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

September 11, 2023

Contents

1 Transposition function

theory *Transposition*
 imports *Main*
begin

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

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

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

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

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

lemma *transpose_eq_iff*:
‹*transpose* a b c = $d \longleftrightarrow (c \neq a \wedge c \neq b \wedge d = c) \vee (c = a \wedge d = b) \vee (c = b \wedge d = a)$›
⟨*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 $\circ$ *transpose* a b = *id*›
⟨*proof*⟩

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

lemma *transpose_comp_triple*:
‹*transpose* a b $\circ$ *transpose* b c $\circ$ *transpose* a b = *transpose* a c›
if ‹$a \neq c$› **and** ‹$b \neq c$›
⟨*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* :: ‹$'a \Rightarrow 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$›
  where ‹*swap a b f* $\equiv$ *f* $\circ$ *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*
  $c \neq a \Longrightarrow c \neq b \Longrightarrow$ *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* $\circ$ *Fun.swap a b* = *id*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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

lemma *swap_image_eq*:
  assumes $a \in A$ $b \in A$
  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 ⇒ 'a ⇒ 'b) ⇒ 'a ⇒ 'a list ⇒ 'b list*
where *zip_with_prev f x xs = map2 f (x # xs) xs*

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

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* $S \longleftrightarrow (\forall x.\ x \notin S \longrightarrow p\ x = x) \wedge (\forall y.\ \exists!x.\ p\ x = y)$›

lemma *bij_imp_permutes*:
‹p *permutes* S› **if** ‹*bij_betw* $p\ S\ S$› **and** *stable*: ‹$\bigwedge x.\ x \notin S \Longrightarrow p\ x = x$›
⟨*proof*⟩

context
fixes p :: ‹$'a \Rightarrow 'a$› **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*:
‹$x \notin S \Longrightarrow p\ x = x$›
⟨*proof*⟩

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

lemma *permutes_in_image*:
‹$p\ x \in S \longleftrightarrow x \in S$›
⟨*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*} $\longleftrightarrow$ *p* = *id*›
⟨*proof*⟩

lemma *permutes_univ*: *p permutes UNIV* $\longleftrightarrow$ ($\forall y.\ \exists!x.\ p\ x = y$)
⟨*proof*⟩

lemma *permutes_swap_id*: $a \in S \Longrightarrow b \in S \Longrightarrow$ *transpose a b permutes S*
⟨*proof*⟩

lemma *permutes_superset*:
‹*p permutes T*› **if** ‹*p permutes S*› ‹$\bigwedge x.\ x \in S - T \Longrightarrow p\ x = x$›
⟨*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 $(\lambda x.\ \textit{if}\ x \in B\ \textit{then}\ f\ (p\ (\textit{inv\_into}\ A\ f\ x))\ \textit{else}\ x)$ *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 $\bigwedge x.\ x \in A \Longrightarrow f\ x = f'\ (p\ x)$
shows *image_mset* f' (*mset_set* A) = *image_mset* f (*mset_set* A)
⟨*proof*⟩

3.3 Group properties

lemma *permutes_compose*: p *permutes* $S \Longrightarrow q$ *permutes* $S \Longrightarrow q \circ p$ *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 Mapping permutations with bijections

lemma *bij_betw_permutations*:
assumes *bij_betw* f A B
shows *bij_betw* $(\lambda\pi\ x.\ \textit{if}\ x \in B\ \textit{then}\ f\ (\pi\ (\textit{inv\_into}\ A\ f\ x))\ \textit{else}\ x)$
$\{\pi.\ \pi$ *permutes* $A\}$ $\{\pi.\ \pi$ *permutes* $B\}$ (**is** *bij_betw* *?f* _ _)
⟨*proof*⟩

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

$\{\pi.\ \pi\ permutes\ A \wedge (\forall x{\in}A.\ \pi\ x \neq x)\}$ $\{\pi.\ \pi\ permutes\ B \wedge (\forall x{\in}B.\ \pi\ x \neq x)\}$
(**is** *bij_betw ?f _ _*)
⟨*proof*⟩

