latex

rraya

August 10, 2019

Contents

theory *Hales*
  imports *Complex-Main HOL−Algebra.Group HOL−Algebra.Bij*
    HOL−Library.Bit

begin
declare [[*quick-and-dirty=true*]]

nitpick-params [*verbose*, *card* = *1*−*20*, *max-potential* = *0*,
    sat-solver = *MiniSat-JNI*, *max-threads* = *1*, *timeout* = *600*]

1 Edwards curves

locale *curve-addition* =
  fixes *c d* :: *real*
begin

definition *e* :: *real* ⇒ *real* ⇒ *real* **where**
 e x y = *x*^*2* + *c* ∗ *y*^*2* − *1* − *d* ∗ *x*^*2* ∗ *y*^*2*

definition *delta-plus* :: *real* ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *real* **where**
 delta-plus x1 y1 x2 y2 = *1* + *d* ∗ *x1* ∗ *y1* ∗ *x2* ∗ *y2*

definition *delta-minus* :: *real* ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *real* **where**
 delta-minus x1 y1 x2 y2 = *1* − *d* ∗ *x1* ∗ *y1* ∗ *x2* ∗ *y2*

definition *delta* :: *real* ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *real* **where**
 delta x1 y1 x2 y2 = (*delta-plus x1 y1 x2 y2*) ∗
    (*delta-minus x1 y1 x2 y2*)

fun *add* :: *real* × *real* ⇒ *real* × *real* ⇒ *real* × *real* **where**

```
 add (x1,y1) (x2,y2) =
    ((x1*x2 - c*y1*y2) div (1-d*x1*y1*x2*y2),
     (x1*y2+y1*x2) div (1+d*x1*y1*x2*y2))

lemma add-with-deltas:
 add (x1,y1) (x2,y2) =
    ((x1*x2 - c*y1*y2) div (delta-minus x1 y1 x2 y2),
     (x1*y2+y1*x2) div (delta-plus x1 y1 x2 y2))
  unfolding delta-minus-def delta-plus-def
  by(simp add: algebra-simps)

lemma commutativity: add z1 z2 = add z2 z1
  by(cases z1,cases z2,simp add: algebra-simps)

lemma add-closure:
  assumes z3 = (x3,y3) z3 = add (x1,y1) (x2,y2)
  assumes delta-minus x1 y1 x2 y2 ≠ 0 delta-plus x1 y1 x2 y2 ≠ 0
  assumes e x1 y1 = 0 e x2 y2 = 0
  shows e x3 y3 = 0
proof -
  have x3-expr: x3 = (x1*x2 - c*y1*y2) div (delta-minus x1 y1 x2 y2)
    using assms add-with-deltas by auto
  have y3-expr: y3 = (x1*y2+y1*x2) div (delta-plus x1 y1 x2 y2)
    using assms add-with-deltas by auto
  define prod where prod =
    -1 + x1^2 * x2^2 + c * x2^2 * y1^2 - d * x1^2 * x2^4 * y1^2 +
     c * x1^2 * y2^2 - d * x1^4 * x2^2 * y2^2 + c^2 * y1^2 * y2^2 -
     4 * c * d * x1^2 * x2^2 * y1^2 * y2^2 +
     2 * d^2 * x1^2 * x2^2 * y1^2 * y2^2 + d^2 * x1^4 * x2^4 * y1^2 * y2^2 -
     c^2 * d * x2^2 * y1^4 * y2^2 + c * d^2 * x1^2 * x2^4 * y1^4 * y2^2 -
     c^2 * d * x1^2 * y1^2 * y2^4 + c * d^2 * x1^4 * x2^2 * y1^2 * y2^4 +
     c^2 * d^2 * x1^2 * x2^2 * y1^4 * y2^4 -
     d^4 * x1^4 * x2^4 * y1^4 * y2^4

  define e1 where e1 = e x1 y1
  define e2 where e2 = e x2 y2


  have prod-eq-1: ∃ r1 r2. prod - (r1 * e1 + r2 * e2) = 0
    unfolding prod-def e1-def e2-def e-def by algebra

  define a where a = x1*x2 - c*y1*y2
  define b where b = x1*y2+y1*x2

  have (e x3 y3)*(delta x1 y1 x2 y2)^2 =
         e (a div (delta-minus x1 y1 x2 y2))
           (b div (delta-plus x1 y1 x2 y2)) * (delta x1 y1 x2 y2)^2
    unfolding a-def b-def
    by (simp add: mult.commute mult.left-commute x3-expr y3-expr)
```

```
  also have ... =
    ((a div delta-minus x1 y1 x2 y2)^2 +
    c * (b div delta-plus x1 y1 x2 y2)^2 −
    1 −
    d * (a div delta-minus x1 y1 x2 y2)^2 *
   (b div delta-plus x1 y1 x2 y2)^2) * (delta x1 y1 x2 y2)^2
    unfolding delta-plus-def delta-minus-def delta-def e-def by simp
  also have ... =
    ((a div delta-minus x1 y1 x2 y2)^2 * (delta x1 y1 x2 y2)^2 +
    c * (b div delta-plus x1 y1 x2 y2)^2 * (delta x1 y1 x2 y2)^2 −
    1 * (delta x1 y1 x2 y2)^2 −
    d * (a div delta-minus x1 y1 x2 y2)^2 *
   (b div delta-plus x1 y1 x2 y2)^2 * (delta x1 y1 x2 y2)^2)
    by(simp add: algebra-simps)
  also have ... =
    ((a * delta-plus x1 y1 x2 y2)^2 + c * (b * delta-minus x1 y1 x2 y2)^2 −
     (delta x1 y1 x2 y2)^2 − d * a^2 * b^2)
   unfolding delta-def by(simp add: field-simps assms(3,4))+
  also have ... − prod = 0
      unfolding prod-def delta-plus-def delta-minus-def delta-def a-def b-def by
algebra
  finally have (e x3 y3)*(delta x1 y1 x2 y2)^2 = prod by simp
  then have prod-eq-2: (e x3 y3) = prod div (delta x1 y1 x2 y2)^2
    using assms(3,4) delta-def by auto

  have e1 = 0 unfolding e1-def using assms(5) by simp
  moreover have e2 = 0 unfolding e2-def using assms(6) by simp
  ultimately have prod = 0 using prod-eq-1 by simp
  then show e x3 y3 = 0 using prod-eq-2 by simp
qed

lemma associativity:
  assumes z1' = (x1',y1') z3' = (x3',y3')
  assumes z1' = add (x1,y1) (x2,y2) z3' = add (x2,y2) (x3,y3)
  assumes delta-minus x1 y1 x2 y2 ≠ 0 delta-plus x1 y1 x2 y2 ≠ 0
          delta-minus x2 y2 x3 y3 ≠ 0 delta-plus x2 y2 x3 y3 ≠ 0
          delta-minus x1' y1' x3 y3 ≠ 0 delta-plus x1' y1' x3 y3 ≠ 0
          delta-minus x1 y1 x3' y3' ≠ 0 delta-plus x1 y1 x3' y3' ≠ 0
  assumes e x1 y1 = 0 e x2 y2 = 0 e x3 y3 = 0
  shows add (add (x1,y1) (x2,y2)) (x3,y3) = add (x1,y1) (add (x2,y2) (x3,y3))

proof −
 define e1 where e1 = e x1 y1
 define e2 where e2 = e x2 y2
 define e3 where e3 = e x3 y3
 define Delta_x where Delta_x =
   (delta-minus x1' y1' x3 y3)*(delta-minus x1 y1 x3' y3')*
   (delta x1 y1 x2 y2)*(delta x2 y2 x3 y3)
 define Delta_y where Delta_y =
```

(delta-plus x1' y1' x3 y3)*(delta-plus x1 y1 x3' y3')*
(delta x1 y1 x2 y2)*(delta x2 y2 x3 y3)
define g_x :: real **where** g_x = fst(add z1' (x3,y3)) − fst(add (x1,y1) z3')
define g_y **where** g_y = snd(add z1' (x3,y3)) − snd(add (x1,y1) z3')
define gxpoly **where** gxpoly = g_x * $Delta_x$
define gypoly **where** gypoly = g_y * $Delta_y$

