The Boustrophedon Transform, the Entringer Numbers, and Related Sequences

Manuel Eberl

March 17, 2025

Abstract

This entry defines the *Boustrophedon transform*, which can be seen as either a transformation of a sequence of numbers or, equivalently, an exponential generating function. We define it in terms of the *Seidel triangle*, a number triangle similar to Pascal's triangle, and then prove the closed form $\mathcal{B}(f) = (\sec + \tan) f$.

We also define several related sequences, such as:

- the *zigzag numbers* E_n, counting the number of alternating permutations on a linearly ordered set with n elements; or, alternatively, the number of increasing binary trees with n elements
- the *Entringer numbers* $E_{n,k}$, which generalise the zigzag numbers and count the number of alternating permutations of $n+1$ elements that start with the k-th smallest element
- the *secant* and *tangent* numbers S_n and T_n, which are the series of numbers such that $\sec x = \sum_{n\geq 0} \frac{S(n)}{(2n)!} x^{2n}$ and $\tan x = \sum_{n\geq 1} \frac{T(n)}{(2n-1)!} x^{2n-1}$, respectively
- the *Euler numbers* $\mathcal{E}_n$ and *Euler polynomials* $\mathcal{E}_n(x)$, which are analogous to Bernoulli numbers and Bernoulli polynomials and satisfy many similar properties, which we also prove

Various relationships between these sequences are shown; notably we have $E_{2n} = S_n$ and $E_{2n+1} = T_{n+1}$ and $\mathcal{E}_{2n} = (-1)^n S_n$ and

$$T_n = \frac{(-1)^{n+1} 2^{2n} (2^{2n} - 1) B_{2n}}{2n}$$

where B_n denotes the Bernoulli numbers.

Reasonably efficient executable algorithms to compute the Boustrophedon transform and the above sequences are also given, including imperative ones for T_n and S_n using Imperative HOL.

Contents

1 Preliminary material

```
theory Boustrophedon_Transform_Library
imports
  "HOL-Computational_Algebra.Computational_Algebra"
  "Polynomial_Interpolation.Ring_Hom_Poly"
  "HOL-Library.FuncSet"
  "HOL-Library.Groups_Big_Fun"
begin
```

1.1 Miscellaneous

```
context comm_monoid_fun
begin

interpretation F: comm_monoid_set f "1"
  ..

lemma expand_superset_cong:
  assumes "finite A" and "⋀a. a ∉ A ⟹ g a = 1" and "⋀a. a ∈ A ⟹
g a = h a"
  shows "G g = F.F h A"
proof -
  have "G g = F.F g A"
    by (rule expand_superset) (use assms(1,2) in auto)
  also have "... = F.F h A"
    by (rule F.cong) (use assms(3) in auto)
  finally show ?thesis .
qed

lemma reindex_bij_witness:
  assumes "⋀x. h1 (h2 x) = x" "⋀x. h2 (h1 x) = x"
  assumes "⋀x. g1 (h1 x) = g2 x"
  shows   "G g1 = G g2"
proof -
  have "bij h1"
    using assms(1,2) by (metis bij_betw_def inj_def surj_def)
  have "G g1 = G (g1 ∘ h1)"
    by (rule reindex_cong[of h1]) (use ‹bij h1› in auto)
  also have "g1 ∘ h1 = g2"
    using assms(3) by auto
  finally show ?thesis .
qed

lemma distrib':
  assumes "⋀x. x ∉ A ⟹ g1 x = 1"
  assumes "⋀x. x ∉ A ⟹ g2 x = 1"
  assumes "finite A"
  shows "G (λx. f (g1 x) (g2 x)) = f (G g1) (G g2)"
proof (rule distrib)
```

```
  show "finite {x. g1 x ≠ 1}"
    by (rule finite_subset[OF _ assms(3)]) (use assms(1) in auto)
  show "finite {x. g2 x ≠ 1}"
    by (rule finite_subset[OF _ assms(3)]) (use assms(2) in auto)
qed

end

lemma of_rat_fact [simp]: "of_rat (fact n) = fact n"
  by (induction n) (auto simp: of_rat_mult of_rat_add)

lemma Pow_conv_subsets_of_size:
  assumes "finite A"
  shows   "Pow A = (⋃k≤card A. {X. X ⊆ A ∧ card X = k})"
  using assms by (auto intro: card_mono)
```

1.2 Linear orders

```
lemma (in linorder) linorder_linear_order [intro]: "linear_order {(x,y).
x ≤ y}"
  unfolding linear_order_on_def partial_order_on_def preorder_on_def antisym_def

            trans_def refl_on_def total_on_def by auto

lemma (in linorder) less_strict_linear_order_on [intro]: "strict_linear_order_on
A {(x,y). x < y}"
  unfolding strict_linear_order_on_def trans_def irrefl_def total_on_def
by auto

lemma (in linorder) greater_strict_linear_order_on [intro]: "strict_linear_order_on
A {(x,y). x > y}"
  unfolding strict_linear_order_on_def trans_def irrefl_def total_on_def
by auto

lemma strict_linear_order_on_asym_on:
  assumes "strict_linear_order_on A R"
  shows   "asym_on A R"
  using assms unfolding strict_linear_order_on_def
  by (meson asym_on_iff_irrefl_on_if_trans_on asym_on_subset top_greatest)

lemma strict_linear_order_on_antisym_on:
  assumes "strict_linear_order_on A R"
  shows   "antisym_on A R"
  using assms unfolding strict_linear_order_on_def
  by (meson antisym_on_def irreflD transD)

lemma monotone_on_imp_inj_on:
  assumes "monotone_on A R R' f" "strict_linear_order_on A {(x,y). R
x y}"
```

```
          "strict_linear_order_on (f ‘ A) {(x,y). R’ x y}"
  shows   "inj_on f A"
proof
  fix x y assume xy: "x ∈ A" "y ∈ A" "f x = f y"
  show "x = y"
  proof (rule ccontr)
    assume "x ≠ y"
    hence "R x y ∨ R y x"
      using assms(2) xy unfolding strict_linear_order_on_def total_on_def
by auto
    hence "R’ (f x) (f y) ∨ R’ (f y) (f x)"
      using assms(1) xy(1,2) by (auto simp: monotone_on_def)
    thus False
      using xy(3) assms(3) unfolding strict_linear_order_on_def irrefl_def
      by auto
  qed
qed

lemma monotone_on_inv_into:
  assumes "monotone_on A R R’ f" "strict_linear_order_on A {(x,y). R
x y}"
          "strict_linear_order_on (f ‘ A) {(x,y). R’ x y}"
  shows   "monotone_on (f ‘ A) R’ R (inv_into A f)"
  unfolding monotone_on_def
proof safe
  fix x y assume xy: "x ∈ A" "y ∈ A" "R’ (f x) (f y)"
  have "inj_on f A"
    using assms(1,2,3) by (rule monotone_on_imp_inj_on)
  have "f x ≠ f y"
    using xy assms(3) by (auto simp: strict_linear_order_on_def irrefl_def)
  have "¬R y x"
  proof
    assume "R y x"
    hence "R’ (f y) (f x)"
      using assms(1) xy by (auto simp: monotone_on_def)
    thus False
      using xy strict_linear_order_on_antisym_on[OF assms(3)] ‹f x ≠
f y›
      by (auto simp: antisym_on_def)
  qed
  hence "R x y"
    using assms(2) xy ‹f x ≠ f y› by (auto simp: strict_linear_order_on_def
total_on_def)
  thus "R (inv_into A f (f x)) (inv_into A f (f y))"
    by (subst (1 2) inv_into_f_f) (use xy ‹inj_on f A› in auto)
qed

lemma sorted_wrt_imp_distinct:
  assumes "sorted_wrt R xs" "⋀x. x ∈ set xs ⟹ ¬R x x"
```

```
  shows   "distinct xs"
  using assms by (induction R xs rule: sorted_wrt.induct) auto

lemma strict_linear_order_on_finite_has_least:
  assumes "strict_linear_order_on A R" "finite A" "A ≠ {}"
  shows   "∃x∈A. ∀y∈A-{x}. (x,y) ∈ R"
  using assms(2,1,3)
proof (induction A rule: finite_psubset_induct)
  case (psubset A)
  from ‹A ≠ {}› obtain x where x: "x ∈ A"
    by blast
  show ?case
  proof (cases "A - {x} = {}")
    case True
    thus ?thesis
      by (intro bexI[of _ x]) (use x in auto)
  next
    case False
    have trans: "(x,z) ∈ R" if "(x,y) ∈ R" "(y,z) ∈ R" for x y z
      using psubset.prems that unfolding strict_linear_order_on_def trans_def
by blast
    have *: "strict_linear_order_on (A - {x}) R"
      using psubset.prems(1) by (auto simp: strict_linear_order_on_def
total_on_def)
    have "∃z∈A-{x}. ∀y∈A-{x}-{z}. (z,y) ∈ R"
      by (rule psubset.IH) (use x False * in auto)
    then obtain z where z: "z ∈ A - {x}" "⋀y. y ∈ A - {x, z} ⟹ (z,y)
∈ R"
      by blast
    have "(x, z) ∈ R ∨ (z, x) ∈ R"
      using psubset.prems x z unfolding strict_linear_order_on_def total_on_def
      by auto
    thus ?thesis
    proof
      assume "(x, z) ∈ R"
      thus ?thesis
        using x z by (auto intro!: bexI[of _ x] intro: trans)
    next
      assume "(z, x) ∈ R"
      thus ?thesis
        using x z by (auto intro!: bexI[of _ z] intro: trans)
    qed
  qed
qed

lemma strict_linear_orderE_sorted_list:
  assumes "strict_linear_order_on A R" "finite A"
  obtains xs where "sorted_wrt (λx y. (x,y) ∈ R) xs" "set xs = A" "distinct
xs"
```

```
proof -
  have "∃xs. sorted_wrt (λx y. (x,y) ∈ R) xs ∧ set xs = A"
    using assms(2,1)
  proof (induction A rule: finite_psubset_induct)
    case (psubset A)
    show ?case
    proof (cases "A = {}")
      case False
      then obtain x where x: "x ∈ A" "⋀y. y ∈ A - {x} ⟹ (x,y) ∈ R"
        using strict_linear_order_on_finite_has_least[OF psubset.prems
psubset.hyps(1)] by blast
      have *: "strict_linear_order_on (A - {x}) R"
        using psubset.prems by (auto simp: strict_linear_order_on_def
total_on_def)
      have "∃xs. sorted_wrt (λx y. (x,y) ∈ R) xs ∧ set xs = A - {x}"
        by (rule psubset.IH) (use x * in auto)
      then obtain xs where xs: "sorted_wrt (λx y. (x,y) ∈ R) xs" "set
xs = A - {x}"
        by blast
      have "sorted_wrt (λx y. (x,y) ∈ R) (x # xs)" "set (x # xs) = A"
        using x xs by auto
      thus ?thesis
        by blast
    qed auto
  qed
  then obtain xs where xs: "sorted_wrt (λx y. (x,y) ∈ R) xs" "set xs
= A"
    by blast
  from xs(1) have "distinct xs"
    by (rule sorted_wrt_imp_distinct) (use assms in ‹auto simp: strict_linear_order_on_def
irrefl_def›)
  with xs show ?thesis
    using that by blast
qed

lemma sorted_wrt_strict_linear_order_unique:
  assumes R: "strict_linear_order_on A R"
  assumes "sorted_wrt (λx y. (x,y) ∈ R) xs" "sorted_wrt (λx y. (x,y)
∈ R) ys"
  assumes "set xs ⊆ A" "set xs = set ys"
  shows   "xs = ys"
  using assms(2-)
proof (induction xs arbitrary: ys)
  case (Cons x xs ys')
  from Cons.prems obtain y ys where [simp]: "ys' = y # ys"
    by (cases ys') auto
  have "set ys' ⊆ A"
    unfolding ‹set (x#xs) = set ys'›[symmetric] by fact
  have [simp]: "(z, z) ∉ R" for z
```

```
    using R by (auto simp: strict_linear_order_on_def irrefl_def)
  have "distinct (x # xs)"
    by (rule sorted_wrt_imp_distinct[OF <sorted_wrt _ (x#xs)>]) auto
  hence "x ∉ set xs"
    by auto
  have "distinct ys'"
    by (rule sorted_wrt_imp_distinct[OF <sorted_wrt _ ys'>]) auto
  hence "y ∉ set ys"
    by auto

  have *: "(x,y) ∈ R ∨ x = y ∨ (y,x) ∈ R"
    using R Cons.prems unfolding total_on_def by auto
  have "x = y"
    by (rule ccontr)
      (use Cons.prems strict_linear_order_on_asym_on[OF R] *
           <set ys' ⊆ A> <x ∉ set xs> <y ∉ set ys>
       in <auto simp: insert_eq_iff asym_on_def>)
  moreover have "xs = ys"
    by (rule Cons.IH)
      (use Cons.prems <x = y> <x ∉ set xs> <y ∉ set ys> in <simp_all
add: insert_eq_iff>)
  ultimately show ?case
    by simp
qed auto

definition sorted_list_of_set_wrt :: "('a × 'a) set ⇒ 'a set ⇒ 'a list"
where
  "sorted_list_of_set_wrt R A =
     (THE xs. sorted_wrt (λx y. (x,y) ∈ R) xs ∧ distinct xs ∧ set xs
= A)"

lemma sorted_list_of_set_wrt:
  assumes "strict_linear_order_on A R" "finite A"
  shows   "sorted_wrt (λx y. (x,y) ∈ R) (sorted_list_of_set_wrt R A)"
          "distinct (sorted_list_of_set_wrt R A)"
          "set (sorted_list_of_set_wrt R A) = A"
proof -
  define P where "P = (λxs. sorted_wrt (λx y. (x,y) ∈ R) xs ∧ distinct
xs ∧ set xs = A)"
  have "∃xs. P xs"
    using strict_linear_orderE_sorted_list[OF assms] unfolding P_def by
blast
  moreover have "xs = ys" if "P xs" "P ys" for xs ys
    using sorted_wrt_strict_linear_order_unique[OF assms(1)] that
    unfolding P_def by blast
  ultimately have *: "∃!xs. P xs"
    by blast
  show "sorted_wrt (λx y. (x,y) ∈ R) (sorted_list_of_set_wrt R A)"
      "distinct (sorted_list_of_set_wrt R A)"
```

```
      "set (sorted_list_of_set_wrt R A) = A"
    using theI'[OF *] unfolding P_def sorted_list_of_set_wrt_def by blast+
qed

lemma sorted_list_of_set_wrt_eqI:
  assumes "strict_linear_order_on A R" "sorted_wrt (λx y. (x,y) ∈ R)
xs" "set xs = A"
  shows   "sorted_list_of_set_wrt R A = xs"
proof (rule sym, rule sorted_wrt_strict_linear_order_unique[OF assms(1,2)])
  have *: "finite A"
    unfolding assms(3) [symmetric] by simp
  show "sorted_wrt (λx y. (x, y) ∈ R) (sorted_list_of_set_wrt R A)"
      "set xs = set (sorted_list_of_set_wrt R A)"
    using assms(3) sorted_list_of_set_wrt[OF assms(1) *] by simp_all
qed (use assms in auto)

lemma strict_linear_orderE_bij_betw:
  assumes "strict_linear_order_on A R" "finite A"
  obtains f where
    "bij_betw f {0..<card A} A" "monotone_on {0..<card A} (<) (λx y. (x,y)
∈ R) f"
proof -
  obtain xs where xs: "sorted_wrt (λx y. (x,y) ∈ R) xs" "set xs = A"
"distinct xs"
    using strict_linear_orderE_sorted_list[OF assms] by blast
  have length_xs: "length xs = card A"
    using distinct_card[of xs] xs by simp
  define f where "f = (λi. xs ! i)"

  have "A = set xs"
    using xs by simp
  also have "... = {f i |i. i < card A}"
    by (simp add: set_conv_nth length_xs f_def)
  also have "... = f ‘ {0..<card A}"
    by auto
  finally have range: "f ‘ {0..<card A} = A"
    by blast

  show ?thesis
  proof (rule that[of f])
    show "monotone_on {0..<card A} (<) (λx y. (x, y) ∈ R) f"
      using xs length_xs by (auto simp: monotone_on_def f_def sorted_wrt_iff_nth_less)
    hence "inj_on f {0..<card A}"
      by (rule monotone_on_imp_inj_on) (use assms range in auto)
    with range show "bij_betw f {0..<card A} A"
      by (simp add: bij_betw_def)
  qed
qed
```

```
lemma strict_linear_orderE_bij_betw':
  assumes "strict_linear_order_on A R" "finite A"
  obtains f where "bij_betw f {1..card A} A" "monotone_on {1..card A}
(<) (λx y. (x,y) ∈ R) f"
proof -
  obtain f where f: "bij_betw f {0..<card A} A" "monotone_on {0..<card
A} (<) (λx y. (x,y) ∈ R) f"
    using strict_linear_orderE_bij_betw[OF assms] .
  have *: "bij_betw (λn. n - 1) {1..card A} {0..<card A}"
    by (rule bij_betwI[of _ _ _ "λn. n + 1"]) auto
  have "bij_betw (f ∘ (λn. n - 1)) {1..card A} A"
    by (rule bij_betw_trans[OF * f(1)])
  moreover have "monotone_on {1..card A} (<) (λx y. (x, y) ∈ R) (f ∘
(λn. n - 1))"
    using f(2) by (rule monotone_on_o) (auto simp: strict_mono_on_def)
  ultimately show ?thesis
    using that by blast
qed

lemma monotone_on_strict_linear_orderD:
  assumes "monotone_on A R R' f"
  assumes "strict_linear_order_on A {(x,y). R x y}" "strict_linear_order_on
(f ` A) {(x,y). R' x y}"
  assumes "x ∈ A" "y ∈ A"
  shows   "R' (f x) (f y) ⟷ R x y"
proof
  assume "R x y"
  thus "R' (f x) (f y)"
    using assms by (auto simp: monotone_on_def)
next
  assume *: "R' (f x) (f y)"
  have "¬R y x"
  proof
    assume "R y x"
    hence "R' (f y) (f x)"
      using assms by (auto simp: monotone_on_def)
    with * show False
      using assms strict_linear_order_on_asym_on[OF assms(3)]
      by (auto simp: asym_on_def)
  qed
  moreover have "x ≠ y"
    using assms * by (auto simp: strict_linear_order_on_def irrefl_def)
  ultimately show "R x y"
    using assms by (auto simp: strict_linear_order_on_def total_on_def)
qed
```

1.3 Polynomials, formal power series and Laurent series

```
lemma lead_coeff_pderiv: "lead_coeff (pderiv p) = of_nat (degree p) *
lead_coeff p"
  for p :: "'a::{comm_semiring_1,semiring_no_zero_divisors,semiring_char_0}
poly"
proof (cases "pderiv p = 0")
  case False
  hence "degree p > 0"
    by (simp add: pderiv_eq_0_iff)
  thus ?thesis
    by (subst coeff_pderiv) (auto simp: degree_pderiv)
next
  case True
  thus ?thesis
    by (simp add: pderiv_eq_0_iff)
qed

lemma of_nat_poly_pderiv:
  "map_poly (of_nat :: nat ⇒ 'a :: {semidom, semiring_char_0}) (pderiv
p) =
     pderiv (map_poly of_nat p)"
proof (induct p rule: pderiv.induct)
  case (1 a p)
  interpret of_nat_poly_hom: map_poly_comm_semiring_hom of_nat
    by standard auto
  show ?case using 1 unfolding pderiv.simps
    by (cases "p = 0") (auto simp: hom_distribs pderiv_pCons)
qed
```

```
lemma fps_mult_left_numeral_nth [simp]:
  "((numeral c :: 'a ::{comm_monoid_add, semiring_1} fps) * f) $ n = numeral
c * f $ n"
  by (simp add: numeral_fps_const)

lemma fps_mult_right_numeral_nth [simp]:
  "(f * (numeral c :: 'a ::{comm_monoid_add, semiring_1} fps)) $ n = f
$ n * numeral c"
  by (simp add: numeral_fps_const)

lemma fps_shift_Suc_times_fps_X [simp]:
  fixes f :: "'a::{comm_monoid_add,mult_zero,monoid_mult} fps"
  shows "fps_shift (Suc n) (f * fps_X) = fps_shift n f"
  by (intro fps_ext) (simp add: nth_less_subdegree_zero)

lemma fps_shift_Suc_times_fps_X' [simp]:
```

```
  fixes f :: "'a::{comm_monoid_add,mult_zero,monoid_mult} fps"
  shows "fps_shift (Suc n) (fps_X * f) = fps_shift n f"
  by (intro fps_ext) (simp add: nth_less_subdegree_zero)

lemma fps_nth_inverse:
  fixes f :: "'a :: division_ring fps"
  assumes "fps_nth f 0 ≠ 0" "n > 0"
  shows   "fps_nth (inverse f) n = -(∑i=0..<n. inverse f $ i * f $ (n
- i)) / f $ 0"
proof -
  have "inverse f * f = 1"
    using assms by (simp add: inverse_mult_eq_1)
  also have "fps_nth ... n = 0"
    using ‹n > 0› by simp
  also have "fps_nth (inverse f * f) n = (∑i=0..n. inverse f $ i * f
$ (n - i))"
    by (simp add: fps_mult_nth)
  also have "{0..n} = insert n {0..<n}"
    by auto
  also have "(∑i∈... inverse f $ i * f $ (n - i)) =
             inverse f $ n * f $ 0 + (∑i=0..<n. inverse f $ i * f $ (n
- i))"
    by (subst sum.insert) auto
  finally show "inverse f $ n = -(∑i=0..<n. inverse f $ i * f $ (n -
i)) / f $ 0"
    using assms by (simp add: field_simps add_eq_0_iff)
qed

lemma fps_compose_of_poly:
  fixes p :: "'a :: idom poly"
  assumes [simp]: "fps_nth f 0 = 0"
  shows "fps_compose (fps_of_poly p) f = poly (map_poly fps_const p) f"
  by (induction p)
     (simp_all add: fps_of_poly_pCons fps_compose_mult_distrib fps_compose_add_distrib
                    algebra_simps)

lemma fps_nth_compose_linear:
  fixes f :: "'a :: comm_ring_1 fps"
  shows "fps_nth (fps_compose f (fps_const c * fps_X)) n = c ^ n * fps_nth
f n"
  by (subst fps_compose_linear) auto

lemma fps_nth_compose_uminus:
  fixes f :: "'a :: comm_ring_1 fps"
  shows "fps_nth (fps_compose f (-fps_X)) n = (-1) ^ n * fps_nth f n"
  using fps_nth_compose_linear[of f "-1" n] by (simp flip: fps_const_neg)

lemma fps_shift_compose_linear:
  fixes f :: "'a :: comm_ring_1 fps"
```

```
  shows "fps_shift n (fps_compose f (fps_const c * fps_X)) = fps_const
(c ^ n) * fps_compose (fps_shift n f) (fps_const c * fps_X)"
  by (auto simp: fps_eq_iff fps_nth_compose_linear power_add)

lemma fps_compose_shift_linear:
  fixes f :: "'a :: field fps"
  assumes "c ≠ 0"
  shows "fps_compose (fps_shift n f) (fps_const c * fps_X) =
           fps_const (1 / c ^ n) * fps_shift n (fps_compose f (fps_const
c * fps_X))"
  using assms by (auto simp: fps_eq_iff fps_nth_compose_linear power_add)
```

```
lemma fls_compose_fps_sum [simp]:
  assumes [simp]: "H ≠ 0" "fps_nth H 0 = 0"
  shows   "fls_compose_fps (∑x∈A. F x) H = (∑x∈A. fls_compose_fps
(F x) H)"
  by (induction A rule: infinite_finite_induct) (auto simp: fls_compose_fps_add)

lemma divide_fps_eqI:
  assumes "F * G = (H :: 'a :: field fps)" "H ≠ 0 ∨ G ≠ 0 ∨ F = 0"
  shows   "H / G = F"
proof (cases "G = 0")
  case True
  with assms show ?thesis
    by auto
next
  case False
  have "(F * G) / G = F"
    by (rule fps_divide_times_eq) (use False in auto)
  thus ?thesis
    using assms by simp
qed

lemma fps_to_fls_sum [simp]: "fps_to_fls (∑x∈A. f x) = (∑x∈A. fps_to_fls
(f x))"
  by (induction A rule: infinite_finite_induct) auto

lemma fps_to_fls_sum_list [simp]: "fps_to_fls (sum_list fs) = (∑f←fs.
fps_to_fls f)"
  by (induction fs) auto

lemma fps_to_fls_sum_mset [simp]: "fps_to_fls (sum_mset F) = (∑f∈#F.
fps_to_fls f)"
  by (induction F) auto
```

lemma *fps_to_fls_prod [simp]: "fps_to_fls (∏x∈A. f x) = (∏x∈A. fps_to_fls (f x))"*
  by *(induction A rule: infinite_finite_induct) (auto simp: fls_times_fps_to_fls)*

lemma *fps_to_fls_prod_list [simp]: "fps_to_fls (prod_list fs) = (∏f←fs. fps_to_fls f)"*
  by *(induction fs) (auto simp: fls_times_fps_to_fls)*

lemma *fps_to_fls_prod_mset [simp]: "fps_to_fls (prod_mset F) = (∏f∈#F. fps_to_fls f)"*
  by *(induction F) (auto simp: fls_times_fps_to_fls)*

1.4 Power series of trigonometric functions

definition *fps_sec :: "'a :: field_char_0 ⇒ 'a fps"*
  where *"fps_sec c = inverse (fps_cos c)"*

lemma *fps_sec_deriv: "fps_deriv (fps_sec c) = fps_const c * fps_sec c * fps_tan c"*
  by *(simp add: fps_sec_def fps_tan_def fps_inverse_deriv fps_cos_deriv fps_divide_unit*
                power2_eq_square flip: fps_const_neg)

lemma *fps_sec_nth_0 [simp]: "fps_nth (fps_sec c) 0 = 1"*
  by *(simp add: fps_sec_def)*

lemma *fps_sec_square_conv_fps_tan_square:*
  "fps_sec c ^ 2 = (1 + fps_tan c ^ 2 :: 'a :: field_char_0 fps)"
proof -
  have *"fps_nth (fps_cos c) 0 ≠ fps_nth 0 0"*
    by *auto*
  hence *[simp]: "fps_cos c ≠ 0"*
    by *metis*
  have *"fps_to_fls (1 + fps_tan c ^ 2) =*
          fps_to_fls 1 + fps_to_fls (fps_sin c) ^ 2 / fps_to_fls (fps_cos c) ^ 2"
    by *(simp add: fps_tan_def field_simps fps_to_fls_power flip: fls_divide_fps_to_fls)*
  also have *"... = (fps_to_fls (fps_cos c ^ 2 + fps_sin c ^ 2)) / fps_to_fls (fps_cos c) ^ 2"*
    by *(simp add: field_simps fps_to_fls_power)*
  also have *"fps_cos c ^ 2 + fps_sin c ^ 2 = 1"*
    by *(rule fps_sin_cos_sum_of_squares)*
  also have *"fps_to_fls 1 / fps_to_fls (fps_cos c) ^ 2 = fps_to_fls (fps_sec c ^ 2)"*
    by *(simp add: fps_sec_def fps_to_fls_power field_simps flip: fls_inverse_fps_to_fls)*
  finally show *?thesis*
    by *(simp only: fps_to_fls_eq_iff)*
qed

```
definition fps_cosh :: "'a :: field_char_0 ⇒ 'a fps"
  where "fps_cosh c = fps_const (1/2) * (fps_exp c + fps_exp (-c))"

lemma fps_nth_cosh_0 [simp]: "fps_nth (fps_cosh c) 0 = 1"
  by (simp_all add: fps_cosh_def)

lemma fps_cos_conv_cosh: "fps_cos c = fps_cosh (i * c)"
  by (simp add: fps_cosh_def fps_cos_fps_exp_ii)

lemma fps_cosh_conv_cos: "fps_cosh c = fps_cos (i * c)"
  by (simp add: fps_cosh_def fps_cos_fps_exp_ii)

lemma fps_cosh_compose_linear [simp]:
  "fps_cosh (d::'a::field_char_0) oo (fps_const c * fps_X) = fps_cosh
(c * d)"
  by (simp add: fps_cosh_def fps_compose_add_distrib fps_compose_mult_distrib)

lemma fps_fps_cosh_compose_minus [simp]:
  "fps_compose (fps_cosh c) (-fps_X) = fps_cosh (-c :: 'a :: field_char_0)"
  by (simp add: fps_cosh_def fps_compose_add_distrib fps_compose_mult_distrib)

lemma fps_nth_cosh: "fps_nth (fps_cosh c) n = (if even n then c ^ n /
fact n else 0)"
proof -
  have "fps_nth (fps_cosh c) n = (c ^ n + (-c) ^ n) / (2 * fact n)"
    by (simp add: fps_cosh_def fps_exp_def fps_mult_left_const_nth add_divide_distrib
mult_ac)
  also have "c ^ n + (-c) ^ n = (if even n then 2 * c ^ n else 0)"
    by (auto simp: uminus_power_if)
  also have "... / (2 * fact n) = (if even n then c ^ n / fact n else
0)"
    by auto
  finally show ?thesis .
qed


definition fps_sech :: "'a :: field_char_0 ⇒ 'a fps"
  where "fps_sech c = inverse (fps_cosh c)"

lemma fps_nth_sech_0 [simp]: "fps_nth (fps_sech c) 0 = 1"
  by (simp_all add: fps_sech_def)

lemma fps_sec_conv_sech: "fps_sec c = fps_sech (i * c)"
  by (simp add: fps_sech_def fps_sec_def fps_cos_conv_cosh)

lemma fps_sech_conv_sec: "fps_sech c = fps_sec (i * c)"
  by (simp add: fps_sech_def fps_sec_def fps_cosh_conv_cos)
```

```
lemma fps_sech_compose_linear [simp]:
  "fps_sech (d::'a::field_char_0) oo (fps_const c * fps_X) = fps_sech
(c * d)"
  by (simp add: fps_sech_def fps_inverse_compose)

lemma fps_fps_sech_compose_minus [simp]:
  "fps_compose (fps_sech c) (-fps_X) = fps_sech (-c :: 'a :: field_char_0)"
  by (simp add: fps_sech_def fps_inverse_compose)


lemma fps_tan_deriv': "fps_deriv (fps_tan 1 :: 'a :: field_char_0 fps)
= 1 + fps_tan 1 ^ 2"
proof -
  have "fps_nth (fps_cos (1::'a)) 0 ≠ fps_nth 0 0"
    by auto
  hence [simp]: "fps_cos (1::'a) ≠ 0"
    by metis
  have "fps_to_fls (fps_deriv (fps_tan (1 :: 'a :: field_char_0))) =
          fps_to_fls 1 / fps_to_fls (fps_cos 1 ^ 2)"
    by (simp add: fls_deriv_fps_to_fls fps_tan_deriv flip: fls_divide_fps_to_fls)
  also have "1 = fps_cos 1 ^ 2 + fps_sin (1::'a) ^ 2"
    using fps_sin_cos_sum_of_squares[of "1::'a"] by simp
  also have "fps_to_fls ... / fps_to_fls (fps_cos 1 ^ 2) = fps_to_fls
(1 + fps_tan 1 ^ 2)"
    by (simp add: field_simps fps_tan_def power2_eq_square fls_times_fps_to_fls
            flip: fls_divide_fps_to_fls)
  finally show ?thesis
    by (simp only: fps_to_fls_eq_iff)
qed

lemma fps_tan_nth_0 [simp]: "fps_nth (fps_tan c) 0 = 0"
  by (simp add: fps_tan_def)


lemma fps_nth_sin_even:
  assumes "even n"
  shows   "fps_nth (fps_sin c) n = 0"
  using assms by (auto simp: fps_sin_def)

lemma fps_nth_cos_odd:
  assumes "odd n"
  shows   "fps_nth (fps_cos c) n = 0"
  using assms by (auto simp: fps_cos_def)

lemma fps_tan_odd: "fps_tan (-c) = -fps_tan c"
  by (simp add: fps_tan_def fps_sin_even fps_cos_odd fps_divide_uminus)

lemma fps_sec_even: "fps_sec (-c) = fps_sec c"
  by (simp add: fps_sec_def fps_cos_odd fps_divide_uminus)
```

```
lemma fps_sin_compose_linear [simp]: "fps_sin c oo (fps_const c' * fps_X)
= fps_sin (c * c')"
  by (rule fps_ext) (simp_all add: fps_sin_def fps_compose_linear power_mult_distrib)

lemma fps_sin_compose_uminus [simp]: "fps_sin c oo (-fps_X) = fps_sin
(-c)"
  using fps_sin_compose_linear[of c "-1"] by (simp flip: fps_const_neg
del: fps_sin_compose_linear)

lemma fps_cos_compose_linear [simp]: "fps_cos c oo (fps_const c' * fps_X)
= fps_cos (c * c')"
  by (rule fps_ext) (simp_all add: fps_cos_def fps_compose_linear power_mult_distrib)

lemma fps_cos_compose_uminus [simp]: "fps_cos c oo (-fps_X) = fps_cos
(-c)"
  using fps_cos_compose_linear[of c "-1"] by (simp flip: fps_const_neg
del: fps_cos_compose_linear)

lemma fps_tan_compose_linear [simp]: "fps_tan c oo (fps_const c' * fps_X)
= fps_tan (c * c')"
  by (simp add: fps_tan_def fps_divide_compose)

lemma fps_tan_compose_uminus [simp]: "fps_tan c oo (-fps_X) = fps_tan
(-c)"
  by (simp add: fps_tan_def fps_divide_compose)

lemma fps_sec_compose_linear [simp]: "fps_sec c oo (fps_const c' * fps_X)
= fps_sec (c * c')"
  by (simp add: fps_sec_def fps_inverse_compose)

lemma fps_sec_compose_uminus [simp]: "fps_sec c oo (-fps_X) = fps_sec
(-c)"
  by (simp add: fps_sec_def fps_inverse_compose)

lemma fps_nth_tan_even:
  assumes "even n"
  shows   "fps_nth (fps_tan c) n = 0"
proof -
  have "fps_tan c oo -fps_X = -fps_tan c"
    by (simp add: fps_tan_odd)
  hence "(fps_tan c oo -fps_X) $ n = (-fps_tan c) $ n"
    by (rule arg_cong)
  thus ?thesis using assms
    unfolding fps_eq_iff fps_nth_compose_uminus
    by (auto simp: minus_one_power_iff)
qed

lemma fps_nth_sec_odd:
```

```
  assumes "odd n"
  shows   "fps_nth (fps_sec c) n = 0"
proof -
  have "fps_sec c oo -fps_X = fps_sec c"
    by (simp add: fps_sec_even)
  hence "(fps_sec c oo -fps_X) $ n = (fps_sec c) $ n"
    by (rule arg_cong)
  thus ?thesis using assms
    unfolding fps_eq_iff fps_nth_compose_uminus
    by (auto simp: minus_one_power_iff)
qed

end
```

2 Alternating permutations

```
theory Alternating_Permutations
  imports "HOL-Combinatorics.Combinatorics" Boustrophedon_Transform_Library
begin
```

Given a strict linear order $<$ on some finite set $A = \{a_1, \ldots, a_n\}$ with $a_1 < \ldots < a_n$ we call a permutation π *alternating* if $f(a_1) > f(a_2) < f(a_3) > f_(a_4) \ldots$.

