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"
  ⟨proof⟩

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⟩

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⟩

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⟩

end

lemma of_rat_fact [simp]: "of_rat (fact n) = fact n"
  ⟨proof⟩

lemma Pow_conv_subsets_of_size:
  assumes "finite A"
  shows   "Pow A = (⋃k≤card A. {X. X ⊆ A ∧ card X = k})"
  ⟨proof⟩
```

1.2 Linear orders

lemma (**in** *linorder*) *linorder_linear_order [intro]: "linear_order {(x,y). x ≤ y}"*
⟨*proof*⟩

lemma (**in** *linorder*) *less_strict_linear_order_on [intro]: "strict_linear_order_on A {(x,y). x < y}"*
⟨*proof*⟩

lemma (**in** *linorder*) *greater_strict_linear_order_on [intro]: "strict_linear_order_on A {(x,y). x > y}"*
⟨*proof*⟩

lemma *strict_linear_order_on_asym_on:*
assumes *"strict_linear_order_on A R"*
shows *"asym_on A R"*
⟨*proof*⟩

lemma *strict_linear_order_on_antisym_on:*
assumes *"strict_linear_order_on A R"*
shows *"antisym_on A R"*
⟨*proof*⟩

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*⟩

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)"*
⟨*proof*⟩

lemma *sorted_wrt_imp_distinct:*
assumes *"sorted_wrt R xs" "⋀x. x ∈ set xs ⟹ ¬R x x"*
shows *"distinct xs"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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⟩

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⟩

lemma of_nat_poly_pderiv:
"map_poly (of_nat :: nat ⇒ 'a :: {semidom, semiring_char_0}) (pderiv p) =
pderiv (map_poly of_nat p)"
⟨proof⟩

lemma fps_mult_left_numeral_nth [simp]:
"((numeral c :: 'a ::{comm_monoid_add, semiring_1} fps) * f) $ n = numeral c * f $ n"
⟨proof⟩

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

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"
⟨proof⟩

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"
⟨proof⟩

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⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *divide_fps_eqI:*
assumes *"F * G = (H :: 'a :: field fps)" "H ≠ 0 ∨ G ≠ 0 ∨ F = 0"*
shows *"H / G = F"*
⟨*proof*⟩

lemma *fps_to_fls_sum [simp]: "fps_to_fls (∑x∈A. f x) = (∑x∈A. fps_to_fls (f x))"*
⟨*proof*⟩

lemma *fps_to_fls_sum_list [simp]: "fps_to_fls (sum_list fs) = (∑f←fs. fps_to_fls f)"*
⟨*proof*⟩

lemma *fps_to_fls_sum_mset [simp]: "fps_to_fls (sum_mset F) = (∑f∈#F. fps_to_fls f)"*
⟨*proof*⟩

lemma *fps_to_fls_prod [simp]: "fps_to_fls (∏x∈A. f x) = (∏x∈A. fps_to_fls (f x))"*
⟨*proof*⟩

lemma *fps_to_fls_prod_list [simp]: "fps_to_fls (prod_list fs) = (∏f←fs. fps_to_fls f)"*
⟨*proof*⟩

