

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

December 17, 2025

Contents

1	Transposition function	1
2	Stirling numbers of first and second kind	5
2.1	Stirling numbers of the second kind	5
2.2	Stirling numbers of the first kind	6
2.2.1	Efficient code	7
3	Permutations, both general and specifically on finite sets.	8
3.1	Auxiliary	8
3.2	Basic definition and consequences	8
3.3	Group properties	11
3.4	Restricting a permutation to a subset	11
3.5	Mapping a permutation	12
3.6	The number of permutations on a finite set	14
3.7	Permutations of index set for iterated operations	14
3.8	Permutations as transposition sequences	14
3.9	Some closure properties of the set of permutations, with lengths	14
3.10	Various combinations of transpositions with 2, 1 and 0 com- mon elements	15
3.11	The identity map only has even transposition sequences . . .	16
3.12	Therefore we have a welldefined notion of parity	16
3.13	And it has the expected composition properties	17
3.14	A more abstract characterization of permutations	17
3.15	Relation to <i>permutes</i>	18
3.16	Sign of a permutation	18
3.17	An induction principle in terms of transpositions	19
3.18	More on the sign of permutations	20
3.19	Transpositions of adjacent elements	21
3.20	Transferring properties of permutations along bijections . . .	22

3.21	Permuting a list	23
3.22	More lemmas about permutations	25
3.23	Sum over a set of permutations (could generalize to iteration)	27
3.24	Constructing permutations from association lists	27
4	Permuted Lists	29
4.1	An existing notion	29
4.2	Nontrivial conclusions	30
4.3	Trivial conclusions:	30
5	Permutations of a Multiset	32
5.1	Permutations of a multiset	32
5.2	Cardinality of permutations	34
5.3	Permutations of a set	35
5.4	Code generation	36
6	Cycles	39
6.1	Definitions	39
6.2	Basic Properties	39
6.3	Conjugation of cycles	40
6.4	When Cycles Commute	40
6.5	Cycles from Permutations	40
6.5.1	Exponentiation of permutations	40
6.5.2	Extraction of cycles from permutations	41
6.6	Decomposition on Cycles	41
6.6.1	Preliminaries	41
6.6.2	Decomposition	42
7	Permutations as abstract type	42
7.1	Abstract type of permutations	43
7.2	Identity, composition and inversion	44
7.3	Orbit and order of elements	45
7.4	Swaps	49
7.5	Permutations specified by cycles	49
7.6	Syntax	50
8	Permutation orbits	50
8.1	Orbits and cyclic permutations	50
8.2	Decomposition of arbitrary permutations	54
8.3	Function-power distance between values	54
9	Basic combinatorics in Isabelle/HOL (and the Archive of Formal Proofs)	56

1 Transposition function

theory *Transposition*
 imports *Main*
begin

definition *transpose* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \rangle$
 where $\langle \text{transpose } a \ b \ c = (\text{if } c = a \text{ then } b \text{ else if } c = b \text{ then } a \text{ else } c) \rangle$

lemma *transpose_apply_first* [*simp*]:
 $\langle \text{transpose } a \ b \ a = b \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_apply_second* [*simp*]:
 $\langle \text{transpose } a \ b \ b = a \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_apply_other* [*simp*]:
 $\langle \text{transpose } a \ b \ c = c \rangle$ **if** $\langle c \neq a \rangle \ \langle c \neq b \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_same* [*simp*]:
 $\langle \text{transpose } a \ a = \text{id} \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_eq_iff*:
 $\langle \text{transpose } a \ b \ c = d \iff (c \neq a \wedge c \neq b \wedge d = c) \vee (c = a \wedge d = b) \vee (c = b \wedge d = a) \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_eq_imp_eq*:
 $\langle c = d \rangle$ **if** $\langle \text{transpose } a \ b \ c = \text{transpose } a \ b \ d \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_commute* [*ac_simps*]:
 $\langle \text{transpose } b \ a = \text{transpose } a \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_involutory* [*simp*]:
 $\langle \text{transpose } a \ b \ (\text{transpose } a \ b \ c) = c \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_comp_involutory* [*simp*]:
 $\langle \text{transpose } a \ b \circ \text{transpose } a \ b = \text{id} \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_eq_id_iff*: *Transposition.transpose* $x \ y = \text{id} \iff x = y$
 $\langle \text{proof} \rangle$

lemma *transpose_triple*:

$\langle \text{transpose } a \ b \ (\text{transpose } b \ c \ (\text{transpose } a \ b \ d)) = \text{transpose } a \ c \ d \rangle$
if $\langle a \neq c \rangle$ **and** $\langle b \neq c \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_comp_triple*:

$\langle \text{transpose } a \ b \circ \text{transpose } b \ c \circ \text{transpose } a \ b = \text{transpose } a \ c \rangle$
if $\langle a \neq c \rangle$ **and** $\langle b \neq c \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_image_eq* [simp]:

$\langle \text{transpose } a \ b \ ' A = A \rangle$ **if** $\langle a \in A \longleftrightarrow b \in A \rangle$
 $\langle \text{proof} \rangle$

lemma *inj_on_transpose* [simp]:

$\langle \text{inj_on } (\text{transpose } a \ b) \ A \rangle$
 $\langle \text{proof} \rangle$

lemma *inj_transpose*:

$\langle \text{inj } (\text{transpose } a \ b) \rangle$
 $\langle \text{proof} \rangle$

lemma *surj_transpose*:

$\langle \text{surj } (\text{transpose } a \ b) \rangle$
 $\langle \text{proof} \rangle$

lemma *bij_betw_transpose_iff* [simp]:

$\langle \text{bij_betw } (\text{transpose } a \ b) \ A \ A \rangle$ **if** $\langle a \in A \longleftrightarrow b \in A \rangle$
 $\langle \text{proof} \rangle$

lemma *bij_transpose* [simp]:

$\langle \text{bij } (\text{transpose } a \ b) \rangle$
 $\langle \text{proof} \rangle$

lemma *bijection_transpose*:

$\langle \text{bijection } (\text{transpose } a \ b) \rangle$
 $\langle \text{proof} \rangle$

lemma *inv_transpose_eq* [simp]:

$\langle \text{inv } (\text{transpose } a \ b) = \text{transpose } a \ b \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_apply_commute*:

$\langle \text{transpose } a \ b \ (f \ c) = f \ (\text{transpose } (\text{inv } f \ a) \ (\text{inv } f \ b) \ c) \rangle$
if $\langle \text{bij } f \rangle$
 $\langle \text{proof} \rangle$

lemma *transpose_comp_eq*:

$\langle \text{transpose } a \ b \circ f = f \circ \text{transpose } (\text{inv } f \ a) \ (\text{inv } f \ b) \rangle$

if $\langle \text{bij } f \rangle$
 $\langle \text{proof} \rangle$

lemma *in_transpose_image_iff*:
 $\langle x \in \text{transpose } a \ b \ \text{' } S \longleftrightarrow \text{transpose } a \ b \ x \in S \rangle$
 $\langle \text{proof} \rangle$

Legacy input alias

$\langle ML \rangle$

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

lemma *swap_def*:
 $\langle \text{Fun.swap } a \ b \ f = f \ (a := f \ b, \ b := f \ a) \rangle$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *swap_apply*:
 $\text{Fun.swap } a \ b \ f \ a = f \ b$
 $\text{Fun.swap } a \ b \ f \ b = f \ a$
 $c \neq a \implies c \neq b \implies \text{Fun.swap } a \ b \ f \ c = f \ c$
 $\langle \text{proof} \rangle$

lemma *swap_self*: $\text{Fun.swap } a \ a \ f = f$
 $\langle \text{proof} \rangle$

lemma *swap_commute*: $\text{Fun.swap } a \ b \ f = \text{Fun.swap } b \ a \ f$
 $\langle \text{proof} \rangle$

lemma *swap_nilpotent*: $\text{Fun.swap } a \ b \ (\text{Fun.swap } a \ b \ f) = f$
 $\langle \text{proof} \rangle$

lemma *swap_comp_involutory*: $\text{Fun.swap } a \ b \circ \text{Fun.swap } a \ b = \text{id}$
 $\langle \text{proof} \rangle$

lemma *swap_triple*:
assumes $a \neq c$ **and** $b \neq c$
shows $\text{Fun.swap } a \ b \ (\text{Fun.swap } b \ c \ (\text{Fun.swap } a \ b \ f)) = \text{Fun.swap } a \ c \ f$
 $\langle \text{proof} \rangle$

lemma *comp_swap*: $f \circ \text{Fun.swap } a \ b \ g = \text{Fun.swap } a \ b \ (f \circ g)$
 $\langle \text{proof} \rangle$

lemma *swap_image_eq*:
assumes $a \in A \ b \in A$
shows $\text{Fun.swap } a \ b \ f \ \text{' } A = f \ \text{' } A$
 $\langle \text{proof} \rangle$

lemma *inj_on_imp_inj_on_swap*: $\text{inj_on } f \ A \implies a \in A \implies b \in A \implies \text{inj_on } (\text{Fun.swap } a \ b \ f) \ A$
 ⟨proof⟩

lemma *inj_on_swap_iff*:
 assumes $A: a \in A \ b \in A$
 shows $\text{inj_on } (\text{Fun.swap } a \ b \ f) \ A \longleftrightarrow \text{inj_on } f \ A$
 ⟨proof⟩

lemma *surj_imp_surj_swap*: $\text{surj } f \implies \text{surj } (\text{Fun.swap } a \ b \ f)$
 ⟨proof⟩

lemma *surj_swap_iff*: $\text{surj } (\text{Fun.swap } a \ b \ f) \longleftrightarrow \text{surj } f$
 ⟨proof⟩

lemma *bij_betw_swap_iff*: $x \in A \implies y \in A \implies \text{bij_betw } (\text{Fun.swap } x \ y \ f) \ A \ B \longleftrightarrow \text{bij_betw } f \ A \ B$
 ⟨proof⟩

lemma *bij_swap_iff*: $\text{bij } (\text{Fun.swap } a \ b \ f) \longleftrightarrow \text{bij } f$
 ⟨proof⟩

lemma *swap_image*:
 ⟨ $\text{Fun.swap } i \ j \ f \ ' \ A = f \ ' \ (A - \{i, j\} \cup (\text{if } i \in A \text{ then } \{j\} \text{ else } \{\}) \cup (\text{if } j \in A \text{ then } \{i\} \text{ else } \{\}))$ ⟩
 ⟨proof⟩

lemma *inv_swap_id*: $\text{inv } (\text{Fun.swap } a \ b \ \text{id}) = \text{Fun.swap } a \ b \ \text{id}$
 ⟨proof⟩

lemma *bij_swap_comp*:
 assumes $\text{bij } p$
 shows $\text{Fun.swap } a \ b \ \text{id} \circ p = \text{Fun.swap } (\text{inv } p \ a) \ (\text{inv } p \ b) \ p$
 ⟨proof⟩

lemma *swap_id_eq*: $\text{Fun.swap } a \ b \ \text{id } x = (\text{if } x = a \text{ then } b \text{ else if } x = b \text{ then } a \text{ else } x)$
 ⟨proof⟩

lemma *swap_unfold*:
 ⟨ $\text{Fun.swap } a \ b \ p = p \circ \text{Fun.swap } a \ b \ \text{id}$ ⟩
 ⟨proof⟩

lemma *swap_id_idempotent*: $\text{Fun.swap } a \ b \ \text{id} \circ \text{Fun.swap } a \ b \ \text{id} = \text{id}$
 ⟨proof⟩

lemma *bij_swap_compose_bij*:
 ⟨ $\text{bij } (\text{Fun.swap } a \ b \ \text{id} \circ p)$ ⟩ **if** ⟨ $\text{bij } p$ ⟩

<proof>

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 \Rightarrow nat \Rightarrow 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

<proof>

lemma *Stirling_less* [simp]: $n < k \implies \text{Stirling } n \ k = 0$

<proof>

lemma *Stirling_same* [simp]: *Stirling* n n = 1

<proof>

lemma *Stirling_2_2*: *Stirling* (Suc (Suc n)) (Suc (Suc 0)) = $2^{\text{Suc } n} - 1$

<proof>

lemma *Stirling_2*: *Stirling* (Suc n) (Suc (Suc 0)) = $2^n - 1$

<proof>

2.2 Stirling numbers of the first kind

fun *stirling* :: nat \Rightarrow nat \Rightarrow 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]: $n > 0 \implies \text{stirling } n \ 0 = 0$

<proof>

lemma *stirling_less* [simp]: $n < k \implies \text{stirling } n \ k = 0$

<proof>

lemma *stirling_same* [simp]: *stirling* *n n* = 1
 ⟨proof⟩

lemma *stirling_Suc_n_1*: *stirling* (*Suc n*) (*Suc 0*) = *fact n*
 ⟨proof⟩

lemma *stirling_Suc_n_n*: *stirling* (*Suc n*) *n* = *Suc n choose 2*
 ⟨proof⟩

lemma *stirling_Suc_n_2*:
 assumes *n* ≥ *Suc 0*
 shows *stirling* (*Suc n*) 2 = ($\sum k=1..n. \text{fact } n \text{ div } k$)
 ⟨proof⟩

lemma *of_nat_stirling_Suc_n_2*:
 assumes *n* ≥ *Suc 0*
 shows (*of_nat* (*stirling* (*Suc n*) 2))::'*a*::field_char_0) = *fact n* * ($\sum k=1..n. (1 / \text{of_nat } k)$)
 ⟨proof⟩

lemma *sum_stirling*: ($\sum k \leq n. \text{stirling } n k$) = *fact n*
 ⟨proof⟩

lemma *stirling_pochhammer*:
 ($\sum k \leq n. \text{of_nat } (\text{stirling } n k) * x^k$) = (*pochhammer* *x n* :: '*a*::comm_semiring_1')
 ⟨proof⟩