Since it is somewhat awkward to specify this for a function, we instead define what an alternating permutation is using the view that a permutation on A is simple the tuple $(f_(a_1), \ldots, f(a_n))$.

2.1 Alternating lists

Given a relation R, we say that a list $[x_1, \ldots, x_n]$ is R-*alternating* if we have $(x_i, x_{i+1}) \in R$ for any even i and $(x_{i+1}, x_i) \in R$ for any odd i.

In other words: if we view R as an order then the list alternates between "rises" and "falls", starting with a "fall".

```
fun alternating_list :: "('a × 'a) set ⇒ 'a list ⇒ bool" where
  "alternating_list R [] ⟷ True"
| "alternating_list R [x] ⟷ True"
| "alternating_list R (x # y # xs) ⟷ (y,x) ∈ R ∧ alternating_list (R⁻¹)
(y # xs)"

lemma alternating_list_Cons_iff:
  "alternating_list R (x # xs) ⟷ xs = [] ∨ ((hd xs, x) ∈ R ∧ alternating_list
(converse R) xs)"
  by (cases xs) auto

lemma alternating_list_append_iff:
  "alternating_list R (xs @ ys) ⟷ (let R' = if even (length xs) then
R else converse R in
```

```
    alternating_list R xs ∧ alternating_list R' ys ∧ (xs = [] ∨ ys =
[] ∨ (last xs, hd ys) ∈ R'))"
  by (induction R xs rule: alternating_list.induct)
    (auto simp: Let_def alternating_list_Cons_iff)
```

A reverse-alternating list is the same as an alternating list except that it starts with a "rise" instead of a "fall". Equivalently, a reverse-alternating list is an alternating list with respect to the converse relation.

```
abbreviation rev_alternating_list :: "('a × 'a) set ⇒ 'a list ⇒ bool"
where
  "rev_alternating_list R ≡ alternating_list (R⁻¹)"

lemma alternating_list_rev:
  "alternating_list R (rev xs) ⟷ alternating_list (if odd (length xs)
then R else converse R) xs"
  by (induction xs arbitrary: R)
    (auto simp: alternating_list_append_iff last_rev alternating_list_Cons_iff)

lemma alternating_list_map:
  assumes "alternating_list R xs"
  assumes "monotone_on (set xs) (λx y. (x, y) ∈ R) (λx y. (x, y) ∈ R')
f"
  shows   "alternating_list R' (map f xs)"
proof -
  define A where "A = set xs"
  have "(f x, f y) ∈ R'" if "(x, y) ∈ R" "x ∈ A" "y ∈ A" for x y
    using assms(2) that by (auto simp: monotone_on_def A_def)
  moreover have "set xs ⊆ A"
    by (simp add: A_def)
  ultimately show ?thesis using assms(1)
    by (induction R xs arbitrary: R' rule: alternating_list.induct) auto
qed

lemma alternating_list_map_iff:
  assumes "monotone_on (set xs) (λx y. (x, y) ∈ R) (λx y. (x, y) ∈ R')
f"
  assumes "strict_linear_order_on (set xs) R" "strict_linear_order_on
(f ` set xs) R'"
  shows   "alternating_list R' (map f xs) ⟷ alternating_list R xs"
proof
  assume "alternating_list R xs"
  thus "alternating_list R' (map f xs)"
    by (intro alternating_list_map) (use assms in simp_all)
next
  assume "alternating_list R' (map f xs)"
  hence "alternating_list R (map (inv_into (set xs) f) (map f xs))"
  proof (rule alternating_list_map)
    have "monotone_on (f ` set xs) (λx y. (x, y) ∈ R') (λx y. (x, y)
∈ R) (inv_into (set xs) f)"
```

```
      by (rule monotone_on_inv_into) (use assms in simp_all)
    thus "monotone_on (set (map f xs)) (λx y. (x, y) ∈ R') (λx y. (x,
y) ∈ R) (inv_into (set xs) f)"
      by simp
  qed
  also have "map (inv_into (set xs) f) (map f xs) = map (λx. x) xs"
    unfolding map_map o_def
    by (intro map_cong inv_into_f_f monotone_on_imp_inj_on[OF assms(1)])
       (use assms in simp_all)
  finally show "alternating_list R xs"
    by simp
qed
```

2.2 The set of alternating permutations on a set

```
definition alternating_permutations_of_set :: "('a × 'a) set ⇒ 'a set
⇒ 'a list set" where
  "alternating_permutations_of_set R A = {ys∈permutations_of_set A. alternating_list
R ys}"

lemma finite_alternating_permutations_of_set [intro]: "finite (alternating_permutations_of
R A)"
  unfolding alternating_permutations_of_set_def by simp

lemma alternating_permutations_of_set_code [code]:
  "alternating_permutations_of_set R A = Set.filter (alternating_list
R) (permutations_of_set A)"
  by (simp add: alternating_permutations_of_set_def Set.filter_def)

abbreviation rev_alternating_permutations_of_set :: "('a × 'a) set ⇒
'a set ⇒ 'a list set" where
  "rev_alternating_permutations_of_set R A ≡ alternating_permutations_of_set
(converse R) A"

definition alt_permutes ("_ alt'_permutes_ _" [40,0,40] 41) where
  "f alt_permutes_R A ⟷ f permutes A ∧ alternating_list R (map f (sorted_list_of_set_wrt
R A))"

abbreviation rev_alt_permutes ("_ rev'_alt'_permutes_ _" [40,0,40] 41)
where
  "f rev_alt_permutes_R A ≡ f alt_permutes_converse R A"

abbreviation alt_permutes_less ("_ alt'_permutes _" [40,40] 41) where
  "f alt_permutes A ≡ f alt_permutes_{(x,y). x < y} A"

abbreviation rev_alt_permutes_less ("_ rev'_alt'_permutes _" [40,40] 41)
where
  "f rev_alt_permutes A ≡ f rev_alt_permutes_{(x,y). x < y} A"
```

```
lemma alternating_permutations_of_set_empty [simp]:
  "alternating_permutations_of_set R {} = {[]}"
  by (auto simp: alternating_permutations_of_set_def)

lemma alternating_permutations_of_set_singleton [simp]:
  "alternating_permutations_of_set R {x} = {[x]}"
  by (auto simp: alternating_permutations_of_set_def)

lemma bij_betw_alternating_permutations_of_set:
  assumes "monotone_on A (λx y. (x,y) ∈ R) (λx y. (x,y) ∈ R') f"
  assumes "strict_linear_order_on A R" "strict_linear_order_on (f ` A)
R'" "B = f  ` A"
  shows   "bij_betw (map f) (alternating_permutations_of_set R A) (alternating_permutations
R' B)"
proof -
  have "inj_on f A"
    by (rule monotone_on_imp_inj_on[OF assms(1)]) (use assms(2,3) in simp_all)
  have inj: "inj_on (map f) (alternating_permutations_of_set R A)"
    by (rule inj_on_mapI[OF inj_on_subset[OF <inj_on f A>]])
       (auto simp: alternating_permutations_of_set_def permutations_of_set_def)

  have "map f ` alternating_permutations_of_set R A = alternating_permutations_of_set
R' (f ` A)"
    (is "_ ` ?lhs = ?rhs")
  proof safe
    fix xs assume "xs ∈ ?lhs"
    thus "map f xs ∈ ?rhs" using assms
      by (auto simp: alternating_permutations_of_set_def permutations_of_set_def
distinct_map alternating_list_map
                    inj_on_subset[OF <inj_on f A>])
  next
    fix xs assume xs: "xs ∈ ?rhs"
    hence set_xs: "set xs = f ` A"
      by (auto simp: alternating_permutations_of_set_def permutations_of_set_def)
    define ys where "ys = map (inv_into A f) xs"
    have mono: "monotone_on (f ` A) (λx y. (x,y) ∈ R') (λx y. (x,y) ∈
R) (inv_into A f)"
      by (intro monotone_on_inv_into) (use assms in simp_all)
    hence inj': "inj_on (inv_into A f) (f ` A)"
      by (rule monotone_on_imp_inj_on) (use assms <inj_on f A> in simp_all)
    have "ys ∈ ?lhs" using xs mono <inj_on f A> inj' assms(2,3)
      by (auto simp: ys_def alternating_permutations_of_set_def permutations_of_set_def
distinct_map
               intro!: inj_on_subset[OF <inj_on f A>] alternating_list_map)
    moreover have "map f ys = map (λx. x) xs"
      unfolding ys_def map_map o_def
      by (intro map_cong inv_into_f_f) (use <inj_on f A> set_xs in auto)
```

```
    ultimately show "xs ∈ map f ‘ ?lhs"
      by auto
  qed
  with inj show ?thesis using ‹B = f ‘ A›
    unfolding bij_betw_def by blast
qed

lemma alternating_permutations_of_set_glue:
  assumes A: "finite A"
  assumes X: "X ⊆ A" and x: "x ∈ A - X" "⋀y. y ∈ A-{x} ⟹ (x,y) ∈
R"
  assumes xs: "xs ∈ alternating_permutations_of_set R X"
  assumes ys: "ys ∈ alternating_permutations_of_set R (A - X - {x})"
  defines "R' ≡ (if odd (card X) then R else R⁻¹)"
  shows   "rev xs @ [x] @ ys ∈ alternating_permutations_of_set R' A"
proof -
  have "set (xs @ ys) ⊆ A - {x}"
    using xs ys X x unfolding alternating_permutations_of_set_def permutations_of_set_def
    by auto
  hence *: "y ∈ A - {x}" if "y ∈ set (xs @ ys)" for y
    using that by blast
  have length_xs: "length xs = card X"
    using xs distinct_card[of xs]
    unfolding alternating_permutations_of_set_def permutations_of_set_def
by simp

  have "xs = [] ∨ (hd xs, x) ∈ R⁻¹"
    using x(2)[OF *, of "hd xs"] by (cases "xs = []") auto
  moreover have "ys = [] ∨ (hd ys, x) ∈ R⁻¹"
    using x(2)[OF *, of "hd ys"] by (cases "ys = []") auto
  ultimately have "alternating_list R' (rev xs @ [x] @ ys)"
    using xs ys unfolding alternating_list_append_iff R'_def alternating_permutations_of_se
    by (simp add: length_xs alternating_list_rev last_rev)
  moreover have "rev xs @ [x] @ ys ∈ permutations_of_set A"
    using xs ys X x unfolding alternating_permutations_of_set_def permutations_of_set_def
    by auto
  ultimately show ?thesis
    unfolding alternating_permutations_of_set_def by blast
qed

lemma alternating_permutations_of_set_split:
  assumes A: "finite A"
  assumes z: "z ∈ A"
  assumes zs: "zs ∈ alternating_permutations_of_set R A"
  assumes k: "k < length zs" "zs ! k = z"
  defines "R' ≡ (if odd k then R else converse R)"
  obtains xs ys where
    "zs = rev xs @ [z] @ ys" "alternating_list R' xs" "alternating_list
R' ys"
```

```
    "distinct xs" "distinct ys" "length xs = k"
proof -
  have "set zs = A" "distinct zs"
    using zs unfolding alternating_permutations_of_set_def permutations_of_set_def
by blast+
  with z(1) have "z ∈ set zs"
    by blast
  then obtain xs ys where zs_eq: "zs = xs @ z # ys"
    by (metis in_set_conv_decomp)

  have "zs ! length xs = z" "length xs < length zs"
    using k by (simp_all add: zs_eq)
  with ‹distinct zs› and k have k_eq: "k = length xs"
    using distinct_conv_nth by blast

  have "alternating_list R (xs @ z # ys)"
    using zs by (simp add: alternating_permutations_of_set_def zs_eq)
  hence "alternating_list R' (rev xs)" "alternating_list R' ys"
    by (auto simp: alternating_list_append_iff alternating_list_Cons_iff
                   Let_def k_eq R'_def alternating_list_rev)
  thus ?thesis
    using ‹distinct zs› k_eq by (intro that[of "rev xs" ys]) (simp_all
add: zs_eq)
qed

lemma inj_on_glue_alternating_permutations_of_set:
  fixes A :: "'a set"
  assumes x: "x ∈ A" "⋀y. y ∈ A - {x} ⟹ (x, y) ∈ R"
  defines "P ≡ (λX::'a set. alternating_permutations_of_set R X)"
  shows   "inj_on (λ(xs, ys). rev xs @ [x] @ ys) ((⋃X∈Pow (A-{x}). P
X × P (A - X - {x})))"
proof (rule inj_onI, clarify, goal_cases)
  case (1 xs1 ys1 xs2 ys2)
  from 1 have "rev xs1 @ x # ys1 = rev xs2 @ x # ys2"
    by simp
  moreover have "x ∉ set xs1" "x ∉ set xs2" "x ∉ set ys1" "x ∉ set
ys2"
    using 1 unfolding P_def alternating_permutations_of_set_def permutations_of_set_def
    by auto
  ultimately show "xs1 = xs2 ∧ ys1 = ys2"
    by (subst (asm) append_Cons_eq_iff) auto
qed
```

2.3 Zigzag numbers

The zigzag numbers E_n count the number of alternating permutations on a linearly ordered set with n elements. Note that varying conventions exist; e.g. these are also sometimes also called "Euler numbers" or "Euler zigzag numbers". [3, A000111]

In our formalisation, "Euler numbers" are something closely related but different, following the conventions of ProofWiki and Mathematica.

It is easy to see that we can w.l.o.g. assume that the set in question is the integers from 1 to n and the order in question is the natural order $<$.

```
definition zigzag_number :: "nat ⇒ nat" where
  "zigzag_number n = card (alternating_permutations_of_set {(x,y). x <
y} {1..n})"

lemma zigzag_number_0 [simp]: "zigzag_number 0 = 1"
  and zigzag_number_1 [simp]: "zigzag_number (Suc 0) = 1"
  by (simp_all add: zigzag_number_def)

lemma card_alternating_permutations_of_set:
  assumes "strict_linear_order_on A R" "finite A"
  shows   "card (alternating_permutations_of_set R A) = zigzag_number
(card A)"
proof -
  obtain f :: "nat ⇒ 'a" where f:
    "bij_betw f {1..card A} A" "monotone_on {1..card A} (<) (λx y. (x,y)
∈ R) f"
    using strict_linear_orderE_bij_betw'[OF assms] .
  define P1 where "P1 = alternating_permutations_of_set {(x, y). x <
y} {1..card A}"
  define P2 where "P2 = alternating_permutations_of_set R A"

  have "zigzag_number (card A) = card P1"
    by (simp add: zigzag_number_def P1_def)
  also have "bij_betw (map f) P1 P2"
    unfolding P1_def P2_def
  proof (rule bij_betw_alternating_permutations_of_set)
    show "strict_linear_order_on (f ` {1..card A}) R" and "A = f ` {1..card
A}"
      using assms f(1) by (simp_all add: bij_betw_def)
  qed (use f(2) in auto)
  hence "card P1 = card P2"
    by (rule bij_betw_same_card)
  finally show ?thesis
    by (simp add: P2_def)
qed
```

The zigzag numbers satisfy the Catalan-like recurrence

$$2E_{n+1} = \sum_{k=0}^{n} \binom{n}{k} E_k E_{n-k} \ .$$

The idea behind the proof is to look at a linearly ordered set A of size $n+1$ (with $n > 0$) and its largest element x. We now do the following:

1. Pick a number $0 \le k \le n$.

2. Pick a subset $X \subseteq A \setminus \{x\}$ of elements to occur to the left of A in our permutation. We have $\binom{n}{k}$ choices for this.

3. Pick an alternating permutation *xs* of X and a reverse-alternating permutation of *ys* of $A \setminus (X \cup \{x\})$. We have E_k and E_{n-k} choices for this, respectively.

4. Return the permutation *rev xs @ [x] @ ys*

This process constructs exactly all alternating and reverse-alternating permutations on A. Moreover, the alternating and reverse-alternating permutations of A are disjoint and have the same cardinality since $|A| \geq 2$.

Thus if we sum the number of possibilities we counted above over all k, we obtain exactly $2E_{n+1}$.

```
theorem zigzag_number_Suc:
  assumes "n > 0"
  shows   "2 * zigzag_number (Suc n) =
             (∑k≤n. (n choose k) * (zigzag_number k * zigzag_number
(n - k)))"
proof -
  define P where "P = (λX::nat set. alternating_permutations_of_set {(x,y).
x < y} X)"
  define P' where "P' = (λX::nat set. alternating_permutations_of_set
{(x,y). x > y} X)"
  define glue :: "nat list × nat list ⇒ nat list" where "glue = (λ(xs,
ys). rev xs @ [1] @ ys)"
  define A where "A = {1..n+1}"
  have [intro]: "finite (P X)" "finite (P' X)" for X
    unfolding P_def P'_def by auto
  let ?less = "{(x,y). x < (y::nat)}"
  let ?greater = "{(x,y). x > (y::nat)}"
  have [simp]: "converse ?less = ?greater" "converse ?greater = ?less"
    by (auto simp: converse_def)
  define R where "R = (λk. if odd (k::nat) then ?less else ?greater)"

  have disjoint: "P A ∩ P' A = {}"
  proof -
    have False if "zs ∈ P A" "zs ∈ P' A" for zs
    proof -
      have zs: "set zs = A" "distinct zs" "alternating_list ?less zs"
"alternating_list ?greater zs"
        using that
        unfolding P_def P'_def alternating_permutations_of_set_def permutations_of_set_def
        by simp_all
      have "length zs ≥ 2"
        using distinct_card[of zs] zs <n > 0> by (simp add: A_def)
      then obtain x y zs' where zs_eq: "zs = x # y # zs'"
        by (auto simp: Suc_le_length_iff numeral_2_eq_2)
```

```
      show False
        using zs by (simp add: zs_eq)
    qed
    thus ?thesis
      by blast
  qed

  have "card (glue ` (⋃X∈Pow (A-{1}). P X × P (A - X - {1}))) =
        card (⋃X∈Pow (A-{1}). P X × P (A - X - {1}))"
    unfolding glue_def P_def
    by (rule card_image, rule inj_on_glue_alternating_permutations_of_set)
       (auto simp: A_def)

  also have "glue ` (⋃X∈Pow (A-{1}). P X × P (A - X - {1})) = P A ∪
P' A"
  proof (rule antisym)
    have "glue (xs, ys) ∈ P A ∪ P' A"
      if X: "X ∈ Pow (A - {1})" and xs: "xs ∈ P X" and ys: "ys ∈ P (A
- X - {1})" for X xs ys
    proof -
      have "rev xs @ [1] @ ys ∈ alternating_permutations_of_set
                (if odd (card X) then ?less else ?less⁻¹) A"
        by (rule alternating_permutations_of_set_glue[of A X 1 ?less xs
ys])
           (use X xs ys in <auto simp: A_def P_def>)
      hence "glue (xs, ys) ∈ (if odd (card X) then P A else P' A)"
        by (auto simp: glue_def P_def P'_def)
      also have "... ⊆ P A ∪ P' A"
        by auto
      finally show "glue (xs, ys) ∈ P A ∪ P' A" .
    qed
    thus "glue ` (⋃X∈Pow (A-{1}). P X × P (A - X - {1})) ⊆ P A ∪ P'
A"
      by blast
  next
    have "zs ∈ glue ` (⋃X∈Pow (A-{1}). P X × P (A - X - {1}))" if zs:
"zs ∈ P A ∪ P' A" for zs
    proof -
      from zs have set_zs: "set zs = A" and "distinct zs"
        by (auto simp: P_def P'_def alternating_permutations_of_set_def
permutations_of_set_def)
      have "length zs = Suc n"
        using set_zs <distinct zs> distinct_card[of zs] by (simp add:
A_def)
      from set_zs have "1 ∈ set zs"
        by (auto simp: A_def)
      then obtain k where k: "k < length zs" "zs ! k = 1"
        by (meson in_set_conv_nth)
      define R' where "R' = (if zs ∈ P A then ?less else ?greater)"
```

```
      obtain xs ys where xs_ys:
        "zs = rev xs @ [1] @ ys" "alternating_list (if odd k then R' else
R'⁻¹) xs"
        "alternating_list (if odd k then R' else R'⁻¹) ys" "distinct xs"
"distinct ys" "length xs = k"
        by (rule alternating_permutations_of_set_split[of A 1 zs R' k])
           (use k zs in ‹auto simp: A_def R'_def P_def P'_def›)
      have set_xs: "set xs ⊆ A - {1}"
        using ‹distinct zs› unfolding set_zs [symmetric] xs_ys(1) by (auto
simp: xs_ys(1))
      have set_ys: "set ys = A - set xs - {1}"
        using ‹distinct zs› unfolding set_zs [symmetric] xs_ys(1) by (auto
simp: xs_ys(1))
      have "odd k ⟷ zs ∈ P A"
      proof -
        have 1: "xs ≠ [] ∨ ys ≠ []"
          using xs_ys(1) ‹n > 0› ‹length zs = Suc n› by (auto simp: A_def)
        have 2: "x ∈ A - {1}" if "x ∈ set (xs @ ys)" for x
        proof -
          have "x ∈ set (xs @ ys)"
            using that by simp
          also have "... ⊆ set zs - {1}"
            using ‹distinct zs› by (auto simp add: xs_ys(1))
          finally show ?thesis
            by (simp add: set_zs)
        qed
        have 3: "xs = [] ∨ 1 < hd xs"
          using 2[of "hd xs"] by (cases "xs = []") (auto simp: hd_in_set
A_def)
        have 4: "ys = [] ∨ 1 < hd ys"
          using 2[of "hd ys"] by (cases "ys = []") (auto simp: hd_in_set
A_def)
        have "alternating_list R' zs"
          using zs by (auto simp: R'_def P_def P'_def alternating_permutations_of_set_def)
        thus ?thesis
          using 1 3 4 xs_ys(2,3) ‹length xs = k› zs
          by (auto simp: xs_ys(1) alternating_list_append_iff alternating_list_Cons_iff
                         alternating_list_rev Let_def R'_def last_rev
split: if_splits)
      qed
      hence "(if odd k then R' else R'⁻¹) = ?less"
        by (auto simp: R'_def)
      with xs_ys and set_ys have "zs = glue (xs, ys)" "xs ∈ P (set xs)"
"ys ∈ P (A - set xs - {1})"
        by (simp_all add: glue_def P_def alternating_permutations_of_set_def
permutations_of_set_def)
      thus "zs ∈ glue ` (⋃X∈Pow (A-{1}). P X × P (A - X - {1}))"
        using set_xs by blast
    qed
```

```
    thus "P A ∪ P' A ⊆ glue ` (⋃X∈Pow (A-{1}). P X × P (A - X - {1}))"
      by blast
  qed

  also have "card (P A ∪ P' A) = card (P A) + card (P' A)"
    by (subst card_Un_disjoint) (use disjoint in auto)
  also have "card (P A) = zigzag_number (Suc n)"
    unfolding P_def by (subst card_alternating_permutations_of_set) (auto
simp: A_def)
  also have "card (P' A) = zigzag_number (Suc n)"
    unfolding P'_def by (subst card_alternating_permutations_of_set) (auto
simp: A_def)

  also have "card (⋃X∈Pow (A-{1}). P X × P (A - X - {1})) =
               (∑X∈Pow (A - {1}). card (P X × P (A - X - {1})))"
  proof (intro card_UN_disjoint ballI impI)
    fix X Y assume "X ∈ Pow (A - {1})" "Y ∈ Pow (A - {1})" "X ≠ Y"
    show "P X × P (A - X - {1}) ∩ P Y × P (A - Y - {1}) = {}"
      using <X ≠ Y> unfolding P_def alternating_permutations_of_set_def
permutations_of_set_def
      by blast
  qed (auto simp: A_def)
  also have "... = (∑X∈Pow (A - {1}). zigzag_number (card X) * zigzag_number
(n - card X))"
  proof (rule sum.cong)
    fix X assume X: "X ∈ Pow (A - {1})"
    have [simp]: "finite X"
      by (rule finite_subset[of _ A]) (use X in <auto simp: A_def>)
    have "card (P X × P (A - X - {1})) = card (P X) * card (P (A - X
- {1}))"
      by (rule card_cartesian_product)
    also have "card (P X) = zigzag_number (card X)"
      unfolding P_def by (rule card_alternating_permutations_of_set) (use
X in auto)
    also have "card (P (A - X - {1})) = zigzag_number (card (A - X - {1}))"
      unfolding P_def by (rule card_alternating_permutations_of_set) (use
X in <auto simp: A_def>)
    also have "card (A - X - {1}) = card (A - X) - 1"
      using X by (subst card_Diff_subset) (auto simp: A_def)
    also have "card (A - X) = card A - card X"
      using X finite_subset[of X A] by (subst card_Diff_subset) (auto
simp: A_def)
    also have "card A = n + 1"
      by (simp add: A_def)
    finally show "card (P X × P (A - X - {1})) =
                    zigzag_number (card X) * zigzag_number (n - card X)"
      by simp
  qed auto
```

```
  also have "Pow (A - {1}) = (⋃k≤n. {X∈Pow (A-{1}). card X = k})"
    by (subst Pow_conv_subsets_of_size) (simp_all add: A_def)
  also have "(∑X∈... zigzag_number (card X) * zigzag_number (n - card
X)) =
               (∑k≤n. card {X. X ⊆ A-{1} ∧ card X = k} * (zigzag_number
k * zigzag_number (n - k)))"
    by (subst sum.UNION_disjoint) (auto simp: A_def)
  also have "... = (∑k≤n. (n choose k) * (zigzag_number k * zigzag_number
(n - k)))"
    using n_subsets[of "A - {1}"] by (simp add: A_def)
  finally show ?thesis
    by simp
qed
```

The exponential generating function of the zigzag numbers is:

$$f(x) = \sum_{n \geq 0} \frac{E_n}{n!} x^n = \sec x + \tan x$$

This follows from the fact that by the above recurrence for E_n, both f and $\sin + \tan$ satisfy the ordinary differential equation $2f'(x) = 1 + f(x)^2$

```
corollary exponential_generating_function_zigzag_number:
  "Abs_fps (λn. of_nat (zigzag_number n) / fact n :: 'a :: field_char_0)
= fps_sec 1 + fps_tan 1"
proof -
  define F where "F ≡ Abs_fps (λn. of_nat (zigzag_number n) / fact n
:: 'a)"
  define G where "G ≡ (fps_sec 1 + fps_tan 1 :: 'a fps)"
  have [simp]: "fps_nth F 0 = 1" "fps_nth F (Suc 0) = 1"
    by (simp_all add: F_def)
  have F_Suc: "fps_nth F (Suc n) = (∑k≤n. fps_nth F k * fps_nth F (n
- k)) / (2 * of_nat (n + 1))"
    if "n > 0" for n
  proof -
    have "2 * fps_nth F (Suc n) = of_nat (2 * zigzag_number (Suc n)) /
fact (Suc n)"
      by (simp add: F_def)
    also have "... = (∑k≤n. fps_nth F k * fps_nth F (n - k)) / of_nat
(n + 1)"
      by (subst zigzag_number_Suc) (use that in ‹auto simp: F_def mult_ac
binomial_fact sum_divide_distrib›)
    finally show ?thesis
      unfolding of_nat_mult by (simp add: divide_simps mult_ac del: of_nat_Suc)
  qed
  have "2 * fps_deriv F = 1 + F ^ 2"
    by (rule fps_ext) (auto simp: fps_nth_power_0 F_Suc fps_square_nth
divide_simps simp del: of_nat_Suc)
  have "2 * fps_deriv G = 1 + G ^ 2"
    using fps_sec_square_conv_fps_tan_square[where ?'a = 'a]
```

```
    by (simp add: G_def fps_sec_deriv fps_tan_deriv' power2_eq_square
algebra_simps)

  have "fps_nth F n = fps_nth G n" for n
  proof (induction rule: less_induct)
    case (less n)
    show ?case
    proof (cases "n = 0")
      case True
      thus ?thesis
        by (auto simp: F_def G_def)
    next
      case n: False
      have "2 * of_nat n * fps_nth F n = fps_nth (2 * fps_deriv F) (n
- 1)"
        using n by simp
      also have "2 * fps_deriv F = 1 + F ^ 2"
        by fact
      also have "fps_nth (1 + F ^ 2) (n - 1) = fps_nth 1 (n - 1) + (∑k≤n-1.
F $ k * F $ (n - Suc k))"
        using n by (simp add: fps_square_nth)
      also have "(∑k≤n-1. F $ k * F $ (n - Suc k)) = (∑k≤n-1. G $
k * G $ (n - Suc k))"
        by (intro sum.cong arg_cong2[of _ _ _ _ "(*)"] less.IH) (use n
in auto)
      also have "fps_nth 1 (n - 1) + ... = fps_nth (1 + G ^ 2) (n - 1)"
        using n by (simp add: fps_square_nth)
      also have "(1 + G ^ 2) = 2 * fps_deriv G"
        using <2 * fps_deriv G = 1 + G ^ 2> ..
      also have "fps_nth ... (n - 1) = 2 * of_nat n * fps_nth G n"
        using n by simp
      finally show ?thesis
        using n by simp
    qed
  qed
  thus "F = G"
    by (rule fps_ext)
qed
```

Lastly, we get the following explicit relationships between the zigzag numbers and the coefficients appearing in the Maclaurin series of sec and tan.

```
corollary zigzag_number_conv_fps_sec:
  assumes "even n"
  shows   "real (zigzag_number n) = fps_nth (fps_sec 1) n * fact n"
proof -
  have "real (zigzag_number n) / fact n =
          fps_nth (Abs_fps (λn. real (zigzag_number n) / fact n)) n"
    by simp
  also have "Abs_fps (λn. real (zigzag_number n) / fact n) = fps_sec 1
```

```
+ fps_tan 1"
    by (rule exponential_generating_function_zigzag_number)
  also have "fps_nth ... n = fps_nth (fps_sec 1) n"
    using assms by (simp add: fps_nth_tan_even)
  finally show ?thesis
    by (simp add: field_simps)
qed

corollary zigzag_number_conv_fps_tan:
  assumes "odd n"
  shows   "real (zigzag_number n) = fps_nth (fps_tan 1) n * fact n"
proof -
  have "real (zigzag_number n) / fact n =
          fps_nth (Abs_fps (λn. real (zigzag_number n) / fact n)) n"
    by simp
  also have "Abs_fps (λn. real (zigzag_number n) / fact n) = fps_sec 1
+ fps_tan 1"
    by (rule exponential_generating_function_zigzag_number)
  also have "fps_nth ... n = fps_nth (fps_tan 1) n"
    using assms by (simp add: fps_nth_sec_odd)
  finally show ?thesis
    by (simp add: field_simps)
qed
```

