Chebyshev Polynomials

Manuel Eberl

November 21, 2023

Abstract

The multiple-angle formulas for cos and sin state that for any natural number n, the values of $\cos nx$ and $\sin nx$ can be expressed in terms of $\cos x$ and $\sin x$. To be more precise, there are polynomials T_n and U_n such that $\cos nx = T_n(\cos x)$ and $\sin nx = U_n(\cos x) \sin x$. These are called the *Chebyshev polynomials of the first and second kind*, respectively.

This entry contains a definition of these two familes of polynomials in Isabelle/HOL along with some of their most important properties. In particular, it is shown that T_n and U_n are *orthogonal* families of polynomials.

Moreover, we show the well-known result that for any monic polynomial p of degree $n > 0$, it holds that $\sup_{x\in[-1,1]} |p(x)| \geq 2^{n-1}$, and that this inequality is sharp since equality holds with $p = 2^{1-n}T_n$. This has important consequences in the theory of function interpolation, since it implies that the roots of T_n (also colled the *Chebyshev nodes*) are exceptionally well-suited as interpolation nodes.

Contents

1 Parametricity of polynomial operations

```
theory Polynomial_Transfer
  imports "HOL-Computational_Algebra.Polynomial"
begin



definition rel_poly :: "('a ⇒ 'b ⇒ bool) ⇒ 'a :: zero poly ⇒ 'b ::
zero poly ⇒ bool" where
  "rel_poly R p q ⟷ rel_fun (=) R (coeff p) (coeff q)"

lemma left_unique_rel_poly [transfer_rule]: "left_unique R ⟹ left_unique
(rel_poly R)"
  unfolding left_unique_def rel_poly_def poly_eq_iff rel_fun_def by meson

lemma right_unique_rel_poly [transfer_rule]: "right_unique R ⟹ right_unique
(rel_poly R)"
  unfolding right_unique_def rel_poly_def poly_eq_iff rel_fun_def by meson

lemma bi_unique_rel_poly [transfer_rule]: "bi_unique R ⟹ bi_unique
(rel_poly R)"
  unfolding bi_unique_def rel_poly_def poly_eq_iff rel_fun_def by meson

lemma rel_poly_swap: "rel_poly R x y ⟷ rel_poly (λy x. R x y) y x"
  by (auto simp: rel_poly_def rel_fun_def)

lemma coeff_transfer [transfer_rule]:
  "rel_fun (rel_poly R) (rel_fun (=) R) coeff coeff"
  by (auto simp: rel_fun_def rel_poly_def)

lemma map_poly_transfer:
  assumes "rel_fun R S f g" "f 0 = 0" "g 0 = 0"
  shows   "rel_fun (rel_poly R) (rel_poly S) (map_poly f) (map_poly g)"
  using assms by (auto simp: rel_fun_def rel_poly_def coeff_map_poly)

lemma map_poly_transfer':
  assumes "rel_fun R S f g" "rel_poly R p q" "f 0 = 0" "g 0 = 0"
  shows   "rel_poly S (map_poly f p) (map_poly g q)"
  using assms by (auto simp: rel_fun_def rel_poly_def coeff_map_poly)

lemma rel_poly_id: "p = q ⟹ rel_poly (=) p q"
  by (auto simp: rel_poly_def)

lemma left_total_rel_poly [transfer_rule]:
  assumes "left_total R" "right_unique R" "R 0 0"
  shows   "left_total (rel_poly R)"
  unfolding left_total_def
proof
```

```
  fix p :: "'a poly"
  from assms have "∀x. ∃y. R x y"
    unfolding left_total_def by blast
  then obtain f where f: "R x (f x)" for x
    by metis
  have [simp]: "f 0 = 0"
    using assms f[of 0] by (auto dest: right_uniqueD)
  have "rel_poly R (map_poly (λx. x) p) (map_poly f p)"
    by (rule map_poly_transfer'[of "(=)"] rel_funI)+ (auto intro: rel_poly_id
f)
  thus "∃q. rel_poly R p q"
    by force
qed

lemma right_total_rel_poly [transfer_rule]:
  assumes "right_total R" "left_unique R" "R 0 0"
  shows   "right_total (rel_poly R)"
  using left_total_rel_poly[of "λx y. R y x"] assms
  by (metis left_totalE left_totalI left_unique_iff rel_poly_swap right_total_def
right_unique_iff)

lemma bi_total_rel_poly [transfer_rule]:
  assumes "bi_total R" "bi_unique R" "R 0 0"
  shows   "bi_total (rel_poly R)"
  using left_total_rel_poly[of R] right_total_rel_poly[of R] assms
  by (simp add: bi_total_alt_def bi_unique_alt_def)

lemma zero_poly_transfer [transfer_rule]: "R 0 0 ⟹ rel_poly R 0 0"
  by (auto simp: rel_fun_def rel_poly_def)

lemma one_poly_transfer [transfer_rule]: "R 0 0 ⟹ R 1 1 ⟹ rel_poly
R 1 1"
  by (auto simp: rel_fun_def rel_poly_def)

lemma pCons_transfer [transfer_rule]:
  "rel_fun R (rel_fun (rel_poly R) (rel_poly R)) pCons pCons"
  by (auto simp: rel_fun_def rel_poly_def coeff_pCons split: nat.splits)

lemma plus_poly_transfer [transfer_rule]:
  "rel_fun R (rel_fun R R) (+) (+) ⟹
   rel_fun (rel_poly R) (rel_fun (rel_poly R) (rel_poly R)) (+) (+)"
  by (auto simp: rel_fun_def rel_poly_def)

lemma minus_poly_transfer [transfer_rule]:
  "rel_fun R (rel_fun R R) (-) (-) ⟹
   rel_fun (rel_poly R) (rel_fun (rel_poly R) (rel_poly R)) (-) (-)"
  by (auto simp: rel_fun_def rel_poly_def)

lemma uminus_poly_transfer [transfer_rule]:
```

```
  "rel_fun R R uminus uminus ⟹ rel_fun (rel_poly R) (rel_poly R) uminus
uminus"
  by (auto simp: rel_fun_def rel_poly_def)

lemma smult_transfer [transfer_rule]:
  "rel_fun R (rel_fun R R) (*) (*) ⟹
   rel_fun R (rel_fun (rel_poly R) (rel_poly R)) smult smult"
  by (auto simp: rel_fun_def rel_poly_def)

lemma monom_transfer [transfer_rule]:
  "R 0 0 ⟹ rel_fun R (rel_fun (=) (rel_poly R)) monom monom"
  by (auto simp: rel_fun_def rel_poly_def)

lemma pderiv_transfer [transfer_rule]:
  assumes "R 0 0" "rel_fun R (rel_fun R R) (+) (+)"
  shows "rel_fun (rel_poly R) (rel_poly R) pderiv pderiv"
proof (rule rel_funI, goal_cases)
  case (1 p q)
  define f :: "nat ⇒ 'a ⇒ 'a" where
    "f = (λn p. of_nat n * p)"
  define g :: "nat ⇒ 'b ⇒ 'b" where
    "g = (λn p. of_nat n * p)"
  have plus: "R (x + y) (x' + y')" if "R x x'" "R y y'" for x x' y y'
    using assms(2) that by (auto simp: rel_fun_def)
  have fg: "R (f m x) (g n y)" if "m = n" "R x y" for x y m n
    unfolding that(1)
    by (induction n) (auto simp: f_def g_def ring_distribs intro!: assms(1)
plus that)
  have "rel_fun (=) R (λn. f (Suc n) (coeff p (Suc n))) (λn. g (Suc n)
(coeff q (Suc n)))"
    using 1 by (intro rel_funI fg) (auto simp: rel_poly_def rel_fun_def)
  thus ?case
    by (auto simp: rel_poly_def coeff_pderiv [abs_def] f_def g_def)
qed

lemma If_transfer':
  assumes "P = P'" "P ⟹ R x x'" "¬P ⟹ R y y'"
  shows   "R (if P then x else y) (if P' then x' else y')"
  using assms by auto

lemma nth_transfer:
  assumes "list_all2 R xs ys" "i = j" "i < length xs"
  shows   "R (xs ! i) (ys ! j)"
  using assms by (simp add: list_all2_nthD)

lemma Poly_transfer [transfer_rule]:
  assumes [transfer_rule]: "R 0 0" "bi_unique R"
  shows   "rel_fun (list_all2 R) (rel_poly R) Poly Poly"
  unfolding rel_poly_def
```

```
proof (intro rel_funI, goal_cases)
  case [transfer_rule]: (1 p q i j)
  show "R (coeff (Poly p) i) (coeff (Poly q) j)"
    unfolding coeff_Poly_eq nth_default_def
  proof (rule If_transfer')
    show "(i < length p) = (j < length q)"
      by transfer_prover
    show "R (p ! i) (q ! j)" if "i < length p"
      by (rule nth_transfer) (use 1 that in auto)
  qed (use assms in auto)
qed

lemma poly_of_list_transfer [transfer_rule]:
  assumes [transfer_rule]: "R 0 0" "bi_unique R"
  shows   "rel_fun (list_all2 R) (rel_poly R) poly_of_list poly_of_list"
  unfolding poly_of_list_def by transfer_prover

lemma degree_transfer [transfer_rule]:
  assumes [transfer_rule]: "R 0 0" "bi_unique R"
  shows   "rel_fun (rel_poly R) (=) degree degree"
proof
  fix p q
  assume *: "rel_poly R p q"
  with assms have "coeff p i = 0 ⟷ coeff q i = 0" for i
    unfolding rel_poly_def rel_fun_def bi_unique_def by metis
  thus "degree p = degree q"
    using antisym degree_le coeff_eq_0 by metis
qed

lemma coeffs_transfer [transfer_rule]:
  assumes [transfer_rule]: "R 0 0" "bi_unique R"
  shows "rel_fun (rel_poly R) (list_all2 R) coeffs coeffs"
proof
  fix p q
  assume [transfer_rule]: "rel_poly R p q"
  have "degree p = degree q"
    by transfer_prover
  show "list_all2 R (coeffs p) (coeffs q)"
    unfolding coeffs_def by transfer_prover
qed

lemma times_poly_transfer [transfer_rule]:
  assumes [transfer_rule]: "rel_fun R (rel_fun R R) (+) (+)"
                           "rel_fun R (rel_fun R R) (*) (*)" "R 0 0" "bi_unique
R"
  shows "rel_fun (rel_poly R) (rel_fun (rel_poly R) (rel_poly R)) (*)
(*)"
  unfolding times_poly_def fold_coeffs_def by transfer_prover
```

```
lemma dvd_poly_transfer [transfer_rule]:
  assumes [transfer_rule]: "rel_fun R (rel_fun R R) (+) (+)"
                           "rel_fun R (rel_fun R R) (*) (*)" "R 0 0" "bi_unique
R" "bi_total R"
  shows "rel_fun (rel_poly R) (rel_fun (rel_poly R) (=)) (dvd) (dvd)"
  unfolding dvd_def by transfer_prover

lemma poly_transfer [transfer_rule]:
  assumes [transfer_rule]: "rel_fun R (rel_fun R R) (+) (+)"
                           "rel_fun R (rel_fun R R) (*) (*)" "R 0 0" "bi_unique
R"
  shows "rel_fun (rel_poly R) (rel_fun R R) poly poly"
  unfolding poly_def horner_sum_foldr by transfer_prover

lemma pcompose_transfer [transfer_rule]:
  assumes [transfer_rule]: "rel_fun R (rel_fun R R) (+) (+)"
                           "rel_fun R (rel_fun R R) (*) (*)" "R 0 0" "bi_unique
R"
  shows "rel_fun (rel_poly R) (rel_fun (rel_poly R) (rel_poly R)) pcompose
pcompose"
  unfolding pcompose_def fold_coeffs_def by transfer_prover

lemma order_0_right: "order x 0 = Least (λ_. False)"
  unfolding order_def by simp


lemma order_poly_transfer [transfer_rule]:
  assumes [transfer_rule]:
    "rel_fun R (rel_fun R R) (+) (+)" "rel_fun R (rel_fun R R) (*) (*)"
    "rel_fun R R uminus uminus"
    "R 0 0" "R 1 1" "bi_unique R" "bi_total R" "R x y" "rel_poly R p q"
  shows "order x p = order y q"
  unfolding order_def by transfer_prover

end
```

2 Missing Library Material

```
theory Chebyshev_Polynomials_Library
  imports "HOL-Computational_Algebra.Polynomial" "HOL-Library.FuncSet"
begin
```

2.1 Miscellaneous

```
lemma bij_betw_Collect:
  assumes "bij_betw f A B" "⋀x. x ∈ A ⟹ Q (f x) ⟷ P x"
  shows   "bij_betw f {x∈A. P x} {y∈B. Q y}"
  using assms by (auto simp add: bij_betw_def inj_on_def)
```

lemma *induct_nat_012[case_names 0 1 ge2]:*
  "P 0 ⟹ P (Suc 0) ⟹ (⋀n. P n ⟹ P (Suc n) ⟹ P (Suc (Suc n))) ⟹ P n"
proof *(induction_schema, pat_completeness)*
  show *"wf (Wellfounded.measure id)"*
    by *simp*
qed *auto*

2.2 Lists

lemma *distinct_adj_conv_length_remdups_adj:*
  "distinct_adj xs ⟷ length (remdups_adj xs) = length xs"
proof *(induction xs rule: remdups_adj.induct)*
  case *(3 x y xs)*
  thus *?case*
    using *remdups_adj_length[of "y # xs"]* **by** *auto*
qed *auto*

lemma *successively_conv_nth:*
  "successively P xs ⟷ (∀i. Suc i < length xs ⟶ P (xs ! i) (xs ! Suc i))"
  by *(induction P xs rule: successively.induct)*
    (force simp: nth_Cons split: nat.splits)+

lemma *successively_nth: "successively P xs ⟹ Suc i < length xs ⟹ P (xs ! i) (xs ! Suc i)"*
  unfolding *successively_conv_nth* **by** *blast*

lemma *distinct_adj_conv_nth:*
  "distinct_adj xs ⟷ (∀i. Suc i < length xs ⟶ xs ! i ≠ xs ! Suc i)"
  by *(simp add: distinct_adj_def successively_conv_nth)*

lemma *distinct_adj_nth: "distinct_adj xs ⟹ Suc i < length xs ⟹ xs ! i ≠ xs ! Suc i"*
  unfolding *distinct_adj_conv_nth* **by** *blast*

The following two lemmas give a full characterisation of the *filter* function: The list *filter P xs* is the only list *ys* for which there exists a strictly increasing function $f : \{0, \ldots, |\text{ys}| - 1\} \to \{0, \ldots, |\text{xs}| - 1\}$ such that:

- $\text{ys}_i = \text{xs}_{f(i)}$
- $P(\text{xs}_i) \longleftrightarrow \exists j{<}n.\ f(j) = i$, i.e. the range of f are precisely the indices of the elements of *xs* that satisfy *P*.

lemma *filterE:*
  fixes *P :: "'a ⇒ bool"* **and** *xs :: "'a list"*

```
  assumes "length (filter P xs) = n"
  obtains f :: "nat ⇒ nat" where
    "strict_mono_on {..<n} f"
    "⋀i. i < n ⟹ f i < length xs"
    "⋀i. i < n ⟹ filter P xs ! i = xs ! f i"
    "⋀i. i < length xs ⟹ P (xs ! i) ⟷ (∃j. j < n ∧ f j = i)"
  using assms(1)
proof (induction xs arbitrary: n thesis)
  case Nil
  thus ?case
    using that[of "λ_. 0"] by auto
next
  case (Cons x xs)
  define n' where "n' = (if P x then n - 1 else n)"
  obtain f :: "nat ⇒ nat" where f:
    "strict_mono_on {..<n'} f"
    "⋀i. i < n'⟹ f i < length xs"
    "⋀i. i < n' ⟹ filter P xs ! i = xs ! f i"
    "⋀i. i < length xs ⟹ P (xs ! i) ⟷ (∃j. j < n' ∧ f j = i)"
  proof (rule Cons.IH[where n = n'])
    show "length (filter P xs) = n'"
      using Cons.prems(2) by (auto simp: n'_def)
  qed auto
  define f' where "f' = (λi. if P x then case i of 0 ⇒ 0 | Suc j ⇒ Suc
(f j) else Suc (f i))"

  show ?case
  proof (rule Cons.prems(1))
    show "strict_mono_on {..<n} f'"
      by (auto simp: f'_def strict_mono_on_def n'_def strict_mono_onD[OF
f(1)] split: nat.splits)
    show "f' i < length (x # xs)" if "i < n" for i
      using that f(2) by (auto simp: f'_def n'_def split: nat.splits)
    show "filter P (x # xs) ! i = (x # xs) ! f' i" if "i < n" for i
      using that f(3) by (auto simp: f'_def n'_def split: nat.splits)
    show "P ((x # xs) ! i) ⟷ (∃j<n. f' j = i)" if "i < length (x #
xs)" for i
    proof (cases i)
      case [simp]: 0
      show ?thesis using that Cons.prems(2)
        by (auto simp: f'_def intro!: exI[of _ 0])
    next
      case [simp]: (Suc i')
      have "P ((x # xs) ! i) ⟷ P (xs ! i')"
        by simp
      also have "... ⟷ (∃j<n'. f j = i')"
        using that by (subst f(4)) simp_all
      also have "... ⟷ {j∈{..<n'}. f j = i'} ≠ {}"
        by blast
```

```
      also have "bij_betw (λj. if P x then j+1 else j) {j∈{..<n'}. f j
= i'} {j∈{..<n}. f' j = i}"
      proof (intro bij_betwI[of _ _ _ "λj. if P x then j-1 else j"], goal_cases)
        case 2
        have "(if P x then j - 1 else j) < n'"
          if "j < n" "f' j = i" for j
          using that by (auto simp: n'_def f'_def split: nat.splits)
        moreover have "f (if P x then j - 1 else j) = i'" if "j < n" "f'
j = i" for j
          using that by (auto simp: n'_def f'_def split: nat.splits if_splits)
        ultimately show ?case by auto
      qed (auto simp: n'_def f'_def split: nat.splits)
      hence "{j∈{..<n'}. f j = i'} ≠ {} ⟷ {j∈{..<n}. f' j = i} ≠ {}"
        unfolding bij_betw_def by blast
      also have "... ⟷ (∃j<n. f' j = i)"
        by auto
      finally show ?thesis .
    qed
  qed
qed
```

The following lemma shows the uniqueness of the above property. It is very useful for finding a "closed form" for `filter P xs` in some concrete situation. For example, if we know that exactly every other element of `xs` satisfies `P`, we can use it to prove that `filter P xs = map ((*) 2) [0..<length xs div 2]`

```
lemma filter_eqI:
  fixes f :: "nat ⇒ nat" and xs ys :: "'a list"
  defines "n ≡ length ys"
  assumes "strict_mono_on {..<n} f"
  assumes "⋀i. i < n ⟹ f i < length xs"
  assumes "⋀i. i < n ⟹ ys ! i = xs ! f i"
  assumes "⋀i. i < length xs ⟹ P (xs ! i) ⟷ (∃j. j < n ∧ f j = i)"
  shows   "filter P xs = ys"
  using assms(2-) unfolding n_def
proof (induction xs arbitrary: ys f)
  case Nil
  thus ?case by auto
next
  case (Cons x xs ys f)
  show ?case
  proof (cases "P x")
    case False
    have "filter P xs = ys"
    proof (rule Cons.IH)
      have pos: "f i > 0" if "i < length ys" for i
        using Cons.prems(4)[of "f i"] Cons.prems(2,3)[of i] that False
        by (auto intro!: Nat.gr0I)
      show "strict_mono_on {..<length ys} ((λx. x - 1) ∘ f)"
```

```
      proof (intro strict_mono_onI)
        fix i j assume ij: "i ∈ {..<length ys}" "j ∈ {..<length ys}"
"i < j"
        thus "((λx. x - 1) ∘ f) i < ((λx. x - 1) ∘ f) j"
          using Cons.prems(1) pos[of i] pos[of j]
          by (auto simp: strict_mono_on_def diff_less_mono)
      qed
      show "((λx. x - 1) ∘ f) i < length xs" if "i < length ys" for i
        using Cons.prems(2)[of i] pos[of i] that by auto
      show "ys ! i = xs ! ((λx. x - 1) ∘ f) i" if "i < length ys" for
i
        using Cons.prems(3)[of i] pos[of i] that by auto
      show "P (xs ! i) ⟷ (∃j<length ys. ((λx. x - 1) ∘ f) j = i)"
if "i < length xs" for i
        using Cons.prems(4)[of "Suc i"] that pos by (auto split: if_splits)
    qed
    thus ?thesis
      using False by simp
  next
    case True
    have "ys ≠ []"
      using Cons.prems(4)[of 0] True by auto
    have [simp]: "f 0 = 0"
    proof -
      obtain j where "j < length ys" "f j = 0"
        using Cons.prems(4)[of 0] True by auto
      with strict_mono_onD[OF Cons.prems(1)] have "j = 0"
        by (metis gr_implies_not_zero lessThan_iff less_antisym zero_less_Suc)
      with ‹f j = 0› show ?thesis
        by simp
    qed
    have pos: "f j > 0" if "j > 0" "j < length ys" for j
      using strict_mono_onD[OF Cons.prems(1), of 0 j] that ‹ys ≠ []› by
auto
    have f_eq_Suc_imp_pos: "j > 0" if "f j = Suc k" for j k
      by (rule Nat.gr0I) (use that in auto)

    define f' where "f' = (λn. f (Suc n) - 1)"
    have "filter P xs = tl ys"
    proof (rule Cons.IH)
      show "strict_mono_on {..<length (tl ys)} f'"
      proof (intro strict_mono_onI)
        fix i j assume ij: "i ∈ {..<length (tl ys)}" "j ∈ {..<length
(tl ys)}" "i < j"
        from ij have "Suc i < length ys" "Suc j < length ys"
          by auto
        thus "f' i < f' j"
          using strict_mono_onD[OF Cons.prems(1), of "Suc i" "Suc j"]
                pos[of "Suc i"] pos[of "Suc j"] ‹ys ≠ []› ‹i < j›
```

```
          by (auto simp: strict_mono_on_def diff_less_mono f'_def)
      qed
      show "f' i < length xs" and "tl ys ! i = xs ! f' i" if "i < length
(tl ys)" for i
      proof -
        have "Suc i < length ys"
          using that by auto
        thus "f' i < length xs"
          using Cons.prems(2)[of "Suc i"] pos[of "Suc i"] that by (auto
simp: f'_def)
        show "tl ys ! i = xs ! f' i"
          using ‹Suc i < length ys› that Cons.prems(3)[of "Suc i"] pos[of
"Suc i"]
          by (auto simp: nth_tl nth_Cons f'_def split: nat.splits)
      qed
      show "P (xs ! i) ⟷ (∃j<length (tl ys). f' j = i)" if "i < length
xs" for i
      proof -
        have "P (xs ! i) ⟷ P ((x # xs) ! Suc i)"
          by simp
        also have "... ⟷ {j ∈ {..<length ys}. f j = Suc i} ≠ {}"
          using that by (subst Cons.prems(4)) auto
        also have "bij_betw (λx. x - 1) {j ∈ {..<length ys}. f j = Suc
i}
                     {j ∈ {..<length (tl ys)}. f' j = i}"
          by (rule bij_betwI[of _ _ _ Suc])
             (auto simp: f'_def Suc_diff_Suc f_eq_Suc_imp_pos diff_less_mono
Suc_leI pos)
        hence "{j ∈ {..<length ys}. f j = Suc i} ≠ {} ⟷ {j ∈ {..<length
(tl ys)}. f' j = i} ≠ {}"
          unfolding bij_betw_def by blast
        also have "... ⟷ (∃j<length (tl ys). f' j = i)"
          by blast
        finally show ?thesis .
      qed
    qed
    moreover have "hd ys = x"
      using True ‹f 0 = 0› ‹ys ≠ []› Cons.prems(3)[of 0] by (auto simp:
hd_conv_nth)
    ultimately show ?thesis
      using ‹ys ≠ []› True by force
  qed
qed

lemma filter_eq_iff:
  "filter P xs = ys ⟷
     (∃f. strict_mono_on {..<length ys} f ∧
          (∀i<length ys. f i < length xs ∧ ys ! i = xs ! f i) ∧
          (∀i<length xs. P (xs ! i) ⟷ (∃j. j < length ys ∧ f j = i)))"
```

```
  (is "?lhs = ?rhs")
proof
  show ?rhs if ?lhs
    unfolding that [symmetric] by (rule filterE[OF refl, of P xs]) blast
  show ?lhs if ?rhs
    using that filter_eqI[of ys _ xs P] by blast
qed
```