A row of the Stirling number triangle

definition *stirling_row* :: *nat* ⇒ *nat list*
 where *stirling_row n* = [*stirling n k*. *k* ← [0..*Suc n*]]

lemma *nth_stirling_row*: *k* ≤ *n* ⇒ *stirling_row n* ! *k* = *stirling n k*
 ⟨proof⟩

lemma *length_stirling_row* [simp]: *length* (*stirling_row n*) = *Suc n*
 ⟨proof⟩

lemma *stirling_row_nonempty* [simp]: *stirling_row n* ≠ []
 ⟨proof⟩

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 ⇒ 'a ⇒ 'b) ⇒ 'a ⇒ 'a list ⇒ '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 ← [0..
 ⟨proof⟩

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]
 ⟨proof⟩

lemma *stirling_row_code* [code]:
stirling_row 0 = [1]
stirling_row (Suc n) = *stirling_row_aux* n 0 (*stirling_row* n)
 ⟨proof⟩

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)
 ⟨proof⟩

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*)⟩
 ⟨*proof*⟩

lemma *bij_betw_fixpoints*:

⟨*bij_betw* *f* (*fixpoints* *f*) (*fixpoints* *f*)⟩
 ⟨*proof*⟩

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*⟩

lemma *inj_imp_permutes*:

assumes *i*: *inj_on* *f S* **and** *fin*: *finite S*
and *fS*: ∧ *x*. *x* ∈ *S* ⟶ *f x* ∈ *S*
and *f*: ∧ *i*. *i* ∉ *S* ⟶ *f i* = *i*
shows *f permutes S*
 ⟨*proof*⟩

context

fixes *p* :: ⟨*'a* ⇒ *'a*⟩ **and** *S* :: ⟨*'a* *set*⟩
assumes *perm*: ⟨*p permutes S*⟩

begin

lemma *permutes_inj*:

⟨*inj* *p*⟩
 ⟨*proof*⟩

lemma *permutes_image*:

⟨*p* ‘ *S* = *S*⟩
 ⟨*proof*⟩

lemma *permutes_not_in*:

⟨*x* ∉ *S* ⟶ *p x* = *x*⟩
 ⟨*proof*⟩

lemma *permutes_image_complement*:

⟨*p* ‘ (− *S*) = − *S*⟩
 ⟨*proof*⟩

lemma *permutes_in_image*:

⟨*p x* ∈ *S* ⟷ *x* ∈ *S*⟩
 ⟨*proof*⟩

```

lemma permutes_surj:
   $\langle \text{surj } p \rangle$ 
 $\langle \text{proof} \rangle$ 

lemma permutes_inv_o:
  shows  $p \circ \text{inv } p = \text{id}$ 
  and  $\text{inv } p \circ p = \text{id}$ 
 $\langle \text{proof} \rangle$ 

lemma permutes_inverses:
  shows  $p (\text{inv } p \ x) = x$ 
  and  $\text{inv } p (p \ x) = x$ 
 $\langle \text{proof} \rangle$ 

lemma permutes_inv_eq:
   $\langle \text{inv } p \ y = x \longleftrightarrow p \ x = y \rangle$ 
 $\langle \text{proof} \rangle$ 

lemma permutes_inj_on:
   $\langle \text{inj\_on } p \ A \rangle$ 
 $\langle \text{proof} \rangle$ 

lemma permutes_bij:
   $\langle \text{bij } p \rangle$ 
 $\langle \text{proof} \rangle$ 

lemma permutes_imp_bij:
   $\langle \text{bij\_betw } p \ S \ S \rangle$ 
 $\langle \text{proof} \rangle$ 

lemma permutes_subset:
   $\langle p \text{ permutes } T \rangle$  if  $\langle S \subseteq T \rangle$ 
 $\langle \text{proof} \rangle$ 

lemma permutes_imp_permutes_insert:
   $\langle p \text{ permutes insert } x \ S \rangle$ 
 $\langle \text{proof} \rangle$ 

end

lemma permutes_id [simp]:
   $\langle \text{id permutes } S \rangle$ 
 $\langle \text{proof} \rangle$ 

lemma permutes_empty [simp]:
   $\langle p \text{ permutes } \{\} \longleftrightarrow p = \text{id} \rangle$ 
 $\langle \text{proof} \rangle$ 

```

lemma *permutes_sing* [simp]:
 $\langle p \text{ permutes } \{a\} \longleftrightarrow p = \text{id} \rangle$
 $\langle \text{proof} \rangle$

lemma *permutes_univ*: $p \text{ permutes } \text{UNIV} \longleftrightarrow (\forall y. \exists! x. p \ x = y)$
 $\langle \text{proof} \rangle$

lemma *permutes_swap_id*: $a \in S \implies b \in S \implies \text{transpose } a \ b \text{ permutes } S$
 $\langle \text{proof} \rangle$

lemma *permutes_altdef*: $p \text{ permutes } A \longleftrightarrow \text{bij_betw } p \ A \ A \wedge \{x. p \ x \neq x\} \subseteq A$
 $\langle \text{proof} \rangle$

lemma *permutes_superset*:
 $\langle p \text{ permutes } T \rangle \text{ if } \langle p \text{ permutes } S \rangle \langle \bigwedge x. x \in S - T \implies p \ x = x \rangle$
 $\langle \text{proof} \rangle$

lemma *permutes_bij_inv_into*:
fixes $A :: 'a \text{ set}$
and $B :: 'b \text{ set}$
assumes $p \text{ permutes } A$
and $\text{bij_betw } f \ A \ B$
shows $(\lambda x. \text{if } x \in B \text{ then } f \ (p \ (\text{inv_into } A \ f \ x)) \text{ else } x) \text{ permutes } B$
 $\langle \text{proof} \rangle$

lemma *permutes_image_mset*:
assumes $p \text{ permutes } A$
shows $\text{image_mset } p \ (\text{mset_set } A) = \text{mset_set } A$
 $\langle \text{proof} \rangle$

lemma *permutes_implies_image_mset_eq*:
assumes $p \text{ permutes } A \wedge x. x \in A \implies f \ x = f' \ (p \ x)$
shows $\text{image_mset } f' \ (\text{mset_set } A) = \text{image_mset } f \ (\text{mset_set } A)$
 $\langle \text{proof} \rangle$

3.3 Group properties

lemma *permutes_compose*: $p \text{ permutes } S \implies q \text{ permutes } S \implies q \circ p \text{ permutes } S$
 $\langle \text{proof} \rangle$

lemma *permutes_inv*:
assumes $p \text{ permutes } S$
shows $\text{inv } p \text{ permutes } S$
 $\langle \text{proof} \rangle$

lemma *permutes_inv_inv*:
assumes $p \text{ permutes } S$
shows $\text{inv } (\text{inv } p) = p$
 $\langle \text{proof} \rangle$

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

lemma *permutes_vimage*: f permutes $A \implies f -' A = A$
 $\langle proof \rangle$

3.4 Restricting a permutation to a subset

definition *restrict_id* :: $('a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'a$
where *restrict_id* $f\ A = (\lambda x. \text{if } x \in A \text{ then } f\ x \text{ else } x)$

lemma *restrict_id_cong* [*cong*]:
assumes $\bigwedge x. x \in A \implies f\ x = g\ x$ $A = B$
shows *restrict_id* $f\ A = \text{restrict_id } g\ B$
 $\langle proof \rangle$

lemma *restrict_id_cong'*:
assumes $x \in A \implies f\ x = g\ x$ $A = B$
shows *restrict_id* $f\ A\ x = \text{restrict_id } g\ B\ x$
 $\langle proof \rangle$

lemma *restrict_id_simps* [*simp*]:
 $x \in A \implies \text{restrict_id } f\ A\ x = f\ x$
 $x \notin A \implies \text{restrict_id } f\ A\ x = x$
 $\langle proof \rangle$

lemma *bij_betw_restrict_id*:
assumes *bij_betw* $f\ A\ A \subseteq B$
shows *bij_betw* $(\text{restrict_id } f\ A)\ B\ B$
 $\langle proof \rangle$

lemma *permutes_restrict_id*:
assumes *bij_betw* $f\ A\ A$
shows *restrict_id* $f\ A$ permutes A
 $\langle proof \rangle$

3.5 Mapping a permutation

definition *map_permutation* :: $'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'b$ **where**
 $\text{map_permutation } A\ f\ p = \text{restrict_id } (f \circ p \circ \text{inv_into } A\ f)\ (f -' A)$

lemma *map_permutation_cong_strong*:
assumes $A = B$ $\bigwedge x. x \in A \implies f\ x = g\ x$ $\bigwedge x. x \in A \implies p\ x = q\ x$
assumes $p -' A \subseteq A$ *inj_on* $f\ A$
shows $\text{map_permutation } A\ f\ p = \text{map_permutation } B\ g\ q$

$\langle proof \rangle$

lemma *map_permutation_cong*:

assumes *inj_on* *f* *A* *p* *permutes* *A*

assumes $A = B \wedge x. x \in A \implies f\ x = g\ x \wedge x. x \in A \implies p\ x = q\ x$

shows $\text{map_permutation}\ A\ f\ p = \text{map_permutation}\ B\ g\ q$

$\langle proof \rangle$

lemma *inv_into_id* [*simp*]: $x \in A \implies \text{inv_into}\ A\ id\ x = x$

$\langle proof \rangle$

lemma *inv_into_ident* [*simp*]: $x \in A \implies \text{inv_into}\ A\ (\lambda x. x)\ x = x$

$\langle proof \rangle$

lemma *map_permutation_id* [*simp*]: $p\ \text{permutes}\ A \implies \text{map_permutation}\ A\ id\ p$

$=\ p$

$\langle proof \rangle$

lemma *map_permutation_ident* [*simp*]: $p\ \text{permutes}\ A \implies \text{map_permutation}\ A$

$(\lambda x. x)\ p = p$

$\langle proof \rangle$

lemma *map_permutation_id'*: $\text{inj_on}\ f\ A \implies \text{map_permutation}\ A\ f\ id = id$

$\langle proof \rangle$

lemma *map_permutation_ident'*: $\text{inj_on}\ f\ A \implies \text{map_permutation}\ A\ f\ (\lambda x. x)$

$=\ (\lambda x. x)$

$\langle proof \rangle$

lemma *map_permutation_permutes*:

assumes *bij_betw* *f* *A* *B* *p* *permutes* *A*

shows $\text{map_permutation}\ A\ f\ p\ \text{permutes}\ B$

$\langle proof \rangle$

lemma *map_permutation_compose*:

fixes $f :: 'a \Rightarrow 'b$ **and** $g :: 'b \Rightarrow 'c$

assumes *bij_betw* *f* *A* *B* *inj_on* *g* *B*

shows $\text{map_permutation}\ B\ g\ (\text{map_permutation}\ A\ f\ p) = \text{map_permutation}\ A\ (g \circ f)\ p$

$\langle proof \rangle$

lemma *map_permutation_compose_inv*:

assumes *bij_betw* *f* *A* *B* *p* *permutes* *A* $\wedge x. x \in A \implies g\ (f\ x) = x$

shows $\text{map_permutation}\ B\ g\ (\text{map_permutation}\ A\ f\ p) = p$

$\langle proof \rangle$

lemma *map_permutation_apply*:

assumes *inj_on* *f* *A* $x \in A$

shows $\text{map_permutation}\ A\ f\ h\ (f\ x) = f\ (h\ x)$

$\langle \text{proof} \rangle$

lemma *map_permutation_compose'*:

fixes $f :: 'a \Rightarrow 'b$

assumes *inj_on* f A q *permutes* A

shows $\text{map_permutation } A f (p \circ q) = \text{map_permutation } A f p \circ \text{map_permutation } A f q$

$\langle \text{proof} \rangle$

lemma *map_permutation_transpose*:

assumes *inj_on* f $a \in A$ $b \in A$

shows $\text{map_permutation } A f (\text{Transposition.transpose } a b) = \text{Transposition.transpose } (f a) (f b)$

$\langle \text{proof} \rangle$

lemma *map_permutation_permutes_iff*:

assumes *bij_betw* f A B $p \text{ ' } A \subseteq A \bigwedge x. x \notin A \implies p x = x$

shows $\text{map_permutation } A f p \text{ permutes } B \longleftrightarrow p \text{ permutes } A$

$\langle \text{proof} \rangle$

lemma *bij_betw_permutations*:

assumes *bij_betw* f A B

shows *bij_betw* $(\lambda \pi x. \text{if } x \in B \text{ then } f (\pi (\text{inv_into } A f x)) \text{ else } x)$
 $\{\pi. \pi \text{ permutes } A\} \{\pi. \pi \text{ permutes } B\} \text{ (is_bij_betw } ?f \text{ ___)}$

$\langle \text{proof} \rangle$

lemma *bij_betw_derangements*:

assumes *bij_betw* f A B

shows *bij_betw* $(\lambda \pi x. \text{if } x \in B \text{ then } f (\pi (\text{inv_into } A f x)) \text{ else } x)$
 $\{\pi. \pi \text{ permutes } A \wedge (\forall x \in A. \pi x \neq x)\} \{\pi. \pi \text{ permutes } B \wedge (\forall x \in B. \pi x \neq x)\}$
 $\text{ (is_bij_betw } ?f \text{ ___)}$

$\langle \text{proof} \rangle$

3.6 The number of permutations on a finite set

lemma *permutes_insert_lemma*:

assumes p *permutes* $(\text{insert } a S)$

shows $\text{transpose } a (p a) \circ p$ *permutes* S

$\langle \text{proof} \rangle$

lemma *permutes_insert*: $\{p. p \text{ permutes } (\text{insert } a S)\} =$

$(\lambda (b, p). \text{transpose } a b \circ p) \text{ ' } \{(b, p). b \in \text{insert } a S \wedge p \in \{p. p \text{ permutes } S\}\}$

$\langle \text{proof} \rangle$

lemma *card_permutations*:

assumes $\text{card } S = n$

and *finite* S

shows $\text{card } \{p. p \text{ permutes } S\} = \text{fact } n$

$\langle proof \rangle$

lemma *finite_permutations*:
assumes *finite S*
shows *finite {p. p permutes S}*
 $\langle proof \rangle$

lemma *permutes_doubleton_iff*: $f \text{ permutes } \{a, b\} \longleftrightarrow f = id \vee f = \text{Transposition.transpose } a \ b$
 $\langle proof \rangle$

3.7 Permutations of index set for iterated operations

lemma (*in comm_monoid_set*) *permute*:
assumes *p permutes S*
shows $F \ g \ S = F \ (g \circ p) \ S$
 $\langle proof \rangle$

