

Complex Lattices, Elliptic Functions, and the Modular Group

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

June 14, 2026

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 $z \rightarrow i\infty$
- 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	5
1.1	The z -plane vs the q -disc	5
1.1.1	The neighbourhood of $i\infty$	5
1.1.2	The parameter q	6
1.2	Parallelogram-shaped paths	9
2	Möbius transforms and the modular group	10
2.1	Basic properties of Möbius transforms	11
2.2	Unimodular Möbius transforms	12
2.3	The modular group	14
2.3.1	Definition	14
2.3.2	Basic operations	16
2.3.3	Basic properties	19
2.4	Code generation	27
2.5	The factor of automorphy and the slash operator	29
2.6	Sending rational numbers to infinity	31
2.7	Representation as product of powers of generators	31
2.8	Induction rules in terms of generators	35
2.9	Reduction map	36
3	Complex lattices	38
3.1	Basic definitions and useful lemmas	38
3.2	Period parallelograms	48
3.3	Canonical representatives and the fundamental parallelogram	50
3.4	Equivalence of fundamental pairs	54
3.5	Additional useful facts	55
3.6	Doubly-periodic functions	60
4	Subgroups of the modular group	60
4.1	Definition and group action on the upper half plane	61
4.2	Conjugation	64
4.3	Elliptic points	65
4.4	Subgroups containing shifts	65
4.4.1	Congruence subgroups	66
4.4.2	Hecke subgroups $\Gamma_0(N)$	69
4.5	The subgroups $\Gamma_1(N)$	71
5	Fundamental regions of the modular group	73
5.1	Definition	73
5.2	The standard fundamental region	74
5.3	Proving that the standard region is fundamental	77
5.4	The corner point of the standard fundamental region	79

5.5	The elliptic points of the modular group	81
5.6	Fundamental regions for congruence subgroups	83
6	Elliptic Functions	84
6.1	Definition	84
6.2	Basic results about zeros and poles	87
6.3	Even elliptic functions	91
6.4	Closure properties of the class of elliptic functions	92
6.5	Affine transformations and surjectivity	95
7	The Weierstraß \wp Function	96
7.1	Preliminary convergence results	97
7.2	Definition and basic properties	104
7.3	Ellipticity and poles	108
7.4	The numbers e_1, e_2, e_3	110
7.5	Injectivity of \wp	111
7.6	Invariance under lattice transformations	112
7.7	Construction of arbitrary elliptic functions from \wp	115
8	Related facts about Jacobi theta functions	118
8.1	Uniqueness of quasi-periodic entire functions	118
8.2	Theta inversion	122
8.3	Theta nullwert inversions in the reals	122
9	Eisenstein series and the differential equations of \wp	125
9.1	Definition	126
9.2	The Laurent series expansion of \wp at the origin	127
9.3	Differential equations for \wp	128
9.4	Lattice invariants and a recurrence for the Eisenstein series	129
9.5	Fourier expansion	131
9.6	Behaviour under lattice transformations	133
9.7	Recurrence relation	134
10	Addition and duplication theorems for \wp	137
11	Connection between complex lattices and theta functions	142
12	Eisenstein series and related invariants as modular forms	144
12.1	Eisenstein series	145
12.2	The monomials of the Eisenstein polynomial	149
12.3	The normalised Eisenstein series	150
12.4	Identities for normalised Eisenstein series	153
12.5	The modular discriminant	156
12.6	Ramanujan's tau function	157
12.7	The modular λ function	158

12.8 Klein's J invariant	160
12.9 Klein's c	161
12.10 Values at specific points	162
12.11 Consequences for the fundamental region	163
13 The Dedekind η function	164
13.1 Definition and basic properties	164
13.2 Relation to the Jacobi ϑ functions	166
13.3 The inversion identity	167
13.4 General transformation law	168
13.5 The transformation law for G_2	169

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 $\text{Im}(z) > 0$ 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 $\infty$ ") where
  "at_ii_inf = filtercomap Im at_top"
```

```
lemma eventually_at_ii_inf:
  "eventually ( $\lambda z. \text{Im } z > c$ ) at_ii_inf"
  <proof>
```

```
lemma eventually_at_ii_inf_iff:
  " $(\forall_F z \text{ in } at\_ii\_inf. P z) \iff (\exists c. \forall z. \text{Im } z > c \longrightarrow P z)$ "
  <proof>
```

```
lemma eventually_at_ii_inf_iff':
  " $(\forall_F z \text{ in } at\_ii\_inf. P z) \iff (\exists c. \forall z. \text{Im } z \geq c \longrightarrow P z)$ "
  <proof>
```

```
lemma filterlim_Im_at_ii_inf: "filterlim Im at_top at_i $\infty$ "
  <proof>
```

```
lemma filterlim_at_ii_infI:
  assumes "filterlim f F at_top"
  shows "filterlim ( $\lambda x. f (\text{Im } x)$ ) F at_i $\infty$ "
  <proof>
```

```
lemma filtermap_scaleR_at_ii_inf:
  assumes "c > 0"
  shows "filtermap ( $\lambda z. c *_R z$ ) at_ii_inf = at_ii_inf"
```

<proof>

lemma *at_ii_inf_neq_bot [simp]: "at_ii_inf \neq 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 $\text{Re}(z) \in (-\frac{1}{2}, \frac{1}{2}]$ as the codomain of the inverse mapping.

definition *to_q :: "nat \Rightarrow complex \Rightarrow 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<..
 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  $\tau$  = to_q m  $\tau'$ " "m > 0"
  obtains n where " $\tau' = \tau + \text{of\_int } n * \text{of\_nat } m$ "
  <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  $\implies$  filterlim
(to_q n) (at 0) at_ii_inf"
  <proof>

lemma eventually_to_q_neq:
  assumes n: "n > 0"
  shows "eventually ( $\lambda w. \text{to\_q } n \ w \neq \text{to\_q } n \ z$ ) (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)  $\longleftrightarrow$  is_pole (f o to_q n) x"
  <proof>

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

lemma Im_of_q: "q  $\neq$  0  $\implies$  n > 0  $\implies$  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 <..

```

1.2 Parallelogram-shaped paths

```
theory Parallelogram_Paths
```

```

imports "HOL-Complex_Analysis.Complex_Analysis"
begin

definition parallelogram_path :: "'a :: real_normed_vector  $\Rightarrow$  'a  $\Rightarrow$  'a
 $\Rightarrow$  real  $\Rightarrow$  '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  $\equiv$  ( $\lambda w. z + \text{Re } w *_R a + \text{Im } w *_R b$ )"
  shows "parallelogram_path z a b = g  $\circ$  rectpath 0 (1 + i)"
  <proof>

lemma
  fixes f :: "complex  $\Rightarrow$  complex" and z  $\omega_1$   $\omega_2$  :: complex
  defines "I  $\equiv$  ( $\lambda a b. \text{contour\_integral (linepath (z + a) (z + b)) f}$ )"
  defines "P  $\equiv$  parallelogram_path z  $\omega_1$   $\omega_2$ "
  assumes "continuous_on (path_image P) f"
  shows contour_integral_parallelogram_path:
    "contour_integral P f =
      (I 0  $\omega_1$  - I  $\omega_2$  ( $\omega_1 + \omega_2$ )) - (I 0  $\omega_2$  - I  $\omega_1$  ( $\omega_1 + \omega_2$ ))"
  and contour_integral_parallelogram_path':
    "contour_integral P f =
      contour_integral (linepath z (z +  $\omega_1$ )) ( $\lambda x. f x - f (x +$ 
 $\omega_2)$ ) -
      contour_integral (linepath z (z +  $\omega_2$ )) ( $\lambda x. f x - f (x +$ 
 $\omega_1)$ )"
  <proof>

end

```

2 Möbius transforms and the modular group

```

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

```

begin

$\langle proof \rangle \langle proof \rangle$

2.1 Basic properties of Möbius transforms

lemma moebius_uminus [simp]: "moebius (-a) (-b) (-c) (-d) = moebius a b c d"
 $\langle proof \rangle$

lemma moebius_uminus': "moebius (-a) b c d = moebius a (-b) (-c) (-d)"
 $\langle proof \rangle$

lemma moebius_diff_eq:
fixes a b c d :: "'a :: field"
defines "f \equiv moebius a b c d"
assumes *: "c = 0 \vee z \neq -d / c \wedge w \neq -d / c"
shows "f w - f z = (a * d - b * c) / ((c * w + d) * (c * z + d)) * (w - z)"
 $\langle proof \rangle$

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"
 $\langle proof \rangle$

lemma moebius_eq_shift: "moebius 1 (of_int n) 0 1 z = z + of_int n"
 $\langle proof \rangle$

lemma moebius_S:
assumes "a * d - b * c \neq 0" "z \neq 0"
shows "moebius a b c d (-1 / z) = moebius b (- a) d (- c) (z :: 'a :: field)"
 $\langle proof \rangle$

lemma moebius_eq_S: "moebius 0 1 (-1) 0 z = -1 / z"
 $\langle proof \rangle$

lemma continuous_on_moebius [continuous_intros]:
fixes a b c d :: "'a :: real_normed_field"
assumes "c \neq 0 \vee d \neq 0" "c = 0 \vee -d / c \notin A"
shows "continuous_on A (moebius a b c d)"
 $\langle proof \rangle$

lemma continuous_on_moebius' [continuous_intros]:
fixes a b c d :: "'a :: real_normed_field"
assumes "continuous_on A f" "c \neq 0 \vee d \neq 0" " $\bigwedge z. z \in A \implies c = 0 \vee f z \neq -d / c$ "

shows "continuous_on A ($\lambda x. \text{moebius } a \ b \ c \ d \ (f \ x)$)"
 <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" " $\bigwedge z. z \in A \implies c = 0 \vee f \ z \neq -d / c$ "
 shows " $(\lambda x. \text{moebius } a \ b \ c \ d \ (f \ x))$ 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" " $\bigwedge z. z \in A \implies c = 0 \vee f \ z \neq -d / c$ "
 shows " $(\lambda x. \text{moebius } a \ b \ c \ d \ (f \ x))$ 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 \Rightarrow complex" where
 " $\varphi = \text{moebius } (\text{of_int } a) \ (\text{of_int } b) \ (\text{of_int } c) \ (\text{of_int } d)$ "

lemma cnj_φ: " $\varphi \ (\text{cnj } z) = \text{cnj } (\varphi \ z)$ "
 <proof>

lemma Im_transform:
 " $\text{Im } (\varphi \ z) = \text{Im } z / \text{norm } (\text{of_int } c * z + \text{of_int } d) ^ 2$ "
 <proof>

```

lemma Im_transform_pos_aux:
  assumes "Im z  $\neq$  0"
  shows "of_int c * z + of_int d  $\neq$  0"
  <proof>

lemma Im_transform_pos: "Im z > 0  $\implies$  Im ( $\varphi$  z) > 0"
  <proof>

lemma Im_transform_neg: "Im z < 0  $\implies$  Im ( $\varphi$  z) < 0"
  <proof>

lemma Im_transform_zero_iff [simp]: "Im ( $\varphi$  z) = 0  $\longleftrightarrow$  Im z = 0"
  <proof>

lemma Im_transform_pos_iff [simp]: "Im ( $\varphi$  z) > 0  $\longleftrightarrow$  Im z > 0"
  <proof>

lemma Im_transform_neg_iff [simp]: "Im ( $\varphi$  z) < 0  $\longleftrightarrow$  Im z < 0"
  <proof>

lemma Im_transform_nonneg_iff [simp]: "Im ( $\varphi$  z)  $\geq$  0  $\longleftrightarrow$  Im z  $\geq$  0"
  <proof>

lemma Im_transform_nonpos_iff [simp]: "Im ( $\varphi$  z)  $\leq$  0  $\longleftrightarrow$  Im z  $\leq$  0"
  <proof>

lemma transform_in_reals_iff [simp]: " $\varphi$  z  $\in \mathbb{R}$   $\longleftrightarrow$  z  $\in \mathbb{R}$ "
  <proof>

end

lemma Im_one_over_neg_iff [simp]: "Im (1 / z) < 0  $\longleftrightarrow$  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_ $\varphi$ :
  assumes "of_int c * z + of_int d  $\neq$  0"
  shows "inv. $\varphi$  ( $\varphi$  z) = z"
  <proof>

lemma inv_ $\varphi$ ':

```

```

    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 all integer tuples  $(a, b, c, d)$  with  $ad - bc = 1$ .

typedef modgrp = "{(a,b,c,d::int). a * d - b * c = 1}"
  <proof>

setup_lifting type_definition_modgrp

instantiation modgrp :: one
begin

lift_definition one_modgrp :: modgrp is "(1, 0, 0, 1)"
  <proof>

instance <proof>
end

instantiation modgrp :: uminus
begin

lift_definition uminus_modgrp :: "modgrp ⇒ modgrp" is "λ(a,b,c,d). (-a,
-b, -c, -d)"
  <proof>

instance <proof>
end

lemma minus_minus_modgrp [simp]: "-(-f :: modgrp) = f"
  <proof>

instantiation modgrp :: sgn
begin

lift_definition sgn_modgrp :: "modgrp ⇒ modgrp"
  is "λ(a,b,c,d). if c < 0 ∨ c = 0 ∧ d < 0 then (-1,0,0,-1) else (1,0,0,1)"
  <proof>

instance <proof>

```

```

end

instantiation modgrp :: abs
begin

lift_definition abs_modgrp :: "modgrp  $\Rightarrow$  modgrp"
  is " $\lambda(a,b,c,d). \text{if } c < 0 \vee c = 0 \wedge d < 0 \text{ then } (-a,-b,-c,-d) \text{ else } (a,b,c,d)$ "
  <proof>

instance <proof>

end

instantiation modgrp :: times
begin

lift_definition times_modgrp :: "modgrp  $\Rightarrow$  modgrp  $\Rightarrow$  modgrp"
  is " $\lambda(a,b,c,d) (a',b',c',d'). (a * a' + b * c', a * b' + b * d', c * a' + d * c', c * b' + d * d')$ "
  <proof>

instance <proof>
end

instantiation modgrp :: inverse
begin

lift_definition inverse_modgrp :: "modgrp  $\Rightarrow$  modgrp"
  is " $\lambda(a, b, c, d). (d, -b, -c, a)$ "
  <proof>

definition divide_modgrp :: "modgrp  $\Rightarrow$  modgrp  $\Rightarrow$  modgrp" where
  "divide_modgrp x y = x * inverse y"

instance <proof>

end

interpretation modgrp: Groups.group "(*)" :: modgrp  $\Rightarrow$  _" 1 inverse
<proof>

instance modgrp :: monoid_mult
  <proof>

lemma inverse_power_modgrp: "inverse (x ^ n :: modgrp) = inverse x ^ n"

```

<proof>

lemma *inverse_mult_assoc1_modgrp [simp]: "inverse f * (f * g) = (g :: modgrp)"*
<proof>

lemma *inverse_mult_assoc2_modgrp [simp]: "f * (inverse f * g) = (g :: modgrp)"*
<proof>

lemma *abs_modgrp_conv_sgn: "abs f = sgn f * (f :: modgrp)"*
<proof>

lemma *modgrp_conv_sgn_abs: "f = sgn f * (abs f :: modgrp)"*
<proof>

lemma *sgn_1_modgrp [simp]: "sgn (1 :: modgrp) = 1"*
<proof>

lemma *sgn_uminus_modgrp [simp]: "sgn (-f :: modgrp) = -sgn f"*
<proof>

lemma *sgn_sgn_modgrp [simp]: "sgn (sgn f) = sgn (f :: modgrp)"*
<proof>

lemma *sgn_abs_modgrp [simp]: "sgn (abs f) = (1 :: modgrp)"*
<proof>

lemma *abs_1_modgrp [simp]: "abs (1 :: modgrp) = 1"*
<proof>

lemma *abs_uminus_modgrp [simp]: "abs (-f :: modgrp) = abs f"*
<proof>

lemma *abs_sgn_modgrp [simp]: "abs (sgn f) = (1 :: modgrp)"*
<proof>

lemma *abs_abs_modgrp [simp]: "abs (abs f) = abs (f :: modgrp)"*
<proof>

lemma *minus_modgrp_neq_self [simp]: "-f ≠ (f :: modgrp)" "f ≠ -f"*
<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 \Rightarrow modgrp" is " $\lambda n. (1, n, 0, 1)$ "
<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 \Rightarrow bool" is " $\lambda(a,b,c,d). c \neq 0$ "
<proof>

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

lift_definition *pole_modgrp* :: "modgrp \Rightarrow 'a :: field" is " $\lambda(a,b,c,d). -\text{of_int } d / \text{of_int } c$ "
<proof>

lemma *pole_modgrp_in_Rats*: "pole_modgrp f \in (Q :: 'a :: real_field set)"
<proof>

lemma *pole_modgrp_in_Reals*: "pole_modgrp f \in (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 \Rightarrow 'a :: field" is " $\lambda(a,b,c,d). \text{of_int } a / \text{of_int } c$ "
<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 \Rightarrow int" is " $\lambda(a,b,c,d). a$ " *<proof>*

lift_definition *modgrp_b* :: "modgrp \Rightarrow int" is " $\lambda(a,b,c,d). b$ " *<proof>*

lift_definition *modgrp_c* :: "modgrp \Rightarrow int" is " $\lambda(a,b,c,d). c$ " *<proof>*

lift_definition *modgrp_d* :: "modgrp \Rightarrow int" is " $\lambda(a,b,c,d). d$ " *<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_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 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⟩

```

```

lemma sgn_modgrp_altdef:
  "sgn f = (if modgrp_c f < 0 ∨ modgrp_c f = 0 ∧ modgrp_d f < 0 then
-1 else 1)"
⟨proof⟩

```

```

lemma abs_modgrp_altdef:
  "abs f = (if modgrp_c f < 0 ∨ modgrp_c f = 0 ∧ modgrp_d f < 0 then
-f else f)"
⟨proof⟩

```

```

lemma abs_eq_modgrpE:

```

```

assumes "abs f = (g :: modgrp)"
obtains "f = g" | "f = -g"
⟨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_eq_iff:
  "f = g ⟷ modgrp_a f = modgrp_a g ∧ modgrp_b f = modgrp_b g ∧
    modgrp_c f = modgrp_c g ∧ modgrp_d f = modgrp_d g"
⟨proof⟩

```

```

lemma modgrp_wrong: "a * d - b * c ≠ 1 ⟹ modgrp a b c d = 1"
⟨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_a_modgrp: "modgrp_a (modgrp a b c d) = a"
    and modgrp_b_modgrp: "modgrp_b (modgrp a b c d) = b"
    and modgrp_c_modgrp: "modgrp_c (modgrp a b c d) = c"
    and modgrp_d_modgrp: "modgrp_d (modgrp a b c d) = d"
⟨proof⟩

```

```

lemmas modgrp_abcd_modgrp = modgrp_a_modgrp modgrp_b_modgrp modgrp_c_modgrp
modgrp_d_modgrp

```

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_commutates:
  "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_uminus_modgrp [simp]: "apply_modgrp (-f) z = apply_modgrp
f 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 apply_modgrp_eq_apply_modgrp_iff:
  assumes "Im w > 0" "Im z > 0"
  shows "apply_modgrp f w = apply_modgrp g z ↔ apply_modgrp (inverse
g * f) w = z"
  ⟨proof⟩

lemma apply_modgrp_sgn [simp]: "apply_modgrp (sgn f) = (λx. x)"
  ⟨proof⟩

lemma apply_modgrp_abs [simp]: "apply_modgrp (abs f) = apply_modgrp f"
  ⟨proof⟩

```

```

lemma is_singular_modgrp_altdef: "is_singular_modgrp x  $\longleftrightarrow$  modgrp_c
x  $\neq$  0"
  <proof>

lemma not_is_singular_modgrpD:
  assumes " $\neg$ is_singular_modgrp x"
  shows "abs x = shift_modgrp (sgn (modgrp_a x) * modgrp_b x)"
  <proof>

lemma is_singular_modgrp_inverse [simp]: "is_singular_modgrp (inverse
x)  $\longleftrightarrow$  is_singular_modgrp x"
  <proof>

lemma is_singular_modgrp_S_left_iff [simp]: "is_singular_modgrp (S_modgrp
* f)  $\longleftrightarrow$  modgrp_a f  $\neq$  0"
  <proof>

lemma is_singular_modgrp_S_right_iff [simp]: "is_singular_modgrp (f *
S_modgrp)  $\longleftrightarrow$  modgrp_d f  $\neq$  0"
  <proof>

lemma is_singular_modgrp_T_left_iff [simp]:
  "is_singular_modgrp (T_modgrp * f)  $\longleftrightarrow$  is_singular_modgrp f"
  <proof>

lemma is_singular_modgrp_T_right_iff [simp]:
  "is_singular_modgrp (f * T_modgrp)  $\longleftrightarrow$  is_singular_modgrp f"
  <proof>

lemma is_singular_modgrp_shift_left_iff [simp]:
  "is_singular_modgrp (shift_modgrp n * f)  $\longleftrightarrow$  is_singular_modgrp f"
  <proof>

lemma is_singular_modgrp_shift_right_iff [simp]:
  "is_singular_modgrp (f * shift_modgrp n)  $\longleftrightarrow$  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  $\in$  ( $\mathbb{R}$  :: '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 is_singular_modgrp_uminus_iff [simp]: "is_singular_modgrp (-f)
↔ is_singular_modgrp f"
  ⟨proof⟩

lemma times_modgrp_uminus_left [simp]: "(-f :: modgrp) * g = -(f * g)"
  ⟨proof⟩

lemma times_modgrp_uminus_right [simp]: "(f :: modgrp) * (-g) = -(f *
g)"
  ⟨proof⟩

lemma inverse_modgrp_uminus [simp]: "inverse (-f :: modgrp) = -inverse
f"
  ⟨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 \neq S_modgrp"
 <proof>

lemma *S_neq_shift_modgrp* [simp]: "S_modgrp \neq shift_modgrp n"
 <proof>

lemma *shift_modgrp_eq_T_iff* [simp]: "shift_modgrp n = T_modgrp \longleftrightarrow n = 1"
 <proof>

lemma *T_eq_shift_modgrp_iff* [simp]: "T_modgrp = shift_modgrp n \longleftrightarrow 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 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⟩

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 not_singular_modgrpD:
  assumes "¬is_singular_modgrp f"
  shows "abs f = shift_modgrp (modgrp_b (abs 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_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_uminus: "a * d - b * c = 1 ⇒ -modgrp a b c d = modgrp
(-a) (-b) (-c) (-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 modgrp_a_uminus [simp]: "modgrp_a (-f) = -modgrp_a f"
  and modgrp_b_uminus [simp]: "modgrp_b (-f) = -modgrp_b f"
  and modgrp_c_uminus [simp]: "modgrp_c (-f) = -modgrp_c f"
  and modgrp_d_uminus [simp]: "modgrp_d (-f) = -modgrp_d f"
  ⟨proof⟩

lemma modgrp_a_inverse [simp]: "modgrp_a (inverse f) = modgrp_d f"
  and modgrp_b_inverse [simp]: "modgrp_b (inverse f) = -modgrp_b f"
  and modgrp_c_inverse [simp]: "modgrp_c (inverse f) = -modgrp_c f"
  and modgrp_d_inverse [simp]: "modgrp_d (inverse f) = modgrp_a f"
  ⟨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_times [simp]: "modgrp_a (f * g) = modgrp_a f * modgrp_a
g + modgrp_b f * modgrp_c g"
  and modgrp_b_times [simp]: "modgrp_b (f * g) = modgrp_a f * modgrp_b
g + modgrp_b f * modgrp_d g"
  and modgrp_c_times [simp]: "modgrp_c (f * g) = modgrp_c f * modgrp_a
g + modgrp_d f * modgrp_c g"
  and modgrp_d_times [simp]: "modgrp_d (f * g) = modgrp_c f * modgrp_b
g + modgrp_d f * modgrp_d g"
  ⟨proof⟩

lemma modgrp_a_code [code]:
  "modgrp_a (modgrp a b c d) = (if a * d - b * c = 1 then a else 1)"
  ⟨proof⟩

lemma modgrp_b_code [code]:
  "modgrp_b (modgrp a b c d) = (if a * d - b * c = 1 then 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 d else 1)"
  ⟨proof⟩

lemma sgn_modgrp_code [code]:
  "sgn (modgrp a b c d) = (if a * d - b * c = 1 ∧ (c < 0 ∨ c = 0 ∧ d
< 0) then -1 else 1)"
  ⟨proof⟩

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

```

2.5 The factor of automorphy and the slash operator

The following will be needed to define the slash operator and it appears in a few other places as well. Diamond and Shurman call it the *factor of automorphy* of a modular transformation.

```

lift_definition automorphy_factor :: "modgrp ⇒ complex ⇒ complex" is
  "λ(a,b,c,d) z. of_int c * z + of_int d" <proof>

```

```

lemma automorphy_factor_altdef:
  "automorphy_factor g z = of_int (modgrp_c g) * z + of_int (modgrp_d
g)"
  <proof>

```

```

lemma automorphy_factor_uminus [simp]: "automorphy_factor (-h) z = -automorphy_factor
h z"
  <proof>

```

```

lemma automorphy_factor_1 [simp]: "automorphy_factor 1 z = 1"
  <proof>

```

```

lemma automorphy_factor_shift [simp]: "automorphy_factor (shift_modgrp
n) z = 1"
  <proof>

```

```

lemma automorphy_factor_T [simp]: "automorphy_factor T_modgrp z = 1"
  <proof>

```

```

lemma automorphy_factor_S [simp]: "automorphy_factor S_modgrp z = z"
  <proof>

```

```

lemma automorphy_factor_shift_right [simp]:
  "automorphy_factor (f * shift_modgrp n) z = automorphy_factor f (z +
of_int n)"
  <proof>

```

```

lemma automorphy_factor_shift_left [simp]:
  "automorphy_factor (shift_modgrp n * f) z = automorphy_factor f z"
  <proof>