2.4 Alternating permutations with a fixed first element

In order to study the *Entringer numbers*, a generalisation of the zigzag numbers, we introduce the set of alternating permutations on a set that start with some fixed element x.

```
definition alternating_permutations_of_set_with_hd ::
  "('a × 'a) set ⇒ 'a set ⇒ 'a ⇒ 'a list set" where
  "alternating_permutations_of_set_with_hd R A x =
     {xs∈alternating_permutations_of_set R A. xs ≠ [] ∧ hd xs = x}"

lemma alternating_permutations_of_set_with_hd_singleton:
  "alternating_permutations_of_set_with_hd R {y} x = (if x = y then {[x]}
else {})"
  by (auto simp: alternating_permutations_of_set_with_hd_def alternating_permutations_of_se

lemma alternating_permutations_of_set_with_hd_outside:
  assumes "x ∉ A"
  shows   "alternating_permutations_of_set_with_hd R A x = {}"
proof -
  {
    fix xs assume "xs ∈ alternating_permutations_of_set_with_hd R A x"
    hence "set xs = A" "xs ≠ []" "hd xs = x"
      by (auto simp: alternating_permutations_of_set_with_hd_def
                   alternating_permutations_of_set_def permutations_of_set_def)
    moreover from this have "hd xs ∈ set xs"
```

```
      by (intro hd_in_set)
    ultimately have "x ∈ A"
      by auto
    hence False
      using assms by simp
  }
  thus ?thesis
    by blast
qed

lemma alternating_permutations_of_set_with_hd_least:
  assumes "strict_linear_order_on A R"
  assumes "⋀y. y ∈ A - {x} ⟹ (x, y) ∈ R" "x ∈ A" "A ≠ {x}" "finite
A"
  shows   "alternating_permutations_of_set_with_hd R A x = {}"
proof -
  from assms have "A - {x} ≠ {}"
    by auto
  hence "card (A - {x}) > 0"
    using ‹finite A› card_gt_0_iff by blast
  hence "card A ≥ 2"
    by (subst (asm) card_Diff_subset) (use assms in auto)

  {
    fix xs assume "xs ∈ alternating_permutations_of_set_with_hd R A x"
    hence xs: "set xs = A" "xs ≠ []" "hd xs = x" "alternating_list R
xs" "distinct xs"
      by (auto simp: alternating_permutations_of_set_with_hd_def
                     alternating_permutations_of_set_def permutations_of_set_def)
    have "length xs ≥ 2"
      using distinct_card[of xs] xs ‹card A ≥ 2› by simp
    then obtain x' y xs' where xs_eq: "xs = x' # y # xs'"
      by (auto simp: Suc_le_length_iff numeral_2_eq_2)
    have [simp]: "x' = x"
      using ‹hd xs = x› by (simp add: xs_eq)
    from xs(4) have "(y, x) ∈ R"
      by (simp add: xs_eq)
    moreover from this and assms(1) have "y ∈ A - {x}"
      using ‹set xs = A›  by (auto simp: strict_linear_order_on_def irrefl_def
xs_eq)
    with assms(2)[of y] and ‹set xs = A› have "(x, y) ∈ R"
      by (auto simp: xs_eq)
    ultimately have False
      using strict_linear_order_on_asym_on[OF assms(1)] ‹x ∈ A› ‹y ∈
A - {x}›
      by (auto simp: asym_on_def)
  }
  thus ?thesis
    by blast
```

```
qed

lemma alternating_permutations_of_set_with_hd_greatest:
  assumes "strict_linear_order_on A R"
  assumes "⋀y. y ∈ A - {x} ⟹ (y, x) ∈ R" "x ∈ A"
  shows   "bij_betw (λxs. x # xs)
             (rev_alternating_permutations_of_set R (A - {x}))
             (alternating_permutations_of_set_with_hd R A x)"
proof -
  have [simp]: "A ≠ {}"
    using ‹x ∈ A› by auto
  show ?thesis
  proof (rule bij_betwI)
    show "(#) x ∈ rev_alternating_permutations_of_set R (A - {x}) →
                    alternating_permutations_of_set_with_hd R A x"
    proof (safe, goal_cases)
      case (1 xs)
      hence "set xs ⊆ A - {x}"
        by (auto simp: alternating_permutations_of_set_def permutations_of_set_def)
      moreover have "hd xs ∈ set xs ∨ xs = []"
        using hd_in_set by blast
      ultimately have "hd xs ∈ A - {x} ∨ xs = []"
        by blast
      hence "(hd xs, x) ∈ R ∨ xs = []"
        using assms(2) by blast
      thus ?case
        using ‹x ∈ A› assms(2) 1
        by (auto simp: alternating_permutations_of_set_with_hd_def alternating_permutations
                       permutations_of_set_nonempty alternating_list_Cons_iff)
    qed
  next
    show "tl ∈ alternating_permutations_of_set_with_hd R A x →
                 rev_alternating_permutations_of_set R (A - {x})"
      by (auto simp: alternating_permutations_of_set_with_hd_def
                     alternating_permutations_of_set_def permutations_of_set_nonempty
                     alternating_list_Cons_iff)
  qed (auto simp: alternating_permutations_of_set_with_hd_def)
qed

lemma UN_alternating_permutations_of_set_with_hd:
  assumes "A ≠ {}"
  shows   "(⋃x∈A. alternating_permutations_of_set_with_hd R A x) =
             alternating_permutations_of_set R A"
  using assms
  by (force simp: alternating_permutations_of_set_with_hd_def
                  alternating_permutations_of_set_def permutations_of_set_def
intro!: hd_in_set)

lemma alternating_permutations_of_set_with_hd_split_first:
```

```
  assumes "strict_linear_order_on A R" "x ∈ A" "A ≠ {x}"
  shows   "bij_betw ((#) x)
            (⋃y∈{y∈A-{x}. (y,x)∈R}. alternating_permutations_of_set_with_hd
(converse R) (A - {x}) y)
            (alternating_permutations_of_set_with_hd R A x)"
proof -
  have [simp]: "A ≠ {}"
    using assms by auto
  have "A - {x} ≠ {}"
    using assms by blast

  show ?thesis
  proof (rule bij_betwI)
    show "(#) x ∈ ⋃ (alternating_permutations_of_set_with_hd (R⁻¹) (A
- {x}) ` {y ∈ A - {x}. (y, x) ∈ R}) →
                  alternating_permutations_of_set_with_hd R A x"
    proof (intro Pi_I; elim UN_E, goal_cases)
      case (1 xs y)
      have xs: "xs ∈ permutations_of_set (A - {x})" "alternating_list
(converse R) xs" "hd xs = y"
        using 1 by (auto simp: alternating_permutations_of_set_with_hd_def

                              alternating_permutations_of_set_def)
      have "x # xs ∈ permutations_of_set A"
        using xs <x ∈ A> by (auto simp: permutations_of_set_nonempty)
      moreover have "alternating_list R (x # xs)"
        using xs 1 by (auto simp: alternating_list_Cons_iff)
      ultimately show "x # xs ∈ alternating_permutations_of_set_with_hd
R A x"
        unfolding alternating_permutations_of_set_with_hd_def
        by (auto simp: alternating_permutations_of_set_def)
    qed
  next
    show "tl ∈ alternating_permutations_of_set_with_hd R A x →
                ⋃ (alternating_permutations_of_set_with_hd (R⁻¹) (A
- {x}) ` {y ∈ A - {x}. (y, x) ∈ R})"
    proof (safe, goal_cases)
      case (1 xs)
      have xs: "xs ∈ permutations_of_set A" "alternating_list R xs" "hd
xs = x"
        using 1 by (auto simp: alternating_permutations_of_set_with_hd_def

                              alternating_permutations_of_set_def)
      have "xs ≠ []"
        using xs assms by (auto simp: permutations_of_set_def)
      then obtain ys where xs_eq: "xs = x # ys"
        using xs(3) by (cases xs) auto

      have ys: "ys ∈ permutations_of_set (A - {x})"
```

```
        using xs by (auto simp: permutations_of_set_nonempty xs_eq)
      hence "set ys = A - {x}"
        by (auto simp: permutations_of_set_def)
      hence "ys ≠ []"
        using ‹A - {x} ≠ {}› by (intro notI) auto

      have "hd ys ∈ A"
        using hd_in_set[of ys] ‹set ys = A - {x}› ‹ys ≠ []› by auto
      moreover have "rev_alternating_list R ys" "(hd ys, x) ∈ R"
        using xs ‹ys ≠ []› by (auto simp: xs_eq alternating_list_Cons_iff)
      moreover have "(hd ys, hd ys) ∉ R"
        using assms(1) by (auto simp: strict_linear_order_on_def irrefl_def)
      ultimately show ?case
        using ‹ys ≠ []› ys
        by (auto simp: xs_eq alternating_permutations_of_set_with_hd_def
                       alternating_permutations_of_set_def)
    qed
  qed (auto simp: alternating_permutations_of_set_with_hd_def)
qed

lemma bij_betw_alternating_permutations_of_set_with_hd_flip:
  assumes "x ≤ n"
  shows   "bij_betw (map (λk. n - k))
             (alternating_permutations_of_set_with_hd {(x::nat,y). x <
y} {0..n} x)
             (alternating_permutations_of_set_with_hd {(x::nat,y). x >
y} {0..n} (n - x))"
proof -
  have *: "bij_betw (λk. n - k) {0..n} {0..n}"
    by (rule bij_betwI[of _ _ _ "λk. n - k"]) auto
  have "bij_betw (map ((-) n))
          (alternating_permutations_of_set {(x, y). x < y} {0..n})
          (alternating_permutations_of_set {(x, y). y < x} {0..n})"
    by (rule bij_betw_alternating_permutations_of_set)
       (use * in ‹auto simp: monotone_on_def image_def bij_betw_def›)

  thus ?thesis
    unfolding alternating_permutations_of_set_with_hd_def
  proof (rule bij_betw_Collect, goal_cases)
    case (1 xs)
    hence "xs ≠ []" "set xs = {0..n}"
      by (auto simp: alternating_permutations_of_set_def permutations_of_set_def)
    with hd_in_set[of xs] have "hd xs ≤ n"
      by auto
    thus ?case using ‹xs ≠ []› assms
      by (auto simp: hd_map)
  qed
qed
```

2.5 Entringer numbers

The Entringer number $E_{n,k}$ now counts the number of alternating permutations on a set with $n+1$ elements that start with the (unique) element of rank k, i.e. the k-th largest element of the set. [3, A008282]

As we will see, it suffices to w.l.o.g. only consider sets of integers of the form $\{0,\ldots,n\}$.

```
definition entringer_number :: "nat ⇒ nat ⇒ nat" where
  "entringer_number n k =
     card (alternating_permutations_of_set_with_hd {(x,y). x < y} {0..n}
k)"

lemma entringer_number_0_0 [simp]: "entringer_number 0 0 = 1"
  and entring_number_0_left [simp]: "k ≠ 0 ⟹ entringer_number 0 k =
0"
  by (simp_all add: entringer_number_def alternating_permutations_of_set_with_hd_singleton)

lemma entringer_number_0_right [simp]:
  assumes "n > 0"
  shows   "entringer_number n 0 = 0"
proof -
  have "alternating_permutations_of_set_with_hd {(x,y). x < y} {0..n}
0 = {}"
    by (rule alternating_permutations_of_set_with_hd_least) (use assms
in auto)
  thus ?thesis
    using assms by (simp add: entringer_number_def)
qed

lemma entringer_number_greater_eq_0 [simp]:
  assumes "k > n"
  shows   "entringer_number n k = 0"
proof -
  have "alternating_permutations_of_set_with_hd {(x,y). x < y} {0..n}
k = {}"
    by (rule alternating_permutations_of_set_with_hd_outside) (use assms
in auto)
  thus ?thesis
    using assms by (simp add: entringer_number_def)
qed

theorem card_alternating_permutations_of_set_with_hd:
  assumes "strict_linear_order_on A R" "finite A" "x ∈ A"
  shows   "card (alternating_permutations_of_set_with_hd R A x) =
             entringer_number (card A - 1) (card {y∈A-{x}. (y,x) ∈ R})"
proof -
  define n where "n = card A - 1"
  have "A ≠ {}"
```

```
    using ‹x ∈ A› by auto
  with ‹finite A› have "card A > 0"
    using card_gt_0_iff by blast
  hence "card A = Suc n"
    by (auto simp: n_def)
  hence *: "{0..n} = {0..<card A}"
    by auto

  obtain f :: "nat ⇒ 'a" where f:
    "bij_betw f {0..n} A" "monotone_on {0..n} (<) (λx y. (x,y) ∈ R) f"
    using strict_linear_orderE_bij_betw[OF assms(1,2)] unfolding * .
  obtain k where k: "k ≤ n" "f k = x"
    using f(1) ‹x ∈ A› by (auto simp: bij_betw_def)
  have R_f_iff: "(f x, f y) ∈ R ⟷ x < y" if "x ≤ n" "y ≤ n" for x
y
    by (rule monotone_on_strict_linear_orderD[OF f(2)])
       (use assms that f(1) in ‹auto simp: bij_betw_def›)
  have f_eq_iff: "f x = f y ⟷ x = y" if "x ≤ n" "y ≤ n" for x y
    using f(1) that by (auto simp: bij_betw_def inj_on_def)

  have "bij_betw f {i∈{0..n}. i < k} {y∈A. (y, x) ∈ R}"
    using f(1) by (rule bij_betw_Collect) (use f(2) k in ‹auto simp: monotone_on_def
R_f_iff›)
  hence "card {i∈{0..n}. i < k} = card {y∈A. (y, x) ∈ R}"
    by (rule bij_betw_same_card)
  also have "{i∈{0..n}. i < k} = {..<k}"
    using k by auto
  also have "{y∈A. (y, x) ∈ R} = {y∈A-{x}. (y, x) ∈ R}"
    using ‹x ∈ A› assms by (auto simp: strict_linear_order_on_def irrefl_def)
  finally have k_eq: "k = card {y∈A-{x}. (y, x) ∈ R}"
    by simp

  have "bij_betw (map f)
          (alternating_permutations_of_set_with_hd {(x,y). x < y} {0..n}
k)
          (alternating_permutations_of_set_with_hd R A x)"
    unfolding alternating_permutations_of_set_with_hd_def
    using bij_betw_alternating_permutations_of_set
  proof (rule bij_betw_Collect)
    show "A = f ` {0..n}" "strict_linear_order_on (f ` {0..n}) R"
      using f(1) assms by (simp_all add: bij_betw_def)
  next
    fix xs assume "xs ∈ alternating_permutations_of_set {(x, y). x <
y} {0..n}"
    hence xs: "set xs = {0..n}" "xs ≠ []"
      by (auto simp: alternating_permutations_of_set_def permutations_of_set_def)
    show "(map f xs ≠ [] ∧ hd (map f xs) = x) ⟷ (xs ≠ [] ∧ hd xs
= k)"
      using k hd_in_set[of xs] xs by (auto simp: hd_map f_eq_iff)
```

```
  qed (use f assms in <auto simp: hd_map>)
  hence "card (alternating_permutations_of_set_with_hd {(x,y). x < y}
{0..n} k) =
         card (alternating_permutations_of_set_with_hd R A x)"
    by (rule bij_betw_same_card)
  also have "card (alternating_permutations_of_set_with_hd {(x,y). x <
y} {0..n} k) =
             entringer_number n k"
    unfolding entringer_number_def by simp
  finally show ?thesis
    by (simp add: n_def k_eq)
qed
```

It is not difficult to show that $E_{n,n} = E_n$, i.e. the Entringer numbers really are a generalisation of the Euler numbers. The idea is that if we have an alternating permutation of n elements $0, 1, \ldots, n$ that starts with largest one (i.e. n) then the list we obtain after dropping the initial element is a reverse-alternating permutation of $0, 1, \ldots, n-1$ with no further restrictions, and this map is one-to-one.

```
lemma entringer_number_same [simp]:
  "entringer_number n n = zigzag_number n"
proof (cases "n = 0")
  case False
  have "bij_betw (λxs. n # xs)
               (rev_alternating_permutations_of_set {(x, y). x < y} ({0..n}-{n}))
               (alternating_permutations_of_set_with_hd {(x, y). x < y}
{0..n} n)"
    by (rule alternating_permutations_of_set_with_hd_greatest) auto
  hence "card (rev_alternating_permutations_of_set {(x, y). x < y} ({0..n}-{n}))
=
         card (alternating_permutations_of_set_with_hd {(x, y). x < y}
{0..n} n)"
    by (rule bij_betw_same_card)
  also have "... = entringer_number n n"
    using False by (simp add: entringer_number_def)
  also have "converse {(x, y). x < y} = {(x::nat, y). x > y}"
    by auto
  also have "card (alternating_permutations_of_set {(x, y). x > y} ({0..n}-{n}))
= zigzag_number n"
    by (subst card_alternating_permutations_of_set) auto
  finally show ?thesis ..
qed auto

lemma card_rev_alternating_permutations_of_set_with_hd:
  assumes x: "x ≤ n"
  shows "card (alternating_permutations_of_set_with_hd {(x::nat,y). x
> y} {0..n} x) =
           entringer_number n (n - x)"
proof -
```

```
  have "card (alternating_permutations_of_set_with_hd {(x::nat,y). x >
y} {0..n} x) =
        entringer_number n (card {y ∈ {0..n} - {x}. x < y})"
    by (subst card_alternating_permutations_of_set_with_hd) (use assms
in auto)
  also have "{y ∈ {0..n} - {x}. x < y} = {x<..n}"
    using x by auto
  finally show ?thesis
    by simp
qed
```

The following summation identity can be visualised as follows: if we have an alternating permutation of the elements $0, \ldots, n$ that starts with k then the next element after k must be a reverse-alternating permutation starting with one of the elements $0, \ldots, k-1$, and this is again a bijection.

```
theorem sum_entringer_numbers:
  assumes k: "k ≤ Suc n"
  shows   "(∑i<k. entringer_number n (n - i)) = entringer_number (Suc
n) k"
proof -
  define A where "A = (λX x. alternating_permutations_of_set_with_hd
{(x::nat,y). x < y} X x)"
  define A' where "A' = (λX x. alternating_permutations_of_set_with_hd
{(x::nat,y). x > y} X x)"
  have converses: "converse {(x::nat,y). x < y} = {(x::nat,y). x > y}"

                  "converse {(x::nat,y). x > y} = {(x::nat,y). x < y}"
    by auto

  have "bij_betw ((#) k)
        (⋃ (alternating_permutations_of_set_with_hd ({(x, y). x < y}⁻¹)
({0..Suc n} - {k}) ` {y ∈ {0..Suc n} - {k}. (y, k) ∈ {(x, y). x < y}}))
         (alternating_permutations_of_set_with_hd {(x, y). x < y} {0..Suc
n} k)"
    by (intro alternating_permutations_of_set_with_hd_split_first) (use
k in auto)
  also have "{y ∈ {0..Suc n} - {k}. (y, k) ∈ {(x, y). x < y}} = {0..<k}"
    using k by auto
  finally have "bij_betw ((#) k) (⋃i<k. A' ({0..Suc n} - {k}) i) (A {0..Suc
n} k)"
    using converses by (simp add: A_def A'_def case_prod_unfold atLeast0LessThan)
  hence "card (⋃i<k. A' ({0..Suc n} - {k}) i) = card (A {0..Suc n} k)"
    by (rule bij_betw_same_card)
  also have "card (A {0..Suc n} k) = entringer_number (Suc n) k"
    by (simp add: entringer_number_def A_def)
  also have "card (⋃i<k. A' ({0..Suc n} - {k}) i) = (∑i<k. card (A'
({0..Suc n} - {k}) i))"
    by (subst card_UN_disjoint)
       (auto simp: A'_def alternating_permutations_of_set_with_hd_def
```

```
alternating_permutations_of_set_def)
  also have "... = (∑i<k. entringer_number n (n - i))"
  proof (intro sum.cong)
    fix i assume i: "i ∈ {..<k}"
    have "card (A' ({0..Suc n} - {k}) i) =
          entringer_number n (card {j ∈ {0..Suc n} - {k} - {i}. i < j})"
      unfolding A'_def using i k
      by (subst card_alternating_permutations_of_set_with_hd) auto
    also have "{j ∈ {0..Suc n} - {k} - {i}. i < j} = {i<..Suc n} - {k}"
      using i k by auto
    also have "card ... = n - i"
      using i k by (subst card_Diff_subset) auto
    finally show "card (A' ({0..Suc n} - {k}) i) = entringer_number n
(n - i)" .
  qed auto
  finally show ?thesis .
qed

lemma sum_entringer_numbers':
  assumes k: "k ≤ n"
  shows   "(∑i≤k. entringer_number n (n - i)) = entringer_number (Suc
n) (Suc k)"
proof -
  have "(∑i<Suc k. entringer_number n (n - i)) = entringer_number (Suc
n) (Suc k)"
    by (rule sum_entringer_numbers) (use k in auto)
  also have "{..<Suc k} = {..k}"
    by auto
  finally show ?thesis .
qed
```

A consequence of this summation identity is that the sum of all the values in the n-th row of the Entringer triangle is exactly the n-th zigzag number.

```
corollary sum_entringer_numbers_row: "(∑k≤n. entringer_number n k) =
zigzag_number (Suc n)"
proof -
  have "(∑k≤n. entringer_number n (n - k)) = zigzag_number (Suc n)"
    using sum_entringer_numbers'[OF order.refl, of n] by simp
  also have "(∑k≤n. entringer_number n (n - k)) = (∑k≤n. entringer_number
n k)"
    by (rule sum.reindex_bij_witness[of _ "λk. n - k" "λk. n - k"]) auto
  finally show ?thesis
    by simp
qed
```

By telescoping the summation identity, we also obtain the following simple recurrence for the Entringer numbers:

```
corollary entringer_number_rec:
  assumes "k ≤ n"
```

```
  shows   "entringer_number (Suc n) (Suc k) =
             entringer_number (Suc n) k + entringer_number n (n - k)"
proof -
  have "entringer_number (Suc n) (Suc k) = (∑i≤k. entringer_number n
(n - i))"
    by (rule sum_entringer_numbers' [symmetric]) (use assms in auto)
  also have "{..k} = insert k {..<k}"
    by auto
  also have "(∑i∈.... entringer_number n (n - i)) =
             (∑i<k. entringer_number n (n - i)) + entringer_number n
(n - k)"
    by (subst sum.insert) auto
  also have "(∑i<k. entringer_number n (n - i)) = entringer_number (Suc
n) k"
    by (rule sum_entringer_numbers) (use assms in auto)
  finally show ?thesis .
qed
```

This recurrence can be used to compute the Entringer numbers (although if one wants this to be efficient one has to be a bit smarter about avoiding double computations; either by memoisation or by finding a smarter way to traverse the triangle).

```
lemma entringer_number_code [code]:
  "entringer_number n k =
     (if n = 0 then if k = 0 then 1 else 0
      else if k = 0 ∨ k > n then 0
      else entringer_number n (k - 1) + entringer_number (n - 1) (n -
k))"
  using entringer_number_rec[of "k - 1" "n - 1"] by (cases n; cases k)
auto

end
```

3 Increasing binary trees

```
theory Increasing_Binary_Trees
  imports Alternating_Permutations "HOL-Library.Tree"
begin
```

We will now look at a second combinatorial application of the zigzag numbers E_n.

An increasing binary trees is one where

- the root contains the smallest element
- no element is contained in the tree twice
- if a node has exactly one non-leaf child, it must be the left child

- if a node has two non-leaf children, the element attached to the left one must be smaller than that of the right one

Another way to think of this is as a heap with no duplicate elements where each node has either 0, 1, or 2 children and the order of the children does not matter. This is however slightly more awkward to express.

We will show below that the number of increasing binary trees with n nodes with values from a set with n elements is E_n.

We do this by showing that the number of increasing binary trees satisfies the same recurrence as E_n.

The following relation represents the condition that a non-leaf child must always be to the left of a leaf child, and a right node child must have a value greater than a left node child.

```
definition le_root :: "'a :: ord tree ⇒ 'a tree ⇒ bool" where
  "le_root t1 t2 =
     (case t1 of
        Leaf ⇒ t2 = Leaf
      | Node _ x _ ⇒ (case t2 of Leaf ⇒ True | Node _ y _ ⇒ x ≤ y))"
```

The following predicate models the notion that a binary tree is increasing.

```
primrec inc_tree :: "'a :: linorder tree ⇒ bool" where
  "inc_tree Leaf = True"
| "inc_tree (Node l x r) ⟷ inc_tree l ∧ inc_tree r ∧ le_root l r ∧
     (∀y∈set_tree l ∪ set_tree r. x < y) ∧ set_tree l ∩ set_tree r =
{}"
```

We introduce the following abbreviation for the set of increasing binary trees that have exactly the values from the given set attached to them.

```
definition Inc_Trees :: "'a :: linorder set ⇒ 'a tree set" where
  "Inc_Trees A = {t. set_tree t = A ∧ inc_tree t}"

lemma Inc_Trees_empty [simp]: "Inc_Trees {} = {Leaf}"
  by (auto simp: Inc_Trees_def)

lemma Inc_Trees_infinite_eq_empty [simp]:
  assumes "¬finite A"
  shows   "Inc_Trees A = {}"
  using assms finite_set_tree unfolding Inc_Trees_def by blast
```

For our proof later, we will need to also consider the set of "almost" increasing binary trees, i.e. binary trees that are increasing if the left and right child of the root are swapped.

```
primrec mirror_root :: "'a tree ⇒ 'a tree" where
  "mirror_root Leaf = Leaf"
| "mirror_root (Node l x r) = Node r x l"
```

```
lemma mirror_root_mirror_root [simp]: "mirror_root (mirror_root t) =
t"
  by (cases t) auto

lemma set_tree_mirror_root [simp]: "set_tree (mirror_root t) = set_tree
t"
  by (cases t) auto

definition Inc_Trees' :: "'a :: linorder set ⇒ 'a tree set" where
  "Inc_Trees' A = {t. set_tree t = A ∧ inc_tree (mirror_root t)}"

lemma Inc_Trees'_empty [simp]: "Inc_Trees' {} = {Leaf}"
  by (auto simp: Inc_Trees'_def)

lemma Inc_Trees'_infinite_eq_empty [simp]:
  assumes "¬finite A"
  shows   "Inc_Trees' A = {}"
  using assms finite_set_tree unfolding Inc_Trees'_def by blast
```

Since swapping the children of the root is an involution, the number of increasing binary trees and the number of almost increasing binary trees is the same.

```
lemma bij_betw_mirror_root_Inc_Trees: "bij_betw mirror_root (Inc_Trees
A) (Inc_Trees' A)"
  by (rule bij_betwI[of mirror_root _ _ mirror_root]) (auto simp: Inc_Trees_def
Inc_Trees'_def)

lemma card_Inc_Trees' [simp]: "card (Inc_Trees' A) = card (Inc_Trees
A)"
  using bij_betw_same_card[OF bij_betw_mirror_root_Inc_Trees[of A]] by
simp
```

Except for the obvious case $|A| \leq 1$, a tree cannot be both increasing and almost increasing.

```
lemma disjoint_Inc_Trees_Inc_Trees':
  assumes "card A > 1"
  shows   "Inc_Trees A ∩ Inc_Trees' A = {}"
proof safe
  fix t assume t: "t ∈ Inc_Trees A" "t ∈ Inc_Trees' A"
  obtain l x r where t_eq: "t = Node l x r"
    using t assms by (cases t) (auto simp: Inc_Trees_def)
  have "le_root l r ∧ le_root r l" "set_tree l ∩ set_tree r = {}"
    using t by (auto simp: t_eq Inc_Trees_def Inc_Trees'_def)
  hence "l = Leaf ∧ r = Leaf"
    by (cases l; cases r; force simp: le_root_def)
  moreover have "A = {x} ∪ set_tree l ∪ set_tree r"
    using t by (simp add: Inc_Trees_def t_eq)
  ultimately have "A = {x}"
```

```
    by simp
  thus "t ∈ {}"
    using assms by simp
qed
```

If we take any subset X of a set A, pick increasing binary trees l on X and r on $A \setminus X$ and then make them the left and right child, respectively, of a new node with a value x that is smaller than all values in A, then we obtain exactly all increasing and almost increasing binary trees on $A \cup \{x\}$.

```
lemma Inc_Trees_insert_min:
  assumes "⋀y. y ∈ A ⟹ x < y"
  shows   "Inc_Trees (insert x A) ∪ Inc_Trees' (insert x A) =
             (⋃X∈Pow A. ⋃l∈Inc_Trees X. ⋃r∈Inc_Trees (A-X). {Node
l x r})"
proof ((intro equalityI subsetI; (elim UN_E)?), goal_cases)
  case (1 t)
  then obtain l x' r where t_eq: "t = Node l x' r"
    using assms by (cases t) (auto simp: Inc_Trees_def Inc_Trees'_def)
  define X where "X = set_tree l"
  have "x ∉ A"
    using assms by force
  have "x' ∉ set_tree l ∪ set_tree r"
    using 1 unfolding Inc_Trees_def Inc_Trees'_def t_eq by auto
  have "set_tree t = insert x' (set_tree l ∪ set_tree r)"
    by (simp add: Inc_Trees_def t_eq)
  also have "set_tree t = insert x A"
    using 1 by (auto simp: Inc_Trees_def Inc_Trees'_def)
  finally have [simp]: "x' = x" using assms
    using assms 1 <x ∉ A> <x' ∉ set_tree l ∪ set_tree r>
    by (fastforce simp: Inc_Trees_def Inc_Trees'_def t_eq insert_eq_iff
Un_commute)
  have "X ∩ set_tree r = {}"
    using 1 unfolding X_def by (auto simp: Inc_Trees_def Inc_Trees'_def
t_eq)
  have "set_tree t = insert x (X ∪ set_tree r)"
    by (simp add: t_eq X_def)
  also have "set_tree t = insert x A"
    using 1 by (auto simp: Inc_Trees_def Inc_Trees'_def t_eq)
  finally have "set_tree r = A - X"
    using <X ∩ set_tree r = {}> <x' ∉ _> <x ∉ A>
    by (auto simp: insert_eq_iff)

  have "X ∈ Pow A"
    using <set_tree t = insert x A> <x' ∉ _> unfolding X_def t_eq by
auto
  moreover have "l ∈ Inc_Trees X"
    using 1 by (auto simp add: X_def Inc_Trees_def Inc_Trees'_def t_eq)
  moreover have "r ∈ Inc_Trees (A - X)"
    using 1 <set_tree r = A - X> by (auto simp add: Inc_Trees_def Inc_Trees'_def
```

```
t_eq)
  ultimately show "t ∈ (⋃X∈Pow A. ⋃l∈Inc_Trees X. ⋃r∈Inc_Trees (A
- X). {⟨l, x, r⟩})"
    unfolding t_eq <x' = x> by blast
next
  case (2 t X l r)
  have "le_root l r ∨ le_root r l"
    by (cases l; cases r) (force simp: le_root_def)+
  thus ?case
    using 2 assms
    by (auto simp: Inc_Trees_def Inc_Trees'_def)
qed

lemma Inc_Trees_singleton [simp]: "Inc_Trees {x} = {Node Leaf x Leaf}"
  and Inc_Trees'_singleton [simp]: "Inc_Trees' {x} = {Node Leaf x Leaf}"
proof -
  have "Inc_Trees {x} ∪ Inc_Trees' {x} = {Node Leaf x Leaf}"
    by (subst Inc_Trees_insert_min) auto
  moreover have "Inc_Trees {x} ≠ {}"
    by (auto simp: Inc_Trees_def le_root_def intro!: exI[of _ "Node Leaf
x Leaf"])
  moreover have "Inc_Trees' {x} ≠ {}"
    by (auto simp: Inc_Trees'_def le_root_def intro!: exI[of _ "Node Leaf
x Leaf"])
  ultimately show "Inc_Trees {x} = {Node Leaf x Leaf}" "Inc_Trees' {x}
= {Node Leaf x Leaf}"
    by (simp_all add: Un_singleton_iff)
qed

lemma Diff_right_commute: "A - B - C = A - C - (B :: 'a set)"
  by blast
```

We can therefore derive the following recurrence on the set of increasing and almost increasing binary trees on a set A: pick the smallest element x in A as a minimum, then pick a subset X of $A \setminus \{x\}$ and any increasing trees on X as the left child and any increasing tree on $X \setminus (A \cup \{x\})$ as the right child.

```
lemma Inc_Trees_rec:
  assumes "finite A" "A ≠ {}"
  defines "x ≡ Min A"
  shows   "Inc_Trees A ∪ Inc_Trees' A =
             (⋃X∈Pow (A-{x}). ⋃l∈Inc_Trees X. ⋃r∈Inc_Trees (A-X-{x}).
{Node l x r})"
proof -
  define A' where "A' = A - {x}"
  have 1: "x ≤ y" if "y ∈ A" for y
    unfolding x_def by (rule Min.coboundedI) (use assms that in auto)
  have 2: "x < y" if "y ∈ A'" for y
    using 1[of y] that by (auto simp: A'_def)
```

```
  have "x ∈ A"
    unfolding x_def by (rule Min_in) (use assms in auto)
  hence "A = insert x A'"
    by (auto simp: A'_def)
  also have "Inc_Trees (insert x A') ∪ Inc_Trees' (insert x A') =
              (⋃X∈Pow A'. ⋃l∈Inc_Trees X. ⋃r∈Inc_Trees (A' - X).
{⟨l, x, r⟩})"
    by (subst Inc_Trees_insert_min) (use 2 in auto)
  finally show ?thesis
    by (simp add: A'_def Diff_right_commute)
qed

lemma Inc_Trees_rec':
  assumes "finite A" "A ≠ {}"
  defines "x ≡ Min A"
  shows   "Inc_Trees A ∪ Inc_Trees' A =
             (λ(_, (l, r)). Node l x r) ` (SIGMA X:Pow (A-{x}). Inc_Trees
X × Inc_Trees (A - X - {x}))"
  unfolding Inc_Trees_rec[OF assms(1,2)] x_def
  unfolding Sigma_def image_UN image_insert image_empty image_Union image_image
prod.case
  by blast

lemma finite_Inc_Trees [intro]: "finite (Inc_Trees A)"
  and finite_Inc_Trees' [intro]: "finite (Inc_Trees' A)"
proof -
  have "finite (Inc_Trees A ∪ Inc_Trees' A)"
  proof (cases "finite A")
    case True
    thus ?thesis
    proof (induction rule: finite_psubset_induct)
      case (psubset A)
      have IH: "finite (Inc_Trees B)" if "B ⊂ A" for B
        using psubset.IH[of B] that by blast
      show ?case
      proof (cases "A = {}")
        case False
        hence "Min A ∈ A"
          using psubset.hyps by (intro Min_in) auto
        have "Inc_Trees A ∪ Inc_Trees' A = (λ(_, l, y). ⟨l, Min A, y⟩)
`
                (SIGMA X:Pow (A - {Min A}). Inc_Trees X × Inc_Trees
(A - X - {Min A}))"
          by (intro Inc_Trees_rec') (use False psubset.hyps in auto)
        also have "finite ..."
          using <Min A ∈ A> psubset.hyps
          by (intro finite_imageI finite_SigmaI IH) auto
        finally show ?thesis .
      qed auto
```

```
    qed
  qed simp_all
  thus "finite (Inc_Trees A)" and "finite (Inc_Trees' A)"
    by auto
qed
```

By taking the cardinality of both sides, we obtain the following recurrence on twice the number of increasing trees. Note that this only holds for $|A| > 1$ since otherwise the set of increasing and almost increasing trees are not disjoint.

```
lemma card_Inc_Trees_rec:
  assumes "finite A" "card A > 1"
  defines "x ≡ Min A"
  shows   "2 * card (Inc_Trees A) =
             (∑X∈Pow (A - {x}). card (Inc_Trees X) * card (Inc_Trees
(A - X - {x})))"
proof -
  have "A ≠ {}"
    using assms by auto
  have "Inc_Trees A ∪ Inc_Trees' A =
          (λ(_, (l, r)). Node l x r) ` (SIGMA X:Pow (A-{x}). Inc_Trees
X × Inc_Trees (A - X - {x}))"
    unfolding x_def by (rule Inc_Trees_rec') fact+
  also have "card ... = card (SIGMA X:Pow (A - {x}). Inc_Trees X × Inc_Trees
(A - X - {x}))"
  proof (rule card_image)
    show "inj_on (λ(_, l, r). ⟨l, x, r⟩)
            (SIGMA X:Pow (A - {x}). Inc_Trees X × Inc_Trees (A - X -
{x}))"
      by (rule inj_onI) (auto simp: Inc_Trees_def)
  qed
  also have "... = (∑X∈Pow (A - {x}). card (Inc_Trees X) * card (Inc_Trees
(A - X - {x})))"
    using assms by (subst card_SigmaI) (auto simp: card_cartesian_product)
  also have "card (Inc_Trees A ∪ Inc_Trees' A) = card (Inc_Trees A) +
card (Inc_Trees' A)"
  proof (rule card_Un_disjoint)
    have False if t: "t ∈ Inc_Trees A ∩ Inc_Trees' A" for t
    proof -
      from t obtain l x r where t_eq: "t = Node l x r"
        using ‹A ≠ {}› by (cases t) (auto simp: Inc_Trees_def)
      have "le_root l r ∧ le_root r l"
        using t by (auto simp: Inc_Trees_def Inc_Trees'_def t_eq)
      hence "A = {x}"
        by (use t in ‹force simp: Inc_Trees_def Inc_Trees'_def le_root_def
t_eq split: tree.splits›)
      with assms show False
        by simp
    qed
```

```
    thus "Inc_Trees A ∩ Inc_Trees' A = {}"
      by blast
  qed auto
  also have "card (Inc_Trees' A) = card (Inc_Trees A)"
    by simp
  also have "... + ... = 2 * ..."
    by simp
  finally show ?thesis .
qed
```

By induction, our main result follows:

```
theorem card_Inc_Trees:
  assumes "finite A"
  shows   "card (Inc_Trees A) = zigzag_number (card A)"
  using assms
proof (induction rule: finite_psubset_induct)
  case (psubset A)
  show ?case
  proof (cases "card A < 2")
    case False
    have "card A > 1"
      using False by (simp add: card_gt_0_iff)
    have "A ≠ {}"
      using False by auto
    define x where "x = Min A"
    have "x ∈ A"
      unfolding x_def by (intro Min_in) fact+
    have "2 * card (Inc_Trees A) =
            (∑X∈Pow (A - {x}). card (Inc_Trees X) * card (Inc_Trees
(A - X - {x})))"
      unfolding x_def by (rule card_Inc_Trees_rec) fact+
    also have "... = (∑X∈Pow (A - {x}). zigzag_number (card X) * zigzag_number
(card A - card X - 1))"
    proof (intro sum.cong, goal_cases)
      case (2 X)
      have "finite X"
        by (rule finite_subset[of _ A]) (use 2 <finite A> in auto)
      have "card (Inc_Trees X) * card (Inc_Trees (A - X - {x})) =
            zigzag_number (card X) * zigzag_number (card (A - X - {x}))"
        by (intro arg_cong2[of _ _ _ _ "(*)"] psubset.IH)
           (use 2 <x ∈ A> in auto)
      also have "card (A - X - {x}) = card (A - X) - 1"
        by (subst card_Diff_subset) (use 2 <x ∈ A> in auto)
      also have "card (A - X) = card A - card X"
        by (subst card_Diff_subset) (use 2 psubset.hyps <finite X> in
auto)
      finally show ?case .
    qed auto
    also have "... = (∑X∈(⋃k≤card (A - {x}). {X. X ⊆ A - {x} ∧ card
```

```
X = k}).
                    zigzag_number (card X) * zigzag_number (card A -
card X - 1))"
      by (subst Pow_conv_subsets_of_size) (use psubset.hyps in simp_all)
    also have "... = (∑k≤card (A - {x}). card {X. X ⊆ A-{x} ∧ card
X = k} *
                    (zigzag_number k * zigzag_number (card A - k - 1)))"
      by (subst sum.UNION_disjoint) (use finite_subset[OF _ ‹finite A›]
in auto)
    also have "... = (∑k≤card (A - {x}). (card (A-{x}) choose k) *
                    (zigzag_number k * zigzag_number (card A - k - 1)))"
      by (intro sum.cong refl, subst n_subsets) (use ‹finite A› in auto)
    also have "card (A - {x}) = card A - 1"
      by (subst card_Diff_subset) (use ‹x ∈ A› ‹finite A› in auto)
    also have "(∑k≤card A - 1. (card A - 1 choose k) * (zigzag_number
k * zigzag_number (card A - k - 1))) =
              2 * zigzag_number (card A)"
      using zigzag_number_Suc[of "card A - 1"] ‹card A > 1› by simp
    finally show ?thesis
      by simp
  next
    case True
    hence "card A = 0 ∨ card A = 1"
      by auto
    then consider "A = {}" | x where "A = {x}"
      using card_1_singletonE[of A] ‹finite A› by auto
    thus ?thesis
      by cases simp_all
  qed
qed

end
```

4 Tangent numbers

```
theory Tangent_Numbers
imports
  "HOL-Computational_Algebra.Computational_Algebra"
  "Bernoulli.Bernoulli_FPS"
  "Polynomial_Interpolation.Ring_Hom_Poly"
  Boustrophedon_Transform_Library
  Alternating_Permutations
begin
```

4.1 The higher derivatives of $\tan x$

The n-th derivatives of $\tan x$ are:

- $\tan x^2 + 1$

- $\tan x^3 + \tan x$
- $6 \tan x^4 + 8 \tan x^2 + 2$
- $24 \tan x^5 + 40 \tan x^3 + 16 \tan x$
- ...

