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

October 24, 2026

Contents

1 Transposition function **1**

2 Stirling numbers of first and second kind **5**
2.1 Stirling numbers of the second kind 5
2.2 Stirling numbers of the first kind 6
2.2.1 Efficient code 7

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

3.21 Permuting a list . . . 23
3.22 More lemmas about permutations . . . 25
3.23 Sum over a set of permutations (could generalize to iteration) 27
3.24 Constructing permutations from association lists . . . 27

4 Permuted Lists 29
4.1 An existing notion . . . 29
4.2 Nontrivial conclusions . . . 30
4.3 Trivial conclusions: . . . 30

5 Permutations of a Multiset 32
5.1 Permutations of a multiset . . . 32
5.2 Cardinality of permutations . . . 34
5.3 Permutations of a set . . . 35
5.4 Code generation . . . 36

6 Cycles 39
6.1 Definitions . . . 39
6.2 Basic Properties . . . 39
6.3 Conjugation of cycles . . . 40
6.4 When Cycles Commute . . . 40
6.5 Cycles from Permutations . . . 40
6.5.1 Exponentiation of permutations . . . 40
6.5.2 Extraction of cycles from permutations . . . 41
6.6 Decomposition on Cycles . . . 41
6.6.1 Preliminaries . . . 41
6.6.2 Decomposition . . . 42

7 Permutations as abstract type 42
7.1 Abstract type of permutations . . . 43
7.2 Identity, composition and inversion . . . 44
7.3 Orbit and order of elements . . . 45
7.4 Swaps . . . 49
7.5 Permutations specified by cycles . . . 49
7.6 Syntax . . . 50

8 Permutation orbits 50
8.1 Orbits and cyclic permutations . . . 50
8.2 Decomposition of arbitrary permutations . . . 54
8.3 Function-power distance between values . . . 54

9 Basic combinatorics in Isabelle/HOL (and the Archive of Formal Proofs) 56

1 Transposition function

theory *Transposition*
imports *Main*
begin

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

lemma *transpose_apply_first* [*simp*]:
‹*transpose a b a* = *b*›
⟨*proof*⟩

lemma *transpose_apply_second* [*simp*]:
‹*transpose a b b* = *a*›
⟨*proof*⟩

lemma *transpose_apply_other* [*simp*]:
‹*transpose a b c* = *c*› **if** ‹*c* ≠ *a*› ‹*c* ≠ *b*›
⟨*proof*⟩

lemma *transpose_same* [*simp*]:
‹*transpose a a* = *id*›
⟨*proof*⟩

lemma *transpose_eq_iff*:
‹*transpose a b c* = *d* ⟷ (*c* ≠ *a* ∧ *c* ≠ *b* ∧ *d* = *c*) ∨ (*c* = *a* ∧ *d* = *b*) ∨ (*c* = *b* ∧ *d* = *a*)›
⟨*proof*⟩

lemma *transpose_eq_imp_eq*:
‹*c* = *d*› **if** ‹*transpose a b c* = *transpose a b d*›
⟨*proof*⟩

lemma *transpose_commute* [*ac_simps*]:
‹*transpose b a* = *transpose a b*›
⟨*proof*⟩

lemma *transpose_involutory* [*simp*]:
‹*transpose a b* (*transpose a b c*) = *c*›
⟨*proof*⟩

lemma *transpose_comp_involutory* [*simp*]:
‹*transpose a b* ∘ *transpose a b* = *id*›
⟨*proof*⟩

lemma *transpose_eq_id_iff*: *Transposition.transpose x y* = *id* ⟷ *x* = *y*
⟨*proof*⟩

lemma *transpose_triple*:
‹*transpose a b* (*transpose b c* (*transpose a b d*)) = *transpose a c d*›
if ‹*a* ≠ *c*› **and** ‹*b* ≠ *c*›
⟨*proof*⟩

lemma *transpose_comp_triple*:
‹*transpose a b* ∘ *transpose b c* ∘ *transpose a b* = *transpose a c*›
if ‹*a* ≠ *c*› **and** ‹*b* ≠ *c*›
⟨*proof*⟩

lemma *transpose_image_eq* [*simp*]:
‹*transpose a b* ‘ *A* = *A*› **if** ‹*a* ∈ *A* ⟷ *b* ∈ *A*›
⟨*proof*⟩

lemma *inj_on_transpose* [*simp*]:
‹*inj_on* (*transpose a b*) *A*›
⟨*proof*⟩

lemma *inj_transpose*:
‹*inj* (*transpose a b*)›
⟨*proof*⟩

lemma *surj_transpose*:
‹*surj* (*transpose a b*)›
⟨*proof*⟩

lemma *bij_betw_transpose_iff* [*simp*]:
‹*bij_betw* (*transpose a b*) *A A*› **if** ‹*a* ∈ *A* ⟷ *b* ∈ *A*›
⟨*proof*⟩

lemma *bij_transpose* [*simp*]:
‹*bij* (*transpose a b*)›
⟨*proof*⟩

lemma *bijection_transpose*:
‹*bijection* (*transpose a b*)›
⟨*proof*⟩

lemma *inv_transpose_eq* [*simp*]:
‹*inv* (*transpose a b*) = *transpose a b*›
⟨*proof*⟩

lemma *transpose_apply_commute*:
‹*transpose a b* (*f c*) = *f* (*transpose* (*inv f a*) (*inv f b*) *c*)›
if ‹*bij f*›
⟨*proof*⟩

lemma *transpose_comp_eq*:
‹*transpose a b* ∘ *f* = *f* ∘ *transpose* (*inv f a*) (*inv f b*)›

if ‹*bij f*›
⟨*proof*⟩

lemma *in_transpose_image_iff*:
‹*x* ∈ *transpose a b* ‘ *S* ⟷ *transpose a b x* ∈ *S*›
⟨*proof*⟩

Legacy input alias

⟨*ML*⟩

abbreviation (*input*) *swap* :: ‹′*a* ⇒ ′*a* ⇒ (′*a* ⇒ ′*b*) ⇒ ′*a* ⇒ ′*b*›
where ‹*swap a b f* ≡ *f* ∘ *transpose a b*›

lemma *swap_def*:
‹*Fun.swap a b f* = *f* (*a* := *f b*, *b*:= *f a*)›
⟨*proof*⟩

⟨*ML*⟩

lemma *swap_apply*:
Fun.swap a b f a = *f b*
Fun.swap a b f b = *f a*
c ≠ *a* ⟹ *c* ≠ *b* ⟹ *Fun.swap a b f c* = *f c*
⟨*proof*⟩

lemma *swap_self*: *Fun.swap a a f* = *f*
⟨*proof*⟩

lemma *swap_commute*: *Fun.swap a b f* = *Fun.swap b a f*
⟨*proof*⟩

lemma *swap_nilpotent*: *Fun.swap a b* (*Fun.swap a b f*) = *f*
⟨*proof*⟩

lemma *swap_comp_involutory*: *Fun.swap a b* ∘ *Fun.swap a b* = *id*
⟨*proof*⟩

lemma *swap_triple*:
assumes *a* ≠ *c* **and** *b* ≠ *c*
shows *Fun.swap a b* (*Fun.swap b c* (*Fun.swap a b f*)) = *Fun.swap a c f*
⟨*proof*⟩

lemma *comp_swap*: *f* ∘ *Fun.swap a b g* = *Fun.swap a b* (*f* ∘ *g*)
⟨*proof*⟩

lemma *swap_image_eq*:
assumes *a* ∈ *A b* ∈ *A*
shows *Fun.swap a b f* ‘ *A* = *f* ‘ *A*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *surj_imp_surj_swap*: *surj f* $\Longrightarrow$ *surj* (*Fun.swap a b f*)
⟨*proof*⟩

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

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

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

lemma *swap_image*:
‹*Fun.swap i j f* ‘ *A* = *f* ‘ (*A* − {*i*, *j*}
∪ (*if i* $\in$ *A then* {*j*} *else* {}) ∪ (*if j* $\in$ *A then* {*i*} *else* {}))›
⟨*proof*⟩

lemma *inv_swap_id*: *inv* (*Fun.swap a b id*) = *Fun.swap a b id*
⟨*proof*⟩

lemma *bij_swap_comp*:
assumes *bij p*
shows *Fun.swap a b id* ∘ *p* = *Fun.swap* (*inv p a*) (*inv p b*) *p*
⟨*proof*⟩

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

lemma *swap_unfold*:
‹*Fun.swap a b p* = *p* ∘ *Fun.swap a b id*›
⟨*proof*⟩

lemma *swap_id_idempotent*: *Fun.swap a b id* ∘ *Fun.swap a b id* = *id*
⟨*proof*⟩

lemma *bij_swap_compose_bij*:
‹*bij* (*Fun.swap a b id* ∘ *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* ⇒ *nat* ⇒ *nat*
  where
    Stirling 0 0 = 1
  | *Stirling 0 (Suc k) = 0*
  | *Stirling (Suc n) 0 = 0*
  | *Stirling (Suc n) (Suc k) = Suc k * Stirling n (Suc k) + Stirling n k*

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

lemma *Stirling_less* [*simp*]: *n < k* ⟹ *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 ^ 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* ⇒ *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*
  ⟨*proof*⟩

lemma *stirling_less* [*simp*]: *n < k* ⟹ *stirling n k = 0*
  ⟨*proof*⟩

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

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

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

lemma *stirling_Suc_n_2*:
assumes *n ≥ Suc 0*
shows *stirling (Suc n) 2 = (∑ k=1..n. fact n div k)*
⟨*proof*⟩

lemma *of_nat_stirling_Suc_n_2*:
assumes *n ≥ Suc 0*
shows *(of_nat (stirling (Suc n) 2)::'a::field_char_0) = fact n * (∑ k=1..n. (1 / of_nat k))*
⟨*proof*⟩

lemma *sum_stirling*: *(∑ k≤n. stirling n k) = fact n*
⟨*proof*⟩

lemma *stirling_pochhammer*:
*(∑ k≤n. of_nat (stirling n k) * x ^ k) = (pochhammer x n :: 'a::comm_semiring_1)*
⟨*proof*⟩

