Complex Lattices, Elliptic Functions, and the Modular Group

Manuel Eberl, Anthony Bordg, Lawrence C. Paulson, Wenda Li

September 1, 2025

Abstract

This entry defines complex lattices, i.e. $\Lambda(\omega_1, \omega_2) = \mathbb{Z}\omega_1 + \mathbb{Z}\omega_2$ where $\omega_1/\omega_2 \notin \mathbb{R}$. Based on this, various other related topics are covered:

- the modular group Γ and its fundamental region
- elliptic functions and their basic properties
- the Weierstraß elliptic function $\wp$ and the fact that every elliptic function can be written in terms of $\wp$
- the Eisenstein series G_n (including the forbidden series G_2)
- the ordinary differential equation satisfied by $\wp$, the recurrence relation for G_n, and the addition and duplication theorems for $\wp$
- the lattice invariants g_2, g_3, and Klein's J invariant
- the non-vanishing of the lattice discriminant Δ
- G_n, Δ, J as holomorphic functions in the upper half plane
- the Fourier expansion of $G_n(z)$ for $z \to i\infty$
- the functional equations of G_n, Δ, J, and η w.r.t. the modular group
- Dedekind's η function
- the inversion formulas for the Jacobi θ functions

In particular, this entry contains most of Chapters 1 and 3 from Apostol's *Modular Functions and Dirichlet Series in Number Theory* [1] and parts of Chapter 2.

The purpose of this entry is to provide a foundation for further formalisation of modular functions and modular forms.

Contents

1 Auxiliary material

1.1 The z-plane vs the q-disc

theory *Z_Plane_Q_Disc*
  imports *"HOL-Complex_Analysis.Complex_Analysis" "Theta_Functions.Nome"*
begin

In the study of modular forms and related subjects, we often need convert between the upper half of the complex plane (typically with a parameter written as z or τ) and the unit disc (with a parameter written as q).

This is particularly interesting for 1-periodic functions $f(z)$ (or more generally n-periodic functions where $n > 0$ is an integer) since such functions have a Fourier expansion in terms of q, i.e. we can view them both as functions $f(z)$ for $\mathrm{Im}(z) > 0$ or $f(q)$ for $|q| < 1$, where the latter is only well-defined due to the periodicity.

1.1.1 The neighbourhood of $i\infty$

The following filter describes the neighbourhood of $i\infty$, i.e. the neighbourhood of all points with sufficiently big imaginary value. In terms of q, this corresponds to the point $q = 0$.

definition *at_ii_inf :: "complex filter" ("at'_i∞")* **where**
  "at_ii_inf = filtercomap Im at_top"

lemma *eventually_at_ii_inf:*
  "eventually (λz. Im z > c) at_ii_inf"
  unfolding *at_ii_inf_def* **using** *filterlim_at_top_dense* **by** *blast*

lemma *eventually_at_ii_inf_iff:*
  "(∀F z in at_ii_inf. P z) ⟷ (∃c. ∀z. Im z > c ⟶ P z)"
  by *(simp add: at_ii_inf_def eventually_filtercomap_at_top_dense)*

lemma *eventually_at_ii_inf_iff':*
  "(∀F z in at_ii_inf. P z) ⟷ (∃c. ∀z. Im z ≥ c ⟶ P z)"
  by *(simp add: at_ii_inf_def eventually_filtercomap_at_top_linorder)*

lemma *filterlim_Im_at_ii_inf: "filterlim Im at_top at_i∞"*
  unfolding *at_ii_inf_def* **by** *(rule filterlim_filtercomap)*

lemma *filterlim_at_ii_infI:*
  assumes *"filterlim f F at_top"*
  shows *"filterlim (λx. f (Im x)) F at_i∞"*
  using *filterlim_filtercomapI[of f F at_top Im] assms* **by** *(simp add: at_ii_inf_def)*

lemma *filtermap_scaleR_at_ii_inf:*
  assumes *"c > 0"*
  shows *"filtermap (λz. c *R z) at_ii_inf = at_ii_inf"*

```
proof (rule antisym)
  show "filtermap (λz. c *R z) at_ii_inf ≤ at_ii_inf"
  proof (rule filter_leI)
    fix P
    assume "eventually P at_ii_inf"
    then obtain b where b: "⋀z. Im z > b ⟹ P z"
      by (auto simp: eventually_at_ii_inf_iff)
    have "eventually (λz. Im z > b / c) at_ii_inf"
      by (intro eventually_at_ii_inf)
    thus "eventually P (filtermap (λz. c *R z) at_ii_inf)"
      unfolding eventually_filtermap
      by eventually_elim (use assms in ‹auto intro!: b simp: field_simps›)
  qed
next
  show "filtermap (λz. c *R z) at_ii_inf ≥ at_ii_inf"
  proof (rule filter_leI)
    fix P
    assume "eventually P (filtermap (λz. c *R z) at_ii_inf)"
    then obtain b where b: "⋀z. Im z > b ⟹ P (c *R z)"
      by (auto simp: eventually_at_ii_inf_iff eventually_filtermap)
    have b': "P z" if "Im z > b * c" for z
      using b[of "z /R c"] that assms by (auto simp: field_simps)
    have "eventually (λz. Im z > b * c) at_ii_inf"
      by (intro eventually_at_ii_inf)
    thus "eventually P at_ii_inf"
      unfolding eventually_filtermap
      by eventually_elim (use assms in ‹auto intro!: b' simp: field_simps›)
  qed
qed

lemma at_ii_inf_neq_bot [simp]: "at_ii_inf ≠ bot"
  unfolding at_ii_inf_def by (intro filtercomap_neq_bot_surj surj_Im)
auto
```

1.1.2 The parameter q

The standard mapping from z to q is $z \mapsto \exp(2i\pi z)$, which is also sometimes referred to as the square of the *nome*. However, if the period of the function is $n > 01$, we have to opt for $z \mapsto \exp(2i\pi z/n)$ instead, so we allow this additional flexibility here.

Note that the inverse mapping from q to z is multivalued. We arbitrarily choose the strip with $\mathrm{Re}(z) \in (-\frac{1}{2}, \frac{1}{2}]$ as the codomain of the inverse mapping.

```
definition to_q :: "nat ⇒ complex ⇒ complex" where
  "to_q n τ = exp (2 * pi * i * τ / n)"

lemma to_nome_conv_to_q: "to_nome = to_q 2"
  by (auto simp: fun_eq_iff to_q_def to_nome_def mult_ac)
```

```
lemma to_q_conv_to_nome: "to_q n z = to_nome (2 * z / of_nat n)"
  by (simp add: to_q_def to_nome_def field_simps)

lemma to_q_add: "to_q n (w + z) = to_q n w * to_q n z"
  and to_q_diff: "to_q n (w - z) = to_q n w / to_q n z"
  and to_q_minus: "to_q n (-w) = inverse (to_q n w)"
  and to_q_power: "to_q n w ^ k = to_q n (of_nat k * w)"
  and to_q_power_int: "to_q n w powi m = to_q n (of_int m * w)"
  by (simp_all add: to_q_def add_divide_distrib diff_divide_distrib
                    exp_add exp_diff exp_minus ring_distribs mult_ac exp_power_int
              flip: exp_of_nat_mult)

interpretation to_q: periodic_fun_simple "to_q n" "of_nat n"
proof
  show "to_q n (z + of_nat n) = to_q n z" for z
    by (cases "n = 0") (simp_all add: to_q_def ring_distribs exp_add add_divide_distrib)
qed

lemma to_q_of_nat_period [simp]: "to_q n (of_nat n) = 1"
  by (simp add: to_q_def exp_eq_polar)

lemma to_q_of_int [simp]:
  assumes "int n dvd m"
  shows   "to_q n (of_int m) = 1"
proof -
  obtain k where m_eq: "m = int n * k"
    using assms by (elim dvdE)
  have "to_q n (of_int m) = to_q n (of_nat n * of_int k)"
    by (simp add: m_eq)
  also have "... = to_q n (of_nat n) powi k"
    by (simp only: to_q_power_int mult_ac)
  also have "... = 1"
    by simp
  finally show ?thesis .
qed

lemma to_q_of_nat [simp]:
  assumes "n dvd m"
  shows   "to_q n (of_nat m) = 1"
  using to_q_of_int[of n "int m"] assms by (simp del: to_q_of_int)

lemma to_q_numeral [simp]:
  assumes "n dvd numeral m"
  shows   "to_q n (numeral m) = 1"
  using to_q_of_nat[of n "numeral m"] assms by (simp del: to_q_of_nat)

lemma to_q_of_nat_period_1 [simp]: "w ∈ ℤ ⟹ to_q (Suc 0) w = 1"
  by (auto elim!: Ints_cases)
```

```
lemma Ln_to_q:
  assumes "x ∈ Re -` {n/2<..n/2}" "n > 0"
  shows "Ln (to_q n x) = 2 * pi * i * x / n"
unfolding to_q_def
proof (rule Ln_exp)
  have "-1/2 * pi < Re x / n * pi" "Re x / n * pi ≤ 1/2 * pi"
    using assms by (auto simp: field_simps)
  thus "-pi < Im (complex_of_real (2 * pi) * i * x / n)"
    using assms(2) by (auto simp: field_simps)
  show "Im (complex_of_real (2 * pi) * i * x / n) ≤ pi"
    using ‹Re x / n * pi ≤ 1/2 * pi› using assms(2) by (auto simp: field_simps)
qed

lemma to_q_nonzero [simp]: "to_q n τ ≠ 0"
  by (auto simp: to_q_def)

lemma norm_to_q [simp]: "norm (to_q n z) = exp (-2 * pi * Im z / n)"
  by (simp add: to_q_def)

lemma to_q_has_field_derivative [derivative_intros]:
  assumes [derivative_intros]: "(f has_field_derivative f') (at z)" and
n: "n > 0"
  shows   "((λz. to_q n (f z)) has_field_derivative (2 * pi * i * f' /
n * to_q n (f z))) (at z)"
  unfolding to_q_def by (rule derivative_eq_intros refl | (use n in simp;
fail))+

lemma deriv_to_q [simp]: "n > 0 ⟹ deriv (to_q n) z = 2 * pi * i / n
* to_q n z"
  by (auto intro!: DERIV_imp_deriv derivative_eq_intros)

lemma to_q_holomorphic_on [holomorphic_intros]:
  "f holomorphic_on A ⟹ n > 0 ⟹ (λz. to_q n (f z)) holomorphic_on
A"
  unfolding to_q_def by (intro holomorphic_intros) auto

lemma to_q_analytic_on [analytic_intros]:
  "f analytic_on A ⟹ n > 0 ⟹ (λz. to_q n (f z)) analytic_on A"
  unfolding to_q_def by (intro analytic_intros) auto

lemma to_q_continuous_on [continuous_intros]:
  "continuous_on A f ⟹ n > 0 ⟹ continuous_on A (λz. to_q n (f z))"
  unfolding to_q_def by (intro continuous_intros) auto

lemma to_q_continuous [continuous_intros]:
  "continuous F f ⟹ n > 0 ⟹ continuous F (λz. to_q n (f z))"
  unfolding to_q_def by (auto intro!: continuous_intros simp: divide_inverse)
```

```
lemma to_q_tendsto [tendsto_intros]:
  "(f ⟶ x) F ⟹ n > 0 ⟹ ((λz. to_q n (f z)) ⟶ to_q n x) F"
  unfolding to_q_def by (intro tendsto_intros) auto

lemma to_q_eq_to_qE:
  assumes "to_q m τ = to_q m τ'" "m > 0"
  obtains n where "τ' = τ + of_int n * of_nat m"
proof -
  from assms obtain n where "2 * pi * i * τ / m = 2 * pi * i * τ' / m
+ real_of_int (2 * n) * pi * i"
    using assms unfolding to_q_def exp_eq by blast
  also have "... = 2 * pi * i * (τ' / m + of_int n)"
    by (simp add: algebra_simps)
  also have "2 * pi * i * τ / m = 2 * pi * i * (τ / m)"
    by simp
  finally have "τ / m = τ' / m + of_int n"
    by (subst (asm) mult_cancel_left) auto
  hence "τ = τ' + of_int n * of_nat m"
    using ‹m > 0› by (metis divide_add_eq_iff divide_cancel_right of_nat_eq_0_iff
rel_simps(70))
  thus ?thesis
    by (intro that[of "-n"]) auto
qed

lemma to_q_inj_on_standard:
  assumes n: "n > 0"
  shows "inj_on (to_q n) (Re -` {-n/2..<n/2})"
  unfolding inj_on_def
proof safe+
  fix x y::complex
  assume eq: "to_q n x = to_q n y"
      and xRe: "Re x ∈ {- real n/2..<real n/2}" and yRe:"Re y ∈ {- real
n/2..<real n/2}"
  obtain rx ix where x:"x=Complex rx ix" by (meson complex.exhaust_sel)
  obtain ry iy where y:"y=Complex ry iy" by (meson complex.exhaust_sel)
  define pp where "pp= 2*pi"
  have rxry:"rx/n∈{-1/2..<1/2}" "ry/n∈{-1/2..<1/2}"
    using xRe yRe x y n by (auto simp: field_simps)

  define prx where "prx = pp*rx / n"
  define pry where "pry = pp*ry / n"

  have coseq:"exp (- (pp * ix / n)) * cos prx
        = exp (- (pp * iy / n)) * cos pry"
  proof -
    have "Re (to_q n x) = Re (to_q n y)"
      using eq by simp
    then show ?thesis
      unfolding x y to_q_def Re_exp Im_exp pp_def prx_def pry_def us-
```

```
ing assms
      by simp
  qed
  moreover have sineq:"exp (- (pp * ix / n)) * sin prx
       = exp (- (pp * iy / n)) * sin pry"
  proof -
    have "Im (to_q n x) = Im (to_q n y)"
      using eq by simp
    then show ?thesis
      unfolding x y to_q_def Re_exp Im_exp pp_def prx_def pry_def
      by simp
  qed
  ultimately have "(exp (- (pp * ix / n)) * sin (prx))
    / (exp (- (pp * ix / n)) * cos (prx))
    = (exp (- (pp * iy / n)) * sin (pry))
    / (exp (- (pp * iy / n)) * cos (pry))"
    by auto
  then have teq:"tan prx  = tan pry"
    by (subst (asm) (1 2) mult_divide_mult_cancel_left) (auto simp: tan_def)

  have sgn_eq_cos:"sgn (cos (prx)) = sgn (cos (pry))"
  proof -
    have "sgn (exp (- (pp * ix / n)) * cos prx)
        = sgn (exp (- (pp * iy / n)) * cos pry)"
      using coseq by simp
    then show ?thesis by (simp add:sgn_mult)
  qed
  have sgn_eq_sin:"sgn (sin (prx)) = sgn (sin (pry))"
  proof -
    have "sgn (exp (- (pp * ix / n)) * sin prx)
        = sgn (exp (- (pp * iy / n)) * sin pry)"
      using sineq by simp
    then show ?thesis by (simp add:sgn_mult)
  qed

  have "prx=pry" if "tan prx = 0"
  proof -
    define pi2 where "pi2 = pi /2"
    have [simp]: "cos pi2 = 0" "cos (-pi2) = 0"
      "sin pi2 = 1" "sin (-pi2) = -1"
      unfolding pi2_def by auto
    have "prx∈{-pi,-pi2,0,pi2}"
    proof -
      from tan_eq_0_Ex[OF that]
      obtain k0 where k0:"prx = real_of_int k0 / 2 * pi"
        by auto
      then have "rx / n = real_of_int k0 / 4"
        unfolding pp_def prx_def using n by (auto simp: field_simps)
      with rxry have "k0∈{-2,-1,0,1}"
```

```
      by auto
    then show ?thesis using k0 pi2_def by auto
  qed
  moreover have "pry∈{-pi,-pi2,0,pi2}"
  proof -
    from tan_eq_0_Ex that teq
    obtain k0 where k0:"pry = real_of_int k0 / 2 * pi"
      by auto
    then have "ry / n = real_of_int k0/4"
      unfolding pp_def pry_def using n by (auto simp: field_simps)
    with rxry have "k0∈{-2,-1,0,1}"
      by auto
    then show ?thesis using k0 pi2_def by auto
  qed
  moreover note sgn_eq_cos sgn_eq_sin
  ultimately show "prx=pry" by auto
qed
moreover have "prx=pry" if "tan prx ≠ 0"
proof -
  from tan_eq[OF teq that]
  obtain k0 where k0:"prx = pry + real_of_int k0 * pi"
    by auto
  then have "pi * (2 * rx / n) = pi* (2 * ry / n + real_of_int k0)"
    unfolding pp_def prx_def pry_def by (auto simp: algebra_simps)
  then have "real_of_int k0 = 2 * ((rx - ry) / n)"
    by (subst diff_divide_distrib, subst (asm) mult_left_cancel) (use
n in auto)
  also have "... ∈ {-2<..<2}"
    using rxry using n by (auto simp: field_simps)
  finally have "real_of_int k0 ∈ {- 2<..<2}" by simp
  then have "k0 ∈ {-1,0,1}" by auto
  then have "prx=pry-pi ∨ prx = pry ∨ prx = pry+pi"
    using k0 by auto
  moreover have False if "prx=pry-pi ∨ prx = pry+pi"
  proof -
    have "cos prx = - cos pry"
      using that by auto
    moreover note sgn_eq_cos
    ultimately have "cos prx = 0"
      by (simp add: sgn_zero_iff)
    then have "tan prx = 0" unfolding tan_def by auto
    then show False
      using ‹tan prx ≠ 0› unfolding prx_def by auto
  qed
  ultimately show "prx = pry" by auto
qed
ultimately have "prx=pry" by auto
then have "rx = ry" unfolding prx_def pry_def pp_def using n by auto
moreover have "ix = iy"
```

```
  proof -
    have "cos prx ≠0 ∨  sin prx≠0"
      using sin_zero_norm_cos_one by force
    then have "exp (- (pp * ix))  = exp (- (pp * iy))"
      using coseq sineq ‹prx = pry› n by auto
    then show "ix= iy" unfolding pp_def by auto
  qed
  ultimately show "x=y" unfolding x y by auto
qed

lemma filterlim_to_q_at_ii_inf' [tendsto_intros]:
  assumes n: "n > 0"
  shows   "filterlim (to_q n) (nhds 0) at_ii_inf"
proof -
  have "((λz. exp (- (2 * pi * Im z) / n)) ⟶ 0) at_ii_inf"
    unfolding at_ii_inf_def
    by (rule filterlim_compose[OF _ filterlim_filtercomap]) (use n in
real_asymp)
  hence "filterlim (λz. norm (to_q n z)) (nhds 0) at_ii_inf"
    by (simp add: to_q_def)
  thus ?thesis
    using tendsto_norm_zero_iff by blast
qed

lemma filterlim_to_q_at_ii_inf [tendsto_intros]: "n > 0 ⟹ filterlim
(to_q n) (at 0) at_ii_inf"
  using filterlim_to_q_at_ii_inf' by (auto simp: filterlim_at)

lemma eventually_to_q_neq:
  assumes n: "n > 0"
  shows "eventually (λw. to_q n w ≠ to_q n z) (at z)"
proof -
  have "eventually (λw. w ∈ ball z 1 - {z}) (at z)"
    by (intro eventually_at_in_open) auto
  thus ?thesis
  proof eventually_elim
    case (elim w)
    show ?case
    proof
      assume "to_q n w = to_q n z"
      then obtain m where eq: "z = w + of_int m * of_nat n"
        using n by (elim to_q_eq_to_qE)
      with elim have "real_of_int (|m| * int n) < 1"
        by (simp add: dist_norm norm_mult)
      hence "|m| * int n < 1"
        by linarith
      with n have "m = 0"
        by (metis add_0 mult_pos_pos not_less of_nat_0_less_iff zero_less_abs_iff
zless_imp_add1_zle)
```

```
      thus False
        using elim eq by simp
    qed
  qed
qed


lemma inj_on_to_q:
  assumes n: "n > 0"
  shows "inj_on (to_q n) (ball z (1/2))"
proof
  fix x y assume xy: "x ∈ ball z (1/2)" "y ∈ ball z (1/2)" "to_q n x
= to_q n y"
  from xy have "dist x z < 1 / 2" "dist y z < 1 / 2"
    by (auto simp: dist_commute)
  hence "dist x y < 1 / 2 + 1 / 2"
    using dist_triangle_less_add by blast
  moreover obtain m where m: "y = x + of_int m * of_nat n"
    by (rule to_q_eq_to_qE[OF xy(3) n]) auto
  ultimately have "real_of_int (|m| * int n) < 1"
    by (auto simp: dist_norm norm_mult)
  hence "|m| * int n < 1"
    by linarith
  with n have "m = 0"
    by (metis add_0 mult_pos_pos not_less of_nat_0_less_iff zero_less_abs_iff
zless_imp_add1_zle)
  thus "x = y"
    using m by simp
qed

lemma filtermap_to_q_nhds:
  assumes n: "n > 0"
  shows "filtermap (to_q n) (nhds z) = nhds (to_q n z)"
proof (rule filtermap_nhds_open_map')
  show "open (ball z (1 / 2))" "z ∈ ball z (1 / 2)" "isCont (to_q n)
z"
    using n by (auto intro!: continuous_intros)
  show "open (to_q n ‘ S)" if "open S" "S ⊆ ball z (1 / 2)" for S
  proof (rule open_mapping_thm3)
    show "inj_on (to_q n) S"
      using that by (intro inj_on_subset[OF inj_on_to_q] n)
  qed (use that n in ‹auto intro!: holomorphic_intros›)
qed

lemma filtermap_to_q_at:
  assumes n: "n > 0"
  shows "filtermap (to_q n) (at z) = at (to_q n z)"
  using filtermap_to_q_nhds
proof (rule filtermap_nhds_eq_imp_filtermap_at_eq)
```

```
  show "eventually (λx. to_q n x = to_q n z ⟶ x = z) (at z)"
    using eventually_to_q_neq[OF n, of z] by eventually_elim (use n in
auto)
qed fact+

lemma is_pole_to_q_iff:
  assumes n: "n > 0"
  shows "is_pole f (to_q n x) ⟷ is_pole (f o to_q n) x"
proof -
  have "filtermap f (at (to_q n x))
        = filtermap f (filtermap (to_q n) (at x)) "
    unfolding filtermap_to_q_at[OF n] by simp
  also have "... = filtermap (f ∘ to_q n) (at x)"
    unfolding filtermap_filtermap unfolding comp_def by simp
  finally show ?thesis unfolding is_pole_def filterlim_def by simp
qed

definition of_q :: "nat ⇒ complex ⇒ complex" where
  "of_q n q = ln q / (2 * pi * i / n)"

lemma Im_of_q: "q ≠ 0 ⟹ n > 0 ⟹ Im (of_q n q) = -n * ln (norm q)
/ (2 * pi)"
  by (simp add: of_q_def Im_divide power2_eq_square)

lemma Im_of_q_gt:
  assumes "norm q < exp (-2 * pi * c / n)" "q ≠ 0" "n > 0"
  shows   "Im (of_q n q) > c"
proof -
  have "-Im (of_q n q) = n * ln (norm q) / (2 * pi)"
    using assms by (subst Im_of_q) auto
  also have "ln (norm q) < ln (exp (-2 * pi * c / n))"
    by (subst ln_less_cancel_iff) (use assms in auto)
  hence "n * ln (norm q) / (2 * pi) < n * ln (exp (-2 * pi * c / n)) /
(2 * pi)"
    using ‹n > 0› by (intro mult_strict_left_mono divide_strict_right_mono)
auto
  also have "... = -c"
    using ‹n > 0› by simp
  finally show ?thesis
    by simp
qed

lemma to_q_of_q [simp]: "q ≠ 0 ⟹ n > 0 ⟹ to_q n (of_q n q) = q"
  by (simp add: to_q_def of_q_def)

lemma of_q_to_q:
  assumes "m > 0"
  shows "∃n. of_q m (to_q m τ) = τ + of_int n * of_nat m"
proof
```

```
  show "of_q m (to_q m τ) =
         τ + of_int (-unwinding (complex_of_real (2 * pi) * i * τ / m))
* of_nat m"
    using unwinding[of "2 * pi * i * τ / m"] assms
    by (simp add: of_q_def to_q_def field_simps)
qed

lemma filterlim_norm_at_0: "filterlim norm (at_right 0) (at 0)"
  unfolding filterlim_at
  by (auto simp: eventually_at tendsto_norm_zero_iff intro: exI[of _ 1])

lemma filterlim_of_q_at_0:
  assumes n: "n > 0"
  shows "filterlim (of_q n) at_ii_inf (at 0)"
proof -
  have "filterlim (λq. -n * ln (norm q) / (2 * pi)) at_top (at 0)"
    by (rule filterlim_compose[OF _ filterlim_norm_at_0]) (use n in real_asymp)
  also have "eventually (λq::complex. q ≠ 0) (at 0)"
    by (auto simp: eventually_at intro: exI[of _ 1])
  hence "eventually (λq. -n * ln (norm q) / (2 * pi) = Im (of_q n q))
(at 0)"
    by eventually_elim (use n in ‹simp add: Im_of_q›)
  hence "filterlim (λq::complex. -n * ln (norm q) / (2 * pi)) at_top (at
0) ⟷
        filterlim (λq. Im (of_q n q)) at_top (at 0)"
    by (intro filterlim_cong) auto
  finally show ?thesis
    by (simp add: at_ii_inf_def filterlim_filtercomap_iff o_def)
qed

lemma at_ii_inf_filtermap:
  assumes "n > 0"
  shows   "filtermap (to_q n) at_ii_inf = at 0"
proof (rule filtermap_fun_inverse[OF  filterlim_of_q_at_0 filterlim_to_q_at_ii_inf])
  have "eventually (λx. x ≠ 0) (at (0 :: complex))"
    by (rule eventually_neq_at_within)
  thus "∀F x in at 0. to_q n (of_q n x) = x"
    by eventually_elim (use assms in auto)
qed fact+

lemma eventually_at_ii_inf_to_q:
  assumes n: "n > 0"
  shows "eventually P (at 0) = (∀F x in at_ii_inf. P (to_q n x))"
proof -
  have "(∀F x in at_ii_inf. P (to_q n x)) ⟷ (∀F x in filtermap (to_q
n) at_ii_inf. P x)"
    by (simp add: eventually_filtermap)
  also have "... ⟷ eventually P (at 0)"
    by (simp add: at_ii_inf_filtermap n)
```

```
  finally show ?thesis ..
qed

lemma of_q_tendsto:
  assumes "x ∈ Re -` {real n / 2<..real n / 2}" "n > 0"
  shows "of_q n −to_q n x→ x"
proof -
  obtain rx ix where x:"x = Complex rx ix"
    using complex.exhaust_sel by blast
  have "Re (to_q n x) > 0" if "Im (to_q n x) = 0"
  proof -
    have "sin (2 * pi * rx / n) = 0"
      using that unfolding to_q_def x Im_exp Re_exp by simp
    then obtain k where "pi * (2 * rx / n) = pi * real_of_int k"
      unfolding sin_zero_iff_int2 by (auto simp: algebra_simps)
    hence k: "2 * rx / n = real_of_int k"
      using mult_cancel_left pi_neq_zero by blast
    moreover have "2*rx/n ∈ {-1<..<1}"
      using assms unfolding x by simp
    ultimately have "k=0" by auto
    then have "rx=0" using k ‹n > 0› by auto
    then show ?thesis unfolding to_q_def x
      using Re_exp by simp
  qed
  then have "to_q n x ∉ ℝ≤0"
    unfolding complex_nonpos_Reals_iff
    unfolding Im_exp Re_exp to_q_def
    by (auto simp add: complex_nonpos_Reals_iff)
  moreover have "Ln (to_q n x) / (2 * complex_of_real pi * i / n) = x"

    by (subst Ln_to_q) (use assms in ‹auto simp: field_simps›)
  ultimately show "of_q n −to_q n x→ x"
    unfolding of_q_def by (auto intro!: tendsto_eq_intros)
qed

lemma of_q_to_q_Re:
  assumes "x ∈ Re -` {real n / 2<..real n / 2}" "n > 0"
  shows "of_q n (to_q n x) = x"
proof -
  have "- pi < Im (complex_of_real (2 * pi) * i * x / n)"
  proof -
    have "pi* (-1/2) < pi * (Re x / n)"
      by (rule mult_strict_left_mono) (use assms in auto)
    then show ?thesis by auto
  qed
  moreover have "Im (complex_of_real (2 * pi) * i * x / n) ≤ pi"
    using assms by auto
  ultimately show ?thesis using assms(2)
    unfolding to_q_def of_q_def
```

```
    by (subst Ln_exp) auto
qed

end
```

1.2 Parallelogram-shaped paths

```
theory Parallelogram_Paths
  imports "HOL-Complex_Analysis.Complex_Analysis"
begin

definition parallelogram_path :: "'a :: real_normed_vector ⇒ 'a ⇒ 'a
⇒ real ⇒ 'a" where
  "parallelogram_path z a b =
     linepath z (z + a) +++ linepath (z + a) (z + a + b) +++
     linepath (z + a + b) (z + b) +++ linepath (z + b) z"

lemma path_parallelogram_path [intro]: "path (parallelogram_path z a
b)"
  and valid_path_parallelogram_path [intro]: "valid_path (parallelogram_path
z a b)"
  and pathstart_parallelogram_path [simp]: "pathstart (parallelogram_path
z a b) = z"
  and pathfinish_parallelogram_path [simp]: "pathfinish (parallelogram_path
z a b) = z"
  by (auto simp: parallelogram_path_def intro!: valid_path_join)

lemma parallelogram_path_altdef:
  fixes z a b :: complex
  defines "g ≡ (λw. z + Re w *R a + Im w *R b)"
  shows   "parallelogram_path z a b = g ∘ rectpath 0 (1 + i)"
  by (simp add: parallelogram_path_def rectpath_def g_def Let_def o_def
joinpaths_def
                fun_eq_iff linepath_def scaleR_conv_of_real algebra_simps)

lemma
  fixes f :: "complex ⇒ complex" and z ω1 ω2 :: complex
  defines "I ≡ (λa b. contour_integral (linepath (z + a) (z + b)) f)"
  defines "P ≡ parallelogram_path z ω1 ω2"
  assumes "continuous_on (path_image P) f"
  shows contour_integral_parallelogram_path:
          "contour_integral P f =
             (I 0 ω1 - I ω2 (ω1 + ω2)) - (I 0 ω2 - I ω1 (ω1 + ω2))"
    and contour_integral_parallelogram_path':
          "contour_integral P f =
             contour_integral (linepath z (z + ω1)) (λx. f x - f (x +
ω2)) -
             contour_integral (linepath z (z + ω2)) (λx. f x - f (x +
ω1))"
```

```
proof -
  define L where "L = (λa b. linepath (z + a) (z + b))"
  have I: "(f has_contour_integral (I a b)) (L a b)"
    if "closed_segment (z + a) (z + b) ⊆ path_image P" for a b
    unfolding I_def L_def
    by (intro has_contour_integral_integral contour_integrable_continuous_linepath
              continuous_on_subset[OF assms(3)] that)
  have "(f has_contour_integral
          (I 0 ω1 + (I ω1 (ω1 + ω2) + ((-I ω2 (ω1 + ω2)) + (-I 0 ω2)))))
          (L 0 ω1 +++ L ω1 (ω1 + ω2) +++ reversepath (L ω2 (ω1 + ω2))
+++ reversepath (L 0 ω2))"
    unfolding P_def parallelogram_path_def
    by (intro has_contour_integral_join valid_path_join valid_path_linepath
has_contour_integral_reversepath I)
       (auto simp: parallelogram_path_def path_image_join closed_segment_commute
P_def L_def add_ac)
  thus "contour_integral P f =
          (I 0 ω1 - I ω2 (ω1 + ω2)) - (I 0 ω2 - I ω1 (ω1 + ω2))"
    by (intro contour_integral_unique)
       (simp_all add: parallelogram_path_def P_def L_def algebra_simps)

  also have "I 0 ω2 - I ω1 (ω1 + ω2) = contour_integral (L 0 ω2) (λx.
f x - f (x + ω1))"
  proof -
    have "(f has_contour_integral I ω1 (ω1 + ω2)) (L ω1 (ω1 + ω2))"
      by (rule I) (auto simp: parallelogram_path_def path_image_join P_def
add_ac)
    also have "L ω1 (ω1 + ω2) = (+) ω1 ∘ L 0 ω2"
      by (simp add: linepath_def L_def fun_eq_iff algebra_simps)
    finally have "((λx. f (x + ω1)) has_contour_integral I ω1 (ω1 + ω2))
(L 0 ω2)"
      unfolding has_contour_integral_translate .
    hence "((λx. f x - f (x + ω1)) has_contour_integral (I 0 ω2 - I ω1
(ω1 + ω2))) (L 0 ω2)"
      by (intro has_contour_integral_diff I)
         (auto simp: parallelogram_path_def path_image_join closed_segment_commute
L_def P_def)
    thus ?thesis
      by (rule contour_integral_unique [symmetric])
  qed

  also have "I 0 ω1 - I ω2 (ω1 + ω2) = contour_integral (L 0 ω1) (λx.
f x - f (x + ω2))"
  proof -
    have "(f has_contour_integral I ω2 (ω1 + ω2)) (L ω2 (ω1 + ω2))"
      by (rule I) (auto simp: parallelogram_path_def path_image_join closed_segment_commute
P_def L_def add_ac)
    also have "L ω2 (ω1 + ω2) = (+) ω2 ∘ L 0 ω1"
      by (simp add: linepath_def L_def o_def algebra_simps)
```

```
    finally have "((λx. f (x + ω2)) has_contour_integral I ω2 (ω1 + ω2))
(L 0 ω1)"
      unfolding has_contour_integral_translate .
    hence "((λx. f x - f (x + ω2)) has_contour_integral (I 0 ω1 - I ω2
(ω1 + ω2))) (L 0 ω1)"
      by (intro has_contour_integral_diff I)
         (auto simp: parallelogram_path_def path_image_join closed_segment_commute
P_def)
    thus ?thesis
      by (rule contour_integral_unique [symmetric])
  qed

  finally show "contour_integral P f =
                  contour_integral (linepath z (z + ω1)) (λx. f x - f
(x + ω2)) -
                  contour_integral (linepath z (z + ω2)) (λx. f x - f
(x + ω1))"
    by (simp add: L_def add_ac)
qed

end
```

2 Möbius transforms and the modular group

```
theory Modular_Group
imports
  "HOL-Complex_Analysis.Complex_Analysis"
  "HOL-Number_Theory.Number_Theory"
begin
```

2.1 Basic properties of Möbius transforms

```
lemma moebius_uminus [simp]: "moebius (-a) (-b) (-c) (-d) = moebius a
b c d"
  by (simp add: fun_eq_iff moebius_def divide_simps) (simp add: algebra_simps
add_eq_0_iff2)

lemma moebius_uminus': "moebius (-a) b c d = moebius a (-b) (-c) (-d)"
  by (simp add: fun_eq_iff moebius_def divide_simps) (simp add: algebra_simps
add_eq_0_iff2)

lemma moebius_diff_eq:
  fixes a b c d :: "'a :: field"
  defines "f ≡ moebius a b c d"
  assumes *: "c = 0 ∨ z ≠ -d / c ∧ w ≠ -d / c"
  shows   "f w - f z = (a * d - b * c) / ((c * w + d) * (c * z + d)) *
(w - z)"
  using * by (auto simp: moebius_def divide_simps f_def add_eq_0_iff)
             (auto simp: algebra_simps)
```

```
lemma continuous_on_moebius [continuous_intros]:
  fixes a b c d :: "'a :: real_normed_field"
  assumes "c ≠ 0 ∨ d ≠ 0" "c = 0 ∨ -d / c ∉ A"
  shows   "continuous_on A (moebius a b c d)"
  unfolding moebius_def
  by (intro continuous_intros) (use assms in ‹auto simp: add_eq_0_iff›)

lemma continuous_on_moebius' [continuous_intros]:
  fixes a b c d :: "'a :: real_normed_field"
  assumes "continuous_on A f" "c ≠ 0 ∨ d ≠ 0" "⋀z. z ∈ A ⟹ c = 0
∨ f z ≠ -d / c"
  shows   "continuous_on A (λx. moebius a b c d (f x))"
proof -
  have "continuous_on A (moebius a b c d ∘ f)" using assms
    by (intro continuous_on_compose assms continuous_on_moebius) force+
  thus ?thesis
    by (simp add: o_def)
qed

lemma holomorphic_on_moebius [holomorphic_intros]:
  assumes "c ≠ 0 ∨ d ≠ 0" "c = 0 ∨ -d / c ∉ A"
  shows   "(moebius a b c d) holomorphic_on A"
  unfolding moebius_def
  by (intro holomorphic_intros) (use assms in ‹auto simp: add_eq_0_iff›)

lemma holomorphic_on_moebius' [holomorphic_intros]:
  assumes "f holomorphic_on A" "c ≠ 0 ∨ d ≠ 0" "⋀z. z ∈ A ⟹ c =
0 ∨ f z ≠ -d / c"
  shows   "(λx. moebius a b c d (f x)) holomorphic_on A"
proof -
  have "(moebius a b c d ∘ f) holomorphic_on A" using assms
    by (intro holomorphic_on_compose assms holomorphic_on_moebius) force+
  thus ?thesis
    by (simp add: o_def)
qed

lemma analytic_on_moebius [analytic_intros]:
  assumes "c ≠ 0 ∨ d ≠ 0" "c = 0 ∨ -d / c ∉ A"
  shows   "(moebius a b c d) analytic_on A"
  unfolding moebius_def
  by (intro analytic_intros) (use assms in ‹auto simp: add_eq_0_iff›)

lemma analytic_on_moebius' [analytic_intros]:
  assumes "f analytic_on A" "c ≠ 0 ∨ d ≠ 0" "⋀z. z ∈ A ⟹ c = 0 ∨
f z ≠ -d / c"
  shows   "(λx. moebius a b c d (f x)) analytic_on A"
proof -
```

```
  have "(moebius a b c d ∘ f) analytic_on A" using assms
    by (intro analytic_on_compose assms analytic_on_moebius) force+
  thus ?thesis
    by (simp add: o_def)
qed

lemma moebius_has_field_derivative:
  assumes "c = 0 ∨ x ≠ -d / c" "c ≠ 0 ∨ d ≠ 0"
  shows   "(moebius a b c d has_field_derivative (a * d - b * c) / (c
* x + d) ^ 2) (at x within A)"
  unfolding moebius_def
  apply (rule derivative_eq_intros refl)+
  using assms
   apply (auto simp: divide_simps add_eq_0_iff power2_eq_square split:
if_splits)
  apply (auto simp: algebra_simps)?
  done
```

2.2 Unimodular Möbius transforms

A unimodular Möbius transform has integer coefficients and determinant ± 1.

```
locale unimodular_moebius_transform =
  fixes a b c d :: int
  assumes unimodular: "a * d - b * c = 1"
begin

definition φ :: "complex ⇒ complex" where
  "φ = moebius (of_int a) (of_int b) (of_int c) (of_int d)"

lemma cnj_φ: "φ (cnj z) = cnj (φ z)"
  by (simp add: moebius_def φ_def)

lemma Im_transform:
  "Im (φ z) = Im z / norm (of_int c * z + of_int d) ^ 2"
proof -
  have "Im (φ z) = Im z * of_int (a * d - b * c) /
                   ((real_of_int c * Re z + real_of_int d)^2 + (real_of_int
c * Im z)^2)"
    by (simp add: φ_def moebius_def Im_divide algebra_simps)
  also have "a * d - b * c = 1"
    using unimodular by simp
  also have "((real_of_int c * Re z + real_of_int d)^2 + (real_of_int c
* Im z)^2) =
             norm (of_int c * z + of_int d) ^ 2"
    unfolding cmod_power2 by (simp add: power2_eq_square algebra_simps)
  finally show ?thesis
    by simp
qed
```

```
lemma Im_transform_pos_aux:
  assumes "Im z ≠ 0"
  shows   "of_int c * z + of_int d ≠ 0"
proof (cases "c = 0")
  case False
  hence "Im (of_int c * z + of_int d) ≠ 0"
    using assms by auto
  moreover have "Im 0 = 0"
    by simp
  ultimately show ?thesis
    by metis
next
  case True
  thus ?thesis using unimodular
    by auto
qed

lemma Im_transform_pos: "Im z > 0 ⟹ Im (φ z) > 0"
  using Im_transform_pos_aux[of z] by (auto simp: Im_transform)

lemma Im_transform_neg: "Im z < 0 ⟹ Im (φ z) < 0"
  using Im_transform_pos[of "cnj z"] by (simp add: cnj_φ)

lemma Im_transform_zero_iff [simp]: "Im (φ z) = 0 ⟷ Im z = 0"
  using Im_transform_pos_aux[of z] by (auto simp: Im_transform)

lemma Im_transform_pos_iff [simp]: "Im (φ z) > 0 ⟷ Im z > 0"
  using Im_transform_pos[of z] Im_transform_neg[of z] Im_transform_zero_iff[of
z]
  by (cases "Im z" "0 :: real" rule: linorder_cases) (auto simp del: Im_transform_zero_iff)

lemma Im_transform_neg_iff [simp]: "Im (φ z) < 0 ⟷ Im z < 0"
  using Im_transform_pos_iff[of "cnj z"] by (simp add: cnj_φ)

lemma Im_transform_nonneg_iff [simp]: "Im (φ z) ≥ 0 ⟷ Im z ≥ 0"
  using Im_transform_neg_iff[of z] by linarith

lemma Im_transform_nonpos_iff [simp]: "Im (φ z) ≤ 0 ⟷ Im z ≤ 0"
  using Im_transform_pos_iff[of z] by linarith

lemma transform_in_reals_iff [simp]: "φ z ∈ ℝ ⟷ z ∈ ℝ"
  using Im_transform_zero_iff[of z] by (auto simp: complex_is_Real_iff)

end


lemma Im_one_over_neg_iff [simp]: "Im (1 / z) < 0 ⟷ Im z > 0"
proof -
```

```
  interpret unimodular_moebius_transform 0 1 "-1" 0
    by standard auto
  show ?thesis
    using Im_transform_pos_iff[of z] by (simp add: φ_def moebius_def)
qed


locale inverse_unimodular_moebius_transform = unimodular_moebius_transform
begin

sublocale inv: unimodular_moebius_transform d "-b" "-c" a
  by unfold_locales (use unimodular in Groebner_Basis.algebra)

lemma inv_φ:
  assumes "of_int c * z + of_int d ≠ 0"
  shows   "inv.φ (φ z) = z"
  using unimodular assms
  unfolding inv.φ_def φ_def of_int_minus
  by (subst moebius_inverse) (auto simp flip: of_int_mult)

lemma inv_φ':
  assumes "of_int c * z - of_int a ≠ 0"
  shows   "φ (inv.φ z) = z"
  using unimodular assms
  unfolding inv.φ_def φ_def of_int_minus
  by (subst moebius_inverse') (auto simp flip: of_int_mult)

end
```

2.3 The modular group

2.3.1 Definition

We define the modular group as a quotient of all integer tuples (a, b, c, d) with $ad - bc = 1$ over a relation that identifies (a, b, c, d) with $(-a, -b, -c, -d)$.

```
definition modgrp_rel :: "int × int × int × int ⇒ int × int × int
× int ⇒ bool" where
  "modgrp_rel =
     (λ(a,b,c,d) (a',b',c',d'). a * d - b * c = 1 ∧
                                ((a,b,c,d) = (a',b',c',d') ∨ (a,b,c,d)
= (-a',-b',-c',-d')))"

lemma modgrp_rel_same_iff: "modgrp_rel x x ⟷ (case x of (a,b,c,d)
⇒ a * d - b * c = 1)"
  by (auto simp: modgrp_rel_def)

lemma part_equivp_modgrp_rel: "part_equivp modgrp_rel"
  unfolding part_equivp_def
proof safe
```

```
  show "∃x. modgrp_rel x x"
    by (rule exI[of _ "(1,0,0,1)"]) (auto simp: modgrp_rel_def)
qed (auto simp: modgrp_rel_def case_prod_unfold fun_eq_iff)


quotient_type modgrp = "int × int × int × int" / partial: modgrp_rel
  by (fact part_equivp_modgrp_rel)

instantiation modgrp :: one
begin

lift_definition one_modgrp :: modgrp is "(1, 0, 0, 1)"
  by (auto simp: modgrp_rel_def)

instance ..
end


instantiation modgrp :: times
begin

lift_definition times_modgrp :: "modgrp ⇒ modgrp ⇒ modgrp"
  is "λ(a,b,c,d) (a',b',c',d'). (a * a' + b * c', a * b' + b * d', c *
a' + d * c', c * b' + d * d')"
  unfolding modgrp_rel_def case_prod_unfold prod_eq_iff fst_conv snd_conv
  by (elim conjE disjE; intro conjI) (auto simp: algebra_simps)

instance ..
end


instantiation modgrp :: inverse
begin

lift_definition inverse_modgrp :: "modgrp ⇒ modgrp"
  is "λ(a, b, c, d). (d, -b, -c, a)"
  by (auto simp: modgrp_rel_def algebra_simps)

definition divide_modgrp :: "modgrp ⇒ modgrp ⇒ modgrp" where
  "divide_modgrp x y = x * inverse y"

instance ..

end


interpretation modgrp: Groups.group "(*) :: modgrp ⇒ _" 1 inverse
proof
  show "a * b * c = a * (b * c :: modgrp)" for a b c
```

```
    by (transfer; unfold modgrp_rel_def case_prod_unfold prod_eq_iff fst_conv
snd_conv;
        intro conjI; elim conjE disjE)
       (auto simp: algebra_simps)
next
  show "1 * a = a" for a :: modgrp
    by transfer (auto simp: modgrp_rel_def)
next
  show "inverse a * a = 1" for a :: modgrp
    by (transfer; unfold modgrp_rel_def case_prod_unfold prod_eq_iff fst_conv
snd_conv;
        intro conjI; elim conjE disjE)
       (auto simp: algebra_simps)
qed


instance modgrp :: monoid_mult
  by standard (simp_all add: modgrp.assoc)

lemma inverse_power_modgrp: "inverse (x ^ n :: modgrp) = inverse x ^
n"
  by (induction n) (simp_all add: algebra_simps modgrp.inverse_distrib_swap
power_commuting_commutes)
```

2.3.2 Basic operations

Application to a field

```
lift_definition apply_modgrp :: "modgrp ⇒ 'a :: field ⇒ 'a" is
  "λ(a,b,c,d). moebius (of_int a) (of_int b) (of_int c) (of_int d)"
  by (auto simp: modgrp_rel_def)
```

The shift operation $z \mapsto z + n$

```
lift_definition shift_modgrp :: "int ⇒ modgrp" is "λn. (1, n, 0, 1)"
  by transfer (auto simp: modgrp_rel_def)
```

The shift operation $z \mapsto z + 1$

```
lift_definition T_modgrp :: modgrp is "(1, 1, 0, 1)"
  by transfer (auto simp: modgrp_rel_def)
```

The operation $z \mapsto -\frac{1}{z}$

```
lift_definition S_modgrp :: modgrp is "(0, -1, 1, 0)"
  by transfer (auto simp: modgrp_rel_def)
```

Whether or not the transformation has a pole in the complex plane

```
lift_definition is_singular_modgrp :: "modgrp ⇒ bool" is "λ(a,b,c,d).
c ≠ 0"
  by transfer (auto simp: modgrp_rel_def)
```

The position of the transformation's pole in the complex plane (if it has one)

lift_definition *pole_modgrp :: "modgrp ⇒ 'a :: field"* **is** *"λ(a,b,c,d). -of_int d / of_int c"*
  by *transfer (auto simp: modgrp_rel_def)*

lemma *pole_modgrp_in_Reals: "pole_modgrp f ∈ (ℝ :: 'a :: real_field set)"*
  by *transfer (auto intro!: Reals_divide)*

lemma *Im_pole_modgrp [simp]: "Im (pole_modgrp f) = 0"*
  by *transfer auto*

The complex number to which complex infinity is mapped by the transformation. This is undefined if the transformation maps complex infinity to itself.

lift_definition *pole_image_modgrp :: "modgrp ⇒ 'a :: field"* **is** *"λ(a,b,c,d). of_int a / of_int c"*
  by *transfer (auto simp: modgrp_rel_def)*

lemma *Im_pole_image_modgrp [simp]: "Im (pole_image_modgrp f) = 0"*
  by *transfer auto*

The normalised coefficients of the transformation. The convention that is chosen is that *c* is always non-negative, and if *c* is zero then *d* is positive.

lift_definition *modgrp_a :: "modgrp ⇒ int"* **is** *"λ(a,b,c,d). if c < 0 ∨ c = 0 ∧ d < 0 then -a else a"*
  by *transfer (auto simp: modgrp_rel_def)*

lift_definition *modgrp_b :: "modgrp ⇒ int"* **is** *"λ(a,b,c,d). if c < 0 ∨ c = 0 ∧ d < 0 then -b else b"*
  by *transfer (auto simp: modgrp_rel_def)*

lift_definition *modgrp_c :: "modgrp ⇒ int"* **is** *"λ(a,b,c,d). |c|"*
  by *transfer (auto simp: modgrp_rel_def)*

lift_definition *modgrp_d :: "modgrp ⇒ int"* **is** *"λ(a,b,c,d). if c < 0 ∨ c = 0 ∧ d < 0 then -d else d"*
  by *transfer (auto simp: modgrp_rel_def)*

lemma *modgrp_abcd_S [simp]:*
  "modgrp_a S_modgrp = 0" "modgrp_b S_modgrp = -1" "modgrp_c S_modgrp = 1" "modgrp_d S_modgrp = 0"
  by *(transfer; simp)+*

lemma *modgrp_abcd_T [simp]:*
  "modgrp_a T_modgrp = 1" "modgrp_b T_modgrp = 1" "modgrp_c T_modgrp = 0" "modgrp_d T_modgrp = 1"
  by *(transfer; simp)+*

```
lemma modgrp_abcd_shift [simp]:
  "modgrp_a (shift_modgrp n) = 1" "modgrp_b (shift_modgrp n) = n"
  "modgrp_c (shift_modgrp n) = 0" "modgrp_d (shift_modgrp n) = 1"
  by (transfer; simp)+

lemma modgrp_c_shift_left [simp]:
  "modgrp_c (shift_modgrp n * f) = modgrp_c f"
  by transfer auto

lemma modgrp_d_shift_left [simp]:
  "modgrp_d (shift_modgrp n * f) = modgrp_d f"
  by transfer auto

lemma modgrp_abcd_det: "modgrp_a x * modgrp_d x - modgrp_b x * modgrp_c
x = 1"
  by transfer (auto simp: modgrp_rel_def)

lemma modgrp_c_nonneg: "modgrp_c x ≥ 0"
  by transfer auto

lemma modgrp_a_nz_or_b_nz: "modgrp_a x ≠ 0 ∨ modgrp_b x ≠ 0"
  by transfer (auto simp: modgrp_rel_def split: if_splits)

lemma modgrp_c_nz_or_d_nz: "modgrp_c x ≠ 0 ∨ modgrp_d x ≠ 0"
  by transfer (auto simp: modgrp_rel_def split: if_splits)

lemma modgrp_cd_signs: "modgrp_c x > 0 ∨ modgrp_c x = 0 ∧ modgrp_d x
> 0"
  by transfer (auto simp: modgrp_rel_def zmult_eq_1_iff)

lemma apply_modgrp_altdef:
  "(apply_modgrp x :: 'a :: field ⇒ _) =
   moebius (of_int (modgrp_a x)) (of_int (modgrp_b x)) (of_int (modgrp_c
x)) (of_int (modgrp_d x))"
proof (transfer, goal_cases)
  case (1 x)
  thus ?case
    by (auto simp: case_prod_unfold moebius_uminus')
qed
```

Converting a quadruple of numbers into an element of the modular group.

```
lift_definition modgrp :: "int ⇒ int ⇒ int ⇒ int ⇒ modgrp" is
  "λa b c d. if a * d - b * c = 1 then (a, b, c, d) else (1, 0, 0, 1)"
  by transfer (auto simp: modgrp_rel_def)

lemma modgrp_wrong: "a * d - b * c ≠ 1 ⟹ modgrp a b c d = 1"
  by transfer (auto simp: modgrp_rel_def algebra_simps)
```

```
lemma modgrp_cong:
  assumes "modgrp_rel (a,b,c,d) (a',b',c',d')"
  shows   "modgrp a b c d = modgrp a' b' c' d'"
  using assms by transfer (auto simp add: modgrp_rel_def)

lemma modgrp_abcd [simp]: "modgrp (modgrp_a x) (modgrp_b x) (modgrp_c
x) (modgrp_d x) = x"
  apply transfer
  apply (auto split: if_splits)
       apply (auto simp: modgrp_rel_def)
  done

lemma
  assumes "a * d - b * c = 1"
  shows   modgrp_c_modgrp: "modgrp_c (modgrp a b c d) = |c|"
    and   modgrp_a_modgrp: "modgrp_a (modgrp a b c d) = (if c < 0 ∨ c
= 0 ∧ d < 0 then -a else a)"
    and   modgrp_b_modgrp: "modgrp_b (modgrp a b c d) = (if c < 0 ∨ c
= 0 ∧ d < 0 then -b else b)"
    and   modgrp_d_modgrp: "modgrp_d (modgrp a b c d) = (if c < 0 ∨ c
= 0 ∧ d < 0 then -d else d)"
  using assms by (transfer; simp add: modgrp_rel_def)+
```

2.3.3 Basic properties

```
lemma continuous_on_apply_modgrp [continuous_intros]:
  fixes g :: "'a :: topological_space ⇒ 'b :: real_normed_field"
  assumes "continuous_on A g" "⋀z. z ∈ A ⟹ ¬is_singular_modgrp f ∨
g z ≠ pole_modgrp f"
  shows   "continuous_on A (λz. apply_modgrp f (g z))"
  using assms
  by transfer (auto intro!: continuous_intros simp: modgrp_rel_def)

lemma holomorphic_on_apply_modgrp [holomorphic_intros]:
  assumes "g holomorphic_on A" "⋀z. z ∈ A ⟹ ¬is_singular_modgrp f
∨ g z ≠ pole_modgrp f"
  shows   "(λz. apply_modgrp f (g z)) holomorphic_on A"
  using assms
  by transfer (auto intro!: holomorphic_intros simp: modgrp_rel_def)

lemma analytic_on_apply_modgrp [analytic_intros]:
  assumes "g analytic_on A" "⋀z. z ∈ A ⟹ ¬is_singular_modgrp f ∨
g z ≠ pole_modgrp f"
  shows   "(λz. apply_modgrp f (g z)) analytic_on A"
  using assms
  by transfer (auto intro!: analytic_intros simp: modgrp_rel_def)

lemma isCont_apply_modgrp [continuous_intros]:
  fixes z :: "'a :: real_normed_field"
```

```
  assumes "¬is_singular_modgrp f ∨ z ≠ pole_modgrp f"
  shows   "isCont (apply_modgrp f) z"
proof -
  define S where "S = (if is_singular_modgrp f then -{pole_modgrp f}
else (UNIV :: 'a set))"
  have "continuous_on S (apply_modgrp f)"
    unfolding S_def by (intro continuous_intros) auto
  moreover have "open S" "z ∈ S"
    using assms by (auto simp: S_def)
  ultimately show ?thesis
    using continuous_on_eq_continuous_at by blast
qed

lemmas tendsto_apply_modgrp [tendsto_intros] = isCont_tendsto_compose[OF
isCont_apply_modgrp]

lift_definition diff_scale_factor_modgrp :: "modgrp ⇒ 'a :: field ⇒ 'a
⇒ 'a" is
  "λ(a,b,c,d) w z. (of_int c * w + of_int d) * (of_int c * z + of_int
d)"
  by (auto simp: modgrp_rel_def algebra_simps)

lemma diff_scale_factor_modgrp_commutes:
  "diff_scale_factor_modgrp f w z = diff_scale_factor_modgrp f z w"
  by transfer (simp add: case_prod_unfold)

lemma diff_scale_factor_modgrp_zero_iff:
  fixes w z :: "'a :: field_char_0"
  shows "diff_scale_factor_modgrp f w z = 0 ⟷ is_singular_modgrp f
∧ pole_modgrp f ∈ {w, z}"
  by transfer
     (auto simp: case_prod_unfold modgrp_rel_def divide_simps add_eq_0_iff
mult.commute
                   minus_equation_iff[of "of_int x" for x])

lemma apply_modgrp_diff_eq:
  fixes g :: modgrp
  defines "f ≡ apply_modgrp g"
  assumes *: "¬is_singular_modgrp g ∨ pole_modgrp g ∉ {w, z}"
  shows   "f w - f z = (w - z) / diff_scale_factor_modgrp g w z"
  unfolding f_def using *
  by transfer
     (auto simp: modgrp_rel_def moebius_diff_eq zmult_eq_1_iff
           simp flip: of_int_diff of_int_mult split: if_splits)

lemma norm_modgrp_dividend_ge:
  fixes z :: complex
  shows   "norm (of_int c * z + of_int d) ≥ |c * Im z|"
proof -
```

```
  define x y where "x = Re z" and "y = Im z"
  have z_eq: "z = Complex x y"
    by (simp add: x_def y_def)
  have "(real_of_int c * y) ^ 2 ≤ (real_of_int c * x + real_of_int d)
^ 2 + (real_of_int c * y) ^ 2"
    by simp
  also have "... = norm (of_int c * z + of_int d) ^ 2"
    by (simp add: cmod_power2 x_def y_def)
  finally show "norm (of_int c * z + of_int d) ≥ |c * y|"
    by (metis abs_le_square_iff abs_norm_cancel)
qed

lemma diff_scale_factor_modgrp_altdef:
  fixes g :: modgrp
  defines "c ≡ modgrp_c g" and "d ≡ modgrp_d g"
  shows "diff_scale_factor_modgrp g w z = (of_int c * w + of_int d) *
(of_int c * z + of_int d)"
  unfolding c_def d_def by transfer (auto simp: algebra_simps)

lemma norm_diff_scale_factor_modgrp_ge_complex:
  fixes w z :: complex
  assumes "w ≠ z"
  shows   "norm (diff_scale_factor_modgrp g w z) ≥ of_int (modgrp_c g)
^ 2 * |Im w * Im z|"
proof -
  have "norm (diff_scale_factor_modgrp g w z) ≥
        |of_int (modgrp_c g) * Im w| * |of_int (modgrp_c g) * Im z|"
    unfolding diff_scale_factor_modgrp_altdef norm_mult
    by (intro mult_mono norm_modgrp_dividend_ge) auto
  thus ?thesis
    by (simp add: algebra_simps abs_mult power2_eq_square)
qed

lemma apply_shift_modgrp [simp]: "apply_modgrp (shift_modgrp n) z = z
+ of_int n"
  by transfer (auto simp: moebius_def)

lemma apply_modgrp_T [simp]: "apply_modgrp T_modgrp z = z + 1"
  by transfer (auto simp: moebius_def)

lemma apply_modgrp_S [simp]: "apply_modgrp S_modgrp z = -1 / z"
  by transfer (auto simp: moebius_def)

lemma apply_modgrp_1 [simp]: "apply_modgrp 1 z = z"
  by transfer (auto simp: moebius_def)

lemma apply_modgrp_mult_aux:
  fixes z :: "'a :: field_char_0"
  assumes ns: "c' = 0 ∨ z ≠ -d' / c'"
```

```
  assumes det: "a * d - b * c = 1" "a' * d' - b' * c' = 1"
  shows   "moebius a b c d (moebius a' b' c' d' z) =
           moebius (a * a' + b * c') (a * b' + b * d')
                   (c * a' + d * c') (c * b' + d * d') z"
proof -
  have det': "c ≠ 0 ∨ d ≠ 0" "c' ≠ 0 ∨ d' ≠ 0"
    using det by auto
  from assms have nz: "c' * z + d' ≠ 0"
    by (auto simp add: divide_simps add_eq_0_iff split: if_splits)
  show ?thesis using det nz
    by (simp add: moebius_def divide_simps) (auto simp: algebra_simps)
qed

lemma apply_modgrp_mult:
  fixes z :: "'a :: field_char_0"
  assumes "¬is_singular_modgrp y ∨ z ≠ pole_modgrp y"
  shows   "apply_modgrp (x * y) z = apply_modgrp x (apply_modgrp y z)"
  using assms
proof (transfer, goal_cases)
  case (1 y z x)
  obtain a b c d where [simp]: "x = (a, b, c, d)"
    using prod_cases4 by blast
  obtain a' b' c' d' where [simp]: "y = (a', b', c', d')"
    using prod_cases4 by blast
  show ?case
    using apply_modgrp_mult_aux[of "of_int c'" z "of_int d'" "of_int a"
"of_int d"
                                   "of_int b" "of_int c" "of_int a'" "of_int
b'"] 1
    by (simp flip: of_int_mult of_int_add of_int_diff of_int_minus add:
modgrp_rel_def)
qed

lemma is_singular_modgrp_altdef: "is_singular_modgrp x ⟷ modgrp_c
x ≠ 0"
  by transfer (auto split: if_splits)

lemma not_is_singular_modgrpD:
  assumes "¬is_singular_modgrp x"
  shows   "x = shift_modgrp (sgn (modgrp_a x) * modgrp_b x)"
  using assms
proof (transfer, goal_cases)
  case (1 x)
  obtain a b c d where [simp]: "x = (a, b, c, d)"
    using prod_cases4 by blast
  from 1 have [simp]: "c = 0"
    by auto
  from 1 have "a = 1 ∧ d = 1 ∨ a = -1 ∧ d = -1"
    by (auto simp: modgrp_rel_def zmult_eq_1_iff)
```

```
  thus ?case
    by (auto simp: modgrp_rel_def)
qed

lemma is_singular_modgrp_inverse [simp]: "is_singular_modgrp (inverse
x) ⟷ is_singular_modgrp x"
  by transfer auto

lemma is_singular_modgrp_S_left_iff [simp]: "is_singular_modgrp (S_modgrp
* f) ⟷ modgrp_a f ≠ 0"
  by transfer (auto simp: modgrp_rel_def split: if_splits)

lemma is_singular_modgrp_S_right_iff [simp]: "is_singular_modgrp (f *
S_modgrp) ⟷ modgrp_d f ≠ 0"
  by transfer (auto simp: modgrp_rel_def split: if_splits)

lemma is_singular_modgrp_T_left_iff [simp]:
  "is_singular_modgrp (T_modgrp * f) ⟷ is_singular_modgrp f"
  by transfer (auto simp: modgrp_rel_def split: if_splits)

lemma is_singular_modgrp_T_right_iff [simp]:
  "is_singular_modgrp (f * T_modgrp) ⟷ is_singular_modgrp f"
  by transfer (auto simp: modgrp_rel_def split: if_splits)

lemma is_singular_modgrp_shift_left_iff [simp]:
  "is_singular_modgrp (shift_modgrp n * f) ⟷ is_singular_modgrp f"
  by transfer (auto simp: modgrp_rel_def split: if_splits)

lemma is_singular_modgrp_shift_right_iff [simp]:
  "is_singular_modgrp (f * shift_modgrp n) ⟷ is_singular_modgrp f"
  by transfer (auto simp: modgrp_rel_def split: if_splits)

lemma pole_modgrp_inverse [simp]: "pole_modgrp (inverse x) = pole_image_modgrp
x"
  by transfer auto

lemma pole_image_modgrp_inverse [simp]: "pole_image_modgrp (inverse x)
= pole_modgrp x"
  by transfer auto

lemma pole_image_modgrp_in_Reals: "pole_image_modgrp x ∈ (ℝ :: 'a ::
{real_field, field} set)"
  by transfer (auto intro!: Reals_divide)

lemma apply_modgrp_inverse_eqI:
  fixes x y :: "'a :: field_char_0"
  assumes "¬is_singular_modgrp f ∨ y ≠ pole_modgrp f" "apply_modgrp
f y = x"
  shows   "apply_modgrp (inverse f) x = y"
```

```
proof -
  have "apply_modgrp (inverse f) x = apply_modgrp (inverse f * f) y"
    using assms by (subst apply_modgrp_mult) auto
  also have "... = y"
    by simp
  finally show ?thesis .
qed

lemma apply_modgrp_eq_iff [simp]:
  fixes x y :: "'a :: field_char_0"
  assumes "¬is_singular_modgrp f ∨ x ≠ pole_modgrp f ∧ y ≠ pole_modgrp
f"
  shows   "apply_modgrp f x = apply_modgrp f y ⟷ x = y"
  using assms by (metis apply_modgrp_inverse_eqI)

lemma is_singular_modgrp_times_aux:
  assumes det: "a * d - b * c = 1" "a' * d' - b' * (c' :: int) = 1"
  shows   "(c * a' + d * c' ≠ 0) ⟷ ((c = 0 ⟶ c' ≠ 0) ∧ (c = 0 ∨
c' = 0 ∨ -d * c' ≠ a' * c))"
  using assms by Groebner_Basis.algebra

lemma is_singular_modgrp_times_iff:
   "is_singular_modgrp (x * y) ⟷
      (is_singular_modgrp x ∨ is_singular_modgrp y) ∧
      (¬is_singular_modgrp x ∨ ¬is_singular_modgrp y ∨ pole_modgrp x
≠ (pole_image_modgrp y :: real))"
proof (transfer, goal_cases)
  case (1 x y)
  obtain a b c d where [simp]: "x = (a, b, c, d)"
    using prod_cases4 by blast
  obtain a' b' c' d' where [simp]: "y = (a', b', c', d')"
    using prod_cases4 by blast
  show ?case
    using is_singular_modgrp_times_aux[of a d b c a' d' b' c'] 1
    by (auto simp: modgrp_rel_def field_simps simp flip: of_int_mult of_int_add
of_int_minus of_int_diff)
qed

lemma shift_modgrp_1: "shift_modgrp 1 = T_modgrp"
  by transfer (auto simp: modgrp_rel_def)

lemma shift_modgrp_eq_iff: "shift_modgrp n = shift_modgrp m ⟷ n =
m"
  by transfer (auto simp: modgrp_rel_def)

lemma shift_modgrp_neq_S [simp]: "shift_modgrp n ≠ S_modgrp"
  by transfer (auto simp: modgrp_rel_def)

lemma S_neq_shift_modgrp [simp]: "S_modgrp ≠ shift_modgrp n"
```

```
  by transfer (auto simp: modgrp_rel_def)

lemma shift_modgrp_eq_T_iff [simp]: "shift_modgrp n = T_modgrp ⟷ n
= 1"
  by transfer (auto simp: modgrp_rel_def)

lemma T_eq_shift_modgrp_iff [simp]: "T_modgrp = shift_modgrp n ⟷ n
= 1"
  by transfer (auto simp: modgrp_rel_def)

lemma shift_modgrp_0 [simp]: "shift_modgrp 0 = 1"
  by transfer (auto simp: modgrp_rel_def)

lemma shift_modgrp_add: "shift_modgrp (m + n) = shift_modgrp m * shift_modgrp
n"
  by transfer (auto simp: modgrp_rel_def)

lemma shift_modgrp_minus: "shift_modgrp (-m) = inverse (shift_modgrp
m)"
proof -
  have "1 = shift_modgrp (m + (-m))"
    by simp
  also have "shift_modgrp (m + (-m)) = shift_modgrp m * shift_modgrp (-m)"
    by (subst shift_modgrp_add) auto
  finally show ?thesis
    by (simp add: modgrp.inverse_unique)
qed

lemma shift_modgrp_power: "shift_modgrp n ^ m = shift_modgrp (n * m)"
  by (induction m) (auto simp: algebra_simps shift_modgrp_add)

lemma shift_modgrp_power_int: "shift_modgrp n powi m = shift_modgrp (n
* m)"
  by (cases "m ≥ 0")
     (auto simp: power_int_def shift_modgrp_power simp flip: shift_modgrp_minus)

lemma shift_shift_modgrp: "shift_modgrp n * (shift_modgrp m * x) = shift_modgrp
(n + m) * x"
  by (simp add: mult.assoc shift_modgrp_add)


lemma shift_modgrp_conv_T_power: "shift_modgrp n = T_modgrp powi n"
  by (simp flip: shift_modgrp_1 add: shift_modgrp_power_int)

lemma modgrp_S_S [simp]: "S_modgrp * S_modgrp = 1"
  by transfer (auto simp: modgrp_rel_def)

lemma inverse_S_modgrp [simp]: "inverse S_modgrp = S_modgrp"
  using modgrp_S_S modgrp.inverse_unique by metis
```

lemma *modgrp_S_S_' [simp]: "S_modgrp * (S_modgrp * x) = x"*
by *(subst mult.assoc [symmetric], subst modgrp_S_S) simp*

lemma *modgrp_S_power: "S_modgrp ^ n = (if even n then 1 else S_modgrp)"*
by *(induction n) auto*

lemma *modgrp_S_S_power_int: "S_modgrp powi n = (if even n then 1 else S_modgrp)"*
by *(auto simp: power_int_def modgrp_S_power even_nat_iff)*

lemma *not_is_singular_1_modgrp [simp]: "¬is_singular_modgrp 1"*
by *transfer auto*

lemma *not_is_singular_T_modgrp [simp]: "¬is_singular_modgrp T_modgrp"*
by *transfer auto*

lemma *not_is_singular_shift_modgrp [simp]: "¬is_singular_modgrp (shift_modgrp n)"*
by *transfer auto*

lemma *is_singular_S_modgrp [simp]: "is_singular_modgrp S_modgrp"*
by *transfer auto*

lemma *pole_modgrp_S [simp]: "pole_modgrp S_modgrp = 0"*
by *transfer auto*

lemma *pole_modgrp_1 [simp]: "pole_modgrp 1 = 0"*
by *transfer auto*

lemma *pole_modgrp_T [simp]: "pole_modgrp T_modgrp = 0"*
by *transfer auto*

lemma *pole_modgrp_shift [simp]: "pole_modgrp (shift_modgrp n) = 0"*
by *transfer auto*

lemma *pole_image_modgrp_1 [simp]: "pole_image_modgrp 1 = 0"*
by *transfer auto*

lemma *pole_image_modgrp_T [simp]: "pole_image_modgrp T_modgrp = 0"*
by *transfer auto*

lemma *pole_image_modgrp_shift [simp]: "pole_image_modgrp (shift_modgrp n) = 0"*
by *transfer auto*

lemma *pole_image_modgrp_S [simp]: "pole_image_modgrp S_modgrp = 0"*
by *transfer auto*

```
lemma minus_minus_power2_eq: "(-x - y :: 'a :: ring_1) ^ 2 = (x + y)
^ 2"
  by (simp add: algebra_simps power2_eq_square)

lift_definition deriv_modgrp :: "modgrp ⇒ 'a :: field ⇒ 'a" is
  "λ(a,b,c,d) x. inverse ((of_int c * x + of_int d) ^ 2)"
  by (auto simp: modgrp_rel_def minus_minus_power2_eq)

lemma deriv_modgrp_nonzero:
  assumes "¬is_singular_modgrp f ∨ (x :: 'a :: field_char_0) ≠ pole_modgrp
f"
  shows   "deriv_modgrp f x ≠ 0"
  using assms
  by transfer (auto simp: modgrp_rel_def add_eq_0_iff split: if_splits)

lemma deriv_modgrp_altdef:
  "deriv_modgrp f z = inverse (of_int (modgrp_c f) * z + of_int (modgrp_d
f)) ^ 2"
  by transfer (auto simp: minus_minus_power2_eq power_inverse)

lemma apply_modgrp_has_field_derivative [derivative_intros]:
  assumes "¬is_singular_modgrp f ∨ x ≠ pole_modgrp f"
  shows   "(apply_modgrp f has_field_derivative deriv_modgrp f x) (at
x within A)"
  using assms
proof (transfer, goal_cases)
  case (1 f x A)
  obtain a b c d where [simp]: "f = (a, b, c, d)"
    using prod_cases4 .
  have "(moebius (of_int a) (of_int b) (of_int c) (of_int d) has_field_derivative
          (of_int (a * d - b * c) / ((of_int c * x + of_int d)^2)))
          (at x within A)"
    unfolding of_int_mult of_int_diff
    by (rule moebius_has_field_derivative) (use 1 in ‹auto simp: modgrp_rel_def›)
  also have "a * d - b * c = 1"
    using 1 by (simp add: modgrp_rel_def)
  finally show ?case
    by (simp add: field_simps)
qed

lemma apply_modgrp_has_field_derivative' [derivative_intros]:
  assumes "(g has_field_derivative g') (at x within A)"
  assumes "¬is_singular_modgrp f ∨ g x ≠ pole_modgrp f"
  shows   "((λx. apply_modgrp f (g x)) has_field_derivative deriv_modgrp
f (g x) * g')
             (at x within A)"
proof -
  have "((apply_modgrp f ∘ g) has_field_derivative deriv_modgrp f (g x)
```

```
* g') (at x within A)"
    by (intro DERIV_chain assms derivative_intros)
  thus ?thesis
    by (simp add: o_def)
qed


lemma modgrp_a_1 [simp]: "modgrp_a 1 = 1"
  and modgrp_b_1 [simp]: "modgrp_b 1 = 0"
  and modgrp_c_1 [simp]: "modgrp_c 1 = 0"
  and modgrp_d_1 [simp]: "modgrp_d 1 = 1"
  by (transfer; simp; fail)+

lemma modgrp_c_0:
  assumes "a * d = 1"
  shows   "modgrp a b 0 d = shift_modgrp (if a > 0 then b else -b)"
  using assms by transfer (auto simp: modgrp_rel_def zmult_eq_1_iff)

lemma not_singular_modgrpD:
  assumes "¬is_singular_modgrp f"
  shows   "f = shift_modgrp (modgrp_b f)"
  using assms by transfer (auto simp: modgrp_rel_def zmult_eq_1_iff)

lemma S_conv_modgrp: "S_modgrp = modgrp 0 (-1) 1 0"
  and T_conv_modgrp: "T_modgrp = modgrp 1 1 0 1"
  and shift_conv_modgrp: "shift_modgrp n = modgrp 1 n 0 1"
  and one_conv_modgrp: "1 = modgrp 1 0 0 1"
  by (transfer; simp add: modgrp_rel_def)+

lemma modgrp_rel_reflI: "(case x of (a,b,c,d) ⇒ a * d - b * c = 1) ⟹
x = y ⟹ modgrp_rel x y"
  by (simp add: modgrp_rel_def case_prod_unfold)

lemma modgrp_times:
  assumes "a * d - b * c = 1"
  assumes "a' * d' - b' * c' = 1"
  shows "modgrp a b c d * modgrp a' b' c' d' =
         modgrp (a * a' + b * c') (a * b' + b * d') (c * a' + d * c')
(c * b' + d * d')"
  using assms by transfer (auto simp: modgrp_rel_def algebra_simps)

lemma modgrp_inverse:
  assumes "a * d - b * c = 1"
  shows   "inverse (modgrp a b c d) = modgrp d (-b) (-c) a"
  by (intro modgrp.inverse_unique, subst modgrp_times)
     (use assms in ‹auto simp: algebra_simps one_conv_modgrp›)

lemma modgrp_a_mult_shift [simp]: "modgrp_a (f * shift_modgrp m) = modgrp_a
f"
```

```
  by transfer auto

lemma modgrp_b_mult_shift [simp]: "modgrp_b (f * shift_modgrp m) = modgrp_a
f * m + modgrp_b f"
  by transfer auto

lemma modgrp_c_mult_shift [simp]: "modgrp_c (f * shift_modgrp m) = modgrp_c
f"
  by transfer auto

lemma modgrp_d_mult_shift [simp]: "modgrp_d (f * shift_modgrp m) = modgrp_c
f * m + modgrp_d f"
  by transfer auto

lemma coprime_modgrp_c_d: "coprime (modgrp_c f) (modgrp_d f)"
proof -
  define a b c d where "a = modgrp_a f" "b = modgrp_b f" "c = modgrp_c
f" "d = modgrp_d f"
  have "[a * 0 - b * c = a * d - b * c] (mod d)"
    by (intro cong_diff cong_refl cong_mult) (auto simp: Cong.cong_def)
  also have "a * d - b * c = 1"
    unfolding a_b_c_d_def modgrp_abcd_det[of f] by simp
  finally have "[c * (-b) = 1] (mod d)"
    by (simp add: mult_ac)
  thus "coprime c d"
    by (subst coprime_iff_invertible_int) (auto intro!: exI[of _ "-b"])
qed

context unimodular_moebius_transform
begin

lift_definition as_modgrp :: modgrp is "(a, b, c, d)"
  using unimodular by (auto simp: modgrp_rel_def)

lemma as_modgrp_altdef: "as_modgrp = modgrp a b c d"
  using unimodular by transfer (auto simp: modgrp_rel_def)

lemma φ_as_modgrp: "φ = apply_modgrp as_modgrp"
  unfolding φ_def by transfer simp

end

interpretation modgrp: unimodular_moebius_transform "modgrp_a x" "modgrp_b
x" "modgrp_c x" "modgrp_d x"
  rewrites "modgrp.as_modgrp = x" and "modgrp.φ = apply_modgrp x"
proof -
  show *: "unimodular_moebius_transform (modgrp_a x) (modgrp_b x) (modgrp_c
x) (modgrp_d x)"
    by unfold_locales (rule modgrp_abcd_det)
```

show *"unimodular_moebius_transform.as_modgrp (modgrp_a x) (modgrp_b x) (modgrp_c x) (modgrp_d x) = x"*
by *(subst unimodular_moebius_transform.as_modgrp_altdef[OF *]) auto*
show *"unimodular_moebius_transform.φ (modgrp_a x) (modgrp_b x) (modgrp_c x) (modgrp_d x) = apply_modgrp x"*
by *(subst unimodular_moebius_transform.φ_def[OF *], subst apply_modgrp_altdef) auto*
qed

2.4 Code generation

code_datatype *modgrp*

lemma *one_modgrp_code [code]:* *"1 = modgrp 1 0 0 1"*
and *S_modgrp_code [code]:* *"S_modgrp = modgrp 0 (-1) 1 0"*
and *T_modgrp_code [code]:* *"T_modgrp = modgrp 1 1 0 1"*
and *shift_modgrp_code [code]:* *"shift_modgrp n = modgrp 1 n 0 1"*
by *(simp_all add: one_conv_modgrp S_conv_modgrp T_conv_modgrp shift_conv_modgrp)*

lemma *inverse_modgrp_code [code]: "inverse (modgrp a b c d) = modgrp d (-b) (-c) a"*
by *(cases "a * d - b * c = 1")*
(auto simp: modgrp_inverse modgrp_wrong algebra_simps)

lemma *times_modgrp_code [code]:*
*"modgrp a b c d * modgrp a' b' c' d' = (*
*if a * d - b * c ≠ 1 then modgrp a' b' c' d'*
*else if a' * d' - b' * c' ≠ 1 then modgrp a b c d*
*else modgrp (a * a' + b * c') (a * b' + b * d') (c * a' + d * c') (c * b' + d * d'))"*
by *(simp add: modgrp_times modgrp_wrong)*

lemma *modgrp_a_code [code]:*
*"modgrp_a (modgrp a b c d) = (if a * d - b * c = 1 then if c < 0 ∨ c = 0 ∧ d < 0 then -a else a else 1)"*
by *transfer auto*

lemma *modgrp_b_code [code]:*
*"modgrp_b (modgrp a b c d) = (if a * d - b * c = 1 then if c < 0 ∨ c = 0 ∧ d < 0 then -b else b else 0)"*
by *transfer (auto simp: modgrp_rel_def)*

lemma *modgrp_c_code [code]:*
*"modgrp_c (modgrp a b c d) = (if a * d - b * c = 1 then |c| else 0)"*
by *transfer (auto simp: modgrp_rel_def)*

lemma *modgrp_d_code [code]:*
*"modgrp_d (modgrp a b c d) = (if a * d - b * c = 1 then if c < 0 ∨ c = 0 ∧ d < 0 then -d else d else 1)"*

```
  by transfer auto

lemma apply_modgrp_code [code]:
  "apply_modgrp (modgrp a b c d) z =
     (if a * d - b * c ≠ 1 then z else (of_int a * z + of_int b) / (of_int
c * z + of_int d))"
  by transfer (auto simp: moebius_def)

lemma is_singular_modgrp_code [code]:
  "is_singular_modgrp (modgrp a b c d) ⟷ a * d - b * c = 1 ∧ c ≠ 0"
  by transfer auto

lemma pole_modgrp_code [code]:
  "pole_modgrp (modgrp a b c d) = (if a * d - b * c = 1 then -of_int d
/ of_int c else 0)"
  by transfer auto

lemma pole_image_modgrp_code [code]:
  "pole_image_modgrp (modgrp a b c d) =
    (if a * d - b * c = 1 ∧ c ≠ 0 then of_int a / of_int c else 0)"
  by transfer auto
```

The following will be needed later to define the slash operator.

```
definition modgrp_factor :: "modgrp ⇒ complex ⇒ complex" where
  "modgrp_factor g z = of_int (modgrp_c g) * z + of_int (modgrp_d g)"

lemma modgrp_factor_1 [simp]: "modgrp_factor 1 z = 1"
  by (auto simp: modgrp_factor_def)

lemma modgrp_factor_shift [simp]: "modgrp_factor (shift_modgrp n) z =
1"
  by (simp add: modgrp_factor_def)

lemma modgrp_factor_T [simp]: "modgrp_factor T_modgrp z = 1"
  by (simp add: modgrp_factor_def)

lemma modgrp_factor_S [simp]: "modgrp_factor S_modgrp z = z"
  by (simp add: modgrp_factor_def)

lemma modgrp_factor_shift_right [simp]:
  "modgrp_factor (f * shift_modgrp n) z = modgrp_factor f (z + of_int
n)"
  unfolding modgrp_factor_def
  by transfer (auto simp: algebra_simps)

lemma modgrp_factor_shift_left [simp]:
  "modgrp_factor (shift_modgrp n * f) z = modgrp_factor f z"
  by (simp add: modgrp_factor_def)
```

```
lemma modgrp_factor_T_right [simp]:
  "modgrp_factor (f * T_modgrp) z = modgrp_factor f (z + 1)"
  unfolding shift_modgrp_1 [symmetric] by (subst modgrp_factor_shift_right)
auto

lemma modgrp_factor_T_left [simp]:
  "modgrp_factor (T_modgrp * f) z = modgrp_factor f z"
  unfolding shift_modgrp_1 [symmetric] by (subst modgrp_factor_shift_left)
auto

lemma has_field_derivative_modgrp_factor [derivative_intros]:
  assumes "(f has_field_derivative f') (at x)"
  shows   "((λx. modgrp_factor g (f x)) has_field_derivative (of_int
(modgrp_c g) * f')) (at x)"
  unfolding modgrp_factor_def by (auto intro!: derivative_eq_intros assms)

lemma modgrp_factor_analytic [analytic_intros]: "modgrp_factor g analytic_on
A"
  unfolding modgrp_factor_def by (auto simp: modgrp_factor_def intro!:
analytic_intros)

lemma modgrp_factor_meromorphic [meromorphic_intros]: "modgrp_factor
h meromorphic_on A"
  unfolding modgrp_factor_def by (auto intro!: meromorphic_intros)

lemma modgrp_factor_nonzero [simp]:
  assumes "Im z ≠ 0"
  shows   "modgrp_factor g z ≠ 0"
proof -
  define c d where "c = modgrp_c g" and "d = modgrp_d g"
  have "c ≠ 0 ∨ d ≠ 0"
    unfolding c_def d_def using modgrp_c_nz_or_d_nz by blast
  have "of_int c * z + of_int d ≠ 0"
    using assms ‹c ≠ 0 ∨ d ≠ 0› by (auto simp: complex_eq_iff)
  thus ?thesis
    by (simp add: modgrp_factor_def c_def d_def)
qed

lemma tendsto_modgrp_factor [tendsto_intros]:
  "(f ⟶ c) F ⟹ ((λx. modgrp_factor g (f x)) ⟶ modgrp_factor
g c) F"
  unfolding modgrp_factor_def by (auto intro!: tendsto_intros)

lemma minus_diff_power_even:
  assumes "even k"
  shows   "(-a - b) ^ k = (a + b :: 'a :: ring_1) ^ k"
proof -
  have "(-a - b) ^ k = (-(a + b)) ^ k"
    by simp
```

```
  also have "... = (a + b) ^ k"
    using assms by (rule power_minus_even)
  finally show ?thesis .
qed

lemma minus_diff_power_int_even:
  assumes "even k"
  shows   "(-a - b) powi k = (a + b :: 'a :: field) powi k"
proof -
  have "(-a - b) powi k = (-(a + b)) powi k"
    by simp
  also have "... = (a + b) powi k"
    by (rule power_int_minus_left_even) fact
  finally show ?thesis .
qed
```

2.5 The slash operator

The typical definition in the literature is that, for a function $f : \mathbb{H} \to \mathbb{C}$ and an element γ of the modular group, the slash operator of weight k is defined as $(f|_k\gamma)(z) = (cz + d)^{-k} f(\gamma z)$.

This has notational advantages, but for formalisation, we think the following is a bit easier for now. Note that in practice, k will always be even, and we do in fact need it to be even here because otherwise the concept would not be well-defined since (c, d) is only determined up to a factor ± 1.

```
lift_definition modgrp_slash :: "modgrp ⇒ int ⇒ complex ⇒ complex" is
  "(λ(a,b,c,d) k z. if even k then (of_int c * z + of_int d) powi k else
0)"
  by standard (auto simp: fun_eq_iff modgrp_rel_def minus_diff_power_int_even)

lemma modgrp_slash_altdef:
  "modgrp_slash f k z = (if even k then modgrp_factor f z powi k else
0)"
  unfolding modgrp_factor_def by transfer (force simp: modgrp_rel_def
minus_diff_power_int_even)

lemma modgrp_slash_1 [simp]: "even k ⟹ modgrp_slash 1 k z = 1"
  by transfer auto

lemma modgrp_slash_shift [simp]: "even k ⟹ modgrp_slash (shift_modgrp
n) k z = 1"
  by transfer auto

lemma modgrp_slash_T [simp]: "even k ⟹ modgrp_slash T_modgrp k z =
1"
  by transfer auto

lemma modgrp_slash_S [simp]: "even k ⟹ modgrp_slash S_modgrp k z =
```

```
z powi k"
  by transfer auto

lemma modgrp_slash_mult:
  assumes "z ∉ ℝ"
  shows   "modgrp_slash (f * g) k z = modgrp_slash f k (apply_modgrp g
z) * modgrp_slash g k z"
proof (use assms in transfer, safe, goal_cases)
  case (1 z a b c d a' b' c' d' k)
  hence "complex_of_int d' + z * complex_of_int c' ≠ 0"
    using 1 by (auto simp: modgrp_rel_def complex_eq_iff complex_is_Real_iff)
  thus ?case
    by (auto simp: moebius_def field_simps power_int_divide_distrib)
qed

lemma modgrp_slash_meromorphic [meromorphic_intros]: "modgrp_slash f
k meromorphic_on A"
  unfolding modgrp_slash_altdef by (cases "even k") (auto intro!: meromorphic_intros)
```

2.6 Representation as product of powers of generators

```
definition modgrp_from_gens :: "int option list ⇒ modgrp" where
  "modgrp_from_gens xs = prod_list (map (λx. case x of None ⇒ S_modgrp
| Some n ⇒ shift_modgrp n) xs)"

lemma modgrp_from_gens_Nil [simp]:
        "modgrp_from_gens [] = 1"
  and modgrp_from_gens_append [simp]:
        "modgrp_from_gens (xs @ ys) = modgrp_from_gens xs * modgrp_from_gens
ys"
  and modgrp_from_gens_Cons1 [simp]:
        "modgrp_from_gens (None # xs) = S_modgrp * modgrp_from_gens xs"
  and modgrp_from_gens_Cons2 [simp]:
        "modgrp_from_gens (Some n # xs) = shift_modgrp n * modgrp_from_gens
xs"
  and modgrp_from_gens_Cons:
        "modgrp_from_gens (x # xs) =
            (case x of None ⇒ S_modgrp | Some n ⇒ shift_modgrp n) *
modgrp_from_gens xs"
  by (simp_all add: modgrp_from_gens_def)

definition invert_modgrp_gens :: "int option list ⇒ int option list"
  where "invert_modgrp_gens = rev ∘ map (map_option uminus)"

lemma invert_modgrp_gens_Nil [simp]:
        "invert_modgrp_gens [] = []"
  and invert_modgrp_gens_append [simp]:
        "invert_modgrp_gens (xs @ ys) = invert_modgrp_gens ys @ invert_modgrp_gens
xs"
```

```
  and invert_modgrp_gens_Cons1 [simp]:
        "invert_modgrp_gens (None # xs) = invert_modgrp_gens xs @ [None]"
  and invert_modgrp_gens_Cons2 [simp]:
        "invert_modgrp_gens (Some n # xs) = invert_modgrp_gens xs @ [Some
(-n)]"
  and invert_modgrp_gens_Cons:
        "invert_modgrp_gens (x # xs) = invert_modgrp_gens xs @ [map_option
uminus x]"
  by (simp_all add: invert_modgrp_gens_def)

lemma modgrp_from_gens_invert [simp]:
  "modgrp_from_gens (invert_modgrp_gens xs) = inverse (modgrp_from_gens
xs)"
  by (induction xs)
     (auto simp: invert_modgrp_gens_Cons map_option_case algebra_simps

                 modgrp.inverse_distrib_swap shift_modgrp_minus split:
option.splits)

function modgrp_genseq :: "int ⇒ int ⇒ int ⇒ int ⇒ int option list"
where
  "modgrp_genseq a b c d =
     (if c = 0 then let b' = (if a > 0 then b else -b) in [Some b']
      else modgrp_genseq (-a * (d div c) + b) (-a) (d mod c) (-c) @ [None,
Some (d div c)])"
  by auto
termination
  by (relation "Wellfounded.measure (nat ∘ abs ∘ (λ(_,_,c,_) ⇒ c))")
(auto simp: abs_mod_less)

lemmas [simp del] = modgrp_genseq.simps

lemma modgrp_genseq_c_0: "modgrp_genseq a b 0 d = (let b' = (if a > 0
then b else -b) in [Some b'])"
  and modgrp_genseq_c_nz:
        "c ≠ 0 ⟹ modgrp_genseq a b c d =
           (let q = d div c in modgrp_genseq (-a * q + b) (-a) (d mod
c) (-c) @ [None, Some q])"
  by (subst modgrp_genseq.simps; simp add: Let_def)+

lemma modgrp_genseq_code [code]:
  "modgrp_genseq a b c d =
     (if c = 0 then [Some (if a > 0 then b else -b)]
      else (let q = d div c in modgrp_genseq (-a * q + b) (-a) (d mod
c) (-c) @ [None, Some q]))"
  by (subst modgrp_genseq.simps) (auto simp: Let_def)

lemma modgrp_genseq_correct:
  assumes "a * d - b * c = 1"
```

```
  shows   "modgrp_from_gens (modgrp_genseq a b c d) = modgrp a b c d"
  using assms
proof (induction a b c d rule: modgrp_genseq.induct)
  case (1 a b c d)
  write S_modgrp ("S")
  write shift_modgrp ("T")

  show ?case
  proof (cases "c = 0")
    case [simp]: True
    thus ?thesis using 1
      by (auto simp: modgrp_genseq_c_0 modgrp_c_0)
  next
    case False
    define q r where "q = d div c" and "r = d mod c"
    have "q * c + r = d"
      by (simp add: q_def r_def)
    with ‹a * d - b * c = 1› have det': "a * r - (b - a * q) * c = 1"
      by Groebner_Basis.algebra

    have "modgrp_from_gens (modgrp_genseq a b c d) = modgrp (-a * q +
b) (-a) r (-c) * (S * T q)"
      using False "1.IH" det' by (simp add: modgrp_genseq_c_nz Let_def
q_def r_def)
    also have "S * T q = modgrp 0 (- 1) 1 q"
      unfolding S_conv_modgrp shift_conv_modgrp by (subst modgrp_times)
simp_all
    also have "modgrp (-a * q + b) (-a) r (-c) * ... = modgrp (- a) (-
b) (- c) (-r - c * q)"
      using det' by (subst modgrp_times) simp_all
    also have "... = modgrp a b c (q * c + r)"
      using det' by (intro modgrp_cong) (auto simp: algebra_simps modgrp_rel_def)
    also have "q * c + r = d"
      by (simp add: q_def r_def)
    finally show ?thesis .
  qed
qed

lemma filterlim_apply_modgrp_at:
  assumes "¬is_singular_modgrp g ∨ z ≠ pole_modgrp g"
  shows   "filterlim (apply_modgrp g) (at (apply_modgrp g z)) (at (z ::
'a :: real_normed_field))"
proof -
  have "∀⇩F x in at z. x ∈ (if is_singular_modgrp g then -{pole_modgrp
g} else UNIV) - {z}"
    by (intro eventually_at_in_open) (use assms in auto)
  hence "∀⇩F x in at z. apply_modgrp g x ≠ apply_modgrp g z"
    by eventually_elim (use assms in ‹auto split: if_splits›)
  thus ?thesis
```

```
    using assms by (auto simp: filterlim_at intro!: tendsto_eq_intros)
qed

lemma apply_modgrp_neq_pole_image [simp]:
  "is_singular_modgrp g ⟹ z ≠ pole_modgrp g ⟹
     apply_modgrp g (z :: 'a :: field_char_0) ≠ pole_image_modgrp g"
  by transfer (auto simp: field_simps add_eq_0_iff moebius_def modgrp_rel_def
                    simp flip: of_int_add of_int_mult of_int_diff)

lemma image_apply_modgrp_conv_vimage:
  fixes A :: "'a :: field_char_0 set"
  assumes "¬is_singular_modgrp f ∨ pole_modgrp f ∉ A"
  defines "S ≡ (if is_singular_modgrp f then -{pole_image_modgrp f ::
'a} else UNIV)"
  shows   "apply_modgrp f ` A = apply_modgrp (inverse f) -` A ∩ S"
proof (intro equalityI subsetI)
  fix z assume z: "z ∈ (apply_modgrp (inverse f) -` A) ∩ S"
  thus "z ∈ apply_modgrp f ` A" using assms
    by (auto elim!: rev_image_eqI simp flip: apply_modgrp_mult simp: S_def
split: if_splits)
next
  fix z assume z: "z ∈ apply_modgrp f ` A"
  then obtain x where x: "x ∈ A" "z = apply_modgrp f x"
    by auto
  have "apply_modgrp (inverse f) (apply_modgrp f x) ∈ A"
    using x assms by (subst apply_modgrp_mult [symmetric]) auto
  moreover have "apply_modgrp f x ≠ pole_image_modgrp f" if "is_singular_modgrp
f"
    using x assms that by (intro apply_modgrp_neq_pole_image) auto
  ultimately show "z ∈ (apply_modgrp (inverse f) -` A) ∩ S"
    using x by (auto simp: S_def)
qed

lemma apply_modgrp_open_map:
  fixes A :: "'a :: real_normed_field set"
  assumes "open A" "¬is_singular_modgrp f ∨ pole_modgrp f ∉ A"
  shows   "open (apply_modgrp f ` A)"
proof -
  define S :: "'a set" where "S = (if is_singular_modgrp f then -{pole_image_modgrp
f} else UNIV)"
  have "open S"
    by (auto simp: S_def)
  have "apply_modgrp f ` A = apply_modgrp (inverse f) -` A ∩ S"
    using image_apply_modgrp_conv_vimage[of f A] assms by (auto simp:
S_def)
  also have "apply_modgrp (inverse f) -` A ∩ S = S ∩ apply_modgrp (inverse
f) -` A"
    by (simp only: Int_commute)
  also have "open ..."
```

```
    using assms by (intro continuous_open_preimage continuous_intros assms
<open S>)
                   (auto simp: S_def)
  finally show ?thesis .
qed

lemma filtermap_at_apply_modgrp:
  fixes z :: "'a :: real_normed_field"
  assumes "¬is_singular_modgrp g ∨ z ≠ pole_modgrp g"
  shows   "filtermap (apply_modgrp g) (at z) = at (apply_modgrp g z)"
proof (rule filtermap_fun_inverse)
  show "filterlim (apply_modgrp g) (at (apply_modgrp g z)) (at z)"
    using assms by (intro filterlim_apply_modgrp_at) auto
next
  have "filterlim (apply_modgrp (inverse g)) (at (apply_modgrp (inverse
g) (apply_modgrp g z)))
           (at (apply_modgrp g z))"
    using assms by (intro filterlim_apply_modgrp_at) auto
  also have "apply_modgrp (inverse g) (apply_modgrp g z) = z"
    using assms by (simp flip: apply_modgrp_mult)
  finally show "filterlim (apply_modgrp (inverse g)) (at z) (at (apply_modgrp
g z))" .
next
  have "eventually (λx. x ∈ (if is_singular_modgrp g then -{pole_image_modgrp
g} else UNIV))
           (at (apply_modgrp g z))"
    by (intro eventually_at_in_open') (use assms in auto)
  thus "∀F x in at (apply_modgrp g z). apply_modgrp g (apply_modgrp (inverse
g) x) = x"
    by eventually_elim (auto simp flip: apply_modgrp_mult split: if_splits)
qed

lemma zorder_moebius_zero:
  assumes "a ≠ 0" "a * d - b * c ≠ 0"
  shows   "zorder (moebius a b c d) (-b / a) = 1"
proof (rule zorder_eqI)
  note [simp del] = div_mult_self3 div_mult_self4 div_mult_self2 div_mult_self1
  define A where "A = (if c = 0 then UNIV else -{-d/c})"
  show "open A"
    by (auto simp: A_def)
  show "-b/a ∈ A"
    using assms by (auto simp: A_def field_simps)
  show "(λx. a / (c * x + d)) holomorphic_on A"
  proof (intro holomorphic_intros)
    fix x assume "x ∈ A"
    thus "c * x + d ≠ 0"
      unfolding A_def using assms
      by (auto simp: A_def field_simps add_eq_0_iff split: if_splits)
  qed
```

```
  show "a / (c * (-b / a) + d) ≠ 0"
    using assms by (auto simp: field_simps)
  show "moebius a b c d w = a / (c * w + d) * (w - - b / a) powi 1"
    if "w ∈ (if c = 0 then UNIV else - {- d / c})" "w ≠ - b / a" for w
    using that assms by (auto simp: divide_simps moebius_def split: if_splits)
qed

lemma zorder_moebius_pole:
  assumes "c ≠ 0" "a * d - b * c ≠ 0"
  shows   "zorder (moebius a b c d) (-d / c) = -1"
proof -
  note [simp del] = div_mult_self3 div_mult_self4 div_mult_self2 div_mult_self1
  have "zorder (moebius a b c d) (-d / c) =
          zorder (λx. (a * x + b) / c / (x - (-d / c)) ^ 1) (-d / c)"
  proof (rule zorder_cong)
    have "eventually (λz. z ≠ -d/c) (at (-d/c))"
      by (simp add: eventually_neq_at_within)
    thus "∀F z in at (- d / c). moebius a b c d z = (a * z + b) / c /
(z - - d / c) ^ 1"
      by eventually_elim (use assms in <auto simp: moebius_def divide_simps
mult_ac>)
  qed auto
  also have "zorder (λx. (a * x + b) / c / (x - (-d / c)) ^ 1) (-d / c)
= -int 1"
  proof (rule zorder_nonzero_div_power)
    show "(λw. (a * w + b) / c) holomorphic_on UNIV"
      using assms by (intro holomorphic_intros)
    show "(a * (-d / c) + b) / c ≠ 0"
      using assms by (auto simp: field_simps)
  qed auto
  finally show ?thesis by simp
qed

lemma zorder_moebius:
  assumes "c = 0 ∨ z ≠ -d / c" "a * d - b * c ≠ 0"
  shows   "zorder (λx. moebius a b c d x - moebius a b c d z) z = 1"
proof (rule zorder_eqI)
  define S where "S = (if c = 0 then UNIV else -{-d/c})"
  define g where "g = (λw. (a * d - b * c) / ((c * w + d) * (c * z +
d)))"
  show "open S" "z ∈ S"
    using assms by (auto simp: S_def)
  show "g holomorphic_on S"
    unfolding g_def using assms
    by (intro holomorphic_intros no_zero_divisors)
       (auto simp: S_def field_simps add_eq_0_iff split: if_splits)
  show "(a * d - b * c) / ((c * z + d) * (c * z + d)) ≠ 0"
    using assms by (auto simp: add_eq_0_iff split: if_splits)
  show "moebius a b c d w - moebius a b c d z =
```

```
        (a * d - b * c) / ((c * w + d) * (c * z + d)) * (w - z) powi
1" if "w ∈ S" for w
    by (subst moebius_diff_eq) (use that assms in ‹auto simp: S_def split:
if_splits›)
qed

lemma zorder_apply_modgrp:
  assumes "¬is_singular_modgrp g ∨ z ≠ pole_modgrp g"
  shows   "zorder (λx. apply_modgrp g x - apply_modgrp g z) z = 1"
  using assms
proof (transfer, goal_cases)
  case (1 f z)
  obtain a b c d where [simp]: "f = (a, b, c, d)"
    using prod_cases4 .
  show ?case using 1 zorder_moebius[of c z d a b]
    by (auto simp: modgrp_rel_def simp flip: of_int_mult)
qed

lemma zorder_fls_modgrp_pole:
  assumes "is_singular_modgrp f"
  shows   "zorder (apply_modgrp f) (pole_modgrp f) = -1"
  using assms
proof (transfer, goal_cases)
  case (1 f)
  obtain a b c d where [simp]: "f = (a, b, c, d)"
    using prod_cases4 .
  show ?case using 1 zorder_moebius_pole[of c a d b]
    by (auto simp: modgrp_rel_def simp flip: of_int_mult)
qed
```

