

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

September 11, 2023

## Contents

<b>1</b>	<b>Transposition function</b>	<b>1</b>
<b>2</b>	<b>Stirling numbers of first and second kind</b>	<b>5</b>
2.1	Stirling numbers of the second kind . . . . .	5
2.2	Stirling numbers of the first kind . . . . .	6
2.2.1	Efficient code . . . . .	9
<b>3</b>	<b>Permutations, both general and specifically on finite sets.</b>	<b>11</b>
3.1	Auxiliary . . . . .	12
3.2	Basic definition and consequences . . . . .	12
3.3	Group properties . . . . .	17
3.4	Mapping permutations with bijections . . . . .	17
3.5	The number of permutations on a finite set . . . . .	19
3.6	Hence a sort of induction principle composing by swaps . . . . .	22
3.7	Permutations of index set for iterated operations . . . . .	23
3.8	Permutations as transposition sequences . . . . .	23
3.9	Some closure properties of the set of permutations, with lengths . . . . .	23
3.10	Various combinations of transpositions with 2, 1 and 0 common elements . . . . .	25
3.11	The identity map only has even transposition sequences . . . . .	26
3.12	Therefore we have a welldefined notion of parity . . . . .	28
3.13	And it has the expected composition properties . . . . .	29
3.14	A more abstract characterization of permutations . . . . .	29
3.15	Relation to <i>permutes</i> . . . . .	32
3.16	Sign of a permutation as a real number . . . . .	32
3.17	Permuting a list . . . . .	33
3.18	More lemmas about permutations . . . . .	35
3.19	Sum over a set of permutations (could generalize to iteration) . . . . .	44
3.20	Constructing permutations from association lists . . . . .	45

<b>4</b>	<b>Permuted Lists</b>	<b>48</b>
4.1	An existing notion . . . . .	48
4.2	Nontrivial conclusions . . . . .	48
4.3	Trivial conclusions: . . . . .	49
<b>5</b>	<b>Permutations of a Multiset</b>	<b>51</b>
5.1	Permutations of a multiset . . . . .	52
5.2	Cardinality of permutations . . . . .	54
5.3	Permutations of a set . . . . .	57
5.4	Code generation . . . . .	58
<b>6</b>	<b>Cycles</b>	<b>62</b>
6.1	Definitions . . . . .	62
6.2	Basic Properties . . . . .	62
6.3	Conjugation of cycles . . . . .	64
6.4	When Cycles Commute . . . . .	65
6.5	Cycles from Permutations . . . . .	66
6.5.1	Exponentiation of permutations . . . . .	66
6.5.2	Extraction of cycles from permutations . . . . .	67
6.6	Decomposition on Cycles . . . . .	69
6.6.1	Preliminaries . . . . .	69
6.6.2	Decomposition . . . . .	72
<b>7</b>	<b>Permutations as abstract type</b>	<b>74</b>
7.1	Abstract type of permutations . . . . .	74
7.2	Identity, composition and inversion . . . . .	76
7.3	Orbit and order of elements . . . . .	79
7.4	Swaps . . . . .	89
7.5	Permutations specified by cycles . . . . .	90
7.6	Syntax . . . . .	90
<b>8</b>	<b>Permutation orbits</b>	<b>91</b>
8.1	Orbits and cyclic permutations . . . . .	91
8.2	Decomposition of arbitrary permutations . . . . .	97
8.3	Function-power distance between values . . . . .	99
<b>9</b>	<b>Basic combinatorics in Isabelle/HOL (and the Archive of Formal Proofs)</b>	<b>103</b>

## 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*]:  
 $\langle \text{transpose } a \ b \ a = b \rangle$   
**by** (*simp add: transpose\_def*)

**lemma** *transpose\_apply\_second* [*simp*]:  
 $\langle \text{transpose } a \ b \ b = a \rangle$   
**by** (*simp add: transpose\_def*)

**lemma** *transpose\_apply\_other* [*simp*]:  
 $\langle \text{transpose } a \ b \ c = c \rangle$  **if**  $\langle c \neq a \rangle \langle c \neq b \rangle$   
**using that by** (*simp add: transpose\_def*)

**lemma** *transpose\_same* [*simp*]:  
 $\langle \text{transpose } a \ a = \text{id} \rangle$   
**by** (*simp add: fun\_eq\_iff transpose\_def*)

**lemma** *transpose\_eq\_iff*:  
 $\langle \text{transpose } a \ b \ c = d \iff (c \neq a \wedge c \neq b \wedge d = c) \vee (c = a \wedge d = b) \vee (c = b \wedge d = a) \rangle$   
**by** (*auto simp add: transpose\_def*)

**lemma** *transpose\_eq\_imp\_eq*:  
 $\langle c = d \rangle$  **if**  $\langle \text{transpose } a \ b \ c = \text{transpose } a \ b \ d \rangle$   
**using that by** (*auto simp add: transpose\_eq\_iff*)

**lemma** *transpose\_commute* [*ac\_simps*]:  
 $\langle \text{transpose } b \ a = \text{transpose } a \ b \rangle$   
**by** (*auto simp add: fun\_eq\_iff transpose\_eq\_iff*)

**lemma** *transpose\_involution* [*simp*]:  
 $\langle \text{transpose } a \ b \ (\text{transpose } a \ b \ c) = c \rangle$   
**by** (*auto simp add: transpose\_eq\_iff*)

**lemma** *transpose\_comp\_involution* [*simp*]:  
 $\langle \text{transpose } a \ b \circ \text{transpose } a \ b = \text{id} \rangle$   
**by** (*rule ext*) *simp*

**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**  $\langle b \neq c \rangle$   
**using that by** (*simp add: transpose\_def*)

**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**  $\langle b \neq c \rangle$   
**using that by** (*simp add: fun\_eq\_iff transpose\_triple*)

**lemma** *transpose\_image\_eq* [*simp*]:  
 ⟨*transpose a b* ‘ $A = A$ ’ **if** ⟨ $a \in A \longleftrightarrow b \in A$ ⟩  
**using that by** (*auto simp add: transpose\_def [abs\_def]*)

**lemma** *inj\_on\_transpose* [*simp*]:  
 ⟨*inj\_on* (*transpose a b*)  $A$ ⟩  
**by rule** (*drule transpose\_eq\_imp\_eq*)

**lemma** *inj\_transpose*:  
 ⟨*inj* (*transpose a b*)⟩  
**by** (*fact inj\_on\_transpose*)

**lemma** *surj\_transpose*:  
 ⟨*surj* (*transpose a b*)⟩  
**by simp**

**lemma** *bij\_betw\_transpose\_iff* [*simp*]:  
 ⟨*bij\_betw* (*transpose a b*)  $A A$ ’ **if** ⟨ $a \in A \longleftrightarrow b \in A$ ⟩  
**using that by** (*auto simp: bij\_betw\_def*)

**lemma** *bij\_transpose* [*simp*]:  
 ⟨*bij* (*transpose a b*)⟩  
**by** (*rule bij\_betw\_transpose\_iff*) *simp*

**lemma** *bijection\_transpose*:  
 ⟨*bijection* (*transpose a b*)⟩  
**by standard** (*fact bij\_transpose*)

**lemma** *inv\_transpose\_eq* [*simp*]:  
 ⟨*inv* (*transpose a b*) = *transpose a b*⟩  
**by** (*rule inv\_unique\_comp*) *simp\_all*

**lemma** *transpose\_apply\_commute*:  
 ⟨*transpose a b* ( $f c$ ) =  $f$  (*transpose* (*inv f a*) (*inv f b*)  $c$ )⟩  
**if** ⟨*bij f*⟩  
**proof** –  
**from that have** ⟨*surj f*⟩  
**by** (*rule bij\_is\_surj*)  
**with that show** *?thesis*  
**by** (*simp add: transpose\_def bij\_inv\_eq\_iff surj\_f\_inv\_f*)  
**qed**

**lemma** *transpose\_comp\_eq*:  
 ⟨*transpose a b*  $\circ f$  =  $f \circ$  *transpose* (*inv f a*) (*inv f b*)⟩  
**if** ⟨*bij f*⟩  
**using that by** (*simp add: fun\_eq\_iff transpose\_apply\_commute*)

**lemma** *in\_transpose\_image\_iff*:

$\langle x \in \text{transpose } a \ b \ 'S \longleftrightarrow \text{transpose } a \ b \ x \in S \rangle$   
**by** (auto intro!: image\_eqI)

Legacy input alias

**setup**  $\langle \text{Context.theory\_map } (\text{Name\_Space.map\_naming } (\text{Name\_Space.qualified\_path } \text{true } \text{binding } \langle \text{Fun} \rangle)) \rangle$

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

**lemma** *swap\_def*:  
 $\langle \text{Fun.swap } a \ b \ f = f \ (a := f \ b, \ b := f \ a) \rangle$   
**by** (simp add: fun\_eq\_iff)

**setup**  $\langle \text{Context.theory\_map } (\text{Name\_Space.map\_naming } (\text{Name\_Space.parent\_path})) \rangle$

**lemma** *swap\_apply*:  
 $\text{Fun.swap } a \ b \ f \ a = f \ b$   
 $\text{Fun.swap } a \ b \ f \ b = f \ a$   
 $c \neq a \implies c \neq b \implies \text{Fun.swap } a \ b \ f \ c = f \ c$   
**by** *simp\_all*

**lemma** *swap\_self*:  $\text{Fun.swap } a \ a \ f = f$   
**by** *simp*

**lemma** *swap\_commute*:  $\text{Fun.swap } a \ b \ f = \text{Fun.swap } b \ a \ f$   
**by** (simp add: ac\_simps)

**lemma** *swap\_nilpotent*:  $\text{Fun.swap } a \ b \ (\text{Fun.swap } a \ b \ f) = f$   
**by** (simp add: comp\_assoc)

**lemma** *swap\_comp\_involutory*:  $\text{Fun.swap } a \ b \circ \text{Fun.swap } a \ b = \text{id}$   
**by** (simp add: fun\_eq\_iff)

**lemma** *swap\_triple*:  
**assumes**  $a \neq c$  **and**  $b \neq c$   
**shows**  $\text{Fun.swap } a \ b \ (\text{Fun.swap } b \ c \ (\text{Fun.swap } a \ b \ f)) = \text{Fun.swap } a \ c \ f$   
**using** *assms* *transpose\_comp\_triple* [of  $a \ c \ b$ ]  
**by** (simp add: comp\_assoc)

**lemma** *comp\_swap*:  $f \circ \text{Fun.swap } a \ b \ g = \text{Fun.swap } a \ b \ (f \circ g)$   
**by** (simp add: comp\_assoc)

**lemma** *swap\_image\_eq*:  
**assumes**  $a \in A \ b \in A$   
**shows**  $\text{Fun.swap } a \ b \ f \ 'A = f \ 'A$   
**using** *assms* **by** (metis *image\_comp* *transpose\_image\_eq*)

**lemma** *inj\_on\_imp\_inj\_on\_swap*:  $\text{inj\_on } f \ A \implies a \in A \implies b \in A \implies \text{inj\_on}$

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

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

**lemma** *surj\_imp\_surj\_swap*:  $\text{surj } f \implies \text{surj } (\text{Fun.swap } a \ b \ f)$   
**by** (*meson comp\_surj surj\_transpose*)

**lemma** *surj\_swap\_iff*:  $\text{surj } (\text{Fun.swap } a \ b \ f) \longleftrightarrow \text{surj } f$   
**by** (*metis fun.set\_map surj\_transpose*)

**lemma** *bij\_betw\_swap\_iff*:  $x \in A \implies y \in A \implies \text{bij\_betw } (\text{Fun.swap } x \ y \ f) \ A \ B$   
 $\longleftrightarrow \text{bij\_betw } f \ A \ B$   
**by** (*meson bij\_betw\_comp\_iff bij\_betw\_transpose\_iff*)

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

**lemma** *swap\_image*:  
 $\langle \text{Fun.swap } i \ j \ f \ 'A = f \ ' (A - \{i, j\})$   
 $\cup (\text{if } i \in A \ \text{then } \{j\} \ \text{else } \{\}) \cup (\text{if } j \in A \ \text{then } \{i\} \ \text{else } \{\}) \rangle$   
**by** (*auto simp add: Fun.swap\_def*)

**lemma** *inv\_swap\_id*:  $\text{inv } (\text{Fun.swap } a \ b \ \text{id}) = \text{Fun.swap } a \ b \ \text{id}$   
**by** *simp*

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

**lemma** *swap\_id\_eq*:  $\text{Fun.swap } a \ b \ \text{id } x = (\text{if } x = a \ \text{then } b \ \text{else if } x = b \ \text{then } a \ \text{else } x)$   
**by** (*simp add: Fun.swap\_def*)

**lemma** *swap\_unfold*:  
 $\langle \text{Fun.swap } a \ b \ p = p \circ \text{Fun.swap } a \ b \ \text{id} \rangle$   
**by** *simp*

**lemma** *swap\_id\_idempotent*:  $\text{Fun.swap } a \ b \ \text{id} \circ \text{Fun.swap } a \ b \ \text{id} = \text{id}$   
**by** *simp*

**lemma** *bij\_swap\_compose\_bij*:  
 $\langle \text{bij } (\text{Fun.swap } a \ b \ \text{id} \circ p) \rangle$  **if**  $\langle \text{bij } p \rangle$   
**using** *that* **by** (*rule bij\_comp*) *simp*

end

## 2 Stirling numbers of first and second kind

theory *Stirling*  
imports *Main*  
begin

### 2.1 Stirling numbers of the second kind

fun *Stirling* :: nat  $\Rightarrow$  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]:  $Stirling\ (Suc\ n)\ (Suc\ 0) = 1$   
by (induct n) simp\_all

lemma *Stirling\_less* [simp]:  $n < k \implies Stirling\ n\ k = 0$   
by (induct n k rule: *Stirling.induct*) simp\_all

lemma *Stirling\_same* [simp]:  $Stirling\ n\ n = 1$   
by (induct n) simp\_all

lemma *Stirling\_2\_2*:  $Stirling\ (Suc\ (Suc\ n))\ (Suc\ (Suc\ 0)) = 2 \wedge^{Suc\ n} - 1$

proof (induct n)

case 0

then show ?case by simp

next

case (Suc n)

have  $Stirling\ (Suc\ (Suc\ (Suc\ n)))\ (Suc\ (Suc\ 0)) =$

$2 * Stirling\ (Suc\ (Suc\ n))\ (Suc\ (Suc\ 0)) + Stirling\ (Suc\ (Suc\ n))\ (Suc\ 0)$

by simp

also have  $\dots = 2 * (2 \wedge^{Suc\ n} - 1) + 1$

by (simp only: *Suc Stirling\_1*)

also have  $\dots = 2 \wedge^{Suc\ (Suc\ n)} - 1$

proof -

have  $(2::nat) \wedge^{Suc\ n} - 1 > 0$

by (induct n) simp\_all

then have  $2 * ((2::nat) \wedge^{Suc\ n} - 1) > 0$

by simp

then have  $2 \leq 2 * ((2::nat) \wedge^{Suc\ n})$

by simp

with *add\_diff\_assoc2* [of  $2\ 2 * 2 \wedge^{Suc\ n}\ 1$ ]

have  $2 * 2 \wedge^{Suc\ n} - 2 + (1::nat) = 2 * 2 \wedge^{Suc\ n} + 1 - 2$ .

then show ?thesis

by (simp add: *nat\_distrib*)

```

qed
finally show ?case by simp
qed

```

```

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

```

## 2.2 Stirling numbers of the first kind

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

lemma *of\_nat\_stirling\_Suc\_n\_2*:

```

    assumes  $n \geq \text{Suc } 0$ 
    shows (of_nat (stirling (Suc n) 2))::'a::field_char_0) = fact n * ( $\sum k=1..n. (1$ 
/ of_nat k))
    using assms
    proof (induct n)
    case 0
    then show ?case by simp
next
    case (Suc n)
    show ?case
    proof (cases n)
    case 0
    then show ?thesis
    by (auto simp add: numerals(2))
next
    case Suc
    then have geq1:  $\text{Suc } 0 \leq n$ 
    by simp
    have (of_nat (stirling (Suc (Suc n)) 2))::'a) =
of_nat (Suc n * stirling (Suc n) 2 + stirling (Suc n) (Suc 0))
    by (simp only: stirling.simps(4))[of Suc n] numerals(2))
    also have ... = of_nat (Suc n) * (fact n * ( $\sum k = 1..n. 1 / \text{of\_nat } k$ )) + fact
n
    using Suc.hyps[OF geq1]
    by (simp only: stirling_Suc_n_1 of_nat_fact of_nat_add of_nat_mult)
    also have ... = fact (Suc n) * ( $\sum k = 1..n. 1 / \text{of\_nat } k$ ) + fact (Suc n) *
(1 / of_nat (Suc n))
    using of_nat_neq_0 by auto
    also have ... = fact (Suc n) * ( $\sum k = 1..Suc\ n. 1 / \text{of\_nat } k$ )
    by (simp add: distrib_left)
    finally show ?thesis .
qed

```

qed

**lemma** *sum\_stirling*:  $(\sum k \leq n. \text{stirling } n \ k) = \text{fact } n$

**proof** (*induct n*)

case 0

then show ?case by simp

next

case (*Suc n*)

have  $(\sum k \leq \text{Suc } n. \text{stirling } (\text{Suc } n) \ k) = \text{stirling } (\text{Suc } n) \ 0 + (\sum k \leq n. \text{stirling } (\text{Suc } n) \ (\text{Suc } k))$

by (*simp only: sum.atMost\_Suc\_shift*)

also have  $\dots = (\sum k \leq n. \text{stirling } (\text{Suc } n) \ (\text{Suc } k))$

by *simp*

also have  $\dots = (\sum k \leq n. n * \text{stirling } n \ (\text{Suc } k) + \text{stirling } n \ k)$

by *simp*

also have  $\dots = n * (\sum k \leq n. \text{stirling } n \ (\text{Suc } k)) + (\sum k \leq n. \text{stirling } n \ k)$

by (*simp add: sum.distrib sum\_distrib\_left*)

also have  $\dots = n * \text{fact } n + \text{fact } n$

**proof** -

have  $n * (\sum k \leq n. \text{stirling } n \ (\text{Suc } k)) = n * ((\sum k \leq \text{Suc } n. \text{stirling } n \ k) - \text{stirling } n \ 0)$

by (*metis add\_diff\_cancel\_left' sum.atMost\_Suc\_shift*)

also have  $\dots = n * (\sum k \leq n. \text{stirling } n \ k)$

by (*cases n*) *simp\_all*

also have  $\dots = n * \text{fact } n$

using *Suc.hyps* by *simp*

finally have  $n * (\sum k \leq n. \text{stirling } n \ (\text{Suc } k)) = n * \text{fact } n$ .

moreover have  $(\sum k \leq n. \text{stirling } n \ k) = \text{fact } n$

using *Suc.hyps*.

ultimately show ?thesis by *simp*

qed

also have  $\dots = \text{fact } (\text{Suc } n)$  by *simp*

finally show ?case.

qed

**lemma** *stirling\_pochhammer*:

$(\sum k \leq n. \text{of\_nat } (\text{stirling } n \ k) * x^k) = (\text{pochhammer } x \ n :: 'a :: \text{comm\_semiring}_1)$

**proof** (*induct n*)

case 0

then show ?case by *simp*

next

case (*Suc n*)

have  $\text{of\_nat } (n * \text{stirling } n \ 0) = (0 :: 'a)$  by (*cases n*) *simp\_all*

then have  $(\sum k \leq \text{Suc } n. \text{of\_nat } (\text{stirling } (\text{Suc } n) \ k) * x^k) =$

$(\text{of\_nat } (n * \text{stirling } n \ 0) * x^0 +$

$(\sum i \leq n. \text{of\_nat } (n * \text{stirling } n \ (\text{Suc } i)) * (x^{\text{Suc } i}))) +$

$(\sum i \leq n. \text{of\_nat } (\text{stirling } n \ i) * (x^{\text{Suc } i}))$

by (*subst sum.atMost\_Suc\_shift*) (*simp add: sum.distrib ring\_distrib*)

also have  $\dots = \text{pochhammer } x \ (\text{Suc } n)$

```

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

```

A row of the Stirling number triangle

```

definition stirling_row :: nat ⇒ 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
  by (simp add: stirling_row_def del: upt_Suc)

```

```

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

```

```

lemma stirling_row_nonempty [simp]: stirling_row n ≠ []
  using length_stirling_row[of n] by (auto simp del: length_stirling_row)

```

### 2.2.1 Efficient code

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

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

```

definition zip_with_prev :: ('a ⇒ '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..length xs -
1]))

```

```

proof (cases xs)
  case Nil
  then show ?thesis
    by (simp add: zip_with_prev_def)
  next
  case (Cons y ys)
  then have zip_with_prev f x xs = f x (hd xs) # zip_with_prev f y ys
    by (simp add: zip_with_prev_def)
  also have zip_with_prev f y ys = map (λi. f (xs ! i) (xs ! (i + 1))) [0..length
xs - 1]
  unfolding Cons
  by (induct ys arbitrary: y)
    (simp_all add: zip_with_prev_def upt_conv Cons flip: map_Suc upt del:
upt_Suc)

```

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

**primrec** *stirling\_row\_aux*  
**where**

$stirling\_row\_aux\ n\ y\ [] = [1]$   
 $|\ stirling\_row\_aux\ n\ y\ (x\#\!xs) = (y + n * x) \#\ stirling\_row\_aux\ n\ x\ xs$

**lemma** *stirling\_row\_aux\_correct*:

$stirling\_row\_aux\ n\ y\ xs = zip\_with\_prev\ (\lambda a\ b.\ a + n * b)\ y\ xs\ @\ [1]$   
**by** (*induct xs arbitrary: y*) (*simp\_all add: zip\_with\_prev\_def*)

**lemma** *stirling\_row\_code* [*code*]:

$stirling\_row\ 0 = [1]$   
 $stirling\_row\ (Suc\ n) = stirling\_row\_aux\ n\ 0\ (stirling\_row\ n)$

**proof** *goal\_cases*

**case** *1*

**show** *?case* **by** (*simp add: stirling\_row\_def*)

**next**

**case** *2*

**have**  $stirling\_row\ (Suc\ n) =$   
 $0 \# [stirling\_row\ n\ !\ i + stirling\_row\ n\ !\ (i+1) * n.\ i \leftarrow [0..<n]]\ @\ [1]$

**proof** (*rule nth\_equalityI, goal\_cases length nth*)

**case** (*nth i*)

**from** *nth* **have**  $i \leq Suc\ n$

**by** *simp*

**then consider**  $i = 0 \vee i = Suc\ n \mid i > 0\ i \leq n$

**by** *linarith*

**then show** *?case*

**proof** *cases*

**case** *1*

**then show** *?thesis*

**by** (*auto simp: nth\_stirling\_row nth\_append*)

**next**

**case** *2*

**then show** *?thesis*

**by** (*cases i*) (*simp\_all add: nth\_append nth\_stirling\_row*)

**qed**

**next**

**case** *length*

**then show** *?case* **by** *simp*

**qed**

**also have**  $0 \# [stirling\_row\ n\ !\ i + stirling\_row\ n\ !\ (i+1) * n.\ i \leftarrow [0..<n]]\ @\ [1] =$

$zip\_with\_prev\ (\lambda a\ b.\ a + n * b)\ 0\ (stirling\_row\ n)\ @\ [1]$

**by** (*cases n*) (*auto simp add: zip\_with\_prev\_altdef stirling\_row\_def hd\_map simp del: upt\_Suc*)

```

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

```

```

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

```

**end**

### 3 Permutations, both general and specifically on finite sets.

```

theory Permutations
  imports
    HOL-Library.Multiset
    HOL-Library.Disjoint_Sets
    Transposition
begin

```

#### 3.1 Auxiliary

```

abbreviation (input) fixpoints :: ⟨('a ⇒ 'a) ⇒ 'a set⟩
  where ⟨fixpoints f ≡ {x. f x = x}⟩

```

```

lemma inj_on_fixpoints:
  ⟨inj_on f (fixpoints f)⟩
  by (rule inj_onI) simp

```

```

lemma bij_betw_fixpoints:
  ⟨bij_betw f (fixpoints f) (fixpoints f)⟩
  using inj_on_fixpoints by (auto simp add: bij_betw_def)

```

#### 3.2 Basic definition and consequences

```

definition permutes :: ⟨('a ⇒ 'a) ⇒ 'a set ⇒ bool⟩ (infixr ⟨permutes⟩ 41)
  where ⟨p permutes S ↔ (∀x. x ∉ S → p x = x) ∧ (∀y. ∃!x. p x = y)⟩

```

```

lemma bij_imp_permutes:
  ⟨p permutes S⟩ if ⟨bij_betw p S S⟩ and stable: ⟨∧x. x ∉ S ⇒ p x = x⟩

```

**proof** –

```

  note ⟨bij_betw p S S⟩

```

```

  moreover have ⟨bij_betw p (– S) (– S)⟩

```

```

    by (auto simp add: stable_intro!: bij_betw_imageI inj_onI)

```

```

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

```

```

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

```

```

lemma permutes_inj:
  ⟨inj p⟩
  using perm by (auto simp: permutes_def inj_on_def)

```

```

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

```

```

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

```

```

lemma permutes_image_complement:
  ⟨p ' (- S) = - S⟩
  by (auto simp add: permutes_not_in)

```

```

lemma permutes_in_image:
  ⟨p x ∈ S ⟷ x ∈ S⟩
  using permutes_image permutes_inj by (auto dest: inj_image_mem_iff)