```

```

lemma automorphy_factor_T_right [simp]:
  "automorphy_factor (f * T_modgrp) z = automorphy_factor f (z + 1)"
  ⟨proof⟩

lemma automorphy_factor_T_left [simp]:
  "automorphy_factor (T_modgrp * f) z = automorphy_factor f z"
  ⟨proof⟩

lemma automorphy_factor_nonzero [simp]:
  assumes "Im z ≠ 0"
  shows "automorphy_factor g z ≠ 0"
  ⟨proof⟩

lemma automorphy_factor_mult:
  assumes "Im z ≠ 0"
  shows "automorphy_factor (f * g) z =
        automorphy_factor f (apply_modgrp g z) * automorphy_factor
g z"
  ⟨proof⟩

lemma has_field_derivative_automorphy_factor [derivative_intros]:
  assumes "(f has_field_derivative f') (at x within A)"
  shows "((λx. automorphy_factor g (f x)) has_field_derivative (of_int
(modgrp_c g) * f')) (at x within A)"
  ⟨proof⟩

lemma automorphy_factor_analytic [analytic_intros]: "automorphy_factor
g analytic_on A"
  ⟨proof⟩

lemma automorphy_factor_meromorphic [meromorphic_intros]: "automorphy_factor
h meromorphic_on A"
  ⟨proof⟩

lemma tendsto_automorphy_factor [tendsto_intros]:
  "(f → c) F ⇒ ((λx. automorphy_factor g (f x)) → automorphy_factor
g c) F"
  ⟨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 (1 / automorphy_factor
f x ^ 2)) (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 $((\lambda x. \text{apply_modgrp } f (g x)) \text{ has_field_derivative } g' / \text{automorphy_factor } f (g x) ^ 2)$
 (at x within A)"
 $\langle \text{proof} \rangle$

lemma Im_apply_modgrp : $\text{"Im (apply_modgrp } f t) = \text{Im } t / \text{norm (automorphy_factor } f t) ^ 2\text{"}$
 $\langle \text{proof} \rangle$

2.6 Sending rational numbers to infinity

The following gives a unimodular transformation that sends a specific rational number to infinity. This is not unique.

lift_definition $\text{modgrp_of_rat} :: \text{"rat} \Rightarrow \text{modgrp}"$ is
 $\text{"}\lambda x. \text{let } (u, v) = \text{quotient_of } x; (a, b) = \text{bezout_coefficients } u v \text{ in } (a, b, -v, u)\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{modgrp_of_rat_of_int}$: $\text{"modgrp_of_rat (of_int } n) = -S_modgrp * \text{shift_modgrp } (-n)\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{modgrp_of_rat_of_nat}$: $\text{"modgrp_of_rat (of_nat } n) = -S_modgrp * \text{shift_modgrp } (-\text{int } n)\text{"}$
 $\langle \text{proof} \rangle$

lemma modgrp_of_rat_0 [simp]: $\text{"modgrp_of_rat } 0 = -S_modgrp\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{pole_modgrp_of_rat}$ [simp]: $\text{"pole_modgrp (modgrp_of_rat } x) = x\text{"}$
 $\langle \text{proof} \rangle$

2.7 Representation as product of powers of generators

definition $\text{modgrp_from_gens} :: \text{"int option list} \Rightarrow \text{modgrp}"$ where
 $\text{"modgrp_from_gens } xs = \text{prod_list (map } (\lambda x. \text{case } x \text{ of None} \Rightarrow S_modgrp \mid \text{Some } n \Rightarrow \text{shift_modgrp } n) xs)\text{"}$

lemma $\text{modgrp_from_gens_Nil}$ [simp]:
 $\text{"modgrp_from_gens } [] = 1\text{"}$
 and $\text{modgrp_from_gens_append}$ [simp]:
 $\text{"modgrp_from_gens (xs @ ys) = modgrp_from_gens } xs * \text{modgrp_from_gens } ys\text{"}$
 and $\text{modgrp_from_gens_Cons1}$ [simp]:
 $\text{"modgrp_from_gens (None \# xs) = S_modgrp * modgrp_from_gens } xs\text{"}$
 and $\text{modgrp_from_gens_Cons2}$ [simp]:
 $\text{"modgrp_from_gens (Some } n \# xs) = \text{shift_modgrp } n * \text{modgrp_from_gens } xs\text{"}$
 and $\text{modgrp_from_gens_Cons}$:

```

      "modgrp_from_gens (x # xs) =
        (case x of None  $\Rightarrow$  S_modgrp | Some n  $\Rightarrow$  shift_modgrp n) *
modgrp_from_gens xs"
    <proof>

definition invert_modgrp_gens :: "int option list  $\Rightarrow$  int option list"
  where "invert_modgrp_gens = rev  $\circ$  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 sgn_S_modgrp [simp]: "sgn S_modgrp = 1"
  <proof>

lemma sgn_T_modgrp [simp]: "sgn T_modgrp = 1"
  <proof>

lemma sgn_shift_modgrp [simp]: "sgn (shift_modgrp n) = 1"
  <proof>

lemma abs_S_modgrp [simp]: "abs S_modgrp = S_modgrp"
  <proof>

lemma abs_T_modgrp [simp]: "abs T_modgrp = T_modgrp"
  <proof>

lemma abs_shift_modgrp [simp]: "abs (shift_modgrp n) = shift_modgrp n"
  <proof>

lemma abs_modgrp_eqE:
  assumes "abs f = abs g"
  obtains "f = g" | "f = (-g :: modgrp)"
  <proof>

lemma abs_modgrp_eq_1_iff: "abs (f :: modgrp) = 1  $\longleftrightarrow$  f = 1  $\vee$  f = -1"
  <proof>

```

```

lemma abs_modgrp_uminus_cong:
  "abs f = abs f'  $\implies$  abs (-f) = abs (-f' :: modgrp)"
  <proof>

lemma abs_modgrp_mult_cong:
  "abs f = abs f'  $\implies$  abs g = abs g'  $\implies$  abs (f * g) = abs (f' * g' ::
modgrp)"
  <proof>

lemma abs_modgrp_inverse_cong:
  "abs f = abs f'  $\implies$  abs (inverse f) = abs (inverse f' :: modgrp)"
  <proof>

lemmas abs_modgrp_congs = abs_modgrp_uminus_cong abs_modgrp_mult_cong
abs_modgrp_inverse_cong

lemma modgrp_from_gens_invert [simp]:
  "abs (modgrp_from_gens (invert_modgrp_gens xs)) = abs (inverse (modgrp_from_gens
xs))"
  <proof>

function modgrp_genseq :: "int  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  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  $\neq$  0  $\implies$  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 "abs (modgrp_from_gens (modgrp_genseq a b c d)) = abs (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 " \neg is_singular_modgrp $g \vee z \neq$ pole_modgrp g "

shows " $zorder (\lambda x. apply_modgrp\ g\ x - apply_modgrp\ g\ z)\ z = 1$ "

<proof>

lemma *zorder_fls_modgrp_pole:*

assumes "is_singular_modgrp f "

shows " $zorder (apply_modgrp\ f)\ (pole_modgrp\ f) = -1$ "

<proof>

2.8 Induction rules in terms of generators

Theorem 2.1

lemma *modgrp_induct_S_shift* [*case_names id uminus S shift*]:

assumes " $P\ 1$ "

" $\bigwedge x. P\ x \implies P\ (-x)$ "

" $\bigwedge x. P\ x \implies P\ (S_modgrp\ *\ x)$ "

" $\bigwedge x\ n. P\ x \implies P\ (shift_modgrp\ n\ *\ x)$ "

shows " $P\ x$ "

<proof>

lemma *modgrp_induct* [*case_names id uminus S T inv_T*]:

assumes " $P\ 1$ "

" $\bigwedge x. P\ x \implies P\ (-x)$ "

" $\bigwedge x. P\ x \implies P\ (S_modgrp\ *\ x)$ "

" $\bigwedge x. P\ x \implies P\ (T_modgrp\ *\ x)$ "

" $\bigwedge x. P\ x \implies P\ (inverse\ T_modgrp\ *\ x)$ "

shows " $P\ x$ "

<proof>

lemma *modgrp_induct_S_shift'* [*case_names id uminus S shift*]:

assumes " $P\ 1$ "

" $\bigwedge x. P\ x \implies P\ (-x)$ "

" $\bigwedge x. abs\ x = x \implies P\ x \implies P\ (x\ * S_modgrp)$ "

" $\bigwedge x\ n. abs\ x = x \implies P\ x \implies P\ (x\ * shift_modgrp\ n)$ "

shows " $P\ x$ "

<proof>

lemma *modgrp_induct'* [*case_names id uminus S T inv_T*]:

assumes " $P\ 1$ "

" $\bigwedge x. P\ x \implies P\ (-x)$ "

" $\bigwedge x. P\ x \implies P\ (x\ * S_modgrp)$ "

" $\bigwedge x. P\ x \implies P\ (x\ * T_modgrp)$ "

" $\bigwedge x. P\ x \implies P\ (x\ * inverse\ T_modgrp)$ "

shows " $P\ x$ "

<proof>

```

definition apply_modgrp' :: "modgrp  $\Rightarrow$  'a  $\times$  'a  $\Rightarrow$  'a  $\times$  'a :: ring_1"
  where "apply_modgrp' f =
    ( $\lambda(x,y).$  (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  $\notin$   $\mathbb{R}$ "
  shows "apply_modgrp' f (z, 1) = (automorphy_factor f z * apply_modgrp
    f z, automorphy_factor f z)"
  <proof>

```

2.9 Reduction map

Two elements of the modular group are considered congruent modulo n if their entries are.

```

definition cong_modgrp :: "modgrp  $\Rightarrow$  modgrp  $\Rightarrow$  int  $\Rightarrow$  bool"
  (<<indent=1 notation=<mixfix cong>>[_ = _] '( ' mod $_{\Gamma}$  _ '))>> where
  "[f = g] (mod $_{\Gamma}$  n)  $\longleftrightarrow$  list_all ( $\lambda h.$  [h f = h g] (mod n)) [modgrp_a,
    modgrp_b, modgrp_c, modgrp_d]"

```

```

lemma cong_modgrp_abs [simp]: "[f = g] (mod $_{\Gamma}$  (abs n))  $\longleftrightarrow$  [f = g] (mod $_{\Gamma}$ 
  n)"
  <proof>

```

```

lemma cong_modgrp_refl [intro]: "[f = f] (mod $_{\Gamma}$  n)"
  <proof>

```

```

lemma cong_modgrp_mult:
  "[f1 = g1] (mod $_{\Gamma}$  n)  $\implies$  [f2 = g2] (mod $_{\Gamma}$  n)  $\implies$  [f1 * f2 = g1 * g2]
  (mod $_{\Gamma}$  n)"
  <proof>

```

```

lemma cong_modgrp_inverse:
  "[f = g] (mod $_{\Gamma}$  n)  $\implies$  [inverse f = inverse g] (mod $_{\Gamma}$  n)"
  <proof>

```

The following version of the Chinese Remainder Theorem also works when the two moduli are not coprime.

```

lemma chinese_remainder_theorem_gen:
  fixes m n :: "'a :: {unique_euclidean_ring, euclidean_ring_gcd}"
  assumes "[x = y] (mod gcd m n)"
  obtains z where "[z = x] (mod m)" "[z = y] (mod n)"
  <proof>

```

```

lemma cong_solve:
  fixes a :: "'a :: {unique_euclidean_semiring, euclidean_ring_gcd}"

```

shows " $\exists x. [a * x = \text{gcd } a \ n] \pmod n$ "
 <proof>

lemma *cong_solve_coprime*:
 fixes a :: "'a :: {unique_euclidean_ring, euclidean_ring_gcd}"
 shows " $\text{coprime } a \ n \implies \exists x. [a * x = 1] \pmod n$ "
 <proof>

lemma *chinese_remainder*:
 fixes A :: "'a set"
 and m :: "'a \Rightarrow 'b :: {unique_euclidean_ring, euclidean_ring_gcd}"
 and u :: "'a \Rightarrow 'b"
 assumes fin: "finite A"
 and cop: " $\forall i \in A. \forall j \in A. i \neq j \longrightarrow \text{coprime } (m \ i) \ (m \ j)$ "
 shows " $\exists x. \forall i \in A. [x = u \ i] \pmod {m \ i}$ "
 <proof>

lemma *coprime_via_prime_factors*:
 fixes x y :: "'a :: factorial_semiring_gcd"
 assumes "x \neq 0" " $\bigwedge p. \text{prime } p \implies p \text{ dvd } x \implies \neg p \text{ dvd } y$ "
 shows "coprime x y"
 <proof>

The following shows that the reduction map $\text{SL}(2, \mathbb{Z}) \rightarrow \text{SL}(2, \mathbb{Z}/n\mathbb{Z})$ is surjective.

This means that if we have a matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $ad - bc = 1 \pmod n$,

there exists a matrix $A' = \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix}$ with $a'd' - b'c' = 1$ and $A = A' \pmod n$.

So, basically, we can take a matrix whose determinant is only congruent to 1 modulo n and turn it into one with determinant exactly 1 while only adding and subtracting multiples of n to the entries.

lemma *exists_coprime_shifted_int_aux*:
 fixes c d :: int
 assumes c: "c \neq 0"
 assumes coprime: "coprime (gcd c d) n"
 obtains d' where "[d' = d] $\pmod n$ " "coprime c d'"
 <proof>

lemma *exists_coprime_shifted_int*:
 fixes c d :: int
 assumes coprime: "coprime (gcd c d) n"
 obtains c' d' where "[c' = c] $\pmod n$ " "[d' = d] $\pmod n$ " "coprime c' d'"
 <proof>

```

theorem modgrp_reduction_surj:
  fixes a b c d n :: int
  assumes "[a * d - b * c = 1] (mod n)"
  obtains f :: modgrp where
    "[modgrp_a f = a] (mod n)" "[modgrp_b f = b] (mod n)"
    "[modgrp_c f = c] (mod n)" "[modgrp_d f = d] (mod n)"
  <proof>

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

```

```

lemma (in -) image_plus_conv_vimage_plus:
  fixes c :: "'a :: group_add"
  shows "(+) c ` A = (+) (-c) -` A"
  <proof>

```

```

lemma filtermap_cosparse_translate:
  "filtermap ((+) (c :: 'a :: real_normed_vector)) (cosparse A) = cosparse
  ((+) c ` A)"
  <proof>

```

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  $\omega_1 \omega_2 :: complex$ 
begin

```

The following function convergs from lattice coordinates into cartesian coordinates.

```

definition of_omega12_coords :: "real  $\times$  real  $\Rightarrow$  complex" where
  "of_omega12_coords = ( $\lambda(x,y)$ . of_real x *  $\omega_1$  + of_real y *  $\omega_2$ )"

```

```

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*]: " $ab \in \mathbb{Z} \times \mathbb{Z} \implies \text{of_}\omega12_coords$
 $ab \in \Lambda$ "
 ⟨*proof*⟩

lemma *lattice_plus_right_cancel* [*simp*]: " $y \in \Lambda \implies x + y \in \Lambda \longleftrightarrow x$
 $\in \Lambda$ "
 ⟨*proof*⟩

lemma *lattice_plus_left_cancel* [*simp*]: " $x \in \Lambda \implies x + y \in \Lambda \longleftrightarrow y \in$
 Λ "
 ⟨*proof*⟩

lemma *lattice_induct* [*consumes 1, case_names zero gen1 gen2 add uminus*]:
 assumes " $z \in \Lambda$ "
 assumes *zero*: " $P\ 0$ "
 assumes *gens*: " $P\ \omega1$ " " $P\ \omega2$ "
 assumes *plus*: " $\bigwedge w\ z. P\ w \implies P\ z \implies P\ (w + z)$ "
 assumes *uminus*: " $\bigwedge w. P\ w \implies P\ (-w)$ "
 shows " $P\ z$ "
 ⟨*proof*⟩

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

definition *rel* :: "*complex* \Rightarrow *complex* \Rightarrow *bool*" **where**
 "*rel* $x\ y \longleftrightarrow (x - y) \in \Lambda$ "

lemma *rel_refl* [*simp, intro*]: "*rel* $x\ x$ "
 ⟨*proof*⟩

lemma *relE*:
 assumes "*rel* $x\ y$ "
 obtains *z* **where** " $z \in \Lambda$ " " $y = x + z$ "
 ⟨*proof*⟩

lemma *rel_symI*: "*rel* $x\ y \implies \text{rel } y\ x$ "
 ⟨*proof*⟩

lemma *rel_sym*: "*rel* $x\ y \longleftrightarrow \text{rel } y\ x$ "
 ⟨*proof*⟩

lemma *rel_0_right_iff*: "*rel* $x\ 0 \longleftrightarrow x \in \Lambda$ "
 ⟨*proof*⟩

lemma *rel_0_left_iff*: "*rel* $0\ x \longleftrightarrow x \in \Lambda$ "
 ⟨*proof*⟩

lemma *rel_trans* [*trans*]: "*rel* $x\ y \implies \text{rel } y\ z \implies \text{rel } x\ z$ "
 ⟨*proof*⟩

```

lemma rel_minus [lattice_intros]: "rel a b  $\implies$  rel (-a) (-b)"
  <proof>

lemma rel_minus_iff: "rel (-a) (-b)  $\longleftrightarrow$  rel a b"
  <proof>

lemma rel_add [lattice_intros]: "rel a b  $\implies$  rel c d  $\implies$  rel (a + c)
(b + d)"
  <proof>

lemma rel_diff [lattice_intros]: "rel a b  $\implies$  rel c d  $\implies$  rel (a - c)
(b - d)"
  <proof>

lemma rel_mult_of_nat_left [lattice_intros]: "rel a b  $\implies$  rel (of_nat
n * a) (of_nat n * b)"
  <proof>

lemma rel_mult_of_nat_right [lattice_intros]: "rel a b  $\implies$  rel (a *
of_nat n) (b * of_nat n)"
  <proof>

lemma rel_mult_of_int_left [lattice_intros]: "rel a b  $\implies$  rel (of_int
n * a) (of_int n * b)"
  <proof>

lemma rel_mult_of_int_right [lattice_intros]: "rel a b  $\implies$  rel (a *
of_int n) (b * of_int n)"
  <proof>

lemma rel_sum [lattice_intros]:
  "( $\bigwedge$  i. i  $\in$  A  $\implies$  rel (f i) (g i))  $\implies$  rel ( $\sum$  i $\in$ A. f i) ( $\sum$  i $\in$ A. g i)"
  <proof>

lemma rel_sum_list [lattice_intros]:
  "list_all2 rel xs ys  $\implies$  rel (sum_list xs) (sum_list ys)"
  <proof>

lemma rel_lattice_trans_left [trans]: "x  $\in$   $\Lambda$   $\implies$  rel x y  $\implies$  y  $\in$   $\Lambda$ "
  <proof>

lemma rel_lattice_trans_right [trans]: "rel x y  $\implies$  y  $\in$   $\Lambda$   $\implies$  x  $\in$   $\Lambda$ "
  <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  $\omega_2 \ \omega_1$  <proof>

lemma swap_of_ $\omega_{12}$ _coords [simp]: "swap.of_ $\omega_{12}$ _coords = of_ $\omega_{12}$ _coords
  o 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  $(\omega_1, \omega_2)$  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  $\times$  complex  $\Rightarrow$  bool" where
  "fundpair =  $(\lambda(a, b). b / a \notin \mathbb{R})$ "

lemma fundpair_swap: "fundpair ab  $\longleftrightarrow$  fundpair (prod.swap ab)"
  <proof>

lemma fundpair_cnj_iff [simp]: "fundpair (cnj a, cnj b) = fundpair (a, b)"
  <proof>

lemma fundpair_altdef: "fundpair =  $(\lambda(a,b). a / b \notin \mathbb{R})$ "
  <proof>

lemma
  assumes "fundpair (a, b)"
  shows fundpair_imp_nonzero [dest]: "a  $\neq$  0" "b  $\neq$  0"
  and fundpair_imp_neq: "a  $\neq$  b" "b  $\neq$  a"
  <proof>

lemma fundpair_imp_independent:
  assumes "fundpair  $(\omega_1, \omega_2)$ "
  shows "independent  $\{\omega_1, \omega_2\}$ "
  <proof>

lemma fundpair_imp_basis:
  assumes "fundpair  $(\omega_1, \omega_2)$ "
  shows "span  $\{\omega_1, \omega_2\} = UNIV$ "
  <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 ( $\omega_1$ ,  $\omega_2$ )"
begin

definition ratio :: complex ("τ") where "τ =  $\omega_2$  /  $\omega_1$ "

lemma ratio_not_in_Reals: "τ  $\notin$   $\mathbb{R}$ "
  <proof>

lemma  $\omega_1$ _neq_ $\omega_2$  [simp]: " $\omega_1 \neq \omega_2$ " and  $\omega_2$ _neq_ $\omega_1$  [simp]: " $\omega_2 \neq \omega_1$ "
  <proof>

lemma  $\omega_1$ _nonzero [simp]: " $\omega_1 \neq 0$ " and  $\omega_2$ _nonzero [simp]: " $\omega_2 \neq 0$ "
  <proof>

lemma lattice0_nonempty [simp]: "lattice0  $\neq$  {}"
  <proof>

lemma  $\omega_{12}$ _independent': "independent { $\omega_1$ ,  $\omega_2$ }"
  <proof>

lemma span_ $\omega_{12}$ : "span { $\omega_1$ ,  $\omega_2$ } = UNIV"
  <proof>

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

definition  $\omega_1$ _coord :: "complex  $\Rightarrow$  real" where
  " $\omega_1$ _coord z = representation { $\omega_1$ ,  $\omega_2$ } z  $\omega_1$ "

definition  $\omega_2$ _coord :: "complex  $\Rightarrow$  real" where
  " $\omega_2$ _coord z = representation { $\omega_1$ ,  $\omega_2$ } z  $\omega_2$ "

definition  $\omega_{12}$ _coords :: "complex  $\Rightarrow$  real  $\times$  real" where
  " $\omega_{12}$ _coords z = ( $\omega_1$ _coord z,  $\omega_2$ _coord z)"

sublocale  $\omega_1$ _coord: bounded_linear  $\omega_1$ _coord
  <proof>

sublocale  $\omega_2$ _coord: bounded_linear  $\omega_2$ _coord
  <proof>

sublocale  $\omega_{12}$ _coords: linear  $\omega_{12}$ _coords
  <proof>

sublocale  $\omega_{12}$ _coords: bounded_linear  $\omega_{12}$ _coords
  <proof>

```

```

lemmas [continuous_intros] =
   $\omega_1$ _coord.continuous_on  $\omega_1$ _coord.continuous
   $\omega_2$ _coord.continuous_on  $\omega_2$ _coord.continuous
   $\omega_{12}$ _coords.continuous_on  $\omega_{12}$ _coords.continuous

lemmas [tendsto_intros] =  $\omega_1$ _coord.tendsto  $\omega_2$ _coord.tendsto  $\omega_{12}$ _coords.tendsto

lemma  $\omega_1$ _coord_ $\omega_1$  [simp]: " $\omega_1$ _coord  $\omega_1 = 1$ "
  and  $\omega_1$ _coord_ $\omega_2$  [simp]: " $\omega_1$ _coord  $\omega_2 = 0$ "
  and  $\omega_2$ _coord_ $\omega_1$  [simp]: " $\omega_2$ _coord  $\omega_1 = 0$ "
  and  $\omega_2$ _coord_ $\omega_2$  [simp]: " $\omega_2$ _coord  $\omega_2 = 1$ "
  <proof>

lemma  $\omega_{12}$ _coords_ $\omega_1$  [simp]: " $\omega_{12}$ _coords  $\omega_1 = (1, 0)$ "
  and  $\omega_{12}$ _coords_ $\omega_2$  [simp]: " $\omega_{12}$ _coords  $\omega_2 = (0, 1)$ "
  <proof>

lemma  $\omega_{12}$ _coords_of_ $\omega_{12}$ _coords [simp]: " $\omega_{12}$ _coords (of_ $\omega_{12}$ _coords z)
= z"
  <proof>

lemma  $\omega_1$ _coord_of_ $\omega_{12}$ _coords [simp]: " $\omega_1$ _coord (of_ $\omega_{12}$ _coords z) =
fst z"
  and  $\omega_2$ _coord_of_ $\omega_{12}$ _coords [simp]: " $\omega_2$ _coord (of_ $\omega_{12}$ _coords z) = snd
z"
  <proof>

lemma of_ $\omega_{12}$ _coords_ $\omega_{12}$ _coords [simp]: "of_ $\omega_{12}$ _coords ( $\omega_{12}$ _coords z)
= z"
  <proof>

lemma  $\omega_{12}$ _coords_eqI:
  assumes "of_ $\omega_{12}$ _coords a = b"
  shows "  $\omega_{12}$ _coords b = a"
  <proof>

lemmas [simp] =  $\omega_1$ _coord.scaleR  $\omega_2$ _coord.scaleR  $\omega_{12}$ _coords.scaleR

lemma  $\omega_{12}$ _coords_times_ $\omega_1$  [simp]: " $\omega_{12}$ _coords (of_real a *  $\omega_1$ ) = (a,
0)"
  and  $\omega_{12}$ _coords_times_ $\omega_2$  [simp]: " $\omega_{12}$ _coords (of_real a *  $\omega_2$ ) = (0,
a)"
  and  $\omega_{12}$ _coords_times_ $\omega_1'$  [simp]: " $\omega_{12}$ _coords ( $\omega_1$  * of_real a) = (a,
0)"
  and  $\omega_{12}$ _coords_times_ $\omega_2'$  [simp]: " $\omega_{12}$ _coords ( $\omega_2$  * of_real a) = (0,
a)"
  and  $\omega_{12}$ _coords_mult_of_real [simp]: " $\omega_{12}$ _coords (of_real c * z) = c
*_R  $\omega_{12}$ _coords z"

```

```

    and  $\omega_{12\_coords\_mult\_of\_int}$  [simp]: " $\omega_{12\_coords} (of\_int\ i * z) = of\_int\ i *_R\ \omega_{12\_coords}\ z$ "
    and  $\omega_{12\_coords\_mult\_of\_nat}$  [simp]: " $\omega_{12\_coords} (of\_nat\ n * z) = of\_nat\ n *_R\ \omega_{12\_coords}\ z$ "
    and  $\omega_{12\_coords\_divide\_of\_real}$  [simp]: " $\omega_{12\_coords} (z / of\_real\ c) = \omega_{12\_coords}\ z /_R\ c$ "
    and  $\omega_{12\_coords\_mult\_numeral}$  [simp]: " $\omega_{12\_coords} (numeral\ num * z) = numeral\ num *_R\ \omega_{12\_coords}\ z$ "
    and  $\omega_{12\_coords\_divide\_numeral}$  [simp]: " $\omega_{12\_coords} (z / numeral\ num) = \omega_{12\_coords}\ z /_R\ numeral\ num$ "
    <proof>

```

```

lemma  $of\_omega_{12\_coords\_eq\_iff}$ : " $of\_omega_{12\_coords}\ z1 = of\_omega_{12\_coords}\ z2 \iff z1 = z2$ "
  <proof>

```

```

lemma  $omega_{12\_coords\_eq\_iff}$ : " $omega_{12\_coords}\ z1 = omega_{12\_coords}\ z2 \iff z1 = z2$ "
  <proof>

```

```

lemma  $of\_omega_{12\_coords\_eq\_0\_iff}$  [simp]: " $of\_omega_{12\_coords}\ z = 0 \iff z = (0,0)$ "
  <proof>

```

```

lemma  $omega_{12\_coords\_eq\_0\_0\_iff}$  [simp]: " $omega_{12\_coords}\ x = (0, 0) \iff x = 0$ "
  <proof>

```

```

lemma  $bij\_of\_omega_{12\_coords}$ : " $bij\ of\_omega_{12\_coords}$ "
  <proof>

```

```

lemma  $bij\_betw\_lattice$ : " $bij\_betw\ of\_omega_{12\_coords}\ (\mathbb{Z} \times \mathbb{Z})\ lattice$ "
  <proof>

```

```

lemma  $bij\_betw\_lattice0$ : " $bij\_betw\ of\_omega_{12\_coords}\ (\mathbb{Z} \times \mathbb{Z} - \{(0,0)\})\ lattice0$ "
  <proof>

```

```

lemma  $bij\_betw\_lattice'$ : " $bij\_betw\ (of\_omega_{12\_coords} \circ map\_prod\ of\_int\ of\_int)\ UNIV\ lattice$ "
  <proof>

```

```

lemma  $bij\_betw\_lattice0'$ : " $bij\_betw\ (of\_omega_{12\_coords} \circ map\_prod\ of\_int\ of\_int)\ (-\{(0,0)\})\ lattice0$ "
  <proof>

```

```

lemma  $infinite\_lattice$ : " $\neg finite\ lattice$ "
  <proof>

```

```

lemma  $omega_{12\_coords\_image\_eq}$ : " $omega_{12\_coords}\ ` X = of\_omega_{12\_coords}\ - ` X$ "
  <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 , $z + \omega_1$, $z + \omega_2$, and $z + \omega_1 + \omega_2$. 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  $\Rightarrow$  complex set" where
  "period_parallelogram z = (+) z ' of_ω12_coords ' ({0.. $<1$ }  $\times$  {0.. $<1$ })"
```