No pattern is readily apparent, but it is obvious that for any n, the n-th derivative of $\tan x$ can be expressed as a polynomial of degree $n+1$ in $\tan x$, i.e. it is of the form $P_n(\tan x)$ for some family of polynomials P_n.
Using the fact that $\tan' x = \tan x^2 + 1$ and the chain rule, one can deduce that $P_{n+1}(X) = (X^2 + 1)P_n'(X)$, and of course $P_0(X) = X$, which gives us a recursive characterisation of P_n.

```
primrec tangent_poly :: "nat ⇒ nat poly" where
  "tangent_poly 0 = [:0, 1:]"
| "tangent_poly (Suc n) = pderiv (tangent_poly n) * [:1,0,1:]"

lemma degree_tangent_poly [simp]: "degree (tangent_poly n) = n + 1"
  by (induction n)
     (auto simp: degree_mult_eq pderiv_eq_0_iff degree_pderiv simp del:
mult_pCons_right)

lemma tangent_poly_altdef [code]:
  "tangent_poly n = ((λp. pderiv p * [:1,0,1:]) ^^ n) [:0, 1:]"
  by (induction n) simp_all

lemma fps_tan_higher_deriv':
  "(fps_deriv ^^ n) (fps_tan (1::'a::field_char_0)) =
     fps_compose (fps_of_poly (map_poly of_nat (tangent_poly n))) (fps_tan
1)"
proof -
  interpret of_nat_poly_hom: map_poly_comm_semiring_hom of_nat
    by standard auto
  show ?thesis
    by (induction n)
       (simp_all add: hom_distribs fps_of_poly_pderiv fps_of_poly_add
                      fps_of_poly_pCons fps_compose_add_distrib fps_compose_mult_distrib
                      fps_compose_deriv fps_tan_deriv' power2_eq_square
of_nat_poly_pderiv)
qed

theorem fps_tan_higher_deriv:
  "(fps_deriv ^^ n) (fps_tan 1) =
     poly (map_poly of_int (tangent_poly n)) (fps_tan (1::'a::field_char_0))"
  using fps_tan_higher_deriv'[of n]
  by (subst (asm) fps_compose_of_poly)
     (simp_all add: map_poly_map_poly o_def fps_of_nat)
```

For easier notation, we give the name "auxiliary tangent numbers" to the coefficients of these polynomials and treat them as a number triangle $T_{n,j}$. These will aid us in the computation of the actual tangent numbers later.

```
definition tangent_number_aux :: "nat ⇒ nat ⇒ nat" where
  "tangent_number_aux n j = poly.coeff (tangent_poly n) j"
```

The coefficients satisfy the following recurrence and boundary conditions:

- $T_{0,1} = 1$
- $T_{0,j} = 0$ if $j \neq 1$
- $T_{n,j} = 0$ if $j > n + 1$ or $n + j$ even
- $T_{n,n+1} = n!$
- $T_{n+1,j+1} = jT_{n,j} + (j + 2)T_{n,j+2}$

```
lemma tangent_number_aux_0_left:
  "tangent_number_aux 0 j = (if j = 1 then 1 else 0)"
  unfolding tangent_number_aux_def by (auto simp: coeff_pCons split: nat.splits)

lemma tangent_number_aux_0_left' [simp]:
  "j ≠ 1 ⟹ tangent_number_aux 0 j = 0"
  "tangent_number_aux 0 (Suc 0) = 1"
  by (simp_all add: tangent_number_aux_0_left)

lemma tangent_number_aux_0_right:
  "tangent_number_aux (Suc n) 0 = poly.coeff (tangent_poly n) 1"
  unfolding tangent_number_aux_def tangent_poly.simps by (auto simp: coeff_pderiv)

lemma tangent_number_aux_rec:
  "tangent_number_aux (Suc n) (Suc j) = j * tangent_number_aux n j + (j
+ 2) * tangent_number_aux n (j + 2)"
  unfolding tangent_number_aux_def tangent_poly.simps
  by (simp_all add: coeff_pderiv coeff_pCons split: nat.splits)

lemma tangent_number_aux_rec':
  "n > 0 ⟹ j > 0 ⟹ tangent_number_aux n j = (j-1) * tangent_number_aux
(n-1) (j-1) + (j+1) * tangent_number_aux (n-1) (j+1)"
  using tangent_number_aux_rec[of "n-1" "j-1"] by simp

lemma tangent_number_aux_odd_eq_0: "even (n + j) ⟹ tangent_number_aux
n j = 0"
  unfolding tangent_number_aux_def
  by (induction n arbitrary: j)
    (auto simp: coeff_pCons coeff_pderiv split: nat.splits)
```

```
lemma tangent_number_aux_eq_0 [simp]: "j > n + 1 ⟹ tangent_number_aux
n j = 0"
  unfolding tangent_number_aux_def by (simp add: coeff_eq_0)

lemma tangent_number_aux_last [simp]: "tangent_number_aux n (Suc n) =
fact n"
  by (induction n) (auto simp: tangent_number_aux_rec)

lemma tangent_number_aux_last': "Suc m = n ⟹ tangent_number_aux m
n = fact m"
  by (cases n) auto

lemma tangent_number_aux_1_right [simp]:
  "tangent_number_aux i (Suc 0) = tangent_number_aux (i + 1) 0"
  by (simp add: tangent_number_aux_def coeff_pderiv)
```

4.2 The tangent numbers

The actual secant numbers T_n are now defined to be the even-index coefficients of the power series expansion of $\tan x$ (the even-index ones are all 0). [3, A000182]

This also turns out to be exactly the same as $T_{n,0}$.

```
definition tangent_number :: "nat ⇒ nat" where
  "tangent_number n = nat (floor (fps_nth (fps_tan 1) (2*n-1) * fact (2*n-1)
:: real))"

lemma tangent_number_conv_zigzag_number:
  "n > 0 ⟹ tangent_number n = zigzag_number (2 * n - 1)"
  unfolding tangent_number_def
  by (subst zigzag_number_conv_fps_tan [symmetric]) auto

lemma tangent_number_0 [simp]: "tangent_number 0 = 0"
  by (simp add: tangent_number_def fps_tan_def)

lemma fps_nth_tan_aux:
  "fps_tan (1::'a::field_char_0) $ (2*n-1) =
     of_nat (tangent_number_aux (2*n-1) 0) / fact (2*n-1)"
proof (cases "n = 0")
  case False
  interpret of_nat_poly_hom: map_poly_comm_semiring_hom of_nat
    by standard auto
  from False have n: "n > 0"
    by simp
  have "fps_nth ((fps_deriv ^^ (2 * n - 1)) (fps_tan (1::'a))) 0 =
          fact (2*n-1) * fps_nth (fps_tan 1) (2*n-1)"
    by (simp add: fps_0th_higher_deriv)
  also have "(fps_deriv ^^ (2*n-1)) (fps_tan (1::'a)) =
               fps_of_poly (map_poly of_nat (tangent_poly (2*n-1))) oo
```

```
fps_tan 1"
    by (subst fps_tan_higher_deriv') auto
  also have "fps_nth ... 0 = of_nat (tangent_number_aux (2*n-1) 0)"
    by (simp add: tangent_number_aux_def)
  finally show ?thesis
    by simp
qed auto

lemma fps_nth_tan:
  "fps_nth (fps_tan (1::'a :: field_char_0)) (2*n - Suc 0) = of_int (tangent_number
n) / fact (2*n-1)"
  using fps_nth_tan_aux[of n, where ?'a = real] fps_nth_tan_aux[of n,
where ?'a = 'a]
  by (simp add: tangent_number_def)

lemma tangent_number_conv_aux [code]:
  "tangent_number n = tangent_number_aux (2*n - Suc 0) 0"
  using fps_nth_tan[of n, where ?'a = real] fps_nth_tan_aux[of n, where
?'a = real] by simp

lemma tangent_number_1 [simp]: "tangent_number (Suc 0) = 1"
  by (simp add: tangent_number_conv_aux tangent_number_aux_0_right)
```

The tangent number T_n can be expressed in terms of the Bernoulli number $\mathcal{B}_n$:

```
theorem tangent_number_conv_bernoulli:
   "2 * real n * of_int (tangent_number n) =
      (-1)^(n+1) * (2^(2*n) * (2^(2*n) - 1)) * bernoulli (2*n)"
proof -
  define F where "F = (λc::complex. fps_compose bernoulli_fps (fps_const
c * fps_X))"
  define E where "E = (λc::complex. fps_to_fls (fps_exp c))"
  have neqI1: "f ≠ g" if "fls_nth f 0 ≠ fls_nth g 0" for f g :: "complex
fls"
    using that by metis
  have [simp]: "fls_nth (E c) n = c ^ nat n / (fact (nat n))" if "n ≥
0" for n c
    using that by (auto simp: E_def)

  have [simp]: "subdegree (1 - fps_exp 1 :: complex fps) = 1"
    by (rule subdegreeI) auto
  have "fps_to_fls (F (2*i) - F (4*i) - fps_const i * fps_X) =
         2 * fls_const i * fls_X / (E (2*i) - 1) -
         4 * fls_const i * fls_X / (E (4*i) - 1) -
         fls_const i * fls_X"
    unfolding F_def bernoulli_fps_def E_def
    apply (simp flip: fls_compose_fps_to_fls)
    apply (simp add: fls_compose_fps_divide fls_times_fps_to_fls fls_compose_fps_diff
               flip: fls_const_mult_const fls_divide_fps_to_fls)
```

```
    done
  also have "E (4 * i) = E (2 * i) ^ 2"
    by (simp add: fps_exp_power_mult E_def flip: fps_to_fls_power)
  also have "E (2 * i) ^ 2 - 1 = (E (2 * i) - 1) * (E (2 * i) + 1)"
    by (simp add: algebra_simps power2_eq_square)
  also have "2 * fls_const i * fls_X / (E (2 * i) - 1) -
            4 * fls_const i * fls_X / ((E (2 * i) - 1) * (E (2 * i) +
1)) =
            2 * fls_const i * fls_X * (1 / (E (2 * i) + 1))"
    unfolding E_def
    apply (simp add: divide_simps)
    apply (auto simp: algebra_simps add_eq_0_iff fls_times_fps_to_fls
neqI1)
    done
  also have "1 / (E (2 * i) + 1) = E (-i) / (E (-i) * (E (2 * i) + 1))"
    by (simp add: divide_simps add_eq_0_iff2 neqI1)
  also have "E (-i) * (E (2 * i) + 1) = E i + E (-i)"
    by (simp add: E_def algebra_simps flip: fls_times_fps_to_fls fps_exp_add_mult)
  also have "2 * fls_const i * fls_X * (E (-i) / (E i + E (-i))) - fls_const
i * fls_X =
            fls_X * (fls_const (-i) * (1 - 2 * E (-i) / (E i + E (-i))))"
    by (simp add: algebra_simps)
  also have "1 - 2 * E (-i) / (E i + E (-i)) = (E i - E (-i)) / (E i + E
(-i))"
    by (simp add: divide_simps neqI1)
  also have "fls_const (-i) * ... = (-fls_const i/2 * (E i - E (-i))) /
((E i + E (-i)) / 2)"
    by (simp add: divide_simps neqI1)
  also have "-fls_const i / 2 * (E i - E (-i)) = fps_to_fls (fps_sin 1)"
    by (simp add: fps_sin_fps_exp_ii E_def fls_times_fps_to_fls flip:
fls_const_divide_const)
  also have "(E i + E (-i)) / 2 = fps_to_fls (fps_cos 1)"
    by (simp add: fps_cos_fps_exp_ii E_def fls_times_fps_to_fls flip:
fls_const_divide_const)
  also have "fls_X * (fps_to_fls (fps_sin 1) / fps_to_fls (fps_cos 1))
=
              fps_to_fls (fps_X * fps_tan (1::complex))"
    by (simp add: fps_tan_def fls_times_fps_to_fls flip: fls_divide_fps_to_fls)
  finally have eq: "F (2 * i) - F (4 * i) - fps_const i * fps_X =
                 fps_X * fps_tan 1" (is "?lhs = ?rhs")
    by (simp only: fps_to_fls_eq_iff)

  show "2 * real n * of_int (tangent_number n) =
          (-1)^(n+1) * (2^(2*n) * (2^(2*n) - 1)) * bernoulli (2*n)"
  proof (cases "n = 0")
    case False
    hence n: "n > 0"
      by simp
    have "fps_nth ?lhs (2*n) = (-1)^n * (2^(2*n) - 4^(2*n)) * of_real
```

```
(bernoulli (2 * n)) / fact (2*n)"
      using n unfolding F_def fps_nth_compose_linear fps_sub_nth
      by (simp add: algebra_simps diff_divide_distrib)
    also note <?lhs = ?rhs>
    also have "fps_nth ?rhs (2*n) = complex_of_int (tangent_number n)
/ fact (2 * n - 1)"
      using n by (simp add: fps_nth_tan)
    finally have "complex_of_int (tangent_number n) * (fact (2*n) / fact
(2 * n - 1)) =
                  (- 1) ^ n * (2 ^ (2 * n) - 4 ^ (2 * n)) * complex_of_real
(bernoulli (2 * n))"
      by (simp add: divide_simps)
    also have "complex_of_int (tangent_number n) * (fact (2*n) / fact
(2 * n - 1)) =
               of_real (fact (2*n) / fact (2 * n - 1) * of_int (tangent_number
n))"
      by (simp add: field_simps)
    also have "fact (2*n) / fact (2 * n - 1) = (2 * of_nat n :: real)"
      using fact_binomial[of 1 "2 * n", where ?'a = real] n by simp
    also have "2 ^ (2 * n) - 4 ^ (2 * n) = -(2 ^ (2 * n) * (2 ^ (2 * n)
- 1 :: complex))"
      by (simp add: algebra_simps flip: power_mult_distrib)
    also have "(- 1) ^ n * - (2 ^ (2 * n) * (2 ^ (2 * n) - 1)) * complex_of_real
(bernoulli (2 * n)) =
               of_real ((-1)^(n+1) * (2^(2*n) * (2^(2*n) - 1)) * bernoulli
(2*n))"
      by simp
    finally show ?thesis
      by (simp only: of_real_eq_iff)
  qed auto
qed
```

4.3 Efficient functional computation

We will now formalise and verify an algorithm to compute the first n tangent numbers relatively efficiently via the auxiliary tangent numbers. The algorithm is a functional variant of the imperative in-place algorithm given by Brent et al. [1]. The functional algorithm could easily be adapted to one that returns a stream of all tangent numbers instead of a list of the first n of them.

The algorithm uses $O(n^2)$ additions and multiplications on integers, but since the numbers grow up to $\Theta(n \log n)$ bits, this translates to $O(n^3 \log 1 + \varepsilon n)$ bit operations.

Note that Brent et al. only define the tangent numbers T_n starting with $n = 1$, whereas we also defined $T_0 = 0$. The algorithm only computes $T_1, \ldots, T_n$.

```
function pochhammer_row_impl :: "nat ⇒ nat ⇒ nat ⇒ nat list" where
```

```
  "pochhammer_row_impl k n x = (if k ≥ n then [] else x # pochhammer_row_impl
(Suc k) n (x * k))"
  by auto
termination by (relation "measure (λ(k,n,_) ⇒ n - k)") auto

lemmas [simp del] = pochhammer_row_impl.simps

lemma pochhammer_rec'': "k > 0 ⟹ pochhammer n k = n * pochhammer (n+1)
(k-1)"
  by (cases k) (auto simp: pochhammer_rec)

lemma pochhammer_row_impl_correct:
  "pochhammer_row_impl k n x = map (λi. x * pochhammer k i) [0..<n-k]"
proof (induction k n x rule: pochhammer_row_impl.induct)
  case (1 k n x)
  show ?case
  proof (cases "k < n")
    case True
    have "pochhammer_row_impl k n x = x # map (λi. x * k * pochhammer
(Suc k) i) [0..<n - (k + 1)]"
      using True by (subst pochhammer_row_impl.simps) (simp_all add: "1.IH")
    also have "map (λi. x * k * pochhammer (Suc k) i) [0..<n - (k + 1)]
=
                 map (λi. x * pochhammer k i) (map Suc [0..<n - (k + 1)])"
      by (simp add: pochhammer_rec)
    also have "map Suc [0..<n - (k + 1)] = [Suc 0..<n-k]"
      using True by (simp add: map_Suc_upt Suc_diff_Suc del: upt_Suc)
    also have "x # map (λi. x * pochhammer k i) [Suc 0..<n-k] =
                 map (λi. x * pochhammer k i) (0 # [Suc 0..<n-k])"
      by simp
    also have "0 # [Suc 0..<n-k] = [0..<n-k]"
      using True by (subst upt_conv_Cons) auto
    finally show ?thesis .
  qed (subst pochhammer_row_impl.simps; auto)
qed


context
  fixes T :: "nat ⇒ nat ⇒ nat"
  defines "T ≡ tangent_number_aux"
begin

primrec tangent_number_impl_aux1 :: "nat ⇒ nat ⇒ nat list ⇒ nat list"
where
  "tangent_number_impl_aux1 j y [] = []"
| "tangent_number_impl_aux1 j y (x # xs) =
     (let x' = j * y + (j+2) * x in x' # tangent_number_impl_aux1 (j+1)
x' xs)"
```

```
lemma length_tangent_number_impl_aux1 [simp]: "length (tangent_number_impl_aux1
j y xs) = length xs"
  by (induction xs arbitrary: j y) (simp_all add: Let_def)

fun tangent_number_impl_aux2 :: "nat list ⇒ nat list" where
  "tangent_number_impl_aux2 [] = []"
| "tangent_number_impl_aux2 (x # xs) = x # tangent_number_impl_aux2 (tangent_number_impl_aux
0 x xs)"

lemma tangent_number_impl_aux1_nth_eq:
  assumes "i < length xs"
  shows   "tangent_number_impl_aux1 j y xs ! i =
             (j+i) * (if i = 0 then y else tangent_number_impl_aux1 j
y xs ! (i-1)) + (j+i+2) * xs ! i"
  using assms
proof (induction xs arbitrary: i j y)
  case (Cons x xs)
  show ?case
  proof (cases i)
    case 0
    thus ?thesis
      by (simp add: Let_def)
  next
    case (Suc i')
    define x' where "x' = j * y + (x + (x + j * x))"
    have "tangent_number_impl_aux1 j y (x # xs) ! i = tangent_number_impl_aux1
(Suc j) x' xs ! i'"
      by (simp add: x'_def Let_def Suc)
    also have "... = (Suc j + i') * (if i' = 0 then x' else tangent_number_impl_aux1
(Suc j) x' xs ! (i'-1)) +
                   (Suc j + i' + 2) * xs ! i'"
      using Cons.prems by (subst Cons.IH) (auto simp: Suc)
    also have "Suc j + i' = j + i"
      by (simp add: Suc)
    also have "xs ! i' = (x # xs) ! i"
      by (auto simp: Suc)
    also have "(if i' = 0 then x' else tangent_number_impl_aux1 (Suc j)
x' xs ! (i'-1)) =
               (x' # tangent_number_impl_aux1 j y (x # xs)) ! i"
      by (auto simp: Suc x'_def Let_def)
    finally show ?thesis
      by (simp add: Suc)
  qed
qed auto

lemma tangent_number_impl_aux2_correct:
  assumes "k ≤ n"
  shows  "tangent_number_impl_aux2 (map (λi. T (2 * k + i) (i + 1)) [0..<n-k])
=
```

```
             map tangent_number [Suc k..<Suc n]"
  using assms
proof (induction k rule: inc_induct)
  case (step k)
  have *: "[0..<n-k] = 0 # map Suc [0..<n-Suc k]"
    by (subst upt_conv_Cons)
       (use step.hyps in ‹auto simp: map_Suc_upt Suc_diff_Suc simp del:
upt_Suc›)
  define ts where
    "ts = tangent_number_impl_aux1 0 (T (2*k) 1) (map (λi. T (2*k+i+1)
(i+2)) [0..<n-Suc k])"
  have T_rec: "T (Suc a) (Suc b) = b * T a b + (b + 2) * T a (b + 2)"
for a b
    unfolding T_def tangent_number_aux_rec ..

  have "tangent_number_impl_aux2 (map (λi. T (2 * k + i) (i + 1)) [0..<n-k])
=
          T (2 * k) 1 # tangent_number_impl_aux2 ts"
    unfolding * list.map tangent_number_impl_aux2.simps
    by (simp add: o_def ts_def algebra_simps numeral_3_eq_3)
  also have "ts = map (λi. T (2 * Suc k + i) (i + 1)) [0..<n - Suc k]"
  proof (rule nth_equalityI)
    fix i assume "i < length ts"
    hence i: "i < n - Suc k"
      by (simp add: ts_def)
    hence "ts ! i = T (2 * Suc k + i) (i + 1)"
    proof (induction i)
      case 0
      thus ?case unfolding ts_def
        by (subst tangent_number_impl_aux1_nth_eq)
           (use T_rec[of "2*k+1" 0] in ‹auto simp: eval_nat_numeral›)
    next
      case (Suc i)
      have "ts ! Suc i = Suc i * T (Suc (Suc (2 * k + i))) (Suc i) +
                  (Suc i + 2) * T (Suc (Suc (2 * k + i))) (Suc i + 2)"
        using Suc unfolding ts_def
        by (subst tangent_number_impl_aux1_nth_eq) (auto simp: eval_nat_numeral)
      also have "... = T (2 * Suc k + Suc i) (Suc i + 1)"
        using T_rec[of "2 * Suc k + i" "Suc i"] by simp
      finally show ?case .
    qed
    thus "ts ! i = map (λi. T (2 * Suc k + i) (i + 1)) [0..<n - Suc k]
! i"
      using i by simp
  qed (simp_all add: ts_def)
  also have "tangent_number_impl_aux2 ... = map tangent_number [Suc (Suc
k)..<Suc n]"
    by (rule step.IH)
  also have "T (2 * k) 1 = tangent_number (Suc k)"
```

```
    by (simp add: tangent_number_conv_aux T_def)
  also have "tangent_number (Suc k) # map tangent_number [Suc (Suc k)..<Suc
n] =
             map tangent_number [Suc k..<Suc n]"
    using step.hyps by (subst upt_conv_Cons) (auto simp del: upt_Suc)
  finally show ?case .
qed auto

definition tangent_numbers :: "nat ⇒ nat list" where
  "tangent_numbers n = map tangent_number [1..<Suc n]"

lemma tangent_numbers_code [code]:
  "tangent_numbers n = tangent_number_impl_aux2 (pochhammer_row_impl 1
(Suc n) 1)"
proof -
  have "pochhammer_row_impl 1 (Suc n) 1 = map (λi. T i (i + 1)) [0..<n]"
    by (simp add: pochhammer_row_impl_correct pochhammer_fact T_def)
  also have "tangent_number_impl_aux2 ... = map tangent_number [Suc 0..<Suc
n]"
    using tangent_number_impl_aux2_correct[of 0 n] by (simp del: upt_Suc)
  finally show ?thesis
    by (simp only: tangent_numbers_def One_nat_def)
qed

lemma tangent_number_code [code]:
  "tangent_number n = (if n = 0 then 0 else last (tangent_numbers n))"
  by (simp add: tangent_numbers_def)

end

end
```

4.4 Imperative in-place computation

```
theory Tangent_Numbers_Imperative
  imports Tangent_Numbers "Refine_Monadic.Refine_Monadic" "Refine_Imperative_HOL.IICF"
"HOL-Library.Code_Target_Numeral"
begin
```

We will now formalise and verify the imperative in-place version of the algorithm given by Brent et al. [1]. We use as storage only an array of n numbers, which will also contain the results in the end. Note however that the size of these numbers grows enormously the longer the algorithm runs.

```
locale tangent_numbers_imperative
begin

context
  fixes n :: nat
begin
```

```
definition I_init :: "nat list × nat ⇒ bool" where
  "I_init = (λ(xs, i).
     (n = 0 ∧ i = 1 ∧ xs = []) ∨
     (i ∈ {1..n} ∧ xs = map fact [0..<i] @ replicate (n-i) 0))"
```

```
definition init_loop_aux :: "nat list nres" where
  "init_loop_aux =
     do {xs ← RETURN (op_array_replicate n 0);
         (if n = 0 then RETURN xs else do {ASSERT (length xs > 0); RETURN
(xs[0 := 1])})}"
```

```
definition init_loop :: "nat list nres" where
  "init_loop =
     do {
       xs ← init_loop_aux;
       (xs', _) ←
         WHILE_T^{I_init}
           (λ(_, i). i < n)
           (λ(xs, i). do {
             ASSERT (i - 1 < length xs);
             x ← RETURN (xs ! (i - 1));
             ASSERT (i < length xs);
             RETURN (xs[i := i * x], i + 1)
           })
           (xs, 1);
       RETURN xs'
     }"
```

```
definition I_inner where
  "I_inner xs i = (λ(xs', j). j ∈ {i..n} ∧ length xs' = n ∧
     (∀k<n. xs' ! k = (if k∈{i..<j} then tangent_number_aux (k+Suc i-1)
(k+2-Suc i) else xs ! k)))"
```

```
definition inner_loop :: "nat list ⇒ nat ⇒ nat list nres" where
  "inner_loop xs i =
     do {
       (xs', _) ←
         WHILE_T^{I_inner xs i} (λ(_, j). j < n)
         (λ(xs, j). do {
           ASSERT (j - 1 < length xs);
           x ← RETURN (xs ! (j - 1));
           ASSERT (j < length xs);
           y ← RETURN (xs ! j);
           RETURN (xs[j := (j - i) * x + (j - i + 2) * y], j + 1)
         })
         (xs, i);
       RETURN xs'
     }"
```

```
definition I_compute :: "nat list × nat ⇒ bool" where
  "I_compute = (λ(xs, i). (n = 0 ∧ i = 1 ∧ xs = []) ∨
     (i ∈ {1..n} ∧ xs = map (λk. if k < i then tangent_number (k+1) else
tangent_number_aux (k+i-1) (k+2-i)) [0..<n]))"

definition compute :: "nat list nres" where
  "compute =
     do {
       xs ← init_loop;
       (xs', _) ←
         WHILE_T^I_compute
           (λ(_, i). i < n)
           (λ(xs, i). do { xs' ← inner_loop xs i; RETURN (xs', i + 1)
})
           (xs, 1);
       RETURN xs'
     }"

lemma init_loop_aux_correct [refine_vcg]:
  "init_loop_aux ≤ SPEC (λxs. xs = (replicate n 0)[0 := 1])"
  unfolding init_loop_aux_def
  by refine_vcg auto

lemma init_loop_correct [refine_vcg]: "init_loop ≤ SPEC (λxs. xs = map
fact [0..<n])"
  unfolding init_loop_def
  apply refine_vcg
  apply (rule wf_measure[of "λ(_, i). n - i"])
  subgoal
    by (auto simp: I_init_def nth_list_update' intro!: nth_equalityI)
  subgoal
    by (auto simp: I_init_def)
  subgoal
    by (auto simp: I_init_def)
  subgoal
    by (auto simp: I_init_def nth_list_update' fact_reduce nth_Cons nth_append
          intro!: nth_equalityI split: nat.splits)
  subgoal
    by auto
  subgoal
    by (auto simp: I_init_def)
  done

lemma I_inner_preserve:
  assumes invar: "I_inner xs i (xs', j)" and invar': "I_compute (xs,
i)"
  assumes j: "j < n"
  defines "y ≡ (j - i) * xs' ! (j - 1) + (j - i + 2) * xs' ! j"
```

```
  defines "xs'' ≡ list_update xs' j y"
  shows   "I_inner xs i (xs'', j + 1)"
  unfolding I_inner_def
proof safe
  show "j + 1 ∈ {i..n}" "length xs'' = n"
    using invar j by (simp_all add: xs''_def I_inner_def)
next
  fix k assume k: "k < n"
  define T where "T = tangent_number_aux"
  have ij: "1 ≤ i" "i ≤ j" "j < n"
    using invar invar' j by (auto simp: I_inner_def I_compute_def)
  have nth_xs': "xs' ! k = (if k ∈ {i..<j} then T (k + Suc i - 1) (k
+ 2 - Suc i) else xs ! k)"
    if "k < n" for k using invar that unfolding I_inner_def T_def by blast
  have nth_xs: "xs ! k = (if k < i then tangent_number (k + 1)
                          else T (k + i - 1) (k + 2 - i))"
    if "k < n" for k using invar' that unfolding I_compute_def T_def by
auto
  have [simp]: "length xs' = n"
    using invar by (simp add: I_inner_def)

  consider "k = j" | "k ∈ {i..<j}" | "k ∉ {i..j}"
    by force
  thus "xs'' ! k = (if k ∈ {i..<j + 1} then T (k + Suc i - 1) (k + 2 -
Suc i) else xs ! k)"
  proof cases
    assume [simp]: "k = j"
    have "xs'' ! k = y"
      using ij by (simp add: xs''_def)
    also have "... = (j - i) * xs' ! (j - 1) + (j - i + 2) * xs' ! j"
      by (simp add: y_def)
    also have "xs' ! j = xs ! j"
      using ij by (subst nth_xs') auto
    also have "... = T (j + i - 1) (j + 2 - i)"
      using ij by (subst nth_xs) auto
    also have "xs' ! (j - 1) = (if i = j then xs ! (i - 1) else T (j +
i - 1) (j - i))"
      using ij by (subst nth_xs')  auto
    also have "xs ! (i - 1) = T (2 * i - 1) 0"
      using ij by (subst nth_xs) (auto simp: tangent_number_conv_aux T_def)
    also have "(if i = j then T (2 * i - 1) 0 else T (j + i - 1) (j -
i)) = T (j + i - 1) (j - i)"
      by (auto simp: mult_2)
    also have "(j - i) * T (j + i - 1) (j - i) + (j - i + 2) * T (j +
i - 1) (j + 2 - i) =
               T (j + i) (j + 1 - i)"
      unfolding T_def by (subst (3) tangent_number_aux_rec') (use ij in
auto)
    finally show ?thesis
```

```
      using ij by simp
  next
    assume k: "k ∈ {i..<j}"
    hence "xs'' ! k = xs' ! k"
      unfolding xs''_def by auto
    also have "... = T (k + i) (Suc k - i)"
      by (subst nth_xs') (use k ij in auto)
    finally show ?thesis
      using k by simp
  next
    assume k: "k ∉ {i..j}"
    hence "xs'' ! k = xs' ! k"
      using ij unfolding xs''_def by auto
    also have "xs' ! k = xs ! k"
      using k <k < n> by (subst nth_xs') auto
    finally show ?thesis
      using k by auto
  qed
qed

lemma inner_loop_correct [refine_vcg]:
  assumes "I_compute (xs, i)" "i < n"
  shows "inner_loop xs i ≤ SPEC (λxs'. xs' =
          map (λk. if k ≥ i then tangent_number_aux (k+Suc i-1) (k+2-Suc
i) else xs ! k) [0..<n])"
  unfolding inner_loop_def
  apply refine_vcg
      apply (rule wf_measure[of "λ(_, j). n - j"])
  subgoal
    using assms by (auto simp: I_inner_def I_compute_def)
  subgoal
    using assms unfolding I_inner_def by auto
  subgoal
    using assms unfolding I_inner_def by auto
  subgoal for s xs' j
    using I_inner_preserve[of xs i xs' j] assms by auto
  subgoal
    by auto
  subgoal using assms
    by (auto simp: I_inner_def intro!: nth_equalityI)
  done

lemma compute_correct [refine_vcg]: "compute ≤ SPEC (λxs'. xs' = tangent_numbers
n)"
  unfolding compute_def
  apply refine_vcg
     apply (rule wf_measure[of "λ(_, i). n - i"])
  subgoal
    by (auto simp: I_compute_def tangent_number_aux_last')
```

```
  subgoal
    by (auto simp: I_compute_def tangent_number_conv_aux less_Suc_eq mult_2)
  subgoal
    by auto
  subgoal
    by (auto simp: I_compute_def tangent_number_conv_aux less_Suc_eq mult_2
intro!: nth_equalityI)
  subgoal
    by auto
  subgoal
    by (auto simp: I_compute_def tangent_numbers_def intro!: nth_equalityI
simp del: upt_Suc)
  done

lemmas defs =
  compute_def inner_loop_def init_loop_def init_loop_aux_def

end

sepref_definition compute_imp is
  "tangent_numbers_imperative.compute" ::
     "nat_assn^d →_a array_assn nat_assn"
  unfolding tangent_numbers_imperative.defs by sepref

lemma imp_correct':
  "(compute_imp, λn. RETURN (tangent_numbers n)) ∈ nat_assn^d →_a array_assn
nat_assn"
proof -
  have *: "(compute, λn. RETURN (tangent_numbers n)) ∈ nat_rel → ⟨Id⟩nres_rel"
    by refine_vcg simp?
  show ?thesis
    using compute_imp.refine[FCOMP *] .
qed

theorem imp_correct:
   "<nat_assn n n> compute_imp n <array_assn nat_assn (tangent_numbers
n)>_t"
proof -
  have [simp]: "nofail (compute n)"
    using compute_correct[of n] le_RES_nofailI by blast
  have 1: "xs = tangent_numbers n" if "RETURN xs ≤ compute n" for xs
    using that compute_correct[of n] by (simp add: pw_le_iff)
  note rl = compute_imp.refine[THEN hfrefD, of n n, THEN hn_refineD, simplified]
  show ?thesis
    apply (rule cons_rule[OF _ _ rl])
    apply (sep_auto simp: pure_def)
    apply (sep_auto simp: pure_def dest!: 1)
    done
qed
```

```
end

lemmas [code] = tangent_numbers_imperative.compute_imp_def

end
```

5 Secant numbers

```
theory Secant_Numbers
  imports
  "HOL-Computational_Algebra.Computational_Algebra"
  "Polynomial_Interpolation.Ring_Hom_Poly"
  Boustrophedon_Transform_Library
  Alternating_Permutations
  Tangent_Numbers
begin
```

5.1 The higher derivatives of $\sec x$

Similarly to what we saw with tangent numbers, the n-th derivatives of $\sec x$ do not follow an easily discernible pattern, but they can all be expressed in the form $\sec x P_n(\tan x)$, where P_n is a polynomial of degree n.

