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"
  ⟨*proof*⟩

lemma *eventually_at_ii_inf_iff:*
  "(∀F z in at_ii_inf. P z) ⟷ (∃c. ∀z. Im z > c ⟶ P z)"
  ⟨*proof*⟩

lemma *eventually_at_ii_inf_iff':*
  "(∀F z in at_ii_inf. P z) ⟷ (∃c. ∀z. Im z ≥ c ⟶ P z)"
  ⟨*proof*⟩

lemma *filterlim_Im_at_ii_inf: "filterlim Im at_top at_i∞"*
  ⟨*proof*⟩

lemma *filterlim_at_ii_infI:*
  assumes *"filterlim f F at_top"*
  shows *"filterlim (λx. f (Im x)) F at_i∞"*
  ⟨*proof*⟩

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

⟨*proof*⟩

lemma *at_ii_inf_neq_bot [simp]: "at_ii_inf ≠ bot"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *to_q_conv_to_nome: "to_q n z = to_nome (2 * z / of_nat n)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

interpretation *to_q: periodic_fun_simple "to_q n" "of_nat n"*
⟨*proof*⟩

lemma *to_q_of_nat_period [simp]: "to_q n (of_nat n) = 1"*
⟨*proof*⟩

lemma *to_q_of_int [simp]:*
assumes *"int n dvd m"*
shows *"to_q n (of_int m) = 1"*
⟨*proof*⟩

lemma *to_q_of_nat [simp]:*
assumes *"n dvd m"*
shows *"to_q n (of_nat m) = 1"*
⟨*proof*⟩

lemma *to_q_numeral [simp]:*
assumes *"n dvd numeral m"*

shows *"to_q n (numeral m) = 1"*
⟨*proof*⟩

lemma *to_q_of_nat_period_1 [simp]: "w ∈ ℤ ⟹ to_q (Suc 0) w = 1"*
⟨*proof*⟩

lemma *Ln_to_q:*
assumes *"x ∈ Re -‘ {n/2<..n/2}" "n > 0"*
shows *"Ln (to_q n x) = 2 * pi * i * x / n"*
⟨*proof*⟩

lemma *to_q_nonzero [simp]: "to_q n τ ≠ 0"*
⟨*proof*⟩

lemma *norm_to_q [simp]: "norm (to_q n z) = exp (-2 * pi * Im z / n)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *deriv_to_q [simp]: "n > 0 ⟹ deriv (to_q n) z = 2 * pi * i / n * to_q n z"*
⟨*proof*⟩

lemma *to_q_holomorphic_on [holomorphic_intros]:*
"f holomorphic_on A ⟹ n > 0 ⟹ (λz. to_q n (f z)) holomorphic_on A"
⟨*proof*⟩

lemma *to_q_analytic_on [analytic_intros]:*
"f analytic_on A ⟹ n > 0 ⟹ (λz. to_q n (f z)) analytic_on A"
⟨*proof*⟩

lemma *to_q_continuous_on [continuous_intros]:*
"continuous_on A f ⟹ n > 0 ⟹ continuous_on A (λz. to_q n (f z))"
⟨*proof*⟩

lemma *to_q_continuous [continuous_intros]:*
"continuous F f ⟹ n > 0 ⟹ continuous F (λz. to_q n (f z))"
⟨*proof*⟩

lemma *to_q_tendsto [tendsto_intros]:*
"(f ⟶ x) F ⟹ n > 0 ⟹ ((λz. to_q n (f z)) ⟶ to_q n x) F"
⟨*proof*⟩

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*⟩

lemma *to_q_inj_on_standard:*
assumes *n: "n > 0"*
shows *"inj_on (to_q n) (Re -` {-n/2..<n/2})"*
⟨*proof*⟩

lemma *filterlim_to_q_at_ii_inf' [tendsto_intros]:*
assumes *n: "n > 0"*
shows *"filterlim (to_q n) (nhds 0) at_ii_inf"*
⟨*proof*⟩

lemma *filterlim_to_q_at_ii_inf [tendsto_intros]: "n > 0 ⟹ filterlim (to_q n) (at 0) at_ii_inf"*
⟨*proof*⟩

lemma *eventually_to_q_neq:*
assumes *n: "n > 0"*
shows *"eventually (λw. to_q n w ≠ to_q n z) (at z)"*
⟨*proof*⟩

lemma *inj_on_to_q:*
assumes *n: "n > 0"*
shows *"inj_on (to_q n) (ball z (1/2))"*
⟨*proof*⟩

lemma *filtermap_to_q_nhds:*
assumes *n: "n > 0"*
shows *"filtermap (to_q n) (nhds z) = nhds (to_q n z)"*
⟨*proof*⟩

lemma *filtermap_to_q_at:*
assumes *n: "n > 0"*
shows *"filtermap (to_q n) (at z) = at (to_q n z)"*
⟨*proof*⟩

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*⟩

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

lemma *Im_of_q: "q ≠ 0 ⟹ n > 0 ⟹ Im (of_q n q) = -n * ln (norm q)*

```
/ (2 * pi)"
```
⟨*proof*⟩

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*⟩

lemma *to_q_of_q [simp]: "q ≠ 0 ⟹ n > 0 ⟹ to_q n (of_q n q) = q"*
⟨*proof*⟩

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*⟩

lemma *filterlim_norm_at_0: "filterlim norm (at_right 0) (at 0)"*
⟨*proof*⟩

lemma *filterlim_of_q_at_0:*
assumes *n: "n > 0"*
shows *"filterlim (of_q n) at_ii_inf (at 0)"*
⟨*proof*⟩

lemma *at_ii_inf_filtermap:*
assumes *"n > 0"*
shows *"filtermap (to_q n) at_ii_inf = at 0"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *moebius_uminus': "moebius (-a) b c d = moebius a (-b) (-c) (-d)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

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)"*
  ⟨*proof*⟩

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)"*
  ⟨*proof*⟩

lemma *Im_transform:*
  *"Im (φ z) = Im z / norm (of_int c * z + of_int d) ^ 2"*
⟨*proof*⟩

lemma *Im_transform_pos_aux:*
  assumes *"Im z ≠ 0"*
  shows *"of_int c * z + of_int d ≠ 0"*
⟨*proof*⟩

lemma *Im_transform_pos: "Im z > 0 ⟹ Im (φ z) > 0"*
  ⟨*proof*⟩

lemma *Im_transform_neg: "Im z < 0 ⟹ Im (φ z) < 0"*
  ⟨*proof*⟩

lemma *Im_transform_zero_iff [simp]: "Im (φ z) = 0 ⟷ Im z = 0"*
  ⟨*proof*⟩

lemma *Im_transform_pos_iff [simp]: "Im (φ z) > 0 ⟷ Im z > 0"*
  ⟨*proof*⟩

lemma *Im_transform_neg_iff [simp]: "Im (φ z) < 0 ⟷ Im z < 0"*

⟨*proof*⟩

lemma *Im_transform_nonneg_iff [simp]: "Im (φ z) ≥ 0 ⟷ Im z ≥ 0"*
⟨*proof*⟩

lemma *Im_transform_nonpos_iff [simp]: "Im (φ z) ≤ 0 ⟷ Im z ≤ 0"*
⟨*proof*⟩

lemma *transform_in_reals_iff [simp]: "φ z ∈ ℝ ⟷ z ∈ ℝ"*
⟨*proof*⟩

end

lemma *Im_one_over_neg_iff [simp]: "Im (1 / z) < 0 ⟷ Im z > 0"*
⟨*proof*⟩

locale *inverse_unimodular_moebius_transform = unimodular_moebius_transform*
begin

sublocale *inv: unimodular_moebius_transform d "-b" "-c" a*
⟨*proof*⟩

lemma *inv_φ:*
assumes *"of_int c * z + of_int d ≠ 0"*
shows *"inv.φ (φ z) = z"*
⟨*proof*⟩

lemma *inv_φ':*
assumes *"of_int c * z - of_int a ≠ 0"*
shows *"φ (inv.φ z) = z"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *part_equivp_modgrp_rel: "part_equivp modgrp_rel"*
⟨*proof*⟩

quotient_type *modgrp = "int × int × int × int" / partial: modgrp_rel*
⟨*proof*⟩

instantiation *modgrp :: one*
begin

lift_definition *one_modgrp :: modgrp* **is** *"(1, 0, 0, 1)"*
⟨*proof*⟩

instance ⟨*proof*⟩
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')"*
⟨*proof*⟩

instance ⟨*proof*⟩
end

instantiation *modgrp :: inverse*
begin

lift_definition *inverse_modgrp :: "modgrp ⇒ modgrp"*
is *"λ(a, b, c, d). (d, -b, -c, a)"*
⟨*proof*⟩

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

instance ⟨*proof*⟩

end

interpretation *modgrp: Groups.group "(*) :: modgrp ⇒ _" 1 inverse*

⟨*proof*⟩

instance *modgrp :: monoid_mult*
⟨*proof*⟩

lemma *inverse_power_modgrp: "inverse (x ^ n :: modgrp) = inverse x ^ n"*
⟨*proof*⟩

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)"
⟨*proof*⟩

The shift operation $z \mapsto z + n$

lift_definition *shift_modgrp :: "int ⇒ modgrp"* **is** *"λn. (1, n, 0, 1)"*
⟨*proof*⟩

The shift operation $z \mapsto z + 1$

lift_definition *T_modgrp :: modgrp* **is** *"(1, 1, 0, 1)"*
⟨*proof*⟩

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

lift_definition *S_modgrp :: modgrp* **is** *"(0, -1, 1, 0)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *pole_modgrp_in_Reals: "pole_modgrp f ∈ (ℝ :: 'a :: real_field set)"*
⟨*proof*⟩

lemma *Im_pole_modgrp [simp]: "Im (pole_modgrp f) = 0"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *Im_pole_image_modgrp [simp]: "Im (pole_image_modgrp f) = 0"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lift_definition *modgrp_b :: "modgrp ⇒ int"* **is** *"λ(a,b,c,d). if c < 0 ∨ c = 0 ∧ d < 0 then -b else b"*
⟨*proof*⟩

lift_definition *modgrp_c :: "modgrp ⇒ int"* **is** *"λ(a,b,c,d). |c|"*
⟨*proof*⟩

lift_definition *modgrp_d :: "modgrp ⇒ int"* **is** *"λ(a,b,c,d). if c < 0 ∨ c = 0 ∧ d < 0 then -d else d"*
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *modgrp_c_shift_left [simp]:*
*"modgrp_c (shift_modgrp n * f) = modgrp_c f"*
⟨*proof*⟩

lemma *modgrp_d_shift_left [simp]:*
*"modgrp_d (shift_modgrp n * f) = modgrp_d f"*
⟨*proof*⟩

lemma *modgrp_abcd_det: "modgrp_a x * modgrp_d x - modgrp_b x * modgrp_c x = 1"*

⟨*proof*⟩

lemma *modgrp_c_nonneg: "modgrp_c x ≥ 0"*
⟨*proof*⟩

lemma *modgrp_a_nz_or_b_nz: "modgrp_a x ≠ 0 ∨ modgrp_b x ≠ 0"*
⟨*proof*⟩

lemma *modgrp_c_nz_or_d_nz: "modgrp_c x ≠ 0 ∨ modgrp_d x ≠ 0"*
⟨*proof*⟩

lemma *modgrp_cd_signs: "modgrp_c x > 0 ∨ modgrp_c x = 0 ∧ modgrp_d x > 0"*
⟨*proof*⟩

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*⟩

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)"*
⟨*proof*⟩

lemma *modgrp_wrong: "a * d - b * c ≠ 1 ⟹ modgrp a b c d = 1"*
⟨*proof*⟩

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'"*
⟨*proof*⟩

lemma *modgrp_abcd [simp]: "modgrp (modgrp_a x) (modgrp_b x) (modgrp_c x) (modgrp_d x) = x"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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)"*
⟨*proof*⟩

lemma *diff_scale_factor_modgrp_commutes:*
"diff_scale_factor_modgrp f w z = diff_scale_factor_modgrp f z w"
⟨*proof*⟩

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}"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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

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)"*
⟨*proof*⟩

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*⟩

lemma *apply_shift_modgrp [simp]: "apply_modgrp (shift_modgrp n) z = z + of_int n"*
⟨*proof*⟩

lemma *apply_modgrp_T [simp]: "apply_modgrp T_modgrp z = z + 1"*
⟨*proof*⟩

lemma *apply_modgrp_S [simp]: "apply_modgrp S_modgrp z = -1 / z"*
⟨*proof*⟩

lemma *apply_modgrp_1 [simp]: "apply_modgrp 1 z = z"*
⟨*proof*⟩

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*⟩

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)"*
⟨*proof*⟩

lemma *is_singular_modgrp_altdef: "is_singular_modgrp x ⟷ modgrp_c x ≠ 0"*
⟨*proof*⟩

lemma *not_is_singular_modgrpD:*
assumes *"¬is_singular_modgrp x"*
shows *"x = shift_modgrp (sgn (modgrp_a x) * modgrp_b x)"*
⟨*proof*⟩

lemma *is_singular_modgrp_inverse [simp]: "is_singular_modgrp (inverse x) ⟷ is_singular_modgrp x"*
⟨*proof*⟩

lemma *is_singular_modgrp_S_left_iff [simp]: "is_singular_modgrp (S_modgrp * f) ⟷ modgrp_a f ≠ 0"*
⟨*proof*⟩

lemma *is_singular_modgrp_S_right_iff [simp]: "is_singular_modgrp (f * S_modgrp) ⟷ modgrp_d f ≠ 0"*
⟨*proof*⟩

lemma *is_singular_modgrp_T_left_iff [simp]:*
*"is_singular_modgrp (T_modgrp * f) ⟷ is_singular_modgrp f"*
⟨*proof*⟩

lemma *is_singular_modgrp_T_right_iff [simp]:*
*"is_singular_modgrp (f * T_modgrp) ⟷ is_singular_modgrp f"*
⟨*proof*⟩

lemma *is_singular_modgrp_shift_left_iff [simp]:*
*"is_singular_modgrp (shift_modgrp n * f) ⟷ is_singular_modgrp f"*
⟨*proof*⟩

lemma *is_singular_modgrp_shift_right_iff [simp]:*
*"is_singular_modgrp (f * shift_modgrp n) ⟷ is_singular_modgrp f"*
⟨*proof*⟩

lemma *pole_modgrp_inverse [simp]: "pole_modgrp (inverse x) = pole_image_modgrp x"*
⟨*proof*⟩

lemma *pole_image_modgrp_inverse [simp]: "pole_image_modgrp (inverse x) = pole_modgrp x"*
⟨*proof*⟩

lemma *pole_image_modgrp_in_Reals: "pole_image_modgrp x ∈ (ℝ :: 'a :: {real_field, field} set)"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