2.7 Induction rules in terms of generators

Theorem 2.1

```
lemma modgrp_induct_S_shift [case_names id S shift]:
  assumes "P 1"
          "⋀x. P x ⟹ P (S_modgrp * x)"
          "⋀x n. P x ⟹ P (shift_modgrp n * x)"
  shows   "P x"
proof -
  define xs where "xs = modgrp_genseq (modgrp_a x) (modgrp_b x) (modgrp_c
x) (modgrp_d x)"
  have "P (modgrp_from_gens xs)"
    by (induction xs) (auto intro: assms simp: modgrp_from_gens_Cons split:
option.splits)
  thus ?thesis using modgrp_abcd_det[of x]
    by (simp add: xs_def modgrp_genseq_correct)
qed

lemma modgrp_induct [case_names id S T inv_T]:
```

```
  assumes "P 1"
          "⋀x. P x ⟹ P (S_modgrp * x)"
          "⋀x. P x ⟹ P (T_modgrp * x)"
          "⋀x. P x ⟹ P (inverse T_modgrp * x)"
  shows   "P x"
proof -
  define xs where "xs = modgrp_genseq (modgrp_a x) (modgrp_b x) (modgrp_c
x) (modgrp_d x)"
  have *: "P (T_modgrp ^ n * x)" if "P x" for n x
    by (induction n) (auto simp: mult.assoc shift_modgrp_1 intro: assms
that)
  have **: "P (inverse T_modgrp ^ n * x)" if "P x" for n x
    by (induction n) (auto simp: shift_modgrp_add mult.assoc shift_modgrp_1
intro: assms that)
  have ***: "P (shift_modgrp n * x)" if "P x" for n x
    using *[of x "nat n"] **[of x "nat (-n)"] that
    by (auto simp: shift_modgrp_conv_T_power power_int_def)
  have "P (modgrp_from_gens xs)"
    by (induction xs) (auto intro: assms *** simp: modgrp_from_gens_Cons
split: option.splits)
  thus ?thesis using modgrp_abcd_det[of x]
    by (simp add: xs_def modgrp_genseq_correct)
qed

lemma modgrp_induct_S_shift' [case_names id S shift]:
  assumes "P 1"
          "⋀x. P x ⟹ P (x * S_modgrp)"
          "⋀x n. P x ⟹ P (x * shift_modgrp n)"
  shows   "P x"
proof -
  define xs where "xs = modgrp_genseq (modgrp_a x) (modgrp_b x) (modgrp_c
x) (modgrp_d x)"
  have "P (modgrp_from_gens xs)"
    by (induction xs rule: rev_induct) (auto intro: assms simp: modgrp_from_gens_Cons
split: option.splits)
  thus ?thesis using modgrp_abcd_det[of x]
    by (simp add: xs_def modgrp_genseq_correct)
qed

lemma modgrp_induct' [case_names id S T inv_T]:
  assumes "P 1"
          "⋀x. P x ⟹ P (x * S_modgrp)"
          "⋀x. P x ⟹ P (x * T_modgrp)"
          "⋀x. P x ⟹ P (x * inverse T_modgrp)"
  shows   "P x"
proof -
  define xs where "xs = modgrp_genseq (modgrp_a x) (modgrp_b x) (modgrp_c
x) (modgrp_d x)"
  have *: "P (x * T_modgrp ^ n)" if "P x" for n x
```

```
    using assms(3)[of "x * T_modgrp ^ n" for n]
    by (induction n)  (auto intro: that simp: algebra_simps power_commuting_commutes)
  have **: "P (x * inverse T_modgrp ^ n)" if "P x" for n x
    using assms(4)[of "x * inverse T_modgrp ^ n" for n]
    by (induction n) (auto intro: that simp: algebra_simps power_commuting_commutes)

  have ***: "P (x * shift_modgrp n)" if "P x" for n x
    using *[of x "nat n"] **[of x "nat (-n)"] that
    by (auto simp: shift_modgrp_conv_T_power power_int_def)
  have "P (modgrp_from_gens xs)"
    by (induction xs rule: rev_induct)
       (auto intro: assms *** simp: modgrp_from_gens_Cons split: option.splits)
  thus ?thesis using modgrp_abcd_det[of x]
    by (simp add: xs_def modgrp_genseq_correct)
qed

lemma moebius_uminus1: "moebius (-a) b c d = moebius a (-b) (-c) (-d)"
  by (auto simp add: moebius_def fun_eq_iff divide_simps) (auto simp:
algebra_simps add_eq_0_iff)

lemma moebius_shift:
  "moebius a b c d (z + of_int n) = moebius a (a * of_int n + b) c (c
* of_int n + d) z"
  by (simp add: moebius_def algebra_simps)

lemma moebius_eq_shift: "moebius 1 (of_int n) 0 1 z = z + of_int n"
  by (simp add: moebius_def)

lemma moebius_S:
  assumes "a * d - b * c ≠ 0" "z ≠ 0"
  shows   "moebius a b c d (-(1 / z)) = moebius b (- a) d (- c) (z ::
'a :: field)"
  using assms by (auto simp: divide_simps moebius_def)

lemma moebius_eq_S: "moebius 0 1 (-1) 0 z = -1 / z"
  by (simp add: moebius_def)

definition apply_modgrp' :: "modgrp ⇒ 'a × 'a ⇒ 'a × 'a :: ring_1"
  where "apply_modgrp' f =
           (λ(x,y). (of_int (modgrp_a f) * x + of_int (modgrp_b f) * y,

                     of_int (modgrp_c f) * x + of_int (modgrp_d f) * y))"

lemma apply_modgrp'_z_one:
  assumes "z ∉ ℝ"
  shows   "apply_modgrp' f (z, 1) = (modgrp_factor f z * apply_modgrp
f z, modgrp_factor f z)"
proof -
  from assms have "complex_of_int (modgrp_c f) * z + complex_of_int (modgrp_d
```

```
f) ≠ 0"
    by (simp add: complex_is_Real_iff modgrp.Im_transform_pos_aux)
  thus ?thesis
    by (simp add: apply_modgrp'_def modgrp_factor_def apply_modgrp_altdef
moebius_def)
qed
```

2.8 Subgroups

```
locale modgrp_subgroup =
  fixes G :: "modgrp set"
  assumes one_in_G [simp, intro]: "1 ∈ G"
  assumes times_in_G [simp, intro]: "x ∈ G ⟹ y ∈ G ⟹ x * y ∈ G"
  assumes inverse_in_G [simp, intro]: "x ∈ G ⟹ inverse x ∈ G"
begin

lemma divide_in_G [intro]: "f ∈ G ⟹ g ∈ G ⟹ f / g ∈ G"
  unfolding divide_modgrp_def by (intro times_in_G inverse_in_G)

lemma power_in_G [intro]: "f ∈ G ⟹ f ^ n ∈ G"
  by (induction n) auto

lemma power_int_in_G [intro]: "f ∈ G ⟹ f powi n ∈ G"
  by (auto simp: power_int_def)

lemma prod_list_in_G [intro]: "(⋀x. x ∈ set xs ⟹ x ∈ G) ⟹ prod_list
xs ∈ G"
  by (induction xs) auto

lemma inverse_in_G_iff [simp]: "inverse f ∈ G ⟷ f ∈ G"
  by (metis inverse_in_G modgrp.inverse_inverse)

definition rel :: "complex ⇒ complex ⇒ bool" where
  "rel x y ⟷ Im x > 0 ∧ Im y > 0 ∧ (∃f∈G. apply_modgrp f x = y)"

definition orbit :: "complex ⇒ complex set" where
  "orbit x = {y. rel x y}"

lemma Im_nonpos_imp_not_rel: "Im x ≤ 0 ∨ Im y ≤ 0 ⟹ ¬rel x y"
  by (auto simp: rel_def)

lemma orbit_empty: "Im x ≤ 0 ⟹ orbit x = {}"
  by (auto simp: orbit_def Im_nonpos_imp_not_rel)

lemma rel_imp_Im_pos [dest]:
  assumes "rel x y"
  shows   "Im x > 0" "Im y > 0"
  using assms by (auto simp: rel_def)
```

```
lemma rel_refl [simp]: "rel x x ⟷ Im x > 0"
  by (auto simp: rel_def intro!: bexI[of _ 1])

lemma rel_sym:
  assumes "rel x y"
  shows   "rel y x"
proof -
  from assms obtain f where f: "f ∈ G" "Im x > 0" "Im y > 0" "apply_modgrp
f x = y"
    by (auto simp: rel_def intro!: bexI[of _ 1])
  from this have "apply_modgrp (inverse f) y = x"
    using pole_modgrp_in_Reals[of f, where ?'a = complex]
    by (intro apply_modgrp_inverse_eqI) (auto simp: complex_is_Real_iff)
  moreover have "inverse f ∈ G"
    using f by auto
  ultimately show ?thesis
    using f by (auto simp: rel_def)
qed

lemma rel_commutes: "rel x y = rel y x"
  using rel_sym by blast

lemma rel_trans [trans]:
  assumes "rel x y" "rel y z"
  shows   "rel x z"
proof -
  from assms obtain f where f: "f ∈ G" "Im x > 0" "Im y > 0" "apply_modgrp
f x = y"
    by (auto simp: rel_def intro!: bexI[of _ 1])
  from assms obtain g where g: "Im z > 0" "g ∈ G" "apply_modgrp g y
= z"
    by (auto simp: rel_def intro!: bexI[of _ 1])
  have "apply_modgrp (g * f) x = z"
    using f g pole_modgrp_in_Reals[of f, where ?'a = complex]
    by (subst apply_modgrp_mult) (auto simp: complex_is_Real_iff)
  with f g show ?thesis
    unfolding rel_def by blast
qed

lemma relI1 [intro]: "rel x y ⟹ f ∈ G ⟹ Im x > 0 ⟹ rel x (apply_modgrp
f y)"
  using modgrp.Im_transform_pos_iff rel_def rel_trans by blast

lemma relI2 [intro]: "rel x y ⟹ f ∈ G ⟹ Im x > 0 ⟹ rel (apply_modgrp
f x) y"
  by (meson relI1 rel_commutes rel_def)

lemma rel_apply_modgrp_left_iff [simp]:
  assumes "f ∈ G"
```

```
  shows   "rel (apply_modgrp f x) y ⟷ Im x > 0 ∧ rel x y"
proof safe
  assume "rel (apply_modgrp f x) y"
  thus "rel x y"
    by (meson assms modgrp.Im_transform_pos_iff rel_def rel_trans)
next
  assume "rel x y" "Im x > 0"
  thus "rel (apply_modgrp f x) y"
    by (meson assms relI2 rel_trans)
qed auto

lemma rel_apply_modgrp_right_iff [simp]:
  assumes "f ∈ G"
  shows   "rel y (apply_modgrp f x) ⟷ Im x > 0 ∧ rel y x"
  using assms by (metis rel_apply_modgrp_left_iff rel_sym)

lemma orbit_refl_iff: "x ∈ orbit x ⟷ Im x > 0"
  by (auto simp: orbit_def)

lemma orbit_refl: "Im x > 0 ⟹ x ∈ orbit x"
  by (auto simp: orbit_def)

lemma orbit_cong: "rel x y ⟹ orbit x = orbit y"
  using rel_trans rel_commutes unfolding orbit_def by blast

lemma orbit_empty_iff [simp]: "orbit x = {} ⟷ Im x ≤ 0" "{} = orbit
x ⟷ Im x ≤ 0"
  using orbit_refl orbit_empty by force+

lemmas [simp] = orbit_refl_iff

lemma orbit_eq_iff: "orbit x = orbit y ⟷ Im x ≤ 0 ∧ Im y ≤ 0 ∨ rel
x y"
proof (cases "Im y ≤ 0 ∨ Im x ≤ 0")
  case True
  thus ?thesis
    by (auto simp: orbit_empty)
next
  case False
  have "(∀z. rel x z ⟷ rel y z) ⟷ rel x y"
    by (meson False not_le rel_commutes rel_refl rel_trans)
  thus ?thesis
    using False unfolding orbit_def by blast
qed

lemma orbit_apply_modgrp [simp]: "f ∈ G ⟹ orbit (apply_modgrp f z)
= orbit z"
  by (subst orbit_eq_iff) auto
```

```
lemma apply_modgrp_in_orbit_iff [simp]: "f ∈ G ⟹ apply_modgrp f z
∈ orbit y ⟷ z ∈ orbit y"
  by (auto simp: orbit_def rel_commutes)

lemma orbit_imp_Im_pos: "x ∈ orbit y ⟹ Im x > 0"
  by (auto simp: orbit_def)

end

interpretation modular_group: modgrp_subgroup UNIV
  by unfold_locales auto

notation modular_group.rel (infixl "∼Γ" 49)

lemma (in modgrp_subgroup) rel_imp_rel: "rel x y ⟹ x ∼Γ y"
  unfolding rel_def modular_group.rel_def by auto

lemma modular_group_rel_plus_int_iff_right1 [simp]:
  assumes "z ∈ ℤ"
  shows   "x ∼Γ y + z ⟷ x ∼Γ y"
proof -
  from assms obtain n where n: "z = of_int n"
    by (elim Ints_cases)
  have "x ∼Γ apply_modgrp (shift_modgrp n) y ⟷ x ∼Γ y"
    by (subst modular_group.rel_apply_modgrp_right_iff) auto
  thus ?thesis
    by (simp add: n)
qed

lemma
  assumes "z ∈ ℤ"
  shows   modular_group_rel_plus_int_iff_right2 [simp]: "x ∼Γ z + y ⟷
x ∼Γ y"
    and   modular_group_rel_plus_int_iff_left1 [simp]:  "z + x ∼Γ y ⟷
x ∼Γ y"
    and   modular_group_rel_plus_int_iff_left2 [simp]:  "x + z ∼Γ y ⟷
x ∼Γ y"
  using modular_group_rel_plus_int_iff_right1[OF assms] modular_group.rel_commutes
add.commute
  by metis+

lemma modular_group_rel_S_iff_right [simp]: "x ∼Γ -(1/y) ⟷ x ∼Γ y"
proof -
  have "x ∼Γ apply_modgrp S_modgrp y ⟷ x ∼Γ y"
    by (subst modular_group.rel_apply_modgrp_right_iff) auto
  thus ?thesis
    by simp
qed
```

```
lemma modular_group_rel_S_iff_left [simp]: "-(1/x) ∼Γ y ⟷ x ∼Γ y"
  using modular_group_rel_S_iff_right[of y x] by (metis modular_group.rel_commutes)
```

2.8.1 Subgroups containing shifts

```
definition modgrp_subgroup_period :: "modgrp set ⇒ nat" where
  "modgrp_subgroup_period G = nat (Gcd {n. shift_modgrp n ∈ G})"

lemma of_nat_modgrp_subgroup_period:
  "of_nat (modgrp_subgroup_period G) = Gcd {n. shift_modgrp n ∈ G}"
  unfolding modgrp_subgroup_period_def by simp

lemma ideal_int_conv_Gcd:
  fixes A :: "int set"
  assumes "0 ∈ A"
  assumes "⋀x y. x ∈ A ⟹ y ∈ A ⟹ x + y ∈ A"
  assumes "⋀x y. x ∈ A ⟹ x * y ∈ A"
  shows   "A = {n. Gcd A dvd n}"
proof
  show "A ⊆ {n. Gcd A dvd n}"
    by auto
next
  have "Gcd A ∈ A"
  proof (cases "A = {0}")
    case False
    define x :: int where "x = int (LEAST x. int x ∈ A ∧ x > 0)"
    have ex: "∃x. int x ∈ A ∧ x > 0"
    proof -
      from False obtain x where "x ∈ A - {0}"
        using assms(1) by auto
      with assms(3)[of x "-1"] show ?thesis
        by (intro exI[of _ "if x > 0 then nat x else nat (-x)"]) auto
    qed
    have x: "x ∈ A ∧ x > 0"
      unfolding x_def using LeastI_ex[OF ex] by auto
    have x': "x ≤ y" if y: "y ∈ A" "y > 0" for y
      using y unfolding x_def
      by (metis of_nat_le_iff wellorder_Least_lemma(2) zero_less_imp_eq_int)
    have "x dvd Gcd A"
    proof (rule Gcd_greatest)
      fix y assume y: "y ∈ A"
      have "y = (y div x) * x + (y mod x)"
        by simp
      have "y + x * (-1) * (y div x) ∈ A"
        by (intro assms) (use x y in auto)
      also have "y + x * (-1) * (y div x) = y mod x"
        by (simp add: algebra_simps)
      finally have "y mod x ∈ A" .
      moreover have "y mod x ≥ 0" "y mod x < x"
```

```
      using x by simp_all
    ultimately have "y mod x = 0"
      using x'[of "y mod x"] by (cases "y mod x = 0") auto
    thus "x dvd y"
      by presburger
  qed
  thus "Gcd A ∈ A"
    using assms(3) x by (auto elim!: dvdE)
qed auto
thus "{n. Gcd A dvd n} ⊆ A"
  by (auto elim!: dvdE intro!: assms)
qed


locale modgrp_subgroup_periodic = modgrp_subgroup +
  assumes periodic': "∃n>0. shift_modgrp n ∈ G"
begin

lemma modgrp_subgroup_period_pos: "modgrp_subgroup_period G > 0"
proof -
  have "Gcd {n. shift_modgrp n ∈ G} ≠ 0"
    using periodic' by (auto intro!: Nat.gr0I simp: Gcd_0_iff)
  moreover have "Gcd {n. shift_modgrp n ∈ G} ≥ 0"
    by simp
  ultimately show ?thesis
    unfolding modgrp_subgroup_period_def by linarith
qed

lemma shift_modgrp_in_G_iff: "shift_modgrp n ∈ G ⟷ int (modgrp_subgroup_period
G) dvd n"
proof -
  let ?A = "{n. shift_modgrp n ∈ G}"
  have "?A = {n. int (modgrp_subgroup_period G) dvd n}"
    unfolding of_nat_modgrp_subgroup_period
    by (rule ideal_int_conv_Gcd) (auto simp: shift_modgrp_add simp flip:
shift_modgrp_power_int)
  thus ?thesis
    by auto
qed

lemma shift_modgrp_in_G_period [intro, simp]:
  "shift_modgrp (int (modgrp_subgroup_period G)) ∈ G"
  by (subst shift_modgrp_in_G_iff) auto

lemma shift_modgrp_in_G [intro]:
  "int (modgrp_subgroup_period G) dvd n ⟹ shift_modgrp n ∈ G"
  by (subst shift_modgrp_in_G_iff) auto

end
```

interpretation *modular_group: modgrp_subgroup_periodic UNIV*
rewrites *"modgrp_subgroup_period UNIV = Suc 0"*
by *unfold_locales (auto intro: exI[of _ 1] simp: modgrp_subgroup_period_def)*

lemma *modgrp_subgroup_period_UNIV [simp]: "modgrp_subgroup_period UNIV = Suc 0"*
by *(simp add: modgrp_subgroup_period_def)*

2.8.2 Congruence subgroups

lift_definition *modgrps_cong :: "int ⇒ modgrp set"* **is**
*"λq. {(a,b,c,d) :: (int × int × int × int) | a b c d. a * d - b * c = 1 ∧ q dvd c}"*
by *(auto simp: rel_set_def modgrp_rel_def)*

lemma *modgrps_cong_altdef: "modgrps_cong q = {f. q dvd modgrp_c f}"*
by *transfer' (auto simp: modgrp_rel_def)*

lemma *modgrp_in_modgrps_cong_iff:*
assumes *"a * d - b * c = 1"*
shows *"modgrp a b c d ∈ modgrps_cong q ⟷ q dvd c"*
using *assms* **by** *(auto simp: modgrps_cong_altdef modgrp_c_code)*

lemma *modgrp_in_modgrps_cong:*
assumes *"q dvd c" "a * d - b * c = 1"*
shows *"modgrp a b c d ∈ modgrps_cong q"*
using *assms* **by** *(auto simp: modgrps_cong_altdef modgrp_c_code)*

lemma *shift_in_modgrps_cong [simp]: "shift_modgrp n ∈ modgrps_cong q"*
by *(auto simp: modgrps_cong_altdef)*

lemma *S_in_modgrps_cong_iff [simp]: "S_modgrp ∈ modgrps_cong q ⟷ is_unit q"*
by *(auto simp: modgrps_cong_altdef)*

locale *hecke_cong_subgroup =*
fixes *q :: int*
assumes *q_pos: "q > 0"*
begin

definition *subgrp ("Γ''")* **where** *"subgrp = modgrps_cong q"*

lemma *shift_in_subgrp [simp]: "shift_modgrp n ∈ subgrp"*
by *(auto simp: subgrp_def)*

lemma *S_in_subgrp_iff [simp]: "S_modgrp ∈ subgrp ⟷ q = 1"*
using *q_pos* **by** *(auto simp: subgrp_def)*

```
sublocale modgrp_subgroup Γ'
proof
  show "inverse x ∈ Γ'" if "x ∈ Γ'" for x
  proof -
    from that have "q dvd modgrp_c x"
      by (simp add: modgrps_cong_altdef subgrp_def)
    hence "q dvd modgrp_c (inverse x)"
      by transfer auto
    thus "inverse x ∈ Γ'"
      by (simp add: modgrps_cong_altdef subgrp_def)
  qed
next
  show "x * y ∈ Γ'" if "x ∈ Γ'" "y ∈ Γ'" for x y
  proof -
    from that have "q dvd modgrp_c x" "q dvd modgrp_c y"
      by (auto simp: modgrps_cong_altdef subgrp_def)
    hence "q dvd modgrp_c (x * y)"
      by transfer auto
    thus ?thesis
      by (auto simp: modgrps_cong_altdef subgrp_def)
  qed
qed (auto simp: subgrp_def modgrps_cong_altdef)

end


locale hecke_prime_subgroup =
  fixes p :: int
  assumes p_prime: "prime p"
begin

lemma p_pos: "p > 0"
  using p_prime by (simp add: prime_gt_0_int)

lemma p_not_1 [simp]: "p ≠ 1"
  using p_prime by auto

sublocale hecke_cong_subgroup p
  by standard (rule p_pos)

notation subgrp ("Γ''")

definition S_shift_modgrp where "S_shift_modgrp n = S_modgrp * shift_modgrp
n"


lemma modgrp_decompose:
  assumes "f ∉ Γ'"
  obtains g k where "g ∈ Γ'" "k ∈ {0..<p}" "f = g * S_modgrp * shift_modgrp
```

```
k"
proof -
  define a b c d where "a = modgrp_a f" "b = modgrp_b f" "c = modgrp_c
f" "d = modgrp_d f"
  have det: "a * d - b * c = 1"
    unfolding a_b_c_d_def using modgrp_abcd_det[of f] by simp
  have "¬p dvd c"
    unfolding a_b_c_d_def using assms by (auto simp: subgrp_def modgrps_cong_altdef)
  hence "coprime p c"
    using p_prime by (intro prime_imp_coprime) auto
  define k where "k = (modular_inverse p c * d) mod p"
  have "[k * c = (modular_inverse p c * d) mod p * c] (mod p)"
    by (simp add: k_def)
  also have "[(modular_inverse p c * d) mod p * c = modular_inverse p
c * d * c] (mod p)"
    by (intro cong_mult cong_mod_leftI cong_refl)
  also have "modular_inverse p c * d * c = modular_inverse p c * c * d"
    by (simp add: mult_ac)
  also have "[... = 1 * d] (mod p)" using ‹coprime p c›
    by (intro cong_mult cong_refl cong_modular_inverse2) (auto simp: coprime_commute)
  finally have "[k * c = d] (mod p)"
    by simp
  hence dvd: "p dvd k * c - d"
    by (simp add: cong_iff_dvd_diff)

  have det': "(k * a - b) * c - a * (k * c - d) = 1"
    using det by (simp add: algebra_simps)
  define g where "g = modgrp (k * a - b) a (k * c - d) c"

  show ?thesis
  proof (rule that)
    show "g ∈ Γ'"
      unfolding subgrp_def g_def using det' dvd
      by (intro modgrp_in_modgrps_cong) auto
  next
    show "k ∈ {0..<p}"
      unfolding k_def using p_pos by simp
  next
    have "g * S_modgrp * shift_modgrp k = modgrp a b c d" using det
      by (auto simp: g_def S_modgrp_code shift_modgrp_code times_modgrp_code
algebra_simps)
    also have "... = f"
      by (simp add: a_b_c_d_def)
    finally show "f = g * S_modgrp * shift_modgrp k" ..
  qed
qed

lemma modgrp_decompose':
  obtains g h
```

```
    where "g ∈ Γ'" "h = 1 ∨ (∃k∈{0..<p}. h = S_shift_modgrp k)" "f
= g * h"
proof (cases "f ∈ Γ'")
  case True
  thus ?thesis
    using that[of f 1] by auto
next
  case False
  thus ?thesis
    using modgrp_decompose[of f] that modgrp.assoc unfolding S_shift_modgrp_def
    by metis
qed

end

end
```

3 Complex lattices

```
theory Complex_Lattices
  imports "HOL-Complex_Analysis.Complex_Analysis" Parallelogram_Paths
begin


lemmas [simp del] = div_mult_self1 div_mult_self2 div_mult_self3 div_mult_self4
```

3.1 Basic definitions and useful lemmas

We define a complex lattice with two generators $\omega_1, \omega_2 \in \mathbb{C}$ as the set $\Lambda(\omega_1, \omega_2) = \omega_1\mathbb{Z} + \omega_2\mathbb{Z}$. For now, we make no restrictions on the generators, but for most of our results we will require that they be independent (i.e. neither is a multiple of the other, or, in terms of complex numbers, their quotient is not real).

```
locale pre_complex_lattice =
  fixes ω1 ω2 :: complex
begin
```

The following function convergs from lattice coordinates into cartesian coordinates.

```
definition of_ω12_coords :: "real × real ⇒ complex" where
  "of_ω12_coords = (λ(x,y). of_real x * ω1 + of_real y * ω2)"

sublocale of_ω12_coords: linear of_ω12_coords
  unfolding of_ω12_coords_def by (auto simp: linear_iff algebra_simps
scaleR_conv_of_real)

sublocale of_ω12_coords: bounded_linear of_ω12_coords
```

using *of_ω12_coords.linear_axioms linear_conv_bounded_linear* **by** *auto*

lemmas *[continuous_intros] = of_ω12_coords.continuous_on of_ω12_coords.continuous*
lemmas *[tendsto_intros] = of_ω12_coords.tendsto*

lemmas *[simp] = of_ω12_coords.add of_ω12_coords.diff of_ω12_coords.neg of_ω12_coords.scaleR*

lemma *of_ω12_coords_fst [simp]: "of_ω12_coords (a, 0) = of_real a * ω1"*
  and *of_ω12_coords_snd [simp]: "of_ω12_coords (0, a) = of_real a * ω2"*
  and *of_ω12_coords_scaleR': "of_ω12_coords (c *$_R$ z) = of_real c * of_ω12_coords z"*
  by *(simp_all add: of_ω12_coords_def algebra_simps case_prod_unfold scaleR_prod_def scaleR_conv_of_real)*

The following is our lattice as a set of lattice points.

definition *lattice :: "complex set" ("Λ")* **where**
  "lattice = of_ω12_coords ‘ (ℤ × ℤ)"

definition *lattice0 :: "complex set" ("Λ*") **where**
  "lattice0 = lattice - {0}"

lemma *countable_lattice [intro]: "countable lattice"*
  unfolding *lattice_def* **by** *(intro countable_image countable_SIGMA countable_int)*

lemma *latticeI: "of_ω12_coords (x, y) = z ⟹ x ∈ ℤ ⟹ y ∈ ℤ ⟹ z ∈ Λ"*
  by *(auto simp: lattice_def)*

lemma *latticeE:*
  assumes *"z ∈ Λ"*
  obtains *x y* **where** *"z = of_ω12_coords (of_int x, of_int y)"*
  using *assms* **unfolding** *lattice_def Ints_def* **by** *auto*

lemma *lattice0I [intro]: "z ∈ Λ ⟹ z ≠ 0 ⟹ z ∈ Λ*"*
  by *(auto simp: lattice0_def)*

lemma *lattice0E [elim]: "⋀P. z ∈ Λ* ⟹ (z ∈ Λ ⟹ z ≠ 0 ⟹ P) ⟹ P"*
  by *(auto simp: lattice0_def)*

lemma *in_lattice0_iff: "z ∈ Λ* ⟷ z ∈ Λ ∧ z ≠ 0"*
  by *(auto simp: lattice0_def)*

named_theorems *lattice_intros*

lemma *zero_in_lattice [lattice_intros, simp]: "0 ∈ lattice"*

```
  by (rule latticeI[of 0 0]) (auto simp: of_ω12_coords_def)

lemma generator_in_lattice [lattice_intros, simp]: "ω1 ∈ lattice" "ω2
∈ lattice"
  by (auto intro: latticeI[of 0 1] latticeI[of 1 0] simp: of_ω12_coords_def)

lemma uminus_in_lattice [lattice_intros]: "z ∈ Λ ⟹ -z ∈ Λ"
proof -
  assume "z ∈ Λ"
  then obtain x y where "z = of_ω12_coords (of_int x, of_int y)"
    by (elim latticeE)
  thus ?thesis
    by (intro latticeI[of "-x" "-y"]) (auto simp: of_ω12_coords_def)
qed

lemma uminus_in_lattice_iff: "-z ∈ Λ ⟷ z ∈ Λ"
  using uminus_in_lattice minus_minus by metis

lemma uminus_in_lattice0_iff: "-z ∈ Λ* ⟷ z ∈ Λ*"
  by (auto simp: lattice0_def uminus_in_lattice_iff)

lemma add_in_lattice [lattice_intros]: "z ∈ Λ ⟹ w ∈ Λ ⟹ z + w ∈
Λ"
proof -
  assume "z ∈ Λ" "w ∈ Λ"
  then obtain a b c d
    where "z = of_ω12_coords (of_int a, of_int b)" "w = of_ω12_coords
(of_int c, of_int d)"
    by (elim latticeE)
  thus ?thesis
    by (intro latticeI[of "a + c" "b + d"]) (auto simp: of_ω12_coords_def
algebra_simps)
qed

lemma lattice_lattice0: "Λ = insert 0 Λ*"
  by (auto simp: lattice0_def)

lemma mult_of_nat_left_in_lattice [lattice_intros]: "z ∈ Λ ⟹ of_nat
n * z ∈ Λ"
  by (induction n) (auto intro!: lattice_intros simp: ring_distribs)

lemma mult_of_nat_right_in_lattice [lattice_intros]: "z ∈ Λ ⟹ z *
of_nat n ∈ Λ"
  using mult_of_nat_left_in_lattice[of z n] by (simp add: mult.commute)

lemma mult_of_int_left_in_lattice [lattice_intros]: "z ∈ Λ ⟹ of_int
n * z ∈ Λ"
  using mult_of_nat_left_in_lattice[of z "nat n"]
        uminus_in_lattice[OF mult_of_nat_left_in_lattice[of z "nat (-n)"]]
```

```
  by (cases "n ≥ 0") auto

lemma mult_of_int_right_in_lattice [lattice_intros]: "z ∈ Λ ⟹ z *
of_int n ∈ Λ"
  using mult_of_int_left_in_lattice[of z n] by (simp add: mult.commute)

lemma diff_in_lattice [lattice_intros]: "z ∈ Λ ⟹ w ∈ Λ ⟹ z - w
∈ Λ"
  using add_in_lattice[OF _ uminus_in_lattice, of z w] by simp

lemma diff_in_lattice_commute: "z - w ∈ Λ ⟷ w - z ∈ Λ"
  using uminus_in_lattice_iff[of "z - w"] by simp

lemma of_ω12_coords_in_lattice [lattice_intros]: "ab ∈ ℤ × ℤ ⟹ of_ω12_coords
ab ∈ Λ"
  unfolding lattice_def by auto

lemma lattice_plus_right_cancel [simp]: "y ∈ Λ ⟹ x + y ∈ Λ ⟷ x
∈ Λ"
  by (metis add_diff_cancel_right' add_in_lattice diff_in_lattice)

lemma lattice_plus_left_cancel [simp]: "x ∈ Λ ⟹ x + y ∈ Λ ⟷ y ∈
Λ"
  by (metis add.commute lattice_plus_right_cancel)

lemma lattice_induct [consumes 1, case_names zero gen1 gen2 add uminus]:
  assumes "z ∈ Λ"
  assumes zero: "P 0"
  assumes gens: "P ω1" "P ω2"
  assumes plus: "⋀w z. P w ⟹ P z ⟹ P (w + z)"
  assumes uminus: "⋀w. P w ⟹ P (-w)"
  shows   "P z"
proof -
  from assms(1) obtain a b where z_eq: "z = of_ω12_coords (of_int a,
of_int b)"
    by (elim latticeE)
  have nat1: "P (of_ω12_coords (of_nat n, 0))" for n
    by (induction n) (auto simp: of_ω12_coords_def ring_distribs intro:
zero plus gens)
  have int1: "P (of_ω12_coords (of_int n, 0))" for n
    using nat1[of "nat n"] uminus[OF nat1[of "nat (-n)"]]
    by (cases "n ≥ 0") (auto simp: of_ω12_coords_def)
  have nat2: "P (of_ω12_coords (of_int a, of_nat n))" for a n
  proof (induction n)
    case 0
    thus ?case
      using int1[of a] by simp
  next
    case (Suc n)
```

```
    from plus[OF Suc gens(2)] show ?case
      by (simp add: of_ω12_coords_def algebra_simps)
  qed
  have int2: "P (of_ω12_coords (of_int a, of_int b))" for a b
    using nat2[of a "nat b"] uminus[OF nat2[of "-a" "nat (-b)"]]
    by (cases "b ≥ 0") (auto simp: of_ω12_coords_def)
  from this[of a b] and z_eq show ?thesis
    by simp
qed
```

The following equivalence relation equates two points if they differ by a lattice point.

```
definition rel :: "complex ⇒ complex ⇒ bool" where
  "rel x y ⟷ (x - y) ∈ Λ"

lemma rel_refl [simp, intro]: "rel x x"
  by (auto simp: rel_def)

lemma relE:
  assumes "rel x y"
  obtains z where "z ∈ Λ" "y = x + z"
  using assms unfolding rel_def using pre_complex_lattice.uminus_in_lattice
by force

lemma rel_symI: "rel x y ⟹ rel y x"
  by (auto simp: rel_def diff_in_lattice_commute)

lemma rel_sym: "rel x y ⟷ rel y x"
  by (auto simp: rel_def diff_in_lattice_commute)

lemma rel_0_right_iff: "rel x 0 ⟷ x ∈ Λ"
  by (simp add: rel_def)

lemma rel_0_left_iff: "rel 0 x ⟷ x ∈ Λ"
  by (simp add: rel_def uminus_in_lattice_iff)

lemma rel_trans [trans]: "rel x y ⟹ rel y z ⟹ rel x z"
  using add_in_lattice rel_def by fastforce

lemma rel_minus [lattice_intros]: "rel a b ⟹ rel (-a) (-b)"
  unfolding rel_def by (simp add: diff_in_lattice_commute)

lemma rel_minus_iff: "rel (-a) (-b) ⟷ rel a b"
  by (auto simp: rel_def diff_in_lattice_commute)

lemma rel_add [lattice_intros]: "rel a b ⟹ rel c d ⟹ rel (a + c)
(b + d)"
  unfolding rel_def by (simp add: add_diff_add add_in_lattice)
```

lemma *rel_diff [lattice_intros]: "rel a b ⟹ rel c d ⟹ rel (a - c) (b - d)"*
by *(metis rel_add rel_minus uminus_add_conv_diff)*

lemma *rel_mult_of_nat_left [lattice_intros]: "rel a b ⟹ rel (of_nat n * a) (of_nat n * b)"*
by *(induction n) (auto intro!: lattice_intros simp: ring_distribs)*

lemma *rel_mult_of_nat_right [lattice_intros]: "rel a b ⟹ rel (a * of_nat n) (b * of_nat n)"*
by *(induction n) (auto intro!: lattice_intros simp: ring_distribs)*

lemma *rel_mult_of_int_left [lattice_intros]: "rel a b ⟹ rel (of_int n * a) (of_int n * b)"*
by *(induction n) (auto intro!: lattice_intros simp: ring_distribs)*

lemma *rel_mult_of_int_right [lattice_intros]: "rel a b ⟹ rel (a * of_int n) (b * of_int n)"*
by *(induction n) (auto intro!: lattice_intros simp: ring_distribs)*

lemma *rel_sum [lattice_intros]:*
"(⋀i. i ∈ A ⟹ rel (f i) (g i)) ⟹ rel (∑ i∈A. f i) (∑ i∈A. g i)"
by *(induction A rule: infinite_finite_induct) (auto intro!: lattice_intros)*

lemma *rel_sum_list [lattice_intros]:*
"list_all2 rel xs ys ⟹ rel (sum_list xs) (sum_list ys)"
by *(induction rule: list_all2_induct) (auto intro!: lattice_intros)*

lemma *rel_lattice_trans_left [trans]: "x ∈ Λ ⟹ rel x y ⟹ y ∈ Λ"*
using *rel_0_left_iff rel_trans* **by** *blast*

lemma *rel_lattice_trans_right [trans]: "rel x y ⟹ y ∈ Λ ⟹ x ∈ Λ"*
using *rel_lattice_trans_left rel_sym* **by** *blast*

end

Exchanging the two generators clearly does not change the underlying lattice.

locale *pre_complex_lattice_swap = pre_complex_lattice*
begin

sublocale *swap: pre_complex_lattice ω2 ω1 .*

lemma *swap_of_ω12_coords [simp]: "swap.of_ω12_coords = of_ω12_coords ∘ prod.swap"*
by *(auto simp: fun_eq_iff swap.of_ω12_coords_def of_ω12_coords_def add_ac)*

lemma *swap_lattice [simp]: "swap.lattice = lattice"*

unfolding *swap.lattice_def lattice_def swap_of_ω12_coords image_comp [symmetric] product_swap* **..**

lemma *swap_lattice0 [simp]: "swap.lattice0 = lattice0"*
unfolding *swap.lattice0_def lattice0_def swap_lattice* **..**

lemma *swap_rel [simp]: "swap.rel = rel"*
unfolding *swap.rel_def rel_def swap_lattice* **..**

end

A pair *(ω₁, ω₂)* of complex numbers with *ω₂ / ω₁* $\notin \mathbb{R}$ is called a *fundamental pair*. Two such pairs are called *equivalent* if

definition *fundpair :: "complex × complex ⇒ bool"* **where**
"fundpair = (λ(a, b). b / a ∉ ℝ)"

lemma *fundpair_swap: "fundpair ab ⟷ fundpair (prod.swap ab)"*
unfolding *fundpair_def prod.swap_def case_prod_unfold fst_conv snd_conv*
by *(metis Un_insert_right collinear_iff_Reals insert_is_Un)*

lemma *fundpair_cnj_iff [simp]: "fundpair (cnj a, cnj b) = fundpair (a, b)"*
by *(auto simp: fundpair_def complex_is_Real_iff simp flip: complex_cnj_divide)*

lemma *fundpair_altdef: "fundpair = (λ(a,b). a / b ∉ ℝ)"*
proof
fix *ab :: "complex × complex"*
show *"fundpair ab = (case ab of (a, b) ⇒ a / b ∉ ℝ)"*
using *fundpair_swap[of ab]* **by** *(auto simp: fundpair_def)*
qed

lemma
assumes *"fundpair (a, b)"*
shows *fundpair_imp_nonzero [dest]: "a ≠ 0" "b ≠ 0"*
and *fundpair_imp_neq: "a ≠ b" "b ≠ a"*
using *assms* **unfolding** *fundpair_def* **by** *(auto split: if_splits)*

lemma *fundpair_imp_independent:*
assumes *"fundpair (ω1, ω2)"*
shows *"independent {ω1, ω2}"*
proof
assume *"dependent {ω1, ω2}"*
then obtain *a b* **where** *ab: "a *ℝ ω1 + b *ℝ ω2 = 0"* **and** *"a ≠ 0 ∨ b ≠ 0"*
by *(subst (asm) real_vector.dependent_finite) (use assms* **in** *<auto dest: fundpair_imp_neq>)*
with *assms* **have** *[simp]: "a ≠ 0" "b ≠ 0"*
by *auto*
from *ab* **have** *"ω2 / ω1 = of_real (-a / b)"*

```
    using assms by (auto simp: field_simps scaleR_conv_of_real add_eq_0_iff)
  also have "... ∈ ℝ"
    by simp
  finally show False
    using assms by (auto simp: fundpair_def)
qed

lemma fundpair_imp_basis:
  assumes "fundpair (ω1, ω2)"
  shows   "span {ω1, ω2} = UNIV"
proof -
  have "dim (span {ω1, ω2}) = card {ω1, ω2}"
    using fundpair_imp_independent[OF assms] by (rule dim_span_eq_card_independent)
  hence "dim (span {ω1, ω2}) = DIM(complex)"
    using fundpair_imp_neq(1)[OF assms] by simp
  thus ?thesis
    using dim_eq_full span_span by blast
qed
```

We now introduce the assumption that the generators be independent. This makes $\{\omega_1, \omega_2\}$ a basis of $\mathbb{C}$ (in the sense of an $\mathbb{R}$-vector space), and we define a few functions to help us convert between these two views.

```
locale complex_lattice = pre_complex_lattice +
  assumes fundpair: "fundpair (ω1, ω2)"
begin

lemma ω1_neq_ω2 [simp]: "ω1 ≠ ω2" and ω2_neq_ω1 [simp]: "ω2 ≠ ω1"
  using fundpair fundpair_imp_neq by blast+

lemma ω1_nonzero [simp]: "ω1 ≠ 0" and ω2_nonzero [simp]: "ω2 ≠ 0"
  using fundpair by auto

lemma lattice0_nonempty [simp]: "lattice0 ≠ {}"
proof -
  have "ω1 ∈ lattice0"
    by auto
  thus ?thesis
    by blast
qed

lemma ω12_independent': "independent {ω1, ω2}"
  using fundpair by (rule fundpair_imp_independent)

lemma span_ω12: "span {ω1, ω2} = UNIV"
  using fundpair by (rule fundpair_imp_basis)
```

The following converts complex numbers into lattice coordinates, i.e. as a linear combination of the two generators.

```
definition ω1_coord :: "complex ⇒ real" where
```

```
  "ω1_coord z = representation {ω1, ω2} z ω1"

definition ω2_coord :: "complex ⇒ real" where
  "ω2_coord z = representation {ω1, ω2} z ω2"

definition ω12_coords :: "complex ⇒ real × real" where
  "ω12_coords z = (ω1_coord z, ω2_coord z)"

sublocale ω1_coord: bounded_linear ω1_coord
  unfolding ω1_coord_def using ω12_independent' span_ω12
  by (rule bounded_linear_representation)

sublocale ω2_coord: bounded_linear ω2_coord
  unfolding ω2_coord_def using ω12_independent' span_ω12
  by (rule bounded_linear_representation)

sublocale ω12_coords: linear ω12_coords
  unfolding ω12_coords_def
  by (auto simp: linear_iff ω1_coord.add ω2_coord.add ω1_coord.scaleR
ω2_coord.scaleR)

sublocale ω12_coords: bounded_linear ω12_coords
  using ω12_coords.linear_axioms linear_conv_bounded_linear by auto

lemmas [continuous_intros] =
  ω1_coord.continuous_on ω1_coord.continuous
  ω2_coord.continuous_on ω2_coord.continuous
  ω12_coords.continuous_on ω12_coords.continuous

lemmas [tendsto_intros] = ω1_coord.tendsto ω2_coord.tendsto ω12_coords.tendsto

lemma ω1_coord_ω1 [simp]: "ω1_coord ω1 = 1"
  and ω1_coord_ω2 [simp]: "ω1_coord ω2 = 0"
  and ω2_coord_ω1 [simp]: "ω2_coord ω1 = 0"
  and ω2_coord_ω2 [simp]: "ω2_coord ω2 = 1"
  unfolding ω1_coord_def ω2_coord_def using ω1_neq_ω2
  by (simp_all add: ω12_independent' representation_basis)

lemma ω12_coords_ω1 [simp]: "ω12_coords ω1 = (1, 0)"
  and ω12_coords_ω2 [simp]: "ω12_coords ω2 = (0, 1)"
  by (simp_all add: ω12_coords_def)

lemma ω12_coords_of_ω12_coords [simp]: "ω12_coords (of_ω12_coords z)
= z"
  by (simp add: of_ω12_coords_def case_prod_unfold ω12_coords.add ω12_coords.scaleR
          flip: scaleR_conv_of_real)

lemma ω1_coord_of_ω12_coords [simp]: "ω1_coord (of_ω12_coords z) =
fst z"
```

```
  and ω2_coord_of_ω12_coords [simp]: "ω2_coord (of_ω12_coords z) = snd
z"
  using ω12_coords_of_ω12_coords[of z]
  by (auto simp del: ω12_coords_of_ω12_coords simp add: ω12_coords_def
prod_eq_iff)

lemma of_ω12_coords_ω12_coords [simp]: "of_ω12_coords (ω12_coords z)
= z"
proof -
  have "(∑ b∈{ω1, ω2}. representation {ω1, ω2} z b *R b) = z"
    by (rule real_vector.sum_representation_eq) (use ω12_independent'
span_ω12 in simp_all)
  thus ?thesis
    by (simp add: ω12_coords_def of_ω12_coords_def scaleR_conv_of_real
                  ω1_coord_def ω2_coord_def ω1_neq_ω2)
qed

lemma ω12_coords_eqI:
  assumes "of_ω12_coords a = b"
  shows   "ω12_coords b = a"
  unfolding assms[symmetric] by auto

lemmas [simp] = ω1_coord.scaleR ω2_coord.scaleR ω12_coords.scaleR

lemma ω12_coords_times_ω1 [simp]: "ω12_coords (of_real a * ω1) = (a,
0)"
  and ω12_coords_times_ω2 [simp]: "ω12_coords (of_real a * ω2) = (0,
a)"
  and ω12_coords_times_ω1' [simp]: "ω12_coords (ω1 * of_real a) = (a,
0)"
  and ω12_coords_times_ω2' [simp]: "ω12_coords (ω2 * of_real a) = (0,
a)"
  and ω12_coords_mult_of_real [simp]: "ω12_coords (of_real c * z) = c
*R  ω12_coords z"
  and ω12_coords_mult_of_int [simp]: "ω12_coords (of_int i * z) = of_int
i *R  ω12_coords z"
  and ω12_coords_mult_of_nat [simp]: "ω12_coords (of_nat n * z) = of_nat
n *R  ω12_coords z"
  and ω12_coords_divide_of_real [simp]: "ω12_coords (z / of_real c) =
ω12_coords z /R c"
  and ω12_coords_mult_numeral [simp]: "ω12_coords (numeral num * z) =
numeral num *R  ω12_coords z"
  and ω12_coords_divide_numeral [simp]: "ω12_coords (z / numeral num)
= ω12_coords z /R numeral num"
  by (rule ω12_coords_eqI; simp add: scaleR_conv_of_real field_simps;
fail)+

lemma of_ω12_coords_eq_iff: "of_ω12_coords z1 = of_ω12_coords z2 ⟷
z1 = z2"
```

```
  using ω12_coords_eqI by blast

lemma ω12_coords_eq_iff: "ω12_coords z1 = ω12_coords z2 ⟷ z1 = z2"
  by (metis of_ω12_coords_ω12_coords)

lemma of_ω12_coords_eq_0_iff [simp]: "of_ω12_coords z = 0 ⟷ z = (0,0)"
  unfolding zero_prod_def [symmetric]
  by (metis ω12_coords.zero ω12_coords_eqI of_ω12_coords_ω12_coords)

lemma ω12_coords_eq_0_0_iff [simp]: "ω12_coords x = (0, 0) ⟷ x =
0"
  by (metis ω12_coords.zero ω12_coords_eq_iff zero_prod_def)

lemma bij_of_ω12_coords: "bij of_ω12_coords"
proof -
  have "∃z'. z = of_ω12_coords z'" for z
    by (rule exI[of _ "ω12_coords z"]) auto
  hence "surj of_ω12_coords"
    by blast
  thus ?thesis
    unfolding bij_def by (auto intro!: injI simp: of_ω12_coords_eq_iff)
qed

lemma bij_betw_lattice: "bij_betw of_ω12_coords (ℤ × ℤ) lattice"
  unfolding lattice_def using bij_of_ω12_coords unfolding bij_betw_def
inj_on_def by blast

lemma bij_betw_lattice0: "bij_betw of_ω12_coords (ℤ × ℤ - {(0,0)})
lattice0"
  unfolding lattice0_def by (intro bij_betw_DiffI bij_betw_singletonI
bij_betw_lattice) auto

lemma bij_betw_lattice': "bij_betw (of_ω12_coords ∘ map_prod of_int
of_int) UNIV lattice"
  by (rule bij_betw_trans[OF _ bij_betw_lattice]) (auto simp: Ints_def
bij_betw_def inj_on_def)

lemma bij_betw_lattice0': "bij_betw (of_ω12_coords ∘ map_prod of_int
of_int) (-{(0,0)}) lattice0"
  by (rule bij_betw_trans[OF _ bij_betw_lattice0]) (auto simp: Ints_def
bij_betw_def inj_on_def)

lemma infinite_lattice: "¬finite lattice"
proof -
  have "finite (UNIV :: (int × int) set) ⟷ finite lattice"
    by (rule bij_betw_finite[OF bij_betw_lattice'])
  moreover have "¬finite (UNIV :: (int × int) set)"
    by (simp add: finite_prod)
  ultimately show ?thesis
```

```
    by blast
qed

lemma ω12_coords_image_eq: "ω12_coords ‘ X = of_ω12_coords -‘ X"
  using bij_of_ω12_coords image_iff by fastforce

lemma of_ω12_coords_image_eq: "of_ω12_coords ‘ X = ω12_coords -‘ X"
  by (metis UNIV_I ω12_coords_eqI ω12_coords_image_eq bij_betw_imp_surj_on
      bij_of_ω12_coords rangeI subsetI subset_antisym surj_image_vimage_eq)

lemma of_ω12_coords_linepath:
   "of_ω12_coords (linepath a b x) = linepath (of_ω12_coords a) (of_ω12_coords
b) x"
  by (simp add: linepath_def scaleR_prod_def scaleR_conv_of_real
                of_ω12_coords_def algebra_simps case_prod_unfold)

lemma of_ω12_coords_linepath':
  "of_ω12_coords o (linepath a b) =
      linepath (of_ω12_coords a) (of_ω12_coords b)"
  unfolding comp_def using of_ω12_coords_linepath
  by auto

lemma ω12_coords_linepath:
   "ω12_coords (linepath a b x) = linepath (ω12_coords a) (ω12_coords
b) x"
  by (rule ω12_coords_eqI) (simp add: of_ω12_coords_linepath)

lemma of_ω12_coords_in_lattice_iff:
  "of_ω12_coords z ∈ Λ ⟷ fst z ∈ ℤ ∧ snd z ∈ ℤ"
proof
  assume "of_ω12_coords z ∈ Λ"
  then obtain m n where mn: "of_ω12_coords z = of_ω12_coords (of_int
m, of_int n)"
    by (elim latticeE)
  hence "z = (of_int m, of_int n)"
    by (simp only: of_ω12_coords_eq_iff)
  thus "fst z ∈ ℤ ∧ snd z ∈ ℤ"
    by auto
next
  assume "fst z ∈ ℤ ∧ snd z ∈ ℤ"
  thus "of_ω12_coords z ∈ Λ"
    by (simp add: latticeI of_ω12_coords_def split_def)
qed

lemma of_ω12_coords_in_lattice [simp, intro]:
  "fst z ∈ ℤ ⟹ snd z ∈ ℤ ⟹ of_ω12_coords z ∈ Λ"
  by (subst of_ω12_coords_in_lattice_iff) auto

lemma in_lattice_conv_ω12_coords: "z ∈ Λ ⟷ ω12_coords z ∈ ℤ × ℤ"
```

```
  using of_ω12_coords_in_lattice_iff[of "ω12_coords z"] by (auto simp:
mem_Times_iff)

lemma ω12_coords_in_Z_times_Z: "z ∈ Λ ⟹ ω12_coords z ∈ ℤ × ℤ"
  by (subst (asm) in_lattice_conv_ω12_coords) auto

lemma half_periods_notin_lattice [simp]:
  "ω1 / 2 ∉ Λ" "ω2 / 2 ∉ Λ" "(ω1 + ω2) / 2 ∉ Λ"
  by (auto simp: in_lattice_conv_ω12_coords ω12_coords.add)

end


locale complex_lattice_swap = complex_lattice
begin

sublocale pre_complex_lattice_swap ω1 ω2 .

sublocale swap: complex_lattice ω2 ω1
proof
  show "fundpair (ω2, ω1)"
    using fundpair by (subst (asm) fundpair_swap) auto
qed

lemma swap_ω12_coords [simp]: "swap.ω12_coords = prod.swap ∘ ω12_coords"
  by (metis (no_types, lifting) ext comp_apply of_ω12_coords_ω12_coords
      pre_complex_lattice_swap.swap_of_ω12_coords swap.ω12_coords_eqI)

lemma swap_ω1_coord [simp]: "swap.ω1_coord = ω2_coord"
  and swap_ω2_coord [simp]: "swap.ω2_coord = ω1_coord"
  using swap_ω12_coords unfolding swap.ω12_coords_def[abs_def] ω12_coords_def[abs_def]
  by (auto simp: fun_eq_iff)

end
```

3.2 Period parallelograms

```
context pre_complex_lattice
begin
```

The period parallelogram at a vertex `z` is the parallelogram with the vertices `z`, `z + ω1`, `z + ω2`, and `z + ω1 + ω2`. For convenience, we define the parallelogram to be contain only two of its four sides, so that one can obtain an exact covering of the complex plane with period parallelograms.

We will occasionally need the full parallelogram with all four sides, or the interior of the parallelogram without its four sides, but these are easily obtained from this using the `closure` and `interior` operators, while the border itself (which is of interest for integration) is obtained with the `frontier` operator.

definition *period_parallelogram :: "complex ⇒ complex set"* **where**
"period_parallelogram z = (+) z ‘ of_ω12_coords ‘ ({0..<1} × {0..<1})"

The following is a path along the border of a period parallelogram, starting at the vertex *z* and going in direction *ω1*.

definition *period_parallelogram_path :: "complex ⇒ real ⇒ complex"* **where**
"period_parallelogram_path z ≡ parallelogram_path z ω1 ω2"

lemma *bounded_period_parallelogram [intro]: "bounded (period_parallelogram z)"*
unfolding *period_parallelogram_def*
by *(rule bounded_translation bounded_linear_image bounded_Times)+*
(auto intro: of_ω12_coords.bounded_linear_axioms)

lemma *convex_period_parallelogram [intro]:*
"convex (period_parallelogram z)"
unfolding *period_parallelogram_def*
by *(intro convex_translation convex_linear_image of_ω12_coords.linear_axioms convex_Times) auto*

lemma *closure_period_parallelogram:*
"closure (period_parallelogram z) = (+) z ‘ of_ω12_coords ‘ (cbox (0,0) (1,1))"
proof -
have *"closure (period_parallelogram z) = (+) z ‘ closure (of_ω12_coords ‘ ({0..<1} × {0..<1}))"*
unfolding *period_parallelogram_def* **by** *(subst closure_translation) auto*
also have *"closure (of_ω12_coords ‘ ({0..<1} × {0..<1})) =*
of_ω12_coords ‘ (closure ({0..<1} × {0..<1}))"
by *(rule closure_bounded_linear_image [symmetric])*
(auto intro: bounded_Times of_ω12_coords.linear_axioms)
also have *"closure ({0..<1::real} × {0..<1::real}) = {0..1} × {0..1}"*
by *(simp add: closure_Times)*
also have *"... = cbox (0,0) (1,1)"*
by *auto*
finally show *?thesis* .
qed

lemma *compact_closure_period_parallelogram [intro]: "compact (closure (period_parallelogram z))"*
unfolding *closure_period_parallelogram*
by *(intro compact_translation compact_continuous_image continuous_intros compact_Times) auto*

lemma *vertex_in_period_parallelogram [simp, intro]: "z ∈ period_parallelogram z"*
unfolding *period_parallelogram_def image_image*
by *(rule image_eqI[of _ _ "(0,0)"]) auto*

```
lemma nonempty_period_parallelogram: "period_parallelogram z ≠ {}"
  using vertex_in_period_parallelogram[of z] by blast

end


lemma (in pre_complex_lattice_swap)
  swap_period_parallelogram [simp]: "swap.period_parallelogram = period_parallelogram"
  unfolding swap.period_parallelogram_def period_parallelogram_def swap_of_ω12_coords
            image_comp [symmetric] product_swap ..


context complex_lattice
begin

lemma simple_path_parallelogram: "simple_path (parallelogram_path z ω1
ω2)"
  unfolding parallelogram_path_altdef
proof (rule simple_path_continuous_image)
  let ?h = "λw. z + Re w *R ω1 + Im w *R ω2"
  show "simple_path (rectpath 0 (1 + i))"
    by (intro simple_path_rectpath) auto
  show "continuous_on (path_image (rectpath 0 (1 + i))) ?h"
    by (intro continuous_intros)
  show "inj_on ?h (path_image (rectpath 0 (1 + i)))"
  proof
    fix x y assume "?h x = ?h y"
    hence "of_ω12_coords (Re x, Im x) = of_ω12_coords (Re y, Im y)"
      by (simp add: of_ω12_coords_def scaleR_conv_of_real)
    thus "x = y"
      by (intro complex_eqI) (simp_all add: of_ω12_coords_eq_iff)
  qed
qed

lemma (in -) image_plus_conv_vimage_plus:
  fixes c :: "'a :: group_add"
  shows "(+) c ‘ A = (+) (-c) -‘ A"
proof safe
  fix z assume "-c + z ∈ A"
  thus "z ∈ (+) c ‘ A"
    by (intro image_eqI[of _ _ "-c + z"]) (auto simp: algebra_simps)
qed auto

lemma period_parallelogram_altdef:
  "period_parallelogram z = {w. ω12_coords (w - z) ∈ {0..<1} × {0..<1}}"
  unfolding period_parallelogram_def of_ω12_coords_image_eq image_plus_conv_vimage_plus
  by auto
```

```
lemma interior_period_parallelogram:
  "interior (period_parallelogram z) = (+) z ‘ of_ω12_coords ‘ box (0,0)
(1,1)"
proof -
  have bij: "bij of_ω12_coords"
    by (simp add: bij_of_ω12_coords)
  have "interior (period_parallelogram z) = (+) z ‘ interior (of_ω12_coords
‘ ({0..<1} × {0..<1}))"
    unfolding period_parallelogram_def by (subst interior_translation)
auto
  also have "interior (of_ω12_coords ‘ ({0..<1} × {0..<1})) =
             of_ω12_coords ‘ (interior ({0..<1} × {0..<1}))"
    using of_ω12_coords.linear_axioms bij
     by (rule interior_bijective_linear_image)
  also have "interior ({0..<1::real} × {0..<1::real}) = {0<..<1} × {0<..<1}"
    by (subst interior_Times) simp_all
  finally show ?thesis by (auto simp: box_prod)
qed

lemma path_image_parallelogram_path':
  "path_image (parallelogram_path z ω1 ω2) =
     (+) z ‘ of_ω12_coords ‘ (cbox (0,0) (1,1) - box (0,0) (1,1))"
proof -
  define f where "f = (λx. (Re x, Im x))"
  have "bij f"
    by (rule bij_betwI[of _ _ _ "λ(x,y). Complex x y"]) (auto simp: f_def)
  hence "inj f" "surj f"
    using bij_is_inj bij_is_surj by auto
  have "path_image (parallelogram_path z ω1 ω2) =
          (λw. z + Re w *R ω1 + Im w *R ω2) ‘ (cbox 0 (1 + i) - box 0
(1 + i))"
    (is "_ = _ ‘ ?S")
    unfolding parallelogram_path_altdef period_parallelogram_altdef path_image_compose
    by (subst path_image_rectpath_cbox_minus_box) auto
  also have "(λw. z + Re w *R ω1 + Im w *R ω2) = (+) z ∘ of_ω12_coords
∘ f"
    by (auto simp: of_ω12_coords_def fun_eq_iff scaleR_conv_of_real f_def)
  also have "... ‘ (cbox 0 (1 + i) - box 0 (1 + i)) =
            (+) z ‘ of_ω12_coords ‘ f ‘ ((cbox 0 (1 + i) - box 0 (1 +
i)))"
    by (simp add: image_image)
  also have "f ‘ ((cbox 0 (1 + i) - box 0 (1 + i))) = f ‘ cbox 0 (1 + i)
- f ‘ box 0 (1 + i)"
    by (rule image_set_diff[OF <inj f>])
  also have "cbox 0 (1 + i) = f -‘ cbox (0,0) (1,1)"
    by (auto simp: f_def cbox_complex_eq)
  also have "f ‘ ... = cbox (0,0) (1,1)"
    by (rule surj_image_vimage_eq[OF <surj f>])
  also have "box 0 (1 + i) = f -‘ box (0,0) (1,1)"
```

```
    by (auto simp: f_def box_complex_eq box_prod)
  also have "f ‘ ... = box (0,0) (1,1)"
    by (rule surj_image_vimage_eq[OF ‹surj f›])
  finally show ?thesis .
qed

lemma fund_period_parallelogram_in_lattice_iff:
  assumes "z ∈ period_parallelogram 0"
  shows   "z ∈ Λ ⟷ z = 0"
proof
  assume "z ∈ Λ"
  then obtain m n where mn: "z = of_ω12_coords (of_int m, of_int n)"
    by (elim latticeE)
  show "z = 0"
    using assms unfolding mn period_parallelogram_altdef  by auto
qed auto

lemma path_image_parallelogram_path:
  "path_image (parallelogram_path z ω1 ω2) = frontier (period_parallelogram
z)"
  unfolding frontier_def closure_period_parallelogram interior_period_parallelogram
            path_image_parallelogram_path'
  by (subst image_set_diff) (auto intro!: inj_onI simp: of_ω12_coords_eq_iff)

lemma path_image_parallelogram_subset_closure:
  "path_image (parallelogram_path z ω1 ω2) ⊆ closure (period_parallelogram
z)"
  unfolding path_image_parallelogram_path' closure_period_parallelogram
by (intro image_mono) auto

lemma path_image_parallelogram_disjoint_interior:
  "path_image (parallelogram_path z ω1 ω2) ∩ interior (period_parallelogram
z) = {}"
  unfolding path_image_parallelogram_path' interior_period_parallelogram
  by (auto simp: of_ω12_coords_eq_iff box_prod)

lemma winding_number_parallelogram_outside:
  assumes "w ∉ closure (period_parallelogram z)"
  shows "winding_number (parallelogram_path z ω1 ω2) w = 0"
  by (rule winding_number_zero_outside[OF _ _ _ assms])
     (use path_image_parallelogram_subset_closure[of z] in auto)
```

The path we take around the period parallelogram is clearly a simple path, and its orientation depends on the angle between our generators.

```
lemma winding_number_parallelogram_inside:
  assumes "w ∈ interior (period_parallelogram z)"
  shows   "winding_number (parallelogram_path z ω1 ω2) w = sgn (Im (ω2
/ ω1))"
proof -
```

```
  let ?P = "parallelogram_path z ω1 ω2"
  have w: "w ∉ path_image ?P"
    using assms unfolding interior_period_parallelogram path_image_parallelogram_path'
    by (auto simp: of_ω12_coords_eq_iff box_prod)
  define ind where "ind = (λa b. winding_number (linepath (z + a) (z
+b)) w)"
  define u where "u = w - z"
  define x y where "x = ω1_coord u" and "y = ω2_coord u"
  have u_eq: "u = of_ω12_coords (x, y)"
    by (simp_all add: x_def y_def flip: ω12_coords_def)
  have xy: "x ∈ {0<..<1}" "y ∈ {0<..<1}" using assms(1)
    unfolding interior_period_parallelogram image_plus_conv_vimage_plus
of_ω12_coords_image_eq
    by (auto simp: x_def y_def ω12_coords_def u_def box_prod)
  have w_eq: "w = z + of_ω12_coords (x, y)"
    using u_eq by (simp add: u_def algebra_simps)

  define W where "W = winding_number (parallelogram_path z ω1 ω2) w"
  have Re_W_eq: "Re W = Re (ind 0 ω1) + Re (ind ω1 (ω1 + ω2)) + Re (ind
(ω1 + ω2) ω2) + Re (ind ω2 0)"
    using w unfolding W_def parallelogram_path_def
    by (simp add: winding_number_join ind_def path_image_join add_ac)

  show ?thesis
  proof (cases "Im (ω2 / ω1)" "0::real" rule: linorder_cases)
    case equal
    hence False
      using fundpair complex_is_Real_iff by (auto simp: fundpair_def)
    thus ?thesis ..

  next
    case greater
    have "W = 1"
      unfolding W_def
    proof (rule simple_closed_path_winding_number_pos; (fold W_def)?)
      from greater have neg: "Im (ω1 * cnj ω2) < 0"
        by (subst (asm) Im_complex_div_gt_0) (auto simp: mult_ac)

      have pos1: "Re (ind 0 ω1) > 0"
      proof -
        have "Im ((z + ω1 - (z + 0)) * cnj (z + ω1 - w)) = y * (-Im (ω1
* cnj ω2))"
          by (simp add: w_eq algebra_simps of_ω12_coords_def)
        also have "... > 0"
          using neg xy by (intro mult_pos_pos) auto
        finally show ?thesis
          unfolding ind_def by (rule winding_number_linepath_pos_lt)
      qed
```

```
      have pos2: "Re (ind ω1 (ω1 + ω2)) > 0"
      proof -
        have "Im ((z + (ω1 + ω2) - (z + ω1)) * cnj (z + (ω1 + ω2) -
w)) =
                (1 - x) * (-Im (ω1 * cnj ω2))"
          by (simp add: w_eq algebra_simps of_ω12_coords_def)
        also have "... > 0"
          using neg xy by (intro mult_pos_pos) auto
        finally show ?thesis
          unfolding ind_def by (rule winding_number_linepath_pos_lt)
      qed

      have pos3: "Re (ind (ω1 + ω2) ω2) > 0"
      proof -
        have "Im ((z + ω2 - (z + (ω1 + ω2))) * cnj (z + ω2 - w)) =
                (1 - y) * (-Im (ω1 * cnj ω2))"
          by (simp add: w_eq algebra_simps of_ω12_coords_def)
        also have "... > 0"
          using neg xy by (intro mult_pos_pos) auto
        finally show ?thesis
          unfolding ind_def by (rule winding_number_linepath_pos_lt)
      qed

      have pos4: "Re (ind ω2 0) > 0"
      proof -
        have "Im ((z + 0 - (z + ω2)) * cnj (z + 0 - w)) =
                x * (-Im (ω1 * cnj ω2))"
          by (simp add: w_eq algebra_simps of_ω12_coords_def)
        also have "... > 0"
          using neg xy by (intro mult_pos_pos) auto
        finally show ?thesis
          unfolding ind_def by (rule winding_number_linepath_pos_lt)
      qed

      show "Re W > 0"
        using pos1 pos2 pos3 pos4 unfolding Re_W_eq by linarith
    qed (use w in <auto intro: simple_path_parallelogram>)

    thus ?thesis
      using greater by (simp add: W_def)

  next
    case less

    have "W = -1"
      unfolding W_def
    proof (rule simple_closed_path_winding_number_neg; (fold W_def)?)
      from less have neg: "Im (ω2 * cnj ω1) < 0"
        by (simp add: Im_complex_div_lt_0)
```

```
      have neg1: "Re (ind 0 ω1) < 0"
      proof -
        have "Im ((z + 0 - (z + ω1)) * cnj (z + 0 - w)) = y * (-Im (ω2
* cnj ω1))"
          by (simp add: w_eq algebra_simps of_ω12_coords_def)
        also have "... > 0"
          using neg xy by (intro mult_pos_pos) auto
        finally show ?thesis
          unfolding ind_def by (rule winding_number_linepath_neg_lt)
      qed

      have neg2: "Re (ind ω1 (ω1 + ω2)) < 0"
      proof -
        have "Im ((z + ω1 - (z + (ω1 + ω2))) * cnj (z + ω1 - w)) =
                (1 - x) * (-Im (ω2 * cnj ω1))"
          by (simp add: w_eq algebra_simps of_ω12_coords_def)
        also have "... > 0"
          using neg xy by (intro mult_pos_pos) auto
        finally show ?thesis
          unfolding ind_def by (rule winding_number_linepath_neg_lt)
      qed

      have neg3: "Re (ind (ω1 + ω2) ω2) < 0"
      proof -
        have "Im ((z + (ω1 + ω2)) - (z + ω2)) * cnj (z + (ω1 + ω2) -
w)) =
                (1 - y) * (-Im (ω2 * cnj ω1))"
          by (simp add: w_eq algebra_simps of_ω12_coords_def)
        also have "... > 0"
          using neg xy by (intro mult_pos_pos) auto
        finally show ?thesis
          unfolding ind_def by (rule winding_number_linepath_neg_lt)
      qed

      have neg4: "Re (ind ω2 0) < 0"
      proof -
        have "Im ((z + ω2 - (z + 0)) * cnj (z + ω2 - w)) =
                x * (-Im (ω2 * cnj ω1))"
          by (simp add: w_eq algebra_simps of_ω12_coords_def)
        also have "... > 0"
          using neg xy by (intro mult_pos_pos) auto
        finally show ?thesis
          unfolding ind_def by (rule winding_number_linepath_neg_lt)
      qed

      show "Re W < 0"
        using neg1 neg2 neg3 neg4 unfolding Re_W_eq by linarith
    qed (use w in <auto intro: simple_path_parallelogram>)
```

```
    thus ?thesis
      using less by (simp add: W_def)
  qed
qed

end
```

3.3 Canonical representatives and the fundamental parallelogram

```
context complex_lattice
begin
```

The following function maps any complex number `z` to its canonical representative `z'` in the fundamental period parallelogram.

```
definition to_fund_parallelogram :: "complex ⇒ complex" where
  "to_fund_parallelogram z =
     (case ω12_coords z of (a, b) ⇒ of_ω12_coords (frac a, frac b))"

lemma to_fund_parallelogram_in_parallelogram [intro]:
  "to_fund_parallelogram z ∈ period_parallelogram 0"
  unfolding to_fund_parallelogram_def
  by (auto simp: period_parallelogram_altdef case_prod_unfold frac_lt_1)

lemma ω1_coord_to_fund_parallelogram [simp]: "ω1_coord (to_fund_parallelogram
z) = frac (ω1_coord z)"
  and ω2_coord_to_fund_parallelogram [simp]: "ω2_coord (to_fund_parallelogram
z) = frac (ω2_coord z)"
  by (auto simp: to_fund_parallelogram_def case_prod_unfold ω12_coords_def)

lemma to_fund_parallelogramE:
  obtains m n where "to_fund_parallelogram z = z + of_int m * ω1 + of_int
n * ω2"
proof -
  define m where "m = floor (fst (ω12_coords z))"
  define n where "n = floor (snd (ω12_coords z))"
  have "z - of_int m * ω1 - of_int n * ω2 =
        of_ω12_coords (ω12_coords z) - of_int m * ω1 - of_int n * ω2"
    by (simp add: m_def n_def)
  also have "... = to_fund_parallelogram z"
    unfolding of_ω12_coords_def
    by (simp add: case_prod_unfold to_fund_parallelogram_def frac_def
                  m_def n_def of_ω12_coords_def algebra_simps)
  finally show ?thesis
    by (intro that[of "-m" "-n"]) auto
qed

lemma rel_to_fund_parallelogram_left: "rel (to_fund_parallelogram z)
```

```
z"
proof -
  obtain m n where "to_fund_parallelogram z = z + of_int m * ω1 + of_int
n * ω2"
    by (elim to_fund_parallelogramE)
  hence "to_fund_parallelogram z - z = of_int m * ω1 + of_int n * ω2"
    by Groebner_Basis.algebra
  also have "... = of_ω12_coords (of_int m, of_int n)"
    by (simp add: of_ω12_coords_def)
  also have "... ∈ Λ"
    by (rule of_ω12_coords_in_lattice) auto
  finally show ?thesis
    by (simp add: rel_def)
qed

lemma rel_to_fund_parallelogram_right: "rel z (to_fund_parallelogram
z)"
  using rel_to_fund_parallelogram_left[of z] by (simp add: rel_sym)

lemma rel_to_fund_parallelogram_left_iff [simp]: "rel (to_fund_parallelogram
z) w ⟷ rel z w"
  using rel_sym rel_to_fund_parallelogram_right rel_trans by blast

lemma rel_to_fund_parallelogram_right_iff [simp]: "rel z (to_fund_parallelogram
w) ⟷ rel z w"
  using rel_sym rel_to_fund_parallelogram_left rel_trans by blast

lemma to_fund_parallelogram_in_lattice_iff [simp]:
  "to_fund_parallelogram z ∈ lattice ⟷ z ∈ lattice"
  using pre_complex_lattice.rel_0_left_iff rel_to_fund_parallelogram_right_iff
by blast

lemma to_fund_parallelogram_in_lattice [lattice_intros]:
  "z ∈ lattice ⟹ to_fund_parallelogram z ∈ lattice"
  by simp
```

`to_fund_parallelogram` is a bijective map from any period parallelogram to the standard period parallelogram:

```
lemma bij_betw_to_fund_parallelogram:
  "bij_betw to_fund_parallelogram (period_parallelogram orig) (period_parallelogram
0)"
proof -
  have "bij_betw (of_ω12_coords ∘ map_prod frac frac ∘ ω12_coords)
         (period_parallelogram orig) (period_parallelogram 0)"
  proof (intro bij_betw_trans)
    show "bij_betw of_ω12_coords ({0..<1}×{0..<1}) (period_parallelogram
0)"
      by (rule bij_betwI[of _ _ _ ω12_coords]) (auto simp: period_parallelogram_altdef)
  next
```

```
    define a b where "a = ω1_coord orig" "b = ω2_coord orig"
    have orig_eq: "orig = of_ω12_coords (a, b)"
      by (auto simp: a_b_def simp flip: ω12_coords_def)

    show "bij_betw ω12_coords (period_parallelogram orig)
            ({ω1_coord orig..<ω1_coord orig+1} × {ω2_coord orig..<ω2_coord
orig+1})"
    proof (rule bij_betwI[of _ _ _ of_ω12_coords])
      show "ω12_coords
              ∈ period_parallelogram orig →
                {ω1_coord orig..<ω1_coord orig + 1} × {ω2_coord orig..<ω2_coord
orig + 1}"
        by (auto simp: orig_eq period_parallelogram_def period_parallelogram_altdef
ω12_coords.add)
    next
      show "of_ω12_coords ∈ {ω1_coord orig..<ω1_coord orig + 1} × {ω2_coord
orig..<ω2_coord orig + 1} →
              period_parallelogram orig"
      proof safe
        fix c d :: real
        assume c: "c ∈ {ω1_coord orig..<ω1_coord orig + 1}"
        assume d: "d ∈ {ω2_coord orig..<ω2_coord orig + 1}"
        have "of_ω12_coords (c, d) = of_ω12_coords (a, b) + of_ω12_coords
(c - a, d - b)"
          by (simp add: of_ω12_coords_def algebra_simps)
        moreover have "(c - a, d - b) ∈ {0..<1} × {0..<1}"
          using c d unfolding a_b_def [symmetric] by auto
        ultimately show "of_ω12_coords (c, d) ∈ period_parallelogram
orig"
          unfolding period_parallelogram_def period_parallelogram_altdef
orig_eq image_image
          by auto
      qed
    qed auto
  next
    show "bij_betw (map_prod frac frac)
            ({ω1_coord orig..<ω1_coord orig + 1} × {ω2_coord orig..<ω2_coord
orig + 1})
            ({0..<1} × {0..<1})"
      by (intro bij_betw_map_prod bij_betw_frac)
  qed
  also have "of_ω12_coords ∘ map_prod frac frac ∘ ω12_coords =
              to_fund_parallelogram"
    by (auto simp: o_def to_fund_parallelogram_def fun_eq_iff case_prod_unfold
map_prod_def)
  finally show ?thesis .
qed
```

There exists a bijection between any two period parallelograms that always

maps points to equivalent points.

```
lemma bij_betw_period_parallelograms:
  obtains f where
    "bij_betw f (period_parallelogram orig) (period_parallelogram orig')"
    "⋀z. rel (f z) z"
proof -
  define h where "h = inv_into (period_parallelogram orig') to_fund_parallelogram"
  show ?thesis
  proof (rule that[of "h ∘ to_fund_parallelogram"])
    show "bij_betw (h ∘ to_fund_parallelogram)
            (period_parallelogram orig) (period_parallelogram orig')"
      unfolding h_def
      using bij_betw_to_fund_parallelogram bij_betw_inv_into[OF bij_betw_to_fund_parallelogr
      by (rule bij_betw_trans)
  next
    fix z :: complex
    have "rel (to_fund_parallelogram (h (to_fund_parallelogram z))) (h
(to_fund_parallelogram z))"
      by auto
    also have "to_fund_parallelogram (h (to_fund_parallelogram z)) = to_fund_parallelogram
z"
      unfolding h_def using bij_betw_to_fund_parallelogram[of orig']
      by (subst f_inv_into_f[of _ to_fund_parallelogram])
         (simp_all add: bij_betw_def to_fund_parallelogram_in_parallelogram)
    finally have *: "rel (to_fund_parallelogram z) (h (to_fund_parallelogram
z))" .
    have "rel ((to_fund_parallelogram z - z)) (to_fund_parallelogram z
- h (to_fund_parallelogram z))"
      using * diff_in_lattice rel_def rel_to_fund_parallelogram_left by
blast
    thus "rel ((h ∘ to_fund_parallelogram) z) z"
      using * pre_complex_lattice.rel_sym by force
  qed
qed

lemma to_fund_parallelogram_0 [simp]: "to_fund_parallelogram 0 = 0"
  by (simp add: to_fund_parallelogram_def zero_prod_def)

lemma to_fund_parallelogram_lattice [simp]: "z ∈ Λ ⟹ to_fund_parallelogram
z = 0"
  by (auto simp: to_fund_parallelogram_def in_lattice_conv_ω12_coords)

lemma to_fund_parallelogram_eq_iff [simp]:
  "to_fund_parallelogram u = to_fund_parallelogram v ⟷ rel u v"
proof
  assume "rel u v"
  then obtain z where z: "z ∈ Λ" "v = u + z"
    by (elim relE)
  from this(1) obtain m n where mn: "z = of_ω12_coords (of_int m, of_int
```

```
n)"
    by (elim latticeE)
  show "to_fund_parallelogram u = to_fund_parallelogram v" unfolding
z(2)
    by (simp add: to_fund_parallelogram_def ω12_coords.add in_lattice_conv_ω12_coords
mn case_prod_unfold)
next
  assume "to_fund_parallelogram u = to_fund_parallelogram v"
  thus "rel u v"
    by (metis rel_to_fund_parallelogram_right rel_to_fund_parallelogram_right_iff)
qed

lemma to_fund_parallelogram_eq_0_iff [simp]: "to_fund_parallelogram u
= 0 ⟷ u ∈ Λ"
  using to_fund_parallelogram_eq_iff[of u 0]
  by (simp del: to_fund_parallelogram_eq_iff add: rel_0_right_iff)

lemma to_fund_parallelogram_of_fund_parallelogram:
  "z ∈ period_parallelogram 0 ⟹ to_fund_parallelogram z = z"
  unfolding to_fund_parallelogram_def period_parallelogram_def
  by (auto simp: of_ω12_coords_eq_iff frac_eq_id)

lemma to_fund_parallelogram_idemp [simp]:
  "to_fund_parallelogram (to_fund_parallelogram z) = to_fund_parallelogram
z"
  by (rule to_fund_parallelogram_of_fund_parallelogram) auto

lemma to_fund_parallelogram_unique:
  assumes "rel z z'" "z' ∈ period_parallelogram 0"
  shows   "to_fund_parallelogram z = z'"
  using assms by (metis to_fund_parallelogram_eq_iff to_fund_parallelogram_of_fund_parallel

lemma to_fund_parallelogram_unique':
  assumes "rel z z'" "z ∈ period_parallelogram 0" "z' ∈ period_parallelogram
0"
  shows   "z = z'"
  using assms
  by (metis to_fund_parallelogram_eq_iff to_fund_parallelogram_of_fund_parallelogram)
```

The following is the "left half" of the fundamental parallelogram. The bottom border is contained, the top border is not. Of the frontier of this parallelogram only the upper half is

```
definition (in pre_complex_lattice) half_fund_parallelogram where
  "half_fund_parallelogram =
     of_ω12_coords ` {(x,y). x ∈ {0..1/2} ∧ y ∈ {0..<1} ∧ (x ∈ {0, 1/2}
⟶ y ≤ 1/2)}"

lemma half_fund_parallelogram_altdef:
  "half_fund_parallelogram = ω12_coords -` {(x,y). x ∈ {0..1/2} ∧ y ∈
```

```
{0..<1} ∧ (x ∈ {0, 1/2} ⟶ y ≤ 1/2)}"
  unfolding half_fund_parallelogram_def by (meson of_ω12_coords_image_eq)

lemma zero_in_half_fund_parallelogram [simp, intro]: "0 ∈ half_fund_parallelogram"
  by (auto simp: half_fund_parallelogram_altdef zero_prod_def)

lemma half_fund_parallelogram_in_lattice_iff:
  assumes "z ∈ half_fund_parallelogram"
  shows   "z ∈ Λ ⟷ z = 0"
proof
  assume "z ∈ Λ"
  then obtain m n where z_eq: "z = of_ω12_coords (of_int m, of_int n)"
    by (elim latticeE)
  thus "z = 0"
    using assms unfolding z_eq half_fund_parallelogram_altdef by auto
qed auto

definition to_half_fund_parallelogram :: "complex ⇒ complex" where
  "to_half_fund_parallelogram z =
     (let (x,y) = map_prod frac frac (ω12_coords z);
          (x',y') = (if x > 1/2 ∨ (x ∈ {0, 1/2} ∧ y > 1 / 2) then (if
x = 0 then 0 else 1 - x, if y = 0 then 0 else 1 - y) else (x, y))
      in  of_ω12_coords (x',y'))"

lemma in_Ints_conv_floor: "x ∈ ℤ ⟷ x = of_int (floor x)"
  by (metis Ints_of_int of_int_floor)

lemma (in complex_lattice) rel_to_half_fund_parallelogram:
  "rel z (to_half_fund_parallelogram z) ∨ rel z (-to_half_fund_parallelogram
z)"
  unfolding rel_def in_lattice_conv_ω12_coords to_half_fund_parallelogram_def
Let_def
            ω1_coord.diff ω2_coord.diff ω1_coord.add ω2_coord.add ω1_coord.neg
ω2_coord.neg
            case_prod_unfold Let_def ω12_coords_def frac_def map_prod_def
  by (simp flip: in_Ints_conv_floor)

lemma (in complex_lattice) to_half_fund_parallelogram_in_half_fund_parallelogram
[intro]:
  "to_half_fund_parallelogram z ∈ half_fund_parallelogram"
  unfolding half_fund_parallelogram_altdef to_half_fund_parallelogram_def
to_half_fund_parallelogram_def Let_def
            ω1_coord.diff ω2_coord.diff ω1_coord.add ω2_coord.add ω1_coord.neg
ω2_coord.neg
            case_prod_unfold Let_def ω12_coords_def frac_def map_prod_def
  apply simp
  apply linarith
  done
```

```
lemma (in complex_lattice) half_fund_parallelogram_subset_period_parallelogram:
  "half_fund_parallelogram ⊆ period_parallelogram 0"
proof -
  have "of_ω12_coords ` {(x, y). x ∈ {0..1 / 2} ∧ y ∈ {0..<1} ∧ (x ∈
{0,1/2} ⟶ y ≤ 1 / 2)} ⊆
        of_ω12_coords ` ({0..<1} × {0..<1})"
    by (intro image_mono) (auto simp: not_less)
  also have "... = (+) 0 ` (of_ω12_coords ` ({0..<1} × {0..<1}))"
    by simp
  finally show ?thesis
    unfolding period_parallelogram_def half_fund_parallelogram_def .
qed

lemma to_half_fund_parallelogram_in_lattice_iff [simp]: "to_half_fund_parallelogram
z ∈ Λ ⟷ z ∈ Λ"
  by (metis rel_lattice_trans_left rel_sym rel_to_half_fund_parallelogram
uminus_in_lattice_iff)

lemma rel_in_half_fund_parallelogram_imp_eq:
  assumes "rel z w ∨ rel z (-w)" "z ∈ half_fund_parallelogram" "w ∈
half_fund_parallelogram"
  shows   "z = w"
  using assms(1)
proof
  assume "rel z w"
  moreover from assms have "z ∈ period_parallelogram 0" "w ∈ period_parallelogram
0"
    using half_fund_parallelogram_subset_period_parallelogram by blast+
  ultimately show "z = w"
    by (metis to_fund_parallelogram_eq_iff
        to_fund_parallelogram_of_fund_parallelogram)
next
  assume "rel z (-w)"
  hence "rel (-w) z"
    by (rule rel_symI)
  then obtain m n where z_eq: "z = -w + of_ω12_coords (of_int m, of_int
n)"
    by (elim relE latticeE) auto
  define x y where "x = ω1_coord w" "y = ω2_coord w"
  have w_eq: "w = of_ω12_coords (x, y)"
    unfolding x_y_def ω12_coords_def [symmetric] by simp
  have 1: "x ≥ 0" "y ≥ 0" "x ≤ 1/2" "y < 1" "x = 0 ∨ x = 1/2 ⟹ y
≤ 1/2"
    using assms(3) unfolding half_fund_parallelogram_altdef w_eq by auto
  have 2: "of_int m - x ≥ 0" "of_int n - y ≥ 0" "of_int m - x ≤ 1/2"
"of_int n - y < 1"
        "of_int m - x = 0 ∨ of_int m - x = 1/2 ⟹ of_int n - y ≤ 1/2"
    using assms(2) unfolding half_fund_parallelogram_altdef z_eq w_eq
```

```
    by (auto simp: ω12_coords.diff)

  have "m ∈ {0,1}" "n ∈ {0,1}"
    using 1 2 by auto
  hence "real_of_int m = 2 * x ∧ real_of_int n = 2 * y"
    using 1 2 by auto
  hence "ω12_coords z = ω12_coords w"
    unfolding z_eq w_eq ω12_coords.add ω12_coords.diff ω12_coords.neg
ω12_coords_of_ω12_coords
    by simp
  thus ?thesis
    by (metis of_ω12_coords_ω12_coords)
qed

lemma to_half_fund_parallelogram_of_half_fund_parallelogram:
  assumes "z ∈ half_fund_parallelogram"
  shows   "to_half_fund_parallelogram z = z"
  by (metis assms rel_to_half_fund_parallelogram to_half_fund_parallelogram_in_half_fund_pa
rel_in_half_fund_parallelogram_imp_eq)

lemma to_half_fund_parallelogram_idemp [simp]:
  "to_half_fund_parallelogram (to_half_fund_parallelogram z) = to_half_fund_parallelogram
z"
  by (rule to_half_fund_parallelogram_of_half_fund_parallelogram) auto

lemma to_half_fund_parallelogram_unique:
  assumes "rel z z' ∨ rel z (-z')" "z' ∈ half_fund_parallelogram"
  shows   "to_half_fund_parallelogram z = z'"
proof (rule rel_in_half_fund_parallelogram_imp_eq)
  show "rel (to_half_fund_parallelogram z) z' ∨ rel (to_half_fund_parallelogram
z) (- z')"
    using rel_to_half_fund_parallelogram[of z] assms rel_sym rel_trans
rel_minus minus_minus
    by metis
qed (use to_half_fund_parallelogram_in_half_fund_parallelogram assms in
auto)

lemma to_half_fund_parallelogram_eq_iff:
  "to_half_fund_parallelogram z = to_half_fund_parallelogram w ⟷ rel
z w ∨ rel z (-w)"
proof
  assume eq: "to_half_fund_parallelogram z = to_half_fund_parallelogram
w"
  define u where "u = to_half_fund_parallelogram w"
  have "rel z u ∨ rel z (-u)" "rel w u ∨ rel w (-u)"
    using rel_to_half_fund_parallelogram[of z] rel_to_half_fund_parallelogram[of
w] eq unfolding u_def by auto
  hence "rel z w ∨ (rel z u ∧ rel w (-u) ∨ rel z (-u) ∧ rel w u)"
    using rel_trans rel_sym by blast
```

```
  moreover have "rel z (-w)" if "rel z u ∧ rel w (-u) ∨ rel z (-u) ∧
rel w u"
    using that by (metis minus_minus pre_complex_lattice.rel_minus rel_sym
rel_trans)
  ultimately show "rel z w ∨ rel z (-w)" by blast
next
  assume *: "rel z w ∨ rel z (-w)"
  define u where "u = to_half_fund_parallelogram w"
  show "to_half_fund_parallelogram z = u"
  proof (rule to_half_fund_parallelogram_unique)
    have "rel w u ∨ rel w (-u)"
      unfolding u_def using rel_to_half_fund_parallelogram[of w] by blast
    with * have "rel z u ∨ (rel (-w) (-u) ∧ rel z (-w) ∨ rel w (-u)
∧ rel z w)"
      using rel_trans rel_sym rel_minus[of w "-u"] rel_minus[of z "-w"]
rel_minus[of w u]
      unfolding minus_minus by blast
    thus "rel z u ∨ rel z (-u)"
      using rel_trans rel_sym by blast
  qed (auto simp: u_def)
qed

lemma in_half_fund_parallelogram_imp_half_lattice:
  assumes "z ∈ half_fund_parallelogram" "to_fund_parallelogram (-z) ∈
half_fund_parallelogram"
  shows   "2 * z ∈ Λ"
  using assms
  by (metis rel_in_half_fund_parallelogram_imp_eq diff_minus_eq_add mult_2
           pre_complex_lattice.rel_def rel_to_fund_parallelogram_left)

end
```

3.4 Equivalence of fundamental pairs

Two fundamental pairs are called *equivalent* if they generate the same complex lattice.

```
definition equiv_fundpair :: "complex × complex ⇒ complex × complex ⇒
bool" where
  "equiv_fundpair = (λ(ω1, ω2) (ω1', ω2').
     pre_complex_lattice.lattice ω1 ω2 = pre_complex_lattice.lattice
ω1' ω2')"

lemma equiv_fundpair_iff_aux:
  fixes p :: int
  assumes "p * c + q * a = 0" "p * d + q * b = 1"
          "r * c + s * a = 1" "r * d + s * b = 0"
  shows "|a * d - b * c| = 1"
proof -
  have "r * b * c + s * a * b = b"
```

```
    by (metis assms(3) distrib_left mult.left_commute mult.right_neutral)
  moreover have "r * a * d + s * a * b = 0"
    by (metis assms(4) distrib_left mult.commute mult.left_commute mult_zero_right)
  ultimately have "r dvd b"
    by (metis mult.assoc dvd_0_right dvd_add_right_iff dvd_triv_left)
  have "p * r * d + p * s * b = 0"
    by (metis assms(4) distrib_left mult.commute mult.left_commute mult_zero_right)
  moreover have "p * r * d + q * r * b = r"
    by (metis assms(2) int_distrib(2) mult.assoc mult.left_commute mult.right_neutral)
  ultimately have "b dvd r"
    by (metis dvd_0_right mult.commute zdvd_reduce)
  have "r * c * d + s * b * c = 0"
    by (metis assms(4) distrib_left mult.commute mult.left_commute mult_zero_right)
  moreover have "r * c * d + s * a * d = d"
    by (metis assms(3) distrib_left mult.commute mult.right_neutral)
  ultimately have "s dvd d"
    by (metis dvd_0_right dvd_add_times_triv_right_iff mult.assoc mult.commute)
  have "q * r * d + q * s * b = 0"
    by (metis mult.assoc assms(4) int_distrib(2) mult_not_zero)
  moreover have "p * s * d + q * s * b = s"
    by (metis assms(2) int_distrib(2) mult.assoc mult.left_commute mult.right_neutral)
  ultimately have "d dvd s"
    by (metis add.commute dvd_0_right mult.commute zdvd_reduce)
  have "|b| = |r|" "|d| = |s|"
    by (meson ‹b dvd r› ‹r dvd b› ‹d dvd s› ‹s dvd d› zdvd_antisym_abs)+
  then show ?thesis
    by (smt (verit, best) assms(3,4) mult.commute mult_cancel_left mult_eq_0_iff
mult_minus_left)
qed
```

The following fact is Theorem 1.2 in Apostol's book: two fundamental pairs are equivalent iff there exists a unimodular transformation that maps one to the other.

```
theorem equiv_fundpair_iff:
  fixes ω1 ω2 ω1' ω2' :: complex
  assumes "fundpair (ω1, ω2)" "fundpair (ω1', ω2')"
  shows "equiv_fundpair (ω1, ω2) (ω1', ω2') ⟷
         (∃a b c d. |a*d - b*c| = 1 ∧
                    ω2' = of_int a * ω2 + of_int b * ω1 ∧ ω1' = of_int
c * ω2 + of_int d * ω1)"
    (is "?lhs = ?rhs")
proof -
  interpret gl: complex_lattice ω1 ω2
    by standard fact
  interpret gl': complex_lattice ω1' ω2'
    by standard fact
  show ?thesis
  proof
    assume L: ?lhs
```

```
    hence lattices_eq: "gl.lattice = gl'.lattice"
      by (simp add: equiv_fundpair_def)

    have "ω1' ∈ gl'.lattice" "ω2' ∈ gl'.lattice"
      by auto
    hence "ω1' ∈ gl.lattice" "ω2' ∈ gl.lattice"
      by (simp_all add: lattices_eq)
    then obtain a b c d
      where ab: "ω2' = of_int b * ω1 + of_int a * ω2"
        and cd: "ω1' = of_int d * ω1 + of_int c * ω2"
      by (elim gl.latticeE) (auto simp: gl.of_ω12_coords_def scaleR_conv_of_real)

    have "ω1 ∈ gl.lattice" "ω2 ∈ gl.lattice"
      by auto
    hence "ω1 ∈ gl'.lattice" "ω2 ∈ gl'.lattice"
      by (simp_all add: lattices_eq)
    then obtain p q r s
      where pq: "ω1 = of_int p * ω1' + of_int q * ω2'" and
            rs: "ω2 = of_int r * ω1' + of_int s * ω2'"
      by (elim gl'.latticeE) (auto simp: gl'.of_ω12_coords_def scaleR_conv_of_real)

    have "ω1 = p * (c * ω2 + d * ω1) + q * (a * ω2 + b * ω1)"
      using pq cd ab add.commute by metis
    also have "... = (p * c + q * a) * ω2 + (p * d + q * b) * ω1"
      by (simp add: algebra_simps)
    finally have "gl.of_ω12_coords (1, 0) = gl.of_ω12_coords (p*d+q*b,
p*c+q*a)"
      by (simp_all add: gl.of_ω12_coords_def add_ac)
    hence pc: "(p * c + q * a) = 0 ∧ p * d + q * b = 1"
      unfolding gl.of_ω12_coords_eq_iff prod.case prod.inject by linarith

    have "ω2 = r * (c * ω2 + d * ω1) + s * (a * ω2 + b * ω1)"
      using cd rs ab add.commute by metis
    also have "... = (r * c + s * a) * ω2 + (r * d + s * b) * ω1"
      by (simp add: algebra_simps)
    finally have "gl.of_ω12_coords (0, 1) = gl.of_ω12_coords (r*d+s*b,
r*c+s*a)"
      by (simp_all add: gl.of_ω12_coords_def add_ac)
    hence rc: "r * c + s * a = 1 ∧ r * d + s * b = 0"
      unfolding gl.of_ω12_coords_eq_iff prod.case prod.inject by linarith
    with pc have "|a*d - b*c| = 1"
      by (meson equiv_fundpair_iff_aux)

    hence "|a*d - b*c| = 1 ∧
             ω2' = of_int a * ω2 + of_int b * ω1 ∧ ω1' = of_int c * ω2
+ of_int d * ω1"
      using cd ab by auto
    thus ?rhs
      by blast
```

```
  next
    assume ?rhs
    then obtain a b c d :: int where 1: "|a * d - b * c| = 1"
      and eq: "ω2' = of_int a * ω2 + of_int b * ω1" "ω1' = of_int c
* ω2 + of_int d * ω1"
      by blast
    define det where "det = a * d - b * c"
    define a' b' c' d' where "a' = det * d" "b' = -det * b" "c' = -det
* c" "d' = det * a"
    have "|det| = 1"
      using 1 by (simp add: det_def)
    hence det_square: "det ^ 2 = 1"
      using abs_square_eq_1 by blast

    have eq': "ω2 = of_int a' * ω2' + of_int b' * ω1'" "ω1 = of_int
c' * ω2' + of_int d' * ω1'"
    proof -
      have "of_int a' * ω2' + of_int b' * ω1' = det^2 * ω2"
        by (simp add: eq algebra_simps det_def a'_b'_c'_d'_def power2_eq_square)
      thus "ω2 = of_int a' * ω2' + of_int b' * ω1'"
        by (simp add: det_square)
    next
      have "of_int c' * ω2' + of_int d' * ω1' = det^2 * ω1"
        by (simp add: eq algebra_simps det_def a'_b'_c'_d'_def power2_eq_square)
      thus "ω1 = of_int c' * ω2' + of_int d' * ω1'"
        by (simp add: det_square)
    qed

    have "gl'.lattice ⊆ gl.lattice"
      by (safe elim!: gl'.latticeE, unfold gl'.of_ω12_coords_def)
         (auto simp: eq ring_distribs intro!: gl'.lattice_intros)
    moreover have "gl.lattice ⊆ gl'.lattice"
      by (safe elim!: gl.latticeE, unfold gl.of_ω12_coords_def)
         (auto simp: eq' ring_distribs intro!: gl.lattice_intros)
    ultimately show ?lhs
      unfolding equiv_fundpair_def by auto
  qed
qed
```

We will now look at the triangle spanned by the origin and the generators. We will prove that the only points that lie in or on this triangle are its three vertices.