The following is a path along the border of a period parallelogram, starting at the vertex z and going in direction ω_1 .

```
definition period_parallelogram_path :: "complex  $\Rightarrow$  real  $\Rightarrow$  complex" where
  "period_parallelogram_path z  $\equiv$  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 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  $\omega_1$   $\omega_2$ )  $\cap$  interior (period_parallelogram  
z) = {}"  
  <proof>
```

```
lemma winding_number_parallelogram_outside:  
  assumes "w  $\notin$  closure (period_parallelogram z)"  
  shows "winding_number (parallelogram_path z  $\omega_1$   $\omega_2$ ) 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  $\in$  interior (period_parallelogram z)"  
  shows "winding_number (parallelogram_path z  $\omega_1$   $\omega_2$ ) w = sgn (Im ( $\omega_2$   
/  $\omega_1$ ))"  
  <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  $\Rightarrow$  complex" where  
  "to_fund_parallelogram z =  
    (case  $\omega_{12}$ _coords z of (a, b)  $\Rightarrow$  of_ $\omega_{12}$ _coords (frac a, frac b))"
```

```
lemma to_fund_parallelogram_in_parallelogram [intro]:  
  "to_fund_parallelogram z  $\in$  period_parallelogram 0"  
  <proof>
```

```
lemma  $\omega_1$ _coord_to_fund_parallelogram [simp]: " $\omega_1$ _coord (to_fund_parallelogram  
z) = frac ( $\omega_1$ _coord z)"  
  and  $\omega_2$ _coord_to_fund_parallelogram [simp]: " $\omega_2$ _coord (to_fund_parallelogram  
z) = frac ( $\omega_2$ _coord z)"  
  <proof>
```

```
lemma to_fund_parallelogramE:  
  obtains m n where "to_fund_parallelogram z = z + of_int m *  $\omega_1$  + of_int  
n *  $\omega_2$ "  
  <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 \longleftrightarrow rel z w"
⟨proof⟩

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

lemma *to_fund_parallelogram_in_lattice_iff [simp]*:
"to_fund_parallelogram z \in lattice \longleftrightarrow z \in lattice"
⟨proof⟩

lemma *to_fund_parallelogram_in_lattice [lattice_intros]*:
"z \in lattice \implies to_fund_parallelogram z \in 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)"
" $\bigwedge z. \text{rel } (f z) z$ "
⟨proof⟩

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

lemma *to_fund_parallelogram_lattice [simp]*: "z \in $\Lambda \implies$ to_fund_parallelogram
z = 0"
⟨proof⟩

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

```

lemma to_fund_parallelogram_eq_0_iff [simp]: "to_fund_parallelogram u
= 0  $\longleftrightarrow$  u  $\in$   $\Lambda$ "
  <proof>

```

```

lemma to_fund_parallelogram_of_fund_parallelogram:
  "z  $\in$  period_parallelogram 0  $\implies$  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'  $\in$  period_parallelogram 0"
  shows "to_fund_parallelogram z = z'"
  <proof>

```

```

lemma to_fund_parallelogram_unique':
  assumes "rel z z'" "z  $\in$  period_parallelogram 0" "z'  $\in$  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_omega12_coords ' {(x,y). x  $\in$  {0..1/2}  $\wedge$  y  $\in$  {0..<1}  $\wedge$  (x  $\in$  {0, 1/2}
 $\longrightarrow$  y  $\leq$  1/2)}"

```

```

lemma half_fund_parallelogram_altdef:
  "half_fund_parallelogram = omega12_coords - ' {(x,y). x  $\in$  {0..1/2}  $\wedge$  y  $\in$ 
{0..<1}  $\wedge$  (x  $\in$  {0, 1/2}  $\longrightarrow$  y  $\leq$  1/2)}"
  <proof>

```

```

lemma zero_in_half_fund_parallelogram [simp, intro]: "0  $\in$  half_fund_parallelogram"
  <proof>

```

```

lemma half_fund_parallelogram_in_lattice_iff:
  assumes "z  $\in$  half_fund_parallelogram"
  shows "z  $\in$   $\Lambda$   $\longleftrightarrow$  z = 0"
  <proof>

```

```

definition to_half_fund_parallelogram :: "complex  $\Rightarrow$  complex" where
  "to_half_fund_parallelogram z =
  (let (x,y) = map_prod frac frac (omega12_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 $\omega_1', \omega_2' \in \Lambda$ then these two points generate Λ if and only if the triangle spanned by $0, \omega_1'$, and ω_2' contains no other lattice points except $0, \omega_1'$, and ω_2' .

```
context complex_lattice
```

begin

lemma *in_triangle_iff*:

fixes *x*

defines "*a* \equiv $\omega_1_coord\ x$ " and "*b* \equiv $\omega_2_coord\ x$ "

shows " $x \in convex\ hull\ \{0, \omega_1, \omega_2\} \longleftrightarrow a \geq 0 \wedge b \geq 0 \wedge a + b \leq 1$ "
<proof>

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

lemma *lattice_Int_triangle*: " $convex\ hull\ \{0, \omega_1, \omega_2\} \cap \Lambda = \{0, \omega_1, \omega_2\}$ "

<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 $0, \omega_1, \omega_2$ contains no lattice points other than its three vertices.

lemma *equiv_fundpair_iff_triangle*:

assumes "*fundpair* (ω_1', ω_2')" " $\omega_1' \in \Lambda$ " " $\omega_2' \in \Lambda$ "

shows "*equiv_fundpair* (ω_1, ω_2) (ω_1', ω_2') $\longleftrightarrow convex\ hull\ \{0, \omega_1', \omega_2'\} \cap \Lambda = \{0, \omega_1', \omega_2'\}$ "
<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* :: " $nat \Rightarrow complex\ set$ " **where**

"*lattice_layer* *k* =
of_ ω_{12} _coords ' map_prod of_int of_int '
($\{int\ k, -int\ k\} \times \{-int\ k..int\ k\} \cup \{-int\ k..int\ k\} \times \{-int\ k, int\ k\}$)"

lemma *in_lattice_layer_iff*:

" $z \in lattice_layer\ k \longleftrightarrow$

$\omega_{12}_coords\ z \in \mathbb{Z} \times \mathbb{Z} \cap (\{int\ k, -int\ k\} \times \{-int\ k..int\ k\} \cup \{-int\ k..int\ k\} \times \{-int\ k, int\ k\})$ "
(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 \subseteq (*) (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 \equiv Inf (norm ' lattice_layer_path)"

lemma *Inf_para_pos*: "Inf_para > 0"
<proof>

lemma *Inf_para_nonzero [simp]*: "Inf_para \neq 0"
<proof>

lemma *Inf_para_le*:
assumes "z \in lattice_layer_path"
shows "Inf_para \leq norm z"
<proof>

lemma *lattice_layer_le_norm*:
assumes " $\omega \in$ lattice_layer k"
shows "k * Inf_para \leq norm ω "
<proof>

corollary *Inf_para_le_norm*:
assumes " $\omega \in \Lambda^*$ "
shows "Inf_para \leq 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 \in Λ " "y \in Λ " "x \neq y"
shows "dist x y \geq Inf_para"
<proof>

definition *Sup_para* :: real where — *R* in the proof of Lemma 1
"Sup_para \equiv Sup (norm ' lattice_layer_path)"

lemma *Sup_para_ge*:
assumes "z \in lattice_layer_path"
shows "Sup_para \geq norm z"
<proof>

lemma *Sup_para_pos*: "Sup_para > 0"

<proof>

lemma *Sup_para_nonzero [simp]: "Sup_para $\neq 0$ "*
<proof>

lemma *lattice_layer_ge_norm:*
 assumes *" $\omega \in \text{lattice_layer } k$ "*
 shows *"norm $\omega \leq k * \text{Sup_para}$ "*
<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: " $\neg \text{islimpt } \Lambda$ "*
<proof>

lemma *closed_lattice: "closed lattice"*
<proof>

lemma *lattice_sparse: " Λ sparse_in UNIV"*
<proof>

lemma *eventually_not_in_lattice_cosparse:*
 "eventually ($\lambda z. z \notin \Lambda$) (cosparse UNIV)"
<proof>

lemma *eventually_not_rel_cosparse:*
 "eventually ($\lambda z. \neg \text{rel } z \ w$) (cosparse 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 *" $X \subseteq \Lambda$ " " $X \neq \{\}$ "*
 obtains *x where " $x \in X$ " " $\bigwedge y. y \in X \implies \text{norm } x \leq \text{norm } y$ "*
<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 *" $x \in \Lambda^*$ "*
 obtains *y where " $y \in \Lambda^*$ " " $y / x \notin \mathbb{R}$ "*
<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  $\omega_1$   $\omega_2$ )  $\cap$  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 " $\wedge z. \neg z$  islimpt avoid"
  assumes " $\wedge$  orig d. finite (closure (period_parallelogram orig)  $\cap$  avoid)"
 $\implies$ 
  finite (closure (period_parallelogram (orig + d))
 $\cap$  avoid)  $\implies$ 
  P orig  $\implies$  P (orig + d)"
  assumes " $\wedge$  orig. finite (closure (period_parallelogram orig)  $\cap$  avoid)"
 $\implies$ 
  path_image (parallelogram_path orig  $\omega_1$   $\omega_2$ )  $\cap$  avoid
= {}  $\implies$ 
  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  $\tau$  :: complex (structure)
  assumes Im_ $\tau$ _pos: "Im  $\tau$  > 0"
begin

```

```

sublocale complex_lattice 1  $\tau$ 
<proof>

```

```

lemma winding_number_parallelogram_inside':
  assumes "w  $\in$  interior (period_parallelogram z)"
  shows "winding_number (parallelogram_path z 1  $\tau$ ) 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 $\lambda z. f z = 0$ 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  $\Rightarrow$  'a"
  assumes f_periodic: "f (z +  $\omega$ 1) = f z" "f (z +  $\omega$ 2) = f z"
begin

lemma lattice_cong:
  assumes "rel x y"
  shows "f x = f y"
  <proof>

end

locale complex_lattice_periodic =
  complex_lattice  $\omega$ 1  $\omega$ 2 + pre_complex_lattice_periodic  $\omega$ 1  $\omega$ 2 f
  for  $\omega$ 1  $\omega$ 2 :: complex and f :: "complex  $\Rightarrow$  'a"
begin

lemma eval_to_fund_parallelogram: "f (to_fund_parallelogram z) = f z"
  <proof>

end

locale complex_lattice_periodic_compose =
  complex_lattice_periodic  $\omega$ 1  $\omega$ 2 f for  $\omega$ 1  $\omega$ 2 :: complex and f :: "complex
 $\Rightarrow$  'a" +
  fixes h :: "'a  $\Rightarrow$  'b"
begin

sublocale compose: complex_lattice_periodic  $\omega$ 1  $\omega$ 2 " $\lambda$ z. h (f z)"
  <proof>

end

end
```

4 Subgroups of the modular group

```
theory Modular_Subgroups
  imports Modular_Group
```

begin

4.1 Definition and group action on the upper half plane

```
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  $\implies$  f  $\in$  G  $\implies$  Im x > 0  $\implies$  rel x (apply_modgrp
f y)"
  <proof>

lemma relI2 [intro]: "rel x y  $\implies$  f  $\in$  G  $\implies$  Im x > 0  $\implies$  rel (apply_modgrp
f x) y"
  <proof>

lemma relE:
  assumes "rel x y"
  obtains h where "Im x > 0" "Im y > 0" "h  $\in$  G" "y = apply_modgrp h x"
  <proof>

lemma relE':
  assumes "rel x y"
  obtains h where "Im x > 0" "Im y > 0" "h  $\in$  G" "x = apply_modgrp h y"
  <proof>

lemma rel_apply_modgrp_left_iff [simp]:
  assumes "f  $\in$  G"
  shows   "rel (apply_modgrp f x) y  $\longleftrightarrow$  Im x > 0  $\wedge$  rel x y"
  <proof>

lemma rel_apply_modgrp_right_iff [simp]:
  assumes "f  $\in$  G"
  shows   "rel y (apply_modgrp f x)  $\longleftrightarrow$  Im x > 0  $\wedge$  rel y x"
  <proof>

lemma orbit_refl_iff: "x  $\in$  orbit x  $\longleftrightarrow$  Im x > 0"
  <proof>

lemma orbit_refl: "Im x > 0  $\implies$  x  $\in$  orbit x"
  <proof>

lemma orbit_cong: "rel x y  $\implies$  orbit x = orbit y"
  <proof>

```

lemma orbit_empty_iff [simp]: "orbit x = {} \longleftrightarrow Im x \leq 0" "{} = orbit x \longleftrightarrow Im x \leq 0"
 <proof>

lemmas [simp] = orbit_refl_iff

lemma orbit_eq_iff: "orbit x = orbit y \longleftrightarrow Im x \leq 0 \wedge Im y \leq 0 \vee rel x y"
 <proof>

lemma orbit_apply_modgrp [simp]: "f \in G \implies orbit (apply_modgrp f z) = orbit z"
 <proof>

lemma apply_modgrp_in_orbit_iff [simp]: "f \in G \implies apply_modgrp f z \in orbit y \longleftrightarrow z \in orbit y"
 <proof>

lemma orbit_imp_Im_pos: "x \in orbit y \implies Im x $>$ 0"
 <proof>

end

interpretation modular_group: modgrp_subgroup UNIV
 <proof>

notation modular_group.rel (infixl " \sim_{Γ} " 49)

lemma (in modgrp_subgroup) rel_imp_rel: "rel x y \implies x \sim_{Γ} y"
 <proof>

lemma modular_group_rel_plus_int_iff_right1 [simp]:
 assumes "z \in \mathbb{Z} "
 shows "x \sim_{Γ} y + z \longleftrightarrow x \sim_{Γ} y"
 <proof>

lemma
 assumes "z \in \mathbb{Z} "
 shows modular_group_rel_plus_int_iff_right2 [simp]: "x \sim_{Γ} z + y \longleftrightarrow x \sim_{Γ} y"
 and modular_group_rel_plus_int_iff_left1 [simp]: "z + x \sim_{Γ} y \longleftrightarrow x \sim_{Γ} y"
 and modular_group_rel_plus_int_iff_left2 [simp]: "x + z \sim_{Γ} y \longleftrightarrow x \sim_{Γ} y"
 <proof>

lemma modular_group_rel_S_iff_right [simp]: "x \sim_{Γ} -(1/y) \longleftrightarrow x \sim_{Γ} y"
 <proof>

lemma modular_group_rel_S_iff_left [simp]: $-(1/x) \sim_{\Gamma} y \longleftrightarrow x \sim_{\Gamma} y$
 ⟨proof⟩

The index of a subgroup is the number of cosets.

definition index_modgrp :: "modgrp set \Rightarrow nat" where
 $"index_modgrp\ G = card\ (range\ (\lambda x. (*)\ x\ 'G))"$

The following defines the group generated by a set of elements.

inductive_set generate_modgrp :: "modgrp set \Rightarrow modgrp set" for X :: "modgrp set" where
 $"x \in X \implies x \in generate_modgrp\ X"$
 $| "1 \in generate_modgrp\ X"$
 $| "x \in generate_modgrp\ X \implies y \in generate_modgrp\ X \implies x * y \in generate_modgrp\ X"$
 $| "x \in generate_modgrp\ X \implies inverse\ x \in generate_modgrp\ X"$

lemma modgrp_subgroup_generate: $"modgrp_subgroup\ (generate_modgrp\ X)"$
 ⟨proof⟩

lemma (in modgrp_subgroup) generate_modgrp_subsetI:
 assumes $"X \subseteq G"$
 shows $"generate_modgrp\ X \subseteq G"$
 ⟨proof⟩

4.2 Conjugation

The conjugation of a subgroup G w.r.t. some $h \in \Gamma$ is $h^{-1}Gh$.

definition conj_modgrp :: "modgrp \Rightarrow modgrp set \Rightarrow modgrp set" where
 $"conj_modgrp\ x\ G = (\lambda y. inverse\ x * y * x)\ 'G"$

lemma conj_modgrp_mono: $"G \subseteq H \implies conj_modgrp\ x\ G \subseteq conj_modgrp\ x\ H"$
 ⟨proof⟩

lemma conj_modgrp_altdef: $"conj_modgrp\ x\ G = (\lambda y. x * y * inverse\ x)\ -'G"$
 ⟨proof⟩

lemma conj_modgrp_UNIV [simp]: $"conj_modgrp\ x\ UNIV = UNIV"$
 ⟨proof⟩

lemma in_conj_modgrp_iff: $"z \in conj_modgrp\ x\ G \longleftrightarrow x * z * inverse\ x \in G"$
 ⟨proof⟩

lemma conj_modgrp_mult: $"conj_modgrp\ (g * f)\ G = conj_modgrp\ f\ (conj_modgrp\ g\ G)"$

```

    <proof>

lemma conj_modgrp_1 [simp]: "conj_modgrp 1 G = G"
  <proof>

context modgrp_subgroup
begin

lemma conj_modgrp_id:
  assumes "x ∈ G"
  shows   "conj_modgrp x G = G"
  <proof>

lemma modgrp_subgroup_conj: "modgrp_subgroup (conj_modgrp x G)"
  <proof>

end

```

4.3 Elliptic points

The elliptic order of a point in the complex plane is the size of its stabiliser group (modulo $\pm I$).

```

definition ellorder_modgrp :: "modgrp set ⇒ complex ⇒ nat" where
  "ellorder_modgrp G z =
    (if Im z > 0 then card {h∈G. apply_modgrp h z = z} div card (G ∩
{1, -1}) else 0)"

```

```

lemma (in modgrp_subgroup) ellorder_modgrp_cong:
  assumes "rel w z"
  shows   "ellorder_modgrp G w = ellorder_modgrp G z"
  <proof>

```

We define the number of elliptic points of a given order, and the number of cusps (sometimes seen as elliptic points of order ∞).

```

definition ellcount_modgrp :: "modgrp set ⇒ nat ⇒ nat" where
  "ellcount_modgrp G k =
    card ({z. Im z > 0 ∧ ellorder_modgrp G z = k} // {(w,z). modgrp_subgroup.rel
G w z})"

```

```

definition cusp_count_modgrp :: "modgrp set ⇒ nat" where
  "cusp_count_modgrp G = 1 + card ((-pole_modgrp ' G) // {(w::rat,z).
∃h. w = apply_modgrp h z})"

```

4.4 Subgroups containing shifts

We will now look at subgroups that contain some shift operator T^n for $n > 0$. The cusp width at infinity is the smallest such n (or equivalently the GCD of all such n).

The cusp width at the other cusps (i.e. a rational numbers) is defined in the same way after conjugation with a modular transformation that sends the rational number to infinity.

definition `cuspid_width_at_ii_inf` :: "modgrp set \Rightarrow nat" ("cuspid_width $_{\infty}$ ")
where

"cuspid_width_at_ii_inf G = nat (Gcd {n. shift_modgrp n \in G})"

definition `cuspid_width_modgrp` :: "modgrp \Rightarrow modgrp set \Rightarrow nat" **where**
"cuspid_width_modgrp h G = cuspid_width $_{\infty}$ (conj_modgrp h G)"

lemma `of_nat_cuspid_width_at_ii_inf`:

"of_nat (cuspid_width_at_ii_inf G) = Gcd {n. shift_modgrp n \in G}"
 \langle proof \rangle

lemma `cuspid_width_at_ii_inf_UNIV` [simp]: "cuspid_width_at_ii_inf UNIV = Suc 0"
 \langle proof \rangle

lemma (in `modgrp_subgroup`) `shift_modgrp_in_G_iff`:

"shift_modgrp n \in G \longleftrightarrow int (cuspid_width_at_ii_inf G) dvd n"
 \langle proof \rangle

locale `modgrp_subgroup_periodic` = `modgrp_subgroup` +
assumes `periodic`: " $\exists n > 0$. shift_modgrp n \in G"
begin

lemma `cuspid_width_at_ii_inf_pos`: "cuspid_width_at_ii_inf G > 0"
 \langle proof \rangle

lemma `shift_modgrp_in_G_period` [intro, simp]:

"shift_modgrp (int (cuspid_width_at_ii_inf G)) \in G"
 \langle proof \rangle

lemma `shift_modgrp_in_G` [intro]:

"int (cuspid_width_at_ii_inf G) dvd n \implies shift_modgrp n \in G"
 \langle proof \rangle

end

4.4.1 Congruence subgroups

The principal congruence subgroup $\Gamma(N)$ consists of those modular transformations A for which $A = I \pmod{N}$ (i.e. they look like the identity modulo N).

lift_definition `modgrps_pcong` :: "int \Rightarrow modgrp set" (" \langle notation= \langle mixfix modgrps_pcong \rangle \rangle Γ' (_')") is

" λN . {(a,b,c,d) :: (int \times int \times int \times int) | a b c d.

$a * d - b * c = 1 \wedge [a = 1] \pmod{N} \wedge [d = 1] \pmod{N} \wedge N \text{ dvd } b \wedge N \text{ dvd } c$ "
 <proof>

lemma modgrps_pcong_altdef:
 "modgrps_pcong N = {f. [f = 1] (mod_Γ N)}"
 <proof>

lemma modgrps_pcong_abs [simp]: "modgrps_pcong (abs n) = modgrps_pcong n"
 <proof>

lemma modgrp_in_modgrps_pcong_iff:
 assumes "a * d - b * c = 1"
 shows "modgrp a b c d ∈ modgrps_pcong N \longleftrightarrow [a = 1] (mod N) \wedge [d = 1] (mod N) \wedge N dvd b \wedge N dvd c"
 <proof>

lemma modgrp_in_modgrps_pcong:
 assumes "[a = 1] (mod N)" "[d = 1] (mod N)" "N dvd b" "N dvd c" "a * d - b * c = 1"
 shows "modgrp a b c d ∈ modgrps_pcong N"
 <proof>

lemma modgrp_pcong_0 [simp]: "modgrps_pcong 0 = {1}"
 <proof>

lemma modgrp_pcong_1 [simp]: "modgrps_pcong 1 = UNIV"
 <proof>

lemma modgrps_pcong_mono: "n dvd m \implies modgrps_pcong m \subseteq modgrps_pcong n"
 <proof>

lemma modgrps_pcong_subset_iff: "modgrps_pcong m \subseteq modgrps_pcong n \longleftrightarrow n dvd m"
 <proof>

lemma shift_in_modgrps_pcong_iff: "shift_modgrp n ∈ modgrps_pcong d \longleftrightarrow d dvd n"
 <proof>

interpretation modgrps_pcong: modgrp_subgroup "modgrps_pcong N"
 <proof>

lemma infinite_modgrp_pcong:
 assumes "n \neq 0"
 shows "infinite (modgrps_pcong n)"
 <proof>

The level of a subgroup is the smallest n such that it contains $\Gamma(n)$. Equivalently (and more elegantly), it is the GCD of all those numbers n .

definition *level_modgrp* where "level_modgrp $G = \text{Gcd } \{n. \text{modgrps_pcong } n \subseteq G\}$ "

lemma *level_modgrp_nonneg*: "level_modgrp $G \geq 0$ "
 ⟨proof⟩

lemma *level_modgrp_UNIV [simp]*: "level_modgrp $UNIV = 1$ "
 ⟨proof⟩

$\Gamma(n)$ is normal.

lemma *conj_modgrps_pcong*: "conj_modgrp $x (\text{modgrps_pcong } n) = \text{modgrps_pcong } n$ "
 ⟨proof⟩

The level of $\Gamma(N)$ is, unsurprisingly, N .

lemma *level_conj_modgrp [simp]*: "level_modgrp (conj_modgrp $x G) = \text{level_modgrp } G$ "
 ⟨proof⟩

lemma *cong_lcm_iff*:
 "[$a = (b :: 'a :: \{\text{unique_euclidean_ring, ring_gcd}\}) \pmod{\text{lcm } m n}$]
 \longleftrightarrow
 [$a = b \pmod{m} \wedge a = b \pmod{n}$]"
 ⟨proof⟩

Next we investigate $\Gamma(\text{lcm}(n, m))$ and $\Gamma(\text{gcd}(n, m))$. The former is very easy.

lemma *modgrps_pcong_lcm*: "modgrps_pcong (lcm $n m) = \text{modgrps_pcong } n \cap \text{modgrps_pcong } m$ "
 ⟨proof⟩

The case for the GCD is slightly more difficult and requires the Chinese Remainder Theorem and the surjectivity of the reduction map.

lemma *modgrps_pcong_gcd_aux*:
 assumes " $h \in \text{modgrps_pcong } (\text{gcd } m n)$ "
 obtains $f g$ where " $f \in \text{modgrps_pcong } m$ " " $g \in \text{modgrps_pcong } n$ " " $h = f * g$ "
 ⟨proof⟩

lemma (in *modgrp_subgroup*) *modgrps_pcong_gcd*:
 fixes $m n :: \text{int}$
 defines " $H \equiv \text{generate_modgrp } (\text{modgrps_pcong } m \cup \text{modgrps_pcong } n)$ "
 shows " $\text{modgrps_pcong } (\text{gcd } m n) = H$ "
 ⟨proof⟩

Finally we prove a key lemma: $\Gamma(N)$ is contained in a subgroup iff N is a multiple of the level.

lemma (in *modgrp_subgroup*) *contains_modgrps_pcong_iff*:
 "modgrps_pcong $n \subseteq G \iff \text{level_modgrp } G \text{ dvd } n$ "
 <proof>

It is also easy to see that the level is a multiple of all cusp widths. It is in fact even exactly the LCM of the cusp widths (as shown by Wohlfahrt), but we do not show this here.

lemma (in *modgrp_subgroup*) *cusp_width_at_ii_inf_dvd_level*:
 "cusp_width_at_ii_inf $G \text{ dvd level_modgrp } G$ "
 <proof>

lemma *level_modgrps_pcong [simp]*: "level_modgrp (modgrps_pcong n) = abs n "
 <proof>

A *congruence subgroup* is a subgroup of the modular group that contains $\Gamma(N)$ for some $N > 0$.

locale *cong_subgroup* = *modgrp_subgroup* +
 assumes *level_pos*: "level_modgrp $G > 0$ "
begin

definition *cusp_width* :: "rat \Rightarrow nat" where
 "cusp_width $x = (\text{let } f = \text{modgrp_of_rat } x \text{ in } \text{Gcd } \{n. \text{inverse } f * \text{shift_modgrp } (\text{int } n) * f \in G\})$ "

lemma *cong_subgroup_conj*: "cong_subgroup (conj_modgrp $x G$)"
 <proof>

sublocale *modgrp_subgroup_periodic* G
 <proof>

lemma *infinite*: "infinite G "
 <proof>

end

lemma *cong_subgroup_pcong*: " $N > 0 \implies \text{cong_subgroup } (\text{modgrps_pcong } N)$ "
 <proof>

interpretation *modular_group*: *cong_subgroup UNIV*
 rewrites "cusp_width_at_ii_inf $UNIV = \text{Suc } 0$ "
 <proof>

lemma *cong_subgroup_UNIV [intro]*: "cong_subgroup $UNIV$ " <proof>

4.4.2 Hecke subgroups $\Gamma_0(N)$

The Hecke subgroup of level N , $\Gamma_0(N)$, is a superset of $\Gamma(N)$. It only requires that $c \equiv 0 \pmod{N}$, i.e. it consists of the matrices that are lower triangular

matrices modulo N .

All cusps have width 1 in $\Gamma_0(N)$.

