

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

December 12, 2021

Contents

1	Transposition function	1
2	Stirling numbers of first and second kind	5
2.1	Stirling numbers of the second kind	5
2.2	Stirling numbers of the first kind	6
2.2.1	Efficient code	7
3	Permutations, both general and specifically on finite sets.	8
3.1	Auxiliary	8
3.2	Basic definition and consequences	8
3.3	Group properties	10
3.4	Mapping permutations with bijections	11
3.5	The number of permutations on a finite set	11
3.6	Hence a sort of induction principle composing by swaps . . .	12
3.7	Permutations of index set for iterated operations	12
3.8	Permutations as transposition sequences	12
3.9	Some closure properties of the set of permutations, with lengths	12
3.10	Various combinations of transpositions with 2, 1 and 0 com- mon elements	13
3.11	The identity map only has even transposition sequences . . .	13
3.12	Therefore we have a welldefined notion of parity	14
3.13	And it has the expected composition properties	14
3.14	A more abstract characterization of permutations	15
3.15	Relation to <i>permutes</i>	16
3.16	Sign of a permutation as a real number	16
3.17	Permuting a list	17
3.18	More lemmas about permutations	18
3.19	Sum over a set of permutations (could generalize to iteration)	20
3.20	Constructing permutations from association lists	20

4	Permuted Lists	23
4.1	An existing notion	23
4.2	Nontrivial conclusions	23
4.3	Trivial conclusions:	24
5	Permutations of a Multiset	25
5.1	Permutations of a multiset	26
5.2	Cardinality of permutations	27
5.3	Permutations of a set	28
5.4	Code generation	30
6	Cycles	32
6.1	Definitions	32
6.2	Basic Properties	33
6.3	Conjugation of cycles	33
6.4	When Cycles Commute	33
6.5	Cycles from Permutations	34
6.5.1	Exponentiation of permutations	34
6.5.2	Extraction of cycles from permutations	34
6.6	Decomposition on Cycles	35
6.6.1	Preliminaries	35
6.6.2	Decomposition	35
7	Permutations as abstract type	36
7.1	Abstract type of permutations	36
7.2	Identity, composition and inversion	37
7.3	Orbit and order of elements	39
7.4	Swaps	42
7.5	Permutations specified by cycles	43
7.6	Syntax	43
8	Permutation orbits	44
8.1	Orbits and cyclic permutations	44
8.2	Decomposition of arbitrary permutations	47
8.3	Function-power distance between values	48
9	Basic combinatorics in Isabelle/HOL (and the Archive of Formal Proofs)	50

1 Transposition function

```
theory Transposition
  imports Main
begin
```

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

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

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

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

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

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

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

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

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

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

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

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

lemma *transpose_image_eq* [*simp*]:
 $\langle \text{transpose } a \ b \ ' \ A = A \rangle \text{ if } \langle a \in A \longleftrightarrow b \in A \rangle$
 $\langle \text{proof} \rangle$

lemma *inj_on_transpose* [*simp*]:
 $\langle \text{inj_on } (\text{transpose } a \ b) \ A \rangle$
 $\langle \text{proof} \rangle$

lemma *inj_transpose*:
 $\langle \text{inj } (\text{transpose } a \ b) \rangle$
 $\langle \text{proof} \rangle$

lemma *surj_transpose*:
 $\langle \text{surj } (\text{transpose } a \ b) \rangle$
 $\langle \text{proof} \rangle$

lemma *bij_betw_transpose_iff* [*simp*]:
 $\langle \text{bij_betw } (\text{transpose } a \ b) \ A \ A \rangle \text{ if } \langle a \in A \longleftrightarrow b \in A \rangle$
 $\langle \text{proof} \rangle$

lemma *bij_transpose* [*simp*]:
 $\langle \text{bij } (\text{transpose } a \ b) \rangle$
 $\langle \text{proof} \rangle$

lemma *bijection_transpose*:
 $\langle \text{bijection } (\text{transpose } a \ b) \rangle$
 $\langle \text{proof} \rangle$

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

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

lemma *transpose_comp_eq*:
 $\langle \text{transpose } a \ b \circ f = f \circ \text{transpose } (\text{inv } f \ a) \ (\text{inv } f \ b) \rangle$
if $\langle \text{bij } f \rangle$
 $\langle \text{proof} \rangle$

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

Legacy input alias

$\langle ML \rangle$

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

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

$\langle ML \rangle$

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

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

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

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

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

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

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

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

lemma *inj_on_imp_inj_on_swap*: $inj_on\ f\ A \implies a \in A \implies b \in A \implies inj_on\ (Fun.swap\ a\ b\ f)\ A$
 $\langle proof \rangle$

lemma *inj_on_swap_iff*:
assumes $A: a \in A\ b \in A$
shows $inj_on\ (Fun.swap\ a\ b\ f)\ A \longleftrightarrow inj_on\ f\ A$
 $\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

end

2 Stirling numbers of first and second kind

theory *Stirling*
imports *Main*
begin

2.1 Stirling numbers of the second kind

fun *Stirling* :: nat $\Rightarrow$ nat $\Rightarrow$ nat

where

Stirling 0 0 = 1
| *Stirling* 0 (Suc k) = 0
| *Stirling* (Suc n) 0 = 0
| *Stirling* (Suc n) (Suc k) = Suc k * *Stirling* n (Suc k) + *Stirling* n k

lemma *Stirling_1* [simp]: *Stirling* (Suc n) (Suc 0) = 1
<proof>

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

lemma *Stirling_same* [simp]: *Stirling* n n = 1
<proof>

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

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

2.2 Stirling numbers of the first kind

fun *stirling* :: nat $\Rightarrow$ nat $\Rightarrow$ nat

where

stirling 0 0 = 1
| *stirling* 0 (Suc k) = 0
| *stirling* (Suc n) 0 = 0
| *stirling* (Suc n) (Suc k) = n * *stirling* n (Suc k) + *stirling* n k

lemma *stirling_0* [simp]: $n > 0 \implies \text{stirling } n \ 0 = 0$
<proof>

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

lemma *stirling_same* [simp]: *stirling* n n = 1
<proof>

lemma *stirling_Suc_n_1*: *stirling* (Suc n) (Suc 0) = fact n
<proof>

lemma *stirling_Suc_n_n*: *stirling* (Suc n) n = Suc n choose 2
<proof>

lemma *stirling_Suc_n_2*:
assumes $n \geq \text{Suc } 0$

shows $\text{stirling } (\text{Suc } n) \ 2 = (\sum k=1..n. \text{fact } n \ \text{div } k)$
 $\langle \text{proof} \rangle$

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

lemma sum_stirling : $(\sum k \leq n. \text{stirling } n \ k) = \text{fact } n$
 $\langle \text{proof} \rangle$

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

A row of the Stirling number triangle

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

lemma nth_stirling_row : $k \leq n \implies \text{stirling_row } n \ ! \ k = \text{stirling } n \ k$
 $\langle \text{proof} \rangle$

lemma $\text{length_stirling_row}$ [simp]: $\text{length } (\text{stirling_row } n) = \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma $\text{stirling_row_nonempty}$ [simp]: $\text{stirling_row } n \neq []$
 $\langle \text{proof} \rangle$

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 $\text{zip_with_prev} :: ('a \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'a \ \text{list} \Rightarrow 'b \ \text{list}$
where $\text{zip_with_prev } f \ x \ xs = \text{map2 } f \ (x \ \# \ xs) \ xs$

lemma $\text{zip_with_prev_altdef}$:
 $\text{zip_with_prev } f \ x \ xs =$
 $(\text{if } xs = [] \ \text{then } [] \ \text{else } f \ x \ (\text{hd } xs) \ \# \ [f \ (xs!i) \ (xs!(i+1)). \ i \leftarrow [0..<\text{length } xs - 1]])$
 $\langle \text{proof} \rangle$

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]
  <proof>

lemma stirling_row_code [code]:
  stirling_row 0 = [1]
  stirling_row (Suc n) = stirling_row_aux n 0 (stirling_row n)
  <proof>

lemma stirling_code [code]:
  stirling n k =
    (if k = 0 then (if n = 0 then 1 else 0)
     else if k > n then 0
     else if k = n then 1
     else stirling_row n ! k)
  <proof>

end

```

3 Permutations, both general and specifically on finite sets.