lemma *fps_to_fls_prod_mset [simp]: "fps_to_fls (prod_mset F) = (∏f∈#F. fps_to_fls f)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *fps_sec_nth_0 [simp]: "fps_nth (fps_sec c) 0 = 1"*
⟨*proof*⟩

lemma *fps_sec_square_conv_fps_tan_square:*
"fps_sec c ^ 2 = (1 + fps_tan c ^ 2 :: 'a :: field_char_0 fps)"
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *fps_cos_conv_cosh: "fps_cos c = fps_cosh (i * c)"*
⟨*proof*⟩

lemma *fps_cosh_conv_cos: "fps_cosh c = fps_cos (i * c)"*
⟨*proof*⟩

lemma *fps_cosh_compose_linear [simp]:*
*"fps_cosh (d::'a::field_char_0) oo (fps_const c * fps_X) = fps_cosh (c * d)"*
⟨*proof*⟩

lemma *fps_fps_cosh_compose_minus [simp]:*
"fps_compose (fps_cosh c) (-fps_X) = fps_cosh (-c :: 'a :: field_char_0)"
⟨*proof*⟩

lemma *fps_nth_cosh: "fps_nth (fps_cosh c) n = (if even n then c ^ n / fact n else 0)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *fps_sec_conv_sech: "fps_sec c = fps_sech (i * c)"*
⟨*proof*⟩

lemma *fps_sech_conv_sec: "fps_sech c = fps_sec (i * c)"*
⟨*proof*⟩

lemma *fps_sech_compose_linear [simp]:*
*"fps_sech (d::'a::field_char_0) oo (fps_const c * fps_X) = fps_sech (c * d)"*
⟨*proof*⟩

lemma *fps_fps_sech_compose_minus [simp]:*
"fps_compose (fps_sech c) (-fps_X) = fps_sech (-c :: 'a :: field_char_0)"
⟨*proof*⟩

lemma *fps_tan_deriv': "fps_deriv (fps_tan 1 :: 'a :: field_char_0 fps) = 1 + fps_tan 1 ^ 2"*
⟨*proof*⟩

lemma *fps_tan_nth_0 [simp]: "fps_nth (fps_tan c) 0 = 0"*
⟨*proof*⟩

lemma *fps_nth_sin_even:*
assumes *"even n"*
shows *"fps_nth (fps_sin c) n = 0"*
⟨*proof*⟩

lemma *fps_nth_cos_odd:*
  assumes *"odd n"*
  shows *"fps_nth (fps_cos c) n = 0"*
  ⟨*proof*⟩

lemma *fps_tan_odd: "fps_tan (-c) = -fps_tan c"*
  ⟨*proof*⟩

lemma *fps_sec_even: "fps_sec (-c) = fps_sec c"*
  ⟨*proof*⟩

lemma *fps_sin_compose_linear [simp]: "fps_sin c oo (fps_const c' * fps_X) = fps_sin (c * c')"*
  ⟨*proof*⟩

lemma *fps_sin_compose_uminus [simp]: "fps_sin c oo (-fps_X) = fps_sin (-c)"*
  ⟨*proof*⟩

lemma *fps_cos_compose_linear [simp]: "fps_cos c oo (fps_const c' * fps_X) = fps_cos (c * c')"*
  ⟨*proof*⟩

lemma *fps_cos_compose_uminus [simp]: "fps_cos c oo (-fps_X) = fps_cos (-c)"*
  ⟨*proof*⟩

lemma *fps_tan_compose_linear [simp]: "fps_tan c oo (fps_const c' * fps_X) = fps_tan (c * c')"*
  ⟨*proof*⟩

lemma *fps_tan_compose_uminus [simp]: "fps_tan c oo (-fps_X) = fps_tan (-c)"*
  ⟨*proof*⟩

lemma *fps_sec_compose_linear [simp]: "fps_sec c oo (fps_const c' * fps_X) = fps_sec (c * c')"*
  ⟨*proof*⟩

lemma *fps_sec_compose_uminus [simp]: "fps_sec c oo (-fps_X) = fps_sec (-c)"*
  ⟨*proof*⟩

lemma *fps_nth_tan_even:*
  assumes *"even n"*
  shows *"fps_nth (fps_tan c) n = 0"*
⟨*proof*⟩

lemma *fps_nth_sec_odd:*

