have {c. transpose a b c ≠ c} ⊆ {a, b}
    by (auto simp add: transpose_def)
  then show finite {c. transpose a b c ≠ c}
    by (rule finite_subset) simp
qed simp

lemma apply_swap_simp [simp]:
  ⟨a ↔ b⟩ ⟨$⟩ a = b
  ⟨a ↔ b⟩ ⟨$⟩ b = a
  by (transfer; simp)+
```

lemma *apply_swap_same* [*simp*]:
  $c \neq a \Longrightarrow c \neq b \Longrightarrow \langle a \leftrightarrow b\rangle \ \langle\$\rangle \ c = c$
  by *transfer simp*

lemma *apply_swap_eq_iff* [*simp*]:
  $\langle a \leftrightarrow b\rangle \ \langle\$\rangle \ c = a \longleftrightarrow c = b$
  $\langle a \leftrightarrow b\rangle \ \langle\$\rangle \ c = b \longleftrightarrow c = a$
  by (*transfer*; *auto simp add*: *transpose_def*)+

lemma *swap_1* [*simp*]:
  $\langle a \leftrightarrow a\rangle = 1$
  by *transfer simp*

lemma *swap_sym*:
  $\langle b \leftrightarrow a\rangle = \langle a \leftrightarrow b\rangle$
  by (*transfer*; *auto simp add*: *transpose_def*)+

lemma *swap_self* [*simp*]:
  $\langle a \leftrightarrow b\rangle * \langle a \leftrightarrow b\rangle = 1$
  by *transfer simp*

lemma *affected_swap*:
  $a \neq b \Longrightarrow$ *affected* $\langle a \leftrightarrow b\rangle = \{a,\ b\}$
  by *transfer* (*auto simp add*: *transpose_def*)

lemma *inverse_swap* [*simp*]:
  inverse $\langle a \leftrightarrow b\rangle = \langle a \leftrightarrow b\rangle$
  by *transfer* (*auto intro*: *inv_equality*)

7.5 Permutations specified by cycles

fun *cycle* :: *'a list* ⇒ *'a perm* (‹⟨_⟩›)
where
  $\langle[]\rangle = 1$
| $\langle[a]\rangle = 1$
| $\langle a \# b \# as\rangle = \langle a \# as\rangle * \langle a{\leftrightarrow}b\rangle$

We do not continue and restrict ourselves to syntax from here. See also introductory note.

7.6 Syntax

bundle *permutation_syntax*
begin
notation *swap*   (‹⟨_ ↔ _⟩›)
notation *cycle*   (‹⟨_⟩›)
notation *apply* (**infixl** ‹⟨\$⟩› *999*)
end

unbundle *no permutation_syntax*

end

8 Permutation orbits

theory *Orbits*
imports
  HOL−Library.FuncSet
  HOL−Combinatorics.Permutations
begin

8.1 Orbits and cyclic permutations

inductive_set *orbit* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$ *set* **for** *f x* **where**
  base: $f\ x \in$ *orbit f x* |
  step: $y \in$ *orbit f x* $\Longrightarrow f\ y \in$ *orbit f x*

definition *cyclic_on* :: $('a \Rightarrow 'a) \Rightarrow 'a$ *set* $\Rightarrow$ *bool* **where**
  cyclic_on f S $\longleftrightarrow$ $(\exists\, s{\in}S.\ S =$ *orbit f s*$)$

lemma *orbit_altdef*: *orbit f x* $= \{(f \verb|^^| n)\ x \mid n.\ 0 < n\}$ (**is** $?L = ?R$)
proof (*intro set_eqI iffI*)
  fix *y* **assume** $y \in ?L$ **then show** $y \in ?R$
    by (*induct rule*: *orbit.induct*) (*auto simp*: *exI*[**where** *x=1*] *exI*[**where** *x=Suc n* **for** *n*])
next
  fix *y* **assume** $y \in ?R$
  then obtain *n* **where** $y = (f \verb|^^| n)\ x\ 0 < n$ **by** *blast*
  then show $y \in ?L$
  proof (*induction n arbitrary*: *y*)
    case (*Suc n*) **then show** *?case* **by** (*cases n = 0*) (*auto intro*: *orbit.intros*)
  qed *simp*
qed

lemma *orbit_trans*:
  assumes $s \in$ *orbit f t* $t \in$ *orbit f u* **shows** $s \in$ *orbit f u*
  using *assms* **by** *induct* (*auto intro*: *orbit.intros*)

lemma *orbit_subset*:
  assumes $s \in$ *orbit f* $(f\ t)$ **shows** $s \in$ *orbit f t*
  using *assms* **by** (*induct*) (*auto intro*: *orbit.intros*)

lemma *orbit_sim_step*:
  assumes $s \in$ *orbit f t* **shows** $f\ s \in$ *orbit f* $(f\ t)$
  using *assms* **by** *induct* (*auto intro*: *orbit.intros*)

lemma *orbit_step*:

```
  assumes y ∈ orbit f x f x ≠ y shows y ∈ orbit f (f x)
  using assms
proof induction
  case (step y) then show ?case by (cases x = y) (auto intro: orbit.intros)
qed simp

lemma self_in_orbit_trans:
  assumes s ∈ orbit f s t ∈ orbit f s shows t ∈ orbit f t
  using assms(2,1) by induct (auto intro: orbit_sim_step)

lemma orbit_swap:
  assumes s ∈ orbit f s t ∈ orbit f s shows s ∈ orbit f t
  using assms(2,1)
proof induction
  case base then show ?case by (cases f s = s) (auto intro: orbit_step)
next
  case (step x) then show ?case by (cases f x = s) (auto intro: orbit_step)
qed

lemma permutation_self_in_orbit:
  assumes permutation f shows s ∈ orbit f s
  unfolding orbit_altdef using permutation_self[OF assms, of s] by simp metis

lemma orbit_altdef_self_in:
  assumes s ∈ orbit f s shows orbit f s = {(f ^^ n) s | n. True}
proof (intro set_eqI iffI)
  fix x assume x ∈ {(f ^^ n) s | n. True}
  then obtain n where x = (f ^^ n) s by auto
  then show x ∈ orbit f s using assms by (cases n = 0) (auto simp: orbit_altdef)
qed (auto simp: orbit_altdef)

lemma orbit_altdef_permutation:
  assumes permutation f shows orbit f s = {(f ^^ n) s | n. True}
  using assms by (intro orbit_altdef_self_in permutation_self_in_orbit)

lemma orbit_altdef_bounded:
  assumes (f ^^ n) s = s 0 < n shows orbit f s = {(f ^^ m) s| m. m < n}
proof −
  from assms have s ∈ orbit f s
    by (auto simp add: orbit_altdef) metis
  then have orbit f s = {(f ^^ m) s|m. True} by (rule orbit_altdef_self_in)
  also have ... = {(f ^^ m) s| m. m < n}
    using assms
    by (auto simp: funpow_mod_eq intro: exI[where x=m mod n for m])
  finally show ?thesis .
qed

lemma funpow_in_orbit:
  assumes s ∈ orbit f t shows (f ^^ n) s ∈ orbit f t
```

```
  using assms by (induct n) (auto intro: orbit.intros)

lemma finite_orbit:
  assumes s ∈ orbit f s shows finite (orbit f s)
proof –
  from assms obtain n where n: 0 < n (f ^^ n) s = s
    by (auto simp: orbit_altdef)
  then show ?thesis by (auto simp: orbit_altdef_bounded)
qed

lemma self_in_orbit_step:
  assumes s ∈ orbit f s shows orbit f (f s) = orbit f s
proof (intro set_eqI iffI)
  fix t assume t ∈ orbit f s then show t ∈ orbit f (f s)
    using assms by (auto intro: orbit_step orbit_sim_step)
qed (auto intro: orbit_subset)

lemma permutation_orbit_step:
  assumes permutation f shows orbit f (f s) = orbit f s
  using assms by (intro self_in_orbit_step permutation_self_in_orbit)

lemma orbit_nonempty:
  orbit f s ≠ {}
  using orbit.base by fastforce

lemma orbit_inv_eq:
  assumes permutation f
  shows orbit (inv f) x = orbit f x (is ?L = ?R)
proof –
  { fix g y assume A: permutation g y ∈ orbit (inv g) x
    have y ∈ orbit g x
    proof –
      have inv_g: ⋀y. x = g y ⟹ inv g x = y ⋀y. inv g (g y) = y
        by (metis A(1) bij_inv_eq_iff permutation_bijective)+

      { fix y assume y ∈ orbit g x
        then have inv g y ∈ orbit g x
          by (cases) (simp_all add: inv_g A(1) permutation_self_in_orbit)
      } note inv_g_in_orb = this

      from A(2) show ?thesis
        by induct (simp_all add: inv_g_in_orb A permutation_self_in_orbit)
    qed
  } note orb_inv_ss = this

  have inv (inv f) = f
    by (simp add: assms inv_inv_eq permutation_bijective)
  then show ?thesis
    using orb_inv_ss[OF assms] orb_inv_ss[OF permutation_inverse[OF assms]]
```