A row of the Stirling number triangle

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

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

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

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

2.2.1 Efficient code

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

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

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

lemma *zip_with_prev_altdef*:
zip_with_prev f x xs =
(*if xs* = [] *then* [] *else f x* (*hd xs*) # [*f* (*xs*!*i*) (*xs*!(*i*+*1*)). *i* ← [*0*..<*length xs* − *1*]])
⟨*proof*⟩

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

lemma *stirling_row_aux_correct*:
stirling_row_aux n y xs = *zip_with_prev* (λ*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* :: ‹('*a* ⇒ '*a*) ⇒ '*a set*›
where ‹*fixpoints f* ≡ {*x*. *f x* = *x*}›

lemma *inj_on_fixpoints*:
‹*inj_on f* (*fixpoints f*)›
⟨*proof*⟩

lemma *bij_betw_fixpoints*:
‹*bij_betw f* (*fixpoints f*) (*fixpoints f*)›
⟨*proof*⟩

3.2 Basic definition and consequences

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

lemma *bij_imp_permutes*:
‹*p permutes S*› **if** ‹*bij_betw p S S*› **and** *stable*: ‹⋀*x*. *x* ∉ *S* ⟹ *p x* = *x*›
⟨*proof*⟩

lemma *inj_imp_permutes*:
assumes *i*: *inj_on f S* **and** *fin*: *finite S*
and *fS*: ⋀*x*. *x* ∈ *S* ⟹ *f x* ∈ *S*
and *f*: ⋀*i*. *i* ∉ *S* ⟹ *f i* = *i*
shows *f permutes S*
⟨*proof*⟩

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

lemma *permutes_inj*:
‹*inj p*›
⟨*proof*⟩

lemma *permutes_image*:
‹*p* ‘ *S* = *S*›
⟨*proof*⟩

lemma *permutes_not_in*:
‹*x* ∉ *S* ⟹ *p x* = *x*›
⟨*proof*⟩

lemma *permutes_image_complement*:
‹*p* ‘ (− *S*) = − *S*›
⟨*proof*⟩

lemma *permutes_in_image*:
‹*p x* ∈ *S* ⟷ *x* ∈ *S*›
⟨*proof*⟩

lemma *permutes_surj*:
‹*surj p*›
⟨*proof*⟩

lemma *permutes_inv_o*:
shows $p \circ inv\ p = id$
and $inv\ p \circ p = id$
⟨*proof*⟩

lemma *permutes_inverses*:
shows $p\ (inv\ p\ x) = x$
and $inv\ p\ (p\ x) = x$
⟨*proof*⟩

lemma *permutes_inv_eq*:
‹$inv\ p\ y = x \longleftrightarrow p\ x = y$›
⟨*proof*⟩

lemma *permutes_inj_on*:
‹*inj_on p A*›
⟨*proof*⟩

lemma *permutes_bij*:
‹*bij p*›
⟨*proof*⟩

lemma *permutes_imp_bij*:
‹*bij_betw p S S*›
⟨*proof*⟩

lemma *permutes_subset*:
‹*p permutes T*› **if** ‹$S \subseteq T$›
⟨*proof*⟩

lemma *permutes_imp_permutes_insert*:
‹*p permutes insert x S*›
⟨*proof*⟩

end

lemma *permutes_id* [*simp*]:
‹*id permutes S*›
⟨*proof*⟩

lemma *permutes_empty* [*simp*]:
‹p *permutes* $\{\} \longleftrightarrow p = id$›
⟨*proof*⟩

lemma *permutes_sing* [*simp*]:
  ‹*p permutes* {*a*} ⟷ *p* = *id*›
⟨*proof*⟩

lemma *permutes_univ*: *p permutes UNIV* ⟷ (∀ *y*. ∃!*x*. *p x* = *y*)
  ⟨*proof*⟩

lemma *permutes_swap_id*: *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *transpose a b permutes S*
  ⟨*proof*⟩

lemma *permutes_altdef*: *p permutes A* ⟷ *bij_betw p A A* ∧ {*x*. *p x* ≠ *x*} ⊆ *A*
  ⟨*proof*⟩

lemma *permutes_superset*:
  ‹*p permutes T*› **if** ‹*p permutes S*› ‹⋀*x*. *x* ∈ *S* − *T* ⟹ *p x* = *x*›
⟨*proof*⟩

lemma *permutes_bij_inv_into*:
  fixes *A* :: *'a set*
    and *B* :: *'b set*
  assumes *p permutes A*
    and *bij_betw f A B*
  shows (λ*x*. *if x* ∈ *B then f* (*p* (*inv_into A f x*)) *else x*) *permutes B*
⟨*proof*⟩

lemma *permutes_image_mset*:
  assumes *p permutes A*
  shows *image_mset p* (*mset_set A*) = *mset_set A*
  ⟨*proof*⟩

lemma *permutes_implies_image_mset_eq*:
  assumes *p permutes A* ⋀*x*. *x* ∈ *A* ⟹ *f x* = *f'* (*p x*)
  shows *image_mset f'* (*mset_set A*) = *image_mset f* (*mset_set A*)
⟨*proof*⟩

3.3 Group properties

lemma *permutes_compose*: *p permutes S* ⟹ *q permutes S* ⟹ *q* ∘ *p permutes S*
  ⟨*proof*⟩

lemma *permutes_inv*:
  assumes *p permutes S*
  shows *inv p permutes S*
  ⟨*proof*⟩

lemma *permutes_inv_inv*:
  assumes *p permutes S*
  shows *inv* (*inv p*) = *p*
  ⟨*proof*⟩

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

lemma *permutes_vimage*: *f permutes A* $\Longrightarrow$ *f* − ‘ *A* = *A*
  ⟨*proof*⟩

3.4 Restricting a permutation to a subset

definition *restrict_id* :: (*'a* ⇒ *'a*) ⇒ *'a set* ⇒ *'a* ⇒ *'a*
  where *restrict_id f A* = (λ*x*. *if x* ∈ *A then f x else x*)

lemma *restrict_id_cong* [*cong*]:
  assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x$ *A* = *B*
  shows *restrict_id f A* = *restrict_id g B*
  ⟨*proof*⟩

lemma *restrict_id_cong'*:
  assumes *x* ∈ *A* $\Longrightarrow$ *f x* = *g x A* = *B*
  shows *restrict_id f A x* = *restrict_id g B x*
  ⟨*proof*⟩

lemma *restrict_id_simps* [*simp*]:
  x ∈ *A* $\Longrightarrow$ *restrict_id f A x* = *f x*
  x ∉ *A* $\Longrightarrow$ *restrict_id f A x* = *x*
  ⟨*proof*⟩

lemma *bij_betw_restrict_id*:
  assumes *bij_betw f A A A* ⊆ *B*
  shows *bij_betw* (*restrict_id f A*) *B B*
⟨*proof*⟩

lemma *permutes_restrict_id*:
  assumes *bij_betw f A A*
  shows *restrict_id f A permutes A*
  ⟨*proof*⟩

3.5 Mapping a permutation

definition *map_permutation* :: *'a set* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'a*) ⇒ *'b* ⇒ *'b* **where**
  map_permutation A f p = *restrict_id* (*f* ∘ *p* ∘ *inv_into A f*) (*f* ‘ *A*)

lemma *map_permutation_cong_strong*:
  assumes *A* = *B* $\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x$ $\bigwedge x.\ x \in A \Longrightarrow p\ x = q\ x$
  assumes *p* ‘ *A* ⊆ *A inj_on f A*
  shows *map_permutation A f p* = *map_permutation B g q*

⟨*proof*⟩

lemma *map_permutation_cong*:
assumes *inj_on f A p permutes A*
assumes $A = B$ $\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x$ $\bigwedge x.\ x \in A \Longrightarrow p\ x = q\ x$
shows *map_permutation A f p = map_permutation B g q*
⟨*proof*⟩

lemma *inv_into_id* [*simp*]: $x \in A \Longrightarrow$ *inv_into A id x = x*
⟨*proof*⟩

lemma *inv_into_ident* [*simp*]: $x \in A \Longrightarrow$ *inv_into A* $(\lambda x.\ x)$ *x = x*
⟨*proof*⟩

lemma *map_permutation_id* [*simp*]: *p permutes A* $\Longrightarrow$ *map_permutation A id p = p*
⟨*proof*⟩

lemma *map_permutation_ident* [*simp*]: *p permutes A* $\Longrightarrow$ *map_permutation A* $(\lambda x.\ x)$ *p = p*
⟨*proof*⟩

lemma *map_permutation_id'*: *inj_on f A* $\Longrightarrow$ *map_permutation A f id = id*
⟨*proof*⟩

lemma *map_permutation_ident'*: *inj_on f A* $\Longrightarrow$ *map_permutation A f* $(\lambda x.\ x) = (\lambda x.\ x)$
⟨*proof*⟩

lemma *map_permutation_permutes*:
assumes *bij_betw f A B p permutes A*
shows *map_permutation A f p permutes B*
⟨*proof*⟩

lemma *map_permutation_compose*:
fixes $f :: 'a \Rightarrow 'b$ **and** $g :: 'b \Rightarrow 'c$
assumes *bij_betw f A B inj_on g B*
shows *map_permutation B g* (*map_permutation A f p*) = *map_permutation A* $(g \circ f)$ *p*
⟨*proof*⟩

lemma *map_permutation_compose_inv*:
assumes *bij_betw f A B p permutes A* $\bigwedge x.\ x \in A \Longrightarrow g\ (f\ x) = x$
shows *map_permutation B g* (*map_permutation A f p*) = *p*
⟨*proof*⟩

lemma *map_permutation_apply*:
assumes *inj_on f A x* $\in$ *A*
shows *map_permutation A f h* $(f\ x) = f\ (h\ x)$