```
lift_definition modgrps_hecke :: "int  $\Rightarrow$  modgrp set" (" $\langle$ notation= $\langle$ mixfix
modgrps_hecke $\rangle$  $\rangle$  $\Gamma_0'$ ( $\_'$ )") is
  " $\lambda N. \{(a,b,c,d) :: (\text{int} \times \text{int} \times \text{int} \times \text{int}) \mid a \ b \ c \ d. a * d - b *
c = 1 \wedge N \ \text{dvd} \ c\}$ "
   $\langle$ proof $\rangle$ 
```

```
lemma modgrps_hecke_altdef: "modgrps_hecke q = {f. q dvd modgrp_c f}"
   $\langle$ proof $\rangle$ 
```

```
lemma modgrp_in_modgrps_hecke_iff:
  assumes "a * d - b * c = 1"
  shows "modgrp a b c d  $\in$  modgrps_hecke q  $\longleftrightarrow$  q dvd c"
   $\langle$ proof $\rangle$ 
```

```
lemma modgrp_in_modgrps_hecke:
  assumes "q dvd c" "a * d - b * c = 1"
  shows "modgrp a b c d  $\in$  modgrps_hecke q"
   $\langle$ proof $\rangle$ 
```

```
lemma shift_in_modgrps_hecke [simp]: "shift_modgrp n  $\in$  modgrps_hecke
q"
   $\langle$ proof $\rangle$ 
```

```
lemma cusp_width_at_ii_inf_hecke [simp]: "cusp_width $_{\infty}$  (modgrps_hecke
q) = 1"
   $\langle$ proof $\rangle$ 
```

```
lemma S_in_modgrps_hecke_iff [simp]: "S_modgrp  $\in$  modgrps_hecke q  $\longleftrightarrow$ 
is_unit q"
   $\langle$ proof $\rangle$ 
```

```
lemma level_modgrps_hecke [simp]: "level_modgrp (modgrps_hecke N) = abs
N"
   $\langle$ proof $\rangle$ 
```

```
locale hecke_subgroup =
  fixes q :: int
  assumes q_pos: "q > 0"
begin
```

```
definition subgrp (" $\Gamma'$ ") where "subgrp = modgrps_hecke q"
```

```
lemma S_in_subgrp_iff [simp]: "S_modgrp  $\in$  subgrp  $\longleftrightarrow$  q = 1"
   $\langle$ proof $\rangle$ 
```

```
sublocale modgrp_subgroup  $\Gamma'$ 
```

<proof>

sublocale *cong_subgroup* Γ'

<proof>

end

Next, we focus on the Hecke subgroups of prime d .

locale *hecke_prime_subgroup* =

fixes $p :: \text{int}$

assumes p_prime : "prime p "

begin

lemma p_pos : " $p > 0$ "

<proof>

lemma p_not_1 [*simp*]: " $p \neq 1$ "

<proof>

sublocale *hecke_subgroup* p

<proof>

notation *subgrp* (" Γ'' ")

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

Every transformation $f \in \Gamma$ that is *not* in the subgroup can be written as a product $f = gST^k$, where g is in the subgroup.

lemma *modgrp_decompose*:

assumes " $f \notin \Gamma'$ "

obtains $g\ k$ **where** " $g \in \Gamma''$ " " $k \in \{0..<p\}$ " " $f = g * S_modgrp * shift_modgrp\ k$ "

<proof>

lemma *modgrp_decompose'*:

obtains $g\ h$

where " $g \in \Gamma''$ " " $h = 1 \vee (\exists k \in \{0..<p\}. h = S_shift_modgrp\ k)$ " " $f = g * h$ "

<proof>

end

4.5 The subgroups $\Gamma_1(N)$

The following subgroups lie inbetween $\Gamma(N)$ and $\Gamma_0(N)$. They consist of those matrices that become upper triangular matrices with 1 on the diagonal when reduced modulo N .

These groups do not seem to have a name in the literature. We call them the “unipotent subgroups modulo N ” (since upper triangular matrices with 1 on the diagonal are exactly the unipotent matrices).

Again, the level of $\Gamma(N)$ is N and the cusp widths are all 1.

```
lift_definition modgrps_unip :: "int  $\Rightarrow$  modgrp set" ("( $\langle$ notation= $\langle$ mixfix
modgrps_unip $\rangle$  $\rangle$  $\Gamma_1'$ ( $\_'$ ))") is
  " $\lambda N. \{(a,b,c,d) :: (int \times int \times int \times int) \mid a \ b \ c \ d. a * d - b *
c = 1 \wedge$ 
       $[a = 1] \pmod{N} \wedge N \text{ dvd } c \wedge [d = 1] \pmod{N}\}$ "
   $\langle$ proof $\rangle$ 
```

```
lemma modgrps_unip_altdef:
  "modgrps_unip N = {f. [modgrp_a f = 1] (mod N)  $\wedge$  [modgrp_d f = 1] (mod
N)  $\wedge$  N dvd modgrp_c f}"
   $\langle$ proof $\rangle$ 
```

```
lemma modgrps_unip_subset_hecke: "modgrps_unip N  $\subseteq$  modgrps_hecke N"
   $\langle$ proof $\rangle$ 
```

```
lemma modgrp_in_modgrps_unip_iff:
  assumes "a * d - b * c = 1"
  shows "modgrp a b c d  $\in$  modgrps_unip N  $\longleftrightarrow$  [a = 1] (mod N)  $\wedge$  [d
= 1] (mod N)  $\wedge$  N dvd c"
   $\langle$ proof $\rangle$ 
```

```
lemma shift_in_modgrps_unip [simp]: "shift_modgrp n  $\in$  modgrps_unip N"
   $\langle$ proof $\rangle$ 
```

```
lemma cusp_width_at_ii_inf_unip [simp]: "cusp_width $_{\infty}$  (modgrps_unip N)
= 1"
   $\langle$ proof $\rangle$ 
```

```
lemma S_in_modgrps_unip_iff [simp]: "S_modgrp  $\in$  modgrps_unip N  $\longleftrightarrow$  is_unit
N"
   $\langle$ proof $\rangle$ 
```

```
lemma level_modgrps_unip [simp]: "level_modgrp (modgrps_unip N) = abs
N"
   $\langle$ proof $\rangle$ 
```

```
locale unip_subgroup =
  fixes N :: int
  assumes N_pos: "N > 0"
begin
```

```
definition subgrp (" $\Gamma'$ ") where "subgrp = modgrps_unip N"
```

```

lemma S_in_subgrp_iff [simp]: "S_modgrp ∈ subgrp ↔ N = 1"
  ⟨proof⟩

sublocale modgrp_subgroup Γ'
  ⟨proof⟩

sublocale cong_subgroup Γ'
  ⟨proof⟩

end

bundle modgrp_notation
begin

notation modular_group.rel (infixl "∼Γ" 49)
notation modgrps_pcong ("(<notation=<mixfix modgrps_pcong>>Γ'(_'))")
notation modgrps_hecke ("(<notation=<mixfix modgrps_hecke>>Γ0'(_'))")
notation modgrps_unip ("(<notation=<mixfix modgrps_unip>>Γ1'(_'))")

end

unbundle no_modgrp_notation

end

```

5 Fundamental regions of the modular group

```

theory Modular_Fundamental_Region
  imports Modular_Subgroups "Elliptic_Functions.Complex_Lattices" "HOL-Library.Real_Mod"
begin

unbundle modgrp_notation

```

5.1 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 $"x \in R"$ $"y \in \text{closure } R"$ $"\text{rel } x \ y"$ $"\text{Im } y > 0"$

shows $"x = y"$

$\langle \text{proof} \rangle$

lemma

pole_modgrp_not_in_region [simp]: $"\text{pole_modgrp } f \notin R"$ **and**

pole_image_modgrp_not_in_region [simp]: $"\text{pole_image_modgrp } f \notin R"$

$\langle \text{proof} \rangle$

end

5.2 The standard fundamental region

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

definition *std_fund_region* :: "complex set" (" \mathcal{R}_Γ ") **where**

$"\mathcal{R}_\Gamma = \text{-cball } 0 \ 1 \cap \text{Re } \{-1/2 < .. < 1/2\} \cap \{z. \text{Im } z > 0\}"$

The following version of \mathcal{R}_Γ is what Apostol refers to as the closure of \mathcal{R}_Γ , 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 $\text{Im}(z) = 0$ axis, we only want one of these copies to be in \mathcal{R}_Γ' , and we choose the left one.

So \mathcal{R}_Γ' is actually \mathcal{R}_Γ 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}_Γ' ") **where**

$"\mathcal{R}_\Gamma' = \mathcal{R}_\Gamma \cup (\text{-ball } 0 \ 1 \cap \text{Re } \{-1/2..0\} \cap \{z. \text{Im } z > 0\})"$

lemma *std_fund_region_altdef*:

$"\mathcal{R}_\Gamma = \{z. \text{norm } z > 1 \wedge \text{norm } (z + \text{cnj } z) < 1 \wedge \text{Im } z > 0\}"$

$\langle \text{proof} \rangle$

lemma *in_std_fund_region_iff*:

$"z \in \mathcal{R}_\Gamma \longleftrightarrow \text{norm } z > 1 \wedge \text{Re } z \in \{-1/2 < .. < 1/2\} \wedge \text{Im } z > 0"$

$\langle \text{proof} \rangle$

lemma *in_std_fund_region'_iff*:

$"z \in \mathcal{R}_\Gamma' \longleftrightarrow \text{Im } z > 0 \wedge ((\text{norm } z > 1 \wedge \text{Re } z \in \{-1/2 < .. < 1/2\}) \vee (\text{norm } z = 1 \wedge \text{Re } z \in \{-1/2..0\}))"$

$\langle \text{proof} \rangle$

lemma *open_std_fund_region [simp, intro]*: "open \mathcal{R}_Γ "

$\langle \text{proof} \rangle$

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

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

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*: " $f \text{ ` } S \subseteq \mathcal{R}_\Gamma$ "
 ⟨*proof*⟩

lemma *image_std_fund_region_subset*: " $g \text{ ` } \mathcal{R}_\Gamma \subseteq S$ "
 ⟨*proof*⟩

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

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

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

lemma *image_subset_std_fund_region'*: " $f \text{ ` } S' \subseteq T$ "
 ⟨*proof*⟩

lemma *image_std_fund_region_subset'*: " $g \text{ ` } T \subseteq S'$ "
 ⟨*proof*⟩

lemma *bij_betw_std_fund_region1'*: " $\text{bij_betw } f \text{ } S' \text{ } T$ "
 ⟨*proof*⟩

lemma *bij_betw_std_fund_region2'*: " $\text{bij_betw } g \text{ } T \text{ } S'$ "
 ⟨*proof*⟩

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

lemma *in_closure_std_fund_region_iff*:
 " $x \in \text{closure } \mathcal{R}_\Gamma \iff \text{norm } x \geq 1 \wedge \text{Re } x \in \{-1/2..1/2\} \wedge \text{Im } x \geq 0$ "
 ⟨*proof*⟩

```

lemma frontier_std_fund_region:
  "frontier  $\mathcal{R}_\Gamma$  =
    {z. norm z  $\geq$  1  $\wedge$  Im z  $>$  0  $\wedge$  |Re z| = 1 / 2}  $\cup$ 
    {z. norm z = 1  $\wedge$  Im z  $>$  0  $\wedge$  |Re z|  $\leq$  1 / 2}" (is "_ = ?rhs")
<proof>

lemma std_fund_region'_subset_closure: " $\mathcal{R}_\Gamma' \subseteq \text{closure } \mathcal{R}_\Gamma$ "
<proof>

lemma std_fund_region'_superset: " $\mathcal{R}_\Gamma \subseteq \mathcal{R}_\Gamma'$ "
<proof>

lemma in_std_fund_region'_not_on_frontier_iff:
  assumes "z  $\notin$  frontier  $\mathcal{R}_\Gamma$ "
  shows "z  $\in \mathcal{R}_\Gamma' \longleftrightarrow z \in \mathcal{R}_\Gamma$ "
<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: " $\mathcal{R}_\Gamma' \subseteq \text{closure } \mathcal{R}_\Gamma$ "
<proof>

lemma closure_std_fund_region_Im_pos: "closure  $\mathcal{R}_\Gamma \subseteq \{z. \text{Im } z > 0\}$ "
<proof>

lemma closure_std_fund_region_Im_ge: "closure  $\mathcal{R}_\Gamma \subseteq \{z. \text{Im } z \geq \text{sqrt }
3 / 2\}$ "
<proof>

lemma std_fund_region'_minus_std_fund_region:
  " $\mathcal{R}_\Gamma' - \mathcal{R}_\Gamma =$ 
    {z. norm z = 1  $\wedge$  Im z  $>$  0  $\wedge$  Re z  $\in$   $\{-1/2..0\}$ }  $\cup$  {z. Re z = -1
  / 2  $\wedge$  Im z  $\geq$  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  $\wedge$  Im z  $>$  0  $\wedge$  Re z  $\in$   $\{0<..1/2\}$ }  $\cup$  {z. Re z = 1 /
  2  $\wedge$  Im z  $\geq$  sqrt 3 / 2}"
  (is "?lhs = ?rhs")
<proof>

lemma cis_in_std_fund_region'_iff:

```

```

    assumes " $\varphi \in \{0..pi\}$ "
    shows " $\text{cis } \varphi \in \mathcal{R}_\Gamma' \iff \varphi \in \{pi/2..2*pi/3\}$ "
  <proof>

lemma imag_axis_in_std_fund_region'_iff: " $y *_R i \in \mathcal{R}_\Gamma' \iff y \geq 1$ "
  <proof>

lemma vertical_left_in_std_fund_region'_iff:
  " $-1 / 2 + y *_R i \in \mathcal{R}_\Gamma' \iff y \geq \text{sqrt } 3 / 2$ "
  <proof>

lemma std_fund_region'_border_aux1:
  " $\{z. \text{norm } z = 1 \wedge 0 < \text{Im } z \wedge \text{Re } z \in \{-1/2..0\}\} = \text{cis } \{ \{pi / 2..2 / 3 * pi\}$ "
  <proof>

lemma std_fund_region'_border_aux2:
  " $\{z. \text{Re } z = -1 / 2 \wedge \text{sqrt } 3 / 2 \leq \text{Im } z\} = (\lambda x. -1 / 2 + x *_R i) \{ \text{sqrt } 3 / 2.. \}$ "
  <proof>

lemma compact_std_fund_region:
  assumes " $B > 1$ "
  shows " $\text{compact } (\text{closure } \mathcal{R}_\Gamma \cap \{z. \text{Im } z \leq B\})$ "
  <proof>

end

```

5.3 Proving that the standard region is fundamental

```

lemma norm_open_segment_less:
  fixes x y z :: "'a :: euclidean_space"
  assumes " $\text{norm } x \leq \text{norm } y$ " " $z \in \text{open\_segment } x y$ "
  shows " $\text{norm } z < \text{norm } y$ "
  <proof>

Lemma 1

lemma (in complex_lattice) std_fund_region_fundamental_lemma1:
  obtains  $\omega_1' \omega_2' :: \text{complex}$  and  $a b c d :: \text{int}$ 
  where " $|a * d - b * c| = 1$ "
    " $\omega_2' = \text{of\_int } a * \omega_2 + \text{of\_int } b * \omega_1$ "
    " $\omega_1' = \text{of\_int } c * \omega_2 + \text{of\_int } d * \omega_1$ "
    " $\text{Im } (\omega_2' / \omega_1') \neq 0$ "
    " $\text{norm } \omega_1' \leq \text{norm } \omega_2'$ " " $\text{norm } \omega_2' \leq \text{norm } (\omega_1' + \omega_2')$ " " $\text{norm } \omega_2' \leq \text{norm } (\omega_1' - \omega_2')$ "
  <proof>

lemma (in complex_lattice) std_fund_region_fundamental_lemma2:
  obtains  $\omega_1' \omega_2' :: \text{complex}$  and  $a b c d :: \text{int}$ 

```

```

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 ~Γ 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 ∈ ℛΓ" "y ∈ ℛΓ"
  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 ∈ ℛΓ" "y ∈ ℛΓ"
  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⟩

```

Every point in the upper half plane has a canonical representative w.r.t. the full modular group in the standard fundamental region.

```

lemma canonical_point_in_std_fund_region':
  assumes "Im z > 0"
  obtains z' where "z' ∈ ℛΓ" "z ~Γ z'"
⟨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>

```

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

5.4 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)"

```

$\langle proof \rangle$

lemma *norm_modfun_rho_plus_1* [*simp*]: " $norm (\varrho + 1) = 1$ "
 $\langle proof \rangle$

lemma *cnj_modfun_rho*: " $cnj \varrho = -\varrho - 1$ "
and *cnj_modfun_rho_plus1*: " $cnj (\varrho + 1) = -\varrho$ "
 $\langle proof \rangle$

lemma *modfun_rho_cube*: " $\varrho^3 = 1$ "
 $\langle proof \rangle$

lemma *modfun_rho_power_mod3_reduce*: " $\varrho^n = \varrho^{(n \bmod 3)}$ "
 $\langle proof \rangle$

lemma *modfun_rho_power_mod3_reduce'*: " $n \geq 3 \implies \varrho^n = \varrho^{(n \bmod 3)}$ "
 $\langle proof \rangle$

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

lemma *modfun_rho_square*: " $\varrho^2 = -\varrho - 1$ "
 $\langle proof \rangle$

lemma *modfun_rho_not_real* [*simp*]: " $\varrho \notin \mathbb{R}$ "
 $\langle proof \rangle$

lemma *modfun_rho_nonzero* [*simp*]: " $\varrho \neq 0$ "
 $\langle proof \rangle$

lemma *modfun_rho_not_one* [*simp*]: " $\varrho \neq 1$ "
 $\langle proof \rangle$

lemma *i_neq_modfun_rho* [*simp*]: " $i \neq \varrho$ "
and *i_neq_modfun_rho_plus1* [*simp*]: " $i \neq \varrho + 1$ "
and *modfun_rho_neg_i* [*simp*]: " $\varrho \neq i$ "
and *modfun_rho_plus1_neg_i* [*simp*]: " $\varrho + 1 \neq i$ "
 $\langle proof \rangle$

lemma *i_in_closure_std_fund_region* [*intro, simp*]: " $i \in \text{closure } \mathcal{R}_\Gamma$ "
and *i_in_std_fund_region'* [*intro, simp*]: " $i \in \mathcal{R}_\Gamma$ "
and *modfun_rho_in_closure_std_fund_region* [*intro, simp*]: " $\varrho \in \text{closure } \mathcal{R}_\Gamma$ "
and *modfun_rho_in_std_fund_region'* [*intro, simp*]: " $\varrho \in \mathcal{R}_\Gamma$ "
and *modfun_rho_plus_1_notin_closure_std_fund_region* [*intro, simp*]: " $\varrho + 1 \in \text{closure } \mathcal{R}_\Gamma$ "
and *modfun_rho_plus_1_notin_std_fund_region'* [*intro, simp*]: " $\varrho + 1 \notin \mathcal{R}_\Gamma$ "
 $\langle proof \rangle$

lemma *modfun_rho_power_eq_1_iff*: " $\varrho^n = 1 \iff 3 \text{ dvd } n$ "
 <proof>

5.5 The elliptic points of the modular group

We will now explicitly compute the stabilisers of i and ρ and, from that, see that they are elliptic points of order 2 and 3, respectively.

We will later see that these are the only elliptic points of the modular group (up to equivalence w.r.t. unimodular transformations, of course).

lemma *abs_le_1_iff_int*: " $\text{abs } n \leq 1 \iff n = 0 \vee n = 1 \vee n = (-1::\text{int})$ "
 <proof>

lemma *abs_eq_1_iff*: " $\text{abs } (x :: 'a :: \text{linordered_idom}) = 1 \iff x = 1 \vee x = -1$ "
 <proof>

The stabilisers of two equivalent points are related by conjugation.

lemma *bij_betw_stabiliser_modgrp_apply_modgrp*:
 fixes $h :: \text{modgrp}$
 assumes $x: "Im\ x > 0"$
 defines " $\text{Stab} \equiv (\lambda x. \{h. \text{apply_modgrp } h\ x = x\})$ "
 defines " $y \equiv \text{apply_modgrp } h\ x$ "
 shows " $\text{bij_betw } (\lambda f. h * f * \text{inverse } h) (\text{Stab } x) (\text{Stab } y)$ "
 <proof>

lemma *stabiliser_ii_modgrp*: " $\text{apply_modgrp } h\ i = i \iff h \in \{1, -1, S_{\text{modgrp}}, -S_{\text{modgrp}}\}$ "
 <proof>

lemma *ellorder_modgrp_UNIV_ii [simp]*: " $\text{ellorder_modgrp UNIV } i = 2$ "
 <proof>

lemma *ellorder_modgrp_UNIV_ii'*: " $z \sim_{\Gamma} i \implies \text{ellorder_modgrp UNIV } z = 2$ "
 <proof>

lemma *stabiliser_rho_modgrp_aux*:
 assumes " $a^2 + a * b + b^2 \leq (1::\text{int})$ "
 shows " $|b| \leq 1$ "
 <proof>

lemma *stabiliser_rho_modgrp*:
 defines " $ST \equiv S_{\text{modgrp}} * T_{\text{modgrp}}$ "
 shows " $\text{apply_modgrp } h\ \varrho = \varrho \iff h \in \{1, -1, ST, -ST, ST^2, -(ST^2)\}$ "
 <proof>

lemma *ellorder_modgrp_UNIV_rho [simp]*: " $\text{ellorder_modgrp UNIV } \varrho = 3$ "

<proof>

lemma *ellorder_modgrp_UNIV_rho'*: " $z \sim_{\Gamma} \rho \implies \text{ellorder_modgrp UNIV } z = 3$ "
<proof>

Other than i and ρ , no non-trivial unimodular transformation has any fixed points in the fundamental region.

lemma *modgrp_fixed_point_trivial_std_fund_region'*:
 assumes " $z \in \mathcal{R}_{\Gamma}' - \{i, \rho\}$ " "*apply_modgrp h z = z*"
 shows "*abs h = 1*"
<proof>

Therefore, all points not equivalent to i or ρ have elliptic order 1, i.e. are not elliptic points.

lemma *ellorder_modgrp_UNIV_eq_1_std_fund_region'*:
 assumes " $z \in \mathcal{R}_{\Gamma}' - \{i, \rho\}$ "
 shows "*ellorder_modgrp UNIV z = 1*"
<proof>

lemma *ellorder_modgrp_UNIV_eq_1*:
 assumes "*Im z > 0*" " $\neg(z \sim_{\Gamma} i)$ " " $\neg(z \sim_{\Gamma} \rho)$ "
 shows "*ellorder_modgrp UNIV z = 1*"
<proof>

lemma *ellorder_modgrp_UNIV_eq*:
 "*ellorder_modgrp UNIV z =*
 (if Im z ≤ 0 then 0 else if z ~_Γ i then 2 else if z ~_Γ ρ then
 3 else 1)"
<proof>

A simple consequence: no unimodular transformation sends ρ to i .

lemma *not_rho_equiv_i [simp]*: " $\neg(\rho \sim_{\Gamma} i)$ "
<proof>

lemma *not_i_equiv_rho [simp]*: " $\neg(i \sim_{\Gamma} \rho)$ "
<proof>

lemma *not_modular_group_rel_rho_i [simp]*: " $z \sim_{\Gamma} \rho \implies \neg z \sim_{\Gamma} i$ "
<proof>

lemma *modular_group_rel_rho_i_cases [case_names rho i neither invalid]*:
 obtains " $z \sim_{\Gamma} \rho$ " " $\neg z \sim_{\Gamma} i$ " | " $z \sim_{\Gamma} i$ " " $\neg z \sim_{\Gamma} \rho$ " | "*Im z > 0*" " $\neg z \sim_{\Gamma} \rho$ " " $\neg z \sim_{\Gamma} i$ " | "*Im z ≤ 0*"
<proof>

lemma *finite_modgrp_fixpoints*:
 assumes "*Im z > 0*"

```

  shows "finite {h∈G. apply_modgrp h z = z}"
⟨proof⟩

```

```

lemma (in modgrp_subgroup) ellorder_modgrp_pos:
  assumes "Im z > 0"
  shows "ellorder_modgrp G z > 0"
⟨proof⟩

```

```

lemma (in modgrp_subgroup)
  assumes "Im z > 0"
  shows ellorder_modgrp_dvd: "card (G ∩ {1, -1}) dvd card {h∈G. apply_modgrp
h z = z}"
  and card_modgrp_fixpoints:
    "card {h∈G. apply_modgrp h z = z} = ellorder_modgrp G z *
card (G ∩ {1, -1})"
⟨proof⟩

```

A point having an elliptic order of 1 means that the point is fixed only by the trivial maps $\pm I$.

```

lemma (in modgrp_subgroup) ellorder_modgrp_eq_1_iff:
  assumes "Im z > 0"
  shows "ellorder_modgrp G z = 1  $\longleftrightarrow$  ( $\forall f \in G. \text{apply\_modgrp } f \text{ } z = z \longleftrightarrow |f| = 1$ )"
⟨proof⟩

```

```

lemma modgrp_fixed_point_trivial:
  assumes "Im z > 0" "~z ~ $\Gamma$  i" "~z ~ $\Gamma$   $\rho$ " "apply_modgrp f z = z"
  shows "|f| = 1"
⟨proof⟩

```

5.6 Fundamental regions for congruence subgroups

```

context hecke_prime_subgroup
begin

```

```

definition std_fund_region_cong ("R") where
  "R = R $\Gamma$   $\cup$  ( $\bigcup_{k \in \{0..<p\}}$ . ( $\lambda z. -1 / (z + \text{of\_int } k)$ ) ' R $\Gamma$ )"

```

```

lemma std_fund_region_cong_altdef:
  "R = R $\Gamma$   $\cup$  ( $\bigcup_{k \in \{0..<p\}}$ . apply_modgrp (S_shift_modgrp k) ' R $\Gamma$ )"
⟨proof⟩

```

```

lemma closure_UN_finite: "finite A  $\implies$  closure ( $\bigcup A$ ) = ( $\bigcup_{X \in A. \text{closure } X$ )"
⟨proof⟩

```

```

sublocale std_region: fundamental_region  $\Gamma$ ' R
⟨proof⟩

```

end

```
bundle modfun_region_notation
begin
notation std_fund_region (" $\mathcal{R}_\Gamma$ ")
notation modfun_rho (" $\rho$ ")
end
```

```
unbundle no_modfun_region_notation
unbundle no_modgrp_notation
```

end

6 Elliptic Functions

```
theory Elliptic_Functions
  imports Complex_Lattices
begin
```

6.1 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  $\omega_1$   $\omega_2$   $f$ 
  for  $\omega_1$   $\omega_2$  :: complex and  $f$  :: "complex  $\Rightarrow$  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  $\omega_1$   $\omega_2$   $f$ 
  for  $\omega_1$   $\omega_2$  :: complex and  $f$  :: "complex  $\Rightarrow$  complex" +
  assumes nicely_meromorphic: " $f$  nicely_meromorphic_on UNIV"
```

```
locale elliptic_function_remove_sings = elliptic_function
begin
```

```
sublocale remove_sings: nicely_elliptic_function  $\omega_1$   $\omega_2$  "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  $\omega_1 \omega_2$  "isolated_zero f"
  ⟨proof⟩

sublocale poles: complex_lattice_periodic  $\omega_1 \omega_2$  "is_pole f"
  ⟨proof⟩

sublocale zorder: complex_lattice_periodic  $\omega_1 \omega_2$  "zorder f"
  ⟨proof⟩

sublocale deriv: complex_lattice_periodic  $\omega_1 \omega_2$  "deriv f"
  ⟨proof⟩

sublocale higher_deriv: complex_lattice_periodic  $\omega_1 \omega_2$  "(deriv ^^ n)
f"
  ⟨proof⟩

sublocale residue: complex_lattice_periodic  $\omega_1 \omega_2$  "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

```

6.2 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  $\longleftrightarrow$  f constant_on UNIV"
<proof>
```

```
lemma order_pos_iff: "elliptic_order f > 0  $\longleftrightarrow$   $\neg$ f constant_on UNIV"
<proof>
```

The following lemma allows us to evaluate an integral of the form $\int_P h(w)f'(w)/f(w) dw$ 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 " $\gamma \equiv$  parallelogram_path orig  $\omega_1$   $\omega_2$ "
  assumes h: "h holomorphic_on UNIV"
  assumes nz: " $\bigwedge z. z \in$  path_image  $\gamma \implies f z \neq 0 \wedge \neg$ is_pole f z"
  shows "contour_integral  $\gamma$  ( $\lambda x. h x * deriv f x / f x$ ) =
        contour_integral (linepath orig (orig +  $\omega_1$ ))
          ( $\lambda z. (h z - h (z + \omega_2)) * deriv f z / f z$ ) -
        contour_integral (linepath orig (orig +  $\omega_2$ ))
          ( $\lambda z. (h z - h (z + \omega_1)) * deriv f z / f z$ )"
<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 " $\gamma \equiv$  parallelogram_path orig  $\omega_1$   $\omega_2$ "
  assumes nz: " $\bigwedge z. z \in$  path_image  $\gamma \implies f z \neq 0 \wedge \neg$ is_pole f z"
  shows "contour_integral (parallelogram_path orig  $\omega_1$   $\omega_2$ ) ( $\lambda x. deriv f x / f x$ ) = 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 " $\gamma \equiv$  parallelogram_path orig  $\omega_1$   $\omega_2$ "
  assumes wf: " $\bigwedge z. z \in$  path_image  $\gamma \implies f z \neq 0 \wedge \neg$ is_pole f z"
  shows "contour_integral  $\gamma$  ( $\lambda z. z * deriv f z / f z$ ) / (2*pi*i)  $\in$   $\Lambda$ "
<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:
  "( $\sum z \mid z \in \text{period\_parallelogram orig} \wedge \text{isolated\_zero } f z. \text{nat } (\text{zorder } f z)) = \text{elliptic\_order } f"$ 
  <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  $\equiv (\lambda \text{orig. } \{z \in \text{period\_parallelogram orig. isolated\_zero } f z \vee \text{is\_pole } f z\})"$ "
  defines "S  $\equiv (\lambda \text{orig. } \sum z \in Z \text{ orig. of\_int } (\text{zorder } f z) * z)"$ "
  shows "S orig  $\in \Lambda"$ "
  <proof>
```

Again, similarly: The residues in a period parallelogram sum to 0.