Moreover, we shall prove that for any lattice Λ, if we have two points ω_1', ω_2' $\in \Lambda$ then these two points generate Λ if and only if the triangle spanned by 0, ω_1', and ω_2' contains no other lattice points except 0, ω_1', and ω_2'.

```
context complex_lattice
begin
```

```
lemma in_triangle_iff:
  fixes x
  defines "a ≡ ω1_coord x" and "b ≡ ω2_coord x"
  shows   "x ∈ convex hull {0, ω1, ω2} ⟷ a ≥ 0 ∧ b ≥ 0 ∧ a + b ≤
1"
proof
  assume "x ∈ convex hull {0, ω1, ω2}"
  then obtain t u where tu: "u ≥ 0" "t ≥ 0" "u + t ≤ 1" "x = of_ω12_coords
(u, t)"
    unfolding convex_hull_3_alt by (auto simp: of_ω12_coords_def scaleR_conv_of_real)
  have "a = u" "b = t"
    by (auto simp: a_def b_def tu(4))
  with tu show "a ≥ 0 ∧ b ≥ 0 ∧ a + b ≤ 1"
    by auto
next
  assume ab: "a ≥ 0 ∧ b ≥ 0 ∧ a + b ≤ 1"
  have "x = of_ω12_coords (a, b)"
    by (auto simp: a_def b_def simp flip: ω12_coords_def)
  hence "x = 0 + a *R (ω1 - 0) + b *R (ω2 - 0)" "0 ≤ a ∧ 0 ≤ b ∧ a
+ b ≤ 1"
    using ab by (auto simp: a_def b_def of_ω12_coords_def scaleR_conv_of_real)
  thus "x ∈ convex hull {0, ω1, ω2}"
    unfolding convex_hull_3_alt by blast
qed
```

The only lattice points inside the fundamental triangle are the generators and the origin.

```
lemma lattice_Int_triangle: "convex hull {0, ω1, ω2} ∩ Λ = {0, ω1,
ω2}"
proof (intro equalityI subsetI)
  fix x assume x: "x ∈ convex hull {0, ω1, ω2} ∩ Λ"
  then obtain a b :: real where ab: "a ≥ 0" "b ≥ 0" "a + b ≤ 1" "x
= of_ω12_coords (a, b)"
    unfolding convex_hull_3_alt by (auto simp: of_ω12_coords_def scaleR_conv_of_real)
  from ab(4) and x have "a ∈ ℤ" "b ∈ ℤ"
    by (auto simp: of_ω12_coords_in_lattice_iff)
  with ab(1-3) have "(a, b) ∈ {(0, 0), (0, 1), (1, 0)}"
    by (auto elim!: Ints_cases)
  with ab show "x ∈ {0, ω1, ω2}"
    by auto
qed (auto intro: hull_inc)
```

The following fact is Theorem 1.1 in Apostol's book: given a fixed lattice Λ, a pair of non-collinear period vectors ω_1, ω_2 is fundamental (i.e. generates Λ) iff the triangle spanned by 0, ω_1, ω_2 contains no lattice points other than its three vertices.

```
lemma equiv_fundpair_iff_triangle:
  assumes "fundpair (ω1', ω2')" "ω1' ∈ Λ" "ω2' ∈ Λ"
```

```
  shows   "equiv_fundpair (ω1, ω2) (ω1', ω2') ⟷ convex hull {0, ω1',
ω2'} ∩ Λ = {0, ω1', ω2'}"
proof -
  interpret lattice': complex_lattice ω1' ω2'
    by standard fact
  show ?thesis
  proof
    assume "equiv_fundpair (ω1, ω2) (ω1', ω2')"
    thus "convex hull {0, ω1', ω2'} ∩ Λ = {0, ω1', ω2'}"
      using lattice'.lattice_Int_triangle by (simp add: equiv_fundpair_def)
  next
    assume triangle: "convex hull {0, ω1', ω2'} ∩ Λ = {0, ω1', ω2'}"

    show "equiv_fundpair (ω1, ω2) (ω1', ω2')"
      unfolding equiv_fundpair_def prod.case
    proof
      show "lattice'.lattice ⊆ Λ"
        by (intro subsetI, elim lattice'.latticeE)
           (auto simp: lattice'.of_ω12_coords_def intro!: lattice_intros
assms)
    next
      show "Λ ⊆ lattice'.lattice"
      proof
        fix x assume x: "x ∈ Λ"
        define y where "y = lattice'.to_fund_parallelogram x"
        have y: "y ∈ Λ"
        proof -
          obtain a b where y_eq: "y = x + of_int a * ω1' + of_int b *
ω2'"
            using lattice'.to_fund_parallelogramE[of x] unfolding y_def
by blast
          show ?thesis by (auto simp: y_eq intro!: lattice_intros assms
x)
        qed

        have "y ∈ lattice'.lattice"
        proof (cases "y ∈ convex hull {0, ω1', ω2'}")
          case True
          hence "y ∈ convex hull {0, ω1', ω2'} ∩ Λ"
            using y by auto
          also note triangle
          finally show ?thesis
            by auto
        next
          case False
          define y' where "y' = ω1' + ω2' - y"
          have y_conv_y': "y = ω1' + ω2' - y'"
            by (simp add: y'_def)
          define a b where "a = lattice'.ω1_coord x" and "b = lattice'.ω2_coord
```

```
x"
          have [simp]: "lattice'.ω1_coord y = frac a" "lattice'.ω2_coord
y = frac b"
            by (simp_all add: y_def a_def b_def)
          from False have "frac a + frac b > 1"
            by (auto simp: lattice'.in_triangle_iff y'_def)
          hence "y' ∈ convex hull {0, ω1', ω2'}"
            by (auto simp: lattice'.in_triangle_iff y'_def less_imp_le[OF
frac_lt_1]
                           lattice'.ω1_coord.add lattice'.ω2_coord.add
                           lattice'.ω1_coord.diff lattice'.ω2_coord.diff)
          hence "y' ∈ convex hull {0, ω1', ω2'} ∩ Λ"
            using y assms by (auto simp: y'_def intro!: lattice_intros)
          also note triangle
          finally have "y' ∈ {0, ω1', ω2'}" .
          thus ?thesis
            by (auto simp: y_conv_y' intro!: lattice'.lattice_intros)
        qed
        thus "x ∈ lattice'.lattice"
          by (simp add: y_def)
      qed
    qed
  qed
qed

end
```

3.5 Additional useful facts

```
context complex_lattice
begin
```

The following partitions the lattice into countably many "layers", starting from the origin, which is the 0-th layer. The k-th layer consists of precisely those points in the lattice whose lattice coordinates (m, n) satisfy $\max(|m|, |n|) = k$.

```
definition lattice_layer :: "nat ⇒ complex set" where
  "lattice_layer k =
     of_ω12_coords ` map_prod of_int of_int `
       ({int k, -int k} × {-int k..int k} ∪ {-int k..int k} × {-int k,
int k})"

lemma in_lattice_layer_iff:
  "z ∈ lattice_layer k ⟷
     ω12_coords z ∈ ℤ × ℤ ∩ ({int k, -int k} × {-int k..int k} ∪ {-int
k..int k} × {-int k, int k})"
  (is "?lhs = ?rhs")
proof
  assume ?lhs
```

```
  thus ?rhs
    unfolding lattice_layer_def of_ω12_coords_image_eq by (auto simp:
case_prod_unfold)
next
  assume ?rhs
  thus ?lhs unfolding lattice_layer_def image_Un map_prod_image of_ω12_coords_image_eq
    by (auto elim!: Ints_cases)
qed

lemma of_ω12_coords_of_int_in_lattice_layer:
  "of_ω12_coords (of_int a, of_int b) ∈ lattice_layer (nat (max |a| |b|))"
  unfolding in_lattice_layer_iff by (auto simp flip: of_int_minus simp:
max_def)

lemma lattice_layer_covers: "Λ = (⋃k. lattice_layer k)"
proof -
  have "(⋃k. lattice_layer k) = of_ω12_coords ‘ map_prod real_of_int
real_of_int ‘
          (⋃k. ({int k, - int k} × {- int k..int k} ∪ {- int k..int
k} × {- int k, int k}))"
    (is "_ = _ ‘ _ ‘ (⋃k. ?A k)") unfolding lattice_layer_def by blast
  also have "(⋃k. ?A k) = UNIV"
  proof safe
    fix a b :: int
    have "(a, b) ∈ ?A (nat (max |a| |b|))"
      unfolding max_def by simp linarith
    thus "(a, b) ∈ (⋃k. ?A k)"
      by blast
  qed blast+
  also have "range (map_prod real_of_int real_of_int) = ℤ × ℤ"
    by (auto elim!: Ints_cases)
  finally show ?thesis
    by (simp add: lattice_def)
qed

lemma finite_lattice_layer: "finite (lattice_layer k)"
  unfolding lattice_layer_def by auto

lemma lattice_layer_0: "lattice_layer 0 = {0}"
  by (auto simp: lattice_layer_def)

lemma zero_in_lattice_layer_iff [simp]: "0 ∈ lattice_layer k ⟷ k =
0"
  by (auto simp: in_lattice_layer_iff zero_prod_def)

lemma lattice_layer_disjoint:
  assumes "m ≠ n"
  shows   "lattice_layer m ∩ lattice_layer n = {}"
  using assms by (auto simp: lattice_layer_def of_ω12_coords_eq_iff)
```

```
lemma lattice0_conv_layers: "Λ* = (⋃i∈{0<..}. lattice_layer i)" (is
"?lhs = ?rhs")
proof -
  have "Λ* = (⋃i∈UNIV. lattice_layer i) - lattice_layer 0"
    by (simp add: lattice0_def lattice_layer_covers lattice_layer_0)
  also have "... = (⋃i∈UNIV-{0}. lattice_layer i)"
    using lattice_layer_disjoint by blast
  also have "UNIV-{0::nat} = {0<..}"
    by auto
  finally show ?thesis .
qed

lemma card_lattice_layer:
  assumes "k > 0"
  shows "card (lattice_layer k) = 8 * k"
proof -
  define f where "f = of_ω12_coords ∘ map_prod real_of_int real_of_int"
  have "lattice_layer k = f ‘ ({int k, - int k} × {- int k..int k} ∪
{- int k..int k} × {- int k, int k})"
    (is "_ = _ ‘ ?A") unfolding lattice_layer_def f_def image_image o_def
..
  also have "card ... = card ?A"
    by (intro card_image)
       (auto simp: inj_on_def of_ω12_coords_eq_iff f_def map_prod_def
case_prod_unfold)
  also have "?A = {int k, -int k} × {-int k..int k} ∪ {-int k+1..int
k-1} × {-int k, int k}"
    by auto
  also have "card ... = 8 * k" using ‹k > 0›
    by (subst card_Un_disjoint)
       (auto simp: nat_diff_distrib nat_add_distrib nat_mult_distrib Suc_diff_Suc)
  finally show ?thesis .
qed

lemma lattice_layer_nonempty: "lattice_layer k ≠ {}"
  by (auto simp: lattice_layer_def)

definition lattice_layer_path :: "complex set" where
  "lattice_layer_path = of_ω12_coords ‘ ({1, -1} × {-1..1} ∪ {-1..1}
× {-1, 1})"

lemma in_lattice_layer_path_iff:
  "z ∈ lattice_layer_path ⟷ ω12_coords z ∈ ({1, -1} × {-1..1} ∪ {-1..1}
× {-1, 1})"
  unfolding lattice_layer_path_def of_ω12_coords_image_eq by blast

lemma lattice_layer_path_nonempty: "lattice_layer_path ≠ {}"
```

```
proof -
  have "ω1 ∈ lattice_layer_path"
    by (auto simp: in_lattice_layer_path_iff)
  thus ?thesis by blast
qed

lemma compact_lattice_layer_path [intro]: "compact lattice_layer_path"
  unfolding lattice_layer_path_def of_ω12_coords_def case_prod_unfold
  by (intro compact_continuous_image continuous_intros compact_Un compact_Times)
auto

lemma lattice_layer_subset: "lattice_layer k ⊆ (*) (of_nat k) ` lattice_layer_path"
proof
  fix x
  assume "x ∈ lattice_layer k"
  then obtain m n where x: "x = of_ω12_coords (of_int m, of_int n)"
    "(m, n) ∈ ({int k, -int k} × {-int k..int k} ∪ {-int k..int k} ×
{-int k, int k})"
    unfolding lattice_layer_def by blast

  show "x ∈ (*) (of_nat k) ` lattice_layer_path"
  proof (cases "k > 0")
    case True
    have "x = of_nat k * of_ω12_coords (of_int m / of_int k, of_int n
/ of_int k)"
         "(of_int m / of_int k, of_int n / of_int k) ∈ {1::real, -1} ×
{-1..1} ∪ {-1..1} × {-1::real, 1}"
      using x True by (auto simp: divide_simps of_ω12_coords_def)
    thus ?thesis
      unfolding lattice_layer_path_def by blast
  qed (use x in ‹auto simp: lattice_layer_path_def image_iff
                      intro!: exI[of _ "ω1"] bexI[of _ "(1, 0)"]›)
qed
```

The shortest and longest distance of any point on the first layer from the origin, respectively.

```
definition Inf_para :: real where — r in the proof of Lemma 1
  "Inf_para ≡ Inf (norm ` lattice_layer_path)"

lemma Inf_para_pos: "Inf_para > 0"
proof -
  have "compact (norm ` lattice_layer_path)"
    by (intro compact_continuous_image continuous_intros) auto
  hence "Inf_para ∈ norm ` lattice_layer_path"
    unfolding Inf_para_def
    by (intro closed_contains_Inf)
       (use lattice_layer_path_nonempty in ‹auto simp: compact_imp_closed
bdd_below_norm_image›)
  moreover have "∀x∈norm ` lattice_layer_path. x > 0"
```

```
    by (auto simp: in_lattice_layer_path_iff zero_prod_def)
  ultimately show ?thesis
    by blast
qed

lemma Inf_para_nonzero [simp]: "Inf_para ≠ 0"
  using Inf_para_pos by linarith

lemma Inf_para_le:
  assumes "z ∈ lattice_layer_path"
  shows   "Inf_para ≤ norm z"
  unfolding Inf_para_def by (rule cInf_lower) (use assms bdd_below_norm_image
in auto)

lemma lattice_layer_le_norm:
  assumes "ω ∈ lattice_layer k"
  shows "k * Inf_para ≤ norm ω"
proof -
  obtain z where z: "z ∈ lattice_layer_path" "ω = of_nat k * z"
    using lattice_layer_subset[of k] assms by auto
  have "real k * Inf_para ≤ real k * norm z"
    by (intro mult_left_mono Inf_para_le z) auto
  also have "... = norm ω"
    by (simp add: z norm_mult)
  finally show ?thesis .
qed

corollary Inf_para_le_norm:
  assumes "ω ∈ Λ*"
  shows "Inf_para ≤ norm ω"
proof -
  from assms obtain k where ω: "ω ∈ lattice_layer k" and "k ≠ 0"
    unfolding lattice0_def by (metis DiffE UN_iff lattice_layer_0 lattice_layer_covers)
  with Inf_para_pos have "Inf_para ≤ real k * Inf_para"
    by auto
  then show ?thesis
    using ω lattice_layer_le_norm by force
qed
```

One easy corollary is now that our lattice is discrete in the sense that there is a positive real number that bounds the distance between any two points from below.

```
lemma Inf_para_le_dist:
  assumes "x ∈ Λ" "y ∈ Λ" "x ≠ y"
  shows   "dist x y ≥ Inf_para"
proof -
  have "x - y ∈ Λ" "x - y ≠ 0"
    using assms by (auto intro: diff_in_lattice)
  hence "x - y ∈ Λ*"
```

```
    by auto
  hence "Inf_para ≤ norm (x - y)"
    by (rule Inf_para_le_norm)
  thus ?thesis
    by (simp add: dist_norm)
qed


definition Sup_para :: real where — R in the proof of Lemma 1
  "Sup_para ≡ Sup (norm ‘ lattice_layer_path)"

lemma Sup_para_ge:
  assumes "z ∈ lattice_layer_path"
  shows   "Sup_para ≥ norm z"
  unfolding Sup_para_def
proof (rule cSup_upper)
  show "bdd_above (norm ‘ lattice_layer_path)"
    unfolding bdd_above_norm by (rule compact_imp_bounded) auto
qed (use assms in auto)

lemma Sup_para_pos: "Sup_para > 0"
proof -
  have "0 < norm ω1"
    using ω1_nonzero by auto
  also have "... ≤ Sup_para"
    by (rule Sup_para_ge) (auto simp: in_lattice_layer_path_iff)
  finally show ?thesis .
qed

lemma Sup_para_nonzero [simp]: "Sup_para ≠ 0"
  using Sup_para_pos by linarith

lemma lattice_layer_ge_norm:
  assumes "ω ∈ lattice_layer k"
  shows "norm ω ≤ k * Sup_para"
proof -
  obtain z where z: "z ∈ lattice_layer_path" "ω = of_nat k * z"
    using lattice_layer_subset[of k] assms by auto
  have "norm ω = real k * norm z"
    by (simp add: z norm_mult)
  also have "... ≤ real k * Sup_para"
    by (intro mult_left_mono Sup_para_ge z) auto
  finally show ?thesis .
qed
```

We can now easily show that our lattice is a sparse set (i.e. it has no limit points). This also implies that it is closed.

```
lemma not_islimpt_lattice: "¬z islimpt Λ"
proof (rule discrete_imp_not_islimpt[of Inf_para])
```

```
  fix x y assume "x ∈ Λ" "y ∈ Λ" "dist x y < Inf_para"
  with Inf_para_le_dist[of x y] show "x = y"
    by (cases "x = y") auto
qed (fact Inf_para_pos)

lemma closed_lattice: "closed lattice"
  unfolding closed_limpt by (auto simp: not_islimpt_lattice)

lemma lattice_sparse: "Λ sparse_in UNIV"
  using not_islimpt_lattice sparse_in_def by blast
```

Any non-empty set of lattice points has one lattice point that is closer to the origin than all others.

```
lemma shortest_lattice_vector_exists:
  assumes "X ⊆ Λ" "X ≠ {}"
  obtains x where "x ∈ X" "⋀y. y ∈ X ⟹ norm x ≤ norm y"
proof -
  obtain x0 where x0: "x0 ∈ X"
    using assms by auto
  have "¬z islimpt X" for z
    using not_islimpt_lattice assms(1) islimpt_subset by blast
  hence "finite (cball 0 (norm x0) ∩ X)"
    by (intro finite_not_islimpt_in_compact) auto
  moreover have "x0 ∈ cball 0 (norm x0) ∩ X"
    using x0 by auto
  ultimately obtain x where x: "is_arg_min norm (λx. x ∈ cball 0 (norm
x0) ∩ X) x"
    using ex_is_arg_min_if_finite[of "cball 0 (norm x0) ∩ X" norm] by
blast
  thus ?thesis
    by (intro that[of x]) (auto simp: is_arg_min_def)
qed
```

If x is a non-zero lattice point then there exists another lattice point that is not collinear with x, i.e. that does not lie on the line through 0 and x.

```
lemma noncollinear_lattice_point_exists:
  assumes "x ∈ Λ*"
  obtains y where "y ∈ Λ*" "y / x ∉ ℝ"
proof -
  from assms obtain m n where x: "x = of_ω12_coords (of_int m, of_int
n)" and "x ≠ 0"
    by (elim latticeE lattice0E DiffE) auto
  define y where "y = of_ω12_coords (of_int (-n), of_int m)"
  have "y ∈ Λ"
    by (auto simp: y_def)
  moreover have "y ≠ 0"
    using ‹x ≠ 0› by (auto simp: x y_def of_ω12_coords_eq_0_iff prod_eq_iff)
  moreover have "y / x ∉ ℝ"
  proof
```

```
    assume "y / x ∈ ℝ"
    then obtain a where "y / x = of_real a"
      by (elim Reals_cases)
    hence y: "y = a *ℝ x"
      using assms ‹x ≠ 0› by (simp add: field_simps scaleR_conv_of_real)
    have "of_ω12_coords (-real_of_int n, real_of_int m) =
          of_ω12_coords (a * real_of_int m, a * real_of_int n)"
      using y by (simp add: x y_def algebra_simps flip: of_ω12_coords.scaleR)
    hence eq: "-real_of_int n = a * real_of_int m" "real_of_int m = a
* real_of_int n"
      unfolding of_ω12_coords_eq_iff prod_eq_iff fst_conv snd_conv by
blast+
    have "m ≠ 0 ∨ n ≠ 0"
      using ‹x ≠ 0› by (auto simp: x)
    with eq[symmetric] have nz: "m ≠ 0" "n ≠ 0"
      by auto
    have "a ^ 2 * real_of_int m = a * (a * real_of_int m)"
      by (simp add: power2_eq_square algebra_simps)
    also have "... = (-1) * real_of_int m"
      by (simp flip: eq)
    finally have "a ^ 2 = -1"
      using ‹m ≠ 0› by (subst (asm) mult_right_cancel) auto
    moreover have "a ^ 2 ≥ 0"
      by simp
    ultimately show False
      by linarith
  qed
  ultimately show ?thesis
    by (intro that) auto
qed
```

We can always easily find a period parallelogram whose border does not touch any given set of points we want to avoid, as long as that set is sparse.

```
lemma shifted_period_parallelogram_avoid:
  assumes "countable avoid"
  obtains orig where "path_image (parallelogram_path orig ω1 ω2) ∩ avoid
= {}"
proof -
  define avoid' where "avoid' = ω12_coords ` avoid"

  define avoid1 where "avoid1 = fst ` avoid'"
  define avoid2 where "avoid2 = snd ` avoid'"
  define avoid''
    where "avoid'' = (avoid1 ∪ (λx. x - 1) ` avoid1) × UNIV ∪ UNIV ×
(avoid2 ∪ (λx. x - 1) ` avoid2)"

  obtain orig where orig: "orig ∉ avoid''"
  proof -
    have *: "avoid1 ∪ (λx. x - 1) ` avoid1 ∈ null_sets lborel"
```

```
          "avoid2 ∪ (λx. x - 1) ‘ avoid2 ∈ null_sets lborel"
      by (rule null_sets.Un; rule countable_imp_null_set_lborel;
          use assms in ‹force simp: avoid1_def avoid2_def avoid'_def›)+
    have "avoid'' ∈ null_sets lborel"
      unfolding lborel_prod[symmetric] avoid''_def using * by (intro null_sets.Un)
auto
    hence "AE z in lborel. z ∉ avoid''"
      using AE_not_in by blast
    from eventually_happens[OF this] show ?thesis using that
      by (auto simp: ae_filter_eq_bot_iff)
  qed

  have *: "(λx. x - 1) ‘ X = ((+) 1) -‘ X" for X :: "real set"
    by force

  have fst_orig: "fst z ∉ {fst orig, fst orig + 1}" if "z ∈ avoid'" for
z
  proof
    assume "fst z ∈ {fst orig, fst orig + 1}"
    hence "orig ∈ (avoid1 ∪ (λx. x - 1) ‘ avoid1) × UNIV"
      using that unfolding avoid1_def * by (cases orig; cases z) force
    thus False using orig
      by (auto simp: avoid''_def)
  qed

  have snd_orig: "snd z ∉ {snd orig, snd orig + 1}" if "z ∈ avoid'" for
z
  proof
    assume "snd z ∈ {snd orig, snd orig + 1}"
    hence "orig ∈ UNIV × (avoid2 ∪ (λx. x - 1) ‘ avoid2)"
      using that unfolding avoid2_def * by (cases orig; cases z) force

    thus False using orig
      by (auto simp: avoid''_def)
  qed

  show ?thesis
  proof (rule that[of "of_ω12_coords orig"], safe)
    fix z assume z: "z ∈ path_image (parallelogram_path (of_ω12_coords
orig) ω1 ω2)" "z ∈ avoid"
    have "ω12_coords z ∈ ω12_coords ‘path_image (parallelogram_path (of_ω12_coords
orig) ω1 ω2)"
      using z(1) by blast
    thus "z ∈ {}" using z(2) fst_orig[of "ω12_coords z"] snd_orig[of
"ω12_coords z"]
      unfolding path_image_parallelogram_path'
      by (auto simp: avoid'_def ω12_coords.add box_prod)
  qed
qed
```

We can also prove a rule that allows us to prove a property about period parallelograms while assuming w.l.o.g. that the border of the parallelogram does not touch an arbitrary sparse set of points we want to avoid and the property we want to prove is invariant under shifting the parallelogram by an arbitrary amount.

This will be useful later for the use case of showing that any period parallelograms contain the same number of zeros as poles, which is proven by integrating along the border of a period parallelogram that is assume w.l.o.g. not to have any zeros or poles on its border.

```
lemma shifted_period_parallelogram_avoid_wlog [consumes 1, case_names
shift avoid]:
  assumes "⋀z. ¬z islimpt avoid"
  assumes "⋀orig d. finite (closure (period_parallelogram orig) ∩ avoid)
⟹
                    finite (closure (period_parallelogram (orig + d))
∩ avoid) ⟹
                    P orig ⟹ P (orig + d)"
  assumes "⋀orig. finite (closure (period_parallelogram orig) ∩ avoid)
⟹
                   path_image (parallelogram_path orig ω1 ω2) ∩ avoid
= {} ⟹
                   P orig"
  shows   "P orig"
proof -
  from assms have countable: "countable avoid"
    using no_limpt_imp_countable by blast

  from shifted_period_parallelogram_avoid[OF countable]
  obtain orig' where orig': "path_image (parallelogram_path orig' ω1
ω2) ∩ avoid = {}"
      by blast
  define d where "d = ω12_coords (orig - orig')"

  have "compact (closure (period_parallelogram orig))" for orig
    by (rule compact_closure_period_parallelogram)
  hence fin: "finite (closure (period_parallelogram orig) ∩ avoid)" for
orig
    using assms by (intro finite_not_islimpt_in_compact) auto

  from orig' have "P orig'"
    by (intro assms fin)
  have "P (orig' + (orig - orig'))"
    by (rule assms(2)) fact+
  thus ?thesis
    by (simp add: algebra_simps)
qed

end
```

The standard lattice is one that has been rotated and scaled such that the first generator is 1 and the second generator τ lies in the upper half plane.

```
locale std_complex_lattice =
  fixes τ :: complex (structure)
  assumes Im_τ_pos: "Im τ > 0"
begin

sublocale complex_lattice 1 τ
  by standard (use Im_τ_pos in ‹auto elim!: Reals_cases simp: fundpair_def›)

lemma winding_number_parallelogram_inside':
  assumes "w ∈ interior (period_parallelogram z)"
  shows   "winding_number (parallelogram_path z 1 τ) w = 1"
  using winding_number_parallelogram_inside[OF assms] Im_τ_pos by simp

end
```

3.6 Doubly-periodic functions

The following locale can be useful to prove that certain things respect the equivalence relation defined by the lattice: it shows that a doubly periodic function gives the same value for all equivalent points. Note that this is useful even for functions `f` that are only doubly quasi-periodic, since one might then still be able to prove that the function `λz. f z = 0` or `zorder f` or `is_pole f` are doubly periodic, so the zeros and poles of `f` are distributed according to the lattice symmetry.

```
locale pre_complex_lattice_periodic = pre_complex_lattice +
  fixes f :: "complex ⇒ 'a"
  assumes f_periodic: "f (z + ω1) = f z" "f (z + ω2) = f z"
begin

lemma lattice_cong:
  assumes "rel x y"
  shows   "f x = f y"
proof -
  define z where "z = y - x"
  from assms have z: "z ∈ Λ"
    using pre_complex_lattice.rel_def pre_complex_lattice.rel_sym z_def
by blast
  have "f (x + z) = f x"
    using z
  proof (induction arbitrary: x rule: lattice_induct)
    case (uminus w x)
    show ?case
      using uminus[of "x - w"] by simp
  qed (auto simp: f_periodic simp flip: add.assoc)
  thus ?thesis
```

```
    by (simp add: z_def)
qed

end

locale complex_lattice_periodic =
  complex_lattice ω1 ω2 + pre_complex_lattice_periodic ω1 ω2 f
  for ω1 ω2 :: complex and f :: "complex ⇒ 'a"
begin

lemma eval_to_fund_parallelogram: "f (to_fund_parallelogram z) = f z"
  by (rule lattice_cong) auto

end

locale complex_lattice_periodic_compose =
  complex_lattice_periodic ω1 ω2 f for ω1 ω2 :: complex and f :: "complex
⇒ 'a" +
fixes h :: "'a ⇒ 'b"
begin

sublocale compose: complex_lattice_periodic ω1 ω2 "λz. h (f z)"
  by standard (auto intro!: arg_cong[of _ _ h] lattice_cong simp: rel_def)

end

end
```

4 Fundamental regions of the modular group

```
theory Modular_Fundamental_Region
  imports Modular_Group Complex_Lattices "HOL-Library.Real_Mod"
begin
```

4.1 Definition

A fundamental region of a subgroup of the modular group is an open subset of the upper half of the complex plane that contains at most one representative of every equivalence class and whose closure contains at least one representative of every equivalence class.

```
locale fundamental_region = modgrp_subgroup +
  fixes R :: "complex set"
  assumes "open": "open R"
  assumes subset: "R ⊆ {z. Im z > 0}"
  assumes unique: "⋀x y. x ∈ R ⟹ y ∈ R ⟹ rel x y ⟹ x = y"
  assumes equiv_in_closure: "⋀x. Im x > 0 ⟹ ∃y∈closure R. rel x y
"
begin
```

The uniqueness property can be extended to the closure of `R`:

```
lemma unique':
  assumes "x ∈ R" "y ∈ closure R" "rel x y" "Im y > 0"
  shows   "x = y"
proof (cases "y ∈ R")
  case False
  show ?thesis
  proof (rule ccontr)
    assume xy: "x ≠ y"
    from assms have "rel y x"
      by (simp add: rel_commutes)
    then obtain f where f: "x = apply_modgrp f y" "f ∈ G"
      unfolding rel_def by blast

    have "continuous_on {z. Im z > 0} (apply_modgrp f)"
      by (intro continuous_intros) auto
    hence "isCont (apply_modgrp f) y"
      using open_halfspace_Im_gt[of 0] assms continuous_on_eq_continuous_at
by blast
    hence lim: "apply_modgrp f -y→ x"
      using f by (simp add: isCont_def)

    define ε where "ε = dist x y / 2"
    have ε: "ε > 0"
      using xy by (auto simp: ε_def)

    have "eventually (λw. w ∈ ball x ε ∩ R) (nhds x)"
      by (intro eventually_nhds_in_open) (use assms ε "open" in auto)
    from this and lim have "eventually (λz. apply_modgrp f z ∈ ball
x ε ∩ R) (at y)"
      by (rule eventually_compose_filterlim)
    moreover have "eventually (λz. z ∈ ball y ε) (nhds y)"
      using assms ε by (intro eventually_nhds_in_open) auto
    hence "eventually (λz. z ∈ ball y ε) (at y)"
      unfolding eventually_at_filter by eventually_elim auto
    ultimately have "eventually (λz. z ∈ ball y ε ∧ apply_modgrp f z
∈ R ∩ ball x ε) (at y)"
      by eventually_elim auto
    moreover have "y islimpt R"
      using ‹y ∈ closure R› ‹y ∉ R› by (auto simp: closure_def)
    hence "frequently (λz. z ∈ R) (at y)"
      using islimpt_conv_frequently_at by blast
    ultimately have "frequently (λz.
                        (z ∈ ball y ε ∧ apply_modgrp f z ∈ R ∩ ball
x ε) ∧ z ∈ R) (at y)"
      by (intro frequently_eventually_conj)
    hence "frequently (λz. False) (at y)"
    proof (rule frequently_elim1)
      fix z assume z: "(z ∈ ball y ε ∧ apply_modgrp f z ∈ R ∩ ball x
```

```
ε) ∧ z ∈ R"
      have "ball y ε ∩ ball x ε = {}"
        by (intro disjoint_ballI) (auto simp: ε_def dist_commute)
      with z have "apply_modgrp f z ≠ z"
        by auto
      with z f subset show False
        using unique[of z "apply_modgrp f z"] by auto
    qed
    thus False
      by simp
  qed
qed (use assms unique in auto)

lemma
  pole_modgrp_not_in_region [simp]: "pole_modgrp f ∉ R" and
  pole_image_modgrp_not_in_region [simp]: "pole_image_modgrp f ∉ R"
  using subset by force+


end
```

4.2 The standard fundamental region

The standard fundamental region $\mathcal{R}_\Gamma$ consists of all the points *z* in the upper half plane with *|z| > 1* and $|\mathrm{Re}(z)| < \frac{1}{2}$.

```
definition std_fund_region :: "complex set" ("\<R>\<Gamma>") where
  "\<R>\<Gamma> = -cball 0 1 ∩ Re -` {-1/2<..<1/2} ∩ {z. Im z > 0}"
```

The following version of $\mathcal{R}_\Gamma$ is what Apostol refers to as the closure of $\mathcal{R}_\Gamma$, but it is actually only part of the closure: since each point at the border of the fundamental region is equivalent to its mirror image w.r.t. the *Im(z) = 0* axis, we only want one of these copies to be in $\mathcal{R}_\Gamma$', and we choose the left one.

So $\mathcal{R}_\Gamma$' is actually $\mathcal{R}_\Gamma$ plus all the points on the left border plus all points on the left half of the semicircle.

```
definition std_fund_region' :: "complex set" ("\<R>\<Gamma>'") where
  "\<R>\<Gamma>' = \<R>\<Gamma> ∪ (-ball 0 1 ∩ Re -` {-1/2..0} ∩ {z. Im z > 0})"

lemma std_fund_region_altdef:
  "\<R>\<Gamma> = {z. norm z > 1 ∧ norm (z + cnj z) < 1 ∧ Im z > 0}"
  by (auto simp: std_fund_region_def complex_add_cnj)

lemma in_std_fund_region_iff:
  "z ∈ \<R>\<Gamma> ⟷ norm z > 1 ∧ Re z ∈ {-1/2<..<1/2} ∧ Im z > 0"
  by (auto simp: std_fund_region_def field_simps)

lemma in_std_fund_region'_iff:
```

```
  "z ∈ ℛΓ' ⟷ Im z > 0 ∧ ((norm z > 1 ∧ Re z ∈ {-1/2..<1/2}) ∨ (norm
z = 1 ∧ Re z ∈ {-1/2..0}))"
  by (auto simp: std_fund_region'_def std_fund_region_def field_simps)

lemma open_std_fund_region [simp, intro]: "open ℛΓ"
  unfolding std_fund_region_def
  by (intro open_Int open_vimage continuous_intros open_halfspace_Im_gt)
auto

lemma Im_std_fund_region: "z ∈ ℛΓ ⟹ Im z > 0"
  by (auto simp: std_fund_region_def)
```

We now show that the closure of the standard fundamental region contains exactly those points z with $|z| \geq 1$ and $|\mathrm{Re}(z)| \leq \frac{1}{2}$.

```
context
  fixes S S' :: "(real × real) set" and T :: "complex set"
  fixes f :: "real × real ⇒ complex" and g :: "complex ⇒ real × real"
  defines "f ≡ (λ(x,y). Complex x (y + sqrt (1 - x ^ 2)))"
  defines "g ≡ (λz. (Re z, Im z - sqrt (1 - Re z ^ 2)))"
  defines "S ≡ ({-1/2<..<1/2} × {0<..})"
  defines "S' ≡ ({-1/2..1/2} × {0..})"
  defines "T ≡ {z. norm z ≥ 1 ∧ Re z ∈ {-1/2..1/2} ∧ Im z ≥ 0}"
begin

lemma image_subset_std_fund_region: "f ` S ⊆ ℛΓ"
  unfolding subset_iff in_std_fund_region_iff S_def
proof safe
  fix a b :: real
  assume ab: "a ∈ {-1/2<..<1/2}" "b > 0"
  have "|a| ^ 2 ≤ (1 / 2) ^ 2"
    using ab by (intro power_mono) auto
  hence "a ^ 2 ≤ 1 / 4"
    by (simp add: power2_eq_square)
  hence "a ^ 2 ≤ 1"
    by simp

  show "Im (f (a, b)) > 0"
    using ab ‹a ^ 2 ≤ 1 / 4› by (auto simp: f_def intro: add_pos_nonneg)

  show "Re (f (a, b)) ∈ {-1/2<..<1/2}"
    using ab by (simp add: f_def)

  have "1 ^ 2 = a^2 + (0 + sqrt (1 - a^2)) ^ 2"
    using ‹a ^ 2 ≤ 1 / 4› by (simp add: power2_eq_square algebra_simps)
  also have "a^2 + (0 + sqrt (1 - a^2)) ^ 2 < a^2 + (b + sqrt (1 - a^2)) ^
2"
    using ab ‹a ^ 2 ≤ 1› by (intro add_strict_left_mono power2_mono power2_strict_mono)
auto
  also have "... = norm (f (a, b)) ^ 2"
```

```
    by (simp add: f_def norm_complex_def)
  finally show "norm (f (a, b)) > 1"
    by (rule power2_less_imp_less) auto
qed

lemma image_std_fund_region_subset: "g ` \<R>\<^sub>\<Gamma> \<subseteq> S"
  unfolding subset_iff g_def S_def
proof safe
  fix z :: complex
  assume "z \<in> \<R>\<^sub>\<Gamma>"
  hence z: "norm z > 1" "Re z \<in> {-1/2<..<1/2}" "Im z > 0"
    by (auto simp: in_std_fund_region_iff)

  have "|Re z| ^ 2 \<le> (1 / 2) ^ 2"
    using z by (intro power_mono) auto
  hence "Re z ^ 2 \<le> 1 / 4"
    by (simp add: power2_eq_square)
  hence "Re z ^ 2 \<le> 1"
    by simp

  from z show "Re z \<in> {- 1 / 2<..<1 / 2}"
    by auto

  have "sqrt (1 - Re z ^ 2) ^ 2 = 1 - Re z ^ 2"
    using \<open>Re z ^ 2 \<le> 1\<close> by simp
  also have "... < Im z ^ 2"
    using z by (simp add: norm_complex_def algebra_simps)
  finally have "sqrt (1 - Re z ^ 2) < Im z"
    by (rule power2_less_imp_less) (use z in auto)
  thus "Im z - sqrt (1 - Re z ^ 2) > 0"
    by simp
qed

lemma std_fund_region_map_inverses: "f (g x) = x" "g (f y) = y"
  by (simp_all add: f_def g_def case_prod_unfold)

lemma bij_betw_std_fund_region1: "bij_betw f S \<R>\<^sub>\<Gamma>"
  using image_std_fund_region_subset image_subset_std_fund_region
  by (intro bij_betwI[of _ _ _ g]) (auto simp: std_fund_region_map_inverses)

lemma bij_betw_std_fund_region2: "bij_betw g \<R>\<^sub>\<Gamma> S"
  using image_std_fund_region_subset image_subset_std_fund_region
  by (intro bij_betwI[of _ _ _ f]) (auto simp: std_fund_region_map_inverses)

lemma image_subset_std_fund_region': "f ` S' \<subseteq> T"
  unfolding subset_iff S'_def T_def
proof safe
  fix a b :: real
  assume ab: "a \<in> {-1/2..1/2}" "b \<ge> 0"
```

```
  have "|a| ^ 2 ≤ (1 / 2) ^ 2"
    using ab by (intro power_mono) auto
  hence "a ^ 2 ≤ 1 / 4"
    by (simp add: power2_eq_square)
  hence "a ^ 2 ≤ 1"
    by simp

  show "Im (f (a, b)) ≥ 0"
    using ab ‹a ^ 2 ≤ 1 / 4› by (auto simp: f_def intro: add_pos_nonneg)

  show "Re (f (a, b)) ∈ {-1/2..1/2}"
    using ab by (simp add: f_def)

  have "1 ^ 2 = a² + (0 + sqrt (1 - a²)) ^ 2"
    using ‹a ^ 2 ≤ 1 / 4› by (simp add: power2_eq_square algebra_simps)
  also have "a² + (0 + sqrt (1 - a²)) ^ 2 ≤ a² + (b + sqrt (1 - a²))
^ 2"
    using ab ‹a ^ 2 ≤ 1› by (intro add_left_mono power2_mono power2_strict_mono)
auto
  also have "... = norm (f (a, b)) ^ 2"
    by (simp add: f_def norm_complex_def)
  finally show "norm (f (a, b)) ≥ 1"
    by (rule power2_le_imp_le) auto
qed

lemma image_std_fund_region_subset': "g ` T ⊆ S'"
  unfolding subset_iff g_def S'_def
proof safe
  fix z :: complex
  assume "z ∈ T"
  hence z: "norm z ≥ 1" "Re z ∈ {-1/2..1/2}" "Im z ≥ 0"
    by (auto simp: T_def)

  have "|Re z| ^ 2 ≤ (1 / 2) ^ 2"
    using z by (intro power_mono) auto
  hence "Re z ^ 2 ≤ 1 / 4"
    by (simp add: power2_eq_square)
  hence "Re z ^ 2 ≤ 1"
    by simp

  from z show "Re z ∈ {-1/2..1/2}"
    by auto

  have "sqrt (1 - Re z ^ 2) ^ 2 = 1 - Re z ^ 2"
    using ‹Re z ^ 2 ≤ 1› by simp
  also have "... ≤ Im z ^ 2"
    using z by (simp add: norm_complex_def algebra_simps)
  finally have "sqrt (1 - Re z ^ 2) ≤ Im z"
    by (rule power2_le_imp_le) (use z in auto)
```

```
  thus "Im z - sqrt (1 - Re z ^ 2) ≥ 0"
    by simp
qed

lemma bij_betw_std_fund_region1': "bij_betw f S' T"
  using image_std_fund_region_subset' image_subset_std_fund_region'
  by (intro bij_betwI[of _ _ _ g]) (auto simp: std_fund_region_map_inverses)

lemma bij_betw_std_fund_region2': "bij_betw g T S'"
  using image_std_fund_region_subset' image_subset_std_fund_region'
  by (intro bij_betwI[of _ _ _ f]) (auto simp: std_fund_region_map_inverses)

lemma closure_std_fund_region: "closure 𝓡_Γ = T"
proof -
  have homeo: "homeomorphism S 𝓡_Γ f g"
    using image_std_fund_region_subset image_subset_std_fund_region
    by (intro homeomorphismI)
       (auto simp: g_def f_def case_prod_unfold intro!: continuous_intros)

  have "closure 𝓡_Γ = closure (f ` S)"
    using bij_betw_std_fund_region1 by (simp add: bij_betw_def)
  also have "... = f ` closure S"
    using bij_betw_std_fund_region1 homeo
  proof (rule closure_bij_homeomorphic_image_eq)
    show "continuous_on UNIV f" "continuous_on UNIV g"
      by (auto simp: f_def g_def case_prod_unfold intro!: continuous_intros)
  qed (auto simp: std_fund_region_map_inverses)
  also have "closure S = {-1 / 2..1 / 2} × {0..}"
    by (simp add: S_def closure_Times)
  also have "... = S'"
    by (simp add: S'_def)
  also have "f ` S' = T"
    using bij_betw_std_fund_region1' by (simp add: bij_betw_def)
  finally show ?thesis .
qed

lemma in_closure_std_fund_region_iff:
  "x ∈ closure 𝓡_Γ ⟷ norm x ≥ 1 ∧ Re x ∈ {-1/2..1/2} ∧ Im x ≥ 0"
  by (simp add: closure_std_fund_region T_def)

lemma frontier_std_fund_region:
  "frontier 𝓡_Γ =
     {z. norm z ≥ 1 ∧ Im z > 0 ∧ |Re z| = 1 / 2} ∪
     {z. norm z = 1 ∧ Im z > 0 ∧ |Re z| ≤ 1 / 2}" (is "_ = ?rhs")
proof -
  have [simp]: "x ^ 2 ≥ 1 ⟷ x ≥ 1 ∨ x ≤ -1" for x :: real
    using abs_le_square_iff[of 1 x] by auto
  have "frontier 𝓡_Γ = closure 𝓡_Γ - 𝓡_Γ"
    unfolding frontier_def by (subst interior_open) simp_all
```

```
  also have "... = ?rhs"
    unfolding closure_std_fund_region unfolding std_fund_region_def
    by (auto simp: cmod_def T_def)
  finally show ?thesis .
qed

lemma std_fund_region'_subset_closure: "\<R>\<Gamma>' \<subseteq> closure \<R>\<Gamma>"
  by (auto simp: in_std_fund_region'_iff in_closure_std_fund_region_iff)

lemma std_fund_region'_superset: "\<R>\<Gamma> \<subseteq> \<R>\<Gamma>'"
  by (auto simp: in_std_fund_region'_iff in_std_fund_region_iff)

lemma in_std_fund_region'_not_on_frontier_iff:
  assumes "z \<notin> frontier \<R>\<Gamma>"
  shows   "z \<in> \<R>\<Gamma>' \<longleftrightarrow> z \<in> \<R>\<Gamma>"
proof
  assume "z \<in> \<R>\<Gamma>'"
  hence "z \<in> closure \<R>\<Gamma>"
    using std_fund_region'_subset_closure by blast
  thus "z \<in> \<R>\<Gamma>"
    using assms by (auto simp: frontier_def interior_open)
qed (use std_fund_region'_superset in auto)

lemma simply_connected_std_fund_region: "simply_connected \<R>\<Gamma>"
proof (rule simply_connected_retraction_gen)
  show "simply_connected S"
    unfolding S_def by (intro convex_imp_simply_connected convex_Times)
auto
  show "continuous_on S f"
    unfolding f_def S_def case_prod_unfold by (intro continuous_intros)
  show "continuous_on \<R>\<Gamma> g"
    unfolding g_def case_prod_unfold by (intro continuous_intros)
  show "f ` S = \<R>\<Gamma>"
    using bij_betw_std_fund_region1 by (simp add: bij_betw_def)
  show "g \<in> \<R>\<Gamma> \<rightarrow> S"
    using bij_betw_std_fund_region2 bij_betw_imp_funcset by blast
  show "f (g x) = x" for x
    by (simp add: f_def g_def)
qed

lemma simply_connected_closure_std_fund_region: "simply_connected (closure
\<R>\<Gamma>)"
proof (rule simply_connected_retraction_gen)
  show "simply_connected S'"
    unfolding S'_def by (intro convex_imp_simply_connected convex_Times)
auto
  show "continuous_on S' f"
    unfolding f_def S'_def case_prod_unfold by (intro continuous_intros)
  show "continuous_on (closure \<R>\<Gamma>) g"
```

```
    unfolding g_def case_prod_unfold by (intro continuous_intros)
  show "f ‘ S' = closure \<R>\<^sub>\<Gamma>"
    using bij_betw_std_fund_region1' by (simp add: bij_betw_def closure_std_fund_region)
  show "g ∈ closure \<R>\<^sub>\<Gamma> → S'"
    using bij_betw_std_fund_region2' bij_betw_imp_funcset closure_std_fund_region
by blast
  show "f (g x) = x" for x
    by (simp add: f_def g_def)
qed

lemma std_fund_region'_subset: "\<R>\<^sub>\<Gamma>' ⊆ closure \<R>\<^sub>\<Gamma>"
  unfolding std_fund_region'_def closure_std_fund_region T_def unfold-
ing std_fund_region_def
  by auto

lemma closure_std_fund_region_Im_pos: "closure \<R>\<^sub>\<Gamma> ⊆ {z. Im z > 0}"
  unfolding closure_std_fund_region
  by (auto intro!: neq_le_trans simp: norm_complex_def field_simps power2_ge_1_iff
T_def)

lemma closure_std_fund_region_Im_ge: "closure \<R>\<^sub>\<Gamma> ⊆ {z. Im z ≥ sqrt
3 / 2}"
proof
  fix z assume "z ∈ closure \<R>\<^sub>\<Gamma>"
  hence *: "norm z ≥ 1" "|Re z| ≤ 1 / 2" "Im z ≥ 0"
    by (auto simp: closure_std_fund_region T_def)
  have "1 ≤ norm z ^ 2"
    using * by simp
  also have "norm z ^ 2 ≤ (1 / 2) ^ 2 + Im z ^ 2"
    unfolding cmod_power2 by (intro add_right_mono power2_mono) (use *
in auto)
  finally have "Im z ^ 2 ≥ (sqrt 3 / 2) ^ 2"
    by (simp add: power2_eq_square)
  hence "Im z ≥ sqrt 3 / 2"
    by (subst (asm) abs_le_square_iff [symmetric]) (use * in auto)
  thus "z ∈ {z. Im z ≥ sqrt 3 / 2}"
    by simp
qed

lemma std_fund_region'_minus_std_fund_region:
  "\<R>\<^sub>\<Gamma>' - \<R>\<^sub>\<Gamma> =
     {z. norm z = 1 ∧ Im z > 0 ∧ Re z ∈ {-1/2..0}} ∪ {z. Re z = -1
/ 2 ∧ Im z ≥ sqrt 3 / 2}"
  (is "?lhs = ?rhs")
proof (intro equalityI subsetI)
  fix z assume z: "z ∈ ?lhs"
  from z have "Im z ≥ sqrt 3 / 2"
    using closure_std_fund_region_Im_ge std_fund_region'_subset by auto
  thus "z ∈ ?rhs" using z
```

```
    by (auto simp: std_fund_region'_def std_fund_region_def not_less)
next
  fix z assume z: "z ∈ ?rhs"
  have "sqrt 3 / 2 > 0"
    by simp
  have "Im z > 0"
    using z less_le_trans[OF ‹sqrt 3 / 2 > 0›, of "Im z"] by auto
  moreover have "norm z ≥ 1"
    using z
  proof
    assume "z ∈ {z. Re z = - 1 / 2 ∧ sqrt 3 / 2 ≤ Im z}"
    hence "norm z ^ 2 ≥ (-1/2) ^ 2 + (sqrt 3 / 2) ^ 2"
      unfolding cmod_power2 by (intro add_mono power2_mono) auto
    also have "(-1/2) ^ 2 + (sqrt 3 / 2) ^ 2 = 1"
      by (simp add: field_simps power2_eq_square)
    finally show "norm z ≥ 1"
      by (simp add: power2_nonneg_ge_1_iff)
  qed auto
  ultimately show "z ∈ ?lhs" using z
    by (auto simp: std_fund_region'_def std_fund_region_def)
qed

lemma closure_std_fund_region_minus_std_fund_region':
  "closure 𝓡Γ - 𝓡Γ' =
      {z. norm z = 1 ∧ Im z > 0 ∧ Re z ∈ {0<..1/2}} ∪ {z. Re z = 1 /
2 ∧ Im z ≥ sqrt 3 / 2}"
  (is "?lhs = ?rhs")
proof (intro equalityI subsetI)
  fix z assume z: "z ∈ closure 𝓡Γ - 𝓡Γ'"
  have "norm z ≥ 1"
    using z by (auto simp: closure_std_fund_region in_std_fund_region'_iff
not_le T_def)
  from z have "Im z > 0" "Im z ≥ sqrt 3 / 2"
    using closure_std_fund_region_Im_pos closure_std_fund_region_Im_ge
by blast+
  thus "z ∈ ?rhs" using z
    by (auto simp: closure_std_fund_region in_std_fund_region'_iff not_le
T_def)
next
  fix z assume "z ∈ ?rhs"
  thus "z ∈ ?lhs"
  proof
    assume "z ∈ {z. cmod z = 1 ∧ 0 < Im z ∧ Re z ∈ {0<..1 / 2}}"
    thus "z ∈ ?lhs"
      by (auto simp: closure_std_fund_region in_std_fund_region'_iff not_le
T_def)
  next
    assume z: "z ∈ {z. Re z = 1 / 2 ∧ sqrt 3 / 2 ≤ Im z}"
    have "0 < sqrt 3 / 2"
```

```
    by simp
  also have "... ≤ Im z"
    using z by auto
  finally have "Im z > 0" .
  have "norm z ^ 2 ≥ (1 / 2) ^ 2 + (sqrt 3 / 2) ^ 2"
    unfolding cmod_power2 by (intro add_mono power2_mono) (use z in
auto)
  also have "(1 / 2) ^ 2 + (sqrt 3 / 2) ^ 2 = 1"
    by (simp add: power2_eq_square)
  finally have "norm z ≥ 1"
    by (simp add: power2_nonneg_ge_1_iff)
  from this and ‹Im z > 0› and z show "z ∈ ?lhs"
    by (auto simp: closure_std_fund_region in_std_fund_region'_iff not_le
T_def)
 qed
qed

lemma cis_in_std_fund_region'_iff:
 assumes "φ ∈ {0..pi}"
 shows   "cis φ ∈ 𝓡_Γ' ⟷ φ ∈ {pi/2..2*pi/3}"
proof
 assume φ: "φ ∈ {pi/2..2*pi/3}"
 have "φ > 0"
   by (rule less_le_trans[of _ "pi / 2"]) (use φ in auto)
 moreover have "φ < pi"
   by (rule le_less_trans[of _ "2 * pi / 3"]) (use φ in auto)
 ultimately have "sin φ > 0"
   by (intro sin_gt_zero) auto
 moreover have "cos φ ≥ cos (2 * pi / 3)"
   using φ by (intro cos_monotone_0_pi_le) auto
 moreover have "cos φ ≤ cos (pi / 2)"
   using φ by (intro cos_monotone_0_pi_le) auto
 ultimately show "cis φ ∈ 𝓡_Γ'"
   by (auto simp: in_std_fund_region'_iff cos_120)
next
 assume "cis φ ∈ 𝓡_Γ'"
 hence *: "cos φ ≥ cos (2 * pi / 3)" "cos φ ≤ cos (pi / 2)"
   by (auto simp: in_std_fund_region'_iff cos_120)
 have "φ ≤ 2 * pi / 3"
   using *(1) assms by (subst (asm) cos_mono_le_eq) auto
 moreover have "φ ≥ pi / 2"
   using *(2) assms by (subst (asm) cos_mono_le_eq) auto
 ultimately show "φ ∈ {pi/2..2*pi/3}"
   by auto
qed

lemma imag_axis_in_std_fund_region'_iff: "y *_R i ∈ 𝓡_Γ' ⟷ y ≥ 1"
 by (auto simp: in_std_fund_region'_iff)
```

```
lemma vertical_left_in_std_fund_region'_iff:
  "-1 / 2 + y *R i ∈ 𝓡Γ' ⟷ y ≥ sqrt 3 / 2"
proof
  assume y: "y ≥ sqrt 3 / 2"
  have "1 = (1 / 2) ^ 2 + (sqrt 3 / 2) ^ 2"
    by (simp add: power2_eq_square)
  also have "... ≤ (1 / 2) ^ 2 + y ^ 2"
    using y by (intro add_mono power2_mono) auto
  also have "... = norm (y *R i - 1 / 2) ^ 2"
    unfolding cmod_power2 by simp
  finally have "norm (y *R i - 1 / 2) ≥ 1"
    by (simp add: power2_nonneg_ge_1_iff)
  moreover have "y > 0"
    by (rule less_le_trans[OF _ y]) auto
  ultimately show "-1 / 2 + y *R i ∈ 𝓡Γ'"
    using y by (auto simp: in_std_fund_region'_iff)
next
  assume *: "-1 / 2 + y *R i ∈ 𝓡Γ'"
  hence "y > 0"
    by (auto simp: in_std_fund_region'_iff)
  from * have "1 ≤ norm (y *R i - 1 / 2)"
    by (auto simp: in_std_fund_region'_iff)
  hence "1 ≤ norm (y *R i - 1 / 2) ^ 2"
    by (simp add: power2_nonneg_ge_1_iff)
  also have "... = (1 / 2) ^ 2 + y ^ 2"
    unfolding cmod_power2 by simp
  finally have "y ^ 2 ≥ (sqrt 3 / 2) ^ 2"
    by (simp add: algebra_simps power2_eq_square)
  hence "y ≥ sqrt 3 / 2"
    by (rule power2_le_imp_le) (use ‹y > 0› in auto)
  thus "y ≥ sqrt 3 / 2" using *
    by (auto simp: in_std_fund_region'_iff)
qed

lemma std_fund_region'_border_aux1:
  "{z. norm z = 1 ∧ 0 < Im z ∧ Re z ∈ {-1/2..0}} = cis ‘ {pi / 2..2 /
3 * pi}"
proof safe
  fix z :: complex assume z: "norm z = 1" "Im z > 0" "Re z ∈ {-1/2..0}"
  show "z ∈ cis ‘ {pi/2..2/3*pi}"
  proof (rule rev_image_eqI)
    from z have [simp]: "z ≠ 0"
      by auto
    have [simp]: "Arg z ≥ 0"
      using z by (auto simp: Arg_less_0)
    have z_eq: "cis (Arg z) = z"
      using z by (auto simp: cis_Arg complex_sgn_def)
    thus "z = cis (Arg z)"
      by simp
```

```
    have "Re (cis (Arg z)) ≥ -1/2"
      using z by (subst z_eq) auto
    hence "cos (Arg z) ≥ cos (2/3*pi)"
      by (simp add: cos_120 cos_120')
    hence "Arg z ≤ 2 / 3 * pi"
      using Arg_le_pi by (subst (asm) cos_mono_le_eq) auto
    moreover have "Re (cis (Arg z)) ≤ 0"
      using z by (subst z_eq) auto
    hence "cos (Arg z) ≤ cos (pi / 2)"
      by simp
    hence "Arg z ≥ pi / 2"
      using Arg_le_pi by (subst (asm) cos_mono_le_eq) auto
    ultimately show "Arg z ∈ {pi/2..2/3*pi}"
      by simp
  qed
next
  fix t :: real assume t: "t ∈ {pi/2..2/3*pi}"
  have "t > 0"
    by (rule less_le_trans[of _ "pi/2"]) (use t in auto)
  have "t < pi"
    by (rule le_less_trans[of _ "2/3*pi"]) (use t in auto)
  have "sin t > 0"
    using ‹t > 0› ‹t < pi› by (intro sin_gt_zero) auto
  moreover have "cos t ≤ cos (pi / 2)"
    using t ‹t < pi› by (intro cos_monotone_0_pi_le) auto
  moreover have "cos t ≥ cos (2*pi/3)"
    using t by (intro cos_monotone_0_pi_le) auto
  ultimately show "norm (cis t) = 1" "Im (cis t) > 0" "Re (cis t) ∈ {-1/2..0}"
    by (auto simp: cos_120 cos_120')
qed

lemma std_fund_region'_border_aux2:
  "{z. Re z = - 1 / 2 ∧ sqrt 3 / 2 ≤ Im z} = (λx. - 1 / 2 + x *R i) ‘
{sqrt 3 / 2..}"
  by (auto simp: complex_eq_iff)

lemma compact_std_fund_region:
  assumes "B > 1"
  shows "compact (closure 𝓡Γ ∩ {z. Im z ≤ B})"
  unfolding compact_eq_bounded_closed
proof
  show "closed (closure 𝓡Γ ∩ {z. Im z ≤ B})"
    by (intro closed_Int closed_halfspace_Im_le) auto
next
  show "bounded (closure 𝓡Γ ∩ {z. Im z ≤ B})"
  proof -
    have "closure 𝓡Γ ∩ {z. Im z ≤ B} ⊆ cbox (-1/2) (1/2 + i * B)"
      by (auto simp: in_closure_std_fund_region_iff in_cbox_complex_iff)
    moreover have "bounded (cbox (-1/2) (1/2 + i * B))"
```

```
    by simp
  ultimately show ?thesis
    using bounded_subset by blast
 qed
qed

end
```

4.3 Proving that the standard region is fundamental

```
lemma norm_open_segment_less:
  fixes x y z :: "'a :: euclidean_space"
  assumes "norm x ≤ norm y" "z ∈ open_segment x y"
  shows   "norm z < norm y"
  using assms
  by (metis (no_types, opaque_lifting) diff_zero dist_decreases_open_segment
        dist_norm norm_minus_commute order_less_le_trans)
```

Lemma 1

```
lemma (in complex_lattice) std_fund_region_fundamental_lemma1:
  obtains ω1' ω2' :: complex and a b c d :: int
  where "|a * d - b * c| = 1"
        "ω2' = of_int a * ω2 + of_int b * ω1"
        "ω1' = of_int c * ω2 + of_int d * ω1"
        "Im (ω2' / ω1') ≠ 0"
        "norm ω1' ≤ norm ω2'" "norm ω2' ≤ norm (ω1' + ω2')" "norm ω2'
≤ norm (ω1' - ω2')"
proof -
  have "Λ* ⊆ Λ" "Λ* ≠ {}"
    by auto
  then obtain ω1' where ω1': "ω1' ∈ Λ*" "⋀y. y ∈ Λ* ⟹ norm ω1' ≤
norm y"
    using shortest_lattice_vector_exists by blast

  define X where "X = {y. y ∈ Λ* ∧ y / ω1' ∉ ℝ}"
  have "X ⊆ Λ"
    by (auto simp: X_def lattice0_def)
  moreover have  "X ≠ {}"
    using noncollinear_lattice_point_exists[of ω1'] ω1'(1) unfolding
X_def by force
  ultimately obtain ω2' where ω2': "ω2' ∈ X" "⋀z. z ∈ X ⟹ norm ω2'
≤ norm z"
    using shortest_lattice_vector_exists by blast

  have [simp]: "ω1' ≠ 0" "ω2' ≠ 0"
    using ω1' ω2' by (auto simp: lattice0_def X_def)
  have noncollinear: "ω2' / ω1' ∉ ℝ"
    using ω2' by (auto simp: X_def)
  hence fundpair': "fundpair (ω1', ω2')"
```

```
    unfolding fundpair_def prod.case by simp
  have Im_nz: "Im (ω2' / ω1') ≠ 0"
    using noncollinear by (auto simp: complex_is_Real_iff)

  have "norm ω1' ≤ norm ω2'"
    by (intro ω1') (use ω2' in ‹auto simp: X_def›)

  have triangle: "z ∉ Λ" if z: "z ∈ convex hull {0, ω1', ω2'}" "z ∉
{0, ω1', ω2'}" for z
  proof
    assume "z ∈ Λ"
    hence "z ∈ Λ*"
      using z by (auto simp: lattice0_def)
    from that obtain a b where ab: "a ≥ 0" "b ≥ 0" "a + b ≤ 1" "z
= a *R ω1' + b *R ω2'"
      unfolding convex_hull_3_alt by (auto simp: scaleR_conv_of_real)

    have "norm z ≤ norm (a *R ω1') + norm (b *R ω2')"
      unfolding ab using norm_triangle_ineq by blast
    also have "... = a * norm ω1' + b * norm ω2'"
      using ab by simp
    finally have norm_z_le: "norm z ≤ a * norm ω1' + b * norm ω2'" .

    also have "... ≤ a * norm ω2' + b * norm ω2'"
      using ab ‹norm ω1' ≤ norm ω2'› by (intro add_mono mult_left_mono)
auto
    also have "... = (a + b) * norm ω2'"
      by (simp add: algebra_simps)
    finally have norm_z_le': "norm z ≤ (a + b) * norm ω2'" .

    have "z / ω1' ∉ ℝ"
    proof
      assume real: "z / ω1' ∈ ℝ"
      show False
      proof (cases "b = 0")
        case False
        hence "ω2' / ω1' = (z / ω1' - of_real a) / of_real b"
          by (simp add: ab field_simps scaleR_conv_of_real)
        also have "... ∈ ℝ"
          using real by (auto intro: Reals_divide Reals_diff)
        finally show False
          using noncollinear by contradiction
      next
        case True
        hence "z = a *R ω1'"
          using ab by simp
        from this and z have "a ≠ 1"
          by auto
        hence "a < 1"
```

```
      using ab by simp
    have "norm z = a * norm ω1'"
      using ‹z = a *R ω1'› ‹a ≥ 0› by simp
    also have "... < 1 * norm ω1'"
      using ‹a < 1› by (intro mult_strict_right_mono) auto
    finally have "norm z < norm ω1'"
      by simp
    moreover have "norm z ≥ norm ω1'"
      by (intro ω1') (use z ‹z ∈ Λ*› in auto)
    ultimately show False
      by simp
  qed
qed
hence "z ∈ X"
  using ‹z ∈ Λ*› by (auto simp: X_def)
hence "norm z ≥ norm ω2'"
  by (intro ω2')

moreover have "norm z ≤ norm ω2'"
proof -
  have "norm z ≤ (a + b) * norm ω2'"
    by (rule norm_z_le')
  also have "... ≤ 1 * norm ω2'"
    using ab by (intro mult_right_mono) auto
  finally show "norm z ≤ norm ω2'"
    by simp
qed

ultimately have norm_z: "norm z = norm ω2'"
  by linarith

have "¬(a + b < 1)"
proof
  assume *: "a + b < 1"
  have "norm z ≤ (a + b) * norm ω2'"
    by (rule norm_z_le')
  also have "... < 1 * norm ω2'"
    by (intro mult_strict_right_mono *) auto
  finally show False
    using norm_z by simp
qed
with ab have b_eq: "b = 1 - a"
  by linarith

have "norm z < norm ω2'"
proof (rule norm_open_segment_less)
  have "a ≠ 0" "a ≠ 1"
    using z ab by (auto simp: b_eq)
  hence "∃u. u > 0 ∧ u < 1 ∧ z = (1 - u) *R ω1' + u *R ω2'"
```

```
      using ab by (intro exI[of _ b]) (auto simp: b_eq)
    thus "z ∈ open_segment ω1' ω2'"
      using z ab noncollinear unfolding in_segment by auto
  next
    show "norm ω1' ≤ norm ω2'"
      by fact
  qed
  with norm_z show False
    by simp
qed
hence "convex hull {0, ω1', ω2'} ∩ Λ ⊆ {0, ω1', ω2'}"
  by blast
moreover have "{0, ω1', ω2'} ⊆ convex hull {0, ω1', ω2'} ∩ Λ"
  using ω1' ω2' by (auto intro: hull_inc simp: X_def)
ultimately have "convex hull {0, ω1', ω2'} ∩ Λ = {0, ω1', ω2'}"
  by blast

hence "equiv_fundpair (ω1, ω2) (ω1', ω2')"
  using fundpair' ω1' ω2' by (subst equiv_fundpair_iff_triangle) (auto
simp: X_def)
then obtain a b c d :: int where
  det: "|a * d - b * c| = 1" and
  abcd: "ω2' = of_int a * ω2 + of_int b * ω1" "ω1' = of_int c * ω2
+ of_int d * ω1"
  using fundpair fundpair' by (subst (asm) equiv_fundpair_iff) auto

have *: "norm (ω1' + of_int n * ω2') ≥ norm ω2'" if n: "n ≠ 0" for
n
proof (rule ω2')
  define z where "z = ω1' + of_int n * ω2'"
  have "z ∈ Λ"
    unfolding z_def using ω1'(1) ω2'(1) by (auto intro!: lattice_intros
simp: X_def)
  moreover have "z / ω1' ∉ ℝ"
  proof
    assume "z / ω1' ∈ ℝ"
    hence "(z / ω1' - 1) / of_int n ∈ ℝ"
      by auto
    also have "(z / ω1' - 1) / of_int n = ω2' / ω1'"
      using n by (simp add: field_simps z_def)
    finally show False
      using noncollinear by contradiction
  qed
  moreover from this have "z ≠ 0"
    by auto
  ultimately show "z ∈ X"
    by (auto simp: X_def lattice0_def)
qed
have norms: "norm (ω1' + ω2') ≥ norm ω2'" "norm (ω1' - ω2') ≥ norm
```

```
ω2'"
    using *[of 1] and *[of "-1"] by simp_all

  show ?thesis
    using det norms abcd noncollinear ‹norm ω1' ≤ norm ω2'›
    by (intro that[of a d b c ω2' ω1']) (simp_all add: complex_is_Real_iff)
qed

lemma (in complex_lattice) std_fund_region_fundamental_lemma2:
  obtains ω1' ω2' :: complex and a b c d :: int
  where "a * d - b * c = 1"
        "ω2' = of_int a * ω2 + of_int b * ω1"
        "ω1' = of_int c * ω2 + of_int d * ω1"
        "Im (ω2' / ω1') ≠ 0"
        "norm ω1' ≤ norm ω2'" "norm ω2' ≤ norm (ω1' + ω2')" "norm ω2'
≤ norm (ω1' - ω2')"
proof -
  obtain ω1' ω2' a b c d
    where abcd: "|a * d - b * c| = 1"
      and eq: "ω2' = of_int a * ω2 + of_int b * ω1" "ω1' = of_int c
* ω2 + of_int d * ω1"
      and nz: "Im (ω2' / ω1') ≠ 0"
      and norms: "norm ω1' ≤ norm ω2'" "norm ω2' ≤ norm (ω1' + ω2')"
"norm ω2' ≤ norm (ω1' - ω2')"
    using std_fund_region_fundamental_lemma1 .

  show ?thesis
  proof (cases "a * d - b * c = 1")
    case True
    thus ?thesis
      by (intro that[of a d b c ω2' ω1'] eq nz norms)
  next
    case False
    show ?thesis
    proof (intro that[of a "-d" b "-c" ω2' "-ω1'"])
      from False have "a * d - b * c = -1"
        using abcd by linarith
      thus "a * (-d) - b * (-c) = 1"
        by simp
    next
      show "ω2' = of_int a * ω2 + of_int b * ω1"
           "-ω1' = of_int (- c) * ω2 + of_int (- d) * ω1"
        using eq by (simp_all add: algebra_simps)
    qed (use norms nz in ‹auto simp: norm_minus_commute add.commute›)
  qed
qed
```

Theorem 2.2

```
lemma std_fund_region_fundamental_aux1:
```

```
  assumes "Im τ' > 0"
  obtains τ where "Im τ > 0" "τ ∼Γ τ'" "norm τ ≥ 1" "norm (τ + 1) ≥
norm τ" "norm (τ - 1) ≥ norm τ"
proof -
  interpret std_complex_lattice τ'
    using assms by unfold_locales (auto simp: complex_is_Real_iff)
  obtain ω1 ω2 a b c d
    where abcd: "a * d - b * c = 1"
      and eq: "ω2 = of_int a * τ' + of_int b * 1" "ω1 = of_int c * τ'
+ of_int d * 1"
      and nz: "Im (ω2 / ω1) ≠ 0"
      and norms: "norm ω1 ≤ norm ω2" "norm ω2 ≤ norm (ω1 + ω2)" "norm
ω2 ≤ norm (ω1 - ω2)"
    using std_fund_region_fundamental_lemma2 .
  from nz have [simp]: "ω1 ≠ 0" "ω2 ≠ 0"
    by auto

  interpret unimodular_moebius_transform a b c d
    by unfold_locales fact

  define τ where "τ = ω2 / ω1"
  have τ_eq: "τ = φ τ'"
    by (simp add: moebius_def τ_def eq add_ac φ_def)

  show ?thesis
  proof (rule that[of τ])
    show "Im τ > 0"
      using assms τ_eq by (simp add: Im_transform_pos)
  next
    show "norm τ ≥ 1" "norm (τ + 1) ≥ norm τ" "norm (τ - 1) ≥ norm
τ"
      using norms by (simp_all add: τ_def norm_divide field_simps norm_minus_commute)
  next
    have "τ = apply_modgrp as_modgrp τ'"
      using τ_eq by (simp add: φ_as_modgrp)
    thus "τ ∼Γ τ'" using ‹Im τ' > 0›
      by auto
  qed
qed

lemma std_fund_region_fundamental_aux2:
  assumes "norm (z + 1) ≥ norm z" "norm (z - 1) ≥ norm z"
  shows   "Re z ∈ {-1/2..1/2}"
proof -
  have "0 ≤ norm (z + 1) ^ 2 - norm z ^ 2"
    using assms by simp
  also have "... = (Re z + 1)^2 - (Re z)^2"
    unfolding norm_complex_def by simp
  also have "... = 1 + 2 * Re z"
```

```
    by (simp add: algebra_simps power2_eq_square)
  finally have "Re z ≥ -1/2"
    by simp

  have "0 ≤ norm (z - 1) ^ 2 - norm z ^ 2"
    using assms by simp
  also have "... = (Re z - 1)^2 - (Re z)^2"
    unfolding norm_complex_def by simp
  also have "... = 1 - 2 * Re z"
    by (simp add: algebra_simps power2_eq_square)
  finally have "Re z ≤ 1/2"
    by simp

  with ‹Re z ≥ -1/2› show ?thesis
    by simp
qed

lemma std_fund_region_fundamental_aux3:
  fixes x y :: complex
  assumes xy: "x ∈ \<R>\<^sub>\<Gamma>" "y ∈ \<R>\<^sub>\<Gamma>"
  assumes f: "y = apply_modgrp f x"
  defines "c ≡ modgrp_c f"
  defines "d ≡ modgrp_d f"
  assumes c: "c ≠ 0"
  shows   "Im y < Im x"
proof -
  have ineq1: "norm (c * x + d) ^ 2 > c ^ 2 - |c * d| + d ^ 2"
  proof -
    have "of_int |c| * 1 < of_int |c| * norm x"
      using xy c by (intro mult_strict_left_mono) (auto simp: std_fund_region_def)
    hence "of_int c ^ 2 < (of_int c * norm x) ^ 2"
      by (intro power2_strict_mono) auto
    also have "... - |c * d| * 1 + d ^ 2 ≤
          (of_int c * norm x) ^ 2 - |c * d| * (2 * |Re x|) + d ^ 2"
      using xy unfolding of_int_add of_int_mult of_int_power of_int_diff
      by (intro add_mono diff_mono mult_left_mono) (auto simp: std_fund_region_def)
    also have "... = c ^ 2 * norm x ^ 2 - |2 * c * d * Re x| + d ^ 2"
      by (simp add: power_mult_distrib abs_mult)
    also have "... ≤ c ^ 2 * norm x ^ 2 + 2 * c * d * Re x + d ^ 2"
      by linarith
    also have "... = norm (c * x + d) ^ 2"
      unfolding cmod_power2 by (simp add: algebra_simps power2_eq_square)
    finally show "norm (c * x + d) ^ 2 > c ^ 2 - |c * d| + d ^ 2"
      by simp
  qed

  have "Im y = Im x / norm (c * x + d) ^ 2"
    using f by (simp add: modgrp.Im_transform c_def d_def)
```

```
  also have "norm (c * x + d) ^ 2 > 1"
  proof (cases "d = 0")
    case [simp]: True
    have "0 < c ^ 2"
      using c by simp
    hence "1 ≤ real_of_int (c ^ 2) * 1"
      by linarith
    also have "... < of_int (c ^ 2) * norm x ^ 2"
      using xy c by (intro mult_strict_left_mono) (auto simp: std_fund_region_def)
    also have "... = norm (c * x + d) ^ 2"
      by (simp add: norm_mult power_mult_distrib)
    finally show ?thesis .
  next
    case False
    have "0 < |c * d|"
      using c False by auto
    hence "1 ≤ |c * d|"
      by linarith
    also have "... ≤ |c * d| + (|c| - |d|) ^ 2"
      by simp
    also have "... = c ^ 2 - |c * d| + d ^ 2"
      by (simp add: algebra_simps power2_eq_square abs_mult)
    finally have "1 ≤ real_of_int (c ^ 2 - |c * d| + d ^ 2)"
      by linarith
    also have "... < norm (c * x + d) ^ 2"
      using ineq1 False by simp
    finally show ?thesis .
  qed

  hence "Im x / norm (c * x + d) ^ 2 < Im x / 1"
    using xy by (intro divide_strict_left_mono) (auto simp: std_fund_region_def)
  finally show ?thesis
    by simp
qed

lemma std_fund_region_fundamental_aux4:
  fixes x y :: complex
  assumes xy: "x ∈ ℛ_Γ" "y ∈ ℛ_Γ"
  assumes f: "y = apply_modgrp f x"
  shows   "f = 1"
proof -
  define a where "a = modgrp_a f"
  define b where "b = modgrp_b f"
  define c where "c = modgrp_c f"
  define d where "d = modgrp_d f"

  have c: "c = 0"
  proof (rule ccontr)
    assume c: "c ≠ 0"
```

```
    have "Im y < Im x" using xy f c
      by (intro std_fund_region_fundamental_aux3[where f = f]) (auto
simp: c_def)
    moreover have "Im y > Im x"
    proof (rule std_fund_region_fundamental_aux3[where f = "inverse f"])
      have "Im x > 0"
        using xy by (auto simp: std_fund_region_def)
      hence "x ≠ pole_modgrp f"
        using pole_modgrp_in_Reals[of f, where ?'a = complex]
        by (auto simp: complex_is_Real_iff)
      with f have "apply_modgrp (inverse f) y = x"
        by (intro apply_modgrp_inverse_eqI) auto
      thus "x = apply_modgrp (inverse f) y" ..
    next
      have "is_singular_modgrp f"
        using c by (simp add: is_singular_modgrp_altdef c_def)
      hence "is_singular_modgrp (inverse f)"
        by simp
      thus "modgrp_c (inverse f) ≠ 0"
        unfolding is_singular_modgrp_altdef by simp
    qed (use xy c in ‹auto simp: c_def›)
    ultimately show False
      by simp
  qed

  define n where "n = sgn a * b"
  from c have "¬is_singular_modgrp f"
    by (auto simp: is_singular_modgrp_altdef c_def)
  hence f_eq: "f = shift_modgrp n"
    using not_is_singular_modgrpD[of f] by (simp add: n_def a_def b_def)
  from xy f have "n = 0"
    by (auto simp: std_fund_region_def f_eq)
  thus "f = 1"
    by (simp add: f_eq)
qed
```

Theorem 2.3

```
interpretation std_fund_region: fundamental_region UNIV std_fund_region
proof
  show "std_fund_region ⊆ {z. Im z > 0}"
    using Im_std_fund_region by blast
next
  fix x y :: complex
  assume xy: "x ∈ ℛ⇩Γ" "y ∈ ℛ⇩Γ" "x ∼⇩Γ y"
  then obtain f where f: "y = apply_modgrp f x"
    by (auto simp: modular_group.rel_def)
  with xy have "f = 1"
    using std_fund_region_fundamental_aux4 by blast
  with f xy show "x = y"
```

```
    by simp
next
  fix x :: complex
  assume x: "Im x > 0"
  obtain y where y: "Im y > 0" "y ∼Γ x"
    "norm y ≥ 1" "norm (y + 1) ≥ norm y" "norm (y - 1) ≥ norm y"
    using std_fund_region_fundamental_aux1[OF x] by blast
  from y have "Re y ∈ {-1/2..1/2}"
    by (intro std_fund_region_fundamental_aux2)
  with y show "∃y∈closure std_fund_region. x ∼Γ y"
    using x unfolding closure_std_fund_region by (auto simp: modular_group.rel_commutes)
qed auto

theorem std_fund_region_no_fixed_point:
  assumes "z ∈ RΓ"
  assumes "apply_modgrp f z = z"
  shows   "f = 1"
  using std_fund_region_fundamental_aux4[of z "apply_modgrp f z" f] assms
by auto

lemma std_fund_region_no_fixed_point':
  assumes "z ∈ RΓ"
  assumes "apply_modgrp f z = apply_modgrp g z"
  shows   "f = g"
proof -
  have z: "Im z > 0"
    using assms by (auto simp: in_std_fund_region_iff)
  have "apply_modgrp (inverse f) (apply_modgrp g z) = apply_modgrp (inverse
f) (apply_modgrp f z)"
    by (simp only: assms(2))
  also have "... = z"
    using z by (intro apply_modgrp_inverse_eqI) auto
  also have "apply_modgrp (inverse f) (apply_modgrp g z) = apply_modgrp
(inverse f * g) z"
    by (rule apply_modgrp_mult [symmetric]) (use z in auto)
  finally have "inverse f * g = 1"
    using assms by (intro std_fund_region_no_fixed_point) auto
  thus ?thesis
    by (metis modgrp.left_cancel modgrp.left_inverse)
qed

lemma equiv_point_in_std_fund_region':
  assumes "Im z > 0"
  obtains z' where "z ∼Γ z'" "z' ∈ RΓ'"
proof -
  obtain z1 where z1: "z ∼Γ z1" "z1 ∈ closure RΓ"
    using std_fund_region.equiv_in_closure assms by blast
  show ?thesis
  proof (cases "z1 ∈ RΓ'")
```

```
    case True
    thus ?thesis
      using z1 by (intro that[of z1]) auto
  next
    case False
    hence "z1 ∈ closure \<R>\<^sub>\<Gamma> - \<R>\<^sub>\<Gamma>'"
      using z1 by blast
    thus ?thesis
      unfolding closure_std_fund_region_minus_std_fund_region'
    proof
      assume z1': "z1 ∈ {z. cmod z = 1 ∧ 0 < Im z ∧ Re z ∈ {0<..1 /
2}}"
      define z2 where "z2 = apply_modgrp S_modgrp z1"
      show ?thesis
      proof (rule that [of z2])
        show "z ∼\<^sub>\<Gamma> z2"
          unfolding z2_def using z1
          by (subst modular_group.rel_apply_modgrp_right_iff) auto
        have "-cnj z1 ∈ \<R>\<^sub>\<Gamma>'"
          using z1' by (auto simp: z2_def in_std_fund_region'_iff)
        also have "-cnj z1 = z2"
          using z1' by (auto simp: z2_def divide_conv_cnj)
        finally show "z2 ∈ \<R>\<^sub>\<Gamma>'" .
      qed
    next
      assume z1': "z1 ∈ {z. Re z = 1 / 2 ∧ sqrt 3 / 2 ≤ Im z}"
      define z2 where "z2 = apply_modgrp (shift_modgrp (-1)) z1"
      show ?thesis
      proof (rule that [of z2])
        show "z ∼\<^sub>\<Gamma> z2"
          unfolding z2_def using z1
          by (subst modular_group.rel_apply_modgrp_right_iff) auto
        have "-cnj z1 ∈ \<R>\<^sub>\<Gamma>'"
          using z1' z1 by (auto simp: z2_def in_std_fund_region'_iff in_closure_std_fund_re
        also have "-cnj z1 = z2"
          using z1' by (auto simp: z2_def complex_eq_iff)
        finally show "z2 ∈ \<R>\<^sub>\<Gamma>'" .
      qed
    qed
  qed
qed
```

The image of the fundamental region under a unimodular transformation is again a fundamental region.

```
locale std_fund_region_image =
  fixes f :: modgrp and R :: "complex set"
  defines "R ≡ apply_modgrp f ` \<R>\<^sub>\<Gamma>"
begin
```

```
lemma R_altdef: "R = {z. Im z > 0} ∩ apply_modgrp (inverse f) -‘ 𝓡Γ"
  unfolding R_def
proof safe
  fix z assume z: "z ∈ 𝓡Γ"
  thus "Im (apply_modgrp f z) > 0"
    by (auto simp: Im_std_fund_region)
  have "apply_modgrp (inverse f) (apply_modgrp f z) ∈ 𝓡Γ"
    by (subst apply_modgrp_mult [symmetric]) (use z in auto)
  thus "apply_modgrp f z ∈ apply_modgrp (inverse f) -‘ 𝓡Γ"
    by (auto simp flip: apply_modgrp_mult)
next
  fix z assume z: "apply_modgrp (inverse f) z ∈ 𝓡Γ" "Im z > 0"
  have "z = apply_modgrp f (apply_modgrp (inverse f) z)"
    by (subst apply_modgrp_mult [symmetric]) (use z(2) in auto)
  with z show "z ∈ apply_modgrp f ‘ 𝓡Γ"
    by blast
qed

lemma R_altdef': "R = apply_modgrp (inverse f) -‘ 𝓡Γ"
  unfolding R_altdef by (auto simp: in_std_fund_region_iff)

sublocale fundamental_region UNIV R
proof
  show "open R"
    unfolding R_altdef
    by (intro continuous_open_preimage continuous_intros) (auto simp:
open_halfspace_Im_gt )
  show "R ⊆ {z. 0 < Im z}"
    unfolding R_altdef by auto
  show "x = y" if "x ∈ R" "y ∈ R" "x ∼Γ y" for x y
    using that unfolding R_def by (auto dest: std_fund_region.unique)
  show "∃y∈closure R. x ∼Γ y" if "Im x > 0" for x
  proof -
    define x' where "x' = apply_modgrp (inverse f) x"
    have x': "Im x' > 0"
      using that by (simp add: x'_def)
    then obtain y where y: "y ∈ closure 𝓡Γ" "x' ∼Γ y"
      using std_fund_region.equiv_in_closure[of x'] by blast
    define y' where "y' = apply_modgrp f y"
    have "y islimpt 𝓡Γ"
      using y by (meson islimpt_closure_open limpt_of_closure open_std_fund_region)
    then obtain h :: "nat ⇒ complex" where h: "⋀n. h n ∈ 𝓡Γ - {y}"
"h ⟶ y"
      unfolding islimpt_sequential by blast
    have "(apply_modgrp f ∘ h) n ∈ R - {y'}" for n
    proof -
      have Ims: "Im y > 0" "Im (h n) > 0"
        using y h(1)[of n] by (auto simp: in_std_fund_region_iff)
      have "apply_modgrp f (h n) ∈ R" "h n ≠ y"
```

```
      using h(1)[of n] by (auto simp: R_def)
    moreover have "apply_modgrp f (h n) ≠ y'"
      unfolding y'_def using y ‹h n ≠ y› Ims by (subst apply_modgrp_eq_iff)
auto
    ultimately show ?thesis
      by auto
  qed
  moreover have "(apply_modgrp f ∘ h) ⟶ y'"
    unfolding y'_def using y by (auto intro!: tendsto_compose_at[OF
h(2)] tendsto_eq_intros)+
  ultimately have "y' islimpt R"
    unfolding islimpt_sequential by blast
  hence "y' ∈ closure R"
    by (simp add: closure_def)

  moreover have "x ∼Γ y'"
    using x' that y unfolding y'_def x'_def
    by (auto simp: modular_group.rel_commutes)
  ultimately show ?thesis
    by blast
 qed
qed

end
```

4.4 The corner point of the standard fundamental region

The point $\rho = \exp(2/3\pi) = -\frac{1}{2} + \frac{\sqrt{3}}{2}i$ is the left corner of the standard fundamental region, and its reflection on the imaginary axis (which is the same as its image under $z \mapsto -1/z$) forms the right corner.

```
definition modfun_rho ("ϱ") where
  "ϱ = cis (2 / 3 * pi)"

lemma modfun_rho_altdef: "ϱ = -1 / 2 + sqrt 3 / 2 * i"
  by (simp add: complex_eq_iff modfun_rho_def Re_exp Im_exp sin_120 cos_120)

lemma Re_modfun_rho [simp]: "Re ϱ = -1 / 2"
  and Im_modfun_rho [simp]: "Im ϱ = sqrt 3 / 2"
  by (simp_all add: modfun_rho_altdef)

lemma norm_modfun_rho [simp]: "norm ϱ = 1"
  by (simp add: modfun_rho_def)

lemma modfun_rho_plus_1_eq: "ϱ + 1 = exp (pi / 3 * i)"
  by (simp add: modfun_rho_altdef complex_eq_iff Re_exp Im_exp sin_60
cos_60)

lemma norm_modfun_rho_plus_1 [simp]: "norm (ϱ + 1) = 1"
```

```
  by (simp add: modfun_rho_plus_1_eq)

lemma cnj_modfun_rho: "cnj ϱ = -ϱ - 1"
  and cnj_modfun_rho_plus1: "cnj (ϱ + 1) = -ϱ"
  by (auto simp: complex_eq_iff)

lemma modfun_rho_cube: "ϱ ^ 3 = 1"
  by (simp add: modfun_rho_def Complex.DeMoivre)

lemma modfun_rho_power_mod3_reduce: "ϱ ^ n = ϱ ^ (n mod 3)"
proof -
  have "ϱ ^ n = ϱ ^ (3 * (n div 3) + (n mod 3))"
    by simp
  also have "... = (ϱ ^ 3) ^ (n div 3) * ϱ ^ (n mod 3)"
    by (subst power_add) (simp add: power_mult)
  also have "... = ϱ ^ (n mod 3)"
    by (simp add: modfun_rho_cube)
  finally show ?thesis .
qed

lemma modfun_rho_power_mod3_reduce': "n ≥ 3 ⟹ ϱ ^ n = ϱ ^ (n mod 3)"
  by (rule modfun_rho_power_mod3_reduce)

lemmas [simp] = modfun_rho_power_mod3_reduce' [of "numeral num" for num]

lemma modfun_rho_square: "ϱ ^ 2 = -ϱ - 1"
  by (simp add: modfun_rho_altdef power2_eq_square field_simps flip: of_real_mult)

lemma modfun_rho_not_real [simp]: "ϱ ∉ ℝ"
  by (simp add: modfun_rho_altdef complex_is_Real_iff)

lemma modfun_rho_nonzero [simp]: "ϱ ≠ 0"
  by (simp add: modfun_rho_def)

lemma modfun_rho_not_one [simp]: "ϱ ≠ 1"
  by (simp add: complex_eq_iff modfun_rho_altdef)

lemma i_neq_modfun_rho [simp]: "i ≠ ϱ"
  and i_neq_modfun_rho_plus1 [simp]: "i ≠ ϱ + 1"
  and modfun_rho_neg_i [simp]: "ϱ ≠ i"
  and modfun_rho_plus1_neg_i [simp]: "ϱ + 1 ≠ i"
  by (auto simp: complex_eq_iff)

lemma i_in_closure_std_fund_region [intro, simp]: "i ∈ closure 𝓡Γ"
  and i_in_std_fund_region' [intro, simp]: "i ∈ 𝓡Γ'"
  and modfun_rho_in_closure_std_fund_region [intro, simp]: "ϱ ∈ closure
𝓡Γ"
  and modfun_rho_in_std_fund_region' [intro, simp]: "ϱ ∈ 𝓡Γ'"
  and modfun_rho_plus_1_notin_closure_std_fund_region [intro, simp]: "ϱ
```

```
+ 1 ∈ closure 𝓡Γ"
  and modfun_rho_plus_1_notin_std_fund_region' [intro, simp]: "ϱ + 1
∉ 𝓡Γ'"
  by (simp_all add: closure_std_fund_region std_fund_region'_def in_std_fund_region_iff)

lemma modfun_rho_power_eq_1_iff: "ϱ ^ n = 1 ⟷ 3 dvd n"
proof -
  have "ϱ ^ n = 1 ⟷ (∃k. real n = 3 * real_of_int k)"
    by (simp add: modfun_rho_def Complex.DeMoivre cis_eq_1_iff)
  also have "(λk. real n = 3 * real_of_int k) = (λk. int n = 3 * k)"
    by (rule ext) linarith
  also have "(∃k. int n = 3 * k) ⟷ 3 dvd n"
    by presburger
  finally show ?thesis .
qed
```

4.5 Fundamental regions for congruence subgroups

```
context hecke_prime_subgroup
begin

definition std_fund_region_cong ("𝓡") where
  "𝓡 = 𝓡Γ ∪ (⋃k∈{0..<p}. (λz. -1 / (z + of_int k)) ` 𝓡Γ)"

lemma std_fund_region_cong_altdef:
  "𝓡 = 𝓡Γ ∪ (⋃k∈{0..<p}. apply_modgrp (S_shift_modgrp k) ` 𝓡Γ)"
proof -
  have "apply_modgrp (S_shift_modgrp k) ` 𝓡Γ = (λz. -1 / (z + of_int
k)) ` 𝓡Γ" for k
    unfolding S_shift_modgrp_def
    by (intro image_cong refl, subst apply_modgrp_mult) auto
  thus ?thesis
    by (simp add: std_fund_region_cong_def)
qed

lemma closure_UN_finite: "finite A ⟹ closure (⋃A) = (⋃X∈A. closure
X)"
  by (induction A rule: finite_induct) auto


sublocale std_region: fundamental_region Γ' 𝓡
proof
  show "open 𝓡"
    unfolding std_fund_region_cong_altdef
    by (intro open_Un open_UN ballI open_std_fund_region apply_modgrp_open_map)
auto
next
  show "𝓡 ⊆ {z. Im z > 0}"
    by (auto simp: std_fund_region_cong_altdef in_std_fund_region_iff)
```

```
next
  fix x assume "Im x > 0"
  then obtain y where y: "y ∈ closure 𝒟Γ" "x ∼Γ y"
    using std_fund_region.equiv_in_closure by blast
  then obtain f where f: "y = apply_modgrp f x" "Im y > 0"
    by (auto simp: modular_group.rel_def)
  obtain g h where gh: "g ∈ Γ'" "h = 1 ∨ (∃k∈{0..<p}. h = S_shift_modgrp
k)" "inverse f = g * h"
    using modgrp_decompose'[of "inverse f"] .
  have inverse_g_eq: "inverse g = h * f"
    using gh(3) by (metis modgrp.assoc modgrp.inverse_unique modgrp.left_inverse)

  show "∃y∈closure 𝒟. rel x y"
    using gh(2)
  proof safe
    assume "h = 1"
    thus ?thesis using y f gh
      by (auto simp: std_fund_region_cong_altdef intro!: bexI[of _ y])
  next
    fix k assume k: "k ∈ {0..<p}" "h = S_shift_modgrp k"
    have "apply_modgrp h y ∈ apply_modgrp h ‘ closure 𝒟Γ"
      using y by blast
    also have "... ⊆ closure (apply_modgrp h ‘ 𝒟Γ)"
      by (intro continuous_image_closure_subset[of "{z. Im z > 0}"])
         (auto intro!: continuous_intros closure_std_fund_region_Im_pos)
    also have "apply_modgrp h y = apply_modgrp (inverse g) x"
      unfolding inverse_g_eq using ‹Im x > 0› f by (subst apply_modgrp_mult)
auto
    finally have "apply_modgrp (inverse g) x ∈ closure (apply_modgrp h
‘ 𝒟Γ)" .
    moreover have "rel x (apply_modgrp (inverse g) x)"
      using ‹Im x > 0› gh by auto
    ultimately show ?thesis
      unfolding std_fund_region_cong_altdef using k by (auto simp: closure_UN_finite)
  qed
next
  fix x y assume xy: "x ∈ 𝒟" "y ∈ 𝒟" "rel x y"
  define ST where "ST = (λk. apply_modgrp (S_shift_modgrp k) :: complex
⇒ complex)"

  have 1: False
    if xy: "x ∈ 𝒟Γ" "y ∈ 𝒟Γ" "rel x (ST k y)" and k: "k ∈ {0..<p}"
for x y k
  proof -
    have "x ∼Γ ST k y"
      using xy(3) by (rule rel_imp_rel)
    hence "x ∼Γ y"
      by (auto simp: ST_def)
    hence [simp]: "x = y"
```

```
    using xy(1,2) std_fund_region.unique by blast
  with xy(3) have "rel (ST k x) x"
    by (simp add: rel_commutes)
  then obtain f where f: "f ∈ Γ'" "apply_modgrp f (ST k x) = x" "Im
x > 0"
    unfolding rel_def by blast
  hence "apply_modgrp (f * S_shift_modgrp k) x = x"
    by (subst apply_modgrp_mult) (auto simp: ST_def)
  hence "f * S_shift_modgrp k = 1"
    using xy by (intro std_fund_region_no_fixed_point) auto
  hence "f = inverse (S_shift_modgrp k)"
    by (metis modgrp.inverse_inverse modgrp.inverse_unique)
  moreover have "modgrp_c (inverse (S_shift_modgrp k)) = 1"
    by (simp add: S_shift_modgrp_def S_modgrp_code shift_modgrp_code
inverse_modgrp_code
                  times_modgrp_code modgrp_c_code)
  moreover have "¬p dvd 1"
    using p_prime using not_prime_unit by blast
  ultimately show False
    using ‹f ∈ Γ'› unfolding subgrp_def modgrps_cong_altdef by auto
qed

have "x ∈ ℛΓ ∪ (⋃k∈{0..<p}. ST k ` ℛΓ)" "y ∈ ℛΓ ∪ (⋃k∈{0..<p}.
ST k ` ℛΓ)"
  using xy unfolding ST_def std_fund_region_cong_altdef by blast+
thus "x = y"
proof safe
  assume "x ∈ ℛΓ" "y ∈ ℛΓ"
  thus "x = y"
    using ‹rel x y› rel_imp_rel std_fund_region.unique by blast
next
  fix k y'
  assume "x ∈ ℛΓ" "k ∈ {0..<p}" "y' ∈ ℛΓ" "y = ST k y'"
  thus "x = ST k y'"
    using 1[of x y' k] ‹rel x y› by auto
next
  fix k x'
  assume "x' ∈ ℛΓ" "k ∈ {0..<p}" "y ∈ ℛΓ" "x = ST k x'"
  thus "ST k x' = y"
    using 1[of y x' k] ‹rel x y› by (auto simp: rel_commutes)
next
  fix x' y' :: complex and k1 k2 :: int
  assume xy': "x' ∈ ℛΓ" "y' ∈ ℛΓ" "x = ST k1 x'" "y = ST k2 y'"
  assume k12: "k1 ∈ {0..<p}" "k2 ∈ {0..<p}"
  have x': "Im x' > 0"
    using xy' by (auto simp: in_std_fund_region_iff)
  have "ST k1 x' ∼Γ ST k2 y'"
    using xy xy' by (intro rel_imp_rel) auto
  hence "x' ∼Γ y'"
```

```
      by (auto simp: ST_def)
    hence [simp]: "x' = y'"
      using xy' by (intro std_fund_region.unique)
    have "rel (ST k1 x') (ST k2 x')"
      using xy xy' by simp
    then obtain f where f: "f ∈ Γ'" "apply_modgrp f (ST k1 x') = ST
k2 y'"
      unfolding rel_def by auto
    note ‹apply_modgrp f (ST k1 x') = ST k2 y'›
    also have "apply_modgrp f (ST k1 x') = apply_modgrp (f * S_shift_modgrp
k1) x'"
      unfolding ST_def using x' by (subst apply_modgrp_mult) auto
    finally have "f * S_shift_modgrp k1 = S_shift_modgrp k2"
      unfolding ST_def using xy' by (intro std_fund_region_no_fixed_point'[of
x']) auto
    hence "f = S_shift_modgrp k2 * inverse (S_shift_modgrp k1)"
      by (metis modgrp.right_inverse modgrp.right_neutral mult.assoc)
    also have "... = S_modgrp * shift_modgrp (k2 - k1) * S_modgrp"
      using shift_modgrp_add[of k2 "-k1"]
      by (simp add: S_shift_modgrp_def modgrp.inverse_distrib_swap modgrp.assoc
            flip: shift_modgrp_minus)
    finally have "f = S_modgrp * shift_modgrp (k2 - k1) * S_modgrp" .
    moreover have "modgrp_c (S_modgrp * shift_modgrp (k2 - k1) * S_modgrp)
= |k2 - k1|"
      by (simp add: S_modgrp_code shift_modgrp_code times_modgrp_code
modgrp_c_code)
    moreover have "p dvd modgrp_c f"
      using f by (auto simp: subgrp_def modgrps_cong_altdef)
    ultimately have "p dvd |k2 - k1|"
      by simp
    moreover from k12 have "|k2 - k1| < p"
      by auto
    ultimately have "k1 = k2"
      using zdvd_not_zless[of "|k2 - k1|" p] by (cases "k1 = k2") auto
    thus "ST k1 x' = ST k2 y'"
      by simp
  qed
qed

end



bundle modfun_region_notation
begin
notation std_fund_region ("\<R>\<^sub>\<Gamma>")
notation modfun_rho ("\<varrho>")
end
```

unbundle *no modfun_region_notation*

end

5 Elliptic Functions

theory *Elliptic_Functions*
  imports *Complex_Lattices*
begin

5.1 Definition

In the context of a complex lattice Λ, a function is called *elliptic* if it is meromorphic and periodic w.r.t. the lattice.

locale *elliptic_function = complex_lattice_periodic ω1 ω2 f*
  for *ω1 ω2 :: complex* **and** *f :: "complex ⇒ complex" +*
  assumes *meromorphic: "f meromorphic_on UNIV"*

We call a function *nicely elliptic* if it additionally is nicely meromorphic, i.e. it has no removable singularities and returns 0 at each pole. It is easy to convert elliptic functions into nicely elliptic ones using the *remove_sings* operator and lift results from the nicely elliptic setting to the "regular" elliptic one.

locale *nicely_elliptic_function = complex_lattice_periodic ω1 ω2 f*
  for *ω1 ω2 :: complex* **and** *f :: "complex ⇒ complex" +*
  assumes *nicely_meromorphic: "f nicely_meromorphic_on UNIV"*

locale *elliptic_function_remove_sings = elliptic_function*
begin

sublocale *remove_sings: nicely_elliptic_function ω1 ω2 "remove_sings f"*
proof
  show *"remove_sings f nicely_meromorphic_on UNIV"*
    using *meromorphic* **by** *(rule remove_sings_nicely_meromorphic)*
  have *∗: "remove_sings f (z + ω) = remove_sings f z"* **if** *ω: "ω ∈ Λ"*
for *z ω*
  proof -
    have *"remove_sings f (z + ω) = remove_sings (λw. f (w + ω)) z"*
      by *(simp add: remove_sings_shift_0' add_ac)*
    also have *"(λw. f (w + ω)) = f"*
      by *(intro ext lattice_cong) (auto simp: rel_def ω)*
    finally show *?thesis* **.**
  qed
  show *"remove_sings f (z + ω1) = remove_sings f z"*
      "remove_sings f (z + ω2) = remove_sings f z" **for** *z*
    by *(rule ∗; simp; fail)+*

qed

end

context *elliptic_function*
begin

interpretation *elliptic_function_remove_sings* **..**

lemma *isolated_singularity [simp, singularity_intros]: "isolated_singularity_at f z"*
  using *meromorphic* **by** *(simp add: meromorphic_on_altdef)*

lemma *not_essential [simp, singularity_intros]: "not_essential f z"*
  using *meromorphic* **by** *(simp add: meromorphic_on_altdef)*

lemma *meromorphic' [meromorphic_intros]: "f meromorphic_on A"*
  by *(rule meromorphic_on_subset[OF meromorphic]) auto*

lemma *meromorphic'' [meromorphic_intros]:*
  assumes *"g analytic_on A"*
  shows *"(λx. f (g x)) meromorphic_on A"*
  by *(rule meromorphic_on_compose[OF meromorphic assms]) auto*

Due to the lattice-periodicity of `f`, its derivative, zeros, poles, multiplicities, and residues are also all lattice-periodic.

sublocale *zeros: complex_lattice_periodic ω1 ω2 "isolated_zero f"*
proof
  fix *z :: complex*
  have **: "isolated_zero f (z + c) ⟷ isolated_zero (λz. f (z + c)) z"* **for** *c*
    by *(simp add: isolated_zero_shift' add_ac)*
  from *this* **show** *"isolated_zero f (z + ω1) ⟷ isolated_zero f z"*
                "isolated_zero f (z + ω2) ⟷ isolated_zero f z"
    by *(simp_all add: f_periodic)*
qed

sublocale *poles: complex_lattice_periodic ω1 ω2 "is_pole f"*
proof
  fix *z :: complex*
  have **: "is_pole f (z + c) ⟷ is_pole (λz. f (z + c)) z"* **for** *c*
    by *(simp add: is_pole_shift_0' add_ac)*
  from *this* **show** *"is_pole f (z + ω1) ⟷ is_pole f z" "is_pole f (z + ω2) ⟷ is_pole f z"*
    by *(simp_all add: f_periodic)*
qed

sublocale *zorder: complex_lattice_periodic ω1 ω2 "zorder f"*

```
proof
  fix z :: complex
  have *: "zorder f (z + c) = zorder (λz. f (z + c)) z" for c
    by (simp add: zorder_shift' add_ac)
  from this show "zorder f (z + ω1) = zorder f z" "zorder f (z + ω2)
= zorder f z"
    by (simp_all add: f_periodic)
qed

sublocale deriv: complex_lattice_periodic ω1 ω2 "deriv f"
proof
  fix z :: complex
  have *: "deriv f (z + c) = deriv (λz. f (z + c)) z" for c
    by (simp add: deriv_shift_0' add_ac o_def)
  from this show "deriv f (z + ω1) = deriv f z" "deriv f (z + ω2) =
deriv f z"
    by (simp_all add: f_periodic)
qed

sublocale higher_deriv: complex_lattice_periodic ω1 ω2 "(deriv ^^ n)
f"
proof
  fix z :: complex
  have *: "(deriv ^^ n) f (z + c) = (deriv ^^ n) (λz. f (z + c)) z" for
c
    by (simp add: higher_deriv_shift_0' add_ac o_def)
  from this show "(deriv ^^ n) f (z + ω1) = (deriv ^^ n) f z"
                 "(deriv ^^ n) f (z + ω2) = (deriv ^^ n) f z"
    by (simp_all add: f_periodic)
qed

sublocale residue: complex_lattice_periodic ω1 ω2 "residue f"
proof
  fix z :: complex
  have *: "residue f (z + c) = residue (λz. f (z + c)) z" for c
    by (simp add: residue_shift_0' add_ac o_def)
  from this show "residue f (z + ω1) = residue f z" "residue f (z +
ω2) = residue f z"
    by (simp_all add: f_periodic)
qed

lemma eventually_remove_sings_eq: "eventually (λw. remove_sings f w =
f w) (cosparse UNIV)"
  by (simp add: eventually_remove_sings_eq meromorphic)

lemma eventually_remove_sings_eq': "eventually (λw. remove_sings f w
= f w) (at z)"
  using eventually_remove_sings_eq by (simp add: eventually_cosparse_open_eq)
```

```
lemma isolated_zero_analytic_iff:
  assumes "f analytic_on {z}" "¬(∀≈z∈UNIV. f z = 0)"
  shows   "isolated_zero f z ⟷ f z = 0"
proof
  assume "isolated_zero f z"
  thus "f z = 0"
    using assms(1) zero_isolated_zero_analytic by blast
next
  assume "f z = 0"
  hence "f −z→ 0"
    using assms(1) by (metis analytic_at_imp_isCont continuous_at)

  have "eventually (λz. remove_sings f z ≠ 0) (at z)"
  proof (rule ccontr)
    assume "¬eventually (λz. remove_sings f z ≠ 0) (at z)"
    hence "frequently (λz. remove_sings f z = 0) (at z)"
      by (auto simp: frequently_def)
    hence "remove_sings f z = 0" for z
      using remove_sings.nicely_meromorphic by (rule frequently_eq_meromorphic_imp_constant
auto
    with eventually_remove_sings_eq show False
      using assms(2) by simp
  qed
  hence "eventually (λz. f z ≠ 0) (at z)"
    using eventually_remove_sings_eq'[of z] by eventually_elim auto
  with ‹f −z→ 0› show "isolated_zero f z"
    by (auto simp: isolated_zero_def)
qed

end


context nicely_elliptic_function
begin

lemma nicely_meromorphic' [meromorphic_intros]: "f nicely_meromorphic_on
A"
  by (rule nicely_meromorphic_on_subset[OF nicely_meromorphic]) auto

lemma analytic:
  assumes "⋀z. z ∈ A ⟹ ¬is_pole f z"
  shows   "f analytic_on A"
  using nicely_meromorphic_on_subset[OF nicely_meromorphic]
  by (rule nicely_meromorphic_without_singularities) (use assms in auto)

lemma holomorphic:
  assumes "⋀z. z ∈ A ⟹ ¬is_pole f z"
  shows   "f holomorphic_on A"
  using analytic by (rule analytic_imp_holomorphic) (use assms in blast)
```

```
lemma continuous_on:
  assumes "⋀z. z ∈ A ⟹ ¬is_pole f z"
  shows   "continuous_on A f"
  using holomorphic by (rule holomorphic_on_imp_continuous_on) (use assms
in blast)

sublocale elliptic_function ω1 ω2 f
proof
  show "f meromorphic_on UNIV"
    using nicely_meromorphic unfolding nicely_meromorphic_on_def by blast
qed

lemma analytic_at_iff_not_pole: "f analytic_on {z} ⟷ ¬is_pole f z"
  using analytic analytic_at_imp_no_pole by blast

lemma constant_or_avoid: "f = (λ_. c) ∨ (∀≈z∈UNIV. f z ≠ c)"
  using nicely_meromorphic_imp_constant_or_avoid[OF nicely_meromorphic,
of c] by auto

lemma isolated_zero_iff:
  assumes "f ≠ (λ_. 0)"
  shows   "isolated_zero f z ⟷ ¬is_pole f z ∧ f z = 0"
proof (cases "is_pole f z")
  case True
  thus ?thesis using pole_is_not_zero[of f z]
    by auto
next
  case False
  have "¬(∀≈z∈UNIV. f z = 0)"
  proof
    assume "∀≈z∈UNIV. f z = 0"
    moreover have "∀≈z∈UNIV. f z ≠ 0"
      using constant_or_avoid[of 0] assms by auto
    ultimately have "∀≈z∈(UNIV :: complex set). False"
      by eventually_elim auto
    thus False
      by simp
  qed
  moreover have "f analytic_on {z}"
    using False by (subst analytic_at_iff_not_pole)
  ultimately have "isolated_zero f z ⟷ f z = 0"
    by (subst isolated_zero_analytic_iff) auto
  thus ?thesis
    using False by simp
qed

end
```

5.2 Basic results about zeros and poles

In this section we will show that an elliptic function has the same number of poles in any period parallelogram. This number is called its *order*. Then we will show that the number of zeros in a period parallelogram is also equal to its order, and that there are no elliptic functions with order *1* and no non-constant elliptic functions with order *0*.

```
context elliptic_function
begin
```

Due to its meromorphicity and the fact that the period parallelograms are bounded, an elliptic function can only have a finite number of poles and zeros in a period parallelogram.

```
lemma finite_poles_in_parallelogram: "finite {z∈period_parallelogram
orig. is_pole f z}"
proof (rule finite_subset)
  show "finite (closure (period_parallelogram orig) ∩ {z. is_pole f z})"
  proof (rule finite_not_islimpt_in_compact)
    show "¬z islimpt {z. is_pole f z}" for z
      by (meson UNIV_I meromorphic meromorphic_on_isolated_singularity
          meromorphic_on_subset not_islimpt_poles subsetI)
  qed auto
qed (use closure_subset in auto)

lemma finite_zeros_in_parallelogram: "finite {z∈period_parallelogram
orig. isolated_zero f z}"
proof (rule finite_subset)
  show "finite (closure (period_parallelogram orig) ∩ {z. isolated_zero
f z})"
  proof (rule finite_not_islimpt_in_compact)
    show "¬z islimpt {z. isolated_zero f z}" for z
      using meromorphic_on_imp_not_zero_cosparse[OF meromorphic]
      by (auto simp: eventually_cosparse_open_eq islimpt_iff_eventually)
  qed auto
qed (use closure_subset in auto)
```

The *order* of an elliptic function is the number of its poles inside a period parallelogram, with multiplicity taken into account. We will later show that this is also the number of zeros.

```
definition (in complex_lattice) elliptic_order :: "(complex ⇒ complex)
⇒ nat" where
  "elliptic_order f = (∑z | z ∈ period_parallelogram 0 ∧ is_pole f z.
nat (-zorder f z))"

lemma elliptic_order_const [simp]: "elliptic_order (λx. c) = 0"
  by (simp add: elliptic_order_def)

lemma poles_eq_elliptic_order:
```

```
  "(∑z | z ∈ period_parallelogram orig ∧ is_pole f z. nat (-zorder f
z)) = elliptic_order f"
proof -
  define h where "h = to_fund_parallelogram"
  define P where "P = period_parallelogram"
  have "(∑z ∈ {z∈P orig. is_pole f z}. nat (-zorder f (h z))) =
        (∑z ∈ {z∈P 0. is_pole f z}. nat (-zorder f z))"
  proof (rule sum.reindex_bij_betw)
    show "bij_betw h {z ∈ P orig. is_pole f z} {z ∈ P 0. is_pole f z}"
    proof (rule bij_betw_Collect)
      show "bij_betw h (P orig) (P 0)"
        unfolding h_def P_def by (rule bij_betw_to_fund_parallelogram)
    next
      fix z
      show "is_pole f (h z) ⟷ is_pole f z"
        unfolding h_def by (simp add: poles.lattice_cong)
    qed
  qed
  also have "(∑z ∈ {z∈P orig. is_pole f z}. nat (-zorder f (h z))) =
             (∑z ∈ {z∈P orig. is_pole f z}. nat (-zorder f z))"
    using zorder.lattice_cong[of "h z" z for z] by (simp add: h_def)
  finally show ?thesis
    by (simp add: elliptic_order_def P_def)
qed

end


context nicely_elliptic_function
begin
```

The order of a (nicely) elliptic function is zero iff it is constant. We will later lift this to non-nicely elliptic functions, where we get that the order is zero iff the function is *mostly* constant (i.e. constant except for a sparse set).