2.3 Polynomials

```
lemma poly_of_nat [simp]: "poly (of_nat n) x = of_nat n"
  by (simp add: of_nat_poly)

lemma poly_of_int [simp]: "poly (of_int n) x = of_int n"
  by (simp add: of_int_poly)

lemma poly_numeral [simp]: "poly (numeral n) x = numeral n"
  by (metis of_nat_numeral poly_of_nat)

lemma order_gt_0_iff: "p ≠ 0 ⟹ order x p > 0 ⟷ poly p x = 0"
  by (subst order_root) auto

lemma order_eq_0_iff: "p ≠ 0 ⟹ order x p = 0 ⟷ poly p x ≠ 0"
  by (subst order_root) auto

lemma coeff_pcompose_monom_linear [simp]:
  fixes p :: "'a :: comm_ring_1 poly"
  shows "coeff (pcompose p (monom c (Suc 0))) k = c ^ k * coeff p k"
  by (induction p arbitrary: k)
     (auto simp: coeff_pCons coeff_monom_mult pcompose_pCons split: nat.splits)

lemma of_nat_mult_conv_smult: "of_nat n * P = smult (of_nat n) P"
  by (simp add: monom_0 of_nat_monom)

lemma numeral_mult_conv_smult: "numeral n * P = smult (numeral n) P"
  by (metis of_nat_mult_conv_smult of_nat_numeral)

lemma has_field_derivative_poly [derivative_intros]:
  assumes "(f has_field_derivative f') (at x within A)"
  shows   "((λx. poly p (f x)) has_field_derivative
              (f' * poly (pderiv p) (f x))) (at x within A)"
  using DERIV_chain[OF poly_DERIV assms, of p] by (simp add: o_def mult_ac)

lemma sum_order_le_degree:
  assumes "p ≠ 0"
  shows   "(∑x | poly p x = 0. order x p) ≤ degree p"
  using assms
proof (induction "degree p" arbitrary: p rule: less_induct)
  case (less p)
```

```
  show ?case
  proof (cases "∃x. poly p x = 0")
    case False
    thus ?thesis
      by auto
  next
    case True
    then obtain x where x: "poly p x = 0"
      by auto
    have "[:-x, 1:] ^ order x p dvd p"
      by (simp add: order_1)
    then obtain q where q: "p = [:-x, 1:] ^ order x p * q"
      by (elim dvdE)
    have [simp]: "q ≠ 0"
      using q less.prems by auto
    have "order x p = order x p + order x q"
      by (subst q, subst order_mult) (auto simp: order_power_n_n)
    hence "order x q = 0"
      by auto
    hence [simp]: "poly q x ≠ 0"
      by (simp add: order_root)
    have deg_p: "degree p = degree q + order x p"
      by (subst q, subst degree_mult_eq) (auto simp: degree_power_eq)
    moreover have "order x p > 0"
      using x less.prems by (simp add: order_root)
    ultimately have "degree q < degree p"
      by linarith
    hence "(∑x | poly q x = 0. order x q) ≤ degree q"
      by (intro less.hyps) auto
    hence "order x p + (∑x | poly q x = 0. order x q) ≤ degree p"
      by (simp add: deg_p)
    also have "{y. poly q y = 0} = {y. poly p y = 0} - {x}"
      by (subst q) auto
    also have "(∑y ∈ {y. poly p y = 0} - {x}. order y q) =
               (∑y ∈ {y. poly p y = 0} - {x}. order y p)"
      by (intro sum.cong refl, subst q)
         (auto simp: order_mult order_power_n_n intro!: order_0I)
    also have "order x p + ... = (∑y ∈ insert x ({y. poly p y = 0} -
{x}). order y p)"
      using ‹p ≠ 0› by (subst sum.insert) (auto simp: poly_roots_finite)
    also have "insert x ({y. poly p y = 0} - {x}) = {y. poly p y = 0}"
      using ‹poly p x = 0› by auto
    finally show ?thesis .
  qed
qed
```

2.4 Trigonometric functions

```
lemma sin_multiple_reduce:
```

```
  "sin (x * numeral n :: 'a :: {real_normed_field, banach}) =
     sin x * cos (x * of_nat (pred_numeral n)) + cos x * sin (x * of_nat
(pred_numeral n))"
proof -
  have "numeral n = of_nat (pred_numeral n) + (1 :: 'a)"
    by (metis add.commute numeral_eq_Suc of_nat_Suc of_nat_numeral)
  also have "sin (x * ...) = sin (x * of_nat (pred_numeral n) + x)"
    unfolding of_nat_Suc by (simp add: ring_distribs)
  finally show ?thesis
    by (simp add: sin_add)
qed

lemma cos_multiple_reduce:
  "cos (x * numeral n :: 'a :: {real_normed_field, banach}) =
     cos (x * of_nat (pred_numeral n)) * cos x - sin (x * of_nat (pred_numeral
n)) * sin x"
proof -
  have "numeral n = of_nat (pred_numeral n) + (1 :: 'a)"
    by (metis add.commute numeral_eq_Suc of_nat_Suc of_nat_numeral)
  also have "cos (x * ...) = cos (x * of_nat (pred_numeral n) + x)"
    unfolding of_nat_Suc by (simp add: ring_distribs)
  finally show ?thesis
    by (simp add: cos_add)
qed

lemma arccos_eq_pi_iff: "x ∈ {-1..1} ⟹ arccos x = pi ⟷ x = -1"
  by (metis arccos arccos_minus_1 atLeastAtMost_iff cos_pi)

lemma arccos_eq_0_iff: "x ∈ {-1..1} ⟹ arccos x = 0 ⟷ x = 1"
  by (metis arccos arccos_1 atLeastAtMost_iff cos_zero)
```

2.5 Hyperbolic functions

```
lemma cosh_double_cosh: "cosh (2 * x :: 'a :: {banach, real_normed_field})
= 2 * (cosh x)^2 - 1"
  using cosh_double[of x] by (simp add: sinh_square_eq)

lemma sinh_multiple_reduce:
  "sinh (x * numeral n :: 'a :: {real_normed_field, banach}) =
     sinh x * cosh (x * of_nat (pred_numeral n)) + cosh x * sinh (x *
of_nat (pred_numeral n))"
proof -
  have "numeral n = of_nat (pred_numeral n) + (1 :: 'a)"
    by (metis add.commute numeral_eq_Suc of_nat_Suc of_nat_numeral)
  also have "sinh (x * ...) = sinh (x * of_nat (pred_numeral n) + x)"
    unfolding of_nat_Suc by (simp add: ring_distribs)
  finally show ?thesis
    by (simp add: sinh_add)
qed
```

```
lemma cosh_multiple_reduce:
  "cosh (x * numeral n :: 'a :: {real_normed_field, banach}) =
     cosh (x * of_nat (pred_numeral n)) * cosh x + sinh (x * of_nat (pred_numeral
n)) * sinh x"
proof -
  have "numeral n = of_nat (pred_numeral n) + (1 :: 'a)"
    by (metis add.commute numeral_eq_Suc of_nat_Suc of_nat_numeral)
  also have "cosh (x * ...) = cosh (x * of_nat (pred_numeral n) + x)"
    unfolding of_nat_Suc by (simp add: ring_distribs)
  finally show ?thesis
    by (simp add: cosh_add)
qed

lemma cosh_arcosh_real [simp]:
  assumes "x ≥ (1 :: real)"
  shows   "cosh (arcosh x) = x"
proof -
  have "eventually (λt::real. cosh t ≥ x) at_top"
    using cosh_real_at_top by (simp add: filterlim_at_top)
  then obtain t where "t ≥ 1" "cosh t ≥ x"
    by (metis eventually_at_top_linorder linorder_not_le order_le_less)
  moreover have "isCont cosh (y :: real)" for y
    by (intro continuous_intros)
  ultimately obtain y where "y ≥ 0" "x = cosh y"
    using IVT[of cosh 0 x t] assms by auto
  thus ?thesis
    by (simp add: arcosh_cosh_real)
qed

lemma arcosh_eq_0_iff_real [simp]: "x ≥ 1 ⟹ arcosh x = 0 ⟷ x = (1
:: real)"
  using cosh_arcosh_real by fastforce

lemma arcosh_nonneg_real [simp]:
  assumes "x ≥ 1"
  shows   "arcosh (x :: real) ≥ 0"
proof -
  have "1 + 0 ≤ x + (x² - 1) powr (1 / 2)"
    using assms by (intro add_mono) auto
  thus ?thesis unfolding arcosh_def by simp
qed

lemma arcosh_real_strict_mono:
  fixes x y :: real
  assumes "1 ≤ x" "x < y"
  shows   "arcosh x < arcosh y"
proof -
  have "cosh (arcosh x) < cosh (arcosh y)"
```

```
    by (subst (1 2) cosh_arcosh_real) (use assms in auto)
  thus ?thesis
    using assms by (subst (asm) cosh_real_nonneg_less_iff) auto
qed

lemma arcosh_less_iff_real [simp]:
  fixes x y :: real
  assumes "1 ≤ x" "1 ≤ y"
  shows   "arcosh x < arcosh y ⟷ x < y"
  using arcosh_real_strict_mono[of x y] arcosh_real_strict_mono[of y x]
assms
  by (cases x y rule: linorder_cases) auto

lemma arcosh_real_gt_1_iff [simp]: "x ≥ 1 ⟹ arcosh x > 0 ⟷ x ≠
(1 :: real)"
  using arcosh_less_iff_real[of 1 x] by (auto simp del: arcosh_less_iff_real)

lemma sinh_arcosh_real: "x ≥ 1 ⟹ sinh (arcosh x) = sqrt (x² - 1)"
  by (rule sym, rule real_sqrt_unique) (auto simp: sinh_square_eq)


lemma sinh_arsinh_real [simp]: "sinh (arsinh x :: real) = x"
proof -
  have "eventually (λt::real. sinh t ≥ x) at_top"
    using sinh_real_at_top by (simp add: filterlim_at_top)
  then obtain t where "sinh t ≥ x"
    by (metis eventually_at_top_linorder linorder_not_le order_le_less)
  moreover have "eventually (λt::real. sinh t ≤ x) at_bot"
    using sinh_real_at_bot by (simp add: filterlim_at_bot)
  then obtain t' where "t' ≤ t" "sinh t' ≤ x"
    by (metis eventually_at_bot_linorder nle_le)
  moreover have "isCont sinh (y :: real)" for y
    by (intro continuous_intros)
  ultimately obtain y where "x = sinh y"
    using IVT[of sinh t' x t] by auto
  thus ?thesis
    by (simp add: arsinh_sinh_real)
qed

lemma arsinh_real_strict_mono:
  fixes x y :: real
  assumes "x < y"
  shows   "arsinh x < arsinh y"
proof -
  have "sinh (arsinh x) < sinh (arsinh y)"
    by (subst (1 2) sinh_arsinh_real) (use assms in auto)
  thus ?thesis
    using assms by (subst (asm) sinh_real_less_iff) auto
qed
```

```
lemma arsinh_less_iff_real [simp]:
  fixes x y :: real
  shows "arsinh x < arsinh y ⟷ x < y"
  using arsinh_real_strict_mono[of x y] arsinh_real_strict_mono[of y x]
  by (cases x y rule: linorder_cases) auto

lemma arsinh_real_eq_0_iff [simp]: "arsinh x = 0 ⟷ x = (0 :: real)"
  by (metis arsinh_0 sinh_arsinh_real)

lemma arsinh_real_pos_iff [simp]: "arsinh x > 0 ⟷ x > (0 :: real)"
  using arsinh_less_iff_real[of 0 x] by (simp del: arsinh_less_iff_real)

lemma arsinh_real_neg_iff [simp]: "arsinh x < 0 ⟷ x < (0 :: real)"
  using arsinh_less_iff_real[of x 0] by (simp del: arsinh_less_iff_real)

lemma cosh_arsinh_real: "cosh (arsinh x) = sqrt (x² + 1)"
  by (rule sym, rule real_sqrt_unique) (auto simp: cosh_square_eq)


end
```

3 Chebyshev Polynomials

```
theory Chebyshev_Polynomials
imports
  "HOL-Analysis.Analysis"
  "HOL-Real_Asymp.Real_Asymp"
  "HOL-Computational_Algebra.Formal_Laurent_Series"
  "Polynomial_Interpolation.Ring_Hom_Poly"
  "Descartes_Sign_Rule.Descartes_Sign_Rule"
  Polynomial_Transfer
  Chebyshev_Polynomials_Library
begin
```

3.1 Definition

We choose the recursive definition of T_n and U_n and do some setup to define both of them at once.

```
locale gen_cheb_poly =
  fixes c :: "'a :: comm_ring_1"
begin

fun f :: "nat ⇒ 'a ⇒ 'a" where
  "f 0 x = 1"
| "f (Suc 0) x = c * x"
| "f (Suc (Suc n )) x = 2 * x * f (Suc n) x - f n x"
```

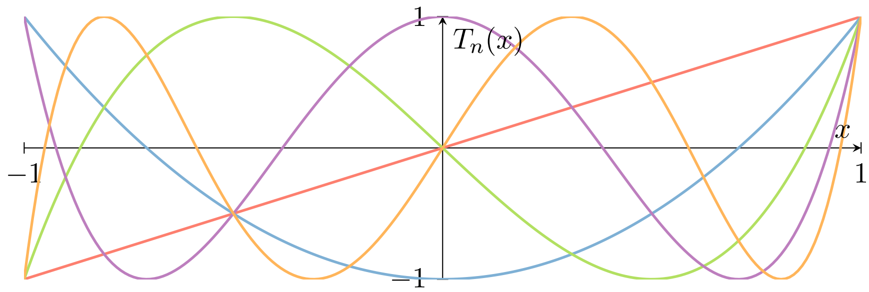

Figure 1: Some of the Chebyshev polynomials of the first kind, T_1 to T_5.

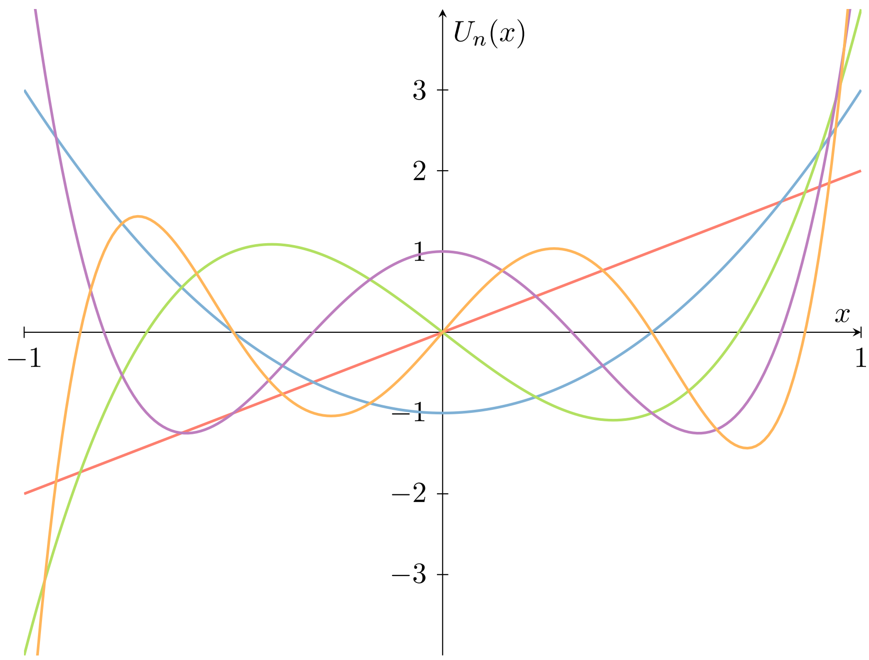

Figure 2: Some of the Chebyshev polynomials of the second kind, U_1 to U_5.

```
fun P :: "nat ⇒ ('a :: comm_ring_1) poly" where
  "P 0 = 1"
| "P (Suc 0) = [:0, c:]"
| "P (Suc (Suc n)) = [:0, 2:] * P (Suc n) - P n"

lemma eval [simp]: "poly (P n) x = f n x"
  by (induction n rule: P.induct) simp_all

lemma eval_0:
  "f n 0 = (if odd n then 0 else (-1) ^ (n div 2))"
  by (induction n rule: induct_nat_012) auto

lemma eval_1 [simp]:
  "f n 1 = of_nat n * (c - 1) + 1"
proof (induction n rule: induct_nat_012)
  case (ge2 n)
  show ?case
    by (auto simp: ge2.IH algebra_simps)
qed auto

lemma uminus [simp]: "f n (-x) = (-1) ^ n  * f n x"
  by (induction n rule: P.induct) (simp_all add: algebra_simps)

lemma pcompose_minus: "pcompose (P n) (monom (-1) 1) = (-1) ^ n * P n"
  by (induction n rule: induct_nat_012)
     (simp_all add: pcompose_diff pcompose_uminus pcompose_smult one_pCons
                    poly_const_pow algebra_simps monom_altdef)

lemma degree_le: "degree (P n) ≤ n"
proof -
  have "i > n ⟹ coeff (P n) i = 0" for i
  by (induction n arbitrary: i rule: P.induct)
     (auto simp: coeff_pCons split: nat.splits)
  thus ?thesis
    using degree_le by blast
qed

lemma lead_coeff:
  "coeff (P n) n = (if n = 0 then 1 else c * 2 ^ (n - 1))"
proof (induction n rule: P.induct)
  case (3 n)
  thus ?case
    using degree_le[of n] by (auto simp: coeff_eq_0 algebra_simps)
qed auto

lemma degree_eq:
  "c * 2 ^ (n - 1) ≠ 0 ⟹ degree (P n :: 'a poly) = n"
  using lead_coeff[of n] degree_le[of n]
  by (metis le_degree nle_le one_neq_zero)
```

```
lemmas [simp del] = f.simps(3) P.simps(3)

end
```

The two related constants *Cheb_poly* and *cheb_poly* denote the *n*-th Chebyshev polynomial of the first kind T_n and its interpretation as a function. We make the definition polymorphic so that it works on every commutative ring; however, many results will only hold for rings (or even only fields) of characteristic *0*.

```
definition cheb_poly :: "nat ⇒ 'a :: comm_ring_1 ⇒ 'a" where
 "cheb_poly = gen_cheb_poly.f 1"

definition Cheb_poly :: "nat ⇒ 'a :: comm_ring_1 poly" where
 "Cheb_poly = gen_cheb_poly.P 1"

interpretation cheb_poly: gen_cheb_poly 1
  rewrites "gen_cheb_poly.f 1 ≡ cheb_poly" and "gen_cheb_poly.P 1 = Cheb_poly"
       and "⋀x :: 'a. 1 * x = x"
       and "⋀n. of_nat n * (1 - 1 :: 'a) + 1 = 1"
  by unfold_locales (simp_all add: cheb_poly_def Cheb_poly_def)

lemmas cheb_poly_simps [code] = cheb_poly.f.simps
lemmas Cheb_poly_simps [code] = cheb_poly.P.simps

lemma Cheb_poly_of_int: "of_int_poly (Cheb_poly n) = Cheb_poly n"
  by (induction n rule: induct_nat_012) (simp_all add: hom_distribs Cheb_poly_simps)

lemma degree_Cheb_poly [simp]:
  "degree (Cheb_poly n :: 'a :: {idom, ring_char_0} poly) = n"
  by (rule cheb_poly.degree_eq) auto

lemma lead_coeff_Cheb_poly [simp]:
  "lead_coeff (Cheb_poly n :: 'a :: {idom, ring_char_0} poly) = 2 ^ (n-1)"
  unfolding degree_Cheb_poly by (subst cheb_poly.lead_coeff) auto

lemma Cheb_poly_nonzero [simp]: "Cheb_poly n ≠ 0"
  by (metis cheb_poly.eval cheb_poly.eval_1 one_neq_zero poly_0)

lemma continuous_cheb_poly [continuous_intros]:
  fixes f :: "'b :: topological_space ⇒ 'a :: {real_normed_algebra_1,
comm_ring_1}"
  shows "continuous_on A f ⟹ continuous_on A (λx. cheb_poly n (f x))"
  unfolding cheb_poly.eval [symmetric]
  by (induction n rule: induct_nat_012) (auto intro!: continuous_intros
simp: cheb_poly_simps)
```

Similarly, we introduce two constants for U_n.

```
definition cheb_poly' :: "nat ⇒ 'a :: comm_ring_1 ⇒ 'a" where
```

```
 "cheb_poly' = gen_cheb_poly.f 2"

definition Cheb_poly' :: "nat ⇒ 'a :: comm_ring_1 poly" where
 "Cheb_poly' = gen_cheb_poly.P 2"

interpretation cheb_poly': gen_cheb_poly 2
  rewrites "gen_cheb_poly.f 2 ≡ cheb_poly'" and "gen_cheb_poly.P 2 =
Cheb_poly'"
       and "⋀n. of_nat n * (2 - 1 :: 'a) + 1 = of_nat (Suc n)"
  by unfold_locales (simp_all add: cheb_poly'_def Cheb_poly'_def)

lemmas cheb_poly'_simps [code] = cheb_poly'.f.simps
lemmas Cheb_poly'_simps [code] = cheb_poly'.P.simps

lemma Cheb_poly'_of_int: "of_int_poly (Cheb_poly' n) = Cheb_poly' n"
  by (induction n rule: induct_nat_012) (simp_all add: hom_distribs Cheb_poly'_simps)

lemma degree_Cheb_poly' [simp]:
  "degree (Cheb_poly' n :: 'a :: {idom, ring_char_0} poly) = n"
  by (rule cheb_poly'.degree_eq) auto

lemma lead_coeff_Cheb_poly' [simp]:
  "lead_coeff (Cheb_poly' n :: 'a :: {idom, ring_char_0} poly) = 2 ^ n"
  unfolding degree_Cheb_poly'
  by (subst cheb_poly'.lead_coeff; cases n) auto

lemma Cheb_poly_nonzero' [simp]: "Cheb_poly' n ≠ (0 :: 'a :: {comm_ring_1,
ring_char_0} poly)"
proof -
  have "poly (Cheb_poly' n) 1 = (of_nat (Suc n) :: 'a)"
    by simp
  also have "... ≠ 0"
    using of_nat_neq_0 by blast
  finally show ?thesis
    by force
qed

lemma continuous_cheb_poly' [continuous_intros]:
  fixes f :: "'b :: topological_space ⇒ 'a :: {real_normed_algebra_1,
comm_ring_1}"
  shows "continuous_on A f ⟹ continuous_on A (λx. cheb_poly' n (f x))"
  by (induction n rule: induct_nat_012) (auto intro!: continuous_intros
simp: cheb_poly'_simps)
```

3.2 Relation to trigonometric functions

Consider the multiple angle formulas for the cosine function:

$$\begin{aligned}
\cos 1x &= \cos x \\
\cos 2x &= 1 + 2\cos^2 x \\
\cos 3x &= -3\cos x + 4\cos^3 x \\
\cos 4x &= 1 - 8\cos^2 x + 8\cos^4 x
\end{aligned}$$

It seems that for any $n \in \mathbb{N}$, we can write $\cos(nx)$ as a sum of powers $\cos^i x$ for $0 \leq i \leq n$, i.e. as a polynomial in $\cos x$ of degree n. It turns out that this polynomial is exactly T_n. This can also serve as an alternative, trigonometric definition of T_n.