3.8 Permutations as transposition sequences

inductive *swapidseq* :: $nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow bool$
where
 $id[simp]: swapidseq \ 0 \ id$
 $| \ comp_Suc: swapidseq \ n \ p \Longrightarrow a \neq b \Longrightarrow swapidseq \ (Suc \ n) \ (transpose \ a \ b \circ p)$

declare *id[unfolded id_def, simp]*

definition *permutation* $p \longleftrightarrow (\exists \ n. \ swapidseq \ n \ p)$

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

lemma *permutation_id[simp]*: *permutation id*
 $\langle proof \rangle$

declare *permutation_id[unfolded id_def, simp]*

lemma *swapidseq_swap*: $swapidseq \ (if \ a = b \ then \ 0 \ else \ 1) \ (transpose \ a \ b)$
 $\langle proof \rangle$

lemma *permutation_swap_id*: *permutation (transpose a b)*
 $\langle proof \rangle$

lemma *swapidseq_comp_add*: $swapidseq \ n \ p \Longrightarrow swapidseq \ m \ q \Longrightarrow swapidseq \ (n + m) \ (p \circ q)$
 $\langle proof \rangle$

lemma *permutation_compose*: $permutation \ p \Longrightarrow permutation \ q \Longrightarrow permutation \ (p \circ q)$

$\langle \text{proof} \rangle$

lemma *swapidseq_endswap*: $\text{swapidseq } n \ p \implies a \neq b \implies \text{swapidseq } (\text{Suc } n) \ (p \circ \text{transpose } a \ b)$
 $\langle \text{proof} \rangle$

lemma *swapidseq_inverse_exists*: $\text{swapidseq } n \ p \implies \exists q. \text{swapidseq } n \ q \wedge p \circ q = \text{id} \wedge q \circ p = \text{id}$
 $\langle \text{proof} \rangle$

lemma *swapidseq_inverse*:
assumes *swapidseq* $n \ p$
shows *swapidseq* $n \ (\text{inv } p)$
 $\langle \text{proof} \rangle$

lemma *permutation_inverse*: $\text{permutation } p \implies \text{permutation } (\text{inv } p)$
 $\langle \text{proof} \rangle$

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

lemma *swap_id_common*: $a \neq c \implies b \neq c \implies \text{transpose } a \ b \circ \text{transpose } a \ c = \text{transpose } b \ c \circ \text{transpose } a \ b$
 $\langle \text{proof} \rangle$

lemma *swap_id_common'*: $a \neq b \implies a \neq c \implies \text{transpose } a \ c \circ \text{transpose } b \ c = \text{transpose } b \ c \circ \text{transpose } a \ b$
 $\langle \text{proof} \rangle$

lemma *swap_id_independent*: $a \neq c \implies a \neq d \implies b \neq c \implies b \neq d \implies \text{transpose } a \ b \circ \text{transpose } c \ d = \text{transpose } c \ d \circ \text{transpose } a \ b$
 $\langle \text{proof} \rangle$

3.11 The identity map only has even transposition sequences

lemma *symmetry_lemma*:
assumes $\bigwedge a \ b \ c \ d. P \ a \ b \ c \ d \implies P \ a \ b \ d \ c$
and $\bigwedge a \ b \ c \ d. a \neq b \implies c \neq d \implies$
 $a = c \wedge b = d \vee a = c \wedge b \neq d \vee a \neq c \wedge b = d \vee a \neq c \wedge a \neq d \wedge b \neq c$
 $\wedge b \neq d \implies$
 $P \ a \ b \ c \ d$
shows $\bigwedge a \ b \ c \ d. a \neq b \longrightarrow c \neq d \longrightarrow P \ a \ b \ c \ d$
 $\langle \text{proof} \rangle$

lemma *swap_general*:
assumes $a \neq b \ c \neq d$
shows $\text{transpose } a \ b \circ \text{transpose } c \ d = \text{id} \vee$
 $(\exists x \ y \ z. x \neq a \wedge y \neq a \wedge z \neq a \wedge x \neq y \wedge$
 $\text{transpose } a \ b \circ \text{transpose } c \ d = \text{transpose } x \ y \circ \text{transpose } a \ z)$

$\langle \text{proof} \rangle$

lemma *swapidseq_id_iff*[simp]: $\text{swapidseq } 0 \ p \longleftrightarrow p = \text{id}$
 $\langle \text{proof} \rangle$

lemma *swapidseq_cases*: $\text{swapidseq } n \ p \longleftrightarrow$
 $n = 0 \wedge p = \text{id} \vee (\exists a \ b \ q \ m. n = \text{Suc } m \wedge p = \text{transpose } a \ b \circ q \wedge \text{swapidseq}$
 $m \ q \wedge a \neq b)$
 $\langle \text{proof} \rangle$

lemma *fixing_swapidseq_decrease*:
assumes *swapidseq* $n \ p$
and $a \neq b$
and $(\text{transpose } a \ b \circ p) \ a = a$
shows $n \neq 0 \wedge \text{swapidseq } (n - 1) \ (\text{transpose } a \ b \circ p)$
 $\langle \text{proof} \rangle$

lemma *swapidseq_identity_even*:
assumes *swapidseq* $n \ (\text{id} :: 'a \Rightarrow 'a)$
shows *even* n
 $\langle \text{proof} \rangle$

3.12 Therefore we have a welldefined notion of parity

definition *evenperm* $p = \text{even } (\text{SOME } n. \text{swapidseq } n \ p)$

lemma *swapidseq_even_even*:
assumes $m: \text{swapidseq } m \ p$
and $n: \text{swapidseq } n \ p$
shows $\text{even } m \longleftrightarrow \text{even } n$
 $\langle \text{proof} \rangle$

lemma *evenperm_unique*:
assumes *swapidseq* $n \ p$ and *even* $n = b$
shows *evenperm* $p = b$
 $\langle \text{proof} \rangle$

3.13 And it has the expected composition properties

lemma *evenperm_id*[simp]: *evenperm* $\text{id} = \text{True}$
 $\langle \text{proof} \rangle$

lemma *evenperm_identity* [simp]:
 $\langle \text{evenperm } (\lambda x. x) \rangle$
 $\langle \text{proof} \rangle$

lemma *evenperm_swap*: *evenperm* $(\text{transpose } a \ b) = (a = b)$
 $\langle \text{proof} \rangle$

lemma *evenperm_comp*:

assumes *permutation p permutation q*
shows $\text{evenperm } (p \circ q) \longleftrightarrow \text{evenperm } p = \text{evenperm } q$
 <proof>

lemma *evenperm_inv*:
assumes *permutation p*
shows $\text{evenperm } (\text{inv } p) = \text{evenperm } p$
 <proof>

3.14 A more abstract characterization of permutations

lemma *permutation_bijective*:
assumes *permutation p*
shows *bij p*
 <proof>

lemma *permutation_finite_support*:
assumes *permutation p*
shows $\text{finite } \{x. p\ x \neq x\}$
 <proof>

lemma *permutation_lemma*:
assumes *finite S*
and *bij p*
and $\forall x. x \notin S \longrightarrow p\ x = x$
shows *permutation p*
 <proof>

lemma *permutation*: $\text{permutation } p \longleftrightarrow \text{bij } p \wedge \text{finite } \{x. p\ x \neq x\}$
 <proof>

lemma *permutation_inverse_works*:
assumes *permutation p*
shows $\text{inv } p \circ p = \text{id}$
and $p \circ \text{inv } p = \text{id}$
 <proof>

lemma *permutation_inverse_compose*:
assumes *p: permutation p*
and *q: permutation q*
shows $\text{inv } (p \circ q) = \text{inv } q \circ \text{inv } p$
 <proof>

3.15 Relation to permutes

lemma *permutes_imp_permutation*:
 <permutation p> **if** <finite S> <p permutes S>
 <proof>

lemma *permutation_permutesE*:

assumes $\langle \text{permutation } p \rangle$
obtains S **where** $\langle \text{finite } S \rangle \langle p \text{ permutes } S \rangle$
 $\langle \text{proof} \rangle$

lemma *permutation_permutes*: $\text{permutation } p \longleftrightarrow (\exists S. \text{finite } S \wedge p \text{ permutes } S)$
 $\langle \text{proof} \rangle$

3.16 Sign of a permutation

definition *sign* :: $\langle 'a \Rightarrow 'a \Rightarrow \text{int} \rangle$ — TODO: prefer less generic name
where $\langle \text{sign } p = (\text{if evenperm } p \text{ then } 1 \text{ else } -1) \rangle$

lemma *sign_cases* [*case_names even odd*]:
obtains $\langle \text{sign } p = 1 \rangle \mid \langle \text{sign } p = -1 \rangle$
 $\langle \text{proof} \rangle$

lemma *sign_nz* [*simp*]: $\text{sign } p \neq 0$
 $\langle \text{proof} \rangle$

lemma *sign_id* [*simp*]: $\text{sign } \text{id} = 1$
 $\langle \text{proof} \rangle$

lemma *sign_identity* [*simp*]:
 $\langle \text{sign } (\lambda x. x) = 1 \rangle$
 $\langle \text{proof} \rangle$

lemma *sign_inverse*: $\text{permutation } p \implies \text{sign } (\text{inv } p) = \text{sign } p$
 $\langle \text{proof} \rangle$

lemma *sign_compose*: $\text{permutation } p \implies \text{permutation } q \implies \text{sign } (p \circ q) = \text{sign } p * \text{sign } q$
 $\langle \text{proof} \rangle$

lemma *sign_swap_id*: $\text{sign } (\text{transpose } a \ b) = (\text{if } a = b \text{ then } 1 \text{ else } -1)$
 $\langle \text{proof} \rangle$

lemma *sign_idempotent* [*simp*]: $\text{sign } p * \text{sign } p = 1$
 $\langle \text{proof} \rangle$

lemma *sign_left_idempotent* [*simp*]:
 $\langle \text{sign } p * (\text{sign } p * \text{sign } q) = \text{sign } q \rangle$
 $\langle \text{proof} \rangle$

lemma *abs_sign* [*simp*]: $|\text{sign } p| = 1$
 $\langle \text{proof} \rangle$

3.17 An induction principle in terms of transpositions

definition *apply_transps* :: $('a \times 'a) \text{ list} \Rightarrow 'a \Rightarrow 'a$ **where**
 $\text{apply_transps } xs = \text{foldr } (\circ) \ (\text{map } (\lambda(a,b). \text{Transposition.transpose } a \ b) \ xs) \ \text{id}$

lemma *apply_transps_Nil* [simp]: *apply_transps [] = id*
 ⟨proof⟩

lemma *apply_transps_Cons* [simp]:
apply_transps (x # xs) = Transposition.transpose (fst x) (snd x) ∘ apply_transps xs
 ⟨proof⟩

lemma *apply_transps_append* [simp]:
apply_transps (xs @ ys) = apply_transps xs ∘ apply_transps ys
 ⟨proof⟩

lemma *permutation_apply_transps* [simp, intro]: *permutation (apply_transps xs)*
 ⟨proof⟩

lemma *permutes_apply_transps*:
assumes $\forall (a,b) \in \text{set } xs. a \in A \wedge b \in A$
shows *apply_transps xs permutes A*
 ⟨proof⟩

lemma *permutes_induct* [consumes 2, case_names id swap]:
assumes *p permutes S finite S*
assumes *P id*
assumes $\bigwedge a b p. a \in S \implies b \in S \implies a \neq b \implies P p \implies p \text{ permutes } S$
 $\implies P (\text{Transposition.transpose } a b \circ p)$
shows *P p*
 ⟨proof⟩

lemma *permutes_rev_induct* [consumes 2, case_names id swap]:
assumes *finite S p permutes S*
assumes *P id*
assumes $\bigwedge a b p. a \in S \implies b \in S \implies a \neq b \implies P p \implies p \text{ permutes } S$
 $\implies P (p \circ \text{Transposition.transpose } a b)$
shows *P p*
 ⟨proof⟩

lemma *map_permutation_apply_transps*:
assumes *f: inj_on f A and set ts ⊆ A × A*
shows *map_permutation A f (apply_transps ts) = apply_transps (map (map_prod f f) ts)*
 ⟨proof⟩

lemma *permutes_from_transpositions*:
assumes *p permutes A finite A*
shows $\exists xs. (\forall (a,b) \in \text{set } xs. a \neq b \wedge a \in A \wedge b \in A) \wedge \text{apply_transps } xs = p$
 ⟨proof⟩

3.18 More on the sign of permutations

lemma *evenperm_apply_transps_iff*:
assumes $\forall (a,b) \in \text{set } xs. a \neq b$
shows $\text{evenperm } (\text{apply_transps } xs) \longleftrightarrow \text{even } (\text{length } xs)$
 $\langle \text{proof} \rangle$

lemma *evenperm_map_permutation*:
assumes $f: \text{inj_on } f \ A$ **and** $p \text{ permutes } A$ *finite A*
shows $\text{evenperm } (\text{map_permutation } A \ f \ p) \longleftrightarrow \text{evenperm } p$
 $\langle \text{proof} \rangle$

lemma *sign_map_permutation*:
assumes $\text{inj_on } f \ A$ $p \text{ permutes } A$ *finite A*
shows $\text{sign } (\text{map_permutation } A \ f \ p) = \text{sign } p$
 $\langle \text{proof} \rangle$

Sometimes it can be useful to consider the sign of a function that is not a permutation in the Isabelle/HOL sense, but its restriction to some finite subset is.