```

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

```

3.1 Auxiliary

```

abbreviation (input) fixpoints ::  $\langle ('a \Rightarrow 'a) \Rightarrow 'a\ set \rangle$ 
  where  $\langle fixpoints\ f \equiv \{x.\ f\ x = x\} \rangle$ 

```

```

lemma inj_on_fixpoints:
   $\langle inj\_on\ f\ (fixpoints\ f) \rangle$ 
  <proof>

```

```

lemma bij_betw_fixpoints:
   $\langle bij\_betw\ f\ (fixpoints\ f)\ (fixpoints\ f) \rangle$ 
  <proof>

```

3.2 Basic definition and consequences

```

definition permutes ::  $\langle ('a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow bool \rangle$  (infixr  $\langle permutes \rangle$  41)

```

where $\langle p \text{ permutes } S \longleftrightarrow (\forall x. x \notin S \longrightarrow p\ x = x) \wedge (\forall y. \exists!x. p\ x = y) \rangle$

lemma *bij_imp_permutes*:
 $\langle p \text{ permutes } S \rangle$ **if** $\langle \text{bij_betw } p\ S\ S \rangle$ **and** *stable*: $\langle \bigwedge x. x \notin S \implies p\ x = x \rangle$
 $\langle \text{proof} \rangle$

context
fixes $p :: \langle 'a \Rightarrow 'a \rangle$ **and** $S :: \langle 'a \text{ set} \rangle$
assumes *perm*: $\langle p \text{ permutes } S \rangle$
begin

lemma *permutes_inj*:
 $\langle \text{inj } p \rangle$
 $\langle \text{proof} \rangle$

lemma *permutes_image*:
 $\langle p\ ` S = S \rangle$
 $\langle \text{proof} \rangle$

lemma *permutes_not_in*:
 $\langle x \notin S \implies p\ x = x \rangle$
 $\langle \text{proof} \rangle$

lemma *permutes_image_complement*:
 $\langle p\ ` (- S) = - S \rangle$
 $\langle \text{proof} \rangle$

lemma *permutes_in_image*:
 $\langle p\ x \in S \longleftrightarrow x \in S \rangle$
 $\langle \text{proof} \rangle$

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

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

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

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