Proving it is a simple induction:

```
lemma cheb_poly_cos [simp]:
  fixes x :: "'a :: {banach, real_normed_field}"
  shows "cheb_poly n (cos x) = cos (of_nat n * x)"
proof (induction n rule: induct_nat_012)
  case (ge2 n)
  have [simp]: "cos (x * 2) = 2 * (cos x)² - 1" "sin (x * 2) = 2 * sin
x * cos x"
    using cos_double_cos[of x] sin_double[of x] by (simp_all add: mult_ac)
  show ?case
    by (simp add: ge2 cheb_poly_simps algebra_simps cos_add power2_eq_square)
qed simp_all
```

If we look at the multiple angular formulae for the sine function, we see a similar pattern:

$$\begin{aligned}
\sin 1x &= \sin x \\
\sin 2x &= 2\sin x\cos x \\
\sin 3x &= \sin x(-1 + 4\cos^2 x) \\
\sin 4x &= \sin x(-4\cos x + 8\cos^3 x)
\end{aligned}$$

It seems that $\sin nx/\sin x$ can be expressed as a polynomial in $\cos x$ of degree $n-1$. This polynomial turns out to be exactly U_{n-1}.

```
lemma cheb_poly'_cos:
  fixes x :: "'a :: {banach, real_normed_field}"
  shows "cheb_poly' n (cos x) * sin x = sin (of_nat (n+1) * x)"
proof (induction n rule: induct_nat_012)
  case (ge2 n)
  have [simp]: "sin x * (sin x * t) = (1 - cos x ^ 2) * t" for x t ::
'a
    using sin_squared_eq[of x] by algebra
  have "cheb_poly' (Suc (Suc n)) (cos x) * sin x =
```

```
        2 * cos x * (cheb_poly' (Suc n) (cos x) * sin x) - cheb_poly'
n (cos x) * sin x"
    by (simp add: algebra_simps cheb_poly'_simps)
  also have "... = 2 * cos x * sin (of_nat (Suc n + 1) * x) - sin (of_nat
(n + 1) * x)"
    by (simp only: ge2.IH)
  also have "... - sin (of_nat (Suc (Suc n) + 1) * x) = 0"
    by (simp add: algebra_simps sin_add cos_add power2_eq_square power3_eq_cube
                  sin_multiple_reduce cos_multiple_reduce)
  finally show ?case by simp
qed (auto simp: sin_double)


lemma cheb_poly_conv_cos:
  assumes "|x::real| ≤ 1"
  shows   "cheb_poly n x = cos (n * arccos x)"
  using cheb_poly_cos[of n "arccos x"] assms by simp

lemma cheb_poly'_cos':
  fixes x :: "'a :: {real_normed_field, banach}"
  shows "sin x ≠ 0 ⟹ cheb_poly' n (cos x) = sin (of_nat (n+1) * x)
/ sin x"
  using cheb_poly'_cos[of n x] by (auto simp: field_simps)

lemma cheb_poly'_conv_cos:
  assumes "|x::real| < 1"
  shows   "cheb_poly' n x = sin (real (n+1) * arccos x) / sqrt (1 - x²)"
proof -
  define y where "y = arccos x"
  have x: "cos y = x"
    unfolding y_def using assms cos_arccos_abs by fastforce
  have "x ^ 2 ≠ 1"
    using assms by (subst abs_square_eq_1) auto
  hence y: "sin y ≠ 0"
    using assms by (simp add: sin_arccos_abs y_def)
  have "cheb_poly' n (cos y) = sin ((1 + real n) * y) / sin y"
    using y by (subst cheb_poly'_cos') auto
  also have "sin y = sqrt (1 - x²)"
    unfolding y_def using assms by (subst sin_arccos_abs) auto
  finally show ?thesis
    using x by (simp add: x y_def)
qed


lemma cos_multiple:
  fixes x :: "'a :: {banach, real_normed_field}"
  shows "cos (numeral n * x) = poly (Cheb_poly (numeral n)) (cos x)"
  using cheb_poly_cos[of "numeral n" x] unfolding of_nat_numeral by simp
```

```
lemma sin_multiple:
  fixes x :: "'a :: {banach, real_normed_field}"
  shows "sin (numeral n * x) = sin x * poly (Cheb_poly' (pred_numeral
n)) (cos x)"
  by (metis Suc_eq_plus1 cheb_poly'.eval cheb_poly'_cos mult.commute numeral_eq_Suc
of_nat_numeral)
```

Example application: quadruple-angle formulas for sin and cos:

```
lemma cos_quadruple:
  fixes x :: "'a :: {banach, real_normed_field}"
  shows "cos (4 * x) = 8 * cos x ^ 4 - 8 * cos x ^ 2 + 1"
  by (subst cos_multiple)
     (simp add: eval_nat_numeral Cheb_poly_simps algebra_simps del: cheb_poly.eval)

lemma sin_quadruple:
  fixes x :: "'a :: {banach, real_normed_field}"
  shows "sin (4 * x) = sin x * (8 * cos x ^ 3 - 4 * cos x)"
  by (subst sin_multiple)
     (simp add: eval_nat_numeral Cheb_poly'_simps algebra_simps del: cheb_poly'.eval)
```

3.3 Relation to hyperbolic functions

```
lemma cheb_poly_cosh [simp]:
  fixes x :: "'a :: {banach, real_normed_field}"
  shows "cheb_poly n (cosh x) = cosh (of_nat n * x)"
proof (induction n rule: induct_nat_012)
  case (ge2 n)
  have [simp]: "cosh (x * 2) = 2 * (cosh x)² - 1" "sinh (x * 2) = 2 *
sinh x * cosh x"
    using cosh_double_cosh[of x] sinh_double[of x] by (simp_all add: mult_ac)
  show ?case
    by (simp add: ge2 cheb_poly_simps algebra_simps cosh_add power2_eq_square)
qed simp_all

lemma cheb_poly'_cosh:
  fixes x :: "'a :: {real_normed_field, banach}"
  shows "cheb_poly' n (cosh x) * sinh x = sinh (of_nat (n+1) * x)"
proof (induction n rule: induct_nat_012)
  case (ge2 n)
  have [simp]: "sinh x * (sinh x * t) = (cosh x ^ 2 - 1) * t" for x t
:: 'a
    using sinh_square_eq[of x] by algebra
  have "cheb_poly' (Suc (Suc n)) (cosh x) * sinh x =
        2 * cosh x * (cheb_poly' (Suc n) (cosh x) * sinh x) - cheb_poly'
n (cosh x) * sinh x"
    by (simp add: algebra_simps cheb_poly'_simps)
  also have "... = 2 * cosh x * sinh (of_nat (Suc n + 1) * x) - sinh (of_nat
(n + 1) * x)"
    by (simp only: ge2.IH)
```

```
  also have "... - sinh (of_nat (Suc (Suc n) + 1) * x) = 0"
    by (simp add: algebra_simps sinh_add cosh_add power2_eq_square power3_eq_cube
                  sinh_multiple_reduce cosh_multiple_reduce)
  finally show ?case by simp
qed (auto simp: sinh_double)

lemma cheb_poly_conv_cosh:
  assumes "(x :: real) ≥ 1"
  shows   "cheb_poly n x = cosh (n * arcosh x)"
  using cheb_poly_cosh[of n "arcosh x"] assms
  by (simp del: cheb_poly_cosh)

lemma cheb_poly'_cosh':
  fixes x :: "'a :: {real_normed_field, banach}"
  shows "sinh x ≠ 0 ⟹ cheb_poly' n (cosh x) = sinh (of_nat (n+1) *
x) / sinh x"
  using cheb_poly'_cosh[of n x] by (auto simp: field_simps)

lemma cheb_poly'_conv_cosh:
  assumes "x > (1 :: real)"
  shows   "cheb_poly' n x = sinh (real (n+1) * arcosh x) / sqrt (x² -
1)"
proof -
  have "x² ≠ 1"
    using assms by (simp add: power2_eq_1_iff)
  hence "cheb_poly' n (cosh (arcosh x)) = sinh ((1 + real n) * arcosh
x) / sqrt (x² - 1)"
    using assms by (subst cheb_poly'_cosh') (auto simp: sinh_arcosh_real)
  thus ?thesis
    using assms by simp
qed
```

3.4 Roots

T_n has n distinct real roots, namely:

$$x_k = \cos\left(\frac{2k+1}{2n}\pi\right)$$

These are called the *Chebyshev nodes* of degree n.

```
definition cheb_node :: "nat ⇒ nat ⇒ real" where
  "cheb_node n k = cos (real (2*k+1) / real (2*n) * pi)"

lemma cheb_poly_cheb_node [simp]:
  assumes "k < n"
  shows   "cheb_poly n (cheb_node n k) = 0"
proof -
  have "cheb_poly n (cheb_node n k) = cos ((1 + 2 * real k) / 2 * pi)"
    using assms by (simp add: cheb_node_def)
```

```
  also have "(1 + 2 * real k) / 2 * pi = pi * real (Suc (2 * k)) / 2"
    by (simp add: field_simps)
  also have "cos ... = 0"
    by (rule cos_pi_eq_zero)
  finally show ?thesis .
qed

lemma strict_antimono_cheb_node: "monotone_on {..<n} (<) (>) (cheb_node
n)"
  unfolding cheb_node_def
proof (intro monotone_onI cos_monotone_0_pi)
  fix k l assume kl: "k ∈ {..<n}" "l ∈ {..<n}"
  have "real (2 * l + 1) / real (2 * n) * pi ≤ 1 * pi"
    by (intro mult_right_mono; use kl in simp; fail)
  thus "real (2 * l + 1) / real (2 * n) * pi ≤ pi"
    by simp
qed (auto simp: field_simps)

lemma cheb_node_pos_iff:
  assumes k: "k < n"
  shows   "cheb_node n k > 0 ⟷ k < n div 2"
proof -
  have "(1 + 2 * real k) / (2 * real n) * pi ≤ 1 * pi"
    by (intro mult_right_mono) (use k in auto)
  hence "cos ((1 + 2 * real k) * pi / (2 * real n)) > cos (pi / 2) ⟷
          (1 + 2 * real k) / real n * pi < 1 * pi"
    by (subst cos_mono_less_eq) auto
  also have "... ⟷ (1 + 2 * real k) / real n < 1"
    using pi_gt_zero by (subst mult_less_cancel_right) (auto simp del:
pi_gt_zero)
  also have "((1 + 2 * real k) / real n < 1) ⟷ 1 + 2 * real k < real
n"
    using k by (auto simp: field_simps)
  also have "... ⟷ k < n div 2"
    by linarith
  finally show "cheb_node n k > 0 ⟷ k < n div 2"
    by (simp add: cheb_node_def)
qed

lemma cheb_poly_roots_bij_betw:
  "bij_betw (cheb_node n) {..<n} {x. cheb_poly n x = 0}"
proof -
  have inj: "inj_on (cheb_node n) {..<n}" (is "inj_on ?h _")
    using strict_antimono_cheb_node[of n] unfolding strict_antimono_iff_antimono
by blast

  have "cheb_node n ` {..<n} = {x. cheb_poly n x = 0}"
  proof (rule card_seteq)
    have "finite {x. poly (Cheb_poly n) (x::real) = 0}"
```

```
    by (intro poly_roots_finite) auto
  thus "finite {x. cheb_poly n (x::real) = 0}" by simp
next
  show "cheb_node n ` {..<n} ⊆ {x. cheb_poly n x = 0}"
    by auto
next
  have "{x. cheb_poly n x = 0} = {x. poly (Cheb_poly n) (x::real) =
0}" by simp
  also have "card ... ≤ degree (Cheb_poly n :: real poly)"
    by (intro poly_roots_degree) auto
  also have "... = n" by simp
  also have "n = card (cheb_node n ` {..<n})"
    using inj by (subst card_image) auto
  finally show "card {x::real. cheb_poly n x = 0} ≤ card (cheb_node
n ` {..<n})" .
qed

with inj show ?thesis
  unfolding bij_betw_def by blast
qed

lemma card_cheb_poly_roots: "card {x::real. cheb_poly n x = 0} = n"
  using bij_betw_same_card[OF cheb_poly_roots_bij_betw[of n]] by simp
```

It is easy to see that all the Chebyshev nodes have order 1 as roots of T_n.

```
lemma order_Cheb_poly_cheb_node [simp]:
  assumes "k < n"
  shows   "order (cheb_node n k) (Cheb_poly n) = 1"
proof -
  have "(∑(x::real) | cheb_poly n x = 0. order x (Cheb_poly n)) ≤ n"
    using sum_order_le_degree[of "Cheb_poly n :: real poly"] by simp
  also have "(∑(x::real) | cheb_poly n x = 0. order x (Cheb_poly n))
=
            (∑k<n. order (cheb_node n k) (Cheb_poly n))"
    by (rule sum.reindex_bij_betw [symmetric], rule cheb_poly_roots_bij_betw)
  finally have "(∑k<n. order (cheb_node n k) (Cheb_poly n)) ≤ n" .

  have "(∑l∈{..<n}-{k}. 1 :: nat) ≤ (∑l∈{..<n}-{k}. order (cheb_node
n l) (Cheb_poly n))"
    by (intro sum_mono) (auto simp: Suc_le_eq order_gt_0_iff)
  also have "... + order (cheb_node n k) (Cheb_poly n) =
             (∑l∈insert k ({..<n}-{k}). order (cheb_node n l) (Cheb_poly
n))"
    by (subst sum.insert) auto
  also have "insert k ({..<n}-{k}) = {..<n}"
    using assms by auto
  also have "(∑k<n. order (cheb_node n k) (Cheb_poly n)) ≤ n"
    by fact
  finally have "order (cheb_node n k) (Cheb_poly n) ≤ 1"
```

```
    using assms by simp
  moreover have "order (cheb_node n k) (Cheb_poly n) > 0"
    using assms by (auto simp: order_gt_0_iff)
  ultimately show ?thesis
    by linarith
qed

lemma order_Cheb_poly [simp]:
  assumes "poly (Cheb_poly n) (x :: real) = 0"
  shows   "order x (Cheb_poly n) = 1"
proof -
  have "x ∈ {x. poly (Cheb_poly n) x = 0}"
    using assms by simp
  also have "... = cheb_node n ` {..<n}"
    using cheb_poly_roots_bij_betw assms by (auto simp: bij_betw_def)
  finally show ?thesis
    by auto
qed
```

This also means that T_n is square-free. We only show this for the case where we view T_n as a real polynomial, but this also holds in every other reasonable ring since $\mathbb{R}$ is a splitting field of T_n (as we have just shown). However, we chose not to do this here.

```
lemma rsquarefree_Cheb_poly_real: "rsquarefree (Cheb_poly n :: real poly)"
  unfolding rsquarefree_def by (auto simp: order_eq_0_iff)
```

Similarly, the n distinct real roots of U_n are:

$$y_i = \cos\left(\frac{k+1}{n+1}\pi\right)$$

```
definition cheb_node' :: "nat ⇒ nat ⇒ real" where
  "cheb_node' n k = cos (real (k+1) / real (n+1) * pi)"

lemma cheb_poly'_cheb_node' [simp]:
  assumes "k < n"
  shows   "cheb_poly' n (cheb_node' n k) = 0"
proof -
  define x where "x = real (k + 1) / real (n + 1)"
  have x: "x ∈ {0<..<1}"
    using assms by (auto simp: x_def)
  have "cheb_poly' n (cos (x * pi)) * sin (x * pi) = sin (real (n + 1)
* (x * pi))"
    using assms by (simp add: cheb_poly'_cos)
  also have "real (n + 1) * (x * pi) = real (k + 1) * pi"
    by (simp add: x_def)
  also have "sin ... = 0"
    by (rule sin_npi)
  finally have "cheb_poly' n (cheb_node' n k) * sin (x * pi) = 0"
```

```
    unfolding cheb_node'_def x_def by simp
  moreover have "sin (x * pi) > 0"
    by (intro sin_gt_zero) (use x in auto)
  ultimately show ?thesis
    by simp
qed

lemma strict_antimono_cheb_node': "monotone_on {..<n} (<) (>) (cheb_node'
n)"
  unfolding cheb_node'_def
proof (intro monotone_onI cos_monotone_0_pi)
  fix k l assume kl: "k ∈ {..<n}" "l ∈ {..<n}"
  have " real (l + 1) / real (n + 1) * pi ≤ 1 * pi"
    by (intro mult_right_mono; use kl in simp; fail)
  thus " real (l + 1) / real (n + 1) * pi ≤ pi"
    by simp
  assume "k < l"
  show "real (k + 1) / real (n + 1) * pi < real (l + 1) / real (n + 1)
* pi"
    using kl ‹k < l› by (intro mult_strict_right_mono divide_strict_right_mono)
auto
qed (auto simp: field_simps)

lemma cheb_node'_pos_iff:
  assumes k: "k < n"
  shows   "cheb_node' n k > 0 ⟷ k < n div 2"
proof -
  have "real (k + 1) / real (n + 1) * pi ≤ 1 * pi"
    by (intro mult_right_mono) (use k in auto)
  hence "cos (real (k + 1) / real (n + 1) * pi) > cos (pi / 2) ⟷
          real (k + 1) / real (n + 1) * pi < 1 / 2 * pi"
    using assms by (subst cos_mono_less_eq) auto
  also have "... ⟷ real (k + 1) / real (n + 1) < 1 / 2"
    using pi_gt_zero by (subst mult_less_cancel_right) (auto simp del:
pi_gt_zero)
  also have "real (k + 1) / real (n + 1) < 1 / 2 ⟷ 2 * real k + 2 <
real n + 1"
    using k by (auto simp: field_simps)
  also have "... ⟷ k < n div 2"
    by linarith
  finally show "cheb_node' n k > 0 ⟷ k < n div 2"
    by (simp add: cheb_node'_def)
qed

lemma cheb_poly'_roots_bij_betw:
  "bij_betw (cheb_node' n) {..<n} {x. cheb_poly' n x = 0}"
proof -
  have inj: "inj_on (cheb_node' n) {..<n}"
    using strict_antimono_cheb_node'[of n] unfolding strict_antimono_iff_antimono
```

```
by blast

  have "cheb_node' n ` {..<n} = {x. cheb_poly' n x = 0}"
  proof (rule card_seteq)
    have "finite {x. poly (Cheb_poly' n) (x::real) = 0}"
      by (intro poly_roots_finite) auto
    thus "finite {x. cheb_poly' n (x::real) = 0}" by simp
  next
    show "cheb_node' n ` {..<n} ⊆ {x. cheb_poly' n x = 0}"
      by auto
  next
    have "{x. cheb_poly' n x = 0} = {x. poly (Cheb_poly' n) (x::real)
= 0}" by simp
    also have "card ... ≤ degree (Cheb_poly' n :: real poly)"
      by (intro poly_roots_degree) auto
    also have "... = n" by simp
    also have "n = card (cheb_node' n ` {..<n})"
      using inj by (subst card_image) auto
    finally show "card {x::real. cheb_poly' n x = 0} ≤ card (cheb_node'
n ` {..<n})" .
  qed

  with inj show ?thesis
    unfolding bij_betw_def by blast
qed

lemma card_cheb_poly'_roots: "card {x::real. cheb_poly' n x = 0} = n"
  using bij_betw_same_card[OF cheb_poly'_roots_bij_betw[of n]] by simp

lemma order_Cheb_poly'_cheb_node' [simp]:
  assumes "k < n"
  shows   "order (cheb_node' n k) (Cheb_poly' n) = 1"
proof -
  have "(∑(x::real) | cheb_poly' n x = 0. order x (Cheb_poly' n)) ≤
n"
    using sum_order_le_degree[of "Cheb_poly' n :: real poly"] by simp
  also have "(∑(x::real) | cheb_poly' n x = 0. order x (Cheb_poly' n))
=
            (∑k<n. order (cheb_node' n k) (Cheb_poly' n))"
    by (rule sum.reindex_bij_betw [symmetric], rule cheb_poly'_roots_bij_betw)
  finally have "(∑k<n. order (cheb_node' n k) (Cheb_poly' n)) ≤ n" .

  have "(∑l∈{..<n}-{k}. 1 :: nat) ≤ (∑l∈{..<n}-{k}. order (cheb_node'
n l) (Cheb_poly' n))"
    by (intro sum_mono) (auto simp: Suc_le_eq order_gt_0_iff)
  also have "... + order (cheb_node' n k) (Cheb_poly' n) =
             (∑l∈insert k ({..<n}-{k}). order (cheb_node' n l) (Cheb_poly'
n))"
    by (subst sum.insert) auto
```

```
  also have "insert k ({..<n}-{k}) = {..<n}"
    using assms by auto
  also have "(∑k<n. order (cheb_node' n k) (Cheb_poly' n)) ≤ n"
    by fact
  finally have "order (cheb_node' n k) (Cheb_poly' n) ≤ 1"
    using assms by simp
  moreover have "order (cheb_node' n k) (Cheb_poly' n) > 0"
    using assms by (auto simp: order_gt_0_iff)
  ultimately show ?thesis
    by linarith
qed

lemma order_Cheb_poly' [simp]:
  assumes "poly (Cheb_poly' n) (x :: real) = 0"
  shows   "order x (Cheb_poly' n) = 1"
proof -
  have "x ∈ {x. poly (Cheb_poly' n) x = 0}"
    using assms by simp
  also have "... = cheb_node' n ` {..<n}"
    using cheb_poly'_roots_bij_betw assms by (auto simp: bij_betw_def)
  finally show ?thesis
    by auto
qed

lemma rsquarefree_Cheb_poly'_real: "rsquarefree (Cheb_poly' n :: real
poly)"
  unfolding rsquarefree_def by (auto simp: order_eq_0_iff)
```

3.5 Generating functions

T_n and U_n have the following rational generating functions:

$$\sum_{n=0}^{\infty} T_n(x) t^n = \frac{1 - tx}{1 - 2tx + t^2} \qquad \sum_{n=0}^{\infty} U_n(x) t^n = \frac{1}{1 - 2tx + t^2}$$

This is a simple consequence of the linear recurrence equations they satisfy (which we used as their definitions).