define gxpoly-expr **where** gxpoly-expr =
d*x2* y2* (x1^2* x2* x3* y1−x1^2* x2* x3^3* y1−c* x1* x3* y1^2* y2+d* x1^3* x2^2* x3* y1^2* y2
+c* x1* x3^3* y1^2* y2−d* x1^3* x2^2* x3^3* y1^2* y2−c* d* x1^2* x2* x3* y1^3* y2^2+c* d* x1^2* x2* x3^3* y1^3* y2^2
−x1* x2* x3^2* y3+x1^3* x2* x3^2* y3+c* x1* x2* y1^2* y3−d* x1^3* x2^3* x3^2* y1^2* y3+c* x1^2* y1* y2* y3
−c* x3^2* y1* y2* y3−c* d* x1^2* x2^2* y1^3* y2* y3+c^2* x3^2* y1^3* y2* y3−c* d* x1^3* x2* y1^2* y2^2* y3
+d^2* x1^3* x2^3* x3^2* y1^2* y2^2* y3−c^2* d* x1^2* x3^2* y1^3* y2^3* y3+c* d^2* x1^2* x2^2* x3^2* y1^3* y2^3* y3
−c* x2* x3* y1* y3^2+d* x1^2* x2^3* x3^3* y1* y3^2+c^2* x2* x3* y1^3* y3^2−c* d* x1^2* x2^3* x3* y1^3* y3^2
+c* x1* x3* y2* y3^2−c* x1^3* x3* y2* y3^2−d* x1* x2^2* x3^3* y2* y3^2+d* x1^3* x2^2* x3^3* y2* y3^2
+c* d* x2* x3^3* y1* y2^2* y3^2−d^2* x1^2* x2^3* x3^3* y1* y2^2* y3^2+c* d^2* x1^2* x2^3* x3* y1^3* y2^2* y3^2
−c^2* d *x2* x3^3* y1^3* y2^2* y3^2+c^2* d* x1^3* x3* y1^2* y2^3* y3^2−c* d^2* x1^3 *x2^2* x3* y1^2* y2^3* y3^2
−c^2* d* x1* x3^3* y1^2* y2^3* y3^2+c* d^2* x1* x2^2* x3^3* y1^2* y2^3* y3^2−c^2* x1* x2* y1^2* y3^3
+c* d* x1* x2^3* x3^2* y1^2* y3^3−c^2* x1^2* y1* y2* y3^3+c* d* x2^2* x3^2* y1* y2* y3^3+c^2* d* x1^2* x2^2* y1^3* y2* y3^3
−c^2* d* x2^2* x3^2* y1^3* y2* y3^3+c* d* x1* x2* x3^2* y2^2* y3^3−c* d *x1^3* x2* x3^2* y2^2* y3^3
+c^2* d* x1^3* x2* y1^2* y2^2* y3^3−c* d^2* x1* x2^3* x3^2* y1^2* y2^2* y3^3+c^2* d* x1^2* x3^2* y1* y2^3* y3^3
−c* d^2* x1^2* x2^2* x3^2* y1* y2^3* y3^3)