⟨*proof*⟩

lemma *map_permutation_compose'*:
fixes $f :: 'a \Rightarrow 'b$
assumes *inj_on f A q permutes A*
shows *map_permutation A f (p ∘ q) = map_permutation A f p ∘ map_permutation A f q*
⟨*proof*⟩

lemma *map_permutation_transpose*:
assumes *inj_on f A a ∈ A b ∈ A*
shows *map_permutation A f (Transposition.transpose a b) = Transposition.transpose (f a) (f b)*
⟨*proof*⟩

lemma *map_permutation_permutes_iff*:
assumes *bij_betw f A B p ‘ A ⊆ A* $\bigwedge x.\ x \notin A \Longrightarrow p\ x = x$
shows *map_permutation A f p permutes B ⟷ p permutes A*
⟨*proof*⟩

lemma *bij_betw_permutations*:
assumes *bij_betw f A B*
shows *bij_betw (λπ x. if x ∈ B then f (π (inv_into A f x)) else x) {π. π permutes A} {π. π permutes B} (***is** *bij_betw ?f _ _)*
⟨*proof*⟩

lemma *bij_betw_derangements*:
assumes *bij_betw f A B*
shows *bij_betw (λπ x. if x ∈ B then f (π (inv_into A f x)) else x) {π. π permutes A ∧ (∀x∈A. π x ≠ x)} {π. π permutes B ∧ (∀x∈B. π x ≠ x)}*
*(***is** *bij_betw ?f _ _)*
⟨*proof*⟩

3.6 The number of permutations on a finite set

lemma *permutes_insert_lemma*:
assumes *p permutes (insert a S)*
shows *transpose a (p a) ∘ p permutes S*
⟨*proof*⟩

lemma *permutes_insert*: *{p. p permutes (insert a S)} =*
(λ(b, p). transpose a b ∘ p) ‘ {(b, p). b ∈ insert a S ∧ p ∈ {p. p permutes S}}
⟨*proof*⟩

lemma *card_permutations*:
assumes *card S = n*
and *finite S*
shows *card {p. p permutes S} = fact n*

⟨*proof*⟩

lemma *finite_permutations*:
assumes *finite S*
shows *finite* {*p*. *p permutes S*}
⟨*proof*⟩

lemma *permutes_doubleton_iff*: *f permutes* {*a*, *b*} ⟷ *f* = *id* ∨ *f* = *Transposition.transpose a b*
⟨*proof*⟩

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*⟩

3.8 Permutations as transposition sequences

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

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

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

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

lemma *permutation_id*[*simp*]: *permutation id*
⟨*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* ⟹ *swapidseq m q* ⟹ *swapidseq* (*n* + *m*) (*p* ∘ *q*)
⟨*proof*⟩

lemma *permutation_compose*: *permutation p* ⟹ *permutation q* ⟹ *permutation* (*p* ∘ *q*)

⟨*proof*⟩

lemma *swapidseq_endswap*: *swapidseq n p* ⟹ *a* ≠ *b* ⟹ *swapidseq* (*Suc n*) (*p* ∘ *transpose a b*)
⟨*proof*⟩

lemma *swapidseq_inverse_exists*: *swapidseq n p* ⟹ ∃ *q*. *swapidseq n q* ∧ *p* ∘ *q* = *id* ∧ *q* ∘ *p* = *id*
⟨*proof*⟩

lemma *swapidseq_inverse*:
assumes *swapidseq n p*
shows *swapidseq n* (*inv p*)
⟨*proof*⟩

lemma *permutation_inverse*: *permutation p* ⟹ *permutation* (*inv p*)
⟨*proof*⟩

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

lemma *swap_id_common*: *a* ≠ *c* ⟹ *b* ≠ *c* ⟹
transpose a b ∘ *transpose a c* = *transpose b c* ∘ *transpose a b*
⟨*proof*⟩

lemma *swap_id_common'*: *a* ≠ *b* ⟹ *a* ≠ *c* ⟹
transpose a c ∘ *transpose b c* = *transpose b c* ∘ *transpose a b*
⟨*proof*⟩

lemma *swap_id_independent*: *a* ≠ *c* ⟹ *a* ≠ *d* ⟹ *b* ≠ *c* ⟹ *b* ≠ *d* ⟹
transpose a b ∘ *transpose c d* = *transpose c d* ∘ *transpose a b*
⟨*proof*⟩

3.11 The identity map only has even transposition sequences

lemma *symmetry_lemma*:
assumes ⋀*a b c d*. *P a b c d* ⟹ *P a b d c*
and ⋀*a b c d*. *a* ≠ *b* ⟹ *c* ≠ *d* ⟹
a = *c* ∧ *b* = *d* ∨ *a* = *c* ∧ *b* ≠ *d* ∨ *a* ≠ *c* ∧ *b* = *d* ∨ *a* ≠ *c* ∧ *a* ≠ *d* ∧ *b* ≠ *c* ∧ *b* ≠ *d* ⟹
P a b c d
shows ⋀*a b c d*. *a* ≠ *b* ⟶ *c* ≠ *d* ⟶ *P a b c d*
⟨*proof*⟩

lemma *swap_general*:
assumes *a* ≠ *b c* ≠ *d*
shows *transpose a b* ∘ *transpose c d* = *id* ∨
(∃ *x y z*. *x* ≠ *a* ∧ *y* ≠ *a* ∧ *z* ≠ *a* ∧ *x* ≠ *y* ∧
transpose a b ∘ *transpose c d* = *transpose x y* ∘ *transpose a z*)

⟨*proof*⟩

lemma *swapidseq_id_iff*[*simp*]: *swapidseq 0 p* ⟷ *p* = *id*
⟨*proof*⟩

lemma *swapidseq_cases*: *swapidseq n p* ⟷
n = *0* ∧ *p* = *id* ∨ (∃ *a b q m*. *n* = *Suc m* ∧ *p* = *transpose a b* ∘ *q* ∧ *swapidseq m q* ∧ *a* ≠ *b*)
⟨*proof*⟩

lemma *fixing_swapidseq_decrease*:
assumes *swapidseq n p*
and *a* ≠ *b*
and (*transpose a b* ∘ *p*) *a* = *a*
shows *n* ≠ *0* ∧ *swapidseq* (*n* − *1*) (*transpose a b* ∘ *p*)
⟨*proof*⟩

lemma *swapidseq_identity_even*:
assumes *swapidseq n* (*id* :: *'a* ⇒ *'a*)
shows *even n*
⟨*proof*⟩

3.12 Therefore we have a welldefined notion of parity

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

lemma *swapidseq_even_even*:
assumes *m*: *swapidseq m p*
and *n*: *swapidseq n p*
shows *even m* ⟷ *even n*
⟨*proof*⟩

lemma *evenperm_unique*:
assumes *swapidseq n p* **and***even n* = *b*
shows *evenperm p* = *b*
⟨*proof*⟩

3.13 And it has the expected composition properties

lemma *evenperm_id*[*simp*]: *evenperm id* = *True*
⟨*proof*⟩

lemma *evenperm_identity* [*simp*]:
‹*evenperm* (λ*x*. *x*)›
⟨*proof*⟩

lemma *evenperm_swap*: *evenperm* (*transpose a b*) = (*a* = *b*)
⟨*proof*⟩

lemma *evenperm_comp*:

assumes *permutation p permutation q*
shows *evenperm* $(p \circ q) \longleftrightarrow$ *evenperm p* $=$ *evenperm q*
⟨*proof*⟩

lemma *evenperm_inv*:
assumes *permutation p*
shows *evenperm* $(inv\ p) =$ *evenperm p*
⟨*proof*⟩

3.14 A more abstract characterization of permutations

lemma *permutation_bijective*:
assumes *permutation p*
shows *bij p*
⟨*proof*⟩

lemma *permutation_finite_support*:
assumes *permutation p*
shows *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* $\{x.\ p\ x \neq x\}$
⟨*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* ⟷ (∃ *S*. *finite S* ∧ *p permutes S*)
⟨*proof*⟩

3.16 Sign of a permutation

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

lemma *sign_cases* [*case_names even odd*]:
obtains ‹*sign p* = *1*› | ‹*sign p* = − *1*›
⟨*proof*⟩

lemma *sign_nz* [*simp*]: *sign p* ≠ *0*
⟨*proof*⟩

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

lemma *sign_identity* [*simp*]:
‹*sign* (λ*x*. *x*) = *1*›
⟨*proof*⟩

lemma *sign_inverse*: *permutation p* ⟹ *sign* (*inv p*) = *sign p*
⟨*proof*⟩

lemma *sign_compose*: *permutation p* ⟹ *permutation q* ⟹ *sign* (*p* ∘ *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*⟩

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

3.17 An induction principle in terms of transpositions

definition *apply_transps* :: ('*a* × '*a*) *list* ⇒ '*a* ⇒ '*a* **where**
apply_transps xs = *foldr* (∘) (*map* (λ(*a*,*b*). *Transposition.transpose a b*) *xs*) *id*