```

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

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

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

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

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

end

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

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

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

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

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

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

lemma permutes_bij_inv_into:
  fixes  $A :: 'a \text{ set}$ 
  and  $B :: 'b \text{ set}$ 
  assumes  $p \text{ permutes } A$ 
  and  $\text{bij\_betw } f \ A \ B$ 

```

shows $(\lambda x. \text{if } x \in B \text{ then } f (p (\text{inv_into } A f x)) \text{ else } x) \text{ permutes } B$
 $\langle \text{proof} \rangle$

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

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

3.3 Group properties

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

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

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

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

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

3.4 Mapping permutations with bijections

lemma *bij_betw_permutations*:
assumes $\text{bij_betw } f A B$
shows $\text{bij_betw } (\lambda \pi x. \text{if } x \in B \text{ then } f (\pi (\text{inv_into } A f x)) \text{ else } x)$
 $\{\pi. \pi \text{ permutes } A\} \{\pi. \pi \text{ permutes } B\} (\text{is } \text{bij_betw } ?f _ _)$
 $\langle \text{proof} \rangle$

lemma *bij_betw_derangements*:
assumes $\text{bij_betw } f A B$
shows $\text{bij_betw } (\lambda \pi x. \text{if } x \in B \text{ then } f (\pi (\text{inv_into } A f x)) \text{ else } x)$

$\{\pi. \pi \text{ permutes } A \wedge (\forall x \in A. \pi x \neq x)\} \{\pi. \pi \text{ permutes } B \wedge (\forall x \in B. \pi x \neq x)\}$
 $(\text{is_bij_betw } ?f _ _)$
 $\langle \text{proof} \rangle$

3.5 The number of permutations on a finite set

lemma *permutes_insert_lemma*:
assumes $p \text{ permutes } (\text{insert } a \ S)$
shows $\text{transpose } a \ (p \ a) \circ p \text{ permutes } S$
 $\langle \text{proof} \rangle$

lemma *permutes_insert*: $\{p. p \text{ permutes } (\text{insert } a \ S)\} =$
 $(\lambda(b, p). \text{transpose } a \ b \circ p) \cdot \{(b, p). b \in \text{insert } a \ S \wedge p \in \{p. p \text{ permutes } S\}\}$
 $\langle \text{proof} \rangle$

lemma *card_permutations*:
assumes $\text{card } S = n$
and $\text{finite } S$
shows $\text{card } \{p. p \text{ permutes } S\} = \text{fact } n$
 $\langle \text{proof} \rangle$

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

3.6 Hence a sort of induction principle composing by swaps

lemma *permutes_induct* [*consumes 2, case_names id swap*]:
 $\langle P \ p \rangle$ **if** $\langle p \text{ permutes } S \rangle \langle \text{finite } S \rangle$
and *id*: $\langle P \ \text{id} \rangle$
and *swap*: $\langle \bigwedge a \ b \ p. a \in S \implies b \in S \implies p \text{ permutes } S \implies P \ p \implies P \ (\text{transpose } a \ b \circ p) \rangle$
 $\langle \text{proof} \rangle$

lemma *permutes_rev_induct* [*consumes 2, case_names id swap*]:
 $\langle P \ p \rangle$ **if** $\langle p \text{ permutes } S \rangle \langle \text{finite } S \rangle$
and *id'*: $\langle P \ \text{id} \rangle$
and *swap'*: $\langle \bigwedge a \ b \ p. a \in S \implies b \in S \implies p \text{ permutes } S \implies P \ p \implies P \ (p \circ \text{transpose } a \ b) \rangle$
 $\langle \text{proof} \rangle$

3.7 Permutations of index set for iterated operations

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

3.8 Permutations as transposition sequences

inductive *swapidseq* :: *nat* $\Rightarrow$ (*'a* $\Rightarrow$ *'a*) $\Rightarrow$ *bool*

where

id[*simp*]: *swapidseq* 0 *id*

| *comp_Suc*: *swapidseq* *n p* $\implies$ *a* $\neq$ *b* $\implies$ *swapidseq* (*Suc* *n*) (*transpose* *a b* $\circ$ *p*)

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

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

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

lemma *permutation_id*[*simp*]: *permutation id*

<proof>

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

lemma *swapidseq_swap*: *swapidseq* (*if* *a* = *b* *then* 0 *else* 1) (*transpose* *a b*)

<proof>

lemma *permutation_swap_id*: *permutation* (*transpose* *a b*)

<proof>

lemma *swapidseq_comp_add*: *swapidseq* *n p* $\implies$ *swapidseq* *m q* $\implies$ *swapidseq* (*n* + *m*) (*p* $\circ$ *q*)

<proof>

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

<proof>

lemma *swapidseq_endswap*: *swapidseq* *n p* $\implies$ *a* $\neq$ *b* $\implies$ *swapidseq* (*Suc* *n*) (*p* $\circ$ *transpose* *a b*)

<proof>

lemma *swapidseq_inverse_exists*: *swapidseq* *n p* $\implies$ $\exists$ *q*. *swapidseq* *n q* $\wedge$ *p* $\circ$ *q* = *id* $\wedge$ *q* $\circ$ *p* = *id*

<proof>

lemma *swapidseq_inverse*:

assumes *swapidseq* *n p*

shows *swapidseq* *n* (*inv* *p*)

<proof>

lemma *permutation_inverse*: *permutation* *p* $\implies$ *permutation* (*inv* *p*)

<proof>

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

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

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

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

3.11 The identity map only has even transposition sequences

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

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

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

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

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

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

$\langle \text{proof} \rangle$

3.12 Therefore we have a welldefined notion of parity

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

lemma $\text{swapidseq_even_even}$:

assumes m : $\text{swapidseq } m \ p$

and n : $\text{swapidseq } n \ p$

shows $\text{even } m \longleftrightarrow \text{even } n$

$\langle \text{proof} \rangle$

lemma evenperm_unique :

assumes p : $\text{swapidseq } n \ p$

and n : $\text{even } n = b$

shows $\text{evenperm } p = b$

$\langle \text{proof} \rangle$

3.13 And it has the expected composition properties

lemma $\text{evenperm_id}[simp]$: $\text{evenperm } \text{id} = \text{True}$

$\langle \text{proof} \rangle$

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

$\langle \text{evenperm } (\lambda x. x) \rangle$

$\langle \text{proof} \rangle$

lemma evenperm_swap : $\text{evenperm } (\text{transpose } a \ b) = (a = b)$

$\langle \text{proof} \rangle$

lemma evenperm_comp :

assumes $\text{permutation } p \ \text{permutation } q$

shows $\text{evenperm } (p \circ q) \longleftrightarrow \text{evenperm } p = \text{evenperm } q$

$\langle \text{proof} \rangle$

lemma evenperm_inv :

assumes $\text{permutation } p$

shows $\text{evenperm } (\text{inv } p) = \text{evenperm } p$

$\langle \text{proof} \rangle$

3.14 A more abstract characterization of permutations

lemma $\text{permutation_bijective}$:

assumes $\text{permutation } p$

shows $\text{bij } p$

$\langle \text{proof} \rangle$

lemma $\text{permutation_finite_support}$:

assumes $\text{permutation } p$

shows $\text{finite } \{x. p \ x \neq x\}$

⟨proof⟩

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

lemma *permutation*: *permutation p* $\longleftrightarrow$ *bij p* $\wedge$ *finite {x. p x $\neq$ x}*
 (**is** *?lhs* $\longleftrightarrow$ *?b* $\wedge$ *?f*)
 ⟨proof⟩

lemma *permutation_inverse_works*:
 assumes *permutation p*
 shows *inv p* $\circ$ *p* = *id*
 and *p* $\circ$ *inv p* = *id*
 ⟨proof⟩

lemma *permutation_inverse_compose*:
 assumes *p*: *permutation p*
 and *q*: *permutation q*
 shows *inv (p* $\circ$ *q)* = *inv q* $\circ$ *inv p*
 ⟨proof⟩

3.15 Relation to *permutes*

lemma *permutes_imp_permutation*:
 ⟨*permutation p*⟩ **if** ⟨*finite S*⟩ ⟨*p permutes S*⟩
 ⟨proof⟩

lemma *permutation_permutesE*:
 assumes ⟨*permutation p*⟩
 obtains *S* **where** ⟨*finite S*⟩ ⟨*p permutes S*⟩
 ⟨proof⟩

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

3.16 Sign of a permutation as a real number

definition *sign* :: $\langle 'a \Rightarrow 'a \rangle \Rightarrow \text{int}$ — 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 $\langle \text{sign } p = 1 \rangle \mid \langle \text{sign } p = -1 \rangle$
 ⟨proof⟩

lemma *sign_nz* [*simp*]: *sign p* $\neq 0$
 ⟨proof⟩

lemma *sign_id* [simp]: *sign id = 1*
 ⟨proof⟩

lemma *sign_identity* [simp]:
 ⟨*sign* ($\lambda x. x$) = 1⟩
 ⟨proof⟩

lemma *sign_inverse*: *permutation p $\implies$ sign (inv p) = sign p*
 ⟨proof⟩

lemma *sign_compose*: *permutation p $\implies$ permutation q $\implies$ sign (p $\circ$ q) = sign p * sign q*
 ⟨proof⟩

lemma *sign_swap_id*: *sign (transpose a b) = (if a = b then 1 else - 1)*
 ⟨proof⟩

lemma *sign_idempotent* [simp]: *sign p * sign p = 1*
 ⟨proof⟩

lemma *sign_left_idempotent* [simp]:
 ⟨*sign p* * (*sign p* * *sign q*) = *sign q*⟩
 ⟨proof⟩

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

3.17 Permuting a list

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

definition *permute_list* :: (*nat* $\Rightarrow$ *nat*) $\Rightarrow$ '*a list* $\Rightarrow$ '*a list*
where *permute_list* *f xs* = *map* ($\lambda i. xs ! (f i)$) [0..*length xs*]

lemma *permute_list_map*:
assumes *f* *permutes* {..*length xs*}
shows *permute_list f* (*map g xs*) = *map g* (*permute_list f xs*)
 ⟨proof⟩

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

lemma *permute_list_Nil* [simp]: *permute_list f [] = []*
 ⟨proof⟩

lemma *length_permute_list* [simp]: *length* (*permute_list f xs*) = *length xs*
 ⟨proof⟩

lemma *permute_list_compose*:
 assumes g permutes $\{.. $\text{length } xs$ \}$
 shows $\text{permute_list } (f \circ g) \text{ } xs = \text{permute_list } g (\text{permute_list } f \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *permute_list_ident* [simp]: $\text{permute_list } (\lambda x. x) \text{ } xs = xs$
 $\langle \text{proof} \rangle$

lemma *permute_list_id* [simp]: $\text{permute_list } \text{id} \text{ } xs = xs$
 $\langle \text{proof} \rangle$

lemma *mset_permute_list* [simp]:
 fixes $xs :: 'a \text{ list}$
 assumes f permutes $\{.. $\text{length } xs$ \}$
 shows $\text{mset } (\text{permute_list } f \text{ } xs) = \text{mset } xs$
 $\langle \text{proof} \rangle$

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

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

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

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

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

3.18 More lemmas about permutations

lemma *permutes_in_funpow_image*:
 assumes f permutes S $x \in S$
 shows $(f \text{ } \frown n) \text{ } x \in S$

$\langle \text{proof} \rangle$

lemma *permutation_self*:
 assumes $\langle \text{permutation } p \rangle$
 obtains n **where** $\langle n > 0 \rangle \langle (p \text{ } ^{\sim} n) x = x \rangle$
 $\langle \text{proof} \rangle$

The following few lemmas were contributed by Lukas Bulwahn.

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

lemma *image_mset_eq_implies_permutes*:
 fixes $f :: 'a \Rightarrow 'b$
 assumes *finite A*
 and $\text{mset_eq: image_mset } f \ (\text{mset_set } A) = \text{image_mset } f' \ (\text{mset_set } A)$
 obtains p **where** p *permutes A* **and** $\forall x \in A. f \ x = f' \ (p \ x)$
 $\langle \text{proof} \rangle$

lemma *mset_eq_permutation*:
 fixes $xs \ ys :: 'a \text{ list}$
 assumes $\text{mset_eq: mset } xs = \text{mset } ys$
 obtains p **where** p *permutes* $\{..<\text{length } ys\}$ $\text{permute_list } p \ ys = xs$
 $\langle \text{proof} \rangle$

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

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

lemma *image_inverse_permutations*: $\{\text{inv } p \mid p. p \text{ permutes } S\} = \{p. p \text{ permutes } S\}$
 $\langle \text{proof} \rangle$

lemma *image_compose_permutations_left*:
 assumes q *permutes S*
 shows $\{q \circ p \mid p. p \text{ permutes } S\} = \{p. p \text{ permutes } S\}$
 $\langle \text{proof} \rangle$

lemma *image_compose_permutations_right*:
 assumes q *permutes S*