define gypoly-expr **where** gypoly-expr =
−d* x2* y2* (x1* x2* x3* y1^2−x1* x2* x3^3* y1^2+x1^2* x3* y1* y2−x1^2* x3^3* y1* y2−d* x1^2* x2^2* x3* y1^3* y2
+d* x1^2* x2^2* x3^3* y1^3* y2−d* x1^3* x2* x3* y1^2* y2^2+d* x1^3* x2* x3^3* y1^2 *y2^2−x1^2* x2* y1* y3
+x2* x3^2* y1* y3−c* x2* x3^2* y1^3* y3+d* x1^2* x2^3* x3^2* y1^3* y3−x1* x3^2* y2* y3+x1^3* x3^2* y2* y3
+c* x1* y1^2* y2* y3−d* x1^3* x2^2* y1^2* y2* y3+c* d* x1^2* x2* y1^3* y2^2* y3−d^2* x1^2* x2^3* x3^2* y1^3* y2^2* y3
−c* d* x1^3* x3^2* y1^2* y2^3* y3+d^2* x1^3* x2^2* x3^2* y1^2* y2^3* y3−x1* x2* x3* y3^2+x1^3* x2* x3* y3^2
−d* x1^3* x2^3* x3* y1^2* y3^2+d* x1* x2^3* x3^3* y1^2* y3^2−c* x3* y1* y2* y3^2+d *x2^2* x3^3* y1* y2* y3^2

```
  +c^2* x3* y1^3* y2* y3^2−c* d* x2^2* x3^3* y1^3* y2 *y3^2+d* x1* x2*
x3^3* y2^2* y3^2−d* x1^3* x2* x3^3* y2^2* y3^2
  +d^2* x1^3* x2^3* x3 *y1^2* y2^2* y3^2−d^2* x1* x2^3* x3^3* y1^2*
y2^2* y3^2+c* d* x1^2* x3^3* y1* y2^3* y3^2
  −d^2* x1^2* x2^2* x3^3* y1* y2^3* y3^2−c^2* d* x1^2* x3* y1^3* y2^3*
y3^2+c* d^2* x1^2* x2^2* x3* y1^3* y2^3* y3^2
  +c* x1^2* x2* y1* y3^3−d* x1^2* x2^3* x3^2* y1* y3^3+d* x1* x2^2*
x3^2* y2* y3^3−d* x1^3* x2^2* x3^2* y2* y3^3
  −c^2* x1* y1^2 *y2* y3^3+c* d *x1^3* x2^2* y1^2* y2* y3^3−c* d* x2*
x3^2* y1* y2^2* y3^3+d^2* x1^2* x2^3* x3^2* y1* y2^2* y3^3
  −c^2* d* x1^2* x2* y1^3* y2^2* y3^3+c^2* d* x2* x3^2* y1^3* y2^2*
y3^3+c^2* d* x1* x3^2* y1^2* y2^3* y3^3
  −c* d^2* x1* x2^2* x3^2* y1^2* y2^3* y3^3)

have x1'-expr: x1' = (x1 * x2 − c * y1 * y2) / (1 − d * x1 * y1 * x2 * y2)
  using assms(1,3) by auto
have y1'-expr: y1' = (x1 * y2 + y1 * x2) / (1 + d * x1 * y1 * x2 * y2)
  using assms(1,3) by auto
have x3'-expr: x3' = (x2 * x3 − c * y2 * y3) / (1 − d * x2 * y2 * x3 * y3)
  using assms(2,4) by auto
have y3'-expr: y3' = (x2 * y3 + y2 * x3) / (1 + d * x2 * y2 * x3 * y3)
  using assms(2,4) by auto

have non-unfolded-adds:
    delta x1 y1 x2 y2 ≠ 0 using delta-def assms(5,6) by auto

have gx-div: ∃ r1 r2 r3. gxpoly-expr = r1 * e1 + r2 * e2 + r3 * e3
  unfolding gxpoly-expr-def e1-def e2-def e3-def e-def
  by algebra

have gy-div: ∃ r1 r2 r3. gypoly-expr = r1 * e1 + r2 * e2 + r3 * e3
  unfolding gypoly-expr-def e1-def e2-def e3-def e-def
  by algebra

have simp1gx:
  (x1' * x3 − c * y1' * y3) * local.delta-minus x1 y1 x3' y3' *
  (local.delta x1 y1 x2 y2 * local.delta x2 y2 x3 y3) =
  ((x1 * x2 − c * y1 * y2) * x3 * local.delta-plus x1 y1 x2 y2 −
   c * (x1 * y2 + y1 * x2) * y3 * local.delta-minus x1 y1 x2 y2) *
  (local.delta-minus x2 y2 x3 y3 * local.delta-plus x2 y2 x3 y3 −
   d * x1 * y1 * (x2 * x3 − c * y2 * y3) * (x2 * y3 + y2 * x3))

 apply((subst x1'-expr)+, (subst y1'-expr)+,(subst x3'-expr)+,(subst y3'-expr)+)
 apply((subst delta-minus-def[symmetric])+,(subst delta-plus-def[symmetric])+)
  apply(subst (3) delta-minus-def)
  unfolding delta-def
  by(simp add: divide-simps assms(5−8))

have simp2gx:
```

```
  (x1 * x3' - c * y1 * y3') * local.delta-minus x1' y1' x3 y3 *
  (local.delta x1 y1 x2 y2 * local.delta x2 y2 x3 y3) =
  (x1 * (x2 * x3 - c * y2 * y3) * local.delta-plus x2 y2 x3 y3 -
   c * y1 * (x2 * y3 + y2 * x3) * local.delta-minus x2 y2 x3 y3) *
  (local.delta-minus x1 y1 x2 y2 * local.delta-plus x1 y1 x2 y2 -
   d * (x1 * x2 - c * y1 * y2) * (x1 * y2 + y1 * x2) * x3 * y3)
 apply((subst x1'-expr)+, (subst y1'-expr)+,(subst x3'-expr)+,(subst y3'-expr)+)
 apply((subst delta-minus-def[symmetric])+,(subst delta-plus-def[symmetric])+)
  apply(subst (3) delta-minus-def)
  unfolding delta-def
  by(simp add: divide-simps assms(5-8))

have gxpoly = gxpoly-expr
  unfolding gxpoly-def g_x-def Delta_x-def
  apply(simp add: assms(1,2))
  apply(subst delta-minus-def[symmetric])+
  apply(simp add: divide-simps assms(9,11))
  apply(subst (3) left-diff-distrib)
  apply(simp add: simp1gx simp2gx)
  unfolding delta-minus-def delta-plus-def
  unfolding gxpoly-expr-def
  by algebra

obtain r1x r2x r3x where gxpoly = r1x * e1 + r2x * e2 + r3x * e3
  using ‹gxpoly = gxpoly-expr› gx-div by auto
then have gxpoly = 0
  using e1-def assms(13-15) e2-def e3-def by auto
have Delta_x ≠ 0
  using Delta_x-def delta-def assms(7-11) non-unfolded-adds by auto
then have g_x = 0
  using ‹gxpoly = 0› gxpoly-def by auto

have simp1gy: (x1' * y3 + y1' * x3) * local.delta-plus x1 y1 x3' y3' *
  (local.delta x1 y1 x2 y2 * local.delta x2 y2 x3 y3) =
  ((x1 * x2 - c * y1 * y2) * y3 * local.delta-plus x1 y1 x2 y2 +
   (x1 * y2 + y1 * x2) * x3 * local.delta-minus x1 y1 x2 y2) *
  (local.delta-minus x2 y2 x3 y3 * local.delta-plus x2 y2 x3 y3 +
   d * x1 * y1 * (x2 * x3 - c * y2 * y3) * (x2 * y3 + y2 * x3))
 apply((subst x1'-expr)+, (subst y1'-expr)+,(subst x3'-expr)+,(subst y3'-expr)+)
 apply((subst delta-minus-def[symmetric])+,(subst delta-plus-def[symmetric])+)
  apply(subst (2) delta-plus-def)
  unfolding delta-def
  by(simp add: divide-simps assms(5-8))

have simp2gy: (x1 * y3' + y1 * x3') * local.delta-plus x1' y1' x3 y3 *
  (local.delta x1 y1 x2 y2 * local.delta x2 y2 x3 y3) =
   (x1 * (x2 * y3 + y2 * x3) * local.delta-minus x2 y2 x3 y3 +
   y1 * (x2 * x3 - c * y2 * y3) * local.delta-plus x2 y2 x3 y3) *
  (local.delta-minus x1 y1 x2 y2 * local.delta-plus x1 y1 x2 y2 +
```

```
    d * (x1 * x2 - c * y1 * y2) * (x1 * y2 + y1 * x2) * x3 * y3)
   apply((subst x1'-expr)+, (subst y1'-expr)+,(subst x3'-expr)+,(subst y3'-expr)+)
   apply((subst delta-minus-def[symmetric])+,(subst delta-plus-def[symmetric])+)
    apply(subst (3) delta-plus-def)
    unfolding delta-def
    by(simp add: divide-simps assms(5-8))

  have gypoly = gypoly-expr
    unfolding gypoly-def g_y-def Delta_y-def
    apply(simp add: assms(1,2))
    apply(subst delta-plus-def[symmetric])+
    apply(simp add: divide-simps assms(10,12))
    apply(subst left-diff-distrib)
    apply(simp add: simp1gy simp2gy)
    unfolding delta-minus-def delta-plus-def
    unfolding gypoly-expr-def
    by algebra

  obtain r1y r2y r3y where gypoly = r1y * e1 + r2y * e2 + r3y * e3
    using ‹gypoly = gypoly-expr› gy-div by auto
  then have gypoly = 0
    using e1-def assms(13-15) e2-def e3-def by auto
  have Delta_y ≠ 0
    using Delta_y-def delta-def assms(7-12) non-unfolded-adds by auto
  then have g_y = 0
    using ‹gypoly = 0› gypoly-def by auto

  show ?thesis
    using ‹g_y = 0› ‹g_x = 0›
    unfolding g_x-def g_y-def assms(3,4)
    by (simp add: prod-eq-iff)
qed

lemma neutral: add z (1,0) = z by(cases z,simp)

lemma inverse:
  assumes e a b = 0 delta-plus a b a b ≠ 0
  shows add (a,b) (a,-b) = (1,0)
  using assms by(simp add: delta-plus-def e-def,algebra)

corollary
  assumes e a b = 0 delta-plus a b a b ≠ 0
  shows delta-minus a b a (-b) ≠ 0
  using inverse[OF assms] assms(1) unfolding e-def delta-def delta-plus-def delta-minus-def
  by(simp)


lemma affine-closure:
  assumes delta x1 y1 x2 y2 = 0 e x1 y1 = 0 e x2 y2 = 0
```

```
  shows ∃ b. (1/d = b^2 ∧ 1/d ≠ 0) ∨ (1/(c*d) = b^2 ∧ 1/(c*d) ≠ 0)
proof −
  define r where r = (1 − c*d*y1^2*y2^2) * (1 − d*y1^2*x2^2)
  define e1 where e1 = e x1 y1
  define e2 where e2 = e x2 y2
  have r = d^2 * y1^2 * y2^2 * x2^2 * e1 + (1 − d * y1^2) * delta x1 y1 x2 y2
− d * y1^2 * e2
    unfolding r-def e1-def e2-def delta-def delta-plus-def delta-minus-def e-def
    by algebra
  then have r = 0
    using assms e1-def e2-def by simp
  then have cases: (1 − c*d*y1^2*y2^2) = 0 ∨ (1 − d*y1^2*x2^2) = 0
    using r-def by auto
  have d ≠ 0 using ‹r = 0› r-def by auto
  {assume (1 − d*y1^2*x2^2) = 0
  then have 1/d = y1^2*x2^2 1/d ≠ 0
    by(auto simp add: divide-simps ‹d ≠ 0›,argo)}
  note case1 = this
  {assume (1 − c*d*y1^2*y2^2) = 0 (1 − d*y1^2*x2^2) ≠ 0
    then have c ≠ 0 by auto
    then have 1/(c*d) = y1^2*y2^2 1/(c*d) ≠ 0
      apply(simp add: divide-simps ‹d ≠ 0› ‹c ≠ 0›)
      using ‹(1 − c*d*y1^2*y2^2) = 0› apply argo
      using ‹c ≠ 0› ‹d ≠ 0› by auto
  }
  note case2 = this

  show ∃ b. (1/d = b^2 ∧ 1/d ≠ 0) ∨ (1/(c*d) = b^2 ∧ 1/(c*d) ≠ 0)
    using cases case1 case2 by (metis power-mult-distrib)
qed

lemma delta-non-zero:
  fixes x1 y1 x2 y2
  assumes e x1 y1 = 0 e x2 y2 = 0
  assumes ∃ b. 1/c = b^2 ¬ (∃ b. b ≠ 0 ∧ 1/d = b^2)
  shows delta x1 y1 x2 y2 ≠ 0
proof(rule ccontr)
  assume ¬ delta x1 y1 x2 y2 ≠ 0
  then have delta x1 y1 x2 y2 = 0 by blast
  then have ∃ b. (1/d = b^2 ∧ 1/d ≠ 0) ∨ (1/(c*d) = b^2 ∧ 1/(c*d) ≠ 0)
   using affine-closure[OF ‹delta x1 y1 x2 y2 = 0›
                         ‹e x1 y1 = 0› ‹e x2 y2 = 0›] by blast
  then obtain b where (1/(c*d) = b^2 ∧ 1/(c*d) ≠ 0)
   using ‹¬ (∃ b. b ≠ 0 ∧ 1/d = b^2)› by fastforce
  then have 1/c ≠ 0 c ≠ 0 d ≠ 0 1/d ≠ 0 by simp+
  then have 1/d = b^2 / (1/c)
   apply(simp add: divide-simps)
  by (metis ‹1 / (c * d) = b² ∧ 1 / (c * d) ≠ 0› eq-divide-eq semiring-normalization-rules(18))
  then have ∃ b. b ≠ 0 ∧ 1/d = b^2
```

```
    using assms(3)
    by (metis ‹1 / d ≠ 0› power-divide zero-power2)
  then show False
    using ‹¬ (∃ b. b ≠ 0 ∧ 1/d = b^2)› by blast
qed

lemma group-law:
  assumes ∃ b. 1/c = b^2 ¬ (∃ b. b ≠ 0 ∧ 1/d = b^2)
  shows comm-group (|carrier = {(x,y). e x y = 0}, mult = add, one = (1,0)|)
proof(unfold-locales)
  {fix x1 y1 x2 y2
  assume e x1 y1 = 0 e x2 y2 = 0
  have e (fst (add (x1,y1) (x2,y2))) (snd (add (x1,y1) (x2,y2))) = 0
    apply(simp)
     using add-closure delta-non-zero[OF ‹e x1 y1 = 0› ‹e x2 y2 = 0› assms(1)
assms(2)]
          delta-def ‹e x1 y1 = 0› ‹e x2 y2 = 0› by auto}
  then show
      ⋀x y. x ∈ carrier (|carrier = {(x, y). local.e x y = 0}, mult = local.add, one
= (1, 0)|) ⟹
             y ∈ carrier (|carrier = {(x, y). local.e x y = 0}, mult = local.add, one
= (1, 0)|) ⟹
           x ⊗(|carrier = {(x, y). local.e x y = 0}, mult = local.add, one = (1, 0)|) y
           ∈ carrier (|carrier = {(x, y). local.e x y = 0}, mult = local.add, one =
(1, 0)|) by auto
next
  {fix x1 y1 x2 y2 x3 y3
   assume e x1 y1 = 0 e x2 y2 = 0 e x3 y3 = 0
   then have delta x1 y1 x2 y2 ≠ 0 delta x2 y2 x3 y3 ≠ 0
     using assms delta-non-zero by blast+
   fix x1' y1' x3' y3'
   assume (x1',y1') = add (x1,y1) (x2,y2)
          (x3',y3') = add (x2,y2) (x3,y3)
   then have e x1' y1' = 0 e x3' y3' = 0
     using add-closure ‹delta x1 y1 x2 y2 ≠ 0› ‹delta x2 y2 x3 y3 ≠ 0›
           ‹e x1 y1 = 0› ‹e x2 y2 = 0› ‹e x3 y3 = 0› delta-def by fastforce+
   then have delta x1' y1' x3 y3 ≠ 0 delta x1 y1 x3' y3' ≠ 0
     using assms delta-non-zero ‹e x3 y3 = 0› apply blast
    by (simp add: ‹e x1 y1 = 0› ‹e x3' y3' = 0› assms delta-non-zero)

  have add (add (x1,y1) (x2,y2)) (x3,y3) =
        add (x1,y1) (local.add (x2,y2) (x3,y3))
    using associativity
    by (metis ‹(x1', y1') = add (x1, y1) (x2, y2)› ‹(x3', y3') = add (x2, y2) (x3,
y3)› ‹delta x1 y1 x2 y2 ≠ 0›
              ‹delta x1 y1 x3' y3' ≠ 0› ‹delta x1' y1' x3 y3 ≠ 0› ‹delta x2 y2 x3 y3
≠ 0› ‹e x1 y1 = 0›
              ‹e x2 y2 = 0› ‹e x3 y3 = 0› delta-def mult-eq-0-iff)}
```