```

```

lemma permutates_surj:
  ⟨surj p⟩
proof –
  have ⟨p ‘ (S ∪ - S) = p ‘ S ∪ p ‘ (- S)⟩
    by (rule image_Un)
  then show ?thesis
    by (simp add: permutates_image permutates_image_complement)
qed

lemma permutates_inv_o:
  shows p ∘ inv p = id
    and inv p ∘ p = id
  using permutates_inj permutates_surj
  unfolding inj_iff [symmetric] surj_iff [symmetric] by auto

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

lemma permutates_inv_eq:
  ⟨inv p y = x ⟷ p x = y⟩
  by (auto simp add: permutates_inverses)

lemma permutates_inj_on:
  ⟨inj_on p A⟩
  by (rule inj_on_subset [of _ UNIV]) (auto intro: permutates_inj)

lemma permutates_bij:
  ⟨bij p⟩
  unfolding bij_def by (metis permutates_inj permutates_surj)

lemma permutates_imp_bij:
  ⟨bij_betw p S S⟩
  by (simp add: bij_betw_def permutates_image permutates_inj_on)

lemma permutates_subset:
  ⟨p permutates T⟩ if ⟨S ⊆ T⟩
proof (rule bij_imp_permutates)
  define R where ⟨R = T - S⟩
  with that have ⟨T = R ∪ S⟩ ⟨R ∩ S = {}⟩
    by auto
  then have ⟨p x = x⟩ if ⟨x ∈ R⟩ for x
    using that by (auto intro: permutates_not_in)
  then have ⟨p ‘ R = R⟩
    by simp
  with ⟨T = R ∪ S⟩ show ⟨bij_betw p T T⟩
  by (simp add: bij_betw_def permutates_inj_on image_Un permutates_image)
fix x

```

```

assume  $\langle x \notin T \rangle$ 
with  $\langle T = R \cup S \rangle$  show  $\langle p x = x \rangle$ 
  by (simp add: permutes_not_in)
qed

```

```

lemma permutes_imp_permutes_insert:
   $\langle p \text{ permutes insert } x S \rangle$ 
  by (rule permutes_subset) auto

```

**end**

```

lemma permutes_id [simp]:
   $\langle id \text{ permutes } S \rangle$ 
  by (auto intro: bij_imp_permutes)

```

```

lemma permutes_empty [simp]:
   $\langle p \text{ permutes } \{\} \longleftrightarrow p = id \rangle$ 
proof
  assume  $\langle p \text{ permutes } \{\} \rangle$ 
  then show  $\langle p = id \rangle$ 
    by (auto simp add: fun_eq_iff permutes_not_in)
next
  assume  $\langle p = id \rangle$ 
  then show  $\langle p \text{ permutes } \{\} \rangle$ 
    by simp
qed

```

```

lemma permutes_sing [simp]:
   $\langle p \text{ permutes } \{a\} \longleftrightarrow p = id \rangle$ 
proof
  assume perm:  $\langle p \text{ permutes } \{a\} \rangle$ 
  show  $\langle p = id \rangle$ 
  proof
    fix x
    from perm have  $\langle p \text{ ' } \{a\} = \{a\} \rangle$ 
      by (rule permutes_image)
    with perm show  $\langle p x = id x \rangle$ 
      by (cases  $\langle x = a \rangle$ ) (auto simp add: permutes_not_in)
  qed
next
  assume  $\langle p = id \rangle$ 
  then show  $\langle p \text{ permutes } \{a\} \rangle$ 
    by simp
qed

```

```

lemma permutes_univ:  $p \text{ permutes } UNIV \longleftrightarrow (\forall y. \exists!x. p x = y)$ 
  by (simp add: permutes_def)

```

```

lemma permutes_swap_id:  $a \in S \implies b \in S \implies \text{transpose } a \ b \text{ permutes } S$ 

```



by (rule bij\_imp\_permutes) (auto intro: transpose\_apply\_other)

**lemma** *permutes\_superset*:

$\langle p \text{ permutes } T \rangle$  if  $\langle p \text{ permutes } S \rangle \wedge \langle \forall x. x \in S - T \implies p x = x \rangle$

**proof** –

**define**  $R U$  **where**  $\langle R = T \cap S \rangle$  **and**  $\langle U = S - T \rangle$

**then have**  $\langle T = R \cup (T - S) \rangle \langle S = R \cup U \rangle \langle R \cap U = \{\} \rangle$

by *auto*

**from that**  $\langle U = S - T \rangle$  **have**  $\langle p \text{ ` } U = U \rangle$

by *simp*

**from**  $\langle p \text{ permutes } S \rangle$  **have**  $\langle \text{bij\_betw } p (R \cup U) (R \cup U) \rangle$

by (*simp add: permutes\_imp\_bij*  $\langle S = R \cup U \rangle$ )

**moreover have**  $\langle \text{bij\_betw } p U U \rangle$

**using that**  $\langle U = S - T \rangle$  **by** (*simp add: bij\_betw\_def permutes\_inj\_on*)

**ultimately have**  $\langle \text{bij\_betw } p R R \rangle$

**using**  $\langle R \cap U = \{\} \rangle \langle R \cap U = \{\} \rangle$  **by** (rule *bij\_betw\_partition*)

**then have**  $\langle p \text{ permutes } R \rangle$

**proof** (rule *bij\_imp\_permutes*)

**fix**  $x$

**assume**  $\langle x \notin R \rangle$

**with**  $\langle R = T \cap S \rangle \langle p \text{ permutes } S \rangle$  **show**  $\langle p x = x \rangle$

by (*cases*  $\langle x \in S \rangle$ ) (*auto simp add: permutes\_not\_in that(2)*)

**qed**

**then have**  $\langle p \text{ permutes } R \cup (T - S) \rangle$

by (rule *permutes\_subset*) *simp*

**with**  $\langle T = R \cup (T - S) \rangle$  **show** *?thesis*

by *simp*

**qed**

**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**  $(\lambda x. \text{if } x \in B \text{ then } f (p (\text{inv\_into } A f x)) \text{ else } x) \text{ permutes } B$

**proof** (rule *bij\_imp\_permutes*)

**from** *assms* **have**  $\text{bij\_betw } p A A \text{ bij\_betw } f A B \text{ bij\_betw } (\text{inv\_into } A f) B A$

by (*auto simp add: permutes\_imp\_bij bij\_betw\_inv\_into*)

**then have**  $\text{bij\_betw } (f \circ p \circ \text{inv\_into } A f) B B$

by (*simp add: bij\_betw\_trans*)

**then show**  $\text{bij\_betw } (\lambda x. \text{if } x \in B \text{ then } f (p (\text{inv\_into } A f x)) \text{ else } x) B B$

by (*subst bij\_betw\_cong* [**where**  $g = f \circ p \circ \text{inv\_into } A f$ ]) *auto*

**next**

**fix**  $x$

**assume**  $\langle x \notin B \rangle$

**then show**  $(\text{if } x \in B \text{ then } f (p (\text{inv\_into } A f x)) \text{ else } x) = x$  **by** *auto*

**qed**

**lemma** *permutes\_image\_mset*:

**assumes**  $p$  permutes  $A$   
**shows**  $\text{image\_mset } p (\text{mset\_set } A) = \text{mset\_set } A$   
**using** *assms* **by** (*metis image\_mset\_mset\_set bij\_betw\_imp\_inj\_on permutes\_imp\_bij permutes\_image*)

**lemma** *permutes\_implies\_image\_mset\_eq*:

**assumes**  $p$  permutes  $A \wedge x. x \in A \implies f x = f' (p x)$   
**shows**  $\text{image\_mset } f' (\text{mset\_set } A) = \text{image\_mset } f (\text{mset\_set } A)$   
**proof** –  
**have**  $f x = f' (p x)$  **if**  $x \in \# \text{mset\_set } A$  **for**  $x$   
**using** *assms*(2)[*of x*] **that** **by** (*cases finite A*) *auto*  
**with** *assms* **have**  $\text{image\_mset } f (\text{mset\_set } A) = \text{image\_mset } (f' \circ p) (\text{mset\_set } A)$   
**by** (*auto intro!: image\_mset\_cong*)  
**also** **have**  $\dots = \text{image\_mset } f' (\text{image\_mset } p (\text{mset\_set } A))$   
**by** (*simp add: image\_mset\_compositionality*)  
**also** **have**  $\dots = \text{image\_mset } f' (\text{mset\_set } A)$   
**proof** –  
**from** *assms permutes\_image\_mset* **have**  $\text{image\_mset } p (\text{mset\_set } A) = \text{mset\_set } A$   
**by** *blast*  
**then** **show** *?thesis* **by** *simp*  
**qed**  
**finally** **show** *?thesis ..*  
**qed**

### 3.3 Group properties

**lemma** *permutes\_compose*:  $p$  permutes  $S \implies q$  permutes  $S \implies q \circ p$  permutes  $S$   
**unfolding** *permutes\_def o\_def* **by** *metis*

**lemma** *permutes\_inv*:

**assumes**  $p$  permutes  $S$   
**shows**  $\text{inv } p$  permutes  $S$   
**using** *assms* **unfolding** *permutes\_def permutes\_inv\_eq[OF assms]* **by** *metis*

**lemma** *permutes\_inv\_inv*:

**assumes**  $p$  permutes  $S$   
**shows**  $\text{inv } (\text{inv } p) = p$   
**unfolding** *fun\_eq\_iff permutes\_inv\_eq[OF assms] permutes\_inv\_eq[OF permutes\_inv[OF assms]]*  
**by** *blast*

**lemma** *permutes\_invI*:

**assumes** *perm*:  $p$  permutes  $S$   
**and** *inv*:  $\wedge x. x \in S \implies p' (p x) = x$   
**and** *outside*:  $\wedge x. x \notin S \implies p' x = x$   
**shows**  $\text{inv } p = p'$

**proof**

```

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

```

```

lemma permutes_vimage:  $f\ permutes\ A \implies f^{-1}\ A = A$ 
  by (simp add: bij_vimage_eq_inv_image permutes_bij permutes_image[OF permutes_inv])

```

### 3.4 Mapping permutations with bijections

```

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

```

```

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

        dest: bij_betwE)
qed
qed

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

          dest: bij_betwE)
qed
qed

```

### 3.5 The number of permutations on a finite set

**lemma** *permutes\_insert\_lemma*:

**assumes**  $p$  permutes (insert  $a$   $S$ )  
**shows**  $\text{transpose } a (p a) \circ p$  permutes  $S$   
**apply** (rule permutes\_superset[**where**  $S = \text{insert } a S$ ])  
**apply** (rule permutes\_compose[OF *assms*])  
**apply** (rule permutes\_swap\_id, *simp*)  
**using** permutes\_in\_image[OF *assms*, of  $a$ ]  
**apply** *simp*  
**apply** (auto *simp* add: *Ball\_def*)  
**done**

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

**proof** –

**have**  $p$  permutes insert  $a$   $S \iff$   
 $(\exists b q. p = \text{transpose } a b \circ q \wedge b \in \text{insert } a S \wedge q \text{ permutes } S)$  **for**  $p$

**proof** –

**have**  $\exists b q. p = \text{transpose } a b \circ q \wedge b \in \text{insert } a S \wedge q \text{ permutes } S$   
**if**  $p$ :  $p$  permutes insert  $a$   $S$

**proof** –

**let**  $?b = p a$   
**let**  $?q = \text{transpose } a (p a) \circ p$   
**have**  $*$ :  $p = \text{transpose } a ?b \circ ?q$   
**by** (*simp* add: *fun\_eq\_iff* *o\_assoc*)  
**have**  $**$ :  $?b \in \text{insert } a S$   
**unfolding** permutes\_in\_image[OF  $p$ ] **by** *simp*  
**from** permutes\_insert\_lemma[OF  $p$ ]  $*$   $**$  **show** *thesis*  
**by** *blast*

**qed**

**moreover** **have**  $p$  permutes insert  $a$   $S$

**if**  $bq$ :  $p = \text{transpose } a b \circ q \wedge b \in \text{insert } a S \wedge q \text{ permutes } S$  **for**  $b q$

**proof** –

**from** permutes\_subset[OF  $bq(3)$ , of insert  $a$   $S$ ] **have**  $q$ :  $q$  permutes insert  $a$   $S$   
**by** *auto*

**have**  $a$ :  $a \in \text{insert } a S$

**by** *simp*

**from**  $bq(1)$  permutes\_compose[OF  $q$  permutes\_swap\_id[OF  $a$   $bq(2)$ ]] **show**

*thesis*

**by** *simp*

**qed**

**ultimately** **show** *thesis* **by** *blast*

**qed**

**then** **show** *thesis* **by** *auto*

**qed**

**lemma** *card\_permutations*:

**assumes**  $\text{card } S = n$

**and** *finite*  $S$

```

shows card {p. p permutes S} = fact n
using assms(2,1)
proof (induct arbitrary: n)
  case empty
  then show ?case by simp
next
case (insert x F)
{
  fix n
  assume card_insert: card (insert x F) = n
  let ?xF = {p. p permutes insert x F}
  let ?pF = {p. p permutes F}
  let ?pF' = {(b, p). b ∈ insert x F ∧ p ∈ ?pF}
  let ?g = (λ(b, p). transpose x b ∘ p)
  have xfgpF': ?xF = ?g ' ?pF'
    by (rule permutes_insert[of x F])
  from ⟨x ∉ F⟩ ⟨finite F⟩ card_insert have Fs: card F = n - 1
    by auto
  from ⟨finite F⟩ insert.hyps Fs have pFs: card ?pF = fact (n - 1)
    by auto
  then have finite ?pF
    by (auto intro: card_ge_0_finite)
  with ⟨finite F⟩ card.insert_remove have pF'f: finite ?pF'
    apply (simp only: Collect_case_prod Collect_mem_eq)
    apply (rule finite_cartesian_product)
    apply simp_all
  done

  have ginj: inj_on ?g ?pF'
  proof -
    {
      fix b p c q
      assume bp: (b, p) ∈ ?pF'
      assume cq: (c, q) ∈ ?pF'
      assume eq: ?g (b, p) = ?g (c, q)
      from bp cq have pF: p permutes F and qF: q permutes F
        by auto
      from pF ⟨x ∉ F⟩ eq have b = ?g (b, p) x
        by (auto simp: permutes_def fun_upd_def fun_eq_iff)
      also from qF ⟨x ∉ F⟩ eq have ... = ?g (c, q) x
        by (auto simp: fun_upd_def fun_eq_iff)
      also from qF ⟨x ∉ F⟩ have ... = c
        by (auto simp: permutes_def fun_upd_def fun_eq_iff)
      finally have b = c .
      then have transpose x b = transpose x c
        by simp
      with eq have transpose x b ∘ p = transpose x b ∘ q
        by simp
      then have transpose x b ∘ (transpose x b ∘ p) = transpose x b ∘ (transpose

```

```

x b ∘ q)
  by simp
  then have p = q
    by (simp add: o_assoc)
  with ⟨b = c⟩ have (b, p) = (c, q)
    by simp
  }
  then show ?thesis
    unfolding inj_on_def by blast
qed
from ⟨x ∉ F⟩ ⟨finite F⟩ card_insert have n ≠ 0
  by auto
then have ∃ m. n = Suc m
  by presburger
then obtain m where n: n = Suc m
  by blast
from pFs card_insert have *: card ?xF = fact n
  unfolding xfgpF' card_image[OF ginj]
  using ⟨finite F⟩ ⟨finite ?pF⟩
  by (simp only: Collect_case_prod Collect_mem_eq card_cartesian_product)
(simp add: n)
from finite_imageI[OF pF'f, of ?g] have xFf: finite ?xF
  by (simp add: xfgpF' n)
from * have card ?xF = fact n
  unfolding xFf by blast
}
with insert show ?case by simp
qed

```

```

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

```

### 3.6 Hence a sort of induction principle composing by swaps

```

lemma permutes_induct [consumes 2, case_names id swap]:
  ⟨P p⟩ if ⟨p permutes S⟩ ⟨finite S⟩
  and id: ⟨P id⟩
  and swap: ⟨∧ a b p. a ∈ S ⇒ b ∈ S ⇒ p permutes S ⇒ P p ⇒ P (transpose
a b ∘ p)⟩
using ⟨finite S⟩ ⟨p permutes S⟩ swap proof (induction S arbitrary: p)
  case empty
  with id show ?case
    by (simp only: permutes_empty)
next
  case (insert x S p)
  define q where ⟨q = transpose x (p x) ∘ p⟩
  then have swap_q: ⟨transpose x (p x) ∘ q = p⟩

```

```

    by (simp add: o_assoc)
  from ⟨p permutes insert x S⟩ have ⟨q permutes S⟩
    by (simp add: q_def permutes_insert_lemma)
  then have ⟨q permutes insert x S⟩
    by (simp add: permutes_imp_permutes_insert)
  from ⟨q permutes S⟩ have ⟨P q⟩
    by (auto intro: insert.IH insert.prem(2) permutes_imp_permutes_insert)
  have ⟨x ∈ insert x S⟩
    by simp
  moreover from ⟨p permutes insert x S⟩ have ⟨p x ∈ insert x S⟩
    using permutes_in_image [of p ⟨insert x S⟩ x] by simp
  ultimately have ⟨P (transpose x (p x) ∘ q)⟩
    using ⟨q permutes insert x S⟩ ⟨P q⟩
    by (rule insert.prem(2))
  then show ?case
    by (simp add: swap_q)
qed

```

```

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

```

### 3.7 Permutations of index set for iterated operations

```

lemma (in comm_monoid_set) permute:
  assumes p permutes S
  shows F g S = F (g ∘ p) S
proof –
  from ⟨p permutes S⟩ have inj p
    by (rule permutes_inj)
  then have inj_on p S
    by (auto intro: subset_inj_on)
  then have F g (p ‘ S) = F (g ∘ p) S
    by (rule reindex)

```



```

moreover from ⟨p permutes S⟩ have  $p \cdot S = S$ 
  by (rule permutes_image)
ultimately show ?thesis
  by simp
qed

```

### 3.8 Permutations as transposition sequences

```

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

```

```

declare id[unfolded id_def, simp]

```

```

definition permutation p  $\longleftrightarrow (\exists n. \text{swapidseq } n \text{ } p)$ 

```

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

```

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

```

```

declare permutation_id[unfolded id_def, simp]

```

```

lemma swapidseq_swap: swapidseq (if  $a = b$  then 0 else 1) (transpose a b)
  apply clarsimp
  using comp_Suc[of 0 id a b]
  apply simp
  done

```

```

lemma permutation_swap_id: permutation (transpose a b)

```

```

proof (cases  $a = b$ )
  case True
    then show ?thesis by simp
  next
    case False
      then show ?thesis
        unfolding permutation_def
        using swapidseq_swap[of a b] by blast
qed

```

```

lemma swapidseq_comp_add: swapidseq n p  $\implies \text{swapidseq}$  m q  $\implies \text{swapidseq}$  (n
+ m) (p  $\circ$  q)

```

```

proof (induct n p arbitrary: m q rule: swapidseq.induct)
  case (id m q)
    then show ?case by simp
  next

```

```

case (comp_Suc n p a b m q)
have eq: Suc n + m = Suc (n + m)
  by arith
show ?case
  apply (simp only: eq comp_assoc)
  apply (rule swapidseq.comp_Suc)
  using comp_Suc.hyps(2)[OF comp_Suc.premis] comp_Suc.hyps(3)
  apply blast+
done
qed

lemma permutation_compose: permutation p  $\implies$  permutation q  $\implies$  permutation
(p  $\circ$  q)
  unfolding permutation_def using swapidseq_comp_add[of _ p _ q] by metis

lemma swapidseq_endswap: swapidseq n p  $\implies$  a  $\neq$  b  $\implies$  swapidseq (Suc n) (p  $\circ$ 
transpose a b)
  by (induct n p rule: swapidseq.induct)
  (use swapidseq_swap[of a b] in <auto simp add: comp_assoc intro: swapid-
seq.comp_Suc>)

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

```

**lemma** *swapidseq\_inverse*:  
**assumes** *swapidseq n p*  
**shows** *swapidseq n (inv p)*  
**using** *swapidseq\_inverse\_exists*[*OF assms*] *inv\_unique\_comp*[*of p*] **by** *auto*

**lemma** *permutation\_inverse*: *permutation p*  $\implies$  *permutation (inv p)*  
**using** *permutation\_def swapidseq\_inverse* **by** *blast*

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

**lemma** *swap\_id\_common*:  $a \neq c \implies b \neq c \implies$   
 $\text{transpose } a \ b \circ \text{transpose } a \ c = \text{transpose } b \ c \circ \text{transpose } a \ b$   
**by** (*simp add: fun\_eq\_iff transpose\_def*)

**lemma** *swap\_id\_common'*:  $a \neq b \implies a \neq c \implies$   
 $\text{transpose } a \ c \circ \text{transpose } b \ c = \text{transpose } b \ c \circ \text{transpose } a \ b$   
**by** (*simp add: fun\_eq\_iff transpose\_def*)

**lemma** *swap\_id\_independent*:  $a \neq c \implies a \neq d \implies b \neq c \implies b \neq d \implies$   
 $\text{transpose } a \ b \circ \text{transpose } c \ d = \text{transpose } c \ d \circ \text{transpose } a \ b$   
**by** (*simp add: fun\_eq\_iff transpose\_def*)

### 3.11 The identity map only has even transposition sequences

**lemma** *symmetry\_lemma*:  
**assumes**  $\bigwedge a \ b \ c \ d. P \ a \ b \ c \ d \implies P \ a \ b \ d \ c$   
**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**  $\bigwedge a \ b \ c \ d. a \neq b \longrightarrow c \neq d \longrightarrow P \ a \ b \ c \ d$   
**using** *assms* **by** *metis*

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

**proof** –

**assume** *neg*:  $a \neq b \ c \neq d$   
**have**  $a \neq b \longrightarrow c \neq d \longrightarrow$   
 $(\text{transpose } a \ b \circ \text{transpose } c \ d = \text{id} \vee$   
 $(\exists x \ y \ z. x \neq a \wedge y \neq a \wedge z \neq a \wedge x \neq y \wedge$   
 $\text{transpose } a \ b \circ \text{transpose } c \ d = \text{transpose } x \ y \circ \text{transpose } a \ z))$   
**apply** (*rule symmetry\_lemma*[**where**  $a=a$  **and**  $b=b$  **and**  $c=c$  **and**  $d=d$ ])  
**apply** (*simp\_all only: ac\_simps*)  
**apply** (*metis id\_comp swap\_id\_common swap\_id\_common' swap\_id\_independent transpose\_comp\_involutory*)

```

done
with neq show ?thesis by metis
qed

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

lemma swapidseq_cases: swapidseq n p  $\longleftrightarrow$ 
  n = 0  $\wedge$  p = id  $\vee$  ( $\exists$  a b q m. n = Suc m  $\wedge$  p = transpose a b  $\circ$  q  $\wedge$  swapidseq
m q  $\wedge$  a  $\neq$  b)
apply (rule iffI)
apply (erule swapidseq.cases[of n p])
apply simp
apply (rule disjI2)
apply (rule_tac x= a in exI)
apply (rule_tac x= b in exI)
apply (rule_tac x= pa in exI)
apply (rule_tac x= na in exI)
apply simp
apply auto
apply (rule comp_Suc, simp_all)
done

lemma fixing_swapidseq_decrease:
assumes swapidseq n p
and a  $\neq$  b
and (transpose a b  $\circ$  p) a = a
shows n  $\neq$  0  $\wedge$  swapidseq (n - 1) (transpose a b  $\circ$  p)
using assms
proof (induct n arbitrary: p a b)
case 0
then show ?case
by (auto simp add: fun_upd_def)
next
case (Suc n p a b)
from Suc.prem1 swapidseq_cases[of Suc n p]
obtain c d q m where
  cdqm: Suc n = Suc m p = transpose c d  $\circ$  q swapidseq m q c  $\neq$  d n = m
by auto
consider transpose a b  $\circ$  transpose c d = id
| x y z where x  $\neq$  a y  $\neq$  a z  $\neq$  a x  $\neq$  y
  transpose a b  $\circ$  transpose c d = transpose x y  $\circ$  transpose a z
using swap_general[OF Suc.prem2 cdqm(4)] by metis
then show ?case
proof cases
case 1
then show ?thesis
by (simp only: cdqm o_assoc) (simp add: cdqm)
next

```

```

case prems: 2
then have az:  $a \neq z$ 
  by simp
from prems have *:  $(\text{transpose } x \ y \circ h) \ a = a \iff h \ a = a$  for h
  by (simp add: transpose_def)
from cdqm(2) have  $\text{transpose } a \ b \circ p = \text{transpose } a \ b \circ (\text{transpose } c \ d \circ q)$ 
  by simp
then have  $\text{transpose } a \ b \circ p = \text{transpose } x \ y \circ (\text{transpose } a \ z \circ q)$ 
  by (simp add: o_assoc prems)
then have  $(\text{transpose } a \ b \circ p) \ a = (\text{transpose } x \ y \circ (\text{transpose } a \ z \circ q)) \ a$ 
  by simp
then have  $(\text{transpose } x \ y \circ (\text{transpose } a \ z \circ q)) \ a = a$ 
  unfolding Suc by metis
then have  $(\text{transpose } a \ z \circ q) \ a = a$ 
  by (simp only: *)
from Suc.hyps[OF cdqm(3)[unfolded cdqm(5)[symmetric]] az this]
have **:  $\text{swapidseq } (n - 1) (\text{transpose } a \ z \circ q) \ n \neq 0$ 
  by blast+
from  $\langle n \neq 0 \rangle$  have ***:  $\text{Suc } n - 1 = \text{Suc } (n - 1)$ 
  by auto
show ?thesis
  apply (simp only: cdqm(2) prems o_assoc ***)
  apply (simp only: Suc_not_Zero simp_thms comp_assoc)
  apply (rule comp_Suc)
  using ** prems
  apply blast+
done
qed
qed

lemma swapidseq_identity_even:
  assumes  $\text{swapidseq } n \ (id :: 'a \Rightarrow 'a)$ 
  shows  $even \ n$ 
  using  $\langle \text{swapidseq } n \ id \rangle$ 
proof (induct n rule: nat_less_induct)
  case H: (1 n)
  consider  $n = 0$ 
  |  $a \ b :: 'a$  and  $q \ m$  where  $n = \text{Suc } m \ id = \text{transpose } a \ b \circ q \ \text{swapidseq } m \ q \ a \neq b$ 
  using H(2)[unfolded swapidseq_cases[of n id]] by auto
  then show ?case
proof cases
  case 1
  then show ?thesis by presburger
next
  case h: 2
  from fixing_swapidseq_decrease[OF h(3,4), unfolded h(2)[symmetric]]
  have  $m: m \neq 0 \ \text{swapidseq } (m - 1) \ (id :: 'a \Rightarrow 'a)$ 
  by auto

```

```

from  $h\ m$  have  $mn: m - 1 < n$ 
  by arith
from  $H(1)[rule\_format, OF\ mn\ m(2)]\ h(1)\ m(1)$  show ?thesis
  by presburger
qed
qed