3.5 The number of permutations on a finite set

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

3.6 Hence a sort of induction principle composing by swaps

lemma *permutes_induct* [*consumes 2*, *case_names id swap*]:
‹*P p*› **if** ‹*p permutes S*› ‹*finite S*›
and *id*: ‹*P id*›
and *swap*: ‹$\bigwedge a\ b\ p.\ a \in S \Longrightarrow b \in S \Longrightarrow p\ permutes\ S \Longrightarrow P\ p \Longrightarrow P\ (transpose\ a\ b \circ p)$›
⟨*proof*⟩

lemma *permutes_rev_induct* [*consumes 2*, *case_names id swap*]:
‹*P p*› **if** ‹*p permutes S*› ‹*finite S*›
and *id'*: ‹*P id*›
and *swap'*: ‹$\bigwedge a\ b\ p.\ a \in S \Longrightarrow b \in S \Longrightarrow p\ permutes\ S \Longrightarrow P\ p \Longrightarrow P\ (p \circ transpose\ a\ b)$›
⟨*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*: $a \neq c \Longrightarrow b \neq c \Longrightarrow$
$transpose\ a\ b \circ transpose\ a\ c = transpose\ b\ c \circ transpose\ a\ b$
⟨*proof*⟩

lemma *swap_id_common'*: $a \neq b \Longrightarrow a \neq c \Longrightarrow$
$transpose\ a\ c \circ transpose\ b\ c = transpose\ b\ c \circ transpose\ a\ b$
⟨*proof*⟩

lemma *swap_id_independent*: $a \neq c \Longrightarrow a \neq d \Longrightarrow b \neq c \Longrightarrow b \neq d \Longrightarrow$
$transpose\ a\ b \circ transpose\ c\ d = transpose\ c\ d \circ transpose\ a\ b$
⟨*proof*⟩

3.11 The identity map only has even transposition sequences

lemma *symmetry_lemma*:
assumes $\bigwedge a\ b\ c\ d.\ P\ a\ b\ c\ d \Longrightarrow P\ a\ b\ d\ c$
and $\bigwedge a\ b\ c\ d.\ a \neq b \Longrightarrow c \neq d \Longrightarrow$
$a = c \wedge b = d \vee a = c \wedge b \neq d \vee a \neq c \wedge b = d \vee a \neq c \wedge a \neq d \wedge b \neq c \wedge b \neq d \Longrightarrow$
$P\ a\ b\ c\ d$
shows $\bigwedge a\ b\ c\ d.\ a \neq b \longrightarrow c \neq d \longrightarrow P\ a\ b\ c\ d$
⟨*proof*⟩

lemma *swap_general*: $a \neq b \Longrightarrow c \neq d \Longrightarrow$
$transpose\ a\ b \circ transpose\ c\ d = id \vee$
$(\exists x\ y\ z.\ x \neq a \wedge y \neq a \wedge z \neq a \wedge x \neq y \wedge$
$transpose\ a\ b \circ transpose\ c\ d = transpose\ x\ y \circ transpose\ a\ z)$
⟨*proof*⟩

lemma *swapidseq_id_iff*[*simp*]: $swapidseq\ 0\ p \longleftrightarrow p = id$
⟨*proof*⟩

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

lemma *fixing_swapidseq_decrease*:
assumes $swapidseq\ n\ p$
and $a \neq b$
and $(transpose\ a\ b \circ p)\ a = a$
shows $n \neq 0 \wedge swapidseq\ (n - 1)\ (transpose\ a\ b \circ p)$
⟨*proof*⟩

lemma *swapidseq_identity_even*:
assumes $swapidseq\ n\ (id :: 'a \Rightarrow 'a)$
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* $\longleftrightarrow$ *even n*
⟨*proof*⟩

lemma *evenperm_unique*:
assumes *p*: *swapidseq n p*
and *n*:*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*: $permutation\ p \longleftrightarrow bij\ p \wedge finite\ \{x.\ p\ x \neq x\}$
  (**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*: $permutation\ p \longleftrightarrow (\exists S.\ finite\ S \wedge p\ permutes\ S)$
  ⟨*proof*⟩

3.16 Sign of a permutation as a real number

definition *sign* :: ‹$('a \Rightarrow 'a) \Rightarrow int$› — 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*]: $sign\ p \neq 0$
  ⟨*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*⟩