definition *sign_on* :: $'a \text{ set} \Rightarrow ('a \Rightarrow 'a) \Rightarrow \text{int}$
where $\text{sign_on } A \ f = \text{sign } (\text{restrict_id } f \ A)$

lemma *sign_on_cong* [*cong*]:
assumes $A = B \ \bigwedge x. x \in A \implies f \ x = g \ x$
shows $\text{sign_on } A \ f = \text{sign_on } B \ g$
 $\langle \text{proof} \rangle$

lemma *sign_on_permutes*:
assumes $f \text{ permutes } A$ $A \subseteq B$
shows $\text{sign_on } B \ f = \text{sign } f$
 $\langle \text{proof} \rangle$

lemma *sign_on_id* [*simp*]: $\text{sign_on } A \ \text{id} = 1$
 $\langle \text{proof} \rangle$

lemma *sign_on_ident* [*simp*]: $\text{sign_on } A \ (\lambda x. x) = 1$
 $\langle \text{proof} \rangle$

lemma *sign_on_transpose*:
assumes $a \in A \ b \in A \ a \neq b$
shows $\text{sign_on } A \ (\text{Transposition.transpose } a \ b) = -1$
 $\langle \text{proof} \rangle$

lemma *sign_on_compose*:
assumes $\text{bij_betw } f \ A \ A$ $\text{bij_betw } g \ A \ A$ *finite A*
shows $\text{sign_on } A \ (f \circ g) = \text{sign_on } A \ f * \text{sign_on } A \ g$
 $\langle \text{proof} \rangle$

3.19 Transpositions of adjacent elements

We have shown above that every permutation can be written as a product of transpositions. We will now furthermore show that any transposition of successive natural numbers $\{m, \dots, n\}$ can be written as a product of transpositions of *adjacent* elements, i.e. transpositions of the form $i \leftrightarrow i+1$.

function *adj_transp_seq* :: *nat* \Rightarrow *nat* \Rightarrow *nat list* **where**

```

  adj_transp_seq a b =
    (if a  $\geq$  b then []
     else if b = a + 1 then [a]
     else a # adj_transp_seq (a+1) b @ [a])
  <proof>

```

termination <proof>

lemmas [*simp del*] = *adj_transp_seq.simps*

lemma *length_adj_transp_seq*:

```

  a < b  $\implies$  length (adj_transp_seq a b) = 2 * (b - a) - 1
  <proof>

```

definition *apply_adj_transps* :: *nat list* \Rightarrow *nat* \Rightarrow *nat*

where *apply_adj_transps* *xs* = foldl (\circ) id (map (λx . *Transposition.transpose* *x* (x+1)) *xs*)

lemma *apply_adj_transps_aux*:

```

  f  $\circ$  foldl ( $\circ$ ) g (map ( $\lambda x$ . Transposition.transpose x (Suc x)) xs) =
    foldl ( $\circ$ ) (f  $\circ$  g) (map ( $\lambda x$ . Transposition.transpose x (Suc x)) xs)
  <proof>

```

lemma *apply_adj_transps_Nil* [*simp*]: *apply_adj_transps* [] = id

and *apply_adj_transps_Cons* [*simp*]:

apply_adj_transps (*x* # *xs*) = *Transposition.transpose* *x* (x+1) \circ *apply_adj_transps* *xs*

and *apply_adj_transps_snoc* [*simp*]:

apply_adj_transps (*xs* @ [*x*]) = *apply_adj_transps* *xs* \circ *Transposition.transpose* *x* (x+1)
<proof>

lemma *adj_transp_seq_correct*:

assumes *a* < *b*

shows *apply_adj_transps* (*adj_transp_seq* *a* *b*) = *Transposition.transpose* *a* *b*
<proof>

lemma *permutation_apply_adj_transps*: *permutation* (*apply_adj_transps* *xs*)

<proof>

lemma *permutes_apply_adj_transps*:

assumes $\forall x \in \text{set } xs. x \in A \wedge \text{Suc } x \in A$

shows *apply_adj_transps xs permutes A*
 ⟨proof⟩

lemma *set_adj_transp_seq*:
 $a < b \implies \text{set } (\text{adj_transp_seq } a \ b) = \{a..<b\}$
 ⟨proof⟩

3.20 Transferring properties of permutations along bijections

locale *permutes_bij* =
fixes $p :: 'a \Rightarrow 'a$ **and** $A :: 'a \text{ set}$ **and** $B :: 'b \text{ set}$
fixes $f :: 'a \Rightarrow 'b$ **and** $f' :: 'b \Rightarrow 'a$
fixes $p' :: 'b \Rightarrow 'b$
defines $p' \equiv (\lambda x. \text{if } x \in B \text{ then } f' (p (f' x)) \text{ else } x)$
assumes *permutes_p*: $p \text{ permutes } A$
assumes *bij_f*: *bij_betw f A B*
assumes *f'_f*: $x \in A \implies f' (f x) = x$
begin

lemma *bij_f'*: *bij_betw f' B A*
 ⟨proof⟩

lemma *f_f'*: $x \in B \implies f (f' x) = x$
 ⟨proof⟩

lemma *f_in_B*: $x \in A \implies f x \in B$
 ⟨proof⟩

lemma *f'_in_A*: $x \in B \implies f' x \in A$
 ⟨proof⟩

lemma *permutes_p'*: $p' \text{ permutes } B$
 ⟨proof⟩

lemma *f_eq_iff [simp]*: $f x = f y \longleftrightarrow x = y$ **if** $x \in A$ $y \in A$ **for** $x \ y$
 ⟨proof⟩

lemma *apply_transps_map_f_aux*:
assumes $\forall (a,b) \in \text{set } xs. a \in A \wedge b \in A \wedge y \in B$
shows $\text{apply_transps } (\text{map } (\text{map_prod } f \ f) \ xs) \ y = f (\text{apply_transps } xs \ (f' y))$
 ⟨proof⟩

lemma *apply_transps_map_f*:
assumes $\forall (a,b) \in \text{set } xs. a \in A \wedge b \in A$
shows $\text{apply_transps } (\text{map } (\text{map_prod } f \ f) \ xs) =$
 $(\lambda y. \text{if } y \in B \text{ then } f (\text{apply_transps } xs \ (f' y)) \text{ else } y)$
 ⟨proof⟩

end


```

locale permutes_bij_finite = permutes_bij +
  assumes finite_A: finite A
begin

lemma evenperm_p'_iff: evenperm p'  $\longleftrightarrow$  evenperm p
  <proof>

lemma sign_p': sign p' = sign p
  <proof>

end

```

3.21 Permuting a list

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

definition *permute_list* :: (nat \Rightarrow nat) \Rightarrow 'a list \Rightarrow 'a list
where *permute_list* f xs = map ($\lambda i.$ 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)
 <proof>

lemma *permute_list_nth*:
assumes f permutes {..*length* xs} *i* < *length* xs
shows *permute_list* f xs ! *i* = xs ! f *i*
 <proof>

lemma *permute_list_Nil* [simp]: *permute_list* f [] = []
 <proof>

lemma *length_permute_list* [simp]: *length* (*permute_list* f xs) = *length* xs
 <proof>

lemma *permute_list_compose*:
assumes g permutes {..*length* xs}
shows *permute_list* (f \circ g) xs = *permute_list* g (*permute_list* f xs)
 <proof>

lemma *permute_list_ident* [simp]: *permute_list* ($\lambda x.$ x) xs = xs
 <proof>

lemma *permute_list_id* [simp]: *permute_list* id xs = xs
 <proof>

lemma *mset_permute_list* [simp]:
fixes xs :: 'a list

assumes f permutes $\{..
shows $mset\ (permute_list\ f\ xs) = mset\ xs$
 $\langle proof \rangle$$

lemma $set_permute_list$ [simp]:
assumes f permutes $\{..
shows $set\ (permute_list\ f\ xs) = set\ xs$
 $\langle proof \rangle$$

lemma $distinct_permute_list$ [simp]:
assumes f permutes $\{..
shows $distinct\ (permute_list\ f\ xs) = distinct\ xs$
 $\langle proof \rangle$$

lemma $permute_list_zip$:
assumes f permutes A $A = \{..
assumes [simp]: $length\ xs = length\ ys$
shows $permute_list\ f\ (zip\ xs\ ys) = zip\ (permute_list\ f\ xs)\ (permute_list\ f\ ys)$
 $\langle proof \rangle$$

lemma $map_of_permute$:
assumes σ permutes fst ‘ $set\ xs$
shows $map_of\ xs \circ \sigma = map_of\ (map\ (\lambda(x,y). (inv\ \sigma\ x,\ y))\ xs)$
 $(is\ _ = map_of\ (map\ ?f\ _))$
 $\langle proof \rangle$

lemma $list_all2_permute_list_iff$:
 $\langle list_all2\ P\ (permute_list\ p\ xs)\ (permute_list\ p\ ys) \longleftrightarrow list_all2\ P\ xs\ ys \rangle$
if $\langle p$ permutes $\{..
 $\langle proof \rangle$$

3.22 More lemmas about permutations

lemma $permutes_in_funpow_image$:
assumes f permutes S $x \in S$
shows $(f \frown n)\ x \in S$
 $\langle proof \rangle$

lemma $permutation_self$:
assumes $\langle permutation\ p \rangle$
obtains n **where** $\langle n > 0 \rangle \langle (p \frown n)\ x = x \rangle$
 $\langle proof \rangle$

The following few lemmas were contributed by Lukas Bulwahn.

lemma $count_image_mset_eq_card_vimage$:
assumes $finite\ A$
shows $count\ (image_mset\ f\ (mset_set\ A))\ b = card\ \{a \in A. f\ a = b\}$
 $\langle proof \rangle$
lemma $image_mset_eq_implies_permutes$:

```

fixes  $f :: 'a \Rightarrow 'b$ 
assumes  $\text{finite } A$ 
  and  $\text{mset\_eq: image\_mset } f \text{ (mset\_set } A) = \text{image\_mset } f' \text{ (mset\_set } A)$ 
obtains  $p$  where  $p$  permutes  $A$  and  $\forall x \in A. f\ x = f' (p\ x)$ 
<proof>

lemma mset_eq_permutation:
  fixes  $xs\ ys :: 'a\ \text{list}$ 
  assumes  $\text{mset\_eq: mset } xs = \text{mset } ys$ 
  obtains  $p$  where  $p$  permutes  $\{..<\text{length } ys\}$   $\text{permute\_list } p\ ys = xs$ 
<proof>

lemma permutes_natset_le:
  fixes  $S :: 'a::\text{wellorder set}$ 
  assumes  $p$  permutes  $S$ 
  and  $\forall i \in S. p\ i \leq i$ 
  shows  $p = \text{id}$ 
<proof>

lemma permutes_natset_ge:
  fixes  $S :: 'a::\text{wellorder set}$ 
  assumes  $p$ :  $p$  permutes  $S$ 
  and  $\text{le: } \forall i \in S. p\ i \geq i$ 
  shows  $p = \text{id}$ 
<proof>

lemma image_inverse_permutations:  $\{\text{inv } p \mid p. p \text{ permutes } S\} = \{p. p \text{ permutes } S\}$ 
<proof>

lemma image_compose_permutations_left:
  assumes  $q$  permutes  $S$ 
  shows  $\{q \circ p \mid p. p \text{ permutes } S\} = \{p. p \text{ permutes } S\}$ 
<proof>

lemma image_compose_permutations_right:
  assumes  $q$  permutes  $S$ 
  shows  $\{p \circ q \mid p. p \text{ permutes } S\} = \{p. p \text{ permutes } S\}$ 
<proof>

lemma permutes_in_seg:  $p \text{ permutes } \{1..n\} \Longrightarrow i \in \{1..n\} \Longrightarrow 1 \leq p\ i \wedge p\ i \leq n$ 
<proof>

lemma sum_permutations_inverse:  $\text{sum } f \text{ } \{p. p \text{ permutes } S\} = \text{sum } (\lambda p. f(\text{inv } p)) \text{ } \{p. p \text{ permutes } S\}$ 
  (is ?lhs = ?rhs)
<proof>

lemma setum_permutations_compose_left:

```

```

assumes q: q permutes S
shows sum f {p. p permutes S} = sum (λp. f(q ∘ p)) {p. p permutes S}
(is ?lhs = ?rhs)
⟨proof⟩

```

```

lemma sum_permutations_compose_right:
assumes q: q permutes S
shows sum f {p. p permutes S} = sum (λp. f(p ∘ q)) {p. p permutes S}
(is ?lhs = ?rhs)
⟨proof⟩

```

```

lemma inv_inj_on_permutes:
  ⟨inj_on inv {p. p permutes S}⟩
⟨proof⟩

```

```

lemma permutes_pair_eq:
  ⟨{(p s, s) | s. s ∈ S} = {(s, inv p s) | s. s ∈ S}⟩ (is ⟨?L = ?R⟩) if ⟨p permutes S⟩
⟨proof⟩

```

```

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⟩

```

```

lemma permutes_nat_inv_less:
  ⟨inv p i < n⟩
⟨proof⟩

```

end

```

context comm_monoid_set
begin

```

```

lemma permutes_inv:
  ⟨F (λs. g (p s) s) S = F (λs. g s (inv p s)) S⟩ (is ⟨?l = ?r⟩)
if ⟨p permutes S⟩
⟨proof⟩

```

end

3.23 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 $\text{sum } f \{p. p \text{ permutes } (\text{insert } a \ S)\} =$
 $\text{sum } (\lambda b. \text{sum } (\lambda q. f (\text{transpose } a \ b \circ q)) \{p. p \text{ permutes } S\}) (\text{insert } a \ S)$
 $\langle \text{proof} \rangle$