Using the facts that $\sec' x = \sec x \tan x$ and $\tan' x = 1 + \tan^2 x$ and the chain rule, one can see that P_n must satisfy the recurrence $P_{n+1}(X) = XP(X) + (1 + X^2)P'(X)$.

```
primrec secant_poly :: "nat ⇒ nat poly" where
  "secant_poly 0 = 1"
| "secant_poly (Suc n) = (let p = secant_poly n in p * [:0, 1:] + pderiv
p * [:1, 0, 1:])"

lemmas [simp del] = secant_poly.simps(2)

lemma degree_secant_poly [simp]: "degree (secant_poly n) = n"
proof (induction n)
  case (Suc n)
  define p where "p = secant_poly n"
  define q where "q = p * [:0, 1:]"
  define r where "r = pderiv p * [:1, 0, 1:]"
  have p: "degree p = n"
    using Suc.IH by (simp add: p_def)
  show ?case
  proof (cases "n = 0")
    case [simp]: True
    show ?thesis
      by (auto simp: secant_poly.simps(2))
  next
```

```
    case n: False
    have [simp]: "p ≠ 0" "pderiv p ≠ 0"
      using p n by (auto simp: pderiv_eq_0_iff)
    have q: "degree q = Suc n"
      unfolding q_def by (subst degree_mult_eq) (use p in auto)
    have r: "degree r = Suc n"
      unfolding r_def by (subst degree_mult_eq) (use p n in <auto simp:
degree_pderiv>)

    have "secant_poly (Suc n) = q + r"
      by (simp add: Let_def secant_poly.simps(2) p_def q_def r_def)
    also have "degree ... = Suc n"
    proof (rule antisym)
      show "degree (q + r) ≤ Suc n"
        using n by (intro degree_add_le) (auto simp: q r)
      show "degree (q + r) ≥ Suc n"
      proof (rule le_degree)
        have "poly.coeff (q + r) (Suc n) = lead_coeff q + lead_coeff r"
          by (simp add: q r)
        also have "... = Suc (degree p) * lead_coeff p"
          by (simp add: q_def r_def lead_coeff_mult lead_coeff_pderiv
del: mult_pCons_right)
        also have "... ≠ 0"
          by (subst mult_eq_0_iff) auto
        finally show "poly.coeff (q + r) (Suc n) ≠ 0" .
      qed
    qed
    finally show ?thesis .
  qed
qed auto

lemma secant_poly_altdef [code]:
  "secant_poly n = ((λp. p * [:0,1:] + pderiv p * [:1, 0, 1:]) ^^ n) 1"
  by (induction n) (simp_all add: secant_poly.simps(2) Let_def)

lemma fps_sec_higher_deriv':
  "(fps_deriv ^^ n) (fps_sec (1::'a::field_char_0)) =
     fps_sec 1 * fps_compose (fps_of_poly (map_poly of_nat (secant_poly
n))) (fps_tan 1)"
proof -
  interpret of_nat_poly_hom: map_poly_comm_semiring_hom of_nat
    by standard auto
  show ?thesis
    by (induction n)
       (simp_all add: hom_distribs fps_of_poly_pderiv fps_of_poly_add
fps_sec_deriv
                      fps_of_poly_pCons fps_compose_add_distrib fps_compose_mult_distrib
                      fps_compose_deriv fps_tan_deriv' power2_eq_square
of_nat_poly_pderiv
```

```
                          secant_poly.simps(2) Let_def)
qed

theorem fps_sec_higher_deriv:
  "(fps_deriv ^^ n) (fps_sec 1) =
     fps_sec 1 * poly (map_poly of_int (secant_poly n)) (fps_tan (1::'a::field_char_0))"
  using fps_sec_higher_deriv'[of n]
  by (subst (asm) fps_compose_of_poly)
     (simp_all add: map_poly_map_poly o_def fps_of_nat)
```

For easier notation, we give the name "auxiliary secant numbers" to the coefficients of these polynomials and treat them as a number triangle $S_{n,j}$. These will aid us in the computation of the actual secant numbers later.

```
definition secant_number_aux :: "nat ⇒ nat ⇒ nat" where
  "secant_number_aux n j = poly.coeff (secant_poly n) j"
```

The coefficients satisfy the following recurrence and boundary conditions:

- $S_{0,0} = 1$
- $S_{n,j} = 0$ if $j > n$ or $n + j$ odd
- $S_{n,n} = n!$
- $S_{n,j} = (j + 1)S_{n,j} + (j + 2)S_{n,j+2}$

```
lemma secant_number_aux_0_left:
  "secant_number_aux 0 j = (if j = 0 then 1 else 0)"
  unfolding secant_number_aux_def by (auto simp: coeff_pCons split: nat.splits)

lemma secant_number_aux_0_left' [simp]:
  "j ≠ 0 ⟹ secant_number_aux 0 j = 0"
  "secant_number_aux 0 0 = 1"
  by (simp_all add: secant_number_aux_0_left)

lemma secant_number_aux_0_right:
  "secant_number_aux (Suc n) 0 = secant_number_aux n 1"
  unfolding secant_number_aux_def secant_poly.simps by (auto simp: coeff_pderiv
Let_def)

lemma secant_number_aux_rec:
  "secant_number_aux (Suc n) (Suc j) =
     (j+1) * secant_number_aux n j + (j + 2) * secant_number_aux n (j
+ 2)"
  unfolding secant_number_aux_def secant_poly.simps
  by (simp_all add: coeff_pderiv coeff_pCons Let_def split: nat.splits)

lemma secant_number_aux_rec':
```

```
  "n > 0 ⟹ j > 0 ⟹ secant_number_aux n j = j * secant_number_aux (n-1)
(j-1) + (j+1) * secant_number_aux (n-1) (j+1)"
  using secant_number_aux_rec[of "n-1" "j-1"] by simp

lemma secant_number_aux_odd_eq_0: "odd (n + j) ⟹ secant_number_aux
n j = 0"
  unfolding secant_number_aux_def
  by (induction n arbitrary: j)
     (auto simp: coeff_pCons coeff_pderiv secant_poly.simps(2) Let_def
elim: oddE split: nat.splits)

lemma secant_number_aux_eq_0 [simp]: "j > n ⟹ secant_number_aux n
j = 0"
  unfolding secant_number_aux_def by (simp add: coeff_eq_0)

lemma secant_number_aux_last [simp]: "secant_number_aux n n = fact n"
  by (induction n) (auto simp: secant_number_aux_rec)

lemma secant_number_aux_last': "m = n ⟹ secant_number_aux m n = fact
m"
  by (cases n) auto

lemma secant_number_aux_1_right [simp]:
  "secant_number_aux i (Suc 0) = secant_number_aux (i + 1) 0"
  by (simp add: secant_number_aux_def coeff_pderiv secant_poly.simps(2)
Let_def)
```

5.2 The secant numbers

The actual secant numbers S_n are now defined to be the even-index coefficients of the power series expansion of $\sec x$ (the odd-index ones are all 0).[3, A000364]

This also turns out to be exactly the same as $S_{n,0}$.

```
definition secant_number :: "nat ⇒ nat" where
  "secant_number n = nat (floor (fps_nth (fps_sec 1) (2*n) * fact (2*n)
:: real))"

lemma secant_number_conv_zigzag_number:
  "secant_number n = zigzag_number (2 * n)"
  unfolding secant_number_def
  by (subst zigzag_number_conv_fps_sec [symmetric]) auto

lemma zigzag_number_conv_sectan [code]:
  "zigzag_number n = (if even n then secant_number (n div 2) else tangent_number
((n+1) div 2))"
  by (auto elim!: evenE simp: secant_number_conv_zigzag_number tangent_number_conv_zigzag_n

lemma secant_number_0 [simp]: "secant_number 0 = 1"
```

```
  by (simp add: secant_number_def fps_sec_def)

lemma fps_nth_sec_aux:
  "fps_sec (1::'a::field_char_0) $ (2*n) =
     of_nat (secant_number_aux (2*n) 0) / fact (2*n)"
proof (cases "n = 0")
  case False
  interpret of_nat_poly_hom: map_poly_comm_semiring_hom of_nat
    by standard auto
  from False have n: "n > 0"
    by simp
  have "fps_nth ((fps_deriv ^^ (2 * n)) (fps_sec (1::'a))) 0 =
          fact (2*n) * fps_nth (fps_sec 1) (2*n)"
    by (simp add: fps_0th_higher_deriv)
  also have "(fps_deriv ^^ (2*n)) (fps_sec (1::'a)) =
               fps_sec 1 * (fps_of_poly (map_poly of_nat (secant_poly
(2*n))) oo fps_tan 1)"
    by (subst fps_sec_higher_deriv') auto
  also have "fps_nth ... 0 = of_nat (secant_number_aux (2*n) 0)"
    by (simp add: secant_number_aux_def)
  finally show ?thesis
    by simp
qed auto

lemma fps_nth_sec:
  "fps_nth (fps_sec (1::'a :: field_char_0)) (2*n) = of_int (secant_number
n) / fact (2*n)"
  using fps_nth_sec_aux[of n, where ?'a = real] fps_nth_sec_aux[of n,
where ?'a = 'a]
  by (simp add: secant_number_def)

lemma secant_number_conv_aux [code]:
  "secant_number n = secant_number_aux (2*n) 0"
  using fps_nth_sec[of n, where ?'a = real] fps_nth_sec_aux[of n, where
?'a = real] by simp

lemma secant_number_1 [simp]: "secant_number 1 = 1"
  by (simp add: secant_number_conv_aux secant_number_aux_def numeral_2_eq_2

                secant_poly.simps(2) Let_def pderiv_pCons)
```

By noting that $\tan'(x) = \sec(x)^2$ and comparing coefficients, one obtains the following identity that expresses the tangent numbers as a sum of secant numbers:

```
theorem tangent_number_conv_secant_number:
  assumes n: "n > 0"
  shows   "tangent_number n =
             (∑k<n. ((2*n-2) choose (2*k)) * secant_number k * secant_number
(n - k - 1))"
```

```
proof -
  have [simp]: "Suc (2 * n - 2) = 2 * n - 1"
    using n by linarith
  define m where "m = 2 * n - 2"
  have "even m"
    using n by (auto simp: m_def)

  have "fps_deriv (fps_tan (1::real)) = fps_sec 1 ^ 2"
    by (simp add: fps_tan_deriv fps_sec_def fps_inverse_power fps_divide_unit)
  hence "fps_nth (fps_deriv (fps_tan (1::real))) (2*n-2) = fps_nth (fps_sec
1 ^ 2) m"
    unfolding fps_eq_iff m_def by blast
  hence "fact m * fps_nth (fps_deriv (fps_tan (1::real))) (2*n-2) =
           fact m * fps_nth (fps_sec 1 ^ 2) m"
    by (rule arg_cong)
  also have "fps_nth (fps_deriv (fps_tan (1::real))) (2*n-2) =
               real (tangent_number n) * ((2 * real n - 1) / fact (2 *
n - 1))"
    using n by (auto simp: fps_nth_tan of_nat_diff Suc_diff_Suc)
  also have "(2 * real n - 1) / fact (2 * n - 1) = 1 / fact m"
    using n by (cases n) (simp_all add: m_def)
  also have "fps_nth (fps_sec 1 ^ 2) m = (∑k≤m. fps_sec 1 $ k * fps_sec
1 $ (m - k))"
    by (simp add: fps_square_nth)
  also have "... = (∑k | k ≤ m ∧ even k. fps_sec 1 $ k * fps_sec 1 $
(m - k))"
    by (rule sum.mono_neutral_right) (use ‹even m› in ‹auto simp: fps_nth_sec_odd›)
  also have "... = (∑k<n. fps_sec 1 $ (2*k) * fps_sec 1 $ (m - 2 * k))"
    by (rule sum.reindex_bij_witness[of _ "λk. 2 * k" "λk. k div 2"])

       (use n in ‹auto simp: m_def elim!: evenE›)
  also have "fact m * ... =
               (∑k<n. real (((2 * n - 2) choose (2 * k)) * secant_number
k * secant_number (n - k - 1)))"
    unfolding sum_distrib_left
  proof (intro sum.cong, goal_cases)
    case (2 k)
    have "fps_nth (fps_sec 1) (2 * (n - Suc k)) = secant_number (n - Suc
k) / fact (2 * (n - Suc k))"
      by (subst fps_nth_sec) auto
    moreover have "2 * (n - Suc k) = m - 2 * k"
      using ‹n > 0› by (auto simp: m_def)
    ultimately have "fps_nth (fps_sec 1) (m - 2 * k) = secant_number (n
- Suc k) / fact (2 * (n - Suc k))"
      by simp
    moreover have "fps_nth (fps_sec 1) (2 * k) = secant_number k / fact
(2 * k)"
      by (subst fps_nth_sec) auto
    ultimately show ?case
```

```
    using 2 by (simp add: m_def diff_mult_distrib2 binomial_fact field_simps)
  qed auto
  also have "fact m * (real (tangent_number n) * (1 / fact m)) = real
(tangent_number n)"
    by simp
  finally show ?thesis
    unfolding of_nat_sum [symmetric] by linarith
qed
```

5.3 Efficient functional computation

We again formalise a functional algorithm similar to what we have done for tangent numbers. This algorithm is again based on the one given by Brent et al. [1] and is completely analogous to the one for tangent numbers.

```
context
  fixes S :: "nat ⇒ nat ⇒ nat"
  defines "S ≡ secant_number_aux"
begin

primrec secant_number_impl_aux1 :: "nat ⇒ nat ⇒ nat list ⇒ nat list"
where
  "secant_number_impl_aux1 j y [] = []"
| "secant_number_impl_aux1 j y (x # xs) =
     (let x' = j * y + (j+1) * x in x' # secant_number_impl_aux1 (j+1)
x' xs)"

lemma length_secant_number_impl_aux1 [simp]: "length (secant_number_impl_aux1
j y xs) = length xs"
  by (induction xs arbitrary: j y) (simp_all add: Let_def)

fun secant_number_impl_aux2 :: "nat list ⇒ nat list" where
  "secant_number_impl_aux2 [] = []"
| "secant_number_impl_aux2 (x # xs) = x # secant_number_impl_aux2 (secant_number_impl_aux1
0 x xs)"

lemma secant_number_impl_aux1_nth_eq:
  assumes "i < length xs"
  shows   "secant_number_impl_aux1 j y xs ! i =
             (j+i) * (if i = 0 then y else secant_number_impl_aux1 j y
xs ! (i-1)) + (j+i+1) * xs ! i"
  using assms
proof (induction xs arbitrary: i j y)
  case (Cons x xs)
  show ?case
  proof (cases i)
    case 0
    thus ?thesis
      by (simp add: Let_def)
  next
```

```
    case (Suc i')
    define x' where "x' = (j) * y + (j+1) * x"
    have "secant_number_impl_aux1 j y (x # xs) ! i = secant_number_impl_aux1
(Suc j) x' xs ! i'"
      by (simp add: x'_def Let_def Suc)
    also have "... = (Suc j + i') * (if i' = 0 then x' else secant_number_impl_aux1
(Suc j) x' xs ! (i'-1)) +
                    (Suc j + i' + 1) * xs ! i'"
      using Cons.prems by (subst Cons.IH) (auto simp: Suc)
    also have "Suc j + i' = j + i"
      by (simp add: Suc)
    also have "xs ! i' = (x # xs) ! i"
      by (auto simp: Suc)
    also have "(if i' = 0 then x' else secant_number_impl_aux1 (Suc j)
x' xs ! (i'-1)) =
               (x' # secant_number_impl_aux1 j y (x # xs)) ! i"
      by (auto simp: Suc x'_def Let_def)
    finally show ?thesis
      by (simp add: Suc)
  qed
qed auto

lemma secant_number_impl_aux2_correct:
  assumes "k ≤ n"
  shows  "secant_number_impl_aux2 (map (λi. S (2 * k + i) i) [0..<n-k])
=
             map secant_number [k..<n]"
  using assms
proof (induction k rule: inc_induct)
  case (step k)
  have *: "[0..<n-k] = 0 # map Suc [0..<n-Suc k]"
    by (subst upt_conv_Cons)
       (use step.hyps in <auto simp: map_Suc_upt Suc_diff_Suc simp del:
upt_Suc>)
  define ts where
    "ts = secant_number_impl_aux1 0 (S (2*k) 0) (map (λi. S (2*k+i+1)
(i+1)) [0..<n-Suc k])"
  have S_rec: "S (Suc a) (Suc b) = (b + 1) * S a b + (b + 2) * S a (b
+ 2)" for a b
    unfolding S_def secant_number_aux_rec ..

  have "secant_number_impl_aux2 (map (λi. S (2 * k + i) i) [0..<n-k])
=
          S (2 * k) 0 # secant_number_impl_aux2 ts"
    unfolding * list.map secant_number_impl_aux2.simps
    by (simp add: o_def ts_def algebra_simps numeral_3_eq_3)
  also have "ts = map (λi. S (2 * Suc k + i) i) [0..<n - Suc k]"
  proof (rule nth_equalityI)
    fix i assume "i < length ts"
```

```
    hence i: "i < n - Suc k"
      by (simp add: ts_def)
    hence "ts ! i = S (2 * Suc k + i) i"
    proof (induction i)
      case 0
      thus ?case unfolding ts_def
        by (subst secant_number_impl_aux1_nth_eq) (simp_all add: S_def)
    next
      case (Suc i)
      have "ts ! Suc i = (i + 1) * S (2 * Suc k + i) i +
                (i + 2) * S (2 * Suc k + i) (Suc i + 1)"
        using Suc unfolding ts_def
        by (subst secant_number_impl_aux1_nth_eq) (simp_all add: eval_nat_numeral
algebra_simps)
      also have "... = S (Suc (2 * Suc k + i)) (Suc i)"
        by (subst S_rec) simp_all
      finally show ?case by simp
    qed
    thus "ts ! i = map (λi. S (2 * Suc k + i) i) [0..<n - Suc k] ! i"
      using i by simp
  qed (simp_all add: ts_def)
  also have "secant_number_impl_aux2 ... = map secant_number [Suc k..<n]"
    by (rule step.IH)
  also have "S (2 * k) 0 = secant_number k"
    by (simp add: secant_number_conv_aux S_def)
  also have "secant_number k # map secant_number [Suc k..<n] =
             map secant_number [k..<n]"
    using step.hyps by (subst upt_conv_Cons) (auto simp del: upt_Suc)
  finally show ?case .
qed auto

definition secant_numbers :: "nat ⇒ nat list" where
  "secant_numbers n = map secant_number [0..<Suc n]"

lemma secant_numbers_code [code]:
  "secant_numbers n = secant_number_impl_aux2 (pochhammer_row_impl 1 (n+2)
1)"
proof -
  have "pochhammer_row_impl 1 (n+2) 1 = map (λi. S i i) [0..<Suc n]"
    by (simp add: pochhammer_row_impl_correct pochhammer_fact S_def del:
upt_Suc)
  also have "secant_number_impl_aux2 ... = map secant_number [0..<Suc
n]"
    using secant_number_impl_aux2_correct[of 0 "Suc n"] by (simp del:
upt_Suc)
  finally show ?thesis
    by (simp only: secant_numbers_def One_nat_def)
qed
```

lemma *secant_number_code [code]: "secant_number n = last (secant_numbers n)"*
  by *(simp add: secant_numbers_def)*

end

definition *zigzag_numbers :: "nat ⇒ nat list"* **where**
  "zigzag_numbers n = map zigzag_number [0..<Suc n]"

```
lemma nth_splice:
  "i < length xs + length ys ⟹
     splice xs ys ! i =
       (if length xs ≤ length ys then
          if i < 2 * length xs then if even i then xs ! (i div 2) else
ys ! (i div 2) else ys ! (i - length xs)
        else if i < 2 * length ys then if even i then xs ! (i div 2) else
ys ! (i div 2) else xs ! (i - length ys))"
proof (induction xs ys arbitrary: i rule: splice.induct)
  case (2 x xs ys)
  show ?case
  proof (cases i)
    case i: (Suc i')
    have "splice (x # xs) ys ! i = splice ys xs ! i'"
      by (simp add: i)
    also have "... = (if length ys ≤ length xs
                    then if i' < 2 * length ys
                    then if even i' then ys ! (i' div 2) else xs ! (i'
div 2) else xs ! (i' - length ys)
                    else if i' < 2 * length xs
                    then if even i' then ys ! (i' div 2) else xs ! (i'
div 2) else ys ! (i' - length xs))"
      by (rule "2.IH") (use "2.prems" i in auto)
    also have "... = (if length (x # xs) ≤ length ys then if i < 2 * length
(x # xs)
                    then if even i then (x # xs) ! (i div 2) else ys
! (i div 2)
                    else ys ! (i - length (x # xs)) else if i < 2 * length
ys
                    then if even i then (x # xs) ! (i div 2) else ys
! (i div 2)
                    else (x # xs) ! (i - length ys))"
      using "2.prems" by (force simp: i not_less intro!: arg_cong2[of
_ _ _ _ nth] elim!: oddE evenE)
    finally show ?thesis .
  qed auto
qed auto
```

lemma *zigzag_numbers_code [code]:*

```
  "zigzag_numbers n = splice (secant_numbers (n div 2)) (tangent_numbers
((n+1) div 2))"
proof (rule nth_equalityI)
  fix i assume "i < length (zigzag_numbers n)"
  hence i: "i ≤ n"
    by (simp add: zigzag_numbers_def)
  define xs where "xs = secant_numbers (n div 2)"
  define ys where "ys = tangent_numbers ((n+1) div 2)"
  have [simp]: "length xs = n div 2 + 1" "length ys = (n+1) div 2"
    by (simp_all add: xs_def ys_def secant_numbers_def tangent_numbers_def)
  have "splice xs ys ! i = (if even i then xs ! (i div 2) else ys ! (i
div 2))"
  proof (subst nth_splice, goal_cases)
    case 2
    show ?case
      by (cases "even n")
         (use i in <auto elim!: evenE oddE simp: not_less double_not_eq_Suc_double

                         intro!: arg_cong2[of _ _ _ _ nth]>)
  qed (use i in auto)
  also have "... = zigzag_numbers n ! i"
    using i by (auto simp: zigzag_numbers_def secant_numbers_def tangent_numbers_def
                           zigzag_number_conv_sectan xs_def ys_def
                      elim!: evenE oddE simp del: upt_Suc)
  finally show "zigzag_numbers n ! i = splice xs ys ! i" ..
qed (auto simp: secant_numbers_def tangent_numbers_def zigzag_numbers_def)

end
```

5.4 Imperative in-place computation

```
theory Secant_Numbers_Imperative
  imports Secant_Numbers "Refine_Monadic.Refine_Monadic" "Refine_Imperative_HOL.IICF"
"HOL-Library.Code_Target_Numeral"
begin
```

We will now formalise and verify the imperative in-place version of the algorithm given by Brent et al. [1]. We use as storage only an array of n numbers, which will also contain the results in the end. Note however that the size of these numbers grows enormously the longer the algorithm runs.

```
locale secant_numbers_imperative
begin

context
  fixes n :: nat
begin

definition I_init :: "nat list × nat ⇒ bool" where
  "I_init = (λ(xs, i).
```

```
    (i ∈ {1..n+1} ∧ xs = map fact [0..<i] @ replicate (n+1-i) 0))"
```

definition *init_loop_aux :: "nat list nres"* **where**

```
  "init_loop_aux =
    do {xs ← RETURN (op_array_replicate (n+1) 0);
        ASSERT (length xs > 0);
        RETURN (xs[0 := 1])}"
```

definition *init_loop :: "nat list nres"* **where**

```
  "init_loop =
    do {
      xs ← init_loop_aux;
      (xs', _) ←
        WHILE_T^I_init
          (λ(_, i). i ≤ n)
          (λ(xs, i). do {
            ASSERT (i - 1 < length xs);
            x ← RETURN (xs ! (i - 1));
            ASSERT (i < length xs);
            RETURN (xs[i := i * x], i + 1)
          })
          (xs, 1);
      RETURN xs'
    }"
```

definition *I_inner* **where**

```
  "I_inner xs i = (λ(xs', j). j ∈ {i+1..n+1} ∧ length xs' = n+1 ∧
    (∀k≤n. xs' ! k = (if k∈{i..<j} then secant_number_aux (k+Suc i-1)
(k+1-Suc i) else xs ! k)))"
```

definition *inner_loop :: "nat list ⇒ nat ⇒ nat list nres"* **where**

```
  "inner_loop xs i =
    do {
      (xs', _) ←
        WHILE_T^(I_inner xs i) (λ(_, j). j ≤ n)
        (λ(xs, j). do {
          ASSERT (j - 1 < length xs);
          x ← RETURN (xs ! (j - 1));
          ASSERT (j < length xs);
          y ← RETURN (xs ! j);
          RETURN (xs[j := (j - i) * x + (j - i + 1) * y], j + 1)
        })
        (xs, i + 1);
      RETURN xs'
    }"
```

definition *I_compute :: "nat list × nat ⇒ bool"* **where**

```
  "I_compute = (λ(xs, i).
    (i ∈ {1..n+1} ∧ xs = map (λk. if k < i then secant_number k else
```

```
secant_number_aux (k+i-1) (k+1-i)) [0..<Suc n]))"

definition compute :: "nat list nres" where
  "compute =
     do {
       xs ← init_loop;
       (xs', _) ←
         WHILE_T^I_compute
           (λ(_, i). i ≤ n)
           (λ(xs, i). do { xs' ← inner_loop xs i; RETURN (xs', i + 1)
})
           (xs, 1);
       RETURN xs'
     }"

lemma init_loop_aux_correct [refine_vcg]:
  "init_loop_aux ≤ SPEC (λxs. xs = (replicate (n+1) 0)[0 := 1])"
  unfolding init_loop_aux_def
  by refine_vcg auto

lemma init_loop_correct [refine_vcg]: "init_loop ≤ SPEC (λxs. xs = map
fact [0..<n+1])"
  unfolding init_loop_def
  apply refine_vcg
  apply (rule wf_measure[of "λ(_, i). n + 1 - i"])
  subgoal
    by (auto simp: I_init_def nth_list_update' intro!: nth_equalityI)
  subgoal
    by (auto simp: I_init_def)
  subgoal
    by (auto simp: I_init_def)
  subgoal
    by (auto simp: I_init_def nth_list_update' fact_reduce nth_Cons nth_append
             intro!: nth_equalityI split: nat.splits)
  subgoal
    by auto
  subgoal
    by (auto simp: I_init_def le_Suc_eq simp del: upt_Suc)
  done

lemma I_inner_preserve:
  assumes invar: "I_inner xs i (xs', j)" and invar': "I_compute (xs,
i)"
  assumes j: "j ≤ n"
  defines "y ≡ (j - i) * xs' ! (j - 1) + (j - i + 1) * xs' ! j"
  defines "xs'' ≡ list_update xs' j y"
  shows   "I_inner xs i (xs'', j + 1)"
  unfolding I_inner_def
proof safe
```

```
  show "j + 1 ∈ {i+1..n+1}" "length xs'' = n + 1"
    using invar j by (simp_all add: xs''_def I_inner_def)
next
  fix k assume k: "k ≤ n"
  define S where "S = secant_number_aux"
  have ij: "1 ≤ i" "i < j" "j ≤ n"
    using invar invar' j by (auto simp: I_inner_def I_compute_def)
  have nth_xs': "xs' ! k = (if k ∈ {i..<j} then S (k + Suc i-1) (k +
1 - Suc i) else xs ! k)"
    if "k ≤ n" for k using invar that unfolding I_inner_def S_def by
blast
  have nth_xs: "xs ! k = (if k < i then secant_number k else S (k + i
- 1) (k + 1 - i))"
    if "k ≤ n" for k using invar' that unfolding I_compute_def S_def by
(auto simp del: upt_Suc)
  have [simp]: "length xs' = n + 1"
    using invar by (simp add: I_inner_def)

  consider "k = j" | "k ∈ {i..<j}" | "k ∉ {i..j}"
    by force
  thus "xs'' ! k = (if k ∈ {i..<j + 1} then S (k + Suc i - 1) (k + 1 -
Suc i) else xs ! k)"
  proof cases
    assume [simp]: "k = j"
    have "xs'' ! k = y"
      using ij by (simp add: xs''_def)
    also have "... = (j - i) * xs' ! (j - 1) + (j - i + 1) * xs' ! j"
      by (simp add: y_def)
    also have "xs' ! j = xs ! j"
      using ij by (subst nth_xs') auto
    also have "... = S (j + i - 1) (j + 1 - i)"
      using ij by (subst nth_xs) auto
    also have "xs' ! (j - 1) = S (j + i - 1) (j - Suc i)"
      using ij by (subst nth_xs') (auto simp: Suc_diff_Suc)
    also have "(j - i) * S (j + i - 1) (j - Suc i) + (j - i + 1) * S (j
+ i - 1) (j + 1 - i) =
               S (j + i) (j - i)"
      unfolding S_def by (subst (3) secant_number_aux_rec') (use ij in
auto)
    finally show ?thesis
      using ij by simp
  next
    assume k: "k ∈ {i..<j}"
    hence "xs'' ! k = xs' ! k"
      unfolding xs''_def by auto
    also have "... = S (k + i) (k - i)"
      by (subst nth_xs') (use k ij in auto)
    finally show ?thesis
      using k by simp
```

```
  next
    assume k: "k ∉ {i..j}"
    hence "xs'' ! k = xs' ! k"
      using ij unfolding xs''_def by auto
    also have "xs' ! k = xs ! k"
      using k <k ≤ n> by (subst nth_xs') auto
    finally show ?thesis
      using k by auto
  qed
qed

lemma inner_loop_correct [refine_vcg]:
  assumes "I_compute (xs, i)" "i ≤ n"
  shows "inner_loop xs i ≤ SPEC (λxs'. xs' =
          map (λk. if k ≥ i then secant_number_aux (k+Suc i-1) (k+1-Suc
i) else xs ! k) [0..<Suc n])"
  unfolding inner_loop_def
  apply refine_vcg
  apply (rule wf_measure[of "λ(_, j). n + 1 - j"])
  subgoal
    unfolding I_inner_def
    by clarify (use assms in <simp_all add: mult_2 I_compute_def del:
upt_Suc>)
  subgoal
    using assms unfolding I_inner_def by auto
  subgoal
    using assms unfolding I_inner_def by auto
  subgoal for s xs' j
    using I_inner_preserve[of xs i xs' j] assms by auto
  subgoal
    by auto
  subgoal using assms
    by (auto simp: I_inner_def intro!: nth_equalityI simp del: upt_Suc)
  done

lemma compute_correct [refine_vcg]: "compute ≤ SPEC (λxs'. xs' = secant_numbers
n)"
  unfolding compute_def
  apply refine_vcg
     apply (rule wf_measure[of "λ(_, i). n + 1 - i"])
  subgoal
    by (auto simp: I_compute_def secant_number_aux_last' simp del: upt_Suc)
  subgoal
    by (auto simp: I_compute_def secant_number_conv_aux less_Suc_eq mult_2)
  subgoal
    by (auto simp: I_compute_def simp del: upt_Suc)
  subgoal
    by (auto simp: I_compute_def secant_number_conv_aux less_Suc_eq mult_2
simp del: upt_Suc
```

```
            intro!: nth_equalityI)
  subgoal
    by auto
  subgoal
    by (auto simp: I_compute_def secant_numbers_def intro!: nth_equalityI
simp del: upt_Suc)
  done

lemmas defs =
  compute_def inner_loop_def init_loop_def init_loop_aux_def

end

sepref_definition compute_imp is
  "secant_numbers_imperative.compute" ::
    "nat_assn^d →_a array_assn nat_assn"
  unfolding secant_numbers_imperative.defs by sepref

lemma imp_correct':
  "(compute_imp, λn. RETURN (secant_numbers n)) ∈ nat_assn^d →_a array_assn
nat_assn"
proof -
  have *: "(compute, λn. RETURN (secant_numbers n)) ∈ nat_rel → ⟨Id⟩nres_rel"
    by refine_vcg simp?
  show ?thesis
    using compute_imp.refine[FCOMP *] .
qed

theorem imp_correct:
   "<nat_assn n n> compute_imp n <array_assn nat_assn (secant_numbers
n)>_t"
proof -
  have [simp]: "nofail (compute n)"
    using compute_correct[of n] le_RES_nofailI by blast
  have 1: "xs = secant_numbers n" if "RETURN xs ≤ compute n" for xs
    using that compute_correct[of n] by (simp add: pw_le_iff)
  note rl = compute_imp.refine[THEN hfrefD, of n n, THEN hn_refineD, simplified]
  show ?thesis
    apply (rule cons_rule[OF _ _ rl])
    apply (sep_auto simp: pure_def)
    apply (sep_auto simp: pure_def dest!: 1)
    done
qed

end

lemmas [code] = secant_numbers_imperative.compute_imp_def

end
```

6 Euler numbers

```
theory Euler_Numbers
  imports Tangent_Numbers Secant_Numbers
begin
```

Euler numbers and Euler polynomials are very similar to Bernoulli numbers and Bernoulli polynomials. They are closely related to the secant numbers – and thereby also to the zigzag numbers (which are, confusingly, also sometimes referred to as "Euler numbers"). [3, A122045]

Our definition of Euler numbers follows the convention in Mathematica (where they are called `EulerE[n]`) and ProofWiki: Let S_n denote the secant numbers. Then:

$$\mathcal{E}_{2n} = (-1)^n S_n \qquad \mathcal{E}_{2n+1} = 0$$

such that in particular:

$$\sum_{n=0}^{\infty} \mathcal{E}_n n! x^n = \operatorname{sech} x = \frac{1}{\cosh x}$$