then show
$\bigwedge x\ y\ z.$
$x \in carrier\ (|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|) \Longrightarrow$
$y \in carrier\ (|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|) \Longrightarrow$
$z \in carrier\ (|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|) \Longrightarrow$
$x \otimes_{(|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)}$
$y \otimes_{(|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)}$
$z =$
$x \otimes_{(|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)}$
$(y \otimes_{(|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)}$
$z)$ **by** *auto*
next
show
$\mathbf{1}_{(|carrier = \{(x, y).\ e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)}$
$\in carrier\ (|carrier = \{(x, y).\ e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)$
by (*simp add*: *e-def*)
next
show
$\bigwedge x.\ x \in carrier\ (|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|) \Longrightarrow$
$\mathbf{1}_{(|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)} \otimes_{(|carrier = \{(x, y).\ local.e\ x\ y = 0\}, m}$
$x = x$
by (*simp add*: *commutativity neutral*)
next
show $\bigwedge x.\ x \in carrier\ (|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|) \Longrightarrow$
$x \otimes_{(|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)}$
$\mathbf{1}_{(|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)} = x$
by (*simp add*: *neutral*)
next
show $\bigwedge x\ y.\ x \in carrier\ (|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|) \Longrightarrow$
$y \in carrier\ (|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|) \Longrightarrow$
$x \otimes_{(|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)} y =$
$y \otimes_{(|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)} x$
using *commutativity* **by** *auto*
next
show
$carrier\ (|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)$
$\subseteq Units\ (|carrier = \{(x, y).\ local.e\ x\ y = 0\}, mult = local.add, one = (1, 0)|)$
proof(*simp*,*standard*)
fix z
assume $z \in \{(x, y).\ local.e\ x\ y = 0\}$
show $z \in Units$

```
      (|carrier = {(x, y). local.e x y = 0}, mult = local.add,
        one = (1, 0)|)
    unfolding Units-def
  proof(simp, cases z, rule conjI)
    fix x y
    assume z = (x,y)
    from this ‹z ∈ {(x, y). local.e x y = 0}›
    show case z of (x, y) ⇒ local.e x y = 0 by blast
    then obtain x y where z = (x,y) e x y = 0 by blast
    have e x (−y) = 0
      using ‹e x y = 0› unfolding e-def by simp
    have add (x,y) (x,−y) = (1,0)
      using inverse[OF ‹e x y = 0› ] delta-non-zero[OF ‹e x y = 0› ‹e x y = 0›
assms] delta-def by fastforce
    then have add (x,−y) (x,y) = (1,0) by simp
    show ∃ a b. e a b = 0 ∧
              add (a, b) z = (1, 0) ∧
              add z (a, b) = (1, 0)
      using ‹add (x, y) (x, − y) = (1, 0)›
            ‹e x (− y) = 0› ‹z = (x, y)› by fastforce
  qed
 qed
qed
end
```

2 Projective curves

```
locale ext-curve-addition = curve-addition +
  assumes c-eq-1: c = 1
  assumes t-intro: ∃ b'. d = (b')^2
  assumes t-ineq: sqrt(d)^2 ≠ 1 sqrt(d) ≠ 0
begin

definition t where t = sqrt(d)
definition e' where e' x y = x^2 + y^2 − 1 − t^2 ∗ x^2 ∗ y^2

lemma c-d-pos: d ≥ 0 using t-intro by auto

lemma delta-plus-self: delta-plus x0 y0 x0 y0 ≠ 0
  unfolding delta-plus-def
  apply(subst (1) mult.assoc,subst (2) mult.assoc,subst (1) mult.assoc)
  apply(subst power2-eq-square[symmetric])
  using mult-nonneg-nonneg[OF c-d-pos zero-le-power2[of x0∗y0]] by auto

lemma t-nz: t ≠ 0 using t-def t-ineq(2) by auto

lemma d-nz: d ≠ 0 using t-def t-nz by simp

lemma t-expr: t^2 = d t^4 = d^2 using t-def t-intro by auto
```

lemma *e-e′-iff*: $e\ x\ y = 0 \longleftrightarrow e'\ x\ y = 0$
unfolding *e-def e′-def* **using** *c-eq-1 t-expr(1)* **by** *simp*

lemma *t-sq-n1*: $t\hat{}2 \neq 1$ **using** *t-ineq(1) t-def* **by** *simp*

The case $t\hat{}2 = 1$ corresponds to a product of intersecting lines which cannot be a group

lemma *t-2-1-lines*:
$t\hat{}2 = 1 \Longrightarrow e'\ x\ y = -\ (1 - x\hat{}2) * (1 - y\hat{}2)$
unfolding *e′-def* **by** *algebra*

The case $t = 0$ corresponds to a circle which has been treated before

lemma *t-0-circle*:
$t = 0 \Longrightarrow e'\ x\ y = x\hat{}2 + y\hat{}2 - 1$
unfolding *e′-def* **by** *auto*

fun ϱ :: *real* $\times$ *real* $\Rightarrow$ *real* $\times$ *real* **where**
$\varrho\ (x,y) = (-y,x)$
fun τ :: *real* $\times$ *real* $\Rightarrow$ *real* $\times$ *real* **where**
$\tau\ (x,y) = (1/(t*x),1/(t*y))$

lemma *tau-sq*: $(\tau \circ \tau)\ (x,y) = (x,y)$ **by**(*simp add*: *t-nz*)

lemma *tau-idemp*: $\tau \circ \tau = id$
using *t-nz comp-def* **by** *auto*

fun i :: *real* $\times$ *real* $\Rightarrow$ *real* $\times$ *real* **where**
$i\ (a,b) = (a,-b)$

fun *ext-add* :: *real* $\times$ *real* $\Rightarrow$ *real* $\times$ *real* $\Rightarrow$ *real* $\times$ *real* **where**
ext-add $(x1,y1)\ (x2,y2) =$
$((x1*y1-x2*y2)$ *div* $(x2*y1-x1*y2)$,
$(x1*y1+x2*y2)$ *div* $(x1*x2+y1*y2))$

lemma *ext-add-comm*:
ext-add $(x1,y1)\ (x2,y2) =$ *ext-add* $(x2,y2)\ (x1,y1)$
by(*simp add*: *divide-simps*,*argo*)

lemma *inversion-invariance-1*:
assumes $x1 \neq 0\ y1 \neq 0\ x2 \neq 0\ y2 \neq 0$
shows *add* $(\tau\ (x1,y1))\ (x2,y2) =$ *add* $(x1,y1)\ (\tau\ (x2,y2))$
apply(*simp*)
apply(*subst c-eq-1*)+
apply(*simp add*: *algebra-simps*)
apply(*subst power2-eq-square*[*symmetric*])+
apply(*subst t-expr*)+
apply(*rule conjI*)
apply(*simp add*: *divide-simps assms t-nz d-nz*)

```
  apply(simp add: algebra-simps)
  apply(simp add: divide-simps assms t-nz d-nz)
  by(simp add: algebra-simps)

lemma inversion-invariance-2:
  assumes x1 ≠ 0 y1 ≠ 0 x2 ≠ 0 y2 ≠ 0
  shows ext-add (τ (x1,y1)) (x2,y2) = ext-add (x1,y1) (τ (x2,y2))
  apply(simp add: algebra-simps)
  apply(subst power2-eq-square[symmetric])+
  apply(subst t-expr)+
  apply(rule conjI)
  apply(simp add: divide-simps assms t-nz d-nz)
  apply(simp add: algebra-simps)
  apply(simp add: divide-simps assms t-nz d-nz)
  by(simp add: algebra-simps)

lemma rotation-invariance-1:
  add (ϱ (x1,y1)) (x2,y2) =
   ϱ (fst (add (x1,y1) (x2,y2)),snd (add (x1,y1) (x2,y2)))
  apply(simp)
  apply(subst c-eq-1)+
  by(simp add: algebra-simps divide-simps)

lemma rotation-invariance-2:
  ext-add (ϱ (x1,y1)) (x2,y2) =
   ϱ (fst (ext-add (x1,y1) (x2,y2)),snd (ext-add (x1,y1) (x2,y2)))
  by(simp add: algebra-simps divide-simps)

definition delta-x :: real ⇒ real ⇒ real ⇒ real ⇒ real where
  delta-x x1 y1 x2 y2 = x2*y1 − x1*y2
definition delta-y :: real ⇒ real ⇒ real ⇒ real ⇒ real where
  delta-y x1 y1 x2 y2 = x1*x2 + y1*y2
definition delta' :: real ⇒ real ⇒ real ⇒ real ⇒ real where
  delta' x1 y1 x2 y2 = delta-x x1 y1 x2 y2 * delta-y x1 y1 x2 y2

lemma rotation-invariance-3:
  delta x1 y1 (fst (ϱ (x2,y2))) (snd (ϱ (x2,y2))) =
   delta x1 y1 x2 y2
  by(simp add: delta-def delta-plus-def delta-minus-def,argo)

lemma rotation-invariance-4:
  delta' x1 y1 (fst (ϱ (x2,y2))) (snd (ϱ (x2,y2))) =
  − delta' x1 y1 x2 y2
  by(simp add: delta'-def delta-x-def delta-y-def,argo)

lemma inverse-rule-1:
  (τ ∘ i ∘ τ) (x,y) = i (x,y) by (simp add: t-nz)
lemma inverse-rule-2:
  (ϱ ∘ i ∘ ϱ) (x,y) = i (x,y) by simp
```