Due to some limitations coming from the type class structure, we cannot currently write this down nicely as an equation, but the following form is almost as good.

```
theorem Abs_fps_Cheb_poly:
  fixes F X T :: "real fps fps"
  defines "X ≡ fps_const fps_X" and "T ≡ fps_X"
  defines "F ≡ Abs_fps (fps_of_poly ∘ Cheb_poly)"
  shows   "F * (1 - 2 * T * X + T²) = 1 - T * X"
proof -
  have "F = 1 - F * T * (T - 2 * X) - T * X"
  proof (rule fps_ext)
```

```
    fix n :: nat
    define foo :: "real fps fps" where "foo = Abs_fps (λna. fps_of_poly
           (pCons 0 (smult 2 (Cheb_poly (Suc na))) - Cheb_poly na))"
    have "fps_nth F n = fps_nth (1 + T * X + T^2 * (foo)) n"
      by (cases n rule: cheb_poly.P.cases)
         (simp_all add: F_def T_def X_def fps_X_power_mult_nth Cheb_poly_simps
foo_def)
    also have "foo = 2 * X * fps_shift 1 F - F"
      by (simp add: foo_def F_def X_def T_def fps_eq_iff numeral_fps_const
                  mult.assoc coeff_pCons split: nat.splits)
    also have "1 + T * X + T^2 * (2 * X * fps_shift 1 F - F) =
              1 + T * X * (1 + 2 * (T * fps_shift 1 F)) - T^2 * F"
      by (simp add: algebra_simps power2_eq_square)
    also have "T * fps_shift 1 F = F - 1"
      by (rule fps_ext) (auto simp: T_def F_def)
    also have "1 + T * X * (1 + 2 * (F - 1)) - T^2 * F = 1 - F * T * (T
- 2 * X) - T * X"
      by (simp add: algebra_simps power2_eq_square)
    finally show "fps_nth F n = fps_nth ... n" .
  qed
  thus ?thesis
    by algebra
qed

theorem Abs_fps_Cheb_poly':
  fixes F X T :: "real fps fps"
  defines "X ≡ fps_const fps_X" and "T ≡ fps_X"
  defines "F ≡ Abs_fps (fps_of_poly ∘ Cheb_poly')"
  shows   "F * (1 - 2 * T * X + T^2) = 1"
proof -
  have "F = 1 - F * T * (T - 2 * X)"
  proof (rule fps_ext)
    fix n :: nat
    define foo :: "real fps fps" where "foo = Abs_fps (λna. fps_of_poly
           (pCons 0 (smult 2 (Cheb_poly' (Suc na))) - Cheb_poly' na))"
    have "fps_nth F n = fps_nth (1 + 2 * T * X + T^2 * (foo)) n"
      by (cases n rule: cheb_poly.P.cases)
         (simp_all add: F_def T_def X_def fps_X_power_mult_nth Cheb_poly'_simps
                      foo_def numeral_fps_const)
    also have "foo = 2 * X * fps_shift 1 F - F"
      by (simp add: foo_def F_def X_def T_def fps_eq_iff numeral_fps_const
                  mult.assoc coeff_pCons split: nat.splits)
    also have "1 + 2 * T * X + T^2 * (2 * X * fps_shift 1 F - F) =
              1 + 2 * T * X * (1 + T * fps_shift 1 F) - T^2 * F"
      by (simp add: algebra_simps power2_eq_square)
    also have "T * fps_shift 1 F = F - 1"
      by (rule fps_ext) (auto simp: T_def F_def)
    also have "1 + 2 * T * X * (1 + (F - 1)) - T^2 * F = 1 - F * T * (T
- 2 * X)"
```

```
    by (simp add: algebra_simps power2_eq_square)
  finally show "fps_nth F n = fps_nth ... n" .
qed
thus ?thesis
  by algebra
qed
```

3.6 Optimality with respect to the ∞-norm

We now turn towards a property of T_n that explains why they are interesting for interpolating smooth functions. If $f : [0,1] \to \mathbb{R}$ is a smooth function on the unit interval, the approximation error attained when interpolating f with a polynomial P of degree n at the interpolation points $x_1, \ldots, x_n$ is

$$\frac{f^{(n)}(\xi)}{n!} \prod_{i=1}^{n} (x - x_i) \ .$$

Therefore, it makes sense to choose the interpolation points such that $\prod_{i=1}^{n}(x - x_i)$ is minimal.
We will show below results that imply that this product cannot be smaller than 2^{1-n}, and it is easy to see that if we choose x_i to be the Chebyshev nodes then the product becomes exactly 2^{1-n} and thus optimal.

Out first result is now the following: The ∞-norm of a monic polynomial of degree n on the unit interval $[-1, 1]$ is at least 2^{1-n}. This gives us a kind of lower bound on the "oscillation" of polynomials: a monic polynomial of degree n cannot stay closer than 2^{1-n} to 0 at every point of the unit interval.

```
lemma Sup_abs_poly_bound_aux:
  fixes p :: "real poly"
  assumes "lead_coeff p = 1"
  shows   "∃x∈{-1..1}. |poly p x| ≥ 1 / 2 ^ (degree p - 1)"
proof (rule ccontr)
  define n where "n = degree p"
  assume "¬(∃x∈{-1..1}. |poly p x| ≥ 1 / 2 ^ (degree p - 1))"
  hence abs_less: "|poly p x| < 1 / 2 ^ (n - 1)" if "x ∈ {-1..1}" for x
    using that unfolding n_def by force

  have "n > 0"
  proof (rule Nat.gr0I)
    assume [simp]: "n = 0"
    hence "p = 1"
      using assms monic_degree_0 unfolding n_def by blast
    with abs_less[of 0] show False
      by simp
  qed

  define q where "q = p - smult (1 / 2 ^ (n - 1)) (Cheb_poly n)"
```

```
  have "coeff q n = 0"
    using assms by (auto simp: q_def n_def cheb_poly.lead_coeff)
  moreover have "degree q ≤ n"
    by (auto simp: n_def q_def degree_diff_le)
  ultimately have "degree q < n"
    using ‹0 < n› eq_zero_or_degree_less[of q n] by force

  define x where "x = (λk. cos (real (2 * k) / real n * pi / 2))"
  have antimono_x: "strict_antimono_on {0..n} x"
    using ‹n > 0› by (auto simp: monotone_on_def x_def cos_mono_less_eq
field_simps)

  have sgn_q_x: "sgn (poly q (x k)) = (-1) ^ Suc k" if k: "k ≤ n" for
k
  proof -
    from k have [simp]: "cheb_poly n (x k) = (-1) ^ k"
      unfolding x_def by auto
    have "poly q (x k) = poly p (x k) - (-1) ^ k / 2 ^ (n-1)"
      by (auto simp: q_def)
    moreover have "|poly p (x k)| < 1 / 2 ^ (n-1)"
      using abs_less[of "x k"] by (auto simp: x_def n_def)

    moreover have "x k ∈ {-1..1}"
      by (auto simp: x_def)
    ultimately have "if even k then poly q (x k) < 0 else poly q (x k)
> 0"
      using abs_less[of "x k"] by (auto simp: q_def sgn_if)
    thus "sgn (poly q (x k)) = (-1) ^ Suc k"
      by (simp add: minus_one_power_iff)
  qed

  have "∃t∈{x (Suc k)<..<x k}. poly q t = 0" if k: "k < n" for k
    using poly_IVT[of "x (Suc k)" "x k" q] sgn_q_x[of k] sgn_q_x[of "Suc
k"] k
          monotone_onD[OF antimono_x, of k "Suc k"]
    by (force simp: sgn_if minus_one_power_iff mult_neg_pos mult_pos_neg
split: if_splits)
  then obtain y where y: "y k ∈ {x (Suc k)<..<x k} ∧ poly q (y k) =
0" if "k < n" for k
    by metis
  have "strict_antimono_on {0..<n} y"
    unfolding monotone_on_def
  proof safe
    fix k l
    assume kl: "k ∈ {0..<n}" "l ∈ {0..<n}" "k < l"
    hence "y k > x (Suc k)" "x l > y l"
      using y[of k] y[of l] by auto
    moreover have "x (Suc k) ≥ x l"
    proof (cases "Suc k = l")
```

```
      case False
      hence "Suc k < 1"
        using kl by linarith
      from monotone_onD[OF antimono_x _ _ this] show ?thesis
        using kl by auto
    qed auto
    ultimately show "y k > y l"
      by linarith
  qed
  hence "inj_on y {0..<n}"
    using strict_antimono_iff_antimono by blast
  hence "card (y ` {0..<n}) = n"
    by (subst card_image) auto

  have "q ≠ 0"
    using abs_less[of 1] by (auto simp: q_def)
  hence "finite {x. poly q x = 0}"
    using poly_roots_finite by blast
  moreover have "y ` {0..<n} ⊆ {x. poly q x = 0}"
    using y by auto
  ultimately have "card (y ` {0..<n}) ≤ card {x. poly q x = 0}"
    using card_mono by blast
  also have "... < n"
    using poly_roots_degree[of q] ‹q ≠ 0› ‹degree q < n› by simp
  also have "card (y ` {0..<n}) = n"
    by fact
  finally show False
    by simp
qed

lemma Sup_abs_poly_bound_unit_ivl:
  fixes p :: "real poly"
  shows   "(SUP x∈{-1..1}. |poly p x|) ≥ |lead_coeff p| / 2 ^ (degree
p - 1)"
proof (cases "p = 0")
  case [simp]: False
  define a where "a = lead_coeff p"
  have [simp]: "a ≠ 0"
    by (auto simp: a_def)
  define q where "q = smult (1 / a) p"
  have [simp]: "lead_coeff q = 1"
    by (auto simp: q_def a_def)
  have p_eq: "p = smult a q"
    by (auto simp: q_def)

  obtain x where x: "x ∈ {-1..1}" "|poly q x| ≥ 1 / 2 ^ (degree q - 1)"
    using Sup_abs_poly_bound_aux[of q] by auto
  show ?thesis
  proof (rule cSup_upper2[of "|poly p x|"])
```

```
    show "bdd_above ((λx. |poly p x|) ` {- 1..1})"
      by (intro bounded_imp_bdd_above compact_imp_bounded compact_continuous_image)
         (auto intro!: continuous_intros)
  qed (use x in ‹auto simp: p_eq abs_mult field_simps›)
qed auto
```

Using an appropriate change of variables, we obtain the following bound in the most general form for a non-constant polynomial $P(x)$ on some non-empty interval $[a, b]$:

$$\sup_{x\in[a,b]} |P(x)| \geq 2 \cdot \mathrm{lc}(p) \cdot \left(\frac{b-a}{4}\right)^{\deg(p)}$$

where $\mathrm{lc}(p)$ denotes the leading coefficient of p.

```
theorem Sup_abs_poly_bound:
  fixes p :: "real poly"
  assumes "a < b" and "degree p > 0"
  shows   "(SUP x∈{a..b}. |poly p x|) ≥ 2 * |lead_coeff p| * ((b - a)
/ 4) ^ degree p"
proof -
  define q where "q = pcompose p [:(a + b) / 2, (b - a) / 2:]"
  define f where "f = (λx. (a + b) / 2 + x * (b - a) / 2)"
  define g where "g = (λx. (a + b) / (a - b) + x * 2 / (b - a))"
  have p_eq: "p = pcompose q [:(a + b) / (a - b), 2 / (b - a):]"
    using assms by (auto simp: q_def field_simps simp flip: pcompose_assoc)
  have "(SUP x∈{-1..1}. |poly q x|) ≥ |lead_coeff q| / 2 ^ (degree q -
1)"
    by (rule Sup_abs_poly_bound_unit_ivl)
  also have "(λx. |poly q x|) = abs ∘ poly p ∘ f"
    by (auto simp: fun_eq_iff q_def poly_pcompose f_def)
  also have "... ` {-1..1} = abs ` poly p ` (f ` {-1..1})"
    by (simp add: image_image)
  also have "f ` {-1..1} = {a..b}"
  proof -
    have "f ` {-1..1} = (+) ((a+b)/2) ` (*) ((b-a)/2) ` {-1..1}"
      by (simp add: image_image f_def algebra_simps)
    also have "(*) ((b-a)/2) ` {-1..1} = {- ((b - a) / 2)..(b - a) / 2}"
      using assms by (subst image_mult_atLeastAtMost) simp_all
    also have "(+) ((a+b)/2) ` ... = {a..b}"
      by (subst image_add_atLeastAtMost) (simp_all add: field_simps)
    finally show ?thesis .
  qed
  also have "abs ` poly p ` {a..b} = (λx. |poly p x|) ` {a..b}"
    by (simp add: image_image o_def)
  also have "lead_coeff q = lead_coeff p * ((b - a) / 2) ^ degree p"
    using assms unfolding q_def by (subst lead_coeff_comp) auto
  also have "degree q = degree p"
    using assms by (auto simp: q_def)
```

```
  also have "|lead_coeff p * ((b - a) / 2) ^ degree p| / (2 ^ (degree p
- 1)) =
             2 * |lead_coeff p| * ((b - a) / 4) ^ degree p"
    using assms
    by (simp add:  power_divide abs_mult power_diff flip: power_mult_distrib)
  finally show ?thesis .
qed
```

If we scale T_n with a factor of 2^{1-n}, it exactly attains the lower bound we just derived. The Chebyshev polynomials of the first kind are, in that sense, the polynomials that stay closest to *0* within the unit interval.

With some more work (that we will not do), one can see that T_n is in fact the *only* polynomial that attains this minimal deviation (see e.g. Corollary 3.4B in Mason & Handscomb [1]). This fact, however, requires proving the Equioscillation Theorem, which is not so easy and beyond the scope of this entry.

```
lemma abs_cheb_poly_le_1:
  assumes "(x :: real) ∈ {-1..1}"
  shows   "|cheb_poly n x| ≤ 1"
proof -
  have "|cheb_poly n (cos (arccos x))| ≤ 1"
    by (subst cheb_poly_cos) auto
  with assms show ?thesis
    by simp
qed

theorem Sup_abs_poly_bound_sharp:
  fixes n :: nat and p :: "real poly"
  defines "p ≡ smult (1 / 2 ^ (n - 1)) (Cheb_poly n)"
  shows   "degree p = n" and "lead_coeff p = 1"
    and   "(SUP x∈{-1..1}. |poly p x|) = 1 / 2 ^ (n - 1)"
proof -
  show p: "degree p = n" "lead_coeff p = 1"
    by (simp_all add: p_def cheb_poly.lead_coeff)
  show "(SUP x∈{- 1..1}. |poly p x|) = 1 / 2 ^ (n - 1)"
  proof (rule antisym)
    show "(SUP x∈{- 1..1}. |poly p x|) ≥ 1 / 2 ^ (n - 1)"
      using Sup_abs_poly_bound_unit_ivl[of p] p by simp
    show "(SUP x∈{- 1..1}. |poly p x|) ≤ 1 / 2 ^ (n - 1)"
    proof (rule cSUP_least)
      fix x :: real assume "x ∈ {-1..1}"
      thus "|poly p x| ≤ 1 / 2 ^ (n - 1)"
        using abs_cheb_poly_le_1[of x n] by (auto simp: p_def field_simps)
    qed auto
  qed
qed
```

A related fact: among all the real polynomials of degree *n* whose absolute value is bounded by 1 within the unit interval, T_n is the one that grows fastest *outside* the unit interval.

```
theorem cheb_poly_fastest_growth:
  fixes p :: "real poly"
  defines "n ≡ degree p"
  assumes p_bounded: "⋀x. |x| ≤ 1 ⟹ |poly p x| ≤ 1"
  assumes x: "x ∉ {-1<..<1}"
  shows   "|cheb_poly n x| ≥ |poly p x|"
proof (cases "n > 0")
  case False
  thus ?thesis
    using p_bounded[of 1] unfolding n_def
    by (auto elim!: degree_eq_zeroE)
next
  case True
  show ?thesis
  proof (rule ccontr)
    assume "¬|poly p x| ≤ |cheb_poly n x|"
    hence gt: "|poly p x| > |cheb_poly n x|" by simp
    define h where "h = smult (cheb_poly n x / poly p x) p"
    have [simp]: "poly h x = cheb_poly n x" using gt by (simp add: h_def)

    have "degree (Cheb_poly n - h) ≤ n"
      by (rule degree_diff_le) (auto simp: n_def h_def)
    from gt have "poly (Cheb_poly n - h) x = 0"
      by (simp add: h_def)
    define a where "a = (λk. cos (real k / n * pi))"
    have cheb_poly_a: "cheb_poly n (a k) = (-1) ^ k" if "k ≤ n" for k
      using ‹n > 0› and ‹k ≤ n›
      by (auto simp: cheb_poly_conv_cos field_simps arccos_cos a_def)
    have a_mono: "a k ≤ a l" if "k ≥ l" "k ≤ n" for k l
      unfolding a_def by (intro cos_monotone_0_pi_le) (insert ‹n > 0›
that, auto simp: field_simps)
    have a_bounds: "|a k| ≤ 1" for k by (simp add: a_def)

    have h_a_bounded: "|poly h (a k)| < 1" if "k ≤ n" for k
    proof -
      have "|poly h (a k)| = |cheb_poly n x / poly p x| * |poly p (a k)|"
        by (simp add: h_def abs_mult)
      also have "... ≤  |cheb_poly n x / poly p x| * 1" using a_bounds[of
k]
        by (intro mult_left_mono) (auto simp: p_bounded)
      also have "... < 1 * 1" using gt
        by (intro mult_strict_right_mono) (auto simp: field_simps)
      finally show ?thesis by simp
    qed

    have "∃t∈{a (Suc k)<..<a k}. cheb_poly n t = poly h t" if "k < n"
for k
```

```
    proof -
      define l where "l = -1 - poly h (a (if even k then Suc k else k))"
      define u where "u = 1 - poly h (a (if even k then k else Suc k))"
      have lu: "l < 0" "u > 0"
        using h_a_bounded[of k] h_a_bounded[of "Suc k"] ‹k < n› by (auto
simp: l_def u_def)

      have "continuous_on {a (Suc k)..a k} (λt. cheb_poly n t - poly h
t)"
        by (intro continuous_intros)
      moreover have "connected {a (Suc k)..a k}" by simp
      ultimately have conn: "connected ((λt. cheb_poly n t - poly h t)
` {a (Suc k)..a k})"
        by (rule connected_continuous_image)

      have "∃t∈{a (Suc k)..a k}. cheb_poly n t - poly h t = l" using
‹k < n›
        by (intro bexI[of _ "a (if even k then Suc k else k)"])
           (auto intro!: a_mono simp: cheb_poly_a l_def)
      moreover have "∃t∈{a (Suc k)..a k}. cheb_poly n t - poly h t =
u" using ‹k < n›
        by (intro bexI[of _ "a (if even k then k else Suc k)"])
           (auto intro!: a_mono simp: cheb_poly_a u_def)
      ultimately have "0 ∈ (λt. cheb_poly n t - poly h t) ` {a (Suc k)..a
k}" using lu
        by (intro connectedD_interval[OF conn, of l u 0]) auto
      then obtain t where t: "t ∈ {a (Suc k)..a k}" "cheb_poly n t =
poly h t"
        by auto
      moreover have "t ≠ a l" if "l ≤ n" for l
      proof
        assume [simp]: "t = a l"
        with t and that have "poly h t = (-1) ^ l" by (simp add: cheb_poly_a)
        hence "|poly h t| = 1" by simp
        with h_a_bounded[OF that] show False by auto
      qed
      from this[of k] and this[of "Suc k"] and ‹k < n›
        have "t ≠ a k" "t ≠ a (Suc k)" by auto
      ultimately show ?thesis by (intro bexI[of _ t]) auto
    qed
    hence "∀k∈{..<n}. ∃t. t ∈ {a (Suc k)<..<a k} ∧ cheb_poly n t = poly
h t" by blast
    then obtain b where b: "⋀k. k < n ⟹ b k ∈ {a (Suc k)<..<a k}"
                           "⋀k. k < n ⟹ cheb_poly n (b k) = poly h
(b k)"
      by (subst (asm) bchoice_iff) blast

    have b_mono: "b k > b l" if "k < l" "l < n" for k l
    proof -
```

```
      have "b l < a l" using b(1)[of l] that by simp
      also have "a l ≤ a (Suc k)" using that by (intro a_mono) auto
      also have "a (Suc k) < b k" using b(1)[of k] that by simp
      finally show ?thesis .
    qed
    have b_inj: "inj_on b {..<n}"
    proof
      fix k l assume "k ∈ {..<n}" "l ∈ {..<n}" "b k = b l"
      thus "k = l" using b_mono[of k l] b_mono[of l k]
        by (cases k l rule: linorder_cases) auto
    qed

    have "Cheb_poly n ≠ h"
    proof
      assume "Cheb_poly n = h"
      hence "poly (Cheb_poly n) 1 = poly h 1" by (simp only: )
      hence "|poly p x| = |cheb_poly n x| * |poly p 1|" using gt
        by (auto simp: h_def field_simps abs_mult)
      also have "... ≤ |cheb_poly n x| * 1" by (intro mult_left_mono p_bounded)
auto
      finally show False using gt by simp
    qed

    have "x ∉ b ` {..<n}"
    proof
      assume "x ∈ b ` {..<n}"
      then obtain k where "k < n" "x = b k" by blast
      hence "abs x < 1" using b(1)[of k] a_bounds[of k] a_bounds[of "Suc
k"] by force
      with x show False by (simp add: abs_if split: if_splits)
    qed
    with b_inj have "Suc n = card (insert x (b ` {..<n}))"
      by (subst card_insert_disjoint) (auto simp: card_image)
    also have "... ≤ card {t. poly (Cheb_poly n - h) t = 0}"
      using b(2) gt ‹Cheb_poly n ≠ h› by (intro card_mono poly_roots_finite)
auto
    also have "... ≤ degree (Cheb_poly n - h)" using ‹Cheb_poly n ≠ h›
      by (intro poly_roots_degree) auto
    also have "... ≤ n" by (intro degree_diff_le) (auto simp: h_def n_def)
    finally show False by simp
  qed
qed
```