In combination with our other results relating `elliptic_order` to the number of zeros and poles inside period parallelograms, this corresponds to Theorems 1.4 and 1.5 in Apostol's book.

```
lemma elliptic_order_eq_0_iff: "elliptic_order f = 0 ⟷ f constant_on
UNIV"
proof
  assume "f constant_on UNIV"
  then obtain c where f_eq: "f = (λ_. c)"
    by (auto simp: constant_on_def)
  have [simp]: "¬is_pole f z" for z
    unfolding f_eq by auto
  show "elliptic_order f = 0"
    by (simp add: elliptic_order_def)
```

```
next
  assume "elliptic_order f = 0"
  define P where "P = period_parallelogram 0"
  have "eventually (λz. ¬is_pole f z) (cosparse UNIV)"
    using meromorphic by (rule meromorphic_on_imp_not_pole_cosparse)
  hence "{z. is_pole f z} sparse_in UNIV"
    by (simp add: eventually_cosparse)
  hence fin: "finite (closure P ∩ {z. is_pole f z})"
    by (intro sparse_in_compact_finite) (use sparse_in_subset in ‹auto
simp: P_def›)

  from ‹elliptic_order f = 0› have "∀z∈{z∈P. is_pole f z}. zorder f
z ≥ 0"
    unfolding elliptic_order_def
  proof (subst (asm) sum_nonneg_eq_0_iff)
    show "finite {z ∈ period_parallelogram 0. is_pole f z}"
      by (rule finite_subset[OF _ fin]) (use closure_subset in ‹auto simp:
P_def›)
  qed (auto simp: P_def)
  moreover have "∀z∈{z∈P. is_pole f z}. zorder f z < 0"
  proof safe
    fix z assume "is_pole f z"
    have "isolated_singularity_at f z"
      using meromorphic by (simp add: meromorphic_on_altdef)
    with ‹is_pole f z› show "zorder f z < 0"
      using isolated_pole_imp_neg_zorder by blast
  qed
  ultimately have no_poles_P: "¬is_pole f z" if "z ∈ P" for z
    using that by force

  have no_poles [simp]: "¬is_pole f z" for z
  proof -
    have "¬is_pole f (to_fund_parallelogram z)"
      by (intro no_poles_P) (auto simp: P_def)
    also have "is_pole f (to_fund_parallelogram z) ⟷ is_pole f z"
      using poles.lattice_cong rel_to_fund_parallelogram_left by blast
    finally show ?thesis .
  qed

  have ana: "f analytic_on UNIV"
    using nicely_meromorphic by (rule nicely_meromorphic_without_singularities)
auto
  have "continuous_on A f" for A
    by (intro analytic_imp_holomorphic holomorphic_on_imp_continuous_on
analytic_on_subset[OF ana])
       auto
  hence "compact (f ‘ closure P)"
    by (rule compact_continuous_image) (auto simp: P_def)
  hence "bounded (f ‘ closure P)"
```

```
    by (rule compact_imp_bounded)
  hence "bounded (f ‘ P)"
    by (rule bounded_subset) (use closure_subset in auto)
  also have "f ‘ P = f ‘ UNIV"
  proof safe
    fix z show "f z ∈ f ‘ P"
      by (rule image_eqI[of _ _ "to_fund_parallelogram z"]) (auto simp:
P_def lattice_cong)
  qed auto
  finally show "f constant_on UNIV"
    by (intro Liouville_theorem) (auto intro!: analytic_imp_holomorphic
ana)
qed

lemma order_pos_iff: "elliptic_order f > 0 ⟷ ¬f constant_on UNIV"
  using elliptic_order_eq_0_iff by linarith
```

The following lemma allows us to evaluate an integral of the form $\int_P h(w)f'(w)/f(w)\,\mathrm{d}w$ more easily, where P is the path along the border of a period parallelogram.

Note that this only works if there are no zeros or pole on the border of the parallelogram.

```
lemma argument_principle_f_gen:
  fixes orig :: complex
  defines "γ ≡ parallelogram_path orig ω1 ω2"
  assumes h: "h holomorphic_on UNIV"
  assumes nz: "⋀z. z ∈ path_image γ ⟹ f z ≠ 0 ∧ ¬is_pole f z"
  shows "contour_integral γ (λx. h x * deriv f x / f x) =
           contour_integral (linepath orig (orig + ω1))
             (λz. (h z - h (z + ω2)) * deriv f z / f z) -
           contour_integral (linepath orig (orig + ω2))
             (λz. (h z - h (z + ω1)) * deriv f z / f z)"
proof -
  define h' where "h' = (λx. h (x + orig) * deriv f (x + orig) / f (x
+ orig))"
  have [analytic_intros]: "h analytic_on A" for A
    using h analytic_on_holomorphic by blast
  have f_analytic: "f analytic_on A" if "⋀z. z ∈ A ⟹ ¬is_pole f z"
for A
    by (rule nicely_meromorphic_without_singularities)
       (use that in ‹auto intro!: nicely_meromorphic_on_subset[OF nicely_meromorphic]›)

  have "contour_integral γ (λx. h x * deriv f x / f x) =
          contour_integral (linepath 0 ω1) (λx. h' x - h' (x + ω2)) -
          contour_integral (linepath 0 ω2) (λx. h' x - h' (x + ω1))"
(is "_ = ?rhs")
    unfolding γ_def
  proof (subst contour_integral_parallelogram_path'; (fold γ_def)?)
    have "(λx. h x * deriv f x / f x) analytic_on {z. f z ≠ 0 ∧ ¬is_pole
f z}"
```

```
      by (auto intro!: analytic_intros f_analytic)
    hence "(λx. h x * deriv f x / f x) analytic_on path_image γ"
      by (rule analytic_on_subset) (use nz in auto)
    thus "continuous_on (path_image γ) (λx. h x * deriv f x / f x)" us-
ing nz
      by (intro continuous_intros holomorphic_on_imp_continuous_on analytic_imp_holomorphic
  next
    have 1: "linepath orig (orig + ω1) = (+) orig ∘ linepath 0 ω1"
     and 2: "linepath orig (orig + ω2) = (+) orig ∘ linepath 0 ω2"
      by (auto simp: linepath_translate add_ac)
    show "contour_integral (linepath orig (orig + ω1))
            (λx. h x * deriv f x / f x - h (x + ω2) * deriv f (x + ω2)
/ f (x + ω2)) -
          contour_integral (linepath orig (orig + ω2))
            (λx. h x * deriv f x / f x - h (x + ω1) * deriv f (x + ω1)
/ f (x + ω1)) = ?rhs"
      unfolding 1 2 contour_integral_translate h'_def by (simp add: algebra_simps)
  qed

  also have "(λz. h' z - h' (z + ω1)) =
             (λz. (h (z + orig) - h (z + orig + ω1)) * deriv f (z + orig)
/ f (z + orig))"
    (is "?lhs = ?rhs")
  proof
    fix z :: complex
    have "h' z - h' (z + ω1) =
            h (z + orig) * deriv f (z + orig) / f (z + orig) -
            h (z + orig + ω1) * deriv f (z + orig + ω1) / f (z + orig
+ ω1)"
      by (simp add: h'_def add_ac)
    also have "... = (h (z + orig) - h (z + orig + ω1)) * deriv f (z +
orig) / f (z + orig)"
      unfolding f_periodic deriv.f_periodic by (simp add: diff_divide_distrib
ring_distribs)
    finally show "?lhs z = ?rhs z" .
  qed
  also have "contour_integral (linepath 0 ω2) ... =
             contour_integral ((+) orig ∘ linepath 0 ω2) (λz. (h z -
h (z + ω1)) * deriv f z / f z)"
    by (rule contour_integral_translate [symmetric])
  also have "(+) orig ∘ linepath 0 ω2 = linepath orig (orig + ω2)"
    by (subst linepath_translate) (simp_all add: add_ac)

  also have "contour_integral (linepath 0 ω1) (λz. h' z - h' (z + ω2))
=
             contour_integral (linepath 0 ω1)
               (λz. (h (z + orig) - h (z + orig + ω2)) * deriv f (z +
orig) / f (z + orig))"
    (is "contour_integral _ ?lhs = contour_integral _ ?rhs")
```

```
  proof (rule contour_integral_cong)
    fix z :: complex
    assume z: "z ∈ path_image (linepath 0 ω1)"
    hence "orig + z ∈ path_image ((+) orig ∘ linepath 0 ω1)"
      by (subst path_image_compose) auto
    also have "(+) orig ∘ linepath 0 ω1 = linepath orig (orig + ω1)"
      by (subst linepath_translate) (auto simp: add_ac)
    finally have "orig + z ∈ path_image γ"
      unfolding γ_def parallelogram_path_def by (auto simp: path_image_join)
    hence nz': "f (orig + z) ≠ 0"
      using nz by blast

    have "h' z - h' (z + ω2) =
            h (z + orig) * deriv f (z + orig) / f (z + orig) -
            h (z + orig + ω2) * (deriv f (z + orig + ω2) / f (z + orig
+ ω2))"
      by (simp add: h'_def add_ac)
    also have "deriv f (z + orig + ω2) / f (z + orig + ω2) =
               deriv f (z + orig) / f (z + orig)"
      unfolding f_periodic deriv.f_periodic using nz'
      by (auto simp: divide_simps add_ac)
    also have "h (z + orig) * deriv f (z + orig) / f (z + orig) - h (z
+ orig + ω2) * ... = ?rhs z"
      by (simp add: ring_distribs diff_divide_distrib)
    finally show "?lhs z = ?rhs z" .
  qed auto
  also have "... = contour_integral ((+) orig ∘ linepath 0 ω1)
                     (λz. (h z - h (z + ω2)) * deriv f z / f z)"
    unfolding contour_integral_translate by (simp add: add_ac)
  also have "(+) orig ∘ linepath 0 ω1 = linepath orig (orig + ω1)"
    by (subst linepath_translate) (simp_all add: add_ac)

  finally show ?thesis .
qed
```

Using our lemma with $h(z) = 1$, we immediately get the fact that the integral over $f'(z)/f(z)$ vanishes.

```
lemma argument_principle_f_1:
  fixes orig :: complex
  defines "γ ≡ parallelogram_path orig ω1 ω2"
  assumes nz: "⋀z. z ∈ path_image γ ⟹ f z ≠ 0 ∧ ¬is_pole f z"
  shows "contour_integral (parallelogram_path orig ω1 ω2) (λx. deriv
f x / f x) = 0"
  using argument_principle_f_gen[of "λ_. 1" orig] nz by (simp add: γ_def)
```

Using our lemma with $h(z) = z$, we see that the integral over $zf'(z)/f(z)$ does not vanish, but it is of the form $2\pi i\omega$, where $\omega \in \Lambda$.

```
lemma argument_principle_f_z:
  fixes orig :: complex
```

```
  defines "γ ≡ parallelogram_path orig ω1 ω2"
  assumes wf: "⋀z. z ∈ path_image γ ⟹ f z ≠ 0 ∧ ¬is_pole f z"
  shows "contour_integral γ (λz. z * deriv f z / f z) / (2*pi*i) ∈ Λ"
proof -
  note [holomorphic_intros del] = holomorphic_deriv
  note [holomorphic_intros] = holomorphic holomorphic_on_subset[OF holomorphic_deriv[of
_ UNIV]]
  define γ1 where "γ1 = linepath orig (orig + ω1)"
  define γ2 where "γ2 = linepath orig (orig + ω2)"
  define γ1' where "γ1' = f ∘ γ1"
  define γ2' where "γ2' = f ∘ γ2"
  have path_image_subset: "path_image γ1 ⊆ path_image γ" "path_image
γ2 ⊆ path_image γ"
    by (auto simp: γ_def γ1_def γ2_def path_image_join closed_segment_commute
parallelogram_path_def)

  have "pathstart γ ∈ path_image γ"
    by blast
  from wf[OF this] have [simp]: "f orig ≠ 0"
    by (simp add: γ_def)
  have [simp, intro]: "valid_path γ1" "valid_path γ2"
    by (simp_all add: γ1_def γ2_def)
  have [simp, intro]: "valid_path γ1'" "valid_path γ2'"
    unfolding γ1'_def γ2'_def using wf path_image_subset
    by (auto intro!: valid_path_compose_analytic[of _ _ "path_image γ"]
analytic)
  have [simp]: "pathstart γ1' = f orig" "pathfinish γ1' = f (orig + ω1)"
               "pathstart γ2' = f orig" "pathfinish γ2' = f (orig + ω2)"
    by (simp_all add: γ1'_def γ1_def γ2'_def γ2_def pathstart_compose
pathfinish_compose)

  have wf': "f z ≠ 0 ∧ ¬is_pole f z" if "z ∈ path_image γ1 ∪ path_image
γ2" for z
  proof -
    note ‹z ∈ path_image γ1 ∪ path_image γ2›
    also have "path_image γ1 ∪ path_image γ2 ⊆ path_image γ"
      using path_image_subset by blast
    finally show ?thesis
      using wf[of z] by blast
  qed
  have [simp]: "0 ∉ path_image γ1'" "0 ∉ path_image γ2'"
    using wf' by (auto simp: γ1'_def γ2'_def path_image_compose)
```

The actual proof starts here.

```
  define I1 where "I1 = contour_integral γ1"
  define I2 where "I2 = contour_integral γ2"

  have "winding_number γ1' 0 ∈ ℤ"
    by (rule integer_winding_number) (auto intro!: valid_path_imp_path
```

```
simp: f_periodic)
  then obtain m where m: "winding_number γ1' 0 = of_int m"
    by (elim Ints_cases)

  have "winding_number γ2' 0 ∈ ℤ"
    by (rule integer_winding_number) (auto intro!: valid_path_imp_path
simp: f_periodic)
  then obtain n where n: "winding_number γ2' 0 = of_int n"
    by (elim Ints_cases)

  have "contour_integral γ (λz. z * deriv f z / f z) / (2*pi*i) =
         (I1 (λz. (-ω2) * (deriv f z / f z)) - I2 (λz. (-ω1) * (deriv
f z / f z))) / (2*pi*i)"
    unfolding γ_def I1_def I2_def γ1_def γ2_def using wf
    by (subst argument_principle_f_gen) (auto simp: diff_divide_distrib
γ_def)
  also have "I1 (λz. (-ω2) * (deriv f z / f z)) - I2 (λz. (-ω1) * (deriv
f z / f z)) =
             (-ω2) * I1 (λz. deriv f z / f z) - (-ω1) * I2 (λz. deriv
f z / f z)"
    using wf' unfolding I1_def I2_def γ1_def γ2_def
    by (subst (1 2) contour_integral_lmul)
       (auto intro!: contour_integrable_continuous_linepath holomorphic_on_imp_continuous_o
                     analytic_imp_holomorphic analytic_intros analytic)
  also have "... = ω1 * contour_integral γ2' (λz. 1 / z) -
                   ω2 * contour_integral γ1' (λz. 1 / z)"
    unfolding I1_def I2_def γ1'_def γ2'_def using wf'
    by (subst (1 2) contour_integral_comp_analyticW[OF analytic[of "path_image
γ1 ∪ path_image γ2"]])
       (auto simp: γ1_def)
  also have "... / (2 * pi * i) = ω1 * winding_number γ2' 0 - ω2 * winding_number
γ1' 0"
    by (subst (1 2) winding_number_valid_path)
       (auto intro!: valid_path_compose_analytic analytic simp: diff_divide_distrib)
  also have "... = of_int n * ω1 - of_int m * ω2"
    by (auto simp: m n)
  also have "... ∈ Λ"
    by (auto intro!: lattice_intros)
  finally show ?thesis .
qed
```

By using the fact that the integral $f'(z)/f(z)$ along the border of a period parallelogram vanishes, we get the following fact: The number of zeros in the period parallelogram equals the number of poles, i.e. the order.

The only difficulty left here is to show that 1. the number of zeros is invariant under which period parallelogram we choose, and 2. there is a period parallelogram whose borders do not contain any zeros or poles.

This is essentially Theorem 1.8 in Apostol's book.

```
lemma zeros_eq_elliptic_order_aux:
  "(∑z | z ∈ period_parallelogram orig ∧ isolated_zero f z. nat (zorder
f z)) = elliptic_order f"
proof (cases "f = (λ_. 0)")
  case True
  hence "elliptic_order f = 0"
    by (subst elliptic_order_eq_0_iff) (auto simp: constant_on_def)
  moreover have "¬isolated_zero f z" for z
    by (simp add: isolated_zero_def True)
  ultimately show ?thesis
    by simp
next
  case False
  define s where "s = complex_of_real (sgn (Im (ω2 / ω1)))"
  have [simp]: "s ≠ 0"
    using fundpair by (auto simp: s_def sgn_if fundpair_def complex_is_Real_iff)

  have ev_nonzero: "eventually (λz. f z ≠ 0) (cosparse UNIV)"
    using nicely_meromorphic False nicely_meromorphic_imp_constant_or_avoid[of
f UNIV 0]
    by auto
  have isolated_zero_iff: "isolated_zero f z ⟷ ¬is_pole f z ∧ f z
= 0" for z
    using isolated_zero_iff[OF False] by blast

  define P where "P = period_parallelogram"
  define avoid where "avoid = {z. isolated_zero f z ∨ is_pole f z}"
  have sparse_avoid: "¬z islimpt avoid" for z
  proof -
    have "∀≈z∈UNIV. ¬isolated_zero f z ∧ ¬is_pole f z" using meromorphic

      by (intro meromorphic_on_imp_not_zero_cosparse
                meromorphic_on_imp_not_pole_cosparse eventually_conj)
    hence "∀≈z∈UNIV. z ∉ avoid"
      by eventually_elim (auto simp: avoid_def)
    thus ?thesis
      by (auto simp: eventually_cosparse_open_eq islimpt_iff_eventually)
  qed
  thus ?thesis
    unfolding P_def [symmetric]
  proof (induction orig rule: shifted_period_parallelogram_avoid_wlog)
    case (shift orig d)
    obtain h where h: "bij_betw h (P (orig + d)) (P orig)" "⋀z. rel
(h z) z"
      using bij_betw_period_parallelograms unfolding P_def by blast
    have "(∑z | z ∈ P (orig + d) ∧ isolated_zero f z. nat (zorder f
(h z))) =
          (∑z | z ∈ P orig ∧ isolated_zero f z. nat (zorder f z))"
      by (rule sum.reindex_bij_betw, rule bij_betw_Collect[OF h(1)])
```

```
        (auto simp: zeros.lattice_cong[OF h(2)])
    also have "(∑z | z ∈ P (orig + d) ∧ isolated_zero f z. nat (zorder
f (h z))) =
               (∑z | z ∈ P (orig + d) ∧ isolated_zero f z. nat (zorder
f z))"
      by (simp add: zorder.lattice_cong[OF h(2)])
    finally show ?case
      using shift by simp
  next
    case (avoid orig)
    define γ where "γ = parallelogram_path orig ω1 ω2"
    define zeros where "zeros = {z∈P orig. isolated_zero f z}"
    define poles where "poles = {z∈P orig. is_pole f z}"
    have γ: "¬isolated_zero f z ∧ ¬is_pole f z" if "z ∈ path_image γ"
for z
      using avoid that by (auto simp: avoid_def γ_def)
    have "compact (closure (P orig))"
      unfolding P_def by auto
    then obtain R where R: "closure (P orig) ⊆ ball 0 R"
      using compact_imp_bounded bounded_subset_ballD by blast
    define A :: "complex set" where "A = ball 0 R"
    have A: "closure (P orig) ⊆ A"
      using R by (simp add: A_def)

    have fin: "finite {w ∈ A. f w = 0 ∨ w ∈ {z. is_pole f z}}"
    proof (rule finite_subset)
      show "finite (cball 0 R ∩ avoid)"
        by (rule finite_not_islimpt_in_compact) (use sparse_avoid in auto)
    next
      show "{w ∈ A. f w = 0 ∨ w ∈ {z. is_pole f z}} ⊆ cball 0 R ∩ avoid"
        by (auto simp: avoid_def A_def isolated_zero_iff)
    qed

    have "contour_integral γ (λx. deriv f x * 1 / f x) =
          of_real (2 * pi) * i * (∑p∈{w∈A. f w = 0 ∨ w ∈ {z. is_pole
f z}}.
            winding_number γ p * 1 * of_int (zorder f p))"
    proof (rule argument_principle)
      show "open A" "connected A"
        by (auto simp: A_def)
    next
      have "f analytic_on A - {z. is_pole f z}"
        using nicely_meromorphic_on_subset[OF nicely_meromorphic]
        by (rule nicely_meromorphic_without_singularities) auto
      thus "f holomorphic_on A - {z. is_pole f z}"
        by (rule analytic_imp_holomorphic)
    next
      show "path_image γ ⊆ A - {w ∈ A. f w = 0 ∨ w ∈ {z. is_pole f
z}}"
```

```
      using A γ path_image_parallelogram_subset_closure[of orig]
      by (auto simp: γ_def P_def isolated_zero_iff)
  next
    show "finite {w ∈ A. f w = 0 ∨ w ∈ {z. is_pole f z}}"
      by (rule finite_subset[OF _ fin]) auto
  next
    show "∀z. z ∉ A ⟶ winding_number γ z = 0"
      using A unfolding γ_def P_def by (auto intro!: winding_number_parallelogram_outside
  qed (auto simp: γ_def)
  also have "contour_integral γ (λx. deriv f x * 1 / f x) = 0"
    using argument_principle_f_1[of orig] γ by (auto simp: γ_def isolated_zero_iff)
  finally have "(∑z∈{z∈A. f z = 0 ∨ is_pole f z}. winding_number γ
z * of_int (zorder f z)) = 0"
    by simp

  also have "(∑z∈{z∈A. f z = 0 ∨ is_pole f z}. winding_number γ z
* of_int (zorder f z)) =
              (∑z∈{z∈P orig. f z = 0 ∨ is_pole f z}. s * of_int (zorder
f z))"
  proof (intro sum.mono_neutral_cong_right ballI, goal_cases)
    case (3 z)
    hence "z ∉ P orig ∪ frontier (P orig)"
      using γ[of z] by (auto simp: isolated_zero_iff P_def γ_def path_image_parallelogram
    also have "P orig ∪ frontier (P orig) = closure (P orig)"
      using closure_Un_frontier by blast
    finally have "winding_number γ z = 0"
      unfolding γ_def P_def using winding_number_parallelogram_outside
by blast
    thus ?case
      by simp
  next
    case (4 z)
    hence "z ∈ P orig - frontier (P orig)"
      using γ[of z] by (auto simp: isolated_zero_iff P_def γ_def path_image_parallelogram
    also have "P orig - frontier (P orig) = interior (P orig)"
      using interior_subset closure_subset by (auto simp: frontier_def)
    finally have "winding_number γ z = s"
      unfolding s_def γ_def P_def by (rule winding_number_parallelogram_inside)
    thus ?case
      by simp
  qed (use A fin closure_subset in auto)
  also have "{z∈P orig. f z = 0 ∨ is_pole f z} = zeros ∪ poles"
    by (auto simp: zeros_def poles_def isolated_zero_iff)
  also have "(∑z∈zeros ∪ poles. s * complex_of_int (zorder f z)) =
              s * of_int (∑z∈zeros ∪ poles. zorder f z)"
    by (simp add: sum_distrib_left)
  finally have "(∑z∈zeros ∪ poles. zorder f z) = 0"
    by (simp del: of_int_sum)
```

```
    also have "(∑z∈zeros ∪ poles. zorder f z) = (∑z∈zeros. zorder
f z) + (∑z∈poles. zorder f z)"
      by (subst sum.union_disjoint)
         (use A closure_subset
           in ‹auto simp: zeros_def poles_def isolated_zero_iff intro!:
finite_subset[OF _ fin]›)
    also have "(∑z∈zeros. zorder f z) = (∑z∈zeros. int (nat (zorder
f z)))"
    proof (intro sum.cong)
      fix z assume z: "z ∈ zeros"
      hence "¬is_pole f z"
        by (auto simp: zeros_def isolated_zero_iff)
      hence "f analytic_on {z}"
        using nicely_meromorphic nicely_meromorphic_on_imp_analytic_at
by blast
      hence "zorder f z > 0"
        using z by (intro zorder_isolated_zero_pos) (auto simp: zeros_def)
      thus "zorder f z = int (nat (zorder f z))"
        by simp
    qed auto
    also have "... = int (∑z∈zeros. nat (zorder f z))"
      by simp

    also have "(∑z∈poles. zorder f z) = (∑z∈poles. -int (nat (-zorder
f z)))"
    proof (intro sum.cong)
      fix z assume "z ∈ poles"
      hence "zorder f z < 0" using meromorphic
        by (auto intro!: isolated_pole_imp_neg_zorder simp: poles_def
meromorphic_on_altdef)
      thus "zorder f z = - int (nat (- zorder f z))"
        by simp
    qed auto
    also have "... = -int (∑z∈poles. nat (-zorder f z))"
      by (simp add: sum_negf)

    also have "(∑z∈poles. nat (-zorder f z)) = elliptic_order f"
      using poles_eq_elliptic_order[of orig] by (simp add: poles_def P_def)
    finally have "(∑z∈zeros. nat (zorder f z)) = elliptic_order f"
      by linarith
    thus ?case
      by (simp add: zeros_def)
  qed
qed
```

In the same vein, we get the following from our earlier result about the integral over $zf'(z)/f(z)$: The sum over all zeros and poles (counted with multiplicity, where poles have negative multiplicity) in a period parallelogram is a lattice point.

This is Exercise 1.2 in Apostol's book.

```
lemma sum_zeros_poles_in_lattice_aux:
  defines "Z ≡ (λorig. {z∈period_parallelogram orig. isolated_zero f
z ∨ is_pole f z})"
  defines "S ≡ (λorig. ∑z∈Z orig. of_int (zorder f z) * z)"
  shows   "S orig ∈ Λ"
proof (cases "f = (λ_. 0)")
  case True
  hence "elliptic_order f = 0"
    by (subst elliptic_order_eq_0_iff) (auto simp: constant_on_def)
  moreover have "¬isolated_zero f z" "¬is_pole f z" for z
    by (auto simp: isolated_zero_def True)
  ultimately show ?thesis
    by (simp add: S_def Z_def)
next
  case False
  define s where "s = complex_of_real (sgn (Im (ω2 / ω1)))"
  have s: "s ∈ {-1, 1}"
    using fundpair by (auto simp: s_def sgn_if fundpair_def complex_is_Real_iff)

  have ev_nonzero: "eventually (λz. f z ≠ 0) (cosparse UNIV)"
    using nicely_meromorphic False nicely_meromorphic_imp_constant_or_avoid[of
f UNIV 0]
    by auto

  have isolated_zero_iff: "isolated_zero f z ⟷ ¬is_pole f z ∧ f z
= 0" for z
  proof
    assume z: "¬is_pole f z ∧ f z = 0"
    hence "f analytic_on {z}"
      using nicely_meromorphic by (simp add: nicely_meromorphic_on_imp_analytic_at)
    with z have "f —z→ 0"
      by (metis analytic_at_imp_isCont continuous_within)
    moreover have "eventually (λz. f z ≠ 0) (at z)"
      using ev_nonzero by (subst (asm) eventually_cosparse_open_eq) auto
    ultimately show "isolated_zero f z"
      unfolding isolated_zero_def by blast
  next
    assume "isolated_zero f z"
    thus "¬is_pole f z ∧ f z = 0"
      by (meson iso_tuple_UNIV_I nicely_meromorphic
          nicely_meromorphic_on_imp_analytic_at pole_is_not_zero
          zero_isolated_zero_analytic)
  qed

  define P where "P = period_parallelogram"
  define avoid where "avoid = {z. isolated_zero f z ∨ is_pole f z}"
  have sparse_avoid: "¬z islimpt avoid" for z
  proof -
```

```
    have "∀≈z∈UNIV. ¬isolated_zero f z ∧ ¬is_pole f z" using meromorphic

      by (intro meromorphic_on_imp_not_zero_cosparse
                meromorphic_on_imp_not_pole_cosparse eventually_conj)
    hence "∀≈z∈UNIV. z ∉ avoid"
      by eventually_elim (auto simp: avoid_def)
    thus ?thesis
      by (auto simp: eventually_cosparse_open_eq islimpt_iff_eventually)
  qed
  thus ?thesis
    unfolding P_def [symmetric]
  proof (induction orig rule: shifted_period_parallelogram_avoid_wlog)
    case (shift orig d)
    obtain h where h: "bij_betw h (P (orig + d)) (P orig)" "⋀z. rel
(h z) z"
      using bij_betw_period_parallelograms unfolding P_def by blast
    have h': "bij_betw h (Z (orig + d)) (Z orig)"
      unfolding Z_def
      by (rule bij_betw_Collect)
         (use h(1) zeros.lattice_cong[OF h(2)] poles.lattice_cong[OF h(2)]
in ‹auto simp: P_def›)

    have "S (orig + d) = (∑z∈Z (orig + d). of_int (zorder f z) * z)"
      by (simp add: S_def)
    also have "rel (∑z∈Z (orig + d). of_int (zorder f z) * z)
                   (∑z∈Z (orig + d). of_int (zorder f z) * h z)"
      by (intro lattice_intros) (use h(2) in ‹auto simp: rel_sym›)
    also have "... = (∑z∈Z (orig + d). of_int (zorder f (h z)) * h z)"
      by (simp add: zorder.lattice_cong[OF h(2)])
    also have "... = (∑z∈Z orig. of_int (zorder f z) * z)"
      using h' by (rule sum.reindex_bij_betw)
    also have "... = S orig"
      by (simp add: S_def)
    also have "S orig ∈ Λ"
      by fact
    finally show ?case .
  next
    case (avoid orig)
    define γ where "γ = parallelogram_path orig ω1 ω2"
    define zeros where "zeros = {z∈P orig. isolated_zero f z}"
    define poles where "poles = {z∈P orig. is_pole f z}"
    have γ: "¬isolated_zero f z ∧ ¬is_pole f z" if "z ∈ path_image γ"
for z
      using avoid that by (auto simp: avoid_def γ_def)
    have "compact (closure (P orig))"
      unfolding P_def by auto
    then obtain R where R: "closure (P orig) ⊆ ball 0 R"
      using compact_imp_bounded bounded_subset_ballD by blast
    define A :: "complex set" where "A = ball 0 R"
```

```
    have A: "closure (P orig) ⊆ A"
      using R by (simp add: A_def)

    have fin: "finite {w ∈ A. f w = 0 ∨ w ∈ {z. is_pole f z}}"
    proof (rule finite_subset)
      show "finite (cball 0 R ∩ avoid)"
        by (rule finite_not_islimpt_in_compact) (use sparse_avoid in auto)
    next
      show "{w ∈ A. f w = 0 ∨ w ∈ {z. is_pole f z}} ⊆ cball 0 R ∩ avoid"
        by (auto simp: avoid_def A_def isolated_zero_iff)
    qed

    have "contour_integral γ (λx. deriv f x * x / f x) =
            of_real (2 * pi) * i * (∑p∈{w∈A. f w = 0 ∨ w ∈ {z. is_pole
f z}}.
              winding_number γ p * p * of_int (zorder f p))"
    proof (rule argument_principle)
      show "open A" "connected A"
        by (auto simp: A_def)
    next
      have "f analytic_on A - {z. is_pole f z}"
        using nicely_meromorphic_on_subset[OF nicely_meromorphic]
        by (rule nicely_meromorphic_without_singularities) auto
      thus "f holomorphic_on A - {z. is_pole f z}"
        by (rule analytic_imp_holomorphic)
    next
      show "path_image γ ⊆ A - {w ∈ A. f w = 0 ∨ w ∈ {z. is_pole f
z}}"
        using A γ path_image_parallelogram_subset_closure[of orig]
        by (auto simp: γ_def P_def isolated_zero_iff)
    next
      show "finite {w ∈ A. f w = 0 ∨ w ∈ {z. is_pole f z}}"
        by (rule finite_subset[OF _ fin]) auto
    next
      show "∀z. z ∉ A ⟶ winding_number γ z = 0"
        using A unfolding γ_def P_def by (auto intro!: winding_number_parallelogram_outside
    qed (auto simp: γ_def)
    hence "(∑p∈{w∈A. f w = 0 ∨ is_pole f w}. winding_number γ p * p
* of_int (zorder f p)) =
           contour_integral γ (λx. x * deriv f x / f x) / (2 * pi * i)"
      by (simp add: field_simps)
    also have "... ∈ Λ"
      unfolding γ_def by (rule argument_principle_f_z) (use γ in ‹auto
simp: γ_def isolated_zero_iff›)

    also have "(∑z∈{z∈A. f z = 0 ∨ is_pole f z}. winding_number γ z
* z * of_int (zorder f z)) =
               (∑z∈Z orig. s * of_int (zorder f z) * z)"
    proof (intro sum.mono_neutral_cong_right ballI, goal_cases)
```

```
      case (3 z)
      hence "z ∉ P orig ∪ frontier (P orig)"
        using γ[of z] by (auto simp: Z_def isolated_zero_iff P_def γ_def
path_image_parallelogram_path)
      also have "P orig ∪ frontier (P orig) = closure (P orig)"
        using closure_Un_frontier by blast
      finally have "winding_number γ z = 0"
        unfolding γ_def P_def using winding_number_parallelogram_outside
by blast
      thus ?case
        by simp
    next
      case (4 z)
      hence "z ∈ P orig - frontier (P orig)"
        using γ[of z] by (auto simp: Z_def isolated_zero_iff P_def γ_def
path_image_parallelogram_path)
      also have "P orig - frontier (P orig) = interior (P orig)"
        using interior_subset closure_subset by (auto simp: frontier_def)
      finally have "winding_number γ z = s"
        unfolding s_def γ_def P_def by (rule winding_number_parallelogram_inside)
      thus ?case
        by simp
    qed (use A fin closure_subset in ‹auto simp: Z_def P_def isolated_zero_iff›)
    also have "(∑z∈Z orig. s * of_int (zorder f z) * z) = s * S orig"
      by (simp add: S_def sum_distrib_left mult.assoc)
    finally show "S orig ∈ Λ"
      using s by (auto simp: uminus_in_lattice_iff)
  qed
qed
```

Again, similarly: The residues in a period parallelogram sum to 0.

```
lemma sum_residues_eq_0_aux:
  defines "Q ≡ (λorig. {z∈period_parallelogram orig. is_pole f z})"
  defines "S ≡ (λorig. ∑z∈Q orig. residue f z)"
  shows   "S orig ∈ Λ"
proof (cases "f = (λ_. 0)")
  case True
  hence "elliptic_order f = 0"
    by (subst elliptic_order_eq_0_iff) (auto simp: constant_on_def)
  moreover have "¬is_pole f z" for z
    by (auto simp: isolated_zero_def True)
  ultimately show ?thesis
    by (simp add: S_def Q_def)
next
  case False
  define s where "s = complex_of_real (sgn (Im (ω2 / ω1)))"
  have s: "s ∈ {-1, 1}"
    using fundpair by (auto simp: s_def sgn_if fundpair_def complex_is_Real_iff)
```

```
  have ev_nonzero: "eventually (λz. f z ≠ 0) (cosparse UNIV)"
    using nicely_meromorphic False nicely_meromorphic_imp_constant_or_avoid[of
f UNIV 0]
    by auto

  define P where "P = period_parallelogram"
  define avoid where "avoid = {z. is_pole f z}"
  have sparse_avoid: "¬z islimpt avoid" for z
    unfolding avoid_def
    by (meson UNIV_I meromorphic meromorphic_on_isolated_singularity
          meromorphic_on_subset not_islimpt_poles subsetI)
  thus ?thesis
    unfolding P_def [symmetric]
  proof (induction orig rule: shifted_period_parallelogram_avoid_wlog)
    case (shift orig d)
    obtain h where h: "bij_betw h (P (orig + d)) (P orig)" "⋀z. rel
(h z) z"
      using bij_betw_period_parallelograms unfolding P_def by blast
    have h': "bij_betw h (Q (orig + d)) (Q orig)"
      unfolding Q_def
      by (rule bij_betw_Collect)
         (use h(1) poles.lattice_cong[OF h(2)] in ‹auto simp: P_def›)

    have "S (orig + d) = (∑z∈Q (orig + d). residue f z)"
      by (simp add: S_def)
    also have "... = (∑z∈Q (orig + d). residue f (h z))"
      by (simp add: rel_sym residue.lattice_cong[OF h(2)])
    also have "... = (∑z∈Q orig. residue f z)"
      using h' by (rule sum.reindex_bij_betw)
    also have "... = S orig"
      by (simp add: S_def)
    also have "S orig ∈ Λ"
      by fact
    finally show ?case .
  next
    case (avoid orig)
    define γ where "γ = parallelogram_path orig ω1 ω2"
    have γ: "¬is_pole f z" if "z ∈ path_image γ" for z
      using avoid that by (auto simp: avoid_def γ_def)
    have "compact (closure (P orig))"
      unfolding P_def by auto
    then obtain R where R: "closure (P orig) ⊆ ball 0 R"
      using compact_imp_bounded bounded_subset_ballD by blast
    define A :: "complex set" where "A = ball 0 R"
    have A: "closure (P orig) ⊆ A"
      using R by (simp add: A_def)

    have fin: "finite {w ∈ A. is_pole f w}"
    proof (rule finite_subset)
```

```
      show "finite (cball 0 R ∩ avoid)"
        by (rule finite_not_islimpt_in_compact) (use sparse_avoid in auto)
    next
      show "{z ∈ A. is_pole f z} ⊆ cball 0 R ∩ avoid"
        by (auto simp: avoid_def A_def)
    qed

    have "contour_integral γ f =
          of_real (2 * pi) * i * (∑p∈{z∈A. is_pole f z}. winding_number
γ p * residue f p)"
    proof (rule Residue_theorem)
      show "open A" "connected A"
        by (auto simp: A_def)
    next
      have "f analytic_on A - {z∈A. is_pole f z}"
        using nicely_meromorphic_on_subset[OF nicely_meromorphic]
        by (rule nicely_meromorphic_without_singularities) auto
      thus "f holomorphic_on A - {z∈A. is_pole f z}"
        by (rule analytic_imp_holomorphic)
    next
      show "path_image γ ⊆ A - {z ∈ A. is_pole f z}"
        using A γ path_image_parallelogram_subset_closure[of orig]
        by (auto simp: γ_def P_def)
    next
      show "finite {z ∈ A. is_pole f z}"
        using fin by simp
    next
      show "∀z. z ∉ A ⟶ winding_number γ z = 0"
        using A unfolding γ_def P_def by (auto intro!: winding_number_parallelogram_outside
    qed (auto simp: γ_def)
    also have "(∑p∈{z∈A. is_pole f z}. winding_number γ p * residue
f p) =
               (∑p∈Q orig. s * residue f p)"
    proof (intro sum.mono_neutral_cong_right ballI, goal_cases)
      case (3 z)
      hence "z ∉ P orig ∪ frontier (P orig)"
        using γ[of z] by (auto simp: Q_def P_def γ_def path_image_parallelogram_path)
      also have "P orig ∪ frontier (P orig) = closure (P orig)"
        using closure_Un_frontier by blast
      finally have "winding_number γ z = 0"
        unfolding γ_def P_def using winding_number_parallelogram_outside
by blast
      thus ?case
        by simp
    next
      case (4 z)
      hence "z ∈ P orig - frontier (P orig)"
        using γ[of z] by (auto simp: Q_def P_def γ_def path_image_parallelogram_path)
      also have "P orig - frontier (P orig) = interior (P orig)"
```

```
      using interior_subset closure_subset by (auto simp: frontier_def)
    finally have "winding_number γ z = s"
      unfolding s_def γ_def P_def by (rule winding_number_parallelogram_inside)
    thus ?case
      by simp
  qed (use A fin closure_subset in ‹auto simp: Q_def P_def›)
  also have "... = s * (∑z∈Q orig. residue f z)"
    by (simp add: sum_distrib_left)
  also have "contour_integral γ f = 0"
    using γ unfolding γ_def
    by (subst contour_integral_parallelogram_path')
       (auto simp: f_periodic intro!: continuous_on)
  finally show ?case
    using s by (auto simp: S_def)
 qed
qed

end
```

We now lift everything we have done to non-nice elliptic functions.

```
context elliptic_function
begin

lemma elliptic_order_remove_sings [simp]: "elliptic_order (remove_sings
f) = elliptic_order f"
  unfolding elliptic_order_def by simp

interpretation elliptic_function_remove_sings ..

theorem zeros_eq_elliptic_order:
  "(∑z | z ∈ period_parallelogram orig ∧ isolated_zero f z. nat (zorder
f z)) = elliptic_order f"
  using remove_sings.zeros_eq_elliptic_order_aux by simp

lemma card_poles_le_order: "card {z∈period_parallelogram orig. is_pole
f z} ≤ elliptic_order f"
proof -
  have "zorder f z < 0" if "is_pole f z" for z
    using that by (simp add: isolated_pole_imp_neg_zorder)
  hence "(∑z | z ∈ period_parallelogram orig ∧ is_pole f z. 1) ≤
         (∑z | z ∈ period_parallelogram orig ∧ is_pole f z. nat (-zorder
f z))"
    by (intro sum_mono) force+
  thus ?thesis
    by (simp add: poles_eq_elliptic_order)
qed

lemma card_zeros_le_order: "card {z∈period_parallelogram orig. isolated_zero
f z} ≤ elliptic_order f"
```

```
proof -
  have "zorder f z > 0" if "isolated_zero f z" for z
    using that by (intro zorder_isolated_zero_pos') auto
  hence "(∑z | z ∈ period_parallelogram orig ∧ isolated_zero f z. 1)
≤
         (∑z | z ∈ period_parallelogram orig ∧ isolated_zero f z. nat
(zorder f z))"
    by (intro sum_mono) force+
  thus ?thesis
    by (simp add: zeros_eq_elliptic_order)
qed

corollary elliptic_order_eq_0_iff_no_poles: "elliptic_order f = 0 ⟷
(∀z. ¬is_pole f z)"
proof
  assume "elliptic_order f = 0"
  hence "card {z∈period_parallelogram 0. is_pole f z} ≤ 0"
    using card_poles_le_order[of 0] by simp
  hence "{z∈period_parallelogram 0. is_pole f z} = {}"
    using finite_poles_in_parallelogram[of 0] by simp
  hence *: "¬is_pole f z" if "z ∈ period_parallelogram 0" for z
    using that by blast
  hence "¬is_pole f z" for z
    using *[of "to_fund_parallelogram z"] poles.lattice_cong[of z "to_fund_parallelogram
z"]
    by (auto simp: to_fund_parallelogram_in_parallelogram)
  thus "∀z. ¬is_pole f z"
    by blast
qed (simp_all add: elliptic_order_def)

corollary elliptic_order_eq_0_iff_no_zeros: "elliptic_order f = 0 ⟷
(∀z. ¬isolated_zero f z)"
proof
  assume "elliptic_order f = 0"
  hence "card {z∈period_parallelogram 0. isolated_zero f z} ≤ 0"
    using card_zeros_le_order[of 0] by simp
  hence "{z∈period_parallelogram 0. isolated_zero f z} = {}"
    using finite_zeros_in_parallelogram[of 0] by simp
  hence *: "¬isolated_zero f z" if "z ∈ period_parallelogram 0" for z
    using that by blast
  hence "¬isolated_zero f z" for z
    using *[of "to_fund_parallelogram z"] zeros.lattice_cong[of z "to_fund_parallelogram
z"]
    by (auto simp: to_fund_parallelogram_in_parallelogram)
  thus "∀z. ¬isolated_zero f z"
    by blast
qed (simp_all flip: zeros_eq_elliptic_order[of 0])

lemma elliptic_order_eq_0_iff_const_cosparse:
```

```
  "elliptic_order f = 0 ⟷ (∃c. ∀≈x∈UNIV. f x = c)"
proof -
  have "elliptic_order f = 0 ⟷ elliptic_order (remove_sings f) = 0"
    by simp
  also have "... ⟷ remove_sings f constant_on UNIV"
    by (subst remove_sings.elliptic_order_eq_0_iff) auto
  also have "... ⟷ (∃c. ∀≈x∈UNIV. f x = c)"
    by (subst remove_sings_constant_on_open_iff) (auto intro: meromorphic)
  finally show ?thesis .
qed

lemma cosparse_eq_or_avoid: "(∀≈z∈UNIV. f z = c) ∨ (∀≈z∈UNIV. f z
≠ c)"
  by (simp add: Convex.connected_UNIV meromorphic meromorphic_imp_constant_or_avoid)

lemma frequently_eq_imp_almost_everywhere_eq:
  assumes "frequently (λz. f z = c) (at z)"
  shows   "eventually (λz. f z = c) (cosparse UNIV)"
proof -
  have "¬eventually (λz. f z ≠ c) (cosparse UNIV)"
    using assms by (auto simp: eventually_cosparse_open_eq frequently_def)
  thus ?thesis
    using cosparse_eq_or_avoid[of c] by auto
qed

lemma eventually_eq_imp_almost_everywhere_eq:
  assumes "eventually (λz. f z = c) (at z)"
  shows   "eventually (λz. f z = c) (cosparse UNIV)"
  using assms frequently_eq_imp_almost_everywhere_eq eventually_frequently
at_neq_bot by blast

lemma avoid: "elliptic_order f > 0 ⟹ ∀≈z∈UNIV. f z ≠ c"
  using cosparse_eq_or_avoid elliptic_order_eq_0_iff_const_cosparse by
auto

lemma avoid': "elliptic_order f > 0 ⟹ eventually (λz. f z ≠ c) (at
z)"
  using avoid eventually_cosparse_imp_eventually_at by blast

theorem sum_zeros_poles_in_lattice:
  fixes orig :: complex
  defines "Z ≡ {z∈period_parallelogram orig. isolated_zero f z ∨ is_pole
f z}"
  shows   "(∑z∈Z. of_int (zorder f z) * z) ∈ Λ"
  using remove_sings.sum_zeros_poles_in_lattice_aux[of orig]
  by (simp add: Z_def)

theorem sum_residues_eq_0:
  fixes orig :: complex
```

```
  defines "Q ≡ {z∈period_parallelogram orig. is_pole f z}"
  shows   "(∑z∈Q. residue f z) ∈ Λ"
  using remove_sings.sum_residues_eq_0_aux[of orig] by (simp add: Q_def)
```

An obvious fact that we use at one point: if $\sum_{x \in A} f(x) = 1$ for $f(x)$ in the positive integers, then $A = \{x\}$ for some x and $f(x) = 1$.

```
lemma (in -) sum_nat_eq_1E:
  fixes f :: "'a ⇒ nat"
  assumes sum_eq: "(∑x∈A. f x) = 1"
  assumes pos: "⋀x. x ∈ A ⟹ f x > 0"
  obtains x where "A = {x}" "f x = 1"
proof -
  have [simp, intro]: "finite A"
    by (rule ccontr) (use sum_eq in auto)
  have "A ≠ {}"
    using sum_eq by auto
  then obtain x where x: "x ∈ A"
    by auto
  have "(∑x∈A. f x) = f x + (∑x∈A-{x}. f x)"
    by (meson ‹finite A› ‹x ∈ A› sum.remove)
  hence "f x + (∑x∈A-{x}. f x) = 1"
    using sum_eq by simp
  with pos[OF x] have "f x = 1" "(∑x∈A-{x}. f x) = 0"
    by linarith+
  from this have "∀y∈ A - {x}. f y = 0"
    by (subst (asm) sum_eq_0_iff) auto
  with pos have "A - {x} = {}"
    by force
  with x have "A = {x}"
    by auto
  with ‹f x = 1› show ?thesis
    using that[of x] by blast
qed
```

A simple consequence of our result about the sums of poles and zeros being a lattice point is that there are no elliptic functions of order 1.

If there were such a function, it would have only one zero and one pole (both simple) in the fundamental parallelogram. Since their sum would be a lattice point, they would be equivalent modulo the lattice and thus identical. But a point cannot be both a zero and a pole.

```
theorem elliptic_order_neq_1: "elliptic_order f ≠ 1"
proof
  assume "elliptic_order f = 1"
  define P where "P = period_parallelogram 0"

  from ‹elliptic_order f = 1› have *: "(∑z | z ∈ P ∧ is_pole f z. nat
(- zorder f z)) = 1"
    by (simp add: elliptic_order_def P_def)
```

```
  moreover have "zorder f z < 0" if "is_pole f z" for z using that meromorphic
    by (meson isolated_pole_imp_neg_zorder meromorphic_at_iff meromorphic_on_subset
top_greatest)
  ultimately obtain pole where pole: "{z. z ∈ P ∧ is_pole f z} = {pole}"
"zorder f pole = -1"
    by (elim sum_nat_eq_1E) auto

  from ‹elliptic_order f = 1› have *: "(∑z | z ∈ P ∧ isolated_zero
f z. nat (zorder f z)) = 1"
    using zeros_eq_elliptic_order[of 0] by (simp add: P_def)
  moreover have "zorder f z > 0" if "isolated_zero f z" for z using that
    by (simp add: zorder_isolated_zero_pos')
  ultimately obtain zero where zero: "{z. z ∈ P ∧ isolated_zero f z}
= {zero}" "zorder f zero = 1"
    by (elim sum_nat_eq_1E) auto

  have zero': "isolated_zero f zero" and pole': "is_pole f pole"
    using zero pole by blast+
  have "zero ≠ pole"
    using zero' pole' pole_is_not_zero by blast

  have "(∑z | z ∈ P ∧ (isolated_zero f z ∨ is_pole f z). of_int (zorder
f z) * z) ∈ Λ"
    unfolding P_def by (rule sum_zeros_poles_in_lattice)
  also have "{z. z ∈ P ∧ (isolated_zero f z ∨ is_pole f z)} = {zero,
pole}"
    using zero pole by auto
  finally have "zero - pole ∈ Λ"
    using ‹zero ≠ pole› zero pole by simp
  hence "rel zero pole"
    by (simp add: rel_def)
  thus False
    using zero' pole' pole_is_not_zero poles.lattice_cong by blast
qed

end


locale nonconst_nicely_elliptic_function = nicely_elliptic_function +
  assumes order_pos: "elliptic_order f > 0"
begin

lemma isolated_zero_iff': "isolated_zero f z ⟷ ¬is_pole f z ∧ f z
= 0"
proof -
  have "f ≠ (λ_. 0)"
    using order_pos by auto
  thus ?thesis
    using isolated_zero_iff[of z] by simp
```

qed

end

5.3 Even elliptic functions

If an elliptic function is even, i.e. $f(-z) = f(z)$, it is invariant not only under the group generated by $z \mapsto z + \omega_1$ and $z \mapsto z + \omega_2$, but also the additional generator $z \mapsto -z$.
Since our prototypical example of an elliptic function – the Weierstraß $\wp$ function – is even, we will examine these a bit more closely here.

```
locale even_elliptic_function = elliptic_function +
  assumes even: "f (-z) = f z"
begin
```

The Laurent series expansion of an even elliptic function at lattice points and half-lattice points only has even-index coefficients. This also means that, at lattice and half-lattice points, an even elliptic function can only have zeros and poles of even order.

```
lemma
  assumes z: "2 * z ∈ Λ" and "¬(∀≈z. f z = 0)"
  shows   odd_laurent_coeffs_eq_0: "odd n ⟹ fls_nth (laurent_expansion
f z) n = 0"
    and   even_zorder: "even (zorder f z)"
proof -
  define F where "F = laurent_expansion f z"
  have F: "(λw. f (z + w)) has_laurent_expansion F"
    unfolding F_def  by (simp add: not_essential_has_laurent_expansion)
  have "F ≠ 0"
  proof
    assume "F = 0"
    with F have "eventually (λw. f w = 0) (at z)"
      using has_laurent_expansion_eventually_zero_iff by blast
    thus False
      using assms(2) eventually_eq_imp_almost_everywhere_eq by blast
  qed

  have "((λw. f (z + w)) ∘ uminus) has_laurent_expansion (fls_compose_fps
F (-fps_X))"
    by (intro laurent_expansion_intros F has_laurent_expansion_fps fps_expansion_intros)
auto
  also have "((λw. f (z + w)) ∘ uminus) = (λw. f (z + w))" (is "?lhs =
?rhs")
  proof
    fix w :: complex
    have "?lhs w = f (z - w)"
      by simp
```

```
    also have "... = f (-(z + w))"
      by (rule lattice_cong) (use assms z in ‹auto simp: rel_def›)
    also have "... = f (z + w)"
      by (rule even)
    finally show "?lhs w = ?rhs w" .
  qed
  finally have "(λw. f (z + w)) has_laurent_expansion fls_compose_fps F
(- fps_X)" .
  with F have "F = fls_compose_fps F (- fps_X)"
    using has_laurent_expansion_unique by blast
  thus even': "fls_nth F n = 0" if "odd n" for n
    using that fls_nth_fls_compose_fps_linear[of "-1" F n]
    by (auto simp: fls_eq_iff power_int_minus_left simp flip: fps_const_neg)

  have "fls_nth F (fls_subdegree F) ≠ 0"
    using ‹F ≠ 0› by auto
  with even' have "even (fls_subdegree F)"
    by blast
  also have "fls_subdegree F = zorder f z"
    using F ‹F ≠ 0› by (metis has_laurent_expansion_zorder)
  finally show "even (zorder f z)" .
qed

lemma lattice_cong': "rel w z ∨ rel w (-z) ⟹ f w = f z"
  using even lattice_cong by metis

lemma eval_to_half_fund_parallelogram: "f (to_half_fund_parallelogram
z) = f z"
  using rel_to_half_fund_parallelogram[of z] even lattice_cong by metis

lemma zorder_to_half_fund_parallelogram: "zorder f (to_half_fund_parallelogram
z) = zorder f z"
proof -
  define z' where "z' = to_half_fund_parallelogram z"
  have "rel z z' ∨ rel z (-z')"
    unfolding z'_def by (rule rel_to_half_fund_parallelogram)
  hence "zorder f z = zorder f z'"
  proof
    assume "rel z z'"
    thus "zorder f z = zorder f z'"
      by (rule zorder.lattice_cong)
  next
    assume "rel z (-z')"
    hence "zorder f z = zorder f (-z')"
      by (rule zorder.lattice_cong)
    also have "... = zorder (λz. f (-z)) z'"
      by (rule zorder_uminus [symmetric]) (auto intro!: meromorphic')
    also have "... = zorder f z'"
      by (simp add: even)
```

```
    finally show "zorder f z = zorder f z'" .
  qed
  thus ?thesis
    by (simp add: z'_def)
qed

lemma zorder_uminus: "zorder f (-z) = zorder f z"
  by (metis rel_refl to_half_fund_parallelogram_eq_iff zorder_to_half_fund_parallelogram)

end
```

5.4 Closure properties of the class of elliptic functions

Elliptic functions are closed under all basic arithmetic operations (addition, subtraction, multiplication, division). Additionally, they are closed under derivative, translation ($f(z) \rightsquigarrow f(z+c)$) and scaling with an integer ($f(z) \rightsquigarrow f(nz)$).

Furthermore, constant functions are elliptic.

```
lemma elliptic_function_unop:
  assumes "elliptic_function ω1 ω2 f"
  assumes "f meromorphic_on UNIV ⟹ (λz. h (f z)) meromorphic_on UNIV"
  shows   "elliptic_function ω1 ω2 (λz. h (f z))"
proof -
  interpret elliptic_function ω1 ω2 f by fact
  interpret complex_lattice_periodic_compose ω1 ω2 f h ..
  show ?thesis
    by standard (use assms(2) meromorphic in auto)
qed

lemma elliptic_function_binop:
  assumes "elliptic_function ω1 ω2 f" "elliptic_function ω1 ω2 g"
  assumes "f meromorphic_on UNIV ⟹ g meromorphic_on UNIV ⟹ (λz. h
(f z) (g z)) meromorphic_on UNIV"
  shows   "elliptic_function ω1 ω2 (λz. h (f z) (g z))"
proof -
  interpret f: elliptic_function ω1 ω2 f by fact
  interpret g: elliptic_function ω1 ω2 g by fact
  interpret complex_lattice_periodic ω1 ω2 "λz. h (f z) (g z)"
    by standard (auto intro!: arg_cong2[of _ _ _ _ h] f.lattice_cong g.lattice_cong
simp: f.rel_def)
  show ?thesis
    by standard (use assms(3) f.meromorphic g.meromorphic in auto)
qed


context complex_lattice
begin
```

```
named_theorems elliptic_function_intros

lemmas (in elliptic_function) [elliptic_function_intros] = elliptic_function_axioms

lemma elliptic_function_const [elliptic_function_intros]:
  "elliptic_function ω1 ω2 (λ_. c)"
  by standard auto

lemma [elliptic_function_intros]:
  assumes "elliptic_function ω1 ω2 f"
  shows   elliptic_function_cmult_left: "elliptic_function ω1 ω2 (λz.
c * f z)"
    and   elliptic_function_cmult_right: "elliptic_function ω1 ω2 (λz.
f z * c)"
    and   elliptic_function_scaleR: "elliptic_function ω1 ω2 (λz. c'
*R f z)"
    and   elliptic_function_uminus: "elliptic_function ω1 ω2 (λz. -f
z)"
    and   elliptic_function_inverse: "elliptic_function ω1 ω2 (λz. inverse
(f z))"
    and   elliptic_function_power: "elliptic_function ω1 ω2 (λz. f z
^ m)"
    and   elliptic_function_power_int: "elliptic_function ω1 ω2 (λz.
f z powi n)"
  by (rule elliptic_function_unop[OF assms(1)]; force intro!: meromorphic_intros)+

lemma [elliptic_function_intros]:
  assumes "elliptic_function ω1 ω2 f" "elliptic_function ω1 ω2 g"
  shows   elliptic_function_cmult_add: "elliptic_function ω1 ω2 (λz.
f z + g z)"
    and   elliptic_function_cmult_diff: "elliptic_function ω1 ω2 (λz.
f z - g z)"
    and   elliptic_function_cmult_mult: "elliptic_function ω1 ω2 (λz.
f z * g z)"
    and   elliptic_function_cmult_divide: "elliptic_function ω1 ω2 (λz.
f z / g z)"
  by (rule elliptic_function_binop[OF assms(1,2)]; force intro!: meromorphic_intros)+


lemma elliptic_function_compose_mult_of_int_left:
  assumes "elliptic_function ω1 ω2 f"
  shows   "elliptic_function ω1 ω2 (λz. f (of_int n * z))"
proof -
  interpret elliptic_function ω1 ω2 f
    by fact
  show ?thesis
    by unfold_locales
       (auto intro!: meromorphic_intros analytic_intros lattice_cong lattice_intros
             simp: rel_def uminus_in_lattice_iff ring_distribs)
```

qed

lemma *elliptic_function_compose_mult_of_nat_left:*
assumes *"elliptic_function ω1 ω2 f"*
shows *"elliptic_function ω1 ω2 (λz. f (of_nat n * z))"*
using *elliptic_function_compose_mult_of_int_left[OF assms, of "int n"]*
by *simp*

lemma *elliptic_function_compose_mult_numeral_left:*
assumes *"elliptic_function ω1 ω2 f"*
shows *"elliptic_function ω1 ω2 (λz. f (numeral n * z))"*
using *elliptic_function_compose_mult_of_int_left[OF assms, of "numeral n"]* **by** *simp*

lemma
assumes *"elliptic_function ω1 ω2 f"*
shows *elliptic_function_compose_mult_of_int_right: "elliptic_function ω1 ω2 (λz. f (z * of_int n))"*
and *elliptic_function_compose_mult_of_nat_right: "elliptic_function ω1 ω2 (λz. f (z * of_nat m))"*
and *elliptic_function_compose_mult_numeral_right: "elliptic_function ω1 ω2 (λz. f (z * numeral num))"*
by *(subst mult.commute,*
rule elliptic_function_compose_mult_of_int_left
elliptic_function_compose_mult_of_nat_left
elliptic_function_compose_mult_numeral_left,
rule assms)+

lemma *elliptic_function_compose_uminus:*
assumes *"elliptic_function ω1 ω2 f"*
shows *"elliptic_function ω1 ω2 (λz. f (-z))"*
using *elliptic_function_compose_mult_of_int_left[OF assms, of "-1"]*
by *simp*

lemma *elliptic_function_shift:*
assumes *"elliptic_function ω1 ω2 f"*
shows *"elliptic_function ω1 ω2 (λz. f (z + w))"*
proof -
interpret *elliptic_function ω1 ω2 f* **by** *fact*
show *?thesis*
proof
show *"(λz. f (z + w)) meromorphic_on UNIV"*
using *meromorphic* **by** *(rule meromorphic_on_compose) (auto intro!: analytic_intros)*
qed *(auto intro!: lattice_cong simp: rel_def)*
qed

definition *shift_fun :: "'a ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a :: plus"* **where**

```
  "shift_fun w f = (λz. f (z + w))"

lemma elliptic_function_shift' [elliptic_function_intros]:
  assumes "elliptic_function ω1 ω2 f"
  shows   "elliptic_function ω1 ω2 (shift_fun w f)"
  unfolding shift_fun_def using assms by (rule elliptic_function_shift)


lemma nicely_elliptic_function_remove_sings [elliptic_function_intros]:
  assumes "elliptic_function ω1 ω2 f"
  shows   "nicely_elliptic_function ω1 ω2 (remove_sings f)"
proof -
  interpret elliptic_function ω1 ω2 f by fact
  interpret elliptic_function_remove_sings ω1 ω2 f ..
  show ?thesis ..
qed

lemma elliptic_function_remove_sings [elliptic_function_intros]:
  assumes "elliptic_function ω1 ω2 f"
  shows   "elliptic_function ω1 ω2 (remove_sings f)"
proof -
  interpret elliptic_function ω1 ω2 f by fact
  interpret elliptic_function_remove_sings ω1 ω2 f ..
  show ?thesis ..
qed

lemma elliptic_function_deriv [elliptic_function_intros]:
  assumes "elliptic_function ω1 ω2 f"
  shows   "elliptic_function ω1 ω2 (deriv f)"
proof -
  interpret elliptic_function ω1 ω2 f by fact
  show ?thesis by standard (auto intro!: meromorphic_intros meromorphic)
qed

lemma elliptic_function_higher_deriv [elliptic_function_intros]:
  assumes "elliptic_function ω1 ω2 f"
  shows   "elliptic_function ω1 ω2 ((deriv ^^ n) f)"
  using assms by (induction n) (auto intro!: elliptic_function_intros)

lemma elliptic_function_sum [elliptic_function_intros]:
  assumes "⋀x. x ∈ X ⟹ elliptic_function ω1 ω2 (f x)"
  shows   "elliptic_function ω1 ω2 (λz. ∑x∈X. f x z)"
  using assms
  by (induction X rule: infinite_finite_induct) (auto intro!: elliptic_function_intros)

lemma elliptic_function_prod [elliptic_function_intros]:
  assumes "⋀x. x ∈ X ⟹ elliptic_function ω1 ω2 (f x)"
  shows   "elliptic_function ω1 ω2 (λz. ∏x∈X. f x z)"
  using assms
```

```
  by (induction X rule: infinite_finite_induct) (auto intro!: elliptic_function_intros)

lemma elliptic_function_sum_list [elliptic_function_intros]:
  assumes "⋀f. f ∈ set fs ⟹ elliptic_function ω1 ω2 f"
  shows   "elliptic_function ω1 ω2 (λz. ∑f←fs. f z)"
  using assms by (induction fs) (auto intro!: elliptic_function_intros)

lemma elliptic_function_prod_list [elliptic_function_intros]:
  assumes "⋀f. f ∈ set fs ⟹ elliptic_function ω1 ω2 f"
  shows   "elliptic_function ω1 ω2 (λz. ∏f←fs. f z)"
  using assms by (induction fs) (auto intro!: elliptic_function_intros)

lemma elliptic_function_sum_mset [elliptic_function_intros]:
  assumes "⋀f. f ∈# F ⟹ elliptic_function ω1 ω2 f"
  shows   "elliptic_function ω1 ω2 (λz. ∑f∈#F. f z)"
  using assms by (induction F) (auto intro!: elliptic_function_intros)

lemma elliptic_function_prod_mset [elliptic_function_intros]:
  assumes "⋀f. f ∈# F ⟹ elliptic_function ω1 ω2 f"
  shows   "elliptic_function ω1 ω2 (λz. ∏f∈#F. f z)"
  using assms by (induction F) (auto intro!: elliptic_function_intros)

end
```

5.5 Affine transformations and surjectivity

In the following we look at the properties of the elliptic function $af(z) + b$, where $a \neq 0$. Obviously this function inherits many properties from $f(z)$.

```
locale elliptic_function_affine = elliptic_function +
  fixes a b :: complex and g :: "complex ⇒ complex"
  defines "g ≡ λz. a * f z + b"
  assumes nonzero_const: "a ≠ 0"
begin

sublocale affine: elliptic_function ω1 ω2 g
  unfolding g_def by (intro elliptic_function_intros elliptic_function_axioms)

lemma is_pole_affine_iff: "is_pole g z ⟷ is_pole f z"
  using nonzero_const by (simp add: g_def flip: is_pole_plus_const_iff)

lemma zorder_pole_affine:
  assumes "is_pole f z"
  shows   "zorder g z = zorder f z"
proof -
  note [simp] = nonzero_const
  have "zorder (λz. a * f z + b) z = zorder (λz. a * f z) z"
  proof (cases "b = 0")
    case [simp]: False
    show ?thesis
```

```
    proof (intro zorder_add1)
      from assms have "zorder (λz. a * f z) z < 0"
        by (intro isolated_pole_imp_neg_zorder) (auto intro!: singularity_intros)
      thus "zorder (λz. a * f z) z < zorder (λz. b) z"
        by simp
    next
      have "elliptic_order f > 0"
        using assms elliptic_order_eq_0_iff_no_poles by blast
      hence "∀F z in at z. a * f z ≠ 0"
        using avoid'[of 0 z] by auto
      thus "∃F z in at z. a * f z ≠ 0"
        using at_neq_bot eventually_frequently by blast
    qed (use nonzero_const in ‹auto intro!: meromorphic_intros meromorphic›)
  qed auto
  also have "... = zorder f z"
    by (rule zorder_cmult) auto
  finally show ?thesis by (simp only: g_def)
qed

lemma order_affine_eq: "elliptic_order g = elliptic_order f"
  unfolding elliptic_order_def using nonzero_const
  by (intro sum.cong Collect_cong conj_cong refl)
     (auto simp flip: is_pole_plus_const_iff simp: zorder_pole_affine
is_pole_affine_iff)

end
```

One consequence of the above is that a non-constant elliptic function takes on each value in $\mathbb{C}$ "equally often". In particular, this means that any non-constant elliptic function is surjective, i.e. for every $c \in \mathbb{C}$ there exists a preimage z with $f(z) = c$ in every period parallelogram.

```
context nonconst_nicely_elliptic_function
begin

theorem surj:
  fixes c :: complex
  obtains z where "¬is_pole f z" "z ∈ period_parallelogram w" "f z =
c"
proof -
  define g where "g = (λz. f z - c)"
  interpret elliptic_function_affine ω1 ω2 f 1 "-c" g
    by unfold_locales (auto simp: g_def)

  have "elliptic_order g > 0"
    using order_affine_eq order_pos by simp
  then obtain z where z: "isolated_zero g z"
    using affine.elliptic_order_eq_0_iff_no_zeros by auto
  moreover have "¬is_pole f z"
    using z pole_is_not_zero is_pole_affine_iff by blast
```

```
  hence "f analytic_on {z}"
    by (simp add: analytic_at_iff_not_pole)
  hence "g analytic_on {z}"
    by (auto simp: g_def intro!: analytic_intros)
  ultimately have g: "g z = 0"
    using zero_isolated_zero_analytic by auto

  obtain h where h: "bij_betw h (period_parallelogram z) (period_parallelogram
w)" "⋀z. rel (h z) z"
    by (rule bij_betw_period_parallelograms[of z w]) blast

  show ?thesis
  proof (rule that[of "h z"])
    show "¬is_pole f (h z)"
      using ‹¬is_pole f z› h(2)[of z] poles.lattice_cong by blast
  next
    show "h z ∈ period_parallelogram w"
      using h(1) by (auto simp: bij_betw_def)
  next
    show "f (h z) = c"
      using g h(2)[of z] unfolding g_def by (simp add: lattice_cong)
  qed
qed

end

end
```

6 The Weierstraß $\wp$ Function

```
theory Weierstrass_Elliptic
imports
  Elliptic_Functions
  Modular_Group
begin
```

In this section, we will define the Weierstraß $\wp$ function, which is in some sense the simplest and most fundamental elliptic function. All elliptic functions can be expressed solely in terms of $\wp$ and $\wp'$.

6.1 Preliminary convergence results

We first examine the uniform convergence of the series

$$\sum_{\omega \in \Lambda^*} \frac{1}{(z-\omega)^n}$$

and

$$\sum_{\omega \in \Lambda} \frac{1}{(z-\omega)^n}$$

for fixed $n \geq 3$.

The second version is an elliptic function that we call the *Eisenstein function* because setting $z = 0$ gives us the Eisenstein series. To our knowledge this function does not have a name of its own in the literature.

This is perhaps because it is up to a constant factor, equal to the $(n-2)$-nth derivative of the Weierstraß $\wp$ function (which we will define a bit afterwards).

```
lemmas [simp del] = div_mult_self1 div_mult_self2 div_mult_self3 div_mult_self4


context complex_lattice
begin

lemma ω_upper:
  assumes "ω ∈ lattice_layer k" and "α > 0" and "k > 0"
  shows "norm ω powr -α ≤ (k*Inf_para) powr -α"
  using lattice_layer_le_norm Inf_para_pos assms powr_mono2' by force

lemma sum_ω_upper:
  assumes "α > 0" and "k > 0"
  shows "(∑ω ∈ lattice_layer k. norm ω powr -α) ≤ 8 * k powr (1-α)
* Inf_para powr -α"
    (is "?lhs ≤ ?rhs")
proof -
  have "?lhs ≤ (8 * k) * (k*Inf_para) powr -α"
    using sum_bounded_above [OF ω_upper] card_lattice_layer [OF <k>0>]
assms
    by fastforce
  also have "... = ?rhs"
    using Inf_para_pos by (simp add: powr_diff powr_minus powr_mult divide_simps)
  finally show ?thesis .
qed

lemma lattice_layer_lower:
  assumes "ω ∈ lattice_layer k" and "k > 0"
  shows "(k * (if α ≥ 0 then Inf_para else Sup_para)) powr α ≤ norm
ω powr α"
proof (cases "α ≥ 0")
  case True
  have [simp]: "ω ≠ 0"
    using assms by auto
  show ?thesis
    by (rule powr_mono2)
       (use True lattice_layer_le_norm[of ω k] Inf_para_pos assms in auto)
```

```
next
  case False
  show ?thesis
    by (rule powr_mono2') (use False lattice_layer_ge_norm[of ω k] assms
in auto)
qed

lemma sum_lattice_layer_lower:
  fixes α :: real
  assumes "k > 0"
  defines "C ≡ (if α ≥ 0 then Sup_para else Inf_para)"
  shows "8 * k powr (1-α) * C powr -α ≤ (∑ω ∈ lattice_layer k. norm
ω powr -α)"
    (is "?lhs ≤ ?rhs")
proof -
  from ‹k > 0› have "?lhs = (∑ω∈lattice_layer k. (k * C) powr -α)"
    by (simp add: powr_minus powr_diff field_simps powr_mult card_lattice_layer)
  also have "... ≤ ?rhs"
    unfolding C_def using lattice_layer_lower[of _ k "-α"] ‹k > 0›
    by (cases "α > 0"; intro sum_mono) (auto split: if_splits)
  finally show ?thesis .
qed

lemma converges_absolutely_iff_aux1:
  fixes α :: real
  assumes "α > 2"
  shows   "summable (λi. ∑ω∈lattice_layer (Suc i). 1 / norm ω powr
α)"
proof (rule summable_comparison_test')
  show "norm (∑ω∈lattice_layer (Suc n). 1 / norm ω powr α) ≤
        8 * real (Suc n) powr (1 - α) * Inf_para powr -α" for n
  proof -
    have "norm (∑ω∈lattice_layer (Suc n). 1 / norm ω powr α) =
          (∑ω∈lattice_layer (Suc n). norm ω powr -α)"
      unfolding real_norm_def
      by (subst abs_of_nonneg) (auto intro!: sum_nonneg simp: powr_minus
field_simps)
    also have "... ≤ 8 * real (Suc n) powr (1 - α) * Inf_para powr -α"
      using sum_ω_upper[of α "Suc n"] assms by simp
    finally show ?thesis .
  qed
next
  show "summable (λn. 8 * real (Suc n) powr (1 - α) * Inf_para powr -α)"
    by (subst summable_Suc_iff, intro summable_mult summable_mult2, subst
summable_real_powr_iff)
       (use assms in auto)
qed

lemma converges_absolutely_iff_aux2:
```

```
  fixes α :: real
  assumes "summable (λi. ∑ω∈lattice_layer (Suc i). 1 / norm ω powr
α)"
  shows   "α > 2"
proof -
  define C where "C = (if α > 0 then Sup_para else Inf_para)"
  have "C > 0"
    using Sup_para_pos Inf_para_pos by (auto simp: C_def)

  have "summable (λn. 8 * real (Suc n) powr (1 - α) * C powr -α)"
  proof (rule summable_comparison_test')
    show "norm (8 * real (Suc n) powr (1 - α) * C powr -α) ≤
            (∑ω∈lattice_layer (Suc n). 1 / norm ω powr α)" for n
    proof -
      have "norm (8 * real (Suc n) powr (1 - α) * C powr -α) =
            8 * real (Suc n) powr (1 - α) * C powr -α"
        unfolding real_norm_def by (subst abs_of_nonneg) (auto intro!:
sum_nonneg)
      also have "... ≤ (∑ω∈lattice_layer (Suc n). 1 / norm ω powr α)"
        using sum_lattice_layer_lower[of "Suc n" α]
        by (auto simp: powr_minus field_simps C_def split: if_splits)
      finally show ?thesis .
    qed
  qed fact
  hence "summable (λn. 8 * C powr -α * real n powr (1 - α))"
    by (subst (asm) summable_Suc_iff) (simp add: mult_ac)
  hence "summable (λn. real n powr (1 - α))"
    using ‹C > 0› by (subst (asm) summable_cmult_iff) auto
  thus "α > 2"
    by (subst (asm) summable_real_powr_iff) auto
qed
```

Apostol's Lemma 1

```
lemma converges_absolutely_iff:
  fixes α:: real
  shows "(λω. 1 / norm ω powr α) summable_on Λ* ⟷ α > 2"
    (is "?P ⟷ _")
proof -
  have "(λω. 1 / norm ω powr α) summable_on Λ* ⟷
        (λω. 1 / norm ω powr α) summable_on (⋃i ∈ {0<..}. lattice_layer
i)"
    by (simp add: lattice0_conv_layers)
  also have "... ⟷ (λi. ∑ω∈lattice_layer i. 1 / norm ω powr α) summable_on
{0<..}"
    using lattice_layer_disjoint
    by (intro summable_on_Union_iff has_sum_finite finite_lattice_layer
refl)
       (auto simp: disjoint_family_on_def)
  also have "{0<..} = Suc ‘ UNIV"
```

```
    by (auto simp: image_iff) presburger?
  also have "(λi. ∑ω∈lattice_layer i. 1 / norm ω powr α) summable_on
Suc ‘ UNIV ⟷
             (λi. ∑ω∈lattice_layer (Suc i). 1 / norm ω powr α) summable_on
UNIV"
    by (subst summable_on_reindex) (auto simp: o_def)
  also have "... ⟷ summable (λi. ∑ω∈lattice_layer (Suc i). 1 / norm
ω powr α)"
    by (rule summable_on_UNIV_nonneg_real_iff) (auto intro: sum_nonneg)
  also have "... ⟷ α > 2"
    using converges_absolutely_iff_aux1 converges_absolutely_iff_aux2
by blast
  finally show ?thesis .
qed

lemma bounded_lattice_finite:
  assumes "bounded B"
  shows   "finite (Λ ∩ B)"
  by (meson not_islimpt_lattice inf.bounded_iff islimpt_subset ‹bounded
B›
            bounded_infinite_imp_islimpt bounded_subset eq_refl)

lemma closed_subset_lattice: "Λ' ⊆ Λ ⟹ closed Λ'"
  unfolding closed_limpt using not_islimpt_lattice islimpt_subset by
blast

corollary closed_lattice0: "closed Λ*"
  unfolding lattice0_def by (rule closed_subset_lattice) auto

lemma weierstrass_summand_bound:
  assumes "α ≥ 1" and "R > 0"
  obtains M where
    "M > 0"
    "⋀ω z. ⟦ω ∈ Λ; cmod ω > R; cmod z ≤ R⟧ ⟹ norm (z - ω) powr -α
≤ M * (norm ω powr -α)"
proof -
  obtain m where m: "of_int m > R / norm ω1" "m ≥ 0"
    by (metis ex_less_of_int le_less_trans linear not_le of_int_0_le_iff)
  obtain W where W: "W ∈ (Λ - cball 0 R) ∩ cball 0 (norm W)"
  proof
    show "of_int m * ω1 ∈ (Λ - cball 0 R) ∩ cball 0 (norm (of_int m
* ω1))"
      using m latticeI [of m 0]
      by (simp add: field_simps norm_mult)
  qed
  define PF where "PF ≡ (Λ - cball 0 R) ∩ cball 0 (norm W)"
  have "finite PF"
    by (simp add: PF_def Diff_Int_distrib2 bounded_lattice_finite)
  then have "finite (norm ‘ PF)"
```

```
    by blast
  then obtain r where "r ∈ norm ‘ PF"  "r ≤ norm W" and r: "⋀x. x
∈ norm ‘ PF ⟹ r ≤ x"
    using PF_def W Min_le Min_in by (metis empty_iff image_eqI)
  then obtain ωr where ωr: "ωr ∈ PF" "norm ωr = r"
    by blast
  with assms have "ωr ∈ Λ" "ωr ≠ 0" "r > 0"
    by (auto simp: PF_def)
  have minr: "r ≤ cmod ω" if "ω ∈ Λ" "cmod ω > R" for ω
    using r  ‹r ≤ cmod W› that unfolding PF_def by fastforce
  have "R < r"
    using ωr by (simp add: PF_def)
  with ‹R > 0› have R_iff_r: "cmod ω > R ⟷ cmod ω ≥ r" if "ω ∈ Λ"
for ω
    using that by (auto simp: minr)
  define M where "M ≡ (1 - R/r) powr -α"
  have "M > 0"
    unfolding M_def using ‹R < r› by auto
  moreover
  have "cmod (z - ω) powr -α ≤ M * cmod ω powr -α"
    if "ω ∈ Λ" "cmod z ≤ R" "R < cmod ω" for z ω
  proof -
    have "r ≤ cmod ω"
      using minr that by blast
    then have "ω ≠ 0"
      using ‹0 < r› that by force
    have "1 - R/r ≤ 1 - norm (z/ω)"
      using that ‹0 < r› ‹0 < R› ‹ω ≠ 0› ‹r ≤ cmod ω›
      by (simp add: field_simps norm_divide) (metis mult.commute mult_mono
norm_ge_zero)
    also have "... ≤ norm (1 - z/ω)"
      by (metis norm_one norm_triangle_ineq2)
    also have "... = norm ((z - ω) / ω)"
      by (simp add: ‹ω ≠ 0› norm_minus_commute diff_divide_distrib)
    finally have *: "1 - R/r ≤ norm ((z - ω) / ω)" .
    have ge_rR: "cmod (z - ω) ≥ r - R"
      by (smt (verit) ‹r ≤ cmod ω› norm_minus_commute norm_triangle_ineq2
that(2))
    have "1/M ≤ norm ((z - ω) / ω) powr α"
    proof -
      have "1/M = (1 - R / r) powr α"
        by (simp add: M_def powr_minus_divide)
      also have "... ≤ norm ((z - ω) / ω) powr α"
        using * ‹0 < r› ‹R < r› ‹1 ≤ α› powr_mono2 by force
      finally show ?thesis .
    qed
    then show ?thesis
      using ‹R > 0› ‹M > 0› ‹ω ≠ 0›
      by (simp add: mult.commute divide_simps powr_divide norm_divide
```

```
powr_minus)
  qed
  ultimately show thesis
    using that by presburger
qed
```