3.24 Constructing permutations from association lists

definition $\text{list_permutes} :: ('a \times 'a) \text{ list} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$

where $\text{list_permutes } xs \ A \longleftrightarrow$
 $\text{set } (\text{map } \text{fst } xs) \subseteq A \wedge$
 $\text{set } (\text{map } \text{snd } xs) = \text{set } (\text{map } \text{fst } xs) \wedge$
 $\text{distinct } (\text{map } \text{fst } xs) \wedge$
 $\text{distinct } (\text{map } \text{snd } xs)$

lemma $\text{list_permutesI} \ [simp]:$

assumes $\text{set } (\text{map } \text{fst } xs) \subseteq A \ \text{set } (\text{map } \text{snd } xs) = \text{set } (\text{map } \text{fst } xs) \ \text{distinct } (\text{map } \text{fst } xs)$
shows $\text{list_permutes } xs \ A$
 $\langle \text{proof} \rangle$

definition $\text{permutation_of_list} :: ('a \times 'a) \text{ list} \Rightarrow 'a \Rightarrow 'a$

where $\text{permutation_of_list } xs \ x = (\text{case } \text{map_of } xs \ x \text{ of } \text{None} \Rightarrow x \mid \text{Some } y \Rightarrow y)$

lemma $\text{permutation_of_list_Cons}:$

$\text{permutation_of_list } ((x, y) \# xs) \ x' = (\text{if } x = x' \text{ then } y \text{ else } \text{permutation_of_list } xs \ x')$
 $\langle \text{proof} \rangle$

fun $\text{inverse_permutation_of_list} :: ('a \times 'a) \text{ list} \Rightarrow 'a \Rightarrow 'a$

where
 $\text{inverse_permutation_of_list } [] \ x = x$
 $\mid \text{inverse_permutation_of_list } ((y, x') \# xs) \ x =$
 $(\text{if } x = x' \text{ then } y \text{ else } \text{inverse_permutation_of_list } xs \ x)$

declare $\text{inverse_permutation_of_list.simps} \ [simp \ \text{del}]$

lemma $\text{inj_on_map_of}:$

assumes $\text{distinct } (\text{map } \text{snd } xs)$
shows $\text{inj_on } (\text{map_of } xs) \ (\text{set } (\text{map } \text{fst } xs))$
 $\langle \text{proof} \rangle$

lemma $\text{inj_on_the}: \text{None} \notin A \implies \text{inj_on the } A$

$\langle \text{proof} \rangle$

lemma $\text{inj_on_map_of'}:$

assumes $\text{distinct } (\text{map } \text{snd } xs)$
shows $\text{inj_on } (\text{the} \circ \text{map_of } xs) \ (\text{set } (\text{map } \text{fst } xs))$
 $\langle \text{proof} \rangle$

lemma *image_map_of*:
 assumes *distinct* (map fst xs)
 shows map_of xs ' set (map fst xs) = Some ' set (map snd xs)
 <proof>

lemma *the_Some_image* [simp]: the ' Some ' A = A
 <proof>

lemma *image_map_of'*:
 assumes *distinct* (map fst xs)
 shows (the ◦ map_of xs) ' set (map fst xs) = set (map snd xs)
 <proof>

lemma *permutation_of_list_permutes* [simp]:
 assumes *list_permutes* xs A
 shows permutation_of_list xs permutes A
 (is ?f permutes _)
 <proof>

lemma *eval_permutation_of_list* [simp]:
 permutation_of_list [] x = x
 x = x' \implies permutation_of_list ((x',y)#xs) x = y
 x \neq x' \implies permutation_of_list ((x',y')#xs) x = permutation_of_list xs x
 <proof>

lemma *eval_inverse_permutation_of_list* [simp]:
 inverse_permutation_of_list [] x = x
 x = x' \implies inverse_permutation_of_list ((y,x')#xs) x = y
 x \neq x' \implies inverse_permutation_of_list ((y',x')#xs) x = inverse_permutation_of_list
 xs x
 <proof>

lemma *permutation_of_list_id*: x \notin set (map fst xs) \implies permutation_of_list xs
 x = x
 <proof>

lemma *permutation_of_list_unique'*:
 distinct (map fst xs) \implies (x, y) \in set xs \implies permutation_of_list xs x = y
 <proof>

lemma *permutation_of_list_unique*:
 list_permutes xs A \implies (x, y) \in set xs \implies permutation_of_list xs x = y
 <proof>

lemma *inverse_permutation_of_list_id*:
 x \notin set (map snd xs) \implies inverse_permutation_of_list xs x = x
 <proof>

lemma *inverse_permutation_of_list_unique'*:
 $distinct\ (map\ snd\ xs) \implies (x, y) \in set\ xs \implies inverse_permutation_of_list\ xs\ y = x$
 <proof>

lemma *inverse_permutation_of_list_unique*:
 $list_permutes\ xs\ A \implies (x, y) \in set\ xs \implies inverse_permutation_of_list\ xs\ y = x$
 <proof>

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>

end

4 Permuted Lists

theory *List_Permutation*
imports *Permutations*
begin

Note that multisets already provide the notion of permuted 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*) $perm :: \langle 'a\ list \Rightarrow 'a\ list \Rightarrow bool \rangle$ (**infixr** $\langle <^{~~} >$ 50)
where $\langle xs\ <^{~~} ys \equiv mset\ xs = mset\ ys \rangle$

4.2 Nontrivial conclusions

proposition *perm_swap*:
 $\langle xs[i := xs ! j, j := xs ! i] <^{~~} xs \rangle$
if $\langle i < length\ xs \rangle \langle j < length\ xs \rangle$
 <proof>

proposition *mset_le_perm_append*: $mset\ xs \subseteq\# mset\ ys \longleftrightarrow (\exists\ zs.\ xs\ @\ zs <^{~~} ys)$
 <proof>

proposition *perm_set_eq*: $xs\ <^{~~} ys \implies set\ xs = set\ ys$
 <proof>

proposition *perm_distinct_iff*: $xs\ <^{~~} ys \implies distinct\ xs \longleftrightarrow distinct\ ys$
 <proof>

theorem *eq_set_perm_remdups*: $set\ xs = set\ ys \implies remdups\ xs <\sim\sim> remdups\ ys$
 <proof>

proposition *perm_remdups_iff_eq_set*: $remdups\ x <\sim\sim> remdups\ y \longleftrightarrow set\ x = set\ y$
 <proof>

theorem *permutation_Ex_bij*:
 assumes $xs <\sim\sim> ys$
 shows $\exists f. bij_betw\ f\ \{..<length\ xs\}\ \{..<length\ ys\} \wedge (\forall i < length\ xs. xs\ !\ i = ys\ !\ (f\ i))$
 <proof>

proposition *perm_finite*: $finite\ \{B. B <\sim\sim> A\}$
 <proof>

4.3 Trivial conclusions:

proposition *perm_empty_imp*: $[] <\sim\sim> ys \implies ys = []$
 <proof>

This more general theorem is easier to understand!

proposition *perm_length*: $xs <\sim\sim> ys \implies length\ xs = length\ ys$
 <proof>

proposition *perm_sym*: $xs <\sim\sim> ys \implies ys <\sim\sim> xs$
 <proof>

We can insert the head anywhere in the list.

proposition *perm_append_Cons*: $a \# xs @ ys <\sim\sim> xs @ a \# ys$
 <proof>

proposition *perm_append_swap*: $xs @ ys <\sim\sim> ys @ xs$
 <proof>

proposition *perm_append_single*: $a \# xs <\sim\sim> xs @ [a]$
 <proof>

proposition *perm_rev*: $rev\ xs <\sim\sim> xs$
 <proof>

proposition *perm_append1*: $xs <\sim\sim> ys \implies l @ xs <\sim\sim> l @ ys$
 <proof>

proposition *perm_append2*: $xs <\sim\sim> ys \implies xs @ l <\sim\sim> ys @ l$
 <proof>

proposition *perm_empty_iff*: $[] <\sim\sim> xs \longleftrightarrow xs = []$

$\langle proof \rangle$

proposition *perm_empty2* [iff]: $xs <\sim\sim> [] \longleftrightarrow xs = []$
 $\langle proof \rangle$

proposition *perm_sing_imp*: $ys <\sim\sim> xs \implies xs = [y] \implies ys = [y]$
 $\langle proof \rangle$

proposition *perm_sing_eq* [iff]: $ys <\sim\sim> [y] \longleftrightarrow ys = [y]$
 $\langle proof \rangle$

proposition *perm_sing_eq2* [iff]: $[y] <\sim\sim> ys \longleftrightarrow ys = [y]$
 $\langle proof \rangle$

proposition *perm_remove*: $x \in \text{set } ys \implies ys <\sim\sim> x \# \text{remove1 } x \text{ } ys$
 $\langle proof \rangle$

Congruence rule

proposition *perm_remove_perm*: $xs <\sim\sim> ys \implies \text{remove1 } z \text{ } xs <\sim\sim> \text{remove1 } z \text{ } ys$
 $\langle proof \rangle$

proposition *remove_hd* [simp]: $\text{remove1 } z \text{ } (z \# xs) = xs$
 $\langle proof \rangle$

proposition *cons_perm_imp_perm*: $z \# xs <\sim\sim> z \# ys \implies xs <\sim\sim> ys$
 $\langle proof \rangle$

proposition *cons_perm_eq* [simp]: $z \# xs <\sim\sim> z \# ys \longleftrightarrow xs <\sim\sim> ys$
 $\langle proof \rangle$

proposition *append_perm_imp_perm*: $zs @ xs <\sim\sim> zs @ ys \implies xs <\sim\sim> ys$
 $\langle proof \rangle$

proposition *perm_append1_eq* [iff]: $zs @ xs <\sim\sim> zs @ ys \longleftrightarrow xs <\sim\sim> ys$
 $\langle proof \rangle$

proposition *perm_append2_eq* [iff]: $xs @ zs <\sim\sim> ys @ zs \longleftrightarrow xs <\sim\sim> ys$
 $\langle proof \rangle$

end

5 Permutations of a Multiset

theory *Multiset_Permutations*

imports

Complex_Main

Permutations

begin

lemma *mset_tl*: $xs \neq [] \implies \text{mset } (\text{tl } xs) = \text{mset } xs - \{\# \text{hd } xs \#\}$
 <proof>

lemma *mset_set_image_inj*:
 assumes *inj_on* *f* *A*
 shows $\text{mset_set } (f \text{ ` } A) = \text{image_mset } f (\text{mset_set } A)$
 <proof>

lemma *multiset_remove_induct* [*case_names empty remove*]:
 assumes $P \{\#\} \wedge A. A \neq \{\#\} \implies (\bigwedge x. x \in \# A \implies P (A - \{\#x\})) \implies P$
A
 shows $P A$
 <proof>

lemma *map_list_bind*: $\text{map } g (\text{List.bind } xs \text{ } f) = \text{List.bind } xs (\text{map } g \circ f)$
 <proof>

lemma *mset_eq_mset_set_imp_distinct*:
 finite *A* $\implies \text{mset_set } A = \text{mset } xs \implies \text{distinct } xs$
 <proof>

5.1 Permutations of a multiset

definition *permutations_of_multiset* :: 'a multiset \Rightarrow 'a list set **where**
permutations_of_multiset *A* = $\{xs. \text{mset } xs = A\}$

lemma *permutations_of_multisetI*: $\text{mset } xs = A \implies xs \in \text{permutations_of_multiset } A$
 <proof>

lemma *permutations_of_multisetD*: $xs \in \text{permutations_of_multiset } A \implies \text{mset } xs = A$
 <proof>

lemma *permutations_of_multiset_Cons_iff*:
 $x \# xs \in \text{permutations_of_multiset } A \longleftrightarrow x \in \# A \wedge xs \in \text{permutations_of_multiset } (A - \{\#x\})$
 <proof>

lemma *permutations_of_multiset_empty* [*simp*]: $\text{permutations_of_multiset } \{\#\} = \{[]\}$
 <proof>

lemma *permutations_of_multiset_nonempty*:
 assumes *nonempty*: $A \neq \{\#\}$
 shows $\text{permutations_of_multiset } A =$

$(\bigcup_{x \in \text{set_mset } A} ((\#) x) \text{ 'permutations_of_multiset } (A - \{\#x\# \}))$
(is _ = ?rhs)
 $\langle \text{proof} \rangle$

lemma *permutations_of_multiset_singleton* [simp]: *permutations_of_multiset* $\{\#x\#\}$
 $= \{[x]\}$
 $\langle \text{proof} \rangle$

lemma *permutations_of_multiset_doubleton*:
permutations_of_multiset $\{\#x,y\#\} = \{[x,y], [y,x]\}$
 $\langle \text{proof} \rangle$

lemma *rev_permutations_of_multiset* [simp]:
 $\text{rev 'permutations_of_multiset } A = \text{permutations_of_multiset } A$
 $\langle \text{proof} \rangle$

lemma *length_finite_permutations_of_multiset*:
 $xs \in \text{permutations_of_multiset } A \implies \text{length } xs = \text{size } A$
 $\langle \text{proof} \rangle$

lemma *permutations_of_multiset_lists*: *permutations_of_multiset* $A \subseteq \text{lists } (\text{set_mset } A)$
 $\langle \text{proof} \rangle$

lemma *finite_permutations_of_multiset* [simp]: *finite* (*permutations_of_multiset* A)
 $\langle \text{proof} \rangle$

lemma *permutations_of_multiset_not_empty* [simp]: *permutations_of_multiset* $A \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *permutations_of_multiset_image*:
permutations_of_multiset (*image_mset* f A) = *map* f ' *permutations_of_multiset* A
 $\langle \text{proof} \rangle$

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*:
 $(\prod_{y \in \text{set_mset } (A + \{\#x\#\})} \text{fact } (\text{count } (A + \{\#x\#\}) y)) =$
 $(\text{count } A x + 1) * (\prod_{y \in \text{set_mset } A} \text{fact } (\text{count } A y))$
 $\langle \text{proof} \rangle$ **lemma** *multiset_prod_fact_remove*:
 $x \in \# A \implies (\prod_{y \in \text{set_mset } A} \text{fact } (\text{count } A y)) =$