```
  by auto
qed

lemma cyclic_on_alldef:
  cyclic_on f S ⟷ S ≠ {} ∧ (∀s∈S. S = orbit f s)
  unfolding cyclic_on_def by (auto intro: orbit.step orbit_swap orbit_trans)

lemma cyclic_on_funpow_in:
  assumes cyclic_on f S s ∈ S shows (f^^n) s ∈ S
  using assms unfolding cyclic_on_def by (auto intro: funpow_in_orbit)

lemma finite_cyclic_on:
  assumes cyclic_on f S shows finite S
  using assms by (auto simp: cyclic_on_def finite_orbit)

lemma cyclic_on_singleI:
  assumes s ∈ S S = orbit f s shows cyclic_on f S
  using assms unfolding cyclic_on_def by blast

lemma cyclic_on_inI:
  assumes cyclic_on f S s ∈ S shows f s ∈ S
  using assms by (auto simp: cyclic_on_def intro: orbit.intros)

lemma orbit_inverse:
  assumes self: a ∈ orbit g a
    and eq: ⋀x. x ∈ orbit g a ⟹ g' (f x) = f (g x)
  shows f ` orbit g a = orbit g' (f a) (is ?L = ?R)
proof (intro set_eqI iffI)
  fix x assume x ∈ ?L
  then obtain x0 where x0 ∈ orbit g a x = f x0 by auto
  then show x ∈ ?R
  proof (induct arbitrary: x)
    case base then show ?case by (auto simp: self orbit.base eq[symmetric])
  next
    case step then show ?case by cases (auto simp: eq[symmetric] orbit.intros)
  qed
next
  fix x assume x ∈ ?R
  then show x ∈ ?L
  proof (induct arbitrary: )
    case base then show ?case by (auto simp: self orbit.base eq)
  next
    case step then show ?case by cases (auto simp: eq orbit.intros)
  qed
qed

lemma cyclic_on_image:
  assumes cyclic_on f S
  assumes ⋀x. x ∈ S ⟹ g (h x) = h (f x)
```

```
  shows cyclic_on g (h ‘ S)
  using assms by (auto simp: cyclic_on_def) (meson orbit_inverse)

lemma cyclic_on_f_in:
  assumes f permutes S cyclic_on f A f x ∈ A
  shows x ∈ A
proof −
  from assms have fx_in_orb: f x ∈ orbit f (f x) by (auto simp: cyclic_on_alldef)
  from assms have A = orbit f (f x) by (auto simp: cyclic_on_alldef)
  moreover
  then have ... = orbit f x using ‹f x ∈ A› by (auto intro: orbit_step orbit_subset)
  ultimately
   show ?thesis by (metis (no_types) orbit.simps permutes_inverses(2)[OF assms(1)])
qed

lemma orbit_cong0:
  assumes x ∈ A f ∈ A → A ⋀y. y ∈ A ⟹ f y = g y shows orbit f x = orbit g
x
proof −
  { fix n have (f ^^ n) x = (g ^^ n) x ∧ (f ^^ n) x ∈ A
      by (induct n rule: nat.induct) (insert assms, auto)
  } then show ?thesis by (auto simp: orbit_altdef)
qed

lemma orbit_cong:
  assumes self_in: t ∈ orbit f t and eq: ⋀s. s ∈ orbit f t ⟹ g s = f s
  shows orbit g t = orbit f t
  using assms(1) _ assms(2) by (rule orbit_cong0) (auto simp: orbit.step eq)

lemma cyclic_cong:
  assumes ⋀s. s ∈ S ⟹ f s = g s shows cyclic_on f S = cyclic_on g S
proof −
  have (∃s∈S. orbit f s = orbit g s) ⟹ cyclic_on f S = cyclic_on g S
    by (metis cyclic_on_alldef cyclic_on_def)
  then show ?thesis by (metis assms orbit_cong cyclic_on_def)
qed

lemma permutes_comp_preserves_cyclic1:
  assumes g permutes B cyclic_on f C
  assumes A ∩ B = {} C ⊆ A
  shows cyclic_on (f o g) C
proof −
  have ∗: ⋀c. c ∈ C ⟹ f (g c) = f c
    using assms by (subst permutes_not_in [of g]) auto
  with assms(2) show ?thesis by (simp cong: cyclic_cong)
qed