Lemma 2 on Apostol p. 8

```
lemma weierstrass_aux_converges_absolutely_in_disk:
  assumes "α > 2" and "R > 0" and "z ∈ cball 0 R"
  shows "(λω. cmod (z - ω) powr -α) summable_on (Λ - cball 0 R)"
proof -
  have Λ: "Λ - cball 0 R ⊆ Λ*"
    using assms by force
  obtain M where "M > 0" and M: "⋀ω. ⟦ω ∈ Λ; cmod ω > R⟧ ⟹ cmod(z
- ω) powr -α ≤ M * (cmod ω powr -α)"
    using weierstrass_summand_bound assms
    by (smt (verit, del_insts) less_eq_real_def mem_cball_0 one_le_numeral)
  have §: "((λω. 1 / cmod ω powr α) summable_on Λ*)"
    using ‹α > 2› converges_absolutely_iff by blast
  have "(λω. M * norm ω powr -α) summable_on Λ*"
    using summable_on_cmult_right [OF §] by (simp add: powr_minus field_class.field_divide_
  with Λ have "(λω. M * norm ω powr -α) summable_on Λ - cball 0 R"
    using summable_on_subset_banach by blast
  then show ?thesis
    by (rule summable_on_comparison_test) (use M in auto)
qed

lemma weierstrass_aux_converges_absolutely_in_disk':
  fixes α :: nat and R :: real and z:: complex
  assumes "α > 2" and "R > 0" and "z ∈ cball 0 R"
  shows "(λω. 1 / norm (z - ω) ^ α) summable_on (Λ - cball 0 R)"
proof -
  have "(λω. norm (z - ω) powr -of_nat α) summable_on (Λ - cball 0 R)"
    using assms by (intro weierstrass_aux_converges_absolutely_in_disk)
auto
  also have "?this ⟷ ?thesis"
    unfolding powr_minus
    by (intro summable_on_cong refl, subst powr_realpow)
       (use assms in ‹auto simp: field_simps›)
  finally show ?thesis .
qed

lemma weierstrass_aux_converges_in_disk':
  fixes α :: nat and R :: real and z:: complex
  assumes "α > 2" and "R > 0" and "z ∈ cball 0 R"
  shows "(λω. 1 / (z - ω) ^ α) summable_on (Λ - cball 0 R)"
  by (rule abs_summable_summable)
     (use weierstrass_aux_converges_absolutely_in_disk'[OF assms]
      in ‹simp add: norm_divide norm_power›)
```

```
lemma weierstrass_aux_converges_absolutely:
  fixes α :: real
  assumes "α > 2" and "Λ' ⊆ Λ"
  shows "(λω. norm (z - ω) powr -α) summable_on Λ'"
proof (cases "z = 0")
  case True
  hence "(λω. norm (z - ω) powr -α) summable_on Λ*"
    using converges_absolutely_iff[of α] assms by (simp add: powr_minus
field_simps)
  hence "(λω. norm (z - ω) powr -α) summable_on Λ"
    by (simp add: lattice_lattice0 summable_on_insert_iff)
  thus ?thesis
    by (rule summable_on_subset_banach) fact
next
  case [simp]: False
  define R where "R = norm z"
  have "(λω. norm (z - ω) powr -α) summable_on (Λ - cball 0 R)"
    using assms by (intro weierstrass_aux_converges_absolutely_in_disk)
(auto simp: R_def)
  hence "(λω. norm (z - ω) powr -α) summable_on (Λ - cball 0 R) ∪ (Λ
∩ cball 0 R)"
    by (intro bounded_lattice_finite summable_on_union[OF _ summable_on_finite])
auto
  also have "... = Λ"
    by blast
  finally show ?thesis
    by (rule summable_on_subset) fact
qed

lemma weierstrass_aux_converges_absolutely':
  fixes α :: nat
  assumes "α > 2" and "Λ' ⊆ Λ"
  shows "(λω. 1 / norm (z - ω) ^ α) summable_on Λ'"
  using weierstrass_aux_converges_absolutely[of "of_nat α" Λ' z] assms
  by (simp add: powr_nat' powr_minus field_simps norm_power norm_divide
powr_realpow')

lemma weierstrass_aux_converges:
  fixes α :: real
  assumes "α > 2" and "Λ' ⊆ Λ"
  shows "(λω. (z - ω) powr -α) summable_on Λ'"
  by (rule abs_summable_summable)
     (use weierstrass_aux_converges_absolutely[OF assms]
      in ‹simp add: norm_divide norm_powr_real_powr'›)

lemma weierstrass_aux_converges':
  fixes α :: nat
  assumes "α > 2" and "Λ' ⊆ Λ"
```

```
  shows "(λω. 1 / (z - ω) ^ α) summable_on Λ'"
  using weierstrass_aux_converges[of "of_nat α" Λ' z] assms
  by (simp add: powr_nat' powr_minus field_simps)

lemma
  fixes α R :: real
  assumes "α > 2" "R > 0"
  shows weierstrass_aux_converges_absolutely_uniformly_in_disk:
          "uniform_limit (cball 0 R)
                         (λX z. ∑ω∈X. norm ((z - ω) powr -α))
                         (λz. ∑∞ω∈Λ-cball 0 R. norm ((z - ω) powr -α))
                         (finite_subsets_at_top (Λ - cball 0 R))" (is
?th1)
  and   weierstrass_aux_converges_uniformly_in_disk:
          "uniform_limit (cball 0 R)
                         (λX z. ∑ω∈X. (z - ω) powr -α)
                         (λz. ∑∞ω∈Λ-cball 0 R. (z - ω) powr -α)
                         (finite_subsets_at_top (Λ - cball 0 R))" (is
?th2)
proof -
  obtain M where M:
    "M > 0" "⋀ω z. ⟦ω ∈ Λ; norm ω > R; norm z ≤ R⟧ ⟹ norm (z - ω)
powr -α ≤ M * norm ω powr -α"
    using weierstrass_summand_bound[of α R] assms by auto

  have 1: "(λω. M * norm ω powr -α) summable_on (Λ - cball 0 R)"
  proof -
    have "(λω. 1 / norm ω powr α) summable_on Λ*"
      using assms by (subst converges_absolutely_iff) auto
    hence "(λω. M * norm ω powr -α) summable_on Λ*"
      by (intro summable_on_cmult_right) (auto simp: powr_minus field_simps)
    thus "(λω. M * norm ω powr -α) summable_on (Λ - cball 0 R)"
      by (rule summable_on_subset) (use assms in ‹auto simp: lattice0_def›)
  qed

  have 2: "norm ((z - ω) powr -α) ≤ M * norm ω powr -α"
    if "ω ∈ Λ - cball 0 R" "z ∈ cball 0 R" for ω z
  proof -
    have "norm ((z - ω) powr -α) = norm (z - ω) powr -α"
      by (auto simp add: powr_def)
    also have "... ≤ M * norm ω powr -α"
      using that by (intro M) auto
    finally show "norm ((z - ω) powr -α) ≤ M * norm ω powr -α"
      by simp
  qed

  show ?th1 ?th2
    by (rule Weierstrass_m_test_general[OF _ 1]; use 2 in simp)+
qed
```

```
lemma
  fixes n :: nat and R :: real
  assumes "n > 2" "R > 0"
  shows weierstrass_aux_converges_absolutely_uniformly_in_disk':
         "uniform_limit (cball 0 R)
                        (λX z. ∑ω∈X. norm (1 / (z - ω) ^ n))
                        (λz. ∑∞ω∈Λ-cball 0 R. norm (1 / (z - ω) ^
n))
                        (finite_subsets_at_top (Λ - cball 0 R))" (is
?th1)
  and   weierstrass_aux_converges_uniformly_in_disk':
         "uniform_limit (cball 0 R)
                        (λX z. ∑ω∈X. 1 / (z - ω) powr n)
                        (λz. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω) ^ n)
                        (finite_subsets_at_top (Λ - cball 0 R))" (is
?th2)
proof -
  have "uniform_limit (cball 0 R)
          (λX z. ∑ω∈X. norm ((z - ω) powr -real n))
          (λz. ∑∞ω∈Λ-cball 0 R. norm ((z - ω) powr -real n))
          (finite_subsets_at_top (Λ - cball 0 R))"
    by (rule weierstrass_aux_converges_absolutely_uniformly_in_disk) (use
assms in auto)
  also have "?this ⟷ ?th1"
    by (intro uniform_limit_cong eventually_finite_subsets_at_top_weakI
sum.cong ballI)
       (auto simp: norm_divide norm_power powr_minus field_simps powr_nat
             intro!: infsum_cong)
  finally show ?th1 .
next
  have "uniform_limit (cball 0 R)
          (λX z. ∑ω∈X. (z - ω) powr -of_nat n)
          (λz. ∑∞ω∈Λ-cball 0 R. (z - ω) powr -of_nat n)
          (finite_subsets_at_top (Λ - cball 0 R))"
    using weierstrass_aux_converges_uniformly_in_disk[of "of_nat n" R]
assms by auto
  also have "?this ⟷ ?th2"
    by (intro uniform_limit_cong eventually_finite_subsets_at_top_weakI
sum.cong ballI)
       (auto simp: norm_divide norm_power powr_minus field_simps powr_nat
             intro!: infsum_cong)
  finally show ?th2 .
qed


definition eisenstein_fun_aux :: "nat ⇒ complex ⇒ complex" where
  "eisenstein_fun_aux n z =
     (if n = 0 then 1 else if n < 3 ∨ z ∈ Λ* then 0 else (∑∞ω∈Λ*.
```

```
1 / (z - ω) ^ n))"

lemma eisenstein_fun_aux_at_pole_eq_0: "n > 0 ⟹ z ∈ Λ* ⟹ eisenstein_fun_aux
n z = 0"
  by (simp add: eisenstein_fun_aux_def)

lemma eisenstein_fun_aux_has_sum:
  assumes "n ≥ 3" "z ∉ Λ*"
  shows   "((λω. 1 / (z - ω) ^ n) has_sum eisenstein_fun_aux n z) Λ*"
proof -
  have "eisenstein_fun_aux n z = (∑∞ω∈Λ*. 1 / (z - ω) ^ n)"
    using assms by (simp add: eisenstein_fun_aux_def)
  also have "((λω. 1 / (z - ω) ^ n) has_sum ...) Λ*"
    using assms by (intro has_sum_infsum weierstrass_aux_converges') (auto
simp: lattice0_def)
  finally show ?thesis .
qed

lemma eisenstein_fun_aux_minus: "eisenstein_fun_aux n (-z) = (-1) ^ n
* eisenstein_fun_aux n z"
proof (cases "n < 3 ∨ z ∈ Λ*")
  case False
  have "eisenstein_fun_aux n (-z) = (∑∞ω∈Λ*. 1 / (- z - ω) ^ n)"
    using False by (auto simp: eisenstein_fun_aux_def lattice0_def uminus_in_lattice_iff)
  also have "... = (∑∞ω∈uminus ‘ Λ*. 1 / (ω - z) ^ n)"
    by (subst infsum_reindex) (auto simp: o_def minus_diff_commute inj_on_def)
  also have "uminus ‘ Λ* = Λ*"
    by (auto simp: lattice0_def uminus_in_lattice_iff image_def intro:
bexI[of _ "-x" for x])
  also have "(λω. 1 / (ω - z) ^ n) = (λω. (-1) ^ n * (1 / (z - ω) ^ n))"
  proof
    fix ω :: complex
    have "1 / (ω - z) ^ n = (1 / (ω - z)) ^ n"
      by (simp add: power_divide)
    also have "1 / (ω - z) = (-1) / (z - ω)"
      by (simp add: divide_simps)
    also have "... ^ n = (-1) ^ n * (1 / (z - ω) ^ n)"
      by (subst power_divide) auto
    finally show "1 / (ω - z) ^ n = (-1) ^ n * (1 / (z - ω) ^ n)" .
  qed
  also have "(∑∞ω∈Λ*. (-1) ^ n * (1 / (z - ω) ^ n)) = (-1) ^ n * eisenstein_fun_aux
n z"
    using False by (subst infsum_cmult_right') (auto simp: eisenstein_fun_aux_def)
  finally show ?thesis .
qed (auto simp: eisenstein_fun_aux_def lattice0_def uminus_in_lattice_iff)

lemma eisenstein_fun_aux_even_minus: "even n ⟹ eisenstein_fun_aux
n (-z) = eisenstein_fun_aux n z"
  by (simp add: eisenstein_fun_aux_minus)
```

```
lemma eisenstein_fun_aux_odd_minus: "odd n ⟹ eisenstein_fun_aux n
(-z) = -eisenstein_fun_aux n z"
  by (simp add: eisenstein_fun_aux_minus)


lemma eisenstein_fun_aux_has_field_derivative_aux:
  fixes α :: nat and R :: real
  defines "F ≡ (λα z. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω) ^ α)"
  assumes "α > 2" "R > 0" "w ∈ ball 0 R"
  shows   "(F α has_field_derivative -of_nat α * F (Suc α) w) (at w)"
proof -
  define α' where "α' = α - 1"
  have α': "α = Suc α'"
    using assms by (simp add: α'_def)
  have 1: "∀F n in finite_subsets_at_top (Λ - cball 0 R).
           continuous_on (cball 0 R) (λz. ∑ω∈n. 1 / (z - ω) ^ α) ∧
           (∀z∈ball 0 R. ((λz. ∑ω∈n. 1 / (z - ω) ^ α) has_field_derivative
(∑ω∈n. -α / (z - ω) ^ Suc α)) (at z))"

  proof (intro eventually_finite_subsets_at_top_weakI conjI continuous_intros
derivative_intros ballI)
    fix z::complex and X n
    assume "finite X"  "X ⊆ Λ - cball 0 R"
        and "z ∈ ball 0 R"  "n ∈ X"
    then show "((λz. 1 / (z - n) ^ α) has_field_derivative of_int (-
int α) / (z - n) ^ Suc α) (at z)"
     apply (auto intro!: derivative_eq_intros simp: divide_simps)
      apply (simp add: algebra_simps α')
      done
  qed auto

  have "uniform_limit (cball 0 R)
                      (λX z. ∑ω∈X. (z - ω) powr of_real (-of_nat α))
                      (λz. ∑∞ω∈Λ-cball 0 R. (z - ω) powr of_real (-of_nat
α))
                      (finite_subsets_at_top (Λ - cball 0 R))"
    using assms by (intro weierstrass_aux_converges_uniformly_in_disk)
auto
  also have "?this ⟷ uniform_limit (cball 0 R) (λX z. ∑ω∈X. 1 / (z
- ω) ^ α) (F α)
                         (finite_subsets_at_top (Λ - cball 0 R))"
    using assms unfolding F_def
    by (intro uniform_limit_cong eventually_finite_subsets_at_top_weakI)
       (auto simp: powr_minus powr_nat field_simps intro!: sum.cong infsum_cong)
  finally have 2: ... .

  have 3: "finite_subsets_at_top (Λ - cball 0 R) ≠ bot"
    by simp
```

```
  obtain g where g: "continuous_on (cball 0 R) (F α)"
                  "⋀w. w ∈ ball 0 R ⟹ (F α has_field_derivative
g w) (at w) ∧
                       ((λω. -of_nat α / (w - ω) ^ Suc α) has_sum g
w) (Λ - cball 0 R)"
    unfolding has_sum_def using has_complex_derivative_uniform_limit[OF
1 2 3 <R > 0>] by auto

  have "((λω. -of_nat α * (1 / (w - ω) ^ Suc α)) has_sum -of_nat α *
F (Suc α) w) (Λ - cball 0 R)"
    unfolding F_def using assms
    by (intro has_sum_cmult_right has_sum_infsum weierstrass_aux_converges') 
auto
  moreover have "((λω.  -of_nat α * (1 / (w - ω) ^ Suc α)) has_sum
g w) (Λ - cball 0 R)"
    using g(2)[of w] assms by simp
  ultimately have "g w = -of_nat α * F (Suc α) w"
    by (metis infsumI)
  thus ?thesis
    using g(2)[of w] assms by (simp add: F_def)
qed

lemma eisenstein_fun_aux_has_field_derivative:
  assumes z: "z ∉ Λ*" and n: "n ≥ 3"
  shows   "(eisenstein_fun_aux n has_field_derivative -of_nat n * eisenstein_fun_aux
(Suc n) z) (at z)"
proof -
  define R where "R = norm z + 1"
  have R: "R > 0" "norm z < R"
    by (auto simp: R_def add_nonneg_pos)
  have "finite (Λ ∩ cball 0 R)"
    by (simp add: bounded_lattice_finite)
  moreover have "Λ* ∩ cball 0 R ⊆ Λ ∩ cball 0 R"
    unfolding lattice0_def by blast
  ultimately have fin: "finite (Λ* ∩ cball 0 R)"
    using finite_subset by blast
  define n' where "n' = n - 1"
  from n have n': "n = Suc n'"
    by (simp add: n'_def)

  define F1 where "F1 = (λn z. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω) ^ n)"
  define F2 where "F2 = (λn z. ∑ω∈Λ*∩cball 0 R. 1 / (z - ω) ^ n)"

  have "(F1 n has_field_derivative -of_nat n * F1 (Suc n) z) (at z)"
    unfolding F1_def
    by (rule eisenstein_fun_aux_has_field_derivative_aux) (use n in <auto
simp: R_def add_nonneg_pos>)
  moreover have "(F2 n has_field_derivative -of_nat n * F2 (Suc n) z)
```

```
(at z)"
    unfolding F2_def sum_distrib_left lattice0_def
    by (rule derivative_eq_intros refl sum.cong | use R z n in ‹force
simp: lattice0_def›)+
       (simp add: divide_simps power3_eq_cube power4_eq_xxxx n')
  ultimately have "((λz. F1 n z + F2 n z) has_field_derivative
                    (-of_nat n * F1 (Suc n) z) + (-of_nat n * F2 (Suc
n) z)) (at z)"
    by (intro derivative_intros)
  also have "?this ⟷ (eisenstein_fun_aux n has_field_derivative (-of_nat
n * F1 (Suc n) z) + (-of_nat n * F2 (Suc n) z)) (at z)"
  proof (intro has_field_derivative_cong_ev refl)
    have "eventually (λz'. z' ∈ -Λ*) (nhds z)"
      using z by (intro eventually_nhds_in_open) (auto simp: closed_lattice0)
    thus "∀F x in nhds z. x ∈ UNIV ⟶ F1 n x + F2 n x = eisenstein_fun_aux
n x"
    proof eventually_elim
      case (elim z)
      have "((λω. 1 / (z - ω) ^ n) has_sum (F1 n z + F2 n z)) ((Λ - cball
0 R) ∪ (Λ* ∩ cball 0 R))"
        unfolding F1_def F2_def using R fin n
        by (intro has_sum_Un_disjoint[OF has_sum_infsum has_sum_finite]
                  summable_on_subset[OF weierstrass_aux_converges']) auto
      also have "(Λ - cball 0 R) ∪ (Λ* ∩ cball 0 R) = Λ*"
        using R unfolding lattice0_def by auto
      finally show ?case using elim n
        unfolding F1_def F2_def by (simp add: infsumI eisenstein_fun_aux_def)
    qed
  qed auto
  also have "(-of_nat n * F1 (Suc n) z) + (-of_nat n * F2 (Suc n) z) =
-of_nat n * (F1 (Suc n) z + F2 (Suc n) z)"
    by (simp add: algebra_simps)
  also have "F1 (Suc n) z + F2 (Suc n) z = eisenstein_fun_aux (Suc n)
z"
  proof -
    have "((λω. 1 / (z - ω) ^ Suc n) has_sum (F1 (Suc n) z + F2 (Suc
n) z)) ((Λ - cball 0 R) ∪ (Λ* ∩ cball 0 R))"
      unfolding F1_def F2_def using R fin n
      by (intro has_sum_Un_disjoint[OF has_sum_infsum has_sum_finite]
weierstrass_aux_converges') auto
    also have "(Λ - cball 0 R) ∪ (Λ* ∩ cball 0 R) = Λ*"
      using R unfolding lattice0_def by auto
    finally show ?thesis using n z
      unfolding F1_def F2_def eisenstein_fun_aux_def by (simp add: infsumI)
  qed
  finally show ?thesis .
qed

lemmas eisenstein_fun_aux_has_field_derivative' [derivative_intros] =
```

```
  DERIV_chain2[OF eisenstein_fun_aux_has_field_derivative]

lemma higher_deriv_eisenstein_fun_aux:
  assumes z: "z ∉ Λ*" and n: "n ≥ 3"
  shows   "(deriv ^^ m) (eisenstein_fun_aux n) z =
             (-1) ^ m * pochhammer (of_nat n) m * eisenstein_fun_aux (n
+ m) z"
  using z n
proof (induction m arbitrary: z n)
  case 0
  thus ?case by simp
next
  case (Suc m z n)
  have ev: "eventually (λz. z ∈ -Λ*) (nhds z)"
    using Suc.prems closed_lattice0 by (intro eventually_nhds_in_open)
auto
  have "(deriv ^^ Suc m) (eisenstein_fun_aux n) z = deriv ((deriv ^^ m)
(eisenstein_fun_aux n)) z"
    by simp
  also have "... = deriv (λz. (-1)^ m * pochhammer (of_nat n) m * eisenstein_fun_aux
(n + m) z) z"
    by (intro deriv_cong_ev eventually_mono[OF ev]) (use Suc in auto)
  also have "... = (-1) ^ Suc m * pochhammer (of_nat n) (Suc m) * eisenstein_fun_aux
(Suc (n + m)) z"
    using Suc.prems
    by (intro DERIV_imp_deriv)
       (auto intro!: derivative_eq_intros simp: pochhammer_Suc algebra_simps)
  finally show ?case
    by simp
qed

lemma eisenstein_fun_aux_holomorphic: "eisenstein_fun_aux n holomorphic_on
-Λ*"
proof (cases "n ≥ 3")
  case True
  thus ?thesis
    using closed_lattice0
    by (subst holomorphic_on_open) (auto intro!: eisenstein_fun_aux_has_field_derivative)
next
  case False
  thus ?thesis
    by (cases "n = 0") (auto simp: eisenstein_fun_aux_def [abs_def])
qed

lemma eisenstein_fun_aux_holomorphic' [holomorphic_intros]:
  assumes "f holomorphic_on A" "⋀z. z ∈ A ⟹ f z ∉ Λ*"
  shows   "(λz. eisenstein_fun_aux n (f z)) holomorphic_on A"
proof -
  have "eisenstein_fun_aux n ∘ f holomorphic_on A"
```

```
    by (rule holomorphic_on_compose_gen assms eisenstein_fun_aux_holomorphic)+
(use assms in auto)
  thus ?thesis by (simp add: o_def)
qed

lemma eisenstein_fun_aux_analytic: "eisenstein_fun_aux n analytic_on
-Λ*"
  by (simp add: analytic_on_open closed_lattice0 open_Compl eisenstein_fun_aux_holomorphic)


lemma eisenstein_fun_aux_analytic' [analytic_intros]:
  assumes "f analytic_on A" "⋀z. z ∈ A ⟹ f z ∉ Λ*"
  shows   "(λz. eisenstein_fun_aux n (f z)) analytic_on A"
proof -
  have "eisenstein_fun_aux n ∘ f analytic_on A"
    by (rule analytic_on_compose_gen assms eisenstein_fun_aux_analytic)+
(use assms in auto)
  thus ?thesis by (simp add: o_def)
qed

lemma eisenstein_fun_aux_continuous_on: "continuous_on (-Λ*) (eisenstein_fun_aux
n)"
  using holomorphic_on_imp_continuous_on eisenstein_fun_aux_holomorphic
by blast

lemma eisenstein_fun_aux_continuous_on' [continuous_intros]:
  assumes "continuous_on A f" "⋀z. z ∈ A ⟹ f z ∉ Λ*"
  shows   "continuous_on A (λz. eisenstein_fun_aux n (f z))"
  by (rule continuous_on_compose2[OF eisenstein_fun_aux_continuous_on
assms(1)]) (use assms in auto)

lemma weierstrass_aux_translate:
  fixes α :: real
  assumes "α > 2"
  shows   "(∑∞ω∈Λ. (z + w - ω) powr -α) = (∑∞ω∈(+) (-w) ‘ Λ. (z
- ω) powr -α)"
  by (subst infsum_reindex) (auto simp: o_def algebra_simps)

lemma weierstrass_aux_holomorphic:
  assumes "α > 2" "Λ' ⊆ Λ" "finite (Λ - Λ')"
  shows   "(λz. ∑∞ω∈Λ'. 1 / (z - ω) ^ α) holomorphic_on -Λ'"
proof -
  define M where "M = Max (insert 0 (norm ‘ (Λ - Λ')))"
  have M: "M ≥ 0" "⋀ω. ω ∈ Λ - Λ' ⟹ M ≥ norm ω"
    using assms by (auto simp: M_def)
  have [simp]: "closed Λ'"
    using assms(2) by (rule closed_subset_lattice)

  have *: "(λz. ∑∞ω∈Λ'. 1 / (z - ω) ^ α) holomorphic_on ball 0 R -
```

```
Λ'" if R: "R > M" for R
  proof -
    define F where "F = (λα z. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω) ^ α)"
    define G where "G = (λα z. ∑ω∈Λ'∩cball 0 R. 1 / (z - ω) ^ α)"

    have "(F α has_field_derivative -of_nat α * F (Suc α) z) (at z)"
if z: "z ∈ ball 0 R" for z
      unfolding F_def using assms R M(1) z by (intro eisenstein_fun_aux_has_field_derivativ
auto
    hence "F α holomorphic_on ball 0 R - Λ'"
      using holomorphic_on_open ‹closed Λ'› by blast
    hence "(λz. F α z + G α z) holomorphic_on ball 0 R - Λ'"
      unfolding G_def using assms M R by (intro holomorphic_intros) auto
    also have "(λz. F α z + G α z) = (λz. ∑∞ω∈Λ'. 1 / (z - ω) ^ α)"
    proof
      fix z :: complex
      have "finite (Λ ∩ cball 0 R)"
        by (intro bounded_lattice_finite) auto
      moreover have "Λ' ∩ cball 0 R ⊆ Λ ∩ cball 0 R"
        using assms by blast
      ultimately have "finite (Λ' ∩ cball 0 R)"
        using finite_subset by blast
      hence "((λω. 1 / (z - ω) ^ α) has_sum (F α z + G α z)) ((Λ - cball
0 R) ∪ (Λ' ∩ cball 0 R))"
        unfolding F_def G_def using assms
        by (intro has_sum_Un_disjoint[OF has_sum_infsum has_sum_finite]
weierstrass_aux_converges') auto
      also have "(Λ - cball 0 R) ∪ (Λ' ∩ cball 0 R) = Λ'"
        using M R assms by (force simp: not_le)
      finally show "F α z + G α z = (∑∞ω∈Λ'. 1 / (z - ω) ^ α)"
        by (simp add: infsumI)
    qed
    finally show ?thesis .
  qed

  have "(λz. ∑∞ω∈Λ'. 1 / (z - ω) ^ α) holomorphic_on (⋃R∈{R. R >
M}. ball 0 R - Λ')"
    by (rule holomorphic_on_UN_open) (use * ‹closed Λ'› in auto)
  also have "... = (⋃R∈{R. R > M}. ball 0 R) - Λ'"
    by blast
  also have "(⋃R∈{R. R > M}. ball 0 R) = (UNIV :: complex set)"
  proof (safe intro!: UN_I)
    fix z :: complex
    show "norm z + M + 1 > M" "z ∈ ball 0 (norm z + M + 1)"
      using M(1) by (auto intro: add_nonneg_pos)
  qed auto
  finally show ?thesis
    by (simp add: Compl_eq_Diff_UNIV)
qed
```

definition *eisenstein_fun :: "nat ⇒ complex ⇒ complex"* **where**
"eisenstein_fun n z = (if n < 3 ∨ z ∈ Λ then 0 else (∑∞ω∈Λ. 1 / (z - ω) ^ n))"

lemma *eisenstein_fun_has_sum:*
"n ≥ 3 ⟹ z ∉ Λ ⟹ ((λω. 1 / (z - ω) ^ n) has_sum eisenstein_fun n z) Λ"
unfolding *eisenstein_fun_def* **by** *(auto intro!: has_sum_infsum weierstrass_aux_converges')*

lemma *eisenstein_fun_at_pole_eq_0: "z ∈ Λ ⟹ eisenstein_fun n z = 0"*
by *(simp add: eisenstein_fun_def)*

lemma *eisenstein_fun_conv_eisenstein_fun_aux:*
assumes *"n ≥ 3" "z ∉ Λ"*
shows *"eisenstein_fun n z = eisenstein_fun_aux n z + 1 / z ^ n"*
proof -
from *assms* **have** *"eisenstein_fun n z = (∑∞ω∈insert 0 Λ*. 1 / (z - ω) ^ n)"*
by *(simp add: eisenstein_fun_def lattice_lattice0)*
also from *assms* **have** *"... = (∑∞ω∈Λ*. 1 / (z - ω) ^ n) + 1 / z ^ n"*
by *(subst infsum_insert) (auto intro!: weierstrass_aux_converges' simp: lattice_lattice0)*
also from *assms* **have** *"... = eisenstein_fun_aux n z + 1 / z ^ n"*
by *(simp add: eisenstein_fun_aux_def lattice_lattice0)*
finally show *?thesis* .
qed

lemma *eisenstein_fun_altdef:*
"eisenstein_fun n z = (if n < 3 ∨ z ∈ Λ then 0 else eisenstein_fun_aux n z + 1 / z ^ n)"
using *eisenstein_fun_conv_eisenstein_fun_aux[of n z]*
by *(auto simp: eisenstein_fun_def eisenstein_fun_aux_def lattice0_def)*

lemma *eisenstein_fun_minus: "eisenstein_fun n (-z) = (-1) ^ n * eisenstein_fun n z"*
by *(auto simp: eisenstein_fun_altdef eisenstein_fun_aux_minus lattice0_def uminus_in_lattice_iff*
power_minus' divide_simps)
(auto simp: algebra_simps)

lemma *eisenstein_fun_even_minus: "even n ⟹ eisenstein_fun n (-z) = eisenstein_fun n z"*
by *(simp add: eisenstein_fun_minus)*

lemma *eisenstein_fun_odd_minus: "odd n ⟹ eisenstein_fun n (-z) = -eisenstein_fun n z"*
by *(simp add: eisenstein_fun_minus)*

```
lemma eisenstein_fun_has_field_derivative:
  assumes "n ≥ 3" "z ∉ Λ"
  shows   "(eisenstein_fun n has_field_derivative -of_nat n * eisenstein_fun
(Suc n) z) (at z)"
proof -
  define n' where "n' = n - 1"
  have n': "n = Suc n'"
    using assms by (simp add: n'_def)
  have ev: "eventually (λz. z ∈ -Λ) (nhds z)"
    using assms closed_lattice by (intro eventually_nhds_in_open) auto

  have "((λz. eisenstein_fun_aux n z + 1 / z ^ n) has_field_derivative
        -of_nat n * eisenstein_fun (Suc n) z) (at z)"
    using assms
    apply (auto intro!: derivative_eq_intros)
     apply (auto simp: eisenstein_fun_conv_eisenstein_fun_aux lattice_lattice0
field_simps n')
    done
  also have "?this ⟷ ?thesis"
    using assms by (intro has_field_derivative_cong_ev refl eventually_mono[OF
ev])
                   (auto simp: eisenstein_fun_conv_eisenstein_fun_aux)
  finally show ?thesis .
qed

lemmas eisenstein_fun_has_field_derivative' [derivative_intros] =
  DERIV_chain2[OF eisenstein_fun_has_field_derivative]

lemma eisenstein_fun_holomorphic: "eisenstein_fun n holomorphic_on -Λ"
proof (cases "n ≥ 3")
  case True
  thus ?thesis using closed_lattice
    by (subst holomorphic_on_open) (auto intro!: eisenstein_fun_has_field_derivative)
qed (auto simp: eisenstein_fun_def [abs_def])

lemma higher_deriv_eisenstein_fun:
  assumes z: "z ∉ Λ" and n: "n ≥ 3"
  shows   "(deriv ^^ m) (eisenstein_fun n) z =
             (-1) ^ m * pochhammer (of_nat n) m * eisenstein_fun (n +
m) z"
  using z n
proof (induction m arbitrary: z n)
  case 0
  thus ?case by simp
next
  case (Suc m z n)
  have ev: "eventually (λz. z ∈ -Λ) (nhds z)"
    using Suc.prems closed_lattice by (intro eventually_nhds_in_open)
```

```
auto
  have "(deriv ^^ Suc m) (eisenstein_fun n) z = deriv ((deriv ^^ m) (eisenstein_fun
n)) z"
    by simp
  also have "... = deriv (λz. (-1)^ m * pochhammer (of_nat n) m * eisenstein_fun
(n + m) z) z"
    by (intro deriv_cong_ev eventually_mono[OF ev]) (use Suc in auto)
  also have "... = (-1) ^ Suc m * pochhammer (of_nat n) (Suc m) * eisenstein_fun
(Suc (n + m)) z"
    using Suc.prems
    by (intro DERIV_imp_deriv)
       (auto intro!: derivative_eq_intros simp: pochhammer_Suc algebra_simps)
  finally show ?case
    by simp
qed

lemma eisenstein_fun_holomorphic' [holomorphic_intros]:
  assumes "f holomorphic_on A" "⋀z. z ∈ A ⟹ n < 3 ∨ f z ∉ Λ"
  shows   "(λz. eisenstein_fun n (f z)) holomorphic_on A"
proof (cases "n ≥ 3")
  case True
  have "eisenstein_fun n ∘ f holomorphic_on A"
    by (rule holomorphic_on_compose_gen assms eisenstein_fun_holomorphic)+
(use assms True in auto)
  thus ?thesis by (simp add: o_def)
qed (auto simp: eisenstein_fun_def)

lemma eisenstein_fun_analytic: "eisenstein_fun n analytic_on -Λ"
  by (simp add: analytic_on_open closed_lattice open_Compl eisenstein_fun_holomorphic)


lemma eisenstein_fun_analytic' [analytic_intros]:
  assumes "f analytic_on A" "⋀z. z ∈ A ⟹ n < 3 ∨ f z ∉ Λ"
  shows   "(λz. eisenstein_fun n (f z)) analytic_on A"
proof (cases "n ≥ 3")
  case True
  have "eisenstein_fun n ∘ f analytic_on A"
    by (rule analytic_on_compose_gen assms True eisenstein_fun_analytic)+
(use assms True in auto)
  thus ?thesis by (simp add: o_def)
qed (auto simp: eisenstein_fun_def)

lemma eisenstein_fun_continuous_on: "n ≥ 3 ⟹ continuous_on (-Λ) (eisenstein_fun
n)"
  using holomorphic_on_imp_continuous_on eisenstein_fun_holomorphic by
blast

lemma eisenstein_fun_continuous_on' [continuous_intros]:
  assumes "continuous_on A f" "⋀z. z ∈ A ⟹ n < 3 ∨ f z ∉ Λ"
```

```
  shows   "continuous_on A (λz. eisenstein_fun n (f z))"
proof (cases "n ≥ 3")
  case True
  show ?thesis
    by (rule continuous_on_compose2[OF eisenstein_fun_continuous_on assms(1)])
       (use assms True in auto)
qed (auto simp: eisenstein_fun_def)

sublocale eisenstein_fun: complex_lattice_periodic ω1 ω2 "eisenstein_fun
n"
proof
  have *: "eisenstein_fun n (w + z) = eisenstein_fun n w" if "z ∈ Λ"
for w z
  proof (cases "n ≥ 3 ∧ w ∉ Λ")
    case True
    show ?thesis
    proof (rule has_sum_unique)
      show "((λω. 1 / (w - ω) ^ n) has_sum eisenstein_fun n w) Λ"
        by (rule eisenstein_fun_has_sum) (use True in auto)
    next
      have "((λω. 1 / (w + z - ω) ^ n) has_sum eisenstein_fun n (w +
z)) Λ"
        by (rule eisenstein_fun_has_sum) (use True that in auto)
      also have "?this ⟷ ((λω. 1 / (w - ω) ^ n) has_sum eisenstein_fun
n (w + z)) Λ"
        by (rule has_sum_reindex_bij_witness[of _ "(+) z" "(+) (-z)"])
           (use that in ‹auto intro!: lattice_intros simp: algebra_simps›)
      finally show "((λω. 1 / (w - ω) ^ n) has_sum eisenstein_fun n (w
+ z)) Λ" .
    qed
  qed (use that in ‹auto simp: eisenstein_fun_def›)
  show "eisenstein_fun n (z + ω1) = eisenstein_fun n z"
       "eisenstein_fun n (z + ω2) = eisenstein_fun n z" for z
    by (rule *; simp)+
qed

lemma is_pole_eisenstein_fun:
  assumes "n ≥ 3" "z ∈ Λ"
  shows   "is_pole (eisenstein_fun n) z"
proof -
  have "eisenstein_fun_aux n -0→ eisenstein_fun_aux n 0"
    by (rule isContD, rule analytic_at_imp_isCont) (auto intro!: analytic_intros)
  moreover have "is_pole (λw. 1 / w ^ n :: complex) 0"
    using assms is_pole_inverse_power[of n 0] by simp
  ultimately have "is_pole (λw. eisenstein_fun_aux n w + 1 / w ^ n) 0"
    unfolding is_pole_def by (rule tendsto_add_filterlim_at_infinity)
  also have "eventually (λw. w ∉ Λ) (at 0)"
    using not_islimpt_lattice by (auto simp: islimpt_iff_eventually)
  hence "eventually (λw. eisenstein_fun_aux n w + 1 / w ^ n = eisenstein_fun
```

```
n w) (at 0)"
    by eventually_elim (use assms in ‹auto simp: eisenstein_fun_altdef›)
  hence "is_pole (λw. eisenstein_fun_aux n w + 1 / w ^ n) 0 ⟷ is_pole
(eisenstein_fun n) 0"
    by (intro is_pole_cong) auto
  also have "eisenstein_fun n = eisenstein_fun n ∘ (λw. w + z)"
    using assms by (auto simp: fun_eq_iff simp: rel_def uminus_in_lattice_iff
eisenstein_fun.lattice_cong)
  also have "is_pole ... 0 ⟷ is_pole (eisenstein_fun n) z"
    by (simp add: is_pole_shift_0' o_def add.commute)
  finally show ?thesis .
qed

sublocale eisenstein_fun: nicely_elliptic_function ω1 ω2 "eisenstein_fun
n"
proof
  show "eisenstein_fun n nicely_meromorphic_on UNIV"
  proof (cases "n ≥ 3")
    case True
    show ?thesis
    proof (rule nicely_meromorphic_onI_open)
      show "eisenstein_fun n analytic_on UNIV - Λ"
        using eisenstein_fun_analytic[of n] by (simp add: Compl_eq_Diff_UNIV)
      show "is_pole (eisenstein_fun n) z ∧ eisenstein_fun n z = 0" if
"z ∈ Λ" for z
        using that by (simp add: True eisenstein_fun_def is_pole_eisenstein_fun)
      show "isolated_singularity_at (eisenstein_fun n) z" for z
      proof (rule analytic_nhd_imp_isolated_singularity[of _ "-(Λ-{z})"])
        show "open (- (Λ - {z}))"
          by (intro open_Compl closed_subset_lattice) auto
      qed (auto intro!: analytic_intros)
    qed simp
  qed (auto simp: eisenstein_fun_def [abs_def] intro!: analytic_on_imp_nicely_meromorphic_o
qed

lemmas [elliptic_function_intros] =
  eisenstein_fun.elliptic_function_axioms eisenstein_fun.nicely_elliptic_function_axioms

end
```

6.2 Definition and basic properties

The Weierstraß $\wp$ function is in a sense the most basic elliptic function, and we will see later on that all elliptic function can be written as a combination of $\wp$ and $\wp'$.

Its derivative, as we noted before, is equal to our Eisenstein function for $n = 3$ (up to a constant factor -2). The function $\wp$ itself is somewhat more awkward to define.

```
context complex_lattice begin

lemma minus_lattice_eq: "uminus ` Λ = Λ"
proof -
  have "uminus ` Λ ⊆ Λ"
    by (auto simp: uminus_in_lattice_iff)
  then show ?thesis
    using equation_minus_iff by blast
qed

lemma minus_latticemz_eq: "uminus ` Λ* = Λ*"
  by (simp add: lattice0_def inj_on_def image_set_diff minus_lattice_eq)

lemma bij_minus_latticemz: "bij_betw uminus Λ* Λ*"
  by (simp add: bij_betw_def inj_on_def minus_latticemz_eq)


definition weierstrass_fun_deriv ("℘''") where
  "weierstrass_fun_deriv z = -2 * eisenstein_fun 3 z"

sublocale weierstrass_fun_deriv: elliptic_function ω1 ω2 weierstrass_fun_deriv
  unfolding weierstrass_fun_deriv_def by (intro elliptic_function_intros)

sublocale weierstrass_fun_deriv: nicely_elliptic_function ω1 ω2 weierstrass_fun_deriv
proof
  show "℘' nicely_meromorphic_on UNIV"
    using eisenstein_fun.nicely_meromorphic unfolding weierstrass_fun_deriv_def
    by (intro nicely_meromorphic_on_uminus nicely_meromorphic_on_cmult_left)
qed

lemmas [elliptic_function_intros] =
  weierstrass_fun_deriv.elliptic_function_axioms weierstrass_fun_deriv.nicely_elliptic_funct

lemma weierstrass_fun_deriv_minus [simp]: "℘' (-z) = -℘' z"
  by (simp add: weierstrass_fun_deriv_def eisenstein_fun_odd_minus)

lemma weierstrass_fun_deriv_has_field_derivative:
  assumes "z ∉ Λ"
  shows   "(℘' has_field_derivative 6 * eisenstein_fun 4 z) (at z)"
  unfolding weierstrass_fun_deriv_def
  using assms by (auto intro!: derivative_eq_intros)

lemma weierstrass_fun_deriv_holomorphic: "℘' holomorphic_on -Λ"
  unfolding weierstrass_fun_deriv_def by (auto intro!: holomorphic_intros)

lemma weierstrass_fun_deriv_holomorphic' [holomorphic_intros]:
  assumes "f holomorphic_on A" "⋀z. z ∈ A ⟹ f z ∉ Λ"
  shows   "(λz. ℘' (f z)) holomorphic_on A"
  using assms unfolding weierstrass_fun_deriv_def by (auto intro!: holomorphic_intros)
```

```
lemma weierstrass_fun_deriv_analytic: "℘' analytic_on -Λ"
  unfolding weierstrass_fun_deriv_def by (auto intro!: analytic_intros)

lemma weierstrass_fun_deriv_analytic' [analytic_intros]:
  assumes "f analytic_on A" "⋀z. z ∈ A ⟹ f z ∉ Λ"
  shows   "(λz. ℘' (f z)) analytic_on A"
  using assms unfolding weierstrass_fun_deriv_def by (auto intro!: analytic_intros)

lemma weierstrass_fun_deriv_continuous_on: "continuous_on (-Λ) ℘'"
  unfolding weierstrass_fun_deriv_def by (auto intro!: continuous_intros)

lemma weierstrass_fun_deriv_continuous_on' [continuous_intros]:
  assumes "continuous_on A f" "⋀z. z ∈ A ⟹ f z ∉ Λ"
  shows   "continuous_on A (λz. ℘' (f z))"
  using assms unfolding weierstrass_fun_deriv_def by (auto intro!: continuous_intros)

lemma tendsto_weierstrass_fun_deriv [tendsto_intros]:
  assumes "(f ⟶ c) F" "c ∉ Λ"
  shows   "((λz. ℘' (f z)) ⟶ ℘' c) F"
proof (rule continuous_on_tendsto_compose[OF _ assms(1)])
  show "continuous_on (-Λ) ℘'"
    by (intro holomorphic_on_imp_continuous_on holomorphic_intros) auto
  show "eventually (λz. f z ∈ -Λ) F"
    by (rule eventually_compose_filterlim[OF _ assms(1)], rule eventually_nhds_in_open)
       (use assms(2) in ‹auto intro: closed_lattice›)
qed (use assms(2) in auto)
```

The following is the Weierstraß function minus its pole at the origin. By convention, it returns 0 at all its remaining poles.

```
definition weierstrass_fun_aux :: "complex ⇒ complex" where
  "weierstrass_fun_aux z = (if z ∈ Λ* then 0 else (∑∞ ω ∈ Λ*. 1 / (z
- ω)^2 - 1 / ω^2))"
```

This is now the Weierstraß function. Again, it returns 0 at all its poles.

```
definition weierstrass_fun :: "complex ⇒ complex" ("℘")
  where "℘ z = (if z ∈ Λ then 0 else 1 / z^2 + weierstrass_fun_aux z)"

lemma weierstrass_fun_aux_0 [simp]: "weierstrass_fun_aux 0 = 0"
  by (simp add: weierstrass_fun_aux_def)

lemma weierstrass_fun_at_pole: "ω ∈ Λ ⟹ ℘ ω = 0"
  by (simp add: weierstrass_fun_def)


lemma
  fixes R :: real
  assumes "R > 0"
  shows weierstrass_fun_aux_converges_absolutely_uniformly_in_disk:
```

```
          "uniform_limit (cball 0 R)
                         (λX z. ∑ω∈X. norm (1 / (z - ω)^2 - 1 / ω^2))
                         (λz. ∑∞ω∈Λ-cball 0 R. norm (1 / (z - ω)^2 -
1 / ω^2))
                         (finite_subsets_at_top (Λ - cball 0 R))" (is
?th1)
  and   weierstrass_fun_aux_converges_uniformly_in_disk:
          "uniform_limit (cball 0 R)
                         (λX z. ∑ω∈X. 1 / (z - ω)^2 - 1 / ω^2)
                         (λz. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω)^2 - 1 / ω^2)
                         (finite_subsets_at_top (Λ - cball 0 R))" (is
?th2)
proof -
  obtain M where M:
    "M > 0" "⋀ω z. ⟦ω ∈ Λ; norm ω > R; norm z ≤ R⟧ ⟹ norm (z - ω)
powr -2 ≤ M * norm ω powr -2"
    using weierstrass_summand_bound[of 2 R] assms by auto

  have 1: "(λω. 3 * M * R * norm ω powr -3) summable_on (Λ - cball 0
R)"
  proof -
    have "(λω. 1 / norm ω powr 3) summable_on Λ*"
      using assms by (subst converges_absolutely_iff) auto
    hence "(λω. 3 * M * R * norm ω powr -3) summable_on Λ*"
      by (intro summable_on_cmult_right) (auto simp: powr_minus field_simps)
    thus "(λω. 3 * M * R * norm ω powr -3) summable_on (Λ - cball 0 R)"
      by (rule summable_on_subset) (use assms in ‹auto simp: lattice0_def›)
  qed

  have 2: "norm (1 / (z - ω)^2 - 1 / ω^2) ≤ 3 * M * R * norm ω powr -3"
    if "ω ∈ Λ - cball 0 R" "z ∈ cball 0 R" for ω z
  proof -
    from that have nz: "ω ≠ 0" "ω ≠ z"
      using ‹R > 0› by auto
    hence "1 / (z - ω)^2 - 1 / ω^2 = z * (2 * ω - z) / (ω^2 * (z - ω)^2)"
      using that by (auto simp: field_simps) (auto simp: power2_eq_square
algebra_simps)
    also have "norm ... = norm z * norm (2 * ω - z) / norm ω ^ 2 * norm
(z - ω) powr - 2"
      by (simp add: norm_divide norm_mult norm_power powr_minus divide_simps)
    also have "... ≤ R * (2 * norm ω + norm z) / norm ω ^ 2 * (M * norm
ω powr -2)"
      using assms that
      by (intro mult_mono frac_le mult_nonneg_nonneg M order.trans[OF
norm_triangle_ineq4]) auto
    also have "... = M * R * (2 + norm z / norm ω) / norm ω ^ 3"
      using nz by (simp add: field_simps powr_minus power2_eq_square power3_eq_cube)
    also have "... ≤ M * R * 3 / norm ω ^ 3"
      using nz assms M(1) that by (intro mult_left_mono divide_right_mono)
```

```
auto
    finally show ?thesis
      by (simp add: field_simps powr_minus)
  qed

  show ?th1 ?th2 unfolding weierstrass_fun_aux_def
     by (rule Weierstrass_m_test_general[OF _ 1]; use 2 in simp)+
qed

lemma weierstrass_fun_has_field_derivative_aux:
  fixes R :: real
  defines "F ≡ (λz. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω)^2 - 1 / ω^2)"
  defines "F' ≡ (λz. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω) ^ 3)"
  assumes "R > 0" "w ∈ ball 0 R"
  shows   "(F has_field_derivative -2 * F' w) (at w)"
proof -
  have 1: "∀F n in finite_subsets_at_top (Λ - cball 0 R).
           continuous_on (cball 0 R) (λz. ∑ω∈n. 1 / (z - ω)^2 - 1 /
ω^2) ∧
           (∀z∈ball 0 R. ((λz. ∑ω∈n. 1 / (z - ω)^2 - 1 / ω^2) has_field_derivative
(∑ω∈n. -2 / (z - ω)^3)) (at z))"

    apply (intro eventually_finite_subsets_at_top_weakI conjI continuous_intros
derivative_intros ballI)
     apply force
    apply (rule derivative_eq_intros refl | force)+
    apply (simp add: divide_simps, simp add: algebra_simps power4_eq_xxxx
power3_eq_cube)
    done

  have "uniform_limit (cball 0 R)
                      (λX z. ∑ω∈X. 1 / (z - ω)^2 - 1 / ω^2)
                      (λz. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω)^2 - 1 / ω^2)
                      (finite_subsets_at_top (Λ - cball 0 R))"
    using assms by (intro weierstrass_fun_aux_converges_uniformly_in_disk)
auto
  also have "?this ⟷ uniform_limit (cball 0 R) (λX z. ∑ω∈X. 1 / (z
- ω)^2 - 1 / ω^2) F
                         (finite_subsets_at_top (Λ - cball 0 R))"
    using assms unfolding F_def
    by (intro uniform_limit_cong eventually_finite_subsets_at_top_weakI)
       (auto simp: powr_minus powr_nat field_simps intro!: sum.cong infsum_cong)
  finally have 2: ... .

  have 3: "finite_subsets_at_top (Λ - cball 0 R) ≠ bot"
    by simp

  obtain g where g: "continuous_on (cball 0 R) F"
                    "⋀w. w ∈ ball 0 R ⟹ (F has_field_derivative g
```

```
w) (at w) ∧
                        ((λω. -2 / (w - ω) ^ 3) has_sum g w) (Λ - cball
0 R)"
    unfolding has_sum_def using has_complex_derivative_uniform_limit[OF
1 2 3 <R > 0>] by auto

  have "((λω. (-2) * (1 / (w - ω) ^ 3)) has_sum (-2) * F' w) (Λ - cball
0 R)"
    unfolding F'_def using assms
    by (intro has_sum_cmult_right has_sum_infsum weierstrass_aux_converges_in_disk')
auto
  moreover have "((λω.  -2 * (1 / (w - ω) ^ 3)) has_sum g w) (Λ - cball
0 R)"
    using g(2)[of w] assms by simp
  ultimately have "g w = -2 * F' w"
    by (metis infsumI)
  thus "(F has_field_derivative -2 * F' w) (at w)"
    using g(2)[of w] assms by simp
qed

lemma norm_summable_weierstrass_fun_aux: "(λω. norm (1 / (z - ω)^2 -
1 / ω^2)) summable_on Λ"
proof -
  define R where "R = norm z + 1"
  have "(λω. norm (1 / (z - ω)^2 - 1 / ω^2)) summable_on (Λ - cball 0 R)"
    unfolding summable_iff_has_sum_infsum has_sum_def
    by (rule tendsto_uniform_limitI[OF weierstrass_fun_aux_converges_absolutely_uniformly_i
       (auto simp: R_def add_nonneg_pos)
  hence "(λω. norm (1 / (z - ω)^2 - 1 / ω^2)) summable_on ((Λ - cball 0
R) ∪ (Λ ∩ cball 0 R))"
    by (intro summable_on_union[OF _ summable_on_finite]) (auto simp:
bounded_lattice_finite)
  also have "... = Λ"
    by blast
  finally show ?thesis .
qed

lemma summable_weierstrass_fun_aux: "(λω. 1 / (z - ω)^2 - 1 / ω^2) summable_on
Λ"
  using norm_summable_weierstrass_fun_aux by (rule abs_summable_summable)

lemma weierstrass_summable: "(λω. 1 / (z - ω)^2 - 1 / ω^2) summable_on
Λ*"
  by (rule summable_on_subset[OF summable_weierstrass_fun_aux]) (auto
simp: lattice0_def)

lemma weierstrass_fun_aux_has_sum:
  "z ∉ Λ* ⟹ ((λω. 1 / (z - ω)^2 - 1 / ω^2) has_sum weierstrass_fun_aux
z) Λ*"
```

```
  unfolding weierstrass_fun_aux_def by (simp add: weierstrass_summable)

lemma weierstrass_fun_aux_has_field_derivative:
  defines "F ≡ weierstrass_fun_aux"
  defines "F' ≡ (λz. ∑∞ω∈Λ*. 1 / (z - ω) ^ 3)"
  assumes z: "z ∉ Λ*"
  shows   "(F has_field_derivative -2 * eisenstein_fun_aux 3 z) (at z)"
proof -
  define R where "R = norm z + 1"
  have R: "R > 0" "norm z < R"
    by (auto simp: R_def add_nonneg_pos)
  have "finite (Λ ∩ cball 0 R)"
    by (simp add: bounded_lattice_finite)
  moreover have "Λ* ∩ cball 0 R ⊆ Λ ∩ cball 0 R"
    unfolding lattice0_def by blast
  ultimately have fin: "finite (Λ* ∩ cball 0 R)"
    using finite_subset by blast

  define F1 where "F1 = (λz. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω)^2 - 1 / ω^2)"
  define F'1 where "F'1 = (λz. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω) ^ 3)"
  define F2 where "F2 = (λz. ∑ω∈Λ*∩cball 0 R. 1 / (z - ω)^2 - 1 / ω^2)"
  define F'2 where "F'2 = (λz. ∑ω∈Λ*∩cball 0 R. 1 / (z - ω) ^ 3)"

  have "(F1 has_field_derivative -2 * F'1 z) (at z)"
    unfolding F1_def F'1_def
    by (rule weierstrass_fun_has_field_derivative_aux) (auto simp: R_def
add_nonneg_pos)
  moreover have "(F2 has_field_derivative -2 * F'2 z) (at z)"
    unfolding F2_def F'2_def sum_distrib_left lattice0_def
    by (rule derivative_eq_intros refl sum.cong | use R z in ‹force simp:
lattice0_def›)+
       (simp add: divide_simps power3_eq_cube power4_eq_xxxx)
  ultimately have "((λz. F1 z + F2 z) has_field_derivative (-2 * F'1 z)
+ (-2 * F'2 z)) (at z)"
    by (intro derivative_intros)
  also have "?this ⟷ (F has_field_derivative (-2 * F'1 z) + (-2 * F'2
z)) (at z)"
  proof (intro has_field_derivative_cong_ev refl)
    have "eventually (λz'. z' ∈ -Λ*) (nhds z)"
      using z by (intro eventually_nhds_in_open) (auto simp: closed_lattice0)
    thus "∀F x in nhds z. x ∈ UNIV ⟶ F1 x + F2 x = F x"
    proof eventually_elim
      case (elim z)
      have "((λω. 1 / (z - ω)^2 - 1 / ω^2) has_sum (F1 z + F2 z)) ((Λ
- cball 0 R) ∪ (Λ* ∩ cball 0 R))"
        unfolding F1_def F2_def using R fin
        by (intro has_sum_Un_disjoint[OF has_sum_infsum has_sum_finite]
                  summable_on_subset_banach[OF summable_weierstrass_fun_aux])
auto
```

```
    also have "(Λ - cball 0 R) ∪ (Λ* ∩ cball 0 R) = Λ*"
      using R unfolding lattice0_def by auto
    finally show ?case using elim
      unfolding F1_def F2_def F_def weierstrass_fun_aux_def by (simp
add: infsumI)
  qed
 qed auto
 also have "(-2 * F'1 z) + (-2 * F'2 z) = -2 * (F'1 z + F'2 z)"
  by (simp add: algebra_simps)
 also have "F'1 z + F'2 z = F' z"
 proof -
  have "((λω. 1 / (z - ω)^3) has_sum (F'1 z + F'2 z)) ((Λ - cball 0
R) ∪ (Λ* ∩ cball 0 R))"
    unfolding F'1_def F'2_def using R fin
    by (intro has_sum_Un_disjoint [OF has_sum_infsum has_sum_finite]
weierstrass_aux_converges') auto
  also have "(Λ - cball 0 R) ∪ (Λ* ∩ cball 0 R) = Λ*"
    using R unfolding lattice0_def by auto
  finally show "F'1 z + F'2 z = F' z"
    unfolding F'1_def F'2_def F'_def by (simp add: infsumI)
 qed
 finally show ?thesis
  using assms by (simp add: eisenstein_fun_aux_def)
qed

lemmas weierstrass_fun_aux_has_field_derivative' [derivative_intros]
=
 weierstrass_fun_aux_has_field_derivative [THEN DERIV_chain2]

lemma weierstrass_fun_aux_holomorphic: "weierstrass_fun_aux holomorphic_on
-Λ*"
 by (subst holomorphic_on_open)
    (auto intro!: weierstrass_fun_aux_has_field_derivative simp: closed_lattice0)

lemma weierstrass_fun_aux_holomorphic' [holomorphic_intros]:
 assumes "f holomorphic_on A" "⋀z. z ∈ A ⟹ f z ∉ Λ*"
 shows   "(λz. weierstrass_fun_aux (f z)) holomorphic_on A"
proof -
 have "weierstrass_fun_aux ∘ f holomorphic_on A"
  by (rule holomorphic_on_compose_gen assms weierstrass_fun_aux_holomorphic)+
(use assms in auto)
 thus ?thesis by (simp add: o_def)
qed

lemma weierstrass_fun_aux_continuous_on: "continuous_on (-Λ*) weierstrass_fun_aux"
 using holomorphic_on_imp_continuous_on weierstrass_fun_aux_holomorphic
by blast

lemma weierstrass_fun_aux_continuous_on' [continuous_intros]:
```

```
  assumes "continuous_on A f" "⋀z. z ∈ A ⟹ f z ∉ Λ*"
  shows   "continuous_on A (λz. weierstrass_fun_aux (f z))"
  by (rule continuous_on_compose2[OF weierstrass_fun_aux_continuous_on
assms(1)]) (use assms in auto)

lemma weierstrass_fun_aux_analytic: "weierstrass_fun_aux analytic_on
-Λ*"
  by (simp add: analytic_on_open closed_lattice0 open_Compl weierstrass_fun_aux_holomorphic


lemma weierstrass_fun_aux_analytic' [analytic_intros]:
  assumes "f analytic_on A" "⋀z. z ∈ A ⟹ f z ∉ Λ*"
  shows   "(λz. weierstrass_fun_aux (f z)) analytic_on A"
proof -
  have "weierstrass_fun_aux ∘ f analytic_on A"
    by (rule analytic_on_compose_gen assms weierstrass_fun_aux_analytic)+
(use assms in auto)
  thus ?thesis by (simp add: o_def)
qed

lemma deriv_weierstrass_fun_aux:
  "z ∉ Λ* ⟹ deriv weierstrass_fun_aux z = -2 * eisenstein_fun_aux 3
z"
  by (rule DERIV_imp_deriv derivative_eq_intros refl | assumption)+ simp


lemma weierstrass_fun_has_field_derivative:
  fixes R :: real
  assumes z: "z ∉ Λ"
  shows   "(℘ has_field_derivative ℘' z) (at z)"
proof -
  note [derivative_intros] = weierstrass_fun_aux_has_field_derivative
  from z have [simp]: "z ≠ 0" "z ∉ Λ*"
    by (auto simp: lattice0_def)
  define D where "D = -2 / z ^ 3 - 2 * eisenstein_fun_aux 3 z"

  have "((λz. 1 / z² + weierstrass_fun_aux z) has_field_derivative D)
(at z)" unfolding D_def
    by (rule derivative_eq_intros refl | simp)+ (simp add: divide_simps
power3_eq_cube power4_eq_xxxx)
  also have "?this ⟷ (weierstrass_fun has_field_derivative D) (at z)"
  proof (intro has_field_derivative_cong_ev refl)
    have "eventually (λz. z ∈ -Λ) (nhds z)"
      using closed_lattice z by (intro eventually_nhds_in_open) auto
    thus "eventually (λz. z ∈ UNIV ⟶ 1 / z ^ 2 + weierstrass_fun_aux
z = ℘ z) (nhds z)"
      by eventually_elim (simp add: weierstrass_fun_def)
  qed auto
  also have "D = -2 * eisenstein_fun 3 z"
```

```
    using z by (simp add: eisenstein_fun_conv_eisenstein_fun_aux D_def)
  finally show ?thesis by (simp add: weierstrass_fun_deriv_def)
qed

lemmas weierstrass_fun_has_field_derivative' [derivative_intros] =
  weierstrass_fun_has_field_derivative [THEN DERIV_chain2]

lemma weierstrass_fun_holomorphic: "℘ holomorphic_on -Λ"
  by (subst holomorphic_on_open)
     (auto intro!: weierstrass_fun_has_field_derivative simp: closed_lattice)

lemma weierstrass_fun_holomorphic' [holomorphic_intros]:
  assumes "f holomorphic_on A" "⋀z. z ∈ A ⟹ f z ∉ Λ"
  shows   "(λz. weierstrass_fun (f z)) holomorphic_on A"
proof -
  have "weierstrass_fun ∘ f holomorphic_on A"
    by (rule holomorphic_on_compose_gen assms weierstrass_fun_holomorphic)+
(use assms in auto)
  thus ?thesis by (simp add: o_def)
qed

lemma weierstrass_fun_analytic: "℘ analytic_on -Λ"
  by (simp add: analytic_on_open closed_lattice open_Compl weierstrass_fun_holomorphic)


lemma weierstrass_fun_analytic' [analytic_intros]:
  assumes "f analytic_on A" "⋀z. z ∈ A ⟹ f z ∉ Λ"
  shows   "(λz. ℘ (f z)) analytic_on A"
proof -
  have "℘ ∘ f analytic_on A"
    by (rule analytic_on_compose_gen assms weierstrass_fun_analytic)+
(use assms in auto)
  thus ?thesis by (simp add: o_def)
qed

lemma weierstrass_fun_continuous_on: "continuous_on (-Λ) weierstrass_fun"
  using holomorphic_on_imp_continuous_on weierstrass_fun_holomorphic by
blast

lemma weierstrass_fun_continuous_on' [continuous_intros]:
  assumes "continuous_on A f" "⋀z. z ∈ A ⟹ f z ∉ Λ"
  shows   "continuous_on A (λz. ℘ (f z))"
  by (rule continuous_on_compose2[OF weierstrass_fun_continuous_on assms(1)])
(use assms in auto)

lemma tendsto_weierstrass_fun [tendsto_intros]:
  assumes "(f ⟶ c) F" "c ∉ Λ"
  shows   "((λz. ℘ (f z)) ⟶ ℘ c) F"
proof (rule continuous_on_tendsto_compose[OF _ assms(1)])
```

```
  show "continuous_on (-Λ) ℘"
    by (intro holomorphic_on_imp_continuous_on holomorphic_intros) auto
  show "eventually (λz. f z ∈ -Λ) F"
    by (rule eventually_compose_filterlim[OF _ assms(1)], rule eventually_nhds_in_open)
       (use assms(2) in ‹auto intro: closed_lattice›)
qed (use assms(2) in auto)

lemma deriv_weierstrass_fun:
  assumes "z ∉ Λ"
  shows   "deriv ℘ z = ℘' z"
  by (rule DERIV_imp_deriv weierstrass_fun_has_field_derivative refl assms)+
```

The following identity is to be read with care: for $\omega = 0$ we get a division by zero, so the term $1 / \omega^2$ simply gets dropped.

```
lemma weierstrass_fun_eq:
  assumes "z ∉ Λ"
  shows   "℘ z = (∑∞ ω ∈ Λ. (1 / (z - ω) ^ 2) - 1 / ω ^ 2)"
proof -
  have *: "((λω. 1 / (z - ω)² - 1 / ω²) has_sum ℘ z - 1 / z^2) Λ*"
    using has_sum_infsum [OF weierstrass_summable, of z] assms
    by (simp add: weierstrass_fun_def weierstrass_fun_aux_def lattice0_def)
  have "((λω. 1 / (z - ω)² - 1 / ω²) has_sum ((1 / (z - 0)^2 - 1 / 0²)
+ (℘ z - 1 / z^2))) Λ"
    unfolding lattice_lattice0 by (rule has_sum_insert) (use * in auto)
  then show ?thesis
    by (simp add: infsumI)
qed
```

6.3 Ellipticity and poles

It can easily be seen from its definition that $\wp$ is an even elliptic function with a double pole at each lattice point and no other poles. Thus it has order 2.

Its derivative is consequently an odd elliptic function with a triple pole at each lattice point, no other poles, and order 3.

The results in this section correspond to Apostol's Theorems 1.9 and 1.10.

```
lemma weierstrass_fun_minus: "℘ (-z) = ℘ z"
proof (cases "z ∈ Λ")
  case False
  have "(∑∞ ω ∈ Λ*. 1 / (- z - ω)² - 1 / ω²) = (∑∞ ω ∈ Λ*. (1 /
(z - ω)² - 1 / ω²))"
    by (rule infsum_reindex_bij_witness[of _ uminus uminus])
       (auto intro!: lattice_intros simp: uminus_in_lattice0_iff power2_commute
add_ac)
  thus ?thesis using False
    by (auto simp: weierstrass_fun_def weierstrass_fun_aux_def uminus_in_lattice0_iff
lattice_lattice0)
```

```
qed (auto simp: weierstrass_fun_at_pole uminus_in_lattice_iff)

sublocale weierstrass_fun: complex_lattice_periodic ω1 ω2 ℘
proof
  have *: "℘ (z + ω) = ℘ z" if ω: "ω ∈ {ω1, ω2}" for ω z
  proof (cases "z ∈ Λ")
    case z: True
    thus ?thesis
      by (subst (1 2) weierstrass_fun_at_pole) (use ω in ‹auto intro!:
lattice_intros›)
  next
    case z: False
    from ω have ω' [simp, intro]: "ω ∈ Λ"
      by auto
    define f where "f = (λz. ℘ (z + ω) - ℘ z)"
    with ‹z ∉ Λ› have "(f has_field_derivative 0) (at z)" if "z ∉ Λ"
for z
    proof -
      from that and ω have "(f has_field_derivative (℘' (z + ω) - ℘'
z)) (at z)"
        unfolding f_def by (auto intro!: derivative_eq_intros)
      also have "℘' (z + ω) = ℘' z"
        by (rule weierstrass_fun_deriv.lattice_cong) (auto simp: rel_def)
      finally show ?thesis
        by simp
    qed
    hence deriv: "∀x∈UNIV - Λ - {}. (f has_field_derivative 0) (at x)"
      by blast
    have cont: "continuous_on (UNIV - Λ) f"
      by (auto simp: f_def intro!: continuous_intros)

    have *: "connected (UNIV - Λ)" "open (UNIV - Λ)" "finite ({} :: complex
set)"
      by (auto simp: closed_lattice countable_lattice intro!: connected_open_diff_countable

    obtain c where c: "⋀z. z ∈ UNIV - Λ ⟹ f z = c"
      using DERIV_zero_connected_constant[OF * cont deriv] by blast

    have "ω / 2 ∉ Λ"
      using of_ω12_coords_in_lattice_iff ω by (auto simp: in_lattice_conv_ω12_coords)
    hence "f (-ω / 2) = c"
      by (intro c) (auto simp: uminus_in_lattice_iff)
    also have "f (-ω / 2) = 0"
      by (simp add: f_def weierstrass_fun_minus)
    finally have "f z = 0"
      using c that z by auto
    thus ?thesis
      by (simp add: f_def)
  qed
```

```
  show "℘ (z + ω1) = ℘ z" "℘ (z + ω2) = ℘ z" for z
    by (rule *; simp)+
qed


lemma zorder_weierstrass_fun_pole:
  assumes "ω ∈ Λ"
  shows   "zorder ℘ ω = -2"
proof -
  define R where "R = Inf_para"
  have R: "R > 0"
    using Inf_para_pos by (auto simp: R_def)
  have R': "ball 0 R ∩ Λ* = {}"
    using Inf_para_le_norm by (force simp: R_def)

  have "zorder weierstrass_fun ω = zorder (λz. weierstrass_fun (z + ω))
0"
    by (rule zorder_shift)
  also have "(λz. weierstrass_fun (z + ω)) = weierstrass_fun"
    by (intro ext weierstrass_fun.lattice_cong) (auto simp: rel_def assms)
  also have "zorder weierstrass_fun 0 = -2"
  proof (rule zorder_eqI)
    show "open (ball 0 R :: complex set)" "(0 :: complex) ∈ ball 0 R"
      using R by auto
    show "(λz. 1 + weierstrass_fun_aux z * z ^ 2) holomorphic_on ball
0 R" using R'
      by (intro holomorphic_intros holomorphic_on_subset[OF weierstrass_fun_aux_holomorphic
auto
    show "⋀w. ⟦w ∈ ball 0 R; w ≠ 0⟧
          ⟹ ℘ w =
              (1 + weierstrass_fun_aux w * w^2) *
              (w - 0) powi - 2"
      using R'
      by (auto simp: weierstrass_fun_def field_simps power_numeral_reduce
powi_minus_numeral_reduce
                               lattice0_def)
  qed auto
  finally show ?thesis .
qed

lemma is_pole_weierstrass_fun:
  assumes ω:  "ω ∈ Λ"
  shows   "is_pole ℘ ω"
proof -
  have "is_pole ℘ 0"
  proof -
    have "eventually (λz. z ∈ -Λ*) (nhds 0)"
      using closed_lattice0  by (intro eventually_nhds_in_open) auto
```

```
    hence ev: "eventually (λz. z ∉ Λ) (at 0)"
      unfolding eventually_at_filter by eventually_elim (auto simp: lattice0_def)
    have "Λ - Λ* = {0}" "Λ* ⊆ Λ"
      by (auto simp: insert_Diff_if lattice_lattice0)
    hence "weierstrass_fun_aux holomorphic_on -Λ*"
      by (auto intro!: holomorphic_intros)
    hence "continuous_on (-Λ*) weierstrass_fun_aux"
      using holomorphic_on_imp_continuous_on by blast
    moreover have "0 ∈ -Λ*"
      by (auto simp: lattice0_def)
    ultimately have "(weierstrass_fun_aux ⟶ weierstrass_fun_aux 0)
(at 0)"
      using closed_lattice0 by (metis at_within_open closed_open continuous_on_def)
    moreover have "filterlim (λz::complex. 1 / z ^ 2) at_infinity (at
0)"
      using is_pole_inverse_power[of 2 0] by (simp add: is_pole_def)
    ultimately have "filterlim (λz. weierstrass_fun_aux z + 1 / z ^ 2)
at_infinity (at 0)"
      by (rule tendsto_add_filterlim_at_infinity)
    also have "?this ⟷ filterlim weierstrass_fun at_infinity (at 0)"
      by (intro filterlim_cong refl eventually_mono[OF ev]) (auto simp:
weierstrass_fun_def)
    finally show ?thesis
      by (simp add: is_pole_def)
  qed
  also have "℘ = ℘ ∘ (λz. z + ω)"
    by (auto simp: fun_eq_iff rel_def assms uminus_in_lattice_iff
             intro!: weierstrass_fun.lattice_cong)
  also have "is_pole ... 0 ⟷ is_pole ℘ ω"
    by (simp add: o_def is_pole_shift_0' add_ac)
  finally show "is_pole ℘ ω" .
qed

sublocale weierstrass_fun: nicely_elliptic_function ω1 ω2 ℘
proof
  show "℘ nicely_meromorphic_on UNIV"
  proof (rule nicely_meromorphic_onI_open)
    show "℘ analytic_on UNIV - Λ"
      by (auto intro!: analytic_intros)
  next
    fix z assume "z ∈ Λ"
    thus "is_pole ℘ z ∧ ℘ z = 0"
      by (auto simp: weierstrass_fun_at_pole is_pole_weierstrass_fun)
  next
    show "isolated_singularity_at ℘ z" for z
    proof (rule analytic_nhd_imp_isolated_singularity[of _ "-(Λ-{z})"])
      show "open (- (Λ - {z}))"
        by (intro open_Compl closed_subset_lattice) auto
    qed (auto intro!: analytic_intros)
```

```
  qed auto
qed

sublocale weierstrass_fun: even_elliptic_function ω1 ω2 ℘
  by standard (auto simp: weierstrass_fun_minus)

lemmas [elliptic_function_intros] =
  weierstrass_fun.elliptic_function_axioms
  weierstrass_fun.nicely_elliptic_function_axioms

lemma is_pole_weierstrass_fun_iff: "is_pole ℘ z ⟷ z ∈ Λ"
  by (meson ComplI analytic_on_analytic_at is_pole_weierstrass_fun
      nicely_elliptic_function.analytic_at_iff_not_pole
      weierstrass_fun.nicely_elliptic_function_axioms weierstrass_fun_analytic)

lemma is_pole_weierstrass_fun_deriv_iff: "is_pole ℘' z ⟷ z ∈ Λ"
proof -
  have "eventually (λw. w ∉ Λ) (at z)"
    using islimpt_iff_eventually not_islimpt_lattice by auto
  hence "eventually (λw. ℘' w = deriv ℘ w) (at z)"
    by eventually_elim (simp add: deriv_weierstrass_fun)
  hence "is_pole ℘' z ⟷ is_pole (deriv ℘) z"
    by (rule is_pole_cong) auto
  also have "... ⟷ is_pole ℘ z"
    by (rule is_pole_deriv_iff) (auto intro!: meromorphic_intros)
  also have "... ⟷ z ∈ Λ"
    by (rule is_pole_weierstrass_fun_iff)
  finally show ?thesis .
qed

lemma zorder_weierstrass_fun_deriv_pole:
  assumes "z ∈ Λ"
  shows   "zorder ℘' z = -3"
proof -
  have "eventually (λw. w ∉ Λ) (at z)"
    using islimpt_iff_eventually not_islimpt_lattice by auto
  hence "eventually (λw. ℘' w = deriv ℘ w) (at z)"
    by eventually_elim (simp add: deriv_weierstrass_fun)
  hence "zorder ℘' z = zorder (deriv ℘) z"
    by (rule zorder_cong) auto
  also have "... = zorder ℘ z - 1"
    by (subst zorder_deriv) (auto simp: is_pole_weierstrass_fun_iff assms)
  also have "... = -3"
    using assms by (simp add: zorder_weierstrass_fun_pole)
  finally show ?thesis .
qed

lemma order_weierstrass_fun [simp]: "elliptic_order ℘ = 2"
proof -
```

```
  have "elliptic_order ℘ = (∑z ∈ period_parallelogram 0 ∩ Λ. nat (-zorder
℘ z))"
    unfolding elliptic_order_def by (rule sum.cong) (auto simp: is_pole_weierstrass_fun_iff
  also have "period_parallelogram 0 ∩ Λ = {0}"
    by (auto elim!: latticeE simp: period_parallelogram_altdef zero_prod_def)
  finally show ?thesis
    by (simp add: zorder_weierstrass_fun_pole)
qed

lemma order_weierstrass_fun_deriv [simp]: "elliptic_order ℘' = 3"
proof -
  have "elliptic_order ℘' = (∑z ∈ period_parallelogram 0 ∩ Λ. nat (-zorder
℘' z))"
    unfolding elliptic_order_def by (rule sum.cong) (auto simp: is_pole_weierstrass_fun_der
  also have "period_parallelogram 0 ∩ Λ = {0}"
    by (auto elim!: latticeE simp: period_parallelogram_altdef zero_prod_def)
  also have "(∑z∈{0}. nat (- zorder ℘' z)) = nat (-zorder ℘' 0)"
    by simp
  finally show ?thesis
    by (simp add: zorder_weierstrass_fun_deriv_pole)
qed

sublocale weierstrass_fun: nonconst_nicely_elliptic_function ω1 ω2 ℘
  by standard auto

sublocale weierstrass_fun_deriv: nonconst_nicely_elliptic_function ω1
ω2 "℘'"
  by standard auto
```

6.4 The numbers e_1, e_2, e_3

The values of $\wp$ at the half-periods $\frac{1}{2}\omega_1$, $\frac{1}{2}\omega_2$, and $\frac{1}{2}(\omega_1+\omega_2)$ are exactly the roots of the polynomial $4X^3 - g_2X - g_3$.

We call these values e_1, e_2, e_3.

```
definition number_e1:: "complex" ("e₁") where
  "e₁ ≡ ℘ (ω1 / 2)"

definition number_e2:: "complex" ("e₂") where
  "e₂ ≡ ℘ (ω2 / 2)"

definition number_e3:: "complex" ("e₃") where
  "e₃ ≡ ℘ ((ω1 + ω2) / 2)"

lemmas number_e123_defs = number_e1_def number_e2_def number_e3_def
```

The half-lattice points are those that are equivalent to one of the three points $\frac{\omega_1}{2}$, $\frac{\omega_2}{2}$, and $\frac{\omega_1+\omega_2}{2}$.

```
lemma to_fund_parallelogram_half_period:
```

```
  assumes "ω ∉ Λ" "2 * ω ∈ Λ"
  shows   "to_fund_parallelogram ω ∈ {ω1 / 2, ω2 / 2, (ω1 + ω2) / 2}"
proof -
  from assms(2) obtain m n where "2 * ω = of_ω12_coords (of_int m, of_int
n)"
    by (elim latticeE)
  hence mn: "ω = of_ω12_coords (of_int m / 2, of_int n / 2)"
    by (auto simp: of_ω12_coords_def field_simps)
  have [simp]: "of_ω12_coords (1 / 2, 1 / 2) = (ω1 + ω2) / 2"
    by (simp add: of_ω12_coords_def field_simps)
  have "odd m ∨ odd n"
    using assms(1) unfolding mn by (auto simp: in_lattice_conv_ω12_coords
elim!: evenE)
  thus ?thesis
    by (cases "even m"; cases "even n")
       (auto elim!: evenE oddE simp: mn to_fund_parallelogram_def add_divide_distrib)
qed

lemma rel_half_period:
  assumes "ω ∉ Λ" "2 * ω ∈ Λ"
  shows   "∃ω'∈{ω1 / 2, ω2 / 2, (ω1 + ω2) / 2}. rel ω ω'"
proof -
  have "rel ω (to_fund_parallelogram ω)"
    by auto
  with to_fund_parallelogram_half_period[OF assms] show ?thesis
    by blast
qed

lemma weierstass_fun_deriv_half_period_eq_0:
  assumes "ω ∈ Λ"
  shows   "℘' (ω / 2) = 0"
  using weierstrass_fun_deriv.lattice_cong[of "-ω/2" "ω/2"] ‹ω ∈ Λ›
  by (simp add: rel_def uminus_in_lattice_iff)

lemma weierstass_fun_deriv_half_root_eq_0 [simp]:
  "℘' (ω1 / 2) = 0" "℘' (ω2 / 2) = 0" "℘' ((ω1 + ω2) / 2) = 0"
  by (rule weierstass_fun_deriv_half_period_eq_0; simp)+

lemma weierstrass_fun_at_half_period:
  assumes "ω ∈ Λ" "ω / 2 ∉ Λ"
  shows   "℘ (ω / 2) ∈ {e₁, e₂, e₃}"
proof -
  have "∃ω'∈{ω1 / 2, ω2 / 2, (ω1 + ω2) / 2}. rel (ω / 2) ω'"
    using rel_half_period[of "ω / 2"] assms by auto
  thus ?thesis
    unfolding number_e123_defs using weierstrass_fun.lattice_cong by
blast
qed
```

```
lemma weierstrass_fun_at_half_period':
  assumes "2 * ω ∈ Λ" "ω ∉ Λ"
  shows   "℘ ω ∈ {e₁, e₂, e₃}"
  using weierstrass_fun_at_half_period[of "2 * ω"] assms by simp
```

$\wp'$ has a simple zero at each half-lattice point, and no other zeros.

```
lemma weierstrass_fun_deriv_eq_0_iff:
  assumes "z ∉ Λ"
  shows   "℘' z = 0 ⟷ 2 * z ∈ Λ"
proof
  assume "℘' z = 0"
  define z' where "z' = to_fund_parallelogram z"
  have z': "℘' z' = 0" "z' ∉ Λ" "z' ∈ period_parallelogram 0"
    using ‹℘' z = 0› assms by (auto simp: z'_def weierstrass_fun_deriv.eval_to_fund_paralle
  have [simp]: "℘' ≠ (λ_. 0)"
    using weierstrass_fun_deriv.elliptic_order_eq_0_iff_no_poles by auto
  have "{ω1 / 2, ω2 / 2, (ω1 + ω2) / 2, z'} ⊆ {z∈period_parallelogram
0. isolated_zero ℘' z}"
    using z' by (auto simp: period_parallelogram_altdef ω12_coords.add
                  weierstrass_fun_deriv.isolated_zero_iff is_pole_weierstrass_fun_deriv_if

  hence "card {ω1 / 2, ω2 / 2, (ω1 + ω2) / 2, z'} ≤ card {z∈period_parallelogram
0. isolated_zero ℘' z}"
    by (intro card_mono weierstrass_fun_deriv.finite_zeros_in_parallelogram)
  also have "... ≤ 3"
    using weierstrass_fun_deriv.card_zeros_le_order[of 0] by simp
  finally have "2 * z' ∈ {ω1, ω2, ω1 + ω2}"
    by (auto simp: card_insert_if split: if_splits)
  also have "... ⊆ Λ"
    by auto
  finally have "2 * z' ∈ Λ" .
  thus "2 * z ∈ Λ" unfolding z'_def
    by (metis rel_to_fund_parallelogram_right mult_2 rel_add rel_lattice_trans_right)
next
  assume "2 * z ∈ Λ"
  thus "℘' z = 0"
    using assms weierstass_fun_deriv_half_period_eq_0[of "2*z"] by auto
qed

lemma zorder_weierstrass_fun_deriv_zero:
  assumes "z ∉ Λ" "2 * z ∈ Λ"
  shows   "zorder ℘' z = 1"
proof -
  have *: "℘' ≠ (λ_. 0)"
    using is_pole_weierstrass_fun_deriv_iff[of 0] by auto
  note ** [simp] = weierstrass_fun_deriv.isolated_zero_iff[OF *] is_pole_weierstrass_fun_de

  define w1 w2 w3 where "w1 = ω1 / 2" "w2 = ω2 / 2" "w3 = (ω1 + ω2) /
2"
```

```
  note w123_defs = this
  have [simp]: "w1 ∉ Λ" "w2 ∉ Λ" "w3 ∉ Λ" "2 * w1 ∈ Λ" "2 * w2 ∈ Λ"
"2 * w3 ∈ Λ"
   and "distinct [w1, w2, w3]"
    by (auto simp: w123_defs add_divide_distrib in_lattice_conv_ω12_coords
ω12_coords.add)
  define A where "A = {w1, w2, w3}"
  have [simp, intro]: "finite A" and [simp]: "card A = 3"
    using ‹distinct [w1, w2, w3]› by (auto simp: A_def)

  have in_parallelogram: "A ⊆ period_parallelogram 0"
    unfolding A_def period_parallelogram_altdef by (auto simp: w123_defs
ω12_coords.add)
  have [simp]: "℘' w = 0" if "w ∈ A" for w
    by (subst weierstrass_fun_deriv_eq_0_iff) (use that in ‹auto simp:
A_def›)
  have [intro]: "isolated_zero ℘' w" if "w ∈ A" for w
    using that by (subst **) (auto simp: A_def)

  have "(∑w∈A. nat (zorder ℘' w)) ≤ elliptic_order ℘'"
    unfolding weierstrass_fun_deriv.zeros_eq_elliptic_order[of 0, symmetric]
using in_parallelogram
    by (intro sum_mono2 weierstrass_fun_deriv.finite_zeros_in_parallelogram)
       (auto simp: A_def)
  also have "... = 3"
    by simp
  finally have le_3: "(∑w∈A. nat (zorder ℘' w)) ≤ 3" .

  have pos: "zorder ℘' w > 0" if "w ∈ A" for w
    by (rule zorder_isolated_zero_pos) (use that in ‹auto intro!: analytic_intros
simp: A_def›)

  have zorder_eq_1: "zorder ℘' w = 1" if "w ∈ A" for w
  proof (rule ccontr)
    assume *: "zorder ℘' w ≠ 1"
    have "(∑w∈A. nat (zorder ℘' w)) > (∑w∈A. 1)"
    proof (rule sum_strict_mono_strong[of _ w])
      show "nat (zorder ℘' w) ≥ 1" if "w ∈ A" for w
        using pos[of w] that by auto
    qed (use pos[of w] * ‹w ∈ A› in auto)
    with le_3 show False
      by simp
  qed

  from assms obtain w where w: "w ∈ A" "rel z w"
    using rel_half_period[of z] unfolding A_def w123_defs by metis
  with zorder_eq_1[of w] show ?thesis
    using weierstrass_fun_deriv.zorder.lattice_cong by metis
qed
```

end

6.5 Injectivity of $\wp$

context *complex_lattice*
begin

The function $\wp$ is almost injective in the sense that $\wp(u) = \wp(v)$ iff $u \sim v$ or $u \sim -v$. Another way to phrase this is that it is injective inside period half-parallelograms.

This is Exercise 1.3(a) in Apostol's book.

```
theorem weierstrass_fun_eq_iff:
  assumes "u ∉ Λ" "v ∉ Λ"
  shows   "℘ u = ℘ v ⟷ rel u v ∨ rel u (-v)"
proof (intro iffI)
  assume "rel u v ∨ rel u (-v)"
  thus "℘ u = ℘ v"
    by (metis weierstrass_fun.lattice_cong weierstrass_fun_minus)
next
  assume *: "℘ u = ℘ v"
  define c where "c = ℘ u"
  define f where "f = (λz. ℘ z - c)"
  interpret f: elliptic_function_affine ω1 ω2 ℘ 1 "-c" f
  proof -
    show "elliptic_function_affine ω1 ω2 ℘ 1"
      by standard auto
  qed (simp_all add: f_def)
  interpret f: even_elliptic_function ω1 ω2 f
    by standard (simp_all add: f_def weierstrass_fun_minus)
  interpret f: elliptic_function_remove_sings ω1 ω2 f ..

  have ana: "f analytic_on {x}" if "x ∉ Λ" for x using that
    unfolding f_def by (auto intro!: analytic_intros)
  have ana': "remove_sings f analytic_on {x}" if "x ∉ Λ" for x using
that
    by (auto simp: f.remove_sings.analytic_at_iff_not_pole f.is_pole_affine_iff

                   is_pole_weierstrass_fun_iff)
  have [simp]: "remove_sings f x = f x" if "x ∉ Λ" for x
    using ana[OF that] by simp
  have [simp]: "f u = 0" "f v = 0"
    using * by (auto simp: f_def c_def)

  have order: "elliptic_order f = 2"
    by (simp add: f.order_affine_eq)
  have nz: "remove_sings f ≠ (λ_. 0)"
  proof
    assume "remove_sings f = (λ_. 0)"
```

```
    hence "remove_sings f constant_on UNIV"
      by (auto simp: constant_on_def)
    thus False
      using f.remove_sings.elliptic_order_eq_0_iff order by simp
  qed
  have zero_f_iff [simp]: "isolated_zero f z ⟷ f z = 0" if "z ∉ Λ"
for z
    using that ana f.affine.isolated_zero_analytic_iff
          f.affine.elliptic_order_eq_0_iff_const_cosparse local.order
by auto

  define u' where "u' = to_half_fund_parallelogram u"
  define v' where "v' = to_half_fund_parallelogram v"
  define Z where "Z = {z ∈ period_parallelogram 0. z ∉ Λ ∧ f z = 0}"
  define Z1 where "Z1 = {z ∈ half_fund_parallelogram. z ∉ Λ ∧ f z =
0}"
  define Z2 where "Z2 = to_fund_parallelogram ‘ uminus ‘ Z1"

  have Z: "(∑z∈Z. nat (zorder f z)) = 2" "finite Z"
  proof -
    have "(∑z∈{z∈period_parallelogram 0. isolated_zero (remove_sings
f) z}.
                  nat (zorder (remove_sings f) z)) = 2 ∧
          finite {z∈period_parallelogram 0. isolated_zero (remove_sings
f) z}"
      using order f.remove_sings.zeros_eq_elliptic_order[of 0]
            f.remove_sings.finite_zeros_in_parallelogram[of 0] by simp
    also have "{z∈period_parallelogram 0. isolated_zero (remove_sings
f) z} =
               {z∈period_parallelogram 0. z ∉ Λ ∧ remove_sings f z =
0}"
      by (subst f.remove_sings.isolated_zero_iff)
         (use nz f.is_pole_affine_iff is_pole_weierstrass_fun_iff in auto)
    also have "... = {z∈period_parallelogram 0. z ∉ Λ ∧ f z = 0}"
      by (intro Collect_cong conj_cong) auto
    finally show "(∑z∈Z. nat (zorder f z)) = 2" "finite Z"
      unfolding Z_def by auto
  qed
  have "finite Z1"
    by (rule finite_subset[OF _ Z(2)])
       (use half_fund_parallelogram_subset_period_parallelogram in ‹auto
simp: Z_def Z1_def›)
  hence "finite Z2"
    by (simp_all add: Z2_def)
  note finite = ‹finite Z› ‹finite Z1› ‹finite Z2›

  have subset: "Z1 ⊆ Z" "Z2 ⊆ Z"
    unfolding Z_def Z1_def Z2_def
    using half_fund_parallelogram_subset_period_parallelogram
```

```
    by (auto simp: uminus_in_lattice_iff f.affine.eval_to_fund_parallelogram
f.even)

  have "card Z1 + card Z2 = card (Z1 ∪ Z2) + card (Z1 ∩ Z2)"
    by (rule card_Un_Int) (use finite in auto)
  also have "card Z2 = card Z1" unfolding Z2_def image_image
    by (intro card_image inj_onI)
       (auto simp: Z1_def rel_minus_iff intro: rel_in_half_fund_parallelogram_imp_eq)
  also have "card (Z1 ∪ Z2) ≤ card Z"
    by (intro card_mono finite) (use subset in auto)
  also have "... = (∑z∈Z. 1)"
    by simp
  also have "card (Z1 ∩ Z2) = (∑z∈Z1 ∩ Z2. 1)"
    by simp
  also have "... = (∑z∈Z. if z ∈ Z1 ∩ Z2 then 1 else 0)"
    by (intro sum.mono_neutral_cong_left) (use finite subset in auto)
  also have "(∑z∈Z. 1) + ... = (∑z∈Z. if z ∈ Z1 ∩ Z2 then 2 else 1)"
    by (subst sum.distrib [symmetric], intro sum.cong) auto
  also have "... ≤ (∑z∈Z. nat (zorder f z))"
  proof (intro sum_mono)
    fix z assume z: "z ∈ Z"
    hence "zorder f z > 0"
      by (intro zorder_isolated_zero_pos ana) (auto simp: Z_def)
    moreover have "zorder f z ≥ 2" if z: "z ∈ Z1 ∩ Z2"
    proof -
      obtain z' where "to_fund_parallelogram (-z') ∈ half_fund_parallelogram"
"z' ∉ Λ"
                      "z = to_fund_parallelogram (-z')" "z' ∈ half_fund_parallelogram"
        using z by (auto simp: Z1_def Z2_def uminus_in_lattice_iff)
      hence "2 * z ∈ Λ"
        by (metis in_half_fund_parallelogram_imp_half_lattice
            rel_in_half_fund_parallelogram_imp_eq rel_to_fund_parallelogram_left)
      moreover have "¬(∀≈z. f z = 0)" using nz
        using f.affine.elliptic_order_eq_0_iff_const_cosparse local.order
by auto
      ultimately have "even (zorder f z)"
        by (intro f.even_zorder)
      with ‹zorder f z > 0› show "zorder f z ≥ 2"
        by presburger
    qed
    ultimately show "(if z ∈ Z1 ∩ Z2 then 2 else 1) ≤ nat (zorder f
z)"
      by auto
  qed
  also have "... = 2"
    by (fact Z(1))
  finally have "card Z1 ≤ 1"
    by simp
  moreover have "card {u', v'} ≤ card Z1" using assms
```

```
    by (intro card_mono finite) (auto simp: Z1_def u'_def v'_def f.eval_to_half_fund_parall
  ultimately have "card {u', v'} ≤ 1"
    by simp
  hence "u' = v'"
    by (cases "u' = v'") simp_all
  thus "rel u v ∨ rel u (-v)"
    unfolding u'_def v'_def by (simp add: to_half_fund_parallelogram_eq_iff)
qed
```

It is also surjective. Together with the fact that it is doubly periodic and even, this means that it takes on every value exactly once inside its period triangles, or twice within its period parallelograms. Note however that the multiplicities of the poles on the lattice points and of the values e_1, e_2, e_3 at the half-lattice points are 2.

```
lemma surj_weierstrass_fun:
  obtains z where "z ∈ period_parallelogram w - Λ" "℘ z = c"
  using weierstrass_fun.surj[of w c]
  by (auto simp: is_pole_weierstrass_fun_iff)

lemma surj_weierstrass_fun_deriv:
  obtains z where "z ∈ period_parallelogram w - Λ" "℘' z = c"
  using weierstrass_fun_deriv.surj[of w c]
  by (auto simp: is_pole_weierstrass_fun_deriv_iff)

end


context complex_lattice_swap
begin

lemma weierstrass_fun_aux_swap [simp]: "swap.weierstrass_fun_aux = weierstrass_fun_aux"
  unfolding weierstrass_fun_aux_def [abs_def] swap.weierstrass_fun_aux_def
[abs_def] by auto

lemma weierstrass_fun_swap [simp]: "swap.weierstrass_fun = weierstrass_fun"
  unfolding weierstrass_fun_def [abs_def] swap.weierstrass_fun_def [abs_def]
by auto

lemma number_e1_swap [simp]: "swap.number_e1 = number_e2"
  and number_e2_swap [simp]: "swap.number_e2 = number_e1"
  and number_e3_swap [simp]: "swap.number_e3 = number_e3"
  unfolding number_e2_def swap.number_e1_def number_e1_def swap.number_e2_def
            number_e3_def swap.number_e3_def by (simp_all add: add_ac)

end
```

6.6 Invariance under lattice transformations

We show how various concepts related to lattices (e.g. the Weierstraß $\wp$ function, the numbers e_1, e_2, e_3) transform under various transformations of the lattice. Namely: complex conjugation, swapping the generators, stretching/rotation, and unimodular Möbius transforms.

```
locale complex_lattice_cnj = complex_lattice
begin

sublocale cnj: complex_lattice "cnj ω1" "cnj ω2"
  by unfold_locales (use fundpair in auto)

lemma bij_betw_lattice_cnj: "bij_betw cnj lattice cnj.lattice"
  by (rule bij_betwI[of _ _ _ cnj])
     (auto elim!: latticeE cnj.latticeE simp: of_ω12_coords_def cnj.of_ω12_coords_def
           intro!: lattice_intros cnj.lattice_intros)

lemma bij_betw_lattice0_cnj: "bij_betw cnj lattice0 cnj.lattice0"
  unfolding lattice0_def cnj.lattice0_def
  by (intro bij_betw_DiffI bij_betw_lattice_cnj) auto

lemma lattice_cnj_eq: "cnj.lattice = cnj ` lattice"
  using bij_betw_lattice_cnj by (auto simp: bij_betw_def)

lemma lattice0_cnj_eq: "cnj.lattice0 = cnj ` lattice0"
  using bij_betw_lattice0_cnj by (auto simp: bij_betw_def)

lemma eisenstein_fun_aux_cnj: "cnj.eisenstein_fun_aux n z = cnj (eisenstein_fun_aux
n (cnj z))"
  unfolding eisenstein_fun_aux_def cnj.eisenstein_fun_aux_def
  by (subst infsum_reindex_bij_betw[OF bij_betw_lattice0_cnj, symmetric])
     (auto simp flip: infsum_cnj simp: lattice0_cnj_eq in_image_cnj_iff)

lemma weierstrass_fun_aux_cnj: "cnj.weierstrass_fun_aux z = cnj (weierstrass_fun_aux
(cnj z))"
  unfolding weierstrass_fun_aux_def cnj.weierstrass_fun_aux_def
  by (subst infsum_reindex_bij_betw[OF bij_betw_lattice0_cnj, symmetric])
     (auto simp flip: infsum_cnj simp: lattice0_cnj_eq in_image_cnj_iff)

lemma weierstrass_fun_cnj: "cnj.weierstrass_fun z = cnj (weierstrass_fun
(cnj z))"
  unfolding weierstrass_fun_def cnj.weierstrass_fun_def
  by (auto simp: lattice_cnj_eq in_image_cnj_iff weierstrass_fun_aux_cnj)

lemma number_e1_cnj [simp]: "cnj.number_e1 = cnj number_e1"
  and number_e2_cnj [simp]: "cnj.number_e2 = cnj number_e2"
  and number_e3_cnj [simp]: "cnj.number_e3 = cnj number_e3"
  by (simp_all add: number_e1_def cnj.number_e1_def number_e2_def
                    cnj.number_e2_def number_e3_def cnj.number_e3_def
```

```
weierstrass_fun_cnj)

end


locale complex_lattice_stretch = complex_lattice +
  fixes c :: complex
  assumes stretch_nonzero: "c ≠ 0"
begin

sublocale stretched: complex_lattice "c * ω1" "c * ω2"
  by unfold_locales (use fundpair in <auto simp: stretch_nonzero fundpair_def>)

lemma stretched_of_ω12_coords: "stretched.of_ω12_coords ab = c * of_ω12_coords
ab"
  unfolding stretched.of_ω12_coords_def of_ω12_coords_def
  by (auto simp: case_prod_unfold algebra_simps)

lemma stretched_ω12_coords: "stretched.ω12_coords ab = ω12_coords (ab
/ c)"
  using stretch_nonzero stretched_of_ω12_coords
  by (metis mult.commute nonzero_divide_eq_eq of_ω12_coords_ω12_coords
stretched.ω12_coords_eqI)

lemma stretched_ω1_coord: "stretched.ω1_coord ab = ω1_coord (ab / c)"
  and stretched_ω2_coord: "stretched.ω2_coord ab = ω2_coord (ab / c)"
  using stretched_ω12_coords[of ab] by (simp_all add: stretched.ω12_coords_def
ω12_coords_def)

lemma mult_into_stretched_lattice: "(*) c ∈ Λ → stretched.lattice"
  by (auto elim!: latticeE simp: stretched.in_lattice_conv_ω12_coords

        stretched_ω12_coords zero_prod_def)

lemma mult_into_stretched_lattice': "(*) (inverse c) ∈ stretched.lattice
→ Λ"
proof -
  interpret inv: complex_lattice_stretch "c * ω1" "c * ω2" "inverse c"
    by unfold_locales (use stretch_nonzero in auto)
  from inv.mult_into_stretched_lattice show ?thesis
    by (simp add: inv.stretched.lattice_def stretched.lattice_def field_simps
stretch_nonzero)
qed

lemma bij_betw_stretch_lattice: "bij_betw ((*) c) lattice stretched.lattice"
proof (rule bij_betwI[of _ _ _ "(*) (inverse c)"])
  show "(*) c ∈ Λ → stretched.lattice"
    by (rule mult_into_stretched_lattice)
  show "(*) (inverse c) ∈ stretched.lattice → Λ"
```

```
    by (rule mult_into_stretched_lattice')
qed (auto simp: stretch_nonzero)

lemma bij_betw_stretch_lattice0:
  "bij_betw ((*) c) lattice0 stretched.lattice0"
  unfolding lattice0_def stretched.lattice0_def
  by (intro bij_betw_DiffI bij_betw_stretch_lattice) auto

end


locale unimodular_moebius_transform_lattice = complex_lattice + unimodular_moebius_transform
begin

definition ω1' where "ω1' = of_int c * ω2 + of_int d * ω1"
definition ω2' where "ω2' = of_int a * ω2 + of_int b * ω1"

sublocale transformed: complex_lattice ω1' ω2'
proof unfold_locales
  define τ where "τ = ω2 / ω1"
  have "Im (φ τ) ≠ 0"
    using fundpair Im_transform_zero_iff[of τ] unfolding τ_def
    by (auto simp: fundpair_def complex_is_Real_iff)
  also have "φ τ = ω2' / ω1'"
    by (simp add: φ_def ω1'_def ω2'_def moebius_def τ_def divide_simps)
  finally show "fundpair (ω1', ω2')"
    by (simp add: φ_def ω1'_def ω2'_def moebius_def τ_def field_simps

                  fundpair_def complex_is_Real_iff)
qed

lemma transformed_lattice_subset: "transformed.lattice ⊆ lattice"
proof safe
  fix z assume "z ∈ transformed.lattice"
  then obtain m n where mn: "z = of_int m * ω1' + of_int n * ω2'"
    by (elim transformed.latticeE) (auto simp: transformed.of_ω12_coords_def)
  also have "of_int m * ω1' + of_int n * ω2' =
             of_int (d * m + b * n) * ω1 + of_int (c * m + a * n) * ω2"
    by (simp add: algebra_simps ω1'_def ω2'_def)
  finally show "z ∈ lattice"
    by (auto intro!: lattice_intros simp: ring_distribs mult.assoc)
qed

lemma transformed_lattice_eq: "transformed.lattice = lattice"
proof -
  interpret inverse_unimodular_moebius_transform a b c d ..
  interpret inv: unimodular_moebius_transform_lattice ω1' ω2' d "-b" "-c"
a ..
  have [simp]: "inv.ω1' = ω1" "inv.ω2' = ω2"
```

```
    unfolding inv.ω1'_def inv.ω2'_def unfolding ω1'_def ω2'_def of_int_minus
using unimodular
    by (simp_all add: algebra_simps flip: of_int_mult)

  have "inv.transformed.lattice ⊆ transformed.lattice"
    by (rule inv.transformed_lattice_subset)
  also have "inv.transformed.lattice = lattice"
    unfolding inv.transformed.lattice_def unfolding lattice_def by simp
  finally show ?thesis
    using transformed_lattice_subset by blast
qed

lemma transformed_lattice0_eq: "transformed.lattice0 = lattice0"
  by (simp add: transformed.lattice0_def lattice0_def transformed_lattice_eq)

lemma eisenstein_fun_aux_transformed [simp]: "transformed.eisenstein_fun_aux
= eisenstein_fun_aux"
  by (intro ext) (simp add: transformed.eisenstein_fun_aux_def eisenstein_fun_aux_def
                            transformed_lattice0_eq)

lemma weierstrass_fun_aux_transformed [simp]: "transformed.weierstrass_fun_aux
= weierstrass_fun_aux"
  by (intro ext, unfold weierstrass_fun_aux_def transformed.weierstrass_fun_aux_def

      transformed_lattice0_eq) simp_all

lemma weierstrass_fun_transformed [simp]: "transformed.weierstrass_fun
= weierstrass_fun"
  by (intro ext, simp add: weierstrass_fun_def transformed.weierstrass_fun_def
transformed_lattice_eq)

end


locale complex_lattice_apply_modgrp = complex_lattice +
  fixes f :: modgrp
begin

sublocale unimodular_moebius_transform_lattice
  ω1 ω2 "modgrp_a f" "modgrp_b f" "modgrp_c f" "modgrp_d f"
  rewrites "modgrp.as_modgrp = (λx. x)" and "modgrp.φ = apply_modgrp"
    by unfold_locales simp_all

end
```

6.7 Construction of arbitrary elliptic functions from $\wp$

In this section we will show that any elliptic function can be written as a combination of $\wp$ and $\wp'$. The key step is to show that every even elliptic

function can be written as a rational function of $\wp$.

The first step is to show that if $w \notin \Lambda$, the function $f(z) = \wp(z) - \wp(w)$ has a double zero at w if w is a half-lattice point and simple zeros at $\pm w$ otherwise, and no other zeros.

```
locale weierstrass_fun_minus_const = complex_lattice +
  fixes w :: complex and f :: "complex ⇒ complex"
  assumes not_in_lattice: "w ∉ Λ"
  defines "f ≡ (λz. ℘ z - ℘ w)"
begin

sublocale elliptic_function_affine ω1 ω2 ℘ 1 "-℘ w" f
  unfolding f_def by unfold_locales (auto simp: f_def)

lemmas order_eq = order_affine_eq
lemmas is_pole_iff = is_pole_affine_iff
lemmas zorder_pole_eq = zorder_pole_affine

lemma isolated_zero_iff: "isolated_zero f z ⟷ rel z w ∨ rel z (-w)"
proof (cases "z ∈ Λ")
  case False
  hence "f analytic_on {z}"
    unfolding f_def by (auto intro!: analytic_intros)
  moreover have "¬(∀⇩≈z. f z = 0)"
    using affine.elliptic_order_eq_0_iff_const_cosparse order_affine_eq
by auto
  ultimately have "isolated_zero f z ⟷ ℘ z = ℘ w"
    by (subst affine.isolated_zero_analytic_iff) (auto simp: f_def)
  also have "... ⟷ rel z w ∨ rel z (-w)"
    by (rule weierstrass_fun_eq_iff) (use not_in_lattice False in auto)
  finally show ?thesis .
next
  case True
  thus ?thesis
    using not_in_lattice is_pole_iff is_pole_weierstrass_fun pole_is_not_zero
      pre_complex_lattice.rel_lattice_trans_left uminus_in_lattice_iff
by blast
qed

lemma zorder_zero_eq:
  assumes "rel z w ∨ rel z (-w)"
  shows   "zorder f z = (if 2 * w ∈ Λ then 2 else 1)"
proof (cases "2 * w ∈ Λ")
  case False
  have z: "z ∉ Λ"
    using assms not_in_lattice pre_complex_lattice.rel_lattice_trans_left
          uminus_in_lattice_iff by blast
  have z': "2 * z ∉ Λ"
    using assms False z
```

```
    by (metis minus_zero minus_minus not_in_lattice weierstrass_fun_deriv.lattice_cong
        weierstrass_fun_deriv_eq_0_iff weierstrass_fun_deriv_minus z)
  have "zorder f z = 1"
  proof (rule zorder_zero_eqI')
    show "f analytic_on {z}"
      using not_in_lattice z by (auto simp: f_def intro!: analytic_intros)
  next
    show "(deriv ^^ i) f z = 0" if "i < nat 1" for i
      using that z not_in_lattice assms by (auto simp: f_def weierstrass_fun_eq_iff)
  next
    have "deriv f z = ℘' z" unfolding f_def
      by (rule DERIV_imp_deriv) (use z in ‹auto intro!: derivative_eq_intros›)
    also have "℘' z ≠ 0"
      using not_in_lattice False z assms z' by (auto simp: weierstrass_fun_deriv_eq_0_iff)
    finally show "(deriv ^^ nat 1) f z ≠ 0"
      by simp
  qed auto
  with False show ?thesis
    by simp
next
  case True
  have z: "z ∉ Λ"
    using assms not_in_lattice pre_complex_lattice.rel_lattice_trans_left
          uminus_in_lattice_iff by blast
  have z': "2 * z ∈ Λ"
    using assms True z
    by (metis minus_zero minus_minus not_in_lattice weierstrass_fun_deriv.lattice_cong
        weierstrass_fun_deriv_eq_0_iff weierstrass_fun_deriv_minus z)

  have "eventually (λw. f w ≠ 0) (at 0)"
    by (simp add: affine.avoid' order_affine_eq)
  moreover have "eventually (λw. w ∉ Λ) (at 0)"
    using islimpt_iff_eventually not_islimpt_lattice by auto
  ultimately have ev: "eventually (λw. f w ≠ 0 ∧ w ∉ Λ) (at 0)"
    by eventually_elim auto
  obtain z0 where z0: "z0 ∉ Λ" "f z0 ≠ 0"
    using eventually_happens[OF ev] by auto

  have "(∑z∈{z}. nat (zorder f z)) ≤
        (∑z' | z' ∈ period_parallelogram z ∧ isolated_zero f z'. nat
(zorder f z'))"
    using assms by (intro sum_mono2 affine.finite_zeros_in_parallelogram)
(auto simp: isolated_zero_iff)
  also have "... = 2"
    using affine.zeros_eq_elliptic_order[of z] order_eq by simp
  finally have "zorder f z ≤ 2"
    by simp

  moreover have "zorder f z ≥ int 2"
```

```
  proof (rule zorder_geI)
    show "f holomorphic_on -Λ"
      by (auto intro!: holomorphic_intros simp: f_def)
  next
    show "z0 ∈ -Λ" "f z0 ≠ 0"
      using z0 by auto
  next
    show "open (-Λ)"
      using closed_lattice by auto
  next
    have "Λ sparse_in UNIV"
      using not_islimpt_lattice sparse_in_def by blast
    hence "connected (UNIV - Λ)"
      by (intro sparse_imp_connected) auto
    also have "UNIV - Λ = -Λ"
      by auto
    finally show "connected (-Λ)" .
  next
    have "deriv f z = ℘' z"
      by (rule DERIV_imp_deriv) (auto simp: f_def intro!: derivative_eq_intros
z)
    also have "... = 0"
      using True z z' weierstrass_fun_deriv_eq_0_iff by blast
    finally have "deriv f z = 0" .
    moreover have "f z = 0"
      using not_in_lattice z z' assms by (simp add: f_def weierstrass_fun_eq_iff)
    ultimately show "(deriv ^^ n) f z = 0" if "n < 2" for n
      using that by (cases n) auto
  qed (use z in ‹auto intro!: analytic_intros simp: f_def›)
  ultimately have "zorder f z = 2"
    by linarith
  thus ?thesis
    using True by simp
qed

lemma zorder_zero_eq':
  assumes "z ∉ Λ"
  shows   "zorder f z = (if rel z w ∨ rel z (-w) then if 2 * w ∈ Λ then
2 else 1 else 0)"
proof (cases "rel z w ∨ rel z (-w)")
  case True
  thus ?thesis
    using zorder_zero_eq[OF True] by auto
next
  case False
  have "f analytic_on {z}"
    using assms by (auto simp: f_def intro!: analytic_intros)
  moreover have "f z ≠ 0"
    using assms not_in_lattice False by (auto simp: f_def weierstrass_fun_eq_iff)
```

```
  ultimately have "zorder f z = 0"
    by (intro zorder_eq_0I) auto
  thus ?thesis
    using False by simp
qed

end


lemma (in complex_lattice) zorder_weierstrass_fun_minus_const:
  assumes "w ∉ Λ" "z ∉ Λ"
  shows   "zorder (λz. ℘ z - ℘ w) z =
             (if rel z w ∨ rel z (-w) then if 2 * w ∈ Λ then 2 else 1
else 0)"
proof -
  interpret weierstrass_fun_minus_const ω1 ω2 w "λz. ℘ z - ℘ w"
    by unfold_locales (use assms in auto)
  show ?thesis
    using zorder_zero_eq'[of z] assms by simp
qed
```

We now construct an elliptic function

$$g(z) = \prod_{w \in A} (\wp(z) - \wp(w))^{h(w)}$$

where $A \subseteq \mathbb{C} \setminus \Lambda$ is finite and $h : A \to \mathbb{Z}$.