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

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

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

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

lemma *permutes_apply_transps*:
assumes ∀(*a*,*b*)∈*set xs*. *a* ∈ *A* ∧ *b* ∈ *A*
shows *apply_transps xs permutes A*
⟨*proof*⟩

lemma *permutes_induct* [*consumes 2*, *case_names id swap*]:
assumes *p permutes S finite S*
assumes *P id*
assumes ⋀*a b p*. *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *a* ≠ *b* ⟹ *P p* ⟹ *p permutes S*
⟹ *P* (*Transposition.transpose a b* ∘ *p*)
shows *P p*
⟨*proof*⟩

lemma *permutes_rev_induct*[*consumes 2*, *case_names id swap*]:
assumes *finite S p permutes S*
assumes *P id*
assumes ⋀*a b p*. *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *a* ≠ *b* ⟹ *P p* ⟹ *p permutes S*
⟹ *P* (*p* ∘ *Transposition.transpose a b*)
shows *P p*
⟨*proof*⟩

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

lemma *permutes_from_transpositions*:
assumes *p permutes A finite A*
shows ∃*xs*. (∀(*a*,*b*)∈*set xs*. *a* ≠ *b* ∧ *a* ∈ *A* ∧ *b* ∈ *A*) ∧ *apply_transps xs* = *p*
⟨*proof*⟩

3.18 More on the sign of permutations

lemma *evenperm_apply_transps_iff*:
assumes $\forall (a,b) \in set\ xs.\ a \neq b$
shows *evenperm* (*apply_transps xs*) $\longleftrightarrow$ *even* (*length xs*)
⟨*proof*⟩

lemma *evenperm_map_permutation*:
assumes *f*: *inj_on f A* **and** *p permutes A finite A*
shows *evenperm* (*map_permutation A f p*) $\longleftrightarrow$ *evenperm p*
⟨*proof*⟩

lemma *sign_map_permutation*:
assumes *inj_on f A p permutes A finite A*
shows *sign* (*map_permutation A f p*) = *sign p*
⟨*proof*⟩

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

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

lemma *sign_on_cong* [*cong*]:
assumes $A = B \bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x$
shows *sign_on A f* = *sign_on B g*
⟨*proof*⟩

lemma *sign_on_permutes*:
assumes *f permutes A A* $\subseteq$ *B*
shows *sign_on B f* = *sign f*
⟨*proof*⟩

lemma *sign_on_id* [*simp*]: *sign_on A id* = *1*
⟨*proof*⟩

lemma *sign_on_ident* [*simp*]: *sign_on A* ($\lambda x.\ x$) = *1*
⟨*proof*⟩

lemma *sign_on_transpose*:
assumes $a \in A\ b \in A\ a \neq b$
shows *sign_on A* (*Transposition.transpose a b*) = -1
⟨*proof*⟩

lemma *sign_on_compose*:
assumes *bij_betw f A A bij_betw g A A finite A*
shows *sign_on A* ($f \circ g$) = *sign_on A f* $*$ *sign_on A g*
⟨*proof*⟩

3.19 Transpositions of adjacent elements

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

function *adj_transp_seq* :: *nat* ⇒ *nat* ⇒ *nat list* **where**
adj_transp_seq a b =
(*if a* ≥ *b then* []
else if b = *a* + *1 then* [*a*]
else a # *adj_transp_seq* (*a*+*1*) *b* @ [*a*])
⟨*proof*⟩
termination ⟨*proof*⟩

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

lemma *length_adj_transp_seq*:
a < *b* ⟹ *length* (*adj_transp_seq a b*) = *2* ∗ (*b* − *a*) − *1*
⟨*proof*⟩

definition *apply_adj_transps* :: *nat list* ⇒ *nat* ⇒ *nat*
where *apply_adj_transps xs* = *foldl* (∘) *id* (*map* (λ*x*. *Transposition.transpose x* (*x*+*1*)) *xs*)

lemma *apply_adj_transps_aux*:
f ∘ *foldl* (∘) *g* (*map* (λ*x*. *Transposition.transpose x* (*Suc x*)) *xs*) =
foldl (∘) (*f* ∘ *g*) (*map* (λ*x*. *Transposition.transpose x* (*Suc x*)) *xs*)
⟨*proof*⟩

lemma *apply_adj_transps_Nil* [*simp*]: *apply_adj_transps* [] = *id*
and *apply_adj_transps_Cons* [*simp*]:
apply_adj_transps (*x* # *xs*) = *Transposition.transpose x* (*x*+*1*) ∘ *apply_adj_transps xs*
and *apply_adj_transps_snoc* [*simp*]:
apply_adj_transps (*xs* @ [*x*]) = *apply_adj_transps xs* ∘ *Transposition.transpose x* (*x*+*1*)
⟨*proof*⟩

lemma *adj_transp_seq_correct*:
assumes *a* < *b*
shows *apply_adj_transps* (*adj_transp_seq a b*) = *Transposition.transpose a b*
⟨*proof*⟩

lemma *permutation_apply_adj_transps*: *permutation* (*apply_adj_transps xs*)
⟨*proof*⟩

lemma *permutes_apply_adj_transps*:
assumes ∀ *x*∈*set xs*. *x* ∈ *A* ∧ *Suc x* ∈ *A*

shows *apply_adj_transps xs permutes A*
⟨*proof*⟩

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

3.20 Transferring properties of permutations along bijections

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

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

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

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

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

lemma $permutes_p'$: p' *permutes B*
⟨*proof*⟩

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

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

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

end

locale *permutes_bij_finite* = *permutes_bij* +
  assumes *finite_A*: *finite A*
begin

lemma *evenperm_p'_iff*: *evenperm p'* ⟷ *evenperm p*
⟨*proof*⟩

lemma *sign_p'*: *sign p'* = *sign p*
  ⟨*proof*⟩

end

3.21 Permuting a list

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

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

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

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

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

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

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

lemma *permute_list_ident* [*simp*]: *permute_list* (λ*x*. *x*) *xs* = *xs*
  ⟨*proof*⟩

lemma *permute_list_id* [*simp*]: *permute_list id xs* = *xs*
  ⟨*proof*⟩

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

assumes *f permutes* {..<*length xs*}
shows *mset* (*permute_list f xs*) = *mset xs*
⟨*proof*⟩

lemma *set_permute_list* [*simp*]:
assumes *f permutes* {..<*length xs*}
shows *set* (*permute_list f xs*) = *set xs*
⟨*proof*⟩

lemma *distinct_permute_list* [*simp*]:
assumes *f permutes* {..<*length xs*}
shows *distinct* (*permute_list f xs*) = *distinct xs*
⟨*proof*⟩

lemma *permute_list_zip*:
assumes *f permutes A A* = {..<*length xs*}
assumes [*simp*]: *length xs* = *length ys*
shows *permute_list f* (*zip xs ys*) = *zip* (*permute_list f xs*) (*permute_list f ys*)
⟨*proof*⟩

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

lemma *list_all2_permute_list_iff*:
‹*list_all2 P* (*permute_list p xs*) (*permute_list p ys*) ⟷ *list_all2 P xs ys*›
if ‹*p permutes* {..<*length xs*}›
⟨*proof*⟩

3.22 More lemmas about permutations

lemma *permutes_in_funpow_image*:
assumes *f permutes S x* ∈ *S*
shows (*f* ^^ *n*) *x* ∈ *S*
⟨*proof*⟩

lemma *permutation_self*:
assumes ‹*permutation p*›
obtains *n* **where** ‹*n* > *0*› ‹(*p* ^^ *n*) *x* = *x*›
⟨*proof*⟩

The following few lemmas were contributed by Lukas Bulwahn.

lemma *count_image_mset_eq_card_vimage*:
assumes *finite A*
shows *count* (*image_mset f* (*mset_set A*)) *b* = *card* {*a* ∈ *A*. *f a* = *b*}
⟨*proof*⟩
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*⟩
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*⟩

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*⟩

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*⟩

lemma *image_inverse_permutations*: $\{inv\ p\ |p.\ p\ permutes\ S\} = \{p.\ p\ permutes\ S\}$
⟨*proof*⟩

lemma *image_compose_permutations_left*:
assumes $q\ permutes\ S$
shows $\{q \circ p\ |p.\ p\ permutes\ S\} = \{p.\ p\ permutes\ S\}$
⟨*proof*⟩

lemma *image_compose_permutations_right*:
assumes $q\ permutes\ S$
shows $\{p \circ q \mid p.\ p\ permutes\ S\} = \{p\ .\ p\ permutes\ S\}$
⟨*proof*⟩

lemma *permutes_in_seg*: $p\ permutes\ \{1\ ..n\} \Longrightarrow i \in \{1..n\} \Longrightarrow 1 \leq p\ i \wedge p\ i \leq n$
⟨*proof*⟩

lemma *sum_permutations_inverse*: $sum\ f\ \{p.\ p\ permutes\ S\} = sum\ (\lambda p.\ f(inv\ p))\ \{p.\ p\ permutes\ S\}$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *setum_permutations_compose_left*:

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

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

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

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

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

lemma *permutes_nat_less*:
‹*p i* < *n*›
⟨*proof*⟩

lemma *permutes_nat_inv_less*:
‹*inv p i* < *n*›
⟨*proof*⟩

end

context *comm_monoid_set*
begin

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

end

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

lemma *sum_over_permutations_insert*:
assumes *fS*: *finite S*

and *aS*: $a \notin S$
shows *sum f* $\{p.\ p$ *permutes* $(insert\ a\ S)\} =$
sum $(\lambda b.\ sum\ (\lambda q.\ f\ (transpose\ a\ b \circ q))\ \{p.\ p\ permutes\ S\})\ (insert\ a\ S)$
⟨*proof*⟩