$\text{count } A \ x * (\prod_{y \in \text{set_mset } (A - \{\#x\})}. \text{fact } (\text{count } (A - \{\#x\}) \ y))$
 $\langle \text{proof} \rangle$

lemma *card_permutations_of_multiset_aux*:
 $\text{card } (\text{permutations_of_multiset } A) * (\prod_{x \in \text{set_mset } A}. \text{fact } (\text{count } A \ x)) = \text{fact } (\text{size } A)$
 $\langle \text{proof} \rangle$

theorem *card_permutations_of_multiset*:
 $\text{card } (\text{permutations_of_multiset } A) = \text{fact } (\text{size } A) \text{ div } (\prod_{x \in \text{set_mset } A}. \text{fact } (\text{count } A \ x))$
 $(\prod_{x \in \text{set_mset } A}. \text{fact } (\text{count } A \ x) :: \text{nat}) \text{ dvd fact } (\text{size } A)$
 $\langle \text{proof} \rangle$

lemma *card_permutations_of_multiset_insert_aux*:
 $\text{card } (\text{permutations_of_multiset } (A + \{\#x\})) * (\text{count } A \ x + 1) =$
 $(\text{size } A + 1) * \text{card } (\text{permutations_of_multiset } A)$
 $\langle \text{proof} \rangle$

lemma *card_permutations_of_multiset_remove_aux*:
assumes $x \in \# \ A$
shows $\text{card } (\text{permutations_of_multiset } A) * \text{count } A \ x =$
 $\text{size } A * \text{card } (\text{permutations_of_multiset } (A - \{\#x\}))$
 $\langle \text{proof} \rangle$

lemma *real_card_permutations_of_multiset_remove*:
assumes $x \in \# \ A$
shows $\text{real } (\text{card } (\text{permutations_of_multiset } (A - \{\#x\}))) =$
 $\text{real } (\text{card } (\text{permutations_of_multiset } A) * \text{count } A \ x) / \text{real } (\text{size } A)$
 $\langle \text{proof} \rangle$

lemma *real_card_permutations_of_multiset_remove'*:
assumes $x \in \# \ A$
shows $\text{real } (\text{card } (\text{permutations_of_multiset } A)) =$
 $\text{real } (\text{size } A * \text{card } (\text{permutations_of_multiset } (A - \{\#x\}))) / \text{real } (\text{count } A \ x)$
 $\langle \text{proof} \rangle$

end

5.3 Permutations of a set

definition *permutations_of_set* :: 'a set \Rightarrow 'a list set **where**
 $\text{permutations_of_set } A = \{xs. \text{set } xs = A \wedge \text{distinct } xs\}$

lemma *permutations_of_set_altdef*:
 $\text{finite } A \Longrightarrow \text{permutations_of_set } A = \text{permutations_of_multiset } (\text{mset_set } A)$
 $\langle \text{proof} \rangle$

lemma *permutations_of_setI* [intro]:
 assumes *set xs = A distinct xs*
 shows *xs ∈ permutations_of_set A*
 ⟨proof⟩

lemma *permutations_of_setD*:
 assumes *xs ∈ permutations_of_set A*
 shows *set xs = A distinct xs*
 ⟨proof⟩

lemma *permutations_of_set_lists*: *permutations_of_set A ⊆ lists A*
 ⟨proof⟩

lemma *permutations_of_set_empty* [simp]: *permutations_of_set {} = {[]}*
 ⟨proof⟩

lemma *UN_set_permutations_of_set* [simp]:
 $\text{finite } A \implies (\bigcup_{xs \in \text{permutations_of_set } A} \text{set } xs) = A$
 ⟨proof⟩

lemma *permutations_of_set_infinite*:
 $\neg \text{finite } A \implies \text{permutations_of_set } A = \{\}$
 ⟨proof⟩

lemma *permutations_of_set_nonempty*:
 $A \neq \{\} \implies \text{permutations_of_set } A =$
 $(\bigcup_{x \in A} (\lambda xs. x \# xs)) \text{ 'permutations_of_set } (A - \{x\}))$
 ⟨proof⟩

lemma *permutations_of_set_singleton* [simp]: *permutations_of_set {x} = {[x]}*
 ⟨proof⟩

lemma *permutations_of_set_doubleton*:
 $x \neq y \implies \text{permutations_of_set } \{x, y\} = \{[x, y], [y, x]\}$
 ⟨proof⟩

lemma *rev_permutations_of_set* [simp]:
 $\text{rev 'permutations_of_set } A = \text{permutations_of_set } A$
 ⟨proof⟩

lemma *length_finite_permutations_of_set*:
 $xs \in \text{permutations_of_set } A \implies \text{length } xs = \text{card } A$
 ⟨proof⟩

lemma *finite_permutations_of_set* [simp]: *finite (permutations_of_set A)*
 ⟨proof⟩

lemma *permutations_of_set_empty_iff* [simp]:

permutations_of_set $A = \{\}$ $\longleftrightarrow \neg \text{finite } A$
 ⟨proof⟩

lemma *card_permutations_of_set* [simp]:
 $\text{finite } A \implies \text{card } (\text{permutations_of_set } A) = \text{fact } (\text{card } A)$
 ⟨proof⟩

lemma *permutations_of_set_image_inj*:
assumes *inj*: *inj_on* f A
shows $\text{permutations_of_set } (f \text{ ` } A) = \text{map } f \text{ ` } \text{permutations_of_set } A$
 ⟨proof⟩

lemma *permutations_of_set_image_permutes*:
 $\sigma \text{ permutes } A \implies \text{map } \sigma \text{ ` } \text{permutations_of_set } A = \text{permutations_of_set } A$
 ⟨proof⟩

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**
 $\text{permutations_of_list_impl } xs = (\text{if } xs = [] \text{ then } [] \text{ else } \\ \text{List.bind } (\text{remdups } xs) (\lambda x. \text{map } ((\#) \ x) (\text{permutations_of_list_impl } (\text{remove1 } x \ xs))))$

fun *permutations_of_list_impl_aux* **where**
 $\text{permutations_of_list_impl_aux } acc \ xs = (\text{if } xs = [] \text{ then } [acc] \text{ else } \\ \text{List.bind } (\text{remdups } xs) (\lambda x. \text{permutations_of_list_impl_aux } (x\#\text{acc}) (\text{remove1 } x \ xs))))$

declare *permutations_of_list_impl_aux.simps* [simp del]

declare *permutations_of_list_impl.simps* [simp del]

lemma *permutations_of_list_impl_Nil* [simp]:
 $\text{permutations_of_list_impl } [] = []$
 ⟨proof⟩

lemma *permutations_of_list_impl_nonempty*:
 $xs \neq [] \implies \text{permutations_of_list_impl } xs = \\ \text{List.bind } (\text{remdups } xs) (\lambda x. \text{map } ((\#) \ x) (\text{permutations_of_list_impl } (\text{remove1 } x \ xs))))$
 ⟨proof⟩

lemma *set_permutations_of_list_impl*:
 $\text{set } (\text{permutations_of_list_impl } xs) = \text{permutations_of_multiset } (\text{mset } xs)$
 ⟨proof⟩

lemma *distinct_permutations_of_list_impl*:

distinct (*permutations_of_list_impl xs*)
 ⟨*proof*⟩

lemma *permutations_of_list_impl_aux_correct'*:
permutations_of_list_impl_aux acc xs =
map (λxs. rev xs @ acc) (permutations_of_list_impl xs)
 ⟨*proof*⟩

lemma *permutations_of_list_impl_aux_correct*:
permutations_of_list_impl_aux [] xs = map rev (permutations_of_list_impl xs)
 ⟨*proof*⟩

lemma *distinct_permutations_of_list_impl_aux*:
distinct (permutations_of_list_impl_aux acc xs)
 ⟨*proof*⟩

lemma *set_permutations_of_list_impl_aux*:
set (permutations_of_list_impl_aux [] xs) = permutations_of_multiset (mset xs)
 ⟨*proof*⟩

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})))
 ⟨*proof*⟩
termination ⟨*proof*⟩

lemma *permutations_of_set_aux_altdef*:
permutations_of_set_aux acc A = (λxs. rev xs @ acc) ‘ permutations_of_set A
 ⟨*proof*⟩

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

lemma *permutations_of_set_aux_correct*:
permutations_of_set_aux [] A = permutations_of_set A
 ⟨*proof*⟩

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)

⟨proof⟩

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)

⟨proof⟩

lemma *distinct_permutations_of_set_list*:

distinct xs ⇒ distinct (permutations_of_set_list xs)

⟨proof⟩

lemma *permutations_of_list*:

permutations_of_set (set xs) = set (permutations_of_set_list (remdups xs))

⟨proof⟩

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))

⟨proof⟩

value [code] *permutations_of_set* (set "abcd")

end

theory *Cycles*

imports

HOL-Library.FuncSet

Permutations

begin

6 Cycles

6.1 Definitions

abbreviation $\text{cycle} :: 'a \text{ list} \Rightarrow \text{bool}$
where $\text{cycle } cs \equiv \text{distinct } cs$

fun $\text{cycle_of_list} :: 'a \text{ list} \Rightarrow 'a \Rightarrow 'a$
where
 $\text{cycle_of_list } (i \# j \# cs) = \text{transpose } i \ j \circ \text{cycle_of_list } (j \# cs)$
 $| \text{cycle_of_list } cs = \text{id}$

6.2 Basic Properties

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

lemma id_outside_supp :
 assumes $x \notin \text{set } cs$ **shows** $(\text{cycle_of_list } cs) \ x = x$
 $\langle \text{proof} \rangle$

lemma $\text{permutation_of_cycle}$: $\text{permutation } (\text{cycle_of_list } cs)$
 $\langle \text{proof} \rangle$

lemma cycle_permutes : $(\text{cycle_of_list } cs) \text{ permutes } (\text{set } cs)$
 $\langle \text{proof} \rangle$

theorem cyclic_rotation :
 assumes $\text{cycle } cs$ **shows** $\text{map } ((\text{cycle_of_list } cs) \ \sim^n) \ cs = \text{rotate } n \ cs$
 $\langle \text{proof} \rangle$

corollary cycle_is_surj :
 assumes $\text{cycle } cs$ **shows** $(\text{cycle_of_list } cs) \ `(\text{set } cs) = (\text{set } cs)$
 $\langle \text{proof} \rangle$

corollary cycle_is_id_root :
 assumes $\text{cycle } cs$ **shows** $(\text{cycle_of_list } cs) \ \sim^{(\text{length } cs)} = \text{id}$
 $\langle \text{proof} \rangle$

corollary $\text{cycle_of_list_rotate_independent}$:
 assumes $\text{cycle } cs$ **shows** $(\text{cycle_of_list } cs) = (\text{cycle_of_list } (\text{rotate } n \ cs))$
 $\langle \text{proof} \rangle$

6.3 Conjugation of cycles

lemma $\text{conjugation_of_cycle}$:
 assumes $\text{cycle } cs$ **and** $\text{bij } p$
 shows $p \circ (\text{cycle_of_list } cs) \circ (\text{inv } p) = \text{cycle_of_list } (\text{map } p \ cs)$
 $\langle \text{proof} \rangle$

6.4 When Cycles Commute

lemma *cycles_commute*:

assumes *cycle p cycle q and set p \cap set q = {}*
shows $(\text{cycle_of_list } p) \circ (\text{cycle_of_list } q) = (\text{cycle_of_list } q) \circ (\text{cycle_of_list } p)$
 $\langle \text{proof} \rangle$

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 \frown n)*
 $\langle \text{proof} \rangle$

lemma *permutes_funpow*:

assumes *p permutes S* **shows** *(p \frown n) permutes S*
 $\langle \text{proof} \rangle$

lemma *funpow_diff*:

assumes *inj p and i \leq j* *(p \frown i) a = (p \frown j) a* **shows** *(p \frown (j - i)) a = a*
 $\langle \text{proof} \rangle$

lemma *permutation_is_nilpotent*:

assumes *permutation p obtains n where (p \frown n) = id and n > 0*
 $\langle \text{proof} \rangle$

lemma *permutation_is_nilpotent'*:

assumes *permutation p obtains n where (p \frown n) = id and n > m*
 $\langle \text{proof} \rangle$

6.5.2 Extraction of cycles from permutations

definition *least_power* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow \text{nat}$

where *least_power f x = (LEAST n. (f \frown n) x = x \wedge n > 0)*

abbreviation *support* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \text{ list}$

where *support p x \equiv map ($\lambda i. (p \frown i) x$) [0.. $(\text{least_power } p \ x)$]*

lemma *least_powerI*:

assumes $(f \frown n) \ x = x$ **and** $n > 0$
shows $(f \frown (\text{least_power } f \ x)) \ x = x$ **and** $\text{least_power } f \ x > 0$
 $\langle \text{proof} \rangle$

lemma *least_power_le*:

assumes $(f \frown n) \ x = x$ **and** $n > 0$ **shows** $\text{least_power } f \ x \leq n$

$\langle \text{proof} \rangle$

lemma *least_power_of_permutation*:

assumes *permutation p* **shows** $(p \rightsquigarrow (\text{least_power } p \ a)) \ a = a$ **and** *least_power*
p a > 0
 $\langle \text{proof} \rangle$

lemma *least_power_gt_one*:

assumes *permutation p* **and** $p \ a \neq a$ **shows** *least_power p a > Suc 0*
 $\langle \text{proof} \rangle$

lemma *least_power_minimal*:

assumes $(p \rightsquigarrow n) \ a = a$ **shows** $(\text{least_power } p \ a) \ \text{dvd} \ n$
 $\langle \text{proof} \rangle$

lemma *least_power_dvd*:

assumes *permutation p* **shows** $(\text{least_power } p \ a) \ \text{dvd} \ n \longleftrightarrow (p \rightsquigarrow n) \ a = a$
 $\langle \text{proof} \rangle$

theorem *cycle_of_permutation*:

assumes *permutation p* **shows** *cycle (support p a)*
 $\langle \text{proof} \rangle$

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** $\text{set} (\text{support } p \ a) = \text{range} (\lambda i. (p \rightsquigarrow i) \ a)$
 $\langle \text{proof} \rangle$

lemma *disjoint_support*:

assumes *permutation p* **shows** $\text{disjoint} (\text{range} (\lambda a. \text{set} (\text{support } p \ a)))$ **(is disjoint**
?A)
 $\langle \text{proof} \rangle$

lemma *disjoint_support'*:

assumes *permutation p*
shows $\text{set} (\text{support } p \ a) \cap \text{set} (\text{support } p \ b) = \{\} \longleftrightarrow a \notin \text{set} (\text{support } p \ b)$
 $\langle \text{proof} \rangle$

lemma *support_coverture*:

assumes *permutation p* **shows** $\bigcup \{ \text{set} (\text{support } p \ a) \mid a. p \ a \neq a \} = \{ a. p \ a \neq a \}$
 $\langle \text{proof} \rangle$

theorem *cycle_restrict*:

assumes *permutation p* **and** $b \in \text{set } (\text{support } p \ a)$ **shows** $p \ b = (\text{cycle_of_list } (\text{support } p \ a)) \ b$
 $\langle \text{proof} \rangle$