```

shows  $\{p \circ q \mid p. p \text{ permutes } S\} = \{p \cdot p \text{ permutes } S\}$ 
  ⟨proof⟩

lemma permutes_in_seg:  $p \text{ permutes } \{1 \dots n\} \implies i \in \{1 \dots n\} \implies 1 \leq p \ i \wedge p \ i \leq n$ 
  ⟨proof⟩

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⟩

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⟩

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⟩

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

lemma permutes_pair_eq:
  ⟨ $\{(p \ s, s) \mid s. s \in S\} = \{(s, \text{inv } p \ s) \mid s. s \in S\}$ ⟩ (is ⟨?L = ?R⟩) if ⟨ $p \text{ permutes } S$ ⟩
  ⟨proof⟩

context
  fixes  $p$  and  $n \ i :: \text{nat}$ 
  assumes  $p: \langle p \text{ permutes } \{0 \dots n\} \rangle$  and  $i: \langle i < n \rangle$ 
begin

lemma permutes_nat_less:
  ⟨ $p \ i < n$ ⟩
  ⟨proof⟩

lemma permutes_nat_inv_less:
  ⟨ $\text{inv } p \ i < n$ ⟩
  ⟨proof⟩

end

context comm_monoid_set
begin

```

```

lemma permutes_inv:
   $\langle F (\lambda s. g (p\ s)\ s)\ S = F (\lambda s. g\ s\ (inv\ p\ s))\ S \rangle$  (is  $\langle ?l = ?r \rangle$ )
  if  $\langle p\ permutes\ S \rangle$ 
   $\langle proof \rangle$ 

end

```

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

```

lemma sum_over_permutations_insert:
  assumes  $fS$ : finite  $S$ 
  and  $aS$ :  $a \notin S$ 
  shows  $sum\ f\ \{p. p\ permutes\ (insert\ a\ S)\} =$ 
     $sum\ (\lambda b. sum\ (\lambda q. f\ (transpose\ a\ b\ \circ\ q))\ \{p. p\ permutes\ S\})\ (insert\ a\ S)$ 
   $\langle proof \rangle$ 

```

3.20 Constructing permutations from association lists

```

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

```

```

lemma list_permutesI [simp]:
  assumes  $set\ (map\ fst\ xs) \subseteq A$   $set\ (map\ snd\ xs) = set\ (map\ fst\ xs)$   $distinct\ (map\ fst\ xs)$ 
  shows list_permutes  $xs\ A$ 
   $\langle proof \rangle$ 

```

```

definition permutation_of_list ::  $('a \times 'a)\ list \Rightarrow 'a \Rightarrow 'a$ 
  where permutation_of_list  $xs\ x = (case\ map\_of\ xs\ x\ of\ None \Rightarrow x \mid Some\ y \Rightarrow y)$ 

```

```

lemma permutation_of_list_Cons:
  permutation_of_list  $((x, y) \# xs)\ x' = (if\ x = x'\ then\ y\ else\ permutation\_of\_list\ xs\ x')$ 
   $\langle proof \rangle$ 

```

```

fun inverse_permutation_of_list ::  $('a \times 'a)\ list \Rightarrow 'a \Rightarrow 'a$ 
  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*⟩

lemma *inj_on_the*: $\text{None} \notin A \implies \text{inj_on the } A$
 ⟨*proof*⟩

lemma *inj_on_map_of'*:
 assumes *distinct* (*map snd xs*)
 shows *inj_on* (*the* $\circ$ *map_of xs*) (*set* (*map fst xs*))
 ⟨*proof*⟩

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

lemma *the_Some_image* [*simp*]: *the* ' *Some* ' *A* = *A*
 ⟨*proof*⟩

lemma *image_map_of'*:
 assumes *distinct* (*map fst xs*)
 shows (*the* $\circ$ *map_of xs*) ' *set* (*map fst xs*) = *set* (*map snd xs*)
 ⟨*proof*⟩

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

lemma *eval_permutation_of_list* [*simp*]:
permutation_of_list [] $x = x$
 $x = x' \implies \text{permutation_of_list } ((x', y) \# xs) \ x = y$
 $x \neq x' \implies \text{permutation_of_list } ((x', y') \# xs) \ x = \text{permutation_of_list } xs \ x$
 ⟨*proof*⟩

lemma *eval_inverse_permutation_of_list* [*simp*]:
inverse_permutation_of_list [] $x = x$
 $x = x' \implies \text{inverse_permutation_of_list } ((y, x') \# xs) \ x = y$
 $x \neq x' \implies \text{inverse_permutation_of_list } ((y', x') \# xs) \ x = \text{inverse_permutation_of_list } xs \ x$
 ⟨*proof*⟩

lemma *permutation_of_list_id*: $x \notin \text{set } (\text{map fst } xs) \implies \text{permutation_of_list } xs$
 $x = x$
 ⟨*proof*⟩

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

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

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

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

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

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>

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*) $\text{perm} :: \langle 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \rangle$ (**infixr** $\langle \sim \sim \rangle$ 50)
 where $\langle xs \sim \sim \rangle ys \equiv \text{mset } xs = \text{mset } ys$

4.2 Nontrivial conclusions

proposition *perm_swap*:
 $\langle xs[i := xs ! j, j := xs ! i] \sim \sim \rangle xs$

if $\langle i < \text{length } xs \rangle \langle j < \text{length } xs \rangle$
 $\langle \text{proof} \rangle$

proposition *mset_le_perm_append*: $\text{mset } xs \subseteq \# \text{ mset } ys \longleftrightarrow (\exists zs. xs @ zs <^{\sim\sim} ys)$
 $\langle \text{proof} \rangle$

proposition *perm_set_eq*: $xs <^{\sim\sim} ys \implies \text{set } xs = \text{set } ys$
 $\langle \text{proof} \rangle$

proposition *perm_distinct_iff*: $xs <^{\sim\sim} ys \implies \text{distinct } xs \longleftrightarrow \text{distinct } ys$
 $\langle \text{proof} \rangle$

theorem *eq_set_perm_remdups*: $\text{set } xs = \text{set } ys \implies \text{remdups } xs <^{\sim\sim} \text{remdups } ys$
 $\langle \text{proof} \rangle$

proposition *perm_remdups_iff_eq_set*: $\text{remdups } x <^{\sim\sim} \text{remdups } y \longleftrightarrow \text{set } x = \text{set } y$
 $\langle \text{proof} \rangle$