lemma permutes_comp_preserves_cyclic2:
  assumes f permutes A cyclic_on g C
```

```
  assumes A ∩ B = {} C ⊆ B
  shows cyclic_on (f o g) C
proof −
  obtain c where c: c ∈ C C = orbit g c c ∈ orbit g c
    using ‹cyclic_on g C› by (auto simp: cyclic_on_def)
  then have ⋀c. c ∈ C ⟹ f (g c) = g c
    using assms c by (subst permutes_not_in [of f]) (auto intro: orbit.intros)
  with assms(2) show ?thesis by (simp cong: cyclic_cong)
qed

lemma permutes_orbit_subset:
  assumes f permutes S x ∈ S shows orbit f x ⊆ S
proof
  fix y assume y ∈ orbit f x
  then show y ∈ S by induct (auto simp: permutes_in_image assms)
qed

lemma cyclic_on_orbit':
  assumes permutation f shows cyclic_on f (orbit f x)
  unfolding cyclic_on_alldef using orbit_nonempty[of f x]
  by (auto intro: assms orbit_swap orbit_trans permutation_self_in_orbit)

lemma cyclic_on_orbit:
  assumes f permutes S finite S shows cyclic_on f (orbit f x)
  using assms by (intro cyclic_on_orbit') (auto simp: permutation_permutes)

lemma orbit_cyclic_eq3:
  assumes cyclic_on f S y ∈ S shows orbit f y = S
  using assms unfolding cyclic_on_alldef by simp

lemma orbit_eq_singleton_iff: orbit f x = {x} ⟷ f x = x (is ?L ⟷ ?R)
proof
  assume A: ?R
  { fix y assume y ∈ orbit f x then have y = x
      by induct (auto simp: A)
  } then show ?L by (metis orbit_nonempty singletonI subsetI subset_singletonD)
next
  assume A: ?L
  then have ⋀y. y ∈ orbit f x ⟹ f x = y
    by − (erule orbit.cases, simp_all)
  then show ?R using A by blast
qed

lemma eq_on_cyclic_on_iff1:
  assumes cyclic_on f S x ∈ S
  obtains f x ∈ S f x = x ⟷ card S = 1
proof
  from assms show f x ∈ S by (auto simp: cyclic_on_def intro: orbit.intros)
  from assms have S = orbit f x by (auto simp: cyclic_on_alldef)
```

then have $f\ x = x \longleftrightarrow S = \{x\}$ **by** (*metis orbit_eq_singleton_iff*)
then show $f\ x = x \longleftrightarrow card\ S = 1$ **using** ‹$x \in S$› **by** (*auto simp*: *card_Suc_eq*)
qed

lemma *orbit_eqI*:
$y = f\ x \Longrightarrow y \in orbit\ f\ x$
$z = f\ y \Longrightarrow y \in orbit\ f\ x \Longrightarrow z \in orbit\ f\ x$
by (*metis orbit.base*) (*metis orbit.step*)

8.2 Decomposition of arbitrary permutations

definition *perm_restrict* :: $('a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow 'a)$ **where**
perm_restrict f S x $\equiv$ *if* $x \in S$ *then f x else x*

lemma *perm_restrict_comp*:
assumes $A \cap B = \{\}$ *cyclic_on f B*
shows *perm_restrict f A o perm_restrict f B* = *perm_restrict f* $(A \cup B)$
proof −
have $\bigwedge x.\ x \in B \Longrightarrow f\ x \in B$ **using** ‹*cyclic_on f B*› **by** (*rule cyclic_on_inI*)
with *assms* **show** *?thesis* **by** (*auto simp*: *perm_restrict_def fun_eq_iff*)
qed

lemma *perm_restrict_simps*:
$x \in S \Longrightarrow$ *perm_restrict f S x* $= f\ x$
$x \notin S \Longrightarrow$ *perm_restrict f S x* $= x$
by (*auto simp*: *perm_restrict_def*)

lemma *perm_restrict_perm_restrict*:
perm_restrict (*perm_restrict f A*) B = *perm_restrict f* $(A \cap B)$
by (*auto simp*: *perm_restrict_def*)

lemma *perm_restrict_union*:
assumes *perm_restrict f A permutes A perm_restrict f B permutes B* $A \cap B = \{\}$
shows *perm_restrict f A o perm_restrict f B* = *perm_restrict f* $(A \cup B)$
using *assms* **by** (*auto simp*: *fun_eq_iff perm_restrict_def permutes_def*) (*metis Diff_iff Diff_triv*)

lemma *perm_restrict_id*[*simp*]:
assumes *f permutes S* **shows** *perm_restrict f S* $= f$
using *assms* **by** (*auto simp*: *permutes_def perm_restrict_def*)

lemma *cyclic_on_perm_restrict*:
cyclic_on (*perm_restrict f S*) $S \longleftrightarrow$ *cyclic_on f S*
by (*simp add*: *perm_restrict_def cong*: *cyclic_cong*)

lemma *perm_restrict_diff_cyclic*:
assumes *f permutes S cyclic_on f A*
shows *perm_restrict f* $(S - A)$ *permutes* $(S - A)$