6.6.2 Decomposition

inductive *cycle_decomp* :: $'a \text{ set} \Rightarrow ('a \Rightarrow 'a) \Rightarrow \text{bool}$
where
empty: $\text{cycle_decomp } \{\} \text{ id}$
| *comp*: $\llbracket \text{cycle_decomp } I \ p; \text{ cycle } cs; \text{ set } cs \cap I = \{\} \rrbracket \Longrightarrow$
 $\text{cycle_decomp } (\text{set } cs \cup I) ((\text{cycle_of_list } cs) \circ p)$

lemma *semidecomposition*:

assumes $p \text{ permutes } S$ **and** *finite S*
shows $(\lambda y. \text{ if } y \in (S - \text{set } (\text{support } p \ a)) \text{ then } p \ y \text{ else } y) \text{ permutes } (S - \text{set } (\text{support } p \ a))$
 $\langle \text{proof} \rangle$

theorem *cycle_decomposition*:

assumes $p \text{ permutes } S$ **and** *finite S* **shows** $\text{cycle_decomp } S \ p$
 $\langle \text{proof} \rangle$

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. Grievously 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 $'a \text{ perm} = \{f :: 'a \Rightarrow 'a. \text{bij } f \wedge \text{finite } \{a. f \ a \neq a\}\}$
morphisms *apply Perm*
 $\langle \text{proof} \rangle$

setup_lifting *type_definition_perm*

notation *apply* (**infixl** $\langle \$ \rangle$ 999)

lemma *bij_apply* [*simp*]:

bij (*apply* *f*)

⟨*proof*⟩

lemma *perm_eqI*:

assumes $\bigwedge a. f \langle \$ \rangle a = g \langle \$ \rangle a$

shows $f = g$

⟨*proof*⟩

lemma *perm_eq_iff*:

$f = g \longleftrightarrow (\forall a. f \langle \$ \rangle a = g \langle \$ \rangle a)$

⟨*proof*⟩

lemma *apply_inj*:

$f \langle \$ \rangle a = f \langle \$ \rangle b \longleftrightarrow a = b$

⟨*proof*⟩

lift_definition *affected* :: $'a \text{ perm} \Rightarrow 'a \text{ set}$

is $\lambda f. \{a. f a \neq a\}$ ⟨*proof*⟩

lemma *in_affected*:

$a \in \text{affected } f \longleftrightarrow f \langle \$ \rangle a \neq a$

⟨*proof*⟩

lemma *finite_affected* [*simp*]:

finite (*affected* *f*)

⟨*proof*⟩

lemma *apply_affected* [*simp*]:

$f \langle \$ \rangle a \in \text{affected } f \longleftrightarrow a \in \text{affected } f$

⟨*proof*⟩

lemma *card_affected_not_one*:

$\text{card } (\text{affected } f) \neq 1$

⟨*proof*⟩

7.2 Identity, composition and inversion

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

begin

lift_definition *one_perm* :: $'a \text{ perm}$

is *id*

⟨*proof*⟩

lemma *apply_one* [*simp*]:

apply 1 = *id*

⟨*proof*⟩

```

lemma affected_one [simp]:
  affected 1 = {}
  ⟨proof⟩

lemma affected_empty_iff [simp]:
  affected f = {}  $\longleftrightarrow$  f = 1
  ⟨proof⟩

lift_definition times_perm :: 'a perm  $\Rightarrow$  'a perm  $\Rightarrow$  'a perm
  is comp
  ⟨proof⟩

lemma apply_times:
  apply (f * g) = apply f  $\circ$  apply g
  ⟨proof⟩

lemma apply_sequence:
  f <$> (g <$> a) = apply (f * g) a
  ⟨proof⟩

lemma affected_times [simp]:
  affected (f * g)  $\subseteq$  affected f  $\cup$  affected g
  ⟨proof⟩

lift_definition inverse_perm :: 'a perm  $\Rightarrow$  'a perm
  is inv
  ⟨proof⟩

instance
  ⟨proof⟩

end

lemma apply_inverse:
  apply (inverse f) = inv (apply f)
  ⟨proof⟩

lemma affected_inverse [simp]:
  affected (inverse f) = affected f
  ⟨proof⟩

global_interpretation perm: group times 1 :: 'a perm inverse
  ⟨proof⟩

declare perm.inverse_distrib_swap [simp]

lemma perm_mult_commute:
  assumes affected f  $\cap$  affected g = {}
  shows g * f = f * g

```

$\langle proof \rangle$

lemma *apply_power*:
 $apply (f \wedge n) = apply f \wedge n$
 $\langle proof \rangle$

lemma *perm_power_inverse*:
 $inverse f \wedge n = inverse ((f :: 'a \text{ perm}) \wedge n)$
 $\langle proof \rangle$

7.3 Orbit and order of elements

definition *orbit* :: $'a \text{ perm} \Rightarrow 'a \Rightarrow 'a \text{ set}$
where
 $orbit f a = range (\lambda n. (f \wedge n) \langle \$ \rangle a)$

lemma *in_orbitI*:
 assumes $(f \wedge n) \langle \$ \rangle a = b$
 shows $b \in orbit f a$
 $\langle proof \rangle$

lemma *apply_power_self_in_orbit* [simp]:
 $(f \wedge n) \langle \$ \rangle a \in orbit f a$
 $\langle proof \rangle$

lemma *in_orbit_self* [simp]:
 $a \in orbit f a$
 $\langle proof \rangle$

lemma *apply_self_in_orbit* [simp]:
 $f \langle \$ \rangle a \in orbit f a$
 $\langle proof \rangle$

lemma *orbit_not_empty* [simp]:
 $orbit f a \neq \{\}$
 $\langle proof \rangle$

lemma *not_in_affected_iff_orbit_eq_singleton*:
 $a \notin affected f \longleftrightarrow orbit f a = \{a\} \text{ (is } ?P \longleftrightarrow ?Q)$
 $\langle proof \rangle$

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

lemma *orbit_subset_eq_affected*:
 assumes $a \in affected f$
 shows $orbit f a \subseteq affected f$
 $\langle proof \rangle$

lemma *finite_orbit* [*simp*]:
 finite (*orbit* *f* *a*)
 ⟨*proof*⟩

lemma *orbit_1* [*simp*]:
 orbit 1 *a* = {*a*}
 ⟨*proof*⟩

lemma *order_1* [*simp*]:
 order 1 *a* = 1
 ⟨*proof*⟩

lemma *card_orbit_eq* [*simp*]:
 card (*orbit* *f* *a*) = *order* *f* *a*
 ⟨*proof*⟩

lemma *order_greater_zero* [*simp*]:
 order *f* *a* > 0
 ⟨*proof*⟩

lemma *order_eq_one_iff*:
 order *f* *a* = *Suc* 0 \longleftrightarrow *a* \notin *affected* *f* (**is** ?*P* \longleftrightarrow ?*Q*)
 ⟨*proof*⟩

lemma *order_greater_eq_two_iff*:
 order *f* *a* \geq 2 \longleftrightarrow *a* \in *affected* *f*
 ⟨*proof*⟩

lemma *order_less_eq_affected*:
 assumes *f* \neq 1
 shows *order* *f* *a* \leq *card* (*affected* *f*)
 ⟨*proof*⟩

lemma *affected_order_greater_eq_two*:
 assumes *a* \in *affected* *f*
 shows *order* *f* *a* \geq 2
 ⟨*proof*⟩

lemma *order_witness_unfold*:
 assumes *n* > 0 **and** (*f* \wedge *n*) (\$) *a* = *a*
 shows *order* *f* *a* = *card* (($\lambda m.$ (*f* \wedge *m*) (\$) *a*) ‘ {0..*n*})
 ⟨*proof*⟩

lemma *inj_on_apply_range*:
 inj_on ($\lambda m.$ (*f* \wedge *m*) (\$) *a*) {..*order* *f* *a*}
 ⟨*proof*⟩

lemma *orbit_unfold_image*:

$orbit\ f\ a = (\lambda n. (f \wedge n)\ \langle \$ \rangle\ a)\ \text{'}\ \{..<order\ f\ a\}\ (\text{is } _ = ?A)$
 $\langle proof \rangle$

lemma *in_orbitE*:

assumes $b \in orbit\ f\ a$

obtains n **where** $b = (f \wedge n)\ \langle \$ \rangle\ a$ **and** $n < order\ f\ a$

$\langle proof \rangle$

lemma *apply_power_order* [simp]:

$(f \wedge order\ f\ a)\ \langle \$ \rangle\ a = a$

$\langle proof \rangle$

lemma *apply_power_left_mult_order* [simp]:

$(f \wedge (n * order\ f\ a))\ \langle \$ \rangle\ a = a$

$\langle proof \rangle$

lemma *apply_power_right_mult_order* [simp]:

$(f \wedge (order\ f\ a * n))\ \langle \$ \rangle\ a = a$

$\langle proof \rangle$

lemma *apply_power_mod_order_eq* [simp]:

$(f \wedge (n \bmod order\ f\ a))\ \langle \$ \rangle\ a = (f \wedge n)\ \langle \$ \rangle\ a$

$\langle proof \rangle$

lemma *apply_power_eq_iff*:

$(f \wedge m)\ \langle \$ \rangle\ a = (f \wedge n)\ \langle \$ \rangle\ a \longleftrightarrow m \bmod order\ f\ a = n \bmod order\ f\ a\ (\text{is } ?P \longleftrightarrow ?Q)$

$\langle proof \rangle$

lemma *apply_inverse_eq_apply_power_order_minus_one*:

$(inverse\ f)\ \langle \$ \rangle\ a = (f \wedge (order\ f\ a - 1))\ \langle \$ \rangle\ a$

$\langle proof \rangle$

lemma *apply_inverse_self_in_orbit* [simp]:

$(inverse\ f)\ \langle \$ \rangle\ a \in orbit\ f\ a$

$\langle proof \rangle$

lemma *apply_inverse_power_eq*:

$(inverse\ (f \wedge n))\ \langle \$ \rangle\ a = (f \wedge (order\ f\ a - n \bmod order\ f\ a))\ \langle \$ \rangle\ a$

$\langle proof \rangle$

lemma *apply_power_eq_self_iff*:

$(f \wedge n)\ \langle \$ \rangle\ a = a \longleftrightarrow order\ f\ a\ dvd\ n$

$\langle proof \rangle$

lemma *orbit_equiv*:

assumes $b \in orbit\ f\ a$

shows $orbit\ f\ b = orbit\ f\ a\ (\text{is } ?B = ?A)$

$\langle proof \rangle$

lemma *orbit_apply [simp]:*
 $\text{orbit } f \ (f \ \langle \$ \rangle \ a) = \text{orbit } f \ a$
 $\langle \text{proof} \rangle$

lemma *order_apply [simp]:*
 $\text{order } f \ (f \ \langle \$ \rangle \ a) = \text{order } f \ a$
 $\langle \text{proof} \rangle$

lemma *orbit_apply_inverse [simp]:*
 $\text{orbit } f \ (\text{inverse } f \ \langle \$ \rangle \ a) = \text{orbit } f \ a$
 $\langle \text{proof} \rangle$

lemma *order_apply_inverse [simp]:*
 $\text{order } f \ (\text{inverse } f \ \langle \$ \rangle \ a) = \text{order } f \ a$
 $\langle \text{proof} \rangle$

lemma *orbit_apply_power [simp]:*
 $\text{orbit } f \ ((f \wedge n) \ \langle \$ \rangle \ a) = \text{orbit } f \ a$
 $\langle \text{proof} \rangle$

lemma *order_apply_power [simp]:*
 $\text{order } f \ ((f \wedge n) \ \langle \$ \rangle \ a) = \text{order } f \ a$
 $\langle \text{proof} \rangle$

lemma *orbit_inverse [simp]:*
 $\text{orbit } (\text{inverse } f) = \text{orbit } f$
 $\langle \text{proof} \rangle$

lemma *order_inverse [simp]:*
 $\text{order } (\text{inverse } f) = \text{order } f$
 $\langle \text{proof} \rangle$

lemma *orbit_disjoint:*
assumes $\text{orbit } f \ a \neq \text{orbit } f \ b$
shows $\text{orbit } f \ a \cap \text{orbit } f \ b = \{\}$
 $\langle \text{proof} \rangle$

7.4 Swaps

lift_definition *swap* :: $'a \Rightarrow 'a \Rightarrow 'a \text{ perm}$ $(\langle \langle _ \leftrightarrow _ \rangle \rangle)$
is $\lambda a \ b. \text{transpose } a \ b$
 $\langle \text{proof} \rangle$

lemma *apply_swap_simp [simp]:*
 $\langle a \leftrightarrow b \rangle \ \langle \$ \rangle \ a = b$
 $\langle a \leftrightarrow b \rangle \ \langle \$ \rangle \ b = a$
 $\langle \text{proof} \rangle$

lemma *apply_swap_same* [simp]:
 $c \neq a \implies c \neq b \implies \langle a \leftrightarrow b \rangle \langle \$ \rangle c = c$
 <proof>

lemma *apply_swap_eq_iff* [simp]:
 $\langle a \leftrightarrow b \rangle \langle \$ \rangle c = a \iff c = b$
 $\langle a \leftrightarrow b \rangle \langle \$ \rangle c = b \iff c = a$
 <proof>

lemma *swap_1* [simp]:
 $\langle a \leftrightarrow a \rangle = 1$
 <proof>

lemma *swap_sym*:
 $\langle b \leftrightarrow a \rangle = \langle a \leftrightarrow b \rangle$
 <proof>

lemma *swap_self* [simp]:
 $\langle a \leftrightarrow b \rangle * \langle a \leftrightarrow b \rangle = 1$
 <proof>

lemma *affected_swap*:
 $a \neq b \implies \text{affected } \langle a \leftrightarrow b \rangle = \{a, b\}$
 <proof>

lemma *inverse_swap* [simp]:
 $\text{inverse } \langle a \leftrightarrow b \rangle = \langle a \leftrightarrow b \rangle$
 <proof>