```
lemma sum_residues_eq_0_aux:
  defines "Q  $\equiv (\lambda \text{orig. } \{z \in \text{period\_parallelogram orig. is\_pole } f z\})"$ "
  defines "S  $\equiv (\lambda \text{orig. } \sum z \in Q \text{ orig. residue } f z)"$ "
  shows "S orig  $\in \Lambda"$ "
  <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:
  "( $\sum z \mid z \in \text{period\_parallelogram orig} \wedge \text{isolated\_zero } f z. \text{nat } (\text{zorder } f z)) = \text{elliptic\_order } f"$ 
  <proof>
```

```
lemma card_poles_le_order: "card {z  $\in$  period\_parallelogram orig. is\_pole f z}  $\leq$  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_cospars*:

"elliptic_order f = 0 ↔ (∃c. ∀_≈x∈UNIV. f x = c)"

<proof>

lemma *cospars_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) (cospars UNIV)"

<proof>

lemma *eventually_eq_imp_almost_everywhere_eq*:

assumes "eventually (λz. f z = c) (at z)"

shows "eventually (λz. f z = c) (cospars 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 " $(\sum z \in Q. \text{residue } f \ z) \in \Lambda$ "
 <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 $f :: 'a \Rightarrow \text{nat}$
assumes *sum_eq*: " $(\sum x \in A. f \ x) = 1$ "
assumes *pos*: " $\bigwedge x. x \in A \implies f \ x > 0$ "
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* $f \neq 1$ "
 <proof>

end

locale *nonconst_nicely_elliptic_function* = *nicely_elliptic_function* +
assumes *order_pos*: "*elliptic_order* $f > 0$ "
begin

lemma *isolated_zero_iff'*: "*isolated_zero* $f \ z \longleftrightarrow \neg \text{is_pole } f \ z \wedge f \ z = 0$ "
 <proof>

end

6.3 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

6.4 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  $\omega_1$   $\omega_2$  ( $\lambda_. c$ )"
  <proof>

lemma [elliptic_function_intros]:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows elliptic_function_cmult_left: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. c * f z$ )"
    and elliptic_function_cmult_right: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f z * c$ )"
    and elliptic_function_scaleR: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. c' *_R f z$ )"
    and elliptic_function_uminus: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. -f z$ )"
    and elliptic_function_inverse: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. \text{inverse } (f z)$ )"
    and elliptic_function_power: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f z ^ m$ )"
    and elliptic_function_power_int: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f z \text{ pow } n$ )"
  <proof>

lemma [elliptic_function_intros]:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f" "elliptic_function  $\omega_1$   $\omega_2$  g"
  shows elliptic_function_cmult_add: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f z + g z$ )"
    and elliptic_function_cmult_diff: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f z - g z$ )"
    and elliptic_function_cmult_mult: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f z * g z$ )"
    and elliptic_function_cmult_divide: "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f z / g z$ )"
  <proof>

lemma elliptic_function_compose_mult_of_int_left:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f (\text{of\_int } n * z)$ )"
  <proof>

lemma elliptic_function_compose_mult_of_nat_left:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f (\text{of\_nat } n * z)$ )"
  <proof>

```

```

lemma elliptic_function_compose_mult_numeral_left:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f$  (numeral n * z))"
  <proof>

lemma
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows elliptic_function_compose_mult_of_int_right: "elliptic_function
 $\omega_1$   $\omega_2$  ( $\lambda z. f$  (z * of_int n))"
    and elliptic_function_compose_mult_of_nat_right: "elliptic_function
 $\omega_1$   $\omega_2$  ( $\lambda z. f$  (z * of_nat m))"
    and elliptic_function_compose_mult_numeral_right: "elliptic_function
 $\omega_1$   $\omega_2$  ( $\lambda z. f$  (z * numeral num))"
  <proof>

lemma elliptic_function_compose_uminus:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f$  (-z))"
  <proof>

lemma elliptic_function_shift:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. f$  (z + w))"
  <proof>

definition shift_fun :: "'a  $\Rightarrow$  ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$  'a :: plus" where
  "shift_fun w f = ( $\lambda z. f$  (z + w))"

lemma elliptic_function_shift' [elliptic_function_intros]:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows "elliptic_function  $\omega_1$   $\omega_2$  (shift_fun w f)"
  <proof>

lemma nicely_elliptic_function_remove_sings [elliptic_function_intros]:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows "nicely_elliptic_function  $\omega_1$   $\omega_2$  (remove_sings f)"
  <proof>

lemma elliptic_function_remove_sings [elliptic_function_intros]:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows "elliptic_function  $\omega_1$   $\omega_2$  (remove_sings f)"
  <proof>

lemma elliptic_function_deriv [elliptic_function_intros]:
  assumes "elliptic_function  $\omega_1$   $\omega_2$  f"
  shows "elliptic_function  $\omega_1$   $\omega_2$  (deriv f)"

```

<proof>

```
lemma elliptic_function_higher_deriv [elliptic_function_intros]:  
  assumes "elliptic_function  $\omega_1$   $\omega_2$   $f$ "  
  shows "elliptic_function  $\omega_1$   $\omega_2$  ((deriv  $\wedge$  n)  $f$ )"  
  <proof>
```

```
lemma elliptic_function_sum [elliptic_function_intros]:  
  assumes " $\wedge x. x \in X \implies$  elliptic_function  $\omega_1$   $\omega_2$  ( $f$   $x$ )"  
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. \sum_{x \in X}. f$   $x$   $z$ )"  
  <proof>
```

```
lemma elliptic_function_prod [elliptic_function_intros]:  
  assumes " $\wedge x. x \in X \implies$  elliptic_function  $\omega_1$   $\omega_2$  ( $f$   $x$ )"  
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. \prod_{x \in X}. f$   $x$   $z$ )"  
  <proof>
```

```
lemma elliptic_function_sum_list [elliptic_function_intros]:  
  assumes " $\wedge f. f \in \text{set } fs \implies$  elliptic_function  $\omega_1$   $\omega_2$   $f$ "  
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. \sum f \leftarrow fs. f$   $z$ )"  
  <proof>
```

```
lemma elliptic_function_prod_list [elliptic_function_intros]:  
  assumes " $\wedge f. f \in \text{set } fs \implies$  elliptic_function  $\omega_1$   $\omega_2$   $f$ "  
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. \prod f \leftarrow fs. f$   $z$ )"  
  <proof>
```

```
lemma elliptic_function_sum_mset [elliptic_function_intros]:  
  assumes " $\wedge f. f \in \# F \implies$  elliptic_function  $\omega_1$   $\omega_2$   $f$ "  
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. \sum f \in \# F. f$   $z$ )"  
  <proof>
```

```
lemma elliptic_function_prod_mset [elliptic_function_intros]:  
  assumes " $\wedge f. f \in \# F \implies$  elliptic_function  $\omega_1$   $\omega_2$   $f$ "  
  shows "elliptic_function  $\omega_1$   $\omega_2$  ( $\lambda z. \prod f \in \# F. f$   $z$ )"  
  <proof>
```

end

6.5 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  $\Rightarrow$  complex"  
  defines " $g \equiv \lambda z. a * f$   $z + b$ "  
  assumes nonzero_const: " $a \neq 0$ "  
begin
```

```

sublocale affine: elliptic_function  $\omega_1 \omega_2 g$ 
  <proof>

lemma is_pole_affine_iff: "is_pole g z  $\longleftrightarrow$  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 " $\neg$ is_pole f z" "z  $\in$  period_parallelogram w" "f z = c"
  <proof>

```

end

end

7 The Weierstraß \wp Function

```

theory Weierstrass_Elliptic
imports
  Elliptic_Functions
  Modular_Group
  Theta_Functions.Theta_Nullwert
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' .

7.1 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  $\omega$ _upper:
```

```
  assumes " $\omega \in$  lattice_layer  $k$ " and " $\alpha > 0$ " and " $k > 0$ "
  shows " $\text{norm } \omega \text{ powr } -\alpha \leq (k * \text{Inf\_para}) \text{ powr } -\alpha$ "
  <proof>
```

```
lemma sum_ $\omega$ _upper:
```

```
  assumes " $\alpha > 0$ " and " $k > 0$ "
  shows " $(\sum \omega \in$  lattice_layer  $k. \text{norm } \omega \text{ powr } -\alpha) \leq 8 * k \text{ powr } (1-\alpha)$ "
  * Inf_para powr  $-\alpha$ "
  (is "?lhs  $\leq$  ?rhs")
  <proof>
```

```
lemma lattice_layer_lower:
```

```
  assumes " $\omega \in$  lattice_layer  $k$ " and " $k > 0$ "
  shows " $(k * (\text{if } \alpha \geq 0 \text{ then Inf\_para else Sup\_para})) \text{ powr } \alpha \leq \text{norm } \omega \text{ powr } \alpha$ "
  <proof>
```

```
lemma sum_lattice_layer_lower:
```

```
  fixes  $\alpha ::$  real
  assumes " $k > 0$ "
  defines " $C \equiv (\text{if } \alpha \geq 0 \text{ then Sup\_para else Inf\_para})$ "
  shows " $8 * k \text{ powr } (1-\alpha) * C \text{ powr } -\alpha \leq (\sum \omega \in$  lattice_layer  $k. \text{norm } \omega \text{ powr } -\alpha)$ "
```

(is "?lhs ≤ ?rhs")
 <proof>

lemma *converges_absolutely_iff_aux1*:
 fixes $\alpha :: \text{real}$
 assumes " $\alpha > 2$ "
 shows "summable ($\lambda i. \sum_{\omega \in \text{lattice_layer } (\text{Suc } i)}. 1 / \text{norm } \omega \text{ powr } \alpha$)"
 <proof>

lemma *converges_absolutely_iff_aux2*:
 fixes $\alpha :: \text{real}$
 assumes "summable ($\lambda i. \sum_{\omega \in \text{lattice_layer } (\text{Suc } i)}. 1 / \text{norm } \omega \text{ powr } \alpha$)"
 shows " $\alpha > 2$ "
 <proof>

Apostol's Lemma 1

lemma *converges_absolutely_iff*:
 fixes $\alpha :: \text{real}$
 shows " $(\lambda \omega. 1 / \text{norm } \omega \text{ powr } \alpha) \text{ summable_on } \Lambda^* \longleftrightarrow \alpha > 2$ "
 (is "?P \longleftrightarrow _")
 <proof>

lemma *bounded_lattice_finite*:
 assumes "bounded B"
 shows "finite ($\Lambda \cap B$)"
 <proof>

lemma *closed_subset_lattice*: " $\Lambda' \subseteq \Lambda \implies \text{closed } \Lambda'$ "
 <proof>

corollary *closed_lattice0*: " $\text{closed } \Lambda^*$ "
 <proof>

lemma *weierstrass_summand_bound*:
 assumes " $\alpha \geq 1$ " and " $R > 0$ "
 obtains M where
 " $M > 0$ "
 " $\bigwedge \omega z. [\omega \in \Lambda; \text{cmod } \omega > R; \text{cmod } z \leq R] \implies \text{norm } (z - \omega) \text{ powr } -\alpha \leq M * (\text{norm } \omega \text{ powr } -\alpha)$ "
 <proof>

Lemma 2 on Apostol p. 8

lemma *weierstrass_aux_converges_absolutely_in_disk*:
 assumes " $\alpha > 2$ " and " $R > 0$ " and " $z \in \text{cball } 0 R$ "
 shows " $(\lambda \omega. \text{cmod } (z - \omega) \text{ powr } -\alpha) \text{ summable_on } (\Lambda - \text{cball } 0 R)$ "
 <proof>

```

lemma weierstrass_aux_converges_absolutely_in_disk':
  fixes  $\alpha :: \text{nat}$  and  $R :: \text{real}$  and  $z :: \text{complex}$ 
  assumes " $\alpha > 2$ " and " $R > 0$ " and " $z \in \text{cball } 0 R$ "
  shows " $(\lambda \omega. 1 / \text{norm } (z - \omega) ^ \alpha) \text{ summable\_on } (\Lambda - \text{cball } 0 R)$ "
  <proof>

lemma weierstrass_aux_converges_in_disk':
  fixes  $\alpha :: \text{nat}$  and  $R :: \text{real}$  and  $z :: \text{complex}$ 
  assumes " $\alpha > 2$ " and " $R > 0$ " and " $z \in \text{cball } 0 R$ "
  shows " $(\lambda \omega. 1 / (z - \omega) ^ \alpha) \text{ summable\_on } (\Lambda - \text{cball } 0 R)$ "
  <proof>

lemma weierstrass_aux_converges_absolutely:
  fixes  $\alpha :: \text{real}$ 
  assumes " $\alpha > 2$ " and " $\Lambda' \subseteq \Lambda$ "
  shows " $(\lambda \omega. \text{norm } (z - \omega) \text{ powr } -\alpha) \text{ summable\_on } \Lambda'$ "
  <proof>

lemma weierstrass_aux_converges_absolutely':
  fixes  $\alpha :: \text{nat}$ 
  assumes " $\alpha > 2$ " and " $\Lambda' \subseteq \Lambda$ "
  shows " $(\lambda \omega. 1 / \text{norm } (z - \omega) ^ \alpha) \text{ summable\_on } \Lambda'$ "
  <proof>

lemma weierstrass_aux_converges:
  fixes  $\alpha :: \text{real}$ 
  assumes " $\alpha > 2$ " and " $\Lambda' \subseteq \Lambda$ "
  shows " $(\lambda \omega. (z - \omega) \text{ powr } -\alpha) \text{ summable\_on } \Lambda'$ "
  <proof>

lemma weierstrass_aux_converges':
  fixes  $\alpha :: \text{nat}$ 
  assumes " $\alpha > 2$ " and " $\Lambda' \subseteq \Lambda$ "
  shows " $(\lambda \omega. 1 / (z - \omega) ^ \alpha) \text{ summable\_on } \Lambda'$ "
  <proof>

lemma
  fixes  $\alpha R :: \text{real}$ 
  assumes " $\alpha > 2$ " " $R > 0$ "
  shows weierstrass_aux_converges_absolutely_uniformly_in_disk:
    "uniform_limit (cball 0 R)
      ( $\lambda X z. \sum \omega \in X. \text{norm } ((z - \omega) \text{ powr } -\alpha)$ )
      ( $\lambda z. \sum_{\infty} \omega \in \Lambda - \text{cball } 0 R. \text{norm } ((z - \omega) \text{ powr } -\alpha)$ )
      (finite_subsets_at_top ( $\Lambda - \text{cball } 0 R$ ))" (is
?th1)
  and weierstrass_aux_converges_uniformly_in_disk:
    "uniform_limit (cball 0 R)
      ( $\lambda X z. \sum \omega \in X. (z - \omega) \text{ powr } -\alpha$ )
      ( $\lambda z. \sum_{\infty} \omega \in \Lambda - \text{cball } 0 R. (z - \omega) \text{ powr } -\alpha$ )

```

```

                                (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  $\alpha :: \text{nat}$  and  $R :: \text{real}$ 
  defines " $F \equiv (\lambda z. \sum_{\omega \in \Lambda\text{-cball } 0 R. 1 / (z - \omega) ^ \alpha)$ "
  assumes " $\alpha > 2$ " " $R > 0$ " " $w \in \text{ball } 0 R$ "
  shows " $(F \alpha \text{ has\_field\_derivative -of\_nat } \alpha * F (\text{Suc } \alpha) w) (\text{at } w)$ "
  <proof>

lemma eisenstein_fun_aux_has_field_derivative:
  assumes  $z: "z \notin \Lambda^*" \text{ and } n: "n \geq 3"$ 
  shows " $(\text{eisenstein\_fun\_aux } n \text{ has\_field\_derivative -of\_nat } n * \text{eisenstein\_fun\_aux } (\text{Suc } n) z) (\text{at } z)$ "
  <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  $z: "z \notin \Lambda^*" \text{ and } n: "n \geq 3"$ 
  shows " $(\text{deriv } ^ m) (\text{eisenstein\_fun\_aux } n) z =$ 
     $(-1) ^ m * \text{pochhammer } (\text{of\_nat } n) m * \text{eisenstein\_fun\_aux } (n$ 
   $+ m) z$ "
  <proof>

lemma eisenstein_fun_aux_holomorphic: "eisenstein_fun_aux n holomorphic_on
 $-\Lambda^*$ "
  <proof>

lemma eisenstein_fun_aux_holomorphic' [holomorphic_intros]:
  assumes " $f \text{ holomorphic\_on } A$ " " $\bigwedge z. z \in A \implies f z \notin \Lambda^*$ "
  shows " $(\lambda z. \text{eisenstein\_fun\_aux } n (f z)) \text{ holomorphic\_on } A$ "
  <proof>

lemma eisenstein_fun_aux_analytic: "eisenstein_fun_aux n analytic_on
 $-\Lambda^*$ "
  <proof>

lemma eisenstein_fun_aux_analytic' [analytic_intros]:
  assumes " $f \text{ analytic\_on } A$ " " $\bigwedge z. z \in A \implies f z \notin \Lambda^*$ "
  shows " $(\lambda z. \text{eisenstein\_fun\_aux } n (f z)) \text{ analytic\_on } A$ "
  <proof>

lemma eisenstein_fun_aux_continuous_on: "continuous_on  $(-\Lambda^*) (\text{eisenstein\_fun\_aux } n)$ "
  <proof>

lemma eisenstein_fun_aux_continuous_on' [continuous_intros]:
  assumes " $\text{continuous\_on } A f$ " " $\bigwedge z. z \in A \implies f z \notin \Lambda^*$ "
  shows " $\text{continuous\_on } A (\lambda z. \text{eisenstein\_fun\_aux } n (f z))$ "

```

$\langle proof \rangle$

lemma weierstrass_aux_translate:
fixes $\alpha :: \text{real}$
assumes " $\alpha > 2$ "
shows " $(\sum_{\omega \in \Lambda} (z + w - \omega) \text{ powr } -\alpha) = (\sum_{\omega \in (+) (-w) ' \Lambda} (z - \omega) \text{ powr } -\alpha)$ "
 $\langle proof \rangle$

lemma weierstrass_aux_holomorphic:
assumes " $\alpha > 2$ " " $\Lambda' \subseteq \Lambda$ " "**finite** $(\Lambda - \Lambda')$ "
shows " $(\lambda z. \sum_{\omega \in \Lambda'} 1 / (z - \omega) ^ \alpha)$ **holomorphic_on** $-\Lambda'$ "
 $\langle proof \rangle$

**definition eisenstein_fun :: "nat \Rightarrow complex \Rightarrow complex" where
 $"\text{eisenstein_fun } n \ z = (\text{if } n < 3 \vee z \in \Lambda \text{ then } 0 \text{ else } (\sum_{\omega \in \Lambda} 1 / (z - \omega) ^ n))"$**

lemma eisenstein_fun_has_sum:
 $"n \geq 3 \implies z \notin \Lambda \implies ((\lambda \omega. 1 / (z - \omega) ^ n) \text{ has_sum } \text{eisenstein_fun } n \ z) \ \Lambda"$
 $\langle proof \rangle$

lemma eisenstein_fun_at_pole_eq_0: " $z \in \Lambda \implies \text{eisenstein_fun } n \ z = 0$ "
 $\langle proof \rangle$

lemma eisenstein_fun_conv_eisenstein_fun_aux:
assumes " $n \geq 3$ " " $z \notin \Lambda$ "
shows " $\text{eisenstein_fun } n \ z = \text{eisenstein_fun_aux } n \ z + 1 / z ^ n$ "
 $\langle proof \rangle$

lemma eisenstein_fun_altdef:
 $"\text{eisenstein_fun } n \ z = (\text{if } n < 3 \vee z \in \Lambda \text{ then } 0 \text{ else } \text{eisenstein_fun_aux } n \ z + 1 / z ^ n)"$
 $\langle proof \rangle$

lemma eisenstein_fun_minus: " $\text{eisenstein_fun } n \ (-z) = (-1) ^ n * \text{eisenstein_fun } n \ z$ "
 $\langle proof \rangle$

lemma eisenstein_fun_even_minus: " $\text{even } n \implies \text{eisenstein_fun } n \ (-z) = \text{eisenstein_fun } n \ z$ "
 $\langle proof \rangle$

lemma eisenstein_fun_odd_minus: " $\text{odd } n \implies \text{eisenstein_fun } n \ (-z) = -\text{eisenstein_fun } n \ z$ "
 $\langle proof \rangle$

```

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  $\omega_1$   $\omega_2$  "eisenstein_fun
n"
<proof>

lemmas [elliptic_function_intros] =
  eisenstein_fun.elliptic_function_axioms eisenstein_fun.nicely_elliptic_function_axioms

end

```

7.2 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 '  $\Lambda = \Lambda$  "
<proof>

```

```

lemma minus_latticemz_eq: "uminus '  $\Lambda^* = \Lambda^*$  "
<proof>

```

```

lemma bij_minus_latticemz: "bij_betw uminus  $\Lambda^* \Lambda^*$  "
<proof>

```

```

definition weierstrass_fun_deriv (" $\wp'$ ") where
  "weierstrass_fun_deriv z = -2 * eisenstein_fun 3 z"

```

```

sublocale weierstrass_fun_deriv: elliptic_function  $\omega_1$   $\omega_2$  weierstrass_fun_deriv
<proof>

```

```

sublocale weierstrass_fun_deriv: nicely_elliptic_function  $\omega_1$   $\omega_2$  weierstrass_fun_deriv
<proof>

```

```

lemmas [elliptic_function_intros] =
  weierstrass_fun_deriv.elliptic_function_axioms weierstrass_fun_deriv.nicely_elliptic_func

```

```

lemma weierstrass_fun_deriv_minus [simp]: " $\wp'$  (-z) = - $\wp'$  z"
<proof>

```

```

lemma weierstrass_fun_deriv_has_field_derivative:
  assumes "z  $\notin \Lambda$  "
  shows " $\wp'$  has_field_derivative 6 * eisenstein_fun 4 z) (at z)"

```

<proof>

lemma *weierstrass_fun_deriv_holomorphic*: " \wp' holomorphic_on $-\Lambda$ "
<proof>

lemma *weierstrass_fun_deriv_holomorphic'* [*holomorphic_intros*]:
assumes "*f* holomorphic_on *A*" " $\bigwedge z. z \in A \implies f z \notin \Lambda$ "
shows " $(\lambda z. \wp' (f z))$ holomorphic_on *A*"
<proof>

lemma *weierstrass_fun_deriv_analytic*: " \wp' analytic_on $-\Lambda$ "
<proof>

lemma *weierstrass_fun_deriv_analytic'* [*analytic_intros*]:
assumes "*f* analytic_on *A*" " $\bigwedge z. z \in A \implies f z \notin \Lambda$ "
shows " $(\lambda z. \wp' (f z))$ analytic_on *A*"
<proof>

lemma *weierstrass_fun_deriv_continuous_on*: "continuous_on $(-\Lambda)$ \wp' "
<proof>

lemma *weierstrass_fun_deriv_continuous_on'* [*continuous_intros*]:
assumes "continuous_on *A* *f*" " $\bigwedge z. z \in A \implies f z \notin \Lambda$ "
shows "continuous_on *A* $(\lambda z. \wp' (f z))$ "
<proof>

lemma *tendsto_weierstrass_fun_deriv* [*tendsto_intros*]:
assumes "*f* \longrightarrow *c*" *F*" "*c* $\notin \Lambda$ "
shows " $(\lambda z. \wp' (f z)) \longrightarrow \wp' c$ " *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 \Rightarrow complex" where
"*weierstrass_fun_aux* *z* = (if *z* $\in \Lambda^*$ then 0 else $(\sum_{\infty} \omega \in \Lambda^*. 1 / (z - \omega)^2 - 1 / \omega^2))$)"

This is now the Weierstraß function. Again, it returns 0 at all its poles.

definition *weierstrass_fun* :: "complex \Rightarrow complex" (" \wp ")
where " $\wp z =$ (if *z* $\in \Lambda$ then 0 else $1 / z^2 + \text{weierstrass_fun_aux } z$)"

lemma *weierstrass_fun_aux_0* [*simp*]: "*weierstrass_fun_aux* 0 = 0"
<proof>

lemma *weierstrass_fun_at_pole*: " $\omega \in \Lambda \implies \wp \omega = 0$ "
<proof>

lemma

```

fixes R :: real
assumes "R > 0"
shows weierstrass_fun_aux_converges_absolutely_uniformly_in_disk:
  "uniform_limit (cball 0 R)
    ( $\lambda X z. \sum_{\omega \in X}. \text{norm } (1 / (z - \omega)^2 - 1 / \omega^2)$ )
    ( $\lambda z. \sum_{\infty \omega \in \Lambda - \text{cball } 0 R}. \text{norm } (1 / (z - \omega)^2 -$ 
1 /  $\omega^2)$ )
    (finite_subsets_at_top ( $\Lambda - \text{cball } 0 R$ ))" (is
?th1)
and weierstrass_fun_aux_converges_uniformly_in_disk:
  "uniform_limit (cball 0 R)
    ( $\lambda X z. \sum_{\omega \in X}. 1 / (z - \omega)^2 - 1 / \omega^2$ )
    ( $\lambda z. \sum_{\infty \omega \in \Lambda - \text{cball } 0 R}. 1 / (z - \omega)^2 - 1 / \omega^2$ )
    (finite_subsets_at_top ( $\Lambda - \text{cball } 0 R$ ))" (is
?th2)
<proof>

lemma weierstrass_fun_has_field_derivative_aux:
  fixes R :: real
  defines "F  $\equiv (\lambda z. \sum_{\infty \omega \in \Lambda - \text{cball } 0 R}. 1 / (z - \omega)^2 - 1 / \omega^2)$ "
  defines "F'  $\equiv (\lambda z. \sum_{\infty \omega \in \Lambda - \text{cball } 0 R}. 1 / (z - \omega) ^ 3)$ "
  assumes "R > 0" "w  $\in \text{ball } 0 R$ "
  shows "(F has_field_derivative -2 * F' w) (at w)"
<proof>

lemma norm_summable_weierstrass_fun_aux: " $(\lambda \omega. \text{norm } (1 / (z - \omega)^2 -$ 
1 /  $\omega^2))$  summable_on  $\Lambda$ "
<proof>

lemma summable_weierstrass_fun_aux: " $(\lambda \omega. 1 / (z - \omega)^2 - 1 / \omega^2)$  summable_on
 $\Lambda$ "
<proof>

lemma weierstrass_summable: " $(\lambda \omega. 1 / (z - \omega)^2 - 1 / \omega^2)$  summable_on
 $\Lambda^*$ "
<proof>

lemma weierstrass_fun_aux_has_sum:
  " $z \notin \Lambda^* \implies ((\lambda \omega. 1 / (z - \omega)^2 - 1 / \omega^2)$  has_sum weierstrass_fun_aux
z)  $\Lambda^*$ "
<proof>

lemma weierstrass_fun_aux_has_field_derivative:
  defines "F  $\equiv \text{weierstrass\_fun\_aux}$ "
  defines "F'  $\equiv (\lambda z. \sum_{\infty \omega \in \Lambda^*}. 1 / (z - \omega) ^ 3)$ "
  assumes z: " $z \notin \Lambda^*$ "
  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
- $\Lambda^*$ "
  <proof>

lemma weierstrass_fun_aux_holomorphic' [holomorphic_intros]:
  assumes "f holomorphic_on A" " $\bigwedge z. z \in A \implies f z \notin \Lambda^*$ "
  shows "( $\lambda z. \text{weierstrass\_fun\_aux } (f z)$ ) holomorphic_on A"
  <proof>

lemma weierstrass_fun_aux_continuous_on: "continuous_on (- $\Lambda^*$ ) weierstrass_fun_aux"
  <proof>

lemma weierstrass_fun_aux_continuous_on' [continuous_intros]:
  assumes "continuous_on A f" " $\bigwedge z. z \in A \implies f z \notin \Lambda^*$ "
  shows "continuous_on A ( $\lambda z. \text{weierstrass\_fun\_aux } (f z)$ )"
  <proof>

lemma weierstrass_fun_aux_analytic: "weierstrass_fun_aux analytic_on
- $\Lambda^*$ "
  <proof>

lemma weierstrass_fun_aux_analytic' [analytic_intros]:
  assumes "f analytic_on A" " $\bigwedge z. z \in A \implies f z \notin \Lambda^*$ "
  shows "( $\lambda z. \text{weierstrass\_fun\_aux } (f z)$ ) analytic_on A"
  <proof>

lemma deriv_weierstrass_fun_aux:
  " $z \notin \Lambda^* \implies \text{deriv weierstrass\_fun\_aux } z = -2 * \text{eisenstein\_fun\_aux } 3$ 
z"
  <proof>

lemma weierstrass_fun_has_field_derivative:
  fixes R :: real
  assumes z: " $z \notin \Lambda$ "
  shows "( $\wp$  has_field_derivative  $\wp'$  z) (at z)"
  <proof>