```
proof -
  { fix y
    have ∃x. perm_restrict f (S - A) x = y
    proof cases
      assume A: y ∈ S - A
      with ‹f permutes S› obtain x where f x = y x ∈ S
        unfolding permutes_def by auto metis
      moreover
      with A have x ∉ A by (metis Diff_iff assms(2) cyclic_on_inI)
      ultimately
      have perm_restrict f (S - A) x = y  by (simp add: perm_restrict_simps)
      then show ?thesis ..
    next
      assume y ∉ S - A
     then have perm_restrict f (S - A) y = y by (simp add: perm_restrict_simps)
      then show ?thesis ..
    qed
  } note X = this

  { fix x y assume perm_restrict f (S - A) x = perm_restrict f (S - A) y
    with assms have x = y
     by (auto simp: perm_restrict_def permutes_def split: if_splits intro: cyclic_on_f_in)
  } note Y = this

  show ?thesis by (auto simp: permutes_def perm_restrict_simps X intro: Y)
qed

lemma permutes_decompose:
  assumes f permutes S finite S
  shows ∃C. (∀c ∈ C. cyclic_on f c) ∧ ⋃C = S ∧ (∀c1 ∈ C. ∀c2 ∈ C. c1 ≠
c2 ⟶ c1 ∩ c2 = {})
  using assms(2,1)
proof (induction arbitrary: f rule: finite_psubset_induct)
  case (psubset S)

  show ?case
  proof (cases S = {})
    case True then show ?thesis by (intro exI[where x={}]) auto
  next
    case False
    then obtain s where s ∈ S by auto
    with ‹f permutes S› have orbit f s ⊆ S
      by (rule permutes_orbit_subset)
    have cyclic_orbit: cyclic_on f (orbit f s)
      using ‹f permutes S› ‹finite S› by (rule cyclic_on_orbit)

    let ?f' = perm_restrict f (S - orbit f s)