lemma *inverse-rule-3*:
i (*add* (*x1*,*y1*) (*x2*,*y2*)) = *add* (*i* (*x1*,*y1*)) (*i* (*x2*,*y2*))
by(*simp add*: *divide-simps*)
lemma *inverse-rule-4*:
i (*ext-add* (*x1*,*y1*) (*x2*,*y2*)) = *ext-add* (*i* (*x1*,*y1*)) (*i* (*x2*,*y2*))
by(*simp add*: *algebra-simps divide-simps*)

lemma *coherence-1*:
assumes *delta-x x1 y1 x2 y2* ≠ *0 delta-minus x1 y1 x2 y2* ≠ *0*
assumes *e′ x1 y1* = *0 e′ x2 y2* = *0*
shows *delta-x x1 y1 x2 y2* ∗ *delta-minus x1 y1 x2 y2* ∗
(*fst* (*ext-add* (*x1*,*y1*) (*x2*,*y2*)) − *fst* (*add* (*x1*,*y1*) (*x2*,*y2*)))
= *x2* ∗ *y2* ∗ *e′ x1 y1* − *x1* ∗ *y1* ∗ *e′ x2 y2*
apply(*simp*)
apply(*subst* (*2*) *delta-x-def*[*symmetric*])
apply(*subst delta-minus-def*[*symmetric*])
apply(*simp add*: *c-eq-1 assms*(*1*,*2*) *divide-simps*)
unfolding *delta-minus-def delta-x-def e′-def*
apply(*subst t-expr*)+
by(*simp add*: *power2-eq-square field-simps*)

lemma *coherence-2*:
assumes *delta-y x1 y1 x2 y2* ≠ *0 delta-plus x1 y1 x2 y2* ≠ *0*
assumes *e′ x1 y1* = *0 e′ x2 y2* = *0*
shows *delta-y x1 y1 x2 y2* ∗ *delta-plus x1 y1 x2 y2* ∗
(*snd* (*ext-add* (*x1*,*y1*) (*x2*,*y2*)) − *snd* (*add* (*x1*,*y1*) (*x2*,*y2*)))
= − *x2* ∗ *y2* ∗ *e′ x1 y1* − *x1* ∗ *y1* ∗ *e′ x2 y2*
apply(*simp*)
apply(*subst* (*2*) *delta-y-def*[*symmetric*])
apply(*subst delta-plus-def*[*symmetric*])
apply(*simp add*: *c-eq-1 assms*(*1*,*2*) *divide-simps*)
unfolding *delta-plus-def delta-y-def e′-def*
apply(*subst t-expr*)+
by(*simp add*: *power2-eq-square field-simps*)

lemma *coherence*:
assumes *delta x1 y1 x2 y2* ≠ *0 delta′ x1 y1 x2 y2* ≠ *0*
assumes *e′ x1 y1* = *0 e′ x2 y2* = *0*
shows *ext-add* (*x1*,*y1*) (*x2*,*y2*) = *add* (*x1*,*y1*) (*x2*,*y2*)
using *coherence-1 coherence-2 delta-def delta′-def assms* **by** *auto*