3.7 Some basic equations

We first set up a mechanism to allow us to prove facts about Chebyshev polynomials on any ring with characteristic *0* by proving them for Chebyshev polynomials over ℝ.

```
definition rel_ring_int :: "'a :: ring_1 ⇒ 'b :: ring_1 ⇒ bool" where
```

"rel_ring_int x y ⟷ (∃n::int. x = of_int n ∧ y = of_int n)"

lemma *rel_ring_int_0: "rel_ring_int 0 0"*
unfolding *rel_ring_int_def* **by** *(rule exI[of _ 0]) auto*

lemma *rel_ring_int_1: "rel_ring_int 1 1"*
unfolding *rel_ring_int_def* **by** *(rule exI[of _ 1]) auto*

lemma *rel_ring_int_add:*
"rel_fun rel_ring_int (rel_fun rel_ring_int rel_ring_int) (+) (+)"
unfolding *rel_ring_int_def rel_fun_def* **by** *(auto intro: exI[of _ "x + y" for x y])*

lemma *rel_ring_int_mult:*
"rel_fun rel_ring_int (rel_fun rel_ring_int rel_ring_int) (*) (*)"
unfolding *rel_ring_int_def rel_fun_def* **by** *(auto intro: exI[of _ "x * y" for x y])*

lemma *rel_ring_int_minus:*
"rel_fun rel_ring_int (rel_fun rel_ring_int rel_ring_int) (-) (-)"
unfolding *rel_ring_int_def rel_fun_def* **by** *(auto intro: exI[of _ "x - y" for x y])*

lemma *rel_ring_int_uminus:*
"rel_fun rel_ring_int rel_ring_int uminus uminus"
unfolding *rel_ring_int_def rel_fun_def* **by** *(auto intro: exI[of _ "-x" for x])*

lemma *sgn_of_int: "sgn (of_int n :: 'a :: linordered_idom) = of_int (sgn n)"*
by *(auto simp: sgn_if)*

lemma *rel_ring_int_sgn:*
"rel_fun rel_ring_int (rel_ring_int :: 'a :: linordered_idom ⇒ 'b :: linordered_idom ⇒ bool) sgn sgn"
unfolding *rel_ring_int_def rel_fun_def* **using** *sgn_of_int* **by** *metis*

lemma *bi_unique_rel_ring_int:*
"bi_unique (rel_ring_int :: 'a :: ring_char_0 ⇒ 'b :: ring_char_0 ⇒ bool)"
by *(auto simp: rel_ring_int_def bi_unique_def)*

lemmas *rel_ring_int_transfer =*
rel_ring_int_0 rel_ring_int_1 rel_ring_int_add rel_ring_int_mult rel_ring_int_minus
rel_ring_int_uminus bi_unique_rel_ring_int

lemma *rel_poly_rel_ring_int:*
"rel_poly rel_ring_int p q ⟷ (∃r. p = of_int_poly r ∧ q = of_int_poly r)"

```
proof
  assume "rel_poly rel_ring_int p q"
  then obtain f where f: "of_int (f i) = coeff p i" "of_int (f i) = coeff
q i" for i
    unfolding rel_poly_def rel_ring_int_def rel_fun_def by metis
  define g where "g = (λi. if coeff p i = 0 ∧ coeff q i = 0 then 0 else
f i)"
  have g: "of_int (g i) = coeff p i" "of_int (g i) = coeff q i" for i
    by (auto simp: g_def f)
  define r where "r = Abs_poly g"
  have "eventually (λi. g i = 0) cofinite"
    unfolding cofinite_eq_sequentially
    using eventually_gt_at_top[of "degree p"] eventually_gt_at_top[of
"degree q"]
    by eventually_elim (auto simp: g_def coeff_eq_0)
  hence r: "coeff r i = g i" for i
    unfolding r_def by (simp add: Abs_poly_inverse)
  show "∃r. p = of_int_poly r ∧ q = of_int_poly r"
    by (intro exI[of _ r]) (auto simp: poly_eq_iff r g)
qed (auto simp: rel_poly_def rel_ring_int_def rel_fun_def)

lemma Cheb_poly_transfer:
  "rel_fun (=) (rel_poly rel_ring_int) Cheb_poly Cheb_poly"
proof
  fix m n :: nat assume "m = n"
  thus "rel_poly rel_ring_int (Cheb_poly m) (Cheb_poly n :: 'b poly)"
    unfolding rel_poly_rel_ring_int
    by (intro exI[of _ "Cheb_poly m"]) (auto simp: Cheb_poly_of_int)
qed

lemma Cheb_poly'_transfer:
  "rel_fun (=) (rel_poly rel_ring_int) Cheb_poly' Cheb_poly'"
proof
  fix m n :: nat assume "m = n"
  thus "rel_poly rel_ring_int (Cheb_poly' m) (Cheb_poly' n :: 'b poly)"
    unfolding rel_poly_rel_ring_int
    by (intro exI[of _ "Cheb_poly' m"]) (auto simp: Cheb_poly'_of_int)
qed

context
  fixes T :: "'a :: {idom, ring_char_0} itself"
  notes [transfer_rule] = rel_ring_int_transfer [where ?'a = real and
?'b = 'a]
                          Cheb_poly_transfer[where ?'a = real and ?'b
= 'a]
                          Cheb_poly'_transfer[where ?'a = real and ?'b
= 'a]
                          transfer_rule_of_nat transfer_rule_numeral
begin
```

The following rule allows us to prove an equality of real polynomials `P = Q` by proving that $P(\cos x) = Q(\cos x)$ for all $x \in (0, \alpha)$ for some $\alpha > 0$. This holds because there are infinitely many such $\cos x$, but $P - Q$, being a polynomial, can only have finitely many roots if $P \neq 0$.

```
lemma Cheb_poly_equalities_aux:
  fixes p q :: "real poly"
  assumes "a > 0"
  assumes "⋀x. x ∈ {0<..<a} ⟹ poly p (cos x) = poly q (cos x)"
  shows   "p = q"
proof -
  define a' where "a' = max 0 (cos (min a (pi/3)))"
  have "cos (min a (pi / 3)) > cos (pi / 2)"
    by (rule cos_monotone_0_pi) (use assms(1) in ‹auto simp: min_def›)
  moreover have "cos (min a (pi / 3)) < cos 0"
    by (rule cos_monotone_0_pi) (use assms(1) in ‹auto simp: min_def›)
  ultimately have "a' ≥ 0" "a' < 1"
    unfolding a'_def using ‹a > 0›
    by (auto intro!: cos_gt_zero simp: min_def)

  have "infinite {a'<..<1}"
    using ‹a' < 1› by simp
  moreover have "poly (p - q) y = 0" if y: "y ∈ {a'<..<1}" for y
  proof -
    define x where "x = arccos y"
    hence "x < arccos a'"
      unfolding x_def using y ‹a' < 1› ‹a' ≥ 0›
      by (subst arccos_less_mono) auto
    also have "arccos a' ≤ a" using assms(1)
      by (auto simp: a'_def max_def min_def arccos_cos intro: cos_ge_zero
split: if_splits)
    finally have "x < a" .
    moreover have "cos x = y"
      unfolding x_def using y ‹a' ≥ 0› by (subst cos_arccos) auto
    moreover have "x > 0"
      unfolding x_def using arccos_lt_bounded[of y] y ‹a' ≥ 0› by auto
    ultimately show ?thesis
      using assms(2)[of x] by simp
  qed
  hence "{a'<..<1} ⊆ {y. poly (p - q) y = 0}"
    by blast
  ultimately have "infinite {x. poly (p - q) x = 0}"
    using finite_subset by blast
  with poly_roots_finite[of "p - q"] show "p = q"
    by auto
qed
```

First, we show that $T_n(x) = nU_{n-1}(x)$:

```
lemma pderiv_Cheb_poly: "pderiv (Cheb_poly n) = of_nat n * (Cheb_poly'
```

```
(n - 1) :: 'a poly)"
proof (transfer fixing: n, goal_cases)
  case 1
  show ?case
  proof (cases "n = 0")
    case False
    hence n: "n > 0"
      by auto
    show ?thesis
    proof (rule Cheb_poly_equalities_aux[OF pi_gt_zero], goal_cases)
      case x: (1 x)
      from x have [simp]: "sin x ≠ 0"
        using sin_gt_zero by force
      define Q :: "real poly" where "Q = Cheb_poly n"
      define Q' :: "real poly" where "Q' = pderiv Q"
      define f :: "real ⇒ real"
        where "f = (λx. cheb_poly n (cos x) - poly Q (cos x))"
      define g where "g = (λx. - (sin (real n * x) * real n) + sin x
* poly Q' (cos x))"
      have "(f has_field_derivative g x) (at x)"
        unfolding cheb_poly_cos g_def f_def
        by (auto intro!: derivative_eq_intros simp: Q'_def)
      moreover have "f = (λ_. 0)"
        by (auto simp: f_def Q_def)
      hence "(f has_field_derivative 0) (at x)"
        by simp
      ultimately have "g x = 0"
        using DERIV_unique by blast
      also have "g x = sin x * (poly (pderiv (Cheb_poly n)) (cos x) -
real n * cheb_poly' (n-1) (cos x))"
        using cheb_poly'_cos[of "n - 1" x] x n
        by (simp add: g_def Q'_def Q_def of_nat_diff algebra_simps)
      finally show "poly (pderiv (Cheb_poly n)) (cos x) = poly (of_nat
n * Cheb_poly' (n-1)) (cos x)"
        using x by simp
    qed
  qed auto
qed
```

Next, we show that:

$$U_n'(x) = \frac{1}{x^2 - 1}((n+1)T_{n+1}(x) - xU_n(x))$$

```
lemma pderiv_Cheb_poly':
  "pderiv (Cheb_poly' n) * [:-1, 0, 1 :: 'a:] =
     of_nat (n+1) * Cheb_poly (n+1) - [:0,1:] * Cheb_poly' n"
proof (transfer fixing: n, rule Cheb_poly_equalities_aux[OF pi_gt_zero],
goal_cases)
  case x: (1 x)
```

```
  from x have [simp]: "sin x ≠ 0"
    using sin_gt_zero by force
  define Q :: "real poly" where "Q = Cheb_poly' n"
  define Q' :: "real poly" where "Q' = pderiv Q"
  define R :: "real poly" where "R = Cheb_poly (n+1)"
  define f :: "real ⇒ real"
    where "f = (λx. sin (real (n+1) * x) / sin x - poly Q (cos x))"
  define g where "g = (λx::real. ((n+1) * cos ((n+1) * x) * sin x -
                            sin ((n+1) * x) * cos x) / sin x ^ 2 +
                          sin x * poly Q' (cos x))"
  have "(f has_field_derivative g x) (at x)"
    unfolding g_def f_def using x
    by (auto intro!: derivative_eq_intros simp: Q'_def power2_eq_square)
  moreover have ev: "eventually (λy. f y = 0) (nhds x)"
  proof -
    have "eventually (λy. y ∈ {0<..<pi}) (nhds x)"
      by (rule eventually_nhds_in_open) (use x in auto)
    thus ?thesis
    proof eventually_elim
      case (elim y)
      hence "sin y > 0"
        by (intro sin_gt_zero) auto
      thus ?case
        using cheb_poly'_cos[of n y] by (auto simp: f_def Q_def field_simps)
    qed
  qed
  ultimately have "((λ_. 0) has_field_derivative g x) (at x)"
    using DERIV_cong_ev[OF refl ev refl] by simp
  hence "g x = 0"
    using DERIV_unique DERIV_const by blast
  also have "g x = sin x * poly Q' (cos x) +
      (sin x * cos ((n+1) * x) + real n * (sin x * cos ((n+1)*x)) - cos
x * sin ((n+1)*x)) / sin x ^ 2"
    using cheb_poly_cos[of "n - 1" x] x
    by (simp add: g_def Q'_def Q_def of_nat_diff algebra_simps)
  finally have "poly Q' (cos x) = -
                 (real (n+1) * sin x * cos ((n+1) * x) -
                  cos x * sin ((n+1) * x)) / sin x ^ 3"
    using ‹sin x ≠ 0›
    by (auto simp: field_simps eval_nat_numeral)
  also have "sin ((n+1) * x) = cheb_poly' n (cos x) * sin x"
    by (rule cheb_poly'_cos [symmetric])
  also have "cos ((n+1) * x) = cheb_poly (n+1) (cos x)"
    by simp
  also have "-(real (n+1) * sin x * cheb_poly (n+1) (cos x) - cos x *
(cheb_poly' n (cos x) * sin x)) / sin x ^ 3 =
              (cos x * cheb_poly' n (cos x) - real (n+1) * cheb_poly
(n+1) (cos x)) / sin x ^ 2"
    using ‹sin x ≠ 0›
```

```
    by (simp add: field_simps power3_eq_cube power2_eq_square)
  finally have "poly Q' (cos x) * sin x ^ 2 =
                cos x * cheb_poly' n (cos x) - real (n + 1) * cheb_poly
(n + 1) (cos x)"
    using ‹sin x ≠ 0› by (simp add: field_simps)
  thus ?case
    unfolding sin_squared_eq Q'_def Q_def
    by (simp add: algebra_simps power2_eq_square)
qed
```

Next, we have $T_n(x) = \frac{1}{2}(U_n(x) - U_{n-2}(x))$.

```
lemma Cheb_poly_rec:
  assumes n: "n ≥ 2"
  shows "2 * Cheb_poly n = Cheb_poly' n - (Cheb_poly' (n - 2) :: 'a poly)"
proof (transfer fixing: n, rule Cheb_poly_equalities_aux[OF pi_gt_zero],
goal_cases)
  case (1 x)
  have *: "sin x * (sin x * t) = (1 - cos x ^ 2) * t" for t
    using sin_squared_eq[of x] by algebra
  from 1 have "sin x > 0"
    by (intro sin_gt_zero) auto
  hence "(poly (2 * Cheb_poly n) (cos x) - poly (Cheb_poly' n - Cheb_poly'
(n - 2)) (cos x)) = 0"
    using n
    by (auto simp: cheb_poly'_cos' * field_simps sin_add sin_diff cos_add
          power2_eq_square power3_eq_cube of_nat_diff)
  thus ?case
    by simp
qed

lemma cheb_poly_rec:
  assumes n: "n ≥ 2"
  shows "2 * cheb_poly n x = cheb_poly' n x - cheb_poly' (n - 2) (x::'a)"
  using arg_cong[OF Cheb_poly_rec[OF assms], of "λP. poly P x", unfolded
cheb_poly.eval cheb_poly'.eval]
  by (simp add: power2_eq_square algebra_simps)
```

Next, we have $U_n(x) = xU_{n-1}(x) + T_n(x)$.

```
lemma Cheb_poly'_rec:
  assumes n: "n > 0"
  shows   "Cheb_poly' n = [:0,1::'a:] * Cheb_poly' (n - 1) + Cheb_poly
n"
proof (transfer fixing: n, rule Cheb_poly_equalities_aux[OF pi_gt_zero],
goal_cases)
  case (1 x)
  have *: "sin x * (sin x * t) = (1 - cos x ^ 2) * t" for t
    using sin_squared_eq[of x] by algebra
  from 1 have "sin x > 0"
    by (intro sin_gt_zero) auto
```

```
  hence "(poly (Cheb_poly' n) (cos x) - poly ([:0, 1:] * Cheb_poly' (n
- 1) + Cheb_poly n) (cos x)) = 0"
    using n
    by (auto simp: cheb_poly'_cos' * field_simps sin_add cos_add power2_eq_square
                   power3_eq_cube of_nat_diff)
  thus ?case
    by simp
qed

lemma cheb_poly'_rec:
  assumes n: "n > 0"
  shows "cheb_poly' n x = x * cheb_poly' (n-1) x + cheb_poly n (x::'a)"
  using arg_cong[OF Cheb_poly'_rec[OF assms], of "λP. poly P x", unfolded
cheb_poly.eval cheb_poly'.eval]
  by (simp add: power2_eq_square algebra_simps)
```

Next, $T_n(x) = xT_{n-1}(x) + (x^2 - 1)U_{n-2}(x)$.

```
lemma Cheb_poly_rec':
  assumes n: "n ≥ 2"
  shows "Cheb_poly n = [:0,1::'a:] * Cheb_poly (n-1) + [:-1,0,1:] * Cheb_poly'
(n-2)"
proof (transfer fixing: n, rule Cheb_poly_equalities_aux[OF pi_gt_zero],
goal_cases)
  case (1 x)
  have *: "sin x * (sin x * t) = (1 - cos x ^ 2) * t" for t
    using sin_squared_eq[of x] by algebra
  from 1 have "sin x > 0"
    by (intro sin_gt_zero) auto
  hence "poly (Cheb_poly n) (cos x) - poly ([:0, 1:] * Cheb_poly (n-1)
- [:1, 0, - 1:] * Cheb_poly' (n-2)) (cos x) = 0"
    using n
    by (auto simp: cheb_poly'_cos' * field_simps sin_add cos_add sin_diff
cos_diff
                   power2_eq_square power3_eq_cube of_nat_diff)
  thus ?case
    by simp
qed

lemma cheb_poly_rec':
  assumes n: "n ≥ 2"
  shows "cheb_poly n x = x * cheb_poly (n-1) x + (x^2 - 1) * cheb_poly'
(n-2) (x::'a)"
  using arg_cong[OF Cheb_poly_rec'[OF assms], of "λP. poly P x", unfolded
cheb_poly.eval cheb_poly'.eval]
  by (simp add: power2_eq_square algebra_simps)
```

T_n and U_{-1} are a solution to a Pell-like equation on polynomials:

$$T_n(x)^2 + (1 - x^2)U_{n-1}(x)^2 = 1$$

lemma *Cheb_poly_Pell:*
  assumes *n: "n > 0"*
  shows *"Cheb_poly n ^ 2 + [:1, 0, -1::'a:] * Cheb_poly' (n - 1) ^ 2 = 1"*
proof *(transfer fixing: n, rule Cheb_poly_equalities_aux[OF pi_gt_zero], goal_cases)*
  case *(1 x)*
  from *1* **have** *"sin x > 0"*
    by *(intro sin_gt_zero) auto*
  hence *"sin x ^ 2 * (poly (Cheb_poly n ^ 2 + [:1, 0, -1::real:] * Cheb_poly' (n - 1) ^ 2) (cos x) - 1) =
        sin x ^ 2 * (cos (n*x) ^ 2 - 1) + (1 - cos x ^ 2) * sin (n*x) ^ 2"*
    using *n* **by** *(auto simp: cheb_poly'_cos' field_simps power2_eq_square)*
  also have *"... = 0"*
    by *(simp add: sin_squared_eq algebra_simps)*
  finally show *?case*
    using *‹sin x > 0›* **by** *simp*
qed

lemma *cheb_poly_Pell:*
  assumes *n: "n > 0"*
  shows *"cheb_poly n x ^ 2 + (1 - x²) * cheb_poly' (n-1) x ^ 2 = (1 :: 'a)"*
  using *arg_cong[OF Cheb_poly_Pell[OF assms], of "λP. poly P x", unfolded cheb_poly.eval cheb_poly'.eval]*
  by *(simp add: power2_eq_square algebra_simps)*

The following Turán-style equation also holds:

$$T_{n+1}(x)^2 - T_{n+2}(x)T_n(x) = 1 - x^2$$

lemma *Cheb_poly_Turan:*
  *"Cheb_poly (n+1) ^ 2 - Cheb_poly (n+2) * Cheb_poly n = [:1,0,-1::'a:]"*
proof *(transfer fixing: n, rule Cheb_poly_equalities_aux[OF pi_gt_zero], goal_cases)*
  case *(1 x)*
  have **: "sin x * sin x = 1 - cos x ^ 2"
          "sin x * (sin x * t) = (1 - cos x ^ 2) * t"* **for** *t x :: real*
    using *sin_squared_eq[of x]* **by** *algebra+*
  from *1* **have** *"sin x > 0"*
    by *(intro sin_gt_zero) auto*
  hence *"(poly ((Cheb_poly (Suc n))² - Cheb_poly (Suc (Suc n)) * Cheb_poly n) (cos x) - (1 - cos x ^ 2)) = 0"*
    using *‹sin x > 0›*
    apply *(simp add: field_simps cheb_poly'_cos')*
    apply *(auto simp: cheb_poly'_cos' field_simps sin_add cos_add power2_eq_square *
                    sin_multiple_reduce cos_multiple_reduce)*
    done

```
  thus ?case
    by (simp add: power2_eq_square)
qed

lemma cheb_poly_Turan:
  "cheb_poly (n+1) x ^ 2  - cheb_poly (n+2) x * cheb_poly n x = (1 - x
^ 2 :: 'a)"
  using arg_cong[OF Cheb_poly_Turan[of n], of "λP. poly P x", unfolded
cheb_poly.eval]
  by (simp add: power2_eq_square algebra_simps)
```