   have f s ∈ S using ‹f permutes S› ‹s ∈ S› by (auto simp: permutes_in_image)
```

then have $S - orbit\ f\ s \subset S$ **using** *orbit.base*[*of f s*] ‹$s \in S$› **by** *blast*
moreover
have *?f′ permutes* $(S - orbit\ f\ s)$
using ‹*f permutes S*› *cyclic_orbit* **by** (*rule perm_restrict_diff_cyclic*)
ultimately
obtain C **where** C: $\bigwedge c.\ c \in C \Longrightarrow cyclic\_on\ ?f'\ c\ \bigcup C = S - orbit\ f\ s$
$\forall c1 \in C.\ \forall c2 \in C.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}$
using *psubset.IH* **by** *metis*

{ fix c **assume** $c \in C$
then have $*$: $\bigwedge x.\ x \in c \Longrightarrow perm\_restrict\ f\ (S - orbit\ f\ s)\ x = f\ x$
using $C(2)$ ‹*f permutes S*› **by** (*auto simp add*: *perm_restrict_def*)
then have *cyclic_on f c* **using** $C(1)$[*OF* ‹$c \in C$›] **by** (*simp cong*: *cyclic_cong add*: $*$)
} **note** *in_C_cyclic* = *this*

have *Un_ins*: $\bigcup(insert\ (orbit\ f\ s)\ C) = S$
using ‹$\bigcup C = \_$› ‹$orbit\ f\ s \subseteq S$› **by** *blast*

have *Disj_ins*: $(\forall c1 \in insert\ (orbit\ f\ s)\ C.\ \forall c2 \in insert\ (orbit\ f\ s)\ C.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\})$
using C **by** *auto*

show *?thesis*
by (*intro conjI Un_ins Disj_ins exI*[**where** *x=insert* (*orbit f s*) *C*])
(*auto simp*: *cyclic_orbit in_C_cyclic*)
qed
qed

8.3 Function-power distance between values

definition *funpow_dist* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow nat$ **where**
$funpow\_dist\ f\ x\ y \equiv LEAST\ n.\ (f\ \hat{}\hat{}\ n)\ x = y$

abbreviation *funpow_dist1* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow nat$ **where**
$funpow\_dist1\ f\ x\ y \equiv Suc\ (funpow\_dist\ f\ (f\ x)\ y)$

lemma *funpow_dist_0*:
assumes $x = y$ **shows** $funpow\_dist\ f\ x\ y = 0$
using *assms* **unfolding** *funpow_dist_def* **by** (*intro Least_eq_0*) *simp*

lemma *funpow_dist_least*:
assumes $n < funpow\_dist\ f\ x\ y$ **shows** $(f\ \hat{}\hat{}\ n)\ x \neq y$
proof (*rule notI*)
assume $(f\ \hat{}\hat{}\ n)\ x = y$
then have $funpow\_dist\ f\ x\ y \leq n$ **unfolding** *funpow_dist_def* **by** (*rule Least_le*)
with *assms* **show** *False* **by** *linarith*
qed

lemma *funpow_dist1_least*:
  assumes $0 < n$ $n < funpow\_dist1\ f\ x\ y$ **shows** $(f \,\widehat{\ }\widehat{\ }\, n)\ x \neq y$
proof (*rule notI*)
  assume $(f \,\widehat{\ }\widehat{\ }\, n)\ x = y$
  then have $(f \,\widehat{\ }\widehat{\ }\, (n - 1))\ (f\ x) = y$
    using ‹$0 < n$› **by** (*cases n*) (*simp_all add*: *funpow_swap1*)
  then have $funpow\_dist\ f\ (f\ x)\ y \leq n - 1$ **unfolding** *funpow_dist_def* **by** (*rule Least_le*)
  with *assms* **show** *False* **by** *simp*
qed

lemma *funpow_dist_prop*:
  $y \in orbit\ f\ x \Longrightarrow (f \,\widehat{\ }\widehat{\ }\, funpow\_dist\ f\ x\ y)\ x = y$
  unfolding *funpow_dist_def* **by** (*rule LeastI_ex*) (*auto simp*: *orbit_altdef*)

lemma *funpow_dist_0_eq*:
  assumes $y \in orbit\ f\ x$ **shows** $funpow\_dist\ f\ x\ y = 0 \longleftrightarrow x = y$
  using *assms* **by** (*auto simp*: *funpow_dist_0 dest*: *funpow_dist_prop*)

lemma *funpow_dist_step*:
  assumes $x \neq y$ $y \in orbit\ f\ x$ **shows** $funpow\_dist\ f\ x\ y = Suc\ (funpow\_dist\ f\ (f\ x)\ y)$
proof −
  from ‹$y \in$ _› **obtain** n **where** $(f \,\widehat{\ }\widehat{\ }\, n)\ x = y$ **by** (*auto simp*: *orbit_altdef*)
  with ‹$x \neq y$› **obtain** n' **where** [*simp*]: $n = Suc\ n'$ **by** (*cases n*) *auto*

  show *?thesis*
    unfolding *funpow_dist_def*
  proof (*rule Least_Suc2*)
    show $(f \,\widehat{\ }\widehat{\ }\, n)\ x = y$ **by** *fact*
    then show $(f \,\widehat{\ }\widehat{\ }\, n')\ (f\ x) = y$ **by** (*simp add*: *funpow_swap1*)
    show $(f \,\widehat{\ }\widehat{\ }\, 0)\ x \neq y$ **using** ‹$x \neq y$› **by** *simp*
    show $\forall k.\ ((f \,\widehat{\ }\widehat{\ }\, Suc\ k)\ x = y) = ((f \,\widehat{\ }\widehat{\ }\, k)\ (f\ x) = y)$
      by (*simp add*: *funpow_swap1*)
  qed
qed

lemma *funpow_dist1_prop*:
  assumes $y \in orbit\ f\ x$ **shows** $(f \,\widehat{\ }\widehat{\ }\, funpow\_dist1\ f\ x\ y)\ x = y$
   by (*metis assms funpow_dist_prop funpow_dist_step funpow_simps_right*(*2*) *o_apply self_in_orbit_step*)

lemma *funpow_neq_less_funpow_dist*:
  assumes $y \in orbit\ f\ x$ $m \leq funpow\_dist\ f\ x\ y$ $n \leq funpow\_dist\ f\ x\ y$ $m \neq n$
  shows $(f \,\widehat{\ }\widehat{\ }\, m)\ x \neq (f \,\widehat{\ }\widehat{\ }\, n)\ x$
proof (*rule notI*)
  assume *A*: $(f \,\widehat{\ }\widehat{\ }\, m)\ x = (f \,\widehat{\ }\widehat{\ }\, n)\ x$

define m' n' **where** $m' = min\ m\ n$ **and** $n' = max\ m\ n$
with A *assms* **have** A': $m' < n'$ $(f\ \widehat{}\widehat{}\ m')\ x = (f\ \widehat{}\widehat{}\ n')\ x$ $n' \leq funpow\_dist\ f\ x$ y