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

proposition *perm_finite*: $\text{finite } \{B. B <^{\sim\sim} A\}$
 $\langle \text{proof} \rangle$

4.3 Trivial conclusions:

proposition *perm_empty_imp*: $[] <^{\sim\sim} ys \implies ys = []$
 $\langle \text{proof} \rangle$

This more general theorem is easier to understand!

proposition *perm_length*: $xs <^{\sim\sim} ys \implies \text{length } xs = \text{length } ys$
 $\langle \text{proof} \rangle$

proposition *perm_sym*: $xs <^{\sim\sim} ys \implies ys <^{\sim\sim} xs$
 $\langle \text{proof} \rangle$

We can insert the head anywhere in the list.

proposition *perm_append_Cons*: $a \# xs @ ys <^{\sim\sim} xs @ a \# ys$
 $\langle \text{proof} \rangle$

proposition *perm_append_swap*: $xs @ ys <^{\sim\sim} ys @ xs$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

Congruence rule

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

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

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

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

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

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

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

end

5 Permutations of a Multiset

theory *Multiset_Permutations*

imports

Complex_Main

Permutations

begin

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

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

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

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

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

5.1 Permutations of a multiset

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

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

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

lemma *permutations_of_multiset_Cons_iff*:
 $x \# xs \in \text{permutations_of_multiset } A \iff x \in \# A \wedge xs \in \text{permutations_of_multiset } (A - \{\#x\})$
 ⟨proof⟩

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

lemma *permutations_of_multiset_nonempty*:
 assumes *nonempty*: $A \neq \{\#\}$
 shows $\text{permutations_of_multiset } A = (\bigcup_{x \in \text{set_mset } A} ((\#) x) \text{ ‘ permutations_of_multiset } (A - \{\#x\}))$
 (is $_ = ?rhs$)
 ⟨proof⟩

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

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

lemma *rev_permutations_of_multiset* [simp]:
 $\text{rev ‘ permutations_of_multiset } A = \text{permutations_of_multiset } A$
 ⟨proof⟩

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

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

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

lemma *permutations_of_multiset_not_empty* [simp]: $\text{permutations_of_multiset } A \neq \{\#\}$
 ⟨proof⟩

lemma *permutations_of_multiset_image*:
 $\text{permutations_of_multiset } (\text{image_mset } f A) = \text{map } f \text{ ‘ permutations_of_multiset } A$
 ⟨proof⟩

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> **lemma** *multiset_prod_fact_remove*:

$$x \in \# \ A \implies \left(\prod_{y \in \text{set_mset } A}. \text{fact } (\text{count } A \ y) \right) =$$

$$\text{count } A \ x * \left(\prod_{y \in \text{set_mset } (A - \{\#x\})}. \text{fact } (\text{count } (A - \{\#x\}) \ y) \right)$$

<proof>

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>

theorem *card_permutations_of_multiset*:

$$\text{card } (\text{permutations_of_multiset } A) = \text{fact } (\text{size } A) \text{ div } \left(\prod_{x \in \text{set_mset } A}. \text{fact } (\text{count } A \ x) \right)$$

$$\left(\prod_{x \in \text{set_mset } A}. \text{fact } (\text{count } A \ x) :: \text{nat} \right) \text{ dvd } \text{fact } (\text{size } A)$$

<proof>

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>

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>

lemma *real_card_permutations_of_multiset_remove*:

assumes $x \in \# \ A$

shows $\text{real } (\text{card } (\text{permutations_of_multiset } (A - \{\#x\}))) =$

$$\text{real } (\text{card } (\text{permutations_of_multiset } A) * \text{count } A \ x) / \text{real } (\text{size } A)$$

<proof>

lemma *real_card_permutations_of_multiset_remove'*:

assumes $x \in \# \ A$

shows $\text{real } (\text{card } (\text{permutations_of_multiset } A)) =$

$$\text{real } (\text{size } A * \text{card } (\text{permutations_of_multiset } (A - \{\#x\}))) / \text{real } (\text{count } A \ x)$$

$\langle \text{proof} \rangle$

end

5.3 Permutations of a set

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

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

lemma *permutations_of_setI* [intro]:
assumes set xs = A distinct xs
shows xs $\in$ *permutations_of_set* A
 $\langle \text{proof} \rangle$

lemma *permutations_of_setD*:
assumes xs $\in$ *permutations_of_set* A
shows set xs = A distinct xs
 $\langle \text{proof} \rangle$

lemma *permutations_of_set_lists*: *permutations_of_set* A $\subseteq$ lists A
 $\langle \text{proof} \rangle$

lemma *permutations_of_set_empty* [simp]: *permutations_of_set* {} = {}
 $\langle \text{proof} \rangle$

lemma *UN_set_permutations_of_set* [simp]:
finite A $\implies$ ($\bigcup$ xs $\in$ *permutations_of_set* A. set xs) = A
 $\langle \text{proof} \rangle$

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

lemma *permutations_of_set_nonempty*:
A $\neq$ {} $\implies$ *permutations_of_set* A =
($\bigcup$ x $\in$ A. (λ xs. x $\#$ xs) ' *permutations_of_set* (A - {x}))
 $\langle \text{proof} \rangle$

lemma *permutations_of_set_singleton* [simp]: *permutations_of_set* {x} = {[x]}
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *permutations_of_set_empty_iff* [simp]:
 $\text{permutations_of_set } A = \{\} \longleftrightarrow \neg \text{finite } A$
 ⟨proof⟩

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

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

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

5.4 Code generation

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

declare *length_remove1* [termination_simp]

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

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

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

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

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

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

lemma *distinct_permutations_of_list_impl*:
 $\text{distinct } (\text{permutations_of_list_impl } xs)$
 <proof>

lemma *permutations_of_list_impl_aux_correct'*:
 $\text{permutations_of_list_impl_aux } acc \ xs =$
 $\text{map } (\lambda xs. \text{rev } xs \ @ \ acc) (\text{permutations_of_list_impl } xs)$
 <proof>

lemma *permutations_of_list_impl_aux_correct*:
 $\text{permutations_of_list_impl_aux } [] \ xs = \text{map rev } (\text{permutations_of_list_impl } xs)$
 <proof>

lemma *distinct_permutations_of_list_impl_aux*:
 $\text{distinct } (\text{permutations_of_list_impl_aux } acc \ xs)$
 <proof>

lemma *set_permutations_of_list_impl_aux*:
 $\text{set } (\text{permutations_of_list_impl_aux } [] \ xs) = \text{permutations_of_multiset } (\text{mset } xs)$
 <proof>

declare *set_permutations_of_list_impl_aux* [symmetric, code]

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

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

function *permutations_of_set_aux* **where**
 $\text{permutations_of_set_aux } acc \ A =$
 $(\text{if } \neg \text{finite } A \text{ then } \{\} \text{ else if } A = \{\} \text{ then } \{acc\} \text{ else}$
 $(\bigcup_{x \in A. \text{permutations_of_set_aux } (x \# acc) (A - \{x\}))$
 <proof>

termination <proof>

lemma *permutations_of_set_aux_altdef*:
 $\text{permutations_of_set_aux } acc \ A = (\lambda xs. \text{rev } xs \ @ \ acc) ' \text{permutations_of_set } A$
 <proof>