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

lemma weierstrass_fun_holomorphic: " $\wp$  holomorphic_on - $\Lambda$ "
  <proof>

lemma weierstrass_fun_holomorphic' [holomorphic_intros]:

```

assumes "f holomorphic_on A" " $\bigwedge z. z \in A \implies f z \notin \Lambda$ "
shows " $(\lambda z. \text{weierstrass_fun } (f z)) \text{ holomorphic_on } A$ "
 <proof>

lemma weierstrass_fun_analytic: " \wp analytic_on $-\Lambda$ "
 <proof>

lemma weierstrass_fun_analytic' [analytic_intros]:
assumes "f analytic_on A" " $\bigwedge z. z \in A \implies f z \notin \Lambda$ "
shows " $(\lambda z. \wp (f z)) \text{ analytic_on } A$ "
 <proof>

lemma weierstrass_fun_continuous_on: "continuous_on $(-\Lambda)$ weierstrass_fun"
 <proof>

lemma weierstrass_fun_continuous_on' [continuous_intros]:
assumes "continuous_on A f" " $\bigwedge z. z \in A \implies f z \notin \Lambda$ "
shows "continuous_on A $(\lambda z. \wp (f z))$ "
 <proof>

lemma tendsto_weierstrass_fun [tendsto_intros]:
assumes " $(f \longrightarrow c) F$ " " $c \notin \Lambda$ "
shows " $((\lambda z. \wp (f z)) \longrightarrow \wp c) F$ "
 <proof>

lemma deriv_weierstrass_fun:
assumes " $z \notin \Lambda$ "
shows " $\text{deriv } \wp z = \wp' z$ "
 <proof>

The following identity is to be read with care: for $\omega = 0$ we get a division by zero, so the term $1 / \omega^2$ simply gets dropped.

lemma weierstrass_fun_eq:
assumes " $z \notin \Lambda$ "
shows " $\wp z = (\sum_{\omega \in \Lambda} (1 / (z - \omega)^2) - 1 / \omega^2)$ "
 <proof>

7.3 Ellipticity and poles

It can easily be seen from its definition that \wp is an even elliptic function with a double pole at each lattice point and no other poles. Thus it has order 2.

Its derivative is consequently an odd elliptic function with a triple pole at each lattice point, no other poles, and order 3.

The results in this section correspond to Apostol's Theorems 1.9 and 1.10.

lemma weierstrass_fun_minus: " $\wp (-z) = \wp z$ "
 <proof>

sublocale *weierstrass_fun*: *complex_lattice_periodic* ω_1 ω_2 \wp
<proof>

lemma *zorder_weierstrass_fun_pole*:
 assumes " $\omega \in \Lambda$ "
 shows "*zorder* \wp $\omega = -2$ "
<proof>

lemma *is_pole_weierstrass_fun*:
 assumes ω : " $\omega \in \Lambda$ "
 shows "*is_pole* \wp ω "
<proof>

sublocale *weierstrass_fun*: *nicely_elliptic_function* ω_1 ω_2 \wp
<proof>

sublocale *weierstrass_fun*: *even_elliptic_function* ω_1 ω_2 \wp
<proof>

lemmas [*elliptic_function_intros*] =
 weierstrass_fun.elliptic_function_axioms
 weierstrass_fun.nicely_elliptic_function_axioms

lemma *is_pole_weierstrass_fun_iff*: "*is_pole* \wp $z \iff z \in \Lambda$ "
<proof>

lemma *is_pole_weierstrass_fun_deriv_iff*: "*is_pole* \wp' $z \iff z \in \Lambda$ "
<proof>

lemma *zorder_weierstrass_fun_deriv_pole*:
 assumes " $z \in \Lambda$ "
 shows "*zorder* \wp' $z = -3$ "
<proof>

lemma *order_weierstrass_fun [simp]*: "*elliptic_order* $\wp = 2$ "
<proof>

lemma *order_weierstrass_fun_deriv [simp]*: "*elliptic_order* $\wp' = 3$ "
<proof>

sublocale *weierstrass_fun*: *nonconst_nicely_elliptic_function* ω_1 ω_2 \wp
<proof>

sublocale *weierstrass_fun_deriv*: *nonconst_nicely_elliptic_function* ω_1
 ω_2 " \wp' "
<proof>

7.4 The numbers e_1, e_2, e_3

The values of \wp at the half-periods $\frac{1}{2}\omega_1$, $\frac{1}{2}\omega_2$, and $\frac{1}{2}(\omega_1 + \omega_2)$ are exactly the roots of the polynomial $4X^3 - g_2X - g_3$.

We call these values e_1, e_2, e_3 .

definition *number_e1*:: "complex" ("e₁") where
 $e_1 \equiv \wp (\omega_1 / 2)$ "

definition *number_e2*:: "complex" ("e₂") where
 $e_2 \equiv \wp (\omega_2 / 2)$ "

definition *number_e3*:: "complex" ("e₃") where
 $e_3 \equiv \wp ((\omega_1 + \omega_2) / 2)$ "

lemmas *number_e123_defs* = *number_e1_def* *number_e2_def* *number_e3_def*

definition *modulus* :: complex where
 $modulus = (e_3 - e_2) / (e_1 - e_2)$ "

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 " $\wp (\omega / 2) \in \{e_1, e_2, e_3\}$ "
<proof>

lemma *weierstrass_fun_at_half_period'*:
assumes " $2 * \omega \in \Lambda$ " " $\omega \notin \Lambda$ "
shows " $\wp \ \omega \in \{e_1, e_2, e_3\}$ "

<proof>

\wp' has a simple zero at each half-lattice point, and no other zeros.

lemma *weierstrass_fun_deriv_eq_0_iff:*

assumes " $z \notin \Lambda$ "

shows " $\wp' z = 0 \iff 2 * z \in \Lambda$ "

<proof>

lemma *zorder_weierstrass_fun_deriv_zero:*

assumes " $z \notin \Lambda$ " " $2 * z \in \Lambda$ "

shows " $\text{zorder } \wp' z = 1$ "

<proof>

end

7.5 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 " $u \notin \Lambda$ " " $v \notin \Lambda$ "

shows " $\wp u = \wp v \iff \text{rel } u v \vee \text{rel } u (-v)$ "

<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** " $z \in \text{period_parallelogram } w - \Lambda$ " " $\wp z = c$ "

<proof>

lemma *surj_weierstrass_fun_deriv:*

obtains z **where** " $z \in \text{period_parallelogram } w - \Lambda$ " " $\wp' z = c$ "

<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

```

7.6 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  $\omega_1$ " "cnj  $\omega_2$ "
    <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>

lemma modulus_cnj [simp]: "cnj.modulus = cnj modulus"
  <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>

lemma in_stretch_lattice_iff: "z ∈ stretched.lattice ↔ z / c ∈ lattice"
  <proof>

```

lemma `in_stretch_lattice0_iff`: " $z \in \text{stretched.lattice0} \iff z / c \in \text{lattice0}$ "
 ⟨*proof*⟩

lemma `weierstrass_fun_aux_stretch`: " $\text{stretched.weierstrass_fun_aux } z = \text{weierstrass_fun_aux } (z / c) / c^2$ "
 ⟨*proof*⟩

lemma `weierstrass_fun_stretch`: " $\text{stretched.weierstrass_fun } z = \text{weierstrass_fun } (z / c) / c^2$ "
 ⟨*proof*⟩

lemma `number_e1_stretch [simp]`: " $\text{stretched.number_e1} = \text{number_e1} / c^2$ "
 ⟨*proof*⟩

lemma `number_e2_stretch [simp]`: " $\text{stretched.number_e2} = \text{number_e2} / c^2$ "
 ⟨*proof*⟩

lemma `number_e3_stretch [simp]`: " $\text{stretched.number_e3} = \text{number_e3} / c^2$ "
 ⟨*proof*⟩

lemma `modulus_stretch [simp]`: " $\text{stretched.modulus} = \text{modulus}$ "
 ⟨*proof*⟩

end

locale `unimodular_moebius_transform_lattice = complex_lattice + unimodular_moebius_transform`
begin

definition `ω_1'` where " $\omega_1' = \text{of_int } c * \omega_2 + \text{of_int } d * \omega_1$ "

definition `ω_2'` where " $\omega_2' = \text{of_int } a * \omega_2 + \text{of_int } b * \omega_1$ "

sublocale `transformed: complex_lattice ω_1' ω_2'`
 ⟨*proof*⟩

lemma `transformed_lattice_subset`: " $\text{transformed.lattice} \subseteq \text{lattice}$ "
 ⟨*proof*⟩

lemma `transformed_lattice_eq`: " $\text{transformed.lattice} = \text{lattice}$ "
 ⟨*proof*⟩

lemma `transformed_lattice0_eq`: " $\text{transformed.lattice0} = \text{lattice0}$ "
 ⟨*proof*⟩

lemma `eisenstein_fun_aux_transformed [simp]`: " $\text{transformed.eisenstein_fun_aux}$

```

= 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

```

7.7 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. φ z - φ w)"
begin

sublocale elliptic_function_affine ω1 ω2 φ 1 "-φ 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  $\vee$  rel z (-w)"
  shows "zorder f z = (if 2 * w  $\in$   $\Lambda$  then 2 else 1)"
  <proof>

lemma zorder_zero_eq':
  assumes "z  $\notin$   $\Lambda$ "
  shows "zorder f z = (if rel z w  $\vee$  rel z (-w) then if 2 * w  $\in$   $\Lambda$  then
2 else 1 else 0)"
  <proof>

end

```

```

lemma (in complex_lattice) zorder_weierstrass_fun_minus_const:
  assumes "w  $\notin$   $\Lambda$ " "z  $\notin$   $\Lambda$ "
  shows "zorder ( $\lambda$ z.  $\wp$  z -  $\wp$  w) z =
      (if rel z w  $\vee$  rel z (-w) then if 2 * w  $\in$   $\Lambda$  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 $h : A \rightarrow \mathbb{Z}$.

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  $\Rightarrow$  int" and g :: "complex
 $\Rightarrow$  complex"
  assumes finite [intro]: "finite A" and no_lattice_points: "A  $\cap$   $\Lambda$  =
  {}"
  defines "g  $\equiv$  ( $\lambda$ z. ( $\prod_{w \in A}$ . ( $\wp$  z -  $\wp$  w) powi h w))"
begin

```

```

sublocale elliptic_function  $\omega$ 1  $\omega$ 2 g
  <proof>

```

```

sublocale even_elliptic_function  $\omega$ 1  $\omega$ 2 g
  <proof>

```

```

lemma no_lattice_points': "w  $\notin$   $\Lambda$ " if "w  $\in$  A" for w
  <proof>

```

lemma eq_0_iff: "g z = 0 \longleftrightarrow ($\exists w \in A. h w \neq 0 \wedge (\text{rel } z w \vee \text{rel } z (-w))$)"
if "z $\notin \Lambda$ " **for** z
 <proof>

lemma nonzero_almost_everywhere: "eventually ($\lambda z. g z \neq 0$) (cosparse UNIV)"
 <proof>

lemma eventually_nonzero_at: "eventually ($\lambda z. g z \neq 0$) (at z)"
 <proof>

lemma zorder_eq:
assumes z: "z $\notin \Lambda$ "
shows "zorder g z =
 ($\sum w \in A. \text{if } \text{rel } z w \vee \text{rel } z (-w) \text{ then if } 2 * w \in \Lambda \text{ then } 2 * h w \text{ else } h w \text{ else } 0$)"
 <proof>

end

lemma (in even_elliptic_function) in_terms_of_weierstrass_fun_even_aux:
assumes nontrivial: " \neg eventually ($\lambda z. f z = 0$) (cosparse UNIV)"
defines "Z $\equiv \{z \in \text{half_fund_parallelogram} - \{0\}. \text{is_pole } f z \vee \text{isolated_zero } f z\}$ "
defines "h $\equiv (\lambda z. \text{zorder } f z \text{ div } (\text{if } 2 * z \in \Lambda \text{ then } 2 \text{ else } 1))"$
obtains c **where** "eventually ($\lambda z. f z = c * (\prod w \in Z. (\wp z - \wp w)^{\text{powi } h w})$) (cosparse UNIV)"
 <proof>

Finally, we show that any even elliptic function can be written as a rational function of \wp . This is Exercise 1.4 in Apostol's book.

lemma (in even_elliptic_function) in_terms_of_weierstrass_fun_even:
obtains p q :: "complex poly" **where** "q $\neq 0$ " " $\forall z. f z = \text{poly } p (\wp z) / \text{poly } q (\wp z)$ "
 <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 elliptic_function) in_terms_of_weierstrass_fun:

```

    obtains p q r s :: "complex poly" where "q ≠ 0" "s ≠ 0"
      "∀ ≈z. f z = poly p (φ z) / poly q (φ z) + φ' z * poly r (φ z) /
poly s (φ z)"
⟨proof⟩

end

```

8 Related facts about Jacobi theta functions

```
theory Theta_Inversion
```

```
imports
```

```
  "Theta_Functions.Jacobi_Triple_Product"
```

```
  "Theta_Functions.Theta_Nullwert"
```

```
  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.

8.1 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 $\text{Im}(t) > 0$.

We will show that any such function is equal to ϑ_{00} up to a constant factor.

```

locale thetalike_function =
  fixes f :: "complex  $\Rightarrow$  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 "( $\lambda$ x. f (g x)) holomorphic_on A"
  <proof>

lemma analytic:
  assumes "g analytic_on A"
  shows "( $\lambda$ x. 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 (n2)
  * 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 (n2)
  * 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  $\longleftrightarrow$  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

$$\int_{\gamma} \frac{h(z)f'(z)}{f(z)} dz$$

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:

$$\int_{\gamma} \frac{h(z)f'(z)}{f(z)} dz = \int_b^{b+1} (h(z)-h(z+t)) \frac{f'(z)}{f(z)} + 2\pi i h(z+t) dz - \int_b^{b+t} (h(z)-h(z+1)) \frac{f'(z)}{f(z)} dz$$

```

lemma argument_principle_f_gen:
  fixes orig :: complex
  defines "\gamma \equiv parallelogram_path orig 1 t"
  assumes h: "h holomorphic_on UNIV"
  assumes nz: "\z. z \in path_image \gamma \implies f z \neq 0"
  shows "contour_integral \gamma (\lambda x. h x * deriv f x / f x) =
        contour_integral (linepath orig (orig + 1))
          (\lambda z. (h z - h (z + t)) * deriv f z / f z + 2 * pi * i * h
(z + t)) -
        contour_integral (linepath orig (orig + t))
          (\lambda z. (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 " $\gamma \equiv \text{parallelogram\_path orig 1 t}$ "
  assumes nz: " $\bigwedge z. z \in \text{path\_image } \gamma \implies f z \neq 0$ "
  shows " $\text{contour\_integral } \gamma (\lambda z. \text{deriv } f z / f z) = 2 * \text{pi} * i$ "
  <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 " $\gamma \equiv \text{parallelogram\_path orig 1 t}$ "
  assumes nz: " $\bigwedge z. z \in \text{path\_image } \gamma \implies f z \neq 0$ "
  shows " $\text{lattice.rel } (\text{contour\_integral } \gamma (\lambda z. z * \text{deriv } f z / f z) / (2 * \text{pi} * i)) ((t+1)/2)$ "
  <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 " $\neg (\forall z. f z = 0)$ "
  shows " $f z = 0 \iff \text{lattice.rel } z ((t + 1) / 2)$ "
  and " $\text{lattice.rel } z ((t + 1) / 2) \implies \text{zorder } f z = 1$ "
  <proof>

```

Finally, we conclude that our quasi-periodic function is in fact a multiple of ϑ_{00} .

```

theorem multiple_jacobi_theta_00: " $\exists c. \forall z. f z = c * \text{jacobi\_theta\_00 } z t$ "
  <proof>

```

end

As a side effect, we also now know that the zeros of ϑ_{00} are all simple zeros.

```

lemma jacobi_theta_00_simple_zero:
  assumes " $\text{Im } t > 0$ " " $\text{jacobi\_theta\_00 } z t = 0$ "
  shows " $\text{zorder } (\lambda z. \text{jacobi\_theta\_00 } z t) z = 1$ "
  <proof>

```

8.2 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 $t \mapsto -\frac{1}{t}$:

$$\vartheta_{00}(z, -1/t) = \sqrt{-ite^{i\pi tz^2}} \vartheta_{00}(tz, t)$$

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⟩

```

8.3 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

$$f : [0, 1] \rightarrow [0, 1], \quad f(q) = \exp(\pi^2 / \log q)$$

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 \Rightarrow real" where
 "q_inversion q = (if q = 0 then 1 else if q = 1 then 0 else exp (pi² / 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 \in {0..1} \implies q_inversion q \geq 0"
 and `q_inversion_le_1`: "q \in {0..1} \implies q_inversion q \leq 1"
 and `q_inversion_pos`: "q \in {0.. $<$ 1} \implies q_inversion q $>$ 0"
 and `q_inversion_less_1`: "q \in {0<..1} \implies 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 \in {0..1}" "q' \in {0..1}"
 shows "q_inversion q $<$ q_inversion q' \longleftrightarrow q $>$ q'"
 <proof>

lemma `q_inversion_le_iff`:
 assumes "q \in {0..1}" "q' \in {0..1}"
 shows "q_inversion q \leq q_inversion q' \longleftrightarrow q \geq q'"
 <proof>

lemma `q_inversion_eq_iff`:
 assumes "q \in {0..1}" "q' \in {0..1}"
 shows "q_inversion q = q_inversion q' \longleftrightarrow q = q'"
 <proof>

lemma `q_inversion_involution`:
 assumes "q \in {0..1}"
 shows "q_inversion (q_inversion q) = q"
 <proof>

lemma `continuous_q_inversion` [continuous_intros]:
 assumes q: "q \in {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 \in A \implies f x \in {0..1}"
  shows "continuous_on A (\x. q_inversion (f x))"
  <proof>

```

```

definition q_inversion_fixedpoint :: real where
  "q_inversion_fixedpoint = exp (-pi)"

```

```

lemma q_inversion_fixedpoint:
  defines "q0 \equiv q_inversion_fixedpoint"
  shows "q0 \in {0..1}" "q_inversion q0 = q0"
  <proof>

```

```

lemma q_inversion_less_self_iff:
  assumes "q \in {0..1}"
  shows "q_inversion q < q \longleftrightarrow q > q_inversion_fixedpoint"
  <proof>

```

```

lemma q_inversion_greater_self_iff:
  assumes "q \in {0..1}"
  shows "q_inversion q > q \longleftrightarrow 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 \in {0<..<1}" and w: "w > 0"
  shows "jacobi_theta_nome (of_real w) (of_real q) =
    complex_of_real (sqrt (- pi / ln q) * exp (- (ln w)^2 / (4 *
ln q))) *
    jacobi_theta_nome (cis (-pi * ln w / ln q)) (of_real (exp (pi^2
/ ln q)))"
  <proof>

```

```

lemma jacobi_theta_nome_1_left_inversion_real:
  assumes q: "q \in {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 $q: "q \in \{0..<1::real\}"$

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 $q: "q \in \{0..<1::real\}"$

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 $q: "q \in \{0..<1::real\}"$

shows $"jacobi_theta_nw_10 (q_inversion\ q) = \sqrt{-\ln\ q / \pi} * jacobi_theta_nw_01\ q"$

<proof>

end

9 Eisenstein series and the differential equations of \wp

theory *Eisenstein_Series*

imports

Weierstrass_Elliptic

"Elliptic_Functions.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"

Theta_Inversion

begin

unbundle *jacobi_theta_notation*

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.

9.1 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 \Rightarrow complex" **where**
 "eisenstein_series k = (if k = 0 then -1 else if odd k then 0 else
 2 / ω_1 ^ k * zeta (of_nat k) + ($\sum_{\infty n \in -\{0\}}$. $\sum_{\infty m}$. 1 / of_omega12_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 \implies eisenstein_series
 k = 0"
 <proof>

lemma *eisenstein_series_Suc_0* [simp]: "eisenstein_series (Suc 0) = 0"
 <proof>

lemma *eisenstein_series_norm_summable*:
 assumes "n \geq 3"
 shows "($\lambda \omega$. 1 / norm ω ^ n) summable_on Λ^* "
 <proof>

lemma *eisenstein_series_summable*:
 assumes "n \geq 3"
 shows "($\lambda \omega$. 1 / ω ^ n) summable_on Λ^* "
 <proof>

lemma *eisenstein_series_has_sum*:
 assumes "k \geq 3"
 shows "(($\lambda \omega$. 1 / ω ^ k) has_sum eisenstein_series k) Λ^* "
 <proof>

lemma *eisenstein_series_altdef*:
 assumes "k \geq 3"

shows "eisenstein_series k = $(\sum_{\omega \in \Lambda^*} 1 / \omega^k)$ "
 ⟨proof⟩

lemma eisenstein_fun_aux_0 [simp]:
assumes "n ≠ 2"
shows "eisenstein_fun_aux n 0 = eisenstein_series n"
 ⟨proof⟩

9.2 The Laurent series expansion of \wp at the origin

lemma higher_deriv_weierstrass_fun_aux_0:
assumes "m > 0"
shows " $(\text{deriv}^m \text{weierstrass_fun_aux } 0) = (-1)^m * \text{fact } (\text{Suc } m) * G(m + 2)$ "
 ⟨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 ($\lambda n. \text{if } n = 0 \text{ then } 0 \text{ else of_nat } (\text{Suc } n) * G(n + 2)$)"

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]*:
 "*φ has_laurent_expansion fls_weierstrass*"
 ⟨*proof*⟩

corollary *fls_weierstrass_deriv_laurent_expansion [laurent_expansion_intros]*:
 "*φ' 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*⟩

9.3 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 "*φ' z ^ 2 = 4 * φ z ^ 3 - 60 * G 4 * φ 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 \equiv fls_weierstrass"

shows "fls_deriv (fls_deriv P) = 6 * P ^ 2 - fls_const (30 * G 4)"

<proof>

theorem *weierstrass_fun_ODE2*:

assumes "z \notin Λ "

shows "deriv \wp' z = 6 * \wp 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 \notin Λ "

shows "((λz . \wp' (f z)) has_field_derivative ((6 * \wp (f z) ^ 2 - 30 * G 4) * f')) (at z within A)"

<proof>

9.4 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" (" g_2 ") where

" $g_2 \equiv 60 * \text{eisenstein_series } 4$ "

definition *invariant_g3*:: "complex" (" g_3 ") where

" $g_3 \equiv 140 * \text{eisenstein_series } 6$ "

lemma *weierstrass_fun_ODE1'*:

assumes "z \notin Λ "

shows " \wp' z ^ 2 = 4 * \wp z ^ 3 - $g_2 * \wp$ z - g_3 "

<proof>

This is the ODE obtained by differentiating the first ODE.

theorem *weierstrass_fun_ODE2'*:

assumes "z \notin Λ "

shows "deriv \wp' z = 6 * \wp z ^ 2 - $g_2 / 2$ "

<proof>

lemma *half_period_weierstrass_fun_is_root*:

assumes " $\omega \in \Lambda$ " " $\omega / 2 \notin \Lambda$ "

defines "z $\equiv \wp$ ($\omega / 2$)"

shows "4 * z ^ 3 - $g_2 * z$ - $g_3 = 0$ "

<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 [e₁, e₂, e₃]"
 ⟨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 = g₂ ^ 3 - 27 * g₃ ^ 2"

definition *invariant_J* :: complex where
 "invariant_J = g₂ ^ 3 / discr"

theorem
fixes z :: "complex"
defines "P ≡ [:-g₃, -g₂, 0, 4:]"
shows *discr_nonzero_aux1*: "P = 4 * [:-e₁, 1:] * [:-e₂, 1:] * [:-e₃, 1:]"
and *discr_nonzero_aux2*: "4 * (ϕ z)^3 - g₂ * (ϕ z) - g₃ = 4 * (ϕ z - e₁) * (ϕ z - e₂) * (ϕ z - e₃)"
and *discr_nonzero*: "discr ≠ 0"
and *discr_altdef*: "discr = (4 * (e₁ - e₂) * (e₁ - e₃) * (e₂ - e₃)) ^ 2"
and *invariant_g3_conv_e123*: "g₃ = 4 * e₁ * e₂ * e₃"
and *invariant_g2_conv_e123*: "g₂ = -4 * (e₁ * e₂ + e₁ * e₃ + e₂ * e₃)"
and *sum_e123_0*: "e₁ + e₂ + e₃ = 0"
 ⟨proof⟩

corollary *modulus_neq_0*: "modulus ≠ 0" and *modulus_neq_1*: "modulus ≠ 1"
 ⟨proof⟩

The J invariant can be expressed as a rational function of the modulus.

```

corollary invariant_J_conv_modulus:
  defines "x ≡ modulus * (1 - modulus)"
  shows "invariant_J = 4 / 27 * (1 - x) ^ 3 / x ^ 2"
  ⟨proof⟩

end

```

```

lemma (in complex_lattice_swap) modulus_swap: "swap.modulus = 1 - modulus"
  ⟨proof⟩

```

```

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

```

9.5 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

$$f_{n+1}(z) = \frac{(2i\pi)^{n+1}}{n!} \text{Li}_{-n}(q)$$

where $q = e^{2i\pi z}$ and Li_{-n} denotes the Polylogarithm function.

definition `eisenstein_fourier_aux` :: "nat \Rightarrow complex \Rightarrow 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 \geq 2"`
`shows "summable (λ k. norm (1 / (z + of_nat k :: complex) ^ n))"`
`<proof>`

lemma `abs_summable_one_over_const_minus_nat_power`:
`assumes "n \geq 2"`
`shows "summable (λ k. norm (1 / (z - of_nat k :: complex) ^ n))"`
`<proof>`

lemma `has_sum_eisenstein_fourier_aux`:
`assumes "n \geq 2" and "even n" and "Im z > 0"`
`shows "((λ m. 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 $\text{Im}(\tau) > 0$ in terms of a Lambert series:

$$G_k = 2\left(\zeta(k) + \frac{(2i\pi)^k}{(k-1)!}L(n^{k-1}, q)\right)$$

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 \geq 2" "even k"`
`defines "x \equiv to_q 1 τ "`
`shows "eisenstein_series k =`
`2 * (zeta k + (2 * i * pi) ^ k / fact (k - 1) * lambert (λ n.`
`of_nat n ^ (k-1)) x)"`
`<proof>`

9.6 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

9.7 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⟩

lemma coeff_0_0_eisenstein_series_poly [simp]:
  "poly.coeff (poly.coeff (eisenstein_series_poly n) 0) 0 = 0"

```

<proof>

definition *coeff2* where "coeff2 p m n = poly.coeff (poly.coeff p n) m"

The polynomial $P(X, Y)$ that gives us $G_{2n+4} = P(G_4, G_6)$ only has monomials of the form $X^i Y^j$ with $4i + 6j = 2n + 4$.

lemma *support_eisenstein_series_poly*:

assumes "coeff2 (eisenstein_series_poly n) i j \neq 0"

shows "4 * i + 6 * j = 2 * n + 4"

<proof>

We now show that the polynomial also gives us the right answer.