And, the analogous one for U_n:

$$U_{n+1}(x)^2 - U_{n+2}(x)U_n(x) = 1$$

```
lemma Cheb_poly'_Turan:
  "Cheb_poly' (n+1) ^ 2  - Cheb_poly' (n+2) * Cheb_poly' n = (1 :: 'a
poly)"
proof (transfer fixing: n, rule Cheb_poly_equalities_aux[OF pi_gt_zero],
goal_cases)
  case (1 x)
  have *: "sin x * sin x = 1 - cos x ^ 2"
          "sin x * (sin x * t) = (1 - cos x ^ 2) * t" for t x :: real
    using sin_squared_eq[of x] by algebra+
  from 1 have "sin x > 0"
    by (intro sin_gt_zero) auto
  hence "sin x * ((poly ((Cheb_poly' (Suc n))² - Cheb_poly' (Suc (Suc
n)) * Cheb_poly' n) (cos x) - 1)) = 0"
    using ‹sin x > 0›
    apply (simp add: field_simps cheb_poly'_cos')
    apply (auto simp: cheb_poly'_cos' field_simps sin_add cos_add power3_eq_cube
power2_eq_square *
                      sin_multiple_reduce cos_multiple_reduce)
    done
  thus ?case
    using ‹sin x > 0› by simp
qed

lemma cheb_poly'_Turan:
  "cheb_poly' (n+1) x ^ 2  - cheb_poly' (n+2) x * cheb_poly' n x = (1
:: 'a)"
  using arg_cong[OF Cheb_poly'_Turan[of n], of "λP. poly P x", unfolded
cheb_poly'.eval]
  by (simp add: mult_ac)
```

There is also a nice formula for the product of two Chebyshev polynomials of the first kind:

$$T_m(x)T_n(x) = \frac{1}{2}(T_{m+n}(x) + T_{m-n}(x))$$

```
lemma Cheb_poly_prod:
  assumes "n ≤ m"
  shows   "2 * Cheb_poly m * Cheb_poly n = Cheb_poly (m + n) + (Cheb_poly
(m - n) :: 'a poly)"
proof (transfer fixing: m n, rule Cheb_poly_equalities_aux[OF pi_gt_zero],
goal_cases)
  case (1 x)
  have *: "sin x * sin x = 1 - cos x ^ 2"
          "sin x * (sin x * t) = (1 - cos x ^ 2) * t" for t x :: real
    using sin_squared_eq[of x] by algebra+
  have "poly (Cheb_poly (m + n) + Cheb_poly (m - n) - 2 * Cheb_poly m
* Cheb_poly n) (cos x) = 0"
    using assms
    by (simp add: * cos_add cos_diff of_nat_diff power2_eq_square algebra_simps)
  thus ?case
    by simp
qed

lemma cheb_poly_prod':
  assumes "n ≤ m"
  shows   "2 * cheb_poly m x * cheb_poly n x = cheb_poly (m + n) x + cheb_poly
(m - n) (x :: 'a)"
  using arg_cong[OF Cheb_poly_prod[OF assms], of "λP. poly P x", unfolded
cheb_poly'.eval]
  by (simp add: poly_pcompose)
```

In particular, this leads to a divide-and-conquer-style recurrence relation for T_n for even and odd n:

$$T_{2n}(x) = 2T_n(x)^2 - 1$$
$$T_{2n+1} = 2T_n(x)T_{n+1}(x) - x$$

```
lemma Cheb_poly_even:
  "Cheb_poly (2 * n) = 2 * Cheb_poly n ^ 2 - (1 :: 'a poly)"
  using Cheb_poly_prod[of n n]
  by (simp add: power2_eq_square algebra_simps flip: mult_2)

lemma cheb_poly_even:
  "cheb_poly (2 * n) x = 2 * cheb_poly n x ^ 2 - (1 :: 'a)"
  using arg_cong[OF Cheb_poly_even[of n], of "λP. poly P x", unfolded
cheb_poly'.eval]
  by (simp add: poly_pcompose)

lemma Cheb_poly_odd:
  "Cheb_poly (2 * n + 1) = 2 * Cheb_poly n * Cheb_poly (Suc n) - [:0,1::'a:]"
  using Cheb_poly_prod[of n "n + 1"]
  by (simp add: power2_eq_square algebra_simps flip: mult_2)

lemma cheb_poly_odd:
```

```
  "cheb_poly (2 * n + 1) x = 2 * cheb_poly n x * cheb_poly (Suc n) x -
(x :: 'a)"
  using arg_cong[OF Cheb_poly_odd[of n], of "λP. poly P x", unfolded cheb_poly'.eval]
  by (simp add: poly_pcompose)
```

Remarkably, we also have the following formula for the composition of two Chebyshev polynomials of the first kind:

$$T_{mn}(x) = T_m(T_n(x))$$

```
theorem Cheb_poly_mult:
  "(Cheb_poly (m * n) :: 'a poly) = pcompose (Cheb_poly m) (Cheb_poly
n)"
proof (transfer fixing: m n, rule ccontr)
  assume neq: "(Cheb_poly (m * n) :: real poly) ≠ pcompose (Cheb_poly
m) (Cheb_poly n)" (is "?lhs ≠ ?rhs")
  have "{-1..1} ⊆ {x. poly (?lhs - ?rhs) x = 0}"
    by (auto simp: cheb_poly_conv_cos mult_ac poly_pcompose)
  moreover have "¬finite ({-1..1} :: real set)" by simp
  ultimately have "¬finite {x. poly (?lhs - ?rhs) x = 0}" using finite_subset
by blast
  moreover have "finite {x. poly (?lhs - ?rhs) x = 0}" using neq
    by (intro poly_roots_finite) auto
  ultimately show False by contradiction
qed

corollary cheb_poly_mult: "cheb_poly m (cheb_poly n x) = cheb_poly (m *
n) (x :: 'a)"
proof -
  have "cheb_poly m (cheb_poly n x) = poly (pcompose (Cheb_poly m) (Cheb_poly
n)) x"
    by (simp add: poly_pcompose)
  also note Cheb_poly_mult[symmetric]
  finally show ?thesis by simp
qed
```

For the Chebyshev polynomials of the second kind, the following more complicated relationship holds:

$$U_{mn-1}(x) = U_{m-1}(T_n(x)) \cdot U_{n-1}(x)$$

```
theorem Cheb_poly'_mult:
  assumes "m > 0" "n > 0"
  shows   "(Cheb_poly' (m * n - 1) :: 'a poly) =
             pcompose (Cheb_poly' (m-1)) (Cheb_poly n) * Cheb_poly' (n-1)"
proof (transfer fixing: m n, rule Cheb_poly_equalities_aux[of "pi / n"],
goal_cases)
  case (2 x)
  have *: "sin x * sin x = 1 - cos x ^ 2"
```

```
        "sin x * (sin x * t) = (1 - cos x ^ 2) * t" for t x :: real
      using sin_squared_eq[of x] by algebra+
    have "x < pi / n"
      using 2 by auto
    also have "pi / n ≤ pi / 1"
      using assms by (intro divide_left_mono) auto
    finally have "x < pi"
      by simp
    hence A: "sin x > 0"
      by (intro sin_gt_zero) (use 2 in auto)
    from 2 have B: "sin (n * x) > 0"
      by (intro sin_gt_zero) (use 2 assms in ‹auto simp: field_simps›)
    have "poly ((Cheb_poly' (m * n - 1) :: real poly) -
               pcompose (Cheb_poly' (m-1)) (Cheb_poly n) * Cheb_poly' (n-1))
(cos x) = 0"
      using assms A B
      by (simp add: * cos_add cos_diff of_nat_diff power2_eq_square algebra_simps
poly_pcompose cheb_poly'_cos')
    thus ?case
      by simp
qed (use assms in auto)

lemma cheb_poly'_mult:
  assumes "m > 0" "n > 0"
  shows   "cheb_poly' (m * n - 1) (x :: 'a) =
             cheb_poly' (m-1) (cheb_poly n x) * cheb_poly' (n-1) x"
  using arg_cong[OF Cheb_poly'_mult[OF assms], of "λP. poly P x",
                 unfolded cheb_poly'.eval]
  by (simp add: poly_pcompose)
```

The following two lemmas tell tell us that

$$U'_n(1) = 2\binom{n+2}{3} = \frac{1}{3}n(n+1)(n+2)$$
$$U'_n(-1) = (-1)^{n+1}2\binom{n+2}{3} = \frac{(-1)^{n+1}}{3}n(n+1)(n+2)$$

This is good to know because our formula for U'_n has a “division by zero” at ± 1, so we cannot use it to establish these values.

```
lemma poly_pderiv_Cheb_poly'_1:
  "3 * poly (pderiv (Cheb_poly' n) :: 'a poly) 1 = of_nat ((n + 2) * (n
+ 1) * n)"
proof (transfer fixing: n)
  have "poly (pderiv (Cheb_poly' n)) 1 = real ((n + 2) * (n + 1) * n)
/ 3"
  proof (induction n rule: induct_nat_012)
    case (ge2 n)
    show ?case
```

```
    by (auto simp: pderiv_pCons Cheb_poly'_simps pderiv_diff pderiv_smult
ge2 field_simps)
  qed (auto simp: pderiv_pCons)
  thus "3 * poly (pderiv (Cheb_poly' n)) 1 = real ((n + 2) * (n + 1) *
n)"
    by (simp add: field_simps)
qed

lemma poly_pderiv_Cheb_poly'_neg_1:
  "3 * poly (pderiv (Cheb_poly' n) :: 'a poly) (-1) = (-1)^Suc n * of_nat
((n + 2) * (n + 1) * n)"
proof -
  have "3 * poly (pderiv (pcompose (Cheb_poly' n) (monom (-1::'a) 1)))
1 =
          -3 * poly (pderiv (Cheb_poly' n)) (- 1)"
    by (subst pderiv_pcompose) (auto simp: pderiv_pCons poly_pcompose
monom_altdef)
  also have "3 * poly (pderiv (pcompose (Cheb_poly' n) (monom (-1::'a)
1))) 1 =
             (- 1) ^ n * (3 * poly (pderiv (Cheb_poly' n)) 1)"
    by (subst cheb_poly'.pcompose_minus)
       (auto simp: pderiv_mult one_pCons poly_const_pow pderiv_smult)
  also have "3 * poly (pderiv (Cheb_poly' n) :: 'a poly) 1 = of_nat ((n
+ 2) * (n + 1) * n)"
    by (rule poly_pderiv_Cheb_poly'_1)
  finally show ?thesis
    by simp
qed
```

Another alternative definition of T_n and U_n is as the solutions of the ordinary differential equations

$$(1-x^2)T_n'' - xT_n' + n^2T_n = 0$$
$$(1-x^2)U_n'' - 3xU_n' + n(n+2)U_n = 0$$

```
lemma Cheb_poly_ODE:
  fixes n :: nat
  defines "p ≡ (Cheb_poly n :: 'a poly)"
  shows   "[:1,0,-1:] * (pderiv ^^ 2) p - [:0,1:] * pderiv p + of_nat
n ^ 2 * p = 0"
proof (cases "n = 0")
  case n: False
  define f where "f = [:-1, 0, 1 :: 'a:]"
  have "[:1,0,-1:] * (pderiv ^^ 2) p - [:0, 1:] * pderiv p + of_nat n
^ 2 * p =
        -(f * (pderiv ^^ 2) p) - [:0, 1:] * pderiv p + of_nat n ^ 2 *
p"
    by (simp add: f_def)
  also have "f * (pderiv ^^ 2) p = of_nat n * (pderiv (Cheb_poly' (n -
1)) * f)"
```

```
    by (simp add: p_def numeral_2_eq_2 pderiv_Cheb_poly pderiv_mult)
  also have "pderiv (Cheb_poly' (n - 1)) * f =
             of_nat n * Cheb_poly n - [:0, 1:] * Cheb_poly' (n - 1)"
    unfolding f_def by (subst pderiv_Cheb_poly') (use n in auto)
  also have "- (of_nat n * (of_nat n * Cheb_poly n - [:0, 1:] * Cheb_poly'
(n - 1))) -
               [:0, 1:] * pderiv p + (of_nat n)^2 * p = 0"
    by (simp add: p_def pderiv_Cheb_poly power2_eq_square algebra_simps)
  finally show ?thesis .
qed (auto simp: p_def numeral_2_eq_2)

lemma Cheb_poly'_ODE:
  fixes n :: nat
  defines "p ≡ (Cheb_poly' n :: 'a poly)"
  shows   "[:1,0,-1:] * (pderiv ^^ 2) p - [:0,3:] * pderiv p + of_nat
(n*(n+2)) * p = 0"
proof (cases "n = 0")
  case n: False
  define f where "f = [:-1, 0, 1 :: 'a:]"
  have "[:1,0,-1:] * (pderiv ^^ 2) p - [:0,3:] * pderiv p + of_nat (n*(n+2))
* p =
        -((pderiv ^^ 2) p * f + [:0,3:] * pderiv p) + of_nat (n*(n+2))
* p"
    by (simp add: algebra_simps f_def)
  also have "(pderiv ^^ 2) p * f = pderiv (pderiv p * f) - pderiv p *
pderiv f"
    by (simp add: numeral_2_eq_2 pderiv_mult)
  also have "pderiv p * f = of_nat (n + 1) * Cheb_poly (n + 1) - [:0,
1:] * Cheb_poly' n"
    unfolding p_def f_def by (subst pderiv_Cheb_poly') auto
  also have "pderiv (of_nat (n + 1) * Cheb_poly (n + 1) - [:0, 1:] * Cheb_poly'
n) -
               pderiv p * pderiv f + [:0, 3:] * pderiv p =
             of_nat (n^2 + 2 * n) * p"
    by (auto simp: p_def f_def pderiv_pCons pderiv_diff pderiv_mult
                   pderiv_add pderiv_Cheb_poly power2_eq_square algebra_simps)
  also have "-... + of_nat (n * (n + 2)) * p = 0"
    by (simp add: power2_eq_square)
  finally show ?thesis .
qed (auto simp: numeral_2_eq_2 p_def)

end

lemma cheb_poly_prod:
  fixes x :: "'a :: field_char_0"
  assumes "n ≤ m"
  shows   "cheb_poly m x * cheb_poly n x = (cheb_poly (m + n) x + cheb_poly
(m - n) x) / 2"
  using cheb_poly_prod'[OF assms, of x] by (simp add: field_simps)
```

```
lemma has_field_derivative_cheb_poly [derivative_intros]:
  assumes "(f has_field_derivative f') (at x within A)"
  shows   "((λx. cheb_poly n (f x)) has_field_derivative
             (of_nat n * cheb_poly' (n- 1) (f x) * f')) (at x within
A)"
  unfolding cheb_poly.eval [symmetric]
  by (rule derivative_eq_intros refl assms)+ (simp add: pderiv_Cheb_poly)

lemma has_field_derivative_cheb_poly' [derivative_intros]:
  "(cheb_poly' n has_field_derivative
     (if x = 1 then of_nat ((n + 2) * (n + 1) * n) / 3
      else if x = -1 then (-1)^Suc n * of_nat ((n + 2) * (n + 1) * n)
/ 3
      else (of_nat (n+1) * cheb_poly (Suc n) x - x * cheb_poly' n x) /
(x² - 1)))
   (at x within A)" (is "(_ has_field_derivative ?f') (at _ within _)")
proof -
  define a where "a = poly (pderiv (Cheb_poly' n)) x"
  have "((λx. cheb_poly' n x) has_field_derivative a) (at x within A)"
    unfolding cheb_poly'.eval [symmetric]
    by (rule derivative_eq_intros refl)+ (simp add: pderiv_Cheb_poly'
a_def)
  also {
    have "(x ^ 2 - 1) * a = poly (pderiv (Cheb_poly' n) * [:-1, 0, 1:])
x"
      by (simp add: a_def power2_eq_square pderiv_minus algebra_simps)
    also have "... = of_nat (n+1) * cheb_poly (Suc n) x - x * cheb_poly'
n x"
      by (subst pderiv_Cheb_poly') auto
    finally have *: "of_nat (n+1) * cheb_poly (Suc n) x - x * cheb_poly'
n x = (x ^ 2 - 1) * a" ..
    have "a = ?f'"
    proof (cases "x ^ 2 = 1")
      case x: True
      show ?thesis
      proof (cases "n = 0")
        case False
        thus ?thesis using x
          using poly_pderiv_Cheb_poly'_1[of n, where ?'a = 'a]
                poly_pderiv_Cheb_poly'_neg_1[of n, where ?'a = 'a]
          by (auto simp: power2_eq_1_iff a_def field_simps)
      qed (auto simp: a_def)
    next
      case False
      thus ?thesis
        by (subst *) auto
    qed
  }
```

```
  finally show ?thesis .
qed

lemmas has_field_derivative_cheb_poly'' [derivative_intros] =
  DERIV_chain'[OF _ has_field_derivative_cheb_poly']
```

3.8 Signs of the coefficients

Since $T_n(-x) = (-1)^n T_n(x)$ and analogously for U_n, the Chebyshev polynomials are even functions when n is even and odd functions when n is odd. Consequently, when n is even, the coefficients of X^k for any odd k are 0 and analogously when n is odd.

```
lemma coeff_Cheb_poly_eq_0:
  assumes "odd (n + k)"
  shows   "coeff (Cheb_poly n :: 'a :: {idom,ring_char_0} poly) k = 0"
proof -
  note [transfer_rule] =
    rel_ring_int_transfer [where ?'a = real and ?'b = 'a]
    Cheb_poly_transfer[where ?'a = real and ?'b = 'a]
    transfer_rule_of_nat transfer_rule_numeral
  show ?thesis
  proof (transfer fixing: n k)
    have "coeff ((-1) ^ n * pcompose (Cheb_poly n) (monom (-1) 1)) k =
          ((-1)^(n+k) * coeff (Cheb_poly n) k :: real)"
      by (simp add: one_pCons poly_const_pow power_add)
    also have "((-1) ^ n * pcompose (Cheb_poly n) (monom (-1) 1)) = (Cheb_poly
n :: real poly)"
      by (subst cheb_poly.pcompose_minus) auto
    finally show "coeff (Cheb_poly n :: real poly) k = 0"
      using assms by auto
  qed
qed

lemma coeff_Cheb_poly'_eq_0:
  assumes "odd (n + k)"
  shows   "coeff (Cheb_poly' n :: 'a :: {idom,ring_char_0} poly) k = 0"
proof -
  note [transfer_rule] =
    rel_ring_int_transfer [where ?'a = real and ?'b = 'a]
    Cheb_poly'_transfer[where ?'a = real and ?'b = 'a]
    transfer_rule_of_nat transfer_rule_numeral
  show ?thesis
  proof (transfer fixing: n k)
    have "coeff ((-1) ^ n * pcompose (Cheb_poly' n) (monom (-1) 1)) k
=
          ((-1)^(n+k) * coeff (Cheb_poly' n) k :: real)"
      by (simp add: one_pCons poly_const_pow power_add)
    also have "((-1) ^ n * pcompose (Cheb_poly' n) (monom (-1) 1)) = (Cheb_poly'
```

```
n :: real poly)"
      by (subst cheb_poly'.pcompose_minus) auto
    finally show "coeff (Cheb_poly' n :: real poly) k = 0"
      using assms by auto
  qed
qed
```

Next, we analyse the behaviour of the signs of the coefficients of T_n and U_n more generally and show that:

- The leading coefficient is positive.
- After that, every second coefficient is `0`.
- The remaining coefficients are non-zero and their signs alternate.