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*⟩

lemma *shift_modgrp_1: "shift_modgrp 1 = T_modgrp"*
⟨*proof*⟩

lemma *shift_modgrp_eq_iff: "shift_modgrp n = shift_modgrp m ⟷ n = m"*
⟨*proof*⟩

lemma *shift_modgrp_neq_S [simp]: "shift_modgrp n ≠ S_modgrp"*
⟨*proof*⟩

lemma *S_neq_shift_modgrp [simp]: "S_modgrp ≠ shift_modgrp n"*
⟨*proof*⟩

lemma *shift_modgrp_eq_T_iff [simp]: "shift_modgrp n = T_modgrp ⟷ n = 1"*
⟨*proof*⟩

lemma *T_eq_shift_modgrp_iff [simp]: "T_modgrp = shift_modgrp n ⟷ n = 1"*
⟨*proof*⟩

lemma *shift_modgrp_0 [simp]: "shift_modgrp 0 = 1"*
⟨*proof*⟩

lemma *shift_modgrp_add: "shift_modgrp (m + n) = shift_modgrp m * shift_modgrp n"*
⟨*proof*⟩

lemma *shift_modgrp_minus: "shift_modgrp (-m) = inverse (shift_modgrp m)"*
⟨*proof*⟩

lemma *shift_modgrp_power: "shift_modgrp n ^ m = shift_modgrp (n * m)"*
⟨*proof*⟩

lemma *shift_modgrp_power_int: "shift_modgrp n powi m = shift_modgrp (n * m)"*
⟨*proof*⟩

lemma *shift_shift_modgrp: "shift_modgrp n * (shift_modgrp m * x) = shift_modgrp (n + m) * x"*
⟨*proof*⟩

lemma *shift_modgrp_conv_T_power: "shift_modgrp n = T_modgrp powi n"*
⟨*proof*⟩

lemma *modgrp_S_S [simp]: "S_modgrp * S_modgrp = 1"*
⟨*proof*⟩

lemma *inverse_S_modgrp [simp]: "inverse S_modgrp = S_modgrp"*
⟨*proof*⟩

lemma *modgrp_S_S_' [simp]: "S_modgrp * (S_modgrp * x) = x"*
⟨*proof*⟩

lemma *modgrp_S_power: "S_modgrp ^ n = (if even n then 1 else S_modgrp)"*
⟨*proof*⟩

lemma *modgrp_S_S_power_int: "S_modgrp powi n = (if even n then 1 else S_modgrp)"*
⟨*proof*⟩

lemma *not_is_singular_1_modgrp [simp]: "¬is_singular_modgrp 1"*
⟨*proof*⟩

lemma *not_is_singular_T_modgrp [simp]: "¬is_singular_modgrp T_modgrp"*
⟨*proof*⟩

lemma *not_is_singular_shift_modgrp [simp]: "¬is_singular_modgrp (shift_modgrp n)"*
⟨*proof*⟩

lemma *is_singular_S_modgrp [simp]: "is_singular_modgrp S_modgrp"*
⟨*proof*⟩

lemma *pole_modgrp_S [simp]: "pole_modgrp S_modgrp = 0"*
⟨*proof*⟩

lemma *pole_modgrp_1 [simp]: "pole_modgrp 1 = 0"*
⟨*proof*⟩

lemma *pole_modgrp_T [simp]: "pole_modgrp T_modgrp = 0"*
⟨*proof*⟩

lemma *pole_modgrp_shift [simp]: "pole_modgrp (shift_modgrp n) = 0"*
⟨*proof*⟩

lemma *pole_image_modgrp_1 [simp]: "pole_image_modgrp 1 = 0"*
⟨*proof*⟩

lemma *pole_image_modgrp_T [simp]: "pole_image_modgrp T_modgrp = 0"*
⟨*proof*⟩

lemma *pole_image_modgrp_shift [simp]: "pole_image_modgrp (shift_modgrp n) = 0"*
⟨*proof*⟩

lemma *pole_image_modgrp_S [simp]: "pole_image_modgrp S_modgrp = 0"*
⟨*proof*⟩

lemma *minus_minus_power2_eq: "(-x - y :: 'a :: ring_1) ^ 2 = (x + y) ^ 2"*
⟨*proof*⟩

lift_definition *deriv_modgrp :: "modgrp ⇒ 'a :: field ⇒ 'a"* **is**
*"λ(a,b,c,d) x. inverse ((of_int c * x + of_int d) ^ 2)"*
⟨*proof*⟩

lemma *deriv_modgrp_nonzero:*
assumes *"¬is_singular_modgrp f ∨ (x :: 'a :: field_char_0) ≠ pole_modgrp f"*
shows *"deriv_modgrp f x ≠ 0"*
⟨*proof*⟩

lemma *deriv_modgrp_altdef:*
*"deriv_modgrp f z = inverse (of_int (modgrp_c f) * z + of_int (modgrp_d f)) ^ 2"*

⟨*proof*⟩

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)"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *modgrp_c_0:*
assumes *"a * d = 1"*
shows *"modgrp a b 0 d = shift_modgrp (if a > 0 then b else -b)"*
⟨*proof*⟩

lemma *not_singular_modgrpD:*
assumes *"¬is_singular_modgrp f"*
shows *"f = shift_modgrp (modgrp_b f)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *modgrp_rel_reflI: "(case x of (a,b,c,d) ⇒ a * d - b * c = 1) ⟹ x = y ⟹ modgrp_rel x y"*
⟨*proof*⟩

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')"*

⟨*proof*⟩

lemma *modgrp_inverse:*
assumes *"a * d - b * c = 1"*
shows *"inverse (modgrp a b c d) = modgrp d (-b) (-c) a"*
⟨*proof*⟩

lemma *modgrp_a_mult_shift [simp]: "modgrp_a (f * shift_modgrp m) = modgrp_a f"*
⟨*proof*⟩

lemma *modgrp_b_mult_shift [simp]: "modgrp_b (f * shift_modgrp m) = modgrp_a f * m + modgrp_b f"*
⟨*proof*⟩

lemma *modgrp_c_mult_shift [simp]: "modgrp_c (f * shift_modgrp m) = modgrp_c f"*
⟨*proof*⟩

lemma *modgrp_d_mult_shift [simp]: "modgrp_d (f * shift_modgrp m) = modgrp_c f * m + modgrp_d f"*
⟨*proof*⟩

lemma *coprime_modgrp_c_d: "coprime (modgrp_c f) (modgrp_d f)"*
⟨*proof*⟩

context *unimodular_moebius_transform*
begin

lift_definition *as_modgrp :: modgrp* **is** *"(a, b, c, d)"*
⟨*proof*⟩

lemma *as_modgrp_altdef: "as_modgrp = modgrp a b c d"*
⟨*proof*⟩

lemma *φ_as_modgrp: "φ = apply_modgrp as_modgrp"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *inverse_modgrp_code [code]: "inverse (modgrp a b c d) = modgrp d (-b) (-c) a"*
⟨*proof*⟩

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'))"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *modgrp_c_code [code]:*
*"modgrp_c (modgrp a b c d) = (if a * d - b * c = 1 then |c| else 0)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *is_singular_modgrp_code [code]:*
*"is_singular_modgrp (modgrp a b c d) ⟷ a * d - b * c = 1 ∧ c ≠ 0"*
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *modgrp_factor_shift [simp]: "modgrp_factor (shift_modgrp n) z =*
1"
⟨*proof*⟩

lemma *modgrp_factor_T [simp]: "modgrp_factor T_modgrp z = 1"*
⟨*proof*⟩

lemma *modgrp_factor_S [simp]: "modgrp_factor S_modgrp z = z"*
⟨*proof*⟩

lemma *modgrp_factor_shift_right [simp]:*
```
  "modgrp_factor (f * shift_modgrp n) z = modgrp_factor f (z + of_int
n)"
```
⟨*proof*⟩

lemma *modgrp_factor_shift_left [simp]:*
```
  "modgrp_factor (shift_modgrp n * f) z = modgrp_factor f z"
```
⟨*proof*⟩

lemma *modgrp_factor_T_right [simp]:*
```
  "modgrp_factor (f * T_modgrp) z = modgrp_factor f (z + 1)"
```
⟨*proof*⟩

lemma *modgrp_factor_T_left [simp]:*
```
  "modgrp_factor (T_modgrp * f) z = modgrp_factor f z"
```
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *modgrp_factor_analytic [analytic_intros]: "modgrp_factor g analytic_on*

```
A"
```
⟨*proof*⟩

lemma *modgrp_factor_meromorphic [meromorphic_intros]: "modgrp_factor h meromorphic_on A"*
⟨*proof*⟩

lemma *modgrp_factor_nonzero [simp]:*
assumes *"Im z ≠ 0"*
shows *"modgrp_factor g z ≠ 0"*
⟨*proof*⟩

lemma *tendsto_modgrp_factor [tendsto_intros]:*
"(f ⟶ c) F ⟹ ((λx. modgrp_factor g (f x)) ⟶ modgrp_factor g c) F"
⟨*proof*⟩

lemma *minus_diff_power_even:*
assumes *"even k"*
shows *"(-a - b) ^ k = (a + b :: 'a :: ring_1) ^ k"*
⟨*proof*⟩

lemma *minus_diff_power_int_even:*
assumes *"even k"*
shows *"(-a - b) powi k = (a + b :: 'a :: field) powi k"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *modgrp_slash_altdef:*
"modgrp_slash f k z = (if even k then modgrp_factor f z powi k else 0)"
⟨*proof*⟩

lemma *modgrp_slash_1 [simp]: "even k ⟹ modgrp_slash 1 k z = 1"*
⟨*proof*⟩

lemma *modgrp_slash_shift [simp]: "even k ⟹ modgrp_slash (shift_modgrp n) k z = 1"*
⟨*proof*⟩

lemma *modgrp_slash_T [simp]: "even k ⟹ modgrp_slash T_modgrp k z = 1"*
⟨*proof*⟩

lemma *modgrp_slash_S [simp]: "even k ⟹ modgrp_slash S_modgrp k z = z powi k"*
⟨*proof*⟩

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*⟩

lemma *modgrp_slash_meromorphic [meromorphic_intros]: "modgrp_slash f k meromorphic_on A"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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]"
```
⟨*proof*⟩

lemma *modgrp_from_gens_invert [simp]:*
```
  "modgrp_from_gens (invert_modgrp_gens xs) = inverse (modgrp_from_gens
xs)"
```
⟨*proof*⟩

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)])"
```
⟨*proof*⟩
termination
⟨*proof*⟩

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])"
```
⟨*proof*⟩

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]))"
```
⟨*proof*⟩