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

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

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

declare *length_remove1* [*termination_simp*]

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

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

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

lemma *permutations_of_set_aux_list_refine*:
assumes *distinct xs*
shows *set (permutations_of_set_aux_list acc xs)* = *permutations_of_set_aux acc (set xs)*
 ⟨*proof*⟩

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

lemma *distinct_permutations_of_set_aux_list*:
distinct xs ⇒ *distinct (permutations_of_set_aux_list acc xs)*
 ⟨*proof*⟩

lemma *distinct_permutations_of_set_list*:
distinct xs ⇒ *distinct (permutations_of_set_list xs)*
 ⟨*proof*⟩

lemma *permutations_of_set_list*:
permutations_of_set (set xs) = *set (permutations_of_set_list (remdups xs))*
 ⟨*proof*⟩

lemma *permutations_of_list_code* [*code*]:
permutations_of_set (set xs) = *set (permutations_of_set_list (remdups xs))*
permutations_of_set (List.coset xs) =
Code.abort (STR "Permutation of set complement not supported")
 (λ_. *permutations_of_set (List.coset xs)*)

```

    <proof>

value [code] permutations_of_set (set "abcd")

end

```

```

theory Cycles
  imports
    HOL-Library.FuncSet
    Permutations
  begin

```

6 Cycles

6.1 Definitions

```

abbreviation 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)  $x = x$ 
  <proof>

```

```

lemma permutation_of_cycle: permutation (cycle_of_list cs)
  <proof>

```

```

lemma cycle_permutes: (cycle_of_list cs) permutes (set cs)
  <proof>

```

```

theorem cyclic_rotation:
  assumes cycle cs shows  $\text{map } ((\text{cycle\_of\_list } cs) \text{ `` } n) \text{ cs} = \text{rotate } n \text{ cs}$ 
  <proof>

```

```

corollary cycle_is_surj:
  assumes cycle cs shows (cycle_of_list cs) ' (set cs) = (set cs)
  <proof>

```

```

corollary cycle_is_id_root:

```

assumes *cycle cs shows* $(\text{cycle_of_list } cs) \rightsquigarrow (\text{length } cs) = \text{id}$
 $\langle \text{proof} \rangle$

corollary *cycle_of_list_rotate_independent*:

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

6.3 Conjugation of cycles

lemma *conjugation_of_cycle*:

assumes *cycle cs and bij p*

shows $p \circ (\text{cycle_of_list } cs) \circ (\text{inv } p) = \text{cycle_of_list } (\text{map } p \text{ } cs)$

$\langle \text{proof} \rangle$

6.4 When Cycles Commute

lemma *cycles_commute*:

assumes *cycle p cycle q and set p $\cap$ set q = {}*

shows $(\text{cycle_of_list } p) \circ (\text{cycle_of_list } q) = (\text{cycle_of_list } q) \circ (\text{cycle_of_list } p)$

$\langle \text{proof} \rangle$

6.5 Cycles from Permutations

6.5.1 Exponentiation of permutations

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

lemma *permutation_funpow*:

assumes *permutation p shows permutation* $(p \rightsquigarrow n)$

$\langle \text{proof} \rangle$

lemma *permutes_funpow*:

assumes *p permutes S shows* $(p \rightsquigarrow n)$ *permutes S*

$\langle \text{proof} \rangle$

lemma *funpow_diff*:

assumes *inj p and* $i \leq j$ $(p \rightsquigarrow i) \ a = (p \rightsquigarrow j) \ a$ **shows** $(p \rightsquigarrow (j - i)) \ a = a$

$\langle \text{proof} \rangle$

lemma *permutation_is_nilpotent*:

assumes *permutation p obtains n where* $(p \rightsquigarrow n) = \text{id}$ **and** $n > 0$

$\langle \text{proof} \rangle$

lemma *permutation_is_nilpotent'*:

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

$\langle \text{proof} \rangle$

6.5.2 Extraction of cycles from permutations

definition $least_power :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow nat$
where $least_power\ f\ x = (LEAST\ n. (f \frown n)\ x = x \wedge n > 0)$

abbreviation $support :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ list$
where $support\ p\ x \equiv map\ (\lambda i. (p \frown i)\ x)\ [0..< (least_power\ p\ x)]$

lemma $least_powerI$:
assumes $(f \frown n)\ x = x$ **and** $n > 0$
shows $(f \frown (least_power\ f\ x))\ x = x$ **and** $least_power\ f\ x > 0$
 $\langle proof \rangle$

lemma $least_power_le$:
assumes $(f \frown n)\ x = x$ **and** $n > 0$ **shows** $least_power\ f\ x \leq n$
 $\langle proof \rangle$

lemma $least_power_of_permutation$:
assumes $permutation\ p$ **shows** $(p \frown (least_power\ p\ a))\ a = a$ **and** $least_power\ p\ a > 0$
 $\langle proof \rangle$

lemma $least_power_gt_one$:
assumes $permutation\ p$ **and** $p\ a \neq a$ **shows** $least_power\ p\ a > Suc\ 0$
 $\langle proof \rangle$

lemma $least_power_minimal$:
assumes $(p \frown n)\ a = a$ **shows** $(least_power\ p\ a)\ dvd\ n$
 $\langle proof \rangle$

lemma $least_power_dvd$:
assumes $permutation\ p$ **shows** $(least_power\ p\ a)\ dvd\ n \longleftrightarrow (p \frown n)\ a = a$
 $\langle proof \rangle$

theorem $cycle_of_permutation$:
assumes $permutation\ p$ **shows** $cycle\ (support\ p\ a)$
 $\langle proof \rangle$

6.6 Decomposition on Cycles

We show that a permutation can be decomposed on cycles

6.6.1 Preliminaries

lemma $support_set$:
assumes $permutation\ p$ **shows** $set\ (support\ p\ a) = range\ (\lambda i. (p \frown i)\ a)$
 $\langle proof \rangle$

lemma *disjoint_support*:
assumes *permutation p* **shows** *disjoint (range (λa. set (support p a))) (is_disjoint ?A)*
 ⟨*proof*⟩

lemma *disjoint_support'*:
assumes *permutation p*
shows *set (support p a) ∩ set (support p b) = {} ⟷ a ∉ set (support p b)*
 ⟨*proof*⟩

lemma *support_coverture*:
assumes *permutation p* **shows** $\bigcup \{ \text{set (support p a)} \mid a. p\ a \neq a \} = \{ a. p\ a \neq a \}$
 ⟨*proof*⟩

theorem *cycle_restrict*:
assumes *permutation p* **and** $b \in \text{set (support p a)}$ **shows** $p\ b = (\text{cycle_of_list (support p a)})\ b$
 ⟨*proof*⟩

6.6.2 Decomposition

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

lemma *semidecomposition*:
assumes *p permutes S* **and** *finite S*
shows $(\lambda y. \text{if } y \in (S - \text{set (support p a)}) \text{ then } p\ y \text{ else } y) \text{ permutes } (S - \text{set (support p a)})$
 ⟨*proof*⟩

theorem *cycle_decomposition*:
assumes *p permutes S* **and** *finite S* **shows** *cycle_decomp S p*
 ⟨*proof*⟩