3.24 Constructing permutations from association lists

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

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

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')
⟨*proof*⟩

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*: $None \notin A \Longrightarrow$ *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* ∘ *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′* ⟹ *permutation_of_list* ((*x′*,*y*)#*xs*) *x* = *y*
x ≠ *x′* ⟹ *permutation_of_list* ((*x′*,*y′*)#*xs*) *x* = *permutation_of_list xs x*
⟨*proof*⟩

lemma *eval_inverse_permutation_of_list* [*simp*]:
inverse_permutation_of_list [] *x* = *x*
x = *x′* ⟹ *inverse_permutation_of_list* ((*y*,*x′*)#*xs*) *x* = *y*
x ≠ *x′* ⟹ *inverse_permutation_of_list* ((*y′*,*x′*)#*xs*) *x* = *inverse_permutation_of_list xs x*
⟨*proof*⟩

lemma *permutation_of_list_id*: *x* ∉ *set* (*map fst xs*) ⟹ *permutation_of_list xs x* = *x*
⟨*proof*⟩

lemma *permutation_of_list_unique′*:
distinct (*map fst xs*) ⟹ (*x*, *y*) ∈ *set xs* ⟹ *permutation_of_list xs x* = *y*
⟨*proof*⟩

lemma *permutation_of_list_unique*:
list_permutes xs A ⟹ (*x*, *y*) ∈ *set xs* ⟹ *permutation_of_list xs x* = *y*
⟨*proof*⟩

lemma *inverse_permutation_of_list_id*:
x ∉ *set* (*map snd xs*) ⟹ *inverse_permutation_of_list xs x* = *x*
⟨*proof*⟩

lemma *inverse_permutation_of_list_unique'*:
distinct (*map snd xs*) $\Longrightarrow$ (*x*, *y*) $\in$ *set xs* $\Longrightarrow$ *inverse_permutation_of_list xs y* = *x*
⟨*proof*⟩

lemma *inverse_permutation_of_list_unique*:
list_permutes xs A $\Longrightarrow$ (*x*,*y*) $\in$ *set xs* $\Longrightarrow$ *inverse_permutation_of_list xs y* = *x*
⟨*proof*⟩

lemma *inverse_permutation_of_list_correct*:
fixes *A* :: *'a set*
assumes *list_permutes xs A*
shows *inverse_permutation_of_list xs* = *inv* (*permutation_of_list xs*)
⟨*proof*⟩

end

4 Permuted Lists

theory *List_Permutation*
imports *Permutations*
begin

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

4.1 An existing notion

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

4.2 Nontrivial conclusions

proposition *perm_swap*:
‹*xs*[*i* := *xs* ! *j*, *j* := *xs* ! *i*] <~~> *xs*›
if ‹*i* < *length xs*› ‹*j* < *length xs*›
⟨*proof*⟩

proposition *mset_le_perm_append*: *mset xs* ⊆# *mset ys* ⟷ (∃ *zs*. *xs* @ *zs* <~~> *ys*)
⟨*proof*⟩

proposition *perm_set_eq*: *xs* <~~> *ys* $\Longrightarrow$ *set xs* = *set ys*
⟨*proof*⟩

proposition *perm_distinct_iff*: *xs* <~~> *ys* $\Longrightarrow$ *distinct xs* ⟷ *distinct ys*
⟨*proof*⟩

theorem *eq_set_perm_remdups*: *set xs = set ys ⟹ remdups xs <~~> remdups ys*
⟨*proof*⟩

proposition *perm_remdups_iff_eq_set*: *remdups x <~~> remdups y ⟷ set x = set y*
⟨*proof*⟩

theorem *permutation_Ex_bij*:
assumes *xs <~~> ys*
shows *∃f. bij_betw f {..<length xs} {..<length ys} ∧ (∀i<length xs. xs ! i = ys ! (f i))*
⟨*proof*⟩

proposition *perm_finite*: *finite {B. B <~~> A}*
⟨*proof*⟩

4.3 Trivial conclusions:

proposition *perm_empty_imp*: *[] <~~> ys ⟹ ys = []*
⟨*proof*⟩

This more general theorem is easier to understand!

proposition *perm_length*: *xs <~~> ys ⟹ length xs = length ys*
⟨*proof*⟩

proposition *perm_sym*: *xs <~~> ys ⟹ ys <~~> xs*
⟨*proof*⟩

We can insert the head anywhere in the list.

proposition *perm_append_Cons*: *a # xs @ ys <~~> xs @ a # ys*
⟨*proof*⟩

proposition *perm_append_swap*: *xs @ ys <~~> ys @ xs*
⟨*proof*⟩

proposition *perm_append_single*: *a # xs <~~> xs @ [a]*
⟨*proof*⟩

proposition *perm_rev*: *rev xs <~~> xs*
⟨*proof*⟩

proposition *perm_append1*: *xs <~~> ys ⟹ l @ xs <~~> l @ ys*
⟨*proof*⟩

proposition *perm_append2*: *xs <~~> ys ⟹ xs @ l <~~> ys @ l*
⟨*proof*⟩

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

⟨*proof*⟩

proposition *perm_empty2* [*iff*]: *xs* <~~> [] ⟷ *xs* = []
⟨*proof*⟩

proposition *perm_sing_imp*: *ys* <~~> *xs* ⟹ *xs* = [*y*] ⟹ *ys* = [*y*]
⟨*proof*⟩

proposition *perm_sing_eq* [*iff*]: *ys* <~~> [*y*] ⟷ *ys* = [*y*]
⟨*proof*⟩

proposition *perm_sing_eq2* [*iff*]: [*y*] <~~> *ys* ⟷ *ys* = [*y*]
⟨*proof*⟩

proposition *perm_remove*: *x* ∈ *set ys* ⟹ *ys* <~~> *x* # *remove1 x ys*
⟨*proof*⟩

Congruence rule

proposition *perm_remove_perm*: *xs* <~~> *ys* ⟹ *remove1 z xs* <~~> *remove1 z ys*
⟨*proof*⟩

proposition *remove_hd* [*simp*]: *remove1 z* (*z* # *xs*) = *xs*
⟨*proof*⟩

proposition *cons_perm_imp_perm*: *z* # *xs* <~~> *z* # *ys* ⟹ *xs* <~~> *ys*
⟨*proof*⟩

proposition *cons_perm_eq* [*simp*]: *z*#*xs* <~~> *z*#*ys* ⟷ *xs* <~~> *ys*
⟨*proof*⟩

proposition *append_perm_imp_perm*: *zs* @ *xs* <~~> *zs* @ *ys* ⟹ *xs* <~~> *ys*
⟨*proof*⟩

proposition *perm_append1_eq* [*iff*]: *zs* @ *xs* <~~> *zs* @ *ys* ⟷ *xs* <~~> *ys*
⟨*proof*⟩

proposition *perm_append2_eq* [*iff*]: *xs* @ *zs* <~~> *ys* @ *zs* ⟷ *xs* <~~> *ys*
⟨*proof*⟩

end

5 Permutations of a Multiset

theory *Multiset_Permutations*
imports
Complex_Main
Permutations

begin

lemma *mset_tl*: $xs \neq [] \Longrightarrow mset\ (tl\ xs) = mset\ xs - \{\#hd\ xs\#\}$
⟨*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 $P\ \{\#\}\ \bigwedge A.\ A \neq \{\#\} \Longrightarrow (\bigwedge x.\ x \in\# A \Longrightarrow P\ (A - \{\#x\#\})) \Longrightarrow P\ A$
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 $\Longrightarrow$ *mset_set A* = *mset xs* $\Longrightarrow$ *distinct xs*
⟨*proof*⟩

5.1 Permutations of a multiset

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

lemma *permutations_of_multisetI*: *mset xs* = *A* $\Longrightarrow$ *xs* ∈ *permutations_of_multiset A*
⟨*proof*⟩

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

lemma *permutations_of_multiset_Cons_iff*:
x # xs ∈ *permutations_of_multiset A* $\longleftrightarrow$ *x* ∈# *A* ∧ *xs* ∈ *permutations_of_multiset* $(A - \{\#x\#\})$
⟨*proof*⟩

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

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

$(\bigcup x{\in}set_mset\ A.\ ((\#)\ x)\ `\ permutations_of_multiset\ (A - \{\#x\#\}))$
(**is** _ = *?rhs*)
⟨*proof*⟩