In conclusion, we have

$$\operatorname{sgn}([X^k]\,T_n(X)) = \operatorname{sgn}([X^k]\,U_n(X)) = \begin{cases} 0 & \text{if } k > n \text{ or } (n+k) \text{ is odd} \\ (-1)^{\frac{n-k}{2}} & \text{otherwise} \end{cases}$$

The proof works using Descartes' rule of signs: We know that T_n and U_n have n distinct real roots and $\lfloor \frac{n}{2} \rfloor$ of them are positive. By Descartes' rule of signs, this implies that the coefficient sequences of T_n and U_n must have at least $\lfloor \frac{n}{2} \rfloor$ sign alternations. However, we also already know that every other coefficient of T_n and U_n starting with $[X^{n-1}]$ is 0, so the number of sign alternations must be *exactly* $\lfloor \frac{n}{2} \rfloor$.

```
lemma sgn_coeff_Cheb_poly_aux:
  fixes n :: nat and P :: "real poly"
  assumes "degree P = n"
  assumes "⋀i. odd (n + i) ⟹ coeff P i = 0"
  assumes "card {x. x > 0 ∧ poly P x = 0} = n div 2"
  assumes "rsquarefree P"
  assumes "coeff P n > 0"
  shows "sgn (coeff P i) = (if i > n ∨ odd (n + i) then 0 else (-1) ^
((n - i) div 2))"
proof (cases "n > 1")
  case False
  hence "n = 0 ∨ n = 1"
    by linarith
  thus ?thesis
  proof (elim disjE)
    assume [simp]: "n = 0"
    show ?thesis
      using assms by (cases "i = 0") (auto simp: coeff_eq_0)
  next
```

```
    assume [simp]: "n = 1"
    consider "i = 0" | "i = 1" | "i > 1"
      by linarith
    thus ?thesis
      by cases (use assms in ‹auto simp: coeff_eq_0›)
  qed
next
  case n: True
  define xs where "xs = coeffs P"
  define ys where "ys = filter (λx. x ≠ 0) (map sgn xs)"
  have [simp]: "P ≠ 0"
    using assms by auto
  note [simp] = ‹degree P = n›

  have "count_roots_with (λx. x > 0) P =
          (∑(x::real) | x > 0 ∧ poly P x = 0. order x P)"
    unfolding count_roots_with_def roots_with_def ..
  also have "... = (∑(x::real) | x > 0 ∧ poly P x = 0. 1)"
    using ‹rsquarefree P› by (intro sum.cong) (auto simp: rsquarefree_def
order_eq_0_iff)
  also have "... = card {x::real. x > 0 ∧ poly P x = 0}"
    by simp
  also have "... = n div 2"
    by fact
  finally have "count_roots_with (λx::real. x > 0) P = n div 2" .
  hence "sign_changes xs ≥ n div 2"
    using descartes_sign_rule_aux[of P] by (simp add: xs_def)
  also have "sign_changes xs = length (remdups_adj ys) - 1"
    by (simp add: sign_changes_def ys_def)
  finally have length_gt: "length (remdups_adj ys) > n div 2"
    using n by simp

  define d where "d = n mod 2"

  have len_ys_conv_card: "length ys = card {i∈{..n div 2}. coeff P (2
* i + d) ≠ 0}"
  proof -
    have "length ys = card {i. i < Suc n ∧ map sgn xs ! i ≠ 0}"
      unfolding ys_def xs_def
      by (subst length_filter_conv_card) (simp_all add: length_coeffs_degree)
    also have "{i. i < Suc n ∧ map sgn xs ! i ≠ 0} = {i∈{..n}. coeff
P i ≠ 0}"
      by (intro Collect_cong conj_cong)
         (auto simp: xs_def map_nth length_coeffs_degree sgn_eq_0_iff
nth_coeffs_coeff)
    also have "... = {i∈{..n}. even (i + n) ∧ coeff P i ≠ 0} ∪
                    {i∈{..n}. odd (i + n) ∧ coeff P i ≠ 0}"
      by blast
    also have "{i∈{..n}. odd (i + n) ∧ coeff P i ≠ 0} = {}"
```

```
      using assms(2) by auto
    finally have "length ys = card {i∈{..n}. even (i + n) ∧ coeff P i
≠ 0}"
      by simp
    also have "bij_betw (λi. i div 2) {i∈{..n}. even (i + n) ∧ coeff
P i ≠ 0}
                {i∈{..n div 2}. coeff P (2 * i + d) ≠ 0}"
      by (rule bij_betwI[of _ _ _ "λi. 2 * i + d"]; cases "even n")
         (auto elim!: evenE oddE simp: Suc_double_not_eq_double d_def)
    hence "card {i∈{..n}. even (i + n) ∧ coeff P i ≠ 0} =
           card {i∈{..n div 2}. coeff P (2 * i + d) ≠ 0}"
      by (rule bij_betw_same_card)
    finally show ?thesis
      by simp
  qed

  have "length ys ≤ n div 2 + 1"
  proof -
    have "card {i∈{..n div 2}. coeff P (2 * i + d) ≠ 0} ≤ card {..n div
2}"
      by (rule card_mono) auto
    with len_ys_conv_card show ?thesis
      by simp
  qed

  have "length (remdups_adj ys) ≤ length ys"
    by (rule remdups_adj_length)
  hence "length (remdups_adj ys) = length ys" and len_ys: "length ys
= n div 2 + 1"
    using length_gt ‹length ys ≤ n div 2 + 1› by linarith+
  hence distinct: "distinct_adj ys"
    by (simp add: distinct_adj_conv_length_remdups_adj)

  have coeff_nz: "coeff P (2 * i + d) ≠ 0" if "i ≤ n div 2" for i
  proof -
    have "{i∈{..n div 2}. coeff P (2 * i + d) ≠ 0} = {..n div 2}"
    proof (rule card_subset_eq)
      show "card {i ∈ {..n div 2}. coeff P (2 * i + d) ≠ 0} = card {..n
div 2}"
        using len_ys len_ys_conv_card by simp
    qed auto
    thus ?thesis using that
      by blast
  qed

  have coeff_eq_0_iff: "coeff P i = 0 ⟷ i > n ∨ odd (n + i)" for i
  proof
    assume "coeff P i = 0"
    hence "odd (n + i)" if "i ≤ n"
```

```
      using coeff_nz[of "i div 2"] that
      by (cases "even n"; cases "even i"; auto simp: d_def elim!: evenE
oddE)
    thus "i > n ∨ odd (n + i)"
      by linarith
  next
    assume "i > n ∨ odd (n + i)"
    thus "coeff P i = 0"
      using coeff_eq_0[of P i] assms(2)[of i] by auto
  qed
  have [simp]: "length (coeffs P) = Suc n"
    by (auto simp: length_coeffs_degree)

  have ys_eq: "ys = map (λi. sgn (coeff P (2 * i + d))) [0..<Suc (n div
2)]"
    unfolding ys_def
  proof (rule filter_eqI[where f = "λi. 2 * i + d"], goal_cases)
    case 1
    thus ?case
      by (auto intro!: strict_mono_onI)
  next
    case (2 i)
    hence "i < Suc (n div 2)"
      by simp
    hence "2 * i + d < Suc n"
      by (cases "even n") (auto elim!: evenE oddE simp: d_def)
    thus ?case
      by (auto simp:  xs_def d_def length_coeffs_degree)
  next
    case (3 i)
    hence "i < Suc (n div 2)"
      by simp
    hence "2 * i + d < Suc n"
      by (cases "even n") (auto elim!: evenE oddE simp: d_def)
    thus ?case
      by (auto simp del: upt_Suc simp: xs_def length_coeffs_degree nth_coeffs_coeff)
  next
    case (4 i)
    from 4 have "i ≤ n"
      by (simp add: xs_def)
    hence "map sgn xs ! i ≠ 0 ⟷ even (n + i)"
      by (simp add: xs_def nth_coeffs_coeff sgn_eq_0_iff coeff_eq_0_iff)
    also have "... ⟷ (∃j<Suc (n div 2). 2 * j + d = i)"
      unfolding d_def using ‹i ≤ n›
      by (cases "even n"; cases "even i")
         (auto elim!: evenE oddE simp: Suc_double_not_eq_double
            eq_commute[of "2 * x" "Suc y" for x y])
    finally show ?case
      by simp
```

```
  qed

  have *: "coeff P (2 * i + d) * coeff P (2 * Suc i + d) < 0" if "i <
n div 2" for i
  proof -
    have "ys ! i ≠ ys ! Suc i"
      using that distinct by (intro distinct_adj_nth) (auto simp: len_ys)
    also have "ys ! i = sgn (coeff P (2 * i + d))"
      using that by (auto simp: ys_eq map_nth simp del: upt_Suc)
    also have "ys ! Suc i = sgn (coeff P (2 * Suc i + d))"
      using that by (auto simp: ys_eq map_nth simp del: upt_Suc)
    finally have "sgn (coeff P (2 * i + d)) ≠ sgn (coeff P (2 * Suc i
+ d))" .
    moreover have "2 * i + d + 2 ≤ n"
      using that unfolding d_def by (cases "even n") (auto elim!: evenE
oddE)
    hence "coeff P (2 * i + d) ≠ 0" "coeff P (2 * Suc i + d) ≠ 0"
      using that by (auto simp: coeff_eq_0_iff d_def)
    ultimately show ?thesis
      by (auto simp: sgn_if mult_neg_pos mult_pos_neg split: if_splits)
  qed
  have **: "coeff P i * coeff P (i + 2) < 0" if "even (n + i)" "i + 1
< n" for i
    using *[of "i div 2"] that by (auto simp: d_def elim!: evenE oddE)

  have ***: "sgn (coeff P (n - 2 * i)) = (-1) ^ i" if "2 * i ≤ n" for
i
    using that
  proof (induction i)
    case 0
    thus ?case
      using assms by (auto simp: sgn_if)
  next
    case (Suc i)
    have "coeff P (n - 2 * Suc i) * coeff P (n - 2 * Suc i + 2) < 0"
      by (intro **) (use Suc in auto)
    hence "sgn (coeff P (n - 2 * Suc i) * coeff P (n - 2 * Suc i + 2))
= -1"
      using sgn_neg by blast
    also have "n - 2 * Suc i + 2 = n - 2 * i"
      using Suc.prems by simp
    also have "sgn (coeff P (n - 2 * Suc i) * coeff P (n - 2 * i)) =
               sgn (coeff P (n - 2 * Suc i)) * sgn (coeff P (n - 2 * i))"
      by (simp add: sgn_mult)
    also have "sgn (coeff P (n - 2 * i)) = (-1) ^ i"
      by (rule Suc.IH) (use Suc.prems in auto)
    finally show ?case
      by (auto simp: sgn_if)
  qed
```

```
  show "sgn (coeff P i) = (if i > n ∨ odd (n + i) then 0 else (-1) ^
((n - i) div 2))"
    using coeff_eq_0[of P i] assms(2)[of i] ***[of "(n - i) div 2"]
    by auto
qed

theorem sgn_coeff_Cheb_poly:
  "sgn (coeff (Cheb_poly n) i :: 'a :: linordered_idom) =
     (if i > n ∨ odd (n + i) then 0 else (-1) ^ ((n - i) div 2))"
proof -
  note [transfer_rule] =
    rel_ring_int_transfer [where ?'a = real and ?'b = 'a]
    rel_ring_int_sgn [where ?'a = real and ?'b = 'a]
    Cheb_poly_transfer[where ?'a = real and ?'b = 'a]
    transfer_rule_of_nat transfer_rule_numeral
  show ?thesis
  proof (transfer fixing: n i, rule sgn_coeff_Cheb_poly_aux)
    have "bij_betw (cheb_node n) {k∈{..<n}. k < n div 2} {x∈{x. cheb_poly
n x = 0}. x > 0}"
      using cheb_poly_roots_bij_betw by (rule bij_betw_Collect) (auto
simp: cheb_node_pos_iff)
    also have "{k∈{..<n}. k < n div 2} = {..<n div 2}"
      by auto
    finally have "bij_betw (cheb_node n) {..<n div 2} {x. x > 0 ∧ cheb_poly
n x = 0}"
      by (simp add: conj_commute)
    from bij_betw_same_card[OF this]
      show "card {x. 0 < x ∧ poly (Cheb_poly n :: real poly) x = 0} =
n div 2"
      by simp
  qed (auto simp: coeff_Cheb_poly_eq_0 cheb_poly.lead_coeff rsquarefree_Cheb_poly_real)
qed

theorem sgn_coeff_Cheb_poly':
  "sgn (coeff (Cheb_poly' n) i :: 'a :: linordered_idom) =
     (if i > n ∨ odd (n + i) then 0 else (-1) ^ ((n - i) div 2))"
proof -
  note [transfer_rule] =
    rel_ring_int_transfer [where ?'a = real and ?'b = 'a]
    rel_ring_int_sgn [where ?'a = real and ?'b = 'a]
    Cheb_poly'_transfer[where ?'a = real and ?'b = 'a]
    transfer_rule_of_nat transfer_rule_numeral
  show ?thesis
  proof (transfer fixing: n i, rule sgn_coeff_Cheb_poly_aux)
    have "bij_betw (cheb_node' n) {k∈{..<n}. k < n div 2} {x∈{x. cheb_poly'
n x = 0}. x > 0}"
      using cheb_poly'_roots_bij_betw by (rule bij_betw_Collect) (auto
simp: cheb_node'_pos_iff)
```

```
    also have "{k∈{..<n}. k < n div 2} = {..<n div 2}"
      by auto
    finally have "bij_betw (cheb_node' n) {..<n div 2} {x. x > 0 ∧ cheb_poly'
n x = 0}"
      by (simp add: conj_commute)
    from bij_betw_same_card[OF this]
      show "card {x. 0 < x ∧ poly (Cheb_poly' n :: real poly) x = 0}
= n div 2"
      by simp
  qed (auto simp: coeff_Cheb_poly'_eq_0 cheb_poly'.lead_coeff rsquarefree_Cheb_poly'_real)
qed
```

3.9 Orthogonality and integrals

```
lemma cis_eq_1_iff: "cis x = 1 ⟷ (∃n. x = 2 * pi * real_of_int n)"
proof
  assume "cis x = 1"
  hence "Re (cis x) = 1"
    by (subst ‹cis x = 1›) auto
  hence "cos x = 1"
    by simp
  thus "∃n. x = 2 * pi * real_of_int n"
    by (subst (asm) cos_one_2pi_int) auto
qed auto


context
  fixes n :: nat and x :: "nat ⇒ real"
  defines "x ≡ (λk. cos (real (Suc (2 * k)) / real (2 * n) * pi))"
begin

lemma cheb_poly_orthogonality_discrete_aux:
  assumes "l ∈ {0<..<2*n}"
  shows "(∑k<n. cos (real l * real (Suc (2 * k)) / real (2 * n) * pi))
= 0"
proof (cases "n = 0")
  case n: False
  define ω where "ω = cis (real l / real (2 * n) * pi)"
  have [simp]: "ω ≠ 0"
    by (auto simp: ω_def)
  have not_one: "ω² ≠ 1"
  proof
    assume "ω² = 1"
    then obtain t where t: "real l * pi / real n = 2 * pi * real_of_int
t"
      unfolding ω_def Complex.DeMoivre cis_eq_1_iff by auto
    have "real_of_int (int l) = real l"
      by simp
    also have "... = real_of_int (2 * t * int n)"
```

```
      using n t by (simp add: field_simps)
    finally have "int l = int (2 * n) * t"
      by (subst (asm) of_int_eq_iff) (simp add: mult_ac)
    hence "int (2 * n) dvd int l"
      unfolding dvd_def ..
    hence "2 * n dvd l"
      by presburger
    thus False
      using assms n by (auto dest!: dvd_imp_le)
  qed

  have [simp]: "Im ω ≠ 0"
  proof
    assume "Im ω = 0"
    have "norm ω = 1"
      by (auto simp: ω_def)
    hence "|Re ω| = 1"
      using ‹Im ω = 0› by (auto simp: norm_complex_def)
    hence "ω ∈ {1, -1}"
      by (auto simp: complex_eq_iff ‹Im ω = 0›)
    hence "ω ^ 2 = 1"
      by auto
    thus False
      using not_one by contradiction
  qed

  have "(∑k<n. cos (real l * real (Suc (2 * k)) / real (2 * n) * pi))
= Re (∑k<n. ω ^ Suc (2 * k))"
    unfolding ω_def Complex.DeMoivre by (simp add: algebra_simps ω_def)
  also have "(∑k<n. ω ^ Suc (2 * k)) = ω * (∑k<n. (ω²) ^ k)"
    by (simp add: sum_distrib_left power_mult)
  also have "... = (1 - ω² ^ n) * (ω / (1 - ω²))"
    by (subst sum_gp_strict) (use not_one in ‹simp_all add: algebra_simps›)
  also have "ω² ^ n = cis (real l * pi)"
    using n by (simp add: ω_def Complex.DeMoivre)
  also have "... = (-1) ^ l"
    unfolding Complex.DeMoivre [symmetric] by simp
  also have "ω / (1 - ω²) = inverse (-(ω - inverse ω))"
    using not_one by (simp add: power2_eq_square field_simps)
  also have "inverse ω = cnj ω"
    by (simp add: ω_def cis_cnj)
  also have "inverse (-(ω - cnj ω)) = i / (2 * Im ω)"
    by (subst complex_diff_cnj) (auto simp: field_simps)
  finally show ?thesis
    by simp
qed auto
```

For $k = 0, \ldots, n-1$ let $x_k = \cos(\frac{2k+1}{2n}\pi)$ be the Chebyshev nodes of order *n*, i.e. the roots of T_n. Then the following discrete orthogonality relation holds for the Chebyshev polynomials of the first kind (for any $i, j < n$):

$$\sum_{k=0}^{n-1} T_i(x_k) T_j(x_k) = \begin{cases} n & \text{if } i = j = 0 \\ \frac{n}{2} & \text{if } i = j \neq 0 \\ 0 & \text{if } i \neq j \end{cases}$$

```
theorem cheb_poly_orthogonality_discrete:
  fixes i j :: nat
  assumes "i < n" "j < n"
  shows "(∑k<n. cheb_poly i (x k) * cheb_poly j (x k)) =
           (if i = j then if i = 0 then n else n / 2 else 0)"
proof (cases "n = 0")
  case False
  hence n: "n > 0"
    by auto
  show ?thesis
    using assms(1,2)
  proof (induction j i rule: linorder_wlog)
    case (le j i)
    have "(∑k<n. cheb_poly i (x k) * cheb_poly j (x k)) =
            (∑k<n. cos (real (i + j) * (real (Suc (2 * k)) / real (2 *
n)) * pi)) / 2 +
            (∑k<n. cos (real (i - j) * (real (Suc (2 * k)) / real (2 *
n)) * pi)) / 2 "
      unfolding cheb_poly_prod [OF le(1)] using le
      by (simp add: x_def sum.distrib add_divide_distrib of_nat_diff mult_ac
               flip: sum_divide_distrib)
    also have "(∑k<n. cos (real (i - j) * (real (Suc (2 * k)) / real
(2 * n)) * pi)) =
               (if i = j then real n else 0)"
      using cheb_poly_orthogonality_discrete_aux[of "i - j"] le by simp
    also have "(∑k<n. cos (real (i + j) * (real (Suc (2 * k)) / real
(2 * n)) * pi)) =
               (if i = j ∧ i = 0 then real n else 0)"
      using cheb_poly_orthogonality_discrete_aux[of "i + j"] le by auto
    also have "(if i = j ∧ i = 0 then real n else 0) / 2 + (if i = j then
real n else 0) / 2 =
               (if i = j then if i = 0 then n else n / 2 else 0)"
      by auto
    finally show ?case .
  next
    case (sym j i)
    thus ?case
      by (simp only: eq_commute mult.commute) auto
  qed
qed auto
```

A similar relation holds for the Chebyshev polynomials of the second kind:

$$\sum_{k=0}^{n-1} U_i(x_k)U_j(x_k)(1-x_k^2) = \begin{cases} n & \text{if } i=j=n-1 \\ \frac{n}{2} & \text{if } i=j\neq 0 \\ 0 & \text{if } i\neq j \end{cases}$$

```
theorem cheb_poly'_orthogonality_discrete:
  fixes i j :: nat
  assumes "i < n" "j < n"
  shows "(∑k<n. cheb_poly' i (x k) * cheb_poly' j (x k) * (1 - x k ^
2)) =
          (if i = j then if i = n - 1 then n else n / 2 else 0)"
  using assms
proof (induction j i rule: linorder_wlog)
  case (le j i)
  have sin_pos: "sin (pi * (1 + real k * 2) / (real n * 2)) > 0" if "k
< n" for k
  proof -
    have "(1 + real k * 2) / (real n * 2) * pi < 1 * pi"
      by (intro mult_strict_right_mono) (use that in auto)
    thus ?thesis
      using that by (intro sin_gt_zero) (auto simp: mult_ac)
  qed

  have "(∑k<n. cheb_poly' i (x k) * cheb_poly' j (x k) * (1 - x k ^
2)) =
         (∑k<n. sin ((i+1) * real (Suc (2 * k)) / real (2 * n) * pi)
*
                 sin ((j+1) * real (Suc (2 * k)) / real (2 * n) * pi))"
  proof (intro sum.cong refl, goal_cases)
    case (1 k)
    thus ?case
      unfolding x_def cos_squared_eq using sin_pos[of k]
      by (auto simp: cheb_poly'_cos' x_def power2_eq_square mult_ac)
  qed
  also have "... = ((∑k<n. cos (real (i - j) * real (Suc (2 * k)) / real
(2 * n) * pi)) -
                   (∑k<n. cos (real (i + j + 2) * real (Suc (2 * k))
/ real (2 * n) * pi))) / 2"
    using le
    by (simp add: sin_times_sin sum_distrib_right sum_distrib_left algebra_simps

                   add_divide_distrib diff_divide_distrib sum_divide_distrib
of_nat_diff
             flip: sum_diff_distrib sum.distrib)
  also have "(∑k<n. cos (real (i - j) * real (Suc (2 * k)) / real (2
* n) * pi)) =
               (if i = j then real n else 0)"
    using cheb_poly_orthogonality_discrete_aux[of "i - j"] le by simp
```

```
  also have "(∑k<n. cos (real (i + j + 2) * real (Suc (2 * k)) / real
(2 * n) * pi)) =
             (if j = n - 1 then -real n else 0)"
  proof (cases "j = n - 1")
    case [simp]: True
    from le have [simp]: "i = n - 1"
      by auto
    have "(∑k<n. cos (real (i + j + 2) * real (Suc (2 * k)) / real (2
* n) * pi)) =
          (∑k<n. cos ((1 + 2 * real k) * pi))"
      by (simp add: of_nat_diff)
    also have "... = -real n"
      by (simp add: distrib_right)
    finally show ?thesis
      by auto
  next
    case False
    thus ?thesis using le
      by (subst cheb_poly_orthogonality_discrete_aux) auto
  qed
  also have "((if i = j then real n else 0) - (if j = n - 1 then - real
n else 0)) / 2 =
             (if i = j then if i = n - 1 then real n else real n / 2 else
0)"
    using le by auto
  finally show ?case .
next
  case (sym j i)
  thus ?case
    by (simp only: eq_commute mult.commute) auto
qed

end
```

We now show the continuous orthogonality relations. For the polynomials of the first kind, the relation is:

$$\int_{-1}^{1} \frac{T_m(x)T_n(x)}{\sqrt{1-x^2}}\,\mathrm{d}x = \begin{cases} \pi & \text{if } m = n = 0 \\ \frac{\pi}{2} & \text{if } m = n \neq 0 \\ 0 & \text{if } m \neq n \end{cases}$$