We will examine what the zeros and poles of this functions are and what their multiplicities are.

This is roughly Exercise 1.3(b) in Apostol's book.

```
locale elliptic_function_construct = complex_lattice +
  fixes A :: "complex set" and h :: "complex ⇒ int" and g :: "complex
⇒ complex"
  assumes finite [intro]: "finite A" and no_lattice_points: "A ∩ Λ =
{}"
  defines "g ≡ (λz. (∏w∈A. (℘ z - ℘ w) powi h w))"
begin

sublocale elliptic_function ω1 ω2 g
  unfolding g_def by (intro elliptic_function_intros)

sublocale even_elliptic_function ω1 ω2 g
  by standard (simp add: g_def weierstrass_fun_minus)

lemma no_lattice_points': "w ∉ Λ" if "w ∈ A" for w
  using no_lattice_points that by blast

lemma eq_0_iff: "g z = 0 ⟷ (∃w∈A. h w ≠ 0 ∧ (rel z w ∨ rel z (-w)))"
if "z ∉ Λ" for z
```

```
    using finite that by (auto simp: g_def weierstrass_fun_eq_iff no_lattice_points')

lemma nonzero_almost_everywhere: "eventually (λz. g z ≠ 0) (cosparse
UNIV)"
proof -
  have "{z. g z = 0} ⊆ Λ ∪ (⋃w∈A. (+) w ‘ Λ) ∪ (⋃w∈A. (+) (-w) ‘
Λ)"
    using eq_0_iff by (force simp: rel_def)
  moreover have "... sparse_in UNIV"
    by (intro sparse_in_union' sparse_in_UN_finite finite_imageI
          finite sparse_in_translate_UNIV lattice_sparse)
  ultimately have "{z. g z = 0} sparse_in UNIV"
    using sparse_in_subset2 by blast
  thus ?thesis
    by (simp add: eventually_cosparse)
qed

lemma eventually_nonzero_at: "eventually (λz. g z ≠ 0) (at z)"
  using nonzero_almost_everywhere by (auto simp: eventually_cosparse_open_eq)

lemma zorder_eq:
  assumes z: "z ∉ Λ"
  shows   "zorder g z =
             (∑w∈A. if rel z w ∨ rel z (-w) then if 2*w ∈ Λ then 2
* h w else h w else 0)"
proof -
  have [simp]: "w ∉ Λ" if "w ∈ A" for w
    using no_lattice_points that by blast

  have "zorder g z = (∑x∈A. zorder (λz. (℘ z - ℘ x) powi h x) z)"
    unfolding g_def
  proof (rule zorder_prod)
    show "∀F z in at z. (∏x∈A. (℘ z - ℘ x) powi h x) ≠ 0"
      using eventually_nonzero_at[of z] by (simp add: g_def)
  qed (auto intro!: meromorphic_intros)
  also have "... = (∑w∈A. if rel z w ∨ rel z (-w) then if 2*w ∈ Λ then
2 * h w else h w else 0)"
  proof (intro sum.cong refl)
    fix w assume "w ∈ A"
    have "zorder (λz. (℘ z - ℘ w) powi h w) z = h w * zorder (λz. (℘
z - ℘ w)) z"
    proof (cases "h w = 0")
      case [simp]: False
      have "∀F z in at z. ℘ z - ℘ w ≠ 0"
        using eventually_nonzero_at[of z]
        by eventually_elim (use ‹w ∈ A› finite in ‹auto simp: g_def›)

      hence "∃F z in at z. ℘ z - ℘ w ≠ 0"
        using eventually_frequently at_neq_bot by blast
```

```
    thus ?thesis
      by (intro zorder_power_int) (auto intro!: meromorphic_intros)
  qed auto
  also have "zorder (λz. ℘ z - ℘ w) z =
              (if rel z w ∨ rel z (-w) then if 2*w ∈ Λ then 2 else
1 else 0)"
    using zorder_weierstrass_fun_minus_const[of w z] z ‹w ∈ A› by simp
  also have "h w * ... = (if rel z w ∨ rel z (-w) then if 2*w ∈ Λ then
2 * h w else h w else 0)"
    by auto
  finally show "zorder (λz. (℘ z - ℘ w) powi h w) z =
                  (if rel z w ∨ rel z (-w) then if 2*w ∈ Λ then 2 *
h w else h w else 0)" .
qed
finally show "zorder g z = (∑w∈A. if rel z w ∨ rel z (-w) then if
2*w ∈ Λ then 2 * h w else h w else 0)" .
qed

end


lemma (in even_elliptic_function) in_terms_of_weierstrass_fun_even_aux:
  assumes nontrivial: "¬eventually (λz. f z = 0) (cosparse UNIV)"
  defines "Z ≡ {z∈half_fund_parallelogram - {0}. is_pole f z ∨ isolated_zero
f z}"
  defines "h ≡ (λz. if z ∈ Z then zorder f z div (if 2 * z ∈ Λ then
2 else 1) else 0)"
  obtains c where "eventually (λz. f z = c * (∏w∈Z. (℘ z - ℘ w) powi
h w)) (cosparse UNIV)"
proof -
  define g where "g = (λz. (∏w∈Z. (℘ z - ℘ w) powi h w))"
  have [intro]: "finite Z"
  proof (rule finite_subset)
    show "Z ⊆ {z∈period_parallelogram 0. is_pole f z} ∪ {z∈period_parallelogram
0. isolated_zero f z}"
      using half_fund_parallelogram_subset_period_parallelogram by (auto
simp: Z_def)
    show "finite ({z∈period_parallelogram 0. is_pole f z} ∪ {z∈period_parallelogram
0. isolated_zero f z})"
      by (intro finite_UnI finite_poles_in_parallelogram finite_zeros_in_parallelogram)
  qed
  have [simp]: "z ∉ Λ" if "z ∈ Z" for z
    using that half_fund_parallelogram_in_lattice_iff[of z] unfolding
Z_def by auto

  interpret g: elliptic_function_construct ω1 ω2 Z h g
    by unfold_locales (auto simp: g_def)

  have zorder_eq_aux: "zorder g z = zorder f z" if z: "z ∈ half_fund_parallelogram
```

```
- Λ" for z
  proof -
    have "zorder g z = (∑w∈Z. if rel z w ∨ rel z (-w) then if 2*w ∈
Λ then 2 * h w else h w else 0)"
      by (rule g.zorder_eq) (use z in auto)
    also have "... = (∑w∈Z∩{z}. if 2*w ∈ Λ then 2 * h w else h w)"
    proof (intro sum.mono_neutral_cong_right ballI)
      fix w assume w: "w ∈ Z - Z ∩ {z}"
      thus "(if rel z w ∨ rel z (-w) then if 2 * w ∈ Λ then 2 * h w else
h w else 0) = 0"
        using rel_in_half_fund_parallelogram_imp_eq[of z w] z by (auto
simp: Z_def)
    qed auto
    also have "... = (if 2 * z ∈ Λ then 2 * h z else h z)"
      by (auto simp: h_def)
    also have "... = (if z ∈ Z then zorder f z else 0)"
      using even_zorder[of z] nontrivial by (auto simp: h_def)
    also have "... = zorder f z"
    proof (cases "z ∈ Z")
      case False
      hence "¬is_pole f z ∧ ¬isolated_zero f z"
        using z by (auto simp: Z_def)
      moreover have "frequently (λz. f z ≠ 0) (at z)"
        using nontrivial by (metis eventually_eq_imp_almost_everywhere_eq
not_eventually)
      ultimately have "zorder f z = 0"
        by (intro not_pole_not_isolated_zero_imp_zorder_eq_0) (auto intro:
meromorphic')
      thus ?thesis
        by simp
    qed auto
    finally show ?thesis .
  qed

  have zorder_eq: "zorder g z = zorder f z" if z: "z ∉ Λ" for z
  proof -
    have "zorder g z = zorder g (to_half_fund_parallelogram z)"
      using g.zorder_to_half_fund_parallelogram by simp
    also have "... = zorder f (to_half_fund_parallelogram z)"
      by (rule zorder_eq_aux) (use z in auto)
    also have "... = zorder f z"
      using zorder_to_half_fund_parallelogram by simp
    finally show ?thesis .
  qed

  define h where "h = (λz. f z / g z)"
  interpret h: elliptic_function ω1 ω2 h
    unfolding h_def by (intro elliptic_function_intros)
```

```
  have h_nonzero: "eventually (λz. h z ≠ 0) (at z)" for z
  proof -
    have "eventually (λz. f z ≠ 0) (at z)"
      using nontrivial frequently_def frequently_eq_imp_almost_everywhere_eq
by blast
    thus ?thesis
      using g.eventually_nonzero_at by eventually_elim (auto simp: h_def)
  qed

  have zorder_h: "zorder h z = 0" if z: "z ∉ Λ" for z
  unfolding h_def
  proof (subst zorder_divide)
    show "∃F z in at z. f z ≠ 0"
      using nontrivial by (metis eventually_eq_imp_almost_everywhere_eq
not_eventually)
    show "∃F z in at z. g z ≠ 0"
      using eventually_frequently g.eventually_nonzero_at trivial_limit_at
by blast
  qed (use z zorder_eq[of z] in ‹auto intro!: meromorphic' g.meromorphic'›)

  have zorder_h': "zorder h z = 0" if z: "z ∈ period_parallelogram 0"
"z ≠ 0" for z
    by (rule zorder_h) (use z fund_period_parallelogram_in_lattice_iff[of
z] in auto)

  have "elliptic_order h = 0"
  proof -
    have "elliptic_order h = (∑z∈(if is_pole h 0 then {0} else {}).
nat (-zorder h z))"
      unfolding elliptic_order_def
      by (intro sum.mono_neutral_right h.finite_poles_in_parallelogram)
         (auto dest: zorder_h')
    also have "... = nat (-zorder h 0)"
      using zorder_neg_imp_is_pole[OF h.meromorphic', of 0] h_nonzero
            linorder_not_less[of "zorder h 0" 0] by auto
    finally have *: "elliptic_order h = nat (-zorder h 0)" .

    have "elliptic_order h = (∑z∈(if isolated_zero h 0 then {0} else
{}). nat (zorder h z))"
      unfolding h.zeros_eq_elliptic_order[of 0, symmetric]
      by (intro sum.mono_neutral_right h.finite_zeros_in_parallelogram)
         (auto dest: zorder_h')
    also have "... = nat (zorder h 0)"
      using zorder_pos_imp_isolated_zero[OF h.meromorphic', of 0] h_nonzero
            linorder_not_less[of 0 "zorder h 0"] by auto
    finally have "elliptic_order h = nat (zorder h 0)" .
    with * show "elliptic_order h = 0"
      by simp
  qed
```

```
  then obtain c where c: "eventually (λz. h z = c) (cosparse UNIV)"
    using h.elliptic_order_eq_0_iff_const_cosparse by blast
  moreover have "eventually (λz. g z ≠ 0) (cosparse UNIV)"
    using g.nonzero_almost_everywhere by blast
  ultimately have "eventually (λz. f z = c * g z) (cosparse UNIV)"
    by eventually_elim (auto simp: h_def)
  thus ?thesis
    using that[of c] unfolding g_def by blast
qed
```

Finally, we show that any even elliptic function can be written as a rational function of $\wp$. This is Exercise 1.4 in Apostol's book.

```
lemma (in even_elliptic_function) in_terms_of_weierstrass_fun_even:
  obtains p q :: "complex poly" where "q ≠ 0" "∀≈z. f z = poly p (℘
z) / poly q (℘ z)"
proof (cases "eventually (λz. f z = 0) (cosparse UNIV)")
  case True
  thus ?thesis
    using that[of 1 0] by simp
next
  case False
  define Z where "Z = {z∈half_fund_parallelogram - {0}. is_pole f z ∨
isolated_zero f z}"
  define h where "h = (λz. if z ∈ Z then zorder f z div (if 2 * z ∈
Λ then 2 else 1) else 0)"
  obtain c where *: "eventually (λz. f z = c * (∏w∈Z. (℘ z - ℘ w) powi
h w)) (cosparse UNIV)"
    using False in_terms_of_weierstrass_fun_even_aux unfolding Z_def h_def
by metis
  define p where "p = Polynomial.smult c (∏w∈{w∈Z. h w ≥ 0}. [:-℘
w, 1:] ^ nat (h w))"
  define q where "q = (∏w∈{w∈Z. h w < 0}. [:-℘ w, 1:] ^ nat (-h w))"

  have finite: "finite Z"
  proof (rule finite_subset)
    show "Z ⊆ {z∈period_parallelogram 0. is_pole f z} ∪ {z∈period_parallelogram
0. isolated_zero f z}"
      using half_fund_parallelogram_subset_period_parallelogram by (auto
simp: Z_def)
    show "finite ({z∈period_parallelogram 0. is_pole f z} ∪ {z∈period_parallelogram
0. isolated_zero f z})"
      by (intro finite_UnI finite_poles_in_parallelogram finite_zeros_in_parallelogram)
  qed

  show ?thesis
  proof (rule that[of q p])
    show "q ≠ 0"
      using finite by (auto simp: q_def)
  next
```

```
    show "∀≈z. f z = poly p (℘ z) / poly q (℘ z)"
      using *
    proof eventually_elim
      case (elim z)
      have "f z = c * (∏w∈Z. (℘ z - ℘ w) powi h w)"
        by (fact elim)
      also have "Z = {w∈Z. h w ≥ 0} ∪ {w∈Z. h w < 0}"
        by auto
      also have "c * (∏w∈.... (℘ z - ℘ w) powi h w) =
                   c * (∏w∈{w∈Z. h w ≥ 0}. (℘ z - ℘ w) powi h w) *
                   (∏w∈{w∈Z. h w < 0}. (℘ z - ℘ w) powi h w)"
        by (subst prod.union_disjoint) (use finite in auto)
      also have "(∏w∈{w∈Z. h w ≥ 0}. (℘ z - ℘ w) powi h w) =
                 (∏w∈{w∈Z. h w ≥ 0}. poly [:-℘ w, 1:] (℘ z) ^ (nat
(h w)))"
        by (intro prod.cong) (auto simp: power_int_def)
      also have "c * ... = poly p (℘ z)"
        by (simp add: p_def poly_prod)
      also have "(∏w∈{w∈Z. h w < 0}. (℘ z - ℘ w) powi h w) =
                 (∏w∈{w∈Z. h w < 0}. inverse (poly [:-℘ w, 1:] (℘ z)
^ (nat (-h w))))"
        by (intro prod.cong) (auto simp: power_int_def field_simps)
      also have "... = inverse (poly q (℘ z))"
        unfolding q_def poly_prod by (subst prod_inversef [symmetric])
auto
      finally show ?case
        by (simp add: field_simps)
    qed
  qed
qed
```

From this, we now show that any elliptic function f can be written in the form $f(z) = g(\wp(z)) + \wp'(z)h(\wp(z))$ where g, h are rational functions.

The proof is fairly simple: We can split $f(z)$ into a sum $f(z) = f_1(z) + f_2(z)$ where f_1 is even and f_2 is odd by defining $f_1(z) = \frac{1}{2}(f(z) + f(-z))$ and $f_2(z) = \frac{1}{2}(f(z) - f(-z))$. We can then further define $f_3(z) = f_2(z)/\wp'(z)$ so that f_3 is also even.

By our previous result, we know that f_1 and f_3 can be written as rational functions of $\wp$, so by combining everything we get the result we want.

This result is Exercise 1.5 in Apostol's book.

```
theorem (in even_elliptic_function) in_terms_of_weierstrass_fun:
  obtains p q r s :: "complex poly" where "q ≠ 0" "s ≠ 0"
     "∀≈z. f z = poly p (℘ z) / poly q (℘ z) + ℘' z * poly r (℘ z) /
poly s (℘ z)"
proof -
  define f1 where "f1 = (λz. (f z + f (-z)) / 2)"
  define f2 where "f2 = (λz. (f z - f (-z)) / 2)"
  define f2' where "f2' = (λz. f2 z / ℘' z)"
```

```
  note [elliptic_function_intros] = elliptic_function_compose_uminus[OF
elliptic_function_axioms]

  interpret f1: elliptic_function ω1 ω2 f1
    unfolding f1_def by (intro elliptic_function_intros)
  interpret f1: even_elliptic_function ω1 ω2 f1
    by standard (auto simp: f1_def)
  obtain p q where pq: "q ≠ 0" "∀≈z. f1 z = poly p (℘ z) / poly q (℘
z)"
    using f1.in_terms_of_weierstrass_fun_even .

  interpret f2': elliptic_function ω1 ω2 f2'
    unfolding f2'_def f2_def by (intro elliptic_function_intros)
  interpret f2': even_elliptic_function ω1 ω2 f2'
    by standard (auto simp: f2'_def f2_def divide_simps)
  obtain r s where rs: "s ≠ 0" "∀≈z. f2' z = poly r (℘ z) / poly s
(℘ z)"
    using f2'.in_terms_of_weierstrass_fun_even .

  have "eventually (λz. ℘' z ≠ 0) (cosparse UNIV)"
    by (simp add: weierstrass_fun_deriv.avoid)
  with pq(2) and rs(2) have "∀≈z. f z = poly p (℘ z) / poly q (℘ z)
+ ℘' z * poly r (℘ z) / poly s (℘ z)"
  proof eventually_elim
    case (elim z)
    have "poly p (℘ z) / poly q (℘ z) + ℘' z * poly r (℘ z) / poly s
(℘ z) =
          f1 z + ℘' z * (poly r (℘ z) / poly s (℘ z))"
      unfolding elim(1) by (simp add: divide_simps)
    also have "poly r (℘ z) / poly s (℘ z) = f2' z"
      using elim(2) by simp
    also have "℘' z * f2' z = f2 z"
      using elim(3) by (simp add: f2'_def)
    also have "f1 z + f2 z = f z"
      by (simp add: f1_def f2_def field_simps)
    finally show ?case ..
  qed
  thus ?thesis
    using that[of q s p r] pq(1) rs(1) by simp
qed

end
```

7 Eisenstein series and the differential equations of ℘

```
theory Eisenstein_Series
```

```
imports
  Weierstrass_Elliptic
  Z_Plane_Q_Disc
  "Polynomial_Factorization.Fundamental_Theorem_Algebra_Factorized"
  "Zeta_Function.Zeta_Function"
  "Polylog.Polylog"
  "Lambert_Series.Lambert_Series"
  "Cotangent_PFD_Formula.Cotangent_PFD_Formula"
  "Algebraic_Numbers.Bivariate_Polynomials"
begin
```

```
lemmas [simp del] = div_mult_self1 div_mult_self2 div_mult_self3 div_mult_self4
```

We define the Eisenstein series G_n, which is the sequence of coefficients of the Laurent series expansion of $\wp$. Both $\wp$ and G_n (for $n \geq 3$) are invariants of the lattice, i.e. they are independent from the choice of generators.

7.1 Definition

For $n \geq 3$, the Eisenstein series G_n is defined simply as the absolutely convergent sum $\sum_{\omega \in \Lambda^*} \omega^{-n}$. However, we want to stay as general as possible here and therefore define it in such a way that the definition also works for $n = 2$, where the sum is only conditionally convergent and much less well-behaved.

Note that all the Eisenstein series with odd $n \geq 3$ vanish due to the symmetry in the sum. As for $n < 3$, we define $G_1 = 0$ in agreement with the the values for other odd n and $G_0 = 1$ since this makes some later theorem statements regarding modular forms more elegant.

```
context complex_lattice
begin

definition eisenstein_series :: "nat ⇒ complex" where
  "eisenstein_series k = (if k = 0 then 1 else if odd k then 0 else
     2 / ω1 ^ k * zeta (of_nat k) + (∑∞n∈-{0}. ∑∞m. 1 / of_ω12_coords
(of_int m, of_int n) ^ k))"

notation eisenstein_series ("G")

lemma eisenstein_series_0 [simp]: "eisenstein_series 0 = 1"
  by (auto simp: eisenstein_series_def)

lemma eisenstein_series_odd_eq_0 [simp]: "odd k ⟹ eisenstein_series
k = 0"
  by (auto simp: eisenstein_series_def elim!: oddE)
```

```
lemma eisenstein_series_Suc_0 [simp]: "eisenstein_series (Suc 0) = 0"
  by (rule eisenstein_series_odd_eq_0) auto

lemma eisenstein_series_norm_summable:
  assumes "n ≥ 3"
  shows   "(λω. 1 / norm ω ^ n) summable_on Λ*"
  using converges_absolutely_iff[of "of_nat n"] assms by (simp add: powr_realpow')

lemma eisenstein_series_summable:
  assumes "n ≥ 3"
  shows   "(λω. 1 / ω ^ n) summable_on Λ*"
  by (rule abs_summable_summable)
     (use eisenstein_series_norm_summable[OF assms] in ‹simp add: norm_divide
norm_power›)

lemma eisenstein_series_has_sum:
  assumes "k ≥ 3"
  shows   "((λω. 1 / ω ^ k) has_sum eisenstein_series k) Λ*"
proof (cases "even k")
  case odd: False
  define S where "S = (∑∞ω∈Λ*. 1 / ω ^ k)"
  have sum1: "((λω. 1 / ω ^ k) has_sum S) Λ*"
    using eisenstein_series_summable[OF assms] has_sum_infsum unfold-
ing S_def by blast
  also have "?this ⟷ ((λω. -(1 / ω ^ k)) has_sum S) Λ*"
    using assms odd
    by (intro has_sum_reindex_bij_witness[of _ uminus uminus]) (auto simp:
uminus_in_lattice0_iff)
  also have "... ⟷ ((λω. 1 / ω ^ k) has_sum (-S)) Λ*"
    by (rule has_sum_uminus)
  finally have "-S = S"
    using sum1 has_sum_unique by blast
  hence "S = 0"
    by simp
  thus ?thesis
    using sum1 odd assms by simp
next
  case even: True
  define f where "f = (λ(m,n). 1 / of_ω12_coords (of_int m, of_int n)
^ k)"

  note k = ‹k ≥ 3› even
  have "((λm. 1 / of_int m ^ k) has_sum (2 * zeta (of_nat k))) (-{0})"
    by (rule has_sum_zeta_symmetric) (use k in auto)
  hence "((λm. (1 / ω1 ^ k) * (1 / of_int m ^ k))
           has_sum ((1 / ω1 ^ k) * (2 * zeta (of_nat k)))) (-{0})"
    by (rule has_sum_cmult_right)
  hence "((λm. 1 / (of_int m * ω1) ^ k) has_sum (2 / ω1 ^ k * zeta (of_nat
k))) (-{0})"
```

```
    by (simp add: field_simps)
  also have "?this ⟷ (f has_sum (2 / ω1 ^ k * zeta (of_nat k))) ((-{0})
× {0})"
    by (rule has_sum_reindex_bij_witness[of _ fst "λm. (m, 0)"]) (auto
simp: f_def)
  finally have sum1: "(f has_sum (2 / ω1 ^ k * zeta (of_nat k))) ((-{0})
× {0})" .

  define S where "S = (∑∞ω∈Λ*. 1 / ω ^ k)"
  define T where "T = (λn. ∑∞m. 1 / of_ω12_coords (of_int m, of_int
n) ^ k)"

  have sum2: "((λω. 1 / ω ^ k) has_sum S) Λ*"
    unfolding S_def using eisenstein_series_summable by (rule has_sum_infsum)
(use k in auto)
  also have "?this ⟷ (f has_sum S) (-{(0,0)})"
    by (subst has_sum_reindex_bij_betw[OF bij_betw_lattice0', symmetric])
       (simp_all add: case_prod_unfold map_prod_def f_def)
  finally have "(f has_sum S) (- {(0, 0)})" .
  hence "(f has_sum (S - 2 / ω1 ^ k * zeta (of_nat k))) (-{(0, 0)} -
((-{0}) × {0}))"
    by (intro has_sum_Diff sum1) auto
  also have "-{(0, 0)} - ((-{0}) × {0}) = (UNIV × (-{0}) :: (int × int)
set)"
    by auto
  finally have "(f has_sum S - 2 / ω1 ^ k * zeta (of_nat k)) (UNIV × (-{0}))"
.
  also have "?this ⟷ ((λ(n,m). f (m,n)) has_sum S - 2 / ω1 ^ k * zeta
(of_nat k)) ((-{0}) × UNIV)"
    by (rule has_sum_reindex_bij_witness[of _ prod.swap prod.swap]) auto
  finally have sum3: "((λ(n,m). f (m,n)) has_sum S - 2 / ω1 ^ k * zeta
(of_nat k)) ((-{0}) × UNIV)" .

  have "(T has_sum S - 2 / ω1 ^ k * zeta (complex_of_nat k)) (-{0})"
    using sum3
  proof (rule has_sum_SigmaD, unfold prod.case)
    fix n :: int assume n: "n ∈ -{0}"
    have "(λm. f (m, n)) summable_on UNIV"
      using summable_on_SigmaD1[OF has_sum_imp_summable[OF sum3, unfolded
f_def], OF n]
      by (simp add: f_def)
    thus "((λm. f (m, n)) has_sum T n) UNIV"
      unfolding T_def f_def prod.case by (rule has_sum_infsum)
  qed
  hence "S = 2 / ω1 ^ k * zeta (of_nat k) + (∑∞m∈-{0}. T m)"
    by (simp add: has_sum_iff)
  also have "... = eisenstein_series k"
    using k by (simp add: eisenstein_series_def T_def)
  finally show ?thesis
```

```
    using sum2 by simp
qed

lemma eisenstein_series_altdef:
  assumes "k ≥ 3"
  shows   "eisenstein_series k = (∑∞ω∈Λ*. 1 / ω ^ k)"
  using eisenstein_series_has_sum[OF assms] by (simp add: has_sum_iff)

lemma eisenstein_fun_aux_0 [simp]:
  assumes "n ≠ 2"
  shows   "eisenstein_fun_aux n 0 = eisenstein_series n"
proof (cases "n ≥ 3")
  case gt3: True
  show ?thesis
  proof (cases "even n")
    case True
    show ?thesis
      using True gt3 by (auto simp: eisenstein_fun_aux_def eisenstein_series_altdef)
  next
    case False
    have "eisenstein_fun_aux n 0 = -(∑∞ω∈Λ*. 1 / ω ^ n)"
      using False gt3
      by (auto simp: eisenstein_fun_aux_def eisenstein_series_def infsum_uminus)
    also have "... = -eisenstein_series n"
      using gt3 by (simp add: eisenstein_series_altdef)
    finally show ?thesis
      using False by (simp add: eisenstein_series_odd_eq_0)
  qed
next
  case False
  hence "n ∈ {0,1,2}"
    by auto
  thus ?thesis using assms
    by (auto simp: eisenstein_fun_aux_def)
qed
```

7.2 The Laurent series expansion of $\wp$ at the origin

```
lemma higher_deriv_weierstrass_fun_aux_0:
  assumes "m > 0"
  shows   "(deriv ^^ m) weierstrass_fun_aux 0 = (- 1) ^ m * fact (Suc
m) * G (m + 2)"
proof -
  define n where "n = m - 1"
  have "(deriv ^^ Suc n) weierstrass_fun_aux 0 = (deriv ^^ n) (λw. -2
* eisenstein_fun_aux 3 w) 0"
    unfolding funpow_Suc_right o_def
  proof (rule higher_deriv_cong_ev)
    have "eventually (λz. z ∈ -Λ*) (nhds 0)"
```

```
    using closed_lattice0 by (intro eventually_nhds_in_open) auto
  thus "∀F x in nhds 0. deriv weierstrass_fun_aux x = -2 * eisenstein_fun_aux
3 x"
    by eventually_elim (simp add: deriv_weierstrass_fun_aux)
 qed auto
 also have "... = -2 * (deriv ^^ n) (eisenstein_fun_aux 3) 0"
  using closed_lattice0
  by (intro higher_deriv_cmult[where A = "-Λ*"]) (auto intro!: holomorphic_intros)
 also have "... = (- 1) ^ Suc n * fact (n + 2) * G (n + 3)"
  by (subst higher_deriv_eisenstein_fun_aux)
     (auto simp: algebra_simps pochhammer_rec pochhammer_fact)
 finally show ?thesis
  using assms by (simp add: n_def)
qed
```

We now show that the Laurent series expansion of $\wp(z)$ at $z = 0$ has the form

$$z^{-2} + \sum_{n\geq 1}(n+1)G_{n+2}z^n \ .$$

We choose a different approach to prove this than Apostol: Apostol converts the sum in question into a double sum and then interchanges the order of summation, claiming the double sum to be absolutely convergent. Since we were unable to see why that sum should be absolutely convergent, we were unable to replicate his argument. In any case, arguing about absolute convergence of double sums is always messy.

Our approach instead simply uses the fact that `weierstrass_fun_aux` (the Weierstrass function with its double pole removed) is analytic at 0 and thus has a power series expansion that is valid within any ball around 0 that does not contain any lattice points.

The coefficients of this power series expansion can be determined simply by taking the `n`-th derivative of `weierstrass_fun_aux` at 0, which is easy to do.

Note that this series converges absolutely in this domain, since it is a power series, but we do not show this here.

```
definition fps_weierstrass :: "complex fps"
 where "fps_weierstrass = Abs_fps (λn. if n = 0 then 0 else of_nat (Suc
n) * G (n + 2))"

lemma weierstrass_fun_aux_fps_expansion: "weierstrass_fun_aux has_fps_expansion
fps_weierstrass"
proof -
 define c where "c = (λn. if n = 0 then 0 else of_nat (Suc n) * G (n
+ 2))"
 have "(λn. (deriv ^^ n) weierstrass_fun_aux 0 / fact n) = c"
        (is "?lhs = ?rhs")
 proof
  fix n :: nat
```

```
    show "?lhs n = ?rhs n" unfolding c_def
      by (cases "even n") (auto simp: higher_deriv_weierstrass_fun_aux_0
eisenstein_series_odd_eq_0)
  qed
  hence "fps_weierstrass = fps_expansion weierstrass_fun_aux 0"
    by (intro fps_ext) (auto simp: fps_expansion_def fps_weierstrass_def
c_def fun_eq_iff)
  also have "weierstrass_fun_aux has_fps_expansion ..."
    using closed_lattice0
    by (intro has_fps_expansion_fps_expansion[of "-Λ*"] holomorphic_intros)
auto
  finally show ?thesis .
qed


definition fls_weierstrass :: "complex fls"
  where "fls_weierstrass = fls_X_intpow (-2) + fps_to_fls fps_weierstrass"

lemma fls_subdegree_weierstrass: "fls_subdegree fls_weierstrass = -2"
  by (intro fls_subdegree_eqI) (auto simp: fls_weierstrass_def)

lemma fls_weierstrass_nz [simp]: "fls_weierstrass ≠ 0"
  using fls_subdegree_weierstrass by auto
```

The following corresponds to Theorem 1.11 in Apostol's book.

```
theorem fls_weierstrass_laurent_expansion [laurent_expansion_intros]:
  "℘ has_laurent_expansion fls_weierstrass"
proof -
  have "(λz. z powi (-2) + weierstrass_fun_aux z) has_laurent_expansion
fls_weierstrass"
    unfolding fls_weierstrass_def
    by (intro laurent_expansion_intros has_laurent_expansion_fps[OF weierstrass_fun_aux_fps
  also have "?this ⟷ ?thesis"
  proof (intro has_laurent_expansion_cong refl)
    have "eventually (λz. z ∈ -Λ* - {0}) (at 0)"
      using closed_lattice0 by (intro eventually_at_in_open) auto
    thus "∀F x in at 0. x powi - 2 + weierstrass_fun_aux x = ℘ x"
      by eventually_elim
         (auto simp: weierstrass_fun_def power_int_minus field_simps lattice_lattice0)
  qed
  finally show ?thesis .
qed

corollary fls_weierstrass_deriv_laurent_expansion [laurent_expansion_intros]:
  "℘' has_laurent_expansion fls_deriv fls_weierstrass"
  by (rule has_laurent_expansion_deriv'[where A = "-Λ*", OF fls_weierstrass_laurent_expans
     (use closed_lattice0 in ‹auto intro!: derivative_eq_intros simp:
lattice_lattice0›)
```

```
lemma fls_nth_weierstrass:
  "fls_nth fls_weierstrass n =
     (if n = -2 then 1 else if n > 0 then of_int (n + 1) * G (nat n +
2) else 0)"
  unfolding fls_weierstrass_def fps_weierstrass_def by (auto simp: not_less)
```

7.3 Differential equations for $\wp$

Using our results on elliptic functions, we can prove the important result that $\wp$ satisfies the ordinary differential equation

$$\wp'^2 = 4\wp^3 - 60G_4\wp - 140G_6 \ .$$

The proof works by simply subtracting the two sides and then looking at the Laurent series expansion, noting that the poles all cancel out. This means that what remains is an elliptic functions without poles and therefore constant.

The constant can then easily be determined, since it is the 0-th coefficient of said Laurent series.

This is Theorem 1.12 in Apostol's book.

```
theorem weierstrass_fun_ODE1:
  assumes "z ∉ Λ"
  shows   "℘' z ^ 2 = 4 * ℘ z ^ 3 - 60 * G 4 * ℘ z - 140 * G 6"
proof -
  note [simp] = fls_subdegree_deriv fls_subdegree_weierstrass
  define f where "f = (λz. ℘' z ^ 2 - 4 * ℘ z ^ 3 + 60 * G 4 * ℘ z)"
  interpret f: elliptic_function ω1 ω2 f
    unfolding f_def by (intro elliptic_function_intros)
  let ?P = fls_weierstrass
  define F :: "complex fls" where "F = fls_deriv ?P ^ 2 - 4 * ?P ^ 3
+ fls_const (60 * G 4) * ?P"


  define g where "g = (λz. if z ∈ Λ then -140 * G 6 else f z)"


  have unwind: "{m..n::int} = insert m {m+1..n}" if "m ≤ n" for m n
    using that by auto
  define fls_terms :: "complex fls ⇒ complex list"
    where "fls_terms = (λF. map (λk. fls_nth F k) [-6,-5,-4,-3,-2,-1,0])"
  have coeffs: "map (λk. fls_nth F k) [-6,-5,-4,-3,-2,-1,0] = [0, 0, 0,
0, 0, 0, - (140 * G 6)]"
    by (simp add: power2_eq_square power3_eq_cube unwind fls_terms_def
fls_times_nth(1)
                  fls_nth_weierstrass F_def eisenstein_series_odd_eq_0
flip: One_nat_def)

  have F: "f has_laurent_expansion F"
```

```
    unfolding f_def F_def by (intro laurent_expansion_intros)

  have "fls_subdegree F ≥ 0"
  proof (cases "F = 0")
    case False
    thus ?thesis
    proof (rule fls_subdegree_geI)
      show "fls_nth F k = 0" if "k < 0" for k
      proof (cases "k < -6")
        case True
        thus ?thesis
          by (simp add: power2_eq_square power3_eq_cube fls_times_nth(1)
F_def)
      next
        case False
        with ‹k < 0› have "k ∈ {-6, -5, -4, -3, -2, -1}"
          by auto
        thus ?thesis using ‹k < 0›
          using coeffs by auto
      qed
    qed
  qed auto

  interpret g: complex_lattice_periodic ω1 ω2 g
    by standard (auto simp: g_def f.lattice_cong rel_def)

  have "f holomorphic_on -Λ* - {0}"
    unfolding f_def by (intro holomorphic_intros) (auto simp: lattice_lattice0)
  moreover have "open (-Λ*)"
    using closed_lattice0 by auto
  moreover have "f −0→ fls_nth F 0"
    using F ‹fls_subdegree F ≥ 0› by (meson has_laurent_expansion_imp_tendsto_0)
  hence "f −0→ -140 * G 6"
    using coeffs by simp
  ultimately have "(λz. if z = 0 then -140 * G 6 else f z) holomorphic_on
-Λ*"
    unfolding g_def by (rule removable_singularity)
  hence "(λz. if z = 0 then -140 * G 6 else f z) analytic_on {0}"
    using closed_lattice0 unfolding analytic_on_holomorphic by blast
  also have "?this ⟷ g analytic_on {0}"
    using closed_lattice0
    by (intro analytic_at_cong refl eventually_mono[OF eventually_nhds_in_open[of
"-Λ*"]])
       (auto simp: g_def f_def lattice_lattice0)
  finally have "g analytic_on {0}" .

  have "g analytic_on (-Λ ∪ (⋃ω∈Λ. {ω}))"
    unfolding analytic_on_Un analytic_on_UN
  proof safe
```

```
    fix ω :: complex
    assume ω: "ω ∈ Λ"
    have "g ∘ (λz. z - ω) analytic_on {ω}"
      using ‹g analytic_on {0}›
      by (intro analytic_on_compose) (auto intro!: analytic_intros)
    also have "g ∘ (λz. z - ω) = g"
      by (auto simp: fun_eq_iff rel_def uminus_in_lattice_iff ω intro!:
g.lattice_cong)
    finally show "g analytic_on {ω}" .
  next
    have "f holomorphic_on (-Λ)"
      unfolding f_def by (auto intro!: holomorphic_intros)
    also have "?this ⟷ g holomorphic_on (-Λ)"
      by (intro holomorphic_cong refl) (auto simp: g_def)
    finally show "g analytic_on (-Λ)"
      using closed_lattice by (subst analytic_on_open) auto
  qed
  also have "... = UNIV"
    by blast
  finally have g_ana: "g analytic_on UNIV" .

  interpret g: nicely_elliptic_function ω1 ω2 g
    by standard (auto intro!: analytic_on_imp_nicely_meromorphic_on g_ana)
  have "elliptic_order g = 0"
  proof (subst g.elliptic_order_eq_0_iff_no_poles, rule allI)
    show "¬is_pole g z" for z
      by (rule analytic_at_imp_no_pole) (auto intro: analytic_on_subset[OF
g_ana])
  qed
  hence "g constant_on UNIV"
    by (simp add: g.elliptic_order_eq_0_iff)

  then obtain c where c: "g z = c" for z
    unfolding constant_on_def by blast
  from c[of 0] have "c = -140 * G 6"
    by (simp add: g_def)
  with c[of z] and assms show ?thesis
    by (simp add: g_def f_def algebra_simps)
qed
```

The above ODE of the meromorphic function $\wp$ can now easily be lifted to a formal ODE on the corresponding Laurent series.

```
lemma fls_weierstrass_ODE1:
  defines "P ≡ fls_weierstrass"
  shows   "fls_deriv P ^ 2 = 4 * P ^ 3 - fls_const (60 * G 4) * P - fls_const
(140 * G 6)"
  (is "?lhs = ?rhs")
proof -
  have ev: "eventually (λz. z ∈ -Λ* - {0}) (at 0)"
```

```
    using closed_lattice0 by (intro eventually_at_in_open) auto
  have "(λz. ℘' z ^ 2) has_laurent_expansion ?lhs"
    unfolding P_def by (intro laurent_expansion_intros)
  also have "?this ⟷ (λz. 4 * ℘ z ^ 3 - 60 * G 4 * ℘ z - 140 * G 6)
has_laurent_expansion ?lhs"
    by (intro has_laurent_expansion_cong refl eventually_mono[OF ev] weierstrass_fun_ODE1)
       (auto simp: lattice_lattice0)
  finally have ... .
  moreover have "(λz. 4 * ℘ z ^ 3 - 60 * G 4 * ℘ z - 140 * G 6) has_laurent_expansion
?rhs"
    unfolding P_def by (intro laurent_expansion_intros)
  ultimately show ?thesis
    using has_laurent_expansion_unique by blast
qed

lemma fls_weierstrass_ODE2:
  defines "P ≡ fls_weierstrass"
  shows   "fls_deriv (fls_deriv P) = 6 * P ^ 2 - fls_const (30 * G 4)"
proof -
  define d where "d = fls_deriv P"
  have "fls_subdegree d = -3"
    using fls_subdegree_weierstrass unfolding d_def
    by (subst fls_subdegree_deriv) (auto simp: P_def)
  hence nz: "d ≠ 0" by auto

  have "2 * d * fls_deriv d = 2 * d * (6 * P^2  - 30 * fls_const (G 4))"
    using arg_cong[OF fls_weierstrass_ODE1, of fls_deriv]
    unfolding P_def [symmetric] fls_const_mult_const [symmetric] d_def
    by (simp add: fls_deriv_power algebra_simps)
  then have "fls_deriv d = 6 * P^2 - 30 * fls_const (G 4)"
    using nz by simp
  then show ?thesis unfolding d_def
    by (metis fls_const_mult_const fls_const_numeral)
qed

theorem weierstrass_fun_ODE2:
  assumes "z ∉ Λ"
  shows   "deriv ℘' z = 6 * ℘ z ^ 2 - 30 * G 4"
proof -
  define P where "P = fls_weierstrass"
  have "(λz. deriv ℘' z - 6 * ℘ z ^ 2 + 30 * G 4) has_laurent_expansion
        (fls_deriv (fls_deriv P) - 6 * P ^ 2 +fls_const (30 * G 4))"
    unfolding P_def by (intro laurent_expansion_intros)
  also have "... = 0"
    by (simp add: fls_weierstrass_ODE2 P_def)
  finally have "deriv ℘' z - 6 * (℘ z)^2 + 30 * G 4 = 0"
  proof (rule has_laurent_expansion_0_analytic_continuation)
    show "(λz. deriv ℘' z - 6 * (℘ z)^2 + 30 * G 4) holomorphic_on ((UNIV
- (Λ-{0})) - {0})"
```

```
    by (auto intro!: holomorphic_intros intro: closed_subset_lattice)
  show "open (UNIV - (Λ-{0}))"
    by (intro open_Diff closed_subset_lattice) auto
  show "connected (UNIV - (Λ-{0}))"
    by (intro sparse_imp_connected sparse_in_subset2[OF lattice_sparse])
auto
 qed (use assms in auto)
 thus ?thesis
   by (simp add: algebra_simps)
qed

lemma has_field_derivative_weierstrass_fun_deriv [derivative_intros]:
 assumes "(f has_field_derivative f') (at z within A)" "f z ∉ Λ"
 shows   "((λz. ℘' (f z)) has_field_derivative ((6 * ℘ (f z) ^ 2 - 30
* G 4) * f')) (at z within A)"
proof (rule DERIV_chain' [OF assms(1)])
 have "(℘' has_field_derivative (deriv ℘' (f z))) (at (f z))"
   by (rule analytic_derivI) (use assms(2) in ‹auto intro!: analytic_intros›)
 thus "(℘' has_field_derivative 6 * (℘ (f z))^2 - 30 * G 4) (at (f z))"
   using weierstrass_fun_ODE2[OF assms(2)] by simp
qed
```

7.4 Lattice invariants and a recurrence for the Eisenstein series

We will see that G_n can always be expressed in terms of G_4 and G_6. These values, up to a constant factor, are referred to as g_2 and g_3.

```
definition invariant_g2:: "complex" ("g₂") where
 "g₂ ≡ 60 * eisenstein_series 4"

definition invariant_g3:: "complex" ("g₃") where
 "g₃ ≡ 140 * eisenstein_series 6"

lemma weierstrass_fun_ODE1':
 assumes "z ∉ Λ"
 shows   "℘' z ^ 2 = 4 * ℘ z ^ 3 - g₂ * ℘ z - g₃"
 using weierstrass_fun_ODE1[OF assms] by (simp add: invariant_g2_def
invariant_g3_def)
```

This is the ODE obtained by differentiating the first ODE.

```
theorem weierstrass_fun_ODE2':
 assumes "z ∉ Λ"
 shows   "deriv ℘' z = 6 * ℘ z ^ 2 - g₂ / 2"
proof -
 define P where "P = fls_weierstrass"
 have "(λz. deriv ℘' z - 6 * ℘ z ^ 2 + g₂ / 2) has_laurent_expansion
       fls_deriv (fls_deriv P) - 6 * P ^ 2 + fls_const (g₂ / 2)" (is "?f
has_laurent_expansion _")
```

```
    unfolding P_def by (intro laurent_expansion_intros)
  also have "... = 0"
    by (simp add: fls_weierstrass_ODE2 P_def invariant_g2_def)
  finally have "?f has_laurent_expansion 0" .
  hence "?f z = 0"
  proof (rule has_laurent_expansion_0_analytic_continuation)
    show "?f holomorphic_on UNIV - Λ* - {0}"
      using closed_lattice0
      by (intro holomorphic_intros) (auto simp: lattice_lattice0)
    show "connected (UNIV - Λ*)"
      by (intro connected_open_diff_countable countable_subset[OF _ countable_lattice])
         (auto simp: lattice_lattice0)
  qed (use assms closed_lattice0 in ‹auto simp: lattice_lattice0›)
  thus ?thesis by (simp add: algebra_simps)
qed

lemma half_period_weierstrass_fun_is_root:
  assumes "ω ∈ Λ" "ω / 2 ∉ Λ"
  defines "z ≡ ℘ (ω / 2)"
  shows   "4 * z ^ 3 - g2 * z - g3 = 0"
proof -
  have "℘' (ω / 2) = 0"
    using weierstrass_fun_deriv.lattice_cong[of "-ω/2" "ω/2"] ‹ω ∈ Λ›
    by (auto simp: rel_def uminus_in_lattice_iff)
  hence "0 = ℘' (ω / 2) ^ 2"
    by simp
  also have "... = 4 * ℘ (ω / 2) ^ 3 - g2 * ℘ (ω / 2) - g3"
    using assms by (subst weierstrass_fun_ODE1') auto
  finally show ?thesis
    by (simp add: z_def)
qed
```

The discriminant of the depressed cubic polynomial $p(x) = cX^3 + aX + b$ is $-4a^3 - 27cb^2$. This is useful since it gives us an algebraic condition for whether p has distinct roots.

```
lemma (in -) depressed_cubic_discriminant:
  fixes a b :: "'a :: idom"
  assumes "[:b, a, 0, c:] = Polynomial.smult c ([:-x1, 1:] * [:-x2, 1:]
* [:-x3, 1:])"
  shows   "c ^ 3 * (x1 - x2)^2 * (x1 - x3)^2 * (x2 - x3)^2 = -4 * a ^ 3 -
27 * c * b ^ 2"
  using assms by (simp add: algebra_simps) Groebner_Basis.algebra
```

The numbers e_1, e_2, e_3 are all distinct and hence the discriminant $\Delta = g_2^3 - 27g_3^2$ does not vanish. This is the first part of Apostol's Theorem 1.14.

```
theorem distinct_e123: "distinct [e1, e2, e3]"
proof -
  define X where "X = {ω1 / 2, ω2 / 2, (ω1 + ω2) / 2}"
  have empty: "X ∩ Λ = {}"
```

```
    by (auto simp: X_def)
  have roots: "∀x∈X. ℘' x = 0"
    using weierstass_fun_deriv_half_root_eq_0 unfolding X_def by blast
  have X_subset: "X ⊆ period_parallelogram 0"
    unfolding period_parallelogram_altdef of_ω12_coords_image_eq
    by (auto simp: X_def ω12_coords.add)

  have *: "℘ w1 ≠ ℘ w2" if w12: "{w1, w2} ⊆ X" and neq: "w1 ≠ w2" for
w1 w2
  proof
    assume eq: "℘ w1 = ℘ w2"
    have not_in_lattice [simp]: "w1 ∉ Λ" "w2 ∉ Λ"
      using empty that(1) by blast+

    define f where "f = (λz. ℘ z - ℘ w1)"

    have deriv_eq: "deriv f w = ℘' w" if "w ∉ Λ" for w
      unfolding f_def using that
      by (auto intro!: derivative_eq_intros DERIV_imp_deriv)
    have "deriv f w1 = 0" "deriv f w2 = 0"
      using w12 roots not_in_lattice deriv_eq by (metis insert_subset)+
    moreover have [simp]: "f w1 = 0" "f w2 = 0"
      using eq by (simp_all add: f_def)

    have "f has_laurent_expansion fls_weierstrass - fls_const (℘ w1)"
      unfolding f_def by (intro laurent_expansion_intros)
    moreover have "fls_subdegree (fls_weierstrass - fls_const (℘ w1))
= -2"
      by (simp add: fls_subdegree_weierstrass fls_subdegree_diff_eq1)
    ultimately have [simp]: "is_pole f 0" "zorder f 0 = -2"
      by (auto dest: has_laurent_expansion_imp_is_pole_0 has_laurent_expansion_zorder_0)
    have [simp]: "¬f constant_on UNIV"
      using pole_imp_not_constant[OF ‹is_pole f 0›, of UNIV UNIV] by
auto

    have "¬(∀x∈-Λ. f x = 0)"
    proof
      assume *: "∀x∈-Λ. f x = 0"
      have "eventually (λx. x ∈ -Λ* - {0}) (at 0)"
        using closed_lattice0 by (intro eventually_at_in_open) auto
      hence "eventually (λx. f x = 0) (at 0)"
        by eventually_elim (use * in ‹auto simp: lattice_lattice0›)
      hence "f —0→ 0"
        by (simp add: tendsto_eventually)
      with ‹is_pole f 0› show False
        unfolding is_pole_def using not_tendsto_and_filterlim_at_infinity[of
"at 0" f 0] by auto
    qed
    then obtain z0 where z0: "z0 ∉ Λ" "f z0 ≠ 0"
```

```
      by auto
    then have nconst:"¬ f constant_on (UNIV-Λ)"
      by (metis (mono_tags, lifting) Diff_iff UNIV_I <f w1 = 0>
          constant_on_def not_in_lattice(1))

    have zorder_ge: "zorder f w ≥ int 2" if "w ∈ {w1, w2}" for w
    proof (rule zorder_geI)
      show "f analytic_on {w}" "f holomorphic_on UNIV - Λ"
        unfolding f_def using that w12 empty
        by (auto intro!: analytic_intros holomorphic_intros)
      show "connected (UNIV - Λ)"
        using countable_lattice by (intro connected_open_diff_countable)
auto
      show "z0 ∈ UNIV - Λ" "f z0 ≠ 0"
        using z0 by auto
    next
      fix k :: nat
      assume "k < 2"
      hence "k = 0 ∨ k = 1"
        by auto
      thus "(deriv ^^ k) f w = 0"
        using that w12 empty deriv_eq[of w] roots by auto
    qed (use closed_lattice that w12 empty in auto)

    have not_pole: "¬is_pole f w" if "w ∈ {w1, w2}" for w
    proof -
      have "f holomorphic_on -Λ"
        unfolding f_def by (intro holomorphic_intros) auto
      moreover have "open (-Λ)" "w ∈ -Λ"
        using that empty w12 closed_lattice by auto
      ultimately show ?thesis
        using not_is_pole_holomorphic by blast
    qed

    have no_further_poles: "¬is_pole f z" if "z ∈ period_parallelogram
0 - {0}" for z
    proof -
      have "f holomorphic_on -Λ"
        unfolding f_def by (intro holomorphic_intros) auto
      moreover have "z ∉ Λ"
      proof
        assume "z ∈ Λ"
        then obtain m n where z_eq: "z = of_ω12_coords (of_int m, of_int
n)"
          by (elim latticeE)
        from that have "m = 0" "n = 0"
          unfolding period_parallelogram_altdef of_ω12_coords_image_eq
          by (auto simp: z_eq)
        with that show False
```

```
        by (auto simp: z_eq)
    qed
    ultimately show "¬is_pole f z"
      using closed_lattice not_is_pole_holomorphic[of "-Λ" z f] by
auto
  qed

  interpret f: elliptic_function ω1 ω2 f
    unfolding f_def by (intro elliptic_function_intros)

  have [intro]: "isolated_zero f w" if "w ∈ {w1, w2}" for w
  proof (subst f.isolated_zero_analytic_iff)
    show "f analytic_on {w}"
      unfolding f_def by (intro analytic_intros) (use that in auto)
    show "f w = 0"
      using that by auto
    have "elliptic_order f ≠ 0"
      using ‹is_pole f 0› f.elliptic_order_eq_0_iff_no_poles by blast
    thus "¬(∀≈z∈UNIV. f z = 0)"
      unfolding f.elliptic_order_eq_0_iff_const_cosparse by metis
  qed

  have "4 = (∑z∈{w1, w2}. 2 :: nat)"
    using neq by simp
  also have "... ≤ (∑z∈{w1, w2}. nat (zorder f z))"
    using zorder_ge[of w1] zorder_ge[of w2] by (intro sum_mono) auto
  also have "{w1, w2} ⊆ {z ∈ period_parallelogram 0. isolated_zero
f z}"
    using w12 X_subset by auto
  hence "(∑z∈{w1, w2}. nat (zorder f z)) ≤ (∑z∈.... nat (zorder
f z))"
    using f.finite_zeros_in_parallelogram[of 0] by (intro sum_mono2)
auto
  also have "... = elliptic_order f"
    by (rule f.zeros_eq_elliptic_order)
  also have "elliptic_order f = (∑z | z ∈ period_parallelogram 0 ∧
is_pole f z. nat (-zorder f z))"
    by (rule f.poles_eq_elliptic_order [symmetric])
  also have "... ≤ (∑z∈{0}. nat (-zorder f z))"
  proof (rule sum_mono2)
    have "¬is_pole f w"  if "w ∈ period_parallelogram 0 - {0}" for w
      using no_further_poles[OF that] is_pole_cong by blast
    then show "{z ∈ period_parallelogram 0. is_pole f z} ⊆ {0}"
      unfolding period_parallelogram_def by auto
  qed simp_all
  also have "... = 2" by simp
  finally show False by simp
qed
show ?thesis
```

```
    by (simp add: number_e1_def number_e2_def number_e3_def; intro conjI
*) (auto simp: X_def)
qed
```

The above implies that the polynomial

$$4(X - e_1)(X - e_2)(X - e_3) = 4X^3 - g_2X - g_3$$

has three distinct roots and therefore its discriminant

$$\Delta = g_2^3 - 27g_3^2$$

is non-zero. This is the second part of Apostol's Theorem 1.14.
From now on, we will refer to Δ as the discriminant of our lattice Λ. We also introduce the related invariant $j = \frac{g_2^3}{\Delta}$.

```
definition discr :: complex where
  "discr = g2 ^ 3 - 27 * g3 ^ 2"

definition invariant_j :: complex where
  "invariant_j = g2 ^ 3 / discr"

theorem
  fixes z:: "complex"
  defines "P ≡ [:-g3, -g2, 0, 4:]"
  shows discr_nonzero_aux1: "P = 4 * [:-e1, 1:] * [:-e2, 1:] * [:-e3,
1:]"
  and   discr_nonzero_aux2: "4 * (℘ z)^3 - g2 * (℘ z) - g3 = 4 * (℘ z
- e1) * (℘ z - e2) * (℘ z - e3)"
  and   discr_nonzero: "discr ≠ 0"
proof -
  have zeroI: "poly P (℘ (ω / 2)) = 0" if "ω ∈ Λ" "ω / 2 ∉ Λ" for ω
    using half_period_weierstrass_fun_is_root[OF that]
    by (simp add: P_def power3_eq_cube algebra_simps)

  obtain xs where "length xs = 3" and "Polynomial.smult 4 (∏x←xs.
[:-x, 1:]) = P"
    using fundamental_theorem_algebra_factorized[of P]
    by (auto simp: P_def numeral_3_eq_3)
  hence P_eq: "P = Polynomial.smult 4 (∏x←xs. [:-x, 1:])"
    by simp
  from ‹length xs = 3› obtain x1 x2 x3 where xs: "xs = [x1, x2, x3]"
    by (auto simp: numeral_3_eq_3 length_Suc_conv)
  have poly_P_eq: "poly P x = 4 * (x - x1) * (x - x2) * (x - x3)" for
x
    by (simp add: algebra_simps P_eq xs)

  have "∀x∈{e1, e2, e3}. poly P x = 0"
    by (auto simp: number_e1_def number_e2_def number_e3_def intro!: zeroI
intro: lattice_intros)
```

```
  hence set_xs: "set xs = {e1, e2, e3}" and distinct: "distinct xs"
    using distinct_e123 by (auto simp: poly_P_eq xs)
  have "P = Polynomial.smult 4 (∏x∈set xs. [:-x, 1:])"
    using distinct P_eq by (subst prod.distinct_set_conv_list) auto
  thus P_eq': "P = 4 * [:-e1, 1:] * [:-e2, 1:] * [:-e3, 1:]"
    unfolding set_xs using distinct_e123  by (simp add: xs numeral_poly
algebra_simps)

  from arg_cong[OF this, of "λP. poly P (℘ z)"]
    show "4 * (℘ z)^3 - g2 * (℘ z) - g3 = 4 * (℘ z - e1) * (℘ z - e2)
* (℘ z - e3)"
    by (simp add: P_def numeral_poly algebra_simps power3_eq_cube)

  have "-4 * (-g2) ^ 3 - 27 * 4 * (-g3) ^ 2 = 4 ^ 3 * (e1 - e2)^2 * (e1
- e3)^2 * (e2 - e3)^2"
    by (rule sym, rule depressed_cubic_discriminant, fold P_def) (simp
add: P_eq' numeral_poly)
  also have "... ≠ 0"
    using distinct_e123 by simp
  finally show "discr ≠ 0"
    by (simp add: discr_def)
qed

end


context std_complex_lattice
begin

lemma eisenstein_series_norm_summable':
  "k ≥ 3 ⟹ (λ(m,n). norm (1 / (of_int m + of_int n * τ) ^ k)) summable_on
(-{(0,0)})"
  using eisenstein_series_norm_summable[of k]
        summable_on_reindex_bij_betw[OF bij_betw_lattice0', where f =
"λω. norm (1 / ω ^ k)"]
  by (auto simp: eisenstein_series_def map_prod_def of_ω12_coords_def
                 case_prod_unfold norm_divide norm_power)

lemma eisenstein_series_2_altdef:
  "eisenstein_series 2 = 2 * zeta 2 + (∑∞n∈-{0}. ∑∞m. 1 / (of_int
m + of_int n * τ) ^ 2)"
  by (simp add: eisenstein_series_def of_ω12_coords_def)

lemma eisenstein_series_altdef':
  "k ≥ 3 ⟹ eisenstein_series k = (∑∞(m,n)∈-{(0,0)}. 1 / (of_int m
+ of_int n * τ) ^ k)"
  using infsum_reindex_bij_betw[OF bij_betw_lattice0', where f = "λω.
1 / ω ^ k"]
  by (auto simp: eisenstein_series_altdef map_prod_def of_ω12_coords_def
```

```
case_prod_unfold)
```

end

7.5 Fourier expansion

In this section we derive the Fourier expansion of the Eisenstein series, following Apostol's Theorem 1.18, but with some alterations. For example, we directly generalise the result in the spirit of Apostol's Exercise 1.11, and we make use of the existing formalisation of Lambert series.

We first define an auxiliary function

$$f_n(z) = \sum_{m\in\mathbb{Z}} (z+m)^{-n} = \frac{1}{(n-1)!}\psi^{(n-1)}(1+z) + \psi^{(n-1)}(1-z) + \frac{1}{z^n}$$

where $\psi^{(n)}$ denotes the Polygamma function. This is well-defined for $n \geq 2$ and $z \in \mathbb{C} \setminus \mathbb{Z}$.

We then prove the Fourier expansion

$$f_{n+1}(z) = \frac{(2i\pi)^{n+1}}{n!}\mathrm{Li}_{-n}(q)$$

where $q = e^{2i\pi z}$ and Li_{-n} denotes the Polylogarithm function.

```
definition eisenstein_fourier_aux :: "nat ⇒ complex ⇒ complex" where
  "eisenstein_fourier_aux n z =
     (Polygamma (n-1) (1 + z) + Polygamma (n-1) (1 - z)) / fact (n - 1)
+ 1 / z ^ n"

lemma abs_summable_one_over_const_plus_nat_power:
  assumes "n ≥ 2"
  shows   "summable (λk. norm (1 / (z + of_nat k :: complex) ^ n))"
proof (rule summable_comparison_test_ev)
  have "eventually (λk. real k > norm z) at_top"
    by real_asymp
  thus "eventually (λk. norm (norm (1 / (z + of_nat k) ^ n)) ≤ 1 / (real
k - norm z) ^ n) at_top"
  proof eventually_elim
    case (elim k)
    have "real k - norm z ≤ norm (z + of_nat k)"
      by (metis add.commute norm_diff_ineq norm_of_nat)
    hence "1 / norm (z + of_nat k) ^ n ≤ 1 / norm (real k - norm z) ^
n"
      using elim
      by (intro power_mono divide_left_mono Rings.mult_pos_pos zero_less_power)
auto
    thus ?case using elim
      by (simp add: norm_divide norm_power)
  qed
```

```
next
  show "summable (λk. 1 / (real k - norm z) ^ n)"
  proof (rule summable_comparison_test_bigo)
    show "summable (λk. norm (1 / real k ^ n))"
      using inverse_power_summable[of n] assms
      by (simp add: norm_power norm_divide field_simps)
  next
    show "(λk. 1 / (real k - norm z) ^ n) ∈ O(λk. 1 / real k ^ n)"
      by real_asymp
  qed
qed

lemma abs_summable_one_over_const_minus_nat_power:
  assumes "n ≥ 2"
  shows   "summable (λk. norm (1 / (z - of_nat k :: complex) ^ n))"
proof -
  have "summable (λk. norm (1 / ((-z) + of_nat k :: complex) ^ n))"
    using assms by (rule abs_summable_one_over_const_plus_nat_power)
  also have "(λk. norm (1 / ((-z) + of_nat k :: complex) ^ n)) =
             (λk. norm (1 / (z - of_nat k :: complex) ^ n))"
    by (simp add: norm_divide norm_power norm_minus_commute)
  finally show ?thesis .
qed

lemma has_sum_eisenstein_fourier_aux:
  assumes "n ≥ 2" and "even n" and "Im z > 0"
  shows   "((λm. 1 / (z + of_int m) ^ n) has_sum eisenstein_fourier_aux
n z) UNIV"
proof -
  define f where "f = (λk. 1 / (z + of_int k) ^ n)"

  from assms have "1 + z ≠ 0"
    by (subst add_eq_0_iff) auto
  have "(λk. 1 / (((1 + z) + of_nat k) ^ n)) sums (Polygamma (n-1) (1+z)
/ fact (n-1))"
    using assms Polygamma_LIMSEQ[of "1 + z" "n - 1"] ‹1 + z ≠ 0› by (simp
add: field_simps)
  moreover have "summable (λk. cmod (1 / (z + (Suc k)) ^ n))"
    by (subst summable_Suc_iff) (rule abs_summable_one_over_const_plus_nat_power,
fact)
  ultimately have "((λk. 1 / (z + of_nat (Suc k)) ^ n) has_sum (Polygamma
(n-1) (1+z) / fact (n-1))) UNIV"
    by (intro norm_summable_imp_has_sum) (simp_all add: algebra_simps)
  also have "?this ⟷ (f has_sum (Polygamma (n-1) (1+z) / fact (n-1)))
{1..}" unfolding f_def
    by (rule has_sum_reindex_bij_witness[of _ "λk. nat (k - 1)" "λk.
of_int (Suc k)"]) auto
  finally have sum1: "(f has_sum (Polygamma (n-1) (1+z) / fact (n-1)))
{1..}" .
```

```
  have "1 - z ≠ 0"
    using assms by auto
  have "(λk. 1 / (((1 - z) + of_nat k) ^ n)) sums (Polygamma (n-1) (1-z)
/ fact (n-1))"
    using assms Polygamma_LIMSEQ[of "1 - z" "n - 1"] <1 - z ≠ 0>
    by (simp add: field_simps)
  also have "(λk. ((1 - z) + of_nat k) ^ n) = (λk. (z - of_nat (Suc k))
^ n)"
    using assms by (subst even_power_diff_commute) (auto simp: algebra_simps)
  finally have "(λk. 1 / (z - of_nat (Suc k)) ^ n) sums (Polygamma (n-1)
(1-z) / fact (n-1))" .
  moreover have "summable (λk. cmod (1 / (z - (Suc k)) ^ n))"
    by (subst summable_Suc_iff) (rule abs_summable_one_over_const_minus_nat_power,
fact)
  ultimately have "((λk. 1 / (z - of_nat (Suc k)) ^ n) has_sum (Polygamma
(n-1) (1-z) / fact (n-1))) UNIV"
    by (intro norm_summable_imp_has_sum)
  also have "?this ⟷ (f has_sum (Polygamma (n-1) (1-z) / fact (n-1)))
{..-1}" unfolding f_def
    by (rule has_sum_reindex_bij_witness[of _ "λk. nat (-k-1)" "λk. -of_int
(Suc k)"])
       (auto simp: algebra_simps)
  finally have sum2: "(f has_sum (Polygamma (n-1) (1-z) / fact (n-1)))
{..-1}" .

  have "(f has_sum (Polygamma (n-1) (1+z) / fact (n-1)) + Polygamma (n-1)
(1-z) / fact (n-1))
          ({1..} ∪ {..-1})"
    by (intro has_sum_Un_disjoint sum1 sum2) auto
  also have "({1..} ∪ {..-1}) = -{0::int}"
    by auto
  finally have "(f has_sum ((Polygamma (n-1) (1+z) + Polygamma (n-1) (1-z))
/ fact (n-1))) (-{0})"
    by (simp add: add_divide_distrib)
  hence "(f has_sum (f 0 + (Polygamma (n-1) (1+z) + Polygamma (n-1) (1-z))
/ fact (n-1))) (insert 0 (-{0}))"
    by (intro has_sum_insert) auto
  also have "insert 0 (-{0}) = (UNIV :: int set)"
    by auto
  finally show ?thesis
    by (simp add: eisenstein_fourier_aux_def f_def algebra_simps)
qed

lemma eisenstein_fourier_aux_expansion:
  assumes n: "odd n" and z: "Im z > 0"
  shows   "eisenstein_fourier_aux (n + 1) z =
             (2 * i * pi) ^ Suc n / fact n * polylog (-int n) (to_q 1 z)"
proof -
```

```
  have eq0: "1 / z + cot_pfd z = -i * pi * (2 * polylog 0 (to_q 1 z) +
1)"
    if z: "Im z > 0" for z :: complex
  proof -
    define x where "x = exp (2 * pi * i * z)"
    have *: "exp (2 * pi * i * z) = exp (pi * i * z) ^ 2"
      by (subst exp_double [symmetric]) (simp add: algebra_simps)
    have "norm x < 1"
      using z by (auto simp: x_def)
    hence "x ≠ 1"
      by auto

    have "1 / z + cot_pfd z = pi * cot (pi * z)"
      using z by (intro cot_pfd_formula_complex [symmetric]) (auto elim:
Ints_cases)
    also have "pi * cos (pi * z) * (x - 1) = pi * i * sin (pi * z) * (x
+ 1)"
      using z *
      by (simp add: sin_exp_eq cos_exp_eq x_def exp_minus field_simps
power2_eq_square
               del: div_mult_self3 div_mult_self4 div_mult_self2 div_mult_self1)
    hence "pi * cot (pi * z) = i * pi * (x + 1) / (x - 1)"
      unfolding cot_def using z by (auto simp: divide_simps sin_eq_0)
    also have "... = -i * pi * (-(x + 1) / (x - 1))"
      by (simp add: field_simps)
    also have "-(x + 1) / (x - 1) = 1 + 2 * x / (1 - x)"
      using ‹x ≠ 1› by (simp add: field_simps)
    also have "... = 2 * polylog 0 x + 1"
      using ‹norm x < 1› and ‹x ≠ 1› by (simp add: polylog_0_left field_simps)
    finally show ?thesis by (simp only: x_def to_q_def) simp
  qed

  define f :: "nat ⇒ complex ⇒ complex" where
    "f = (λn z. if n = 0 then 1 / z + cot_pfd z
                else (-1) ^ n * Polygamma n (1 - z) - Polygamma n (1 +
z) +
                     (-1) ^ n * fact n / z ^ (Suc n))"

  have f_odd_eq: "f n z = -fact n * eisenstein_fourier_aux (n+1) z" if
"odd n" for n z
    using that by (auto simp add: f_def eisenstein_fourier_aux_def field_simps)

  have DERIV_f: "(f n has_field_derivative f (Suc n) z) (at z)" if "z
∉ ℤ" for n z
  proof (cases "n = 0")
    case [simp]: True
    have "((λz. 1 / z + cot_pfd z) has_field_derivative f 1 z) (at z)"
      unfolding f_def by (rule derivative_eq_intros refl | use that in
force)+
```

```
    thus ?thesis by (simp add: f_def)
  next
    case False
    have 1: "1 - z ∉ ℤ≤0" "1 + z ∉ ℤ≤0"
      using that by (metis Ints_1 Ints_diff add_diff_cancel_left' diff_diff_eq2
nonpos_Ints_Int)+
    have 2: "z ≠ 0"
      using that by auto

    have "((λz. (-1) ^ n * Polygamma n (1 - z) - Polygamma n (1 + z) +
(-1) ^ n * fact n / z ^ (Suc n))
          has_field_derivative (-1) ^ Suc n * Polygamma (Suc n) (1 -
z) -
            Polygamma (Suc n) (1 + z) + (-1) ^ Suc n * fact (Suc n) /
z ^ (Suc (Suc n))) (at z)"
      by (rule derivative_eq_intros refl | use that 1 2 in force)+
    thus ?thesis
      using that False by (simp add: f_def)
  qed

  define g :: "nat ⇒ complex ⇒ complex" where
    "g = (λn z. -pi * ı * (2 * ı * pi) ^ n * (2 * polylog (-n) (to_q 1
z) + (if n = 0 then 1 else 0)))"

  have g_eq: "g n z = -((2 * ı * pi) ^ Suc n) * polylog (-n) (to_q 1 z)"
if "n > 0" for n z
    using that unfolding g_def by simp

  note [derivative_intros del] = DERIV_sum
  have DERIV_g: "(g n has_field_derivative g (Suc n) z) (at z)" if z:
"Im z > 0" for z n
  proof -
    have "norm (to_q 1 z) = exp (- (2 * pi * Im z))"
      by simp
    also have "... < exp 0"
      using that by (subst exp_less_cancel_iff) auto
    finally have "norm (to_q (Suc 0) z) ≠ 1"
      by auto
    moreover have "norm (1 :: complex) = 1"
      by simp
    ultimately have [simp]: "to_q (Suc 0) z ≠ 1"
      by metis
    show ?thesis unfolding g_def
      by (rule derivative_eq_intros refl | (use z in simp; fail))+
         (auto simp: algebra_simps minus_diff_commute)
  qed

  have eq: "f n z = g n z" if "Im z > 0" for z n
    using that
```

```
  proof (induction n arbitrary: z)
    case (0 z)
    have "norm (to_q 1 z) < 1"
      unfolding to_q_def using 0 by simp
    hence "polylog 0 (to_q 1 z) = to_q 1 z / (1 - to_q 1 z)"
      by (subst polylog_0_left) auto
    thus ?case
      using eq0[of z] 0 by (auto simp: complex_is_Int_iff f_def g_def)
  next
    case (Suc n z)
    have "(f n has_field_derivative f (Suc n) z) (at z)"
      by (rule DERIV_f) (use Suc.prems in ‹auto simp: complex_is_Int_iff›)
    also have "?this ⟷ (g n has_field_derivative f (Suc n) z) (at z)"
    proof (rule DERIV_cong_ev)
      have "eventually (λz. z ∈ {z. Im z > 0}) (nhds z)"
        using Suc.prems by (intro eventually_nhds_in_open) (auto simp:
open_halfspace_Im_gt)
      thus "eventually (λz. f n z = g n z) (nhds z)"
        by eventually_elim (use Suc.IH in auto)
    qed auto
    finally have "(g n has_field_derivative f (Suc n) z) (at z)" .
    moreover have "(g n has_field_derivative g (Suc n) z) (at z)"
      by (rule DERIV_g) fact
    ultimately show "f (Suc n) z = g (Suc n) z"
      using DERIV_unique by blast
  qed

  from z have "f n z = g n z"
    by (intro eq) auto
  also have "f n z = -fact n * eisenstein_fourier_aux (n+1) z"
    using n by (subst f_odd_eq) auto
  also have "g n z = - ((2 * ı * pi) ^ Suc n) * polylog (-n) (to_q 1 z)"
    using n by (subst g_eq) (auto elim: oddE)
  finally show ?thesis
    by (simp add: to_q_def field_simps)
qed
```

With this, we can now express the Fourier expansion of the Eisenstein series of the lattice $\Lambda(1, \tau)$ with $\mathrm{Im}(\tau) > 0$ in terms of a Lambert series:

$$G_k = 2(\zeta(k) + \frac{(2i\pi)^k}{(k-1)!} L(n^{k-1}, q))$$

Here, as usual, $q = e^{2i\pi\tau}$ and

$$L(n^{k-1}, q) = \sum_{n \geq 1} n^{k-1} \frac{q^n}{1-q^n} = \sum_{n \geq 1} \sigma_{k-1}(n) q^n$$

```
lemma (in std_complex_lattice) eisenstein_series_conv_lambert:
```

```
  assumes k: "k ≥ 2" "even k"
  defines "x ≡ to_q 1 τ"
  shows "eisenstein_series k =
           2 * (zeta k + (2 * i * pi) ^ k / fact (k - 1) * lambert (λn.
of_nat n ^ (k-1)) x)"
proof -
  have x: "norm x < 1"
    using Im_τ_pos by (simp add: x_def to_q_def)

  define g where "g = (λ(n,m). 1 / (of_int m + of_int n * τ) ^ k)"
  define C where "C = (2 * i * pi) ^ k / fact (k-1)"
  define S where "S = lambert (λn. of_nat n ^ (k-1)) x"

  have sum1: "((λm. g (int n, m)) has_sum C * polylog (1 - int k) (x ^
n)) UNIV" if n: "n > 0" for n
  proof -
    have "((λm. g (int n, m)) has_sum eisenstein_fourier_aux k (of_nat
n * τ)) UNIV"
      using has_sum_eisenstein_fourier_aux[of k "of_nat n * τ"] n k Im_τ_pos
      by (simp add: g_def add_ac)
    also have "eisenstein_fourier_aux k (of_nat n * τ) =
                 C * polylog (1 - int k) (to_q 1 (of_nat n * τ))"
      using eisenstein_fourier_aux_expansion[of "k-1" "of_nat n * τ"]
Im_τ_pos n k
      by (simp add: C_def)
    also have "to_q 1 (of_nat n * τ) = x ^ n"
      by (simp add: x_def to_q_power)
    finally show ?thesis .
  qed

  have "((λn. polylog (1 - int k) (x ^ n)) has_sum S) {1..}"
    using lambert_power_int_has_sum_polylog_gen[of x 1 "int k - 1"] x
k
    by (simp add: power_int_def nat_diff_distrib S_def)
  hence "((λn. C * polylog (1 - int k) (x ^ n)) has_sum (C * S)) {1..}"
    by (rule has_sum_cmult_right)
  also have "?this ⟷ ((λn. (∑∞m. g (int n, m))) has_sum (C * S))
{1..}"
    by (rule has_sum_cong) (use sum1 in ‹auto simp: has_sum_iff›)
  also have "... ⟷ ((λn. (∑∞m. g (n, m))) has_sum (C * S)) {1..}"
    by (rule has_sum_reindex_bij_witness[of _ nat int]) auto
  finally have sum2: "((λn. (∑∞m. g (n, m))) has_sum (C * S)) {1..}"
.

  also have "?this ⟷ ((λn. (∑∞m. g (-n, m))) has_sum (C * S)) {..-1}"
    by (rule has_sum_reindex_bij_witness[of _ uminus uminus]) (auto simp:
g_def)
  also have "(λn. ∑∞m. g (-n, m)) = (λn. ∑∞m. g (n, m))"
  proof
```

```
    fix n :: int
    show "(∑∞m. g (-n, m)) = (∑∞m. g (n, m))"
      by (rule infsum_reindex_bij_witness[of _ uminus uminus])
         (use k in ‹auto simp: g_def even_power_diff_commute›)
  qed
  finally have "((λn. ∑∞m. g (n, m)) has_sum C * S) {..-1}" .
  hence "((λn. ∑∞m. g (n, m)) has_sum (C * S + C * S)) ({..-1} ∪ {1..})"
    by (intro has_sum_Un_disjoint sum2) auto
  also have "C * S + C * S = 2 * C * S"
    by simp
  also have "{..-1} ∪ {1..} = -{0::int}"
    by auto
  finally have sum3: "((λn. ∑∞m. g (n, m)) has_sum 2 * C * S) (-{0})"
.

  have "eisenstein_series k = 2 * zeta (of_nat k) + (∑∞n∈-{0}. ∑∞m.
g (n, m))"
    using k by (simp add: eisenstein_series_def g_def of_ω12_coords_def)
  also have "(∑∞n∈-{0}. ∑∞m. g (n, m)) = 2 * C * S"
    using sum3 by (simp add: has_sum_iff)
  finally show ?thesis
    by (simp add: C_def S_def)
qed
```

7.6 Behaviour under lattice transformations

In this section, we will show how the Eisenstein series and related lattice properties behave under various lattice operations such as unimodular transformations and stretching.

In particular, we will see that the invariant j is actually invariant under unimodular transformations and stretching. This is Apostol's Theorem 1.16.

```
context complex_lattice_swap
begin

lemma eisenstein_series_swap [simp]:
  assumes "k ≠ 2"
  shows   "swap.eisenstein_series k = eisenstein_series k"
proof (cases "k ≥ 3")
  case True
  thus ?thesis
    unfolding eisenstein_series_altdef[OF True] swap.eisenstein_series_altdef[OF
True] by simp
next
  case False
  with assms have "k = 0 ∨ k = 1"
    by auto
  thus ?thesis
    by auto
```

qed

lemma *eisenstein_fun_aux_swap [simp]: "swap.eisenstein_fun_aux = eisenstein_fun_aux"*
  unfolding *eisenstein_fun_aux_def [abs_def] swap.eisenstein_fun_aux_def [abs_def]* **by** *(auto cong: if_cong)*

lemma *invariant_g2_swap [simp]: "swap.invariant_g2 = invariant_g2"*
  and *invariant_g3_swap [simp]: "swap.invariant_g3 = invariant_g3"*
  unfolding *invariant_g2_def [abs_def] swap.invariant_g2_def [abs_def] invariant_g3_def [abs_def] swap.invariant_g3_def [abs_def]*
by *auto*

lemma *discr_swap [simp]: "swap.discr = discr"*
  by *(simp add: discr_def swap.discr_def)*

lemma *invariant_j_swap [simp]: "swap.invariant_j = invariant_j"*
  by *(simp add: invariant_j_def swap.invariant_j_def)*

end

context *complex_lattice_cnj*
begin

lemma *eisenstein_series_cnj [simp]: "cnj.eisenstein_series n = cnj (eisenstein_series n)"*
  unfolding *eisenstein_series_def cnj.eisenstein_series_def*
  by *(simp flip: zeta_cnj infsum_cnj add: of_ω12_coords_def cnj.of_ω12_coords_def)*

lemma *invariant_g2_cnj [simp]: "cnj.invariant_g2 = cnj invariant_g2"*
  and *invariant_g3_cnj [simp]: "cnj.invariant_g3 = cnj invariant_g3"*
  by *(simp_all add: cnj.invariant_g2_def invariant_g2_def cnj.invariant_g3_def invariant_g3_def)*

lemma *discr_cnj [simp]: "cnj.discr = cnj discr"*
  by *(simp add: discr_def cnj.discr_def)*

lemma *invariant_j_cnj [simp]: "cnj.invariant_j = cnj invariant_j"*
  by *(simp add: invariant_j_def cnj.invariant_j_def)*

end

context *complex_lattice_stretch*
begin

lemma *eisenstein_series_stretch:*
  *"stretched.eisenstein_series n = c powi (-n) * eisenstein_series n"*
proof *(cases "n ≥ 3")*

```
  case True
  have "stretched.eisenstein_series n = (∑∞x∈Λ*. c powi (-n) * (1 /
x ^ n))"
    using infsum_reindex_bij_betw[OF bij_betw_stretch_lattice0, where
f = "λω. 1 / ω ^ n"] True
    unfolding stretched.eisenstein_series_altdef[OF True] by (simp add:
power_int_minus field_simps)
  also have "... = c powi (-n) * eisenstein_series n"
    using True by (subst infsum_cmult_right') (auto simp: eisenstein_series_altdef)
  finally show ?thesis .
next
  case False
  hence "n = 0 ∨ n = 1 ∨ n = 2"
    by auto
  thus ?thesis
  proof (elim disjE)
    assume [simp]: "n = 2"
    show "stretched.eisenstein_series n = c powi -int n * G n"
      unfolding eisenstein_series_def stretched.eisenstein_series_def
stretched_of_ω12_coords
      apply (simp add: ring_distribs flip: infsum_cmult_right')
      apply (simp add: power_int_minus field_simps)?
      done
  qed auto
qed

lemma invariant_g2_stretch [simp]: "stretched.invariant_g2 = invariant_g2
/ c ^ 4"
  and invariant_g3_stretch [simp]: "stretched.invariant_g3 = invariant_g3
/ c ^ 6"
  unfolding invariant_g2_def stretched.invariant_g2_def
            invariant_g3_def stretched.invariant_g3_def eisenstein_series_stretch
  by (simp_all add: power_int_minus field_simps)

lemma discr_stretch [simp]: "stretched.discr = discr / c ^ 12"
  unfolding stretched.discr_def discr_def invariant_g2_stretch invariant_g3_stretch
  by (simp add: field_simps stretch_nonzero)

lemma invariant_j_stretch [simp]: "stretched.invariant_j = invariant_j"
  unfolding stretched.invariant_j_def invariant_j_def invariant_g2_stretch
discr_stretch
  by (simp add: field_simps stretch_nonzero)

end


context unimodular_moebius_transform_lattice
begin
```

```
lemma eisenstein_series_transformed [simp]:
  assumes "k ≠ 2"
  shows   "transformed.eisenstein_series k = eisenstein_series k"
proof (cases "k ≥ 3")
  case True
  thus ?thesis
    by (simp add: transformed.eisenstein_series_altdef eisenstein_series_altdef
transformed_lattice0_eq)
next
  case False
  with assms have "k = 0 ∨ k = 1"
    by auto
  thus ?thesis
    by auto
qed

lemma invariant_g2_transformed [simp]: "transformed.invariant_g2 = invariant_g2"
  and invariant_g3_transformed [simp]: "transformed.invariant_g3 = invariant_g3"
     by ((intro ext)?, unfold invariant_g2_def invariant_g3_def number_e1_def
          transformed.invariant_g2_def transformed.invariant_g3_def transformed_lattice0_eq
simp_all

lemma discr_transformed [simp]: "transformed.discr = discr"
  by (simp add: transformed.discr_def discr_def)

lemma invariant_j_transformed [simp]: "transformed.invariant_j = invariant_j"
  by (simp add: transformed.invariant_j_def invariant_j_def)

end
```

7.7 Recurrence relation

```
context complex_lattice
begin
```

Using our formal ODE from above, we find the following recurrence for G_n. By unfolding this repeatedly, we can write any G_n as a polynomial in G_4 and G_6 – or, equivalently, in g_2 and g_3.

This is Theorem 1.13 in Apostol's book.

```
lemma eisenstein_series_recurrence_aux:
  defines "b ≡ λn. (2*n + 1) * (G (2*n + 2))"
  shows "b 1 = g₂ / 20"
    and "b 2 = g₃ / 28"
    and "⋀n. n ≥ 3 ⟹ (2 * of_nat n + 3) * (of_nat n - 2) * b n = 3
* (∑ i=1..n-2. b i * b (n - i - 1))"
proof -
  show "b 1 = g₂ / 20" "b 2 = g₃ / 28"
    by (simp_all add: b_def invariant_g2_def invariant_g3_def)
next
```

```
  fix n :: nat assume n: "n ≥ 3"

  define c where "c = fls_nth fls_weierstrass"
  have b_c: "b n = c (2 * n)" if "n > 0" for n
    using that by (simp add: c_def fls_nth_weierstrass nat_add_distrib
b_def nat_mult_distrib)

  have "(2 * of_nat n) * (2 * of_nat n - 1) * b n =
        6 * fls_nth (fls_weierstrass^2) (2 * int n - 2)"
    using arg_cong[OF fls_weierstrass_ODE2, of "λF. fls_nth F (2 * (n
- 1))"] n
    by (simp add: algebra_simps of_nat_diff nat_mult_distrib b_c c_def)
  also have "fls_nth (fls_weierstrass^2) (2 * int n - 2) =
             (∑ i=-2..2 * int n. c i * c (2 * int n - 2 - i))"
    by (simp add: power2_eq_square fls_times_nth(2) fls_subdegree_weierstrass
flip: c_def)
  also have "... = (∑ i∈{-2..2 * int n}-{n. odd n}. c i * c (2 * int n
- 2 - i))"
    by (intro sum.mono_neutral_right)
       (auto simp: c_def fls_nth_weierstrass eisenstein_series_odd_eq_0
even_nat_iff)
  also have "... = (∑ i∈{-1, int n} ∪ {0..<int n}. c (2 * i) * c (2 *
(int n - i - 1)))"
    by (intro sum.reindex_bij_witness[of _ "λk. 2 * k" "λk. k div 2"])
(auto simp: algebra_simps)
  also have "... = 2 * b n + (∑ i=0..<int n. c (2 * i) * c (2 * (int n
- i - 1)))"
    using n by (subst sum.union_disjoint) (auto simp: c_def fls_nth_weierstrass
b_c)
  also have "(∑ i=0..<int n. c (2 * i) * c (2 * (int n - i - 1))) =
             (∑ i=0..<n. c (2 * int i) * c (2 * (int n - int i - 1)))"
    by (intro sum.reindex_bij_witness[of _ int nat]) (auto simp: of_nat_diff)
  also have "... = (∑ i=1..n-2. c (2 * int i) * c (2 * (int n - int i
- 1)))"
    by (intro sum.mono_neutral_right) (auto simp: c_def fls_nth_weierstrass)
  also have "... = (∑ i=1..n-2. b i * b (n - i - 1))"
    using n by (intro sum.cong) (auto simp: b_c of_nat_diff algebra_simps)
  finally show "(2 * of_nat n + 3) * (of_nat n - 2) * b n
      = 3 * (∑ i=1..n-2. b i * b (n - i - 1))"
    by Groebner_Basis.algebra
qed

theorem eisenstein_series_recurrence:
  assumes "n ≥ 2"
  shows "G (2*n+4) = 3 / of_nat ((2*n+5) * (n-1) * (2*n+3)) *
           (∑ i≤n-2. of_nat ((2*i+3) * (2*(n-i)-1)) * G (2*i+4) * G (2*(n-2-i)+4))"
proof -
  define c :: nat where "c = (2*(n+1)+3) * ((n+1)-2) * (2*(n+1)+1)"
  have c_altdef: "c = (2*n+5) * (n-1) * (2*n+3)"
```

```
    using assms unfolding c_def by (intro arg_cong2[of _ _ _ _ "(*)"])
auto
  define S where "S = (∑ i≤n-2. of_nat ((2*i+3) * (2*(n-i)-1)) * G (2*i+4)
* G (2*(n-2-i)+4))"
  have [simp]: "c ≠ 0"
    using assms unfolding c_def mult_eq_0_iff by auto
  have *: "n + 1 ≥ 3"
    using assms by linarith
  have "of_nat c * G (2 * (n+1) + 2) =
          3 * (∑ i=1..(n+1) - 2. (2 * i + 1) * (2 * ((n+1) - i - 1) +
1) * G (2 * i + 2) * G (2 * ((n+1) - i - 1) + 2))"
    using eisenstein_series_recurrence_aux(3)[OF *] assms unfolding c_def
    apply simp
    apply (simp add: algebra_simps)
    done
  also have "2 * (n+1) + 2 = 2 * n + 4"
    using assms by simp
  also have "(∑ i=1..(n+1) - 2. (2 * i + 1) * (2 * ((n+1) - i - 1) + 1)
* G (2 * i + 2) * G (2 * ((n+1) - i - 1) + 2)) =
             (∑ i≤(n+1) - 3. (2 * (i+1) + 1) * (2 * ((n+1) - (i+1) -
1) + 1) * G (2 * (i+1) + 2) * G (2 * ((n+1) - (i+1) - 1) + 2))"
    by (intro sum.reindex_bij_witness[of _ "λi. i+1" "λi. i-1"]) (use
assms in auto)
  also have "(n+1) - 3 = n - 2"
    using assms by linarith
  also have "(∑ i≤n-2. (2 * (i+1) + 1) * (2 * ((n+1) - (i+1) - 1) + 1)
* G (2 * (i+1) + 2) * G (2 * ((n+1) - (i+1) - 1) + 2)) = S"
    using assms unfolding S_def
    apply (intro sum.cong refl arg_cong2[of _ _ _ _ "(*)"] arg_cong[of
_ _ of_nat] arg_cong[of _ _ G])
       apply (auto simp: algebra_simps Suc_diff_Suc)
    done
  finally have "G (2 * n + 4) = 3 / of_nat c * S"
    by (simp add: field_simps)
  thus ?thesis
    unfolding S_def c_altdef .
qed

end
```

With this we can now write some code to compute representations of G_n in terms of G_4 and G_6. Our code returns a bivariate polynomial with rational coefficients.

```
fun eisenstein_series_poly :: "nat ⇒ rat poly poly" where
  "eisenstein_series_poly n =
     (if n = 0 then [: [:0, 1:] :]
      else if n = 1 then [:0, 1:]
      else
        Polynomial.smult [:3 / of_nat ((2*n+5) * (n-1) * (2*n+3)):]
```

```
        (∑ i≤n-2. Polynomial.smult (of_nat ((2*i+3)*(2*(n-i)-1)))
          (eisenstein_series_poly i * eisenstein_series_poly (n-2-i))))"

lemmas [simp del] = eisenstein_series_poly.simps

lemma eisenstein_series_poly_0 [simp]: "eisenstein_series_poly 0 = [:
[:0, 1:] :]"
  and eisenstein_series_poly_1 [simp]: "eisenstein_series_poly (Suc 0)
= [:0, 1:]"
  and eisenstein_series_poly_rec:
      "n ≥ 2 ⟹ eisenstein_series_poly n =
         Polynomial.smult [:3 / of_nat ((2*n+5) * (n-1) * (2*n+3)):]
           (∑ i≤n-2. Polynomial.smult (of_nat ((2*i+3)*(2*(n-i)-1)))

             (eisenstein_series_poly i * eisenstein_series_poly (n-2-i)))"
  by (subst eisenstein_series_poly.simps; simp; fail)+

context complex_lattice
begin

lemma eisenstein_series_poly_correct:
  "poly2 (map_poly2 of_rat (eisenstein_series_poly n)) (G 4) (G 6) = G
(2 * n + 4)"
proof (induction n rule: eisenstein_series_poly.induct)
  case (1 n)
  interpret map1: map_poly_comm_ring_hom "of_rat :: rat ⇒ complex"
    by standard auto
  interpret map2: map_poly_comm_ring_hom "map_poly (of_rat :: rat ⇒ complex)"
    by standard auto
  consider "n = 0" | "n = 1" | "n ≥ 2"
    by linarith
  thus ?case
  proof cases
    case 3
    define c where "c = 3 / complex_of_nat ((2 * n + 5) * (n - 1) * (2
* n + 3))"
    have "poly2 (map_poly2 of_rat (eisenstein_series_poly n)) (G 4) (G
6) =
          c * (∑ i≤n-2. (of_nat ((2 * i + 3) * (2 * (n - i) - 1)) *
                        G (2 * i + 4) * G (2 * (n - 2 - i) + 4)))"
      using 3 1
      apply (simp add: eisenstein_series_poly_rec map_poly2_def hom_distribs
c_def sum_distrib_left)
      apply (simp add: algebra_simps)?
      done
    also have "... = G (2 * n + 4)"
      unfolding c_def by (rule eisenstein_series_recurrence[OF ‹n ≥ 2›,
symmetric])
    finally show ?thesis .
```

```
  qed (auto simp: map_poly_pCons map_poly2_def)
qed

end
```

We employ memoisation for better performance:

```
definition eisenstein_polys_step :: "rat poly poly list ⇒ rat poly poly
list" where
  "eisenstein_polys_step ps =
     (let n = length ps
      in  ps @ [Polynomial.smult [:3 / of_nat ((2*n+5) * (n-1) * (2*n+3)):]
             (∑ i≤n-2. Polynomial.smult (of_nat ((2*i+3)*(2*(n-i)-1)))

               (ps ! i * ps ! (n-2-i)))])"

definition eisenstein_series_polys :: "nat ⇒ rat poly poly list" where
  "eisenstein_series_polys n = (eisenstein_polys_step ^^ (n - 2)) [[:
[:0, 1:] :], [:0, 1:]]"

lemma eisenstein_polys_step_correct:
  assumes n: "n ≥ 2" and ps_eq: "ps = map eisenstein_series_poly [0..<n]"
  shows   "eisenstein_polys_step ps = map eisenstein_series_poly [0..<Suc
n]"
proof (rule nth_equalityI)
  fix i assume "i < length (eisenstein_polys_step ps)"
  have length: "length ps = n"
    by (simp add: ps_eq)
  define p where "p = Polynomial.smult [:3 / of_nat ((2*n+5) * (n-1)
* (2*n+3)):]
             (∑ i≤n-2. Polynomial.smult (of_nat ((2*i+3)*(2*(n-i)-1)))

               (ps ! i * ps ! (n-2-i)))"
  have step: "eisenstein_polys_step ps = ps @ [p]"
    by (simp add: eisenstein_polys_step_def p_def length Let_def)
  have i: "i ≤ n"
    using ‹i < _› unfolding ps_eq length eisenstein_polys_step_def by
simp
  show "eisenstein_polys_step ps ! i = map eisenstein_series_poly [0..<Suc
n] ! i"
  proof (cases "i = n")
    case False
    thus ?thesis
      using i unfolding step by (auto simp: ps_eq nth_append simp del:
upt_Suc)
  next
    case [simp]: True
    have "eisenstein_polys_step ps ! i = p"
      by (auto simp: step nth_append length)
    also have "p = eisenstein_series_poly n"
```

```
      unfolding eisenstein_series_poly_rec[OF n] p_def ps_eq
      by (intro arg_cong2[of _ _ _ _ Polynomial.smult] refl sum.cong)
(use n in auto)
    also have "... = map eisenstein_series_poly [0..<Suc n] ! i"
      by (simp del: upt_Suc)
    finally show ?thesis .
  qed
qed (auto simp: eisenstein_polys_step_def ps_eq)

lemma eisenstein_series_polys_correct:
  "eisenstein_series_polys n = map eisenstein_series_poly [0..<max 2 n]"
proof (induction n rule: less_induct)
  case (less n)
  show ?case
  proof (cases "n ≥ 3")
    case False
    thus ?thesis
      by (auto simp: eisenstein_series_polys_def upt_rec)
  next
    case True
    define m where "m = n - 3"
    have n_eq: "n = Suc (Suc (Suc m))"
      using True unfolding m_def by simp
    have "eisenstein_polys_step (eisenstein_series_polys (n-1)) =
            map eisenstein_series_poly [0..<Suc (n-1)]"
    proof (rule eisenstein_polys_step_correct)
      have "eisenstein_series_polys (n - 1) = map eisenstein_series_poly
[0..<max 2 (n-1)]"
        by (rule less.IH) (use True in auto)
      thus "eisenstein_series_polys (n - 1) = map eisenstein_series_poly
[0..<n-1]"
        using True by simp
    qed (use True in auto)
    also have "eisenstein_polys_step (eisenstein_series_polys (n-1)) =
eisenstein_series_polys n"
      by (simp add: eisenstein_series_polys_def eval_nat_numeral n_eq)
    also have "Suc (n - 1) = max 2 n"
      using True by auto
    finally show ?thesis .
  qed
qed


lemma funpow_rec_right: "n > 0 ⟹ (f ^^ n) xs = (f ^^ (n-1)) (f xs)"
  by (cases n) (auto simp del: funpow.simps simp: funpow_Suc_right)

context complex_lattice
begin
```

```
lemma eisenstein_series_polys_correct':
  assumes "eisenstein_series_polys m = ps"
  shows   "list_all (λi. G (2*i+4) = poly2 (map_poly2 of_rat (ps ! i))
(G 4) (G 6)) [0..<m]"
  unfolding assms [symmetric] eisenstein_series_polys_correct
  using eisenstein_series_poly_correct by (auto simp: list_all_iff)
```

We now compute the relations up to G_{20} for demonstration purposes. This could in principle be turned into a proof method as well.

```
lemma eisenstein_series_relations:
  "G  8 = 3 / 7 * G 4 ^ 2" (is ?th8)
  "G 10 = 5 / 11 * G 4 * G 6" (is ?th10)
  "G 12 = 18 / 143 * G 4 ^ 3 + 25 / 143 * G 6 ^ 2" (is ?th12)
  "G 14 = 30 / 143 * G 4 ^ 2 * G 6" (is ?th14)
  "G 16 = 27225 / 668525 * G 4 ^ 4 + 300 / 2431 * G 4 * G 6 ^ 2" (is ?th16)
  "G 18 = 3915 / 46189 * G 4 ^ 3 * G 6 + 2750 / 92378 * G 6 ^ 3" (is ?th18)
  "G 20 = 54 / 4199 * G 4 ^ 5 + 36375 / 508079 * G 4 ^ 2 * G 6 ^ 2" (is
?th20)
proof -
  have eq: "eisenstein_series_polys 9 =
              [[:[:0, 1:]:], [:0, [:1:]:], [:[:0, 0, 3 / 7:]:],
               [:0, [:0, 5 / 11:]:], [:[:0, 0, 0, 18 / 143:], 0, [:25
/ 143:]:],
               [:0, [:0, 0, 30 / 143:]:], [:[:0, 0, 0, 0, 9 / 221:], 0,
[:0, 300 / 2431:]:],
               [:0, [:0, 0, 0, 3915 / 46189:], 0, [:125 / 4199:]:],
               [:[:0, 0, 0, 0, 0, 54 / 4199:], 0, [:0, 0, 36375 / 508079:]:]]"
    by (simp add: eisenstein_series_polys_def eisenstein_polys_step_def
                  funpow_rec_right numeral_poly smult_add_right smult_diff_right
flip: pCons_one)
  thus ?th8 ?th10 ?th12 ?th14 ?th16 ?th18 ?th20
    using eisenstein_series_polys_correct'[OF eq]
    by (simp_all add: upt_rec map_poly2_def of_rat_divide power_numeral_reduce
field_simps)
qed

end

end
```

8 Addition and duplication theorems for $\wp$

```
theory Weierstrass_Addition
  imports Eisenstein_Series
begin
```

In this section, we shall derive the addition theorem for $\wp$, and from it the

duplication theorem. The addition theorem is:

$$\wp(w+z) = -\wp(w) - \wp(z) + \tfrac{1}{4}\left(\frac{\wp'(w)-\wp'(z)}{\wp(w)-\wp(z)}\right)^2$$

We first prove this with the additional assumptions that w and z are in "general position", i.e. we have neither $w + 2z \in \Lambda$ nor $z + 2w \in \Lambda$.