```

### 3.12 Therefore we have a welldefined notion of parity

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

```

lemma swapidseq_even_even:
  assumes  $m: swapidseq\ m\ p$ 
    and  $n: swapidseq\ n\ p$ 
  shows  $even\ m \longleftrightarrow even\ n$ 
proof -
  from swapidseq_inverse_exists[ $OF\ n$ ] obtain  $q$  where  $q: swapidseq\ n\ q\ p \circ q$ 
     $= id\ q \circ p = id$ 
    by blast
  from swapidseq_identity_even[ $OF\ swapidseq\_comp\_add[OF\ m\ q(1),\ unfolded\ q]$ ] show ?thesis
    by arith
qed

```

```

lemma evenperm_unique:
  assumes  $p: swapidseq\ n\ p$ 
    and  $n: even\ n = b$ 
  shows  $evenperm\ p = b$ 
unfolding  $n[symmetric]\ evenperm\_def$ 
apply (rule swapidseq_even_even[where  $p = p$ ])
apply (rule someI[where  $x = n$ ])
using  $p$ 
apply blast+
done

```

### 3.13 And it has the expected composition properties

```

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

```

```

lemma evenperm_identity [simp]:
   $\langle evenperm\ (\lambda x.\ x) \rangle$ 
  using evenperm_id by (simp add: id_def [abs_def])

```

```

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

```

```

lemma evenperm_comp:
  assumes  $permutation\ p\ permutation\ q$ 

```

**shows**  $\text{evenperm } (p \circ q) \longleftrightarrow \text{evenperm } p = \text{evenperm } q$   
**proof** –  
**from** *assms* **obtain**  $n\ m$  **where**  $n: \text{swapidseq } n\ p$  **and**  $m: \text{swapidseq } m\ q$   
**unfolding** *permutation\_def* **by** *blast*  
**have**  $\text{even } (n + m) \longleftrightarrow (\text{even } n \longleftrightarrow \text{even } m)$   
**by** *arith*  
**from**  $\text{evenperm\_unique}[OF\ n\ refl]\ \text{evenperm\_unique}[OF\ m\ refl]$   
**and**  $\text{evenperm\_unique}[OF\ \text{swapidseq\_comp\_add}[OF\ n\ m]\ \text{this}]$  **show** *?thesis*  
**by** *blast*  
**qed**

**lemma** *evenperm\_inv*:  
**assumes** *permutation p*  
**shows**  $\text{evenperm } (\text{inv } p) = \text{evenperm } p$   
**proof** –  
**from** *assms* **obtain**  $n$  **where**  $n: \text{swapidseq } n\ p$   
**unfolding** *permutation\_def* **by** *blast*  
**show** *?thesis*  
**by** (*rule evenperm\_unique[OF swapidseq\_inverse[OF n] evenperm\_unique[OF n refl, symmetric]]*)  
**qed**

### 3.14 A more abstract characterization of permutations

**lemma** *permutation\_bijective*:  
**assumes** *permutation p*  
**shows** *bij p*  
**proof** –  
**from** *assms* **obtain**  $n$  **where**  $n: \text{swapidseq } n\ p$   
**unfolding** *permutation\_def* **by** *blast*  
**from**  $\text{swapidseq\_inverse\_exists}[OF\ n]$  **obtain**  $q$  **where**  $q: \text{swapidseq } n\ q\ p \circ q$   
 $= \text{id } q \circ p = \text{id}$   
**by** *blast*  
**then show** *?thesis*  
**unfolding** *bij\_iff*  
**apply** (*auto simp add: fun\_eq\_iff*)  
**apply** *metis*  
**done**  
**qed**

**lemma** *permutation\_finite\_support*:  
**assumes** *permutation p*  
**shows** *finite {x. p x  $\neq$  x}*  
**proof** –  
**from** *assms* **obtain**  $n$  **where**  $\text{swapidseq } n\ p$   
**unfolding** *permutation\_def* **by** *blast*  
**then show** *?thesis*  
**proof** (*induct n p rule: swapidseq.induct*)  
**case** *id*

```

    then show ?case by simp
  next
  case (comp_Suc n p a b)
  let ?S = insert a (insert b {x. p x ≠ x})
  from comp_Suc.hyps(2) have *: finite ?S
    by simp
  from ⟨a ≠ b⟩ have **: {x. (transpose a b ∘ p) x ≠ x} ⊆ ?S
    by auto
  show ?case
    by (rule finite_subset[OF ** *])
qed

```

```

lemma permutation_lemma:
  assumes finite S
    and bij p
    and ∀x. x ∉ S ⟶ p x = x
  shows permutation p
  using assms
proof (induct S arbitrary: p rule: finite_induct)
  case empty
  then show ?case
    by simp
next
  case (insert a F p)
  let ?r = transpose a (p a) ∘ p
  let ?q = transpose a (p a) ∘ ?r
  have *: ?r a = a
    by simp
  from insert * have **: ∀x. x ∉ F ⟶ ?r x = x
    by (metis bij_pointE comp_apply id_apply insert_iff swap_apply(3))
  have bij ?r
    using insert by (simp add: bij_comp)
  have permutation ?r
    by (rule insert(3)[OF ⟨bij ?r⟩ **])
  then have permutation ?q
    by (simp add: permutation_compose permutation_swap_id)
  then show ?case
    by (simp add: o_assoc)
qed

```

```

lemma permutation: permutation p ⟷ bij p ∧ finite {x. p x ≠ x}
  (is ?lhs ⟷ ?b ∧ ?f)
proof
  assume ?lhs
  with permutation_bijective permutation_finite_support show ?b ∧ ?f
    by auto
next
  assume ?b ∧ ?f

```



**then have**  $?f ?b$  **by** *blast+*  
**from** *permutation\_lemma* [*OF this*] **show**  $?lhs$   
**by** *blast*  
**qed**

**lemma** *permutation\_inverse\_works*:  
**assumes** *permutation p*  
**shows**  $inv\ p \circ p = id$   
**and**  $p \circ inv\ p = id$   
**using** *permutation\_bijective* [*OF assms*] **by** (*auto simp: bij\_def inj\_iff surj\_iff*)

**lemma** *permutation\_inverse\_compose*:  
**assumes** *p: permutation p*  
**and** *q: permutation q*  
**shows**  $inv\ (p \circ q) = inv\ q \circ inv\ p$   
**proof** –  
**note**  $ps = permutation\_inverse\_works$  [*OF p*]  
**note**  $qs = permutation\_inverse\_works$  [*OF q*]  
**have**  $p \circ q \circ (inv\ q \circ inv\ p) = p \circ (q \circ inv\ q) \circ inv\ p$   
**by** (*simp add: o\_assoc*)  
**also have**  $\dots = id$   
**by** (*simp add: ps qs*)  
**finally have**  $*$ :  $p \circ q \circ (inv\ q \circ inv\ p) = id$  .  
**have**  $inv\ q \circ inv\ p \circ (p \circ q) = inv\ q \circ (inv\ p \circ p) \circ q$   
**by** (*simp add: o\_assoc*)  
**also have**  $\dots = id$   
**by** (*simp add: ps qs*)  
**finally have**  $**$ :  $inv\ q \circ inv\ p \circ (p \circ q) = id$  .  
**show**  $?thesis$   
**by** (*rule inv\_unique\_comp* [*OF \* \*\**])  
**qed**

### 3.15 Relation to *permutes*

**lemma** *permutes\_imp\_permutation*:  
 $\langle permutation\ p \rangle$  **if**  $\langle finite\ S \rangle$   $\langle p\ permutes\ S \rangle$   
**proof** –  
**from**  $\langle p\ permutes\ S \rangle$  **have**  $\langle \{x. p\ x \neq x\} \subseteq S \rangle$   
**by** (*auto dest: permutes\_not\_in*)  
**then have**  $\langle finite\ \{x. p\ x \neq x\} \rangle$   
**using**  $\langle finite\ S \rangle$  **by** (*rule finite\_subset*)  
**moreover from**  $\langle p\ permutes\ S \rangle$  **have**  $\langle bij\ p \rangle$   
**by** (*auto dest: permutes\_bij*)  
**ultimately show**  $?thesis$   
**by** (*simp add: permutation*)  
**qed**

**lemma** *permutation\_permutesE*:  
**assumes**  $\langle permutation\ p \rangle$

**obtains**  $S$  **where**  $\langle \text{finite } S \rangle \langle p \text{ permutes } S \rangle$   
**proof** –  
**from** *assms* **have**  $\text{fin}: \langle \text{finite } \{x. p\ x \neq x\} \rangle$   
   **by** (*simp add: permutation*)  
**from** *assms* **have**  $\langle \text{bij } p \rangle$   
   **by** (*simp add: permutation*)  
**also have**  $\langle \text{UNIV} = \{x. p\ x \neq x\} \cup \{x. p\ x = x\} \rangle$   
   **by** *auto*  
**finally have**  $\langle \text{bij\_betw } p\ \{x. p\ x \neq x\}\ \{x. p\ x = x\} \rangle$   
   **by** (*rule bij\_betw\_partition*) (*auto simp add: bij\_betw\_fixpoints*)  
**then have**  $\langle p \text{ permutes } \{x. p\ x \neq x\} \rangle$   
   **by** (*auto intro: bij\_imp\_permutes*)  
**with** *fin* **show** *thesis* ..  
**qed**

**lemma** *permutation\_permutes*:  $\text{permutation } p \longleftrightarrow (\exists S. \text{finite } S \wedge p \text{ permutes } S)$   
   **by** (*auto elim: permutation\_permutesE intro: permutes\_imp\_permutation*)

### 3.16 Sign of a permutation as a real number

**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**  $\langle \text{sign } p = 1 \rangle \mid \langle \text{sign } p = -1 \rangle$   
   **by** (*cases*  $\langle \text{evenperm } p \rangle$ ) (*simp\_all add: sign\_def*)

**lemma** *sign\_nz* [*simp*]:  $\text{sign } p \neq 0$   
   **by** (*cases p rule: sign\_cases*) *simp\_all*

**lemma** *sign\_id* [*simp*]:  $\text{sign } \text{id} = 1$   
   **by** (*simp add: sign\_def*)

**lemma** *sign\_identity* [*simp*]:  
    $\langle \text{sign } (\lambda x. x) = 1 \rangle$   
   **by** (*simp add: sign\_def*)

**lemma** *sign\_inverse*:  $\text{permutation } p \Longrightarrow \text{sign } (\text{inv } p) = \text{sign } p$   
   **by** (*simp add: sign\_def evenperm\_inv*)

**lemma** *sign\_compose*:  $\text{permutation } p \Longrightarrow \text{permutation } q \Longrightarrow \text{sign } (p \circ q) = \text{sign } p * \text{sign } q$   
   **by** (*simp add: sign\_def evenperm\_comp*)

**lemma** *sign\_swap\_id*:  $\text{sign } (\text{transpose } a\ b) = (\text{if } a = b \text{ then } 1 \text{ else } -1)$   
   **by** (*simp add: sign\_def evenperm\_swap*)

**lemma** *sign\_idempotent* [*simp*]:  $\text{sign } p * \text{sign } p = 1$   
   **by** (*simp add: sign\_def*)

**lemma** *sign\_left\_idempotent* [*simp*]:  
 $\langle \text{sign } p * (\text{sign } p * \text{sign } q) = \text{sign } q \rangle$   
**by** (*simp add: sign\_def*)

**term** (*bij, bij\_betw, permutation*)

### 3.17 Permuting a list

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

**definition** *permute\_list* ::  $(\text{nat} \Rightarrow \text{nat}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$   
**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*)  
**using** *permutes\_in\_image*[*OF assms*] **by** (*auto simp: permute\_list\_def*)

**lemma** *permute\_list\_nth*:  
**assumes** *f* *permutes*  $\{..<length\ xs\}$  *i* < *length xs*  
**shows** *permute\_list* *f xs* ! *i* = *xs* ! *f i*  
**using** *permutes\_in\_image*[*OF assms*(1)] *assms*(2)  
**by** (*simp add: permute\_list\_def*)

**lemma** *permute\_list\_Nil* [*simp*]: *permute\_list* *f* [] = []  
**by** (*simp add: permute\_list\_def*)

**lemma** *length\_permute\_list* [*simp*]: *length* (*permute\_list* *f xs*) = *length xs*  
**by** (*simp add: permute\_list\_def*)

**lemma** *permute\_list\_compose*:  
**assumes** *g* *permutes*  $\{..<length\ xs\}$   
**shows** *permute\_list* (*f*  $\circ$  *g*) *xs* = *permute\_list* *g* (*permute\_list* *f xs*)  
**using** *assms*[*THEN permutes\_in\_image*] **by** (*auto simp add: permute\_list\_def*)

**lemma** *permute\_list\_ident* [*simp*]: *permute\_list*  $(\lambda x. x)$  *xs* = *xs*  
**by** (*simp add: permute\_list\_def map\_nth*)

**lemma** *permute\_list\_id* [*simp*]: *permute\_list* *id* *xs* = *xs*  
**by** (*simp add: id\_def*)

**lemma** *mset\_permute\_list* [*simp*]:  
**fixes** *xs* ::  $'a \text{ list}$   
**assumes** *f* *permutes*  $\{..<length\ xs\}$   
**shows** *mset* (*permute\_list* *f xs*) = *mset xs*  
**proof** (*rule multiset\_eqI*)  
**fix** *y* ::  $'a$   
**from** *assms* **have** [*simp*]:  $f\ x < \text{length}\ xs \longleftrightarrow x < \text{length}\ xs$  **for** *x*  
**using** *permutes\_in\_image*[*OF assms*] **by** *auto*

**have**  $\text{count } (\text{mset } (\text{permute\_list } f \text{ } xs)) \text{ } y = \text{card } ((\lambda i. \text{xs } ! f i) - \{y\} \cap \{..<\text{length } xs\})$   
**by** (*simp add: permute\_list\_def count\_image\_mset atLeast0LessThan*)  
**also have**  $(\lambda i. \text{xs } ! f i) - \{y\} \cap \{..<\text{length } xs\} = f - \{i. i < \text{length } xs \wedge y = \text{xs } ! i\}$   
**by** *auto*  
**also from** *assms* **have**  $\text{card } \dots = \text{card } \{i. i < \text{length } xs \wedge y = \text{xs } ! i\}$   
**by** (*intro card\_vimage\_inj*) (*auto simp: permutes\_inj permutes\_surj*)  
**also have**  $\dots = \text{count } (\text{mset } xs) \text{ } y$   
**by** (*simp add: count\_mset length\_filter\_conv\_card*)  
**finally show**  $\text{count } (\text{mset } (\text{permute\_list } f \text{ } xs)) \text{ } y = \text{count } (\text{mset } xs) \text{ } y$   
**by** *simp*  
**qed**

**lemma** *set\_permute\_list* [*simp*]:  
**assumes**  $f \text{ permutes } \{..<\text{length } xs\}$   
**shows**  $\text{set } (\text{permute\_list } f \text{ } xs) = \text{set } xs$   
**by** (*rule mset\_eq\_setD[OF mset\_permute\_list]*) *fact*

**lemma** *distinct\_permute\_list* [*simp*]:  
**assumes**  $f \text{ permutes } \{..<\text{length } xs\}$   
**shows**  $\text{distinct } (\text{permute\_list } f \text{ } xs) = \text{distinct } xs$   
**by** (*simp add: distinct\_count\_atmost\_1 assms*)

**lemma** *permute\_list\_zip*:  
**assumes**  $f \text{ permutes } A \text{ } A = \{..<\text{length } xs\}$   
**assumes** [*simp*]:  $\text{length } xs = \text{length } ys$   
**shows**  $\text{permute\_list } f \text{ } (\text{zip } xs \text{ } ys) = \text{zip } (\text{permute\_list } f \text{ } xs) \text{ } (\text{permute\_list } f \text{ } ys)$   
**proof** –  
**from** *permutes\_in\_image*[*OF assms(1)*] *assms(2)* **have**  $*$ :  $f \text{ } i < \text{length } ys \longleftrightarrow i < \text{length } ys$  **for**  $i$   
**by** *simp*  
**have**  $\text{permute\_list } f \text{ } (\text{zip } xs \text{ } ys) = \text{map } (\lambda i. \text{zip } xs \text{ } ys \text{ } ! f i) \text{ } [0..<\text{length } ys]$   
**by** (*simp\_all add: permute\_list\_def zip\_map\_map*)  
**also have**  $\dots = \text{map } (\lambda(x, y). (\text{xs } ! f x, \text{ys } ! f y)) \text{ } (\text{zip } [0..<\text{length } ys] \text{ } [0..<\text{length } ys])$   
**by** (*intro nth\_equalityI*) (*simp\_all add: \**)  
**also have**  $\dots = \text{zip } (\text{permute\_list } f \text{ } xs) \text{ } (\text{permute\_list } f \text{ } ys)$   
**by** (*simp\_all add: permute\_list\_def zip\_map\_map*)  
**finally show** *?thesis* .  
**qed**

**lemma** *map\_of\_permute*:  
**assumes**  $\sigma \text{ permutes } \text{fst } \text{' } \text{set } xs$   
**shows**  $\text{map\_of } xs \circ \sigma = \text{map\_of } (\text{map } (\lambda(x,y). (\text{inv } \sigma \text{ } x, y)) \text{ } xs)$   
 $(\text{is } \_ = \text{map\_of } (\text{map } ?f \text{ } \_))$   
**proof**  
**from** *assms* **have**  $\text{inj } \sigma \text{ } \text{surj } \sigma$   
**by** (*simp\_all add: permutes\_inj permutes\_surj*)

then show  $(\text{map\_of } xs \circ \sigma) x = \text{map\_of } (\text{map } ?f xs) x$  for  $x$   
 by  $(\text{induct } xs) (\text{auto simp: inv\_f\_f surj\_f\_inv\_f})$   
 qed

**lemma** *list\_all2\_permute\_list\_iff*:  
 $\langle \text{list\_all2 } P (\text{permute\_list } p xs) (\text{permute\_list } p ys) \longleftrightarrow \text{list\_all2 } P xs ys \rangle$   
 if  $\langle p \text{ permutes } \{..<\text{length } xs\}\rangle$   
 using that by  $(\text{auto simp add: list\_all2\_iff simp flip: permute\_list\_zip})$

### 3.18 More lemmas about permutations

**lemma** *permutes\_in\_funpow\_image*:  
 assumes  $f \text{ permutes } S \ x \in S$   
 shows  $(f \text{ ^^ } n) x \in S$   
 using *assms* by  $(\text{induction } n) (\text{auto simp: permutes\_in\_image})$

**lemma** *permutation\_self*:  
 assumes  $\langle \text{permutation } p \rangle$   
 obtains  $n$  where  $\langle n > 0 \rangle \langle (p \text{ ^^ } n) x = x \rangle$   
**proof**  $(\text{cases } \langle p x = x \rangle)$   
 case *True*  
 with that [of 1] show *thesis* by *simp*  
**next**  
 case *False*  
**from**  $\langle \text{permutation } p \rangle$  **have**  $\langle \text{inj } p \rangle$   
 by  $(\text{intro permutation\_bijective bij\_is\_inj})$   
**moreover from**  $\langle p x \neq x \rangle$  **have**  $\langle (p \text{ ^^ } \text{Suc } n) x \neq (p \text{ ^^ } n) x \rangle$  for  $n$   
**proof**  $(\text{induction } n \text{ arbitrary: } x)$   
 case 0 then show *?case* by *simp*  
**next**  
 case  $(\text{Suc } n)$   
**have**  $p (p x) \neq p x$   
**proof**  $(\text{rule notI})$   
 assume  $p (p x) = p x$   
 then show *False* using  $\langle p x \neq x \rangle \langle \text{inj } p \rangle$  by  $(\text{simp add: inj\_eq})$   
**qed**  
**have**  $(p \text{ ^^ } \text{Suc } (\text{Suc } n)) x = (p \text{ ^^ } \text{Suc } n) (p x)$   
 by  $(\text{simp add: funpow\_swap1})$   
**also have**  $\dots \neq (p \text{ ^^ } n) (p x)$   
 by  $(\text{rule Suc}) \text{ fact}$   
**also have**  $(p \text{ ^^ } n) (p x) = (p \text{ ^^ } \text{Suc } n) x$   
 by  $(\text{simp add: funpow\_swap1})$   
**finally show** *?case* by *simp*  
**qed**  
**then have**  $\{y. \exists n. y = (p \text{ ^^ } n) x\} \subseteq \{x. p x \neq x\}$   
 by *auto*  
**then have finite**  $\{y. \exists n. y = (p \text{ ^^ } n) x\}$   
 using *permutation\_finite\_support[OF assms]* by  $(\text{rule finite\_subset})$   
**ultimately obtain**  $n$  where  $\langle n > 0 \rangle \langle (p \text{ ^^ } n) x = x \rangle$

by (rule funpow\_inj\_finite)  
 with that [of n] show thesis by blast  
 qed

The following few lemmas were contributed by Lukas Bulwahn.

**lemma** *count\_image\_mset\_eq\_card\_vimage*:  
 assumes *finite A*  
 shows  $\text{count } (\text{image\_mset } f \ (\text{mset\_set } A)) \ b = \text{card } \{a \in A. f \ a = b\}$   
 using *assms*  
**proof** (*induct A*)  
 case *empty*  
 show ?case by *simp*  
**next**  
 case (*insert x F*)  
 show ?case  
**proof** (*cases f x = b*)  
 case *True*  
 with *insert.hyps*  
 have  $\text{count } (\text{image\_mset } f \ (\text{mset\_set } (\text{insert } x \ F))) \ b = \text{Suc } (\text{card } \{a \in F. f \ a = f \ x\})$   
 by *auto*  
 also from *insert.hyps(1,2)* have  $\dots = \text{card } (\text{insert } x \ \{a \in F. f \ a = f \ x\})$   
 by *simp*  
 also from  $\langle f \ x = b \rangle$  have  $\text{card } (\text{insert } x \ \{a \in F. f \ a = f \ x\}) = \text{card } \{a \in \text{insert } x \ F. f \ a = b\}$   
 by (*auto intro: arg\_cong[where f=card]*)  
 finally show ?thesis  
 using *insert* by *auto*  
**next**  
 case *False*  
 then have  $\{a \in F. f \ a = b\} = \{a \in \text{insert } x \ F. f \ a = b\}$   
 by *auto*  
 with *insert False* show ?thesis  
 by *simp*  
 qed  
 qed

— Prove *image\_mset\_eq\_implies\_permutes ...*

**lemma** *image\_mset\_eq\_implies\_permutes*:  
 fixes  $f :: 'a \Rightarrow 'b$   
 assumes *finite A*  
 and *mset\_eq: image\_mset f (mset\_set A) = image\_mset f' (mset\_set A)*  
 obtains *p* where *p permutes A* and  $\forall x \in A. f \ x = f' \ (p \ x)$   
**proof** –  
 from  $\langle \text{finite } A \rangle$  have [*simp*]:  $\text{finite } \{a \in A. f \ a = (b :: 'b)\}$  for *f b* by *auto*  
 have  $f \ 'A = f' \ 'A$   
**proof** –  
 from  $\langle \text{finite } A \rangle$  have  $f \ 'A = f \ '(\text{set\_mset } (\text{mset\_set } A))$   
 by *simp*

```

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

```

— ... and derive the existing property:

**lemma** *mset\_eq\_permutation*:

**fixes** *xs ys* :: 'a list

**assumes** *mset\_eq*: *mset xs = mset ys*

**obtains** *p* **where** *p* permutes {..*length ys*} *permute\_list p ys = xs*

**proof** —

**from** *mset\_eq* **have** *length\_eq*: *length xs = length ys*

**by** (*rule mset\_eq\_length*)

**have** *mset\_set* {..*length ys*} = *mset* [0..*length ys*]

**by** (*rule mset\_set\_upto\_eq\_mset\_upto*)

**with** *mset\_eq length\_eq* **have** *image\_mset* ( $\lambda i. xs ! i$ ) (*mset\_set* {..*length ys*})

=

*image\_mset* ( $\lambda i. ys ! i$ ) (*mset\_set* {..*length ys*})

**by** (*metis map\_nth mset\_map*)

**from** *image\_mset\_eq\_implies\_permutes*[*OF this*]

**obtain** *p* **where** *p*: *p* permutes {..*length ys*} **and**  $\forall i \in \{..*length ys*\}. xs ! i = ys ! (p i)$

**by** *auto*

**with** *length\_eq* **have** *permute\_list p ys = xs*

**by** (*auto intro!*: *nth\_equalityI simp: permute\_list\_nth*)

**with** *p* **show** *thesis* ..

**qed**

**lemma** *permutes\_natset\_le*:

**fixes** *S* :: 'a::wellorder set

**assumes** *p* permutes *S*

**and**  $\forall i \in S. p i \leq i$

**shows** *p = id*

**proof** —

**have** *p n = n* **for** *n*

**using** *assms*

**proof** (*induct n arbitrary: S rule: less\_induct*)

**case** (*less n*)

**show** ?*case*

**proof** (*cases n ∈ S*)

**case** *False*

**with** *less(2)* **show** ?*thesis*

**unfolding** *permutes\_def* **by** *metis*

**next**

**case** *True*

**with** *less(3)* **have** *p n < n*  $\vee$  *p n = n*

**by** *auto*

**then show** ?*thesis*

**proof**

**assume** *p n < n*

**with** *less* **have** *p (p n) = p n*

**by** *metis*

**with** *permutes\_inj*[*OF less(2)*] **have** *p n = n*

**unfolding** *inj\_def* **by** *blast*



```

    with ⟨p n < n⟩ have False
      by simp
    then show ?thesis ..
  qed
qed
qed
then show ?thesis by (auto simp: fun_eq_iff)
qed

```

```

lemma permutes_natset_ge:
  fixes S :: 'a::wellorder set
  assumes p: p permutes S
    and le:  $\forall i \in S. p\ i \geq i$ 
  shows p = id
proof -
  have  $i \geq \text{inv } p\ i$  if  $i \in S$  for i
  proof -
    from that permutes_in_image[OF permutes_inv[OF p]] have  $\text{inv } p\ i \in S$ 
    by simp
    with le have  $p\ (\text{inv } p\ i) \geq \text{inv } p\ i$ 
    by blast
    with permutes_inverses[OF p] show ?thesis
    by simp
  qed
  then have  $\forall i \in S. \text{inv } p\ i \leq i$ 
  by blast
  from permutes_natset_le[OF permutes_inv[OF p] this] have  $\text{inv } p = \text{inv } id$ 
  by simp
  then show ?thesis
  apply (subst permutes_inv_inv[OF p, symmetric])
  apply (rule inv_unique_comp)
  apply simp_all
  done
qed