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

3.17 Permuting a list

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

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

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

3.18 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* ⇒ *'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** ∀ *x*∈*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 ∀ *i* ∈ *S*. *p i* ≤ *i*
shows *p* = *id*
⟨*proof*⟩

lemma *permutes_natset_ge*:
fixes *S* :: *'a*::*wellorder set*
assumes *p*: *p permutes S*
and *le*: ∀ *i* ∈ *S*. *p i* ≥ *i*
shows *p* = *id*
⟨*proof*⟩

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* ∘ *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 $sum\ f\ \{p.\ p\ permutes\ S\} = sum\ (\lambda p.\ f(q \circ p))\ \{p.\ p\ permutes\ S\}$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

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

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

lemma *permutes_pair_eq*:
‹$\{(p\ s,\ s)\ |s.\ s \in S\} = \{(s,\ inv\ p\ s)\ |s.\ s \in S\}$› (**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 ($\lambda s.$ g (p s) s) S = F ($\lambda s.$ g s (*inv* p s)) S› (**is** ‹?l = ?r›)
  if ‹p *permutes* S›
⟨*proof*⟩

end

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

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

3.20 Constructing permutations from association lists

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

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

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

lemma *permutation_of_list_Cons*:
  permutation_of_list ((x, y) # *xs*) x' = (*if* $x = x'$ *then y else permutation_of_list xs* x')
  ⟨*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* ∘ *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'* $\Longrightarrow$ *permutation_of_list* ((*x'*,*y*)#*xs*) *x* = *y*
x $\neq$ *x'* $\Longrightarrow$ *permutation_of_list* ((*x'*,*y'*)#*xs*) *x* = *permutation_of_list xs x*
⟨*proof*⟩

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

lemma *permutation_of_list_id*: *x* $\notin$ *set* (*map fst xs*) $\Longrightarrow$ *permutation_of_list xs x* = *x*
⟨*proof*⟩

lemma *permutation_of_list_unique'*:
  distinct (*map fst xs*) $\Longrightarrow$ (*x*, *y*) $\in$ *set xs* $\Longrightarrow$ *permutation_of_list xs x* = *y*
  ⟨*proof*⟩

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

lemma *inverse_permutation_of_list_id*:
  x $\notin$ *set* (*map snd xs*) $\Longrightarrow$ *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* ⟹ *set xs* = *set ys*
⟨*proof*⟩

proposition *perm_distinct_iff*: *xs* <~~> *ys* ⟹ *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* ≠ [] ⟹ *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* {#} ⋀*A*. *A* ≠ {#} ⟹ (⋀*x*. *x* ∈# *A* ⟹ *P* (*A* − {#*x*#})) ⟹ *P* *A*
shows *P A*
⟨*proof*⟩

lemma *map_list_bind*: *map g* (*List.bind xs f*) = *List.bind xs* (*map g* ∘ *f*)
⟨*proof*⟩

lemma *mset_eq_mset_set_imp_distinct*:
finite A ⟹ *mset_set A* = *mset xs* ⟹ *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* ⟹ *xs* ∈ *permutations_of_multiset* *A*
⟨*proof*⟩

lemma *permutations_of_multisetD*: *xs* ∈ *permutations_of_multiset A* ⟹ *mset* *xs* = *A*
⟨*proof*⟩

lemma *permutations_of_multiset_Cons_iff*:
x # *xs* ∈ *permutations_of_multiset A* ⟷ *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* =
(⋃*x*∈*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*∈*set_mset* (*A*+{#*x*#}). *fact* (*count* (*A*+{#*x*#}) *y*)) =
(*count A x* + *1*) ∗ ($\prod$ *y*∈*set_mset A*. *fact* (*count A y*))
⟨*proof*⟩ **lemma** *multiset_prod_fact_remove*:
x ∈# *A* ⟹ ($\prod$ *y*∈*set_mset A*. *fact* (*count A y*)) =
count A x ∗ ($\prod$ *y*∈*set_mset* (*A*−{#*x*#}). *fact* (*count* (*A*−{#*x*#}) *y*))
⟨*proof*⟩

lemma *card_permutations_of_multiset_aux*:
card (*permutations_of_multiset A*) ∗ ($\prod$ *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* ($\prod$ *x*∈*set_mset A*. *fact* (*count A x*))
($\prod$ *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*) (λ*x*. *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 (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_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* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow nat$
  where *least_power f x* = $(LEAST\ n.\ (f\ \hat{}\hat{}\ n)\ x = x \wedge n > 0)$