7.5 Permutations specified by cycles

fun *cycle* :: 'a list \Rightarrow 'a perm ($\langle _ \rangle$)
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* ($\langle _ \leftrightarrow _ \rangle$)
notation *cycle* ($\langle _ \rangle$)
notation *apply* (**infixl** $\langle _ \$ \rangle$ 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* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \text{ set}$ **for** $f\ x$ **where**

base: $f\ x \in \text{orbit}\ f\ x \mid$

step: $y \in \text{orbit}\ f\ x \implies f\ y \in \text{orbit}\ f\ x$

definition *cyclic_on* :: $('a \Rightarrow 'a) \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$ **where**

cyclic_on $f\ S \longleftrightarrow (\exists s \in S. S = \text{orbit}\ f\ s)$

lemma *orbit_altdef*: $\text{orbit}\ f\ x = \{(f \smallfrown n)\ x \mid n. 0 < n\}$ (**is** $?L = ?R$)

<proof>

lemma *orbit_trans*:

assumes $s \in \text{orbit}\ f\ t \ t \in \text{orbit}\ f\ u$ **shows** $s \in \text{orbit}\ f\ u$

<proof>

lemma *orbit_subset*:

assumes $s \in \text{orbit}\ f\ (f\ t)$ **shows** $s \in \text{orbit}\ f\ t$

<proof>

lemma *orbit_sim_step*:

assumes $s \in \text{orbit}\ f\ t$ **shows** $f\ s \in \text{orbit}\ f\ (f\ t)$

<proof>

lemma *orbit_step*:

assumes $y \in \text{orbit}\ f\ x \ f\ x \neq y$ **shows** $y \in \text{orbit}\ f\ (f\ x)$

<proof>

lemma *self_in_orbit_trans*:

assumes $s \in \text{orbit}\ f\ s \ t \in \text{orbit}\ f\ s$ **shows** $t \in \text{orbit}\ f\ t$

<proof>

lemma *orbit_swap*:

assumes $s \in \text{orbit}\ f\ s \ t \in \text{orbit}\ f\ s$ **shows** $s \in \text{orbit}\ f\ t$

<proof>

lemma *permutation_self_in_orbit*:

assumes *permutation f* **shows** $s \in \text{orbit } f \ s$
 $\langle \text{proof} \rangle$

lemma *orbit_altdef_self_in*:
assumes $s \in \text{orbit } f \ s$ **shows** $\text{orbit } f \ s = \{(f \smallfrown n) \ s \mid n. \text{True}\}$
 $\langle \text{proof} \rangle$

lemma *orbit_altdef_permutation*:
assumes *permutation f* **shows** $\text{orbit } f \ s = \{(f \smallfrown n) \ s \mid n. \text{True}\}$
 $\langle \text{proof} \rangle$

lemma *orbit_altdef_bounded*:
assumes $(f \smallfrown n) \ s = s \ 0 < n$ **shows** $\text{orbit } f \ s = \{(f \smallfrown m) \ s \mid m. m < n\}$
 $\langle \text{proof} \rangle$

lemma *funpow_in_orbit*:
assumes $s \in \text{orbit } f \ t$ **shows** $(f \smallfrown n) \ s \in \text{orbit } f \ t$
 $\langle \text{proof} \rangle$

lemma *finite_orbit*:
assumes $s \in \text{orbit } f \ s$ **shows** *finite* ($\text{orbit } f \ s$)
 $\langle \text{proof} \rangle$

lemma *self_in_orbit_step*:
assumes $s \in \text{orbit } f \ s$ **shows** $\text{orbit } f \ (f \ s) = \text{orbit } f \ s$
 $\langle \text{proof} \rangle$

lemma *permutation_orbit_step*:
assumes *permutation f* **shows** $\text{orbit } f \ (f \ s) = \text{orbit } f \ s$
 $\langle \text{proof} \rangle$

lemma *orbit_nonempty*:
 $\text{orbit } f \ s \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *orbit_inv_eq*:
assumes *permutation f*
shows $\text{orbit } (\text{inv } f) \ x = \text{orbit } f \ x$ (**is** $?L = ?R$)
 $\langle \text{proof} \rangle$

lemma *cyclic_on_alldef*:
 $\text{cyclic_on } f \ S \longleftrightarrow S \neq \{\} \wedge (\forall s \in S. S = \text{orbit } f \ s)$
 $\langle \text{proof} \rangle$

lemma *cyclic_on_funpow_in*:
assumes $\text{cyclic_on } f \ S \ s \in S$ **shows** $(f \smallfrown n) \ s \in S$
 $\langle \text{proof} \rangle$

lemma *finite_cyclic_on*:

assumes *cyclic_on* f S **shows** *finite* S
 ⟨*proof*⟩

lemma *cyclic_on_singleI*:
assumes $s \in S$ $S = \text{orbit } f \ s$ **shows** *cyclic_on* f S
 ⟨*proof*⟩

lemma *cyclic_on_inI*:
assumes *cyclic_on* f S $s \in S$ **shows** $f \ s \in S$
 ⟨*proof*⟩

lemma *orbit_inverse*:
assumes *self*: $a \in \text{orbit } g \ a$
and *eq*: $\bigwedge x. x \in \text{orbit } g \ a \implies g' (f \ x) = f (g \ x)$
shows $f \ ' \ \text{orbit } g \ a = \text{orbit } g' (f \ a)$ (**is** $?L = ?R$)
 ⟨*proof*⟩

lemma *cyclic_on_image*:
assumes *cyclic_on* f S
assumes $\bigwedge x. x \in S \implies g (h \ x) = h (f \ x)$
shows *cyclic_on* g $(h \ ' \ S)$
 ⟨*proof*⟩

lemma *cyclic_on_f_in*:
assumes f *permutes* S *cyclic_on* f A $f \ x \in A$
shows $x \in A$
 ⟨*proof*⟩

lemma *orbit_cong0*:
assumes $x \in A$ $f \in A \rightarrow A$ $\bigwedge y. y \in A \implies f \ y = g \ y$ **shows** $\text{orbit } f \ x = \text{orbit } g \ x$
 ⟨*proof*⟩

lemma *orbit_cong*:
assumes *self_in*: $t \in \text{orbit } f \ t$ **and** *eq*: $\bigwedge s. s \in \text{orbit } f \ t \implies g \ s = f \ s$
shows $\text{orbit } g \ t = \text{orbit } f \ t$
 ⟨*proof*⟩

lemma *cyclic_cong*:
assumes $\bigwedge s. s \in S \implies f \ s = g \ s$ **shows** *cyclic_on* f $S = \text{cyclic_on } g \ S$
 ⟨*proof*⟩

lemma *permutes_comp_preserves_cyclic1*:
assumes g *permutes* B *cyclic_on* f C
assumes $A \cap B = \{\}$ $C \subseteq A$
shows *cyclic_on* $(f \circ g)$ C
 ⟨*proof*⟩

lemma *permutes_comp_preserves_cyclic2*:

assumes f permutes A *cyclic_on* g C
assumes $A \cap B = \{\}$ $C \subseteq B$
shows *cyclic_on* $(f \circ g)$ C
 $\langle \text{proof} \rangle$

lemma *permutes_orbit_subset*:
assumes f permutes S $x \in S$ **shows** *orbit* f $x \subseteq S$
 $\langle \text{proof} \rangle$

lemma *cyclic_on_orbit'*:
assumes *permutation* f **shows** *cyclic_on* f (*orbit* f x)
 $\langle \text{proof} \rangle$

lemma *cyclic_on_orbit*:
assumes f permutes S *finite* S **shows** *cyclic_on* f (*orbit* f x)
 $\langle \text{proof} \rangle$

lemma *orbit_cyclic_eq3*:
assumes *cyclic_on* f S $y \in S$ **shows** *orbit* f $y = S$
 $\langle \text{proof} \rangle$

lemma *orbit_eq_singleton_iff*: *orbit* f $x = \{x\} \longleftrightarrow f$ $x = x$ (**is** $?L \longleftrightarrow ?R$)
 $\langle \text{proof} \rangle$

lemma *eq_on_cyclic_on_iff1*:
assumes *cyclic_on* f S $x \in S$
obtains f $x \in S$ f $x = x \longleftrightarrow \text{card } S = 1$
 $\langle \text{proof} \rangle$

lemma *orbit_eqI*:
 $y = f$ $x \implies y \in \text{orbit } f$ x
 $z = f$ $y \implies y \in \text{orbit } f$ $x \implies z \in \text{orbit } f$ x
 $\langle \text{proof} \rangle$

8.2 Decomposition of arbitrary permutations

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

lemma *perm_restrict_comp*:
assumes $A \cap B = \{\}$ *cyclic_on* f B
shows $\text{perm_restrict } f$ $A \circ \text{perm_restrict } f$ $B = \text{perm_restrict } f$ $(A \cup B)$
 $\langle \text{proof} \rangle$

lemma *perm_restrict_simps*:
 $x \in S \implies \text{perm_restrict } f$ S $x = f$ x
 $x \notin S \implies \text{perm_restrict } f$ S $x = x$
 $\langle \text{proof} \rangle$

lemma *perm_restrict_perm_restrict*:

perm_restrict (*perm_restrict* *f* *A*) *B* = *perm_restrict* *f* (*A* \cap *B*)

\langle *proof* \rangle

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* \circ *perm_restrict* *f* *B* = *perm_restrict* *f* (*A* \cup *B*)

\langle *proof* \rangle

lemma *perm_restrict_id*[*simp*]:

assumes *f* *permutes* *S* **shows** *perm_restrict* *f* *S* = *f*

\langle *proof* \rangle

lemma *cyclic_on_perm_restrict*:

cyclic_on (*perm_restrict* *f* *S*) *S* \longleftrightarrow *cyclic_on* *f* *S*

\langle *proof* \rangle

lemma *perm_restrict_diff_cyclic*:

assumes *f* *permutes* *S* *cyclic_on* *f* *A*

shows *perm_restrict* *f* (*S* $-$ *A*) *permutes* (*S* $-$ *A*)

\langle *proof* \rangle

lemma *permutes_decompose*:

assumes *f* *permutes* *S* *finite* *S*

shows $\exists C. (\forall c \in C. \text{cyclic_on } f \ c) \wedge \bigcup C = S \wedge (\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\})$

\langle *proof* \rangle

8.3 Function-power distance between values

definition *funpow_dist* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{nat}$ **where**

funpow_dist *f* *x* *y* $\equiv \text{LEAST } n. (f \rightsquigarrow n) \ x = y$

abbreviation *funpow_dist1* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{nat}$ **where**

funpow_dist1 *f* *x* *y* $\equiv \text{Suc } (\text{funpow_dist } f \ (f \ x) \ y)$

lemma *funpow_dist_0*:

assumes *x* = *y* **shows** *funpow_dist* *f* *x* *y* = 0

\langle *proof* \rangle

lemma *funpow_dist_least*:

assumes *n* < *funpow_dist* *f* *x* *y* **shows** $(f \rightsquigarrow n) \ x \neq y$

\langle *proof* \rangle

lemma *funpow_dist1_least*:

assumes 0 < *n* < *funpow_dist1* *f* *x* *y* **shows** $(f \rightsquigarrow n) \ x \neq y$

\langle *proof* \rangle

lemma *funpow_dist_prop*:
 $y \in \text{orbit } f \ x \implies (f \rightsquigarrow \text{funpow_dist } f \ x \ y) \ x = y$
 $\langle \text{proof} \rangle$

lemma *funpow_dist_0_eq*:
assumes $y \in \text{orbit } f \ x$ **shows** $\text{funpow_dist } f \ x \ y = 0 \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *funpow_dist_step*:
assumes $x \neq y \in \text{orbit } f \ x$ **shows** $\text{funpow_dist } f \ x \ y = \text{Suc } (\text{funpow_dist } f \ (f \ x) \ y)$
 $\langle \text{proof} \rangle$

lemma *funpow_dist1_prop*:
assumes $y \in \text{orbit } f \ x$ **shows** $(f \rightsquigarrow \text{funpow_dist1 } f \ x \ y) \ x = y$
 $\langle \text{proof} \rangle$

lemma *funpow_neq_less_funpow_dist*:
assumes $y \in \text{orbit } f \ x \ m \leq \text{funpow_dist } f \ x \ y \ n \leq \text{funpow_dist } f \ x \ y \ m \neq n$
shows $(f \rightsquigarrow m) \ x \neq (f \rightsquigarrow n) \ x$
 $\langle \text{proof} \rangle$

lemma *funpow_neq_less_funpow_dist1*:
assumes $y \in \text{orbit } f \ x \ m < \text{funpow_dist1 } f \ x \ y \ n < \text{funpow_dist1 } f \ x \ y \ m \neq n$
shows $(f \rightsquigarrow m) \ x \neq (f \rightsquigarrow n) \ x$
 $\langle \text{proof} \rangle$

lemma *inj_on_funpow_dist*:
assumes $y \in \text{orbit } f \ x$ **shows** $\text{inj_on } (\lambda n. (f \rightsquigarrow n) \ x) \ \{0.. \text{funpow_dist } f \ x \ y\}$
 $\langle \text{proof} \rangle$

lemma *inj_on_funpow_dist1*:
assumes $y \in \text{orbit } f \ x$ **shows** $\text{inj_on } (\lambda n. (f \rightsquigarrow n) \ x) \ \{0.. \text{funpow_dist1 } f \ x \ y\}$
 $\langle \text{proof} \rangle$

lemma *orbit_conv_funpow_dist1*:
assumes $x \in \text{orbit } f \ x$
shows $\text{orbit } f \ x = (\lambda n. (f \rightsquigarrow n) \ x) \text{ `` } \{0.. \text{funpow_dist1 } f \ x \ x\} \text{ (is ?L = ?R)}$
 $\langle \text{proof} \rangle$

lemma *funpow_dist1_prop1*:
assumes $(f \rightsquigarrow n) \ x = y \ 0 < n$ **shows** $(f \rightsquigarrow \text{funpow_dist1 } f \ x \ y) \ x = y$
 $\langle \text{proof} \rangle$

lemma *funpow_dist1_dist*:
assumes $\text{funpow_dist1 } f \ x \ y < \text{funpow_dist1 } f \ x \ z$
assumes $\{y, z\} \subseteq \text{orbit } f \ x$

```

  shows funpow_dist1 f x z = funpow_dist1 f x y + funpow_dist1 f y z (is ?L =
?R)
<proof>

lemma funpow_dist1_le_self:
  assumes (f  $\sim$  m) x = x 0 < m y  $\in$  orbit f x
  shows funpow_dist1 f x y  $\leq$  m
<proof>

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

```