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"
```

⟨*proof*⟩

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*⟩

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"
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *zorder_moebius_zero:*
assumes *"a ≠ 0" "a * d - b * c ≠ 0"*
shows *"zorder (moebius a b c d) (-b / a) = 1"*
⟨*proof*⟩

lemma *zorder_moebius_pole:*
assumes *"c ≠ 0" "a * d - b * c ≠ 0"*
shows *"zorder (moebius a b c d) (-d / c) = -1"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *zorder_fls_modgrp_pole:*
assumes *"is_singular_modgrp f"*
shows *"zorder (apply_modgrp f) (pole_modgrp f) = -1"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *moebius_uminus1: "moebius (-a) b c d = moebius a (-b) (-c) (-d)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *moebius_eq_shift: "moebius 1 (of_int n) 0 1 z = z + of_int n"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *moebius_eq_S: "moebius 0 1 (-1) 0 z = -1 / z"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *power_in_G [intro]: "f ∈ G ⟹ f ^ n ∈ G"*
⟨*proof*⟩

lemma *power_int_in_G [intro]: "f ∈ G ⟹ f powi n ∈ G"*
⟨*proof*⟩

lemma *prod_list_in_G [intro]: "(⋀x. x ∈ set xs ⟹ x ∈ G) ⟹ prod_list xs ∈ G"*
⟨*proof*⟩

lemma *inverse_in_G_iff [simp]: "inverse f ∈ G ⟷ f ∈ G"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *orbit_empty: "Im x ≤ 0 ⟹ orbit x = {}"*
⟨*proof*⟩

lemma *rel_imp_Im_pos [dest]:*
assumes *"rel x y"*
shows *"Im x > 0" "Im y > 0"*
⟨*proof*⟩

lemma *rel_refl [simp]: "rel x x ⟷ Im x > 0"*
⟨*proof*⟩

lemma *rel_sym:*
assumes *"rel x y"*
shows *"rel y x"*
⟨*proof*⟩

lemma *rel_commutes: "rel x y = rel y x"*
⟨*proof*⟩

lemma *rel_trans [trans]:*
assumes *"rel x y" "rel y z"*
shows *"rel x z"*
⟨*proof*⟩

lemma *relI1 [intro]: "rel x y ⟹ f ∈ G ⟹ Im x > 0 ⟹ rel x (apply_modgrp f y)"*
⟨*proof*⟩

lemma *relI2 [intro]: "rel x y ⟹ f ∈ G ⟹ Im x > 0 ⟹ rel (apply_modgrp f x) y"*
⟨*proof*⟩

lemma *rel_apply_modgrp_left_iff [simp]:*
assumes *"f ∈ G"*
shows *"rel (apply_modgrp f x) y ⟷ Im x > 0 ∧ rel x y"*
⟨*proof*⟩

lemma *rel_apply_modgrp_right_iff [simp]:*
assumes *"f ∈ G"*

shows *"rel y (apply_modgrp f x) ⟷ Im x > 0 ∧ rel y x"*
⟨*proof*⟩

lemma *orbit_refl_iff: "x ∈ orbit x ⟷ Im x > 0"*
⟨*proof*⟩

lemma *orbit_refl: "Im x > 0 ⟹ x ∈ orbit x"*
⟨*proof*⟩

lemma *orbit_cong: "rel x y ⟹ orbit x = orbit y"*
⟨*proof*⟩

lemma *orbit_empty_iff [simp]: "orbit x = {} ⟷ Im x ≤ 0" "{} = orbit x ⟷ Im x ≤ 0"*
⟨*proof*⟩

lemmas *[simp] = orbit_refl_iff*

lemma *orbit_eq_iff: "orbit x = orbit y ⟷ Im x ≤ 0 ∧ Im y ≤ 0 ∨ rel x y"*
⟨*proof*⟩

lemma *orbit_apply_modgrp [simp]: "f ∈ G ⟹ orbit (apply_modgrp f z) = orbit z"*
⟨*proof*⟩

lemma *apply_modgrp_in_orbit_iff [simp]: "f ∈ G ⟹ apply_modgrp f z ∈ orbit y ⟷ z ∈ orbit y"*
⟨*proof*⟩

lemma *orbit_imp_Im_pos: "x ∈ orbit y ⟹ Im x > 0"*
⟨*proof*⟩

end

interpretation *modular_group: modgrp_subgroup UNIV*
⟨*proof*⟩

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

lemma (**in** *modgrp_subgroup*) *rel_imp_rel: "rel x y ⟹ x ∼Γ y"*
⟨*proof*⟩

lemma *modular_group_rel_plus_int_iff_right1 [simp]:*
assumes *"z ∈ ℤ"*
shows *"x ∼Γ y + z ⟷ x ∼Γ y"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *modular_group_rel_S_iff_right [simp]: "x ∼Γ -(1/y) ⟷ x ∼Γ y"*
⟨*proof*⟩

lemma *modular_group_rel_S_iff_left [simp]: "-(1/x) ∼Γ y ⟷ x ∼Γ y"*
⟨*proof*⟩

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}"
⟨*proof*⟩

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*⟩

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*⟩

lemma *shift_modgrp_in_G_iff: "shift_modgrp n ∈ G ⟷ int (modgrp_subgroup_period G) dvd n"*
⟨*proof*⟩

lemma *shift_modgrp_in_G_period [intro, simp]:*
"shift_modgrp (int (modgrp_subgroup_period G)) ∈ G"
⟨*proof*⟩

lemma *shift_modgrp_in_G [intro]:*

"int (modgrp_subgroup_period G) dvd n ⟹ shift_modgrp n ∈ G"
⟨*proof*⟩

end

interpretation *modular_group: modgrp_subgroup_periodic UNIV*
rewrites *"modgrp_subgroup_period UNIV = Suc 0"*
⟨*proof*⟩

lemma *modgrp_subgroup_period_UNIV [simp]: "modgrp_subgroup_period UNIV = Suc 0"*
⟨*proof*⟩

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}"*
⟨*proof*⟩

lemma *modgrps_cong_altdef: "modgrps_cong q = {f. q dvd modgrp_c f}"*
⟨*proof*⟩

lemma *modgrp_in_modgrps_cong_iff:*
assumes *"a * d - b * c = 1"*
shows *"modgrp a b c d ∈ modgrps_cong q ⟷ q dvd c"*
⟨*proof*⟩

lemma *modgrp_in_modgrps_cong:*
assumes *"q dvd c" "a * d - b * c = 1"*
shows *"modgrp a b c d ∈ modgrps_cong q"*
⟨*proof*⟩

lemma *shift_in_modgrps_cong [simp]: "shift_modgrp n ∈ modgrps_cong q"*
⟨*proof*⟩

lemma *S_in_modgrps_cong_iff [simp]: "S_modgrp ∈ modgrps_cong q ⟷ is_unit q"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *S_in_subgrp_iff [simp]: "S_modgrp ∈ subgrp ⟷ q = 1"*
⟨*proof*⟩

sublocale *modgrp_subgroup* Γ'
⟨*proof*⟩

end

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

lemma *p_pos: "p > 0"*
⟨*proof*⟩

lemma *p_not_1 [simp]: "p ≠ 1"*
⟨*proof*⟩

sublocale *hecke_cong_subgroup p*
⟨*proof*⟩

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*⟩

lemma *modgrp_decompose':*
obtains *g h*
where *"g ∈ Γ'" "h = 1 ∨ (∃k∈{0..<p}. h = S_shift_modgrp k)" "f = g * h"*
⟨*proof*⟩

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*
⟨*proof*⟩

sublocale *of_ω12_coords: bounded_linear of_ω12_coords*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *latticeI: "of_ω12_coords (x, y) = z ⟹ x ∈ ℤ ⟹ y ∈ ℤ ⟹ z ∈ Λ"*
⟨*proof*⟩

lemma *latticeE:*
assumes *"z ∈ Λ"*
obtains *x y* **where** *"z = of_ω12_coords (of_int x, of_int y)"*
⟨*proof*⟩

lemma *lattice0I [intro]: "z ∈ Λ ⟹ z ≠ 0 ⟹ z ∈ Λ*⁺*"*
⟨*proof*⟩

lemma *lattice0E [elim]: "⋀P. z ∈ Λ*⁺ *⟹ (z ∈ Λ ⟹ z ≠ 0 ⟹ P) ⟹ P"*
⟨*proof*⟩

lemma *in_lattice0_iff: "z ∈ Λ*⁺ *⟷ z ∈ Λ ∧ z ≠ 0"*
⟨*proof*⟩

named_theorems *lattice_intros*

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

lemma *generator_in_lattice [lattice_intros, simp]: "ω1 ∈ lattice" "ω2 ∈ lattice"*
⟨*proof*⟩

lemma *uminus_in_lattice [lattice_intros]: "z ∈ Λ ⟹ -z ∈ Λ"*
⟨*proof*⟩

lemma *uminus_in_lattice_iff: "-z ∈ Λ ⟷ z ∈ Λ"*
⟨*proof*⟩

lemma *uminus_in_lattice0_iff: "-z ∈ Λ*⁺ *⟷ z ∈ Λ*⁺*"*
⟨*proof*⟩

lemma *add_in_lattice [lattice_intros]: "z ∈ Λ ⟹ w ∈ Λ ⟹ z + w ∈ Λ"*
⟨*proof*⟩

lemma *lattice_lattice0: "Λ = insert 0 Λ*⁺*"*
⟨*proof*⟩

lemma *mult_of_nat_left_in_lattice [lattice_intros]: "z ∈ Λ ⟹ of_nat*

n * z ∈ Λ"
⟨proof⟩

lemma mult_of_nat_right_in_lattice [lattice_intros]: "z ∈ Λ ⟹ z * of_nat n ∈ Λ"
⟨proof⟩

lemma mult_of_int_left_in_lattice [lattice_intros]: "z ∈ Λ ⟹ of_int n * z ∈ Λ"
⟨proof⟩

lemma mult_of_int_right_in_lattice [lattice_intros]: "z ∈ Λ ⟹ z * of_int n ∈ Λ"
⟨proof⟩

lemma diff_in_lattice [lattice_intros]: "z ∈ Λ ⟹ w ∈ Λ ⟹ z - w ∈ Λ"
⟨proof⟩

lemma diff_in_lattice_commute: "z - w ∈ Λ ⟷ w - z ∈ Λ"
⟨proof⟩

lemma of_ω12_coords_in_lattice [lattice_intros]: "ab ∈ ℤ × ℤ ⟹ of_ω12_coords ab ∈ Λ"
⟨proof⟩

lemma lattice_plus_right_cancel [simp]: "y ∈ Λ ⟹ x + y ∈ Λ ⟷ x ∈ Λ"
⟨proof⟩

lemma lattice_plus_left_cancel [simp]: "x ∈ Λ ⟹ x + y ∈ Λ ⟷ y ∈ Λ"
⟨proof⟩

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⟩

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"

⟨*proof*⟩

lemma *relE:*
assumes *"rel x y"*
obtains *z* **where** *"z ∈ Λ" "y = x + z"*
⟨*proof*⟩

lemma *rel_symI: "rel x y ⟹ rel y x"*
⟨*proof*⟩

lemma *rel_sym: "rel x y ⟷ rel y x"*
⟨*proof*⟩

lemma *rel_0_right_iff: "rel x 0 ⟷ x ∈ Λ"*
⟨*proof*⟩

lemma *rel_0_left_iff: "rel 0 x ⟷ x ∈ Λ"*
⟨*proof*⟩

lemma *rel_trans [trans]: "rel x y ⟹ rel y z ⟹ rel x z"*
⟨*proof*⟩

lemma *rel_minus [lattice_intros]: "rel a b ⟹ rel (-a) (-b)"*
⟨*proof*⟩

lemma *rel_minus_iff: "rel (-a) (-b) ⟷ rel a b"*
⟨*proof*⟩

lemma *rel_add [lattice_intros]: "rel a b ⟹ rel c d ⟹ rel (a + c) (b + d)"*
⟨*proof*⟩

lemma *rel_diff [lattice_intros]: "rel a b ⟹ rel c d ⟹ rel (a - c) (b - d)"*
⟨*proof*⟩

lemma *rel_mult_of_nat_left [lattice_intros]: "rel a b ⟹ rel (of_nat n * a) (of_nat n * b)"*
⟨*proof*⟩

lemma *rel_mult_of_nat_right [lattice_intros]: "rel a b ⟹ rel (a * of_nat n) (b * of_nat n)"*
⟨*proof*⟩

lemma *rel_mult_of_int_left [lattice_intros]: "rel a b ⟹ rel (of_int n * a) (of_int n * b)"*
⟨*proof*⟩

lemma *rel_mult_of_int_right [lattice_intros]: "rel a b ⟹ rel (a **

```
of_int n) (b * of_int n)"
```
⟨*proof*⟩

lemma *rel_sum [lattice_intros]:*
```
"(⋀i. i ∈ A ⟹ rel (f i) (g i)) ⟹ rel (∑i∈A. f i) (∑i∈A. g i)"
```
⟨*proof*⟩

lemma *rel_sum_list [lattice_intros]:*
```
"list_all2 rel xs ys ⟹ rel (sum_list xs) (sum_list ys)"
```
⟨*proof*⟩

lemma *rel_lattice_trans_left [trans]:* `"x ∈ Λ ⟹ rel x y ⟹ y ∈ Λ"`
⟨*proof*⟩

lemma *rel_lattice_trans_right [trans]:* `"rel x y ⟹ y ∈ Λ ⟹ x ∈ Λ"`
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *swap_of_ω12_coords [simp]:* `"swap.of_ω12_coords = of_ω12_coords`
`∘ prod.swap"`
⟨*proof*⟩

lemma *swap_lattice [simp]:* `"swap.lattice = lattice"`
⟨*proof*⟩

lemma *swap_lattice0 [simp]:* `"swap.lattice0 = lattice0"`
⟨*proof*⟩

lemma *swap_rel [simp]:* `"swap.rel = rel"`
⟨*proof*⟩

end

A pair (ω_1, ω_2) of complex numbers with $\omega_2 / \omega_1 \notin \mathbb{R}$ 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)"`
⟨*proof*⟩

lemma *fundpair_cnj_iff [simp]: "fundpair (cnj a, cnj b) = fundpair (a, b)"*
⟨*proof*⟩

lemma *fundpair_altdef: "fundpair = (λ(a,b). a / b ∉ ℝ)"*
⟨*proof*⟩

lemma
assumes *"fundpair (a, b)"*
shows *fundpair_imp_nonzero [dest]: "a ≠ 0" "b ≠ 0"*
and *fundpair_imp_neq: "a ≠ b" "b ≠ a"*
⟨*proof*⟩

lemma *fundpair_imp_independent:*
assumes *"fundpair (ω1, ω2)"*
shows *"independent {ω1, ω2}"*
⟨*proof*⟩

lemma *fundpair_imp_basis:*
assumes *"fundpair (ω1, ω2)"*
shows *"span {ω1, ω2} = UNIV"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *ω1_nonzero [simp]: "ω1 ≠ 0"* **and** *ω2_nonzero [simp]: "ω2 ≠ 0"*
⟨*proof*⟩

lemma *lattice0_nonempty [simp]: "lattice0 ≠ {}"*
⟨*proof*⟩

lemma *ω12_independent': "independent {ω1, ω2}"*
⟨*proof*⟩

lemma *span_ω12: "span {ω1, ω2} = UNIV"*
⟨*proof*⟩

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*
⟨*proof*⟩

sublocale *ω2_coord: bounded_linear ω2_coord*
⟨*proof*⟩

sublocale *ω12_coords: linear ω12_coords*
⟨*proof*⟩

sublocale *ω12_coords: bounded_linear ω12_coords*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *ω12_coords_ω1 [simp]: "ω12_coords ω1 = (1, 0)"*
and *ω12_coords_ω2 [simp]: "ω12_coords ω2 = (0, 1)"*
⟨*proof*⟩

lemma *ω12_coords_of_ω12_coords [simp]: "ω12_coords (of_ω12_coords z)*
= z"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *of_ω12_coords_ω12_coords [simp]: "of_ω12_coords (ω12_coords z)*
= z"
⟨*proof*⟩

lemma *ω12_coords_eqI:*
assumes *"of_ω12_coords a = b"*
shows *"ω12_coords b = a"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *of_ω12_coords_eq_iff: "of_ω12_coords z1 = of_ω12_coords z2 ⟷ z1 = z2"*
⟨*proof*⟩

lemma *ω12_coords_eq_iff: "ω12_coords z1 = ω12_coords z2 ⟷ z1 = z2"*
⟨*proof*⟩

lemma *of_ω12_coords_eq_0_iff [simp]: "of_ω12_coords z = 0 ⟷ z = (0,0)"*
⟨*proof*⟩

lemma *ω12_coords_eq_0_0_iff [simp]: "ω12_coords x = (0, 0) ⟷ x = 0"*
⟨*proof*⟩

lemma *bij_of_ω12_coords: "bij of_ω12_coords"*
⟨*proof*⟩

lemma *bij_betw_lattice: "bij_betw of_ω12_coords (ℤ × ℤ) lattice"*
⟨*proof*⟩

lemma *bij_betw_lattice0: "bij_betw of_ω12_coords (ℤ × ℤ - {(0,0)}) lattice0"*
⟨*proof*⟩

lemma *bij_betw_lattice': "bij_betw (of_ω12_coords ∘ map_prod of_int of_int) UNIV lattice"*
⟨*proof*⟩

lemma *bij_betw_lattice0': "bij_betw (of_ω12_coords ∘ map_prod of_int of_int) (-{(0,0)}) lattice0"*
⟨*proof*⟩

lemma *infinite_lattice: "¬finite lattice"*
⟨*proof*⟩

lemma *ω12_coords_image_eq: "ω12_coords ‘ X = of_ω12_coords -‘ X"*
⟨*proof*⟩

lemma *of_ω12_coords_image_eq: "of_ω12_coords ‘ X = ω12_coords -‘ X"*
⟨*proof*⟩

lemma *of_ω12_coords_linepath:*
"of_ω12_coords (linepath a b x) = linepath (of_ω12_coords a) (of_ω12_coords b) x"
⟨*proof*⟩

lemma *of_ω12_coords_linepath':*
"of_ω12_coords o (linepath a b) =
linepath (of_ω12_coords a) (of_ω12_coords b)"
⟨*proof*⟩

lemma *ω12_coords_linepath:*
"ω12_coords (linepath a b x) = linepath (ω12_coords a) (ω12_coords b) x"
⟨*proof*⟩

lemma *of_ω12_coords_in_lattice_iff:*
"of_ω12_coords z ∈ Λ ⟷ fst z ∈ ℤ ∧ snd z ∈ ℤ"
⟨*proof*⟩

lemma *of_ω12_coords_in_lattice [simp, intro]:*
"fst z ∈ ℤ ⟹ snd z ∈ ℤ ⟹ of_ω12_coords z ∈ Λ"
⟨*proof*⟩

lemma *in_lattice_conv_ω12_coords: "z ∈ Λ ⟷ ω12_coords z ∈ ℤ × ℤ"*
⟨*proof*⟩

lemma *ω12_coords_in_Z_times_Z: "z ∈ Λ ⟹ ω12_coords z ∈ ℤ × ℤ"*

⟨*proof*⟩

lemma *half_periods_notin_lattice [simp]:*
"ω1 / 2 ∉ Λ" "ω2 / 2 ∉ Λ" "(ω1 + ω2) / 2 ∉ Λ"
⟨*proof*⟩

end

locale *complex_lattice_swap = complex_lattice*
begin

sublocale *pre_complex_lattice_swap ω1 ω2* ⟨*proof*⟩

sublocale *swap: complex_lattice ω2 ω1*
⟨*proof*⟩

lemma *swap_ω12_coords [simp]: "swap.ω12_coords = prod.swap ∘ ω12_coords"*
⟨*proof*⟩

lemma *swap_ω1_coord [simp]: "swap.ω1_coord = ω2_coord"*
and *swap_ω2_coord [simp]: "swap.ω2_coord = ω1_coord"*
⟨*proof*⟩

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* + ω_1, *z* + ω_2, and *z* + ω_1 + ω_2. 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)"*
⟨*proof*⟩

lemma *convex_period_parallelogram [intro]:*
"convex (period_parallelogram z)"
⟨*proof*⟩

lemma *closure_period_parallelogram:*
"closure (period_parallelogram z) = (+) z ‘ of_ω12_coords ‘ (cbox (0,0) (1,1))"
⟨*proof*⟩

lemma *compact_closure_period_parallelogram [intro]: "compact (closure (period_parallelogram z))"*
⟨*proof*⟩

lemma *vertex_in_period_parallelogram [simp, intro]: "z ∈ period_parallelogram z"*
⟨*proof*⟩

lemma *nonempty_period_parallelogram: "period_parallelogram z ≠ {}"*
⟨*proof*⟩

end

lemma (**in** *pre_complex_lattice_swap*)
swap_period_parallelogram [simp]: "swap.period_parallelogram = period_parallelogram"
⟨*proof*⟩

context *complex_lattice*
begin

lemma *simple_path_parallelogram: "simple_path (parallelogram_path z ω1 ω2)"*
⟨*proof*⟩

lemma (**in** *-*) *image_plus_conv_vimage_plus:*
fixes *c :: "'a :: group_add"*
shows *"(+) c ‘ A = (+) (-c) -‘ A"*
⟨*proof*⟩

lemma *period_parallelogram_altdef:*
"period_parallelogram z = {w. ω12_coords (w - z) ∈ {0..<1} × {0..<1}}"
⟨*proof*⟩

lemma *interior_period_parallelogram:*