```

```

lemma image_inverse_permutations:  $\{\text{inv } p \mid p. p \text{ permutes } S\} = \{p. p \text{ permutes } S\}$ 
  apply (rule set_eqI)
  apply auto
  using permutes_inv_inv permutes_inv
  apply auto
  apply (rule_tac x= $\text{inv } x$  in exI)
  apply auto
  done

```

```

lemma image_compose_permutations_left:
  assumes q permutes S
  shows  $\{q \circ p \mid p. p \text{ permutes } S\} = \{p. p \text{ permutes } S\}$ 
  apply (rule set_eqI)

```

```

apply auto
apply (rule permutes_compose)
using assms
apply auto
apply (rule_tac x = inv q ◦ x in exI)
apply (simp add: o_assoc permutes_inv permutes_compose permutes_inv_o)
done

lemma image_compose_permutations_right:
assumes q permutes S
shows  $\{p \circ q \mid p. p \text{ permutes } S\} = \{p . p \text{ permutes } S\}$ 
apply (rule set_eqI)
apply auto
apply (rule permutes_compose)
using assms
apply auto
apply (rule_tac x = x ◦ inv q in exI)
apply (simp add: o_assoc permutes_inv permutes_compose permutes_inv_o
comp_assoc)
done

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

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

```

**lemma** *setum\_permutations\_compose\_left*:  
**assumes**  $q: q \text{ permutes } S$   
**shows**  $\text{sum } f \{p. p \text{ permutes } S\} = \text{sum } (\lambda p. f(q \circ p)) \{p. p \text{ permutes } S\}$   
**(is ?lhs = ?rhs)**

**proof** –  
**let**  $?S = \{p. p \text{ permutes } S\}$   
**have**  $*$ :  $?rhs = \text{sum } (f \circ ((\circ) q)) ?S$   
**by** (*simp add: o\_def*)  
**have**  $**$ :  $\text{inj\_on } ((\circ) q) ?S$   
**proof** (*auto simp add: inj\_on\_def*)  
**fix**  $p r$   
**assume**  $p \text{ permutes } S$   
**and**  $r: r \text{ permutes } S$   
**and**  $rp: q \circ p = q \circ r$   
**then have**  $\text{inv } q \circ q \circ p = \text{inv } q \circ q \circ r$   
**by** (*simp add: comp\_assoc*)  
**with** *permutes\_inj[OF q, unfolded inj\_iff]* **show**  $p = r$   
**by** *simp*

**qed**  
**have**  $((\circ) q) ' ?S = ?S$   
**using** *image\_compose\_permutations\_left[OF q]* **by** *auto*  
**with**  $* \text{sum.reindex}[OF **, of f]$  **show** *?thesis*  
**by** (*simp only:*)

**qed**

**lemma** *sum\_permutations\_compose\_right*:  
**assumes**  $q: q \text{ permutes } S$   
**shows**  $\text{sum } f \{p. p \text{ permutes } S\} = \text{sum } (\lambda p. f(p \circ q)) \{p. p \text{ permutes } S\}$   
**(is ?lhs = ?rhs)**

**proof** –  
**let**  $?S = \{p. p \text{ permutes } S\}$   
**have**  $*$ :  $?rhs = \text{sum } (f \circ (\lambda p. p \circ q)) ?S$   
**by** (*simp add: o\_def*)  
**have**  $**$ :  $\text{inj\_on } (\lambda p. p \circ q) ?S$   
**proof** (*auto simp add: inj\_on\_def*)  
**fix**  $p r$   
**assume**  $p \text{ permutes } S$   
**and**  $r: r \text{ permutes } S$   
**and**  $rp: p \circ q = r \circ q$   
**then have**  $p \circ (q \circ \text{inv } q) = r \circ (q \circ \text{inv } q)$   
**by** (*simp add: o\_assoc*)  
**with** *permutes\_surj[OF q, unfolded surj\_iff]* **show**  $p = r$   
**by** *simp*

**qed**  
**from** *image\_compose\_permutations\_right[OF q]* **have**  $(\lambda p. p \circ q) ' ?S = ?S$   
**by** *auto*  
**with**  $* \text{sum.reindex}[OF **, of f]$  **show** *?thesis*  
**by** (*simp only:*)

**qed**

```

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

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

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

lemma permutes_nat_less:
  ⟨p i < n⟩
proof -
  have ⟨?thesis ⟷ p i ∈ {0..<n}⟩

```

```

    by simp
  also from p have ⟨p i ∈ {0..<n} ↔ i ∈ {0..<n}⟩
    by (rule permutes_in_image)
  finally show ?thesis
    using i by simp
qed

```

```

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

```

end

```

context comm_monoid_set
begin

```

```

lemma permutes_inv:
  ⟨F (λs. g (p s) s) S = F (λs. g s (inv p s)) S⟩ (is ⟨?l = ?r⟩)
  if ⟨p permutes S⟩
proof -
  let ?g = λ(x, y). g x y
  let ?ps = λs. (p s, s)
  let ?ips = λs. (s, inv p s)
  have inj1: inj_on ?ps S by (rule inj_onI) auto
  have inj2: inj_on ?ips S by (rule inj_onI) auto
  have ?l = F ?g (?ps ` S)
    using reindex [OF inj1, of ?g] by simp
  also have ?ps ` S = {(p s, s) | s. s ∈ S} by auto
  also have ... = {(s, inv p s) | s. s ∈ S}
    unfolding permutes_pair_eq [OF that] by simp
  also have ... = ?ips ` S by auto
  also have F ?g ... = ?r
    using reindex [OF inj2, of ?g] by simp
  finally show ?thesis.
qed

```

end

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

```

lemma sum_over_permutations_insert:
  assumes fS: finite S
  and aS: a ∉ S

```

```

shows  $\sum f \{p. p \text{ permutes } (\text{insert } a \ S)\} =$ 
 $\sum (\lambda b. \sum (\lambda q. f (\text{transpose } a \ b \circ q)) \{p. p \text{ permutes } S\}) (\text{insert } a \ S)$ 
proof –
have *:  $\bigwedge f \ a \ b. (\lambda(b, p). f (\text{transpose } a \ b \circ p)) = f \circ (\lambda(b, p). \text{transpose } a \ b \circ p)$ 
by (simp add: fun_eq_iff)
have **:  $\bigwedge P \ Q. \{(a, b). a \in P \wedge b \in Q\} = P \times Q$ 
by blast
show ?thesis
unfolding * ** sum.cartesian_product permutes_insert
proof (rule sum.reindex)
let ?f =  $(\lambda(b, y). \text{transpose } a \ b \circ y)$ 
let ?P =  $\{p. p \text{ permutes } S\}$ 
{
fix b c p q
assume b:  $b \in \text{insert } a \ S$ 
assume c:  $c \in \text{insert } a \ S$ 
assume p:  $p \text{ permutes } S$ 
assume q:  $q \text{ permutes } S$ 
assume eq:  $\text{transpose } a \ b \circ p = \text{transpose } a \ c \circ q$ 
from p q aS have pa:  $p \ a = a$  and qa:  $q \ a = a$ 
unfolding permutes_def by metis+
from eq have  $(\text{transpose } a \ b \circ p) \ a = (\text{transpose } a \ c \circ q) \ a$ 
by simp
then have bc:  $b = c$ 
by (simp add: permutes_def pa qa o_def fun_upd_def id_def
cong del: if_weak_cong split: if_split_asm)
from eq[unfolded bc] have  $(\lambda p. \text{transpose } a \ c \circ p) (\text{transpose } a \ c \circ p) =$ 
 $(\lambda p. \text{transpose } a \ c \circ p) (\text{transpose } a \ c \circ q)$  by simp
then have p = q
unfolding o_assoc swap_id_idempotent by simp
with bc have  $b = c \wedge p = q$ 
by blast
}
then show inj_on ?f (insert a S × ?P)
unfolding inj_on_def by clarify metis
qed
qed

```

### 3.20 Constructing permutations from association lists

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

```

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

```

**lemma** *list\_permutesI* [*simp*]:

```

assumes  $\text{set } (\text{map fst } xs) \subseteq A \text{ set } (\text{map snd } xs) = \text{set } (\text{map fst } xs) \text{ distinct } (\text{map}$ 

```

```

fst xs)
  shows list_permutes xs A
proof -
  from assms(2,3) have distinct (map snd xs)
    by (intro card_distinct) (simp_all add: distinct_card del: set_map)
  with assms show ?thesis
    by (simp add: list_permutes_def)
qed

definition permutation_of_list :: ('a × 'a) list ⇒ 'a ⇒ 'a
  where permutation_of_list xs x = (case map_of xs x of None ⇒ x | Some y ⇒
y)

lemma permutation_of_list_Cons:
  permutation_of_list ((x, y) # xs) x' = (if x = x' then y else permutation_of_list
xs x')
  by (simp add: permutation_of_list_def)

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

declare inverse_permutation_of_list.simps [simp del]

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

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

lemma inj_on_map_of':
  assumes distinct (map snd xs)

```

**shows**  $\text{inj\_on } (the \circ \text{map\_of } xs) \text{ (set (map fst xs))}$   
**by** (*intro comp\_inj\_on inj\_on\_map\_of assms inj\_on\_the*)  
*(force simp: eq\_commute[of None] map\_of\_eq\_None\_iff)*

**lemma** *image\_map\_of*:  
**assumes** *distinct (map fst xs)*  
**shows**  $\text{map\_of } xs \text{ ' set (map fst xs) = Some ' set (map snd xs)}$   
**using** *assms* **by** (*auto simp: rev\_image\_eqI*)

**lemma** *the\_Some\_image* [*simp*]:  $the \text{ ' Some ' } A = A$   
**by** (*subst image\_image*) *simp*

**lemma** *image\_map\_of'*:  
**assumes** *distinct (map fst xs)*  
**shows**  $(the \circ \text{map\_of } xs) \text{ ' set (map fst xs) = set (map snd xs)}$   
**by** (*simp only: image\_comp [symmetric] image\_map\_of assms the\_Some\_image*)

**lemma** *permutation\_of\_list\_permutes* [*simp*]:  
**assumes** *list\_permutes xs A*  
**shows**  $\text{permutation\_of\_list } xs \text{ permutes } A$   
*(is ?f permutes \_)*

**proof** (*rule permutes\_subset[OF bij\_imp\_permutes]*)  
**from** *assms* **show**  $\text{set (map fst xs)} \subseteq A$   
**by** (*simp add: list\_permutes\_def*)  
**from** *assms* **have**  $\text{inj\_on } (the \circ \text{map\_of } xs) \text{ (set (map fst xs)) (is ?P)}$   
**by** (*intro inj\_on\_map\_of'*) (*simp\_all add: list\_permutes\_def*)  
**also** **have**  $?P \longleftrightarrow \text{inj\_on } ?f \text{ (set (map fst xs))}$   
**by** (*intro inj\_on\_cong*)  
*(auto simp: permutation\_of\_list\_def map\_of\_eq\_None\_iff split: option.splits)*  
**finally** **have**  $\text{bij\_betw } ?f \text{ (set (map fst xs)) ( ?f ' set (map fst xs))}$   
**by** (*rule inj\_on\_imp\_bij\_betw*)  
**also** **from** *assms* **have**  $?f \text{ ' set (map fst xs) = (the \circ \text{map\_of } xs) \text{ ' set (map fst xs)}$   
*(is ?f permutes \_)*  
**by** (*intro image\_cong refl*)  
*(auto simp: permutation\_of\_list\_def map\_of\_eq\_None\_iff split: option.splits)*  
**also** **from** *assms* **have**  $\dots = \text{set (map fst xs)}$   
**by** (*subst image\_map\_of'*) (*simp\_all add: list\_permutes\_def*)  
**finally** **show**  $\text{bij\_betw } ?f \text{ (set (map fst xs)) (set (map fst xs))}$  .

**qed** (*force simp: permutation\_of\_list\_def dest!: map\_of\_SomeD split: option.splits*)<sup>+</sup>

**lemma** *eval\_permutation\_of\_list* [*simp*]:  
 $\text{permutation\_of\_list } [] \ x = x$   
 $x = x' \implies \text{permutation\_of\_list } ((x',y)\#xs) \ x = y$   
 $x \neq x' \implies \text{permutation\_of\_list } ((x',y')\#xs) \ x = \text{permutation\_of\_list } xs \ x$   
**by** (*simp\_all add: permutation\_of\_list\_def*)

**lemma** *eval\_inverse\_permutation\_of\_list* [*simp*]:  
 $\text{inverse\_permutation\_of\_list } [] \ x = x$   
 $x = x' \implies \text{inverse\_permutation\_of\_list } ((y,x')\#xs) \ x = y$



$x \neq x' \implies \text{inverse\_permutation\_of\_list } ((y', x') \# xs) x = \text{inverse\_permutation\_of\_list } xs x$

**by** (*simp\_all add: inverse\_permutation\_of\_list.simps*)

**lemma permutation\_of\_list\_id:**  $x \notin \text{set } (\text{map fst } xs) \implies \text{permutation\_of\_list } xs x = x$

**by** (*induct xs*) (*auto simp: permutation\_of\_list\_Cons*)

**lemma permutation\_of\_list\_unique':**

$\text{distinct } (\text{map fst } xs) \implies (x, y) \in \text{set } xs \implies \text{permutation\_of\_list } xs x = y$

**by** (*induct xs*) (*force simp: permutation\_of\_list\_Cons*)<sup>+</sup>

**lemma permutation\_of\_list\_unique:**

$\text{list\_permutes } xs A \implies (x, y) \in \text{set } xs \implies \text{permutation\_of\_list } xs x = y$

**by** (*intro permutation\_of\_list\_unique'*) (*simp\_all add: list\_permutes\_def*)

**lemma inverse\_permutation\_of\_list\_id:**

$x \notin \text{set } (\text{map snd } xs) \implies \text{inverse\_permutation\_of\_list } xs x = x$

**by** (*induct xs*) *auto*

**lemma inverse\_permutation\_of\_list\_unique':**

$\text{distinct } (\text{map snd } xs) \implies (x, y) \in \text{set } xs \implies \text{inverse\_permutation\_of\_list } xs y = x$

**by** (*induct xs*) (*force simp: inverse\_permutation\_of\_list.simps(2)*)<sup>+</sup>

**lemma inverse\_permutation\_of\_list\_unique:**

$\text{list\_permutes } xs A \implies (x, y) \in \text{set } xs \implies \text{inverse\_permutation\_of\_list } xs y = x$

**by** (*intro inverse\_permutation\_of\_list\_unique'*) (*simp\_all add: list\_permutes\_def*)

**lemma inverse\_permutation\_of\_list\_correct:**

**fixes**  $A :: 'a \text{ set}$

**assumes**  $\text{list\_permutes } xs A$

**shows**  $\text{inverse\_permutation\_of\_list } xs = \text{inv } (\text{permutation\_of\_list } xs)$

**proof** (*rule ext, rule sym, subst permutes\_inv\_eq*)

**from** *assms* **show**  $\text{permutation\_of\_list } xs \text{ permutes } A$

**by** *simp*

**show**  $\text{permutation\_of\_list } xs (\text{inverse\_permutation\_of\_list } xs x) = x$  **for**  $x$

**proof** (*cases*  $x \in \text{set } (\text{map snd } xs)$ )

**case** *True*

**then obtain**  $y$  **where**  $(y, x) \in \text{set } xs$  **by** *auto*

**with** *assms* **show** *?thesis*

**by** (*simp add: inverse\_permutation\_of\_list\_unique permutation\_of\_list\_unique*)

**next**

**case** *False*

**with** *assms* **show** *?thesis*

**by** (*auto simp: list\_permutes\_def inverse\_permutation\_of\_list\_id permutation\_of\_list\_id*)

**qed**

**qed**

end

## 4 Permuted Lists

**theory** *List\_Permutation*  
**imports** *Permutations*  
**begin**

Note that multisets already provide the notion of 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* ::  $\langle 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \rangle$  (**infixr**  $\langle \langle \sim \sim \rangle \rangle$  50)  
**where**  $\langle xs \langle \sim \sim \rangle ys \equiv \text{mset } xs = \text{mset } ys \rangle$

### 4.2 Nontrivial conclusions

**proposition** *perm\_swap*:  
 $\langle xs[i := xs ! j, j := xs ! i] \langle \sim \sim \rangle xs \rangle$   
**if**  $\langle i < \text{length } xs \rangle \langle j < \text{length } xs \rangle$   
**using that by** (*simp add: mset\_swap*)

**proposition** *mset\_le\_perm\_append*:  $\text{mset } xs \subseteq\# \text{mset } ys \longleftrightarrow (\exists zs. xs @ zs \langle \sim \sim \rangle ys)$   
**by** (*auto simp add: mset\_subset\_eq\_exists\_conv ex\_mset dest: sym*)

**proposition** *perm\_set\_eq*:  $xs \langle \sim \sim \rangle ys \implies \text{set } xs = \text{set } ys$   
**by** (*rule mset\_eq\_setD simp*)

**proposition** *perm\_distinct\_iff*:  $xs \langle \sim \sim \rangle ys \implies \text{distinct } xs \longleftrightarrow \text{distinct } ys$   
**by** (*rule mset\_eq\_imp\_distinct\_iff simp*)

**theorem** *eq\_set\_perm\_remdups*:  $\text{set } xs = \text{set } ys \implies \text{remdups } xs \langle \sim \sim \rangle \text{remdups } ys$   
**by** (*simp add: set\_eq\_iff\_mset\_remdups\_eq*)

**proposition** *perm\_remdups\_iff\_eq\_set*:  $\text{remdups } x \langle \sim \sim \rangle \text{remdups } y \longleftrightarrow \text{set } x = \text{set } y$   
**by** (*simp add: set\_eq\_iff\_mset\_remdups\_eq*)

**theorem** *permutation\_Ex\_bij*:  
**assumes**  $xs \langle \sim \sim \rangle ys$   
**shows**  $\exists f. \text{bij\_betw } f \{..<\text{length } xs\} \{..<\text{length } ys\} \wedge (\forall i < \text{length } xs. xs ! i = ys ! (f i))$   
**proof** –  
**from** *assms* **have**  $\langle \text{mset } xs = \text{mset } ys \rangle \langle \text{length } xs = \text{length } ys \rangle$

```

  by (auto simp add: dest: mset_eq_length)
  from ⟨mset xs = mset ys⟩ obtain p where ⟨p permutes {..

```

**proposition** *perm\_finite*:  $finite\ B \implies B \langle \sim \sim \rangle A$   
 using *mset\_eq\_finite* by auto

### 4.3 Trivial conclusions:

**proposition** *perm\_empty\_imp*:  $\langle \sim \sim \rangle ys \implies ys = []$   
 by *simp*

This more general theorem is easier to understand!

**proposition** *perm\_length*:  $xs \langle \sim \sim \rangle ys \implies length\ xs = length\ ys$   
 by (rule *mset\_eq\_length*) *simp*

**proposition** *perm\_sym*:  $xs \langle \sim \sim \rangle ys \implies ys \langle \sim \sim \rangle xs$   
 by *simp*

We can insert the head anywhere in the list.

**proposition** *perm\_append\_Cons*:  $a \# xs @ ys \langle \sim \sim \rangle xs @ a \# ys$   
 by *simp*

**proposition** *perm\_append\_swap*:  $xs @ ys \langle \sim \sim \rangle ys @ xs$   
 by *simp*

**proposition** *perm\_append\_single*:  $a \# xs \langle \sim \sim \rangle xs @ [a]$   
 by *simp*

**proposition** *perm\_rev*:  $rev\ xs \langle \sim \sim \rangle xs$   
 by *simp*

**proposition** *perm\_append1*:  $xs \langle \sim \sim \rangle ys \implies l @ xs \langle \sim \sim \rangle l @ ys$   
 by *simp*

**proposition** *perm\_append2*:  $xs \langle \sim \sim \rangle ys \implies xs @ l \langle \sim \sim \rangle ys @ l$   
 by *simp*

**proposition** *perm\_empty [iff]*:  $\langle \sim \sim \rangle xs \iff xs = []$

by *simp*

**proposition** *perm\_empty2* [iff]:  $xs <^{\sim\sim}> [] \longleftrightarrow xs = []$   
by *simp*

**proposition** *perm\_sing\_imp*:  $ys <^{\sim\sim}> xs \implies xs = [y] \implies ys = [y]$   
by *simp*

**proposition** *perm\_sing\_eq* [iff]:  $ys <^{\sim\sim}> [y] \longleftrightarrow ys = [y]$   
by *simp*

**proposition** *perm\_sing\_eq2* [iff]:  $[y] <^{\sim\sim}> ys \longleftrightarrow ys = [y]$   
by *simp*

**proposition** *perm\_remove*:  $x \in \text{set } ys \implies ys <^{\sim\sim}> x \# \text{remove1 } x \text{ } ys$   
by *simp*

Congruence rule

**proposition** *perm\_remove\_perm*:  $xs <^{\sim\sim}> ys \implies \text{remove1 } z \text{ } xs <^{\sim\sim}> \text{remove1 } z \text{ } ys$   
by *simp*

**proposition** *remove\_hd* [simp]:  $\text{remove1 } z \text{ } (z \# xs) = xs$   
by *simp*

**proposition** *cons\_perm\_imp\_perm*:  $z \# xs <^{\sim\sim}> z \# ys \implies xs <^{\sim\sim}> ys$   
by *simp*

**proposition** *cons\_perm\_eq* [simp]:  $z \# xs <^{\sim\sim}> z \# ys \longleftrightarrow xs <^{\sim\sim}> ys$   
by *simp*

**proposition** *append\_perm\_imp\_perm*:  $zs @ xs <^{\sim\sim}> zs @ ys \implies xs <^{\sim\sim}> ys$   
by *simp*

**proposition** *perm\_append1\_eq* [iff]:  $zs @ xs <^{\sim\sim}> zs @ ys \longleftrightarrow xs <^{\sim\sim}> ys$   
by *simp*

**proposition** *perm\_append2\_eq* [iff]:  $xs @ zs <^{\sim\sim}> ys @ zs \longleftrightarrow xs <^{\sim\sim}> ys$   
by *simp*

end

## 5 Permutations of a Multiset

**theory** *Multiset\_Permutations*

**imports**

*Complex\_Main*

*Permutations*

**begin**

**lemma** *mset\_tl*:  $xs \neq [] \implies mset (tl\ xs) = mset\ xs - \{\#hd\ xs\# \}$   
**by** (*cases xs*) *simp\_all*

**lemma** *mset\_set\_image\_inj*:  
**assumes** *inj\_on f A*  
**shows**  $mset\_set (f \ ' A) = image\_mset\ f (mset\_set\ A)$   
**proof** (*cases finite A*)  
**case True**  
**from this and assms show ?thesis by (induction A) auto**  
**qed** (*insert assms, simp add: finite\_image\_iff*)

**lemma** *multiset\_remove\_induct* [*case\_names empty remove*]:  
**assumes**  $P \ \{\#\} \wedge A. A \neq \{\#\} \implies (\bigwedge x. x \in\# A \implies P (A - \{\#x\# \})) \implies P$   
*A*  
**shows**  $P\ A$   
**proof** (*induction A rule: full\_multiset\_induct*)  
**case (less A)**  
**hence IH: P B if B  $\subset\#$  A for B using that by blast**  
**show ?case**  
**proof** (*cases A = \{\#\}*)  
**case True**  
**thus ?thesis by (simp add: assms)**  
**next**  
**case False**  
**hence P (A - \{\#x\# \}) if x  $\in\#$  A for x**  
**using that by (intro IH) (simp add: mset\_subset\_diff\_self)**  
**from False and this show P A by (rule assms)**  
**qed**  
**qed**

**lemma** *map\_list\_bind*:  $map\ g (List.bind\ xs\ f) = List.bind\ xs (map\ g \circ f)$   
**by** (*simp add: List.bind\_def map\_concat*)

**lemma** *mset\_eq\_mset\_set\_imp\_distinct*:  
*finite A  $\implies mset\_set\ A = mset\ xs \implies distinct\ xs$*   
**proof** (*induction xs arbitrary: A*)  
**case (Cons x xs A)**  
**from Cons.prem(2) have x  $\in\#$  mset\_set A by simp**  
**with Cons.prem(1) have [simp]: x  $\in$  A by simp**  
**from Cons.prem have x  $\notin\#$  mset\_set (A - \{x\}) by simp**  
**also from Cons.prem have mset\_set (A - \{x\}) = mset\_set A - \{\#x\# \}**  
**by (subst mset\_set\_Diff) simp\_all**  
**also have mset\_set A = mset (x#xs) by (simp add: Cons.prem)**  
**also have ... - \{\#x\# \} = mset xs by simp**  
**finally have [simp]: x  $\notin$  set xs by (simp add: in\_multiset\_in\_set)**  
**from Cons.prem show ?case by (auto intro!: Cons.IH[of A - \{x\}] simp:**

*mset\_set\_Diff*)  
**qed** *simp\_all*

## 5.1 Permutations of a multiset

**definition** *permutations\_of\_multiset* :: 'a multiset  $\Rightarrow$  'a list set **where**  
*permutations\_of\_multiset* A = {xs. mset xs = A}

**lemma** *permutations\_of\_multisetI*: mset xs = A  $\implies$  xs  $\in$  *permutations\_of\_multiset* A  
**by** (*simp add: permutations\_of\_multiset\_def*)

**lemma** *permutations\_of\_multisetD*: xs  $\in$  *permutations\_of\_multiset* A  $\implies$  mset xs = A  
**by** (*simp add: permutations\_of\_multiset\_def*)

**lemma** *permutations\_of\_multiset\_Cons\_iff*:  
x # xs  $\in$  *permutations\_of\_multiset* A  $\iff$  x  $\in$  # A  $\wedge$  xs  $\in$  *permutations\_of\_multiset* (A - {#x#})  
**by** (*auto simp: permutations\_of\_multiset\_def*)

**lemma** *permutations\_of\_multiset\_empty* [*simp*]: *permutations\_of\_multiset* {#} = {}  
**unfolding** *permutations\_of\_multiset\_def* **by** *simp*