```
  assumes "odd n"
  shows   "fps_nth (fps_sec c) n = 0"
⟨proof⟩

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)"
  ⟨proof⟩

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'))"
  ⟨proof⟩
```

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"
⟨*proof*⟩

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*⟩

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*⟩

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)"*
⟨*proof*⟩

lemma *alternating_permutations_of_set_code [code]:*
"alternating_permutations_of_set R A = Set.filter (alternating_list R) (permutations_of_set A)"
⟨*proof*⟩

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 {} = {[]}"
  ⟨proof⟩

lemma alternating_permutations_of_set_singleton [simp]:
  "alternating_permutations_of_set R {x} = {[x]}"
  ⟨proof⟩

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⟩

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⟩

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⟩

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⟩

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"*
  ⟨proof⟩

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⟩

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 $0 \leq k \leq n$.
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⟩
```

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⟩
```

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⟩

corollary zigzag_number_conv_fps_tan:
  assumes "odd n"
  shows   "real (zigzag_number n) = fps_nth (fps_tan 1) n * fact n"
⟨proof⟩
```

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 {})"
  ⟨*proof*⟩

lemma *alternating_permutations_of_set_with_hd_outside:*
  assumes *"x ∉ A"*
  shows *"alternating_permutations_of_set_with_hd R A x = {}"*
⟨*proof*⟩

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*⟩

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*⟩

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"
  ⟨*proof*⟩

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*⟩

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⟩
```

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"
  ⟨proof⟩

lemma entringer_number_0_right [simp]:
  assumes "n > 0"
  shows   "entringer_number n 0 = 0"
⟨proof⟩

lemma entringer_number_greater_eq_0 [simp]:
  assumes "k > n"
  shows   "entringer_number n k = 0"
⟨proof⟩

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⟩
```

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⟩
```

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*⟩

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*⟩

lemma *sum_entringer_numbers':*
assumes *k: "k ≤ n"*
shows *"(∑i≤k. entringer_number n (n - i)) = entringer_number (Suc n) (Suc k)"*
⟨*proof*⟩

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*⟩

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*⟩

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))"
```

⟨proof⟩

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}"*
⟨*proof*⟩

lemma *Inc_Trees_infinite_eq_empty [simp]:*
assumes *"¬finite A"*
shows *"Inc_Trees A = {}"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *set_tree_mirror_root [simp]: "set_tree (mirror_root t) = set_tree t"*
⟨*proof*⟩

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}"*
⟨*proof*⟩

lemma *Inc_Trees'_infinite_eq_empty [simp]:*
assumes *"¬finite A"*
shows *"Inc_Trees' A = {}"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *card_Inc_Trees' [simp]: "card (Inc_Trees' A) = card (Inc_Trees A)"*

⟨*proof*⟩

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⟩
```

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⟩

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⟩

lemma Diff_right_commute: "A - B - C = A - C - (B :: 'a set)"
  ⟨proof⟩
```

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⟩

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}))"
  ⟨proof⟩
```

```
lemma finite_Inc_Trees [intro]: "finite (Inc_Trees A)"
  and finite_Inc_Trees' [intro]: "finite (Inc_Trees' A)"
⟨proof⟩
```

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⟩
```

By induction, our main result follows:

```
theorem card_Inc_Trees:
  assumes "finite A"
  shows   "card (Inc_Trees A) = zigzag_number (card A)"
  ⟨proof⟩

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"
  ⟨proof⟩

lemma tangent_poly_altdef [code]:
  "tangent_poly n = ((λp. pderiv p * [:1,0,1:]) ^^ n) [:0, 1:]"
  ⟨proof⟩

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⟩

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))"
  ⟨proof⟩
```

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)"
```

⟨proof⟩

lemma *tangent_number_aux_0_left' [simp]:*
"j ≠ 1 ⟹ tangent_number_aux 0 j = 0"
"tangent_number_aux 0 (Suc 0) = 1"
⟨proof⟩

lemma *tangent_number_aux_0_right:*
"tangent_number_aux (Suc n) 0 = poly.coeff (tangent_poly n) 1"
⟨proof⟩

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)"*
⟨proof⟩

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)"*
⟨proof⟩

lemma *tangent_number_aux_odd_eq_0: "even (n + j) ⟹ tangent_number_aux n j = 0"*
⟨proof⟩

lemma *tangent_number_aux_eq_0 [simp]: "j > n + 1 ⟹ tangent_number_aux n j = 0"*
⟨proof⟩

lemma *tangent_number_aux_last [simp]: "tangent_number_aux n (Suc n) = fact n"*
⟨proof⟩

lemma *tangent_number_aux_last': "Suc m = n ⟹ tangent_number_aux m n = fact m"*
⟨proof⟩

lemma *tangent_number_aux_1_right [simp]:*
"tangent_number_aux i (Suc 0) = tangent_number_aux (i + 1) 0"
⟨proof⟩

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)"
```