```
  "interior (period_parallelogram z) = (+) z ‘ of_ω12_coords ‘ box (0,0)
(1,1)"
```
⟨*proof*⟩

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*⟩

lemma *fund_period_parallelogram_in_lattice_iff:*
assumes `"z ∈ period_parallelogram 0"`
shows `"z ∈ Λ ⟷ z = 0"`
⟨*proof*⟩

lemma *path_image_parallelogram_path:*
```
  "path_image (parallelogram_path z ω1 ω2) = frontier (period_parallelogram
z)"
```
⟨*proof*⟩

lemma *path_image_parallelogram_subset_closure:*
```
  "path_image (parallelogram_path z ω1 ω2) ⊆ closure (period_parallelogram
z)"
```
⟨*proof*⟩

lemma *path_image_parallelogram_disjoint_interior:*
```
  "path_image (parallelogram_path z ω1 ω2) ∩ interior (period_parallelogram
z) = {}"
```
⟨*proof*⟩

lemma *winding_number_parallelogram_outside:*
assumes `"w ∉ closure (period_parallelogram z)"`
shows `"winding_number (parallelogram_path z ω1 ω2) w = 0"`
⟨*proof*⟩

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*⟩

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"
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *to_fund_parallelogramE:*
obtains *m n* **where** *"to_fund_parallelogram z = z + of_int m * ω1 + of_int n * ω2"*
⟨*proof*⟩

lemma *rel_to_fund_parallelogram_left: "rel (to_fund_parallelogram z) z"*
⟨*proof*⟩

lemma *rel_to_fund_parallelogram_right: "rel z (to_fund_parallelogram z)"*
⟨*proof*⟩

lemma *rel_to_fund_parallelogram_left_iff [simp]: "rel (to_fund_parallelogram z) w ⟷ rel z w"*
⟨*proof*⟩

lemma *rel_to_fund_parallelogram_right_iff [simp]: "rel z (to_fund_parallelogram w) ⟷ rel z w"*
⟨*proof*⟩

lemma *to_fund_parallelogram_in_lattice_iff [simp]:*
"to_fund_parallelogram z ∈ lattice ⟷ z ∈ lattice"
⟨*proof*⟩

lemma *to_fund_parallelogram_in_lattice [lattice_intros]:*
"z ∈ lattice ⟹ to_fund_parallelogram z ∈ lattice"
⟨*proof*⟩

`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*⟩

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*⟩

lemma *to_fund_parallelogram_0 [simp]: "to_fund_parallelogram 0 = 0"*
⟨*proof*⟩

lemma *to_fund_parallelogram_lattice [simp]: "z ∈ Λ ⟹ to_fund_parallelogram
z = 0"*
⟨*proof*⟩

lemma *to_fund_parallelogram_eq_iff [simp]:*
```
  "to_fund_parallelogram u = to_fund_parallelogram v ⟷ rel u v"
```
⟨*proof*⟩

lemma *to_fund_parallelogram_eq_0_iff [simp]: "to_fund_parallelogram u
= 0 ⟷ u ∈ Λ"*
⟨*proof*⟩

lemma *to_fund_parallelogram_of_fund_parallelogram:*
```
  "z ∈ period_parallelogram 0 ⟹ to_fund_parallelogram z = z"
```
⟨*proof*⟩

lemma *to_fund_parallelogram_idemp [simp]:*
```
  "to_fund_parallelogram (to_fund_parallelogram z) = to_fund_parallelogram
z"
```
⟨*proof*⟩

lemma *to_fund_parallelogram_unique:*
```
  assumes "rel z z'" "z' ∈ period_parallelogram 0"
  shows   "to_fund_parallelogram z = z'"
```
⟨*proof*⟩

lemma *to_fund_parallelogram_unique':*
```
  assumes "rel z z'" "z ∈ period_parallelogram 0" "z' ∈ period_parallelogram
0"
  shows   "z = z'"
```
⟨*proof*⟩

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)}"
⟨*proof*⟩

lemma *zero_in_half_fund_parallelogram [simp, intro]: "0 ∈ half_fund_parallelogram"*
⟨*proof*⟩

lemma *half_fund_parallelogram_in_lattice_iff:*
assumes *"z ∈ half_fund_parallelogram"*
shows *"z ∈ Λ ⟷ z = 0"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma (**in** *complex_lattice*) *rel_to_half_fund_parallelogram:*
"rel z (to_half_fund_parallelogram z) ∨ rel z (-to_half_fund_parallelogram
z)"
⟨*proof*⟩

lemma (**in** *complex_lattice*) *to_half_fund_parallelogram_in_half_fund_parallelogram*
[intro]:
"to_half_fund_parallelogram z ∈ half_fund_parallelogram"
⟨*proof*⟩

lemma (**in** *complex_lattice*) *half_fund_parallelogram_subset_period_parallelogram:*
"half_fund_parallelogram ⊆ period_parallelogram 0"
⟨*proof*⟩

lemma *to_half_fund_parallelogram_in_lattice_iff [simp]: "to_half_fund_parallelogram*
z ∈ Λ ⟷ z ∈ Λ"
⟨*proof*⟩

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"
  ⟨proof⟩

lemma to_half_fund_parallelogram_of_half_fund_parallelogram:
  assumes "z ∈ half_fund_parallelogram"
  shows   "to_half_fund_parallelogram z = z"
  ⟨proof⟩

lemma to_half_fund_parallelogram_idemp [simp]:
  "to_half_fund_parallelogram (to_half_fund_parallelogram z) = to_half_fund_parallelogram
z"
  ⟨proof⟩

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⟩

lemma to_half_fund_parallelogram_eq_iff:
  "to_half_fund_parallelogram z = to_half_fund_parallelogram w ⟷ rel
z w ∨ rel z (-w)"
⟨proof⟩

lemma in_half_fund_parallelogram_imp_half_lattice:
  assumes "z ∈ half_fund_parallelogram" "to_fund_parallelogram (-z) ∈
half_fund_parallelogram"
  shows   "2 * z ∈ Λ"
  ⟨proof⟩

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⟩
```

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *of_ω12_coords_of_int_in_lattice_layer:*
"of_ω12_coords (of_int a, of_int b) ∈ lattice_layer (nat (max |a| |b|))"
⟨*proof*⟩

lemma *lattice_layer_covers: "Λ = (⋃k. lattice_layer k)"*
⟨*proof*⟩

lemma *finite_lattice_layer: "finite (lattice_layer k)"*
⟨*proof*⟩

lemma *lattice_layer_0: "lattice_layer 0 = {0}"*
⟨*proof*⟩

lemma *zero_in_lattice_layer_iff [simp]: "0 ∈ lattice_layer k ⟷ k = 0"*
⟨*proof*⟩

lemma *lattice_layer_disjoint:*
assumes *"m ≠ n"*
shows *"lattice_layer m ∩ lattice_layer n = {}"*
⟨*proof*⟩

lemma *lattice0_conv_layers: "Λ* = (⋃i∈{0<..}. lattice_layer i)"* **(is** *"?lhs = ?rhs")*
⟨*proof*⟩

lemma *card_lattice_layer:*
assumes *"k > 0"*
shows *"card (lattice_layer k) = 8 * k"*
⟨*proof*⟩

lemma *lattice_layer_nonempty: "lattice_layer k ≠ {}"*
⟨*proof*⟩

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})"
⟨*proof*⟩

lemma *lattice_layer_path_nonempty: "lattice_layer_path ≠ {}"*
⟨*proof*⟩

lemma *compact_lattice_layer_path [intro]: "compact lattice_layer_path"*
⟨*proof*⟩

lemma *lattice_layer_subset: "lattice_layer k ⊆ (*) (of_nat k) ‘ lattice_layer_path"*
⟨*proof*⟩

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*⟩

lemma *Inf_para_nonzero [simp]: "Inf_para ≠ 0"*
⟨*proof*⟩

lemma *Inf_para_le:*
assumes *"z ∈ lattice_layer_path"*
shows *"Inf_para ≤ norm z"*
⟨*proof*⟩

lemma *lattice_layer_le_norm:*
assumes *"ω ∈ lattice_layer k"*
shows *"k * Inf_para ≤ norm ω"*
⟨*proof*⟩

corollary *Inf_para_le_norm:*
assumes *"ω ∈ Λ*"*

shows *"Inf_para ≤ norm ω"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *Sup_para_pos: "Sup_para > 0"*
⟨*proof*⟩

lemma *Sup_para_nonzero [simp]: "Sup_para ≠ 0"*
⟨*proof*⟩

lemma *lattice_layer_ge_norm:*
assumes *"ω ∈ lattice_layer k"*
shows *"norm ω ≤ k * Sup_para"*
⟨*proof*⟩

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*⟩

lemma *closed_lattice: "closed lattice"*
⟨*proof*⟩

lemma *lattice_sparse: "Λ sparse_in UNIV"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *winding_number_parallelogram_inside':*
assumes *"w ∈ interior (period_parallelogram z)"*
shows *"winding_number (parallelogram_path z 1 τ) w = 1"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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*⟩

lemma
pole_modgrp_not_in_region [simp]: "pole_modgrp f ∉ R" **and**
pole_image_modgrp_not_in_region [simp]: "pole_image_modgrp f ∉ R"
⟨*proof*⟩