**lemma** *permutations\_of\_multiset\_nonempty*:  
**assumes** *nonempty*: A  $\neq$  {#}  
**shows** *permutations\_of\_multiset* A =  
 $(\bigcup x \in \text{set\_mset } A. ((\#) x) \text{ 'permutations\_of\_multiset } (A - \{ \#x\# \}))$   
(*is \_ = ?rhs*)

**proof** *safe*

**fix** xs **assume** xs  $\in$  *permutations\_of\_multiset* A  
**hence** mset xs: mset xs = A **by** (*simp add: permutations\_of\_multiset\_def*)  
**hence** xs  $\neq$  [] **by** (*auto simp: nonempty*)  
**then obtain** x xs' **where** xs: xs = x # xs' **by** (*cases xs*) *simp\_all*  
**with** mset xs **have** x  $\in$  set\_mset A xs'  $\in$  *permutations\_of\_multiset* (A - {#x#})  
**by** (*auto simp: permutations\_of\_multiset\_def*)  
**with** xs **show** xs  $\in$  ?rhs **by** *auto*  
**qed** (*auto simp: permutations\_of\_multiset\_def*)

**lemma** *permutations\_of\_multiset\_singleton* [*simp*]: *permutations\_of\_multiset* {#x#} = {[x]}  
**by** (*simp add: permutations\_of\_multiset\_nonempty*)

**lemma** *permutations\_of\_multiset\_doubleton*:  
*permutations\_of\_multiset* {#x,y#} = {[x,y], [y,x]}  
**by** (*simp add: permutations\_of\_multiset\_nonempty insert\_commute*)

```

lemma rev_permutations_of_multiset [simp]:
  rev ' permutations_of_multiset A = permutations_of_multiset A
proof
  have rev ' rev ' permutations_of_multiset A ⊆ rev ' permutations_of_multiset A
  unfolding permutations_of_multiset_def by auto
  also have rev ' rev ' permutations_of_multiset A = permutations_of_multiset A
  by (simp add: image_image)
  finally show permutations_of_multiset A ⊆ rev ' permutations_of_multiset A
  .
next
  show rev ' permutations_of_multiset A ⊆ permutations_of_multiset A
  unfolding permutations_of_multiset_def by auto
qed

lemma length_finite_permutations_of_multiset:
  xs ∈ permutations_of_multiset A ⇒ length xs = size A
  by (auto simp: permutations_of_multiset_def)

lemma permutations_of_multiset_lists: permutations_of_multiset A ⊆ lists (set_mset A)
  by (auto simp: permutations_of_multiset_def)

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

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

lemma permutations_of_multiset_image:
  permutations_of_multiset (image_mset f A) = map f ' permutations_of_multiset A
proof safe
  fix xs assume A: xs ∈ permutations_of_multiset (image_mset f A)
  from ex_mset[of A] obtain ys where mset ys = A ..
  with A have mset xs = mset (map f ys)
  by (simp add: permutations_of_multiset_def)

```

**then obtain**  $\sigma$  **where**  $\sigma: \sigma$  *permutes*  $\{..<length (map f ys)\}$  *permute\_list*  $\sigma$   
 $(map f ys) = xs$   
**by** (*rule mset\_eq\_permutation*)  
**with**  $ys$  **have**  $xs = map f (permute\_list \sigma ys)$   
**by** (*simp add: permute\_list\_map*)  
**moreover from**  $\sigma$   $ys$  **have** *permute\_list*  $\sigma$   $ys \in permutations\_of\_multiset A$   
**by** (*simp add: permutations\_of\_multiset\_def*)  
**ultimately show**  $xs \in map f ' permutations\_of\_multiset A$  **by blast**  
**qed** (*auto simp: permutations\_of\_multiset\_def*)

## 5.2 Cardinality of permutations

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

**context**

**begin**

**private lemma** *multiset\_prod\_fact\_insert:*

$$\left(\prod_{y \in \text{set\_mset } (A + \{x\})} \text{fact } (\text{count } (A + \{x\}) y)\right) = (\text{count } A x + 1) * \left(\prod_{y \in \text{set\_mset } A} \text{fact } (\text{count } A y)\right)$$

**proof** –

$$\text{have } \left(\prod_{y \in \text{set\_mset } (A + \{x\})} \text{fact } (\text{count } (A + \{x\}) y)\right) = \left(\prod_{y \in \text{set\_mset } (A + \{x\})} (\text{if } y = x \text{ then } \text{count } A x + 1 \text{ else } 1) * \text{fact } (\text{count } A y)\right)$$

**by** (*intro prod.cong*) *simp\_all*

$$\text{also have } \dots = (\text{count } A x + 1) * \left(\prod_{y \in \text{set\_mset } (A + \{x\})} \text{fact } (\text{count } A y)\right)$$

**by** (*simp add: prod.distrib*)

$$\text{also have } \left(\prod_{y \in \text{set\_mset } (A + \{x\})} \text{fact } (\text{count } A y)\right) = \left(\prod_{y \in \text{set\_mset } A} \text{fact } (\text{count } A y)\right)$$

**by** (*intro prod.mono\_neutral\_right*) (*auto simp: not\_in\_iff*)

**finally show** *?thesis* .

**qed**

**private lemma** *multiset\_prod\_fact\_remove:*

$$x \in \# A \implies \left(\prod_{y \in \text{set\_mset } A} \text{fact } (\text{count } A y)\right) = \text{count } A x * \left(\prod_{y \in \text{set\_mset } (A - \{x\})} \text{fact } (\text{count } (A - \{x\}) y)\right)$$

**using** *multiset\_prod\_fact\_insert*[of  $A - \{x\}$   $x$ ] **by simp**

**lemma** *card\_permutations\_of\_multiset\_aux:*

$$\text{card } (\text{permutations\_of\_multiset } A) * \left(\prod_{x \in \text{set\_mset } A} \text{fact } (\text{count } A x)\right) = \text{fact } (\text{size } A)$$

**proof** (*induction A rule: multiset\_remove\_induct*)

**case** (*remove A*)

**have**  $\text{card } (\text{permutations\_of\_multiset } A) =$

$$\text{card } \left(\bigcup_{x \in \text{set\_mset } A} \{x\} ' \text{permutations\_of\_multiset } (A - \{x\})\right)$$

**by** (*simp add: permutations\_of\_multiset\_nonempty remove.hyps*)

$$\text{also have } \dots = \left(\sum_{x \in \text{set\_mset } A} \text{card } (\text{permutations\_of\_multiset } (A - \{x\}))\right)$$



**by** (*subst card\_UN\_disjoint*) (*auto simp: card\_image*)  
**also have** ... \* ( $\prod_{x \in \text{set\_mset } A} \text{fact } (\text{count } A \ x)$ ) =  
 $(\sum_{x \in \text{set\_mset } A} \text{card } (\text{permutations\_of\_multiset } (A - \{\#x\#})) * (\prod_{y \in \text{set\_mset } A} \text{fact } (\text{count } A \ y)))$   
**by** (*subst sum\_distrib\_right*) *simp\_all*  
**also have** ... =  $(\sum_{x \in \text{set\_mset } A} \text{count } A \ x * \text{fact } (\text{size } A - 1))$   
**proof** (*intro sum.cong refl*)  
**fix** *x* **assume** *x*:  $x \in \# \ A$   
**have**  $\text{card } (\text{permutations\_of\_multiset } (A - \{\#x\#})) * (\prod_{y \in \text{set\_mset } A} \text{fact } (\text{count } A \ y)) =$   
 $\text{count } A \ x * (\text{card } (\text{permutations\_of\_multiset } (A - \{\#x\#})) * (\prod_{y \in \text{set\_mset } (A - \{\#x\#})} \text{fact } (\text{count } (A - \{\#x\#}) \ y)))$  (*is ?lhs*)  
= \_)  
**by** (*subst multiset\_prod\_fact\_remove[OF x]*) *simp\_all*  
**also note** *remove.IH[OF x]*  
**also from** *x* **have**  $\text{size } (A - \{\#x\#}) = \text{size } A - 1$  **by** (*simp add: size\_Diff\_subset*)  
**finally show** *?lhs* =  $\text{count } A \ x * \text{fact } (\text{size } A - 1)$  .  
**qed**  
**also have**  $(\sum_{x \in \text{set\_mset } A} \text{count } A \ x * \text{fact } (\text{size } A - 1)) = \text{size } A * \text{fact } (\text{size } A - 1)$   
**by** (*simp add: sum\_distrib\_right size\_multiset\_overloaded\_eq*)  
**also from** *remove.hyps* **have** ... =  $\text{fact } (\text{size } A)$   
**by** (*cases size A*) *auto*  
**finally show** *?case* .  
**qed simp\_all**

**theorem card\_permutations\_of\_multiset:**  
 $\text{card } (\text{permutations\_of\_multiset } A) = \text{fact } (\text{size } A) \text{ div } (\prod_{x \in \text{set\_mset } A} \text{fact } (\text{count } A \ x))$   
 $(\prod_{x \in \text{set\_mset } A} \text{fact } (\text{count } A \ x) :: \text{nat}) \text{ dvd } \text{fact } (\text{size } A)$   
**by** (*simp\_all flip: card\_permutations\_of\_multiset\_aux[of A]*)

**lemma card\_permutations\_of\_multiset\_insert\_aux:**  
 $\text{card } (\text{permutations\_of\_multiset } (A + \{\#x\#})) * (\text{count } A \ x + 1) = (\text{size } A + 1) * \text{card } (\text{permutations\_of\_multiset } A)$   
**proof** -  
**note** *card\_permutations\_of\_multiset\_aux[of A + {\#x\#}]*  
**also have**  $\text{fact } (\text{size } (A + \{\#x\#})) = (\text{size } A + 1) * \text{fact } (\text{size } A)$  **by** *simp*  
**also note** *multiset\_prod\_fact\_insert[of A x]*  
**also note** *card\_permutations\_of\_multiset\_aux[of A, symmetric]*  
**finally have**  $\text{card } (\text{permutations\_of\_multiset } (A + \{\#x\#})) * (\text{count } A \ x + 1) =$   
\*  
 $(\prod_{y \in \text{set\_mset } A} \text{fact } (\text{count } A \ y)) =$   
 $(\text{size } A + 1) * \text{card } (\text{permutations\_of\_multiset } A) *$   
 $(\prod_{x \in \text{set\_mset } A} \text{fact } (\text{count } A \ x))$  **by** (*simp only: mult\_ac*)  
**thus** *?thesis* **by** (*subst (asm) mult\_right\_cancel*) *simp\_all*  
**qed**

**lemma card\_permutations\_of\_multiset\_remove\_aux:**

```

assumes  $x \in \# A$ 
shows  $\text{card} (\text{permutations\_of\_multiset } A) * \text{count } A x =$ 
 $\text{size } A * \text{card} (\text{permutations\_of\_multiset } (A - \{\#x\}))$ 
proof -
from assms have  $A - \{\#x\} + \{\#x\} = A$  by simp
from assms have  $\text{size } A = \text{size } (A - \{\#x\}) + 1$ 
by (subst  $A$  [symmetric], subst size_union) simp
show ?thesis
using card_permutations_of_multiset_insert_aux[of  $A - \{\#x\}$   $x$ , unfolded
 $A$ ] assms
by (simp add:  $B$ )
qed

```

```

lemma real_card_permutations_of_multiset_remove:
assumes  $x \in \# A$ 
shows  $\text{real} (\text{card} (\text{permutations\_of\_multiset } (A - \{\#x\}))) =$ 
 $\text{real} (\text{card} (\text{permutations\_of\_multiset } A) * \text{count } A x) / \text{real} (\text{size } A)$ 
using assms by (subst card_permutations_of_multiset_remove_aux[OF assms])
auto

```

```

lemma real_card_permutations_of_multiset_remove':
assumes  $x \in \# A$ 
shows  $\text{real} (\text{card} (\text{permutations\_of\_multiset } A)) =$ 
 $\text{real} (\text{size } A * \text{card} (\text{permutations\_of\_multiset } (A - \{\#x\}))) / \text{real}$ 
 $(\text{count } A x)$ 
using assms by (subst card_permutations_of_multiset_remove_aux[OF assms,
symmetric]) simp

```

**end**

### 5.3 Permutations of a set

```

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

```

```

lemma permutations_of_set_altdef:
 $\text{finite } A \Longrightarrow \text{permutations\_of\_set } A = \text{permutations\_of\_multiset } (\text{mset\_set } A)$ 
by (auto simp add: permutations_of_set_def permutations_of_multiset_def mset_set_set
in_multiset_in_set [symmetric] mset_eq_mset_set_imp_distinct)

```

```

lemma permutations_of_setI [intro]:
assumes  $\text{set } xs = A$  distinct  $xs$ 
shows  $xs \in \text{permutations\_of\_set } A$ 
using assms unfolding permutations_of_set_def by simp

```

```

lemma permutations_of_setD:
assumes  $xs \in \text{permutations\_of\_set } A$ 
shows  $\text{set } xs = A$  distinct  $xs$ 

```

**using** *assms* **unfolding** *permutations\_of\_set\_def* **by** *simp\_all*

**lemma** *permutations\_of\_set\_lists*: *permutations\_of\_set*  $A \subseteq$  *lists*  $A$   
**unfolding** *permutations\_of\_set\_def* **by** *auto*

**lemma** *permutations\_of\_set\_empty* [*simp*]: *permutations\_of\_set*  $\{\}$  =  $\{\{\}\}$   
**by** (*auto simp: permutations\_of\_set\_def*)

**lemma** *UN\_set\_permutations\_of\_set* [*simp*]:  
 $\text{finite } A \implies (\bigcup xs \in \text{permutations\_of\_set } A. \text{set } xs) = A$   
**using** *finite\_distinct\_list* **by** (*auto simp: permutations\_of\_set\_def*)

**lemma** *permutations\_of\_set\_infinite*:  
 $\neg \text{finite } A \implies \text{permutations\_of\_set } A = \{\}$   
**by** (*auto simp: permutations\_of\_set\_def*)

**lemma** *permutations\_of\_set\_nonempty*:  
 $A \neq \{\} \implies \text{permutations\_of\_set } A =$   
 $(\bigcup x \in A. (\lambda xs. x \# xs) \text{ 'permutations\_of\_set } (A - \{x\}))$   
**by** (*cases finite A*)  
(*simp\_all add: permutations\_of\_multiset\_nonempty mset\_set\_empty\_iff mset\_set\_Diff*)

*permutations\_of\_set\_altdef permutations\_of\_set\_infinite*

**lemma** *permutations\_of\_set\_singleton* [*simp*]: *permutations\_of\_set*  $\{x\} = \{\{x\}\}$   
**by** (*subst permutations\_of\_set\_nonempty*) *auto*

**lemma** *permutations\_of\_set\_doubleton*:  
 $x \neq y \implies \text{permutations\_of\_set } \{x,y\} = \{\{x,y\}, \{y,x\}\}$   
**by** (*subst permutations\_of\_set\_nonempty*)  
(*simp\_all add: insert\_Diff\_if insert\_commute*)

**lemma** *rev\_permutations\_of\_set* [*simp*]:  
 $\text{rev 'permutations\_of\_set } A = \text{permutations\_of\_set } A$   
**by** (*cases finite A*) (*simp\_all add: permutations\_of\_set\_altdef permutations\_of\_set\_infinite*)

**lemma** *length\_finite\_permutations\_of\_set*:  
 $xs \in \text{permutations\_of\_set } A \implies \text{length } xs = \text{card } A$   
**by** (*auto simp: permutations\_of\_set\_def distinct\_card*)

**lemma** *finite\_permutations\_of\_set* [*simp*]: *finite* (*permutations\_of\_set*  $A$ )  
**by** (*cases finite A*) (*simp\_all add: permutations\_of\_set\_infinite permutations\_of\_set\_altdef*)

**lemma** *permutations\_of\_set\_empty\_iff* [*simp*]:  
 $\text{permutations\_of\_set } A = \{\} \iff \neg \text{finite } A$   
**unfolding** *permutations\_of\_set\_def* **using** *finite\_distinct\_list* [*of A*] **by** *auto*

**lemma** *card\_permutations\_of\_set* [*simp*]:  
 $\text{finite } A \implies \text{card } (\text{permutations\_of\_set } A) = \text{fact } (\text{card } A)$

**by** (*simp* add: *permutations\_of\_set\_altdef* *card\_permutations\_of\_multiset* *del*:  
*One\_nat\_def*)

**lemma** *permutations\_of\_set\_image\_inj*:  
**assumes** *inj*: *inj\_on* *f* *A*  
**shows** *permutations\_of\_set* (*f* ‘ *A*) = *map* *f* ‘ *permutations\_of\_set* *A*  
**by** (*cases* *finite* *A*)  
(*simp\_all* add: *permutations\_of\_set\_infinite* *permutations\_of\_set\_altdef*  
*permutations\_of\_multiset\_image* *mset\_set\_image\_inj* *inj*  
*finite\_image\_iff*)

**lemma** *permutations\_of\_set\_image\_permutes*:  
 $\sigma$  *permutes* *A*  $\implies$  *map*  $\sigma$  ‘ *permutations\_of\_set* *A* = *permutations\_of\_set* *A*  
**by** (*subst* *permutations\_of\_set\_image\_inj* [*symmetric*])  
(*simp\_all* add: *permutes\_inj\_on* *permutes\_image*)

## 5.4 Code generation

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

**declare** *length\_remove1* [*termination\_simp*]

**fun** *permutations\_of\_list\_impl* **where**  
*permutations\_of\_list\_impl* *xs* = (*if* *xs* = [] *then* [] *else*  
*List.bind* (*remdups* *xs*) ( $\lambda x.$  *map* ((#) *x*) (*permutations\_of\_list\_impl* (*remove1*  
*x* *xs*))))

**fun** *permutations\_of\_list\_impl\_aux* **where**  
*permutations\_of\_list\_impl\_aux* *acc* *xs* = (*if* *xs* = [] *then* [*acc*] *else*  
*List.bind* (*remdups* *xs*) ( $\lambda x.$  *permutations\_of\_list\_impl\_aux* (*x*#*acc*) (*remove1*  
*x* *xs*))))

**declare** *permutations\_of\_list\_impl\_aux.simps* [*simp* *del*]

**declare** *permutations\_of\_list\_impl.simps* [*simp* *del*]

**lemma** *permutations\_of\_list\_impl\_Nil* [*simp*]:  
*permutations\_of\_list\_impl* [] = []  
**by** (*simp* add: *permutations\_of\_list\_impl.simps*)

**lemma** *permutations\_of\_list\_impl\_nonempty*:  
 $xs \neq [] \implies$  *permutations\_of\_list\_impl* *xs* =  
*List.bind* (*remdups* *xs*) ( $\lambda x.$  *map* ((#) *x*) (*permutations\_of\_list\_impl* (*remove1*  
*x* *xs*)))  
**by** (*subst* *permutations\_of\_list\_impl.simps*) *simp\_all*

**lemma** *set\_permutations\_of\_list\_impl*:  
*set* (*permutations\_of\_list\_impl* *xs*) = *permutations\_of\_multiset* (*mset* *xs*)  
**by** (*induction* *xs* *rule*: *permutations\_of\_list\_impl.induct*)  
(*subst* *permutations\_of\_list\_impl.simps*,  
*simp\_all* add: *permutations\_of\_multiset\_nonempty* *set\_list\_bind*)

**lemma** *distinct\_permutations\_of\_list\_impl*:  
*distinct (permutations\_of\_list\_impl xs)*  
**by** (*induction xs rule: permutations\_of\_list\_impl.induct,*  
*subst permutations\_of\_list\_impl.simps*)  
*(auto intro!: distinct\_list\_bind simp: distinct\_map o\_def disjoint\_family\_on\_def)*

**lemma** *permutations\_of\_list\_impl\_aux\_correct'*:  
*permutations\_of\_list\_impl\_aux acc xs =*  
*map (λxs. rev xs @ acc) (permutations\_of\_list\_impl xs)*  
**by** (*induction acc xs rule: permutations\_of\_list\_impl\_aux.induct,*  
*subst permutations\_of\_list\_impl\_aux.simps, subst permutations\_of\_list\_impl.simps*)  
*(auto simp: map\_list\_bind intro!: list\_bind\_cong)*

**lemma** *permutations\_of\_list\_impl\_aux\_correct*:  
*permutations\_of\_list\_impl\_aux [] xs = map rev (permutations\_of\_list\_impl xs)*  
**by** (*simp add: permutations\_of\_list\_impl\_aux\_correct'*)

**lemma** *distinct\_permutations\_of\_list\_impl\_aux*:  
*distinct (permutations\_of\_list\_impl\_aux acc xs)*  
**by** (*simp add: permutations\_of\_list\_impl\_aux\_correct' distinct\_map*  
*distinct\_permutations\_of\_list\_impl\_inj\_on\_def*)

**lemma** *set\_permutations\_of\_list\_impl\_aux*:  
*set (permutations\_of\_list\_impl\_aux [] xs) = permutations\_of\_multiset (mset xs)*  
**by** (*simp add: permutations\_of\_list\_impl\_aux\_correct set\_permutations\_of\_list\_impl*)

**declare** *set\_permutations\_of\_list\_impl\_aux [symmetric, code]*

**value** [*code*] *permutations\_of\_multiset {#1,2,3,4::int#}*

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

**function** *permutations\_of\_set\_aux* **where**  
*permutations\_of\_set\_aux acc A =*  
*(if ¬finite A then {} else if A = {} then {acc} else*  
*(⋃ x∈A. permutations\_of\_set\_aux (x#acc) (A - {x})))*  
**by** *auto*  
**termination** **by** (*relation Wellfounded.measure (card ∘ snd)*) (*simp\_all add: card\_gt\_0\_iff*)

**lemma** *permutations\_of\_set\_aux\_altdef*:  
*permutations\_of\_set\_aux acc A = (λxs. rev xs @ acc) ‘ permutations\_of\_set A*  
**proof** (*cases finite A*)  
**assume** *finite A*  
**thus** *?thesis*  
**proof** (*induction A arbitrary: acc rule: finite\_psubset\_induct*)  
**case** (*psubset A acc*)  
**show** *?case*

```

proof (cases A = {})
  case False
  note [simp del] = permutations_of_set_aux.simps
  from psubset.hyps False
  have permutations_of_set_aux acc A =
    (⋃ y∈A. permutations_of_set_aux (y#acc) (A - {y}))
  by (subst permutations_of_set_aux.simps) simp_all
  also have ... = (⋃ y∈A. (λxs. rev xs @ acc) ‘ (λxs. y # xs) ‘ permuta-
tions_of_set (A - {y}))
  apply (rule arg_cong [of _ _ Union], rule image_cong)
  apply (simp_all add: image_image)
  apply (subst psubset)
  apply auto
  done
  also from False have ... = (λxs. rev xs @ acc) ‘ permutations_of_set A
  by (subst (2) permutations_of_set_nonempty) (simp_all add: image_UN)
  finally show ?thesis .
qed simp_all
qed
qed (simp_all add: permutations_of_set_infinite)

```

```

declare permutations_of_set_aux.simps [simp del]

```

```

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

```

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

```

declare length_remove1 [termination_simp]

```

```

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

```

```

definition permutations_of_set_list where
  permutations_of_set_list xs = permutations_of_set_aux_list [] xs

```

```

declare permutations_of_set_aux_list.simps [simp del]

```

```

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

```

```
simp_all add: set_list_bind)
```

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

**lemma** *distinct\_permutations\_of\_set\_aux\_list*:

```
distinct xs  $\implies$  distinct (permutations_of_set_aux_list acc xs)
```

```
by (induction acc xs rule: permutations_of_set_aux_list.induct)
```

```
(subst permutations_of_set_aux_list.simps,
```

```
auto intro!: distinct_list_bind simp: disjoint_family_on_def
```

```
permutations_of_set_aux_list_refine permutations_of_set_aux_altdef)
```

**lemma** *distinct\_permutations\_of\_set\_list*:

```
distinct xs  $\implies$  distinct (permutations_of_set_list xs)
```

```
by (simp add: permutations_of_set_list_def distinct_permutations_of_set_aux_list)
```

**lemma** *permutations\_of\_list*:

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

```
by (simp add: permutations_of_set_aux_correct [symmetric]
```

```
permutations_of_set_aux_list_refine permutations_of_set_list_def)
```

**lemma** *permutations\_of\_list\_code* [code]:

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

```
permutations_of_set (List.coset xs) =
```

```
Code.abort (STR "Permutation of set complement not supported")
```

```
( $\lambda$ _. permutations_of_set (List.coset xs))
```

```
by (simp_all add: permutations_of_list)
```

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

```
end
```

**theory** *Cycles*

```
imports
```

```
  HOL-Library.FuncSet
```

```
  Permutations
```

```
begin
```

## 6 Cycles

### 6.1 Definitions

**abbreviation** *cycle* :: 'a list  $\Rightarrow$  bool

```
where cycle cs  $\equiv$  distinct cs
```

```

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

```

## 6.2 Basic Properties

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

```

lemma id_outside_supp:
  assumes  $x \notin \text{set } cs$  shows (cycle_of_list cs) x = x
  using assms by (induct cs rule: cycle_of_list.induct) (simp_all)

```

```

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

```

```

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

```

```

theorem cyclic_rotation:
  assumes cycle cs shows map ((cycle_of_list cs)  $\overset{\sim}{\sim}$  n) cs = rotate n cs
proof -
  { have map (cycle_of_list cs) cs = rotate1 cs using assms(1)
    proof (induction cs rule: cycle_of_list.induct)
      case (1 i j cs)
      then have  $\langle i \notin \text{set } cs \rangle \langle j \notin \text{set } cs \rangle$ 
        by auto
      then have  $\langle \text{map } (\text{Transposition.transpose } i j) cs = cs \rangle$ 
        by (auto intro: map_idI simp add: transpose_eq_iff)
      show ?case
    proof (cases)
      assume cs = Nil thus ?thesis by simp
    next
      assume cs  $\neq$  Nil hence ge_two: length (j # cs)  $\geq$  2
        using not_less by auto
      have map (cycle_of_list (i # j # cs)) (i # j # cs) =
        map (transpose i j) (map (cycle_of_list (j # cs)) (i # j # cs)) by
    simp
      also have ... = map (transpose i j) (i # (rotate1 (j # cs)))
        by (metis 1.IH 1.prem distinct.simps(2) id_outside_supp list.simps(9))
      also have ... = map (transpose i j) (i # (cs @ [j])) by simp
      also have ... = j # (map (transpose i j) cs) @ [i] by simp
      also have ... = j # cs @ [i]