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⟩
```

```
setup_lifting type_definition_perm
```

```
notation apply (infixl ⟨$⟩ 999)
```

```
lemma bij_apply [simp]:
  bij (apply f)
⟨proof⟩
```

```
lemma perm_eqI:
  assumes ∧ a. f ⟨$⟩ a = g ⟨$⟩ a
  shows f = g
⟨proof⟩
```

```
lemma perm_eq_iff:
  f = g ⟷ (∀ a. f ⟨$⟩ a = g ⟨$⟩ a)
⟨proof⟩
```

```
lemma apply_inj:
  f ⟨$⟩ a = f ⟨$⟩ b ⟷ a = b
⟨proof⟩
```

```
lift_definition affected :: 'a perm ⇒ 'a set
is λf. {a. f a ≠ a} ⟨proof⟩
```

```
lemma in_affected:
  a ∈ affected f ⟷ f ⟨$⟩ a ≠ a
⟨proof⟩
```

```
lemma finite_affected [simp]:
  finite (affected f)
⟨proof⟩
```

```
lemma apply_affected [simp]:
  f ⟨$⟩ a ∈ affected f ⟷ a ∈ affected f
⟨proof⟩
```

```
lemma card_affected_not_one:
  card (affected f) ≠ 1
```

$\langle proof \rangle$

7.2 Identity, composition and inversion

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

lift_definition *one_perm* :: 'a perm
 is *id*
 $\langle proof \rangle$

lemma *apply_one* [*simp*]:
 $apply\ 1 = id$
 $\langle proof \rangle$

lemma *affected_one* [*simp*]:
 $affected\ 1 = \{\}$
 $\langle proof \rangle$

lemma *affected_empty_iff* [*simp*]:
 $affected\ f = \{\} \longleftrightarrow f = 1$
 $\langle proof \rangle$

lift_definition *times_perm* :: 'a perm $\Rightarrow$ 'a perm $\Rightarrow$ 'a perm
 is *comp*
 $\langle proof \rangle$

lemma *apply_times*:
 $apply\ (f * g) = apply\ f \circ apply\ g$
 $\langle proof \rangle$

lemma *apply_sequence*:
 $f\ \langle \$ \rangle\ (g\ \langle \$ \rangle\ a) = apply\ (f * g)\ a$
 $\langle proof \rangle$

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

lift_definition *inverse_perm* :: 'a perm $\Rightarrow$ 'a perm
 is *inv*
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

lemma *apply_inverse*:

$apply\ (inverse\ f) = inv\ (apply\ f)$
 $\langle proof \rangle$

lemma *affected_inverse* [simp]:
 $affected\ (inverse\ f) = affected\ f$
 $\langle proof \rangle$

global_interpretation *perm*: *group times 1 :: 'a perm inverse*
 $\langle proof \rangle$

declare *perm.inverse_distrib_swap* [simp]

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

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

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

7.3 Orbit and order of elements

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

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

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

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

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

lemma *orbit_not_empty* [simp]:

$orbit\ f\ a \neq \{\}$
 $\langle proof \rangle$

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

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

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

lemma *finite_orbit* [simp]:
 $finite\ (orbit\ f\ a)$
 $\langle proof \rangle$

lemma *orbit_1* [simp]:
 $orbit\ 1\ a = \{a\}$
 $\langle proof \rangle$

lemma *order_1* [simp]:
 $order\ 1\ a = 1$
 $\langle proof \rangle$

lemma *card_orbit_eq* [simp]:
 $card\ (orbit\ f\ a) = order\ f\ a$
 $\langle proof \rangle$

lemma *order_greater_zero* [simp]:
 $order\ f\ a > 0$
 $\langle proof \rangle$

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

lemma *order_greater_eq_two_iff*:
 $order\ f\ a \geq 2 \longleftrightarrow a \in affected\ f$
 $\langle proof \rangle$

lemma *order_less_eq_affected*:
assumes $f \neq 1$
shows $order\ f\ a \leq card\ (affected\ f)$
 $\langle proof \rangle$

lemma *affected_order_greater_eq_two*:

assumes $a \in \text{affected } f$

shows $\text{order } f \ a \geq 2$

<proof>

lemma *order_witness_unfold*:

assumes $n > 0$ and $(f \wedge n) \langle \$ \rangle a = a$

shows $\text{order } f \ a = \text{card } ((\lambda m. (f \wedge m) \langle \$ \rangle a) \cdot \{0..<n\})$

<proof>

lemma *inj_on_apply_range*:

inj_on $(\lambda m. (f \wedge m) \langle \$ \rangle a) \{..<\text{order } f \ a\}$

<proof>

lemma *orbit_unfold_image*:

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

<proof>

lemma *in_orbitE*:

assumes $b \in \text{orbit } f \ a$

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

<proof>

lemma *apply_power_order [simp]*:

$(f \wedge \text{order } f \ a) \langle \$ \rangle a = a$

<proof>

lemma *apply_power_left_mult_order [simp]*:

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

<proof>

lemma *apply_power_right_mult_order [simp]*:

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

<proof>

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>

lemma *apply_power_eq_iff*:

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

$\longleftrightarrow ?Q$)

<proof>

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>

lemma *apply_inverse_self_in_orbit [simp]*:

$(\text{inverse } f) \langle \$ \rangle a \in \text{orbit } f a$
 $\langle \text{proof} \rangle$

lemma *apply_inverse_power_eq*:
 $(\text{inverse } (f \wedge n)) \langle \$ \rangle a = (f \wedge (\text{order } f a - n \bmod \text{order } f a)) \langle \$ \rangle a$
 $\langle \text{proof} \rangle$

lemma *apply_power_eq_self_iff*:
 $(f \wedge n) \langle \$ \rangle a = a \iff \text{order } f a \text{ dvd } n$
 $\langle \text{proof} \rangle$

lemma *orbit_equiv*:
 assumes $b \in \text{orbit } f a$
 shows $\text{orbit } f b = \text{orbit } f a$ (is $?B = ?A$)
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

lemma *orbit_disjoint*:

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

7.4 Swaps

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

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

lemma $\text{apply_swap_same} \ [simp]:$
 $c \neq a \implies c \neq b \implies \langle a \leftrightarrow b \rangle \langle \$ \rangle \ c = c$
 $\langle \text{proof} \rangle$

lemma $\text{apply_swap_eq_iff} \ [simp]:$
 $\langle a \leftrightarrow b \rangle \langle \$ \rangle \ c = a \longleftrightarrow c = b$
 $\langle a \leftrightarrow b \rangle \langle \$ \rangle \ c = b \longleftrightarrow c = a$
 $\langle \text{proof} \rangle$

lemma $\text{swap_1} \ [simp]:$
 $\langle a \leftrightarrow a \rangle = 1$
 $\langle \text{proof} \rangle$

lemma $\text{swap_sym}:$
 $\langle b \leftrightarrow a \rangle = \langle a \leftrightarrow b \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{swap_self} \ [simp]:$
 $\langle a \leftrightarrow b \rangle * \langle a \leftrightarrow b \rangle = 1$
 $\langle \text{proof} \rangle$

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

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

7.5 Permutations specified by cycles

fun $\text{cycle} :: 'a \text{ list} \Rightarrow 'a \text{ perm } (\langle _ \rangle)$
where
 $\langle [] \rangle = 1$
 $| \langle [a] \rangle = 1$

$$| \langle a \# b \# as \rangle = \langle a \# as \rangle * \langle a \leftrightarrow b \rangle$$

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

7.6 Syntax