lemma *ext-add-closure*:
assumes *delta′ x1 y1 x2 y2* ≠ *0*
assumes *e′ x1 y1* = *0 e′ x2 y2* = *0*
assumes (*x3*,*y3*) = *ext-add* (*x1*,*y1*) (*x2*,*y2*)
shows *e′ x3 y3* = *0*
proof −

```
  have deltas-nz: delta-x x1 y1 x2 y2 ≠ 0
                  delta-y x1 y1 x2 y2 ≠ 0
    using assms(1) delta'-def by auto

  define closure1 where closure1 =
    2 − t^2 + t^2 ∗ x1^2 − 2 ∗ x2^2 − t^2 ∗ x1^2 ∗ x2^2 +
    t^2 ∗ x2^4 + t^2 ∗ y1^2 + t^4 ∗ x1^2 ∗ y1^2 −
    t^2 ∗ x2^2 ∗ y1^2 − 2 ∗ y2^2 − t^2 ∗ x1^2 ∗ y2^2 +
    (4 ∗ t^2 − 2 ∗ t^4) ∗ x2^2 ∗ y2^2 − t^2 ∗ y1^2 ∗ y2^2 +
    t^2 ∗ y2^4

  define closure2 where closure2 =
    −2 + t^2 + (2 − 2 ∗ t^2) ∗ x1^2 + t^2 ∗ x1^4 + t^2 ∗ x2^2 −
    t^2 ∗ x1^2 ∗ x2^2 + (2 − 2 ∗ t^2) ∗ y1^2 − t^2 ∗ x2^2 ∗ y1^2 +
    t^2 ∗ y1^4 + t^2 ∗ y2^2 − t^2 ∗ x1^2 ∗ y2^2 + t^4 ∗ x2^2 ∗ y2^2 −
    t^2 ∗ y1^2 ∗ y2^2

  define p where p =
    −1 ∗ t^4 ∗ (x1^2 ∗ x2^4 ∗ y1^2 −x1^4 ∗ x2^2 ∗ y1^2 +
    t^2 ∗ x1^4 ∗ y1^4 − x1^2 ∗ x2^2 ∗ y1^4 + x1^4 ∗ x2^2 ∗ y2^2 −
    x1^2 ∗ x2^4 ∗ y2^2 − x1^4 ∗ y1^2 ∗ y2^2 + 4 ∗ x1^2 ∗ x2^2 ∗ y1^2 ∗ y2^2
−
     2 ∗ t^2 ∗ x1^2 ∗ x2^2 ∗ y1^2 ∗ y2^2 − x2^4 ∗ y1^2 ∗ y2^2 − x1^2 ∗ y1^4
∗ y2^2 +
     x2^2 ∗ y1^4 ∗ y2^2 − x1^2 ∗ x2^2 ∗ y2^4 + t^2 ∗ x2^4 ∗ y2^4 + x1^2 ∗
y1^2 ∗ y2^4 −
    x2^2 ∗ y1^2 ∗ y2^4)

  have v3: x3 = fst (ext-add (x1,y1) (x2,y2))
           y3 = snd (ext-add (x1,y1) (x2,y2))
    using assms(4) by simp+

  have t^4 ∗ (delta-x x1 y1 x2 y2)^2 ∗ (delta-y x1 y1 x2 y2)^2 ∗ e' x3 y3 = p
    unfolding e'-def v3
    apply(simp)
    apply(subst (2) delta-x-def[symmetric])+
    apply(subst (2) delta-y-def[symmetric])+
    apply(subst power-divide)+
    apply(simp add: divide-simps deltas-nz)
    unfolding p-def delta-x-def delta-y-def
    by algebra
  also have ... = closure1 ∗ e' x1 y1 +  closure2 ∗ e' x2 y2
    unfolding p-def e'-def closure1-def closure2-def by algebra
  finally have t^4 ∗ (delta-x x1 y1 x2 y2)^2 ∗ (delta-y x1 y1 x2 y2)^2 ∗ e' x3 y3
=
               closure1 ∗ e' x1 y1 +  closure2 ∗ e' x2 y2
    by blast

  then show e' x3 y3 = 0
```

```
    using assms(2,3) deltas-nz t-nz by auto
qed


end

locale projective-curve =
 ext-curve-addition
begin
  definition e-aff = {(x,y). e' x y = 0}
  definition e-circ = {(x,y). x ≠ 0 ∧ y ≠ 0 ∧ (x,y) ∈ e-aff}

  lemma group (BijGroup (Reals × Reals))
    using group-BijGroup by blast

  lemma bij-ϱ: bij-betw ϱ ((Reals − {0}) × (Reals − {0}))
                         ((Reals − {0}) × (Reals − {0}))
    unfolding bij-betw-def inj-on-def image-def
    apply(rule conjI,safe,auto)
   by (metis Reals-minus-iff add.inverse-neutral equation-minus-iff member-remove
remove-def)

lemma bij-τ: bij-betw τ ((Reals − {0}) × (Reals − {0}))
                        ((Reals − {0}) × (Reals − {0}))
    unfolding bij-betw-def inj-on-def image-def
    apply(rule conjI,safe)
    apply(simp add: t-nz)+
   apply(metis Reals-of-real mult.right-neutral real-scaleR-def scaleR-conv-of-real)
       apply (simp add: t-nz)
   apply (metis Reals-of-real mult.right-neutral real-scaleR-def scaleR-conv-of-real)
     apply (simp add: t-nz)
    apply(simp add: t-nz)
  proof −
    fix a :: real and b :: real
    assume a1: a ≠ 0
    assume a2: (∀x∈ℝ − {0}. a ≠ 1 / (t ∗ x)) ∨ (∀y∈ℝ − {0}. b ≠ 1 / (t ∗
y))
    obtain bb :: bool where
      f3: (¬ bb) = (∀A-x. A-x ∉ ℝ − {0} ∨ 1 / (t ∗ A-x) ≠ a)
      by (metis (full-types))
    have f4: ∀R r ra. ((ra::real) = r ∨ ra ∈ R − {r}) ∨ ra ∉ R
      by blast
    have f5: ∀r. (r::real) ∈ ℝ
    by (metis (no-types) Reals-of-real mult.right-neutral real-scaleR-def scaleR-conv-of-real)
then have f6: ∀r. (r = 0 ∨ bb) ∨ 1 / t / r ≠ a
  using f4 f3 by (metis (no-types) divide-divide-eq-left)
  have f7: ∀r ra. (ra::real) / (ra / r) = r ∨ ra = 0
    by auto
  obtain bba :: bool where
```

```
    f8: (¬ bba) = (∀ X1. X1 ∉ ℝ − {0} ∨ 1 / (t ∗ X1) ≠ b)
    by moura
  then have f9: ∀ r. (r = 0 ∨ bba) ∨ 1 / t / r ≠ b
    using f5 f4 by (metis (no-types) divide-divide-eq-left)
  have ∀ r. (r::real) ∗ 0 = 0 ∨ r = 0
    by linarith
  then have bb
    using f7 f6 a1 by (metis divide-eq-0-iff mult.right-neutral t-nz)
  then show b = 0
    using f9 f8 f7 f3 a2 a1 by (metis divide-eq-0-iff t-nz)
qed

lemma ϱ ∈ carrier (BijGroup
            ((Reals − {0}) × (Reals − {0})))
    unfolding BijGroup-def
    apply(simp)
    unfolding Bij-def extensional-def
    apply(simp, rule conjI)
    defer 1
    using bij-ϱ apply blast
    apply(safe)
     apply (metis Reals-of-real mult.right-neutral real-scaleR-def scaleR-conv-of-real)
   apply (metis Reals-of-real mult.right-neutral real-scaleR-def scaleR-conv-of-real)
    sorry

definition G where
  G ≡ {id,ϱ,ϱ ∘ ϱ,ϱ ∘ ϱ ∘ ϱ,τ,τ ∘ ϱ,τ ∘ ϱ ∘ ϱ,τ ∘ ϱ ∘ ϱ ∘ ϱ}

lemma g-no-fp:
  assumes g ∈ G p ∈ e-circ g p = p
  shows g = id
proof −
  obtain x y where p-def: p = (x,y) by fastforce
  {assume g = ϱ ∨ g = ϱ ∘ ϱ ∨ g = ϱ ∘ ϱ ∘ ϱ
  then consider (1) g = ϱ | (2) g = ϱ ∘ ϱ | (3) g = ϱ ∘ ϱ ∘ ϱ by blast
  note cases = this
  from cases have x = 0
    apply(cases)
    using assms(3) p-def by(simp)+
  from cases have y = 0
    apply(cases)
    using assms(3) p-def by(simp)+
  have p ∉ e-circ using e-circ-def ‹x = 0› ‹y = 0› p-def by blast}
  note rotations = this
  {assume g = τ ∨ g = τ ∘ ϱ ∨ g = τ ∘ ϱ ∘ ϱ ∨ g = τ ∘ ϱ ∘ ϱ ∘ ϱ
  then consider (1) g = τ | (2) g = τ ∘ ϱ | (3) g = τ ∘ ϱ ∘ ϱ | (4) g = τ ∘ ϱ ∘
ϱ ∘ ϱ by blast
  note cases = this
  from cases have 2∗t∗x∗y = 0 ∨ (t∗x^2 ∈ {−1,1} ∧ t∗y^2 ∈ {−1,1})
```

```
    apply(cases)
    using assms(3) p-def
    apply(simp,metis eq-divide-eq mult.left-commute power2-eq-square)
    using assms(3) p-def apply auto[1]
    using assms(3) p-def apply(simp)
    apply (smt c-d-pos real-sqrt-ge-0-iff t-def zero-le-divide-1-iff zero-le-mult-iff)
    using assms(3) p-def by auto[1]
  then have t = 0 ∨ x = 0 ∨ y = 0 ∨
    (t * x^2 = - 1 ∨ t * x^2 = 1) ∧ (t * y^2 = - 1 ∨ t * y^2 = 1)
    unfolding e'-def by(simp)
  then consider (1) t = 0 | (2) x = 0 | (3) y = 0 |
           (4) t * x^2 = - 1 ∧ t * y^2 = - 1 |
           (5) t * x^2 = - 1 ∧ t * y^2 = 1 |
           (6) t * x^2 = 1 ∧ t * y^2 = - 1 |
           (7) t * x^2 = 1 ∧ t * y^2 = 1 by blast
  then have e' x y = 2 * (1 - t) / t ∨ e' x y = 2 * (-1 - t) / t
    unfolding e'-def
    apply(cases)
       apply(simp add: t-nz)
    using assms(2) unfolding e-circ-def p-def apply blast
    using assms(2) unfolding e-circ-def p-def apply blast
   apply (metis abs-of-nonneg c-d-pos c-eq-1 nonzero-mult-div-cancel-left one-neq-neg-one
power2-eq-1-iff power2-minus real-sqrt-abs real-sqrt-ge-0-iff t-def t-intro t-nz zero-le-mult-iff
zero-le-one zero-le-power-eq-numeral)
     apply (metis abs-of-nonneg c-d-pos c-eq-1 one-neq-neg-one power2-eq-1-iff
power2-minus real-sqrt-abs real-sqrt-ge-0-iff t-def t-intro zero-le-mult-iff zero-le-one
zero-le-power-eq-numeral)
     apply (metis abs-of-nonneg c-d-pos c-eq-1 one-neq-neg-one power2-eq-1-iff
power2-minus real-sqrt-abs real-sqrt-ge-0-iff t-def t-intro zero-le-mult-iff zero-le-one
zero-le-power-eq-numeral)
    proof -
      assume as: t * x^2 = 1 ∧ t * y^2 = 1
      then have t^2 * x^2 * y^2 = 1 by algebra
      then have x^2 + y^2 - 1 - t^2 * x^2 * y^2 = x^2 + y^2 - 2 by simp
      also have ... = 2 / t - 2
      proof -
        have x^2 = 1 / t y^2 = 1 / t using as t-nz
          by(simp add: divide-simps,simp add: mult.commute)+
        then show ?thesis by simp
      qed
      also have ... = 2 * (1-t) / t
        using t-nz by(simp add: divide-simps)
      finally show x^2 + y^2 - 1 - t^2 * x^2 * y^2 = 2 * (1 - t) / t ∨
                 x^2 + y^2 - 1 - t^2 * x^2 * y^2 = 2 * (- 1 - t) / t by blast
    qed
  then have e' x y ≠ 0
    using t-sq-n1 t-nz by auto
  then have p ∉ e-circ
    unfolding e-circ-def e-aff-def p-def by blast}
```

```
  note symmetries = this
  from rotations symmetries
  show ?thesis using G-def assms(1,2) by blast
qed

definition symmetries where
  symmetries = {τ,τ ∘ ϱ,τ ∘ ϱ ∘ ϱ,τ ∘ ϱ ∘ ϱ ∘ ϱ}

definition rotations where
  rotations = {id,ϱ,ϱ ∘ ϱ,ϱ ∘ ϱ ∘ ϱ}

lemma tau-rot-sym:
  assumes r ∈ rotations
  shows τ ∘ r ∈ symmetries
  using assms unfolding rotations-def symmetries-def by auto

definition e-aff-0 where
  e-aff-0 = {((x1,y1),(x2,y2)). (x1,y1) ∈ e-aff ∧
                                (x2,y2) ∈ e-aff ∧
                                delta x1 y1 x2 y2 ≠ 0 }

definition e-aff-1 where
  e-aff-1 = {((x1,y1),(x2,y2)). (x1,y1) ∈ e-aff ∧
                                (x2,y2) ∈ e-aff ∧
                                delta' x1 y1 x2 y2 ≠ 0 }

lemma dichotomy-1:
  assumes p ∈ e-aff q ∈ e-aff
  shows (p ∈ e-circ ∧ (∃ g ∈ symmetries. q = (g ∘ i) p)) ∨
        (p,q) ∈ e-aff-0 ∨ (p,q) ∈ e-aff-1