```



```

    using ⟨map (Transposition.transpose i j) cs = cs⟩ by simp
    also have ... = rotate1 (i # j # cs) by simp
    finally show ?thesis .
  qed
  qed simp_all }
  note cyclic_rotation' = this

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

  corollary cycle_is_surj:
  assumes cycle cs shows (cycle_of_list cs) ' (set cs) = (set cs)
  using cyclic_rotation[OF assms, of Suc 0] by (simp add: image_set)

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

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

  moreover have (cycle_of_list cs) i = (cycle_of_list (rotate1 cs)) i if i ∉
set cs for i
  using that by (simp add: id_outside_supp)
}
}

```

```

    ultimately show (cycle_of_list cs) = (cycle_of_list (rotate1 cs))
      by blast
  qed } note rotate1_lemma = this

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

```

### 6.3 Conjugation of cycles

```

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

```

### 6.4 When Cycles Commute

```

lemma cycles_commute:
  assumes cycle p cycle q and set p ∩ set q = {}
  shows (cycle_of_list p) ∘ (cycle_of_list q) = (cycle_of_list q) ∘ (cycle_of_list
p)
proof
  { fix p :: 'a list and q :: 'a list and i :: 'a
    assume A: cycle p cycle q set p ∩ set q = {} i ∈ set p i ∉ set q
    have ((cycle_of_list p) ∘ (cycle_of_list q)) i =
      ((cycle_of_list q) ∘ (cycle_of_list p)) i
    proof -
      have ((cycle_of_list p) ∘ (cycle_of_list q)) i = (cycle_of_list p) i
        using id_outside_supp[OF A(5)] by simp
    }
  }

```

```

    also have ... = ((cycle_of_list q) o (cycle_of_list p)) i
      using id_outside_supp[of (cycle_of_list p) i] cycle_is_surj[OF A(1)]
A(3,4) by fastforce
    finally show ?thesis .
  qed } note aui_lemma = this

```

```

fix i consider i ∈ set p i ∉ set q | i ∉ set p i ∈ set q | i ∉ set p i ∉ set q
  using ⟨set p ∩ set q = {}⟩ by blast
thus ((cycle_of_list p) o (cycle_of_list q)) i = ((cycle_of_list q) o (cycle_of_list
p)) i
proof cases
  case 1 thus ?thesis
    using aui_lemma[OF assms] by simp
next
  case 2 thus ?thesis
    using aui_lemma[OF assms(2,1)] assms(3) by (simp add: ac_simps)
next
  case 3 thus ?thesis
    by (simp add: id_outside_supp)
qed
qed

```

## 6.5 Cycles from Permutations

### 6.5.1 Exponentiation of permutations

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

```

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

```

```

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

```

```

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

```

```

lemma permutation_is_nilpotent:
  assumes permutation p obtains n where (p ^^ n) = id and n > 0

```

**proof** –  
**obtain**  $S$  **where**  $\text{finite } S$  **and**  $p$  *permutes*  $S$   
**using**  $\text{assms unfolding permutation\_permutes}$  **by**  $\text{blast}$   
**hence**  $\exists n. (p \text{ } \sim n) = \text{id} \wedge n > 0$   
**proof** (*induct*  $S$  *arbitrary*:  $p$ )  
**case**  $\text{empty}$  **thus**  $?case$   
**using**  $\text{id\_funpow[of 1] unfolding permutes\_empty}$  **by**  $\text{blast}$   
**next**  
**case** ( $\text{insert } s S$ )  
**have**  $(\lambda n. (p \text{ } \sim n) s) \text{ } \text{UNIV} \subseteq (\text{insert } s S)$   
**using**  $\text{permutes\_in\_image[OF permutes\_funpow[OF insert(4)], of \_ s]}$  **by**  
 $\text{auto}$   
**hence**  $\neg \text{inj\_on } (\lambda n. (p \text{ } \sim n) s) \text{ } \text{UNIV}$   
**using**  $\text{insert(1) infinite\_iff\_countable\_subset unfolding sym[OF finite\_insert, of } S s]$  **by**  $\text{metis}$   
**then obtain**  $i j$  **where**  $ij: i < j (p \text{ } \sim i) s = (p \text{ } \sim j) s$   
**unfolding**  $\text{inj\_on\_def}$  **by** ( $\text{metis nat\_neq\_iff}$ )  
**hence**  $(p \text{ } \sim (j - i)) s = s$   
**using**  $\text{funpow\_diff[OF permutes\_inj[OF insert(4)]] le\_eq\_less\_or\_eq}$  **by**  
 $\text{blast}$   
**hence**  $p \text{ } \sim (j - i)$  *permutes*  $S$   
**using**  $\text{permutes\_superset[OF permutes\_funpow[OF insert(4), of } j - i], of } S]$   
**by**  $\text{auto}$   
**then obtain**  $n$  **where**  $n: ((p \text{ } \sim (j - i)) \text{ } \sim n) = \text{id} \wedge n > 0$   
**using**  $\text{insert(3)}$  **by**  $\text{blast}$   
**thus**  $?case$   
**using**  $\text{ij(1) nat\_0\_less\_mult\_iff zero\_less\_diff unfolding funpow\_mult}$  **by**  
 $\text{metis}$   
**qed**  
**thus**  $\text{thesis}$   
**using**  $\text{that}$  **by**  $\text{blast}$   
**qed**

**lemma**  $\text{permutation\_is\_nilpotent'}$ :

**assumes**  $\text{permutation } p$  **obtains**  $n$  **where**  $(p \text{ } \sim n) = \text{id}$  **and**  $n > m$

**proof** –

**obtain**  $n$  **where**  $(p \text{ } \sim n) = \text{id}$  **and**  $n > 0$

**using**  $\text{permutation\_is\_nilpotent[OF assms]}$  **by**  $\text{blast}$

**then obtain**  $k$  **where**  $n * k > m$

**by** ( $\text{metis dividend\_less\_times\_div mult\_Suc\_right}$ )

**from**  $\langle (p \text{ } \sim n) = \text{id} \rangle$  **have**  $p \text{ } \sim (n * k) = \text{id}$

**by** (*induct*  $k$ ) ( $\text{simp, metis funpow\_mult id\_funpow}$ )

**with**  $\langle n * k > m \rangle$  **show**  $\text{thesis}$

**using**  $\text{that}$  **by**  $\text{blast}$

**qed**

## 6.5.2 Extraction of cycles from permutations

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

where  $\text{least\_power } f x = (\text{LEAST } n. (f \sim n) x = x \wedge n > 0)$

**abbreviation**  $\text{support} :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \text{ list}$   
 where  $\text{support } p x \equiv \text{map } (\lambda i. (p \sim i) x) [0..< (\text{least\_power } p x)]$

**lemma**  $\text{least\_powerI}$ :  
 assumes  $(f \sim n) x = x$  and  $n > 0$   
 shows  $(f \sim (\text{least\_power } f x)) x = x$  and  $\text{least\_power } f x > 0$   
 using *assms unfolding least\_power\_def* by (*metis (mono\_tags, lifting) LeastI*)+

**lemma**  $\text{least\_power\_le}$ :  
 assumes  $(f \sim n) x = x$  and  $n > 0$  shows  $\text{least\_power } f x \leq n$   
 using *assms unfolding least\_power\_def* by (*simp add: Least\_le*)

**lemma**  $\text{least\_power\_of\_permutation}$ :  
 assumes *permutation*  $p$  shows  $(p \sim (\text{least\_power } p a)) a = a$  and  $\text{least\_power } p a > 0$   
 using *permutation\_is\_nilpotent[OF assms] least\_powerI* by (*metis id\_apply*)+

**lemma**  $\text{least\_power\_gt\_one}$ :  
 assumes *permutation*  $p$  and  $p a \neq a$  shows  $\text{least\_power } p a > \text{Suc } 0$   
 using *least\_power\_of\_permutation[OF assms(1)] assms(2)*  
 by (*metis Suc\_lessI funpow.simps(2) funpow\_simps\_right(1) o\_id*)

**lemma**  $\text{least\_power\_minimal}$ :  
 assumes  $(p \sim n) a = a$  shows  $(\text{least\_power } p a) \text{ dvd } n$   
**proof** (*cases n = 0, simp*)  
 let  $?lpow = \text{least\_power } p$

assume  $n \neq 0$  then have  $n > 0$  by *simp*  
 hence  $(p \sim (?lpow a)) a = a$  and  $\text{least\_power } p a > 0$   
 using *assms unfolding least\_power\_def* by (*metis (mono\_tags, lifting) LeastI*)+  
 hence *aux\_lemma*:  $(p \sim ((?lpow a) * k)) a = a$  for  $k :: \text{nat}$   
 by (*induct k (simp\_all add: funpow\_add)*)

have  $(p \sim (n \text{ mod } ?lpow a)) ((p \sim (n - (n \text{ mod } ?lpow a))) a) = (p \sim n) a$   
 by (*metis add\_diff\_inverse\_nat funpow\_add mod\_less\_eq\_dividend not\_less o\_apply*)  
 with  $\langle (p \sim n) a = a \rangle$  have  $(p \sim (n \text{ mod } ?lpow a)) a = a$   
 using *aux\_lemma* by (*simp add: minus\_mod\_eq\_mult\_div*)  
 hence  $?lpow a \leq n \text{ mod } ?lpow a$  if  $n \text{ mod } ?lpow a > 0$   
 using *least\_power\_le[OF \_ that, of p a]* by *simp*  
 with  $\langle \text{least\_power } p a > 0 \rangle$  show  $(\text{least\_power } p a) \text{ dvd } n$   
 using *mod\_less\_divisor not\_le* by *blast*

qed

**lemma**  $\text{least\_power\_dvd}$ :  
 assumes *permutation*  $p$  shows  $(\text{least\_power } p a) \text{ dvd } n \longleftrightarrow (p \sim n) a = a$

```

proof
  show  $(p \sim n) a = a \implies (\text{least\_power } p a) \text{ dvd } n$ 
    using least_power_minimal[of _ p] by simp
next
  have  $(p \sim ((\text{least\_power } p a) * k)) a = a$  for  $k :: \text{nat}$ 
    using least_power_of_permutation(1)[OF assms(1)] by (induct k) (simp_all
add: funpow_add)
  thus  $(\text{least\_power } p a) \text{ dvd } n \implies (p \sim n) a = a$  by blast
qed

```

**theorem** *cycle\_of\_permutation:*

```

  assumes permutation p shows cycle (support p a)
proof -
  have  $(\text{least\_power } p a) \text{ dvd } (j - i)$  if  $i \leq j$   $j < \text{least\_power } p a$  and  $(p \sim i) a = (p \sim j) a$  for  $i j$ 
    using funpow_diff[OF bij_is_inj that(1,3)] assms by (simp add: permutation
least_power_dvd)
  moreover have  $i = j$  if  $i \leq j$   $j < \text{least\_power } p a$  and  $(\text{least\_power } p a) \text{ dvd } (j - i)$  for  $i j$ 
    using that le_eq_less_or_eq nat_dvd_not_less by auto
  ultimately have inj_on  $(\lambda i. (p \sim i) a)$   $\{.. < (\text{least\_power } p a)\}$ 
    unfolding inj_on_def by (metis le_cases lessThan_iff)
  thus ?thesis
    by (simp add: atLeast_upt distinct_map)
qed

```

## 6.6 Decomposition on Cycles

We show that a permutation can be decomposed on cycles

### 6.6.1 Preliminaries

**lemma** *support\_set:*

```

  assumes permutation p shows  $\text{set } (\text{support } p a) = \text{range } (\lambda i. (p \sim i) a)$ 
proof
  show  $\text{set } (\text{support } p a) \subseteq \text{range } (\lambda i. (p \sim i) a)$ 
    by auto
next
  show  $\text{range } (\lambda i. (p \sim i) a) \subseteq \text{set } (\text{support } p a)$ 
    proof (auto)
      fix  $i$ 
      have  $(p \sim i) a = (p \sim (i \bmod (\text{least\_power } p a))) ((p \sim (i - (i \bmod (\text{least\_power } p a)))) a)$ 
        by (metis add_diff_inverse_nat funpow_add mod_less_eq_dividend not_le
o_apply)
      also have  $\dots = (p \sim (i \bmod (\text{least\_power } p a))) a$ 
        using least_power_dvd[OF assms] by (metis dvd_minus_mod)
      also have  $\dots \in (\lambda i. (p \sim i) a)$   $\{0.. < (\text{least\_power } p a)\}$ 
        using least_power_of_permutation(2)[OF assms] by fastforce

```

```

    finally show  $(p \sim i) a \in (\lambda i. (p \sim i) a) \setminus \{0..< (least\_power\ p\ a)\}$  .
qed
qed

lemma disjoint_support:
  assumes permutation p shows disjoint (range  $(\lambda a. set (support\ p\ a))$ ) (is disjoint ?A)
proof (rule disjointI)
  { fix i j a b
    assume  $set (support\ p\ a) \cap set (support\ p\ b) \neq \{\}$  have  $set (support\ p\ a) \subseteq set (support\ p\ b)$ 
    unfolding support_set[OF assms]
    proof (auto)
      from  $set (support\ p\ a) \cap set (support\ p\ b) \neq \{\}$ 
      obtain i j where  $ij: (p \sim i) a = (p \sim j) b$ 
      by auto

      fix k
      have  $(p \sim k) a = (p \sim (k + (least\_power\ p\ a) * l)) a$  for l
      using least_power_dvd[OF assms] by (induct l) (simp, metis dvd_triv_left funpow_add o_def)
      then obtain m where  $m \geq i$  and  $(p \sim m) a = (p \sim k) a$ 
      using least_power_of_permutation(2)[OF assms]
      by (metis dividend_less_times_div le_eq_less_or_eq mult_Suc_right trans_less_add2)
      hence  $(p \sim m) a = (p \sim (m - i)) ((p \sim i) a)$ 
      by (metis Nat.le_imp_diff_is_add funpow_add o_apply)
      with  $\langle (p \sim m) a = (p \sim k) a \rangle$  have  $(p \sim k) a = (p \sim ((m - i) + j)) b$ 
      unfolding ij by (simp add: funpow_add)
      thus  $(p \sim k) a \in range (\lambda i. (p \sim i) b)$ 
      by blast
    qed } note aux_lemma = this

  fix supp_a supp_b
  assume  $supp_a \in ?A$  and  $supp_b \in ?A$ 
  then obtain a b where  $a: supp_a = set (support\ p\ a)$  and  $b: supp_b = set (support\ p\ b)$ 
  by auto
  assume  $supp_a \neq supp_b$  thus  $supp_a \cap supp_b = \{\}$ 
  using aux_lemma unfolding a b by blast
qed

lemma disjoint_support':
  assumes permutation p
  shows  $set (support\ p\ a) \cap set (support\ p\ b) = \{\} \iff a \notin set (support\ p\ b)$ 
proof -
  have  $a \in set (support\ p\ a)$ 
  using least_power_of_permutation(2)[OF assms] by force
  show ?thesis

```

```

proof
  assume  $set (support\ p\ a) \cap set (support\ p\ b) = \{\}$ 
  with  $\langle a \in set (support\ p\ a) \rangle$  show  $a \notin set (support\ p\ b)$ 
  by blast
next
  assume  $a \notin set (support\ p\ b)$  show  $set (support\ p\ a) \cap set (support\ p\ b) = \{\}$ 
  proof (rule ccontr)
    assume  $set (support\ p\ a) \cap set (support\ p\ b) \neq \{\}$ 
    hence  $set (support\ p\ a) = set (support\ p\ b)$ 
    using disjoint_support[OF assms] by (meson UNIV_I disjoint_def image_iff)
    with  $\langle a \in set (support\ p\ a) \rangle$  and  $\langle a \notin set (support\ p\ b) \rangle$  show False
    by simp
  qed
qed
qed

```

**lemma** *support\_coverture*:

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

**proof**

**show**  $\{ a. p\ a \neq a \} \subseteq \bigcup \{ set (support\ p\ a) \mid a. p\ a \neq a \}$

**proof**

**fix** *a* **assume**  $a \in \{ a. p\ a \neq a \}$

**have**  $a \in set (support\ p\ a)$

**using** *least\_power\_of\_permutation(2)[OF assms, of a]* **by** *force*

**with**  $\langle a \in \{ a. p\ a \neq a \} \rangle$  **show**  $a \in \bigcup \{ set (support\ p\ a) \mid a. p\ a \neq a \}$

**by** *blast*

**qed**

**next**

**show**  $\bigcup \{ set (support\ p\ a) \mid a. p\ a \neq a \} \subseteq \{ a. p\ a \neq a \}$

**proof**

**fix** *b* **assume**  $b \in \bigcup \{ set (support\ p\ a) \mid a. p\ a \neq a \}$

**then obtain** *a i* **where**  $p\ a \neq a$  **and**  $(p \ \overset{\sim}{\sim} \ i)\ a = b$

**by** *auto*

**have**  $p\ a = a$  **if**  $(p \ \overset{\sim}{\sim} \ i)\ a = (p \ \overset{\sim}{\sim} \ Suc\ i)\ a$

**using** *funpow\_diff[OF bij\_is\_inj \_ that] assms unfolding permutation by*

*simp*

**with**  $\langle p\ a \neq a \rangle$  **and**  $\langle (p \ \overset{\sim}{\sim} \ i)\ a = b \rangle$  **show**  $b \in \{ a. p\ a \neq a \}$

**by** *auto*

**qed**

**qed**

**theorem** *cycle\_restrict*:

**assumes** *permutation p* **and**  $b \in set (support\ p\ a)$  **shows**  $p\ b = (cycle\_of\_list (support\ p\ a))\ b$

**proof** –

**note** *least\_power\_props [simp] = least\_power\_of\_permutation[OF assms(1)]*



```

have map (cycle_of_list (support p a)) (support p a) = rotate1 (support p a)
  using cyclic_rotation[OF cycle_of_permutation[OF assms(1)], of 1 a] by simp
hence map (cycle_of_list (support p a)) (support p a) = tl (support p a) @ [ a ]
  by (simp add: hd_map rotate1_hd_tl)
also have ... = map p (support p a)
proof (rule nth_equalityI, auto)
  fix i assume i < least_power p a show (tl (support p a) @ [a]) ! i = p ((p ~
i) a)
  proof (cases)
    assume i: i = least_power p a - 1
    hence (tl (support p a) @ [ a ]) ! i = a
      by (metis (no_types, lifting) diff_zero length_map length_tl length_upt
nth_append_length)
    also have ... = p ((p ~ i) a)
      by (metis (mono_tags, opaque_lifting) least_power_props i Suc_diff_1
funpow_simps_right(2) funpow_swap1 o_apply)
    finally show ?thesis .
  next
    assume i ≠ least_power p a - 1
    with ⟨i < least_power p a⟩ have i < least_power p a - 1
      by simp
    hence (tl (support p a) @ [ a ]) ! i = (p ~ (Suc i)) a
      by (metis One_nat_def Suc_eq_plus1 add commute length_map length_upt
map_tl_nth_append_nth_map_upt_tl_upt)
    thus ?thesis
      by simp
  qed
qed
finally have map (cycle_of_list (support p a)) (support p a) = map p (support
p a) .
thus ?thesis
  using assms(2) by auto
qed

```

## 6.6.2 Decomposition

```

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

```

**lemma** semidecomposition:

```

assumes p permutes S and finite S
shows (λy. if y ∈ (S - set (support p a)) then p y else y) permutes (S - set
(support p a))
proof (rule bij_imp_permutes)
show (if b ∈ (S - set (support p a)) then p b else b) = b if b ∉ S - set (support

```

```

p a) for b
  using that by auto
next
have is_permutation: permutation p
  using assms unfolding permutation_permutes by blast

let ?q =  $\lambda y. \text{if } y \in (S - \text{set } (\text{support } p \ a)) \text{ then } p \ y \text{ else } y$ 
show bij_betw ?q (S - set (support p a)) (S - set (support p a))
proof (rule bij_betw_imageI)
  show inj_on ?q (S - set (support p a))
    using permutes_inj[OF assms(1)] unfolding inj_on_def by auto
next
  have aux_lemma: set (support p s)  $\subseteq$  (S - set (support p a)) if s  $\in$  S - set
(support p a) for s
  proof -
    have (p  $\hat{\sim}$  i) s  $\in$  S for i
      using that unfolding permutes_in_image[OF permutes_funpow[OF assms(1)]]
    by simp
    thus ?thesis
      using that disjoint_support'[OF is_permutation, of s a] by auto
  qed
  have (p  $\hat{\sim}$  1) s  $\in$  set (support p s) for s
    unfolding support_set[OF is_permutation] by blast
  hence p s  $\in$  set (support p s) for s
    by simp
  hence p '(S - set (support p a))  $\subseteq$  S - set (support p a)
    using aux_lemma by blast
  moreover have (p  $\hat{\sim}$  ((least_power p s) - 1)) s  $\in$  set (support p s) for s
    unfolding support_set[OF is_permutation] by blast
  hence  $\exists s' \in \text{set } (\text{support } p \ s). \ p \ s' = s$  for s
    using least_power_of_permutation[OF is_permutation] by (metis Suc_diff_1
funpow.simps(2) o_apply)
  hence S - set (support p a)  $\subseteq$  p '(S - set (support p a))
    using aux_lemma
  by (clarsimp simp add: image_iff) (metis image_subset_iff)
  ultimately show ?q '(S - set (support p a)) = (S - set (support p a))
    by auto
  qed
qed

theorem cycle_decomposition:
  assumes p_permutes S and finite S shows cycle_decomp S p
  using assms
proof (induct card S arbitrary: S p rule: less_induct)
  case less show ?case
  proof (cases)
    assume S = {} thus ?thesis
      using empty_less(2) by auto
  next

```

```

have is_permutation: permutation p
  using less(2-3) unfolding permutation_permutes by blast

assume S ≠ {} then obtain s where s ∈ S
  by blast
define q where q = (λy. if y ∈ (S - set (support p s)) then p y else y)
have (cycle_of_list (support p s) ∘ q) = p
proof
  fix a
  consider a ∈ S - set (support p s) | a ∈ set (support p s) | a ∉ S a ∉ set
(support p s)
  by blast
  thus ((cycle_of_list (support p s) ∘ q)) a = p a
  proof cases
    case 1
    have (p ~ 1) a ∈ set (support p a)
      unfolding support_set[OF is_permutation] by blast
    with ⟨a ∈ S - set (support p s)⟩ have p a ∉ set (support p s)
      using disjoint_support'[OF is_permutation, of a s] by auto
    with ⟨a ∈ S - set (support p s)⟩ show ?thesis
      using id_outside_supp[of _ support p s] unfolding q_def by simp
  next
    case 2 thus ?thesis
      using cycle_restrict[OF is_permutation] unfolding q_def by simp
  next
    case 3 thus ?thesis
      using id_outside_supp[OF 3(2)] less(2) permutes_not_in unfolding
q_def by fastforce
  qed
qed

moreover from ⟨s ∈ S⟩ have (p ~ i) s ∈ S for i
  unfolding permutes_in_image[OF permutes_funpow[OF less(2)]] .
hence set (support p s) ∪ (S - set (support p s)) = S
  by auto

moreover have s ∈ set (support p s)
  using least_power_of_permutation[OF is_permutation] by force
with ⟨s ∈ S⟩ have card (S - set (support p s)) < card S
  using less(3) by (metis DiffE card_seteq linorder_not_le subsetI)
hence cycle_decomp (S - set (support p s)) q
  using less(1)[OF _ semidecomposition[OF less(2-3)], of s] less(3) unfolding
q_def by blast

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

```

qed

end

## 7 Permutations as abstract type

```
theory Perm
  imports
    Transposition
begin
```

This theory introduces basics about permutations, i.e. almost everywhere fix bijections. But it is by no means complete. 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 perm = {f :: 'a ⇒ 'a. bij f ∧ finite {a. f a ≠ a}}
  morphisms apply Perm
proof
  show id ∈ ?perm by simp
qed
```

```
setup_lifting type_definition_perm
```

```
notation apply (infixl <$> 999)
```

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

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

```
lemma perm_eq_iff:
   $f = g \iff (\forall a. f \langle \$ \rangle a = g \langle \$ \rangle a)$ 
  by (auto intro: perm_eqI)
```

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

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

**lemma** *in\_affected*:

$a \in \text{affected } f \longleftrightarrow f \langle \$ \rangle a \neq a$   
**by** *transfer simp*

**lemma** *finite\_affected [simp]*:

*finite (affected f)*  
**by** *transfer simp*

**lemma** *apply\_affected [simp]*:

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

**proof** *transfer*

**fix**  $f :: 'a \Rightarrow 'a$  **and**  $a :: 'a$

**assume**  $\text{bij } f \wedge \text{finite } \{b. f b \neq b\}$

**then have** *bij f* **by** *simp*

**interpret** *bijection f* **by** *standard (rule ‹bij f›)*

**have**  $f a \in \{a. f a = a\} \longleftrightarrow a \in \{a. f a = a\}$  (**is**  $?P \longleftrightarrow ?Q$ )  
**by** *auto*

**then show**  $f a \in \{a. f a \neq a\} \longleftrightarrow a \in \{a. f a \neq a\}$   
**by** *simp*

**qed**

**lemma** *card\_affected\_not\_one*:

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

**proof**

**interpret** *bijection apply f*

**by** *standard (rule bij\_apply)*

**assume**  $\text{card } (\text{affected } f) = 1$

**then obtain**  $a$  **where**  $*$ :  $\text{affected } f = \{a\}$

**by** (*rule card\_1\_singletonE*)

**then have**  $*$ :  $f \langle \$ \rangle a \neq a$

**by** (*simp flip: in\_affected*)

**with**  $*$  **have**  $f \langle \$ \rangle a \notin \text{affected } f$

**by** *simp*

**then have**  $f \langle \$ \rangle (f \langle \$ \rangle a) = f \langle \$ \rangle a$

**by** (*simp add: in\_affected*)

**then have**  $\text{inv } (\text{apply } f) (f \langle \$ \rangle (f \langle \$ \rangle a)) = \text{inv } (\text{apply } f) (f \langle \$ \rangle a)$

**by** *simp*

**with**  $*$  **show** *False* **by** *simp*

**qed**

## 7.2 Identity, composition and inversion

**instantiation** *Perm.perm* :: (type) {*monoid\_mult, inverse*}

**begin**

**lift\_definition** *one\_perm* :: 'a perm

**is** *id*

**by** *simp*