  ⟨*proof*⟩

lemma *tangent_number_0 [simp]: "tangent_number 0 = 0"*

  ⟨*proof*⟩

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*⟩

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)"
```

  ⟨*proof*⟩

lemma *tangent_number_conv_aux [code]:*

```
  "tangent_number n = tangent_number_aux (2*n - Suc 0) 0"
```

  ⟨*proof*⟩

lemma *tangent_number_1 [simp]: "tangent_number (Suc 0) = 1"*

  ⟨*proof*⟩

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*⟩

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))"*
  ⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *[simp del] = pochhammer_row_impl.simps*

lemma *pochhammer_rec'': "k > 0 ⟹ pochhammer n k = n * pochhammer (n+1) (k-1)"*
  ⟨*proof*⟩

lemma *pochhammer_row_impl_correct:*
  *"pochhammer_row_impl k n x = map (λi. x * pochhammer k i) [0..<n-k]"*
⟨*proof*⟩

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"*
  ⟨*proof*⟩

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_au 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"*
  ⟨*proof*⟩

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]"
  ⟨proof⟩
```

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⟩
```

lemma *tangent_number_code [code]:*

```
  "tangent_number n = (if n = 0 then 0 else last (tangent_numbers n))"
  ⟨proof⟩
```

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])"
  ⟨*proof*⟩

lemma *init_loop_correct [refine_vcg]: "init_loop ≤ SPEC (λxs. xs = map fact [0..<n])"*
  ⟨*proof*⟩

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)"*
  ⟨*proof*⟩

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])"
  ⟨*proof*⟩

lemma *compute_correct [refine_vcg]: "compute ≤ SPEC (λxs'. xs' = tangent_numbers n)"*
  ⟨*proof*⟩

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 $\rightarrow_a$ array_assn nat_assn"
  ⟨*proof*⟩

lemma *imp_correct':*
  "(compute_imp, λn. RETURN (tangent_numbers n)) ∈ nat_assn^d $\rightarrow_a$ array_assn nat_assn"
⟨*proof*⟩

theorem *imp_correct:*
```
  "<nat_assn n n> compute_imp n <array_assn nat_assn (tangent_numbers
n)>t"
```
⟨*proof*⟩

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*⟩

lemma *secant_poly_altdef [code]:*
```
  "secant_poly n = ((λp. p * [:0,1:] + pderiv p * [:1, 0, 1:]) ^^ n) 1"
```
⟨*proof*⟩

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⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

lemma *secant_number_aux_0_left' [simp]:*
```
  "j ≠ 0 ⟹ secant_number_aux 0 j = 0"
  "secant_number_aux 0 0 = 1"
```
⟨proof⟩

lemma *secant_number_aux_0_right:*
```
  "secant_number_aux (Suc n) 0 = secant_number_aux n 1"
```
⟨proof⟩

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)"
```
⟨proof⟩

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)"
```
⟨proof⟩

lemma *secant_number_aux_odd_eq_0: "odd (n + j) ⟹ secant_number_aux n j = 0"*
⟨*proof*⟩

lemma *secant_number_aux_eq_0 [simp]: "j > n ⟹ secant_number_aux n j = 0"*
⟨*proof*⟩

lemma *secant_number_aux_last [simp]: "secant_number_aux n n = fact n"*
⟨*proof*⟩

lemma *secant_number_aux_last': "m = n ⟹ secant_number_aux m n = fact m"*
⟨*proof*⟩

lemma *secant_number_aux_1_right [simp]:*
"secant_number_aux i (Suc 0) = secant_number_aux (i + 1) 0"
⟨*proof*⟩

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)"*
⟨*proof*⟩

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))"
⟨*proof*⟩

lemma *secant_number_0 [simp]: "secant_number 0 = 1"*
⟨*proof*⟩

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*⟩

lemma *fps_nth_sec:*
*"fps_nth (fps_sec (1::'a :: field_char_0)) (2*n) = of_int (secant_number*