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 \Rightarrow 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)):]
($\sum_{i \leq n-2}$. Polynomial.smult (of_nat ((2*i+3)*(2*(n-i)-1)))

(ps ! i * ps ! (n-2-i)))]])"

definition *eisenstein_series_polys* :: "nat \Rightarrow 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 \geq 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 \implies (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

```

10 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:

$$\wp(w + z) = -\wp(w) - \wp(z) + \frac{1}{4} \left(\frac{\wp'(w) - \wp'(z)}{\wp(w) - \wp(z)} \right)^2$$

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*: " $u_1 \notin \Lambda$ " " $u_2 \notin \Lambda$ " " $\neg \text{rel } u_1 \ u_2$ " " $\neg \text{rel } u_1 \ (-u_2)$ "
 assumes *general_position*: " $u_1 + 2 * u_2 \notin \Lambda$ " " $2 * u_1 + u_2 \notin \Lambda$ "
 shows " $\wp(u_1 + u_2) = -\wp u_1 - \wp u_2 + ((\wp' u_1 - \wp' u_2) / (\wp u_1 - \wp u_2))^2 / 4$ "
 and " $\wp'(u_1 + u_2) = -\wp' u_1 + (\wp u_1 - \wp(u_1 + u_2)) * (\wp' u_1 - \wp' u_2) / (\wp u_1 - \wp u_2)$ "
<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*: " $u_1 \notin \Lambda$ " " $u_2 \notin \Lambda$ " " $\neg \text{rel } u_1 \ u_2$ " " $\neg \text{rel } u_1 \ (-u_2)$ "
 shows " $\wp(u_1 + u_2) = -\wp u_1 - \wp u_2 + ((\wp' u_1 - \wp' u_2) / (\wp u_1 - \wp u_2))^2 / 4$ "
 (is " $?lhs \ u_2 = ?rhs \ u_2$ ")
<proof>

lemma *weierstrass_fun_diff*:
 assumes *u12*: " $u_1 \notin \Lambda$ " " $u_2 \notin \Lambda$ " " $\neg \text{rel } u_1 \ u_2$ " " $\neg \text{rel } u_1 \ (-u_2)$ "
 shows " $\wp(u_1 - u_2) = -\wp u_1 - \wp u_2 + ((\wp' u_1 + \wp' u_2) / (\wp u_1 - \wp u_2))^2 / 4$ "
<proof>

Using the addition theorem for $\wp(z + w)$ and letting $w \rightarrow z$ 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*: " $2 * z \notin \Lambda$ "
 shows " $\wp(2 * z) = -2 * \wp z + (\text{deriv } \wp' \ z / \wp' \ z)^2 / 4$ "
<proof>

theorem *weierstrass_fun_deriv_add*:
 assumes *u12*: " $u_1 \notin \Lambda$ " " $u_2 \notin \Lambda$ " " $\neg \text{rel } u_1 \ u_2$ " " $\neg \text{rel } u_1 \ (-u_2)$ "
 shows " $\wp'(u_1 + u_2) = -\wp' u_1 + (\wp u_1 - \wp(u_1 + u_2)) * (\wp' u_1 - \wp' u_2) / (\wp u_1 - \wp u_2)$ "
 (is " $?lhs \ u_2 = ?rhs \ u_2$ ")
<proof>

theorem *weierstrass_fun_deriv_double*:

```

    assumes z: "2 * z ∉ Λ"
    defines "a ≡ deriv ϕ' z / ϕ' z"
    shows "ϕ' (2 * z) = -ϕ' z + 3 * a * ϕ z - a ^ 3 / 4"
  <proof>

end

end
<proof><proof><proof><proof><proof><proof><proof><proof><proof>
theory FPS_Homomorphism
  imports "HOL-Computational_Algebra.Formal_Laurent_Series" "Polynomial_Interpolation.Ring_
begin

interpretation fps_to_fls: comm_ring_hom "fps_to_fls :: 'a :: comm_ring_1
fps ⇒ 'a fls"
  <proof>

interpretation fps_to_fls: inj_idom_hom "fps_to_fls :: 'a :: idom fps ⇒
'a fls"
  <proof>

interpretation fps_const: comm_ring_hom "fps_const :: 'a :: comm_ring_1
⇒ 'a fps"
  <proof>

interpretation fps_const: inj_idom_hom "fps_const :: 'a :: idom ⇒ 'a fps"
  <proof>

interpretation fls_const: comm_ring_hom "fls_const :: 'a :: comm_ring_1
⇒ 'a fls"
  <proof>

interpretation fls_const: inj_idom_hom "fls_const :: 'a :: idom ⇒ 'a fls"
  <proof>

interpretation fls_const: field_hom "fls_const :: 'a :: field ⇒ 'a fls"
  <proof>

interpretation fls_const: field_char_0_hom "fls_const :: 'a :: field_char_0
⇒ 'a fls"
  <proof>

locale fps_homomorphism =
  fixes f :: "'a :: comm_semiring_1 ⇒ 'b :: comm_semiring_1"
  assumes f_0: "f 0 = 0"
  assumes f_1: "f 1 = 1"
  assumes f_add: "f (x + y) = f x + f y"
  assumes f_mult: "f (x * y) = f x * f y"

```

begin

lemma *f_sum*: "f ($\sum x \in A. g x$) = ($\sum x \in A. f (g x)$)"
<proof>

lemma *f_prod*: "f ($\prod x \in A. g x$) = ($\prod x \in A. f (g x)$)"
<proof>

lemma *f_sum_list*: "f (sum_list xs) = sum_list (map f xs)"
<proof>

lemma *f_prod_list*: "f (prod_list xs) = prod_list (map f xs)"
<proof>

lemma *f_power*: "f (x ^ n) = f x ^ n"
<proof>

lemmas *f_simps* = *f_0 f_1 f_add f_mult f_sum f_prod f_power*

definition *fps* :: "'a fps \Rightarrow 'b fps" where
"fps F = Abs_fps ($\lambda n. f (fps_nth F n)$)"

lemma *fps_nth [simp]*: "fps_nth (fps F) n = f (fps_nth F n)"
<proof>

lemma *Abs_fps [simp]*: "fps (Abs_fps g) = Abs_fps ($\lambda n. f (g n)$)"
<proof>

lemma *fps_0 [simp]*: "fps 0 = 0"
and *fps_1 [simp]*: "fps 1 = 1"
and *fps_const [simp]*: "fps (fps_const c) = fps_const (f c)"
and *fps_add [simp]*: "fps (F + G) = fps F + fps G"
and *fps_mult [simp]*: "fps (F * G) = fps F * fps G"
and *fps_shift [simp]*: "fps (fps_shift n F) = fps_shift n (fps F)"
<proof>

lemma *fps_sum*: "fps ($\sum x \in A. g x$) = ($\sum x \in A. fps (g x)$)"
<proof>

lemma *fps_prod*: "fps ($\prod x \in A. g x$) = ($\prod x \in A. fps (g x)$)"
<proof>

lemma *fps_sum_list*: "fps (sum_list xs) = sum_list (map fps xs)"
<proof>

lemma *fps_prod_list*: "fps (prod_list xs) = prod_list (map fps xs)"
<proof>

lemma *fps_power [simp]*: "fps (x ^ n) = fps x ^ n"

```

    <proof>

lemma fps_of_nat [simp]: "fps (of_nat n) = of_nat n"
  <proof>

lemma fps_numeral [simp]: "fps (numeral num) = numeral num"
  <proof>

end

locale fps_homomorphism_ring =
  fps_homomorphism f for f :: "'a :: comm_ring_1 ⇒ 'b :: comm_ring_1"
begin

lemma fps_minus [simp]: "fps (-x) = -fps x"
  <proof>

lemma fps_diff [simp]: "fps (F - G) = fps F - fps G"
  <proof>

lemma fps_of_int [simp]: "fps (of_int m) = of_int m"
  <proof>

end

interpretation of_nat: fps_homomorphism of_nat
  <proof>

interpretation of_nat_fps: comm_semiring_hom "of_nat.fps"
  <proof>

interpretation of_nat_fps: inj_semiring_hom "of_nat.fps :: nat fps ⇒ 'a
:: {comm_semiring_1, semiring_char_0} fps"
  <proof>

interpretation of_int: fps_homomorphism of_int
  <proof>

interpretation of_int: fps_homomorphism_ring of_int
  <proof>

interpretation of_int_fps: comm_ring_hom "of_int.fps"
  <proof>

interpretation of_int_fps: inj_idom_hom "of_int.fps :: int fps ⇒ 'a ::
{idom, ring_char_0} fps"
  <proof>

```

end

11 Connection between complex lattices and theta functions

theory *Complex_Lattices_Theta*

imports *Eisenstein_Series* "*Theta_Functions.Theta_Nullwert*"
begin

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

unbundle *jacobi_theta_nw_notation*

We make the connection to theta functions. In order to do that, we first assume that the generators ω_1 and ω_2 are such that their ratio $\tau := \omega_2/\omega_1$ has positive imaginary part.

locale *complex_lattice_Im_pos* = *complex_lattice* +
assumes *Im_ratio_pos*: "*Im ratio > 0*"
begin

We fix this ratio τ as the second parameter of the theta functions so that the theta functions become quasi-elliptic functions in one variable z .

definition *theta_00* (" \langle notation= \langle mixfix *complex_lattice_Im_pos.theta_00* $\rangle\rangle\vartheta_{00}'$ ($_$)")
where "*theta_00* $z \equiv$ *jacobi_theta_00* (z / ω_1) τ "

definition *theta_01* (" \langle notation= \langle mixfix *complex_lattice_Im_pos.theta_00* $\rangle\rangle\vartheta_{01}'$ ($_$)")
where "*theta_01* $z \equiv$ *jacobi_theta_01* (z / ω_1) τ "

definition *theta_10* (" \langle notation= \langle mixfix *complex_lattice_Im_pos.theta_00* $\rangle\rangle\vartheta_{10}'$ ($_$)")
where "*theta_10* $z \equiv$ *jacobi_theta_10* (z / ω_1) τ "

definition *theta_11* (" \langle notation= \langle mixfix *complex_lattice_Im_pos.theta_00* $\rangle\rangle\vartheta_{11}'$ ($_$)")
where "*theta_11* $z \equiv$ *jacobi_theta_11* (z / ω_1) τ "

lemma *theta_01_conv_00*: "*theta_01* $z =$ *theta_00* ($z + \omega_1 / 2$)"
<proof>

lemma *theta_10_conv_00*: "*theta_10* $z =$ *to_nome* ($z / \omega_1 + \tau / 4$) * *theta_00* ($z + \omega_2 / 2$)"
<proof>

lemma *theta_11_conv_00*:
"*theta_11* $z =$ *to_nome* ($z / \omega_1 + \tau / 4 + 1 / 2$) * *theta_00* ($z + (\omega_1 + \omega_2) / 2$)"
<proof>

The four zeta functions then each have their zeros at various lattice or half-lattice points.

lemma theta_00_eq_0_iff: " $\vartheta_{00}(z) = 0 \iff \text{rel } z ((\omega_1 + \omega_2) / 2)$ " for z
 ⟨proof⟩

lemma theta_01_eq_0_iff: " $\vartheta_{01}(z) = 0 \iff \text{rel } z (\omega_2 / 2)$ "
 ⟨proof⟩

lemma theta_10_eq_0_iff: " $\vartheta_{10}(z) = 0 \iff \text{rel } z (\omega_1 / 2)$ "
 ⟨proof⟩

lemma theta_11_eq_0_iff: " $\vartheta_{11}(z) = 0 \iff z \in \Lambda$ "
 ⟨proof⟩

lemma zorder_theta_00: " $\text{zorder } \theta_{00} ((\omega_1 + \omega_2) / 2) = 1$ "
 ⟨proof⟩

By comparing the zeros of $\wp(z) - e_2$ and $(\vartheta_{01}(z)/\vartheta_{11}(z))^2$ we find that the two functions are identical up to a constant factor, which we then determine to be $(\pi\vartheta_{10}(0)\vartheta_{00}(0)/\omega_1)^2$ by comparing the Laurent series expansions of the two functions at their pole at the origin.

It follows that we can express \wp in terms of the constant e_2 and the theta functions.

lemma weierstrass_fun_conv_theta:
 assumes $z: "z \notin \Lambda"$
 shows " $\wp z = e_2 + (\pi * \vartheta_{10}(0) * \vartheta_{00}(0) / \omega_1)^2 * \vartheta_{01}(z)^2 / \vartheta_{11}(z)^2$ "
 ⟨proof⟩

By plugging in values into the above identity, we derive expressions for e_1 , e_2 , e_3 and the lattice modulus λ purely in terms of theta functions.

lemma e12_conv_theta: " $e_1 - e_2 = (\pi / \omega_1)^2 * \vartheta_{00}(0)^4$ "
and e32_conv_theta: " $e_3 - e_2 = (\pi / \omega_1)^2 * \vartheta_{10}(0)^4$ "
and e13_conv_theta: " $e_1 - e_3 = (\pi / \omega_1)^2 * \vartheta_{01}(0)^4$ "
and e1_conv_theta: " $e_1 = (\pi / \omega_1)^2 / 3 * (\vartheta_{00}(0)^4 + \vartheta_{01}(0)^4)$ "
and e2_conv_theta: " $e_2 = -(\pi / \omega_1)^2 / 3 * (\vartheta_{00}(0)^4 + \vartheta_{10}(0)^4)$ "
and e3_conv_theta: " $e_3 = (\pi / \omega_1)^2 / 3 * (\vartheta_{10}(0)^4 - \vartheta_{01}(0)^4)$ "
and modulus_conv_theta: " $\text{modulus} = \vartheta_{10}(0)^4 / \vartheta_{00}(0)^4$ "
 ⟨proof⟩

Using this, we also obtain an expression of \wp purely in terms of theta functions. This immediately shows that $\wp(z, \tau)$ (which we have not defined yet) is holomorphic in both z and τ .

lemma weierstrass_fun_conv_theta':
 assumes " $z \notin \Lambda$ "

```

shows "∅ z = (pi / ω1)2 * (-1/3 * (∅00(0)4 + ∅10(0)4) + (∅10(0)
* ∅00(0))2 * ∅01(z)2 / ∅11(z)2)"
  ⟨proof⟩

end

context std_complex_lattice
begin

sublocale complex_lattice_Im_pos 1 τ
  rewrites "ratio = τ"
  ⟨proof⟩

end

unbundle no_jacobi_theta_nw_notation

end

```

12 Eisenstein series and related invariants as modular forms

```

theory Basic_Modular_Forms
imports
  Eisenstein_Series Modular_Fundamental_Region
  Elliptic_Functions_Library FPS_Homomorphism
  "Kummer_Congruence.Kummer_Library"
  Complex_Lattices_Theta
begin

```

```

lemma zeta_of_nat_eq_0_iff: "zeta (of_nat n) = 0 ⟷ n = 1"
  ⟨proof⟩

```

```

unbundle modgrp_notation

```

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. $z \rightarrow i\infty$) 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.

12.1 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 \Rightarrow complex \Rightarrow complex" where
`"Eisenstein_G k z = (if z \in \mathbb{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 (ω_2 / ω_1) / ω_1 ^ k"`
`<proof>`

lemma (in `complex_lattice`) `invariant_g2_eq_Eisenstein_G`:
`"invariant_g2 = 60 * Eisenstein_G 4 (ω_2 / ω_1) / ω_1 ^ 4"`
`<proof>`

lemma (in `complex_lattice`) `invariant_g3_eq_Eisenstein_G`:
`"invariant_g3 = 140 * Eisenstein_G 6 (ω_2 / ω_1) / ω_1 ^ 6"`
`<proof>`

lemma `Eisenstein_G_real_eq_0 [simp]`: " $z \in \mathbb{R} \implies \text{Eisenstein_G } k \ z = 0$ "
`<proof>`

lemma `Eisenstein_G_0 [simp]`:
`assumes [simp]: "z \notin \mathbb{R} "`
`shows "Eisenstein_G 0 z = -1"`
`<proof>`

lemma `Eisenstein_G_cnj`: " $\text{Eisenstein_G } k \ (\text{cnj } z) = \text{cnj } (\text{Eisenstein_G } k \ z)$ "
`<proof>`

lemma `Eisenstein_G_odd [simp]`:
`assumes "odd k"`
`shows "Eisenstein_G k z = 0"`
`<proof>`

lemma `Eisenstein_G_uminus`: " $\text{Eisenstein_G } k \ (-z) = \text{Eisenstein_G } k \ z$ "

<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>
```

lemma tendsto_Eisenstein_G [tendsto_intros]:

```
assumes "(f ⟶ z) F" "odd k ∨ z ∉ ℝ"
shows "((λz. Eisenstein_G k (f z)) ⟶ Eisenstein_G k z) F"
<proof>
```

lemma continuous_Eisenstein_G [continuous_intros]:

```
"continuous (at x within A) f ⇒ odd k ∨ f x ∉ ℝ ⇒
continuous (at x within A) (λz. Eisenstein_G k (f z))"
<proof>
```

lemma continuous_on_Eisenstein_G [continuous_intros]:

```
"continuous_on A f ⇒ odd k ∨ (∀x∈A. f x ∉ ℝ) ⇒
continuous_on A (λz. Eisenstein_G k (f z))"
<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 explicitly define the q -expansion of G_n as a holomorphic function on the unit disc.

```

definition q_Eisenstein_G :: "nat ⇒ complex ⇒ complex" where
  "q_Eisenstein_G k q = (if k = 0 then -1 else if odd k then 0 else
    2 * (zeta k + (2*i*pi) ^ k / fact (k - 1) * lambert (λn. of_nat n
^ (k - 1)) q))"

```

```

lemma holomorphic_on_q_Eisenstein_G [holomorphic_intros]:
  "f holomorphic_on A ⇒ (λz. z ∈ A ⇒ norm (f z) < 1) ⇒
    (λz. q_Eisenstein_G k (f z)) holomorphic_on A"
⟨proof⟩

```

```

lemma analytic_on_q_Eisenstein_G [analytic_intros]:
  "f analytic_on A ⇒ (λz. z ∈ A ⇒ norm (f z) < 1) ⇒
    (λz. q_Eisenstein_G k (f z)) analytic_on A"
⟨proof⟩

```

```

lemma tendsto_q_Eisenstein_G [tendsto_intros]:
  assumes "(f ⟶ q) F" "norm q < 1"
  shows "((λz. q_Eisenstein_G k (f z)) ⟶ q_Eisenstein_G k q) F"
⟨proof⟩

```

```

lemma continuous_q_Eisenstein_G [continuous_intros]:
  "continuous (at x within A) f ⇒ norm (f x) < 1 ⇒
    continuous (at x within A) (λz. q_Eisenstein_G k (f z))"
⟨proof⟩

```

The following is the formal q -expansion of G_n .

```

definition fps_Eisenstein_G :: "nat ⇒ complex fps" where
  "fps_Eisenstein_G k = (if k = 0 then -1 else if odd k then 0 else
    2 * (fps_const (zeta k) +
      fps_const ((2*i*pi) ^ k / fact (k-1)) * Abs_fps (divisor_sigma
(k-1))))"

```

```

lemma has_fps_expansion_lambert_sigma_power [fps_expansion_intros]:
  "lambert (λn. of_nat n ^ k :: 'a :: {nat_power_normed_field,banach})
has_fps_expansion
  Abs_fps (λn. of_nat (divisor_sigma k n))"
⟨proof⟩

```

```

lemma has_fps_expansion_q_Eisenstein_G [fps_expansion_intros]:
  "q_Eisenstein_G k has_fps_expansion fps_Eisenstein_G k"
⟨proof⟩

```

```

lemma Eisenstein_G_conv_q:
  assumes z: "Im z > 0"
  shows "Eisenstein_G k z = q_Eisenstein_G k (to_q 1 z)"
<proof>

```

We translate our machinery for deducing representations of G_n in terms of G_4 and G_6 to work for our new views of G_n , including its q -series.

```

lemma eisenstein_series_poly_Eisenstein_G:
  "poly2 (map_poly2 of_rat (eisenstein_series_poly n))
    (Eisenstein_G 4 z) (Eisenstein_G 6 z) = Eisenstein_G (2 * n + 4)
  z"
<proof>

```

```

lemma eisenstein_series_poly_q_Eisenstein_G:
  assumes "norm q < 1"
  shows "poly2 (map_poly2 of_rat (eisenstein_series_poly n))
    (q_Eisenstein_G 4 q) (q_Eisenstein_G 6 q) = q_Eisenstein_G
  (2 * n + 4) q"
  (is "?lhs q = ?rhs q")
<proof>

```

```

lemma eisenstein_series_poly_fps_Eisenstein_G:
  "poly2 (map_poly2 (fps_const o of_rat) (eisenstein_series_poly n))
    (fps_Eisenstein_G 4) (fps_Eisenstein_G 6) = fps_Eisenstein_G (2 *
  n + 4)" (is "?lhs = ?rhs")
<proof>

```

As an application, we show that the identities for G_n give rise to related identities on the divisor function by looking at the coefficients of the q -expansions, i.e. $G_8 = \frac{3}{7}G_4^2$ gives rise to

$$\sigma_7(n) = \sigma_3(n) + 120 \sum_{k=1}^{n-1} \sigma_3(k)\sigma_3(n-k) .$$

```

theorem divisor_sigma_7_conv_3:
  "divisor_sigma 7 n =
  divisor_sigma 3 n + 120 * (∑ k=1..<n. divisor_sigma 3 k * divisor_sigma
  3 (n - k))"
<proof>

```

We now 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) = automorphy_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⟩

lemma Eisenstein_G_plus1: "Eisenstein_G k (z + 1) = Eisenstein_G k z"
 ⟨proof⟩

12.2 The monomials of the Eisenstein polynomial

The polynomial $P(X, Y)$ that gives us $E_{2n+4} = P(E_4, E_6)$ only has monomials of the form $X^i Y^j$ with $4i + 6j = 2n + 4$.

definition coeff2 where "coeff2 p m n = poly.coeff (poly.coeff p n) m"

definition is_46_poly :: "nat \Rightarrow 'a :: comm_ring_1 poly poly \Rightarrow bool" where
 "is_46_poly n p \longleftrightarrow ($\forall i j. \text{coeff2 } p \ i \ j \neq 0 \longrightarrow 4 * i + 6 * j = n$)"

lemma is_46_poly_induct [consumes 1, case_names zero add smult monom]:
 assumes "is_46_poly n p"
 assumes "P 0"
 assumes " $\bigwedge p q. P p \implies P q \implies P (p + q)$ "
 assumes " $\bigwedge p c. P p \implies P (\text{Polynomial.smult } [:c:] p)$ "
 assumes " $\bigwedge i j. 4 * i + 6 * j = n \implies P (\text{Polynomial.monom } (\text{Polynomial.monom } 1 \ i) \ j)$ "
 shows "P p"
 ⟨proof⟩

lemma is_46_poly_0: "is_46_poly n 0"
 ⟨proof⟩

lemma is_46_poly_1: "is_46_poly 0 1"
 ⟨proof⟩

lemma is_46_poly_const: "is_46_poly 0 [: [:c:] :]"
 ⟨proof⟩

lemma is_46_poly_x: "is_46_poly 4 [: [0, 1 :] :]"
 ⟨proof⟩

lemma is_46_poly_y: "is_46_poly 6 [: 0, [:1:] :]"
 ⟨proof⟩

lemma is_46_poly_x_power: "m = 4 * n \implies is_46_poly m [: Polynomial.monom c n :]"
 ⟨proof⟩

lemma is_46_poly_y_power: "m = 6 * n \implies is_46_poly m (Polynomial.monom [:c:] n)"
 ⟨proof⟩

```

lemma is_46_poly_smult:
  assumes "is_46_poly n p"
  shows "is_46_poly n (Polynomial.smult [:c:] p)"
  <proof>

lemma is_46_poly_add:
  assumes "is_46_poly n p" "is_46_poly n q"
  shows "is_46_poly n (p + q)"
  <proof>

lemma is_46_poly_diff:
  assumes "is_46_poly n p" "is_46_poly n q"
  shows "is_46_poly n (p - q)"
  <proof>

lemma is_46_poly_uminus:
  assumes "is_46_poly n p"
  shows "is_46_poly n (-p)"
  <proof>

lemma is_46_poly_sum:
  assumes "\x. x \in A \implies is_46_poly n (p x)"
  shows "is_46_poly n (\sum x \in A. p x)" <proof>

lemma is_46_poly_mult:
  assumes "is_46_poly n1 p" "is_46_poly n2 q" "n = n1 + n2"
  shows "is_46_poly n (p * q)"
  <proof>

lemma is_46_poly_power:
  assumes "is_46_poly n' p" "n = m * n'"
  shows "is_46_poly n (p ^ m)" <proof>

lemma is_46_poly_eisenstein_series_poly: "is_46_poly (2*n+4) (eisenstein_series_poly
n)"
<proof>

```

12.3 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 \Rightarrow complex \Rightarrow complex" where
  "Eisenstein_E k z = Eisenstein_G k z / (2 * zeta k)"

```

```

lemma of_real_of_rat [simp]: "of_real (of_rat x) = of_rat x"
  ⟨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_plus_int: "Eisenstein_E k (z + of_int m) = Eisenstein_E k z"
  ⟨proof⟩

lemma Eisenstein_E_plus1: "Eisenstein_E k (z + 1) = Eisenstein_E k z"
  ⟨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⟩

lemma continuous_on_Eisenstein_E [continuous_intros]:
  "continuous_on A f ⇒ odd k ∨ (∀x∈A. f x ∉ ℝ) ⇒
  continuous_on A (λz. Eisenstein_E k (f z))"

```

<proof>

theorem Eisenstein_E_apply_modgrp:

assumes "k ≠ 2"

shows "Eisenstein_E k (apply_modgrp f z) = automorphy_factor f z ^ k * Eisenstein_E k z"

<proof>

The Fourier expansion of $E_k(z)$:

definition q_Eisenstein_E :: "nat ⇒ complex ⇒ complex" where

"q_Eisenstein_E k q = (if k = 0 then 1 else if odd k then 0 else 1 - 2 * of_nat k / of_real (bernoulli k) * lambert (λn. of_nat n ^ (k - 1)) q)"

lemma q_Eisenstein_G_conv_E: "q_Eisenstein_G k q = 2 * zeta k * q_Eisenstein_E k q"

<proof>

lemma Eisenstein_E_conv_q:

assumes z: "Im z > 0"

shows "Eisenstein_E k z = q_Eisenstein_E k (to_q 1 z)"

<proof>

lemma Eisenstein_E_fourier:

assumes "Im z > 0"

shows "Eisenstein_E k z = q_Eisenstein_E k (to_q 1 z)"

<proof>

The formal power series corresponding to this Fourier expansion:

definition fps_Eisenstein_E :: "nat ⇒ complex fps" where

"fps_Eisenstein_E k =
(if k = 0 then 1 else if odd k then 0
else 1 - fps_const (2 * of_nat k / bernoulli k) * Abs_fps (λn. of_nat (divisor_sigma (k-1) n)))"

lemma fps_nth_0_Eisenstein_E [simp]: "even k ⇒ fps_nth (fps_Eisenstein_E k) 0 = 1"