```

bundle 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>

lemma *orbit_trans*:

assumes *s* ∈ *orbit* *f* *t* *t* ∈ *orbit* *f* *u* **shows** *s* ∈ *orbit* *f* *u*
 <proof>

lemma *orbit_subset*:

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

$\langle \text{proof} \rangle$

lemma *orbit_sim_step*:

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

$\langle \text{proof} \rangle$

lemma *orbit_step*:

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

$\langle \text{proof} \rangle$

lemma *self_in_orbit_trans*:

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

$\langle \text{proof} \rangle$

lemma *orbit_swap*:

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

$\langle \text{proof} \rangle$

lemma *permutation_self_in_orbit*:

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

$\langle \text{proof} \rangle$

lemma *orbit_altdef_self_in*:

assumes $s \in \text{orbit } f \ s$ **shows** $\text{orbit } f \ s = \{(f \ \frown \ n) \ s \mid n. \text{True}\}$

$\langle \text{proof} \rangle$

lemma *orbit_altdef_permutation*:

assumes *permutation* f **shows** $\text{orbit } f \ s = \{(f \ \frown \ n) \ s \mid n. \text{True}\}$

$\langle \text{proof} \rangle$

lemma *orbit_altdef_bounded*:

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

$\langle \text{proof} \rangle$

lemma *funpow_in_orbit*:

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

$\langle \text{proof} \rangle$

lemma *finite_orbit*:

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

$\langle \text{proof} \rangle$

lemma *self_in_orbit_step*:

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

$\langle \text{proof} \rangle$

lemma *permutation_orbit_step*:

assumes *permutation f* **shows** $\text{orbit } f (f s) = \text{orbit } f s$
 <proof>

lemma *orbit_nonempty*:
 $\text{orbit } f s \neq \{\}$
 <proof>

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

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

lemma *cyclic_on_funpow_in*:
assumes *cyclic_on f S s* $s \in S$ **shows** $(f^{\sim n}) s \in S$
 <proof>

lemma *finite_cyclic_on*:
assumes *cyclic_on f S* **shows** *finite S*
 <proof>

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

lemma *cyclic_on_inI*:
assumes *cyclic_on f S s* $s \in S$ **shows** $f s \in S$
 <proof>

lemma *orbit_inverse*:
assumes *self*: $a \in \text{orbit } g a$
and *eq*: $\bigwedge x. x \in \text{orbit } g a \implies g' (f x) = f (g x)$
shows $f^{-1} \text{orbit } g a = \text{orbit } g' (f a)$ (**is** ?L = ?R)
 <proof>

lemma *cyclic_on_image*:
assumes *cyclic_on f S*
assumes $\bigwedge x. x \in S \implies g (h x) = h (f x)$
shows *cyclic_on g (h ` S)*
 <proof>

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

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

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

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

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

lemma *permutes_comp_preserves_cyclic2*:
 assumes f *permutes* A *cyclic_on* $g C$
 assumes $A \cap B = \{\}$ $C \subseteq B$
shows *cyclic_on* $(f \circ g) C$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

lemma *orbit_eqI*:

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

8.2 Decomposition of arbitrary permutations

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

lemma *perm_restrict_comp*:

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

lemma *perm_restrict_simps*:

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

lemma *perm_restrict_perm_restrict*:

$\text{perm_restrict } (\text{perm_restrict } f A) B = \text{perm_restrict } f (A \cap B)$
 $\langle \text{proof} \rangle$

lemma *perm_restrict_union*:

assumes $\text{perm_restrict } f A \text{ permutes } A$ $\text{perm_restrict } f B \text{ permutes } B$ $A \cap B = \{\}$
shows $\text{perm_restrict } f A \circ \text{perm_restrict } f B = \text{perm_restrict } f (A \cup B)$
 $\langle \text{proof} \rangle$

lemma *perm_restrict_id[simp]*:

assumes $f \text{ permutes } S$ **shows** $\text{perm_restrict } f S = f$
 $\langle \text{proof} \rangle$

lemma *cyclic_on_perm_restrict*:

$\text{cyclic_on } (\text{perm_restrict } f S) S \longleftrightarrow \text{cyclic_on } f S$
 $\langle \text{proof} \rangle$

lemma *perm_restrict_diff_cyclic*:

assumes $f \text{ permutes } S$ $\text{cyclic_on } f A$
shows $\text{perm_restrict } f (S - A) \text{ permutes } (S - A)$
 $\langle \text{proof} \rangle$

lemma *permutes_decompose*:

assumes $f \text{ permutes } S$ *finite* S
shows $\exists C. (\forall c \in C. \text{cyclic_on } f c) \wedge \bigcup C = S \wedge (\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\})$
 $\langle \text{proof} \rangle$

8.3 Function-power distance between values

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

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 funpow_dist_0 :
assumes $x = y$ **shows** $\text{funpow_dist } f \ x \ y = 0$
 $\langle \text{proof} \rangle$

lemma funpow_dist_least :
assumes $n < \text{funpow_dist } f \ x \ y$ **shows** $(f \ \frown \ n) \ x \neq y$
 $\langle \text{proof} \rangle$

lemma $\text{funpow_dist1_least}$:
assumes $0 < n \ n < \text{funpow_dist1 } f \ x \ y$ **shows** $(f \ \frown \ n) \ x \neq y$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma $\text{funpow_neq_less_funpow_dist}$:
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 $(f \ \frown \ m) \ x \neq (f \ \frown \ n) \ x$
 $\langle \text{proof} \rangle$

lemma $\text{funpow_neq_less_funpow_dist1}$:
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 $(f \ \frown \ m) \ x \neq (f \ \frown \ n) \ x$
 $\langle \text{proof} \rangle$

lemma $\text{inj_on_funpow_dist}$:

```

assumes  $y \in \text{orbit } f \ x$  shows  $\text{inj\_on } (\lambda n. (f \rightsquigarrow n) \ x) \ \{0.. \text{funpow\_dist } f \ x \ y\}$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{inj\_on\_funpow\_dist1}$ :
assumes  $y \in \text{orbit } f \ x$  shows  $\text{inj\_on } (\lambda n. (f \rightsquigarrow n) \ x) \ \{0.. \text{funpow\_dist1 } f \ x \ y\}$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{orbit\_conv\_funpow\_dist1}$ :
assumes  $x \in \text{orbit } f \ x$ 
shows  $\text{orbit } f \ x = (\lambda n. (f \rightsquigarrow n) \ x) \ ^\cdot \ \{0.. \text{funpow\_dist1 } f \ x \ x\}$  (is  $?L = ?R$ )
 $\langle \text{proof} \rangle$ 

lemma  $\text{funpow\_dist1\_prop1}$ :
assumes  $(f \rightsquigarrow n) \ x = y \ 0 < n$  shows  $(f \rightsquigarrow \text{funpow\_dist1 } f \ x \ y) \ x = y$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{funpow\_dist1\_dist}$ :
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  $?L = ?R$ )
 $\langle \text{proof} \rangle$ 

lemma  $\text{funpow\_dist1\_le\_self}$ :
assumes  $(f \rightsquigarrow m) \ x = x \ 0 < m \ y \in \text{orbit } f \ x$ 
shows  $\text{funpow\_dist1 } f \ x \ y \leq m$ 
 $\langle \text{proof} \rangle$ 

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

```