proof −
  obtain x1 y1 where p-def: p = (x1,y1) by fastforce
  obtain x2 y2 where q-def: q = (x2,y2) by fastforce

  consider (1) (p,q) ∈ e-aff-0 |
           (2) (p,q) ∈ e-aff-1 |
           (3) ¬ ((p,q) ∈ e-aff-0) ∧ ¬ ((p,q) ∈ e-aff-1) by blast
  then show ?thesis
  proof(cases)
    case 1 then show ?thesis by blast
  next
    case 2 then show ?thesis by simp
  next
    case 3
    then have delta x1 y1 x2 y2 = 0
      unfolding p-def q-def  e-aff-0-def e-aff-1-def using assms
      by (simp add: assms p-def q-def)
    from 3 have delta' x1 y1 x2 y2 = 0
      unfolding p-def q-def  e-aff-0-def e-aff-1-def using assms
```

```
  by (simp add: assms p-def q-def)
have x1 ≠ 0 y1 ≠ 0 x2 ≠ 0 y2 ≠ 0
  using ‹delta x1 y1 x2 y2 = 0›
  unfolding delta-def delta-plus-def delta-minus-def by auto
then have p ∈ e-circ q ∈ e-circ
  unfolding e-circ-def using assms p-def q-def by blast+
have (∃ g ∈ symmetries. q = (g ∘ i) p)
proof −
  obtain a0 b0 where tq-expr: τ q = (a0,b0) by fastforce
  obtain a1 b1 where p = (a1,b1) by fastforce
  have a0-nz: a0 ≠ 0 b0 ≠ 0
   using ‹τ q = (a0, b0)› ‹x2 ≠ 0› ‹y2 ≠ 0› comp-apply q-def tau-sq by auto
  have a1-nz: a1 ≠ 0 b1 ≠ 0
    using ‹p = (a1, b1)› ‹x1 ≠ 0› ‹y1 ≠ 0› p-def by auto
  define δ' :: real ⇒ real ⇒ real where
    δ'= (λ x0 y0. x0 ∗ y0 ∗ delta-minus a1 b1 (1/(t∗x0)) (1/(t∗y0)))
  define δ-plus :: real ⇒ real ⇒ real where
    δ-plus = (λ x0 y0. t ∗ x0 ∗ y0 ∗ delta-x a1 b1 (1/(t∗x0)) (1/(t∗y0)))
  define δ-minus :: real ⇒ real ⇒ real where
    δ-minus = (λ x0 y0. t ∗ x0 ∗ y0 ∗ delta-y a1 b1 (1/(t∗x0)) (1/(t∗y0)))
  show ?thesis
  proof(cases delta-minus a1 b1 (fst q) (snd q) = 0)
    case True
    then have t1: delta-minus a1 b1 (fst q) (snd q) = 0 by auto
    then show ?thesis
    proof(cases δ-plus a0 b0 = 0)
      case True
      then have cas1: delta-minus a1 b1 (fst q) (snd q) = 0
                    δ-plus a0 b0 = 0 using t1 by auto
      have δ'-expr: δ' a0 b0 = a0∗b0 − a1∗b1
        unfolding δ'-def delta-minus-def
        apply(simp add: algebra-simps a0-nz a1-nz)
        apply(subst power2-eq-square[symmetric],subst t-expr(1))
        by(simp add: d-nz)
      then have eq1': a0∗b0 − a1∗b1 = 0
      proof −
        have (fst q) = (1 / (t ∗ a0))
             (snd q) = (1 / (t ∗ b0))
          using tq-expr q-def tau-sq by auto
        then have δ' a0 b0 = a0 ∗ b0 ∗ delta-minus a1 b1 (fst q) (snd q)
          unfolding δ'-def by auto
        then show ?thesis using δ'-expr cas1 by auto
      qed
      then have eq1: a0 = a1 ∗ (b1 / b0)
        using a0-nz(2) by(simp add: divide-simps)

      have 0 = δ-plus a0 b0
        using cas1 by auto
      also have δ-plus a0 b0 = −a0∗a1+b0∗b1
```

```
      unfolding δ-plus-def delta-x-def
      by(simp add: algebra-simps t-nz a0-nz)
    also have ... = b0*b1 − a1^2 * (b1 / b0)
      by(simp add: divide-simps a0-nz eq1 power2-eq-square[symmetric])
    also have ... = (b1 / b0) * (b0^2 − a1^2)
      apply(simp add: divide-simps a0-nz)
      by(simp add: algebra-simps power2-eq-square[symmetric])
    finally have (b1 / b0) * (b0^2 − a1^2) = 0 by auto
    then have eq2: (b0^2 − a1^2) = 0
      by(simp add: a0-nz a1-nz)

    have a0^2 − b1^2 = a1^2 * (b1^2 / b0^2) − b1^2
      by(simp add: algebra-simps eq1 power2-eq-square)
    also have ... = (b1^2 / b0^2) * (a1^2 − b0^2)
      by(simp add: divide-simps a0-nz right-diff-distrib')
    also have ... = 0
      using eq2 by auto
    finally have eq3: a0^2 − b1^2 = 0 by blast

    from eq2 have pos1: a1 = b0 ∨ a1 = −b0 by algebra
    from eq3 have pos2: a0 = b1 ∨ a0 = −b1 by algebra
    have (a0 = b1 ∧ a1 = b0) ∨ (a0 = −b1 ∧ a1 = −b0)
      using pos1 pos2 eq2 eq3 eq1' by fastforce
    then have (a0,b0) = (b1,a1) ∨ (a0,b0) = (−b1,−a1) by auto
    then have (a0,b0) ∈ {(b1,a1),(−b1,−a1)} by simp
    moreover have {(b1,a1),(−b1,−a1)} ⊆ {i p, (ϱ ∘ i) p, (ϱ ∘ ϱ ∘ i) p, (ϱ
∘ ϱ ∘ ϱ ∘ i) p}
      using ‹p = (a1, b1)› p-def by auto
    ultimately have (a0,b0) ∈ {i p, (ϱ ∘ i) p, (ϱ ∘ ϱ ∘ i) p, (ϱ ∘ ϱ ∘ ϱ ∘ i)
p}
      by blast
    then have (∃ g ∈ rotations. τ q = (g ∘ i) p)
      unfolding rotations-def by (simp add: ‹τ q = (a0, b0)›)
    then obtain g where g ∈ rotations ∧ τ q = (g ∘ i) p
      by blast
    then have q = (τ ∘ g ∘ i) p
      using tau-sq ‹τ q = (a0, b0)› q-def by auto
    then show (∃ g ∈ symmetries. q = (g ∘ i) p)
      unfolding symmetries-def rotations-def
     using tau-rot-sym ‹g ∈ rotations ∧ τ q = (g ∘ i) p› symmetries-def by
blast
  next
    case False
     then have cas2: delta-minus a1 b1 (fst q) (snd q) = 0
                 δ-minus a0 b0 = 0
       using t1 apply blast
       using False δ-minus-def δ-plus-def ‹delta' x1 y1 x2 y2 = 0› ‹p = (a1,
b1)› delta'-def p-def q-def tq-expr by auto
     have δ'-expr: δ' a0 b0 = a0*b0 − a1*b1
```

```
    unfolding δ'-def delta-minus-def
    apply(simp add: algebra-simps a0-nz a1-nz)
    apply(subst power2-eq-square[symmetric],subst t-expr(1))
    by(simp add: d-nz)
  then have eq1': a0*b0 − a1*b1 = 0
  proof −
    have (fst q) = (1 / (t * a0))
         (snd q) = (1 / (t * b0))
     using tq-expr q-def tau-sq by auto
     then have δ' a0 b0 = a0 * b0 * delta-minus a1 b1 (fst q) (snd q)
       unfolding δ'-def by auto
     then show ?thesis using δ'-expr cas2 by auto
   qed
   then have eq1: a0 = a1 * (b1 / b0)
     using a0-nz(2) by(simp add: divide-simps)

   have 0 = δ-minus a0 b0
     using cas2 by auto
   also have δ-minus a0 b0 = a0 * b1 + a1 * b0
     unfolding δ-minus-def delta-y-def
     by(simp add: algebra-simps t-nz a0-nz)

   also have ... = a1 * (b1 / b0) * b1 + a1 * b0
     by(simp add: eq1)
   also have ... = (a1^2 − b0^2)
     sorry
   also have ... = b0*b1 − a1^2 * (b1 / b0)
      sorry
   also have ... = (b1 / b0) * (b0^2 − a1^2)
     apply(simp add: divide-simps a0-nz)
     sorry
   finally have (b1 / b0) * (b0^2 − a1^2) = 0 by auto
   then have eq2: (b0^2 − a1^2) = 0
     by(simp add: a0-nz a1-nz)

 have a0^2 − b1^2 = a1^2 * (b1^2 / b0^2) − b1^2
   by(simp add: algebra-simps eq1 power2-eq-square)
 also have ... = (b1^2 / b0^2) * (a1^2 − b0^2)
   by(simp add: divide-simps a0-nz right-diff-distrib')
 also have ... = 0
   using eq2 by auto
 finally have eq3: a0^2 − b1^2 = 0 by blast

 from eq2 have pos1: a1 = b0 ∨ a1 = −b0 by algebra
 from eq3 have pos2: a0 = b1 ∨ a0 = −b1 by algebra
 have (a0 = b1 ∧ a1 = b0) ∨ (a0 = −b1 ∧ a1 = −b0)
   using pos1 pos2 eq2 eq3 eq1' by fastforce
 then have (a0,b0) = (b1,a1) ∨ (a0,b0) = (−b1,−a1) by auto
 then have (a0,b0) ∈ {(b1,a1),(−b1,−a1)} by simp
```

```
        moreover have {(b1,a1),(−b1,−a1)} ⊆ {i p, (ϱ ∘ i) p, (ϱ ∘ ϱ ∘ i) p, (ϱ
∘ ϱ ∘ ϱ ∘ i) p}
          using ‹p = (a1, b1)› p-def by auto
        ultimately have (a0,b0) ∈ {i p, (ϱ ∘ i) p, (ϱ ∘ ϱ ∘ i) p, (ϱ ∘ ϱ ∘ ϱ ∘ i)
p}
          by blast
        then have (∃ g ∈ rotations. τ q = (g ∘ i) p)
          unfolding rotations-def by (simp add: ‹τ q = (a0, b0)›)
        then obtain g where g ∈ rotations ∧ τ q = (g ∘ i) p
          by blast
        then have q = (τ ∘ g ∘ i) p
          using tau-sq ‹τ q = (a0, b0)› q-def by auto
        then show (∃ g ∈ symmetries. q = (g ∘ i) p)
          unfolding symmetries-def rotations-def
         using tau-rot-sym ‹g ∈ rotations ∧ τ q = (g ∘ i) p› symmetries-def by
blast
        qed
      next
        case False

        then show ?thesis sorry
      qed
    qed
    show ?thesis sorry
  qed
qed

lemma dichotomy-2:
  assumes add (x1,y1) (x2,y2) = (1,0)
          ((x1,y1),(x2,y2)) ∈ e-aff-0
  shows (x2,y2) = i (x1,y1)
  using assms unfolding delta-def delta-plus-def delta-minus-def
                     e-aff-0-def e-aff-def e′-def
  apply(simp)
  apply(rule conjI)
   defer 1

  sorry

lemma add-cancel-2:
  assumes add (x0,y0) (x1,y1) = add (x0,y0) (i (x0,y0))
          ((x0,y0),(x1,y1)) ∈ e-aff-0
  shows (x1,y1) = i (x0,y0)
proof −
  have e x0 y0 = 0
    using assms(2) unfolding e-aff-0-def e-aff-def
    apply(simp)
    using e-e′-iff by blast
  have add (x0, y0) (i (x0, y0)) = (1,0)
```

```
    using inverse[OF ‹e x0 y0 = 0› delta-plus-self] by fastforce
  then have add (x0,y0) (x1,y1) = (1,0) using assms(1) by argo
  then show ?thesis using dichotomy-2[OF - assms(2)] by fast
qed


lemma dichotomy-3:
  assumes delta' x1 y1 x2 y2 ≠ 0
          add (x1,y1) (x2,y2) = (1,0)
          ((x1,y1),(x2,y2)) ∈ e-aff-1
  shows (x2,y2) = i (x1,y1)
  sorry

lemma add-cancel-3:
  assumes ext-add (x0,y0) (x1,y1) = ext-add (x0,y0) (i (x0,y0))
          ((x0,y0),(x1,y1)) ∈ e-aff-1
  shows (x1,y1) = i (x0,y0)
proof -
  have e x0 y0 = 0
    using assms(2) unfolding e-aff-1-def e-aff-def
    apply(simp)
    using e-e'-iff by blast

  oops
```