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

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

lemma *rev_permutations_of_multiset* [*simp*]:
rev ` *permutations_of_multiset A* = *permutations_of_multiset A*
⟨*proof*⟩

lemma *length_finite_permutations_of_multiset*:
xs ∈ *permutations_of_multiset A* ⟹ *length xs* = *size A*
⟨*proof*⟩

lemma *permutations_of_multiset_lists*: *permutations_of_multiset A* ⊆ *lists* (*set_mset*
A)
⟨*proof*⟩

lemma *finite_permutations_of_multiset* [*simp*]: *finite* (*permutations_of_multiset*
A)
⟨*proof*⟩

lemma *permutations_of_multiset_not_empty* [*simp*]: *permutations_of_multiset*
A ≠ {}
⟨*proof*⟩

lemma *permutations_of_multiset_image*:
permutations_of_multiset (*image_mset f A*) = *map f* ` *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*:
$(\prod y{\in}set_mset\ (A+\{\#x\#\}).\ fact\ (count\ (A+\{\#x\#\})\ y)) =$
$(count\ A\ x + 1) * (\prod y{\in}set_mset\ A.\ fact\ (count\ A\ y))$
⟨*proof*⟩ **lemma** *multiset_prod_fact_remove*:
$x \in\# A \Longrightarrow (\prod y{\in}set_mset\ A.\ fact\ (count\ A\ y)) =$

count A x ∗ (∏ *y*∈*set_mset* (*A*−{#*x*#}). *fact* (*count* (*A*−{#*x*#}) *y*))
⟨*proof*⟩

lemma *card_permutations_of_multiset_aux*:
card (*permutations_of_multiset A*) ∗ (∏ *x*∈*set_mset A*. *fact* (*count A x*)) = *fact* (*size A*)
⟨*proof*⟩

theorem *card_permutations_of_multiset*:
card (*permutations_of_multiset A*) = *fact* (*size A*) *div* (∏ *x*∈*set_mset A*. *fact* (*count A x*))
(∏ *x*∈*set_mset A*. *fact* (*count A x*) :: *nat*) *dvd fact* (*size A*)
⟨*proof*⟩

lemma *card_permutations_of_multiset_insert_aux*:
card (*permutations_of_multiset* (*A* + {#*x*#})) ∗ (*count A x* + *1*) =
(*size A* + *1*) ∗ *card* (*permutations_of_multiset A*)
⟨*proof*⟩

lemma *card_permutations_of_multiset_remove_aux*:
assumes *x* ∈# *A*
shows *card* (*permutations_of_multiset A*) ∗ *count A x* =
size A ∗ *card* (*permutations_of_multiset* (*A* − {#*x*#}))
⟨*proof*⟩

lemma *real_card_permutations_of_multiset_remove*:
assumes *x* ∈# *A*
shows *real* (*card* (*permutations_of_multiset* (*A* − {#*x*#}))) =
real (*card* (*permutations_of_multiset A*) ∗ *count A x*) / *real* (*size A*)
⟨*proof*⟩

lemma *real_card_permutations_of_multiset_remove′*:
assumes *x* ∈# *A*
shows *real* (*card* (*permutations_of_multiset A*)) =
real (*size A* ∗ *card* (*permutations_of_multiset* (*A* − {#*x*#}))) / *real* (*count A x*)
⟨*proof*⟩

end

5.3 Permutations of a set

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

lemma *permutations_of_set_altdef*:
finite A ⟹ *permutations_of_set A* = *permutations_of_multiset* (*mset_set A*)
⟨*proof*⟩

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

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

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

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

lemma *UN_set_permutations_of_set* [*simp*]:
finite A ⟹ (⋃ *xs*∈*permutations_of_set A*. *set xs*) = *A*
⟨*proof*⟩

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

lemma *permutations_of_set_nonempty*:
A ≠ {} ⟹ *permutations_of_set A* =
(⋃ *x*∈*A*. (λ*xs*. *x* # *xs*) ‘ *permutations_of_set* (*A* − {*x*}))
⟨*proof*⟩

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

lemma *permutations_of_set_doubleton*:
x ≠ *y* ⟹ *permutations_of_set* {*x*,*y*} = {[*x*,*y*], [*y*,*x*]}
⟨*proof*⟩

lemma *rev_permutations_of_set* [*simp*]:
rev ‘ *permutations_of_set A* = *permutations_of_set A*
⟨*proof*⟩

lemma *length_finite_permutations_of_set*:
xs ∈ *permutations_of_set A* ⟹ *length xs* = *card A*
⟨*proof*⟩

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

lemma *permutations_of_set_empty_iff* [*simp*]:

permutations_of_set A = {} ⟷ ¬finite A
⟨*proof*⟩

lemma *card_permutations_of_set* [*simp*]:
finite A ⟹ card (permutations_of_set A) = fact (card A)
⟨*proof*⟩

lemma *permutations_of_set_image_inj*:
assumes *inj*: *inj_on f A*
shows *permutations_of_set (f ‘ A) = map f ‘ permutations_of_set A*
⟨*proof*⟩

lemma *permutations_of_set_image_permutes*:
σ permutes A ⟹ map σ ‘ permutations_of_set A = 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**
permutations_of_list_impl xs = (if xs = [] then [[]] else
List.bind (remdups xs) (λx. 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) (λx. 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 [] = [[]]
⟨*proof*⟩

lemma *permutations_of_list_impl_nonempty*:
xs ≠ [] ⟹ permutations_of_list_impl xs =
List.bind (remdups xs) (λx. map ((#) x) (permutations_of_list_impl (remove1 x xs)))
⟨*proof*⟩

lemma *set_permutations_of_list_impl*:
set (permutations_of_list_impl xs) = permutations_of_multiset (mset xs)
⟨*proof*⟩

lemma *distinct_permutations_of_list_impl*:

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

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

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

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

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

declare *set_permutations_of_list_impl_aux* [*symmetric*, *code*]

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

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

function *permutations_of_set_aux* **where**
permutations_of_set_aux acc A =
(*if* ¬*finite A then* {} *else if A* = {} *then* {*acc*} *else*
(⋃*x*∈*A*. *permutations_of_set_aux* (*x*#*acc*) (*A* − {*x*}))))
⟨*proof*⟩
termination ⟨*proof*⟩

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

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

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

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

declare *length_remove1* [*termination_simp*]

fun *permutations_of_set_aux_list* **where**
permutations_of_set_aux_list acc xs =

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

definition permutations_of_set_list where
  permutations_of_set_list xs = permutations_of_set_aux_list [] xs

declare permutations_of_set_aux_list.simps [simp del]