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" ("$\mathcal{R}_\Gamma$")* **where**
"$\mathcal{R}_\Gamma$ = -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" ("$\mathcal{R}_\Gamma$''")* **where**
"$\mathcal{R}_\Gamma$' = $\mathcal{R}_\Gamma$ ∪ (-ball 0 1 ∩ Re -‘ {-1/2..0} ∩ {z. Im z > 0})"

lemma *std_fund_region_altdef:*
"$\mathcal{R}_\Gamma$ = {z. norm z > 1 ∧ norm (z + cnj z) < 1 ∧ Im z > 0}"
⟨*proof*⟩

lemma *in_std_fund_region_iff:*
"z ∈ $\mathcal{R}_\Gamma$ ⟷ norm z > 1 ∧ Re z ∈ {-1/2<..<1/2} ∧ Im z > 0"
⟨*proof*⟩

lemma *in_std_fund_region'_iff:*
"z ∈ $\mathcal{R}_\Gamma$' ⟷ Im z > 0 ∧ ((norm z > 1 ∧ Re z ∈ {-1/2..<1/2}) ∨ (norm z = 1 ∧ Re z ∈ {-1/2..0}))"
⟨*proof*⟩

lemma *open_std_fund_region [simp, intro]: "open $\mathcal{R}_\Gamma$"*
⟨*proof*⟩

lemma *Im_std_fund_region: "z ∈ $\mathcal{R}_\Gamma$ ⟹ Im z > 0"*
⟨*proof*⟩

We now show that the closure of the standard fundamental region contains exactly those points *z* with *|z| ≥ 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 ⊆ $\mathcal{R}_\Gamma$"*
⟨*proof*⟩

lemma *image_std_fund_region_subset: "g ‘ $\mathcal{R}_\Gamma$ ⊆ S"*

⟨*proof*⟩

lemma *std_fund_region_map_inverses: "f (g x) = x" "g (f y) = y"*
⟨*proof*⟩

lemma *bij_betw_std_fund_region1: "bij_betw f S* $\mathcal{R}_\Gamma$*"*
⟨*proof*⟩

lemma *bij_betw_std_fund_region2: "bij_betw g* $\mathcal{R}_\Gamma$ *S"*
⟨*proof*⟩

lemma *image_subset_std_fund_region': "f ` S' ⊆ T"*
⟨*proof*⟩

lemma *image_std_fund_region_subset': "g ` T ⊆ S'"*
⟨*proof*⟩

lemma *bij_betw_std_fund_region1': "bij_betw f S' T"*
⟨*proof*⟩

lemma *bij_betw_std_fund_region2': "bij_betw g T S'"*
⟨*proof*⟩

lemma *closure_std_fund_region: "closure* $\mathcal{R}_\Gamma$ *= T"*
⟨*proof*⟩

lemma *in_closure_std_fund_region_iff:*
"x ∈ closure $\mathcal{R}_\Gamma$ *⟷ norm x ≥ 1 ∧ Re x ∈ {-1/2..1/2} ∧ Im x ≥ 0"*
⟨*proof*⟩

lemma *frontier_std_fund_region:*
"frontier $\mathcal{R}_\Gamma$ *=*
{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*⟩

lemma *std_fund_region'_subset_closure: "*$\mathcal{R}_\Gamma$*' ⊆ closure* $\mathcal{R}_\Gamma$*"*
⟨*proof*⟩

lemma *std_fund_region'_superset: "*$\mathcal{R}_\Gamma$ *⊆* $\mathcal{R}_\Gamma$*'"*
⟨*proof*⟩

lemma *in_std_fund_region'_not_on_frontier_iff:*
assumes *"z ∉ frontier* $\mathcal{R}_\Gamma$*"*
shows *"z ∈* $\mathcal{R}_\Gamma$*' ⟷ z ∈* $\mathcal{R}_\Gamma$*"*
⟨*proof*⟩

lemma *simply_connected_std_fund_region: "simply_connected* $\mathcal{R}_\Gamma$*"*
⟨*proof*⟩

lemma *simply_connected_closure_std_fund_region: "simply_connected (closure* $\mathcal{R}_\Gamma$*)"*
⟨*proof*⟩

lemma *std_fund_region'_subset: "*$\mathcal{R}_\Gamma$*' ⊆ closure* $\mathcal{R}_\Gamma$*"*
⟨*proof*⟩

lemma *closure_std_fund_region_Im_pos: "closure* $\mathcal{R}_\Gamma$ *⊆ {z. Im z > 0}"*
⟨*proof*⟩

lemma *closure_std_fund_region_Im_ge: "closure* $\mathcal{R}_\Gamma$ *⊆ {z. Im z ≥ sqrt 3 / 2}"*
⟨*proof*⟩

lemma *std_fund_region'_minus_std_fund_region:*
"$\mathcal{R}_\Gamma$*' - *$\mathcal{R}_\Gamma$ *=*
{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*⟩

lemma *closure_std_fund_region_minus_std_fund_region':*
"closure $\mathcal{R}_\Gamma$ *-* $\mathcal{R}_\Gamma$*' =*
{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*⟩

lemma *cis_in_std_fund_region'_iff:*
assumes *"φ ∈ {0..pi}"*
shows *"cis φ ∈* $\mathcal{R}_\Gamma$*' ⟷ φ ∈ {pi/2..2*pi/3}"*
⟨*proof*⟩

lemma *imag_axis_in_std_fund_region'_iff: "y *$_R$ i ∈ $\mathcal{R}_\Gamma$*' ⟷ y ≥ 1"*
⟨*proof*⟩

lemma *vertical_left_in_std_fund_region'_iff:*
*"-1 / 2 + y *$_R$ i ∈ $\mathcal{R}_\Gamma$*' ⟷ y ≥ sqrt 3 / 2"*
⟨*proof*⟩

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*⟩

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..}"*

⟨*proof*⟩

lemma *compact_std_fund_region:*
assumes *"B > 1"*
shows *"compact (closure $\mathcal{R}_\Gamma$ ∩ {z. Im z ≤ B})"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

Theorem 2.2

lemma *std_fund_region_fundamental_aux1:*
assumes *"Im τ' > 0"*
obtains *τ* **where** *"Im τ > 0" "τ ∼$_\Gamma$ τ'" "norm τ ≥ 1" "norm (τ + 1) ≥ norm τ" "norm (τ - 1) ≥ norm τ"*
⟨*proof*⟩

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*⟩

lemma *std_fund_region_fundamental_aux3:*
fixes *x y :: complex*
assumes *xy: "x ∈ $\mathcal{R}_\Gamma$" "y ∈ $\mathcal{R}_\Gamma$"*
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*⟩

lemma *std_fund_region_fundamental_aux4:*
fixes *x y :: complex*
assumes *xy: "x ∈ $\mathcal{R}_\Gamma$" "y ∈ $\mathcal{R}_\Gamma$"*
assumes *f: "y = apply_modgrp f x"*
shows *"f = 1"*
⟨*proof*⟩

Theorem 2.3

interpretation *std_fund_region: fundamental_region UNIV std_fund_region*
⟨*proof*⟩

theorem *std_fund_region_no_fixed_point:*
assumes *"z ∈ $\mathcal{R}_\Gamma$"*
assumes *"apply_modgrp f z = z"*
shows *"f = 1"*
⟨*proof*⟩

lemma *std_fund_region_no_fixed_point':*
assumes *"z ∈ $\mathcal{R}_\Gamma$"*
assumes *"apply_modgrp f z = apply_modgrp g z"*
shows *"f = g"*
⟨*proof*⟩

lemma *equiv_point_in_std_fund_region':*
assumes *"Im z > 0"*
obtains *z'* **where** *"z ∼$_\Gamma$ z'" "z' ∈ $\mathcal{R}_\Gamma$'"*
⟨*proof*⟩

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 ‘ $\mathcal{R}_\Gamma$"*
begin

lemma *R_altdef: "R = {z. Im z > 0} ∩ apply_modgrp (inverse f) -‘ $\mathcal{R}_\Gamma$"*
⟨*proof*⟩

lemma *R_altdef': "R = apply_modgrp (inverse f) -‘* $\mathcal{R}_\Gamma$ *"*
⟨*proof*⟩

sublocale *fundamental_region UNIV R*
⟨*proof*⟩

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*"*
⟨*proof*⟩

lemma *Re_modfun_rho [simp]: "Re* ϱ *= -1 / 2"*
and *Im_modfun_rho [simp]: "Im* ϱ *= sqrt 3 / 2"*
⟨*proof*⟩

lemma *norm_modfun_rho [simp]: "norm* ϱ *= 1"*
⟨*proof*⟩

lemma *modfun_rho_plus_1_eq: "*ϱ *+ 1 = exp (pi / 3 ** i*)"*
⟨*proof*⟩

lemma *norm_modfun_rho_plus_1 [simp]: "norm (*ϱ *+ 1) = 1"*
⟨*proof*⟩

lemma *cnj_modfun_rho: "cnj* ϱ *= -*ϱ *- 1"*
and *cnj_modfun_rho_plus1: "cnj (*ϱ *+ 1) = -*ϱ*"*
⟨*proof*⟩

lemma *modfun_rho_cube: "*ϱ *^ 3 = 1"*
⟨*proof*⟩

lemma *modfun_rho_power_mod3_reduce: "*ϱ *^ n = *ϱ *^ (n mod 3)"*
⟨*proof*⟩

lemma *modfun_rho_power_mod3_reduce': "n* $\geq$ *3* $\Longrightarrow$ ϱ *^ n =* ϱ *^ (n mod 3)"*
⟨*proof*⟩

lemmas *[simp] = modfun_rho_power_mod3_reduce' [of "numeral num"* **for** *num]*

lemma *modfun_rho_square: "*ϱ *^ 2 = -*ϱ *- 1"*

⟨*proof*⟩

lemma *modfun_rho_not_real [simp]: "ϱ ∉ ℝ"*
⟨*proof*⟩

lemma *modfun_rho_nonzero [simp]: "ϱ ≠ 0"*
⟨*proof*⟩

lemma *modfun_rho_not_one [simp]: "ϱ ≠ 1"*
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *i_in_closure_std_fund_region [intro, simp]:* "i ∈ *closure* $\mathcal{R}_\Gamma$"
and *i_in_std_fund_region' [intro, simp]:* "i ∈ $\mathcal{R}_\Gamma$'"
and *modfun_rho_in_closure_std_fund_region [intro, simp]: "ϱ ∈ closure* $\mathcal{R}_\Gamma$"
and *modfun_rho_in_std_fund_region' [intro, simp]: "ϱ* ∈ $\mathcal{R}_\Gamma$'"
and *modfun_rho_plus_1_notin_closure_std_fund_region [intro, simp]: "ϱ + 1 ∈ closure* $\mathcal{R}_\Gamma$"
and *modfun_rho_plus_1_notin_std_fund_region' [intro, simp]: "ϱ + 1* ∉ $\mathcal{R}_\Gamma$'"
⟨*proof*⟩

lemma *modfun_rho_power_eq_1_iff: "ϱ ^ n = 1* ⟷ *3 dvd n"*
⟨*proof*⟩

4.5 Fundamental regions for congruence subgroups

context *hecke_prime_subgroup*
begin

definition *std_fund_region_cong ("*$\mathcal{R}$*")* **where**
"$\mathcal{R}$ *=* $\mathcal{R}_\Gamma$ ∪ (⋃*k*∈*{0..<p}. (λz. -1 / (z + of_int k)) ‘* $\mathcal{R}_\Gamma$*)"*

lemma *std_fund_region_cong_altdef:*
"$\mathcal{R}$ *=* $\mathcal{R}_\Gamma$ ∪ (⋃*k*∈*{0..<p}. apply_modgrp (S_shift_modgrp k) ‘* $\mathcal{R}_\Gamma$*)"*
⟨*proof*⟩

lemma *closure_UN_finite: "finite A* ⟹ *closure (*⋃*A) = (*⋃*X*∈*A. closure X)"*
⟨*proof*⟩

sublocale *std_region: fundamental_region* Γ' $\mathcal{R}$

⟨*proof*⟩

end

bundle *modfun_region_notation*
begin
notation *std_fund_region* ("$\mathcal{R}_\Gamma$")
notation *modfun_rho* ("ϱ")
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*⟩

end

context *elliptic_function*
begin

interpretation *elliptic_function_remove_sings* ⟨*proof*⟩

lemma *isolated_singularity [simp, singularity_intros]: "isolated_singularity_at f z"*
⟨*proof*⟩

lemma *not_essential [simp, singularity_intros]: "not_essential f z"*
⟨*proof*⟩

lemma *meromorphic' [meromorphic_intros]: "f meromorphic_on A"*
⟨*proof*⟩

lemma *meromorphic'' [meromorphic_intros]:*
assumes *"g analytic_on A"*
shows *"(λx. f (g x)) meromorphic_on A"*
⟨*proof*⟩

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*⟩

sublocale *poles: complex_lattice_periodic ω1 ω2 "is_pole f"*
⟨*proof*⟩

sublocale *zorder: complex_lattice_periodic ω1 ω2 "zorder f"*
⟨*proof*⟩

sublocale *deriv: complex_lattice_periodic ω1 ω2 "deriv f"*
⟨*proof*⟩

sublocale *higher_deriv: complex_lattice_periodic ω1 ω2 "(deriv ^^ n) f"*
⟨*proof*⟩

sublocale *residue: complex_lattice_periodic ω1 ω2 "residue f"*
⟨*proof*⟩

lemma *eventually_remove_sings_eq: "eventually (λw. remove_sings f w = f w) (cosparse UNIV)"*
⟨*proof*⟩

lemma *eventually_remove_sings_eq': "eventually (λw. remove_sings f w = f w) (at z)"*

⟨*proof*⟩

lemma *isolated_zero_analytic_iff:*
assumes *"f analytic_on {z}"* *"¬(∀≈z∈UNIV. f z = 0)"*
shows *"isolated_zero f z ⟷ f z = 0"*
⟨*proof*⟩

end

context *nicely_elliptic_function*
begin

lemma *nicely_meromorphic' [meromorphic_intros]: "f nicely_meromorphic_on A"*
⟨*proof*⟩

lemma *analytic:*
assumes *"⋀z. z ∈ A ⟹ ¬is_pole f z"*
shows *"f analytic_on A"*
⟨*proof*⟩

lemma *holomorphic:*
assumes *"⋀z. z ∈ A ⟹ ¬is_pole f z"*
shows *"f holomorphic_on A"*
⟨*proof*⟩

lemma *continuous_on:*
assumes *"⋀z. z ∈ A ⟹ ¬is_pole f z"*
shows *"continuous_on A f"*
⟨*proof*⟩

sublocale *elliptic_function ω1 ω2 f*
⟨*proof*⟩

lemma *analytic_at_iff_not_pole: "f analytic_on {z} ⟷ ¬is_pole f z"*
⟨*proof*⟩

lemma *constant_or_avoid: "f = (λ_. c) ∨ (∀≈z∈UNIV. f z ≠ c)"*
⟨*proof*⟩

lemma *isolated_zero_iff:*
assumes *"f ≠ (λ_. 0)"*
shows *"isolated_zero f z ⟷ ¬is_pole f z ∧ f z = 0"*
⟨*proof*⟩

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*⟩

lemma `finite_zeros_in_parallelogram: "finite {z∈period_parallelogram orig. isolated_zero f z}"`
⟨*proof*⟩

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"`
⟨*proof*⟩

lemma `poles_eq_elliptic_order:`
`"(∑z | z ∈ period_parallelogram orig ∧ is_pole f z. nat (-zorder f z)) = elliptic_order f"`
⟨*proof*⟩