  by (*auto simp*: *min_def max_def*)

have $y = (f\ \widehat{}\widehat{}\ funpow\_dist\ f\ x\ y)\ x$
  using ‹$y \in$ _› **by** (*simp only*: *funpow_dist_prop*)
also have $\ldots = (f\ \widehat{}\widehat{}\ ((funpow\_dist\ f\ x\ y - n') + n'))\ x$
  using ‹$n' \leq$ _› **by** *simp*
also have $\ldots = (f\ \widehat{}\widehat{}\ ((funpow\_dist\ f\ x\ y - n') + m'))\ x$
  by (*simp add*: *funpow_add* ‹$(f\ \widehat{}\widehat{}\ m')\ x =$ _›)
also have $(f\ \widehat{}\widehat{}\ ((funpow\_dist\ f\ x\ y - n') + m'))\ x \neq y$
  using A' **by** (*intro funpow_dist_least*) *linarith*
finally show *False* **by** *simp*
qed

lemma *funpow_neq_less_funpow_dist1*:
assumes $y \in orbit\ f\ x$ $m < funpow\_dist1\ f\ x\ y$ $n < funpow\_dist1\ f\ x\ y$ $m \neq n$
shows $(f\ \widehat{}\widehat{}\ m)\ x \neq (f\ \widehat{}\widehat{}\ n)\ x$
proof (*rule notI*)
assume A: $(f\ \widehat{}\widehat{}\ m)\ x = (f\ \widehat{}\widehat{}\ n)\ x$

define m' n' **where** $m' = min\ m\ n$ **and** $n' = max\ m\ n$
with A *assms* **have** A': $m' < n'$ $(f\ \widehat{}\widehat{}\ m')\ x = (f\ \widehat{}\widehat{}\ n')\ x$ $n' < funpow\_dist1\ f$ $x\ y$
  by (*auto simp*: *min_def max_def*)

have $y = (f\ \widehat{}\widehat{}\ funpow\_dist1\ f\ x\ y)\ x$
  using ‹$y \in$ _› **by** (*simp only*: *funpow_dist1_prop*)
also have $\ldots = (f\ \widehat{}\widehat{}\ ((funpow\_dist1\ f\ x\ y - n') + n'))\ x$
  using ‹$n' <$ _› **by** *simp*
also have $\ldots = (f\ \widehat{}\widehat{}\ ((funpow\_dist1\ f\ x\ y - n') + m'))\ x$
  by (*simp add*: *funpow_add* ‹$(f\ \widehat{}\widehat{}\ m')\ x =$ _›)
also have $(f\ \widehat{}\widehat{}\ ((funpow\_dist1\ f\ x\ y - n') + m'))\ x \neq y$
  using A' **by** (*intro funpow_dist1_least*) *linarith+*
finally show *False* **by** *simp*
qed

lemma *inj_on_funpow_dist*:
assumes $y \in orbit\ f\ x$ **shows** *inj_on* $(\lambda n.\ (f\ \widehat{}\widehat{}\ n)\ x)$ $\{0..funpow\_dist\ f\ x\ y\}$
using *funpow_neq_less_funpow_dist*[*OF assms*] **by** (*intro inj_onI*) *auto*

lemma *inj_on_funpow_dist1*:
assumes $y \in orbit\ f\ x$ **shows** *inj_on* $(\lambda n.\ (f\ \widehat{}\widehat{}\ n)\ x)$ $\{0..<funpow\_dist1\ f\ x\ y\}$
using *funpow_neq_less_funpow_dist1*[*OF assms*] **by** (*intro inj_onI*) *auto*

lemma *orbit_conv_funpow_dist1*:
assumes $x \in orbit\ f\ x$

shows *orbit f x* = (λ*n*. (*f* ^^ *n*) *x*) ‘ {*0..<funpow_dist1 f x x*} (**is** *?L* = *?R*)
using *funpow_dist1_prop*[*OF assms*]
by (*auto simp*: *orbit_altdef_bounded*[**where** *n*=*funpow_dist1 f x x*])

lemma *funpow_dist1_prop1*:
assumes (*f* ^^ *n*) *x* = *y* *0* < *n* **shows** (*f* ^^ *funpow_dist1 f x y*) *x* = *y*
proof −
from *assms* **have** *y* ∈ *orbit f x* **by** (*auto simp*: *orbit_altdef*)
then show *?thesis* **by** (*rule funpow_dist1_prop*)
qed

lemma *funpow_dist1_dist*:
assumes *funpow_dist1 f x y* < *funpow_dist1 f x z*
assumes {*y*,*z*} ⊆ *orbit f x*
shows *funpow_dist1 f x z* = *funpow_dist1 f x y* + *funpow_dist1 f y z* (**is** *?L* = *?R*)
proof −
define *n* **where** ‹*n* = *funpow_dist1 f x z* − *funpow_dist1 f x y* − *1*›
with *assms* **have** ∗: ‹*funpow_dist1 f x z* = *Suc* (*funpow_dist1 f x y* + *n*)›
by *simp*
have *x_z*: (*f* ^^ *funpow_dist1 f x z*) *x* = *z* **using** *assms* **by** (*blast intro*: *funpow_dist1_prop*)
have *x_y*: (*f* ^^ *funpow_dist1 f x y*) *x* = *y* **using** *assms* **by** (*blast intro*: *funpow_dist1_prop*)

have (*f* ^^ (*funpow_dist1 f x z* − *funpow_dist1 f x y*)) *y*
= (*f* ^^ (*funpow_dist1 f x z* − *funpow_dist1 f x y*)) ((*f* ^^ *funpow_dist1 f x y*) *x*)
using *x_y* **by** *simp*
also have . . . = *z*
using *assms x_z* **by** (*simp add*: ∗ *funpow_add ac_simps funpow_swap1*)
finally have *y_z_diff*: (*f* ^^ (*funpow_dist1 f x z* − *funpow_dist1 f x y*)) *y* = *z* **.**
then have (*f* ^^ *funpow_dist1 f y z*) *y* = *z*
using *assms* **by** (*intro funpow_dist1_prop1*) *auto*
then have (*f* ^^ *funpow_dist1 f y z*) ((*f* ^^ *funpow_dist1 f x y*) *x*) = *z*
using *x_y* **by** *simp*
then have (*f* ^^ (*funpow_dist1 f y z* + *funpow_dist1 f x y*)) *x* = *z*
by (*simp add*: ∗ *funpow_add funpow_swap1*)
show *?thesis*
proof (*rule antisym*)
from *y_z_diff* **have** (*f* ^^ *funpow_dist1 f y z*) *y* = *z*
using *assms* **by** (*intro funpow_dist1_prop1*) *auto*
then have (*f* ^^ *funpow_dist1 f y z*) ((*f* ^^ *funpow_dist1 f x y*) *x*) = *z*
using *x_y* **by** *simp*
then have (*f* ^^ (*funpow_dist1 f y z* + *funpow_dist1 f x y*)) *x* = *z*
by (*simp add*: ∗ *funpow_add funpow_swap1*)
then have *funpow_dist1 f x z* ≤ *funpow_dist1 f y z* + *funpow_dist1 f x y*
using *funpow_dist1_least not_less* **by** *fastforce*
then show *?L* ≤ *?R* **by** *presburger*