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

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

```
lemma distinct_permutations_of_set_aux_list:
  distinct xs ⟹ distinct (permutations_of_set_aux_list acc xs)
  ⟨proof⟩

lemma distinct_permutations_of_set_list:
    distinct xs ⟹ distinct (permutations_of_set_list xs)
  ⟨proof⟩

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

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

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

end


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

6 Cycles

6.1 Definitions

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

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

6.2 Basic Properties

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

lemma *id_outside_supp*:
  assumes *x* ∉ *set cs* **shows** (*cycle_of_list cs*) *x* = *x*
  ⟨*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** *map* ((*cycle_of_list cs*) ^^ *n*) *cs* = *rotate n 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** (*cycle_of_list cs*) ^^ (*length cs*) = *id*
⟨*proof*⟩

corollary *cycle_of_list_rotate_independent*:
  assumes *cycle cs* **shows** (*cycle_of_list cs*) = (*cycle_of_list* (*rotate n cs*))
⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *permutes_funpow*:
assumes *p permutes S* **shows** (*p* ^^ *n*) *permutes S*
⟨*proof*⟩

lemma *funpow_diff*:
assumes *inj p* **and** *i* ≤ *j* (*p* ^^ *i*) *a* = (*p* ^^ *j*) *a* **shows** (*p* ^^ (*j* − *i*)) *a* = *a*
⟨*proof*⟩

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

lemma *permutation_is_nilpotent'*:
assumes *permutation p* **obtains** *n* **where** (*p* ^^ *n*) = *id* **and** *n* > *m*
⟨*proof*⟩

6.5.2 Extraction of cycles from permutations

definition *least_power* :: ('*a* ⇒ '*a*) ⇒ '*a* ⇒ *nat*
where *least_power f x* = (*LEAST n*. (*f* ^^ *n*) *x* = *x* ∧ *n* > *0*)

abbreviation *support* :: ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a list*
where *support p x* ≡ *map* (λ*i*. (*p* ^^ *i*) *x*) [*0*..< (*least_power p x*)]

lemma *least_powerI*:
assumes (*f* ^^ *n*) *x* = *x* **and** *n* > *0*
shows (*f* ^^ (*least_power f x*)) *x* = *x* **and** *least_power f x* > *0*
⟨*proof*⟩

lemma *least_power_le*:
assumes (*f* ^^ *n*) *x* = *x* **and** *n* > *0* **shows** *least_power f x* ≤ *n*

⟨*proof*⟩

lemma *least_power_of_permutation*:
assumes *permutation p* **shows** $(p \hat{\hat{}} (\mathit{least_power}\ p\ a))\ a = a$ **and** *least_power p a* > *0*
⟨*proof*⟩

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

lemma *least_power_minimal*:
assumes $(p \hat{\hat{}} n)\ a = a$ **shows** $(\mathit{least_power}\ p\ a)\ \mathit{dvd}\ n$
⟨*proof*⟩

lemma *least_power_dvd*:
assumes *permutation p* **shows** $(\mathit{least_power}\ p\ a)\ \mathit{dvd}\ n \longleftrightarrow (p \hat{\hat{}} n)\ a = a$
⟨*proof*⟩

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

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** $\mathit{set}\ (\mathit{support}\ p\ a) = \mathit{range}\ (\lambda i.\ (p \hat{\hat{}} i)\ a)$
⟨*proof*⟩

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

lemma *disjoint_support′*:
assumes *permutation p*
shows $\mathit{set}\ (\mathit{support}\ p\ a) \cap \mathit{set}\ (\mathit{support}\ p\ b) = \{\} \longleftrightarrow a \notin \mathit{set}\ (\mathit{support}\ p\ b)$
⟨*proof*⟩

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

theorem *cycle_restrict*:

assumes *permutation p* **and** $b \in$ *set* (*support p a*) **shows** *p b* = (*cycle_of_list* (*support p a*)) *b*
⟨*proof*⟩

6.6.2 Decomposition

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

lemma *semidecomposition*:
assumes *p permutes S* **and** *finite S*
shows (λ*y*. *if y* ∈ (*S* − *set* (*support p a*)) *then p y else y*) *permutes* (*S* − *set* (*support p a*))
⟨*proof*⟩

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. Grieviously missing are cycles since these would require more elaboration, e.g. the concept of distinct lists equivalent under rotation, which maybe would also deserve its own theory. But see theory *src/HOL/ex/Perm_Fragments.thy* for fragments on that.

7.1 Abstract type of permutations

typedef $'a$ *perm* = {*f* :: $'a$ ⇒ $'a$. *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 λ*f*. {*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*
⟨*proof*⟩

7.2 Identity, composition and inversion

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

lift_definition *one_perm* :: *'a perm*
is *id*
⟨*proof*⟩

lemma *apply_one* [*simp*]:
apply 1 = *id*
⟨*proof*⟩

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

lemma *affected_empty_iff* [*simp*]:
affected f = {} ⟷ *f* = *1*
⟨*proof*⟩

lift_definition *times_perm* :: *'a perm* ⇒ *'a perm* ⇒ *'a perm*
is *comp*
⟨*proof*⟩

lemma *apply_times*:
apply (*f* ∗ *g*) = *apply f* ∘ *apply g*
⟨*proof*⟩

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

lemma *affected_times* [*simp*]:
affected (*f* ∗ *g*) ⊆ *affected f* ∪ *affected g*
⟨*proof*⟩

lift_definition *inverse_perm* :: *'a perm* ⇒ *'a perm*
is *inv*
⟨*proof*⟩

instance
⟨*proof*⟩

end

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

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

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

declare *perm.inverse_distrib_swap* [*simp*]

lemma *perm_mult_commute*:
assumes *affected f* ∩ *affected g* = {}
shows *g* ∗ *f* = *f* ∗ *g*

⟨*proof*⟩

lemma *apply_power*:
  apply (*f* ^ *n*) = *apply f* ^^ *n*
  ⟨*proof*⟩

lemma *perm_power_inverse*:
  inverse f ^ *n* = *inverse* ((*f* :: *'a perm*) ^ *n*)
⟨*proof*⟩

7.3 Orbit and order of elements

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

lemma *in_orbitI*:
  assumes (*f* ^ *n*) ⟨$⟩ *a* = *b*
  shows *b* ∈ *orbit f a*
  ⟨*proof*⟩

lemma *apply_power_self_in_orbit* [*simp*]:
  (*f* ^ *n*) ⟨$⟩ *a* ∈ *orbit f a*
  ⟨*proof*⟩

lemma *in_orbit_self* [*simp*]:
  a ∈ *orbit f a*
  ⟨*proof*⟩

lemma *apply_self_in_orbit* [*simp*]:
  f ⟨$⟩ *a* ∈ *orbit f a*
  ⟨*proof*⟩

lemma *orbit_not_empty* [*simp*]:
  orbit f a ≠ {}
  ⟨*proof*⟩

lemma *not_in_affected_iff_orbit_eq_singleton*:
  a ∉ *affected f* ⟷ *orbit f a* = {*a*} (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

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*⟩

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

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

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

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

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

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

lemma *order_greater_eq_two_iff*:
  order f a ≥ *2* ⟷ *a* ∈ *affected f*
  ⟨*proof*⟩

lemma *order_less_eq_affected*:
  assumes *f* ≠ *1*
  shows *order f a* ≤ *card* (*affected f*)
⟨*proof*⟩

lemma *affected_order_greater_eq_two*:
  assumes *a* ∈ *affected f*
  shows *order f a* ≥ *2*
⟨*proof*⟩

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

lemma *inj_on_apply_range*:
  inj_on (λ*m*. (*f* ^ *m*) ⟨$⟩ *a*) {..<*order f a*}
⟨*proof*⟩

lemma *orbit_unfold_image*:

orbit f a = (λ*n*. (*f* ^ *n*) ⟨$⟩ *a*) ‘ {..<*order f a*} (**is** _ = *?A*)
⟨*proof*⟩

lemma *in_orbitE*:
assumes *b* ∈ *orbit f a*
obtains *n* **where** *b* = (*f* ^ *n*) ⟨$⟩ *a* **and** *n* < *order f a*
⟨*proof*⟩

lemma *apply_power_order* [*simp*]:
(*f* ^ *order f a*) ⟨$⟩ *a* = *a*
⟨*proof*⟩

lemma *apply_power_left_mult_order* [*simp*]:
(*f* ^ (*n* ∗ *order f a*)) ⟨$⟩ *a* = *a*
⟨*proof*⟩

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

lemma *apply_power_mod_order_eq* [*simp*]:
(*f* ^ (*n mod order f a*)) ⟨$⟩ *a* = (*f* ^ *n*) ⟨$⟩ *a*
⟨*proof*⟩

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

lemma *apply_inverse_eq_apply_power_order_minus_one*:
(*inverse f*) ⟨$⟩ *a* = (*f* ^ (*order f a* − *1*)) ⟨$⟩ *a*
⟨*proof*⟩

lemma *apply_inverse_self_in_orbit* [*simp*]:
(*inverse f*) ⟨$⟩ *a* ∈ *orbit f a*
⟨*proof*⟩

lemma *apply_inverse_power_eq*:
(*inverse* (*f* ^ *n*)) ⟨$⟩ *a* = (*f* ^ (*order f a* − *n mod order f a*)) ⟨$⟩ *a*
⟨*proof*⟩

lemma *apply_power_eq_self_iff*:
(*f* ^ *n*) ⟨$⟩ *a* = *a* ⟷ *order f a dvd n*
⟨*proof*⟩

lemma *orbit_equiv*:
assumes *b* ∈ *orbit f a*
shows *orbit f b* = *orbit f a* (**is** *?B* = *?A*)
⟨*proof*⟩

lemma *orbit_apply* [*simp*]:
orbit f (*f* ⟨$⟩ *a*) = *orbit f a*
⟨*proof*⟩

lemma *order_apply* [*simp*]:
order f (*f* ⟨$⟩ *a*) = *order f a*
⟨*proof*⟩

lemma *orbit_apply_inverse* [*simp*]:
orbit f (*inverse f* ⟨$⟩ *a*) = *orbit f a*
⟨*proof*⟩

lemma *order_apply_inverse* [*simp*]:
order f (*inverse f* ⟨$⟩ *a*) = *order f a*
⟨*proof*⟩

lemma *orbit_apply_power* [*simp*]:
orbit f ((*f* ^ *n*) ⟨$⟩ *a*) = *orbit f a*
⟨*proof*⟩

lemma *order_apply_power* [*simp*]:
order f ((*f* ^ *n*) ⟨$⟩ *a*) = *order f a*
⟨*proof*⟩

lemma *orbit_inverse* [*simp*]:
orbit (*inverse f*) = *orbit f*
⟨*proof*⟩

lemma *order_inverse* [*simp*]:
order (*inverse f*) = *order f*
⟨*proof*⟩

lemma *orbit_disjoint*:
assumes *orbit f a* ≠ *orbit f b*
shows *orbit f a* ∩ *orbit f b* = {}
⟨*proof*⟩

7.4 Swaps