```
n) / fact (2*n)"
  ⟨proof⟩

lemma secant_number_conv_aux [code]:
  "secant_number n = secant_number_aux (2*n) 0"
  ⟨proof⟩

lemma secant_number_1 [simp]: "secant_number 1 = 1"
  ⟨proof⟩
```

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⟩
```

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"
  ⟨proof⟩

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"
  ⟨proof⟩

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]"
  ⟨proof⟩

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⟩

lemma secant_number_code [code]: "secant_number n = last (secant_numbers
n)"
  ⟨proof⟩

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⟩

lemma zigzag_numbers_code [code]:
  "zigzag_numbers n = splice (secant_numbers (n div 2)) (tangent_numbers
((n+1) div 2))"
⟨proof⟩

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])"
  ⟨proof⟩
```

```
lemma init_loop_correct [refine_vcg]: "init_loop ≤ SPEC (λxs. xs = map
fact [0..<n+1])"
  ⟨proof⟩
```

```
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)"
  ⟨proof⟩

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])"
  ⟨proof⟩

lemma compute_correct [refine_vcg]: "compute ≤ SPEC (λxs'. xs' = secant_numbers
n)"
  ⟨proof⟩

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"
  ⟨proof⟩

lemma imp_correct':
  "(compute_imp, λn. RETURN (secant_numbers n)) ∈ nat_assn^d →_a array_assn
nat_assn"
⟨proof⟩

theorem imp_correct:
   "<nat_assn n n> compute_imp n <array_assn nat_assn (secant_numbers
n)>_t"
⟨proof⟩

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"
  ⟨proof⟩

lemma secant_number_conv_euler_number: "secant_number n = (-1) ^ n *
euler_number (2 * n)"
  ⟨proof⟩

lemma euler_number_odd_eq_0: "odd n ⟹ euler_number n = 0"
  ⟨proof⟩

lemma euler_number_odd_numeral [simp]: "euler_number (numeral (Num.Bit1
n)) = 0"
  ⟨proof⟩

lemma euler_number_Suc_0 [simp]: "euler_number (Suc 0) = 0"
  ⟨proof⟩

lemma euler_number_0 [simp]: "euler_number 0 = 1"
  and euler_number_2 [simp]: "euler_number 2 = -1"
  ⟨proof⟩

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⟩

lemma exponential_generating_function_euler_numbers:
  "Abs_fps (λn. of_int (euler_number n) / fact n :: 'a :: field_char_0)
= fps_sech 1"
```

⟨proof⟩

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⟩
```

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⟩

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))"
  ⟨proof⟩

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⟩
```

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*⟩

lemma *degree_Euler_poly [simp]: "degree (Euler_poly n) = n"*
⟨*proof*⟩

lemma *poly_Euler_poly [simp]: "poly (Euler_poly n) = euler_poly n"*
⟨*proof*⟩

lemma *euler_poly_onehalf:*
"euler_poly n (1 / 2) = (of_int (euler_number n) / 2 ^ n :: 'a :: field_char_0)"
⟨*proof*⟩

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:]"*
⟨*proof*⟩

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*⟩

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))"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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*⟩

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*⟩

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))"*
  ⟨*proof*⟩

lemma *degree_Bernpoly [simp]: "degree (Bernpoly n) = n"*
⟨*proof*⟩

lemma *lead_coeff_Bernpoly [simp]: "poly.coeff (Bernpoly n) n = 1"*
  ⟨*proof*⟩

lemma *poly_Bernpoly [simp]: "poly (Bernpoly n) x = bernpoly n x"*
⟨*proof*⟩

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⟩

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⟩

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⟩

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⟩



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⟩

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⟩

lemma euler_poly_1_even:
  assumes "even n" "n > 1"
  shows   "euler_poly n 1 = 0"
⟨proof⟩
```