```

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

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

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

lift_definition times_perm :: 'a perm  $\Rightarrow$  'a perm  $\Rightarrow$  'a perm
  is comp
proof
  fix f g :: 'a  $\Rightarrow$  'a
  assume bij f  $\wedge$  finite {a. f a  $\neq$  a}
    bij g  $\wedge$  finite {a. g a  $\neq$  a}
  then have finite ({a. f a  $\neq$  a}  $\cup$  {a. g a  $\neq$  a})
    by simp
  moreover have {a. (f  $\circ$  g) a  $\neq$  a}  $\subseteq$  {a. f a  $\neq$  a}  $\cup$  {a. g a  $\neq$  a}
    by auto
  ultimately show finite {a. (f  $\circ$  g) a  $\neq$  a}
    by (auto intro: finite_subset)
qed (auto intro: bij_comp)

lemma apply_times:
  apply (f * g) = apply f  $\circ$  apply g
  by (fact times_perm.rep_eq)

lemma apply_sequence:
  f  $\langle$ $ $\rangle$  (g  $\langle$ $ $\rangle$  a) = apply (f * g) a
  by (simp add: apply_times)

lemma affected_times [simp]:
  affected (f * g)  $\subseteq$  affected f  $\cup$  affected g
  by transfer auto

lift_definition inverse_perm :: 'a perm  $\Rightarrow$  'a perm
  is inv
proof transfer
  fix f :: 'a  $\Rightarrow$  'a and a
  assume bij f  $\wedge$  finite {b. f b  $\neq$  b}
  then have bij f and fin: finite {b. f b  $\neq$  b}
    by auto
  interpret bijection f by standard (rule  $\langle$ bij f $\rangle$ )
  from fin show bij (inv f)  $\wedge$  finite {a. inv f a  $\neq$  a}
    by (simp add: bij_inv)

```

```

qed

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

end

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

lemma affected_inverse [simp]:
  affected (inverse f) = affected f
proof transfer
  fix f :: 'a ⇒ 'a and a
  assume bij f ∧ finite {b. f b ≠ b}
  then have bij f by simp
  interpret bijection f by standard (rule ⟨bij f⟩)
  show {a. inv f a ≠ a} = {a. f a ≠ a}
  by simp
qed

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

declare perm.inverse_distrib_swap [simp]

lemma perm_mult_commute:
  assumes affected f ∧ affected g = {}
  shows g * f = f * g
proof (rule perm_eqI)
  fix a
  from assms have *: a ∈ affected f ⇒ a ∉ affected g
  a ∈ affected g ⇒ a ∉ affected f for a
  by auto
  consider a ∈ affected f ∧ a ∉ affected g

```

```

       $\wedge f \langle \$ \rangle a \in \text{affected } f$ 
    |  $a \notin \text{affected } f \wedge a \in \text{affected } g$ 
       $\wedge f \langle \$ \rangle a \notin \text{affected } f$ 
    |  $a \notin \text{affected } f \wedge a \notin \text{affected } g$ 
  using assms by auto
then show  $(g * f) \langle \$ \rangle a = (f * g) \langle \$ \rangle a$ 
proof cases
  case 1
  with * have  $f \langle \$ \rangle a \notin \text{affected } g$ 
  by auto
  with 1 show ?thesis by (simp add: in_affected apply_times)
next
  case 2
  with * have  $g \langle \$ \rangle a \notin \text{affected } f$ 
  by auto
  with 2 show ?thesis by (simp add: in_affected apply_times)
next
  case 3
  then show ?thesis by (simp add: in_affected apply_times)
qed
qed

```

```

lemma apply_power:
   $\text{apply } (f \wedge n) = \text{apply } f \wedge n$ 
  by (induct n) (simp_all add: apply_times)

```

```

lemma perm_power_inverse:
   $\text{inverse } f \wedge n = \text{inverse } ((f :: 'a \text{ perm}) \wedge n)$ 
proof (induct n)
  case 0 then show ?case by simp
next
  case (Suc n)
  then show ?case
    unfolding power_Suc2 [of f] by simp
qed

```

### 7.3 Orbit and order of elements

```

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

```

```

lemma in_orbitI:
  assumes  $(f \wedge n) \langle \$ \rangle a = b$ 
  shows  $b \in \text{orbit } f a$ 
  using assms by (auto simp add: orbit_def)

```

```

lemma apply_power_self_in_orbit [simp]:
   $(f \wedge n) \langle \$ \rangle a \in \text{orbit } f a$ 

```



```

    by (rule in_orbitI) rule

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

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

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

lemma not_in_affected_iff_orbit_eq_singleton:
  a ∉ affected f ⟷ orbit f a = {a} (is ?P ⟷ ?Q)
proof
  assume ?P
  then have f ⟨$⟩ a = a
    by (simp add: in_affected)
  then have (f ^ n) ⟨$⟩ a = a for n
    by (induct n) (simp_all add: apply_times)
  then show ?Q
    by (auto simp add: orbit_def)
next
  assume ?Q
  then show ?P
    by (auto simp add: orbit_def in_affected dest: range_eq_singletonD [of _ _
1])
qed

definition order :: 'a perm ⇒ 'a ⇒ nat
where
  order f = card ∘ orbit f

lemma orbit_subset_eq_affected:
  assumes a ∈ affected f
  shows orbit f a ⊆ affected f
proof (rule ccontr)
  assume ¬ orbit f a ⊆ affected f
  then obtain b where b ∈ orbit f a and b ∉ affected f
    by auto
  then have b ∈ range (λn. (f ^ n) ⟨$⟩ a)
    by (simp add: orbit_def)
  then obtain n where b = (f ^ n) ⟨$⟩ a
    by blast
  with ⟨b ∉ affected f⟩
  have (f ^ n) ⟨$⟩ a ∉ affected f
    by simp

```

```

then have  $f \langle \$ \rangle a \notin \text{affected } f$ 
  by (induct n) (simp_all add: apply_times)
with assms show False
  by simp
qed

lemma finite_orbit [simp]:
  finite (orbit f a)
proof (cases a ∈ affected f)
  case False then show ?thesis
    by (simp add: not_in_affected_iff_orbit_eq_singleton)
next
  case True then have  $\text{orbit } f \ a \subseteq \text{affected } f$ 
    by (rule orbit_subset_eq_affected)
  then show ?thesis using finite_affected
    by (rule finite_subset)
qed

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

lemma order_1 [simp]:
   $\text{order } 1 \ a = 1$ 
  unfolding order_def by simp

lemma card_orbit_eq [simp]:
   $\text{card } (\text{orbit } f \ a) = \text{order } f \ a$ 
  by (simp add: order_def)

lemma order_greater_zero [simp]:
   $\text{order } f \ a > 0$ 
  by (simp only: card_gt_0_iff_order_def comp_def) simp

lemma order_eq_one_iff:
   $\text{order } f \ a = \text{Suc } 0 \longleftrightarrow a \notin \text{affected } f \ (\text{is } ?P \longleftrightarrow ?Q)$ 
proof
  assume ?P then have  $\text{card } (\text{orbit } f \ a) = 1$ 
    by simp
  then obtain b where  $\text{orbit } f \ a = \{b\}$ 
    by (rule card_1_singletonE)
  with in_orbit_self [of a f]
    have  $b = a$  by simp
  with  $\langle \text{orbit } f \ a = \{b\} \rangle$  show ?Q
    by (simp add: not_in_affected_iff_orbit_eq_singleton)
next
  assume ?Q
  then have  $\text{orbit } f \ a = \{a\}$ 
    by (simp add: not_in_affected_iff_orbit_eq_singleton)

```

```

then have card (orbit f a) = 1
  by simp
then show ?P
  by simp
qed

```

```

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

```

```

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

```

```

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

```

```

lemma order_witness_unfold:
  assumes  $n > 0$  and  $(f \wedge n) \langle \$ \rangle a = a$ 
  shows  $order\ f\ a = card\ ((\lambda m. (f \wedge m) \langle \$ \rangle a) \cdot \{0..<n\})$ 
proof -
  have  $orbit\ f\ a = (\lambda m. (f \wedge m) \langle \$ \rangle a) \cdot \{0..<n\}$  (is  $\_ = ?B$ )
  proof (rule set_eqI, rule)
    fix  $b$ 
    assume  $b \in orbit\ f\ a$ 
    then obtain  $m$  where  $(f \wedge m) \langle \$ \rangle a = b$ 
      by (auto simp add: orbit_def)
    then have  $b = (f \wedge (m\ mod\ n + n * (m\ div\ n))) \langle \$ \rangle a$ 
      by simp
    also have  $\dots = (f \wedge (m\ mod\ n)) \langle \$ \rangle ((f \wedge (n * (m\ div\ n))) \langle \$ \rangle a)$ 
      by (simp only: power_add apply_times) simp
    also have  $(f \wedge (n * q)) \langle \$ \rangle a = a$  for  $q$ 
      by (induct q)
      (simp_all add: power_add apply_times assms)
    finally have  $b = (f \wedge (m\ mod\ n)) \langle \$ \rangle a$  .
    moreover from  $\langle n > 0 \rangle$ 
    have  $m\ mod\ n < n$ 
      by simp
    ultimately show  $b \in ?B$ 
      by auto
  next
  fix  $b$ 
  assume  $b \in ?B$ 
  then obtain  $m$  where  $(f \wedge m) \langle \$ \rangle a = b$ 
    by blast
  then show  $b \in orbit\ f\ a$ 
    by (rule in_orbitI)
qed
then have  $card\ (orbit\ f\ a) = card\ ?B$ 
  by (simp only:)
then show  $?thesis$ 
  by simp
qed

```

```

lemma inj_on_apply_range:
   $inj\_on\ (\lambda m. (f \wedge m) \langle \$ \rangle a) \{..<order\ f\ a\}$ 
proof -
  have  $inj\_on\ (\lambda m. (f \wedge m) \langle \$ \rangle a) \{..<n\}$ 
    if  $n \leq order\ f\ a$  for  $n$ 
  using that proof (induct n)
    case 0 then show  $?case$  by simp
  next
  case (Suc n)
  then have  $prem: n < order\ f\ a$ 
    by simp
  with Suc.hyps have  $hyp: inj\_on\ (\lambda m. (f \wedge m) \langle \$ \rangle a) \{..<n\}$ 

```

```

    by simp
  have  $(f \wedge n) \langle \$ \rangle a \notin (\lambda m. (f \wedge m) \langle \$ \rangle a) \text{ ' } \{..<n\}$ 
  proof
    assume  $(f \wedge n) \langle \$ \rangle a \in (\lambda m. (f \wedge m) \langle \$ \rangle a) \text{ ' } \{..<n\}$ 
    then obtain  $m$  where  $*$ :  $(f \wedge m) \langle \$ \rangle a = (f \wedge n) \langle \$ \rangle a$  and  $m < n$ 
      by auto
    interpret bijection apply  $(f \wedge m)$ 
      by standard simp
    from  $\langle m < n \rangle$  have  $n = m + (n - m)$ 
      and  $nm: 0 < n - m \ n - m \leq n$ 
      by arith+
    with  $*$  have  $(f \wedge m) \langle \$ \rangle a = (f \wedge (m + (n - m))) \langle \$ \rangle a$ 
      by simp
    then have  $(f \wedge m) \langle \$ \rangle a = (f \wedge m) \langle \$ \rangle ((f \wedge (n - m)) \langle \$ \rangle a)$ 
      by (simp add: power_add_apply_times)
    then have  $(f \wedge (n - m)) \langle \$ \rangle a = a$ 
      by simp
    with  $\langle n - m > 0 \rangle$ 
      have  $order\ f\ a = card\ ((\lambda m. (f \wedge m) \langle \$ \rangle a) \text{ ' } \{0..<n - m\})$ 
        by (rule order_witness_unfold)
      also have  $card\ ((\lambda m. (f \wedge m) \langle \$ \rangle a) \text{ ' } \{0..<n - m\}) \leq card\ \{0..<n - m\}$ 
        by (rule card_image_le) simp
      finally have  $order\ f\ a \leq n - m$ 
        by simp
    with  $prem$  show False by simp
  qed
  with  $hyp$  show  $?case$ 
    by (simp add: lessThan_Suc)
  qed
  then show  $?thesis$  by simp
  qed

```

```

lemma orbit_unfold_image:
  orbit  $f\ a = (\lambda n. (f \wedge n) \langle \$ \rangle a) \text{ ' } \{..<order\ f\ a\}$  (is  $\_ = ?A$ )
proof (rule sym, rule card_subset_eq)
  show finite (orbit  $f\ a$ )
    by simp
  show  $?A \subseteq orbit\ f\ a$ 
    by (auto simp add: orbit_def)
  from inj_on_apply_range [of  $f\ a$ ]
  have  $card\ ?A = order\ f\ a$ 
    by (auto simp add: card_image)
  then show  $card\ ?A = card\ (orbit\ f\ a)$ 
    by simp
  qed

```

```

lemma in_orbitE:
  assumes  $b \in orbit\ f\ a$ 
  obtains  $n$  where  $b = (f \wedge n) \langle \$ \rangle a$  and  $n < order\ f\ a$ 

```

using *assms* **unfolding** *orbit\_unfold\_image* **by** *blast*

**lemma** *apply\_power\_order* [*simp*]:  
 $(f \text{ } ^{\text{order } f} a) \langle \$ \rangle a = a$   
**proof** –  
**have**  $(f \text{ } ^{\text{order } f} a) \langle \$ \rangle a \in \text{orbit } f a$   
**by** *simp*  
**then obtain** *n* **where**  
 $\ast: (f \text{ } ^{\text{order } f} a) \langle \$ \rangle a = (f \text{ } ^n) \langle \$ \rangle a$   
**and**  $n < \text{order } f a$   
**by** (*rule in\_orbitE*)  
**show** *?thesis*  
**proof** (*cases n*)  
**case 0 with**  $\ast$  **show** *?thesis* **by** *simp*  
**next**  
**case** (*Suc m*)  
**from** *order\_greater\_zero* [*of f a*]  
**have**  $\text{Suc } (\text{order } f a - 1) = \text{order } f a$   
**by** *arith*  
**from**  $\langle n < \text{order } f a \rangle$   
**have**  $m < \text{order } f a$   
**by** *simp*  
**with** *Suc*  $\ast$   
**have**  $(\text{inverse } f) \langle \$ \rangle ((f \text{ } ^{\text{Suc } (\text{order } f a - 1)}) \langle \$ \rangle a) =$   
 $(\text{inverse } f) \langle \$ \rangle ((f \text{ } ^{\text{Suc } m}) \langle \$ \rangle a)$   
**by** *simp*  
**then have**  $(f \text{ } ^{(\text{order } f a - 1)}) \langle \$ \rangle a =$   
 $(f \text{ } ^m) \langle \$ \rangle a$   
**by** (*simp only: power\_Suc apply\_times*)  
*(simp add: apply\_sequence mult.assoc [symmetric])*  
**with** *inj\_on\_apply\_range*  
**have**  $\text{order } f a - 1 = m$   
**by** (*rule inj\_onD*)  
*(simp\_all add: \langle m < order f a \rangle)*  
**with** *Suc* **have**  $n = \text{order } f a$   
**by** *auto*  
**with**  $\langle n < \text{order } f a \rangle$   
**show** *?thesis* **by** *simp*  
**qed**  
**qed**

**lemma** *apply\_power\_left\_mult\_order* [*simp*]:  
 $(f \text{ } ^{(n \ast \text{order } f} a)) \langle \$ \rangle a = a$   
**by** (*induct n*) (*simp\_all add: power\_add apply\_times*)

**lemma** *apply\_power\_right\_mult\_order* [*simp*]:  
 $(f \text{ } ^{(\text{order } f a \ast n)}) \langle \$ \rangle a = a$   
**by** (*simp add: ac\_simps*)

**lemma** *apply\_power\_mod\_order\_eq* [*simp*]:  
 $(f \wedge^{(n \bmod \text{order } f a)}) \langle \$ \rangle a = (f \wedge^n) \langle \$ \rangle a$   
**proof** –  
**have**  $(f \wedge^n) \langle \$ \rangle a = (f \wedge^{(n \bmod \text{order } f a + \text{order } f a * (n \text{ div } \text{order } f a))}) \langle \$ \rangle a$   
**by** *simp*  
**also have**  $\dots = (f \wedge^{(n \bmod \text{order } f a)} * f \wedge^{(\text{order } f a * (n \text{ div } \text{order } f a))}) \langle \$ \rangle a$   
**by** (*simp flip: power\_add*)  
**finally show** *?thesis*  
**by** (*simp add: apply\_times*)  
**qed**

**lemma** *apply\_power\_eq\_iff*:  
 $(f \wedge^m) \langle \$ \rangle a = (f \wedge^n) \langle \$ \rangle a \iff m \bmod \text{order } f a = n \bmod \text{order } f a$  (**is** *?P*  
 $\iff$  *?Q*)  
**proof**  
**assume** *?Q*  
**then have**  $(f \wedge^{(m \bmod \text{order } f a)}) \langle \$ \rangle a = (f \wedge^{(n \bmod \text{order } f a)}) \langle \$ \rangle a$   
**by** *simp*  
**then show** *?P*  
**by** *simp*  
**next**  
**assume** *?P*  
**then have**  $(f \wedge^{(m \bmod \text{order } f a)}) \langle \$ \rangle a = (f \wedge^{(n \bmod \text{order } f a)}) \langle \$ \rangle a$   
**by** *simp*  
**with** *inj\_on\_apply\_range*  
**show** *?Q*  
**by** (*rule inj\_onD*) *simp\_all*  
**qed**

**lemma** *apply\_inverse\_eq\_apply\_power\_order\_minus\_one*:  
 $(\text{inverse } f) \langle \$ \rangle a = (f \wedge^{(\text{order } f a - 1)}) \langle \$ \rangle a$   
**proof** (*cases order f a*)  
**case 0 with** *order\_greater\_zero [of f a]* **show** *?thesis*  
**by** *simp*  
**next**  
**case** (*Suc n*)  
**moreover have**  $(f \wedge^{\text{order } f a}) \langle \$ \rangle a = a$   
**by** *simp*  
**then have**  $*$ :  $(\text{inverse } f) \langle \$ \rangle ((f \wedge^{\text{order } f a}) \langle \$ \rangle a) = (\text{inverse } f) \langle \$ \rangle a$   
**by** *simp*  
**ultimately show** *?thesis*  
**by** (*simp add: apply\_sequence mult.assoc [symmetric]*)  
**qed**

**lemma** *apply\_inverse\_self\_in\_orbit* [*simp*]:  
 $(\text{inverse } f) \langle \$ \rangle a \in \text{orbit } f a$   
**using** *apply\_inverse\_eq\_apply\_power\_order\_minus\_one [symmetric]*  
**by** (*rule in\_orbitI*)

```

lemma apply_inverse_power_eq:
  (inverse (f ^ n)) ⟨$⟩ a = (f ^ (order f a - n mod order f a)) ⟨$⟩ a
proof (induct n)
  case 0 then show ?case by simp
next
  case (Suc n)
  define m where m = order f a - n mod order f a - 1
  moreover have order f a - n mod order f a > 0
    by simp
  ultimately have *: order f a - n mod order f a = Suc m
    by arith
  moreover from * have m2: order f a - Suc n mod order f a = (if m = 0 then
order f a else m)
    by (auto simp add: mod_Suc)
  ultimately show ?case
    using Suc
    by (simp_all add: apply_times power_Suc2 [of _ n] power_Suc [of _ m] del:
power_Suc)
      (simp add: apply_sequence mult.assoc [symmetric])
qed

```

```

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

```

```

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

```

```

lemma orbit_apply [simp]:

```



```

orbit f (f ⟨$⟩ a) = orbit f a
by (rule orbit_equiv) simp

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

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

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

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

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

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

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

```

```

by (simp add: order_def)

lemma orbit_disjoint:
  assumes orbit f a ≠ orbit f b
  shows orbit f a ∩ orbit f b = {}
proof (rule ccontr)
  assume orbit f a ∩ orbit f b ≠ {}
  then obtain c where c ∈ orbit f a ∩ orbit f b
    by blast
  then have c ∈ orbit f a and c ∈ orbit f b
    by auto
  then obtain m n where c = (f ^ m) ⟨$⟩ a
    and c = apply (f ^ n) b by (blast elim!: in_orbitE)
  then have (f ^ m) ⟨$⟩ a = apply (f ^ n) b
    by simp
  then have apply (inverse f ^ m) ((f ^ m) ⟨$⟩ a) =
    apply (inverse f ^ m) (apply (f ^ n) b)
    by simp
  then have *: apply (inverse f ^ m * f ^ n) b = a
    by (simp add: apply_sequence perm_power_inverse)
  have a ∈ orbit f b
  proof (cases n m rule: linorder_cases)
    case equal with * show ?thesis
      by (simp add: perm_power_inverse)
    next
    case less
      moreover define q where q = m - n
      ultimately have m = q + n by arith
      with * have apply (inverse f ^ q) b = a
        by (simp add: power_add mult.assoc perm_power_inverse)
      then have a ∈ orbit (inverse f) b
        by (rule in_orbitI)
      then show ?thesis
        by simp
    next
    case greater
      moreover define q where q = n - m
      ultimately have n = m + q by arith
      with * have apply (f ^ q) b = a
        by (simp add: power_add mult.assoc [symmetric] perm_power_inverse)
      then show ?thesis
        by (rule in_orbitI)
  qed
with assms show False
  by (auto dest: orbit_equiv)
qed

```

## 7.4 Swaps

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

**proof**

**fix** *a b* :: 'a

**have**  $\{c. \text{transpose } a b c \neq c\} \subseteq \{a, b\}$

**by** (*auto simp add: transpose\_def*)

**then show** *finite*  $\{c. \text{transpose } a b c \neq c\}$

**by** (*rule finite\_subset*) *simp*

**qed** *simp*

**lemma** *apply\_swap\_simp* [*simp*]:

$\langle a \leftrightarrow b \rangle \langle \$ \rangle a = b$

$\langle a \leftrightarrow b \rangle \langle \$ \rangle b = a$

**by** (*transfer; simp*)<sup>+</sup>

**lemma** *apply\_swap\_same* [*simp*]:

$c \neq a \implies c \neq b \implies \langle a \leftrightarrow b \rangle \langle \$ \rangle c = c$

**by** *transfer simp*

**lemma** *apply\_swap\_eq\_iff* [*simp*]:

$\langle a \leftrightarrow b \rangle \langle \$ \rangle c = a \iff c = b$

$\langle a \leftrightarrow b \rangle \langle \$ \rangle c = b \iff c = a$

**by** (*transfer; auto simp add: transpose\_def*)<sup>+</sup>

**lemma** *swap\_1* [*simp*]:

$\langle a \leftrightarrow a \rangle = 1$

**by** *transfer simp*

**lemma** *swap\_sym*:

$\langle b \leftrightarrow a \rangle = \langle a \leftrightarrow b \rangle$

**by** (*transfer; auto simp add: transpose\_def*)<sup>+</sup>

**lemma** *swap\_self* [*simp*]:

$\langle a \leftrightarrow b \rangle * \langle a \leftrightarrow b \rangle = 1$

**by** *transfer simp*

**lemma** *affected\_swap*:

$a \neq b \implies \text{affected } \langle a \leftrightarrow b \rangle = \{a, b\}$

**by** *transfer (auto simp add: transpose\_def)*

**lemma** *inverse\_swap* [*simp*]:

*inverse*  $\langle a \leftrightarrow b \rangle = \langle a \leftrightarrow b \rangle$

**by** *transfer (auto intro: inv\_equality)*

## 7.5 Permutations specified by cycles

**fun** *cycle* :: 'a list  $\Rightarrow$  'a perm ( $\langle \_ \rangle$ )

**where**

```

⟨[]⟩ = 1
| ⟨[a]⟩ = 1
| ⟨a # b # as⟩ = ⟨a # as⟩ * ⟨a↔b⟩

```

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

## 7.6 Syntax

```

bundle no_permutation_syntax
begin
  no_notation swap    ((_ ↔ _))
  no_notation cycle   ((_))
  no_notation apply (infixl ($) 999)
end

```

```

bundle permutation_syntax
begin
  notation swap      ((_ ↔ _))
  notation cycle     ((_))
  notation apply    (infixl ($) 999)
end

```

```

unbundle no_permutation_syntax

end

```

## 8 Permutation orbits

```

theory Orbits
imports
  HOL-Library.FuncSet
  HOL-Combinatorics.Permutations
begin

```

### 8.1 Orbits and cyclic permutations

```

inductive_set orbit :: ('a ⇒ 'a) ⇒ 'a ⇒ 'a set for f x where
  base: f x ∈ orbit f x |
  step: y ∈ orbit f x ⇒ f y ∈ orbit f x

```

```

definition cyclic_on :: ('a ⇒ 'a) ⇒ 'a set ⇒ bool where
  cyclic_on f S ↔ (∃ s∈S. S = orbit f s)

```

```

lemma orbit_altdef: orbit f x = {(f ^^ n) x | n. 0 < n} (is ?L = ?R)

```

```

proof (intro set_eqI iffI)

```

```

  fix y assume y ∈ ?L then show y ∈ ?R

```

```

    by (induct rule: orbit.induct) (auto simp: exI[where x=1] exI[where x=Suc
n for n])

```

```

next
  fix y assume y ∈ ?R
  then obtain n where y = (f ^^ n) x 0 < n by blast
  then show y ∈ ?L
  proof (induction n arbitrary: y)
    case (Suc n) then show ?case by (cases n = 0) (auto intro: orbit.intros)
  qed simp
qed

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

lemma orbit_subset:
  assumes s ∈ orbit f (f t) shows s ∈ orbit f t
  using assms by (induct) (auto intro: orbit.intros)

lemma orbit_sim_step:
  assumes s ∈ orbit f t shows f s ∈ orbit f (f t)
  using assms by induct (auto intro: orbit.intros)