lift_definition *swap* :: *'a* ⇒ *'a* ⇒ *'a perm* (‹⟨_ ↔ _⟩›)
is λ*a b*. *transpose a b*
⟨*proof*⟩

lemma *apply_swap_simp* [*simp*]:
⟨*a* ↔ *b*⟩ ⟨$⟩ *a* = *b*
⟨*a* ↔ *b*⟩ ⟨$⟩ *b* = *a*
⟨*proof*⟩

lemma *apply_swap_same* [*simp*]:
$c \neq a \Longrightarrow c \neq b \Longrightarrow \langle a \leftrightarrow b\rangle\ \langle\$\rangle\ c = c$
⟨*proof*⟩

lemma *apply_swap_eq_iff* [*simp*]:
$\langle a \leftrightarrow b\rangle\ \langle\$\rangle\ c = a \longleftrightarrow c = b$
$\langle a \leftrightarrow b\rangle\ \langle\$\rangle\ c = b \longleftrightarrow c = a$
⟨*proof*⟩

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

lemma *swap_sym*:
$\langle b \leftrightarrow a\rangle = \langle a \leftrightarrow b\rangle$
⟨*proof*⟩

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

lemma *affected_swap*:
$a \neq b \Longrightarrow \mathit{affected}\ \langle a \leftrightarrow b\rangle = \{a, b\}$
⟨*proof*⟩

lemma *inverse_swap* [*simp*]:
$\mathit{inverse}\ \langle a \leftrightarrow b\rangle = \langle a \leftrightarrow b\rangle$
⟨*proof*⟩

7.5 Permutations specified by cycles

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

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

7.6 Syntax

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

unbundle *no permutation_syntax*

end

8 Permutation orbits

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

8.1 Orbits and cyclic permutations

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

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

lemma *orbit_altdef*: $orbit\ f\ x = \{(f\ \widehat{\ }\widehat{\ }\ n)\ x \mid n.\ 0 < n\}$ (**is** $?L = ?R$)
⟨*proof*⟩

lemma *orbit_trans*:
assumes $s \in orbit\ f\ t\ t \in orbit\ f\ u$ **shows** $s \in orbit\ f\ u$
⟨*proof*⟩

lemma *orbit_subset*:
assumes $s \in orbit\ f\ (f\ t)$ **shows** $s \in orbit\ f\ t$
⟨*proof*⟩

lemma *orbit_sim_step*:
assumes $s \in orbit\ f\ t$ **shows** $f\ s \in orbit\ f\ (f\ t)$
⟨*proof*⟩

lemma *orbit_step*:
assumes $y \in orbit\ f\ x\ f\ x \neq y$ **shows** $y \in orbit\ f\ (f\ x)$
⟨*proof*⟩

lemma *self_in_orbit_trans*:
assumes $s \in orbit\ f\ s\ t \in orbit\ f\ s$ **shows** $t \in orbit\ f\ t$
⟨*proof*⟩

lemma *orbit_swap*:
assumes $s \in orbit\ f\ s\ t \in orbit\ f\ s$ **shows** $s \in orbit\ f\ t$
⟨*proof*⟩

lemma *permutation_self_in_orbit*:

assumes *permutation f* **shows** $s \in$ *orbit f s*
⟨*proof*⟩

lemma *orbit_altdef_self_in*:
assumes $s \in$ *orbit f s* **shows** *orbit f s* $= \{(f \frown n)\ s \mid n.\ True\}$
⟨*proof*⟩

lemma *orbit_altdef_permutation*:
assumes *permutation f* **shows** *orbit f s* $= \{(f \frown n)\ s \mid n.\ True\}$
⟨*proof*⟩

lemma *orbit_altdef_bounded*:
assumes $(f \frown n)\ s = s\ 0 < n$ **shows** *orbit f s* $= \{(f \frown m)\ s \mid m.\ m < n\}$
⟨*proof*⟩

lemma *funpow_in_orbit*:
assumes $s \in$ *orbit f t* **shows** $(f \frown n)\ s \in$ *orbit f t*
⟨*proof*⟩

lemma *finite_orbit*:
assumes $s \in$ *orbit f s* **shows** *finite* (*orbit f s*)
⟨*proof*⟩

lemma *self_in_orbit_step*:
assumes $s \in$ *orbit f s* **shows** *orbit f* (*f s*) $=$ *orbit f s*
⟨*proof*⟩

lemma *permutation_orbit_step*:
assumes *permutation f* **shows** *orbit f* (*f s*) $=$ *orbit f s*
⟨*proof*⟩

lemma *orbit_nonempty*:
orbit f s $\neq \{\}$
⟨*proof*⟩

lemma *orbit_inv_eq*:
assumes *permutation f*
shows *orbit* (*inv f*) $x =$ *orbit f x* (**is** *?L* $=$ *?R*)
⟨*proof*⟩

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

lemma *cyclic_on_funpow_in*:
assumes *cyclic_on f S* $s \in S$ **shows** $(f\frown n)\ s \in S$
⟨*proof*⟩

lemma *finite_cyclic_on*:

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

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

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

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

lemma *cyclic_on_image*:
assumes *cyclic_on f S*
assumes $\bigwedge x.\ x \in S \Longrightarrow g\ (h\ x) = h\ (f\ x)$
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 $x \in A$ $f \in A \to A$ $\bigwedge y.\ y \in A \Longrightarrow f\ y = g\ y$ **shows** *orbit f x* = *orbit g x*
⟨*proof*⟩

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

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

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

lemma *permutes_comp_preserves_cyclic2*:

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

lemma *permutes_orbit_subset*:
assumes *f permutes S x* $\in$ *S* **shows** *orbit f x* $\subseteq$ *S*
⟨*proof*⟩

lemma *cyclic_on_orbit'*:
assumes *permutation f* **shows** *cyclic_on f (orbit f x)*
⟨*proof*⟩

lemma *cyclic_on_orbit*:
assumes *f permutes S finite S* **shows** *cyclic_on f (orbit f x)*
⟨*proof*⟩

lemma *orbit_cyclic_eq3*:
assumes *cyclic_on f S y* $\in$ *S* **shows** *orbit f y* = *S*
⟨*proof*⟩

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

lemma *eq_on_cyclic_on_iff1*:
assumes *cyclic_on f S x* $\in$ *S*
obtains *f x* $\in$ *S f x* = *x* $\longleftrightarrow$ *card S* = *1*
⟨*proof*⟩

lemma *orbit_eqI*:
y = *f x* $\Longrightarrow$ *y* $\in$ *orbit f x*
z = *f y* $\Longrightarrow$ *y* $\in$ *orbit f x* $\Longrightarrow$ *z* $\in$ *orbit f x*
⟨*proof*⟩

8.2 Decomposition of arbitrary permutations

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

lemma *perm_restrict_comp*:
assumes $A \cap B = \{\}$ *cyclic_on f B*
shows *perm_restrict f A o perm_restrict f B* = *perm_restrict f* $(A \cup B)$
⟨*proof*⟩

lemma *perm_restrict_simps*:
x $\in$ *S* $\Longrightarrow$ *perm_restrict f S x* = *f x*
x $\notin$ *S* $\Longrightarrow$ *perm_restrict f S x* = *x*
⟨*proof*⟩

lemma *perm_restrict_perm_restrict*:
perm_restrict (*perm_restrict f A*) *B* = *perm_restrict f* (*A* ∩ *B*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *perm_restrict_id*[*simp*]:
assumes *f permutes S* **shows** *perm_restrict f S* = *f*
⟨*proof*⟩

lemma *cyclic_on_perm_restrict*:
cyclic_on (*perm_restrict f S*) *S* ⟷ *cyclic_on f S*
⟨*proof*⟩

lemma *perm_restrict_diff_cyclic*:
assumes *f permutes S cyclic_on f A*
shows *perm_restrict f* (*S* − *A*) *permutes* (*S* − *A*)
⟨*proof*⟩

lemma *permutes_decompose*:
assumes *f permutes S finite S*
shows ∃ *C*. (∀ *c* ∈ *C*. *cyclic_on f c*) ∧ ⋃ *C* = *S* ∧ (∀ *c1* ∈ *C*. ∀ *c2* ∈ *C*. *c1* ≠ *c2* ⟶ *c1* ∩ *c2* = {})
⟨*proof*⟩

8.3 Function-power distance between values

definition *funpow_dist* :: ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a* ⇒ *nat* **where**
funpow_dist f x y ≡ *LEAST n*. (*f* ^^ *n*) *x* = *y*

abbreviation *funpow_dist1* :: ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a* ⇒ *nat* **where**
funpow_dist1 f x y ≡ *Suc* (*funpow_dist f* (*f x*) *y*)

lemma *funpow_dist_0*:
assumes *x* = *y* **shows** *funpow_dist f x y* = *0*
⟨*proof*⟩

lemma *funpow_dist_least*:
assumes *n* < *funpow_dist f x y* **shows** (*f* ^^ *n*) *x* ≠ *y*
⟨*proof*⟩

lemma *funpow_dist1_least*:
assumes *0* < *n n* < *funpow_dist1 f x y* **shows** (*f* ^^ *n*) *x* ≠ *y*
⟨*proof*⟩

lemma *funpow_dist_prop*:
$y \in orbit\ f\ x \Longrightarrow (f\ \widehat{\ }\widehat{\ }\ funpow_dist\ f\ x\ y)\ x = y$
⟨*proof*⟩

lemma *funpow_dist_0_eq*:
assumes $y \in orbit\ f\ x$ **shows** $funpow_dist\ f\ x\ y = 0 \longleftrightarrow x = y$
⟨*proof*⟩

lemma *funpow_dist_step*:
assumes $x \neq y\ y \in orbit\ f\ x$ **shows** $funpow_dist\ f\ x\ y = Suc\ (funpow_dist\ f\ (f\ x)\ y)$
⟨*proof*⟩

lemma *funpow_dist1_prop*:
assumes $y \in orbit\ f\ x$ **shows** $(f\ \widehat{\ }\widehat{\ }\ funpow_dist1\ f\ x\ y)\ x = y$
⟨*proof*⟩

lemma *funpow_neq_less_funpow_dist*:
assumes $y \in orbit\ f\ x\ m \leq funpow_dist\ f\ x\ y\ n \leq funpow_dist\ f\ x\ y\ m \neq n$
shows $(f\ \widehat{\ }\widehat{\ }\ m)\ x \neq (f\ \widehat{\ }\widehat{\ }\ n)\ x$
⟨*proof*⟩

lemma *funpow_neq_less_funpow_dist1*:
assumes $y \in orbit\ f\ x\ m < funpow_dist1\ f\ x\ y\ n < funpow_dist1\ f\ x\ y\ m \neq n$
shows $(f\ \widehat{\ }\widehat{\ }\ m)\ x \neq (f\ \widehat{\ }\widehat{\ }\ n)\ x$
⟨*proof*⟩

lemma *inj_on_funpow_dist*:
assumes $y \in orbit\ f\ x$ **shows** $inj_on\ (\lambda n.\ (f\ \widehat{\ }\widehat{\ }\ n)\ x)\ \{0..funpow_dist\ f\ x\ y\}$
⟨*proof*⟩

lemma *inj_on_funpow_dist1*:
assumes $y \in orbit\ f\ x$ **shows** $inj_on\ (\lambda n.\ (f\ \widehat{\ }\widehat{\ }\ n)\ x)\ \{0..<funpow_dist1\ f\ x\ y\}$
⟨*proof*⟩

lemma *orbit_conv_funpow_dist1*:
assumes $x \in orbit\ f\ x$
shows $orbit\ f\ x = (\lambda n.\ (f\ \widehat{\ }\widehat{\ }\ n)\ x)\ `\ \{0..<funpow_dist1\ f\ x\ x\}$ (**is** *?L = ?R*)
⟨*proof*⟩

lemma *funpow_dist1_prop1*:
assumes $(f\ \widehat{\ }\widehat{\ }\ n)\ x = y\ 0 < n$ **shows** $(f\ \widehat{\ }\widehat{\ }\ funpow_dist1\ f\ x\ y)\ x = y$
⟨*proof*⟩

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

shows *funpow_dist1 f x z = funpow_dist1 f x y + funpow_dist1 f y z* (**is** *?L = ?R*)
⟨*proof*⟩

lemma *funpow_dist1_le_self*:
assumes $(f \frown\frown m)\ x = x\ 0 < m\ y \in orbit\ f\ x$
shows *funpow_dist1 f x y* $\leq m$
⟨*proof*⟩

end

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

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

end