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⟩
```

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⟩

theorem euler_poly_reflect:
  "euler_poly n (1 - x) = (-1) ^ n * euler_poly n x"
⟨proof⟩

theorem euler_poly_forward_sum: "euler_poly n x + euler_poly n (x + 1)
= 2 * x ^ n"
⟨proof⟩

lemma euler_poly_plus1: "euler_poly n (x + 1) = -euler_poly n x + 2 *
x ^ n"
  ⟨proof⟩

lemma euler_poly_minus:
  "(-1) ^ n * euler_poly n (-x) = -euler_poly n x + 2 * x ^ n"
  ⟨proof⟩
```

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⟩
```

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*⟩

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*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *nth_binomial_row [simp]: "k ≤ n ⟹ binomial_row n ! k = of_nat (n choose k)"*
⟨*proof*⟩

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)"
```

⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *Bernpolys_aux_correct:*

```
  "Bernpolys_aux (binomial_row (Suc n)) (map Bernpoly (rev [0..<n])) m
= map Bernpoly (rev [0..<m+n])"
```

⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *Bernpolys_correct: "Bernpolys n = map Bernpoly [0..<n]"*
⟨*proof*⟩

```
lemma Bernpoly_code [code]: "Bernpoly n = hd (Bernpolys_aux [1, 1] []
(Suc n))"
  ⟨proof⟩


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"
  ⟨proof⟩

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)"
  ⟨proof⟩

lemma bernpoly_code [code]:
  "bernpoly n x = hd (bernpoly_aux [1, 1] [] (Suc n) x)"
⟨proof⟩
```

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"
  ⟨proof⟩

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⟩
```

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"
  ⟨proof⟩

lemma Euler_polys_correct: "Euler_polys n = map Euler_poly [0..<n]"
  ⟨proof⟩

lemma Euler_poly_code [code]: "Euler_poly n = hd (Euler_polys_aux [1]
[] (Suc n))"
  ⟨proof⟩


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"
  ⟨proof⟩

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)"
  ⟨proof⟩

lemma euler_poly_code [code]:
  "euler_poly n x = hd (euler_poly_aux [1] [] (Suc n) x)"
⟨proof⟩

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"
  ⟨proof⟩
```

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))"
  ⟨proof⟩
```

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*⟩

lemma *seidel_triangle_aux_same2 [simp]: "seidel_triangle_aux n k n = 1"*
  ⟨*proof*⟩

lemma *seidel_triangle_aux_0_middle [simp]:*
  "i < n ⟹ seidel_triangle_aux n 0 i = 0"
  ⟨*proof*⟩

lemma *seidel_triangle_aux_0_right [simp]:*
  assumes *"k ≤ n"*
  shows *"seidel_triangle_aux n k 0 = entringer_number n k"*
⟨*proof*⟩

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*⟩

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)"*
  ⟨*proof*⟩

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)))"*
  ⟨*proof*⟩

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)))"*
  ⟨*proof*⟩

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*⟩

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)"
  ⟨*proof*⟩

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))"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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⟩

lemma inv_Boustrophedon_Boustrophedon [simp]:
  fixes A :: "'a :: field_char_0 fps"
  shows "inv_Boustrophedon (Boustrophedon A) = A"
⟨proof⟩
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"
  ⟨proof⟩

lemma length_seidel_triangle_rows_step [simp]:
  "length (seidel_triangle_rows_step a xs) = Suc (length xs)"
  ⟨proof⟩

lemma nth_seidel_triangle_rows_step:
  "i ≤ length xs ⟹ seidel_triangle_rows_step a xs ! i = a + sum_list
(take i xs)"
```

⟨proof⟩

```
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⟩

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⟩

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]"
```
⟨proof⟩

```
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)"
  ⟨proof⟩

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)"
  ⟨proof⟩
```

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"
  ⟨proof⟩
```

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"
  ⟨proof⟩

lemma seidel_triangle_code [code]:
  "seidel_triangle a n k = (if k > n then 0 else seidel_triangle_impl_aux
a [] 0 n k)"
  ⟨proof⟩

lemma entringer_number_code [code]:
  "entringer_number n k = seidel_triangle (λn. if n = 0 then 1 else 0)
n k"
  ⟨proof⟩

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*⟩

```
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*⟩

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.