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

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

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

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

lemma orbit_altdef_self_in:
  assumes s ∈ orbit f s shows orbit f s = {(f ^^ n) s | n. True}
proof (intro set_eqI iffI)
  fix x assume x ∈ {(f ^^ n) s | n. True}

```

**then obtain**  $n$  **where**  $x = (f \overset{\sim}{\sim} n) s$  **by** *auto*  
**then show**  $x \in \text{orbit } f s$  **using** *assms* **by** (*cases*  $n = 0$ ) (*auto simp: orbit\_altdef*)  
**qed** (*auto simp: orbit\_altdef*)

**lemma** *orbit\_altdef\_permutation*:

**assumes** *permutation f* **shows**  $\text{orbit } f s = \{(f \overset{\sim}{\sim} n) s \mid n. \text{True}\}$   
**using** *assms* **by** (*intro orbit\_altdef\_self\_in permutation\_self\_in\_orbit*)

**lemma** *orbit\_altdef\_bounded*:

**assumes**  $(f \overset{\sim}{\sim} n) s = s$   $0 < n$  **shows**  $\text{orbit } f s = \{(f \overset{\sim}{\sim} m) s \mid m. m < n\}$

**proof** –

**from** *assms* **have**  $s \in \text{orbit } f s$

**by** (*auto simp add: orbit\_altdef*) *metis*

**then have**  $\text{orbit } f s = \{(f \overset{\sim}{\sim} m) s \mid m. \text{True}\}$  **by** (*rule orbit\_altdef\_self\_in*)

**also have**  $\dots = \{(f \overset{\sim}{\sim} m) s \mid m. m < n\}$

**using** *assms*

**by** (*auto simp: funpow\_mod\_eq intro: exI[where  $x=m \bmod n$  for  $m$ ]*)

**finally show** *?thesis* .

**qed**

**lemma** *funpow\_in\_orbit*:

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

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

**lemma** *finite\_orbit*:

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

**proof** –

**from** *assms* **obtain**  $n$  **where**  $n: 0 < n$   $(f \overset{\sim}{\sim} n) s = s$

**by** (*auto simp: orbit\_altdef*)

**then show** *?thesis* **by** (*auto simp: orbit\_altdef\_bounded*)

**qed**

**lemma** *self\_in\_orbit\_step*:

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

**proof** (*intro set\_eqI iffI*)

**fix**  $t$  **assume**  $t \in \text{orbit } f s$  **then show**  $t \in \text{orbit } f (f s)$

**using** *assms* **by** (*auto intro: orbit\_step orbit\_sim\_step*)

**qed** (*auto intro: orbit\_subset*)

**lemma** *permutation\_orbit\_step*:

**assumes** *permutation f* **shows**  $\text{orbit } f (f s) = \text{orbit } f s$

**using** *assms* **by** (*intro self\_in\_orbit\_step permutation\_self\_in\_orbit*)

**lemma** *orbit\_nonempty*:

$\text{orbit } f s \neq \{\}$

**using** *orbit.base* **by** *fastforce*

**lemma** *orbit\_inv\_eq*:

**assumes** *permutation f*

```

shows orbit (inv f) x = orbit f x (is ?L = ?R)
proof -
{ fix g y assume A: permutation g y ∈ orbit (inv g) x
  have y ∈ orbit g x
  proof -
    have inv_g:  $\bigwedge y. x = g y \implies inv\ g\ x = y \bigwedge y. inv\ g\ (g\ y) = y$ 
      by (metis A(1) bij_inv_eq_iff permutation_bijective)+

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

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

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

lemma cyclic_on_alldef:
  cyclic_on f S  $\longleftrightarrow$  S  $\neq$  {}  $\wedge$  ( $\forall s \in S. S = orbit\ f\ s$ )
  unfolding cyclic_on_def by (auto intro: orbit.step orbit_swap orbit_trans)

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

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

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

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

lemma orbit_inverse:
  assumes self: a ∈ orbit g a
    and eq:  $\bigwedge x. x \in orbit\ g\ a \implies g'\ (f\ x) = f\ (g\ x)$ 
  shows f ' orbit g a = orbit g' (f a) (is ?L = ?R)

```

```

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

lemma cyclic_on_image:
  assumes cyclic_on f S
  assumes  $\bigwedge x. x \in S \implies g (h x) = h (f x)$ 
  shows cyclic_on g (h ` S)
  using assms by (auto simp: cyclic_on_def) (meson orbit_inverse)

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

lemma orbit_cong0:
  assumes x ∈ A f ∈ A → A  $\bigwedge y. y \in A \implies f y = g y$  shows orbit f x = orbit g
  x
proof –
  { fix n have (f  $\overset{\sim}{\sim}$  n) x = (g  $\overset{\sim}{\sim}$  n) x  $\wedge$  (f  $\overset{\sim}{\sim}$  n) x ∈ A
    by (induct n rule: nat.induct) (insert assms, auto)
  } then show ?thesis by (auto simp: orbit_altdef)
qed

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

```



```

using assms(1) _ assms(2) by (rule orbit_cong0) (auto simp: orbit.step eq)

lemma cyclic_cong:
  assumes  $\bigwedge s. s \in S \implies f s = g s$  shows  $\text{cyclic\_on } f S = \text{cyclic\_on } g S$ 
proof -
  have  $(\exists s \in S. \text{orbit } f s = \text{orbit } g s) \implies \text{cyclic\_on } f S = \text{cyclic\_on } g S$ 
    by (metis cyclic_on_alldef cyclic_on_def)
  then show ?thesis by (metis assms orbit_cong cyclic_on_def)
qed

lemma permutes_comp_preserves_cyclic1:
  assumes  $g \text{ permutes } B \text{ cyclic\_on } f C$ 
  assumes  $A \cap B = \{\}$   $C \subseteq A$ 
  shows  $\text{cyclic\_on } (f \circ g) C$ 
proof -
  have *:  $\bigwedge c. c \in C \implies f (g c) = f c$ 
    using assms by (subst permutes_not_in [of g]) auto
  with assms(2) show ?thesis by (simp cong: cyclic_cong)
qed

lemma permutes_comp_preserves_cyclic2:
  assumes  $f \text{ permutes } A \text{ cyclic\_on } g C$ 
  assumes  $A \cap B = \{\}$   $C \subseteq B$ 
  shows  $\text{cyclic\_on } (f \circ g) C$ 
proof -
  obtain  $c$  where  $c: c \in C \ C = \text{orbit } g c \ c \in \text{orbit } g c$ 
    using  $\langle \text{cyclic\_on } g C \rangle$  by (auto simp: cyclic_on_def)
  then have  $\bigwedge c. c \in C \implies f (g c) = g c$ 
    using assms  $c$  by (subst permutes_not_in [of f]) (auto intro: orbit.intros)
  with assms(2) show ?thesis by (simp cong: cyclic_cong)
qed

lemma permutes_orbit_subset:
  assumes  $f \text{ permutes } S \ x \in S$  shows  $\text{orbit } f x \subseteq S$ 
proof
  fix  $y$  assume  $y \in \text{orbit } f x$ 
  then show  $y \in S$  by induct (auto simp: permutes_in_image assms)
qed

lemma cyclic_on_orbit':
  assumes  $\text{permutation } f$  shows  $\text{cyclic\_on } f (\text{orbit } f x)$ 
  unfolding cyclic_on_alldef using orbit_nonempty[of f x]
  by (auto intro: assms orbit_swap orbit_trans permutation_self_in_orbit)

lemma cyclic_on_orbit:
  assumes  $f \text{ permutes } S \ \text{finite } S$  shows  $\text{cyclic\_on } f (\text{orbit } f x)$ 
  using assms by (intro cyclic_on_orbit') (auto simp: permutation_permutes)

lemma orbit_cyclic_eq3:

```

**assumes** *cyclic\_on*  $f$   $S$   $y \in S$  **shows**  $\text{orbit } f y = S$   
**using** *assms* **unfolding** *cyclic\_on\_alldef* **by** *simp*

**lemma** *orbit\_eq\_singleton\_iff*:  $\text{orbit } f x = \{x\} \longleftrightarrow f x = x$  (**is**  $?L \longleftrightarrow ?R$ )

**proof**

**assume**  $A: ?R$

{ **fix**  $y$  **assume**  $y \in \text{orbit } f x$  **then have**  $y = x$

**by** *induct* (*auto simp: A*)

} **then show**  $?L$  **by** (*metis orbit\_nonempty\_singletonI subsetI subset\_singletonD*)

**next**

**assume**  $A: ?L$

**then have**  $\bigwedge y. y \in \text{orbit } f x \implies f x = y$

**by** - (*erule orbit.cases, simp\_all*)

**then show**  $?R$  **using**  $A$  **by** *blast*

**qed**

**lemma** *eq\_on\_cyclic\_on\_iff1*:

**assumes** *cyclic\_on*  $f$   $S$   $x \in S$

**obtains**  $f x \in S$   $f x = x \longleftrightarrow \text{card } S = 1$

**proof**

**from** *assms* **show**  $f x \in S$  **by** (*auto simp: cyclic\_on\_def intro: orbit.intros*)

**from** *assms* **have**  $S = \text{orbit } f x$  **by** (*auto simp: cyclic\_on\_alldef*)

**then have**  $f x = x \longleftrightarrow S = \{x\}$  **by** (*metis orbit\_eq\_singleton\_iff*)

**then show**  $f x = x \longleftrightarrow \text{card } S = 1$  **using**  $\langle x \in S \rangle$  **by** (*auto simp: card\_Suc\_eq*)

**qed**

**lemma** *orbit\_eqI*:

$y = f x \implies y \in \text{orbit } f x$

$z = f y \implies y \in \text{orbit } f x \implies z \in \text{orbit } f x$

**by** (*metis orbit.base*) (*metis orbit.step*)

## 8.2 Decomposition of arbitrary permutations

**definition** *perm\_restrict* ::  $('a \Rightarrow 'a) \Rightarrow 'a \text{ set} \Rightarrow ('a \Rightarrow 'a)$  **where**

*perm\_restrict*  $f$   $S$   $x \equiv \text{if } x \in S \text{ then } f x \text{ else } x$

**lemma** *perm\_restrict\_comp*:

**assumes**  $A \cap B = \{\}$  *cyclic\_on*  $f$   $B$

**shows** *perm\_restrict*  $f$   $A$   $\circ$  *perm\_restrict*  $f$   $B = \text{perm_restrict } f$   $(A \cup B)$

**proof** -

**have**  $\bigwedge x. x \in B \implies f x \in B$  **using**  $\langle \text{cyclic\_on } f B \rangle$  **by** (*rule cyclic\_on\_inI*)

**with** *assms* **show**  $?thesis$  **by** (*auto simp: perm\_restrict\_def fun\_eq\_iff*)

**qed**

**lemma** *perm\_restrict\_simps*:

$x \in S \implies \text{perm\_restrict } f S x = f x$

$x \notin S \implies \text{perm\_restrict } f S x = x$

**by** (*auto simp: perm\_restrict\_def*)

```

lemma perm_restrict_perm_restrict:
  perm_restrict (perm_restrict f A) B = perm_restrict f (A ∩ B)
  by (auto simp: perm_restrict_def)

lemma perm_restrict_union:
  assumes perm_restrict f A permutes A perm_restrict f B permutes B A ∩ B =
  {}
  shows perm_restrict f A o perm_restrict f B = perm_restrict f (A ∪ B)
  using assms by (auto simp: fun_eq_iff perm_restrict_def permutes_def) (metis
  Diff_iff Diff_triv)

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

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

lemma perm_restrict_diff_cyclic:
  assumes f permutes S cyclic_on f A
  shows perm_restrict f (S - A) permutes (S - A)
proof -
  { fix y
    have  $\exists x. \text{perm\_restrict } f (S - A) x = y$ 
    proof cases
      assume A:  $y \in S - A$ 
      with  $\langle f \text{ permutes } S \rangle$  obtain x where  $f x = y \ x \in S$ 
      unfolding permutes_def by auto metis
      moreover
      with A have  $x \notin A$  by (metis Diff_iff assms(2) cyclic_on_inI)
      ultimately
      have  $\text{perm\_restrict } f (S - A) x = y$  by (simp add: perm_restrict_simps)
      then show ?thesis ..
    next
      assume  $y \notin S - A$ 
      then have  $\text{perm\_restrict } f (S - A) y = y$  by (simp add: perm_restrict_simps)
      then show ?thesis ..
    qed
  } note X = this

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

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

```

```

lemma permutes_decompose:
  assumes f permutes S finite S
  shows  $\exists C. (\forall c \in C. \text{cyclic\_on } f \ c) \wedge \bigcup C = S \wedge (\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\})$ 
  using assms(2,1)
proof (induction arbitrary: f rule: finite_psubset_induct)
  case (psubset S)

  show ?case
proof (cases S = \{\})
  case True then show ?thesis by (intro exI[where x=\{\}] auto)
next
  case False
  then obtain s where  $s \in S$  by auto
  with  $\langle f \text{ permutes } S \rangle$  have  $\text{orbit } f \ s \subseteq S$ 
  by (rule permutes_orbit_subset)
  have cyclic_orbit: cyclic_on f (orbit f s)
  using  $\langle f \text{ permutes } S \rangle \langle \text{finite } S \rangle$  by (rule cyclic_on_orbit)

  let  $?f' = \text{perm\_restrict } f \ (S - \text{orbit } f \ s)$ 

  have  $f \ s \in S$  using  $\langle f \text{ permutes } S \rangle \langle s \in S \rangle$  by (auto simp: permutes_in_image)
  then have  $S - \text{orbit } f \ s \subset S$  using orbit.base[of f s] \langle s \in S \rangle by blast
  moreover
  have  $?f' \text{ permutes } (S - \text{orbit } f \ s)$ 
  using  $\langle f \text{ permutes } S \rangle \text{cyclic\_orbit}$  by (rule perm_restrict_diff_cyclic)
  ultimately
  obtain C where  $C: \bigwedge c. c \in C \implies \text{cyclic\_on } ?f' \ c \ \bigcup C = S - \text{orbit } f \ s$ 
   $\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}$ 
  using psubset.IH by metis

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

  have Un_ins:  $\bigcup (\text{insert } (\text{orbit } f \ s) \ C) = S$ 
  using  $\langle \bigcup C = \_ \rangle \langle \text{orbit } f \ s \subseteq S \rangle$  by blast

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

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

```

qed

### 8.3 Function-power distance between values

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

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

**lemma**  $\text{funpow\_dist\_0}$ :  
**assumes**  $x = y$  **shows**  $\text{funpow\_dist } f \ x \ y = 0$   
**using**  $\text{assms}$  **unfolding**  $\text{funpow\_dist\_def}$  **by** ( $\text{intro } \text{Least\_eq\_0}$ )  $\text{simp}$

**lemma**  $\text{funpow\_dist\_least}$ :  
**assumes**  $n < \text{funpow\_dist } f \ x \ y$  **shows**  $(f \ \overset{\sim}{\sim} \ n) \ x \neq y$   
**proof** ( $\text{rule } \text{notI}$ )  
**assume**  $(f \ \overset{\sim}{\sim} \ n) \ x = y$   
**then have**  $\text{funpow\_dist } f \ x \ y \leq n$  **unfolding**  $\text{funpow\_dist\_def}$  **by** ( $\text{rule } \text{Least\_le}$ )  
**with**  $\text{assms}$  **show**  $\text{False}$  **by**  $\text{linarith}$

qed

**lemma**  $\text{funpow\_dist1\_least}$ :  
**assumes**  $0 < n \ n < \text{funpow\_dist1 } f \ x \ y$  **shows**  $(f \ \overset{\sim}{\sim} \ n) \ x \neq y$   
**proof** ( $\text{rule } \text{notI}$ )  
**assume**  $(f \ \overset{\sim}{\sim} \ n) \ x = y$   
**then have**  $(f \ \overset{\sim}{\sim} \ (n - 1)) \ (f \ x) = y$   
**using**  $\langle 0 < n \rangle$  **by** ( $\text{cases } n$ ) ( $\text{simp\_all } \text{add: } \text{funpow\_swap1}$ )  
**then have**  $\text{funpow\_dist } f \ (f \ x) \ y \leq n - 1$  **unfolding**  $\text{funpow\_dist\_def}$  **by** ( $\text{rule } \text{Least\_le}$ )  
**with**  $\text{assms}$  **show**  $\text{False}$  **by**  $\text{simp}$

qed

**lemma**  $\text{funpow\_dist\_prop}$ :  
 $y \in \text{orbit } f \ x \implies (f \ \overset{\sim}{\sim} \ \text{funpow\_dist } f \ x \ y) \ x = y$   
**unfolding**  $\text{funpow\_dist\_def}$  **by** ( $\text{rule } \text{LeastI\_ex}$ ) ( $\text{auto } \text{simp: } \text{orbit\_altdef}$ )

**lemma**  $\text{funpow\_dist\_0\_eq}$ :  
**assumes**  $y \in \text{orbit } f \ x$  **shows**  $\text{funpow\_dist } f \ x \ y = 0 \iff x = y$   
**using**  $\text{assms}$  **by** ( $\text{auto } \text{simp: } \text{funpow\_dist\_0 } \text{dest: } \text{funpow\_dist\_prop}$ )

**lemma**  $\text{funpow\_dist\_step}$ :  
**assumes**  $x \neq y \ y \in \text{orbit } f \ x$  **shows**  $\text{funpow\_dist } f \ x \ y = \text{Suc } (\text{funpow\_dist } f \ (f \ x) \ y)$   
**proof** –  
**from**  $\langle y \in \_ \rangle$  **obtain**  $n$  **where**  $(f \ \overset{\sim}{\sim} \ n) \ x = y$  **by** ( $\text{auto } \text{simp: } \text{orbit\_altdef}$ )  
**with**  $\langle x \neq y \rangle$  **obtain**  $n'$  **where**  $[\text{simp}]: n = \text{Suc } n'$  **by** ( $\text{cases } n$ )  $\text{auto}$   
  
**show**  $?thesis$

**unfolding** *funpow\_dist\_def*  
**proof** (*rule Least\_Suc2*)  
**show**  $(f \text{ ^^ } n) x = y$  **by** *fact*  
**then show**  $(f \text{ ^^ } n') (f x) = y$  **by** (*simp add: funpow\_swap1*)  
**show**  $(f \text{ ^^ } 0) x \neq y$  **using**  $\langle x \neq y \rangle$  **by** *simp*  
**show**  $\forall k. ((f \text{ ^^ } \text{Suc } k) x = y) = ((f \text{ ^^ } k) (f x) = y)$   
**by** (*simp add: funpow\_swap1*)  
**qed**  
**qed**

**lemma** *funpow\_dist1\_prop*:  
**assumes**  $y \in \text{orbit } f x$  **shows**  $(f \text{ ^^ } \text{funpow\_dist1 } f x y) x = y$   
**by** (*metis assms funpow\_dist\_prop funpow\_dist\_step funpow\_simps\_right(2)*)  
*o\_apply self\_in\_orbit\_step*

**lemma** *funpow\_neq\_less\_funpow\_dist*:  
**assumes**  $y \in \text{orbit } f x$   $m \leq \text{funpow\_dist } f x y$   $n \leq \text{funpow\_dist } f x y$   $m \neq n$   
**shows**  $(f \text{ ^^ } m) x \neq (f \text{ ^^ } n) x$   
**proof** (*rule notI*)  
**assume**  $A: (f \text{ ^^ } m) x = (f \text{ ^^ } n) x$

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

**have**  $y = (f \text{ ^^ } \text{funpow\_dist } f x y) x$   
**using**  $\langle y \in \_ \rangle$  **by** (*simp only: funpow\_dist\_prop*)  
**also have**  $\dots = (f \text{ ^^ } ((\text{funpow\_dist } f x y - n') + n')) x$   
**using**  $\langle n' \leq \_ \rangle$  **by** *simp*  
**also have**  $\dots = (f \text{ ^^ } ((\text{funpow\_dist } f x y - n') + m')) x$   
**by** (*simp add: funpow\_add (f ^^ m') x = \\_*)  
**also have**  $(f \text{ ^^ } ((\text{funpow\_dist } f x y - n') + m')) x \neq y$   
**using**  $A'$  **by** (*intro funpow\_dist\_least linarith*)  
**finally show** *False* **by** *simp*  
**qed**

**lemma** *funpow\_neq\_less\_funpow\_dist1*:  
**assumes**  $y \in \text{orbit } f x$   $m < \text{funpow\_dist1 } f x y$   $n < \text{funpow\_dist1 } f x y$   $m \neq n$   
**shows**  $(f \text{ ^^ } m) x \neq (f \text{ ^^ } n) x$   
**proof** (*rule notI*)  
**assume**  $A: (f \text{ ^^ } m) x = (f \text{ ^^ } n) x$   
  
**define**  $m' n'$  **where**  $m' = \min m n$  **and**  $n' = \max m n$   
**with**  $A$  **assms have**  $A': m' < n' (f \text{ ^^ } m') x = (f \text{ ^^ } n') x$   $n' < \text{funpow\_dist1 } f x y$   
*x y*  
**by** (*auto simp: min\_def max\_def*)

**have**  $y = (f \text{ ^^ } \text{funpow\_dist1 } f \ x \ y) \ x$   
**using**  $\langle y \in \_ \rangle$  **by** (*simp only: funpow\_dist1\_prop*)  
**also have**  $\dots = (f \text{ ^^ } ((\text{funpow\_dist1 } f \ x \ y - n') + n')) \ x$   
**using**  $\langle n' < \_ \rangle$  **by** *simp*  
**also have**  $\dots = (f \text{ ^^ } ((\text{funpow\_dist1 } f \ x \ y - n') + m')) \ x$   
**by** (*simp add: funpow\_add (f ^^ m') x = \_*)  
**also have**  $(f \text{ ^^ } ((\text{funpow\_dist1 } f \ x \ y - n') + m')) \ x \neq y$   
**using**  $A'$  **by** (*intro funpow\_dist1\_least linarith+*)  
**finally show** *False* **by** *simp*  
**qed**

**lemma** *inj\_on\_funpow\_dist*:  
**assumes**  $y \in \text{orbit } f \ x$  **shows** *inj\_on*  $(\lambda n. (f \text{ ^^ } n) \ x) \ \{0.. \text{funpow\_dist } f \ x \ y\}$   
**using** *funpow\_neq\_less\_funpow\_dist[OF assms]* **by** (*intro inj\_onI*) *auto*

**lemma** *inj\_on\_funpow\_dist1*:  
**assumes**  $y \in \text{orbit } f \ x$  **shows** *inj\_on*  $(\lambda n. (f \text{ ^^ } n) \ x) \ \{0.. < \text{funpow\_dist1 } f \ x \ y\}$   
**using** *funpow\_neq\_less\_funpow\_dist1[OF assms]* **by** (*intro inj\_onI*) *auto*

**lemma** *orbit\_conv\_funpow\_dist1*:  
**assumes**  $x \in \text{orbit } f \ x$   
**shows**  $\text{orbit } f \ x = (\lambda n. (f \text{ ^^ } n) \ x) \ \{0.. < \text{funpow\_dist1 } f \ x \ x\}$  (**is**  $?L = ?R$ )  
**using** *funpow\_dist1\_prop[OF assms]*  
**by** (*auto simp: orbit\_altdef\_bounded[where n=funpow\_dist1 f x x]*)

**lemma** *funpow\_dist1\_prop1*:  
**assumes**  $(f \text{ ^^ } n) \ x = y \ 0 < n$  **shows**  $(f \text{ ^^ } \text{funpow\_dist1 } f \ x \ y) \ x = y$   
**proof** –  
**from** *assms* **have**  $y \in \text{orbit } f \ x$  **by** (*auto simp: orbit\_altdef*)  
**then show**  $?thesis$  **by** (*rule funpow\_dist1\_prop*)  
**qed**

**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**  $\text{funpow\_dist1 } f \ x \ z = \text{funpow\_dist1 } f \ x \ y + \text{funpow\_dist1 } f \ y \ z$  (**is**  $?L = ?R$ )  
**proof** –  
**define**  $n$  **where**  $\langle n = \text{funpow\_dist1 } f \ x \ z - \text{funpow\_dist1 } f \ x \ y - 1 \rangle$   
**with** *assms* **have**  $*$ :  $\langle \text{funpow\_dist1 } f \ x \ z = \text{Suc } (\text{funpow\_dist1 } f \ x \ y + n) \rangle$   
**by** *simp*  
**have**  $x\_z$ :  $(f \text{ ^^ } \text{funpow\_dist1 } f \ x \ z) \ x = z$  **using** *assms* **by** (*blast intro: funpow\_dist1\_prop*)  
**have**  $x\_y$ :  $(f \text{ ^^ } \text{funpow\_dist1 } f \ x \ y) \ x = y$  **using** *assms* **by** (*blast intro: funpow\_dist1\_prop*)

**have**  $(f \text{ ^^ } (\text{funpow\_dist1 } f \ x \ z - \text{funpow\_dist1 } f \ x \ y)) \ y$   
 $= (f \text{ ^^ } (\text{funpow\_dist1 } f \ x \ z - \text{funpow\_dist1 } f \ x \ y)) \ ((f \text{ ^^ } \text{funpow\_dist1 } f \ x \ y) \ x)$

```

y) x)
  using x_y by simp
also have ... = z
  using assms x_z by (simp add: * funpow_add ac_simps funpow_swap1)
finally have y_z_diff: (f ^^ (funpow_dist1 f x z - funpow_dist1 f x y)) y = z .
then have (f ^^ funpow_dist1 f y z) y = z
  using assms by (intro funpow_dist1_prop1) auto
then have (f ^^ funpow_dist1 f y z) ((f ^^ funpow_dist1 f x y) x) = z
  using x_y by simp
then have (f ^^ (funpow_dist1 f y z + funpow_dist1 f x y)) x = z
  by (simp add: * funpow_add funpow_swap1)
show ?thesis
proof (rule antisym)
  from y_z_diff have (f ^^ funpow_dist1 f y z) y = z
    using assms by (intro funpow_dist1_prop1) auto
  then have (f ^^ funpow_dist1 f y z) ((f ^^ funpow_dist1 f x y) x) = z
    using x_y by simp
  then have (f ^^ (funpow_dist1 f y z + funpow_dist1 f x y)) x = z
    by (simp add: * funpow_add funpow_swap1)
  then have funpow_dist1 f x z ≤ funpow_dist1 f y z + funpow_dist1 f x y
    using funpow_dist1_least_not_less by fastforce
  then show ?L ≤ ?R by presburger
next
  have funpow_dist1 f y z ≤ funpow_dist1 f x z - funpow_dist1 f x y
    using y_z_diff assms(1) by (metis not_less zero_less_diff funpow_dist1_least)
  then show ?R ≤ ?L by linarith
qed
qed

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

```



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

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

end
```