That is, the exponential generating function of the $\mathcal{E}_n$ is the hyperbolic secant.

```
definition euler_number :: "nat ⇒ int" where
  "euler_number n = (if odd n then 0 else (-1) ^ (n div 2) * secant_number
(n div 2))"

lemma euler_number_odd: "euler_number (2 * n) = (-1) ^ n * secant_number
n"
  by (auto simp: euler_number_def)

lemma secant_number_conv_euler_number: "secant_number n = (-1) ^ n *
euler_number (2 * n)"
  by (auto simp: euler_number_def)

lemma euler_number_odd_eq_0: "odd n ⟹ euler_number n = 0"
  by (simp add: euler_number_def)

lemma euler_number_odd_numeral [simp]: "euler_number (numeral (Num.Bit1
n)) = 0"
  by (subst euler_number_odd_eq_0) auto

lemma euler_number_Suc_0 [simp]: "euler_number (Suc 0) = 0"
  by (subst euler_number_odd_eq_0) auto

lemma euler_number_0 [simp]: "euler_number 0 = 1"
  and euler_number_2 [simp]: "euler_number 2 = -1"
  by (simp_all add: euler_number_def secant_number_conv_aux secant_number_aux_def
```

```
                    secant_poly.simps(2) numeral_2_eq_2 Let_def pderiv_pCons)

lemma fps_nth_sech_conv_of_rat_fps_nth_sech:
  "fps_nth (fps_sech (1 :: 'a :: field_char_0)) n = of_rat (fps_nth (fps_sech
(1 :: rat)) n)"
proof (induction n rule: less_induct)
  case (less n)
  show ?case
  proof (cases "n = 0")
    case False
    hence "fps_nth (fps_sech (1 :: 'a :: field_char_0)) n =
            -(∑i = 0..<n. fps_sech 1 $ i * fps_cosh 1 $ (n - i))"
      by (simp add: fps_sech_def fps_nth_inverse)
    also have "(∑i = 0..<n. fps_sech (1::'a) $ i * fps_cosh 1 $ (n -
i)) =
               (∑i = 0..<n. of_rat (fps_sech 1 $ i) * fps_cosh 1 $ (n
- i))"
      by (intro sum.cong arg_cong2[of _ _ _ _ "(*)"] less.IH refl) auto
    also have "-... = of_rat (-(∑i = 0..<n. fps_sech 1 $ i * fps_cosh
1 $ (n - i)))"
      by (simp add: fps_cosh_def of_rat_sum of_rat_mult of_rat_divide

                    of_rat_add of_rat_power of_rat_minus)
    also have "-(∑i = 0..<n. fps_sech 1 $ i * fps_cosh 1 $ (n - i)) =
fps_nth (fps_sech (1::rat)) n"
      using False by (simp add: fps_sech_def fps_nth_inverse)
    finally show ?thesis .
  qed auto
qed

lemma exponential_generating_function_euler_numbers:
  "Abs_fps (λn. of_int (euler_number n) / fact n :: 'a :: field_char_0)
= fps_sech 1"
proof (rule fps_ext)
  fix n :: nat
  have "fps_sech 1 = fps_sec 1 oo (fps_const i * fps_X)"
    by (simp add: fps_sech_conv_sec)
  also have "fps_nth ... n = i ^ n * fps_nth (fps_sec 1) n"
    by (subst fps_nth_compose_linear) auto
  also have "fps_nth (fps_sec (1::complex)) n =
               (if even n then of_nat (secant_number (n div 2)) / fact
n else 0)"
    by (auto elim!: evenE simp: fps_nth_sec fps_nth_sec_odd)
  also have "i ^ n * ... = (euler_number n / fact n)"
    by (auto simp: euler_number_def)
  finally have *: "fps_nth (fps_sech (1 :: complex)) n = euler_number n
/ fact n"
    by simp
```

```
  have "of_rat (of_int (euler_number n) / fact n) = of_int (euler_number
n) / fact n"
    by (simp add: of_rat_divide)
  also have "... = fps_nth (fps_sech (1::complex)) n"
    by (simp add: *)
  also have "... = of_rat (fps_sech 1 $ n)"
    by (subst fps_nth_sech_conv_of_rat_fps_nth_sech) auto
  finally have "fps_sech (1::rat) $ n = of_int (euler_number n) / fact
n"
    unfolding of_rat_eq_iff ..

  have "fps_nth (fps_sech (1::'a)) n = of_rat (fps_sech 1 $ n)"
    by (subst fps_nth_sech_conv_of_rat_fps_nth_sech) auto
  also have "fps_sech (1::rat) $ n = of_int (euler_number n) / fact n"
    by fact
  also have "of_rat ... = of_int (euler_number n) / fact n"
    by (simp add: of_rat_divide)
  finally show "fps_nth (Abs_fps (λn. of_int (euler_number n) / fact n
:: 'a :: field_char_0)) n =
                  fps_nth (fps_sech 1) n"
    by simp
qed
```

From the above, it easily follows that the sum over the Euler numbers $\mathcal{E}_0$ to $\mathcal{E}_n$ weighted by binomial coefficients vanishes.

```
theorem sum_binomial_euler_number_eq_0:
  assumes n: "n > 0" "even n"
  shows   "(∑k≤n. int (n choose k) * euler_number k) = 0"
proof -
  have "Abs_fps (λn. euler_number n / fact n) * fps_cosh 1 = 1"
    unfolding exponential_generating_function_euler_numbers fps_sech_def
    by (rule inverse_mult_eq_1) auto
  hence "fps_nth (Abs_fps (λn. euler_number n / fact n) * fps_cosh 1)
n = fps_nth 1 n"
    by (rule arg_cong)
  hence "0 = fact n * (∑i=0..n. real_of_int (euler_number i) *
                        (if even n = even i then 1 / fact (n - i) else
0) / fact i)"
    using n by (simp add: fps_eq_iff fps_mult_nth fps_nth_cosh cong: if_cong)
  also have "... = (∑i=0..n. real_of_int (euler_number i) *
                        (if even n = even i then 1 / fact (n - i) else
0) / fact i * fact n)"
    by (simp add: sum_distrib_left sum_distrib_right mult_ac)
  also have "... = (∑i=0..n. real (n choose i) * euler_number i)"
    using n by (intro sum.cong) (auto simp: euler_number_odd_eq_0 binomial_fact
mult_ac)
  also have "... = real_of_int (∑i≤n. int (n choose i) * euler_number
i)"
    by (simp add: atLeast0AtMost)
```

```
  finally show ?thesis
    by linarith
qed
```

This in particular gives us the following full-history recurrence for $\mathcal{E}_n$ that is reminiscent of the Bernoulli numbers:

```
corollary euler_number_rec:
  assumes n: "n > 0" "even n"
  shows   "euler_number n = -(∑k<n. int (n choose k) * euler_number
k)"
proof -
  have "(∑k≤n. int (n choose k) * euler_number k) = 0"
    by (rule sum_binomial_euler_number_eq_0) fact+
  also have "{..n} = insert n {..<n}"
    by auto
  also have "(∑k∈... int (n choose k) * euler_number k) =
               euler_number n + (∑k<n. int (n choose k) * euler_number
k)"
    by (subst sum.insert) (use n in auto)
  finally show ?thesis
    by linarith
qed

lemma euler_number_rec':
  "euler_number n =
     (if n = 0 then 1 else if odd n then 0 else -(∑k<n. int (n choose
k) * euler_number k))"
  using euler_number_rec[of n] by (auto simp: euler_number_odd_eq_0)

lemma tangent_number_conv_euler_number:
  assumes n: "n > 0"
  defines "E ≡ euler_number"
  shows   "int (tangent_number n) =
             (-1) ^ Suc n * (∑k≤2*n-2. int ((2*n-2) choose k) * E k
* E (2*n-k-2))"
proof -
  have "int (tangent_number n) =
          (∑k<n. int (((2 * n - 2) choose (2*k)) * secant_number k *
secant_number (n - k - 1)))"
    using n by (subst tangent_number_conv_secant_number) auto
  also have "... = (∑k<n. ((2 * n - 2) choose (2*k)) * (-1)^(n - 1) *
E (2*k) * E (2*(n-k-1)))"
    by (rule sum.cong) (simp_all add: E_def euler_number_def flip: power_add)
  also have "... = (-1)^(n-1) * (∑k<n. ((2 * n - 2) choose (2*k)) * E
(2*k) * E (2*(n - k - 1)))"
    by (simp add: sum_distrib_left sum_distrib_right mult_ac)
  also have "(-1)^(n-1) = ((-1)^Suc n :: int)"
    using n by (cases n) auto
  also have "(∑k<n. ((2 * n - 2) choose (2*k)) * E (2*k) * E (2*(n -
```

```
k - 1))) =
            (∑k | k ≤ 2 * n - 2 ∧ even k. ((2 * n - 2) choose k) *
E k * E (2 * n - 2 - k))"
    by (rule sum.reindex_bij_witness[of _ "λk. k div 2" "λk. 2 * k"])
       (use n in <auto simp: diff_mult_distrib2>)
  also have "... = (∑k≤2*n-2. ((2 * n - 2) choose k) * E k * E (2 *
n - 2 - k))"
    by (rule sum.mono_neutral_left) (auto simp: E_def euler_number_odd_eq_0)
  finally show ?thesis
    by simp
qed
```

7 Euler polynomials

7.1 Definition and basic properties

Similarly to Bernoulli polynomials, one can also define Euler polynomials based on Euler numbers:

```
definition euler_poly :: "nat ⇒ 'a :: field_char_0 ⇒ 'a" where
  "euler_poly n x = (∑k≤n. of_int ((n choose k) * euler_number k) /
2 ^ k * (x - 1/2) ^ (n - k))"

definition Euler_poly :: "nat ⇒ 'a :: field_char_0 poly" where
  "Euler_poly n =
     (∑k≤n. Polynomial.smult (of_int (int (n choose k) * euler_number
k) / 2 ^ k)
               ((Polynomial.monom 1 1 - [:1/2:]) ^ (n - k)))"

lemma lead_coeff_Euler_poly [simp]: "poly.coeff (Euler_poly n) n = 1"
proof -
  define P :: "nat ⇒ 'a poly" where "P = (λk. (Polynomial.monom 1 1
- [:1 / 2:]) ^ (n - k))"
  have "poly.coeff (Euler_poly n :: 'a poly) n =
          (∑k≤n. of_nat (n choose k) * of_int (euler_number k) * poly.coeff
(P k) n / 2 ^ k)"
    unfolding Euler_poly_def by (simp add: coeff_sum P_def)
  also have "... = (∑k∈{0}. of_nat (n choose k) * of_int (euler_number
k) * poly.coeff (P k) n / 2 ^ k)"
  proof (intro sum.mono_neutral_right ballI, goal_cases)
    case (3 k)
    have "degree (P k) = n - k"
      unfolding P_def by (simp add: monom_altdef degree_power_eq)
    with 3 have "poly.coeff (P k) n = 0"
      by (intro coeff_eq_0) auto
    thus ?case
      by simp
  qed auto
  also have "... = lead_coeff ([:- (1 / 2), 1:] ^ n)"
```

```
    by (simp add: P_def monom_altdef degree_power_eq)
  also have "... = 1"
    by (subst lead_coeff_power) auto
  finally show "poly.coeff (Euler_poly n :: 'a poly) n = 1" .
qed

lemma degree_Euler_poly [simp]: "degree (Euler_poly n) = n"
proof (rule antisym)
  show "degree (Euler_poly n) ≤ n"
    unfolding Euler_poly_def
    by (intro degree_sum_le) (auto simp: degree_power_eq monom_altdef)
  show "degree (Euler_poly n) ≥ n"
    by (rule le_degree) simp
qed

lemma poly_Euler_poly [simp]: "poly (Euler_poly n) = euler_poly n"
  by (rule ext) (simp add: Euler_poly_def poly_sum euler_poly_def poly_monom)

lemma euler_poly_onehalf:
  "euler_poly n (1 / 2) = (of_int (euler_number n) / 2 ^ n :: 'a :: field_char_0)"
proof -
  have "euler_poly n (1 / 2) =
          (∑k≤n. of_nat (n choose k) * of_int (euler_number k) * (0::'a)
^ (n - k) / 2 ^ k)"
    by (simp add: euler_poly_def)
  also have "... = (∑k∈{n}. of_int (euler_number n) / 2 ^ k)"
    by (rule sum.mono_neutral_cong_right) auto
  also have "... = of_int (euler_number n) / 2 ^ n"
    by simp
  finally show ?thesis .
qed

lemma Euler_poly_0 [simp]: "Euler_poly 0 = 1"
  and Euler_poly_1: "Euler_poly 1 = [:-(1 / 2), 1:]"
  and Euler_poly_2: "Euler_poly 2 = [:0, - 1, 1:]"
  using euler_number_2
  by (simp_all add: Euler_poly_def monom_altdef numeral_2_eq_2 del: euler_number_2)
```

Like Bernoulli polynomials, the Euler polynomials are an Appell sequence, i.e. they satisfy $\mathcal{E}_n'(x) = n\mathcal{E}_{n-1}(x)$:

```
lemma pderiv_Euler_poly: "pderiv (Euler_poly n) = of_nat n * Euler_poly
(n - 1)"
proof (cases "n = 0")
  case False
  define m where "m = n - 1"
  have n: "n = Suc m"
    using False by (auto simp: m_def)
  define E where "E = euler_number"
  define X where "X = Polynomial.monom (1::'a) 1"
```

```
  write Polynomial.smult (infixl "*p" 69)
  have "pderiv (Euler_poly n) =
          (∑i≤n. Polynomial.smult (of_nat (Suc m choose i) *
              of_int (E i * (n-i)) / 2^i) ((X - [:1/2:]) ^ (n - Suc i)))"
    using False
    by (simp add: Euler_poly_def pderiv_sum pderiv_smult pderiv_diff pderiv_power
pderiv_monom
                  X_def E_def m_def mult_ac)
  also have "... = (∑i≤m. Polynomial.smult (of_nat (Suc m choose i)
*
                     of_int (E i * (n-i)) / 2^i) ((X - [:1/2:]) ^ (n -
Suc i)))"
    by (rule sum.mono_neutral_right) (use False in <auto simp: m_def>)
  also have "... = (∑i≤m. of_nat n * (of_nat (m choose i) *
                     of_int (E i) / 2 ^ i *p (X - [:1 / 2:]) ^ (m - i)))"
    by (intro sum.cong refl, subst of_nat_binomial_Suc) (use False in
<auto simp: m_def>)
  also have "... = Polynomial.smult (of_nat n) (Euler_poly (n - 1))"
    by (simp add: Euler_poly_def smult_sum2 m_def E_def X_def mult_ac
of_nat_poly)
  finally show ?thesis
    by (simp add: of_nat_poly)
qed auto


lemma continuous_on_euler_poly [continuous_intros]:
  fixes f :: "'a :: topological_space ⇒ 'b :: {real_normed_field, field_char_0}"
  assumes "continuous_on A f"
  shows   "continuous_on A (λx. euler_poly n (f x))"
  unfolding poly_Euler_poly [symmetric] by (intro continuous_on_poly assms)

lemma continuous_euler_poly [continuous_intros]:
  fixes f :: "'a :: t2_space ⇒ 'b :: {real_normed_field, field_char_0}"
  assumes "continuous F f"
  shows   "continuous F (λx. euler_poly n (f x))"
  unfolding poly_Euler_poly [symmetric] by (rule continuous_poly [OF assms])

lemma tendsto_euler_poly [tendsto_intros]:
  fixes f :: "'a :: t2_space ⇒ 'b :: {real_normed_field, field_char_0}"
  assumes "(f ⟶ c) F"
  shows   "((λx. euler_poly n (f x)) ⟶ euler_poly n c) F"
  unfolding poly_Euler_poly [symmetric] by (rule tendsto_intros assms)+

lemma has_field_derivative_euler_poly [derivative_intros]:
  assumes "(f has_field_derivative f') (at x within A)"
  shows   "((λx. euler_poly n (f x)) has_field_derivative
             (of_nat n * f' * euler_poly (n - 1) (f x))) (at x within
A)"
  unfolding poly_Euler_poly [symmetric]
```

```
  by (rule derivative_eq_intros assms)+ (simp_all add: pderiv_Euler_poly)
```

The exponential generating function of the Euler polynomials is:

$$\sum_{n=0}^{\infty} \frac{\mathcal{E}_n(x)}{n!} t^n = \operatorname{sech}(t/2) e^{(x-\frac{1}{2})t} = \frac{2e^{xt}}{e^t + 1}$$

```
theorem exponential_generating_function_euler_poly:
  "Abs_fps (λn. euler_poly n x / fact n :: 'a :: field_char_0) =
     fps_sech (1 / 2) * fps_exp (x - 1 / 2)"
  "Abs_fps (λn. euler_poly n x / fact n) =
     2 * fps_exp x / (fps_exp 1 + 1)"
proof -
  define E where "E = (λc. fps_to_fls (fps_exp (c :: 'a)))"
  have [simp]: "E c ≠ 0" for c
    by (auto simp: E_def)
  have "Abs_fps (λn. euler_poly n x / fact n :: 'a) =
          Abs_fps (λn. (1/2)^n * of_int (euler_number n) / fact n) *
          Abs_fps (λn. (x - 1 / 2) ^ n / fact n)"
    by (simp add: euler_poly_def fps_eq_iff sum_divide_distrib binomial_fact
fps_mult_nth
                  field_simps atLeast0AtMost)
  also have "Abs_fps (λn. (1/2)^n * of_int (euler_number n) / fact n ::
'a) =
             Abs_fps (λn. of_int (euler_number n) / fact n) oo (fps_const
(1/2) * fps_X)"
    unfolding fps_compose_linear by simp
  also have "... = fps_sech (1 / 2)"
    unfolding exponential_generating_function_euler_numbers by simp
  also have "Abs_fps (λn. (x - 1 / 2) ^ n / fact n) = fps_exp (x - 1 /
2)"
    by (simp add: fps_exp_def)
  finally show "Abs_fps (λn. euler_poly n x / fact n :: 'a :: field_char_0)
=
                  fps_sech (1 / 2) * fps_exp (x - 1 / 2)" .

  also {
    have "fps_to_fls (fps_sech (1 / 2) * fps_exp (x - 1 / 2)) =
            2 * E x / (E (1/2) * (E (1/2) + 1 / E (1/2)))"
      using fps_exp_add_mult[of x "-1/2"]
      by (simp add: fps_sech_def fps_cosh_def fls_times_fps_to_fls fls_inverse_const
            fps_exp_neg E_def divide_simps flip: fls_inverse_fps_to_fls
fls_const_divide_const)
    also have "E (1/2) * (E (1/2) + 1 / E (1/2)) = E (1/2) ^ 2 + 1"
      by (simp add: algebra_simps power2_eq_square)
    also have "E (1 / 2) ^ 2 = E 1"
      by (simp add: E_def fps_exp_power_mult flip: fps_to_fls_power)
    also have "2 * E x / (E 1 + 1) = fps_to_fls (2 * fps_exp x / (fps_exp
1 + 1))"
```

```
      by (simp add: E_def fls_times_fps_to_fls flip: fls_divide_fps_to_fls)
    finally have "fps_sech (1 / 2) * fps_exp (x - 1 / 2) =
                  2 * fps_exp x / (fps_exp 1 + 1)"
      by (simp only: fps_to_fls_eq_iff)
  }
  finally show "Abs_fps (λn. euler_poly n x / fact n) =
                  2 * fps_exp x / (fps_exp 1 + 1)" .
qed
```

We also show the corresponding fact for Bernoulli theorems, namely

$$\sum_{n\geq 0} \frac{\mathcal{B}_n(x)}{n!} t^n = \frac{te^{tx}}{e^t - 1}$$

```
theorem exponential_generating_function_bernpoly:
  fixes x :: "'a :: {field_char_0, real_normed_field}"
  shows "Abs_fps (λn. bernpoly n x / fact n) = fps_X * fps_exp x / (fps_exp
1 - 1)"
proof -
  define E where "E = (λc. fps_to_fls (fps_exp (c :: 'a)))"
  have [simp]: "E c ≠ 0" for c
    by (auto simp: E_def)
  have [simp]: "subdegree (1 - fps_exp (1 :: 'a)) = 1"
    by (rule subdegreeI) auto
  have "Abs_fps (λn. bernpoly n x / fact n :: 'a) = bernoulli_fps * fps_exp
x"
    unfolding fps_times_def
    by (simp add: bernpoly_def fps_eq_iff sum_divide_distrib binomial_fact
                  field_simps atLeast0AtMost)
  also have "... = fps_X * fps_exp x / (fps_exp 1 - 1)"
    unfolding bernoulli_fps_def by (subst fps_divide_times2) auto
  finally show ?thesis .
qed


definition Bernpoly :: "nat ⇒ 'a :: {real_algebra_1, field_char_0} poly"
where
  "Bernpoly n = poly_of_list (map (λk. of_nat (n choose k) * of_real (bernoulli
(n - k))) [0..<Suc n])"

lemma coeff_Bernpoly:
  "poly.coeff (Bernpoly n) k = of_nat (n choose k) * of_real (bernoulli
(n - k))"
  by (simp add: Bernpoly_def nth_default_def del: upt_Suc)

lemma degree_Bernpoly [simp]: "degree (Bernpoly n) = n"
proof (rule antisym)
  show "degree (Bernpoly n) ≤ n"
```

```
    by (rule degree_le) (auto simp: coeff_Bernpoly)
  show "degree (Bernpoly n) ≥ n"
    by (rule le_degree) (auto simp: coeff_Bernpoly)
qed

lemma lead_coeff_Bernpoly [simp]: "poly.coeff (Bernpoly n) n = 1"
  by (subst coeff_Bernpoly) auto

lemma poly_Bernpoly [simp]: "poly (Bernpoly n) x = bernpoly n x"
proof -
  have "poly (Bernpoly n) x = (∑i≤n. of_nat (n choose i) * of_real (bernoulli
(n - i)) * x ^ i)"
    by (simp add: poly_altdef coeff_Bernpoly)
  also have "... = bernpoly n x"
    unfolding bernpoly_def
    by (rule sum.reindex_bij_witness[of _ "λi. n - i" "λi. n - i"])
       (auto simp flip: binomial_symmetric)
  finally show ?thesis .
qed
```

The following two recurrences allow computing Bernoulli and Euler polynomials recursively.

```
theorem bernpoly_recurrence:
  fixes x :: "'a :: {field_char_0, real_normed_field}"
  assumes n: "n > 0"
  shows "(∑s<n. of_nat (n choose s) * bernpoly s x) = of_nat n * x ^
(n - 1)"
proof -
  define F where "F = Abs_fps (λn. bernpoly n x / fact n)"
  have F_eq: "F = fps_X * fps_exp x / (fps_exp 1 - 1)"
    unfolding F_def exponential_generating_function_bernpoly ..

  have "(∑s<n. of_nat (n choose s) * bernpoly s x / fact n) =
          fps_nth (F * (fps_exp 1 - 1)) n"
    unfolding F_def fps_mult_nth by (rule sum.mono_neutral_cong_left)
(auto simp: binomial_fact)
  also have "F * (fps_exp 1 - 1) = fps_X * fps_exp x"
    unfolding F_eq by (metis bernoulli_fps_aux dvd_mult2 dvd_mult_div_cancel
dvd_triv_right mult.commute)
  also have "fps_nth ... n = x ^ (n - 1) / fact (n - 1)"
    using n by simp
  finally have "(∑s<n. of_nat (n choose s) * bernpoly s x) = x ^ (n -
1) * (fact n / fact (n - 1))"
    by (simp add: field_simps flip: sum_divide_distrib)
  also have "fact n / fact (n - 1) = (of_nat n :: 'a)"
    using ‹n > 0› by (subst fact_binomial [symmetric]) auto
  finally show "(∑s<n. of_nat (n choose s) * bernpoly s x) = of_nat n
* x ^ (n - 1)"
    by (simp add: mult.commute)
```

```
qed

corollary bernpoly_recurrence':
  fixes x :: "'a :: {field_char_0, real_normed_field}"
  shows "bernpoly n x = x ^ n - (∑s<n. of_nat (Suc n choose s) * bernpoly
s x) / of_nat (Suc n)"
proof -
  have "(∑s<Suc n. of_nat (Suc n choose s) * bernpoly s x) = of_nat
(Suc n) * x ^ n"
    by (subst bernpoly_recurrence) auto
  also have "(∑s<Suc n. of_nat (Suc n choose s) * bernpoly s x) =
              of_nat (Suc n) * bernpoly n x + (∑s<n. of_nat (Suc n choose
s) * bernpoly s x)"
    by simp
  finally have "of_nat (Suc n) * bernpoly n x =
                  of_nat (Suc n) * x ^ n - (∑s<n. of_nat (Suc n choose
s) * bernpoly s x)"
    by (simp add: algebra_simps)
  thus "bernpoly n x = x ^ n - (∑s<n. of_nat (Suc n choose s) * bernpoly
s x) / of_nat (Suc n)"
    by (simp add: field_simps del: of_nat_Suc)
qed

theorem Bernpoly_recurrence:
  assumes "n > 0"
  shows   "(∑s<n. Polynomial.smult (of_nat (n choose s)) (Bernpoly s))
=
             Polynomial.monom (of_nat n :: 'a :: {field_char_0, real_normed_field})
(n - 1)"
    (is "?lhs = ?rhs")
proof -
  have "poly ?lhs x = poly ?rhs x" for x
    using bernpoly_recurrence[of n x] assms by (simp add: poly_sum poly_monom)
  thus "?lhs = ?rhs"
    by blast
qed

theorem Bernpoly_recurrence':
  shows   "Bernpoly n = Polynomial.monom (1 :: 'a :: {field_char_0, real_normed_field})
n -
             Polynomial.smult (1 / of_nat (Suc n))
               (∑s<n. Polynomial.smult (of_nat (Suc n choose s)) (Bernpoly
s))"
    (is "?lhs = ?rhs")
proof -
  have "poly ?lhs x = poly ?rhs x" for x
    using bernpoly_recurrence'[of n x] by (simp add: poly_sum poly_monom)
  thus "?lhs = ?rhs"
    by blast
```

```
qed


theorem euler_poly_recurrence:
  fixes x :: "'a :: {field_char_0}"
  shows "euler_poly n x = x ^ n - (∑s<n. of_nat (n choose s) * euler_poly
s x) / 2"
proof -
  define F where "F = Abs_fps (λn. euler_poly n x / fact n)"
  have F_eq: "F = 2 * fps_exp x / (fps_exp 1 + 1)"
    unfolding F_def exponential_generating_function_euler_poly(2) ..

  have "2 * euler_poly n x / fact n +
          (∑s<n. (if s = n then 2 else 1) * of_nat (n choose s) * euler_poly
s x / fact n) =
         (∑s∈insert n {..<n}. (if s = n then 2 else 1) * of_nat (n
choose s) * euler_poly s x / fact n)"
    by (subst sum.insert) auto
  also have "insert n {..<n} = {..n}"
    by auto
  also have "(∑s<n. (if s = n then 2 else 1) * of_nat (n choose s) *
euler_poly s x / fact n) =
           (∑s<n. of_nat (n choose s) * euler_poly s x / fact n)"
    by (rule sum.cong) auto
  also have "(∑s≤n. (if s = n then 2 else 1) * of_nat (n choose s) *
euler_poly s x / fact n) =
         fps_nth (F * (fps_exp 1 + 1)) n"
    unfolding F_def fps_mult_nth by (rule sum.mono_neutral_cong_left)
(auto simp: binomial_fact)
  also have "F * (fps_exp 1 + 1) = 2 * fps_exp x"
    unfolding F_eq  by (subst fps_divide_unit) auto
  also have "fps_nth ... n = 2 * x ^ n / fact n"
    by simp
  finally show "euler_poly n x = x ^ n - (∑s<n. of_nat (n choose s) *
euler_poly s x) / 2"
    by (simp add: field_simps flip: sum_divide_distrib)
qed

theorem Euler_poly_recurrence:
  "Euler_poly n = (Polynomial.monom 1 n :: 'a :: field_char_0 poly) -

     Polynomial.smult (1/2) (∑s<n. Polynomial.smult (of_nat (n choose
s)) (Euler_poly s))"
    (is "_ = ?rhs")
proof -
  have "poly (Euler_poly n) x = poly ?rhs x" for x
  proof -
    have "poly (Euler_poly n) x = euler_poly n x"
```

```
      by simp
    also have "... = poly ?rhs x"
      by (subst euler_poly_recurrence) (simp_all add: poly_monom poly_sum)
    finally show "poly (Euler_poly n) x = poly ?rhs x" .
  qed
  thus "Euler_poly n = ?rhs"
    by blast
qed

lemma euler_poly_1_even:
  assumes "even n" "n > 1"
  shows   "euler_poly n 1 = 0"
proof -
  have "euler_poly n 1 = of_int (∑k≤n. int (n choose k) * (euler_number
k)) / 2 ^ n"
    by (simp add: euler_poly_def power_diff field_simps flip: sum_divide_distrib)
  also have "(∑k≤n. int (n choose k) * (euler_number k)) = 0"
    by (rule sum_binomial_euler_number_eq_0) (use assms in auto)
  finally show ?thesis
    by simp
qed
```

7.2 Addition and reflection theorems

The Euler polynomials satisfy the following addition theorem:

$$\mathcal{E}_n(x+y) = \sum_{k=0}^{n} \binom{n}{k} \mathcal{E}_k(x) y^{n-k}$$

```
theorem euler_poly_addition:
  "euler_poly n (x + y) = (∑k≤n. of_nat (n choose k) * euler_poly k
x * y ^ (n - k))"
proof -
  define E where "E = (λk. of_int (euler_number k) :: 'a)"
  have "euler_poly n (x + y) =
          (∑k≤n. of_nat (n choose k) * E k * (x + y - 1 / 2) ^ (n -
k) / 2 ^ k)"
    by (simp add: euler_poly_def E_def)
  also have "... = (∑k≤n. of_nat (n choose k) * E k *
                   (∑i≤n-k. of_nat (n - k choose i) * (x - 1/2) ^ i
* y ^ (n - k - i)) / 2 ^ k)"
  proof (rule sum.cong, goal_cases)
    case (2 k)
    have "((x - 1 / 2) + y) ^ (n - k) =
            (∑i≤n-k. of_nat (n - k choose i) * (x - 1/2) ^ i * y ^ (n
- k - i))"
      by (subst binomial_ring) auto
    thus ?case
      by (simp add: algebra_simps)
```

```
  qed auto
  also have "... = (∑(k,i)∈(SIGMA k:{..n}. {..n-k}).
                    of_nat (n choose k) * E k * of_nat (n - k choose i)
*
                    (x - 1/2) ^ i * y ^ (n - k - i) / 2 ^ k)"
    by (simp add: sum_distrib_left sum_distrib_right sum_divide_distrib
mult_ac sum.Sigma)
  also have "... = (∑(k,i)∈(SIGMA k:{..n}. {..k}).
                    of_nat (n choose k) * E i * of_nat (k choose i) *
                    (x - 1/2) ^ (k - i) * y ^ (n - k) / 2 ^ i)"
    by (rule sum.reindex_bij_witness[of _ "λ(k,i). (i, k - i)" "λ(k,
i). (i + k, k)"])
       (auto simp: binomial_fact algebra_simps)
  also have "... = (∑k≤n. of_nat (n choose k) * euler_poly k x * y ^
(n - k))"
    by (simp add: euler_poly_def E_def sum_distrib_left sum_distrib_right

                  sum_divide_distrib mult_ac sum.Sigma)
  finally show ?thesis .
qed
```

The Bernoulli polynomials actually satisfy an analogous theorem.

```
theorem bernpoly_addition:
  fixes x y :: "'a :: {field_char_0, real_normed_field}"
  shows "bernpoly n (x + y) = (∑k≤n. of_nat (n choose k) * bernpoly
k x * y ^ (n - k))"
proof -
  define B where "B = (λk. of_real (bernoulli k) :: 'a)"
  have "bernpoly n (x + y) =
          (∑k≤n. of_nat (n choose k) * B k * (x + y) ^ (n - k))"
    by (simp add: bernpoly_def B_def)
  also have "... = (∑k≤n. of_nat (n choose k) * B k *
                    (∑i≤n-k. of_nat (n - k choose i) * x ^ i * y ^ (n
- k - i)))"
  proof (rule sum.cong, goal_cases)
    case (2 k)
    have "(x + y) ^ (n - k) =
            (∑i≤n-k. of_nat (n - k choose i) * x ^ i * y ^ (n - k -
i))"
      by (subst binomial_ring) auto
    thus ?case
      by (simp add: algebra_simps)
  qed auto
  also have "... = (∑(k,i)∈(SIGMA k:{..n}. {..n-k}).
                    of_nat (n choose k) * B k * of_nat (n - k choose i)
*
                    x ^ i * y ^ (n - k - i))"
    by (simp add: sum_distrib_left sum_distrib_right sum_divide_distrib
mult_ac sum.Sigma)
```

```
  also have "... = (∑(k,i)∈(SIGMA k:{..n}. {..k}).
                   of_nat (n choose k) * B i * of_nat (k choose i) *
                   x ^ (k - i) * y ^ (n - k))"
    by (rule sum.reindex_bij_witness[of _ "λ(k,i). (i, k - i)" "λ(k,
i). (i + k, k)"])
       (auto simp: binomial_fact algebra_simps)
  also have "... = (∑k≤n. of_nat (n choose k) * bernpoly k x * y ^ (n
- k))"
    by (simp add: bernpoly_def B_def sum_distrib_left sum_distrib_right

                  sum_divide_distrib mult_ac sum.Sigma)
  finally show ?thesis .
qed

theorem euler_poly_reflect:
  "euler_poly n (1 - x) = (-1) ^ n * euler_poly n x"
proof -
  have "(-1) ^ n * euler_poly n x =
          (∑k≤n. of_nat (n choose k) * of_int (euler_number k) *
                   ((-1) ^ n * ((x - 1 / 2)) ^ (n - k)) / 2 ^ k)"
    unfolding sum_distrib_left euler_poly_def
    by (intro sum.cong) (simp_all add: mult_ac)
  also have "... = (∑k≤n. of_nat (n choose k) * of_int (euler_number
k) *
                   ((-1) ^ (n - k) * (x - 1 / 2) ^ (n - k)) / 2 ^ k)"
    by (intro sum.cong) (auto simp: uminus_power_if euler_number_odd_eq_0)
  also have "... = (∑k≤n. of_nat (n choose k) * of_int (euler_number
k) *
                   (1 / 2 - x) ^ (n - k) / 2 ^ k)"
    unfolding power_mult_distrib [symmetric] by simp
  also have "... = euler_poly n (1 - x)"
    by (simp add: euler_poly_def)
  finally show ?thesis ..
qed

theorem euler_poly_forward_sum: "euler_poly n x + euler_poly n (x + 1)
= 2 * x ^ n"
proof -
  have "Abs_fps (λn. euler_poly n x / fact n) + Abs_fps (λn. euler_poly
n (x + 1) / fact n) =
          2 * fps_exp x / (fps_exp 1 + 1) + fps_exp 1 * (2 * fps_exp x)
/ (fps_exp 1 + 1)"
    unfolding exponential_generating_function_euler_poly(2) fps_exp_add_mult
    by (simp add: mult_ac)
  also have "fps_exp 1 * (2 * fps_exp x) / (fps_exp 1 + 1) =
               fps_exp 1 * (2 * fps_exp x / (fps_exp 1 + 1))"
    by (subst fps_divide_times) auto
  also have "2 * fps_exp x / (fps_exp 1 + 1) + fps_exp 1 * (2 * fps_exp
x / (fps_exp 1 + 1)) =
```

```
            (fps_exp 1 + 1) * (2 * fps_exp x / (fps_exp 1 + 1))"
    by Groebner_Basis.algebra
  also have "... = 2 * fps_exp x"
    by simp
  also have "fps_nth ... n = 2 * x ^ n / fact n"
    by simp
  finally show ?thesis
    by (simp add: field_simps)
qed

lemma euler_poly_plus1: "euler_poly n (x + 1) = -euler_poly n x + 2 *
x ^ n"
  using euler_poly_forward_sum[of n x] by (simp add: algebra_simps)

lemma euler_poly_minus:
  "(-1) ^ n * euler_poly n (-x) = -euler_poly n x + 2 * x ^ n"
  using euler_poly_reflect[of n "-x"] euler_poly_plus1[of n "x"]
  by (simp add: algebra_simps)
```