After that, we will drop these unnecessary assumptions using analytic continuation. Our proof follows Lang's presentation [2].

```
lemma pos_sum_eq_0_imp_empty:
  fixes f :: "'a ⇒ 'b :: ordered_comm_monoid_add"
  assumes "(∑x∈A. f x) = 0" "⋀x. x ∈ A ⟹ f x > 0" "finite A"
  shows   "A = {}"
proof (rule ccontr)
  assume "A ≠ {}"
  hence "(∑x∈A. f x) > 0"
    by (intro sum_pos) (use assms in auto)
  with assms(1) show False by simp
qed

context complex_lattice
begin

lemma weierstrass_fun_add_aux:
  assumes u12: "u1 ∉ Λ" "u2 ∉ Λ" "¬rel u1 u2" "¬rel u1 (-u2)"
  assumes general_position: "u1 + 2 * u2 ∉ Λ" "2 * u1 + u2 ∉ Λ"
  shows   "℘ (u1 + u2) = -℘ u1 - ℘ u2 + ((℘' u1 - ℘' u2) / (℘ u1 - ℘
u2))^2 / 4"
proof -
  note [simp] = weierstrass_fun.eval_to_fund_parallelogram
                weierstrass_fun_deriv.eval_to_fund_parallelogram
```

We introduce the copies u_1', u_2' of u_1 and u_2 in the fundamental parallelogram for convenience.

```
  define u1' where "u1' = to_fund_parallelogram u1"
  define u2' where "u2' = to_fund_parallelogram u2"
  have [simp]: "u1' ≠ u2'" "u2' ≠ u1'"
    using u12 by (auto simp: u1'_def u2'_def rel_sym)
```

Let a and b be such that $(u_1, \wp(u_1))$ and $(u_2, \wp(u_2))$ lie on the line $ax+b = 0$, i.e. such that u_1 and u_2 are both solutions of the linear equation $\wp'(u) = a\wp(u) + b$.

```
  define a where "a = (℘' u1 - ℘' u2) / (℘ u1 - ℘ u2)"
  define b where "b = ℘' u1 - a * ℘ u1"
  have ab: "℘' u1 = a * ℘ u1 + b" "℘' u2 = a * ℘ u2 + b"
  proof -
    show "℘' u1 = a * ℘ u1 + b"
      using u12 by (auto simp: b_def)
```

```
    have "a * (℘ u1 - ℘ u2) = (℘' u1 - ℘' u2)"
      unfolding a_def using u12 by (simp add: weierstrass_fun_eq_iff)
    thus "℘' u2 = a * ℘ u2 + b"
      by (simp add: algebra_simps b_def)
  qed
```

We define the function $f(z) = \wp'(z) - (a\wp(z) + b)$ and note that it has a triple pole at the lattice points and no other poles and therefore order 3.

```
  define f where "f = remove_sings (λz. ℘' z - (a * ℘ z + b))"
  interpret f: nicely_elliptic_function ω1 ω2 f
    unfolding f_def by (intro elliptic_function_intros)
  note [simp] = f.zorder.eval_to_fund_parallelogram f.eval_to_fund_parallelogram
  have f_eq: "f z = ℘' z - (a * ℘ z + b)" if "z ∉ Λ" for z
    unfolding f_def by (rule remove_sings_at_analytic) (auto intro!: analytic_intros
simp: that)

  have pole_f: "is_pole f z" "zorder f z = -3" if "z ∈ Λ" for z
  proof -
    define F where "F = fls_deriv fls_weierstrass - (fls_const a * fls_weierstrass
+ fls_const b)"
    have F: "f has_laurent_expansion F" unfolding f_def F_def
      by (intro laurent_expansion_intros)

    have "fls_nth F n = 0" if "n < -3" for n
      unfolding F_def using that by (auto simp: fls_weierstrass_def)
    moreover have "fls_nth F (-3) ≠ 0"
      unfolding F_def by (auto simp: fls_weierstrass_def)
    ultimately have "fls_subdegree F = -3"
      using fls_subdegree_eqI by blast
    with F have "zorder f 0 = -3" "is_pole f 0"
      using has_laurent_expansion_imp_is_pole_0 has_laurent_expansion_zorder_0
by fastforce+
    thus "is_pole f z" "zorder f z = -3"
      using f.poles.lattice_cong[of z 0] f.zorder.lattice_cong[of z 0]
that
      by (simp_all add: rel_def)
  qed

  have is_pole_f_iff: "is_pole f z ⟷ z ∈ Λ" for z
  proof -
    have "f analytic_on -Λ"
      unfolding f_def by (intro analytic_intros remove_sings_analytic_on)
auto
    with pole_f[of z] show ?thesis
      by (meson ComplI analytic_at_imp_no_pole analytic_on_analytic_at)
  qed

  have analytic_at_f: "f analytic_on {z}" if "z ∉ Λ" for z
    unfolding f_def using that by (intro analytic_intros remove_sings_analytic_on)
```

```
auto

  interpret f: nonconst_nicely_elliptic_function ω1 ω2 f
  proof
    have "elliptic_order f ≠ 0"
      using pole_f[of 0] by (subst f.elliptic_order_eq_0_iff_no_poles)
auto
    thus "elliptic_order f > 0"
      by linarith
  qed

  have order_f: "elliptic_order f = 3"
  proof -
    have "elliptic_order f = (∑z | z ∈ period_parallelogram 0 ∧ is_pole
f z. nat (- zorder f z))"
      by (rule f.poles_eq_elliptic_order [of 0, symmetric])
    also have "... = (∑z∈{0::complex}. 3)"
    proof (intro sum.cong)
      have "{z ∈ period_parallelogram 0. is_pole f z} ⊆ {0}"
        using fund_period_parallelogram_in_lattice_iff by (auto simp:
is_pole_f_iff)
      moreover have "{0} ⊆ {z ∈ period_parallelogram 0. is_pole f z}"
        using pole_f[of 0] by auto
      ultimately show "{z ∈ period_parallelogram 0. is_pole f z} = {0}"
        by blast
    qed (use pole_f in auto)
    finally show ?thesis
      by simp
  qed
```

We now look at the zeros of f. We know that u_1 and u_2 are zeros.

```
  define Z where "Z = {z. z ∈ period_parallelogram 0 ∧ isolated_zero
f z}"
  have "finite Z"
    unfolding Z_def by (rule f.finite_zeros_in_parallelogram)
  have in_Z_iff: "z ∈ Z ⟷ z ∈ period_parallelogram 0 - {0} ∧ f z =
0" for z
    by (auto simp: f.isolated_zero_iff' Z_def is_pole_f_iff fund_period_parallelogram_in_la

  have "{u1', u2'} ⊆ Z"
    using u12 ab by (auto simp: analytic_at_f is_pole_f_iff u1'_def u2'_def
f_eq in_Z_iff)
  have zorder_pos: "zorder f z > 0" if "z ∈ Z" for z using that
    by (auto simp: Z_def analytic_at_f u1'_def u2'_def f.isolated_zero_iff'
is_pole_f_iff
            intro!: zorder_isolated_zero_pos)
  have zorder_pos': "zorder f u1 > 0" "zorder f u2 > 0"
    using zorder_pos[of u1'] zorder_pos[of u2'] ‹{u1', u2'} ⊆ Z› by (auto
simp: u1'_def u2'_def)
```

We know that the sum of the multiplicities of the zeros must be 3. We already split the sum into the contributions of u_1 and u_2 and those of any remaining zeros.

```
  have sum_zorder_eq:
    "nat (zorder f u1) + nat (zorder f u2) + (∑z∈Z-{u1',u2'}. nat (zorder
f z)) = 3"
  proof -
    have "elliptic_order f = (∑z∈Z. nat (zorder f z))"
      unfolding Z_def by (rule f.zeros_eq_elliptic_order[of 0, symmetric])
    also have "Z = {u1', u2'} ∪ (Z - {u1', u2'})"
      using ‹{u1', u2'} ⊆ Z› by auto
    also have "(∑z∈.... nat (zorder f z)) =
                nat (zorder f u1) + nat (zorder f u2) + (∑z∈Z-{u1',u2'}.
nat (zorder f z))"
      by (subst sum.union_disjoint)
         (use ‹finite Z› u12 zorder_pos' ‹{u1', u2'} ⊆ Z› in ‹auto simp:
u1'_def u2'_def›)
    finally show ?thesis
      using order_f by simp
  qed
```

We also know that the sum of the zeros and poles, weighted by their multiplicity, is a lattice point. Since the pole is at the origin, it does not contribute anything to the sum.

```
  have sum_zeros_in_lattice:
    "of_int (zorder f u1) * u1 + of_int (zorder f u2) * u2 +
       (∑z∈Z-{u1',u2'}. of_int (zorder f z) * z) ∈ Λ"
  proof -
    have "(∑z | z ∈ period_parallelogram 0 ∧ (isolated_zero f z ∨ is_pole
f z).
             of_int (zorder f z) * z) ∈ Λ"
      by (rule f.sum_zeros_poles_in_lattice[of 0])
    also have "{z. z ∈ period_parallelogram 0 ∧ (isolated_zero f z ∨
is_pole f z)} = insert 0 Z"
      using pole_f by (auto simp: Z_def fund_period_parallelogram_in_lattice_iff
is_pole_f_iff)
    also have "(∑z∈insert 0 Z. of_int (zorder f z) * z) = (∑z∈Z. of_int
(zorder f z) * z)"
      by (rule sum.mono_neutral_right) (use ‹finite Z› in auto)
    also have "Z = {u1',u2'} ∪ (Z-{u1',u2'})"
      using ‹{u1', u2'} ⊆ Z› by auto
    also have "(∑z∈.... of_int (zorder f z) * z) =
                of_int (zorder f u1) * u1' + of_int (zorder f u2) * u2'
+
                (∑z∈Z-{u1',u2'}. of_int (zorder f z) * z)"
      by (subst sum.union_disjoint)
         (use ‹finite Z› u12 zorder_pos' ‹{u1', u2'} ⊆ Z› in ‹auto simp:
u1'_def u2'_def›)
```

```
    also have "rel (of_int (zorder f u1) * u1' + of_int (zorder f u2)
* u2' +
                    (∑z∈Z-{u1',u2'}. of_int (zorder f z) * z))
                  (of_int (zorder f u1) * u1 + of_int (zorder f u2) *
u2 +
                    (∑z∈Z-{u1',u2'}. of_int (zorder f z) * z))"
      unfolding u1'_def u2'_def by (intro lattice_intros) auto
    finally show ?thesis .
  qed
```

We now show that the zeros u_1 and u_2 must be simple. If they were not simple, they would be the only zeros and consequently we would have either $2u_1 + u_2$ or $u_1 + 2u_2$, which contradicts our assumption that u_1 and u_2 are in general position.

```
  have [simp]: "zorder f u1 = 1"
  proof (rule ccontr)
    assume "zorder f u1 ≠ 1"
    hence [simp]: "zorder f u1 = 2" "zorder f u2 = 1"
      using sum_zorder_eq zorder_pos' ‹{u1', u2'} ⊆ Z›
      by (auto simp add: u1'_def u2'_def)
    have "(∑z∈Z-{u1',u2'}. nat (zorder f z)) = 0"
      using sum_zorder_eq by simp
    hence *: "Z-{u1',u2'} = {}"
      by (rule pos_sum_eq_0_imp_empty) (use ‹finite Z› in ‹auto intro:
zorder_pos›)
    have "2 * u1 + u2 ∈ Λ"
      using sum_zeros_in_lattice unfolding * by simp
    with general_position show False by simp
  qed

  have [simp]: "zorder f u2 = 1"
  proof (rule ccontr)
    assume "zorder f u2 ≠ 1"
    hence [simp]: "zorder f u1 = 1" "zorder f u2 = 2"
      using sum_zorder_eq zorder_pos' ‹{u1', u2'} ⊆ Z›
      by (auto simp add: u1'_def u2'_def)
    have "(∑z∈Z-{u1',u2'}. nat (zorder f z)) = 0"
      using sum_zorder_eq by simp
    hence *: "Z-{u1',u2'} = {}"
      by (rule pos_sum_eq_0_imp_empty) (use ‹finite Z› in ‹auto intro:
zorder_pos›)
    have "u1 + 2 * u2 ∈ Λ"
      using sum_zeros_in_lattice unfolding * by simp
    with general_position show False by simp
  qed
```

Thus we can conclude that there must be a third root u_3. It is simple, and there are no further roots.

```
  obtain u3 where u3: "u3 ∈ Z - {u1', u2'}" "zorder f u3 = 1" and Z_eq:
```

```
"Z = {u1', u2', u3}"
  proof -
    have "(∑z∈Z - {u1', u2'}. nat (zorder f z)) = 1"
      using sum_zorder_eq by simp
    then obtain u3 where u3: "Z - {u1', u2'} = {u3}" "nat (zorder f u3)
= 1"
    proof (rule sum_nat_eq_1E)
      fix u assume u: "u ∈ Z - {u1', u2'}"
      have "f analytic_on {u}"
        unfolding f_def
        by (intro analytic_intros remove_sings_analytic_on)
           (use u in ‹auto simp: in_Z_iff fund_period_parallelogram_in_lattice_iff›)
      thus "nat (zorder f u) > 0" using u
        by (auto intro!: zorder_isolated_zero_pos simp: Z_def)
    qed auto
    show ?thesis
    proof (rule that[of u3])
      have "u1' ∈ Z" "u2' ∈ Z"
        using u12 ab by (auto simp: u1'_def u2'_def in_Z_iff f.eval_to_fund_parallelogram
f_eq)
      thus "Z = {u1', u2', u3}"
        using u3(1) by blast
    qed (use u3 in auto)
  qed
  have "u3 ∉ Λ"
    using u3(1) by (auto simp: in_Z_iff fund_period_parallelogram_in_lattice_iff)
```

Since the zeros sum to a lattice point, we have $u_3 \sim -(u_1 + u_2)$.

```
  have rel_u3: "rel u3 (-(u1 + u2))"
    using sum_zeros_in_lattice u3 unfolding Z_eq
    by (simp add: u1'_def u2'_def insert_Diff_if rel_def algebra_simps
split: if_splits)
```

By definition of f, the fact that u_3 is a zero means that $(u_3, \wp(u_3))$ also lies on the line $ax + b = 0$.

```
  have "f u3 = 0"
    using u3 f.isolated_zero_iff[of u3] by (auto simp: Z_def)
  hence u3_ab: "℘' u3 = a * ℘ u3 + b"
    using ‹u3 ∉ Λ› by (subst (asm) f_eq) auto
```

From our assumptions, it also follows that $\wp$ takes on a different value for each of u_1, u_2, u_3.

```
  have inj: "inj_on ℘ {u1', u2', u3}"
  proof -
    have "-(u1 + u2) ∉ Λ"
      using ‹u3 ∉ Λ› rel_lattice_trans_right rel_u3 by blast
    have "℘ u1' ≠ ℘ u2'"
      using u12 by (auto simp: u1'_def u2'_def weierstrass_fun_eq_iff)
```

```
    moreover have "℘ u1' ≠ ℘ u3"
    proof
      assume "℘ u1' = ℘ u3"
      hence "℘ u1 = ℘ (-(u1 + u2))"
        using weierstrass_fun.lattice_cong[OF rel_u3] by (auto simp: u1'_def)
      thus False
        using u12 general_position <-(u1 + u2) ∉ Λ>
        by (auto simp: weierstrass_fun_eq_iff rel_def uminus_in_lattice_iff)
    qed
    moreover have "℘ u2' ≠ ℘ u3"
    proof
      assume "℘ u2' = ℘ u3"
      hence "℘ u2 = ℘ (-(u1 + u2))"
        using weierstrass_fun.lattice_cong[OF rel_u3] by (auto simp: u2'_def)
      thus False
        using u12 general_position <-(u1 + u2) ∉ Λ>
        by (auto simp: weierstrass_fun_eq_iff rel_def uminus_in_lattice_iff
add_ac)
    qed
    ultimately show ?thesis
      by auto
  qed
```

We now define the polynomial $P(X) = 4X^3 - g_2X - g_3 - (aX - b)^2$. Combining the fact that u_1, u_2, u_3 are all solutions of $\wp'(u) = a\wp(u) + b$ with the ODE satisfied by $\wp$, we know that $\wp(u_1), \wp(u_2), \wp(u_3)$ are roots of P. Since we also showed that the three are distinct, P has no other roots.

```
  define P where "P = [: -g₃, -g₂, 0,  4 :] - [: b, a :] ^ 2"
  have P_roots: "poly P (℘ u) = 0" if "u ∉ Λ" "℘' u = a * ℘ u + b" for
u
  proof -
    have "poly P (℘ u) = 4 * ℘ u ^ 3 - g₂ * ℘ u - g₃ - (a * ℘ u + b)
^ 2"
      by (simp add: P_def algebra_simps power3_eq_cube)
    also have "4 * ℘ u ^ 3 - g₂ * ℘ u - g₃ = ℘' u ^ 2"
      using that weierstrass_fun_ODE1'[of u] by simp
    also have "a * ℘ u + b = ℘' u"
      using that by (simp add: algebra_simps)
    finally show ?thesis
      by simp
  qed
```

Consequently, we can write P in the form $P(X) = 4(X - \wp(u_1))(X - \wp(u_2))(X - \wp(u_3))$.

```
  define Q where "Q = 4 * (∏u∈{u1',u2',u3}. [: -℘ u, 1 :])"

  have P_Q: "P = Q"
  proof (rule poly_eqI_degree_lead_coeff)
    have "poly.coeff P 3 = 4"
```

```
      by (simp add: P_def power2_eq_square eval_nat_numeral)
    moreover have "lead_coeff Q = 4"
      by (simp add: lead_coeff_mult lead_coeff_prod numeral_poly Q_def)
    moreover have "degree Q = 3"
      using u3 u12 by (auto simp: degree_mult_eq degree_prod_eq_sum_degree
card_insert_if Q_def)
    ultimately show "poly.coeff P 3 = poly.coeff Q 3"
      by simp
  next
    show "degree P ≤ 3"
      by (rule degree_le) (auto simp: P_def eval_nat_numeral Suc_less_eq2
coeff_pCons')
  next
    show "degree Q ≤ 3"
      by (auto simp: degree_mult_eq degree_prod_eq_sum_degree card_insert_if
Q_def)
  next
    show "3 ≤ card (℘ ‘ {u1', u2', u3})"
    proof -
      have "3 ≤ card {u1', u2', u3}"
        using u3 by (auto simp: card_insert_if)
      also have "... = card (℘ ‘ {u1', u2', u3})"
        using inj by (rule card_image [symmetric])
      finally show ?thesis .
    qed
  next
    show "poly P z = poly Q z"
      if "z ∈ ℘ ‘ {u1', u2', u3}" for z
    proof -
      from that obtain u where [simp]: "z = ℘ u" and u: "u ∈ {u1',
u2', u3}"
        by auto
      have "poly P (℘ u) = 0"
        by (rule P_roots) (use u u3 u3_ab u12 ‹u3 ∉ Λ› ab in ‹auto simp:
u1'_def u2'_def›)
      moreover have "poly Q (℘ u) = 0"
        using u by (auto simp: poly_prod Q_def)
      ultimately show ?thesis by simp
    qed
  qed
```

All that remains now is to compare the coefficient of X^2 in these polynomials for X^2 and simplify.

```
  have "℘ (u1 + u2) = ℘ (-(u1 + u2))"
    by (simp only: weierstrass_fun.even)
  also have "... = ℘ u3"
    by (rule weierstrass_fun.lattice_cong) (use rel_u3 in ‹auto simp:
rel_sym›)
  also have "... =  - ℘ u1' - ℘ u2' + a ^ 2 / 4"
```

```
  proof -
    have "poly.coeff P 2 = poly.coeff Q 2"
      by (simp only: P_Q)
    also have "poly.coeff P 2 = -a^2"
      by (simp add: P_def power2_eq_square eval_nat_numeral)
    also have "poly.coeff Q 2 = -4 * (℘ u1' + ℘ u2' + ℘ u3)"
      using u3 by (auto simp: Q_def eval_nat_numeral numeral_poly)
    finally show ?thesis
      by (simp add: field_simps)
  qed
  finally show "℘ (u1 + u2) = -℘ u1 - ℘ u2 + ((℘' u1 - ℘' u2) / (℘ u1
- ℘ u2))^2 / 4"
    by (simp add: u1'_def u2'_def a_def)
qed
```

We now use analytic continuation to get rid of the "general position" assumption.

For this purpose, we regard u_1 as fixed and view the left-hand side and the right-hand side as a function of u_2. The set of values u_2 that we have to exclude is sparse, so analytic continuation works.

```
theorem weierstrass_fun_add:
  assumes u12: "u1 ∉ Λ" "u2 ∉ Λ" "¬rel u1 u2" "¬rel u1 (-u2)"
  shows   "℘ (u1 + u2) = -℘ u1 - ℘ u2 + ((℘' u1 - ℘' u2) / (℘ u1 - ℘
u2))^2 / 4"
    (is "?lhs u2 = ?rhs u2")
proof -
  define A where "A = -(Λ ∪ {z. rel u1 z} ∪ {z. rel u1 (-z)})"
  define B where "B = A - {z. u1 + 2 * z ∈ Λ} - {z. 2 * u1 + z ∈ Λ}"

  have A_altdef: "A = UNIV - (Λ ∪ ((+) (-u1) -` Λ) ∪ ((+) u1 -` Λ))"
    by (auto simp: A_def rel_def add_ac diff_in_lattice_commute)
  have B_altdef: "B = A - (λz. 2 * z + u1) -` Λ -  ((+) (2*u1)) -` Λ"
    unfolding B_def A_altdef by (auto simp: A_def add_ac)

  show ?thesis
  proof (rule analytic_continuation_open[where f = ?lhs and g = ?rhs])
```

Our set B can be written as the complex plane minus some shifted and scaled copies of the lattice, i.e. an uncountable set minus a countable set. Therefore, B is clearly non-empty.

```
    show "B ≠ {}"
    proof -
      have "B = UNIV - (Λ ∪ (+) u1 ` Λ ∪ (+) (-u1) ` Λ ∪ (λz. 2 * z
+ u1) -` Λ ∪ (+) (-2*u1) ` Λ)"
        unfolding A_altdef B_altdef unfolding image_plus_conv_vimage_plus
by auto
      also have "(λz. 2 * z + u1) -` Λ = (λz. (z - u1) / 2) ` Λ"
      proof safe
```

```
      fix z assume "2 * z + u1 ∈ Λ"
      thus "z ∈ (λz. (z - u1) / 2) ‘ Λ"
        by (intro image_eqI[of _ _ "u1 + 2 * z"]) (auto simp: algebra_simps)
    qed auto
    finally have "B = UNIV - (Λ ∪ (+) u1 ‘ Λ ∪ (+) (-u1) ‘ Λ ∪
                              (λz. (z - u1) / 2) ‘ Λ ∪ (+) (-2*u1)
‘ Λ)" .
    also have "uncountable ..."
      by (intro uncountable_minus_countable countable_Un countable_lattice
                countable_image uncountable_UNIV_complex)
    finally have "uncountable B" .
    thus "B ≠ {}"
      by auto
  qed
next
```

Similarly, A can be written as the complex plane minus some shifted copies of the lattice, i.e. the complement of a sparse set. Clearly, what remains is still connected.

```
  show "connected A"
  proof -
    have "(Λ ∪ (+) u1 ‘ Λ ∪ (+) (-u1) ‘ Λ) sparse_in UNIV"
      unfolding A_altdef by (intro sparse_in_union' sparse_in_translate_UNIV
lattice_sparse)
    also have "Λ ∪ (+) u1 ‘ Λ ∪ (+) (-u1) ‘ Λ = Λ ∪ (+) (-u1) -‘ Λ
∪ (+) u1 -‘ Λ"
      unfolding image_plus_conv_vimage_plus by (simp add: Un_ac)
    finally have "connected (UNIV - ...)"
      by (intro sparse_imp_connected) auto
    also have "... = A"
      by (auto simp: A_altdef)
    finally show "connected A" .
  qed
next
```

A and B can both be written in the form "the complex plane minus continuous deformations of the lattice". Since the lattice is a closed set, A and B are open.

```
  show "open A" unfolding A_altdef
    by (intro open_Diff closed_Un closed_lattice closed_vimage continuous_intros)
auto?
  show "open B" unfolding B_altdef
    by (intro open_Diff <open A> closed_lattice closed_vimage continuous_intros)
auto?
next
```

Finally, we apply the restricted version of the identity we already proved before.

```
    show "?lhs u2 = ?rhs u2" if "u2 ∈ B" for u2
      using weierstrass_fun_add_aux[of u1 u2] u12(1) that by (auto simp:
A_def B_def)
  qed (use u12 in ‹auto simp: A_def B_def intro!: holomorphic_intros simp:
rel_def weierstrass_fun_eq_iff›)
qed

lemma weierstrass_fun_diff:
  assumes u12: "u1 ∉ Λ" "u2 ∉ Λ" "¬rel u1 u2" "¬rel u1 (-u2)"
  shows   "℘ (u1 - u2) = -℘ u1 - ℘ u2 + ((℘' u1 + ℘' u2) / (℘ u1 - ℘
u2))^2 / 4"
proof -
  have "℘ (u1 + (-u2)) = -℘ u1 - ℘ (- u2) + ((℘' u1 + ℘' u2) / (℘ u1
- ℘ (- u2)))^2 / 4"
    by (subst weierstrass_fun_add) (use u12 in ‹auto simp: uminus_in_lattice_iff›)
  thus ?thesis
    by (simp add: weierstrass_fun.even)
qed
```

Using the addition theorem for $\wp(z+w)$ and letting $w \to z$ gives us the duplication theorem: $\wp(2z) = -2\wp(z) + \frac{1}{4}(\wp''(z)/\wp'(z))^2$

This is Exercise 1.9 in Apostol's book.

```
theorem weierstrass_fun_double:
  assumes z: "2 * z ∉ Λ"
  shows   "℘ (2 * z) = -2 * ℘ z + (deriv ℘' z / ℘' z)^2 / 4"
proof (rule tendsto_unique)
  have z': "z ∉ Λ"
    using z by (metis assms add_in_lattice mult_2)

  show "(λw. -℘ z - ℘ w + ((℘' z - ℘' w) / (℘ z - ℘ w))^2 / 4) -z→ ℘
(2 * z)"
  proof (rule Lim_transform_eventually)
    show "(λw. ℘ (z + w)) -z→ ℘ (2 * z)"
      by (rule tendsto_eq_intros refl)+ (use z in auto)
  next
    have "eventually (λw. w ∈ -(Λ ∪ (+) z -‘ Λ ∪ (+) (-z) -‘ (Λ-{0}))
- {z}) (at z)"
      using z z' by (intro eventually_at_in_open open_Compl closed_Un
closed_vimage
                          closed_subset_lattice) (auto intro!: continuous_intros)
    thus "eventually (λw. ℘ (z + w) = -℘ z - ℘ w + ((℘' z - ℘' w) / (℘
z - ℘ w))^2 / 4) (at z)"
    proof eventually_elim
      case (elim w)
      show ?case
        by (subst weierstrass_fun_add) (use z' elim in ‹auto simp: rel_def
diff_in_lattice_commute›)
    qed
  qed
```

```
  show "(λw. -℘ z - ℘ w + ((℘' z - ℘' w) / (℘ z - ℘ w))^2 / 4) -z→
        -2 * ℘ z + (deriv ℘' z / ℘' z) ^ 2 / 4"
  proof -
    have *: "(λw. (℘' z - ℘' w) / (℘ z - ℘ w)) -z→ (-deriv ℘' z) /
(-℘' z)"
      by (rule lhopital_complex_simple[OF _ _ _ _ _ refl])
        (use z z' in ‹auto simp: weierstrass_fun_deriv_eq_0_iff weierstrass_fun_ODE2

                          intro!: derivative_eq_intros›)
    have "(λw. -℘ z - ℘ w + ((℘' z - ℘' w) / (℘ z - ℘ w))^2 / 4)
                -z→ -℘ z - ℘ z + ((-deriv ℘' z) / (-℘' z)) ^ 2 / 4"
      by (rule * tendsto_intros z')+ auto
    thus ?thesis
      by simp
  qed
qed auto

end

end
```

9 Eisenstein series and related invariants as modular forms

```
theory Basic_Modular_Forms
  imports Eisenstein_Series Modular_Fundamental_Region
begin
```

In a previous section we defined the Eisenstein series g_k, the modular discriminant Δ, and Klein's invariant J in the context of a fixed complex lattice $\Lambda(\omega_1, \omega 2)$.

In this section, we will look at them for the lattice $\Lambda(1, z)$ with variable $z \in \mathbb{C} \setminus \mathbb{R}$. Since $\Lambda(1, z) = \Lambda(1, -z)$, all of these notions are symmetric with respect to negation of z, so we will often assume $\text{Im}(z) > 0$, as is common in the literature.

We will show that all the above notions satisfy simple functional equations with respect to the modular group, namely if $h(z) = \frac{az+b}{cz+d}$ then $f(h(z)) = (cz + d)^k f(z)$ for some integer k specific to f (the *weight* of f).

Meromorphic functions that satisfy such a functional equation and additionally have a meromorphic Fourier expansion at $q = 0$ (i.e. $z \to i\infty$) are called *meromorphic modular forms*. This notion will be introduced more formally in a future AFP entry, but we already show everything that is required to see that G_k (for $k \geq 3$), Δ, and J are meromorphic modular forms of weight k, 12, and 0, respectively.

9.1 Eisenstein series

First, we look at the Eisenstein series $G_k(z)$, which we define to be the Eisenstein series of the lattice generated by 1 and z. For the case where 1 and z are collinear (i.e. z lying on the real line), we return 0 by convention.

```
definition Eisenstein_G :: "nat ⇒ complex ⇒ complex" where
  "Eisenstein_G k z = (if z ∈ ℝ then 0 else complex_lattice.eisenstein_series
1 z k)"

lemma (in complex_lattice) eisenstein_series_eq_Eisenstein_G:
  "eisenstein_series k = Eisenstein_G k (ω2 / ω1) / ω1 ^ k"
proof -
  interpret complex_lattice_stretch ω1 ω2 "1 / ω1"
    by standard auto
  have "stretched.eisenstein_series k = ω1 ^ k * eisenstein_series k"
    unfolding eisenstein_series_stretch by (simp add: power_int_minus
field_simps)
  also have "stretched.eisenstein_series k = Eisenstein_G k (ω2 / ω1)"
    using stretched.fundpair unfolding Eisenstein_G_def fundpair_def by
simp
  finally show ?thesis
    by simp
qed

lemma Eisenstein_G_real_eq_0 [simp]: "z ∈ ℝ ⟹ Eisenstein_G k z = 0"
  by (simp add: Eisenstein_G_def)

lemma Eisenstein_G_0 [simp]:
  assumes [simp]: "z ∉ ℝ"
  shows   "Eisenstein_G 0 z = 1"
proof -
  interpret complex_lattice 1 z
    by unfold_locales (auto simp: fundpair_def)
  show ?thesis
    by (auto simp: Eisenstein_G_def)
qed

lemma Eisenstein_G_cnj: "Eisenstein_G k (cnj z) = cnj (Eisenstein_G k
z)"
proof (cases "z ∈ ℝ")
  case False
  interpret complex_lattice 1 z
    using False by unfold_locales (auto simp: fundpair_def)
  interpret complex_lattice_cnj 1 z ..
  show ?thesis
    using eisenstein_series_cnj[of k] eisenstein_series_eq_Eisenstein_G[of
k]
          cnj.eisenstein_series_eq_Eisenstein_G[of k] by simp
qed auto
```

```
lemma Eisenstein_G_odd [simp]:
  assumes "odd k"
  shows   "Eisenstein_G k z = 0"
proof (cases "z ∈ ℝ")
  case [simp]: False
  interpret complex_lattice 1 z
    by unfold_locales (auto simp: fundpair_def)
  show ?thesis using assms
    by (auto simp: Eisenstein_G_def)
qed auto

lemma Eisenstein_G_uminus: "Eisenstein_G k (-z) = Eisenstein_G k z"
proof (cases "z ∈ ℝ")
  case False
  interpret lattice1: complex_lattice 1 z
    by standard (use False in ‹auto simp: fundpair_def›)
  interpret lattice2: complex_lattice 1 "-z"
    by standard (use False in ‹auto simp: fundpair_def›)
  have "lattice1.eisenstein_series k = lattice2.eisenstein_series k"
    unfolding lattice1.eisenstein_series_def lattice2.eisenstein_series_def
    by (auto simp: lattice1.of_ω12_coords_def lattice2.of_ω12_coords_def
             intro!: infsum_reindex_bij_witness[of "-{0}" uminus uminus])
  thus ?thesis
    by (simp add: Eisenstein_G_def)
qed (auto simp: Eisenstein_G_def)

lemma
  assumes "k ≥ 3" "(z::complex) ∉ ℝ"
  shows   abs_summable_Eisenstein_G:
            "(λ(m,n). 1 / norm (of_int m + of_int n * z) ^ k) summable_on
(-{(0,0)})"
    and   summable_Eisenstein_G:
            "(λ(m,n). 1 / (of_int m + of_int n * z) ^ k) summable_on (-{(0,0)})"
    and   has_sum_Eisenstein_G:
            "((λ(m,n). 1 / (of_int m + of_int n * z) ^ k) has_sum Eisenstein_G
k z) (-{(0,0)})"
proof -
  from assms interpret complex_lattice 1 z
    by unfold_locales (auto simp: fundpair_def)
  have "(λω. 1 / norm ω ^ k) summable_on lattice0"
    by (rule eisenstein_series_norm_summable) (use assms in auto)
  also have "?this ⟷ (λ(m,n). 1 / norm (of_int m + of_int n * z) ^
k) summable_on (-{(0,0)})"
    by (subst summable_on_reindex_bij_betw[OF bij_betw_lattice0', symmetric])
       (simp_all add: of_ω12_coords_def case_prod_unfold)
  finally show "(λ(m,n). 1 / norm (of_int m + of_int n * z) ^ k) summable_on
(-{(0,0)})" .
```

```
  have "((λω. 1 / ω ^ k) has_sum G k) lattice0"
    by (rule eisenstein_series_has_sum) (use assms in auto)
  also have "?this ⟷ ((λ(m,n). 1 / (of_int m + of_int n * z) ^ k) has_sum

                          eisenstein_series k) (-{(0,0)})"
    by (subst has_sum_reindex_bij_betw[OF bij_betw_lattice0', symmetric])
       (simp_all add: of_ω12_coords_def case_prod_unfold)
  finally show "((λ(m,n). 1 / (of_int m + of_int n * z) ^ k) has_sum Eisenstein_G
k z) (-{(0,0)})"
    unfolding eisenstein_series_eq_Eisenstein_G by simp
  thus "(λ(m,n). 1 / (of_int m + of_int n * z) ^ k) summable_on (-{(0,0)})"
    by (rule has_sum_imp_summable)
qed

lemma Eisenstein_G_analytic [analytic_intros]:
  assumes "f analytic_on A" "⋀z. z ∈ A ⟹ odd k ∨ f z ∉ ℝ"
  shows   "(λz. Eisenstein_G k (f z)) analytic_on A"
proof (cases "k = 0 ∨ odd k")
  case True
  thus ?thesis
  proof
    assume [simp]: "k = 0"
    have "(λ_. 1) holomorphic_on -ℝ"
      by (rule holomorphic_intros)
    also have "?this ⟷ Eisenstein_G k holomorphic_on -ℝ"
      by (rule holomorphic_cong) auto
    finally have ana: "Eisenstein_G k analytic_on -ℝ"
      by (subst analytic_on_open) (auto intro!: closed_complex_Reals)
    have "(Eisenstein_G k ∘ f) analytic_on A"
      by (rule analytic_on_compose_gen[OF _ ana]) (use assms in auto)
    thus ?thesis
      by (simp add: o_def)
  qed auto
next
  case False
  hence k: "k ≥ 2" "even k"
    by auto
  define g :: "complex ⇒ complex" where
    "g = (λz. 2 * (zeta k + (2 * i * pi) ^ k / fact (k - 1) *
                lambert (λn. of_nat n ^ (k-1)) (to_q 1 z)))"

  have "g holomorphic_on {z. Im z > 0}"
    unfolding g_def by (auto intro!: holomorphic_intros simp: lambert_conv_radius_power_of_
  also have "?this ⟷ Eisenstein_G k holomorphic_on {z. Im z > 0}"
  proof (intro holomorphic_cong refl)
    fix z assume z: "z ∈ {z. Im z > 0}"
    interpret std_complex_lattice z
      by standard (use z in auto)
    show "g z = Eisenstein_G k z"
```

```
    unfolding g_def using eisenstein_series_conv_lambert[of k] k z
    by (simp add: Eisenstein_G_def complex_is_Real_iff)
  qed
  finally have "Eisenstein_G k holomorphic_on {z. 0 < Im z}" .
  hence ana: "Eisenstein_G k analytic_on {z. 0 < Im z}"
    by (subst analytic_on_open) (simp_all add: open_halfspace_Im_gt)

  have "(Eisenstein_G k ∘ uminus) analytic_on {z. Im z < 0}"
    by (rule analytic_on_compose_gen[OF _ ana]) (auto intro!: analytic_intros)
  also have "Eisenstein_G k ∘ uminus = Eisenstein_G k"
    by (auto simp: Eisenstein_G_uminus)
  finally have "Eisenstein_G k analytic_on ({z. 0 < Im z} ∪ {z. Im z <
0})"
    by (subst analytic_on_Un) (use ana in auto)
  also have "... = -ℝ"
    by (auto simp: complex_is_Real_iff)
  finally have ana2: "Eisenstein_G k analytic_on -ℝ" .

  have "(Eisenstein_G k ∘ f) analytic_on A"
    by (rule analytic_on_compose_gen[OF _ ana2]) (use assms False in auto)
  thus ?thesis
    by (simp add: o_def)
qed

lemma Eisenstein_G_holomorphic [holomorphic_intros]:
  assumes "f holomorphic_on A" "⋀z. z ∈ A ⟹ odd k ∨ f z ∉ ℝ"
  shows   "(λz. Eisenstein_G k (f z)) holomorphic_on A"
proof -
  from assms(1) have "(Eisenstein_G k ∘ f) holomorphic_on A"
    by (rule holomorphic_on_compose)
       (use assms in ‹auto intro!: analytic_imp_holomorphic analytic_intros›)
  thus ?thesis
    by (simp add: o_def)
qed

lemma Eisenstein_G_meromorphic [meromorphic_intros]:
  assumes "f analytic_on A" "⋀z. z ∈ A ⟹ odd k ∨ f z ∉ ℝ"
  shows   "(λz. Eisenstein_G k (f z)) meromorphic_on A"
  by (rule meromorphic_on_compose[OF _ assms(1) order.refl])
     (use assms(2) in ‹auto intro!: analytic_intros analytic_on_imp_meromorphic_on›)
```

We can also lift our earlier results about the Fourier expansion of G_k to this new viewpoint of $G_k(z)$. This is Theorem 1.18 in Apostol's book.

```
theorem Eisenstein_G_fourier_expansion:
  fixes z :: complex and k :: nat
  assumes z: "Im z > 0"
  assumes k: "k ≥ 2" "even k"
  shows "Eisenstein_G k z =
           2 * (zeta k + (2*i*pi) ^ k / fact (k - 1) * lambert (λn. of_nat
```

```
n ^ (k - 1)) (to_q 1 z))"
proof -
  interpret std_complex_lattice z
    by standard fact
  have "Eisenstein_G k z = eisenstein_series k"
    using eisenstein_series_eq_Eisenstein_G[of k] by simp
  also have "... = 2 * (zeta k + (2*i*pi) ^ k / fact (k - 1) *
                  lambert (λn. of_nat n ^ (k - 1)) (to_q 1 z))"
    by (rule eisenstein_series_conv_lambert[OF k])
  finally show ?thesis .
qed
```

We show how the modular group acts on G_k. The case $k = 2$ is more complicated and will be dealt with later.

```
theorem Eisenstein_G_apply_modgrp:
  assumes "k ≠ 2"
  shows   "Eisenstein_G k (apply_modgrp f z) = modgrp_factor f z ^ k *
Eisenstein_G k z"
proof (cases "z ∈ ℝ")
  case False
  interpret complex_lattice 1 z
    by standard (use False in ‹auto simp: fundpair_def›)
  interpret complex_lattice_apply_modgrp 1 z f ..

  have "Eisenstein_G k (apply_modgrp f z) = Eisenstein_G k (ω2' / ω1')"
    unfolding moebius_def modgrp.φ_def ω1'_def ω2'_def
    by (simp add: apply_modgrp_altdef moebius_def)
  also have "... = modgrp_factor f z ^ k * transformed.eisenstein_series
k"
    by (subst transformed.eisenstein_series_eq_Eisenstein_G)
       (auto simp: ω1'_def power_int_minus field_simps modgrp_factor_def)
  also have "... = modgrp_factor f z ^ k * eisenstein_series k"
    using assms by simp
  also have "... = modgrp_factor f z ^ k * Eisenstein_G k z"
    by (subst eisenstein_series_eq_Eisenstein_G) auto
  finally show ?thesis .
qed auto

lemma Eisenstein_G_plus_int: "Eisenstein_G k (z + of_int m) = Eisenstein_G
k z"
proof (cases "k = 2")
  case False
  thus ?thesis
    using Eisenstein_G_apply_modgrp[of k "shift_modgrp m" z] by simp
next
  case [simp]: True
  have *: "Eisenstein_G 2 (z + of_int m) = Eisenstein_G 2 z" if z: "Im
z > 0" for z m
    using z by (simp add: Eisenstein_G_fourier_expansion to_q_add)
```

```
  show ?thesis
  proof (cases "Im z" "0 :: real" rule: linorder_cases)
    case greater
    thus ?thesis by (simp add: *)
  next
    case equal
    hence "z ∈ ℝ"
      by (auto simp: complex_is_Real_iff)
    thus ?thesis
      by simp
  next
    case less
    have "Eisenstein_G k (z + of_int m) = Eisenstein_G 2 (-(z + of_int
m))"
      by (subst Eisenstein_G_uminus) auto
    also have "... = Eisenstein_G 2 (-z + of_int (-m))"
      by simp
    also have "... = Eisenstein_G 2 (-z)"
      using less by (intro *) auto
    also have "... = Eisenstein_G k z"
      by (simp add: Eisenstein_G_uminus)
    finally show ?thesis .
  qed
qed
```

9.2 The normalised Eisenstein series

The literature also often defines the *normalised* Eisenstein series E_k, which is G_k divided by the constant $2\zeta(k)$. This leads to the somewhat nicer Fourier expansion

$$E_k(z) = 1 - \frac{2k}{B_k} \sum_{n=1}^{\infty} \sigma_{k-1}(n) q^n \ .$$

```
definition Eisenstein_E :: "nat ⇒ complex ⇒ complex" where
  "Eisenstein_E k z = (if k = 0 then if z ∈ ℝ then 0 else 1 else Eisenstein_G
k z / (2 * zeta k))"

lemma Eisenstein_E_fourier:
  assumes "Im z > 0" "k ≥ 2" "even k"
  shows   "Eisenstein_E k z = 1 - 2 * k / bernoulli k * lambert (λn.
n^(k-1)) (to_q 1 z)"
proof -
  obtain l where l: "k = 2 * l" "l > 0"
    using assms by (elim evenE) auto
  have "Eisenstein_E k z = 1 + (2 * i * pi) ^ k / (fact (k - 1) * zeta
k) * lambert (λn. n^(k-1)) (to_q 1 z)"
    using assms unfolding Eisenstein_E_def
```

```
    by (subst Eisenstein_G_fourier_expansion)
       (auto simp: add_divide_distrib zeta_Re_gt_1_nonzero)
  also have "fact (k - 1) = fact k / complex_of_nat k"
    using assms by (subst fact_reduce) auto
  also have "(2 * i * pi) ^ k / (... * zeta k) = -2 * k / bernoulli k"
    using l(2) by (auto simp: l zeta_even_nat power_mult_distrib)
  finally show ?thesis
    by simp
qed

lemma Eisenstein_E_0 [simp]: "z ∉ ℝ ⟹ Eisenstein_E 0 z = 1"
  by (simp add: Eisenstein_E_def)

lemma Eisenstein_E_real_eq_0 [simp]: "z ∈ ℝ ⟹ Eisenstein_E k z = 0"
  by (simp add: Eisenstein_E_def)

lemma Eisenstein_E_cnj: "Eisenstein_E k (cnj z) = cnj (Eisenstein_E k
z)"
  by (simp add: Eisenstein_E_def Eisenstein_G_cnj flip: zeta_cnj)

lemma Eisenstein_E_odd [simp]:
  assumes "odd k"
  shows   "Eisenstein_E k z = 0"
  using assms by (auto simp: Eisenstein_E_def elim!: oddE)

lemma Eisenstein_E_uminus: "Eisenstein_E k (-z) = Eisenstein_E k z"
  by (simp add: Eisenstein_E_def Eisenstein_G_uminus)

lemma Eisenstein_E_analytic [analytic_intros]:
  assumes "f analytic_on A" "⋀z. z ∈ A ⟹ odd k ∨ f z ∉ ℝ"
  shows   "(λz. Eisenstein_E k (f z)) analytic_on A"
proof -
  consider "k = 0" | "k = 1" | "k ≥ 2"
    by linarith
  thus ?thesis
  proof cases
    assume [simp]: "k = 0"
    have "(λz. Eisenstein_G 0 (f z)) analytic_on A"
      using assms by (auto intro!: analytic_intros)
    also have "(λz. Eisenstein_G 0 (f z)) = (λz. Eisenstein_E 0 (f z))"
      by (auto simp: Eisenstein_E_def)
    finally show ?thesis
      by simp
  next
    assume "k ≥ 2"
    thus ?thesis
      unfolding Eisenstein_E_def using assms
      by (auto intro!: analytic_intros simp: zeta_Re_gt_1_nonzero)
  qed (auto simp: Eisenstein_E_def)
```

qed

lemma *Eisenstein_E_holomorphic [holomorphic_intros]:*
assumes *"f holomorphic_on A"* *"⋀z. z ∈ A ⟹ odd k ∨ f z ∉ ℝ"*
shows *"(λz. Eisenstein_E k (f z)) holomorphic_on A"*
proof -
from *assms(1)* **have** *"(Eisenstein_E k ∘ f) holomorphic_on A"*
by *(rule holomorphic_on_compose)*
(use assms **in** *‹auto intro!: analytic_imp_holomorphic analytic_intros›)*
thus *?thesis*
by *(simp add: o_def)*
qed

lemma *Eisenstein_E_meromorphic [meromorphic_intros]:*
assumes *"f analytic_on A"* *"⋀z. z ∈ A ⟹ odd k ∨ f z ∉ ℝ"*
shows *"(λz. Eisenstein_E k (f z)) meromorphic_on A"*
by *(rule meromorphic_on_compose[OF _ assms(1) order.refl])*
(use assms(2) **in** *‹auto intro!: analytic_intros analytic_on_imp_meromorphic_on›)*

theorem *Eisenstein_E_apply_modgrp:*
assumes *"k ≠ 2"*
shows *"Eisenstein_E k (apply_modgrp f z) = modgrp_factor f z ^ k * Eisenstein_E k z"*
unfolding *Eisenstein_E_def* **by** *(subst Eisenstein_G_apply_modgrp) (use assms* **in** *auto)*

9.3 The modular discriminant

definition *modular_discr :: "complex ⇒ complex"* **where**
*"modular_discr z = (60 * Eisenstein_G 4 z) ^ 3 - 27 * (140 * Eisenstein_G 6 z) ^ 2"*

lemma (**in** *complex_lattice) discr_eq_modular_discr: "discr = modular_discr (ω2 / ω1) / ω1 ^ 12"*
unfolding *discr_def modular_discr_def invariant_g2_def invariant_g3_def*
eisenstein_series_eq_Eisenstein_G
by *(simp add: field_simps)*

lemma *modular_discr_real_eq_0 [simp]: "z ∈ ℝ ⟹ modular_discr z = 0"*
by *(simp add: modular_discr_def)*

lemma *modular_discr_cnj: "modular_discr (cnj z) = cnj (modular_discr z)"*
by *(simp add: modular_discr_def Eisenstein_G_cnj)*

lemma *modular_discr_analytic [analytic_intros]:*
assumes *"f analytic_on A"* *"⋀z. z ∈ A ⟹ f z ∉ ℝ"*
shows *"(λz. modular_discr (f z)) analytic_on A"*
unfolding *modular_discr_def* **using** *assms* **by** *(auto intro!: analytic_intros)*

```
lemma modular_discr_holomorphic [holomorphic_intros]:
  assumes "f holomorphic_on A" "⋀z. z ∈ A ⟹ f z ∉ ℝ"
  shows   "(λz. modular_discr (f z)) holomorphic_on A"
  unfolding modular_discr_def using assms by (auto intro!: holomorphic_intros)

lemma modular_discr_uminus: "modular_discr (-z) = modular_discr z"
  by (simp add: modular_discr_def Eisenstein_G_uminus)

lemma modular_discr_nonzero:
  assumes "z ∉ ℝ"
  shows   "modular_discr z ≠ 0"
proof -
  interpret complex_lattice 1 z
    by standard (use assms in ‹auto simp: fundpair_def›)
  have "modular_discr z = discr"
    by (simp add: discr_eq_modular_discr)
  also have "... ≠ 0"
    by (rule discr_nonzero)
  finally show ?thesis .
qed

lemma modular_discr_eq_0_iff: "modular_discr z = 0 ⟷ z ∈ ℝ"
  using modular_discr_nonzero[of z] by auto

theorem modular_discr_apply_modgrp:
  "modular_discr (apply_modgrp f z) = modgrp_factor f z ^ 12 * modular_discr
z"
  by (simp add: modular_discr_def Eisenstein_G_apply_modgrp algebra_simps)

lemma modular_discr_plus_int: "modular_discr (z + of_int m) = modular_discr
z"
  using modular_discr_apply_modgrp[of "shift_modgrp m" z] by simp

lemma modular_discr_minus_one_over: "modular_discr (-(1/z)) = z ^ 12
* modular_discr z"
  using modular_discr_apply_modgrp[of "S_modgrp" z] by simp
```

9.4 Klein's J invariant

```
definition Klein_J :: "complex ⇒ complex" where
  "Klein_J z = (60 * Eisenstein_G 4 z) ^ 3 / modular_discr z"

lemma (in complex_lattice) invariant_j_eq_Klein_J:
  "invariant_j = Klein_J (ω2 / ω1)"
  unfolding invariant_j_def discr_eq_modular_discr Klein_J_def invariant_g2_def
            eisenstein_series_eq_Eisenstein_G by (simp add: field_simps)

lemma Klein_J_real_eq_0 [simp]: "z ∈ ℝ ⟹ Klein_J z = 0"
```

```
  by (simp add: Klein_J_def)

lemma Klein_J_uminus: "Klein_J (-z) = Klein_J z"
  by (simp add: Klein_J_def modular_discr_uminus Eisenstein_G_uminus)

lemma Klein_J_cnj: "Klein_J (cnj z) = cnj (Klein_J z)"
  by (simp add: Klein_J_def Eisenstein_G_cnj modular_discr_cnj)

lemma Klein_J_analytic [analytic_intros]:
  assumes "f analytic_on A" "⋀z. z ∈ A ⟹ f z ∉ ℝ"
  shows   "(λz. Klein_J (f z)) analytic_on A"
  unfolding Klein_J_def using assms by (auto intro!: analytic_intros
simp: modular_discr_nonzero)

lemma Klein_J_holomorphic [holomorphic_intros]:
  assumes "f holomorphic_on A" "⋀z. z ∈ A ⟹ f z ∉ ℝ"
  shows   "(λz. Klein_J (f z)) holomorphic_on A"
  unfolding Klein_J_def using assms by (auto intro!: holomorphic_intros
simp: modular_discr_nonzero)
```

It is trivial to show that Klein's J function is invariant under the modular group. This is Apostol's Theorem 1.16.

```
theorem Klein_J_apply_modgrp:
  "Klein_J (apply_modgrp f z) = Klein_J z"
proof (cases "z ∈ ℝ")
  case False
  thus ?thesis
    by (simp add: Klein_J_def Eisenstein_G_apply_modgrp modular_discr_apply_modgrp
algebra_simps
                  complex_is_Real_iff)
qed auto

lemma Klein_J_plus_int: "Klein_J (z + of_int m) = Klein_J z"
  using Klein_J_apply_modgrp[of "shift_modgrp m" z] by simp

lemma Klein_J_minus_one_over: "Klein_J (-(1/z)) = Klein_J z"
  using Klein_J_apply_modgrp[of "S_modgrp" z] by simp

lemma Klein_J_cong:
  assumes "w ∼Γ z"
  shows   "Klein_J w = Klein_J z"
  using assms by (metis Klein_J_apply_modgrp modular_group.rel_def)
```

9.5 Values at specific points

```
unbundle modfun_region_notation
```

Let $k \geq 2$. The points i and ρ are fixed points of the modular transformations S and ST, respectively. Using this together with the modular transformation

law for G_k, it directly follows that $G_k(i) = 0$ unless k is a multiple of 4 and $G_k(\rho) = 0$ unless k is a multiple of 6.

These facts are part of Apostol's Exercise 1.12 and generalise some facts derived in his Theorem 2.7.

The values $G_2(i) = \pi$ and $G_2(\rho) = \frac{2\pi}{\sqrt{3}}$ can be determined in the same fashion once we have proven the transformation law for G_2.

```
lemma Eisenstein_G_ii_eq_0:
  assumes "k ≠ 2" "¬4 dvd k"
  shows   "Eisenstein_G k i = 0"
proof (cases "even k")
  case True
  thus ?thesis
    using Eisenstein_G_apply_modgrp[of k S_modgrp "i"] assms
    by (auto elim!: evenE simp: power_neg_one_If split: if_splits)
qed auto

lemma Eisenstein_G_6_ii [simp]: "Eisenstein_G 6 i = 0"
  by (rule Eisenstein_G_ii_eq_0) auto

lemma Eisenstein_G_rho_eq_0:
  assumes "k ≠ 2" "¬6 dvd k"
  shows   "Eisenstein_G k ϱ = 0"
proof (cases "even k")
  case True
  show ?thesis
  proof (rule ccontr)
    assume nz: "Eisenstein_G k ϱ ≠ 0"
    have "Eisenstein_G k (- (1 / (ϱ + 1))) = (ϱ + 1) ^ k * Eisenstein_G
k ϱ"
      using Eisenstein_G_apply_modgrp[of k "S_modgrp * T_modgrp" "ϱ"]
assms
      by (auto elim!: evenE simp: power_neg_one_If apply_modgrp_mult split:
if_splits)
    also have "-(1 / (ϱ + 1)) = ϱ"
      by (auto simp: field_simps modfun_rho_altdef complex_eq_iff Re_divide
Im_divide)
    finally have "(ϱ + 1) ^ k = 1"
      using nz by simp
    also have "ϱ + 1 = -1 / ϱ"
      by (auto simp: complex_eq_iff modfun_rho_altdef field_simps Re_divide
Im_divide)
    finally have "ϱ ^ k = 1"
      using True by (auto simp: field_simps uminus_power_if)
    hence "3 dvd k"
      by (simp add: modfun_rho_power_eq_1_iff)
    with True have "6 dvd k"
      by presburger
    thus False
```

```
    using assms by simp
  qed
qed auto

lemma Eisenstein_G_4_rho [simp]: "Eisenstein_G 4 ϱ = 0"
  by (rule Eisenstein_G_rho_eq_0) auto


corollary Eisenstein_G_6_rho_nonzero: "Eisenstein_G 6 ϱ ≠ 0"
proof -
  have "modular_discr ϱ ≠ 0"
    by (rule modular_discr_nonzero) auto
  thus ?thesis
    by (auto simp: modular_discr_def)
qed

corollary Eisenstein_G_4_ii_nonzero: "Eisenstein_G 4 i ≠ 0"
proof -
  have "modular_discr i ≠ 0"
    by (rule modular_discr_nonzero) auto
  thus ?thesis
    by (auto simp: modular_discr_def)
qed


corollary Klein_J_rho [simp]: "Klein_J ϱ = 0"
  by (simp add: Klein_J_def)

corollary Klein_J_ii [simp]: "Klein_J i = 1"
  using modular_discr_nonzero[of i]
  by (simp add: Klein_J_def modular_discr_def complex_is_Real_iff)
```

9.6 Consequences for the fundamental region

One immediate consequence of the fact that $J(\rho) = 0$ and $J(i) = 1$ is that ρ and i are not equivalent w.r.t. the modular group.

```
lemma not_rho_equiv_i [simp]: "¬(ϱ ∼Γ i)"
proof
  assume "ϱ ∼Γ i"
  hence "Klein_J ϱ = Klein_J i"
    by (rule Klein_J_cong)
  thus False
    by simp
qed

lemma not_i_equiv_rho [simp]: "¬(i ∼Γ ϱ)"
  by (subst modular_group.rel_commutes) simp

lemma not_modular_group_rel_rho_i [simp]: " z ∼Γ ϱ ⟹ ¬z ∼Γ i"
```

```
  by (meson modular_group.rel_sym modular_group.rel_trans not_i_equiv_rho)

lemma modular_group_rel_rho_i_cases [case_names rho i neither invalid]:
  obtains "z ∼Γ ϱ" "¬z ∼Γ i" | "z ∼Γ i" "¬z ∼Γ ϱ" | "Im z > 0" "¬z
∼Γ ϱ" "¬z ∼Γ i" | "Im z ≤ 0"
  by (cases "Im z > 0"; cases "z ∼Γ ϱ"; cases "z ∼Γ i") auto
```

Another application of the Klein J function: We can show that subgroups of the modular group have discrete orbits. That is: every point has a neighbourhood in which no equivalent points are.

```
lemma (in modgrp_subgroup) isolated_in_orbit:
  assumes "Im y > 0"
  shows   "¬y islimpt orbit x"
proof
  assume *: "y islimpt orbit x"
  have "Klein_J z - Klein_J x = 0" if z: "Im z > 0" for z
  proof (rule analytic_continuation[of "λz. Klein_J z - Klein_J x"])
    show "(λz. Klein_J z - Klein_J x) holomorphic_on {z. Im z > 0}"
      by (auto intro!: holomorphic_intros simp: complex_is_Real_iff)
    show "open {z. Im z > 0}" and "connected {z. Im z > 0}"
      by (auto simp: open_halfspace_Im_gt intro!: convex_connected convex_halfspace_Im_gt)
    show "y islimpt orbit x" by fact
    show "Klein_J z - Klein_J x = 0" if "z ∈ orbit x" for z
    proof -
      have "z ∼Γ x"
        using that by (auto simp: orbit_def rel_commutes intro: rel_imp_rel)
      then obtain g where g: "apply_modgrp g z = x" "Im z > 0" "Im x
> 0"
        by (auto simp: modular_group.rel_def)
      show ?thesis
        using g(2,3) by (simp add: Klein_J_apply_modgrp flip: g(1))
    qed
  qed (use assms z in ‹auto simp: orbit_def›)
  from this[of "ϱ"] and this[of i] show False
    by simp
qed

lemma (in modgrp_subgroup) discrete_orbit: "discrete (orbit x)"
  unfolding discrete_def
proof safe
  fix y assume y: "y ∈ orbit x"
  hence "Im y > 0"
    by (simp add: orbit_def rel_imp_Im_pos(2))
  have *: "orbit y = orbit x"
    by (intro orbit_cong) (use y in ‹auto simp: orbit_def rel_commutes›)
  have "y isolated_in orbit y"
    using isolated_in_orbit[of y] y * ‹Im y > 0› isolated_in_islimpt_iff
by blast
  thus "y isolated_in orbit x"
```

```
    by (simp add: *)
qed

lemma (in modgrp_subgroup) eventually_not_rel_at:
  "Im x > 0 ⟹ eventually (λy. ¬rel y x) (at x)"
  using isolated_in_orbit[of x x]
  by (simp add: islimpt_conv_frequently_at frequently_def orbit_def rel_commutes)

lemma (in modgrp_subgroup) closed_orbit [intro]: "closedin (top_of_set
{z. Im z > 0}) (orbit x)"
  using isolated_in_orbit[of _ x] by (auto simp: closedin_limpt orbit_imp_Im_pos)

unbundle no modfun_region_notation

end
```

10 Related facts about Jacobi theta functions

```
theory Theta_Inversion
imports
  "Theta_Functions.Jacobi_Triple_Product"
  "Theta_Functions.Theta_Nullwert"
  "Polylog.Polylog_Library"
  Complex_Lattices
begin


lemmas [simp del] = div_mult_self1 div_mult_self2 div_mult_self3 div_mult_self4
```

In this section we will re-use some of the lemmas we proved to study elliptic functions in order to show two non-trivial facts about the Jacobi theta functions. The first one is a uniqueness result:

We know that $\vartheta_{00}(z,t)$, viewed as a function of z for fixed t, is entire, periodic with period 1, and quasi-periodic with period t and factor e^{-2z-t}. We will show that for any fixed t in the upper half plane, $\vartheta_{00}(\cdot,t)$ is actually uniquely defined by these relations, up to a constant factor.

10.1 Uniqueness of quasi-periodic entire functions

We first show a fairly obvious fact: in any complete real normed field, the separable ordinary differential equation $f'(x) = g(x)f(x)$ has at most one solution up to a constant factor in any complex domain.

If a non-vanishing function G satisfies $G'(x) = g(x)G(X)$, then G is that solution. This allows us to "certify" a solution easily.

```
lemma separable_ODE_simple_unique:
  fixes f :: "'a :: {banach, real_normed_field} ⇒ 'a"
```

```
  assumes eq: "⋀x. x ∈ A ⟹ f' x = g x * f x"
  assumes deriv_f: "⋀x. x ∈ A ⟹ (f has_field_derivative f' x) (at
x within A)"
  assumes deriv_g: "⋀x. x ∈ A ⟹ (G has_field_derivative (g x * G x))
(at x within A)"
  assumes nonzero [simp]: "⋀x. x ∈ A ⟹ G x ≠ 0"
  assumes "convex A"
  shows "∃c. ∀x∈A. f x = c * G x"
proof -
  have "∃c. ∀x∈A. f x / G x = c"
  proof (rule has_field_derivative_zero_constant)
    show "((λx. f x / G x) has_field_derivative 0) (at x within A)" if
x: "x ∈ A" for x
      using x by (auto intro!: derivative_eq_intros deriv_f deriv_g simp:
eq)
  qed fact
  then obtain c where c: "⋀x. x ∈ A ⟹ f x / G x = c"
    by blast
  thus ?thesis
    by (auto simp: field_simps)
qed
```

The following locale captures the notion of an entire function in the complex plane that satisfies the same (quasi-)periodicity as the Jacobi theta function ϑ_{00}, namely $f(z+1) = f(z)$ and $f(z+t) = e^{-2z-t}f(z)$ for some fixed t with $\mathrm{Im}(t) > 0$.

We will show that any such function is equal to ϑ_{00} up to a constant factor.

```
locale thetalike_function =
  fixes f :: "complex ⇒ complex" and t :: complex
  assumes entire: "f holomorphic_on UNIV"
  assumes Im_t: "Im t > 0"
  assumes f_plus_1: "f (z + 1) = f z"
  assumes f_plus_quasiperiod: "f (z + t) = f z / to_nome (2*z+t)"
begin

lemma holomorphic:
  assumes "g holomorphic_on A"
  shows   "(λx. f (g x)) holomorphic_on A"
proof -
  have "(f ∘ g) holomorphic_on A"
    using assms entire by (rule holomorphic_on_compose_gen) auto
  thus ?thesis
    by (simp add: o_def)
qed

lemma analytic:
  assumes "g analytic_on A"
  shows   "(λx. f (g x)) analytic_on A"
proof -
```

```
  have *: "f analytic_on UNIV"
    by (subst analytic_on_open) (auto intro!: holomorphic)
  have "(f ∘ g) analytic_on A"
    using assms * by (rule analytic_on_compose_gen) auto
  thus ?thesis
    by (simp add: o_def)
qed
```

We first show some straightforward facts about the behaviour of f on the lattice generated by 1 and t.

```
sublocale lattice: std_complex_lattice t
  by standard (use Im_t in ‹auto elim!: Reals_cases›)

lemma f_plus_of_nat: "f (z + of_nat n) = f z"
proof (induction n)
  case (Suc n)
  thus ?case
    using f_plus_1[of "z + of_nat n"] by (simp add: algebra_simps)
qed auto

lemma f_plus_of_int: "f (z + of_int n) = f z"
  using f_plus_of_nat[of z "nat n"] f_plus_of_nat[of "z + of_int n" "nat
(-n)"]
  by (cases "n ≥ 0") (auto simp: algebra_simps)

lemma f_plus_of_nat_quasiperiod:
  "f (z + of_nat n * t) = f z / to_nome (2 * of_nat n * z + of_nat (n^2)
* t)"
proof (induction n)
  case (Suc n)
  thus ?case
    using f_plus_quasiperiod[of "z + of_nat n * t"]
    by (simp add: algebra_simps power2_eq_square flip: to_nome_add)
qed auto

lemma f_plus_of_int_quasiperiod:
  "f (z + of_int n * t) = f z / to_nome (2 * of_int n * z + of_int (n^2)
* t)"
proof (cases "n ≥ 0")
  case True
  thus ?thesis
    using f_plus_of_nat_quasiperiod[of z "nat n"] by (auto simp: sgn_if)
next
  case False
  thus ?thesis
    using f_plus_of_nat_quasiperiod[of "z + of_int n * t" "nat (-n)"]
    by (simp add: field_simps to_nome_add to_nome_diff power2_eq_square
to_nome_minus)
qed
```

```
lemma relE:
  assumes "lattice.rel z z'"
  obtains m n :: int where "z = z' + of_int m + of_int n * t"
  using assms unfolding lattice.rel_def
  by (elim lattice.latticeE) (auto simp: lattice.of_ω12_coords_def algebra_simps)

lemma f_zero_cong_lattice:
  assumes "lattice.rel z z'"
  shows   "f z = 0 ⟷ f z' = 0"
proof -
  interpret lattice: complex_lattice_periodic 1 t "λz. f z = 0"
    by standard (auto simp: f_plus_1 f_plus_quasiperiod)
  show ?thesis
    using lattice.lattice_cong[OF assms] .
qed

lemma zorder_f_cong_lattice:
  assumes "lattice.rel z z'"
  shows   "zorder f z = zorder f z'"
proof -
  from assms obtain m n where mn: "z = z' + of_int m + of_int n * t"
    by (elim relE)
  define h where "h = (λz. z + of_int m + of_int n * t)"
  define g where "g = (λz. to_nome (-(2 * of_int n * (z + of_int m) +
(of_int n)^2 * t)))"
  have "zorder f (h z') = zorder (f ∘ h) z'"
    by (simp add: zorder_shift' h_def algebra_simps)
  also have "... = zorder f z'"
  proof (cases "∀z. f z = 0")
    case True
    hence [simp]: "f = (λ_. 0)"
      by auto
    show ?thesis
      by simp
  next
    case False
    have ev_nz: "eventually (λz. f z ≠ 0) (cosparse UNIV)"
    proof -
      have "(∀x∈UNIV. f x = 0) ∨ (∀≈x∈UNIV. f x ≠ 0)"
        by (intro nicely_meromorphic_imp_constant_or_avoid
                  analytic_on_imp_nicely_meromorphic_on analytic) auto
      moreover have "¬(∀x∈UNIV. f x = 0)"
        using False by auto
      ultimately show ?thesis
        by blast
    qed
    hence ev: "eventually (λw. f w ≠ 0) (at z')"
      by (rule eventually_cosparse_imp_eventually_at) auto
```

```
    have "f ∘ h = (λz. f z * g z)"
      unfolding to_nome_minus g_def
      by (auto simp: h_def fun_eq_iff f_plus_of_int_quasiperiod f_plus_of_int
divide_inverse)
    hence "zorder (f ∘ h) z' = zorder (λz. f z * g z) z'"
      by (rule arg_cong)
    also have "zorder (λz. f z * g z) z' = zorder f z' + zorder g z'"
      by (rule zorder_times_analytic)
         (use ev in ‹auto intro!: analytic_intros analytic simp: g_def›)
    also have "zorder g z' = 0"
      by (rule zorder_eq_0I) (auto simp: g_def intro!: analytic_intros)
    finally show ?thesis
      by (simp add: o_def)
  qed
  finally show ?thesis
    by (simp add: h_def mn o_def)
qed

lemma deriv_f_plus_1: "deriv f (z + 1) = deriv f z"
proof -
  have "deriv f (z + 1) = deriv (λz. f (z + 1)) z"
    by (simp add: deriv_shift_0' o_def add_ac)
  also have "(λz. f (z + 1)) = f"
    by (subst f_plus_1) auto
  finally show ?thesis .
qed

lemma deriv_f_plus_quasiperiod:
  "deriv f (z + t) = (deriv f z - 2 * pi * i * f z) / to_nome (2 * z +
t)"
proof -
  have "deriv f (z + t) = deriv (λz. f (z + t)) z"
    by (simp add: deriv_shift_0' o_def add_ac)
  also have "(λz. f (z + t)) = (λz. f z * to_nome (-2 * z - t))"
    by (subst f_plus_quasiperiod) (auto simp: to_nome_diff to_nome_minus
field_simps to_nome_add)
  also have "deriv ... z = f z * deriv (λz. to_nome (- (2 * z) - t)) z
+ deriv f z * to_nome (- (2 * z) - t)"
    by (subst complex_derivative_mult_at) (auto intro!: analytic_intros
analytic)
  also have "deriv (λz. to_nome (- (2 * z) - t)) z = -2 * pi * i * to_nome
(-(2*z)-t)"
    by (rule DERIV_imp_deriv) (auto intro!: derivative_eq_intros)
  finally show "deriv f (z + t) = (deriv f z - 2 * pi * i * f z) / to_nome
(2 * z + t)"
    by (auto simp: field_simps to_nome_minus to_nome_diff to_nome_add)
qed
```

Next, we will simplify the integral

$$\int_\gamma \frac{h(z)f'(z)}{f(z)}\,\mathrm{d}z$$

for an arbitrary function $h(z)$ using the shift relations for f. Here, γ is a counter-clockwise contour along the border of a period parallelogram with lower left corner b and no zeros of f on it.
We find that:

$$\int_\gamma \frac{h(z)f'(z)}{f(z)}\,\mathrm{d}z = \int_b^{b+1}(h(z)-h(z+t))\frac{f'(z)}{f(z)}+2\pi i h(z+t)\,\mathrm{d}z - \int_b^{b+t}(h(z)-h(z+1))\frac{f'(z)}{f(z)}\,\mathrm{d}z$$

```
lemma argument_principle_f_gen:
  fixes orig :: complex
  defines "γ ≡ parallelogram_path orig 1 t"
  assumes h: "h holomorphic_on UNIV"
  assumes nz: "⋀z. z ∈ path_image γ ⟹ f z ≠ 0"
  shows "contour_integral γ (λx. h x * deriv f x / f x) =
           contour_integral (linepath orig (orig + 1))
             (λz. (h z - h (z + t)) * deriv f z / f z + 2 * pi * i * h
(z + t)) -
           contour_integral (linepath orig (orig + t))
             (λz. (h z - h (z + 1)) * deriv f z / f z)"
proof -
  define h' where "h' = (λx. h (x + orig) * deriv f (x + orig) / f (x
+ orig))"
  have [analytic_intros]: "h analytic_on A" for A
    using h analytic_on_holomorphic by blast

  have "contour_integral γ (λx. h x * deriv f x / f x) =
          contour_integral (linepath 0 1) (λx. h' x - h' (x + t)) -
          contour_integral (linepath 0 t) (λx. h' x - h' (x + 1))" (is
"_ = ?rhs")
    unfolding γ_def
  proof (subst contour_integral_parallelogram_path'; (fold γ_def)?)
    show "continuous_on (path_image γ) (λx. h x * deriv f x / f x)" us-
ing nz
      by (intro continuous_intros holomorphic_on_imp_continuous_on analytic_imp_holomorphic
         (auto intro!: analytic_intros analytic)
  next
    have 1: "linepath orig (orig + 1) = (+) orig ∘ linepath 0 1"
     and 2: "linepath orig (orig + t) = (+) orig ∘ linepath 0 t"
      by (auto simp: linepath_translate add_ac)
    show "contour_integral (linepath orig (orig + 1))
            (λx. h x * deriv f x / f x - h (x + t) * deriv f (x + t) /
f (x + t)) -
          contour_integral (linepath orig (orig + t))
```

```
            (λx. h x * deriv f x / f x - h (x + 1) * deriv f (x + 1) /
f (x + 1)) = ?rhs"
      unfolding 1 2 contour_integral_translate h'_def by (simp add: algebra_simps)
  qed

  also have "(λz. h' z - h' (z + 1)) =
             (λz. (h (z + orig) - h (z + orig + 1)) * deriv f (z + orig)
/ f (z + orig))"
    (is "?lhs = ?rhs")
  proof
    fix z :: complex
    have "h' z - h' (z + 1) =
            h (z + orig) * deriv f (z + orig) / f (z + orig) -
            h (z + orig + 1) * deriv f (z + orig + 1) / f (z + orig +
1)"
      by (simp add: h'_def add_ac)
    also have "... = (h (z + orig) - h (z + orig + 1)) * deriv f (z +
orig) / f (z + orig)"
      unfolding f_plus_1 deriv_f_plus_1 by (simp add: diff_divide_distrib
ring_distribs)
    finally show "?lhs z = ?rhs z" .
  qed
  also have "contour_integral (linepath 0 t) ... =
             contour_integral ((+) orig ∘ linepath 0 t) (λz. (h z - h
(z + 1)) * deriv f z / f z)"
    by (rule contour_integral_translate [symmetric])
  also have "(+) orig ∘ linepath 0 t = linepath orig (orig + t)"
    by (subst linepath_translate) (simp_all add: add_ac)

  also have "contour_integral (linepath 0 1) (λz. h' z - h' (z + t)) =
             contour_integral (linepath 0 1)
               (λz. (h (z + orig) - h (z + orig + t)) * deriv f (z + orig)
/ f (z + orig) +
                    2 * pi * i * h (z + orig + t))"
    (is "contour_integral _ ?lhs = contour_integral _ ?rhs")
  proof (rule contour_integral_cong)
    fix z :: complex
    assume z: "z ∈ path_image (linepath 0 1)"
    hence "orig + z ∈ path_image ((+) orig ∘ linepath 0 1)"
      by (subst path_image_compose) auto
    also have "(+) orig ∘ linepath 0 1 = linepath orig (orig + 1)"
      by (subst linepath_translate) (auto simp: add_ac)
    finally have "orig + z ∈ path_image γ"
      unfolding γ_def parallelogram_path_def by (auto simp: path_image_join)
    hence nz': "f (orig + z) ≠ 0"
      using nz by blast

    have "h' z - h' (z + t) =
            h (z + orig) * deriv f (z + orig) / f (z + orig) -
```

```
          h (z + orig + t) * (deriv f (z + orig + t) / f (z + orig +
t))"
      by (simp add: h'_def add_ac)
    also have "deriv f (z + orig + t) / f (z + orig + t) =
              deriv f (z + orig) / f (z + orig) - 2 * pi * i"
      unfolding f_plus_quasiperiod deriv_f_plus_quasiperiod using nz'
      by (auto simp: divide_simps add_ac)
    also have "h (z + orig) * deriv f (z + orig) / f (z + orig) - h (z
+ orig + t) * ... = ?rhs z"
      by (simp add: ring_distribs diff_divide_distrib)
    finally show "?lhs z = ?rhs z" .
  qed auto
  also have "... = contour_integral ((+) orig ∘ linepath 0 1)
                     (λz. (h z - h (z + t)) * deriv f z / f z + 2 * pi
* i * h (z + t))"
    unfolding contour_integral_translate by (simp add: add_ac)
  also have "(+) orig ∘ linepath 0 1 = linepath orig (orig + 1)"
    by (subst linepath_translate) (simp_all add: add_ac)

  finally show ?thesis .
qed
```

We now instantiate the above fact with $h(z) = 1$ and see that the corresponding integral divided by $2\pi i$ evaluates to 1. This will later tell us that every period parallelogram contains exactly one root of f, and that it is a simple root.

```
lemma argument_principle_f_1:
  fixes orig :: complex
  defines "γ ≡ parallelogram_path orig 1 t"
  assumes nz: "⋀z. z ∈ path_image γ ⟹ f z ≠ 0"
  shows "contour_integral γ (λz. deriv f z / f z) = 2 * pi * i"
  using argument_principle_f_gen[OF _ nz, of "λ_. 1"] by (simp add: γ_def)
```

Next, we instantiate the lemma with $h(z) = z$ and see that the integral divided by $2\pi i$ evaluates to some value of the form $\frac{t+1}{2} + m + nt$ for integers m, n. In other words: it evaluates to some value equivalent to $\frac{t+1}{2}$ modulo our lattice.

This will later tell us that the roots of f in any period parallelogram sum to something equivalent to $\frac{t+1}{2}$. Since we know there is only one root and it is simple, this means that the only root in each period parallelogram is the copy of $\frac{t+1}{2}$ contained in it.

```
lemma argument_principle_f_z:
  fixes orig :: complex
  defines "γ ≡ parallelogram_path orig 1 t"
  assumes nz: "⋀z. z ∈ path_image γ ⟹ f z ≠ 0"
  shows "lattice.rel (contour_integral γ (λz. z * deriv f z / f z) /
(2*pi*i)) ((t+1)/2)"
proof -
```

```
  note [holomorphic_intros del] = holomorphic_deriv
  note [holomorphic_intros] = holomorphic holomorphic_on_subset[OF holomorphic_deriv[of
_ UNIV]]
  define γ1 where "γ1 = linepath orig (orig + 1)"
  define γ2 where "γ2 = linepath orig (orig + t)"
  define γ2' where "γ2' = f ∘ γ2"
  define γ3 where "γ3 = (λz. f orig * to_nome z) ∘ linepath 0 (-2 * orig
- t)"

  have "pathstart γ ∈ path_image γ"
    by blast
  from nz[OF this] have [simp]: "f orig ≠ 0"
    by (simp add: γ_def)

  have [simp, intro]: "valid_path γ1" "valid_path γ2"
    by (simp_all add: γ1_def γ2_def)
  have [simp, intro]: "valid_path γ2'"
    unfolding γ2'_def by (intro valid_path_compose_analytic[of _ _ UNIV]
analytic) auto
  have [simp, intro]: "valid_path γ3"
    unfolding γ3_def by (intro valid_path_compose_analytic[of _ _ UNIV]
analytic_intros) auto

  have [simp]: "pathstart γ2' = f orig" "pathfinish γ2' = f (orig + t)"
    by (simp_all add: γ2'_def γ2_def pathstart_compose pathfinish_compose)
  have [simp]: "pathstart γ3 = f orig" "pathfinish γ3 = f (orig + t)"
    by (simp_all add: γ3_def pathstart_compose pathfinish_compose f_plus_quasiperiod
                      to_nome_diff to_nome_minus to_nome_add field_simps)

  have nz': "f z ≠ 0" if "z ∈ path_image γ1 ∪ path_image γ2" for z
  proof -
    note ‹z ∈ path_image γ1 ∪ path_image γ2›
    also have "path_image γ1 ∪ path_image γ2 ⊆ path_image γ"
      by (auto simp: γ1_def γ2_def γ_def path_image_compose path_image_join

                     parallelogram_path_def image_Un closed_segment_commute
               simp flip: closed_segment_translation)
    finally show ?thesis
      using nz[of z] by blast
  qed

  have [simp]: "0 ∉ path_image γ3"
    by (auto simp: γ3_def path_image_compose)
  have [simp]: "0 ∉ path_image γ2'"
    using nz' unfolding γ2'_def by (auto simp: γ2'_def path_image_compose)
```

The actual proof starts here.

```
  define I1 where "I1 = contour_integral γ1"
  define I2 where "I2 = contour_integral γ2"
```

```
  have "winding_number (f ∘ γ1) 0 ∈ ℤ"
  proof (rule integer_winding_number)
    have "valid_path (f ∘ γ1)"
      by (intro valid_path_compose_analytic[of _ _ UNIV] analytic) auto
    thus "path (f ∘ γ1)"
      by (rule valid_path_imp_path)
  qed (auto simp: path_image_compose nz' pathfinish_def pathstart_def
γ1_def linepath_def f_plus_1)
  then obtain m where m: "winding_number (f ∘ γ1) 0 = of_int m"
    by (elim Ints_cases)

  have "winding_number γ2' 0 + orig + t / 2 ∈ ℤ"
  proof -
    define r where "r = unwinding (i * pi * (- (2 * orig) - t))"
    have "winding_number (γ2' +++ reversepath γ3) 0 ∈ ℤ"
    proof (rule integer_winding_number)
      have "valid_path (γ2' +++ reversepath γ3)"
        by (intro valid_path_join valid_path_compose_analytic[of _ _ UNIV]
analytic)
           (auto simp: pathfinish_compose f_plus_quasiperiod)
      thus "path (γ2' +++ reversepath γ3)"
        by (rule valid_path_imp_path)
    next
      show "pathfinish (γ2' +++ reversepath γ3) =
            pathstart (γ2' +++ reversepath γ3)"
        by (auto simp: pathfinish_compose pathstart_compose γ2_def)
    next
      have "0 ∉ path_image γ2' ∪ path_image γ3"
        by auto
      also have "path_image γ2' ∪ path_image γ3 = path_image (γ2' +++
reversepath γ3)"
        by (subst path_image_join)
           (simp_all add: f_plus_quasiperiod)
      finally show "0 ∉ path_image (γ2' +++ reversepath γ3)" .
    qed

    also have "winding_number (γ2' +++ reversepath γ3) 0 =
               winding_number γ2' 0 + winding_number (reversepath γ3)
0"
      by (rule winding_number_join) (auto simp: valid_path_imp_path)
    also have "winding_number (reversepath γ3) 0 = -winding_number γ3
0"
      by (subst winding_number_reversepath) (auto simp: valid_path_imp_path)
    also have "winding_number γ3 0 = contour_integral γ3 (λw. 1 / w)
/ (2 * pi * i)"
      by (subst winding_number_valid_path) auto
    also have "contour_integral γ3 (λw. 1 / w) =
                 contour_integral (linepath 0 (-2 * orig - t))
```

```
                  (λw. deriv (λz. f orig * to_nome z) w / (f orig * to_nome
w))" unfolding γ3_def
      by (subst contour_integral_comp_analyticW[of _ UNIV]) (auto intro!:
analytic_intros)
    also have "(λw. deriv (λz. f orig * to_nome z) w / (f orig * to_nome
w)) = (λw. pi * i)"
    proof
      fix w :: complex
      have "deriv (λz. f orig * to_nome z) w = pi * i * to_nome w * f
orig"
        by (rule DERIV_imp_deriv) (auto intro!: derivative_eq_intros)
      thus "deriv (λz. f orig * to_nome z) w / (f orig * to_nome w) =
pi * i"
        by simp
    qed
    also have "contour_integral (linepath 0 (-2 * orig - t)) (λw. pi*i)
/ (2*pi*i) = -orig - t / 2"
      by simp
    finally show ?thesis
      by (simp add: algebra_simps o_def)
  qed
  then obtain n where n: "winding_number γ2' 0 + orig + t / 2 = of_int
n"
    by (elim Ints_cases)

  have "contour_integral γ (λz. z * deriv f z / f z) / (2*pi*i) =
          I1 (λz. -t * deriv f z / f z + 2 * pi * i * (z + t)) / (2*pi*i)
-
          I2 (λz. -(deriv f z / f z)) / (2*pi*i)"
    using nz unfolding γ_def I1_def I2_def γ1_def γ2_def
    by (subst argument_principle_f_gen) (auto simp: diff_divide_distrib)
  also have "I1 (λz. -t * deriv f z / f z + 2 * pi * i * (z + t)) =
             I1 (λz. (-t) * (deriv f z / f z)) + I1 (λz. 2 * pi * i *
(z + t))"
    using nz' unfolding I1_def γ1_def
    by (subst contour_integral_add)
       (auto intro!: contour_integrable_continuous_linepath holomorphic_on_imp_continuous_o
                     holomorphic_intros)
  also have "... / (2*pi*i) = I1 (λz. (-t) * (deriv f z / f z)) / (2*pi*i)
+
                              I1 (λz. 2 * pi * i * (z + t)) / (2*pi*i)"
    by (simp add: add_divide_distrib)
  also have "I1 (λz. 2 * pi * i * (z + t)) = 2 * pi * i * (orig + t + 1
/ 2)"
    unfolding I1_def
  proof (rule contour_integral_unique)
    define F where "F = (λz. pi * i * z ^ 2 + 2 * pi * i * t * z)"
    have "((λz. 2 * pi * i * (z + t)) has_contour_integral (F (pathfinish
γ1) - F (pathstart γ1))) γ1"
```

```
    by (rule contour_integral_primitive[of UNIV])
       (auto simp: γ1_def F_def field_simps intro!: derivative_eq_intros)
  also have "F (pathfinish γ1) - F (pathstart γ1) = 2 * pi * i * (orig
+ t + 1 / 2)"
    by (simp add: F_def γ1_def power2_eq_square field_simps)
  finally show "((λz. 2 * pi * i * (z + t)) has_contour_integral
                  (2 * pi * i * (orig + t + 1 / 2))) γ1" .
qed
also have "... / (2*pi*i) = (orig + t + 1/2)"
  by (simp add: divide_simps)
also have "I1 (λz. (-t) * (deriv f z / f z)) = (-t) * I1 (λz. deriv
f z / f z)"
  using nz' unfolding I1_def γ1_def
  by (subst contour_integral_lmul)
     (auto intro!: contour_integrable_continuous_linepath holomorphic_on_imp_continuous_o
                   holomorphic_intros)

also have "I1 (λz. deriv f z / f z) = contour_integral (f ∘ γ1) (λz.
1 / z)"
  unfolding I1_def γ1_def by (subst contour_integral_comp_analyticW[OF
analytic[of _ UNIV]]) auto
also have "-t * ... / (2 * pi * i) = -t * winding_number (f ∘ γ1) 0"
  by (subst winding_number_valid_path)
     (auto simp: path_image_compose nz' intro!: valid_path_compose_analytic
analytic)
also have "winding_number (f ∘ γ1) 0 = of_int m"
  by (rule m)

also have "I2 (λz. -(deriv f z / f z)) = -I2 (λz. deriv f z / f z)"
  unfolding I2_def by (subst contour_integral_neg) auto
also have "I2 (λz. deriv f z / f z) = contour_integral γ2' (λz. 1 /
z)"
  unfolding I2_def γ2'_def γ2_def
  by (subst contour_integral_comp_analyticW[OF analytic[of _ UNIV]])
auto
also have "(-...) / (2 * pi * i) = -winding_number γ2' 0"
  by (subst winding_number_valid_path) auto
also have "-t * of_int m + (orig + t + 1 / 2) - - winding_number γ2'
0 =
           (t + 1) / 2 + of_int n + of_int (-m) * t"
  unfolding n [symmetric] by (simp add: field_simps)
finally have "contour_integral γ (λz. z * deriv f z / f z) / (2*pi*i)
- (t+1)/2 =
                complex_of_int n + complex_of_int (- m) * t"
  by simp
also have "... = lattice.of_ω12_coords (of_int n, of_int (-m))"
  by (simp add: lattice.of_ω12_coords_def)
also have "... ∈ lattice.lattice"
  by (rule lattice.of_ω12_coords_in_lattice) auto
```

```
  finally show ?thesis
    unfolding lattice.rel_def by blast
qed
```

We now tie everything together and prove the fact mentioned above: the zeros of f are precisely the shifted copies of $\frac{t+1}{2}$, and they are all simple. Unless of course $f(z)$ is identically zero.

```
lemma zero_iff:
  assumes "¬(∀z. f z = 0)"
  shows   "f z = 0 ⟷ lattice.rel z  ((t + 1) / 2)"
    and   "lattice.rel z ((t + 1) / 2) ⟹ zorder f z = 1"
proof -
  note [holomorphic_intros] = holomorphic
  define avoid where "avoid = {z. f z = 0}"
  define parallelogram where "parallelogram = (λz. closure (lattice.period_parallelogram
z))"
  define parallelogram' where "parallelogram' = (λz. interior (lattice.period_parallelogram
z))"

  have ev_nz: "eventually (λz. f z ≠ 0) (cosparse UNIV)"
  proof -
    have "(∀x∈UNIV. f x = 0) ∨ (∀≈x∈UNIV. f x ≠ 0)"
      by (intro nicely_meromorphic_imp_constant_or_avoid
                analytic_on_imp_nicely_meromorphic_on analytic) auto
    moreover have "¬(∀x∈UNIV. f x = 0)"
      using assms by auto
    ultimately show ?thesis
      by blast
  qed

  have sparse_avoid: "¬z islimpt avoid" for z
  proof -
    from ev_nz have "eventually (λz. f z ≠ 0) (at z)"
      by (rule eventually_cosparse_imp_eventually_at) auto
    thus ?thesis
      unfolding avoid_def by (auto simp: islimpt_iff_eventually)
  qed

  have countable: "countable avoid"
    using sparse_avoid no_limpt_imp_countable by blast
  obtain orig where avoid: "path_image (parallelogram_path orig 1 t)
∩ avoid = {}"
   using lattice.shifted_period_parallelogram_avoid[OF countable] by blast

  have "compact (parallelogram orig)"
    unfolding parallelogram_def by (rule lattice.compact_closure_period_parallelogram)
  then obtain R where R: "parallelogram orig ⊆ ball 0 R"
    by (meson bounded_subset_ballD compact_imp_bounded)
```

```
  define γ where "γ = parallelogram_path orig 1 t"
  have γ_subset: "path_image γ ⊆ parallelogram orig"
   and γ_disjoint: "path_image γ ∩ interior (lattice.period_parallelogram
orig) = {}"
    using lattice.path_image_parallelogram_subset_closure[of orig]
          lattice.path_image_parallelogram_disjoint_interior[of orig]
    unfolding parallelogram_def γ_def by blast+
  have path_image_γ_eq: "path_image γ = parallelogram orig - parallelogram'
orig"
    unfolding γ_def parallelogram_def parallelogram'_def
              lattice.path_image_parallelogram_path frontier_def ..
  have "parallelogram' orig ⊆ parallelogram orig"
    unfolding parallelogram_def parallelogram'_def
    by (meson closure_subset dual_order.trans interior_subset)
  hence parallelogram_conv_union: "parallelogram orig = parallelogram'
orig ∪ path_image γ"
    using γ_subset path_image_γ_eq by blast

  have γ: "valid_path γ" "path γ" "pathfinish γ = pathstart γ"
          "path_image γ ⊆ ball 0 R - {w ∈ ball 0 R. f w = 0}"
    using avoid R γ_subset by (auto simp: γ_def avoid_def intro!: valid_path_imp_path)

  have inside: "winding_number γ w = 1" if w: "w ∈ parallelogram orig"
"f w = 0" for w
    using Im_t w avoid γ_disjoint parallelogram_conv_union unfolding γ_def
    by (subst lattice.winding_number_parallelogram_inside)
       (auto simp: avoid_def parallelogram'_def parallelogram_def)

  have outside: "winding_number γ w = 0"
    if w: "w ∈ {w ∈ ball 0 R. f w = 0} - parallelogram orig" for w
  proof (rule winding_number_zero_outside)
    show "convex (parallelogram orig)"
      unfolding parallelogram_def by auto
    show "w ∉ parallelogram orig" using w avoid
      by auto
    show "path_image γ ⊆ parallelogram orig"
      using path_image_γ_eq by blast
  qed (use γ in auto)

  have outside': "∀z. z ∉ ball 0 R ⟶ winding_number γ z = 0"
  proof safe
    fix z :: complex assume z: "z ∉ ball 0 R"
    show "winding_number γ z = 0"
      by (rule winding_number_zero_outside[of _ "ball 0 R"])
         (use z γ in ‹auto simp: γ_def›)
  qed

  have "finite (cball 0 R ∩ {w. f w = 0})"
    by (rule sparse_in_compact_finite)
```

```
      (use ev_nz in ‹auto simp: eventually_cosparse intro: sparse_in_subset›)
  hence fin: "finite {w∈ball 0 R. f w = 0}"
    by (rule finite_subset [rotated]) auto

  have "contour_integral γ (λx. deriv f x / f x) / (2 * pi * i) = 1"
    unfolding γ_def by (subst argument_principle_f_1) (use avoid in ‹auto
simp: avoid_def›)
  also have "contour_integral γ (λx. deriv f x / f x) =
             contour_integral γ (λx. deriv f x * 1 / f x)"
    by simp
  also have "... / (2 * pi * i) =
               (∑p∈{w∈ball 0 R. f w = 0}. winding_number γ p * 1 *
of_int (zorder f p))"
    by (subst argument_principle[of "ball 0 R"])
       (use γ fin outside' in ‹auto intro!: holomorphic›)
  also have "(∑p∈{w ∈ ball 0 R. f w = 0}. winding_number γ p * 1 *
of_int (zorder f p)) =
             (∑p∈{w ∈ parallelogram orig. f w = 0}. of_int (zorder f
p))"
  proof (intro sum.mono_neutral_cong_right ballI)
    show "{w ∈ parallelogram orig. f w = 0} ⊆ {w ∈ ball 0 R. f w = 0}"
      using R by (auto simp: parallelogram_def path_image_γ_eq)
  qed (use fin in ‹auto simp: outside inside›)
  finally have "of_int (∑p∈{w ∈ parallelogram orig. f w = 0}. zorder
f p) = complex_of_int 1"
    unfolding of_int_sum by simp
  hence sum_zorder_eq: "(∑p∈{w ∈ parallelogram orig. f w = 0}. zorder
f p) = 1"
    by (simp only: of_int_eq_iff)

  obtain root where root: "{w ∈ parallelogram orig. f w = 0} = {root}"

                          "f root = 0" "zorder f root = 1"
  proof -
    have "int (card {w ∈ parallelogram orig. f w = 0}) = (∑p∈{w ∈ parallelogram
orig. f w = 0}. 1)"
      by simp
    also have "... ≤ (∑p∈{w ∈ parallelogram orig. f w = 0}. zorder f
p)"
    proof (rule sum_mono)
      fix w assume w: "w ∈ {w ∈ parallelogram orig. f w = 0}"
      have "∃ᶠ z in at w. f z ≠ 0" using ev_nz
        by (intro eventually_frequently eventually_cosparse_imp_eventually_at[OF
ev_nz]) auto
      hence "zorder f w > 0"
        by (subst zorder_pos_iff') (use w in ‹auto intro!: analytic›)
      thus "zorder f w ≥ 1"
        by simp
    qed
```

```
    also have "... = 1"
      by (fact sum_zorder_eq)
    finally have "card {w ∈ parallelogram orig. f w = 0} ≤ 1"
      by simp
    moreover have "card {w ∈ parallelogram orig. f w = 0} > 0"
    proof (subst card_gt_0_iff, safe)
      assume *: "{w ∈ parallelogram orig. f w = 0} = {}"
      show False
        using sum_zorder_eq unfolding * by simp
    next
      show "finite {w ∈ parallelogram orig. f w = 0}"
        by (rule finite_subset[OF _ fin]) (use R in <auto simp: parallelogram_def
path_image_γ_eq>)
    qed
    ultimately have "card {w ∈ parallelogram orig. f w = 0} = 1"
      by linarith
    then obtain w where w: "{w ∈ parallelogram orig. f w = 0} = {w}"
      by (auto simp: card_1_singleton_iff)
    moreover have "zorder f w = 1"
      using sum_zorder_eq unfolding w by simp
    ultimately show ?thesis
      using that by blast
  qed

  have "lattice.rel (contour_integral γ (λx. x * deriv f x / f x) / (2
* pi * i)) ((t + 1) / 2)"
    unfolding γ_def by (rule argument_principle_f_z) (use avoid in <auto
simp: avoid_def>)
  also have "contour_integral γ (λx. x * deriv f x / f x) =
             contour_integral γ (λx. deriv f x * x / f x)"
    by (simp add: mult_ac)
  also have "... / (2 * pi * i) =
               (∑p∈{w∈ball 0 R. f w = 0}. winding_number γ p * p *
of_int (zorder f p))"
    by (subst argument_principle[of "ball 0 R"])
       (use γ fin outside' in <auto intro!: holomorphic>)
  also have "(∑p∈{w ∈ ball 0 R. f w = 0}. winding_number γ p * p *
of_int (zorder f p)) =
             (∑p∈{w ∈ parallelogram orig. f w = 0}. p * of_int (zorder
f p))"
  proof (intro sum.mono_neutral_cong_right ballI)
    show "{w ∈ parallelogram orig. f w = 0} ⊆ {w ∈ ball 0 R. f w = 0}"
      using R by (auto simp: parallelogram_def path_image_γ_eq)
  qed (use fin in <auto simp: outside inside>)
  also have "... = root"
    unfolding root(1) using root(3) by simp
  finally have root': "lattice.rel root ((t + 1) / 2)" .

  show "f z = 0 ⟷ lattice.rel z ((t + 1) / 2)"
```

```
  proof
    assume "lattice.rel z ((t + 1) / 2)"
    also have "lattice.rel ((t + 1) / 2) root"
      using root' by (simp add: lattice.rel_sym)
    finally show "f z = 0"
      using f_zero_cong_lattice[of z root] using root by simp
  next
    assume "f z = 0"
    obtain h where h: "bij_betw h (lattice.period_parallelogram z)
                                  (lattice.period_parallelogram orig)"
                      "⋀w. lattice.rel (h w) w"
      using lattice.bij_betw_period_parallelograms[of z orig] by blast
    have "z ∈ lattice.period_parallelogram z"
      by auto
    hence "h z ∈ {z∈parallelogram orig. f z = 0}"
      using h(1) f_zero_cong_lattice[OF h(2)[of z]] ‹f z = 0›
      using bij_betw_apply closure_subset parallelogram_def by fastforce
    hence "h z = root"
      unfolding root(1) by simp
    hence "lattice.rel (h z) ((t+1) / 2)"
      using root' by simp
    thus "lattice.rel z ((t+1)/2)"
      using h(2) pre_complex_lattice.rel_symI pre_complex_lattice.rel_trans
by blast
  qed

  show "zorder f z = 1" if *: "lattice.rel z ((t+1)/2)"
  proof -
    have "zorder f z = zorder f ((t+1)/2)"
      by (rule zorder_f_cong_lattice) fact
    also have "... = zorder f root"
      by (rule sym, rule zorder_f_cong_lattice) fact
    also have "... = 1"
      by fact
    finally show ?thesis .
  qed
qed
```

Finally, we conclude that our quasi-periodic function is in fact a multiple of ϑ_{00}.

```
theorem multiple_jacobi_theta_00: "∃c. ∀z. f z = c * jacobi_theta_00
z t"
proof -
  define g where "g = (λz. jacobi_theta_00 0 t * f z - f 0 * jacobi_theta_00
z t)"
  interpret g: thetalike_function g t
  proof
    show "g holomorphic_on UNIV"
      unfolding g_def using Im_t by (auto intro!: holomorphic_intros
```

```
holomorphic)
    show "Im t > 0"
      by (fact Im_t)
    show "g (z + 1) = g z" for z
      unfolding g_def
      by (simp add: f_plus_1 jacobi_theta_00_left.plus_1)
    show "g (z + t) = g z / to_nome (2 * z + t)" for z
      unfolding g_def using f_plus_quasiperiod[of z] jacobi_theta_00_plus_quasiperiod[of
z t]
      by (simp add: diff_divide_distrib add_ac)
  qed

  show ?thesis
  proof (rule exI[of _ "f 0 / jacobi_theta_00 0 t"], rule allI)
    fix z :: complex
    have "g z = 0"
    proof (rule ccontr)
      assume "g z ≠ 0"
      hence *: "¬(∀z. g z = 0)"
        by auto
      have "g 0 = 0"
        by (simp add: g_def)
      also have "?this ⟷ lattice.rel 0 ((t + 1) / 2)"
        by (rule g.zero_iff[OF *])
      also have "... ⟷ (0 - ((t + 1) / 2)) ∈ lattice.lattice"
        by (simp add: lattice.rel_def)
      also have "0 - ((t + 1) / 2) = lattice.of_ω12_coords (-1/2, -1/2)"
        by (simp add: lattice.of_ω12_coords_def field_simps)
      finally show False
        by (subst (asm) lattice.of_ω12_coords_in_lattice_iff) auto
    qed
    thus "f z = f 0 / jacobi_theta_00 0 t * jacobi_theta_00 z t"
      using Im_t by (auto simp: field_simps jacobi_theta_00_0_left_nonzero
g_def)
  qed
qed

end
```

10.2 Theta inversion

Using the fact that any quasiperiodic function (in the sense used above) is a multiple of ϑ_{00} and the heat equation for ϑ_{00}, we can now relatively easily prove the theta inversion identity, which describes how ϑ_{00} transforms under the modular transformation $t \mapsto -\frac{1}{t}$:

$$\vartheta_{00}(z, -1/t) = \sqrt{-it} e^{i\pi t z^2} \vartheta_{00}(tz, t)$$

In particular, this means that $\vartheta_{00}(0, t)$ is a modular form of weight $\frac{1}{2}$.

```
theorem jacobi_theta_00_minus_one_over:
  fixes z t :: complex
  assumes t: "Im t > 0"
  shows "jacobi_theta_00 z (-(1/t)) = csqrt (-(i*t)) * to_nome (t*z^2)
* jacobi_theta_00 (t*z) t"
proof -
```

First of all, we look at the quotient

$$\frac{e^{i\pi tz^2}\vartheta_{00}(tz,t)}{\vartheta_{00}(z,-1/t)}$$

and show that it does not depend on z.