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*⟩

lemma *order_pos_iff: "elliptic_order f > 0 ⟷ ¬f constant_on UNIV"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

interpretation *elliptic_function_remove_sings* ⟨*proof*⟩

theorem *zeros_eq_elliptic_order:*
"(∑z | z ∈ period_parallelogram orig ∧ isolated_zero f z. nat (zorder f z)) = elliptic_order f"
⟨*proof*⟩

lemma *card_poles_le_order: "card {z∈period_parallelogram orig. is_pole f z} ≤ elliptic_order f"*

⟨*proof*⟩

lemma *card_zeros_le_order: "card {z∈period_parallelogram orig. isolated_zero f z} ≤ elliptic_order f"*
⟨*proof*⟩

corollary *elliptic_order_eq_0_iff_no_poles: "elliptic_order f = 0 ⟷ (∀z. ¬is_pole f z)"*
⟨*proof*⟩

corollary *elliptic_order_eq_0_iff_no_zeros: "elliptic_order f = 0 ⟷ (∀z. ¬isolated_zero f z)"*
⟨*proof*⟩

lemma *elliptic_order_eq_0_iff_const_cosparse:*
"elliptic_order f = 0 ⟷ (∃c. ∀≈x∈UNIV. f x = c)"
⟨*proof*⟩

lemma *cosparse_eq_or_avoid: "(∀≈z∈UNIV. f z = c) ∨ (∀≈z∈UNIV. f z ≠ c)"*
⟨*proof*⟩

lemma *frequently_eq_imp_almost_everywhere_eq:*
assumes *"frequently (λz. f z = c) (at z)"*
shows *"eventually (λz. f z = c) (cosparse UNIV)"*
⟨*proof*⟩

lemma *eventually_eq_imp_almost_everywhere_eq:*
assumes *"eventually (λz. f z = c) (at z)"*
shows *"eventually (λz. f z = c) (cosparse UNIV)"*
⟨*proof*⟩

lemma *avoid: "elliptic_order f > 0 ⟹ ∀≈z∈UNIV. f z ≠ c"*
⟨*proof*⟩

lemma *avoid': "elliptic_order f > 0 ⟹ eventually (λz. f z ≠ c) (at z)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *lattice_cong': "rel w z ∨ rel w (-z) ⟹ f w = f z"*
  ⟨*proof*⟩

lemma *eval_to_half_fund_parallelogram: "f (to_half_fund_parallelogram z) = f z"*
  ⟨*proof*⟩

lemma *zorder_to_half_fund_parallelogram: "zorder f (to_half_fund_parallelogram z) = zorder f z"*
⟨*proof*⟩

lemma *zorder_uminus: "zorder f (-z) = zorder f z"*
  ⟨*proof*⟩

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*⟩

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*⟩

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)"
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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*⟩

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))"*
⟨*proof*⟩

lemma *elliptic_function_compose_mult_numeral_left:*
assumes *"elliptic_function ω1 ω2 f"*
shows *"elliptic_function ω1 ω2 (λz. f (numeral n * z))"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *elliptic_function_compose_uminus:*
assumes *"elliptic_function ω1 ω2 f"*
shows *"elliptic_function ω1 ω2 (λz. f (-z))"*
⟨*proof*⟩

lemma *elliptic_function_shift:*
assumes *"elliptic_function ω1 ω2 f"*
shows *"elliptic_function ω1 ω2 (λz. f (z + w))"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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*⟩

lemma *elliptic_function_remove_sings [elliptic_function_intros]:*
assumes *"elliptic_function ω1 ω2 f"*
shows *"elliptic_function ω1 ω2 (remove_sings f)"*
⟨*proof*⟩

lemma *elliptic_function_deriv [elliptic_function_intros]:*
assumes *"elliptic_function ω1 ω2 f"*
shows *"elliptic_function ω1 ω2 (deriv f)"*

⟨*proof*⟩

lemma *elliptic_function_higher_deriv [elliptic_function_intros]:*
assumes *"elliptic_function ω1 ω2 f"*
shows *"elliptic_function ω1 ω2 ((deriv ^^ n) f)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is_pole_affine_iff: "is_pole g z ⟷ is_pole f z"*
⟨*proof*⟩

lemma *zorder_pole_affine:*
assumes *"is_pole f z"*
shows *"zorder g z = zorder f z"*
⟨*proof*⟩

lemma *order_affine_eq: "elliptic_order g = elliptic_order f"*
⟨*proof*⟩

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*⟩

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"*
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *bounded_lattice_finite:*
assumes *"bounded B"*
shows *"finite (Λ ∩ B)"*
⟨*proof*⟩

lemma *closed_subset_lattice: "Λ' ⊆ Λ ⟹ closed Λ'"*
⟨*proof*⟩

corollary *closed_lattice0: "closed Λ*"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *weierstrass_aux_converges_absolutely_in_disk':*
fixes α *:: nat* **and** *R :: real* **and** *z:: complex*
assumes *"α > 2"* **and** *"R > 0"* **and** *"z $\in$ cball 0 R"*
shows *"($\lambda\omega$. 1 / norm (z - ω) ^ α) summable_on (Λ - cball 0 R)"*
⟨*proof*⟩

lemma *weierstrass_aux_converges_in_disk':*
fixes α *:: nat* **and** *R :: real* **and** *z:: complex*
assumes *"α > 2"* **and** *"R > 0"* **and** *"z $\in$ cball 0 R"*
shows *"($\lambda\omega$. 1 / (z - ω) ^ α) summable_on (Λ - cball 0 R)"*
⟨*proof*⟩

lemma *weierstrass_aux_converges_absolutely:*
fixes α *:: real*
assumes *"α > 2"* **and** *"Λ' $\subseteq$ Λ"*
shows *"($\lambda\omega$. norm (z - ω) powr -α) summable_on Λ'"*
⟨*proof*⟩

lemma *weierstrass_aux_converges_absolutely':*
fixes α *:: nat*
assumes *"α > 2"* **and** *"Λ' $\subseteq$ Λ"*
shows *"($\lambda\omega$. 1 / norm (z - ω) ^ α) summable_on Λ'"*
⟨*proof*⟩

lemma *weierstrass_aux_converges:*
fixes α *:: real*
assumes *"α > 2"* **and** *"Λ' $\subseteq$ Λ"*
shows *"($\lambda\omega$. (z - ω) powr -α) summable_on Λ'"*
⟨*proof*⟩

lemma *weierstrass_aux_converges':*
fixes α *:: nat*
assumes *"α > 2"* **and** *"Λ' $\subseteq$ Λ"*
shows *"($\lambda\omega$. 1 / (z - ω) ^ α) summable_on Λ'"*
⟨*proof*⟩

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. $\sum\omega\in$X. norm ((z - ω) powr -α))
(λz. $\sum_\infty\omega\in\Lambda$-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. $\sum\omega\in$X. (z - ω) powr -α)
(λz. $\sum_\infty\omega\in\Lambda$-cball 0 R. (z - ω) powr -α)

```
                        (finite_subsets_at_top (Λ - cball 0 R))" (is
?th2)
```
⟨*proof*⟩

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*⟩

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"*
  ⟨*proof*⟩

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*⟩

lemma *eisenstein_fun_aux_minus: "eisenstein_fun_aux n (-z) = (-1) ^ n
* eisenstein_fun_aux n z"*
⟨*proof*⟩

lemma *eisenstein_fun_aux_even_minus: "even n ⟹ eisenstein_fun_aux
n (-z) = eisenstein_fun_aux n z"*
  ⟨*proof*⟩

lemma *eisenstein_fun_aux_odd_minus: "odd n ⟹ eisenstein_fun_aux n
(-z) = -eisenstein_fun_aux n z"*
  ⟨*proof*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *eisenstein_fun_aux_holomorphic: "eisenstein_fun_aux n holomorphic_on -Λ*"*
⟨*proof*⟩

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*⟩

lemma *eisenstein_fun_aux_analytic: "eisenstein_fun_aux n analytic_on -Λ*"*
⟨*proof*⟩

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*⟩

lemma *eisenstein_fun_aux_continuous_on: "continuous_on (-Λ*) (eisenstein_fun_aux n)"*
⟨*proof*⟩

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))"*

⟨*proof*⟩

lemma *weierstrass_aux_translate:*
fixes α *:: real*
assumes *"*α *> 2"*
shows *"(*$\sum_\infty \omega \in \Lambda$*. (z + w - ω) powr -α) = (*$\sum_\infty \omega \in$*(+) (-w) ‘ Λ. (z - ω) powr -α)"*
⟨*proof*⟩

lemma *weierstrass_aux_holomorphic:*
assumes *"*α *> 2" "*$\Lambda' \subseteq \Lambda$*" "finite (*$\Lambda$ *-* Λ*')"*
shows *"(*$\lambda z.$ *∑*$_\infty \omega \in \Lambda'$*. 1 / (z - ω) ^ α) holomorphic_on -*Λ*'"*
⟨*proof*⟩

definition *eisenstein_fun :: "nat ⇒ complex ⇒ complex"* **where**
"eisenstein_fun n z = (if n < 3 ∨ z ∈ Λ *then 0 else (*$\sum_\infty \omega \in \Lambda$*. 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*"*
⟨*proof*⟩

lemma *eisenstein_fun_at_pole_eq_0: "z ∈* Λ *⟹ eisenstein_fun n z = 0"*
⟨*proof*⟩

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*⟩

lemma *eisenstein_fun_altdef:*
"eisenstein_fun n z = (if n < 3 ∨ z ∈ Λ *then 0 else eisenstein_fun_aux n z + 1 / z ^ n)"*
⟨*proof*⟩

lemma *eisenstein_fun_minus: "eisenstein_fun n (-z) = (-1) ^ n * eisenstein_fun n z"*
⟨*proof*⟩

lemma *eisenstein_fun_even_minus: "even n ⟹ eisenstein_fun n (-z) = eisenstein_fun n z"*
⟨*proof*⟩

lemma *eisenstein_fun_odd_minus: "odd n ⟹ eisenstein_fun n (-z) = -eisenstein_fun n z"*
⟨*proof*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

lemma *eisenstein_fun_analytic: "eisenstein_fun n analytic_on -Λ"*
⟨*proof*⟩

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*⟩

lemma *eisenstein_fun_continuous_on: "n ≥ 3 ⟹ continuous_on (-Λ) (eisenstein_fun n)"*
⟨*proof*⟩

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*⟩

sublocale *eisenstein_fun: complex_lattice_periodic ω1 ω2 "eisenstein_fun n"*
⟨*proof*⟩

lemma *is_pole_eisenstein_fun:*
assumes *"n ≥ 3" "z ∈ Λ"*
shows *"is_pole (eisenstein_fun n) z"*
⟨*proof*⟩

sublocale *eisenstein_fun: nicely_elliptic_function ω1 ω2 "eisenstein_fun n"*
⟨*proof*⟩

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*⟩

lemma *minus_latticemz_eq: "uminus ‘ Λ* = Λ*"*
⟨*proof*⟩

lemma *bij_minus_latticemz: "bij_betw uminus Λ* Λ*"*
⟨*proof*⟩

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*
⟨*proof*⟩

sublocale *weierstrass_fun_deriv: nicely_elliptic_function ω1 ω2 weierstrass_fun_deriv*
⟨*proof*⟩

lemmas *[elliptic_function_intros] =*
weierstrass_fun_deriv.elliptic_function_axioms weierstrass_fun_deriv.nicely_elliptic_funct

lemma *weierstrass_fun_deriv_minus [simp]: "℘' (-z) = -℘' z"*
⟨*proof*⟩

lemma *weierstrass_fun_deriv_has_field_derivative:*
assumes *"z ∉ Λ"*
shows *"(℘' has_field_derivative 6 * eisenstein_fun 4 z) (at z)"*

⟨*proof*⟩

lemma *weierstrass_fun_deriv_holomorphic: "℘' holomorphic_on -Λ"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *weierstrass_fun_deriv_analytic: "℘' analytic_on -Λ"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *weierstrass_fun_deriv_continuous_on: "continuous_on (-Λ) ℘'"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *tendsto_weierstrass_fun_deriv [tendsto_intros]:*
assumes *"(f ⟶ c) F" "c ∉ Λ"*
shows *"((λz. ℘' (f z)) ⟶ ℘' c) F"*
⟨*proof*⟩

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 / ω²))"*

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 / z² + weierstrass_fun_aux z)"*

lemma *weierstrass_fun_aux_0 [simp]: "weierstrass_fun_aux 0 = 0"*
⟨*proof*⟩

lemma *weierstrass_fun_at_pole: "ω ∈ Λ ⟹ ℘ ω = 0"*
⟨*proof*⟩

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 / ω²))
(λz. ∑∞ω∈Λ-cball 0 R. norm (1 / (z - ω)² - 1 / ω²))
(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 / ω²)
(λz. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω)² - 1 / ω²)
(finite_subsets_at_top (Λ - cball 0 R))" **(is** *?th2)*
⟨*proof*⟩

lemma *weierstrass_fun_has_field_derivative_aux:*
fixes *R :: real*
defines *"F ≡ (λz. ∑∞ω∈Λ-cball 0 R. 1 / (z - ω)² - 1 / ω²)"*
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*⟩

lemma *norm_summable_weierstrass_fun_aux: "(λω. norm (1 / (z - ω)² - 1 / ω²)) summable_on Λ"*
⟨*proof*⟩

lemma *summable_weierstrass_fun_aux: "(λω. 1 / (z - ω)² - 1 / ω²) summable_on Λ"*
⟨*proof*⟩

lemma *weierstrass_summable: "(λω. 1 / (z - ω)² - 1 / ω²) summable_on Λ*⁺*"*
⟨*proof*⟩

lemma *weierstrass_fun_aux_has_sum:*
"z ∉ Λ⁺ *⟹ ((λω. 1 / (z - ω)² - 1 / ω²) has_sum weierstrass_fun_aux z) Λ*⁺*"*
⟨*proof*⟩

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*⟩

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*"*
⟨*proof*⟩

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*⟩

lemma *weierstrass_fun_aux_continuous_on: "continuous_on (-Λ*) weierstrass_fun_aux"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *weierstrass_fun_aux_analytic: "weierstrass_fun_aux analytic_on -Λ*"*
⟨*proof*⟩