<proof>

lemma fps_Eisenstein_G_conv_E:

"fps_Eisenstein_G k = fps_const (2 * zeta k) * fps_Eisenstein_E k"

<proof>

lemma fps_Eisenstein_E_eq_0_iff [simp]: "fps_Eisenstein_E k = 0 ⇔ odd k"

<proof>

lemma subdegree_fps_Eisenstein_E [simp]: "subdegree (fps_Eisenstein_E

$k) = 0$
 $\langle proof \rangle$

lemma `has_fps_expansion_q_Eisenstein_E [fps_expansion_intros]:`
`"q_Eisenstein_E k has_fps_expansion fps_Eisenstein_E k"`
 $\langle proof \rangle$

lemma `holomorphic_on_q_Eisenstein_E [holomorphic_intros]:`
`"f holomorphic_on A \implies ($\bigwedge z. z \in A \implies \text{norm } (f z) < 1$) \implies`
`($\lambda z. q_Eisenstein_E k (f z)$) holomorphic_on A"`
 $\langle proof \rangle$

lemma `analytic_on_q_Eisenstein_E [analytic_intros]:`
`"f analytic_on A \implies ($\bigwedge z. z \in A \implies \text{norm } (f z) < 1$) \implies`
`($\lambda z. q_Eisenstein_E k (f z)$) analytic_on A"`
 $\langle proof \rangle$

lemma `tendsto_q_Eisenstein_E [tendsto_intros]:`
`assumes "(f \longrightarrow q) F" "norm q < 1"`
`shows "($\lambda z. q_Eisenstein_E k (f z)$) \longrightarrow q_Eisenstein_E k q) F"`
 $\langle proof \rangle$

lemma `continuous_q_Eisenstein_E [continuous_intros]:`
`"continuous (at x within A) f \implies norm (f x) < 1 \implies`
`continuous (at x within A) ($\lambda z. q_Eisenstein_E k (f z)$)"`
 $\langle proof \rangle$

12.4 Identities for normalised Eisenstein series

Just like G_{2n} , E_{2n} can be given as a polynomial in E_4 and E_6 with rational coefficients. The identities resulting from this tend to look a bit nicer than those for G .

context

`fixes B :: "nat \Rightarrow rat"`

`defines "B \equiv bernoulli"`

begin

definition `eisenstein_series_poly'_aux :: "nat \Rightarrow nat \Rightarrow rat" where`
`"eisenstein_series_poly'_aux n i =`
`- (B (2 * (i + 2)) * B (2 * (n - i)) / B (2 * (n + 2))) *`
`(of_nat (2 * (n + 2) choose 2 * (i + 2)))"`

lemma `of_rat_eisenstein_series_poly'_aux:`
`assumes "i + 2 \leq n"`
`shows "of_rat (eisenstein_series_poly'_aux n i) =`
`2 * zeta (2 * of_nat (i+2)) * zeta (2 * of_nat (n-i)) / zeta`
`(2 * of_nat (n+2))"`
 $\langle proof \rangle$

```

fun eisenstein_series_poly' :: "nat  $\Rightarrow$  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)):]
      ( $\sum$  i<n-2. Polynomial.smult [: of_nat ((2*i+3)*(2*(n-i)-1))
* eisenstein_series_poly'_aux n i:]
      (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  $\geq$  2  $\implies$  eisenstein_series_poly' n =
      Polynomial.smult [:3 / of_nat ((2*n+5) * (n-1) * (2*n+3)):]
      ( $\sum$  i<n-2. Polynomial.smult [: of_nat ((2*i+3)*(2*(n-i)-1))
* eisenstein_series_poly'_aux n i:]
      (eisenstein_series_poly' i * eisenstein_series_poly' (n-2-i)))"
    <proof>

lemma is_46_poly_eisenstein_series_poly': "is_46_poly (2*n+4) (eisenstein_series_poly'
n)"
  <proof>

lemma eisenstein_series_poly'_correct:
  assumes z: "Im z > 0"
  defines "E  $\equiv$  ( $\lambda$ n. Eisenstein_E n z)"
  shows "poly2 (map_poly2 of_rat (eisenstein_series_poly' n)) (E 4) (E
6) = E (2 * n + 4)"
  <proof>

end

lemma eisenstein_series_poly_q_Eisenstein_E:
  assumes "norm q < 1"
  shows "poly2 (map_poly2 of_rat (eisenstein_series_poly' n))
      (q_Eisenstein_E 4 q) (q_Eisenstein_E 6 q) = q_Eisenstein_E
(2 * n + 4) q"
      (is "?lhs q = ?rhs q")
  <proof>

lemma eisenstein_series_poly'_fps_Eisenstein_E:
  "poly2 (map_poly2 (fps_const  $\circ$  of_rat) (eisenstein_series_poly' n))
      (fps_Eisenstein_E 4) (fps_Eisenstein_E 6) = fps_Eisenstein_E (2 *
n + 4)" (is "?lhs = ?rhs")

```

<proof>

More efficient computation:

```
definition eisenstein_polys'_step :: "rat poly poly list  $\Rightarrow$  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)):]
        ( $\sum_{i \leq n-2}$ . Polynomial.smult [: of_nat ((2*i+3)*(2*(n-i)-1))
* eisenstein_series_poly'_aux n i :]
        (ps ! i * ps ! (n-2-i)))]])"
```

```
definition eisenstein_series_polys' :: "nat  $\Rightarrow$  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  $\geq$  2" and ps_eq: "ps = map eisenstein_series_poly' [0..shows "eisenstein_polys'_step ps = map eisenstein_series_poly' [0.. $\text{Suc } n$ ]"
<proof>
```

```
lemma eisenstein_series_polys'_correct:
  "eisenstein_series_polys' n = map eisenstein_series_poly' [0.. $\text{max } 2$  n]"
<proof>
```

```
lemma eisenstein_series_poly_code [code]:
  "eisenstein_series_poly' n = eisenstein_series_polys' (Suc n) ! n"
<proof>
```

Again, a number of identities for the normalised Eisenstein series follows directly by simply expressing both sides in terms of E_4 and E_6 and simplifying. Corresponding identities for the divisor function can be read off directly from the coefficients of these power series.

context

fixes E

defines " $E \equiv \text{fps_Eisenstein_E}$ "

begin

```
lemma eisenstein_series_polys'_correct':
  assumes "eisenstein_series_polys' m = ps"
  shows "list_all ( $\lambda i$ .  $E (2*i+4) = \text{poly2} (\text{map\_poly2} (\text{fps\_const} \circ \text{of\_rat})$ 
(ps ! i)) (E 4) (E 6)) [0.. $m$ ]"
<proof>
```

```
lemma Eisenstein_E_identities:
  " $E 8 = E 4 \wedge 2$ " (is ?th8)
  " $E 10 = E 4 * E 6$ " (is ?th10)
```

```

"E 12 = 441 / 691 * E 4 ^ 3 + 250 / 691 * E 6 ^ 2" (is ?th12)
"E 14 = E 4 ^ 2 * E 6" (is ?th14)
"E 16 = 1617 / 3617 * E 4 ^ 4 + 2000 / 3617 * E 4 * E 6 ^ 2" (is ?th16)
"E 18 = 38367 / 43867 * E 4 ^ 3 * E 6 + 5500 / 43867 * E 6 ^ 3" (is ?th18)
"E 20 = 53361 / 174611 * E 4 ^ 5 + 121250 / 174611 * E 4 ^ 2 * E 6 ^
2" (is ?th20)
<proof>

```

end

As an example, we derive the identity that expresses σ_9 in terms of σ_3 and σ_5 :

```

lemma divisor_sigam_9_conv_3_5:
  fixes n :: nat
  assumes n: "n > 0"
  fixes sigma :: "nat  $\Rightarrow$  nat  $\Rightarrow$  int" (" $\sigma$  ")
  defines "sigma  $\equiv$  ( $\lambda$ k n. int (divisor_sigma k n))"
  shows "11 * ( $\sigma$ _9 n) = -10 * ( $\sigma$ _3 n) + 21 * ( $\sigma$ _5 n) +
        5040 * ( $\sum_{i \in \{0 <..<n\}} \sigma_3 i * \sigma_5 (n - i)$ )"
<proof>

```

12.5 The modular discriminant

As before, the modular discriminant is defined as $(60G_4^3) - 27(140G_6)^2$. It is non-zero everywhere and (as we will see later) vanishes at $i\infty$.

```

definition modular_discr :: "complex  $\Rightarrow$  complex" where
  "modular_discr z = (60 * Eisenstein_G 4 z) ^ 3 - 27 * (140 * Eisenstein_G
6 z) ^ 2"

```

```

lemma modular_discr_altdef:
  "modular_discr z = (4/3)^3 * of_real pi ^ 12 * (Eisenstein_E 4 z ^ 3
- Eisenstein_E 6 z ^ 2)"
<proof>

```

```

lemma (in complex_lattice) discr_eq_modular_discr: "discr = modular_discr
( $\omega$ 2 /  $\omega$ 1) /  $\omega$ 1 ^ 12"
<proof>

```

```

lemma modular_discr_real_eq_0 [simp]: "z  $\in$   $\mathbb{R} \implies$  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" " $\wedge$ z. z  $\in$  A  $\implies$  f z  $\notin$   $\mathbb{R}$ "
  shows "( $\lambda$ z. modular_discr (f z)) analytic_on A"

```

<proof>

lemma modular_discr_holomorphic [holomorphic_intros]:
 assumes "f holomorphic_on A" " $\wedge z. z \in A \implies f z \notin \mathbb{R}$ "
 shows " $(\lambda z. \text{modular_discr } (f z))$ holomorphic_on A"
 <proof>

lemma continuous_modular_discr [continuous_intros]:
 "continuous (at x within A) f $\implies f x \notin \mathbb{R} \implies$
 continuous (at x within A) $(\lambda z. \text{modular_discr } (f z))$ "
 <proof>

lemma continuous_on_modular_discr [continuous_intros]:
 "continuous_on A f $\implies (\forall x \in A. f x \notin \mathbb{R}) \implies$
 continuous_on A $(\lambda z. \text{modular_discr } (f z))$ "
 <proof>

lemma modular_discr_uminus: "modular_discr (-z) = modular_discr z"
 <proof>

lemma modular_discr_nonzero:
 assumes "z $\notin \mathbb{R}$ "
 shows "modular_discr z $\neq 0$ "
 <proof>

lemma modular_discr_eq_0_iff: "modular_discr z = 0 $\longleftrightarrow z \in \mathbb{R}$ "
 <proof>

theorem modular_discr_apply_modgrp:
 "modular_discr (apply_modgrp f z) = automorphy_factor f z $^ 12 * \text{modular_discr } z$ "
 <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 $^ 12 * \text{modular_discr } z$ "
 <proof>

12.6 Ramanujan's tau function

The Ramanujan τ function are the integer coefficients of the normalised modular discriminant $\Delta/(2\pi)^{12}$.

definition ramanujan_tau :: "nat \Rightarrow int" where
 "ramanujan_tau n =
 (let F = $(\lambda k. \text{Abs_fps } (\text{divisor_sigma } k))$
 in fps_nth $((1 + 240 * F 3) ^ 3 - (1 - 504 * F 5) ^ 2)$ n div 12

$\wedge 3)$ "

definition *fps_modular_discr* :: "complex fps" where
"fps_modular_discr = fps_const ((2 * pi) ^ 12) * of_int.fps (Abs_fps ramanujan_tau)"

lemma *fps_ramanujan_tau_eq*:
"of_int.fps (Abs_fps ramanujan_tau) =
fps_const (1 / 12 ^ 3) * (fps_Eisenstein_E 4 ^ 3 - fps_Eisenstein_E
6 ^ 2)"
<proof>

lemma *fps_modular_discr_conv_Eisenstein*:
"fps_modular_discr =
fps_const ((4/3)^3 * of_real pi ^ 12) *
(fps_Eisenstein_E 4 ^ 3 - fps_Eisenstein_E 6 ^ 2)"
<proof>

lemma *atLeastAtMost_nat_numeral_right*:
"a ≤ numeral b ⇒ {(a::nat)..numeral b} = insert (numeral b) {a..pred_numeral
b}"
<proof>

lemma *ramanujan_tau_0 [simp]*: "ramanujan_tau 0 = 0"
and *ramanujan_tau_1 [simp]*: "ramanujan_tau (Suc 0) = 1"
and *ramanujan_tau_2*: "ramanujan_tau 2 = -24"
and *ramanujan_tau_3*: "ramanujan_tau 3 = 252"
and *ramanujan_tau_4*: "ramanujan_tau 4 = -1472"
and *ramanujan_tau_5*: "ramanujan_tau 5 = 4830"
and *ramanujan_tau_6*: "ramanujan_tau 6 = -6048"
<proof>

lemma *fps_modular_discr_nonzero [simp]*: "fps_modular_discr ≠ 0"
and *subdegree_fps_modular_discr [simp]*: "subdegree fps_modular_discr
= 1"
<proof>

12.7 The modular λ function

definition *modular_lambda* :: "complex ⇒ complex" where
"modular_lambda z = (if z ∈ ℝ then 0 else complex_lattice.modulus 1
z)"

lemma (in *complex_lattice*) *modulus_eq_modular_lambda*:
"modulus = modular_lambda (ω2 / ω1)"
<proof>

lemma *modular_lambda_real_eq_0 [simp]*: "z ∈ ℝ ⇒ modular_lambda z =

0"
 ⟨proof⟩

lemma modular_lambda_cnj: "modular_lambda (cnj z) = cnj (modular_lambda z)"
 ⟨proof⟩

lemma modular_lambda_uminus: "modular_lambda (-z) = modular_lambda z"
 ⟨proof⟩

lemma modular_lambda_conv_jacobi_theta:
 assumes z: "Im z > 0"
 shows "modular_lambda z = jacobi_theta_10 0 z ^ 4 / jacobi_theta_00 0 z ^ 4"
 ⟨proof⟩

lemma modular_lambda_analytic [analytic_intros]:
 assumes "f analytic_on A" " $\bigwedge z. z \in A \implies f z \notin \mathbb{R}$ "
 shows " $(\lambda z. \text{modular_lambda } (f z)) \text{ analytic_on } A$ "
 ⟨proof⟩

lemma modular_lambda_holomorphic [holomorphic_intros]:
 assumes "f holomorphic_on A" " $\bigwedge z. z \in A \implies f z \notin \mathbb{R}$ "
 shows " $(\lambda z. \text{modular_lambda } (f z)) \text{ holomorphic_on } A$ "
 ⟨proof⟩

lemma modular_lambda_meromorphic [meromorphic_intros]:
 assumes "f analytic_on A" " $\bigwedge z. z \in A \implies f z \notin \mathbb{R}$ "
 shows " $(\lambda z. \text{modular_lambda } (f z)) \text{ meromorphic_on } A$ "
 ⟨proof⟩

lemma tendsto_modular_lambda [tendsto_intros]:
 assumes "(f \longrightarrow z) F" "z $\notin \mathbb{R}$ "
 shows " $(\lambda z. \text{modular_lambda } (f z)) \longrightarrow \text{modular_lambda } z) F$ "
 ⟨proof⟩

lemma continuous_modular_lambda [continuous_intros]:
 "continuous (at x within A) f $\implies f x \notin \mathbb{R} \implies$
 continuous (at x within A) $(\lambda z. \text{modular_lambda } (f z))$ "
 ⟨proof⟩

lemma continuous_on_modular_lambda [continuous_intros]:
 "continuous_on A f $\implies (\bigwedge x. x \in A \implies f x \notin \mathbb{R}) \implies$
 continuous_on A $(\lambda z. \text{modular_lambda } (f z))$ "
 ⟨proof⟩

12.8 Klein's J invariant

Klein's J invariant is defined as $(60G_4)^3/\Delta$, or equivalently $E_4^3/(E_4^3 - E_6^2)$. It is exactly the J invariant of a lattice we saw earlier.

Note that there are a number of different conventions, e.g. sometimes one also calls what we would write as $1728J$ the j invariant (with a lower case j).

definition `Klein_J` :: "complex \Rightarrow 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 (ω_2 / ω_1)"`
`<proof>`

lemma `Klein_J_conv_modular_lambda`:
`assumes "z \notin \mathbb{R} "`
`defines "x \equiv modular_lambda z"`
`shows "Klein_J z = 4 * (1 - x * (1 - x)) ^ 3 / (27 * (x * (1 - x))`
`^ 2)"`
`<proof>`

lemma `Klein_J_real_eq_0 [simp]`: "`z \in \mathbb{R} \implies 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" " $\bigwedge z. z \in A \implies f z \notin \mathbb{R}$ "`
`shows " $(\lambda z. Klein_J (f z))$ analytic_on A"`
`<proof>`

lemma `Klein_J_holomorphic [holomorphic_intros]`:
`assumes "f holomorphic_on A" " $\bigwedge z. z \in A \implies f z \notin \mathbb{R}$ "`
`shows " $(\lambda z. Klein_J (f z))$ holomorphic_on A"`
`<proof>`

lemma `continuous_Klein_J [continuous_intros]`:
`"continuous (at x within A) f \implies f x \notin \mathbb{R} \implies`
`continuous (at x within A) ($\lambda z. Klein_J (f z)$)"`
`<proof>`

lemma `continuous_on_Klein_J [continuous_intros]`:
`"continuous_on A f \implies ($\forall x \in A. f x \notin \mathbb{R}$) \implies`
`continuous_on A ($\lambda z. Klein_J (f z)$)"`

<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_plus1:* "Klein_J (z + 1) = Klein_J z"

<proof>

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

<proof>

lemma *Klein_J_cong:*

assumes "w \sim_{Γ} z"

shows "Klein_J w = Klein_J z"

<proof>

12.9 Klein's c

definition *Klein_c* :: "nat \Rightarrow int" where

"Klein_c n = (let A = of_nat.fps (Abs_fps (divisor_sigma 3));

C = fps_left_inverse (fps_shift 1 (Abs_fps ramanujan_tau))

1

in fps_nth (C * (1 + 240 * A) ^ 3) (n + 1))"

definition *fls_Klein_j* :: "complex fls"

where "fls_Klein_j = fls_X_inv + fps_to_fls (Abs_fps (λ x. of_int (Klein_c x)))"

definition *fls_Klein_J* :: "complex fls"

where "fls_Klein_J = fls_const (1 / 1728) * fls_Klein_j"

lemma *fls_Klein_J_conv_modular_discr:*

"fls_Klein_J = fps_to_fls ((60 * fps_Eisenstein_G 4) ^ 3) / fps_to_fls fps_modular_discr"

<proof>

lemma *fls_Klein_J_conv_Eisenstein_E:*

"fls_Klein_J =

fps_to_fls (fps_Eisenstein_E 4 ^ 3) /

fps_to_fls (fps_Eisenstein_E 4 ^ 3 - fps_Eisenstein_E 6 ^ 2)"

<proof>

lemma *fps_left_inverse_constructor_natural:*

```

"fps_left_inverse_constructor a b (numeral n) =
  - ( $\sum i = 0..pred\_numeral\ n.$  fps_left_inverse_constructor a b i *
a $ (Suc (pred_numeral n) - i)) * b"
<proof>

```

```

lemma Klein_c_0: "Klein_c 0 = 744"
and Klein_c_1: "Klein_c (Suc 0) = 196884"
and Klein_c_2: "Klein_c 2 = 21493760"
and Klein_c_3: "Klein_c 3 = 864299970"
and Klein_c_4: "Klein_c 4 = 20245856256"
<proof>

```

```

lemma fls_Klein_j_subdegree [simp]: "fls_subdegree fls_Klein_j = -1"
<proof>

```

```

lemma fls_Klein_j_nonzero [simp]: "fls_Klein_j  $\neq$  0"
<proof>

```

```

lemma fls_Klein_J_subdegree [simp]: "fls_subdegree fls_Klein_J = -1"
<proof>

```

```

lemma fls_Klein_J_nonzero [simp]: "fls_Klein_J  $\neq$  0"
<proof>

```

12.10 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  $\neq$  2" " $\neg$ 4 dvd k"
  shows "Eisenstein_G k i = 0"
<proof>

```

```

lemma Eisenstein_E_ii_eq_0:
  assumes "k  $\neq$  2" " $\neg$ 4 dvd k"
  shows "Eisenstein_E k i = 0"
<proof>

```

```

lemma Eisenstein_G_6_ii [simp]: "Eisenstein_G 6 i = 0"
<proof>

```

lemma *Eisenstein_E_6_ii* [simp]: "Eisenstein_E 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_E_rho_eq_0*:
 assumes "k ≠ 2" "¬6 dvd k"
 shows "Eisenstein_E k ρ = 0"
 ⟨proof⟩

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

lemma *Eisenstein_E_4_rho* [simp]: "Eisenstein_E 4 ρ = 0"
 ⟨proof⟩

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

corollary *Eisenstein_E_6_rho_nonzero*: "Eisenstein_E 6 ρ ≠ 0"
 ⟨proof⟩

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

corollary *Eisenstein_E_4_ii_nonzero*: "Eisenstein_E 4 i ≠ 0"
 ⟨proof⟩

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

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

12.11 Consequences for the fundamental region

One 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.

Otherwise, if there were a point x and a sequence x_n of points equivalent to it and converging to it, the function $f(z) = J(z) - J(x)$ would have a zero at every x_n and therefore be identically zero by analytic continuation. This would make J a constant function, but since $J(i) = 1$ and $J(\rho) = 0$, this

gives a contradiction.

```
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  $\implies$  eventually ( $\lambda y. \neg \text{rel } y \ x$ ) (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
unbundle no modgrp_notation
```

end

13 The Dedekind η function

```
theory Dedekind_Eta
imports
  Bernoulli.Bernoulli
  Theta_Inversion
  Complex_Lattices_Theta
  Basic_Modular_Forms
  Dedekind_Sums.Dedekind_Sums
  Pentagonal_Number_Theorem.Pentagonal_Number_Theorem
begin
```

```
hide_const (open) Unique_Factorization.coprime
```

13.1 Definition and basic properties

```
definition dedekind_eta:: "complex  $\Rightarrow$  complex" ("η") where
  "η z = to_nome (z / 12) * euler_phi (to_nome (2*z))"
```

```
lemma dedekind_eta_nonzero [simp]: "Im z > 0  $\implies$  η z  $\neq$  0"
  <proof>
```

```
lemma holomorphic_dedekind_eta [holomorphic_intros]:
  assumes "A  $\subseteq$  {z. Im z > 0}"
  shows   "η holomorphic_on A"
  <proof>
```

```

lemma holomorphic_dedekind_eta' [holomorphic_intros]:
  assumes "f holomorphic_on A" " $\bigwedge z. z \in A \implies \text{Im } (f z) > 0$ "
  shows " $(\lambda z. \eta (f z))$  holomorphic_on A"
  <proof>

lemma analytic_dedekind_eta [analytic_intros]:
  assumes "A  $\subseteq$  {z. Im z > 0}"
  shows " $\eta$  analytic_on A"
  <proof>

lemma analytic_dedekind_eta' [analytic_intros]:
  assumes "f analytic_on A" " $\bigwedge z. z \in A \implies \text{Im } (f z) > 0$ "
  shows " $(\lambda z. \eta (f z))$  analytic_on A"
  <proof>

lemma meromorphic_on_dedekind_eta [meromorphic_intros]:
  "f analytic_on A  $\implies$  ( $\bigwedge z. z \in A \implies \text{Im } (f z) > 0$ )  $\implies$  ( $\lambda z. \eta (f z)$ )
  meromorphic_on A"
  <proof>

lemma continuous_on_dedekind_eta [continuous_intros]:
  "A  $\subseteq$  {z. Im z > 0}  $\implies$  continuous_on A  $\eta$ "
  <proof>

lemma continuous_on_dedekind_eta' [continuous_intros]:
  assumes "continuous_on A f" " $\bigwedge z. z \in A \implies \text{Im } (f z) > 0$ "
  shows "continuous_on A ( $\lambda z. \eta (f z)$ )"
  <proof>

lemma tendsto_dedekind_eta [tendsto_intros]:
  assumes "(f  $\longrightarrow$  c) F" "Im c > 0"
  shows "(( $\lambda x. \eta (f x)$ )  $\longrightarrow$   $\eta c$ ) F"
  <proof>

lemma tendsto_at_cusp_dedekind_eta [tendsto_intros]: " $(\eta \longrightarrow 0)$  at_ $i\infty$ "
  <proof>

lemma dedekind_eta_plus1:
  assumes z: "Im z > 0"
  shows " $\eta (z + 1) = \text{cis } (\text{pi}/12) * \eta z$ "
  <proof>

lemma dedekind_eta_plus_nat:
  assumes z: "Im z > 0"
  shows " $\eta (z + \text{of\_nat } n) = \text{cis } (\text{pi} * n / 12) * \eta z$ "
  <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⟩

```

13.2 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 qepochhammer_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 qepochhammer_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 qepochhammer_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 qepochhammer_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 qepochhammer_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>

```

Since theta nullwert functions can be expressed as quotients of Dedekind's η function, we also get the following deep connection between the discriminant of a complex lattice and η .

This can also alternatively be derived very elegantly using modular forms. More precisely: η^2 and the modular discriminant are both cusp forms of weight 12 and that the space of cusp forms of weight 12 is one-dimensional. However, since we already have access to the theta functions and the above connections to the lattice properties, this proof is very simple now as well, without using the heavy tooling of modular forms.

```

theorem (in complex_lattice_Im_pos) discr_conv_dedekind_eta:
  "discr = 4096 * (pi / ω1) ^ 12 * dedekind_eta τ ^ 24"
  <proof>

```

13.3 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 qepochhammer_inf_notation
  <proof>

```

```

theorem dedekind_eta_minus_one_over:
  assumes t: "Im t > 0"

```

shows " $\eta(-1/t) = \text{csqrt}(-i*t) * \eta t$ "
 ⟨proof⟩

13.4 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 $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \text{SL}(2)$ 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 this section, we will give a concrete definition of this 24-th root ϵ in terms of A .

definition `dedekind_eps` :: "modgrp \Rightarrow complex" ("ε") where

```
"ε f = (let f = abs f in
  (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_abs` [simp]: "dedekind_eps (abs f) = dedekind_eps f"
 ⟨proof⟩

lemma `dedekind_eps_uminus` [simp]: "dedekind_eps (-f) = dedekind_eps f"
 ⟨proof⟩

lemma `is_singular_modgrp_abs_iff` [simp]: "is_singular_modgrp (abs f)
 \longleftrightarrow is_singular_modgrp f"
 ⟨proof⟩

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

lemma `dedekind_eps_shift_left`:
 "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 \neq 0"
shows " $\varepsilon (f * S_modgrp) = \text{cis} (-\text{sgn} (\text{modgrp_d } f) * \text{sgn} (\text{modgrp_c } f) * \text{pi} / 4) * \varepsilon f$ "
<proof>

lemma *dedekind_eps_root_of_unity:* " $\varepsilon f ^ 24 = 1$ "

<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 " $\eta (\text{apply_modgrp } f z) = \varepsilon f * \text{csqrt} (\text{automorphy_factor } |f| z) * \eta z$ "
<proof>

no_notation *dedekind_eta* (" η ")

no_notation *dedekind_eps* (" ε ")

end

13.5 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 " $\text{Eisenstein_G } 2 (-1/t) = t^2 * \text{Eisenstein_G } 2 t - 2 * \text{pi} * i * t$ "
<proof>

lemma *Eisenstein_E2_minus_one_over:*

```

assumes t: "Im t > 0"
shows "Eisenstein_E 2 (-(1/t)) = t2 * 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) =
automorphy_factor f z ^ 2 * Eisenstein_G 2 z -
2 * i * pi * modgrp_c f * automorphy_factor f z"
⟨proof⟩

```

```

lemma Eisenstein_E2_apply_modgrp:
assumes "Im z > 0"
shows "Eisenstein_E 2 (apply_modgrp f z) =
automorphy_factor f z ^ 2 * Eisenstein_E 2 z - 6 * i / pi
* modgrp_c f * automorphy_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.