As an analogon of Faulhaber's formula for sums of the form $x^k+(x+1)^k+\ldots$, we can express an alternating sum of the form $x^k-(x+1)^k+(x+2)^k+\ldots$ in terms of the k-th Euler polynomial.

```
corollary alternating_power_sum_conv_euler_poly:
  "(∑i<k. (-1) ^ i * (x + of_nat i) ^ n) =
     (euler_poly n x - (-1) ^ k * euler_poly n (x + of_nat k)) / 2"
proof -
  define E :: "'a ⇒ 'a" where "E = euler_poly n"
  have "(∑i<k. (-1) ^ i * (x + of_nat i) ^ n) = (E x - (-1) ^ k * E
(x + of_nat k)) / 2"
  proof (induction k)
    case (Suc k)
    have "(∑i<Suc k. (-1) ^ i * (x + of_nat i) ^ n) =
            (∑i<k. (-1) ^ i * (x + of_nat i) ^ n) + (-1) ^ k * (x + of_nat
k) ^ n"
      by simp
    also have "(∑i<k. (-1) ^ i * (x + of_nat i) ^ n) = (E x - (-1) ^
k * E (x + of_nat k)) / 2"
      by (rule Suc.IH)
    also have "(x + of_nat k) ^ n = (E (x + of_nat k) + E (x + of_nat
(Suc k))) / 2"
      using euler_poly_forward_sum[of n "x + of_nat k"] by (simp add:
E_def add_ac)
    finally show ?case
      by (simp add: diff_divide_distrib add_divide_distrib ring_distribs)
  qed auto
  thus ?thesis
    by (simp add: E_def)
qed
```

7.3 Multiplication theorems

For any positive integer m, the Bernoulli polynomials satisfy:

$$\mathcal{B}_n(mx) = m^{n-1} \sum_{k=0}^{m-1} \mathcal{B}_n(x + k/m)$$

```
theorem bernpoly_mult:
  fixes x :: "'a :: {real_normed_field, field_char_0}"
  assumes m: "m > 0"
  shows   "bernpoly n (of_nat m * x) =
             of_nat m powi (int n - 1) * (∑k<m. bernpoly n (x + of_nat
k / of_nat m))"
proof -
  define F where "F = (λc (x::'a). Abs_fps (λn. bernpoly n (of_nat c
* x) / fact n))"
  have F_eq: "F c x = fps_X * fps_exp (of_nat c * x) / (fps_exp 1 - 1)"
for c x
    by (simp add: F_def exponential_generating_function_bernpoly fps_exp_power_mult)
  define E where "E = (λc::'a. fps_to_fls (fps_exp c))"
  have E_add: "E (c + c') = E c * E c'" for c c'
    by (simp add: E_def fps_exp_add_mult fls_times_fps_to_fls)
  have E_power: "E c ^ m = E (of_nat m * c)" for c m
    by (simp add: E_def fps_exp_power_mult flip: fps_to_fls_power)
  have minus_one_power_fps: "(-1)^k = fps_const ((-1::'a) ^ k)" for k
    by (simp flip: fps_const_power fps_const_neg)
  have fls_neqI: "F ≠ G" if "fls_nth F 0 ≠ fls_nth G 0" for F G :: "'a
fls"
    using that by metis
  have fls_neqI': "F ≠ G" if "fls_nth F 1 ≠ fls_nth G 1" for F G :: "'a
fls"
    using that by metis
  have fps_neqI: "F ≠ G" if "fps_nth F 0 ≠ fps_nth G 0" for F G :: "'a
fps"
    using that by metis
  have [simp]: "fls_nth (E c) n = c ^ (nat n) / fact (nat n)" if "n ≥
0" for c n
    using that by (auto simp: E_def)
  have [simp]: "subdegree (1 - fps_exp 1 :: 'a fps) = 1"
    by (rule subdegreeI) auto

  have "fps_to_fls (of_nat m * F m x -fps_compose (∑k<m. F 1 (x + of_nat
k / of_nat m)) (of_nat m * fps_X)) =
          of_nat m * (fls_X * E (of_nat m * x)) / (E 1 - 1) -
         (∑k<m. of_nat m * (fls_X * E (of_nat m * x + of_nat k)) / (E
(of_nat m) - 1))"
    unfolding F_eq using m
    by (simp add: fls_times_fps_to_fls flip: fps_of_nat fls_compose_fps_to_fls)
```

```
        (simp add: fls_times_fps_to_fls fps_to_fls_sum fps_to_fls_power
fps_shift_to_fls E_def
                  mult.assoc fls_compose_fps_divide fls_compose_fps_diff
fls_compose_fps_mult
                  fls_compose_fps_power ring_distribs
             flip: fps_of_nat fls_divide_fps_to_fls fls_of_nat)
  also have "(∑k<m. of_nat m * (fls_X * E (of_nat m * x + of_nat k))
/ (E (of_nat m) - 1)) =
             of_nat m * fls_X * E x ^ m * (∑i<m. E 1 ^ i) / (E (of_nat
m) - 1)"
    by (simp add: sum_divide_distrib sum_distrib_left sum_distrib_right

                  algebra_simps E_power E_add power_minus')
  also have "(∑i<m. E 1 ^ i) =  (1 - E 1 ^ m) / (1 - E 1)"
    by (subst sum_gp_strict) (use <m > 0> in <auto simp: fls_neqI'>)
  also have "E (of_nat m) = E 1 ^ m"
    by (simp add: E_power)
  also have "of_nat m * fls_X * E x ^ m * ((1 - E 1 ^ m) / (1 - E 1))
/ (E 1 ^ m - 1) =
             -of_nat m * fls_X * E x ^ m / (1 - E 1)"
    using m by (simp add: divide_simps fls_neqI fls_neqI' E_power) (auto
simp: algebra_simps)
  also have "... = of_nat m * fls_X * E x ^ m / (E 1 - 1)"
    by (simp add: field_simps fls_neqI')
  also have "of_nat m * (fls_X * E (of_nat m * x)) / (E 1 - 1) -
             of_nat m * fls_X * E x ^ m / (E 1 - 1) = 0"
    by (simp add: E_power)
  also have "fls_nth ... n = 0"
    by simp
  finally have "of_nat m * bernpoly n (of_nat m * x) =
                of_nat m ^ n * (∑k<m. bernpoly n (x + of_nat k / of_nat
m))"
    by (simp add: F_def minus_one_power_fps fps_sum_nth fps_nth_compose_linear
nat_add_distrib
                  mult.assoc flip: fps_of_nat sum_divide_distrib)
  also have "of_nat m ^ n = (of_nat m * of_nat m powi (int n - 1) :: 'a)"
    using <m > 0> by (subst power_int_diff) auto
  finally show ?thesis
    using <m > 0> by simp
qed
```

The corresponding theorem for the Euler polynomials is more complicated. For odd positive integers m, we have following still very simple theorem:

$$\mathcal{E}_n(mx) = m^n \sum_{k=0}^{m-1} (-1)^k \mathcal{E}_n(x + k/m)$$

```
theorem euler_poly_mult_odd:
  assumes "odd m"
```

```
  shows   "euler_poly n (of_nat m * x) =
             of_nat m ^ n * (∑k<m. (-1) ^ k * euler_poly n (x + of_nat
k / of_nat m))"
proof -
  define F where "F = (λc (x::'a). Abs_fps (λn. euler_poly n (of_nat
c * x) / fact n))"
  have F_eq: "F c x = 2 * fps_exp x ^ c / (fps_exp 1 + 1)" for c x
    by (simp add: F_def exponential_generating_function_euler_poly(2)
fps_exp_power_mult)
  define E where "E = (λc::'a. fps_to_fls (fps_exp c))"
  have E_add: "E (c + c') = E c * E c'" for c c'
    by (simp add: E_def fps_exp_add_mult fls_times_fps_to_fls)
  have E_power: "E c ^ m = E (of_nat m * c)" for c m
    by (simp add: E_def fps_exp_power_mult flip: fps_to_fls_power)
  have minus_one_power_fps: "(-1)^k = fps_const ((-1::'a) ^ k)" for k
    by (simp flip: fps_const_power fps_const_neg)
  have fls_neqI: "F ≠ G" if "fls_nth F 0 ≠ fls_nth G 0" for F G :: "'a
fls"
    using that by metis
  have fps_neqI: "F ≠ G" if "fps_nth F 0 ≠ fps_nth G 0" for F G :: "'a
fps"
    using that by metis
  have [simp]: "fls_nth (E c) n = c ^ (nat n) / fact (nat n)" if "n ≥
0" for c n
    using that by (auto simp: E_def)

  have "F m x - fps_compose (∑k<m. (-1)^k * F 1 (x + of_nat k / of_nat
m)) (of_nat m * fps_X) =
          2 * fps_exp x ^ m / (fps_exp 1 + 1) -
           (∑k<m. (-1)^k * (2 * fps_exp (of_nat m * x + of_nat k) /
(fps_exp (of_nat m) + 1)))"
    unfolding exponential_generating_function_euler_poly(2)
    by (simp add: fps_exp_power_mult F_eq fps_compose_sum_distrib
                  fps_compose_mult_distrib fps_compose_divide_distrib
fps_compose_add_distrib
                  fps_compose_uminus fps_neqI ring_distribs flip: fps_compose_power
fps_of_nat)
  also have "fps_to_fls ... =
               2 * E x ^ m / (E 1 + 1) -
               (∑k<m. (-1)^k * (2 * E (of_nat m * x + of_nat k)) / (E
(of_nat m) + 1))"
    by (simp add: fls_times_fps_to_fls fps_to_fls_power E_def
             flip: fls_divide_fps_to_fls )
  also have "... = 2 * (E x ^ m / (E 1 + 1) - E x ^ m * (∑k<m. (-E 1)
^ k) / (E (of_nat m) + 1))"
    by (simp add: diff_divide_distrib sum_distrib_left sum_distrib_right
mult_ac E_add E_power
                  power_minus' flip: sum_divide_distrib)
  also have "(∑k<m. (-E 1) ^ k) = (1 - (-E 1) ^ m) / (1 + E 1)"
```

```
    by (subst sum_gp_strict) (auto simp: fls_neqI)
  also have "... = (1 + E 1 ^ m) / (1 + E 1)"
    using ‹odd m› by (auto simp: uminus_power_if)
  also have "E 1 ^ m = E (of_nat m)"
    using ‹odd m› by (auto simp: E_power)
  also have "2 * (E x ^ m / (E 1 + 1) - E x ^ m * ((1 + E (of_nat m))
/ (1 + E 1)) / (E (of_nat m) + 1)) = 0"
    by (simp add: divide_simps add_ac fls_neqI)
  also have "fls_nth ... n = 0"
    by simp
  finally show ?thesis
    by (simp add: F_def fps_sum_nth fps_compose_linear minus_one_power_fps
             flip: fps_of_nat sum_divide_distrib)
qed
```

For even positive m on the other hand, we have the following:

$$\mathcal{E}_n(mx) = -\frac{2m^n}{n+1}\sum_{k=0}^{m-1}(-1)^k\mathcal{B}_{n+1}(x+k/m)$$

```
theorem euler_poly_mult_even:
  fixes x :: "'a :: {real_normed_field, field_char_0}"
  assumes m: "even m" "m > 0"
  shows   "euler_poly n (of_nat m * x) =
             -2 * of_nat m ^ n / of_nat (Suc n) *
               (∑k<m. (-1) ^ k * bernpoly (Suc n) (x + of_nat k / of_nat
m))"
proof -
  define F where "F = (λc (x::'a). Abs_fps (λn. euler_poly n (of_nat
c * x) / fact n))"
  define G where "G = (λc (x::'a). Abs_fps (λn. bernpoly n (of_nat c
* x) / fact n))"
  have *: "(-1) ^ k = fps_const ((-1)^k :: 'a)" for k
    by auto
  have F_eq: "F c x = 2 * fps_exp x ^ c / (fps_exp 1 + 1)" for c x
    by (simp add: F_def exponential_generating_function_euler_poly(2)
fps_exp_power_mult)
  have G_eq: "G c x = fps_X * fps_exp (of_nat c * x) / (fps_exp 1 - 1)"
for c x
    by (simp add: G_def exponential_generating_function_bernpoly fps_exp_power_mult)
  define E where "E = (λc::'a. fps_to_fls (fps_exp c))"
  have E_add: "E (c + c') = E c * E c'" for c c'
    by (simp add: E_def fps_exp_add_mult fls_times_fps_to_fls)
  have E_power: "E c ^ m = E (of_nat m * c)" for c m
    by (simp add: E_def fps_exp_power_mult flip: fps_to_fls_power)
  have minus_one_power_fps: "(-1)^k = fps_const ((-1::'a) ^ k)" for k
    by (simp flip: fps_const_power fps_const_neg)
  have fls_neqI: "F ≠ G" if "fls_nth F 0 ≠ fls_nth G 0" for F G :: "'a
fls"
```

```
    using that by metis
  have fls_neqI': "F ≠ G" if "fls_nth F 1 ≠ fls_nth G 1" for F G :: "'a
fls"
    using that by metis
  have fps_neqI: "F ≠ G" if "fps_nth F 0 ≠ fps_nth G 0" for F G :: "'a
fps"
    using that by metis
  have [simp]: "fls_nth (E c) n = c ^ (nat n) / fact (nat n)" if "n ≥
0" for c n
    using that by (auto simp: E_def)
  have [simp]: "subdegree (1 - fps_exp 1 :: 'a fps) = 1"
    by (rule subdegreeI) auto

  have "fps_to_fls (fps_X * of_nat m * F m x + 2 * fps_compose (∑k<m.
(-1)^k * (G 1 (x + of_nat k / of_nat m))) (of_nat m * fps_X)) =
          fls_X * (of_nat m * (2 * E x ^ m / (E 1 + 1))) +
          2 * (∑i<m. (-1) ^ i * of_nat m * fls_X * E (of_nat m * x +
of_nat i) / (E (of_nat m) - 1))"
    unfolding F_eq G_eq using m
    by (simp add: fls_times_fps_to_fls flip: fps_of_nat fls_compose_fps_to_fls)
       (simp add: fls_times_fps_to_fls fps_to_fls_sum fps_to_fls_power
fps_shift_to_fls E_def
                 mult.assoc fls_compose_fps_divide fls_compose_fps_diff
fls_compose_fps_mult
                 fls_compose_fps_power ring_distribs
            flip: fps_of_nat fls_divide_fps_to_fls fls_of_nat)
  also have "(∑i<m. (-1) ^ i * of_nat m * fls_X * E (of_nat m * x + of_nat
i) / (E (of_nat m) - 1)) =
            of_nat m * fls_X * E x ^ m * (∑i<m. (-E 1) ^ i) / (E (of_nat
m) - 1)"
    by (simp add: sum_divide_distrib sum_distrib_left sum_distrib_right

                 algebra_simps E_power E_add power_minus')
  also have "(∑i<m. (-E 1) ^ i) = (1 - (-E 1) ^ m) / (1 + E 1)"
    by (subst sum_gp_strict) (auto simp: fls_neqI)
  also have "1 - (-E 1) ^ m = 1 - E 1 ^ m"
    using <even m> by auto
  also have "E (of_nat m) = E 1 ^ m"
    by (simp add: E_power)
  also have "of_nat m * fls_X * E x ^ m * ((1 - E 1 ^ m) / (1 + E 1))
/ (E 1 ^ m - 1) =
            -of_nat m * fls_X * E x ^ m / (1 + E 1)"
    using m by (simp add: divide_simps fls_neqI fls_neqI' E_power) (auto
simp: algebra_simps)
  also have "fls_X * (of_nat m * (2 * E x ^ m / (E 1 + 1))) +
              2 * (- of_nat m * fls_X * E x ^ m / (1 + E 1)) = 0"
    by (simp add: algebra_simps)
  also have "fls_nth ... (Suc n) = 0"
    by simp
```

```
  finally have "0 = (of_nat m * euler_poly n (of_nat m * x) / fact n) +
                2 * (of_nat m * (of_nat m ^ n *
                  (∑k<m. (-1) ^ k * bernpoly (Suc n) (x + of_nat k
/ of_nat m)))) /
                    ((1 + of_nat n) * fact n)"
    by (simp add: F_def G_def * fps_sum_nth fps_nth_compose_linear nat_add_distrib
                mult.assoc flip: fps_of_nat sum_divide_distrib)
  also have "... = of_nat m / fact n * (euler_poly n (of_nat m * x) +

                  2 * of_nat m ^ n / of_nat (Suc n) *
                  (∑k<m. (-1) ^ k * bernpoly (Suc n) (x + of_nat k
/ of_nat m)))"
    by (simp add: algebra_simps)
  finally show ?thesis
    using m by (simp add: add_eq_0_iff)
qed
```

The Euler polynomials can be written as the difference of two Bernoulli polynomials.

```
corollary euler_poly_conv_bernpoly:
  fixes x :: "'a :: {real_normed_field, field_char_0}"
  assumes m: "even m" "m > 0"
  shows "euler_poly n x =
           2 / of_nat (Suc n) * (bernpoly (Suc n) x - 2 ^ Suc n * bernpoly
(Suc n) (x / 2))"
proof -
  have "euler_poly n x = -(2^Suc n * (bernpoly (Suc n) (x / 2) -
          bernpoly (Suc n) (x / 2 + 1 / 2)) / of_nat (Suc n))"
    using euler_poly_mult_even[of 2 n "x/2"]
    by (simp add: numeral_2_eq_2)
  also have "... = 2 / of_nat (Suc n) * (2^n * bernpoly (Suc n) (x/2 +
1/2) - 2^n * bernpoly (Suc n) (x/2))"
    by (simp del: of_nat_Suc add: field_simps)
  also have "2^n * bernpoly (Suc n) (x/2 + 1/2) - 2^n * bernpoly (Suc
n) (x/2) =
             bernpoly (Suc n) x - 2 ^ Suc n * bernpoly (Suc n) (x / 2)"
    using bernpoly_mult[of 2 "Suc n" "x/2"]
    by (simp add: numeral_2_eq_2 ring_distribs)
  finally show ?thesis .
qed
```

7.4 Computing Bernoulli polynomials

```
definition binomial_row :: "nat ⇒ 'a :: semiring_1 list" where
  "binomial_row n = map (λk. of_nat (n choose k)) [0..<Suc n]"

lemma length_binomial_row [simp]: "length (binomial_row n) = Suc n"
  by (simp add: binomial_row_def del: upt_Suc)
```

```
lemma nth_binomial_row [simp]: "k ≤ n ⟹ binomial_row n ! k = of_nat
(n choose k)"
  by (simp add: binomial_row_def del: upt_Suc)

definition pascal_step :: "'a :: semiring_1 list ⇒ 'a list" where
  "pascal_step xs = map2 (+) (xs @ [0]) (0 # xs)"

lemma pascal_step_correct [simp]:
  "pascal_step (binomial_row n) = binomial_row (Suc n)"
  by (rule nth_equalityI)
     (auto simp: pascal_step_def binomial_row_def nth_Cons nth_append
add.commute
                  not_less less_Suc_eq binomial_eq_0
           simp del: upt_Suc split: nat.splits)


primrec Bernpolys_aux :: "nat list ⇒ 'a :: {field_char_0, real_normed_field}
poly list ⇒ nat ⇒ 'a poly list" where
  "Bernpolys_aux cs xs 0 = xs"
| "Bernpolys_aux cs xs (Suc k) =
     (let n = length xs;
          p = Polynomial.monom 1 n - Polynomial.smult (1 / of_nat (Suc
n))
                (∑(p,c)←zip xs (drop 2 cs). Polynomial.smult (of_nat
c) p)
      in Bernpolys_aux (pascal_step cs) (p # xs) k)"

lemma length_Bernpolys_aux [simp]: "length (Bernpolys_aux cs xs n) =
length xs + n"
  by (induction n arbitrary: xs cs) (simp_all add: Let_def)

lemma Bernpolys_aux_correct:
  "Bernpolys_aux (binomial_row (Suc n)) (map Bernpoly (rev [0..<n])) m
= map Bernpoly (rev [0..<m+n])"
proof (induction m arbitrary: n)
  case (Suc m n)
  define xs :: "'a poly list" where "xs = map Bernpoly (rev [0..<n])"
  define cs :: "nat list" where "cs = binomial_row (Suc n)"
  define S where "S = (∑(p,c)←zip xs (drop 2 cs). Polynomial.smult
(of_nat c) p)"
  define q where "q = Polynomial.monom 1 n - Polynomial.smult (1 / of_nat
(Suc n)) S"
  have [simp]: "length xs = n"
    by (simp add: xs_def)

  have "Bernpolys_aux cs (map Bernpoly (rev [0..<n]) :: 'a poly list)
(Suc m) =
          Bernpolys_aux (binomial_row (Suc (Suc n))) (q # xs) m"
    by (simp del: upt_Suc add: q_def S_def xs_def cs_def)
```

```
  also have "q # xs = map Bernpoly (rev [0..<Suc n])"
  proof -
    have "q = Polynomial.monom 1 n - Polynomial.smult (1 / of_nat (Suc
n)) S"
      by (simp add: q_def)
    also have "S = (∑s<n. Polynomial.smult (of_nat (Suc n choose (s+2)))
(xs ! s))"
      unfolding S_def
      by (subst sum.list_conv_set_nth) (simp_all add: atLeast0LessThan
cs_def del: upt_Suc)
    also have "... = (∑s<n. Polynomial.smult (of_nat (Suc n choose (s+2)))
(Bernpoly (n - Suc s)))"
      by (intro sum.cong) (auto simp: xs_def rev_nth)
    also have "... = (∑s<n. Polynomial.smult (of_nat (Suc n choose (Suc
n - s))) (Bernpoly s))"
      by (rule sum.reindex_bij_witness[of _ "λs. n - Suc s" "λs. n -
Suc s"])
         (auto simp del: binomial_Suc_Suc)
    also have "... = (∑s<n. Polynomial.smult (of_nat (Suc n choose s))
(Bernpoly s))"
      by (intro sum.cong refl, subst binomial_symmetric) (auto simp del:
binomial_Suc_Suc)
    also have "Polynomial.monom 1 n - Polynomial.smult (1 / of_nat (Suc
n)) ... = Bernpoly n"
      using Bernpoly_recurrence' [symmetric, of n] by simp
    finally show ?thesis
      by (simp add: xs_def)
  qed
  also have "Bernpolys_aux (binomial_row (Suc (Suc n))) ... m = map Bernpoly
(rev [0..<m + Suc n])"
    by (rule Suc.IH)
  finally show ?case
    by (simp del: upt_Suc add: cs_def)
qed auto
```

The following function recursively computes a list of the Bernoulli polynomials $B_0, \ldots, B_{n-1}$.

```
definition Bernpolys :: "nat ⇒ 'a :: {field_char_0, real_normed_field}
poly list"
  where "Bernpolys n = rev (Bernpolys_aux [1, 1] [] n)"

lemma length_Bernpolys [simp]: "length (Bernpolys n) = n"
  by (simp add: Bernpolys_def)

lemma Bernpolys_correct: "Bernpolys n = map Bernpoly [0..<n]"
  using Bernpolys_aux_correct[of 0 n, where ?'a = 'a]
  by (simp add: Bernpolys_def rev_swap binomial_row_def flip: rev_map)

lemma Bernpoly_code [code]: "Bernpoly n = hd (Bernpolys_aux [1, 1] []
```

```
(Suc n))"
  using Bernpolys_aux_correct[of 0 "Suc n", where ?'a = 'a]
  by (simp flip: rev_map add: hd_rev last_map binomial_row_def del: Bernpolys_aux.simps)


primrec bernpoly_aux :: "nat list ⇒ 'a :: {field_char_0, real_normed_field}
list ⇒ nat ⇒ 'a ⇒ 'a list" where
  "bernpoly_aux cs ys 0 x = ys"
| "bernpoly_aux cs ys (Suc k) x =
     (let n = length ys;
          y = x ^ n - (∑ (y,c)←zip ys (drop 2 cs). of_nat c * y) / of_nat
(Suc n)
      in bernpoly_aux (pascal_step cs) (y # ys) k x)"

lemma length_bernpoly_aux [simp]: "length (bernpoly_aux cs xs n x) =
length xs + n"
  by (induction n arbitrary: xs cs) (simp_all add: Let_def)

lemma bernpoly_aux_correct:
  "bernpoly_aux cs (map (λp. poly p x) ps) n x =
     map (λp. poly p x) (Bernpolys_aux cs ps n)"
  by (rule sym, induction n arbitrary: ps cs)
     (simp_all add: Let_def poly_sum_list poly_monom o_def case_prod_unfold
zip_map1
               del: upt_Suc of_nat_Suc)

lemma bernpoly_code [code]:
  "bernpoly n x = hd (bernpoly_aux [1, 1] [] (Suc n) x)"
proof -
  have "length (Bernpolys_aux [1, 1] ([] :: 'a poly list) (Suc n)) ≠
0"
    by (subst length_Bernpolys_aux) auto
  hence "Bernpolys_aux [1, 1] ([] :: 'a poly list) (Suc n) ≠ []"
    by (subst (asm) length_0_conv)
  thus ?thesis
    unfolding poly_Bernpoly [symmetric] Bernpoly_code
    using bernpoly_aux_correct[of "[1, 1]" x "[]" "Suc n"]
    by (simp add: hd_map del: Bernpolys_aux.simps bernpoly_aux.simps)
qed
```

7.5 Computing Euler polynomials

```
primrec Euler_polys_aux :: "nat list ⇒ 'a :: field_char_0 poly list ⇒
nat ⇒ 'a poly list" where
  "Euler_polys_aux cs xs 0 = xs"
| "Euler_polys_aux cs xs (Suc k) =
     (let n = length xs;
          p = Polynomial.monom 1 n - Polynomial.smult (1/2)
               (∑ (p,c)←zip xs (tl cs). Polynomial.smult (of_nat c)
```

```
p)
      in Euler_polys_aux (pascal_step cs) (p # xs) k)"

lemma length_Euler_polys_aux [simp]: "length (Euler_polys_aux cs xs n)
= length xs + n"
  by (induction n arbitrary: xs cs) (simp_all add: Let_def)

lemma Euler_polys_aux_correct:
  "Euler_polys_aux (binomial_row n) (map Euler_poly (rev [0..<n])) m =
map Euler_poly (rev [0..<m+n])"
proof (induction m arbitrary: n)
  case (Suc m n)
  define xs :: "'a poly list" where "xs = map Euler_poly (rev [0..<n])"
  define S where "S = (∑ (p,c)←zip xs (tl (binomial_row n)). Polynomial.smult
(of_nat c) p)"
  define q where "q = Polynomial.monom 1 n - Polynomial.smult (1/2) S"
  have [simp]: "length xs = n"
    by (simp add: xs_def)

  have "Euler_polys_aux (binomial_row n) (map Euler_poly (rev [0..<n])
:: 'a poly list) (Suc m) =
          Euler_polys_aux (binomial_row (Suc n)) (q # xs) m"
    by (simp del: upt_Suc add: q_def S_def xs_def)
  also have "q # xs = map Euler_poly (rev [0..<Suc n])"
  proof -
    have "q = Polynomial.monom 1 n - Polynomial.smult (1/2) S"
      by (simp add: q_def)
    also have "S = (∑ s<n. Polynomial.smult (of_nat (n choose Suc s))
(xs ! s))" unfolding S_def
      by (subst sum.list_conv_set_nth) (simp_all add: atLeast0LessThan
nth_tl del: upt_Suc)
    also have "... = (∑ s<n. Polynomial.smult (of_nat (n choose Suc s))
(Euler_poly (n - Suc s)))"
      by (intro sum.cong) (auto simp: xs_def rev_nth)
    also have "... = (∑ s<n. Polynomial.smult (of_nat (n choose (n - s)))
(Euler_poly s))"
      by (rule sum.reindex_bij_witness[of _ "λs. n - Suc s" "λs. n -
Suc s"]) auto
    also have "... = (∑ s<n. Polynomial.smult (of_nat (n choose s)) (Euler_poly
s))"
      by (intro sum.cong refl, subst binomial_symmetric) auto
    also have "Polynomial.monom 1 n - Polynomial.smult (1/2) ... = Euler_poly
n"
      by (rule Euler_poly_recurrence [symmetric])
    finally show ?thesis
      by (simp add: xs_def)
  qed
  also have "Euler_polys_aux (binomial_row (Suc n)) ... m = map Euler_poly
(rev [0..<m + Suc n])"
```

```
    by (rule Suc.IH)
  finally show ?case
    by (simp del: upt_Suc)
qed auto
```

The following function recursively computes a list of the Euler polynomials $E_0, \ldots, E_{n-1}$.

```
definition Euler_polys :: "nat ⇒ 'a :: field_char_0 poly list"
  where "Euler_polys n = rev (Euler_polys_aux [1] [] n)"

lemma length_Euler_polys [simp]: "length (Euler_polys n) = n"
  by (simp add: Euler_polys_def)

lemma Euler_polys_correct: "Euler_polys n = map Euler_poly [0..<n]"
  using Euler_polys_aux_correct[of 0 n, where ?'a = 'a]
  by (simp add: Euler_polys_def rev_swap binomial_row_def flip: rev_map)

lemma Euler_poly_code [code]: "Euler_poly n = hd (Euler_polys_aux [1]
[] (Suc n))"
  using Euler_polys_aux_correct[of 0 "Suc n", where ?'a = 'a]
  by (simp flip: rev_map add: hd_rev last_map binomial_row_def del: Euler_polys_aux.simps)


primrec euler_poly_aux :: "nat list ⇒ 'a :: {field_char_0, real_normed_field}
list ⇒ nat ⇒ 'a ⇒ 'a list" where
  "euler_poly_aux cs ys 0 x = ys"
| "euler_poly_aux cs ys (Suc k) x =
     (let n = length ys;
          y = x ^ n - (∑(y,c)←zip ys (tl cs). of_nat c * y) / 2
      in euler_poly_aux (pascal_step cs) (y # ys) k x)"

lemma length_euler_poly_aux [simp]: "length (euler_poly_aux cs xs n x)
= length xs + n"
  by (induction n arbitrary: xs cs) (simp_all add: Let_def)

lemma euler_poly_aux_correct:
  "euler_poly_aux cs (map (λp. poly p x) ps) n x = map (λp. poly p x)
(Euler_polys_aux cs ps n)"
  by (rule sym, induction n arbitrary: ps cs)
     (simp_all add: Let_def poly_sum_list poly_monom o_def case_prod_unfold
zip_map1
               del: upt_Suc of_nat_Suc)

lemma euler_poly_code [code]:
  "euler_poly n x = hd (euler_poly_aux [1] [] (Suc n) x)"
proof -
  have "length (Euler_polys_aux [1] ([] :: 'a poly list) (Suc n)) ≠ 0"
    by (subst length_Euler_polys_aux) auto
  hence "Euler_polys_aux [1] ([] :: 'a poly list) (Suc n) ≠ []"
```

```
    by (subst (asm) length_0_conv)
  thus ?thesis
    unfolding poly_Euler_poly [symmetric] Euler_poly_code
    using euler_poly_aux_correct[of "[1]" x "[]" "Suc n"]
    by (simp add: hd_map del: Euler_polys_aux.simps euler_poly_aux.simps)
qed

end
```

8 The Boustrophedon transform

```
theory Boustrophedon_Transform
  imports "HOL-Computational_Algebra.Computational_Algebra" Alternating_Permutations
begin
```

The Boustrophedon transform maps one sequence of numbers to another sequence of numbers – or, equivalently, one exponential generating function to another exponential generating function. It was first described in its full generality by Millar et al. [2].

Its name derives from the Ancient Greek βοῦς ("ox"), στροφή ("turn"), and -ηδόν ("in the manner of") because the number triangle from which it is obtained can be visualised as being traversed left-to-right, then right-to-left, etc. the same way an ox plows a field.

8.1 The Seidel triangle

We define the triangle via its simplest recurrence. Let $T_{n,k}$ denote the k-th entry of the n-th row. The first entry of the n-th row is always $a(n)$, where a is the input sequence. The $k+1$-th entry of a row is the sum of the previous entry in the same row and the k-th last entry of the previous row.

That is: $T_{n,0} = a(n)$ and $T_{n+1,k+1} = T_{n+1,k} + T_{n,n-k}$.

In other words: one produces a new row of the triangle by starting with $a(n)$ and then adding the entries of the previous row, in right-to-left order, adding each intermediate sum to the new row.

```
fun seidel_triangle :: "(nat ⇒ 'a :: monoid_add) ⇒ nat ⇒ nat ⇒ 'a"
where
  "seidel_triangle a n 0 = a n"
| "seidel_triangle a 0 (Suc k) = 0"
| "seidel_triangle a (Suc n) (Suc k) =
     (if k > n then 0 else seidel_triangle a (Suc n) k + seidel_triangle
a n (n - k))"

lemmas seidel_triangle_rec [simp del] = seidel_triangle.simps(3)

lemma seidel_triangle_greater_eq_0 [simp]: "k > n ⟹ seidel_triangle
a n k = 0"
```

```
  by (cases n; cases k) (auto simp: seidel_triangle_rec)
```

There is also the following recurrence where the right-hand side contains only the entries of the previous row. Namely: The entry $T_{n,k}$ is equal to the sum of a_n and the last k entries of the previous row.

```
lemma seidel_triangle_conv_rowsum:
  assumes "k ≤ n"
  shows   "seidel_triangle a n k = a n + (∑j<k. seidel_triangle a (n
- 1) (n - Suc j))"
  using assms
proof (induction k)
  case (Suc k)
  then obtain n' where [simp]: "n = Suc n'"
    by (cases n) auto
  show ?case
    using Suc.IH Suc.prems by (simp add: seidel_triangle_rec add_ac)
qed auto
```

The following function is the function $\pi(n, k, i)$ from the paper by Millar et al. They define it via the number of paths from one node to another node in a triangular directed graph.