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

lemma *least_powerI*:
  assumes $(f\ \hat{}\hat{}\ n)\ x = x$ **and** $n > 0$
  shows $(f\ \hat{}\hat{}\ (least\_power\ f\ x))\ x = x$ **and** $least\_power\ f\ x > 0$
  ⟨*proof*⟩

lemma *least_power_le*:
  assumes $(f\ \hat{}\hat{}\ n)\ x = x$ **and** $n > 0$ **shows** $least\_power\ f\ x \leq n$
  ⟨*proof*⟩

lemma *least_power_of_permutation*:
  assumes *permutation p* **shows** $(p\ \hat{}\hat{}\ (least\_power\ p\ a))\ a = a$ **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** $(least\_power\ p\ a)\ dvd\ n$
⟨*proof*⟩

lemma *least_power_dvd*:
  assumes *permutation p* **shows** $(least\_power\ p\ a)\ dvd\ n \longleftrightarrow (p\ \hat{}\hat{}\ n)\ a = a$
⟨*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** $set\ (support\ p\ a) = range\ (\lambda i.\ (p\ \hat{}\hat{}\ i)\ a)$
⟨*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 *set* (*support p a*) ∩ *set* (*support p b*) = {} ⟷ *a* ∉ *set* (*support p b*)
⟨*proof*⟩

lemma *support_coverture*:
assumes *permutation p* **shows** ⋃ { *set* (*support p a*) | *a*. *p a* ≠ *a* } = { *a*. *p a* ≠ *a* }
⟨*proof*⟩

theorem *cycle_restrict*:
assumes *permutation p* **and** *b* ∈ *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 $a \in$ *affected f*
  shows *order f a* $\geq$ *2*
⟨*proof*⟩

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

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

lemma *orbit_unfold_image*:
  orbit f a = $(\lambda n.\ (f \mathbin{\widehat{}} n)$ ⟨$⟩ $a)$ ‘ {..<*order f a*} (**is** _ = *?A*)
⟨*proof*⟩

lemma *in_orbitE*:
  assumes $b \in$ *orbit f a*
  obtains *n* **where** $b = (f \mathbin{\widehat{}} n)$ ⟨$⟩ *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*) ⟨$⟩ $a \in$ *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 $b \in$ *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 ≠ *a* ⟹ *c* ≠ *b* ⟹ ⟨*a* ↔ *b*⟩ ⟨$⟩ *c* = *c*
⟨*proof*⟩

lemma *apply_swap_eq_iff* [*simp*]:
⟨*a* ↔ *b*⟩ ⟨$⟩ *c* = *a* ⟷ *c* = *b*
⟨*a* ↔ *b*⟩ ⟨$⟩ *c* = *b* ⟷ *c* = *a*
⟨*proof*⟩

lemma *swap_1* [*simp*]:
⟨*a* ↔ *a*⟩ = *1*
⟨*proof*⟩

lemma *swap_sym*:
⟨*b* ↔ *a*⟩ = ⟨*a* ↔ *b*⟩
⟨*proof*⟩

lemma *swap_self* [*simp*]:
⟨*a* ↔ *b*⟩ * ⟨*a* ↔ *b*⟩ = *1*
⟨*proof*⟩

lemma *affected_swap*:
a ≠ *b* ⟹ *affected* ⟨*a* ↔ *b*⟩ = {*a*, *b*}
⟨*proof*⟩

lemma *inverse_swap* [*simp*]:
inverse ⟨*a* ↔ *b*⟩ = ⟨*a* ↔ *b*⟩
⟨*proof*⟩

7.5 Permutations specified by cycles

fun *cycle* :: *'a list* ⇒ *'a perm* (⟨_⟩)
where
⟨[]⟩ = *1*
| ⟨[*a*]⟩ = *1*

| ⟨a # b # as⟩ = ⟨a # as⟩ ∗ ⟨a↔b⟩

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

7.6 Syntax

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

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

unbundle *no_permutation_syntax*

end

8 Permutation orbits

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

8.1 Orbits and cyclic permutations

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

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

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

lemma *orbit_trans*:
  assumes *s* ∈ *orbit f t t* ∈ *orbit f u* **shows** *s* ∈ *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* $= \{(f\ \hat{}\hat{}\ n)\ s \mid n.\ True\}$
⟨*proof*⟩