The proof works by nothing that, for fixed t with $\mathrm{Im}(t) > 0$, the numerator is a theta-like function in z and must therefore be a constant multiple of the denominator.

```
  define f where "f = (λz t. to_nome (t * z^2) * jacobi_theta_00 (t*z)
t)"
  define g where "g = (λz t. f z t / jacobi_theta_00 z (-1/t))"
  define c where "c = (λt. g 0 t)"
  have [analytic_intros]: "c analytic_on A" if "A ⊆ {t. Im t > 0}" for
A
    unfolding c_def g_def f_def using that
    by (auto intro!: analytic_intros simp: Im_complex_div_lt_0 jacobi_theta_00_0_left_nonze

  have f_eq: "f z t = c t * jacobi_theta_00 z (-1/t)" if t: "Im t > 0"
for z t
  proof -
    from t have [simp]: "t ≠ 0"
      by auto
    interpret f: thetalike_function "λz. f z t" "-1/t"
    proof
      show "(λz. f z t) holomorphic_on UNIV"
        using t unfolding f_def by (auto intro!: holomorphic_intros)
    next
      show "Im (-1/t) > 0"
        using t by (simp add: Im_complex_div_lt_0)
    next
      show "f (z + 1) t = f z t" for z
        using jacobi_theta_00_plus_quasiperiod[of "t*z" t]
        by (simp add: f_def ring_distribs power2_eq_square field_simps
to_nome_add)
    next
      show "f (z + (-1/t)) t = f z t / to_nome (2 * z + (-1/t))" for
z
      proof -
        have "f (z + (-1/t)) t = to_nome (t * (z - 1 / t)^2) * jacobi_theta_00
(t * z - 1) t"
          by (simp add: f_def ring_distribs)
```

```
      also have "jacobi_theta_00 (t * z - 1) t = jacobi_theta_00 (t
* z) t"
        by (rule jacobi_theta_00_left.minus_1)
      also have "t * (z - 1 / t) ^ 2 = t * z ^ 2 - 2 * z + 1 / t"
        by (simp add: field_simps power2_eq_square)
      also have "to_nome ... * jacobi_theta_00 (t * z) t = f z t / to_nome
(2*z + (-1/t))"
        by (simp add: f_def to_nome_add to_nome_diff)
      finally show ?thesis .
    qed
  qed

  obtain c' where "f z t = c' * jacobi_theta_00 z (-1/t)" for z
    using f.multiple_jacobi_theta_00 by blast
  from this[of 0] and this[of z] show ?thesis
    using jacobi_theta_00_0_left_nonzero[of "-1/t"] t
    by (auto simp: g_def divide_simps Im_complex_div_lt_0 c_def)
qed
```

Next, we take the equation

$$e^{i\pi tz^2}\vartheta_{00}(tz,t) = c(t)\vartheta_{00}(z,-1/t)$$

and take the derivatives $\frac{\partial^2}{\partial z}$ of both sides and then, separately, $\frac{\partial}{\partial t}$ of both sides. We then use the heat equation for ϑ_{00} and combine both equations, which gives us the following ordinary differential equation:

$$c'(t) = -\frac{c(t)}{2t}$$

```
have c_ODE: "deriv c t = (-1 / (2*t)) * c t" if t: "Im t > 0" for t
proof -
  have [simp]: "t ≠ 0"
    using t by auto

  have "(deriv ^^ 2) (λz. f z t) 0 = (deriv ^^ 2) (λz. c t * jacobi_theta_00
z (-1/t)) 0"
    using t by (subst f_eq) auto
  also have "... = c t * (deriv ^^ 2) (λz. jacobi_theta_00 z (- 1 /
t)) 0"
    by (rule higher_deriv_cmult')
       (use t in ‹auto intro!: analytic_intros simp: Im_complex_div_lt_0›)
  also have "(deriv ^^ 2) (λz. jacobi_theta_00 z (- 1 / t)) 0 =
               4 * pi * i * deriv (jacobi_theta_00 0) (-(1/t))"
    by (subst jacobi_theta_00_heat_equation) (use t in ‹auto simp: Im_complex_div_lt_0›)
  also have "(deriv ^^ 2) (λz. f z t) 0 =
               deriv (deriv (λz. jacobi_theta_00 (t * z) t)) 0 +
               2 * deriv (λz. to_nome (t*z^2)) 0 * deriv (λz. jacobi_theta_00
(t * z) t) 0 +
```

```
              deriv (deriv (λz. to_nome (t*z²))) 0 * jacobi_theta_00
0 t"
      unfolding f_def using t
      by (subst higher_deriv_mult[of _ UNIV]) (auto intro!: holomorphic_intros
simp: eval_nat_numeral)
    also have "deriv (λz. to_nome (t*z²)) = (λz. 2 * pi * ı * t * z *
to_nome (t*z²))"
      by (intro ext DERIV_imp_deriv) (auto intro!: derivative_eq_intros)
    also have "deriv ... = (λz. 2 * pi * ı * t * to_nome (t*z²) - 4 * pi²
* t² * z² * to_nome (t*z²))"
      by (intro ext DERIV_imp_deriv)
         (auto intro!: derivative_eq_intros simp: algebra_simps power2_eq_square)
    also have "deriv (λz. jacobi_theta_00 (t * z) t) = (λz. t * deriv
(λz. jacobi_theta_00 z t) (t*z))"
    proof (rule ext, rule DERIV_imp_deriv)
      fix z :: complex
      have "((λz. jacobi_theta_00 z t) ∘ (λz. t * z) has_field_derivative

              deriv (λz. jacobi_theta_00 z t) (t*z) * t) (at z)"
        by (rule DERIV_chain) (use t in ‹auto intro!: analytic_derivI
analytic_intros›)
      thus "((λz. jacobi_theta_00 (t*z) t) has_field_derivative
              t * deriv (λz. jacobi_theta_00 z t) (t*z)) (at z)"
        by (simp add: algebra_simps o_def)
    qed
    also have "deriv ... 0 = t * deriv (λz. deriv (λz. jacobi_theta_00
z t) (t*z)) 0"
      by (subst deriv_cmult') (use t in ‹auto intro!: analytic_intros
simp: eval_nat_numeral›)
    also have "deriv (λz. deriv (λz. jacobi_theta_00 z t) (t*z)) 0 =
               t * (deriv ^^ 2) (λz. jacobi_theta_00 z t) 0"
    proof (rule DERIV_imp_deriv)
      have "(deriv (λz. jacobi_theta_00 z t) ∘ (λz. t * z) has_field_derivative

              deriv (deriv (λz. jacobi_theta_00 z t)) 0 * t) (at 0)"
        by (rule DERIV_chain) (use t in ‹auto intro!: analytic_derivI
analytic_intros›)
      thus "((λz. deriv (λz. jacobi_theta_00 z t) (t * z)) has_field_derivative
              t * (deriv ^^ 2) (λz. jacobi_theta_00 z t) 0) (at 0)"
        by (simp add: algebra_simps o_def eval_nat_numeral)
    qed
    also have "(deriv ^^ 2) (λz. jacobi_theta_00 z t) 0 = 4 * pi * ı *
deriv (jacobi_theta_00 0) t"
      using t by (subst jacobi_theta_00_heat_equation) simp_all
    finally have "4 * pi * ı * (t² * deriv (jacobi_theta_00 0) t +
                                t * jacobi_theta_00 0 t / 2) =
                  4 * pi * ı * (c t * deriv (jacobi_theta_00 0) (- (1 /
t)))"
      unfolding ring_distribs by (simp add: field_simps power2_eq_square)
```

```
    hence *: "c t * deriv (jacobi_theta_00 0) (-(1/t)) =
                t^2 * deriv (jacobi_theta_00 0) t + t * jacobi_theta_00
0 t / 2"
      by (subst (asm) mult_cancel_left) auto

    have "deriv (f 0) t = deriv (λt. c t * jacobi_theta_00 0 (-1/t)) t"
    proof (rule deriv_cong_ev)
      have "eventually (λt. t ∈ {t. Im t > 0}) (nhds t)"
        by (rule eventually_nhds_in_open) (use t in ‹auto simp: open_halfspace_Im_gt›)
      thus "∀F x in nhds t. f 0 x = c x * jacobi_theta_00 0 (- 1 / x)"
        by eventually_elim (auto simp: f_eq)
    qed auto
    also have "... = c t * deriv (λt. jacobi_theta_00 0 (- (1 / t))) t
+
                     deriv c t * jacobi_theta_00 0 (- (1 / t))"
      by (subst complex_derivative_mult_at)
         (use t in ‹auto intro!: analytic_intros simp: Im_complex_div_lt_0›)
    also have "deriv (λt. jacobi_theta_00 0 (- (1 / t))) t =
                deriv (jacobi_theta_00 0) (-(1/t)) * deriv (λt. -(1/t))
t"
      by (rule deriv_compose_analytic)
         (use t in ‹auto intro!: analytic_intros simp: Im_complex_div_lt_0›)
    also have "deriv (λt. -(1/t)) t = 1 / t ^ 2"
      using t by (auto intro!: DERIV_imp_deriv derivative_eq_intros simp:
power2_eq_square)
    also have "deriv (f 0) t = deriv (jacobi_theta_00 0) t"
      unfolding f_def by simp
    finally have "deriv (jacobi_theta_00 0) t =
                    c t * deriv (jacobi_theta_00 0) (-(1/t)) / t ^ 2
+
                    deriv c t * jacobi_theta_00 0 (-(1/t))"
      by simp
    also note *
    finally have "jacobi_theta_00 0 t / (2*t) + deriv c t * jacobi_theta_00
0 (-(1/t)) = 0"
      by (simp add: add_divide_distrib power2_eq_square)

    also have "jacobi_theta_00 0 t = c t * jacobi_theta_00 0 (- (1 / t))"
      using f_eq[of t 0] t by (simp add: f_def)
    also have "c t * jacobi_theta_00 0 (-(1/t)) / (2 * t) + deriv c t
* jacobi_theta_00 0 (-(1/t)) =
               jacobi_theta_00 0 (-1/t) * (c t / (2*t) + deriv c t)"
      by (simp add: algebra_simps)
    finally have "c t / (2*t) + deriv c t = 0"
      unfolding mult_eq_0_iff using jacobi_theta_00_0_left_nonzero[of
"-1/t"] t
      by (simp_all add: Im_complex_div_lt_0)
    thus "deriv c t = (-1 / (2*t)) * c t"
      using t by (simp add: field_simps add_eq_0_iff)
```

```
  qed
```

This is a particularly simple separable ODE, which has the unique general solution $c(t) = C/\sqrt{t}$ for some constant C.

```
  have "∃C. ∀t∈{t. Im t > 0}. c t = C * (1 / csqrt t)"
    using c_ODE
  proof (rule separable_ODE_simple_unique)
    show "(c has_field_derivative deriv c t) (at t within {t. 0 < Im
t})"
      if "t ∈ {t. Im t > 0}" for t
      by (intro analytic_derivI analytic_intros) (use that in auto)
  next
    show "((λx. 1 / csqrt x) has_field_derivative - 1 / (2 * t) * (1
/ csqrt t))
            (at t within {t. 0 < Im t})" if "t ∈ {t. Im t > 0}" for t
using that
      by (auto intro!: derivative_eq_intros simp: complex_nonpos_Reals_iff

              simp flip: power2_eq_square)
  qed (auto simp: convex_halfspace_Im_gt)
  then obtain C where C: "⋀t. Im t > 0 ⟹ c t = C / csqrt t"
    by auto
```

Noting that $c(i) = 1$, we find that $C = \sqrt{i}$ and therefore $c(t) = 1/\sqrt{-it}$ as desired.

```
  have "c i = C / csqrt i"
    by (rule C) auto
  also have "c i = 1"
    by (simp add: c_def g_def f_def jacobi_theta_00_0_left_nonzero)
  finally have C_eq: "C = csqrt i"
    by (simp add: field_simps)

  have c_eq: "c t = 1 / csqrt (- i * t)" if t: "Im t > 0" for t
  proof -
    have [simp]: "t ≠ 0"
      using t by auto
    have "csqrt (i * (-i * t)) = csqrt i * csqrt (-i * t)"
    proof (rule csqrt_mult)
      have "Arg (-i * t) = Arg (-i) + Arg t"
        by (subst Arg_times) (use Arg_lt_pi[of t] t in auto)
      thus "Arg i + Arg (- i * t) ∈ {- pi<..pi}"
        using Arg_lt_pi[of t] t by auto
    qed
    hence "csqrt i / csqrt t = 1 / csqrt (-i * t)"
      by (simp add: field_simps del: csqrt_ii)
    also have "csqrt i / csqrt t = c t"
      by (subst C) (use t in ‹auto simp: C_eq›)
    finally show ?thesis .
  qed
```

```
  have "to_nome (t * z^2) * jacobi_theta_00 (t * z) t = c t * jacobi_theta_00
z (- 1 / t)"
    using f_eq[of t z] t by (simp add: f_def)
  also have "c t = 1 / csqrt (-i * t)"
    using t by (rule c_eq)
  finally show ?thesis
    using t by (auto simp add: field_simps)
qed
```

Equivalent identities for the other ϑ_{xx} follow:

```
lemma jacobi_theta_01_minus_one_over:
  fixes z t :: complex
  assumes "Im t > 0"
  shows "jacobi_theta_01 z (-(1/t)) = csqrt (-(i*t)) * to_nome (t*z^2)
* jacobi_theta_10 (t*z) t"
  by (simp add: jacobi_theta_01_def jacobi_theta_10_def jacobi_theta_00_minus_one_over
                ring_distribs power2_eq_square to_nome_add add_divide_distrib
assms)

lemma jacobi_theta_10_minus_one_over:
  fixes z t :: complex
  assumes "Im t > 0"
  shows "jacobi_theta_10 z (-(1/t)) = csqrt (-(i*t)) * to_nome (t*z^2)
* jacobi_theta_01 (t*z) t"
proof -
  have [simp]: "t ≠ 0"
    using assms by auto
  have "jacobi_theta_10 z (-1/t) = csqrt (-i*t) *
          (to_nome (z - 1 / (t * 4)) * to_nome (t * (z - 1 / (t * 2))^2))
*
          jacobi_theta_00 (t * z - 1 / 2) t" using assms
    by (simp add: jacobi_theta_10_def jacobi_theta_00_minus_one_over
                  power2_eq_square ring_distribs mult_ac)
  also have "to_nome (z - 1 / (t * 4)) * to_nome (t * (z - 1 / (t * 2))^2)
=
             to_nome (z - 1 / (t * 4) + t * (z - 1 / (t * 2))^2)"
    by (rule to_nome_add [symmetric])
  also have "z - 1 / (t * 4) + t * (z - 1 / (t * 2))^2 = t * z^2"
    by (auto simp: field_simps power2_eq_square)
  also have "jacobi_theta_00 (t * z - 1 / 2) t = jacobi_theta_01 (t *
z - 1) t"
    by (simp add: jacobi_theta_01_def)
  also have "jacobi_theta_01 (t * z - 1) t = jacobi_theta_01 (t * z) t"
    by (rule jacobi_theta_01_left.minus_1)
  finally show ?thesis by simp
qed

lemma jacobi_theta_11_minus_one_over:
```

```
  fixes z t :: complex
  assumes t: "Im t > 0"
  shows "jacobi_theta_11 z (-(1/t)) = -i * csqrt (-(i*t)) * to_nome (t*z^2)
* jacobi_theta_11 (t*z) t"
proof -
  have [simp]: "t ≠ 0"
    using assms by auto
  have "jacobi_theta_11 z (-1/t) = i * csqrt (-(i*t)) * to_nome (t * z^2)
*
         (to_nome (t*z + t/4 - 1/2) * jacobi_theta_00 (t * z - 1 / 2
+ t / 2) t)" using assms
    by (simp add: jacobi_theta_11_def jacobi_theta_00_minus_one_over add_ac
                  power2_eq_square ring_distribs mult_ac to_nome_add to_nome_diff)
  also have "to_nome (t * z + t / 4 - 1 / 2) * jacobi_theta_00 (t * z
- 1 / 2 + t / 2) t =
             jacobi_theta_11 (t*z - 1) t"
    by (simp add: jacobi_theta_11_def algebra_simps)
  also have "... = -jacobi_theta_11 (t*z) t"
    by (rule jacobi_theta_11_minus1_left)
  finally show ?thesis
    by simp
qed
```

10.3 Theta nullwert inversions in the reals

We can thus translate the above theta inversion identities into the q-disc. For simplicity, we only do this for real q with $0 < q < 1$, and we will focus mostly on the theta nullwert functions, where the identities are particularly nice (and stay within the reals).

We introduce the "q inversion" function

$$f : [0,1] \to [0,1],\ f(q) = \exp(\pi^2 / \log q)$$

with the border values $f(0) = 1$ and $f(1) = 0$. This function is a strictly decreasing involution on the real interval $[0,1]$. It corresponds to translating q from the q-disc to the z-plane, doing the transformation $z \mapsto -1/z$, and then translating the result back into the q-disc.

This is useful for computing $\vartheta_i(q)$, since we can apply the inversion to bring any q in the unit disc very close to 0, where the power series of ϑ_i converges extremely quickly.

```
definition q_inversion :: "real ⇒ real" where
  "q_inversion q = (if q = 0 then 1 else if q = 1 then 0 else exp (pi^2
/ ln q))"

lemma q_inversion_0 [simp]: "q_inversion 0 = 1"
  and q_inversion_1 [simp]: "q_inversion 1 = 0"
  by (simp_all add: q_inversion_def)
```

```
lemma q_inversion_nonneg: "q ∈ {0..1} ⟹ q_inversion q ≥ 0"
  and q_inversion_le_1: "q ∈ {0..1} ⟹ q_inversion q ≤ 1"
  and q_inversion_pos: "q ∈ {0..<1} ⟹ q_inversion q > 0"
  and q_inversion_less_1: "q ∈ {0<..1} ⟹ q_inversion q < 1"
  by (auto simp: q_inversion_def field_simps)

lemma q_inversion_strict_antimono: "strict_antimono_on {0..1} q_inversion"
  unfolding monotone_on_def by (auto simp: q_inversion_def field_simps)

lemma q_inversion_less_iff:
  assumes "q ∈ {0..1}" "q' ∈ {0..1}"
  shows   "q_inversion q < q_inversion q' ⟷ q > q'"
  using assms by (auto simp: q_inversion_def field_simps)

lemma q_inversion_le_iff:
  assumes "q ∈ {0..1}" "q' ∈ {0..1}"
  shows   "q_inversion q ≤ q_inversion q' ⟷ q ≥ q'"
  using assms by (auto simp: q_inversion_def field_simps)

lemma q_inversion_eq_iff:
  assumes "q ∈ {0..1}" "q' ∈ {0..1}"
  shows   "q_inversion q = q_inversion q' ⟷ q = q'"
  using assms by (auto simp: q_inversion_def field_simps)

lemma q_inversion_involution:
  assumes "q ∈ {0..1}"
  shows   "q_inversion (q_inversion q) = q"
  using assms by (auto simp: q_inversion_def)

lemma continuous_q_inversion [continuous_intros]:
  assumes q: "q ∈ {0..1}"
  shows   "continuous (at q within {0..1}) q_inversion"
proof -
  define f where "f = (λq. exp (pi ^ 2 / ln q))"

consider "q = 0" | "q = 1" | "q ∈ {0<..<1}"
    using q by force
  hence "(f ⟶ q_inversion q) (at q within {0..1})"
  proof cases
    assume q: "q = 0"
    have "(f ⟶ 1) (at_right 0)"
      unfolding f_def by real_asymp
    thus ?thesis
      by (simp add: q f_def at_within_Icc_at_right)
  next
    assume q: "q = 1"
    have "(f ⟶ 0) (at_left 1)"
      unfolding f_def by real_asymp
```

```
    thus ?thesis
      by (simp add: q f_def at_within_Icc_at_left)
  next
    assume q: "q ∈ {0<..<1}"
    have "isCont f q"
      using q unfolding f_def by (auto intro!: continuous_intros)
    hence "(f ⟶ f q) (at q within {0..1})"
      by (meson Lim_at_imp_Lim_at_within continuous_within)
    thus ?thesis
      using q by (simp add: q_inversion_def f_def)
  qed
  moreover have "eventually (λq. f q = q_inversion q) (at q within {0..1})"
    using eventually_neq_at_within[of 0] eventually_neq_at_within[of 1]
    by eventually_elim (auto simp: q_inversion_def f_def)
  ultimately have "(q_inversion ⟶ q_inversion q) (at q within {0..1})"
    by (rule Lim_transform_eventually)
  thus ?thesis
    using continuous_within by blast
qed

lemma continuous_on_q_inversion [continuous_intros]: "continuous_on {0..1}
q_inversion"
  using continuous_q_inversion continuous_on_eq_continuous_within by blast

lemma continuous_on_q_inversion' [continuous_intros]:
  assumes "continuous_on A f" "⋀x. x ∈ A ⟹ f x ∈ {0..1}"
  shows   "continuous_on A (λx. q_inversion (f x))"
  by (rule continuous_on_compose2[OF continuous_on_q_inversion assms(1)])
(use assms(2) in auto)


definition q_inversion_fixedpoint :: real where
  "q_inversion_fixedpoint = exp (-pi)"

lemma q_inversion_fixedpoint:
  defines "q0 ≡ q_inversion_fixedpoint"
  shows   "q0 ∈ {0..1}" "q_inversion q0 = q0"
proof -
  show "q0 ∈ {0..1}"
    by (auto simp: q0_def q_inversion_fixedpoint_def)
  show "q_inversion q0 = q0"
    by (auto simp: q_inversion_def q0_def q_inversion_fixedpoint_def field_simps
power2_eq_square)
qed

lemma q_inversion_less_self_iff:
  assumes "q ∈ {0..1}"
  shows   "q_inversion q < q ⟷ q > q_inversion_fixedpoint"
  using q_inversion_less_iff[OF assms q_inversion_fixedpoint(1)] q_inversion_fixedpoint(2)
```

```
  by auto

lemma q_inversion_greater_self_iff:
  assumes "q ∈ {0..1}"
  shows   "q_inversion q > q ⟷ q < q_inversion_fixedpoint"
  using q_inversion_less_iff[OF q_inversion_fixedpoint(1) assms] q_inversion_fixedpoint(2)
  by auto
```

From the theta inversion identities, we get three identities of the form $\vartheta_i(f(q)) = \sqrt{-\ln q/\pi}\vartheta_j(q)$. This can be harnessed to evaluate the theta nullwert functions very rapidly: their power series converge extremely quickly for small q, and since $f(q)$ has a unique fixed point $q_0 = e^{-\pi} \approx 0.0432$, we can reduce the computation of theta nullwert functions to computing them for q with $q \leq q_0$ via the inversion formulas.

```
lemma jacobi_theta_nome_inversion_real:
  fixes w q :: real
  assumes q: "q ∈ {0<..<1}" and w: "w > 0"
  shows "jacobi_theta_nome (of_real w) (of_real q) =
           complex_of_real (sqrt (- pi / ln q) * exp (- (ln w)^2 / (4 *
ln q))) *
           jacobi_theta_nome (cis (-pi * ln w / ln q)) (of_real (exp (pi^2
/ ln q)))"
proof -
  define x where "x = ln (of_real w) / (2 * pi * i)"
  define y where "y = of_real (ln q) / (pi * i)"
  have w_eq: "w = to_nome x ^ 2"
    using w by (simp add: x_def to_nome_def exp_of_real flip: exp_of_nat_mult)
  have q_eq: "q = to_nome y"
    using q by (simp add: y_def to_nome_def exp_of_real)
  have y: "y ≠ 0" "Im y > 0"
    using q by (auto simp: y_def Im_complex_div_gt_0 mult_neg_pos)

  have "jacobi_theta_nome (of_real w) (of_real q) = jacobi_theta_00 x
y"
    by (simp add: w_eq q_eq jacobi_theta_00_def flip: jacobi_theta_nome_of_real)
  also have "... = jacobi_theta_00 x (-(1/(-1/y)))"
    using ‹y ≠ 0› by auto
  also have "... = csqrt (i / y) * to_nome (- (x^2 / y)) * jacobi_theta_00
(-(x/y)) (-(1/y))"
    by (subst jacobi_theta_00_minus_one_over) (use y in ‹auto simp: Im_complex_div_lt_0›)
  also have "i / y = -of_real (pi / ln q)"
    by (auto simp: y_def field_simps)
  also have "jacobi_theta_00 (-(x/y)) (-(1/y)) =
             jacobi_theta_nome (inverse (to_nome (x / y)) ^ 2) (inverse
(to_nome (1 / y)))"
    by (simp add: jacobi_theta_00_def to_nome_power to_nome_minus)
  also have "inverse (to_nome (1 / y)) = exp (of_real (pi ^ 2 / ln q))"
    using q by (simp add: to_nome_def y_def field_simps exp_minus power2_eq_square)
  also have "... = of_real (exp (pi ^ 2 / ln q))"
```

```
    by (rule exp_of_real)
  also have "inverse (to_nome (x / y)) ^ 2 =
              exp (-i * (complex_of_real pi * Ln (complex_of_real w))
/ complex_of_real (ln q))"
    by (simp add: to_nome_def x_def y_def exp_minus field_simps flip:
exp_of_nat_mult)
  also have "-i * (complex_of_real pi * Ln (complex_of_real w)) / complex_of_real
(ln q) =
             -i * complex_of_real (pi * ln w / ln q)"
    using q w by (subst Ln_of_real') (auto simp: field_simps power2_eq_square)
  also have "exp ... = cis (-pi * ln w / ln q)"
    by (auto simp: exp_diff exp_add cis_conv_exp exp_minus field_simps)
  also have "csqrt (- complex_of_real (pi / ln q)) = csqrt (of_real (-pi
/ ln q))"
    by simp
  also have "... = of_real (sqrt (-pi / ln q))"
    by (subst of_real_sqrt) (use q in ‹auto simp: field_simps›)
  also have "to_nome (- (x^2 / y)) = exp (-(ln (of_real w) ^ 2 / of_real
(4 * ln q)))"
    unfolding x_def y_def by (simp add: field_simps to_nome_def power2_eq_square)
  also have "-(ln (complex_of_real w) ^ 2 / of_real (4 * ln q)) =
               of_real (-(ln w ^ 2) / (4 * ln q))"
    by (subst Ln_of_real') (use w q in ‹auto simp: field_simps power2_eq_square›)
  also have "exp ... = complex_of_real (exp (-(ln w ^ 2) / (4 * ln q)))"
    by (rule exp_of_real)
  finally show ?thesis by simp
qed

lemma jacobi_theta_nome_1_left_inversion_real:
  assumes q: "q ∈ {0..<1}"
  shows "jacobi_theta_nome 1 (q_inversion q) = sqrt (-ln q / pi) * jacobi_theta_nome
1 q"
proof (cases "q = 0")
  case False
  with assms have q: "q ∈ {0<..<1}"
    by auto
  have "complex_of_real (jacobi_theta_nome 1 q) = jacobi_theta_nome (of_real
1) (of_real q)"
    by (simp flip: jacobi_theta_nome_of_real)
  also have "... = of_real (sqrt (-pi / ln q) * jacobi_theta_nome 1 (exp
(pi^2 / ln q)))"
    by (subst jacobi_theta_nome_inversion_real) (use q in ‹auto simp flip:
jacobi_theta_nome_of_real›)
  finally have "jacobi_theta_nome 1 q = sqrt (-pi / ln q) * jacobi_theta_nome
1 (q_inversion q)"
    unfolding of_real_eq_iff q_inversion_def using q by simp
  thus ?thesis
    using q by (simp add: real_sqrt_divide field_simps real_sqrt_minus)
qed auto
```

```
lemma jacobi_theta_nw_00_inversion_real:
  assumes q: "q ∈ {0..<1::real}"
  shows   "jacobi_theta_nw_00 (q_inversion q) = sqrt (-ln q / pi) * jacobi_theta_nw_00
q"
  unfolding jacobi_theta_nome_00_def power_one
  by (subst jacobi_theta_nome_1_left_inversion_real [OF q]) auto

lemma jacobi_theta_nw_01_inversion_real:
  assumes q: "q ∈ {0..<1::real}"
  shows   "jacobi_theta_nw_01 (q_inversion q) = sqrt (-ln q / pi) * jacobi_theta_nw_10
q"
proof (cases "q = 0")
  case False
  with assms have q: "q ∈ {0<..<1}"
    by auto
  have "complex_of_real (jacobi_theta_nw_10 q) =
          of_real (q powr (1/4)) * jacobi_theta_nome (of_real q) (of_real
q)"
    by (simp add: jacobi_theta_nome_10_def jacobi_theta_nome_of_real)
  also have "... = of_real (sqrt (-(pi/ln q))) *
                   jacobi_theta_nome (-1) (complex_of_real (exp (pi^2 /
ln q)))"
    by (subst jacobi_theta_nome_inversion_real)
       (use q in ‹auto simp flip: cis_inverse simp: power2_eq_square powr_def
exp_minus›)
  also have "... = complex_of_real (sqrt (-pi/ln q) * jacobi_theta_nw_01
(exp (pi^2 / ln q)))"
    by (simp flip: jacobi_theta_nome_of_real add: jacobi_theta_nome_01_def)
  finally have "jacobi_theta_nw_10 q = sqrt (-pi / ln q) * jacobi_theta_nw_01
(q_inversion q)"
    unfolding of_real_eq_iff q_inversion_def using q by simp
  thus ?thesis
    using q by (simp add: real_sqrt_divide field_simps real_sqrt_minus)
qed auto

lemma jacobi_theta_nw_10_inversion_real:
  assumes q: "q ∈ {0<..1::real}"
  shows   "jacobi_theta_nw_10 (q_inversion q) = sqrt (-ln q / pi) * jacobi_theta_nw_01
q"
proof -
  have "jacobi_theta_nw_01 q = jacobi_theta_nw_01 (q_inversion (q_inversion
q))"
    using q by (simp add: q_inversion_involution)
  also have "... = sqrt (-ln (q_inversion q) / pi) * jacobi_theta_nw_10
(q_inversion q)"
    by (subst jacobi_theta_nw_01_inversion_real)
       (use q in ‹auto simp: q_inversion_less_1 q_inversion_nonneg›)
  also have "-ln (q_inversion q) / pi = -pi / ln q"
```

```
    using q by (simp add: q_inversion_def power2_eq_square)
  finally show ?thesis
    using q by (simp add: real_sqrt_divide field_simps real_sqrt_minus)
qed

end
```

11 The Dedekind η function

```
theory Dedekind_Eta
imports
  Bernoulli.Bernoulli
  Theta_Inversion
  Basic_Modular_Forms
  Dedekind_Sums.Dedekind_Sums
  Pentagonal_Number_Theorem.Pentagonal_Number_Theorem
begin


hide_const (open) Unique_Factorization.coprime
```

11.1 Definition and basic properties

```
definition dedekind_eta:: "complex ⇒ complex" ("η") where
  "η z = to_nome (z / 12) * euler_phi (to_nome (2*z))"

lemma dedekind_eta_nonzero [simp]: "Im z > 0 ⟹ η z ≠ 0"
  by (auto simp: dedekind_eta_def norm_to_nome norm_power power_less_one_iff)

lemma holomorphic_dedekind_eta [holomorphic_intros]:
  assumes "A ⊆ {z. Im z > 0}"
  shows   "η holomorphic_on A"
  using assms unfolding dedekind_eta_def
  by (auto intro!: holomorphic_intros simp: norm_to_nome norm_power power_less_one_iff)

lemma holomorphic_dedekind_eta' [holomorphic_intros]:
  assumes "f holomorphic_on A" "⋀z. z ∈ A ⟹ Im (f z) > 0"
  shows   "(λz. η (f z)) holomorphic_on A"
  using assms unfolding dedekind_eta_def
  by (auto intro!: holomorphic_intros simp: norm_to_nome norm_power power_less_one_iff)

lemma analytic_dedekind_eta [analytic_intros]:
  assumes "A ⊆ {z. Im z > 0}"
  shows   "η analytic_on A"
  using assms unfolding dedekind_eta_def
  by (auto intro!: analytic_intros simp: norm_to_nome norm_power power_less_one_iff)

lemma analytic_dedekind_eta' [analytic_intros]:
  assumes "f analytic_on A" "⋀z. z ∈ A ⟹ Im (f z) > 0"
```

```
  shows   "(λz. η (f z)) analytic_on A"
  using assms unfolding dedekind_eta_def
  by (auto intro!: analytic_intros simp: norm_to_nome norm_power power_less_one_iff)

lemma meromorphic_on_dedekind_eta [meromorphic_intros]:
  "f analytic_on A ⟹ (⋀z. z ∈ A ⟹ Im (f z) > 0) ⟹ (λz. η (f z))
meromorphic_on A"
  by (rule analytic_on_imp_meromorphic_on) (auto intro!: analytic_intros)

lemma continuous_on_dedekind_eta [continuous_intros]:
  "A ⊆ {z. Im z > 0} ⟹ continuous_on A η"
  unfolding dedekind_eta_def
  by (intro continuous_intros) (auto simp: norm_to_nome norm_power power_less_one_iff)

lemma continuous_on_dedekind_eta' [continuous_intros]:
  assumes "continuous_on A f" "⋀z. z ∈ A ⟹ Im (f z) > 0"
  shows   "continuous_on A (λz. η (f z))"
  using assms unfolding dedekind_eta_def
  by (intro continuous_intros) (auto simp: norm_to_nome norm_power power_less_one_iff)

lemma tendsto_dedekind_eta [tendsto_intros]:
  assumes "(f ⟶ c) F" "Im c > 0"
  shows   "((λx. η (f x)) ⟶ η c) F"
  unfolding dedekind_eta_def using assms
  by (intro tendsto_intros assms) (auto simp: norm_to_nome norm_power
power_less_one_iff)

lemma tendsto_at_cusp_dedekind_eta [tendsto_intros]: "(η ⟶ 0) at_i∞"
proof -
  have "filterlim (λx::real. x / 12) at_top at_top"
    by real_asymp
  hence "filterlim (λz. Im z / 12) at_top at_i∞"
    by (rule filterlim_at_ii_infI)
  hence *: "((λz. to_nome (z / 12)) ⟶ 0) at_i∞"
    by (intro tendsto_0_to_nome) simp_all

  have "filterlim (λx::real. 2 * x) at_top at_top"
    by real_asymp
  hence "filterlim (λz. 2 * Im z) at_top at_i∞"
    by (rule filterlim_at_ii_infI)
  hence **: "((λz. to_nome (2 * z)) ⟶ 0) at_i∞"
    by (intro tendsto_0_to_nome) simp_all

  have "((λz. to_nome (z / 12) * euler_phi (to_nome (2*z))) ⟶ 0 *
euler_phi 0) at_i∞"
    by (intro tendsto_intros * **) auto
  thus ?thesis
    by (simp add: dedekind_eta_def [abs_def])
qed
```

```
lemma dedekind_eta_plus1:
  assumes z: "Im z > 0"
  shows   "η (z + 1) = cis (pi/12) * η z"
proof -
  have "η (z + 1) = to_nome ((z + 1) / 12) * euler_phi (to_nome (2 * (z
+ 1)))"
    by (simp add: dedekind_eta_def)
  also have "to_nome (2 * (z + 1)) = to_nome (2 * z)"
    by (simp add: to_nome_add ring_distribs)
  also have "to_nome ((z + 1) / 12) = to_nome (1/12) * to_nome (z / 12)"
    by (simp add: to_nome_add add_divide_distrib)
  also have "to_nome (1/12) = cis (pi/12)"
    by (simp add: to_nome_def cis_conv_exp)
  also have "... * to_nome (z / 12) * euler_phi (to_nome (2*z)) = cis
(pi/12) * η z"
    by (simp add: dedekind_eta_def mult_ac)
  finally show ?thesis by (simp add: cis_conv_exp mult_ac)
qed

lemma dedekind_eta_plus_nat:
  assumes z: "Im z > 0"
  shows   "η (z + of_nat n) = cis (pi * n / 12) * η z"
proof (induction n)
  case (Suc n)
  have "η (z + of_nat (Suc n)) = η (z + of_nat n + 1)"
    by (simp add: add_ac)
  also have "... = cis (pi/12) * η (z + of_nat n)"
    using z by (intro dedekind_eta_plus1) auto
  also have "η (z + of_nat n) = cis (pi*n/12) * η z"
    by (rule Suc.IH)
  also have "cis (pi/12) * (cis (pi*n/12) * η z) = cis (pi*Suc n/12) *
η z"
    by (simp add: ring_distribs add_divide_distrib exp_add mult_ac cis_mult)
  finally show ?case .
qed auto

lemma dedekind_eta_plus_int:
  assumes z: "Im z > 0"
  shows   "η (z + of_int n) = cis (pi*n/12) * η z"
proof (cases "n ≥ 0")
  case True
  thus ?thesis
    using dedekind_eta_plus_nat[OF assms, of "nat n"] by simp
next
  case False
  thus ?thesis
    using dedekind_eta_plus_nat[of "z + of_int n" "nat (-n)"] assms
    by (auto simp: cis_mult field_simps)
```

qed

The logarithmic derivative of η is, up to a constant factor, the "forbidden" Eisenstein series G_2. This follows relatively easily from the logarithmic derivative of Euler's function ϕ and the Fourier expansion of G_2, both of which involve the Lambert series $\sum_{k=1}^{\infty} k\frac{q^k}{1-q^k}$.

```
theorem deriv_dedekind_eta:
  assumes z: "Im z > 0"
  shows   "deriv η z = i / (4 * of_real pi) * Eisenstein_G 2 z * η z"
proof -
  define f :: "complex ⇒ complex" where "f = deriv euler_phi"
  have *: "(euler_phi has_field_derivative f q) (at q within A)" if "norm
q < 1" for q A
    unfolding f_def using that by (auto intro!: analytic_derivI analytic_intros)
  have [derivative_intros]:
    "((λz. euler_phi (g z)) has_field_derivative f (g z) * g') (at z within
A)"
    if "(g has_field_derivative g') (at z within A)" "norm (g z) < 1" for
g g' z A
    by (rule DERIV_chain'[OF that(1) *]) fact

  define L where "L = lambert complex_of_nat (to_nome (2 * z))"
  have L_eq: "L = 1 / 24 - Eisenstein_G 2 z / (8 * pi ^ 2)"
    by (subst Eisenstein_G_fourier_expansion)
       (use z in ‹simp_all add: L_def zeta_2 field_simps to_q_conv_to_nome›)

  have "deriv dedekind_eta z = i * pi * (dedekind_eta z / 12 +
                               2 * (to_nome (z / 12) * (to_nome (2*z)
* f (to_nome (2*z)))))"
    by (rule DERIV_imp_deriv)
       (use z in ‹auto intro!: derivative_eq_intros DERIV_imp_deriv
                      simp: norm_to_nome dedekind_eta_def [abs_def] algebra_simps›)
  also have "to_nome (2 * z) * f (to_nome (2 * z)) = -L * euler_phi (to_nome
(2 * z))"
    unfolding f_def by (subst deriv_euler_phi) (use z in ‹auto simp: norm_to_nome
L_def›)
  also have "to_nome (z / 12) * ... = -L * dedekind_eta z"
    by (simp add: dedekind_eta_def)
  finally have "deriv η z = (i * pi * (1 / 12 - 2 * L)) * η z"
    using z by (simp add: field_simps)
  also have "(i * pi * (1 / 12 - 2 * L)) = i / (4 * of_real pi) * Eisenstein_G
2 z"
    unfolding L_eq by (simp add: power2_eq_square)
  finally show ?thesis .
qed

lemma has_field_derivative_dedekind_eta:
  assumes "(f has_field_derivative f') (at x within A)" "Im (f x) > 0"
  shows   "((λx. η (f x)) has_field_derivative
```

```
            (i / (4 * of_real pi) * Eisenstein_G 2 (f x) * η (f x) * f'))
(at x within A)"
proof (rule DERIV_chain2[OF _ assms(1)])
  have "(η has_field_derivative deriv η (f x)) (at (f x))"
    by (rule analytic_derivI) (use assms(2) in ‹auto intro!: analytic_intros›)
  thus "(η has_field_derivative i / (4 * complex_of_real pi) *
          Eisenstein_G 2 (f x) * η (f x)) (at (f x))"
    by (subst (asm) deriv_dedekind_eta) (use assms(2) in auto)
qed
```

11.2 Relation to the Jacobi ϑ functions

```
lemma dedekind_eta_conv_jacobi_theta_01:
  assumes t: "Im t > 0"
  shows   "η t = to_nome (t / 12) * jacobi_theta_01 (-t/2) (3 * t)"
proof -
  include qpochhammer_inf_notation
  define q where "q = to_nome t"
  have q: "q ≠ 0" "norm q < 1"
    using t by (auto simp: q_def norm_to_nome)

  have "η t = to_nome (t / 12) * euler_phi (q ^ 2)"
    unfolding dedekind_eta_def by (simp add: q_def to_nome_power)
  also have "euler_phi (q ^ 2) = ramanujan_theta (- q^2) (-(q ^ 4))"
    by (subst pentagonal_number_theorem_complex)
       (use q in ‹simp_all add: norm_power power_less_1_iff›)
  also have "ramanujan_theta (-(q ^ 2)) (- (q ^ 4)) = jacobi_theta_nome
(-1/q) (q^3)"
    using q by (simp add: jacobi_theta_nome_def eval_nat_numeral mult_ac)
  also have "... = jacobi_theta_00 (-t/2 + 1/2) (3 * t)"
    by (simp add: jacobi_theta_00_def q_def to_nome_power to_nome_diff
                  power_divide diff_divide_distrib)
  also have "... = jacobi_theta_01 (-t/2) (3 * t)"
    by (simp add: jacobi_theta_01_def)
  finally show ?thesis .
qed

lemma jacobi_theta_01_nw_conv_dedekind_eta:
  assumes t: "Im t > 0"
  shows   "jacobi_theta_01 0 t = η (t/2) ^ 2 / η t"
proof -
  include qpochhammer_inf_notation
  define q where "q = to_nome t"
  have q: "q ≠ 0" "norm q < 1"
    using t by (auto simp: q_def norm_to_nome)
  have nz: "(q^2 ; q^2)∞ ≠ 0"
    by (rule qpochhammer_inf_nonzero) (use q in ‹auto simp: norm_power
power_less_1_iff›)
  have eq: "(q ; q)∞ = (q ; q^2)∞ * (q^2 ; q^2)∞"
```

```
    using prod_qpochhammer_inf_group[of q 2 q] q by (simp add: eval_nat_numeral)

  have "jacobi_theta_01 0 t = jacobi_theta_nome (-1) q"
    by (simp add: q_def jacobi_theta_01_def jacobi_theta_00_def)
  also have "... = (q^2 ; q^2)∞ * (q ; q^2)∞ ^ 2"
    by (subst jacobi_theta_nome_triple_product_complex) (use q in ‹simp_all
add: power2_eq_square›)
  also have "... = euler_phi q ^ 2 / euler_phi (q ^ 2)"
    by (simp add: field_simps euler_phi_def eq power2_eq_square)
  also have "... = η (t/2) ^ 2 / η t"
    by (simp add: dedekind_eta_def power_mult_distrib to_nome_power q_def)
  finally show ?thesis .
qed

lemma jacobi_theta_00_nw_conv_dedekind_eta:
  assumes t: "Im t > 0"
  shows   "jacobi_theta_00 0 t = η ((t+1)/2) ^ 2 / η (t+1)"
proof -
  include qpochhammer_inf_notation
  define q where "q = to_nome t"
  have q: "q ≠ 0" "norm q < 1"
    using t by (auto simp: q_def norm_to_nome)
  have nz: "(q^2 ; q^2)∞ ≠ 0"
    by (rule qpochhammer_inf_nonzero) (use q in ‹auto simp: norm_power
power_less_1_iff›)
  have eq: "(-q ; -q)∞ = (- q ; q^2)∞ * (q^2 ; q^2)∞"
    using prod_qpochhammer_inf_group[of "-q" 2 "-q"] q by (simp add: eval_nat_numeral)

  have "jacobi_theta_00 0 t = jacobi_theta_nome 1 q"
    by (simp add: q_def jacobi_theta_00_def)
  also have "... = (q^2; q^2)∞ * (-q; q^2)∞ ^ 2"
    by (subst jacobi_theta_nome_triple_product_complex) (use q in ‹simp_all
add: power2_eq_square›)
  also have "... = euler_phi (-q) ^ 2 / euler_phi (q ^ 2)"
    using nz by (simp add: field_simps euler_phi_def power2_eq_square
eq)
  also have "... = η ((t+1)/2) ^ 2 / η (t + 1)"
    by (simp add: dedekind_eta_def power_mult_distrib to_nome_power
                  to_nome_add add_divide_distrib q_def)
  finally show ?thesis .
qed

lemma jacobi_theta_00_nw_conv_dedekind_eta':
  assumes t: "Im t > 0"
  shows   "jacobi_theta_00 0 t = η t ^ 5 / (η (t/2) * η (2*t)) ^ 2"
proof -
  include qpochhammer_inf_notation
  define q where "q = to_nome t"
  have q: "q ≠ 0" "norm q < 1"
```

```
    using t by (auto simp: q_def norm_to_nome)
  have nz': "(q ; q)∞ ≠ 0"
    by (rule qpochhammer_inf_nonzero) (use q in <auto simp: norm_power
power_less_1_iff>)
  have nz'': " (- q² ; q²)∞ ≠ 0"
    by (rule qpochhammer_inf_nonzero) (use q in <auto simp: norm_power
power_less_1_iff>)
  have nz: "(q^2 ; q^2)∞ ≠ 0"
    by (rule qpochhammer_inf_nonzero) (use q in <auto simp: norm_power
power_less_1_iff>)
  have eq: "(-q ; q)∞ = (- q ; q^2)∞ * (-q² ; q²)∞"
    using prod_qpochhammer_inf_group[of q 2 "-q"] q by (simp add: eval_nat_numeral)

  have "η t ^ 5 / (η (t/2) * η (2*t)) ^ 2 =
        to_nome (5 * t / 12) / to_nome (t / 12) / to_nome (t / 3) *

        euler_phi (q ^ 2) ^ 5 / (euler_phi q * euler_phi (q ^ 4))²"
    by (simp add: dedekind_eta_def power_mult_distrib to_nome_power q_def)
  also have "to_nome (5 * t / 12) / to_nome (t / 12) / to_nome (t / 3)
=
             to_nome (5 * t / 12 - t / 12 - t / 3)"
    unfolding to_nome_diff ..
  also have "5 * t / 12 - t / 12 - t / 3 = 0"
    by simp
  also have "to_nome 0 * euler_phi (q²) ^ 5 / (euler_phi q * euler_phi
(q ^ 4))² =
             (q² ; q²)∞ ^ 5 / ((q ; q)∞ * (q ^ 4 ; q ^ 4)∞)²"
    by (simp add: euler_phi_def)
  finally have "η t ^ 5 / (η (t / 2) * η (2 * t))² =
                (q² ; q²)∞ ^ 5 / ((q ; q)∞ * (q ^ 4 ; q ^ 4)∞)²" .
  also have "... = (q² ; q²)∞ * (q² ; q²)∞ ^ 2 / ((q ; q)∞ * ((q ^ 4
; q ^ 4)∞ / (q² ; q²)∞))²"
    by (simp add: eval_nat_numeral)
  also have "(q ^ 4 ; q ^ 4)∞ / (q² ; q²)∞ = (-q² ; q²)∞"
    using qpochhammer_inf_square[of "q^2" "q^2"] q nz
    by (simp add: norm_power power_less_1_iff field_simps)
  also have "(q² ; q²)∞ * (q² ; q²)∞ ^ 2 / ((q ; q)∞ * (- q² ; q²)∞)²
=
            (q² ; q²)∞ * ((q² ; q²)∞ / (q ; q)∞ / (- q² ; q²)∞) ^
2"
    by (simp add: power_divide power_mult_distrib)
  also have "(q² ; q²)∞ / (q ; q)∞ = (-q ; q)∞"
    using qpochhammer_inf_square[of "q" "q"] q nz'
    by (simp add: norm_power power_less_1_iff field_simps)
  also have "(-q ; q)∞ / (- q² ; q²)∞ = (- q ; q²)∞"
    using eq nz'' by simp
  also have "(q² ; q²)∞ * (- q ; q²)∞² = jacobi_theta_nome 1 q"
    by (subst jacobi_theta_nome_triple_product_complex) (use q in <simp_all
add: power2_eq_square>)
```

```
  also have "... = jacobi_theta_00 0 t"
    by (simp add: q_def jacobi_theta_00_def)
  finally show ?thesis ..
qed

lemma jacobi_theta_10_nw_conv_dedekind_eta:
  assumes t: "Im t > 0"
  shows   "jacobi_theta_10 0 t = 2 * η (2*t) ^ 2 / η t"
proof -
  include qpochhammer_inf_notation
  define q where "q = to_nome t"
  have q: "q ≠ 0" "norm q < 1"
    using t by (auto simp: q_def norm_to_nome)
  have nz: "(q^2 ; q^2)∞ ≠ 0"
    by (rule qpochhammer_inf_nonzero) (use q in ‹auto simp: norm_power
power_less_1_iff›)

  have "2 * η (2*t) ^ 2 / η t =
        2 * (to_nome (t / 3) / to_nome (t / 12)) * euler_phi (q^4) ^
2 / euler_phi (q^2)"
    by (auto simp: dedekind_eta_def power_mult_distrib to_nome_power q_def)
  also have "to_nome (t / 3) / to_nome (t / 12) = to_nome (t / 3 - t /
12)"
    by (subst to_nome_diff) auto
  also have "t / 3 - t / 12 = t / 4"
    by auto
  finally have *: "2 * η (2*t) ^ 2 / η t =
                   2 * to_nome (t/4) * euler_phi (q^4) ^ 2 / euler_phi
(q^2)" .

  have "jacobi_theta_10 0 t = to_nome (t / 4) * jacobi_theta_nome q q"
    by (simp add: q_def jacobi_theta_10_def jacobi_theta_00_def to_nome_power)
  also have "... = to_nome (t / 4) * ((q² ; q²)∞ * (-(q²) ; q²)∞ * (-1
; q²)∞)"
    by (subst jacobi_theta_nome_triple_product_complex) (use q in ‹auto
simp: power2_eq_square›)
  also have "(-1 ; q²)∞ = 2 * (-q² ; q²)∞"
    using qpochhammer_inf_mult_q[of "q^2" "-1"] q by (simp add: norm_power
power_less_1_iff)
  also have "(q² ; q²)∞ * (- q² ; q²)∞ * (2 * (-q² ; q²)∞) =
             2 * ((q² ; q²)∞ * (-q² ; q²)∞) ^ 2 / (q² ; q²)∞"
    using nz by (simp add: power2_eq_square)
  also have "... = 2 * (q ^ 4 ; q ^ 4)∞ ^ 2 / (q² ; q²)∞"
    by (subst qpochhammer_inf_square) (use q in ‹auto simp: norm_power
power_less_1_iff›)
  also have "... = 2 * euler_phi (q^4) ^ 2 / euler_phi (q^2)"
    using nz by (simp add: euler_phi_def power2_eq_square)
  also have "to_nome (t / 4) * ... = 2 * η (2*t) ^ 2 / η t"
    using * by simp
```

```
  finally show ?thesis .
qed

lemma jacobi_theta_00_01_10_nw_conv_dedekind_eta:
  assumes t: "Im t > 0"
  shows   "jacobi_theta_00 0 t * jacobi_theta_01 0 t * jacobi_theta_10
0 t = 2 * η t ^ 3"
  using t by (simp add: jacobi_theta_00_nw_conv_dedekind_eta' field_simps
eval_nat_numeral
               jacobi_theta_01_nw_conv_dedekind_eta jacobi_theta_10_nw_conv_dedekind_eta)
```

11.3 The inversion identity

The inversion identity for Jacobi's ϑ function together with the Jacobi triple product allows us to give a rather short proof of the inversion law for η. This is remarkable: Apostol spends the majority of the chapter on proving this.

We would like to thank Alexey Ustinov, who answered a question of ours on MathOverflow and clarified how to prove the following lemma.

```
lemma dedekind_eta_minus_one_over_aux:
  assumes "Im t > 0"
  shows   "jacobi_theta_10 (1 / 6) (t / 3) =
             of_real (sqrt 3) * to_nome (t / 12) * jacobi_theta_01 (-t
/ 2) (3 * t)"
proof -
  include qpochhammer_inf_notation
  define q where "q = to_nome (t / 12)"
  define r where "r = to_nome (1 / 6)"
  define r' where "r' = to_nome (2/3)"

  have cos_120': "cos (pi * 2 / 3) = -1/2"
    using cos_120 by (simp add: field_simps)
  have sin_120': "sin (pi * 2 / 3) = sqrt 3 / 2"
    using sin_120 by (simp add: field_simps)

  have [simp]: "q ≠ 0" "r ≠ 0"
    by (auto simp: q_def r_def)
  have q: "norm q < 1"
    using assms by (simp add: q_def norm_to_nome)
  have [simp]: "norm (q ^ n) < 1 ⟷ n > 0" for n
    using q by (auto simp: norm_power power_less_1_iff)
  have "jacobi_theta_10 (1 / 6) (t / 3) = r * q * jacobi_theta_nome (r^2
* q^4) (q^4)"
    by (simp add: jacobi_theta_10_def jacobi_theta_00_def power2_eq_square
                  q_def r_def to_nome_power add_ac flip: to_nome_add)
  also have "... = r * q * ((q^8 ; q^8)∞ * ((-(r^2)) * q^8 ; q ^ 8)∞
* (-(1/r^2) ; q^8)∞)"
    by (subst jacobi_theta_nome_triple_product_complex)
```

```
      (auto simp: q norm_power power_less_one_iff algebra_simps)
  also have "-(r^2) = r'^2"
  proof -
    have "-(r^2) = to_nome (1 + 1/3)"
      unfolding to_nome_add by (simp add: r_def to_nome_power)
    also have "... = r' ^ 2"
      by (simp add: r'_def to_nome_power)
    finally show ?thesis .
  qed
  also have "-(1/r^2) = r'"
    by (auto simp: r_def r'_def to_nome_power field_simps simp flip: to_nome_add)
  also have "(r' ; q ^ 8)∞ = (r' * q ^ 8 ; q ^ 8)∞ * (1 - r')"
    by (subst qpochhammer_inf_mult_q) auto
  finally have "jacobi_theta_10 (1 / 6) (t / 3) =
            r * (1 - r') * q * (∏k<3. (r' ^k * q ^ 8 ; q ^ 8)∞)"
    by (simp add: numeral_3_eq_3 mult_ac power2_eq_square)
  also have "r * (1 - r') = of_real (sqrt 3)"
    by (simp add: r_def r'_def to_nome_def exp_eq_polar complex_eq_iff
                cos_30 sin_30 cos_120' sin_120' field_simps)
  also have "(∏k<3. (r' ^ k * q ^ 8 ; q ^ 8)∞) = (q ^ 24 ; q ^ 24)∞"
  proof -
    interpret primroot_cis 3 1
      rewrites "cis (2 * pi * 1 / 3) ≡ r'" and
               "cis (2 * pi * j * of_int 1 / of_nat 3) ≡ r' ^ j" for j
      by unfold_locales (auto simp: r'_def cis_conv_to_nome to_nome_power
field_simps)
    show ?thesis
      using prod_qpochhammer_group_cis[of "q^8" "q^8"] by simp
  qed
  finally have eq1: "jacobi_theta_10 (1 / 6) (t / 3) =
                   complex_of_real (sqrt 3) * q * (q ^ 24 ; q ^ 24)∞"
.


  have "jacobi_theta_01 (-t / 2) (3 * t) = jacobi_theta_nome (r^6 / q^12)
(q^36)"
    by (simp add: jacobi_theta_01_def jacobi_theta_00_def power2_eq_square
                  q_def r_def to_nome_power flip: to_nome_add to_nome_diff
to_nome_minus)
       (simp add: to_nome_diff to_nome_minus field_simps)?
  also have "... = (q^72; q^72)∞ * (q^48 ; q^72)∞ * ((q^36 / q^12) ;
q^72)∞"
    by (subst jacobi_theta_nome_triple_product_complex)
       (auto simp: q norm_power power_less_one_iff to_nome_power field_simps
r_def)
  also have "q^36 / q^12 = q^24"
    by (auto simp: field_simps)
  finally have "jacobi_theta_01 (- t / 2) (3 * t) = (∏k<3. (q^24 * (q^24)^k
; q^72)∞)"
```

```
    by (simp add: numeral_3_eq_3 mult_ac)
  also have "... = (q ^ 24 ; q ^ 24)∞"
    using prod_qpochhammer_inf_group[of "q^24" 3 "q^24"] by simp
  finally have "jacobi_theta_01 (-t/2) (3*t) = (q^24; q^24)∞" .
  hence eq2: "of_real (sqrt 3) * to_nome (t / 12) * jacobi_theta_01 (-t/2)
(3*t) =
              of_real (sqrt 3) * q * (q^24; q^24)∞"
    by (simp add: q_def)

  from eq1 eq2 show ?thesis
    by simp
qed

theorem dedekind_eta_minus_one_over:
  assumes t: "Im t > 0"
  shows   "η (-(1/t)) = csqrt (-i*t) * η t"
proof -
  have [simp]: "t ≠ 0"
    using t by auto
  have "η (-1/t) = to_nome (- (1 / (t * 12))) * jacobi_theta_01 (1 / (t
* 2)) (-(1 / (t / 3)))"
    by (subst dedekind_eta_conv_jacobi_theta_01) (use assms in ‹auto simp:
Im_complex_div_lt_0›)
  also have "... = csqrt ((1/3) * (-i * t)) * jacobi_theta_10 (1 / 6) (t
/ 3)"
    by (subst jacobi_theta_01_minus_one_over)
       (auto simp: to_nome_diff to_nome_minus to_nome_add field_simps
t power2_eq_square)
  also have "csqrt ((1/3) * (-i * t)) = csqrt (-i * t) / sqrt 3"
    by (subst csqrt_mult) (use Arg_eq_0[of "1/3"] in ‹auto simp: Arg_bounded
real_sqrt_divide›)
  also have "jacobi_theta_10 (1 / 6) (t / 3) =
               of_real (sqrt 3) * to_nome (t / 12) * jacobi_theta_01 (-t
/ 2) (3 * t)"
    by (rule dedekind_eta_minus_one_over_aux) fact
  also have "... = sqrt 3 * η t"
    using t by (simp add: jacobi_theta_10_def dedekind_eta_conv_jacobi_theta_01)
  finally show ?thesis
    by simp
qed
```

11.4 General transformation law

From our results so far, it is easy to see that η^{24} is a modular form of weight 12. Thus it follows that if $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{SL}(2)$ is a modular transformation, then $\eta(Az) = \epsilon(A)\sqrt{z}\eta(z)$, where $\epsilon(A)$ is a 24-th unit root that depends on A but not on z.

In the this section, we will give a concrete definition of this 24-th root ε in terms of A.

```
definition dedekind_eps :: "modgrp ⇒ complex" ("ε") where
  "ε f =
     (if is_singular_modgrp f then
        cis (pi * ((modgrp_a f + modgrp_d f) / (12 * modgrp_c f) -
          dedekind_sum (modgrp_d f) (modgrp_c f) - 1 / 4))
      else
        cis (pi * modgrp_b f / 12)
     )"

lemma dedekind_eps_1 [simp]: "dedekind_eps 1 = 1"
  by (simp add: dedekind_eps_def)

lemma dedekind_eps_shift [simp]: "ε (shift_modgrp m) = cis (pi * m /
12)"
  by (simp add: dedekind_eps_def dedekind_sum_def)

lemma dedekind_eps_S [simp]: "dedekind_eps S_modgrp = cis (-pi / 4)"
  by (simp add: dedekind_eps_def dedekind_sum_def complex_eq_iff)

lemma dedekind_eps_shift_right [simp]: "ε (f * shift_modgrp m) = cis
(pi * m / 12) * ε f"
proof (cases "is_singular_modgrp f")
  case True
  have [simp]: "modgrp_c f ≠ 0"
    using True by (simp add: is_singular_modgrp_altdef)
  have "dedekind_sum (modgrp_c f * m + modgrp_d f) (modgrp_c f) =
        dedekind_sum (modgrp_d f) (modgrp_c f)"
  proof (intro dedekind_sum_cong)
    have "[modgrp_c f * m + modgrp_d f = 0 + modgrp_d f] (mod modgrp_c
f)"
      by (intro cong_add) (auto simp: Cong.cong_def)
    thus "[modgrp_c f * m + modgrp_d f = modgrp_d f] (mod modgrp_c f)"
      by simp
  qed (use coprime_modgrp_c_d[of f] in ‹auto simp: Rings.coprime_commute›)
  thus ?thesis using True
    by (simp add: dedekind_eps_def add_divide_distrib ring_distribs is_singular_modgrp_time
          flip: cis_mult cis_divide)
next
  case False
  define n where "n = modgrp_b f"
  have f: "f = shift_modgrp n"
    unfolding n_def using False by (rule not_singular_modgrpD)
  have "f * shift_modgrp m = shift_modgrp (n + m)"
    by (simp add: shift_modgrp_add f)
  also have "ε ... = cis (pi * m / 12) * ε f"
    by (simp add: f dedekind_eps_def cis_mult ring_distribs add_divide_distrib
add_ac)
```

```
  finally show ?thesis .
qed

lemma dedekind_eps_shift_left [simp]: "ε (shift_modgrp m * f) = cis (pi
* m / 12) * ε f"
proof (cases "is_singular_modgrp f")
  case True
  have [simp]: "modgrp_c f ≠ 0"
    using True by (simp add: is_singular_modgrp_altdef)
  have a: "modgrp_a (shift_modgrp m * f) = modgrp_a f + m * modgrp_c f"
    unfolding shift_modgrp_code using modgrp.unimodular[of f] modgrp_c_nonneg[of
f]
    by (subst (3) modgrp_abcd [symmetric], subst times_modgrp_code) (auto
simp: modgrp_a_code algebra_simps)
  show ?thesis using True
    by (auto simp: dedekind_eps_def a add_divide_distrib ring_distribs
simp flip: cis_mult cis_divide)
next
  case False
  then obtain n where [simp]: "f = shift_modgrp n"
    using not_singular_modgrpD by blast
  show ?thesis
    by simp
qed

lemma dedekind_eps_S_right:
  assumes "is_singular_modgrp f" "modgrp_d f ≠ 0"
  shows   "ε (f * S_modgrp) = cis (-sgn (modgrp_d f) * pi / 4) * ε f"
proof -
  note [simp del] = div_mult_self3 div_mult_self4 div_mult_self2 div_mult_self1
  define a b c d where "a = modgrp_a f" "b = modgrp_b f" "c = modgrp_c
f" "d = modgrp_d f"
  have "c > 0"
    using assms modgrp_c_nonneg[of f] unfolding is_singular_modgrp_altdef
a_b_c_d_def by auto
  from assms have [simp]: "d ≠ 0"
    by (auto simp: a_b_c_d_def)
  have "coprime c d"
    unfolding a_b_c_d_def by (intro coprime_modgrp_c_d)
  have det: "a * d - b * c = 1"
    unfolding a_b_c_d_def by (rule modgrp_abcd_det)
  hence det': "a * d = b * c + 1"
    by linarith

  have "pole_modgrp f ≠ (0 :: real)"
    using assms by transfer (auto simp: modgrp_rel_def split: if_splits)
  hence sing: "is_singular_modgrp (f * S_modgrp)"
    using assms by (auto simp: is_singular_modgrp_times_iff)
```

```
  show ?thesis
  proof (cases d "0 :: int" rule: linorder_cases)
    case greater
    have [simp]: "modgrp_a (f * S_modgrp) = b"
      using greater unfolding a_b_c_d_def by transfer auto
    have [simp]: "modgrp_b (f * S_modgrp) = -a"
      using greater unfolding a_b_c_d_def by transfer auto
    have [simp]: "modgrp_c (f * S_modgrp) = d"
      using greater unfolding a_b_c_d_def by transfer auto
    have [simp]: "modgrp_d (f * S_modgrp) = -c"
      using greater unfolding a_b_c_d_def by transfer (auto split: if_splits)

    have "dedekind_sum (-c) d = -dedekind_sum c d"
      using ‹coprime c d› by (simp add: dedekind_sum_negate)
    also have "... = dedekind_sum d c - c / d / 12 - d / c / 12 + 1 /
4 - 1 / (12 * c * d)"
      using ‹c > 0› ‹d > 0› ‹coprime c d› by (subst dedekind_sum_reciprocity')
simp_all
    finally have *: "dedekind_sum (-c) d = ..." .
    have [simp]: "cnj (cis (pi / 4)) = 1 / cis (pi / 4)"
      by (subst divide_conv_cnj) auto

    have "ε (f * S_modgrp) = cis (pi * ((b - c) / (12 * d) + c / (12*d)
+
                                    d / (12*c) + 1 / (12 * c * d) - dedekind_sum
d c - 1 / 2))"
      unfolding dedekind_eps_def a_b_c_d_def [symmetric] using ‹d > 0›
sing assms
      by (simp add: * algebra_simps)
    also have "(b - c) / (12 * d) + c / (12*d) + d / (12*c) + 1 / (12
* c * d) =
               (b * c + 1 + d * d) / (12 * c * d)"
      using ‹c > 0› ‹d > 0› by (simp add: field_simps)
    also have "b * c + 1 = a * d"
      using det by (simp add: algebra_simps)
    also have "(a * d + d * d) / (12 * c * d) = (a + d) / (12 * c)"
      using ‹c > 0› ‹d > 0› by (simp add: field_simps)
    also have "cis (pi * ((a + d) / (12 * c) - dedekind_sum d c - 1 /
2)) =
                cis (-pi / 4) * ε f"
      unfolding dedekind_eps_def a_b_c_d_def [symmetric] using ‹d > 0›
sing assms
      by (auto simp: cis_mult algebra_simps diff_divide_distrib add_divide_distrib)
    finally show ?thesis
      using ‹d > 0› by (simp add: a_b_c_d_def)

  next
    case less
    have [simp]: "modgrp_a (f * S_modgrp) = -b"
```

```
      using less unfolding a_b_c_d_def by transfer (auto split: if_splits)
    have [simp]: "modgrp_b (f * S_modgrp) = a"
      using less unfolding a_b_c_d_def by transfer (auto split: if_splits)
    have [simp]: "modgrp_c (f * S_modgrp) = -d"
      using less unfolding a_b_c_d_def by transfer (auto split: if_splits)
    have [simp]: "modgrp_d (f * S_modgrp) = c"
      using less unfolding a_b_c_d_def by transfer (auto split: if_splits)

    have "dedekind_sum c (-d) =
            -dedekind_sum (-d) c - c / d / 12 - d / c / 12 - 1 / 4 - 1
/ (12 * c * d)"
      using ‹c > 0› ‹d < 0› ‹coprime c d› by (subst dedekind_sum_reciprocity')
simp_all
    also have "-dedekind_sum (-d) c = dedekind_sum d c"
      using ‹coprime c d› by (subst dedekind_sum_negate) (auto simp: Rings.coprime_commute)
    finally have *: "dedekind_sum c (-d) =
                      dedekind_sum d c - c / d / 12 - d / c / 12 - 1 /
4 - 1 / (12 * c * d)" .

    have "ε (f * S_modgrp) =
            cis (pi * (c / (12 * d) + d / (12 * c) + 1 / (12 * c * d)
- (c - b) / (12 * d) -
                       dedekind_sum d c))"
      unfolding dedekind_eps_def a_b_c_d_def [symmetric] using ‹d < 0›
sing assms
      by (simp add: * algebra_simps)
    also have "c / (12 * d) + d / (12 * c) + 1 / (12 * c * d) - (c - b)
/ (12 * d) =
               (d * d + (1 + b * c)) / (12 * c * d)"
      using ‹c > 0› ‹d < 0› by (simp add: field_simps)
    also have "1 + b * c = a * d"
      using det by (simp add: algebra_simps)
    also have "(d * d + a * d) / (12 * c * d) = (a + d) / (12 * c)"
      using ‹c > 0› ‹d < 0› by (simp add: field_simps)
    also have "cis (pi * ((a + d) / (12 * c) - dedekind_sum d c)) = cis
(pi / 4) * ε f"
      unfolding dedekind_eps_def a_b_c_d_def [symmetric] using ‹d < 0›
sing assms
      by (auto simp: cis_mult algebra_simps diff_divide_distrib add_divide_distrib)
    finally show ?thesis
      using ‹d < 0› by (simp add: a_b_c_d_def)
  qed auto
qed

lemma dedekind_eps_root_of_unity: "ε f ^ 24 = 1"
proof -
  have not_sing: "ε f ^ 24 = 1" if "¬is_singular_modgrp f" for f
  proof -
    have "ε f ^ 24 = cis (2 * (pi * real_of_int (modgrp_b f)))"
```

```
      using that by (auto simp: dedekind_eps_def Complex.DeMoivre)
    also have "2 * (pi * real_of_int (modgrp_b f)) = 2 * pi * real_of_int
(modgrp_b f)"
      by (simp add: mult_ac)
    also have "cis ... = 1"
      by (rule cis_multiple_2pi) auto
    finally show ?thesis .
  qed

  show ?thesis
  proof (induction f rule: modgrp_induct_S_shift')
    case (S f)
    show ?case
    proof (cases "modgrp_d f = 0")
      case True
      hence "ε (f * S_modgrp) ^ 24 = cis (real_of_int (modgrp_b (f * S_modgrp))
* (2 * pi))"
        by (simp add: dedekind_eps_def Complex.DeMoivre mult_ac)
      also have "... = 1"
        by (subst cis_power_int [symmetric]) auto
      finally show ?thesis
        by simp
    next
      case d: False
      show ?thesis
      proof (cases "is_singular_modgrp f")
        case sing: True
        have "ε (f * S_modgrp) ^ 24 = cis (- (pi * (real_of_int (sgn (modgrp_d
f)) * 6)))"
          using d sing by (simp add: dedekind_eps_S_right field_simps
Complex.DeMoivre S)
        also have "- (pi * (real_of_int (sgn (modgrp_d f)) * 6)) = 2 *
pi * of_int (-3 * sgn (modgrp_d f))"
          by simp
        also have "cis ... = 1"
          by (rule cis_multiple_2pi) auto
        finally show ?thesis .
      next
        case False
        then obtain n where [simp]: "f = shift_modgrp n"
          using not_singular_modgrpD by blast
        have "ε (f * S_modgrp) ^ 24 = cis (pi * (real_of_int n * 2) -
pi * 6)"
          by (simp add: algebra_simps Complex.DeMoivre cis_mult)
        also have "pi * (real_of_int n * 2) - pi * 6 = (real_of_int (n-3)
* (2 * pi))"
          by (simp add: algebra_simps)
        also have "cis ... = 1"
          by (subst cis_power_int [symmetric]) auto
```

```
      finally show ?thesis .
    qed
  qed
next
  case (shift f n)
  have "ε (f * shift_modgrp n) ^ 24 = cis (of_int n * (2 * pi))"
    by (simp add: power_mult_distrib shift Complex.DeMoivre mult_ac)
  also have "... = 1"
    by (subst cis_power_int [symmetric]) auto
  finally show ?case .
qed auto
qed
```

The following theorem is Apostol's Theorem 3.4: the general functional equation for Dedekind's η function.

Our version is actually more general than Apostol's lemma since it also holds for modular groups with $c = 0$ (i.e. shifts, i.e. T^n). We also use a slightly different definition of ε though, namely the one from Wikipedia. This makes the functional equation look a bit nicer than Apostol's version.

```
theorem dedekind_eta_apply_modgrp:
  assumes "Im z > 0"
  shows   "η (apply_modgrp f z) = ε f * csqrt (modgrp_factor f z) * η
z"
  using assms
proof (induction f arbitrary: z rule: modgrp_induct_S_shift')
  case id
  thus ?case by simp
next
  case (shift f n z)
  have "η (apply_modgrp (f * shift_modgrp n) z) = η (apply_modgrp f (z
+ of_int n))"
    using shift.prems by (subst apply_modgrp_mult) auto
  also have "... = ε f * csqrt (modgrp_factor f (z + of_int n)) * η (z
+ of_int n)"
    using shift.prems by (subst shift.IH) auto
  also have "η (z + of_int n) = cis (pi * n / 12) * η z"
    using shift.prems by (subst dedekind_eta_plus_int) auto
  also have "ε f * csqrt (modgrp_factor f (z + of_int n)) * (cis (pi *
n / 12) * η z) =
             ε (f * shift_modgrp n) * csqrt (modgrp_factor (f * shift_modgrp
n) z) * η z"
    by simp
  finally show ?case .
next
  case (S f z)
  note [simp del] = div_mult_self3 div_mult_self4 div_mult_self2 div_mult_self1
  define a b c d where "a = modgrp_a f" "b = modgrp_b f" "c = modgrp_c
f" "d = modgrp_d f"
```

```
  have det: "a * d - b * c = 1"
    using modgrp_abcd_det[of f] by (simp add: a_b_c_d_def)
  from S.prems have [simp]: "z ≠ 0"
    by auto
  show ?case
  proof (cases "is_singular_modgrp f")
    case False
    hence f: "f = shift_modgrp b"
      unfolding a_b_c_d_def by (rule not_singular_modgrpD)
    have *: "f * S_modgrp = modgrp b (-1) 1 0"
      unfolding f shift_modgrp_code S_modgrp_code times_modgrp_code by
simp
    have [simp]: "modgrp_a (f * S_modgrp) = b"
                 "modgrp_b (f * S_modgrp) = -1"
                 "modgrp_c (f * S_modgrp) = 1"
                 "modgrp_d (f * S_modgrp) = 0"
      by (simp_all add: * modgrp_a_code modgrp_b_code modgrp_c_code modgrp_d_code)
    have eps: "ε (f * S_modgrp) = cis (pi * (b / 12 - 1 / 4))"
      by (simp add: dedekind_eps_def dedekind_sum_def is_singular_modgrp_altdef)

    have "η (apply_modgrp (f * S_modgrp) z) = η (-1 / z + of_int b)"
      using S.prems by (subst apply_modgrp_mult) (auto simp: f algebra_simps)
    also have "... = cis (pi * b / 12) * η (-(1 / z))"
      using S.prems by (subst dedekind_eta_plus_int) auto
    also have "... = cis (pi * b / 12) * csqrt (- i * z) * η z"
      using S.prems by (subst dedekind_eta_minus_one_over) auto
    also have "... = cis (pi / 4) * csqrt (- i * z) * ε (shift_modgrp b
* S_modgrp) * η z"
      using eps by (auto simp: f ring_distribs simp flip: cis_divide)
    also have "csqrt (-i * z) = rcis (norm (csqrt (-i * z))) (Arg (csqrt
(-i * z)))"
      by (rule rcis_cmod_Arg [symmetric])
    also have "... = rcis (sqrt (cmod z)) (Arg (- (i * z)) / 2)"
      by (simp add: norm_mult)
    also have "cis (pi / 4) * ... = rcis (sqrt (norm z)) ((Arg (-i*z) +
pi / 2) / 2)"
      by (simp add: rcis_def cis_mult add_divide_distrib algebra_simps)
    also have "Arg (-i*z) + pi / 2 = Arg z"
    proof (rule cis_Arg_unique [symmetric])
      have "cis (Arg (-i * z) + pi / 2) = - (sgn (i * z) * i)"
        by (simp flip: cis_mult add: cis_Arg)
      also have "... = sgn z"
        by (simp add: complex_sgn_def scaleR_conv_of_real field_simps
norm_mult)
      finally show "sgn z = cis (Arg (-i * z) + pi / 2)" ..
    next
      show "-pi < Arg (-i * z) + pi / 2" "Arg (- i * z) + pi / 2 ≤ pi"
        using Arg_Re_pos[of "-i * z"] S.prems by auto
    qed
```

```
    also have "rcis (sqrt (norm z)) (Arg z / 2) = rcis (norm (csqrt z))
(Arg (csqrt z))"
      by simp
    also have "... = csqrt z"
      by (rule rcis_cmod_Arg)
    finally show ?thesis
      by (simp add: f)
  next
    case sing: True
    hence "c > 0"
      unfolding a_b_c_d_def by (meson is_singular_modgrp_altdef modgrp_cd_signs)
    have "Im (1 / z) < 0"
      using S.prems Im_one_over_neg_iff by blast
    have Arg_z: "Arg z ∈ {0<..<pi}"
      using S.prems by (simp add: Arg_lt_pi)
    have Arg_z': "Arg (-i * z) = -pi/2 + Arg z"
      using Arg_z by (subst Arg_times) auto
    have [simp]: "Arg (-z) = Arg z - pi"
      using Arg_z by (subst Arg_minus) auto

    show ?thesis
    proof (cases d "0 :: int" rule: linorder_cases)
      case equal
      hence *: "¬is_singular_modgrp (f * S_modgrp)"
        unfolding a_b_c_d_def
        by transfer (auto simp: modgrp_rel_def split: if_splits)
      define n where "n = modgrp_b (f * S_modgrp)"
      have **: "f * S_modgrp = shift_modgrp n"
        unfolding n_def using * by (rule not_singular_modgrpD)
      show ?thesis using S.prems
        by (simp add: ** dedekind_eta_plus_int)
    next
      case greater
      have "modgrp a b c d * S_modgrp = modgrp b (-a) d (-c)"
        unfolding shift_modgrp_code S_modgrp_code times_modgrp_code det
by simp
      hence *: "f * S_modgrp = modgrp b (-a) d (-c)"
        by (simp add: a_b_c_d_def)
      have [simp]: "modgrp_a (f * S_modgrp) = b" "modgrp_b (f * S_modgrp)
= -a"
                   "modgrp_c (f * S_modgrp) = d" "modgrp_d (f * S_modgrp)
= -c"
        unfolding * modgrp_a_code modgrp_b_code modgrp_c_code modgrp_d_code
        using greater det by auto

      have "η (apply_modgrp (f * S_modgrp) z) = η (apply_modgrp f (- (1
/ z)))"
        using S.prems by (subst apply_modgrp_mult) auto
      also have "... = ε f * csqrt (modgrp_factor f (- (1 / z))) * η (-
```

```
(1 / z))"
        using S.prems by (subst S.IH) auto
      also have "modgrp_factor f (- (1 / z)) = d - c / z"
        unfolding modgrp_factor_def by (simp add: a_b_c_d_def)
      also have "η (- (1 / z)) = csqrt (-i * z) * η z"
        using S.prems by (subst dedekind_eta_minus_one_over) auto
      also have "ε f * csqrt (d - c / z) * (csqrt (-i * z) * η z) =
                 (csqrt (d - c / z) * csqrt (-i * z)) * ε f * η z"
        by (simp add: mult_ac)
      also have "csqrt (d - c / z) * csqrt (-i * z) = csqrt ((d - c / z)
* (-i * z))"
      proof (rule csqrt_mult [symmetric])
        have "Im (of_int d - of_int c / z) = -of_int c * Im (1 / z)"
          by (simp add: Im_divide)
        hence Im: "Im (of_int d - of_int c / z) > 0"
          using ‹Im (1 / z) < 0› ‹c > 0› by (auto simp: mult_less_0_iff)
        hence Arg_pos: "Arg (of_int d - of_int c / z) > 0"
          using Arg_pos_iff by blast

        have "Arg (of_int d - of_int c / z) + Arg z ≤ 3 / 2 * pi"
        proof (cases "Re z ≥ 0")
          case True
          have "Arg (of_int d - of_int c / z) ≤ pi"
            by (rule Arg_le_pi)
          moreover have "Arg z ≤ pi / 2"
            using Arg_Re_nonneg[of z] True by auto
          ultimately show ?thesis by simp
        next
          case False
          have "Re (of_int d - of_int c / z) = of_int d - of_int c * Re
z / norm z ^ 2"
            by (simp add: Re_divide norm_complex_def)
          also have "... ≥ 0 - 0"
            using ‹d > 0› ‹c > 0› False
            by (intro diff_mono divide_nonpos_pos mult_nonneg_nonpos)
auto
          finally have "Arg (of_int d - of_int c / z) ≤ pi / 2"
            using Arg_Re_nonneg[of "of_int d - of_int c / z"] by simp
          moreover have "Arg z ≤ pi"
            by (rule Arg_le_pi)
          ultimately show ?thesis by simp
        qed
        moreover have "Arg (of_int d - of_int c / z) + Arg z > 0 + 0"
          using Arg_z by (intro add_strict_mono Arg_pos) auto
        ultimately show "Arg (of_int d - of_int c / z) + Arg (- i * z)
∈ {-pi<..pi}"
          using Arg_z' by auto
      qed
      also have "(d - c / z) * (-i * z) = (-i) * (d * z - c)"
```

```
      using S.prems by (auto simp: field_simps)
    also have "csqrt ... = csqrt (-i) * csqrt (d * z - c)"
    proof (intro csqrt_mult)
      have "Arg (of_int d * z - of_int c) > 0"
        using ‹d > 0› S.prems by (subst Arg_pos_iff) auto
      moreover have "Arg (of_int d * z - of_int c) ≤ pi"
        by (rule Arg_le_pi)
      ultimately show "Arg (-i) + Arg (of_int d * z - of_int c) ∈ {-pi<..pi}"
        by auto
    qed
    also have "csqrt (-i) = cis (-pi / 4)"
      by (simp add: csqrt_exp_Ln cis_conv_exp)
    also have "cis (-pi / 4) * csqrt (d * z - c) * ε f * η z =
               ε (f * S_modgrp) * csqrt (d * z - c) * η z"
      using ‹d > 0› sing by (subst dedekind_eps_S_right) (auto simp:
a_b_c_d_def)
    also have "... = ε (f * S_modgrp) * csqrt (modgrp_factor (f * S_modgrp)
z) * η z"
      unfolding modgrp_factor_def by simp
    finally show ?thesis .
  next
    case less
    have "modgrp a b c d * S_modgrp = modgrp b (-a) d (-c)"
      unfolding shift_modgrp_code S_modgrp_code times_modgrp_code det
by simp
    hence *: "f * S_modgrp = modgrp b (-a) d (-c)"
      by (simp add: a_b_c_d_def)
    have [simp]: "modgrp_a (f * S_modgrp) = -b" "modgrp_b (f * S_modgrp)
= a"
                 "modgrp_c (f * S_modgrp) = -d" "modgrp_d (f * S_modgrp)
= c"
      unfolding * modgrp_a_code modgrp_b_code modgrp_c_code modgrp_d_code
      using less det by auto

    have "η (apply_modgrp (f * S_modgrp) z) = η (apply_modgrp f (- (1
/ z)))"
      using S.prems by (subst apply_modgrp_mult) auto
    also have "... = ε f * csqrt (modgrp_factor f (- (1 / z))) * η (-
(1 / z))"
      using S.prems by (subst S.IH) auto
    also have "modgrp_factor f (- (1 / z)) = d - c / z"
      unfolding modgrp_factor_def by (simp add: a_b_c_d_def)
    also have "η (- (1 / z)) = csqrt (-i * z) * η z"
      using S.prems by (subst dedekind_eta_minus_one_over) auto
    also have "ε f * csqrt (d - c / z) * (csqrt (-i * z) * η z) =
               (csqrt (d - c / z) * csqrt (-i * z)) * ε f * η z"
      by (simp add: mult_ac)
    also have "csqrt (-i * z) = csqrt (i * -z)"
      by simp
```

```
      also have "... = csqrt i * csqrt (-z)"
      proof (rule csqrt_mult)
        show "Arg i + Arg (- z) ∈ {- pi<..pi}"
          using Arg_z by auto
      qed
      also have "csqrt (d - c / z) * ... = csqrt i * (csqrt (d - c / z)
* csqrt (-z))"
        by (simp add: mult_ac)
      also have "csqrt (d - c / z) * csqrt (-z) = csqrt ((d - c / z) *
(-z))"
      proof (rule csqrt_mult [symmetric])
        have "Im (of_int d - of_int c / z) = of_int c * Im z / norm z
^ 2"
          by (simp add: Im_divide norm_complex_def)
        also have "... > 0"
          using S.prems ‹c > 0› by (intro mult_pos_pos divide_pos_pos)
auto
        finally have "Arg (of_int d - of_int c / z) ∈ {0<..<pi}"
          using Arg_lt_pi[of "of_int d - of_int c / z"] by auto
        thus "Arg (of_int d - of_int c / z) + Arg (- z) ∈ {- pi<..pi}"
          using Arg_z by auto
      qed
      also have "(d - c / z) * (-z) = c - d * z"
        using S.prems by (simp add: field_simps)
      also have "csqrt i = cis (pi / 4)"
        by (simp add: csqrt_exp_Ln complex_eq_iff cos_45 sin_45 field_simps)
      also have "cis (pi / 4) * csqrt (c - d * z) * ε f * η z =
                 ε (f * S_modgrp) * csqrt (c - d * z) * η z"
        using ‹d < 0› sing by (subst dedekind_eps_S_right) (auto simp:
a_b_c_d_def)
      also have "... = ε (f * S_modgrp) * csqrt (modgrp_factor (f * S_modgrp)
z) * η z"
        unfolding modgrp_factor_def by simp
      finally show ?thesis .
    qed
  qed
qed

no_notation dedekind_eta ("η")
no_notation dedekind_eps ("ε")

end
```

11.5 The transformation law for G_2

```
theory Eisenstein_G2
  imports Dedekind_Eta
begin
```

In his book, Apostol derives the inversion law for G_2 in the exercises to

Chapter 3 and remarks that it leads to a proof of the inversion law for η. Since we already have a nice and short proof for the inversion law for η, we instead go the other direction. We differentiate the inversion law for η and easily obtain the corresponding law for G_2

```
theorem Eisenstein_G2_minus_one_over:
  assumes t: "Im t > 0"
  shows   "Eisenstein_G 2 (-(1/t)) = t^2 * Eisenstein_G 2 t - 2 * pi *
i * t"
proof -
  write dedekind_eta ("η")
  from assms have "t ≠ 0"
    by auto
  note [derivative_intros] = has_field_derivative_dedekind_eta
  have "deriv (λt. η (-(1/t))) t =
          i * Eisenstein_G 2 (-(1 / t)) * η (-(1 / t)) / (4 * pi * t^2)"
    by (rule DERIV_imp_deriv) (use t in ‹auto intro!: derivative_eq_intros
simp: power2_eq_square›)
  also have "η (-(1 / t)) = csqrt (- (i * t)) * η t"
    by (subst dedekind_eta_minus_one_over) (use t in auto)
  finally have "i * η t * csqrt (-i*t) / (4 * pi * t^2) * Eisenstein_G 2
(-(1 / t)) =
                deriv (λt. η (-(1/t))) t"
    by simp

  also have "deriv (λt. η (-(1/t))) t = deriv (λt. csqrt (-(i*t)) * η
t) t"
  proof (intro deriv_cong_ev refl)
    have "eventually (λz. z ∈ {z. Im z > 0}) (nhds t)"
      by (rule eventually_nhds_in_open) (use t in ‹auto simp: open_halfspace_Im_gt›)
    thus "∀F x in nhds t. η (- (1 / x)) = csqrt (- (i * x)) * η x"
      by eventually_elim (subst dedekind_eta_minus_one_over, auto)
  qed
  also have "deriv (λt. csqrt (- (i * t)) * η t) t =
               i * η t * (Eisenstein_G 2 t * csqrt (-i*t) / (4 * pi) -
1 / (2 * csqrt (-i*t)))"
    by (rule DERIV_imp_deriv)
       (use t in ‹auto intro!: derivative_eq_intros simp: complex_nonpos_Reals_iff
field_simps›)
  also have "1 / (2 * csqrt (-i*t)) = csqrt (-i*t) / (2 * (-i * t))"
  proof -
    have *: "-i * t = csqrt (-i * t) ^ 2"
      by simp
    show ?thesis
      by (subst (3) *, unfold power2_eq_square) (use ‹t ≠ 0› in ‹auto
simp: field_simps›)
  qed
  also have "i * η t * (Eisenstein_G 2 t * csqrt (-i*t) / (4 * pi) - ...)
=
             i * η t * csqrt (-i * t) / (4 * pi) * (Eisenstein_G 2 t +
```

```
2 * pi / (i*t))"
    by (simp add: field_simps)
  also have "... = i * η t * csqrt (-i * t) / (4 * pi * t^2) * (t^2 * Eisenstein_G
2 t - 2 * i * pi * t)"
    using ‹t ≠ 0› by (simp add: field_simps power2_eq_square)
  finally show "Eisenstein_G 2 (-(1 / t)) = t^2 * Eisenstein_G 2 t - 2
* pi * i * t"
    by (subst (asm) mult_cancel_left) (use t in auto)
qed

lemma Eisenstein_E2_minus_one_over:
  assumes t: "Im t > 0"
  shows   "Eisenstein_E 2 (-(1/t)) = t^2 * Eisenstein_E 2 t - 6 * i / pi
* t"
  using assms
  by (simp add: Eisenstein_E_def Eisenstein_G2_minus_one_over[OF t]
                zeta_2 power2_eq_square field_simps)
```

In a similar fashion to the η function, we can prove the general modular transformation law for G_2:

```
theorem Eisenstein_G2_apply_modgrp:
  assumes "Im z > 0"
  shows   "Eisenstein_G 2 (apply_modgrp f z) =
             modgrp_factor f z ^ 2 * Eisenstein_G 2 z -
             2 * i * pi * modgrp_c f * modgrp_factor f z"
  using assms
proof (induction f arbitrary: z rule: modgrp_induct_S_shift')
  case id
  thus ?case by simp
next
  case (shift f n z)
  have "Eisenstein_G 2 (apply_modgrp (f * shift_modgrp n) z) =
          Eisenstein_G 2 (apply_modgrp f (z + of_int n))"
    using shift.prems by (subst apply_modgrp_mult) auto
  also have "... = (modgrp_factor f (z + of_int n))^2 * Eisenstein_G 2 (z
+ of_int n) -
                   2 * i * of_real pi * of_int (modgrp_c f) * modgrp_factor
f (z + of_int n)"
    using shift.prems by (subst shift.IH) auto
  also have "Eisenstein_G 2 (z + of_int n) = Eisenstein_G 2 z"
    by (rule Eisenstein_G_plus_int)
  finally show ?case
    by simp
next
  case (S f z)
  note [simp del] = div_mult_self3 div_mult_self4 div_mult_self2 div_mult_self1
  define a b c d where "a = modgrp_a f" "b = modgrp_b f" "c = modgrp_c
f" "d = modgrp_d f"
  have det: "a * d - b * c = 1"
```

```
    using modgrp_abcd_det[of f] by (simp add: a_b_c_d_def)

  from S.prems have [simp]: "z ≠ 0"
    by auto
  show ?case
  proof (cases "is_singular_modgrp f")
    case False
    hence f: "f = shift_modgrp b"
      unfolding a_b_c_d_def by (rule not_singular_modgrpD)
    have *: "f * S_modgrp = modgrp b (-1) 1 0"
      unfolding f shift_modgrp_code S_modgrp_code times_modgrp_code by
simp
    have [simp]: "modgrp_a (f * S_modgrp) = b"
                 "modgrp_b (f * S_modgrp) = -1"
                 "modgrp_c (f * S_modgrp) = 1"
                 "modgrp_d (f * S_modgrp) = 0"
      by (simp_all add: * modgrp_a_code modgrp_b_code modgrp_c_code modgrp_d_code)

    have "Eisenstein_G 2 (apply_modgrp (f * S_modgrp) z) = Eisenstein_G
2 (-1 / z + of_int b)"
      using S.prems by (subst apply_modgrp_mult) (auto simp: f algebra_simps)
    also have "... = Eisenstein_G 2 (-(1 / z))"
      by (subst Eisenstein_G_plus_int) auto
    also have "... = z^2 * Eisenstein_G 2 z - 2 * pi * i * z"
      by (subst Eisenstein_G2_minus_one_over) (use S.prems in auto)
    also have "... = (modgrp_factor (f * S_modgrp) z)^2 * Eisenstein_G 2
z -
                     2 * i * pi * complex_of_int (modgrp_c (f * S_modgrp))
* modgrp_factor (f * S_modgrp) z"
      by (simp add: modgrp_factor_def)
    finally show ?thesis .
  next
    case sing: True
    hence "c > 0"
      unfolding a_b_c_d_def by (meson is_singular_modgrp_altdef modgrp_cd_signs)
    have "Im (1 / z) < 0"
      using S.prems Im_one_over_neg_iff by blast
    have Arg_z: "Arg z ∈ {0<..<pi}"
      using S.prems by (simp add: Arg_lt_pi)
    have Arg_z': "Arg (-i * z) = -pi/2 + Arg z"
      using Arg_z by (subst Arg_times) auto
    have [simp]: "Arg (-z) = Arg z - pi"
      using Arg_z by (subst Arg_minus) auto

    show ?thesis
    proof (cases d "0 :: int" rule: linorder_cases)
      case equal
      hence *: "¬is_singular_modgrp (f * S_modgrp)"
        unfolding a_b_c_d_def
```

```
      by transfer (auto simp: modgrp_rel_def split: if_splits)
    define n where "n = modgrp_b (f * S_modgrp)"
    have **: "f * S_modgrp = shift_modgrp n"
      unfolding n_def using * by (rule not_singular_modgrpD)
    show ?thesis using S.prems
      by (simp add: ** Eisenstein_G_plus_int)
  next
    case greater
    have "modgrp a b c d * S_modgrp = modgrp b (-a) d (-c)"
      unfolding shift_modgrp_code S_modgrp_code times_modgrp_code det
by simp
    hence *: "f * S_modgrp = modgrp b (-a) d (-c)"
      by (simp add: a_b_c_d_def)
    have [simp]: "modgrp_a (f * S_modgrp) = b" "modgrp_b (f * S_modgrp)
= -a"
                 "modgrp_c (f * S_modgrp) = d" "modgrp_d (f * S_modgrp)
= -c"
      unfolding * modgrp_a_code modgrp_b_code modgrp_c_code modgrp_d_code
      using greater det by auto
    define F where "F = modgrp_factor (f * S_modgrp) z"

    have "Eisenstein_G 2 (apply_modgrp (f * S_modgrp) z) =
            Eisenstein_G 2 (apply_modgrp f (- (1 / z)))"
      using S.prems by (subst apply_modgrp_mult) auto
    also have "... = (modgrp_factor f (- (1 / z)))^2 * Eisenstein_G 2
(-(1 / z)) -
                     2 * i * complex_of_real pi * c * modgrp_factor f
(- (1 / z))"
      using S.prems by (subst S.IH) (auto simp flip: a_b_c_d_def)
    also have "modgrp_factor f (- (1 / z)) = F / z"
      unfolding F_def modgrp_factor_def by (simp add: a_b_c_d_def field_simps)
    also have "Eisenstein_G 2 (-(1 / z)) = z^2 * Eisenstein_G 2 z - 2
* pi * i * z"
      by (subst Eisenstein_G2_minus_one_over) (use S.prems in auto)
    also have "(F / z)^2 * (z^2 * Eisenstein_G 2 z - of_real (2 * pi)
* i * z) =
               F ^ 2 * Eisenstein_G 2 z - 2 * pi * i * F ^ 2 / z"
      using S.prems by (simp add: field_simps power2_eq_square modgrp_factor_def
F_def)
    also have "F^2 * Eisenstein_G 2 z - of_real (2 * pi) * i * F^2 / z
-
                 2 * i * of_real pi * of_int c * (F / z) =
               F^2 * Eisenstein_G 2 z - 2 * pi * i * ((F^2 + of_int c
* F) / z)"
      by (simp add: field_simps)
    also have "(F^2 + of_int c * F) / z = of_int (modgrp_c (f * S_modgrp))
* F"
      by (simp add: F_def modgrp_factor_def field_simps power2_eq_square
flip: modgrp_c_def)
```

```
      finally show ?thesis
        unfolding F_def by simp
    next
      case less
      have "modgrp a b c d * S_modgrp = modgrp b (-a) d (-c)"
        unfolding shift_modgrp_code S_modgrp_code times_modgrp_code det
by simp
      hence *: "f * S_modgrp = modgrp b (-a) d (-c)"
        by (simp add: a_b_c_d_def)
      have [simp]: "modgrp_a (f * S_modgrp) = -b" "modgrp_b (f * S_modgrp)
= a"
                   "modgrp_c (f * S_modgrp) = -d" "modgrp_d (f * S_modgrp)
= c"
        unfolding * modgrp_a_code modgrp_b_code modgrp_c_code modgrp_d_code
        using less det by auto
      define F where "F = modgrp_factor (f * S_modgrp) z"

      have "Eisenstein_G 2 (apply_modgrp (f * S_modgrp) z) =
              Eisenstein_G 2 (apply_modgrp f (- (1 / z)))"
        using S.prems by (subst apply_modgrp_mult) auto
      also have "... = (modgrp_factor f (- (1 / z)))^2 * Eisenstein_G 2
(-(1 / z)) -
                       2 * i * complex_of_real pi * c * modgrp_factor f
(- (1 / z))"
        using S.prems by (subst S.IH) (auto simp flip: a_b_c_d_def)
      also have "modgrp_factor f (- (1 / z)) = -F / z"
        unfolding F_def modgrp_factor_def by (simp add: a_b_c_d_def field_simps)
      also have "Eisenstein_G 2 (-(1 / z)) = z^2 * Eisenstein_G 2 z - 2
* pi * i * z"
        by (subst Eisenstein_G2_minus_one_over) (use S.prems in auto)
      also have "(-F / z)^2 * (z^2 * Eisenstein_G 2 z - of_real (2 * pi)
* i * z) =
                 F ^ 2 * Eisenstein_G 2 z - 2 * pi * i * F ^ 2 / z"
        using S.prems by (simp add: field_simps power2_eq_square modgrp_factor_def
F_def)
      also have "F^2 * Eisenstein_G 2 z - of_real (2 * pi) * i * F^2 / z
-
                   2 * i * of_real pi * of_int c * (-F / z) =
                 F^2 * Eisenstein_G 2 z - 2 * pi * i * ((F^2 - of_int c
* F) / z)"
        by (simp add: field_simps)
      also have "(F^2 - of_int c * F) / z = of_int (modgrp_c (f * S_modgrp))
* F"
        by (simp add: F_def modgrp_factor_def field_simps power2_eq_square
flip: modgrp_c_def)
      finally show ?thesis
        unfolding F_def by simp
    qed
  qed
```

qed

```
lemma Eisenstein_E2_apply_modgrp:
  assumes "Im z > 0"
  shows   "Eisenstein_E 2 (apply_modgrp f z) =
             modgrp_factor f z ^ 2 * Eisenstein_E 2 z - 6 * i / pi * modgrp_c
f * modgrp_factor f z"
  unfolding Eisenstein_E_def
  by (simp add: Eisenstein_G2_apply_modgrp[OF assms] power2_eq_square
zeta_2 field_simps)
```

We can now also easily derive the values $G_2(i) = \pi$ and $G_2(\rho) = \frac{2\pi}{\sqrt{3}}$ using the same technique we used earlier for general G_k with $k \geq 3$.

```
lemma Eisenstein_G2_ii: "Eisenstein_G 2 i = of_real pi"
  using Eisenstein_G2_minus_one_over[of "i"] by simp

lemma Eisenstein_E2_ii: "Eisenstein_E 2 i = 3 / of_real pi"
  by (simp add: Eisenstein_G2_ii Eisenstein_E_def zeta_2 power2_eq_square)

lemma Eisenstein_G2_rho: "Eisenstein_G 2 modfun_rho = of_real (2 / sqrt
3 * pi)"
proof -
  have "Eisenstein_G 2 (- (1 / modfun_rho)) = modfun_rho^2 * Eisenstein_G
2 modfun_rho -
          complex_of_real (2 * pi) * i * modfun_rho"
    using Eisenstein_G2_minus_one_over[of modfun_rho] by simp
  also have "- (1 / modfun_rho) = modfun_rho + 1"
    by (auto simp: modfun_rho_altdef field_simps simp flip: of_real_mult)
  also have "modfun_rho ^ 2 = -(modfun_rho + 1)"
    by (auto simp: modfun_rho_altdef field_simps power2_eq_square simp
flip: of_real_mult)
  also have "Eisenstein_G 2 (modfun_rho + 1) = Eisenstein_G 2 modfun_rho"
    using Eisenstein_G_plus_int[of 2 "modfun_rho" 1] by simp
  finally have *: "(modfun_rho + 2) * Eisenstein_G 2 modfun_rho = -2 *
i * pi * modfun_rho"
    unfolding of_real_mult of_real_numeral by Groebner_Basis.algebra

  have "modfun_rho + 2 ≠ 0"
    by (auto simp: modfun_rho_altdef complex_eq_iff)
  hence "Eisenstein_G 2 modfun_rho = (-2 * i * pi * modfun_rho) / (modfun_rho
+ 2)"
    by (subst * [symmetric]) auto
  also have "(-2 * i * pi * modfun_rho) / (modfun_rho + 2) = of_real (2
/ sqrt 3 * pi)"
    by (auto simp: complex_eq_iff modfun_rho_altdef Re_divide Im_divide
field_simps)
  finally show ?thesis .
qed
```

```
lemma Eisenstein_E2_rho: "Eisenstein_E 2 modfun_rho = of_real (2 * sqrt
3 / pi)"
  by (simp add: Eisenstein_G2_rho Eisenstein_E_def zeta_2 power2_eq_square
field_simps
           flip: of_real_mult[of "sqrt 3" "sqrt 3"])

end
```

References

[1] T. M. Apostol. *Modular Functions and Dirichlet Series in Number Theory*. Graduate Texts in Mathematics. Springer New York, 1990.

[2] S. Lang. *Elliptic Functions*. Graduate Texts in Mathematics. Springer New York, 1973.