The proof works by a change of variables $x = \cos\theta$, which converts the integral to the easier form $\int_0^\pi \cos(mt)\cos(nt)\,\mathrm{d}x$, which can then be solved by a computing an indefinite integral (with appropriate case distinctions on *m* and *n*).

```
theorem cheb_poly_orthogonality:
  fixes m n :: nat
  defines "I ≡ if m = n then if m = 0 then pi else pi / 2 else 0"
```

```
  shows "((λx. cheb_poly m x * cheb_poly n x / sqrt (1 - x²)) has_integral
I) {-1..1}"
proof -
  let ?f = "λt::real. -cos (m * t) * cos (n * t)"
  let ?I = "integral {0..pi} (λt. cos (real m * t) * cos (real n * t))"

  have "finite {-1, 1 :: real}" "-1 ≤ (1::real)" "arccos ` {-1..1} ⊆
{0..pi}"
       "continuous_on {0..pi} ?f" "continuous_on {-1..1} arccos"
       "(⋀x. x ∈ {- 1..1} - {- 1, 1} ⟹
        (arccos has_real_derivative -inverse (sqrt (1 - x ^ 2))) (at x
within {- 1..1}))"
    by (auto intro!: arccos_lbound arccos_ubound continuous_intros derivative_eq_intros)

  from has_integral_substitution_general[OF this]
    have "((λx. cos (m * arccos x) * cos (n * arccos x) / sqrt (1 - x²))
has_integral ?I) {-1..1}"
    by (simp add: divide_simps)
  also have "?this ⟷ ((λx. cheb_poly m x * cheb_poly n x / sqrt (1 -
x²)) has_integral ?I) {-1..1}"
    by (intro has_integral_cong) (auto simp: cheb_poly_conv_cos)
  also consider "n = 0" "m = 0" | "n = m" "m ≠ 0" | "n ≠ m" by blast
  hence "?I = I"
  proof cases
    assume mn: "n = m" "m ≠ 0"
    let ?h = "λx::real. (2 * m * x + sin (2 * m * x)) / (4 * m)"
    have "(?h has_field_derivative cos (m * x) * cos (n * x)) (at x within
A)" for x :: real and A
    proof -
      have "(?h has_field_derivative (1 + cos (2 * (m * x))) / 2) (at
x within A)" using mn
        by (auto intro!: derivative_eq_intros simp: field_simps)
      also have "(1 + cos (2 * (m * x))) / 2 = cos (m * x) * cos (n *
x)" using mn
        by (subst cos_double_cos) (auto simp: power2_eq_square)
      finally show ?thesis .
    qed
    hence "((λt. cos (real m * t) * cos (real n * t)) has_integral (?h
pi - ?h 0)) {0..pi}"
      using mn by (intro fundamental_theorem_of_calculus)
                  (simp_all add: has_real_derivative_iff_has_vector_derivative)
    thus ?thesis using mn by (simp add: has_integral_iff I_def)
  next
    assume mn: "n ≠ m"
    let ?h = "λx::real. (m * sin (m * x) * cos (n * x) - n * cos (m *
x) * sin (n * x)) /
                (real m ^ 2 - real n ^ 2)"
    {
      fix x :: real and A :: "real set"
```

```
      have "m * (m * cos (m * x) * cos (n * x) - n * sin (m * x) * sin
(n * x)) -
            n * (n * cos (m * x) * cos (n * x) - m * sin (m * x) * sin
(n * x)) =
              cos (m * x) * cos (n * x) * (real m ^ 2 - real n ^ 2)"
        by (simp add: algebra_simps power2_eq_square)
      moreover from mn have "real m ^ 2 ≠ real n ^ 2" by simp
      ultimately have "(?h has_field_derivative cos (m * x) * cos (n *
x)) (at x within A)"
        by (auto intro!: derivative_eq_intros simp: divide_simps power2_eq_square
mult_ac)
    }
    hence "((λt. cos (real m * t) * cos (real n * t)) has_integral (?h
pi - ?h 0)) {0..pi}"
      using mn by (intro fundamental_theorem_of_calculus)
                  (simp_all add: has_real_derivative_iff_has_vector_derivative)
    thus ?thesis using mn by (simp add: has_integral_iff I_def)
  qed (simp_all add: I_def)
  finally show ?thesis .
qed
```

For the polynomials of the second kind, the relation is:

$$\int_{-1}^{1} U_m(x)U_n(x)\sqrt{1-x^2}\,\mathrm{d}x = \begin{cases} \frac{\pi}{2} & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases}$$

The proof works the same as before.

```
theorem cheb_poly'_orthogonality:
  fixes m n :: nat
  defines "I ≡ if m = n then pi / 2 else 0"
  shows "((λx. cheb_poly' m x * cheb_poly' n x * sqrt (1 - x²)) has_integral
I) {-1..1}"
proof -
  define h :: "nat ⇒ real ⇒ real" where
    "h = (λn x. if x = 0 then real n else if x = pi then (-1)^Suc n *
real n else sin (n * x) / sin x)"
  have h_eq: "h n x = sin (n * x) / sin x" if "x ∉ {0, pi}" for n x
    using that by (auto simp: h_def)
  have h_cont: "continuous_on {0..pi} (h n)" if "n > 0" for n
  proof -
    have "continuous (at x within {0..pi}) (h n)" if "x ∈ {0..pi}" for
x n
    proof -
      consider "x = 0" | "x = pi" | "x ∈ {0<..<pi}"
        using ‹x ∈ {0..pi}› by force
      thus ?thesis
      proof cases
        assume x: "x ∈ {0<..<pi}"
```

```
        have "isCont (λx. sin (n * x) / sin x) x"
          by (intro continuous_intros) (use x in ‹auto simp: sin_zero_pi_iff›)
        also from x have "∀F x in nhds x. x ∈ {0<..<pi}"
          by (intro eventually_nhds_in_open) auto
        hence "∀F x in nhds x. sin (real n * x) / sin x = h n x"
          by eventually_elim (auto simp: h_def)
        hence "isCont (λx. sin (n * x) / sin x) x ⟷ isCont (h n) x"
          by (intro isCont_cong)
        finally show ?thesis
          using continuous_at_imp_continuous_at_within by auto
      next
        assume [simp]: "x = 0"
        have "filterlim (λx::real. sin (n * x) / sin x) (nhds (of_nat
n)) (at_right 0)"
          by real_asymp
        also have "eventually (λx::real. x ∈ {0<..<pi}) (at_right 0)"
          by (rule eventually_at_right_real) auto
        hence "eventually (λx::real. sin (n * x) / sin x = h n x) (at_right
0)"
          by eventually_elim (auto simp: h_def)
        hence "filterlim (λx::real. sin (n * x) / sin x) (nhds (of_nat
n)) (at_right 0) ⟷
               filterlim (h n) (nhds (of_nat n)) (at_right 0)"
          by (intro filterlim_cong refl)
        finally have "filterlim (h n) (nhds (of_nat n)) (at 0 within {0..pi})"
          by (simp add: at_within_Icc_at_right)
        thus ?thesis
          by (simp add: continuous_within h_def)
      next
        assume [simp]: "x = pi"
        have "filterlim (λx::real. sin (n * x) / sin x) (nhds ((-1)^Suc
n * real n)) (at_left pi)"
          by real_asymp
        also have "eventually (λx::real. x ∈ {0<..<pi}) (at_left pi)"
          by (rule eventually_at_left_real) auto
        hence "eventually (λx::real. sin (n * x) / sin x = h n x) (at_left
pi)"
          by eventually_elim (auto simp: h_def)
        hence "filterlim (λx::real. sin (n * x) / sin x) (nhds ((-1)^Suc
n * real n)) (at_left pi) ⟷
               filterlim (h n) (nhds ((-1)^Suc n * real n)) (at_left pi)"
          by (intro filterlim_cong refl)
        finally have "filterlim (h n) (nhds ((-1)^Suc n * real n)) (at
pi within {0..pi})"
          by (simp add: at_within_Icc_at_left)
        thus ?thesis
          by (simp add: continuous_within h_def)
      qed
    qed
```

```
    thus ?thesis
      unfolding continuous_on_eq_continuous_within by blast
  qed

  define f where "f = (λt::real. -sin ((m+1) * t) * sin ((n+1) * t))"
  define g where "g = (λt. sin (real (m+1) * t) * sin (real (n+1) * t))"
  let ?I = "integral {0..pi} g"

  have "finite {-1, 1 :: real}" "-1 ≤ (1::real)" "arccos ` {-1..1} ⊆
{0..pi}"
       "continuous_on {0..pi} f" "continuous_on {-1..1} arccos"
       "(⋀x. x ∈ {- 1..1} - {- 1, 1} ⟹
        (arccos has_real_derivative -inverse (sqrt (1 - x ^ 2))) (at x
within {- 1..1}))"
    by (auto intro!: arccos_lbound arccos_ubound continuous_intros h_cont
derivative_eq_intros simp: f_def)

  from has_integral_substitution_general[OF this]
  have "((λx. - inverse (sqrt (1 - x²)) * (- sin ((m + 1) * arccos x)
* sin ((n + 1) * arccos x)))
          has_integral ?I) {-1..1}"
      by (simp add: divide_simps f_def g_def)
  have I: "((λx. cheb_poly' m x * cheb_poly' n x * sqrt (1 - x²)) has_integral
?I) {-1..1}"
  proof (rule has_integral_spike)
    show "negligible {1, -1 :: real}"
      by simp
    show "cheb_poly' m x * cheb_poly' n x * sqrt (1 - x²) =
          - inverse (sqrt (1 - x²)) * (- sin ((m + 1) * arccos x) * sin
((n + 1) * arccos x))"
      if "x ∈ {-1..1} - {1, -1}" for x :: real
      using that by (auto simp: arccos_eq_0_iff arccos_eq_pi_iff cheb_poly'_conv_cos
field_simps sin_arccos)
  qed fact+

  have sin_double'': "sin (x * (y * 2)) = 2 * sin (x * y) * cos (x * y)"
for x y :: real
    using sin_double[of "x * y"] by (simp add: mult_ac)
  have cos_double'': "cos (x * (y * 2)) = (cos (x * y))² - (sin (x * y))²"
for x y :: real
    using cos_double[of "x * y"] by (simp add: mult_ac)
  have sin_squared_eq': "sin x * sin x = 1 - cos x ^ 2" for x :: real
    using sin_squared_eq[of x] by algebra
  have sin_squared_eq'': "sin x * (sin x * y) = (1 - cos x ^ 2) * y" for
x y :: real
    using sin_squared_eq[of x] by algebra

  have "(g has_integral I) {0..pi}"
  proof (cases "m = n")
```

```
    case [simp]: True
    define G where "G = (λx::real. x/2 - sin (2*(n+1)*x)/(4*(n+1)))"
    have "(g has_integral (G pi - G 0)) {0..pi}"
      apply (rule fundamental_theorem_of_calculus)
      apply (auto simp: G_def g_def divide_simps simp flip: has_real_derivative_iff_has_vec
                  intro!: derivative_eq_intros)
      apply (auto simp: algebra_simps cos_add sin_add cos_multiple_reduce
sin_multiple_reduce
                       sin_double'' cos_double'' power2_eq_square sin_squared_eq'
sin_squared_eq'')
      done
    also have "G 0 = 0"
      by (simp add: G_def)
    also have "G pi = pi / 2 - sin (real (2 * (n + 1)) * pi) / real (4
* (n + 1))"
      unfolding G_def ..
    also have "sin (real (2 * (n + 1)) * pi) = 0"
      using sin_npi by blast
    finally show ?thesis
      by (simp add: I_def)
  next
    case False
    define G where "G = (λx::real. sin ((real m-real n)*x) / (2*(real
m-real n)) - sin ((2+m+n)*x)/(2*(2+m+n)))"
    have "(g has_integral (G pi - G 0)) {0..pi}"
      using False
      apply (intro fundamental_theorem_of_calculus)
      apply (auto simp: G_def g_def divide_simps simp flip: has_real_derivative_iff_has_vec
                  intro!: derivative_eq_intros)
      apply (auto simp: algebra_simps cos_add sin_add cos_diff sin_diff
cos_multiple_reduce sin_multiple_reduce
                       sin_double'' cos_double'' power2_eq_square sin_squared_eq'
sin_squared_eq'')
      done
    also have "G 0 = 0"
      by (simp add: G_def)
    also have "G pi = sin ((real m - real n) * pi) / (2 * (real m - real
n)) -
                     sin (real (2 + m + n) * pi) / real (2 * (2 + m +
n))"
      unfolding G_def by (simp add: G_def)
    also have "real m - real n = of_int (int m - int n)"
      by linarith
    also have "sin (... * pi) = 0"
      using sin_zero_iff_int2 by blast
    also have "sin (real (2 + m + n) * pi) = 0"
      using sin_npi by blast
    finally show ?thesis
      using False by (simp add: I_def)
```

```
  qed
  with I show ?thesis
    using integral_unique by blast
qed
```

We additionally show the following property about the integral from *-1* to *1*:

$$\int_{-1}^{1} T_n(x)\,\mathrm{d}x = \frac{1+(-1)^n}{1-n^2}$$

```
theorem cheb_poly_integral_neg1_1:
  "(cheb_poly n has_integral ((1 + (-1)^n) / (1 - n²))) {-1..1::real}"
proof -
  consider "n = 0" | "n = 1" | "n > 1"
    by linarith
  thus ?thesis
  proof cases
    assume [simp]: "n = 0"
    have "cheb_poly 0 = (λ_. 1 :: real)"
      by auto
    thus ?thesis
      by (auto simp: has_integral_iff_emeasure_lborel)
  next
    assume [simp]: "n = 1"
    have "cheb_poly 1 = (λx. x :: real)"
      by (auto simp: fun_eq_iff)
    thus ?thesis
      using ident_has_integral[of "-1" "1 :: real"] by simp
  next
    assume n: "n > 1"
    define P :: "real poly" where "P = smult (1/(2*(n+1))) (Cheb_poly
(n+1)) - smult (1/(2*(n-1))) (Cheb_poly (n-1))"
    have "(cheb_poly n has_integral (poly P 1 - poly P (-1))) {-1..1::real}"
    proof (rule fundamental_theorem_of_calculus)
      define a b where "a = n+1" and "b = n-1"
      have [simp]: "a ≠ 0" "b ≠ 0"
        using n by (auto simp: a_def b_def)
      have "pderiv P = smult (1 / 2) (Cheb_poly' (a-1) - Cheb_poly' (b-1))"
        using n unfolding P_def a_def [symmetric] b_def [symmetric]
        by (auto simp: P_def of_nat_diff pderiv_Cheb_poly pderiv_diff
pderiv_smult of_nat_mult_conv_smult smult_diff_right)
      also have "2 * ... = Cheb_poly' (a-1) - Cheb_poly' (b-1)"
        by (auto simp: numeral_mult_conv_smult)
      also have "... = 2 * Cheb_poly n"
        using Cheb_poly_rec[of n, where ?'a = real] cheb_poly'.P.simps(3)[of
"n - 2"] n
        by (simp add: a_def b_def numeral_2_eq_2)
      finally have [simp]: "pderiv P = Cheb_poly n"
```

```
        by simp
      show "(poly P has_vector_derivative cheb_poly n x) (at x within
{- 1..1})" for x
        unfolding cheb_poly.eval [symmetric] cheb_poly'.eval [symmetric]
                  has_real_derivative_iff_has_vector_derivative [symmetric]
        by (rule derivative_eq_intros refl)+ auto
    qed auto
    also have "real n ^ 2 ≠ 1"
      by (metis n nat_power_eq_Suc_0_iff numeral_1_eq_Suc_0 numeral_One
numeral_less_iff of_nat_1 of_nat_eq_iff of_nat_power semiring_norm(75)
zero_neq_numeral)
    hence "poly P 1 - poly P (-1) = (if even n then 2 / (1 - real n ^
2) else 0)"
      using n
      apply (simp add: P_def of_nat_diff minus_one_power_iff divide_simps
del: of_nat_Suc)
      apply (auto simp: field_simps power2_eq_square)
      done
    also have "... = (1 + (-1) ^ n) / (1 - real n ^ 2)"
      by auto
    finally show ?thesis .
  qed
qed
```

And, for the polynomials of the second kind:

$$\int_{-1}^{1} U_n(x)\,\mathrm{d}x = \frac{1+(-1)^n}{n+1}$$

```
theorem cheb_poly'_integral_neg1_1:
  "(cheb_poly' n has_integral (1 + (-1) ^ n) / (n+1)) {-1..1::real}"
proof -
  define F :: "real poly" where "F = smult (1 / of_nat (Suc n)) (Cheb_poly
(Suc n))"
  have [simp]: "pderiv F = Cheb_poly' n"
    by (auto simp: F_def pderiv_smult pderiv_Cheb_poly of_nat_mult_conv_smult
simp del: of_nat_Suc)
  have "(poly (Cheb_poly' n) has_integral (poly F 1 - poly F (-1))) {-1..1}"
    by (rule fundamental_theorem_of_calculus)
       (auto intro!: derivative_eq_intros simp flip: has_real_derivative_iff_has_vector_der
  also have "poly F 1 - poly F (-1) = (1 + (-1) ^ n) / (n+1)"
    by (simp add: F_def add_divide_distrib)
  finally show ?thesis
    by (simp add: add_ac)
qed
```

3.10 Clenshaw's algorithm

Clenshaw's algorithm allows us to efficiently evaluate a weighted sum of Chebyshev polynomials of the first kind, i.e.

$$\sum_{i=0}^{n} w_i \cdot T_i(x) \ .$$

This is useful when evaluating interpolations.

```
locale clenshaw =
  fixes g :: "nat ⇒ 'a :: comm_ring_1"
  fixes a b :: "nat ⇒ 'a"
  assumes g_rec: "⋀n. g (Suc (Suc n)) = a n * g (Suc n) + b n * g n"
begin

context
  fixes N :: nat and c :: "nat ⇒ 'a"
begin

function clenshaw_aux where
  "n ≥ N ⟹ clenshaw_aux n = 0"
| "n < N ⟹ clenshaw_aux n =
     c (Suc n) + a n * clenshaw_aux (n+1) + b (Suc n) * clenshaw_aux (n+2)"
  by force+
termination by (relation "Wellfounded.measure (λn. Suc N - n)") simp_all

lemma clenshaw_aux_correct_aux:
  assumes "n ≤ N"
  shows "g n * c n + g (Suc n) * clenshaw_aux n + b n * g n * clenshaw_aux
(Suc n) = (∑k=n..N. c k * g k)"
  using assms
proof (induction rule: inc_induct)
  case (step k)
  show ?case
  proof (cases "Suc k < N")
    case False
    with step.hyps have k: "k = N - 1" by simp
    from step.hyps have "{N - Suc 0..N} = {N - 1, N}" by auto
    with k show ?thesis using step.hyps by simp
  next
    case True
    have "(∑k = k..N. c k * g k) = c k * g k + (∑k = Suc k..N. c k
* g k)"
      using True by (intro sum.atLeast_Suc_atMost) auto
    also have "(∑k = Suc k..N. c k * g k) = g (Suc k) * c (Suc k) +
                 g (Suc (Suc k)) * clenshaw_aux (Suc k) + b (Suc k) *
g (Suc k) * clenshaw_aux (Suc (Suc k))"
      by (subst step.IH [symmetric]) simp_all
```

```
    also have "c k * g k + ... = g k * c k + g (Suc k) * clenshaw_aux k
+ b k * g k * clenshaw_aux (Suc k)"
      using step.prems step.hyps True by (simp add: algebra_simps g_rec)
    finally show ?thesis ..
  qed
qed auto

fun clenshaw_aux' where
  "clenshaw_aux' 0 acc1 acc2 = g 0 * c 0 + g 1 * acc1 + b 0 * g 0 * acc2"
| "clenshaw_aux' (Suc n) acc1 acc2 = clenshaw_aux' n (c (Suc n) + a n
* acc1 + b (Suc n) * acc2) acc1"

lemma clenshaw_aux'_correct: "clenshaw_aux' N 0 0 = (∑k≤N. c k * g
k)"
proof -
  have "(∑k≤N. c k * g k) = g 0 * c 0 + g 1 * clenshaw_aux 0 + b 0 *
g 0 * clenshaw_aux 1"
    using clenshaw_aux_correct_aux[of 0] by (simp add: atLeast0AtMost
clenshaw_def)
  moreover have "clenshaw_aux' n (clenshaw_aux n) (clenshaw_aux (Suc
n)) =
                   g 0 * c 0 + g 1 * clenshaw_aux 0 + b 0 * g 0 * clenshaw_aux
1"
    if "n ≤ N" for n using that by (induction n) auto
  from this[of N] have "g 0 * c 0 + g 1 * clenshaw_aux 0 + b 0 * g 0
* clenshaw_aux 1 =
                          clenshaw_aux' N 0 0" by simp
  ultimately show ?thesis by simp
qed

lemmas [simp del] = clenshaw_aux'.simps

end

lemma clenshaw_aux'_cong:
  "(⋀k. k ≤ n ⟹ c k = c' k) ⟹ clenshaw_aux' c n acc1 acc2 = clenshaw_aux'
c' n acc1 acc2"
  by (induction n acc1 acc2 rule: clenshaw_aux'.induct) (auto simp: clenshaw_aux'.simps)

definition clenshaw where  "clenshaw N c = clenshaw_aux' c N 0 0"

theorem clenshaw_correct: "clenshaw N c = (∑k≤N. c k * g k)"
  using clenshaw_aux'_correct by (simp add: clenshaw_def)

end


definition cheb_eval :: "'a :: comm_ring_1 list ⇒ 'a ⇒ 'a" where
  "cheb_eval cs x = (∑k<length cs. cs ! k * cheb_poly k x)"
```

```
interpretation cheb_poly: clenshaw "λn. cheb_poly n x" "λ_. 2 * x" "λ_.
-1"
  by standard (simp_all add: cheb_poly_simps)

fun cheb_eval_aux where
  "cheb_eval_aux 0 cs x acc1 acc2 = hd cs + x * acc1 - acc2"
| "cheb_eval_aux (Suc n) cs x acc1 acc2 =
     cheb_eval_aux n (tl cs) x (hd cs + 2 * x * acc1 - acc2) acc1"

lemma cheb_eval_aux_altdef:
  "length cs = Suc n ⟹
     cheb_eval_aux n cs x acc1 acc2 =
     cheb_poly.clenshaw_aux' x (λk. rev cs ! k) n acc1 acc2"
proof (induction n cs x acc1 acc2 rule: cheb_eval_aux.induct)
  case (1 cs x acc1 acc2)
  hence "hd cs = cs ! 0"
    by (intro hd_conv_nth) auto
  with 1 show ?case
    by (auto simp: rev_nth cheb_poly.clenshaw_aux'.simps)
next
  case (2 n cs x acc1 acc2)
  hence "hd cs = cs ! 0"
    by (intro hd_conv_nth) auto
  with 2 show ?case
    by (auto simp: rev_nth cheb_poly.clenshaw_aux'.simps nth_tl Suc_diff_le
             intro!: cheb_poly.clenshaw_aux'_cong)
qed

lemmas [simp del] = cheb_eval_aux.simps

lemma cheb_eval_code [code]:
  "cheb_eval [] x = 0"
  "cheb_eval [c] x = c"
  "cheb_eval (c1 # c2 # cs) x =
     cheb_eval_aux (Suc (length cs)) (rev (c1 # c2 # cs)) x 0 0"
proof -
  have "cheb_eval (c1 # c2 # cs) x =
          (∑k≤Suc (length cs). (c1 # c2 # cs) ! k * cheb_poly k x)"
    unfolding cheb_eval_def by (intro sum.cong) auto
  also have "... = cheb_eval_aux (Suc (length cs)) (rev (c1 # c2 # cs))
x 0 0"
    unfolding cheb_poly.clenshaw_def cheb_poly.clenshaw_correct [symmetric]
    using cheb_eval_aux_altdef[of "rev (c1 # c2 # cs)" "Suc (length cs)"
x 0 0]
    by (simp_all add: cheb_poly.clenshaw_def )
  finally show "cheb_eval (c1 # c2 # cs) x = ..." .
qed (simp_all add: cheb_eval_def)
```

end

References

[1] J. Mason and D. Handscomb. *Chebyshev Polynomials.* CRC Press, 2002.