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*⟩

lemma *deriv_weierstrass_fun_aux:*
"z ∉ Λ ⟹ deriv weierstrass_fun_aux z = -2 * eisenstein_fun_aux 3 z"*
⟨*proof*⟩

lemma *weierstrass_fun_has_field_derivative:*
fixes *R :: real*
assumes *z: "z ∉ Λ"*
shows *"(℘ has_field_derivative ℘' z) (at z)"*
⟨*proof*⟩

lemmas *weierstrass_fun_has_field_derivative' [derivative_intros] =*
weierstrass_fun_has_field_derivative [THEN DERIV_chain2]

lemma *weierstrass_fun_holomorphic: "℘ holomorphic_on -Λ"*
⟨*proof*⟩

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*⟩

lemma *weierstrass_fun_analytic: "℘ analytic_on -Λ"*
⟨*proof*⟩

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*⟩

lemma *weierstrass_fun_continuous_on: "continuous_on (-Λ) weierstrass_fun"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *tendsto_weierstrass_fun [tendsto_intros]:*
assumes *"(f ⟶ c) F" "c ∉ Λ"*
shows *"((λz. ℘ (f z)) ⟶ ℘ c) F"*
⟨*proof*⟩

lemma *deriv_weierstrass_fun:*
assumes *"z ∉ Λ"*
shows *"deriv ℘ z = ℘' z"*
⟨*proof*⟩

The following identity is to be read with care: for *ω = 0* we get a division by zero, so the term *1 / ω²* simply gets dropped.

lemma *weierstrass_fun_eq:*
assumes *"z ∉ Λ"*
shows *"℘ z = (∑∞ ω ∈ Λ. (1 / (z - ω) ^ 2) - 1 / ω ^ 2)"*
⟨*proof*⟩

6.3 Ellipticity and poles

It can easily be seen from its definition that ℘ 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*⟩

sublocale *weierstrass_fun: complex_lattice_periodic ω1 ω2 ℘*
⟨*proof*⟩

lemma *zorder_weierstrass_fun_pole:*
assumes *"ω ∈ Λ"*
shows *"zorder ℘ ω = -2"*
⟨*proof*⟩

lemma *is_pole_weierstrass_fun:*
assumes *ω: "ω ∈ Λ"*
shows *"is_pole ℘ ω"*
⟨*proof*⟩

sublocale *weierstrass_fun: nicely_elliptic_function ω1 ω2 ℘*
⟨*proof*⟩

sublocale *weierstrass_fun: even_elliptic_function ω1 ω2 ℘*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *is_pole_weierstrass_fun_deriv_iff: "is_pole ℘' z ⟷ z ∈ Λ"*
⟨*proof*⟩

lemma *zorder_weierstrass_fun_deriv_pole:*
assumes *"z ∈ Λ"*
shows *"zorder ℘' z = -3"*
⟨*proof*⟩

lemma *order_weierstrass_fun [simp]: "elliptic_order ℘ = 2"*
⟨*proof*⟩

lemma *order_weierstrass_fun_deriv [simp]: "elliptic_order ℘' = 3"*
⟨*proof*⟩

sublocale *weierstrass_fun: nonconst_nicely_elliptic_function ω1 ω2 ℘*
⟨*proof*⟩

sublocale *weierstrass_fun_deriv: nonconst_nicely_elliptic_function ω1 ω2 "℘'"*
⟨*proof*⟩

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" ("*e_1*") where*
  "$\mathrm{e}_1 \equiv \wp\ (\omega 1\ /\ 2)$*"*

definition *number_e2:: "complex" ("*e_2*") where*
  "$\mathrm{e}_2 \equiv \wp\ (\omega 2\ /\ 2)$*"*

definition *number_e3:: "complex" ("*e_3*") where*
  "$\mathrm{e}_3 \equiv \wp\ ((\omega 1 + \omega 2)\ /\ 2)$*"*

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 *"*$\omega \notin \Lambda$*" "*$2 * \omega \in \Lambda$*"*
  shows *"to_fund_parallelogram* $\omega \in \{\omega 1\ /\ 2, \omega 2\ /\ 2, (\omega 1 + \omega 2)\ /\ 2\}$*"*
⟨*proof*⟩

lemma *rel_half_period:*
  assumes *"*$\omega \notin \Lambda$*" "*$2 * \omega \in \Lambda$*"*
  shows *"*$\exists \omega' \in \{\omega 1\ /\ 2, \omega 2\ /\ 2, (\omega 1 + \omega 2)\ /\ 2\}.$ *rel* $\omega\ \omega'$*"*
⟨*proof*⟩

lemma *weierstass_fun_deriv_half_period_eq_0:*
  assumes *"*$\omega \in \Lambda$*"*
  shows *"*$\wp'\ (\omega\ /\ 2) = 0$*"*
  ⟨*proof*⟩

lemma *weierstass_fun_deriv_half_root_eq_0 [simp]:*
  "$\wp'\ (\omega 1\ /\ 2) = 0$*" "*$\wp'\ (\omega 2\ /\ 2) = 0$*" "*$\wp'\ ((\omega 1 + \omega 2)\ /\ 2) = 0$*"*
  ⟨*proof*⟩

lemma *weierstrass_fun_at_half_period:*
  assumes *"*$\omega \in \Lambda$*" "*$\omega\ /\ 2 \notin \Lambda$*"*
  shows *"*$\wp\ (\omega\ /\ 2) \in \{\mathrm{e}_1, \mathrm{e}_2, \mathrm{e}_3\}$*"*
⟨*proof*⟩

lemma *weierstrass_fun_at_half_period':*
  assumes *"*$2 * \omega \in \Lambda$*" "*$\omega \notin \Lambda$*"*
  shows *"*$\wp\ \omega \in \{\mathrm{e}_1, \mathrm{e}_2, \mathrm{e}_3\}$*"*
  ⟨*proof*⟩

$\wp'$ 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*⟩

lemma *zorder_weierstrass_fun_deriv_zero:*
  assumes *"z ∉ Λ" "2 * z ∈ Λ"*
  shows *"zorder ℘' z = 1"*
⟨*proof*⟩

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*⟩

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"*
  ⟨*proof*⟩

lemma *surj_weierstrass_fun_deriv:*
  obtains *z* **where** *"z ∈ period_parallelogram w - Λ" "℘' z = c"*
  ⟨*proof*⟩

end

context *complex_lattice_swap*
begin

lemma *weierstrass_fun_aux_swap [simp]: "swap.weierstrass_fun_aux = weierstrass_fun_aux"*
  ⟨*proof*⟩

lemma *weierstrass_fun_swap [simp]: "swap.weierstrass_fun = weierstrass_fun"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *bij_betw_lattice_cnj: "bij_betw cnj lattice cnj.lattice"*
⟨*proof*⟩

lemma *bij_betw_lattice0_cnj: "bij_betw cnj lattice0 cnj.lattice0"*
⟨*proof*⟩

lemma *lattice_cnj_eq: "cnj.lattice = cnj ‘ lattice"*
⟨*proof*⟩

lemma *lattice0_cnj_eq: "cnj.lattice0 = cnj ‘ lattice0"*
⟨*proof*⟩

lemma *eisenstein_fun_aux_cnj: "cnj.eisenstein_fun_aux n z = cnj (eisenstein_fun_aux n (cnj z))"*
⟨*proof*⟩

lemma *weierstrass_fun_aux_cnj: "cnj.weierstrass_fun_aux z = cnj (weierstrass_fun_aux (cnj z))"*
⟨*proof*⟩

lemma *weierstrass_fun_cnj: "cnj.weierstrass_fun z = cnj (weierstrass_fun (cnj z))"*
⟨*proof*⟩

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"*
⟨*proof*⟩

end

locale *complex_lattice_stretch = complex_lattice +*
fixes *c :: complex*
assumes *stretch_nonzero: "c ≠ 0"*
begin

sublocale *stretched: complex_lattice "c * ω1" "c * ω2"*
⟨*proof*⟩

lemma *stretched_of_ω12_coords: "stretched.of_ω12_coords ab = c * of_ω12_coords ab"*
⟨*proof*⟩

lemma *stretched_ω12_coords: "stretched.ω12_coords ab = ω12_coords (ab / c)"*
⟨*proof*⟩

lemma *stretched_ω1_coord: "stretched.ω1_coord ab = ω1_coord (ab / c)"*
and *stretched_ω2_coord: "stretched.ω2_coord ab = ω2_coord (ab / c)"*
⟨*proof*⟩

lemma *mult_into_stretched_lattice: "(*) c ∈ Λ → stretched.lattice"*
⟨*proof*⟩

lemma *mult_into_stretched_lattice': "(*) (inverse c) ∈ stretched.lattice → Λ"*
⟨*proof*⟩

lemma *bij_betw_stretch_lattice: "bij_betw ((*) c) lattice stretched.lattice"*
⟨*proof*⟩

lemma *bij_betw_stretch_lattice0:*
"bij_betw ((*) c) lattice0 stretched.lattice0"
⟨*proof*⟩

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*⟩

lemma *transformed_lattice_subset: "transformed.lattice ⊆ lattice"*
⟨*proof*⟩

lemma *transformed_lattice_eq: "transformed.lattice = lattice"*
⟨*proof*⟩

lemma *transformed_lattice0_eq: "transformed.lattice0 = lattice0"*
⟨*proof*⟩

lemma *eisenstein_fun_aux_transformed [simp]: "transformed.eisenstein_fun_aux = eisenstein_fun_aux"*
⟨*proof*⟩

lemma *weierstrass_fun_aux_transformed [simp]: "transformed.weierstrass_fun_aux = weierstrass_fun_aux"*
⟨*proof*⟩

lemma *weierstrass_fun_transformed [simp]: "transformed.weierstrass_fun = weierstrass_fun"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *zorder_zero_eq:*
assumes *"rel z w ∨ rel z (-w)"*
shows *"zorder f z = (if 2 * w ∈ Λ then 2 else 1)"*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

sublocale *even_elliptic_function ω1 ω2 g*
⟨*proof*⟩

lemma *no_lattice_points': "w ∉ Λ"* **if** *"w ∈ A"* **for** *w*
⟨*proof*⟩

lemma *eq_0_iff: "g z = 0 ⟷ (∃w∈A. h w ≠ 0 ∧ (rel z w ∨ rel z (-w)))"*
if *"z ∉ Λ"* **for** *z*
⟨*proof*⟩

lemma *nonzero_almost_everywhere: "eventually (λz. g z ≠ 0) (cosparse UNIV)"*
⟨*proof*⟩

lemma *eventually_nonzero_at: "eventually (λz. g z ≠ 0) (at z)"*
⟨*proof*⟩

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*⟩

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*⟩

Finally, we show that any even elliptic function can be written as a rational function of ℘. 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*⟩

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*⟩

end

7 Eisenstein series and the differential equations of $\wp$

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"*
⟨*proof*⟩

lemma *eisenstein_series_odd_eq_0 [simp]: "odd k ⟹ eisenstein_series k = 0"*
⟨*proof*⟩

lemma *eisenstein_series_Suc_0 [simp]: "eisenstein_series (Suc 0) = 0"*
⟨*proof*⟩

lemma *eisenstein_series_norm_summable:*
assumes *"n ≥ 3"*
shows *"(λω. 1 / norm ω ^ n) summable_on Λ*"*
⟨*proof*⟩

lemma *eisenstein_series_summable:*
assumes *"n ≥ 3"*
shows *"(λω. 1 / ω ^ n) summable_on Λ*"*
⟨*proof*⟩

lemma *eisenstein_series_has_sum:*
assumes *"k ≥ 3"*
shows *"((λω. 1 / ω ^ k) has_sum eisenstein_series k) Λ*"*
⟨*proof*⟩

lemma *eisenstein_series_altdef:*
assumes *"k ≥ 3"*
shows *"eisenstein_series k = (∑∞ω∈Λ*. 1 / ω ^ k)"*
⟨*proof*⟩

lemma *eisenstein_fun_aux_0 [simp]:*
assumes *"n ≠ 2"*

shows *"eisenstein_fun_aux n 0 = eisenstein_series n"*
⟨*proof*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *fls_weierstrass_nz [simp]: "fls_weierstrass ≠ 0"*
⟨*proof*⟩

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*⟩

corollary *fls_weierstrass_deriv_laurent_expansion [laurent_expansion_intros]:*
"℘' has_laurent_expansion fls_deriv fls_weierstrass"
⟨*proof*⟩

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)"
⟨*proof*⟩

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*⟩

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*⟩

lemma *fls_weierstrass_ODE2:*
defines *"P ≡ fls_weierstrass"*
shows *"fls_deriv (fls_deriv P) = 6 * P ^ 2 - fls_const (30 * G 4)"*
⟨*proof*⟩

theorem *weierstrass_fun_ODE2:*
assumes *"z ∉ Λ"*
shows *"deriv ℘' z = 6 * ℘ z ^ 2 - 30 * G 4"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

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 [*e_1*,* e_2*,* e_3*]"*
⟨*proof*⟩

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 = g_2 *^ 3 - 27 *g_3 *^ 2"*

definition *invariant_j :: complex* **where**
"invariant_j = g_2 *^ 3 / discr"*

theorem
fixes *z:: "complex"*
defines *"P ≡ [:-*g_3*, -*g_2*, 0, 4:]"*
shows *discr_nonzero_aux1: "P = 4 * [:-*e_1*, 1:] * [:-*e_2*, 1:] * [:-*e_3*, 1:]"*
and *discr_nonzero_aux2: "4 * (℘ z)^3 -* g_2 ** (℘ z) -* g_3 *= 4 * (℘ z -* e_1*) * (℘ z -* e_2*) * (℘ z -* e_3*)"*
and *discr_nonzero: "discr ≠ 0"*
⟨*proof*⟩

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)})"*
⟨*proof*⟩

lemma *eisenstein_series_2_altdef:*

```
  "eisenstein_series 2 = 2 * zeta 2 + (∑∞n∈-{0}. ∑∞m. 1 / (of_int
m + of_int n * τ) ^ 2)"
```
⟨*proof*⟩

lemma *eisenstein_series_altdef':*
```
  "k ≥ 3 ⟹ eisenstein_series k = (∑∞(m,n)∈-{(0,0)}. 1 / (of_int m
+ of_int n * τ) ^ k)"
```
⟨*proof*⟩