```
  next
    have funpow_dist1 f y z ≤ funpow_dist1 f x z − funpow_dist1 f x y
     using y_z_diff assms(1) by (metis not_less zero_less_diff funpow_dist1_least)
    then show ?R ≤ ?L by linarith
  qed
qed

lemma funpow_dist1_le_self:
  assumes (f ^^ m) x = x 0 < m y ∈ orbit f x
  shows funpow_dist1 f x y ≤ m
proof (cases x = y)
  case True with assms show ?thesis by (auto dest!: funpow_dist1_least)
next
  case False
  have (f ^^ funpow_dist1 f x y) x = (f ^^ (funpow_dist1 f x y mod m)) x
    using assms by (simp add: funpow_mod_eq)
  with False ‹y ∈ orbit f x› have funpow_dist1 f x y ≤ funpow_dist1 f x y mod m
     by auto (metis ‹(f ^^ funpow_dist1 f x y) x = (f ^^ (funpow_dist1 f x y mod
m)) x› funpow_dist1_prop funpow_dist_least funpow_dist_step leI)
  with ‹m > 0› show ?thesis
    by (auto intro: order_trans)
qed

end
```

9 Basic combinatorics in Isabelle/HOL (and the Archive of Formal Proofs)

```
theory Combinatorics
imports
  Transposition
  Stirling
  Permutations
  List_Permutation
  Multiset_Permutations
  Cycles
  Perm
  Orbits
begin

end
```