However, they also give a closed-form expression for $\pi(n, k, i)$ as a sum of binomial coefficients and Entringer numbers, and we directly use this since it seemed easier to formalise.

```
definition seidel_triangle_aux :: "nat ⇒ nat ⇒ nat ⇒ nat" where
  "seidel_triangle_aux n k i =
     (∑s≤min k (n-i). (k choose s) * ((n-k) choose (n-i-s)) * entringer_number
(n-i) s)"

lemma seidel_triangle_aux_same:
  assumes i: "i ≤ n"
  shows   "seidel_triangle_aux n n i = (n choose i) * zigzag_number (n
- i)"
proof -
  have "seidel_triangle_aux n n i =
          (∑s≤n - i. (n choose s) * (0 choose (n - (i + s))) * entringer_number
(n - i) s)"
    by (simp add: seidel_triangle_aux_def)
  also have "... = (∑s∈{n-i}. (n choose s) * (0 choose (n - (i + s)))
* entringer_number (n - i) s)"
    by (rule sum.mono_neutral_right) auto
  also have "... = (n choose i) * zigzag_number (n - i)"
    using i by (simp flip: binomial_symmetric)
  finally show ?thesis .
qed

lemma seidel_triangle_aux_same2 [simp]: "seidel_triangle_aux n k n =
1"
```

```
  by (simp add: seidel_triangle_aux_def)

lemma seidel_triangle_aux_0_middle [simp]:
  "i < n ⟹ seidel_triangle_aux n 0 i = 0"
  by (simp add: seidel_triangle_aux_def flip: binomial_symmetric)

lemma seidel_triangle_aux_0_right [simp]:
  assumes "k ≤ n"
  shows   "seidel_triangle_aux n k 0 = entringer_number n k"
proof -
  have "seidel_triangle_aux n k 0 = (∑s≤k. (k choose s) * (n - k choose
(n - s)) * entringer_number n s)"
    using assms by (simp add: seidel_triangle_aux_def)
  also have "... = (∑s∈{k}. (k choose s) * (n - k choose (n - s)) * entringer_number
n s)"
    by (rule sum.mono_neutral_right) (use assms in auto)
  finally show ?thesis
    by simp
qed
```

The following lemma is where most of the proof work is done. Millar et al. do not mention it expicitly, but π satistifies the recurrence $\pi(n+1, k+1, i) = \pi(n+1, k, i) + \pi(n, n-k, i)$.

Note that this is the same type of recurrence that we have in the Seidel triangle and the Entringer numbers.

```
lemma seidel_triangle_aux_rec:
  defines "S ≡ seidel_triangle_aux"
  assumes k: "k ≤ n" and i: "i ≤ n"
  shows   "S (Suc n) (Suc k) i = S (Suc n) k i + S n (n - k) i"
proof -
  define N where "N = int n"
  define K where "K = int k"
  define I where "I = int i"

  define B where "B = (λn k. if n < 0 ∨ k < 0 then 0 else ((nat n) choose
(nat k)))"
  have [simp]: "B n k = 0" if "k < 0 ∨ k > n ∨ n < 0" for n k
    using that by (auto simp: B_def)
  have B_rec: "B (N+1) (K+1) = B N (K+1) + B N K" if "N ≥ 0" for N K
    using that by (auto simp: B_def nat_add_distrib not_less)
  have B_eq: "B n' k' = (n choose k)" if "int n = n'" "int k = k'" for
n n' k k'
    unfolding B_def using that by auto
  have B_mult_cong: "B x y * z = B x y * z'" if "x ≥ 0 ∧ y ≥ 0 ∧ y ≤
x ⟶ z = z'" for x y z z'
    using that by (auto simp: B_def)

  define E where "E = (λn k. if n < 0 ∨ k < 0 then 0 else entringer_number
(nat n) (nat k))"
```

```
  have [simp]: "E n k = 0" if "k < 0 ∨ k > n ∨ n < 0" for n k
    using that by (auto simp: E_def)
  have E_rec: "E (n+1) (k+1) = E (n+1) k + E n (n-k)" if "n ≥ 0" "k ≤
n" for n k
    using that by (auto simp: E_def nat_add_distrib entringer_number_rec
nat_diff_distrib)
  have E_eq: "E n' k' = entringer_number n k" if "int n = n'" "int k =
k'" for n n' k k'
    unfolding E_def using that by auto

  have S_eq: "S n k i = (∑?s. B k' s * B (n'-k') (n'-i'-s) * E (n'-i')
s)"
    if "k ≤ n" "i ≤ n" "k' = int k" "n' = int n" "i' = int i" for k n
i :: nat and k' n' i' :: int
  proof -
    have "S n k i = (∑s≤min k (n - i). B k' s * B (n'-k') (n'-i'-s)
* E (n'-i') s)"
      unfolding S_def seidel_triangle_aux_def using that
      by (intro sum.cong arg_cong2[of _ _ _ _ "(*)"] B_eq[symmetric] E_eq[symmetric])
auto
    also have "... = (∑s∈{0..min k' (n' - i')}. B k' s * B (n'-k') (n'-i'-s)
* E (n'-i') s)"
      by (rule sum.reindex_bij_witness[of _ nat int]) (use that in auto)
    also have "... = (∑?s. B k' s * B (n'-k') (n'-i'-s) * E (n'-i') s)"
      by (rule Sum_any.expand_superset_cong [symmetric]) auto
    finally show ?thesis .
  qed

  have "S (Suc n) (Suc k) i =
          (∑?s. B (K+1) s * B ((N+1)-(K+1)) (N+1-I-s) * E (N+1-I) s)"
    by (rule S_eq) (use assms in ‹auto simp: N_def K_def I_def›)
  also have "... = (∑?s. B (K+1) (s+1) * (B (N-K) (N-I-s) * E (N-I+1)
(s+1)))"
    by (rule Sum_any.reindex_bij_witness[of "λs. s+1" "λs. s-1"]) (auto
simp: algebra_simps)
  also have "... = (∑?s. B (K+1) (s+1) * (B (N-K) (N-I-s) * (E (N-I+1)
s + E (N-I) (N-I-s))))"
    by (intro Sum_any.cong B_mult_cong impI, subst E_rec)
       (use assms in ‹auto simp: N_def I_def›)
  also have "... = (∑?s. B (K+1) (s+1) * B (N-K) (N-I-s) * E (N-I+1)
s) +
                 (∑?s. B (K+1) (s+1) * B (N-K) (N-I-s) * E (N-I) (N-I-s))"
    unfolding ring_distribs mult.assoc [symmetric]
    by (rule Sum_any.distrib'[where A = "{0..N-I}"]) auto
  also have "(∑?s. B (K+1) (s+1) * B (N-K) (N-I-s) * E (N-I) (N-I-s))
=
             (∑?s. B (K+1) (N-I-s+1) * B (N-K) s * E (N-I) s)"
    by (rule Sum_any.reindex_bij_witness[of "λs. N-I-s" "λs. N-I-s"])
auto
```

```
  also have "K ≥ 0"
    by (simp add: K_def)
  have "(∑?s. B (K+1) (s+1) * B (N-K) (N-I-s) * E (N-I+1) s) =
          (∑?s. B K (s+1) * B (N-K) (N-I-s) * E (N-I+1) s) +
          (∑?s. B K s * B (N-K) (N-I-s) * E (N-I+1) s)"
    unfolding B_rec[OF <K ≥ 0>] ring_distribs
    by (rule Sum_any.distrib'[where A = "{0..K}"]) auto

  also have "(∑?s. B (K+1) (N-I-s+1) * B (N-K) s * E (N-I) s) =
              (∑?s. B K (N-I-s+1) * B (N-K) s * E (N-I) s) +
              (∑?s. B K (N-I-s) * B (N-K) s * E (N-I) s)"
    unfolding B_rec[OF <K ≥ 0>] ring_distribs
    by (rule Sum_any.distrib'[where A = "{0..N-I+1}"]) auto
  finally have eq: "S (Suc n) (Suc k) i =
                  (∑?s. B K (s+1) * B (N-K) (N-I-s) * E (N-I+1) s) +
                  (∑?s. B K s * B (N-K) (N-I-s) * E (N-I+1) s) +
                  (∑?s. B K (N-I-s+1) * B (N-K) s * E (N-I) s) +
                  (∑?s. B (N-K) s * B K (N-I-s) * E (N-I) s)"
    (is "_ = ?S1 + ?S2 + ?S3 + ?S4")
    by (simp only: add_ac mult.commute)

  have "S (Suc n) k i + S n (n - k) i =
         (∑?s. B K s * B (N+1-K) (N+1-I-s) * E (N+1-I) s) +
         (∑?s. B (N - K) s * B (N-(N - K)) (N-I-s) * E (N-I) s)"
    using assms by (intro arg_cong2[of _ _ _ _ "(+)"] S_eq) (auto simp:
N_def K_def I_def)
  also have "... = (∑?s. B K s * B (N-K+1) (N-I-s+1) * E (N-I+1) s) +
                  (∑?s. B (N - K) s * B K (N-I-s) * E (N-I) s)"
    by (simp add: algebra_simps)
  also have "N - K ≥ 0"
    using assms by (simp add: N_def K_def)
  have "(∑?s. B K s * B (N-K+1) (N-I-s+1) * E (N-I+1) s) =
              (∑?s. B K s * B (N-K) (N-I-s+1) * E (N-I+1) s) + ?S2"
    unfolding B_rec[OF <N - K ≥ 0>] ring_distribs
    by (rule Sum_any.distrib'[where A = "{0..K}"]) auto

  also have "(∑?s. B K s * B (N-K) (N-I-s+1) * E (N-I+1) s) = ?S1 + ?S3"
  proof -
    have "N - I ≥ 0"
      using assms by (auto simp: N_def I_def)
    have "(∑?s. B K s * B (N-K) (N-I-s+1) * E (N-I+1) s) =
          (∑?s. B K (s+1) * (B (N-K) (N-I-s) * E (N-I+1) (s+1)))"
      by (rule Sum_any.reindex_bij_witness[of "λs. s+1" "λs. s-1"]) (auto
simp: algebra_simps)
    also have "... = (∑?s. B K (s+1) * (B (N-K) (N-I-s) * (E (N-I+1)
s + E (N-I) (N-I-s))))"
      by (intro Sum_any.cong B_mult_cong impI, subst E_rec) (use <N -
I ≥ 0> in auto)
```

```
    also have "... = ?S1 + (∑?s. B K (s+1) * B (N-K) (N-I-s) * E (N-I)
(N-I-s))"
      unfolding ring_distribs mult.assoc [symmetric]
      by (rule Sum_any.distrib'[where A = "{0..K}"]) auto
    also have "(∑?s. B K (s+1) * B (N-K) (N-I-s) * E (N-I) (N-I-s)) =
               (∑?s. B K (N-I-s+1) * B (N-K) s * E (N-I) s)"
      by (rule Sum_any.reindex_bij_witness[of "λs. N-I-s" "λs. N-I-s"])
(auto simp: algebra_simps)
    finally show ?thesis .
  qed

  finally show ?thesis
    using eq by algebra
qed
```

With this, we can prove the following closed form for the entry $T_{n,k}$ in the Seidel triangle.

```
theorem seidel_triangle_eq:
  assumes "k ≤ n"
  shows   "seidel_triangle a n k = (∑i≤n. of_nat (seidel_triangle_aux
n k i) * a i)"
  using assms
proof (induction a n k rule: seidel_triangle.induct)
  case (1 a n)
  have "(∑i≤n. of_nat (seidel_triangle_aux n 0 i) * a i) =
        (∑i∈{n}. of_nat (seidel_triangle_aux n 0 i) * a i)"
    by (rule sum.mono_neutral_right) (auto simp: seidel_triangle_aux_def)
  thus ?case
    by (simp add: seidel_triangle_aux_def)
next
  case (3 a n k)
  define S where "S = (λn k i. of_nat (seidel_triangle_aux n k i) ::
'a)"
  from "3.prems" have "k ≤ n"
    by simp
  have "seidel_triangle a (Suc n) (Suc k) =
          seidel_triangle a (Suc n) k + seidel_triangle a n (n - k)"
    using ‹k ≤ n› by (simp add: seidel_triangle_rec)
  also have "seidel_triangle a (Suc n) k = (∑i≤n. S (Suc n) k i * a
i) + a (Suc n)"
    unfolding S_def by (subst "3.IH") (use ‹k ≤ n› in auto)
  also have "seidel_triangle a n (n - k) = (∑i≤n. S n (n - k) i * a
i)"
    unfolding S_def by (subst "3.IH") (use ‹k ≤ n› in auto)
  also have "(∑i≤n. S (Suc n) k i * a i) + a (Suc n) + (∑i≤n. S n
(n - k) i * a i) =
             (∑i≤n. (S (Suc n) k i + S n (n - k) i) * a i) + a (Suc
n)"
    by (simp add: sum.distrib add_ac ring_distribs)
```

```
  also have "(∑i≤n. (S (Suc n) k i + S n (n - k) i) * a i) = (∑i≤n.
S (Suc n) (Suc k) i * a i)"
    by (rule sum.cong) (use ‹k ≤ n› in ‹simp_all add: S_def seidel_triangle_aux_rec›)
  also have "... + a (Suc n) = (∑i≤Suc n. S (Suc n) (Suc k) i * a i)"
    by (simp add: S_def)
  finally show ?case
    by (simp add: S_def)
qed auto
```

8.2 The Boustrophedon transform of a sequence

The Boustrophedon transform of a sequence a_n is defined by taking the last entry of each row of the Seidel triangle of a_n.

```
definition boustrophedon :: "(nat ⇒ 'a :: monoid_add) ⇒ nat ⇒ 'a" where
  "boustrophedon a n = seidel_triangle a n n"

definition inv_boustrophedon :: "(nat ⇒ 'a :: comm_ring_1) ⇒ nat ⇒ 'a"
where
  "inv_boustrophedon a n = (-1)^n * boustrophedon (λk. (-1)^k * a k) n"
```

The Boustrophedon transform has the following nice closed form, which of course follows directly from our above closed form for the Seidel triangle:

```
theorem boustrophedon_eq:
  fixes a :: "nat ⇒ 'a :: comm_semiring_1"
  shows "boustrophedon a n = (∑k≤n. of_nat (n choose k) * a k * of_nat
(zigzag_number (n - k)))"
  by (simp add: boustrophedon_def seidel_triangle_eq seidel_triangle_aux_same
mult_ac)
```

The inverse Boustrophedon transform is the same as the normal Boustrophedon transform except that we must negate every other number in the input and output sequences.

```
theorem inv_boustrophedon_eq:
  fixes a :: "nat ⇒ 'a :: comm_ring_1"
  shows "inv_boustrophedon a n = (∑k≤n. (-1) ^ (n - k) * of_nat (n
choose k) * a k * of_nat (zigzag_number (n - k)))"
  unfolding inv_boustrophedon_def boustrophedon_eq sum_distrib_left
  by (intro sum.cong) (auto simp: uminus_power_if)
```

In particular, the Entringer numbers are the Seidel triangle of the sequence $1, 0, 0, 0, \ldots$.

```
corollary entringer_number_conv_seidel_triangle:
  "seidel_triangle (λn. if n = 0 then 1 else 0 :: 'a :: comm_semiring_1)
n k =
     of_nat (entringer_number n k)"
proof (cases "k ≤ n")
  case True
```

```
  have "k ≤ n"
    using True by auto
  have "seidel_triangle (λn. if n = 0 then 1 else 0 :: 'a) n k =
          of_nat (∑i≤n. seidel_triangle_aux n k i * (if i = 0 then 1
else 0))"
    unfolding seidel_triangle_eq[OF <k ≤ n>] of_nat_sum
    by (rule sum.cong) (use True in auto)
  also have "(∑i≤n. seidel_triangle_aux n k i * (if i = 0 then 1 else
0)) =
             (∑i∈{0}. seidel_triangle_aux n k i * (if i = 0 then 1 else
0))"
    by (rule sum.mono_neutral_right) auto
  also have "... = entringer_number n k"
    using True by simp
  finally show ?thesis .
qed auto
```

And consequently, the zigzag numbers are the Boustrophedon transform of the sequence $1, 0, 0, 0, \ldots$.

```
corollary zigzag_number_conv_boustrophedon:
  "boustrophedon (λn. if n = 0 then 1 else 0 :: 'a :: comm_semiring_1)
n =
     of_nat (zigzag_number n)"
  unfolding boustrophedon_def
  by (subst entringer_number_conv_seidel_triangle) auto
```

8.3 The Boustrophedon transform of a function

Analogously, one can define the Boustrophedon transform $\mathcal{B}(f)(x)$ of an exponential generating function $f(x) = \sum_{n\geq 0} f(n)/n!x^n$ and its inverse $\mathcal{B}^{-1}(f)(x)$:

```
definition Boustrophedon :: "'a :: field_char_0 fps ⇒ 'a fps" where
  "Boustrophedon A = Abs_fps (λn. boustrophedon (λn. fps_nth A n * fact
n) n / fact n)"

definition inv_Boustrophedon :: "'a :: field_char_0 fps ⇒ 'a fps" where
  "inv_Boustrophedon A = Abs_fps (λn. inv_boustrophedon (λn. fps_nth A
n * fact n) n / fact n)"

lemma fps_nth_Boustrophedon:
  fixes A :: "'a :: field_char_0 fps"
  shows "fps_nth (Boustrophedon A) n =
           (∑k≤n. fps_nth A k * of_nat (zigzag_number (n - k)) / fact
(n - k))"
  by (simp add: Boustrophedon_def boustrophedon_eq field_simps sum_distrib_left
sum_distrib_right
                binomial_fact)
```

```
lemma fps_nth_inv_Boustrophedon:
  fixes A :: "'a :: field_char_0 fps"
  shows "fps_nth (inv_Boustrophedon A) n =
           (∑k≤n. (-1)^(n-k) * fps_nth A k * of_nat (zigzag_number (n
- k)) / fact (n - k))"
  by (simp add: inv_Boustrophedon_def inv_boustrophedon_eq field_simps

                sum_distrib_left sum_distrib_right binomial_fact)
```

We have the closed form $\mathcal{B}(f) = (\sec + \tan)f$:

```
theorem Boustrophedon_altdef:
  fixes A :: "'a :: field_char_0 fps"
  shows "Boustrophedon A = (fps_sec 1 + fps_tan 1) * A"
  by (subst mult.commute, rule fps_ext,
      subst exponential_generating_function_zigzag_number [symmetric])
     (simp add: fps_nth_Boustrophedon fps_mult_nth atLeast0AtMost)
```

It is also easy to see from the definition of $\mathcal{B}^{-1}$ that we have $\mathcal{B}^{-1}(f)(x) = \mathcal{B}(g)(-x)$, where $g(x) = f(-x)$.

```
theorem inv_Boustrophedon_altdef1:
  fixes A :: "'a :: field_char_0 fps"
  shows "inv_Boustrophedon A = fps_compose (Boustrophedon (fps_compose
A (-fps_X))) (-fps_X)"
  by (rule fps_ext)
     (simp_all add: inv_Boustrophedon_def Boustrophedon_def fps_nth_compose_uminus
                    inv_boustrophedon_def mult.assoc)
```

Or, yet another view on $\mathcal{B}^{-1}$: $\mathcal{B}^{-1}(f)(x) = (\sec(-x) + \tan(-x))f(x)$.

```
lemma inv_Boustrophedon_altdef2:
  fixes A :: "'a :: field_char_0 fps"
  shows "inv_Boustrophedon A = (fps_sec 1 - fps_tan 1) * A"
proof -
  have "inv_Boustrophedon A =
          (A * fps_compose (Abs_fps (λk. of_nat (zigzag_number k) / fact
k)) (-fps_X))"
    unfolding fps_eq_iff fps_nth_inv_Boustrophedon fps_mult_nth
    by (simp add: fps_nth_compose_uminus mult_ac atLeast0AtMost)
  also have "Abs_fps (λk. of_nat (zigzag_number k) / fact k) = fps_sec
(1::'a) + fps_tan 1"
    by (simp add: exponential_generating_function_zigzag_number)
  also have "fps_compose ... (-fps_X) = fps_sec 1 - fps_tan 1"
    by (simp add: fps_compose_add_distrib fps_sec_even fps_tan_odd)
  finally show ?thesis by (simp add: mult.commute)
qed

lemma fps_sec_plus_tan_times_sec_minus_tan:
  "(fps_sec (c ::'a :: field_char_0) + fps_tan c) * (fps_sec c - fps_tan
c) = 1"
proof -
```

```
  define S where "S = fps_to_fls (fps_sin c)"
  define C where "C = fps_to_fls (fps_cos c)"
  have "fls_nth C 0 = 1"
    by (simp add: C_def)
  hence [simp]: "C ≠ 0"
    by auto

  have "fps_to_fls ((fps_sec c + fps_tan c) * (fps_sec c - fps_tan c))
=
          (inverse C + S / C) * (inverse C - S / C)"
    by (simp add: fps_sec_def fps_tan_def fls_times_fps_to_fls S_def C_def
             flip: fls_inverse_fps_to_fls fls_divide_fps_to_fls)
  also have "(inverse C - S / C) = (1 - S) / C"
    by (simp add: divide_simps)
  also have "(inverse C + S / C) = (1 + S) / C"
    by (simp add: divide_simps)
  also have "(1 + S) / C * ((1 - S) / C) = (1 - S ^ 2) / C ^ 2"
    by (simp add: algebra_simps power2_eq_square)
  also have "1 - S ^ 2 = C ^ 2"
  proof -
    have "1 - S ^ 2 = fps_to_fls (1 - fps_sin c ^ 2)"
      by (simp add: S_def fps_to_fls_power)
    also have "1 - fps_sin c ^ 2 = fps_cos c ^ 2"
      using fps_sin_cos_sum_of_squares[of c]  by algebra
    also have "fps_to_fls ... = C ^ 2"
      by (simp add: C_def fps_to_fls_power)
    finally show ?thesis .
  qed
  also have "C ^ 2 / C ^ 2 = fps_to_fls 1"
    by simp
  finally show ?thesis
    by (simp only: fps_to_fls_eq_iff)
qed
```

Or, equivalently: $\mathcal{B}^{-1}(f) = f/(\sec + \tan)$.

```
theorem inv_Boustrophedon_altdef3:
  fixes A :: "'a :: field_char_0 fps"
  shows "inv_Boustrophedon A = A / (fps_sec 1 + fps_tan 1)"
proof (rule sym, rule divide_fps_eqI)
  have "inv_Boustrophedon A * (fps_sec 1 + fps_tan 1) =
          ((fps_sec 1 + fps_tan 1) * (fps_sec 1 - fps_tan 1)) * A"
    unfolding inv_Boustrophedon_altdef2 by (simp only: mult_ac)
  thus "inv_Boustrophedon A * (fps_sec 1 + fps_tan 1) = A"
    by (simp only: fps_sec_plus_tan_times_sec_minus_tan mult_1_left)
next
  have "fps_nth (fps_sec 1 + fps_tan (1::'a)) 0 = 1"
    by auto
  hence "fps_sec 1 + fps_tan (1::'a) ≠ 0"
    by (intro notI) simp_all
```

```
  thus "A ≠ 0 ∨ fps_sec 1 + fps_tan (1::'a) ≠ 0 ∨ inv_Boustrophedon
A = 0"
    by blast
qed
```

It is now obvious that $\mathcal{B}$ and $\mathcal{B}^{-1}$ really are inverse to one another.

```
lemma Boustrophedon_inv_Boustrophedon [simp]:
  fixes A :: "'a :: field_char_0 fps"
  shows "Boustrophedon (inv_Boustrophedon A) = A"
proof -
  have "Boustrophedon (inv_Boustrophedon A) =
          A * ((fps_sec (1::'a) + fps_tan 1) * (fps_sec 1 - fps_tan 1))"
    by (simp add: Boustrophedon_altdef inv_Boustrophedon_altdef2)
  also have "(fps_sec (1::'a) + fps_tan 1) * (fps_sec 1 - fps_tan 1) =
1"
    by (rule fps_sec_plus_tan_times_sec_minus_tan)
  finally show ?thesis
    by simp
qed

lemma inv_Boustrophedon_Boustrophedon [simp]:
  fixes A :: "'a :: field_char_0 fps"
  shows "inv_Boustrophedon (Boustrophedon A) = A"
proof -
  have "inv_Boustrophedon (Boustrophedon A) =
          A * ((fps_sec (1::'a) + fps_tan 1) * (fps_sec 1 - fps_tan 1))"
    by (simp add: Boustrophedon_altdef inv_Boustrophedon_altdef2)
  also have "(fps_sec (1::'a) + fps_tan 1) * (fps_sec 1 - fps_tan 1) =
1"
    by (rule fps_sec_plus_tan_times_sec_minus_tan)
  finally show ?thesis
    by simp
qed


end
theory Boustrophedon_Transform_Impl
  imports Boustrophedon_Transform Secant_Numbers Tangent_Numbers "HOL-Library.Stream"
begin
```

8.4 Implementation

In the following we will provide some simple functions based on infinite streams to compute the Seidel triangle and the Boustrophedon transform of a sequence efficiently.

The core functionality is the following auxiliary function, which produces the next row of the Seidel triangle from the current row and the corresponding entry in the input sequence.

```
primrec seidel_triangle_rows_step :: "'a :: monoid_add ⇒ 'a list ⇒
'a list" where
  "seidel_triangle_rows_step a [] = [a]"
| "seidel_triangle_rows_step a (x # xs) = a # seidel_triangle_rows_step
(a + x) xs"

primrec seidel_triangle_rows_step_tailrec :: "'a :: monoid_add ⇒ 'a list
⇒ 'a list ⇒ 'a list" where
  "seidel_triangle_rows_step_tailrec a [] acc = a # acc"
| "seidel_triangle_rows_step_tailrec a (x # xs) acc =
     seidel_triangle_rows_step_tailrec (a + x) xs (a # acc)"

lemma seidel_triangle_rows_step_tailrec_correct [simp]:
  "seidel_triangle_rows_step_tailrec a xs acc =
   rev (seidel_triangle_rows_step a xs) @ acc"
  by (induction xs arbitrary: a acc) simp_all

lemma length_seidel_triangle_rows_step [simp]:
  "length (seidel_triangle_rows_step a xs) = Suc (length xs)"
  by (induction xs arbitrary: a) auto

lemma nth_seidel_triangle_rows_step:
  "i ≤ length xs ⟹ seidel_triangle_rows_step a xs ! i = a + sum_list
(take i xs)"
  by (induction xs arbitrary: i a) (auto simp: nth_Cons add_ac split:
nat.splits)

lemma seidel_triangle_rows_step_correct:
  fixes a :: "nat ⇒ 'a :: comm_monoid_add"
  shows "seidel_triangle_rows_step (a n) (map (seidel_triangle a (n-Suc
0)) (rev [0..<n])) =
           map (seidel_triangle a n) [0..<Suc n]"
proof (rule nth_equalityI, goal_cases)
  case i: (2 i)
  have "seidel_triangle_rows_step (a n) (map (seidel_triangle a (n-1))
(rev [0..<n])) ! i =
          a n + sum_list (take i (map (seidel_triangle a (n - Suc 0))
(rev [0..<n])))"
    using i by (subst nth_seidel_triangle_rows_step) auto
  also have "sum_list (take i (map (seidel_triangle a (n - Suc 0)) (rev
[0..<n]))) =
               (∑j<i. seidel_triangle a (n - 1) (n - Suc j))"
    using i by (subst sum.list_conv_set_nth) (simp_all add: atLeast0LessThan
rev_nth)
  also have "a n + ... = seidel_triangle a n i"
    by (rule seidel_triangle_conv_rowsum [symmetric]) (use i in auto)
  also have "... = map (seidel_triangle a n) [0..<Suc n] ! i"
    using i by (subst nth_map) (auto simp del: upt_Suc)
  finally show ?case by simp
```

```
qed auto
```

This auxiliary function produces an infinite stream of all the subsequent rows of the Seidel triangle, given the current row and a stream of the remaining elements of the input sequence.

```
primcorec seidel_triangle_rows_aux :: "'a :: comm_monoid_add stream ⇒
'a list ⇒ 'a list stream" where
  "seidel_triangle_rows_aux as xs =
     (let ys = seidel_triangle_rows_step_tailrec (shd as) xs []
      in  rev ys ## seidel_triangle_rows_aux (stl as) ys)"

lemma seidel_triangle_rows_aux_correct:
  "seidel_triangle_rows_aux (sdrop n as)
     (map (seidel_triangle (λi. as !! i) (n-Suc 0)) (rev [0..<n])) !!
m =
   map (seidel_triangle (λi. as !! i) (n + m)) [0..<Suc (n+m)]"
proof (induction m arbitrary: n)
  case 0
  show ?case
    by (simp add: seidel_triangle_rows_step_correct del: upt_Suc)
next
  case (Suc m n)
  have "seidel_triangle_rows_aux (sdrop n as)
          (map (seidel_triangle ((!!) as) (n - 1)) (rev [0..<n])) !! Suc
m =
        seidel_triangle_rows_aux (sdrop (Suc n) as)
                 (map (seidel_triangle ((!!) as) n) (rev [0..<Suc n]))
!! m"
    by (simp add: seidel_triangle_rows_step_correct rev_map del: upt_Suc)
  also have "... = map (seidel_triangle ((!!) as) (Suc (n + m))) [0..<n+m+2]"
    using Suc.IH[of "Suc n"] by (simp del: upt_Suc)
  finally show ?case
    by simp
qed
```

This function produces an infinite stream of all the rows of the Seidel triangle of the sequence given by the input stream.

Note that in the literature the triangle is often printed with every other row reversed, to emphasise the "ox-plow" nature of the recurrence. It is however mathematically more natural to not do this, so our version does not do this.

```
definition seidel_triangle_rows :: "'a :: comm_monoid_add stream ⇒ 'a
list stream" where
  "seidel_triangle_rows as = seidel_triangle_rows_aux as []"

lemma seidel_triangle_rows_correct:
  "seidel_triangle_rows as !! n = map (seidel_triangle (λi. as !! i) n)
[0..<Suc n]"
  using seidel_triangle_rows_aux_correct[of 0 as n]
```

```
  by (simp del: upt_Suc add: seidel_triangle_rows_def)


primcorec boustrophedon_stream_aux :: "'a :: comm_monoid_add stream ⇒
'a list ⇒ 'a stream" where
  "boustrophedon_stream_aux as xs =
     (let ys = seidel_triangle_rows_step_tailrec (shd as) xs []
      in  hd ys ## boustrophedon_stream_aux (stl as) ys)"

lemma boustrophedon_stream_aux_conv_seidel_triangle_rows_aux:
  "boustrophedon_stream_aux as xs = smap last (seidel_triangle_rows_aux
as xs)"
  by (coinduction arbitrary: as xs) (auto simp: hd_rev)

lemma boustrophedon_stream_aux_correct:
  "boustrophedon_stream_aux (sdrop n as)
     (map (seidel_triangle (λi. as !! i) (n - Suc 0)) (rev [0..<n])) !!
m =
   boustrophedon (λi. as !! i) (n + m)"
  by (subst boustrophedon_stream_aux_conv_seidel_triangle_rows_aux, subst
snth_smap,
      subst seidel_triangle_rows_aux_correct)
     (simp add: boustrophedon_def)
```

This function produces the Boustrophedon transform of a stream.

```
definition boustrophedon_stream :: "'a :: comm_monoid_add stream ⇒ 'a
stream" where
  "boustrophedon_stream as = boustrophedon_stream_aux as []"

lemma boustrophedon_stream_correct:
  "boustrophedon_stream as !! n = boustrophedon (λi. as !! i) n"
  using boustrophedon_stream_aux_correct[of 0 as n]
  by (simp add: boustrophedon_stream_def)
```

Lastly, we also provide a function to compute a single number in the transformed sequence to avoid code-generation problems related to streams.

```
fun seidel_triangle_impl_aux :: "(nat ⇒ 'a :: comm_monoid_add) ⇒ 'a
list ⇒ nat ⇒ nat ⇒ nat ⇒ 'a" where
  "seidel_triangle_impl_aux a xs i n k =
     (let ys = seidel_triangle_rows_step_tailrec (a i) xs []
      in  if n = 0 then ys ! (i - k) else seidel_triangle_impl_aux a ys
(i + 1) (n - 1) k)"

lemmas [simp del] = seidel_triangle_impl_aux.simps


lemma seidel_triangle_impl_aux_correct:
  assumes "k ≤ n + i" "length xs = i"
  shows   "seidel_triangle_impl_aux a xs i n k =
```

```
            seidel_triangle_rows_aux (smap a (fromN i)) xs !! n ! k"
  using assms
  by (induction n arbitrary: k i xs)
     (subst seidel_triangle_impl_aux.simps; simp add: Let_def rev_nth)+

lemma seidel_triangle_code [code]:
  "seidel_triangle a n k = (if k > n then 0 else seidel_triangle_impl_aux
a [] 0 n k)"
  using seidel_triangle_impl_aux_correct[of k n 0 "[]" a]
        seidel_triangle_rows_aux_correct[of 0 "smap a nats" n]
  by (simp del: upt_Suc)

lemma entringer_number_code [code]:
  "entringer_number n k = seidel_triangle (λn. if n = 0 then 1 else 0)
n k"
  by (subst entringer_number_conv_seidel_triangle) auto

end
```

9 Code generation tests

```
theory Boustrophedon_Transform_Impl_Test
imports
  Boustrophedon_Transform_Impl
  Euler_Numbers
  "HOL-Library.Code_Lazy"
  "HOL-Library.Code_Target_Numeral"
begin
```

We now test all the various functions we have implemented.

```
value "zigzag_number 100"
value "zigzag_numbers 100"
value "secant_number 100"
value "secant_numbers 100"
value "tangent_number 100"
value "tangent_numbers 100"
value "euler_number 100"
value "entringer_number 100 32"

value "Bernpolys 20 :: real poly list"
value "Bernpoly 10 :: real poly"
value "Bernpoly 51 :: real poly"
value "bernpoly 10 (1/2) :: real"

value "Euler_polys 20 :: rat poly list"
value "Euler_poly 10 :: rat poly"
value "Euler_poly 51 :: rat poly"
value "euler_poly 51 (3/2) :: real"
```

```
code_lazy_type stream
```

As an example of the Boustrophedon transform, the following is the transform of the sequence $1, 0, 0, 0, \ldots$ with the exponential generating function 1. The transformed sequence is the zigzag numbers, with the exponential generating function $\sec x + \tan x$.

```
value "stake 20 (seidel_triangle_rows (1 ## sconst (0::int)))"
value "stake 20 (boustrophedon_stream (1 ## sconst (0::int)))"
```

The following is another example from the paper by Millar et al: the Boustrophedon transform of the sequence $1, 1, 1, \ldots$ with the exponential generating function e^x. The exponential generating function of the transformed sequence is $e^x(\sec x + \tan x)$.

```
value "stake 20 (seidel_triangle_rows (sconst (1::int)))"
value "stake 20 (boustrophedon_stream (sconst (1::int)))"

end
theory Tangent_Secant_Imperative_Test
  imports Tangent_Numbers_Imperative Secant_Numbers_Imperative
begin

definition "tangent_number_imp n =
  do {
    a ← tangent_numbers_imperative.compute_imp (nat_of_integer n);
    xs ← Array.freeze a;
    return (map integer_of_nat xs)
  }"

ML_val ‹@{code tangent_number_imp} 100 ()›


definition "secant_number_imp n =
  do {
    a ← secant_numbers_imperative.compute_imp (nat_of_integer n);
    xs ← Array.freeze a;
    return (map integer_of_nat xs)
  }"

ML_val ‹@{code secant_number_imp} 100 ()›

end
```

References

[1] R. P. Brent and D. Harvey. *Fast Computation of Bernoulli, Tangent and Secant Numbers*, pages 127–142. Springer New York, 2013.

[2] J. Millar, N. Sloane, and N. Young. A new operation on sequences: The boustrophedon transform. *Journal of Combinatorial Theory, Series A*, 76(1):44–54, Oct. 1996.

[3] OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences, 2024. Published electronically at http://oeis.org.