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*⟩

lemma *abs_summable_one_over_const_minus_nat_power:*
```
  assumes "n ≥ 2"
  shows   "summable (λk. norm (1 / (z - of_nat k :: complex) ^ n))"
```
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *eisenstein_fun_aux_swap [simp]: "swap.eisenstein_fun_aux = eisenstein_fun_aux"*
⟨*proof*⟩

lemma *invariant_g2_swap [simp]: "swap.invariant_g2 = invariant_g2"*
  and *invariant_g3_swap [simp]: "swap.invariant_g3 = invariant_g3"*
  ⟨*proof*⟩

lemma *discr_swap [simp]: "swap.discr = discr"*
  ⟨*proof*⟩

lemma *invariant_j_swap [simp]: "swap.invariant_j = invariant_j"*
  ⟨*proof*⟩

end

context *complex_lattice_cnj*
begin

lemma *eisenstein_series_cnj [simp]: "cnj.eisenstein_series n = cnj (eisenstein_series n)"*
  ⟨*proof*⟩

lemma *invariant_g2_cnj [simp]: "cnj.invariant_g2 = cnj invariant_g2"*
  and *invariant_g3_cnj [simp]: "cnj.invariant_g3 = cnj invariant_g3"*
  ⟨*proof*⟩

lemma *discr_cnj [simp]: "cnj.discr = cnj discr"*
  ⟨*proof*⟩

lemma *invariant_j_cnj [simp]: "cnj.invariant_j = cnj invariant_j"*
  ⟨*proof*⟩

end

context *complex_lattice_stretch*
begin

lemma *eisenstein_series_stretch:*
  *"stretched.eisenstein_series n = c powi (-n) * eisenstein_series n"*
⟨*proof*⟩

lemma *invariant_g2_stretch [simp]: "stretched.invariant_g2 = invariant_g2 / c ^ 4"*
  and *invariant_g3_stretch [simp]: "stretched.invariant_g3 = invariant_g3 / c ^ 6"*
  ⟨*proof*⟩

lemma *discr_stretch [simp]: "stretched.discr = discr / c ^ 12"*
  ⟨*proof*⟩

lemma *invariant_j_stretch [simp]: "stretched.invariant_j = invariant_j"*
⟨*proof*⟩

end

context *unimodular_moebius_transform_lattice*
begin

lemma *eisenstein_series_transformed [simp]:*
assumes *"k ≠ 2"*
shows *"transformed.eisenstein_series k = eisenstein_series k"*
⟨*proof*⟩

lemma *invariant_g2_transformed [simp]: "transformed.invariant_g2 = invariant_g2"*
and *invariant_g3_transformed [simp]: "transformed.invariant_g3 = invariant_g3"*
⟨*proof*⟩

lemma *discr_transformed [simp]: "transformed.discr = discr"*
⟨*proof*⟩

lemma *invariant_j_transformed [simp]: "transformed.invariant_j = invariant_j"*
⟨*proof*⟩

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*⟩

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*⟩

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)))"
  ⟨proof⟩

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⟩

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*⟩

lemma *eisenstein_series_polys_correct:*
```
  "eisenstein_series_polys n = map eisenstein_series_poly [0..<max 2 n]"
```
⟨*proof*⟩

lemma *funpow_rec_right:* `"n > 0 ⟹ (f ^^ n) xs = (f ^^ (n-1)) (f xs)"`
⟨*proof*⟩

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]"
```
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *Eisenstein_G_real_eq_0 [simp]: "z ∈ ℝ ⟹ Eisenstein_G k z = 0"*
⟨*proof*⟩

lemma *Eisenstein_G_0 [simp]:*
assumes *[simp]: "z ∉ ℝ"*
shows *"Eisenstein_G 0 z = 1"*
⟨*proof*⟩

lemma *Eisenstein_G_cnj: "Eisenstein_G k (cnj z) = cnj (Eisenstein_G k z)"*
⟨*proof*⟩

lemma *Eisenstein_G_odd [simp]:*
assumes *"odd k"*
shows *"Eisenstein_G k z = 0"*
⟨*proof*⟩

lemma *Eisenstein_G_uminus: "Eisenstein_G k (-z) = Eisenstein_G k z"*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *Eisenstein_G_plus_int: "Eisenstein_G k (z + of_int m) = Eisenstein_G*
k z"
⟨*proof*⟩

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*⟩

lemma *Eisenstein_E_0 [simp]: "z ∉ ℝ ⟹ Eisenstein_E 0 z = 1"*
  ⟨*proof*⟩

lemma *Eisenstein_E_real_eq_0 [simp]: "z ∈ ℝ ⟹ Eisenstein_E k z = 0"*
  ⟨*proof*⟩

lemma *Eisenstein_E_cnj: "Eisenstein_E k (cnj z) = cnj (Eisenstein_E k*
z)"

⟨*proof*⟩

lemma *Eisenstein_E_odd [simp]:*
assumes *"odd k"*
shows *"Eisenstein_E k z = 0"*
⟨*proof*⟩

lemma *Eisenstein_E_uminus: "Eisenstein_E k (-z) = Eisenstein_E k z"*
⟨*proof*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *modular_discr_real_eq_0 [simp]: "z ∈ ℝ ⟹ modular_discr z = 0"*
⟨*proof*⟩

lemma *modular_discr_cnj: "modular_discr (cnj z) = cnj (modular_discr z)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *modular_discr_uminus: "modular_discr (-z) = modular_discr z"*
⟨*proof*⟩

lemma *modular_discr_nonzero:*
assumes *"z ∉ ℝ"*
shows *"modular_discr z ≠ 0"*
⟨*proof*⟩

lemma *modular_discr_eq_0_iff: "modular_discr z = 0 ⟷ z ∈ ℝ"*
⟨*proof*⟩

theorem *modular_discr_apply_modgrp:*
*"modular_discr (apply_modgrp f z) = modgrp_factor f z ^ 12 * modular_discr z"*
⟨*proof*⟩

lemma *modular_discr_plus_int: "modular_discr (z + of_int m) = modular_discr z"*
⟨*proof*⟩

lemma *modular_discr_minus_one_over: "modular_discr (-(1/z)) = z ^ 12 * modular_discr z"*
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *Klein_J_real_eq_0 [simp]: "z ∈ ℝ ⟹ Klein_J z = 0"*
⟨*proof*⟩

lemma *Klein_J_uminus: "Klein_J (-z) = Klein_J z"*
⟨*proof*⟩

lemma *Klein_J_cnj: "Klein_J (cnj z) = cnj (Klein_J z)"*

⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

lemma *Klein_J_plus_int: "Klein_J (z + of_int m) = Klein_J z"*
⟨*proof*⟩

lemma *Klein_J_minus_one_over: "Klein_J (-(1/z)) = Klein_J z"*
⟨*proof*⟩

lemma *Klein_J_cong:*
assumes *"w ∼Γ z"*
shows *"Klein_J w = Klein_J z"*
⟨*proof*⟩

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*⟩

lemma *Eisenstein_G_6_ii [simp]: "Eisenstein_G 6 i = 0"*
⟨*proof*⟩

lemma *Eisenstein_G_rho_eq_0:*
assumes *"k ≠ 2" "¬6 dvd k"*
shows *"Eisenstein_G k ϱ = 0"*
⟨*proof*⟩

lemma *Eisenstein_G_4_rho [simp]: "Eisenstein_G 4 ϱ = 0"*
⟨*proof*⟩

corollary *Eisenstein_G_6_rho_nonzero: "Eisenstein_G 6 ϱ ≠ 0"*
⟨*proof*⟩

corollary *Eisenstein_G_4_ii_nonzero: "Eisenstein_G 4* i *≠ 0"*
⟨*proof*⟩

corollary *Klein_J_rho [simp]: "Klein_J ϱ = 0"*
⟨*proof*⟩

corollary *Klein_J_ii [simp]: "Klein_J* i *= 1"*
⟨*proof*⟩

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* $\sim_\Gamma$ i*)"*
⟨*proof*⟩

lemma *not_i_equiv_rho [simp]: "¬(*i $\sim_\Gamma$ *ϱ)"*
⟨*proof*⟩

lemma *not_modular_group_rel_rho_i [simp]: " z* $\sim_\Gamma$ *ϱ* $\Longrightarrow$ *¬z* $\sim_\Gamma$ i*"*
⟨*proof*⟩

lemma *modular_group_rel_rho_i_cases [case_names rho i neither invalid]:*
obtains *"z* $\sim_\Gamma$ *ϱ" "¬z* $\sim_\Gamma$ i*" | "z* $\sim_\Gamma$ i*" "¬z* $\sim_\Gamma$ *ϱ" | "Im z > 0" "¬z* $\sim_\Gamma$ *ϱ" "¬z* $\sim_\Gamma$ i*" | "Im z* $\leq$ *0"*
⟨*proof*⟩

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*⟩

lemma (**in** *modgrp_subgroup*) *discrete_orbit: "discrete (orbit x)"*
⟨*proof*⟩

lemma (**in** *modgrp_subgroup*) *eventually_not_rel_at:*
"Im x > 0 ⟹ eventually (λy. ¬rel y x) (at x)"
⟨*proof*⟩

lemma (**in** *modgrp_subgroup*) *closed_orbit [intro]: "closedin (top_of_set {z. Im z > 0}) (orbit x)"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *analytic:*
assumes *"g analytic_on A"*
shows *"(λx. f (g x)) analytic_on A"*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *f_plus_of_nat: "f (z + of_nat n) = f z"*
⟨*proof*⟩

lemma *f_plus_of_int: "f (z + of_int n) = f z"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *relE:*
assumes *"lattice.rel z z'"*
obtains *m n :: int* **where** *"z = z' + of_int m + of_int n * t"*
⟨*proof*⟩

lemma *f_zero_cong_lattice:*
assumes *"lattice.rel z z'"*
shows *"f z = 0 ⟷ f z' = 0"*
⟨*proof*⟩

lemma *zorder_f_cong_lattice:*
assumes *"lattice.rel z z'"*
shows *"zorder f z = zorder f z'"*
⟨*proof*⟩

lemma *deriv_f_plus_1: "deriv f (z + 1) = deriv f z"*
⟨*proof*⟩

lemma *deriv_f_plus_quasiperiod:*
*"deriv f (z + t) = (deriv f z - 2 * pi * i * f z) / to_nome (2 * z + t)"*
⟨*proof*⟩

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*⟩

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"*
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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"*
  ⟨*proof*⟩

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*⟩

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*⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

lemma q_inversion_strict_antimono: "strict_antimono_on {0..1} q_inversion"
  ⟨proof⟩

lemma q_inversion_less_iff:
  assumes "q ∈ {0..1}" "q' ∈ {0..1}"
  shows   "q_inversion q < q_inversion q' ⟷ q > q'"
  ⟨proof⟩

lemma q_inversion_le_iff:
  assumes "q ∈ {0..1}" "q' ∈ {0..1}"
  shows   "q_inversion q ≤ q_inversion q' ⟷ q ≥ q'"
  ⟨proof⟩

lemma q_inversion_eq_iff:
  assumes "q ∈ {0..1}" "q' ∈ {0..1}"
  shows   "q_inversion q = q_inversion q' ⟷ q = q'"
  ⟨proof⟩

lemma q_inversion_involution:
  assumes "q ∈ {0..1}"
  shows   "q_inversion (q_inversion q) = q"
```

⟨*proof*⟩

lemma *continuous_q_inversion [continuous_intros]:*
assumes *q: "q ∈ {0..1}"*
shows *"continuous (at q within {0..1}) q_inversion"*
⟨*proof*⟩

lemma *continuous_on_q_inversion [continuous_intros]: "continuous_on {0..1} q_inversion"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

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*⟩

lemma *q_inversion_less_self_iff:*
assumes *"q ∈ {0..1}"*
shows *"q_inversion q < q ⟷ q > q_inversion_fixedpoint"*
⟨*proof*⟩

lemma *q_inversion_greater_self_iff:*
assumes *"q ∈ {0..1}"*
shows *"q_inversion q > q ⟷ q < q_inversion_fixedpoint"*
⟨*proof*⟩

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 (pi² / ln q)))"*

⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *holomorphic_dedekind_eta [holomorphic_intros]:*
assumes *"A ⊆ {z. Im z > 0}"*
shows *"η holomorphic_on A"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *analytic_dedekind_eta [analytic_intros]:*
assumes *"A ⊆ {z. Im z > 0}"*
shows *"η analytic_on A"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *continuous_on_dedekind_eta [continuous_intros]:*
"A ⊆ {z. Im z > 0} ⟹ continuous_on A η"
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *tendsto_dedekind_eta [tendsto_intros]:*
assumes *"(f ⟶ c) F" "Im c > 0"*
shows *"((λx. η (f x)) ⟶ η c) F"*
⟨*proof*⟩

lemma *tendsto_at_cusp_dedekind_eta [tendsto_intros]: "(η ⟶ 0) at_i∞"*
⟨*proof*⟩

lemma *dedekind_eta_plus1:*
assumes *z: "Im z > 0"*
shows *"η (z + 1) = cis (pi/12) * η z"*
⟨*proof*⟩

lemma *dedekind_eta_plus_nat:*

assumes *z: "Im z > 0"*
shows *"η (z + of_nat n) = cis (pi * n / 12) * η z"*
⟨*proof*⟩

lemma *dedekind_eta_plus_int:*
assumes *z: "Im z > 0"*
shows *"η (z + of_int n) = cis (pi*n/12) * η z"*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *dedekind_eta_minus_one_over:*
assumes *t: "Im t > 0"*
shows *"η (-(1/t)) = csqrt (-i*t) * η t"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *dedekind_eps_shift [simp]: "ε (shift_modgrp m) = cis (pi * m / 12)"*
⟨*proof*⟩

lemma *dedekind_eps_S [simp]: "dedekind_eps S_modgrp = cis (-pi / 4)"*
⟨*proof*⟩

lemma *dedekind_eps_shift_right [simp]: "ε (f * shift_modgrp m) = cis (pi * m / 12) * ε f"*
⟨*proof*⟩

lemma *dedekind_eps_shift_left [simp]: "ε (shift_modgrp m * f) = cis (pi * m / 12) * ε f"*
⟨*proof*⟩

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*⟩

lemma *dedekind_eps_root_of_unity: "ε f ^ 24 = 1"*
⟨*proof*⟩

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"*

⟨*proof*⟩

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*⟩

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"*
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *Eisenstein_E2_ii: "Eisenstein_E 2* i *= 3 / of_real pi"*
⟨*proof*⟩

lemma *Eisenstein_G2_rho: "Eisenstein_G 2 modfun_rho = of_real (2 / sqrt 3 * pi)"*
⟨*proof*⟩

lemma *Eisenstein_E2_rho: "Eisenstein_E 2 modfun_rho = of_real (2 * sqrt 3 / pi)"*
⟨*proof*⟩

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.