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

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

lemma *funpow_in_orbit*:
assumes $s \in$ *orbit f t* **shows** $(f\ \hat{}\hat{}\ n)\ s \in$ *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 ≠ {}
⟨*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 ⟷ *S* ≠ {} ∧ (∀ *s*∈*S*. *S* = *orbit f s*)
⟨*proof*⟩

lemma *cyclic_on_funpow_in*:
assumes *cyclic_on f S s* ∈ *S* **shows** (*f*^^*n*) *s* ∈ *S*
⟨*proof*⟩

lemma *finite_cyclic_on*:
assumes *cyclic_on f S* **shows** *finite S*
⟨*proof*⟩

lemma *cyclic_on_singleI*:
assumes *s* ∈ *S S* = *orbit f s* **shows** *cyclic_on f S*
⟨*proof*⟩

lemma *cyclic_on_inI*:
assumes *cyclic_on f S s* ∈ *S* **shows** *f s* ∈ *S*
⟨*proof*⟩

lemma *orbit_inverse*:
assumes *self*: *a* ∈ *orbit g a*
and *eq*: ⋀*x*. *x* ∈ *orbit g a* ⟹ *g'* (*f x*) = *f* (*g x*)
shows *f* ‘ *orbit g a* = *orbit g'* (*f a*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *cyclic_on_image*:
assumes *cyclic_on f S*
assumes ⋀*x*. *x* ∈ *S* ⟹ *g* (*h x*) = *h* (*f x*)
shows *cyclic_on g* (*h* ‘ *S*)
⟨*proof*⟩

lemma *cyclic_on_f_in*:
assumes *f permutes S cyclic_on f A f x* ∈ *A*
shows *x* ∈ *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 ≡ *if x* ∈ *S then f x else x*

lemma *perm_restrict_comp*:
  assumes *A* ∩ *B* = {} *cyclic_on f B*
  shows *perm_restrict f A o perm_restrict f B* = *perm_restrict f* (*A* ∪ *B*)
⟨*proof*⟩

lemma *perm_restrict_simps*:
  x ∈ *S* ⟹ *perm_restrict f S x* = *f x*
  x ∉ *S* ⟹ *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 ∈ *orbit f x* ⟹ (*f* ^^ *funpow_dist f x y*) *x* = *y*
⟨*proof*⟩

lemma *funpow_dist_0_eq*:
assumes *y* ∈ *orbit f x* **shows** *funpow_dist f x y* = *0* ⟷ *x* = *y*
⟨*proof*⟩

lemma *funpow_dist_step*:
assumes *x* ≠ *y y* ∈ *orbit f x* **shows** *funpow_dist f x y* = *Suc* (*funpow_dist f* (*f x*) *y*)
⟨*proof*⟩

lemma *funpow_dist1_prop*:
assumes *y* ∈ *orbit f x* **shows** (*f* ^^ *funpow_dist1 f x y*) *x* = *y*
⟨*proof*⟩

lemma *funpow_neq_less_funpow_dist*:
assumes *y* ∈ *orbit f x m* ≤ *funpow_dist f x y n* ≤ *funpow_dist f x y m* ≠ *n*
shows (*f* ^^ *m*) *x* ≠ (*f* ^^ *n*) *x*
⟨*proof*⟩

lemma *funpow_neq_less_funpow_dist1*:
assumes *y* ∈ *orbit f x m* < *funpow_dist1 f x y n* < *funpow_dist1 f x y m* ≠ *n*
shows (*f* ^^ *m*) *x* ≠ (*f* ^^ *n*) *x*
⟨*proof*⟩

lemma *inj_on_funpow_dist*:

assumes $y \in$ *orbit f x* **shows** *inj_on* ($\lambda n.$ (f ^^ 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 ^^ 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 ^^ n) x) ‘ {*0..<funpow_dist1 f x x*} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *funpow_dist1_prop1*:
assumes (f ^^ n) $x = y$ $0 < n$ **shows** (f ^^ *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 ^^ 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