3 Projective addition

```
definition gluing :: (((real × real) × bit) × ((real × real) × bit)) set where
  gluing = {(((x0,y0),l),((x1,y1),j)).
               ((x0,y0) ∈ e-aff ∧ (x1,y1) ∈ e-aff) ∧
               (((x0,y0) ∈ e-circ ∧ (x1,y1) = τ (x0,y0) ∧ j = l+1) ∨
                ((x0,y0) ∈ e-aff ∧ x0 = x1 ∧ y0 = y1 ∧ l = j))}

lemma gluing-char:
  assumes (((x0,y0),l),((x1,y1),j)) ∈ gluing
  shows ((x0,y0) = (x1,y1) ∧ l = j) ∨
        ((x1,y1) = τ (x0,y0) ∧ l = j+1)
  using assms gluing-def by force+

lemma gluing-char-zero:
  assumes (((x0,y0),l),((x1,y1),j)) ∈ gluing x0 = 0 ∨ y0 = 0
  shows (x0,y0) = (x1,y1) ∧ l = j
proof -
  consider (1) x0 = 0 | (2) y0 = 0 using assms by auto
  then show ?thesis
    apply(cases)
    using assms(1) unfolding gluing-def
    by(simp add: e-circ-def)+
```

qed

definition *Bits* = *range Bit*
definition *e-aff-bit* :: ((*real* × *real*) × *bit*) *set* **where**
e-aff-bit = *e-aff* × *Bits*

lemma *eq-rel*: *equiv e-aff-bit gluing*
  unfolding *equiv-def*
proof(*intro conjI*)
  show *refl-on e-aff-bit gluing*
    unfolding *refl-on-def*
  proof
    show (∀ *x*∈*e-aff-bit*. (*x*, *x*) ∈ *gluing*)
      unfolding *e-aff-bit-def gluing-def* **by** *auto*
    have *range Bit* = (*UNIV*::*bit set*)
      by (*simp add*: *type-definition.Abs-image*[*OF type-definition-bit*])
    show *gluing* ⊆ *e-aff-bit* × *e-aff-bit*
      unfolding *e-aff-bit-def gluing-def Bits-def*
      using ‹*range Bit* = (*UNIV*::*bit set*)› **by** *auto*
  qed

  show *sym gluing*
    unfolding *sym-def gluing-def*
    by(*auto simp add*: *e-circ-def t-nz*)

  show *trans gluing*
    unfolding *trans-def gluing-def*
     by(*auto simp add*: *e-circ-def t-nz*)
qed

definition *e-proj* **where** *e-proj* = *e-aff-bit* // *gluing*

lemma *rho-circ*:
  assumes *p* ∈ *e-circ*
  shows ϱ *p* ∈ *e-circ*
  using *assms* **unfolding** *e-circ-def e-aff-def e*′*-def*
  by(*simp split*: *prod.splits*,*argo*)

lemma *i-circ*:
  assumes (*x*,*y*) ∈ *e-circ*
  shows *i* (*x*,*y*) ∈ *e-circ*
  using *assms* **unfolding** *e-circ-def e-aff-def e*′*-def* **by** *auto*

lemma *rot-circ*:
  assumes *p* ∈ *e-circ tr* ∈ *rotations*
  shows *tr p* ∈ *e-circ*
proof −

```
  consider (1) tr = id | (2) tr = ϱ  | (3) tr = ϱ ∘ ϱ | (4) tr = ϱ ∘ ϱ ∘ ϱ
    using assms(2) unfolding rotations-def by blast
  then show ?thesis by(cases,auto simp add: assms(1) rho-circ)
qed

lemma τ-circ:
  assumes p ∈ e-circ
  shows τ p ∈ e-circ
  using assms unfolding e-circ-def
  apply(simp split: prod.splits)
  apply(simp add: divide-simps t-nz)
  unfolding e-aff-def e′-def
  apply(simp split: prod.splits)
  apply(simp add: divide-simps t-nz)
  apply(subst power-mult-distrib)+
  apply(subst ring-distribs(1)[symmetric])+
  apply(subst (1) mult.assoc)
  apply(subst right-diff-distrib[symmetric])
  apply(simp add: t-nz)
  by(simp add: algebra-simps)

lemma e-proj-eq:
  assumes p ∈ e-proj
  shows ∃ x y l. (p = {((x,y),l)} ∨ p = {((x,y),l),(τ (x,y),l+1)}) ∧ (x,y) ∈ e-aff

proof −
  obtain g where p-expr: p = gluing `` {g} g ∈ e-aff-bit
    using assms unfolding e-proj-def quotient-def by blast+
  then obtain x y l where g-expr: g = ((x,y),l) (x,y) ∈ e-aff
    using e-aff-bit-def by auto
  then have p-simp: p = gluing `` {((x,y),l)} ((x,y),l) ∈ e-aff-bit (x,y) ∈ e-aff
    using p-expr by simp+
  {fix x′ y′ l′
  assume ((x′,y′), l′) ∈ gluing `` {((x,y),l)}
  then have (x′ = x ∧ y′ = y ∧ l′ = l) ∨
        ((x′,y′) = τ (x,y) ∧ l′ = l + 1)
    unfolding gluing-def Image-def by auto}
  note pair-form = this
  have p = {((x,y),l), (τ (x,y), l+1)} ∨ p = {((x,y),l)}
  proof −
    have ((x,y),l) ∈ p
      using p-simp eq-rel unfolding equiv-def refl-on-def by blast
    then show ?thesis using pair-form p-simp by auto
  qed
  then show ?thesis using p-simp by auto
qed

lemma rot-comp:
  assumes t1 ∈ rotations t2 ∈ rotations
```

shows $t1 \circ t2 \in$ *rotations*
using *assms* **unfolding** *rotations-def* **by** *auto*

definition *p-delta* :: $(real \times real) \times bit \Rightarrow (real \times real) \times bit \Rightarrow real$ **where**
p-delta p1 p2 =
delta (*fst* (*fst p1*)) (*snd* (*fst p1*)) (*fst* (*fst p2*)) (*snd* (*fst p2*))

definition *p-delta'* :: $(real \times real) \times bit \Rightarrow (real \times real) \times bit \Rightarrow real$ **where**
p-delta' p1 p2 =
delta' (*fst* (*fst p1*)) (*snd* (*fst p1*)) (*fst* (*fst p2*)) (*snd* (*fst p2*))

partial-function (*option*) *proj-add* ::
$(real \times real) \times bit \Rightarrow (real \times real) \times bit \Rightarrow ((real \times real) \times bit)$ *option* **where**

proj-add p1 p2 =
(
if (*p-delta p1 p2* $\neq$ *0* $\wedge$ *fst p1* $\in$ *e-aff* $\wedge$ *fst p2* $\in$ *e-aff*)
then Some (*add* (*fst p1*) (*fst p2*), (*snd p1*) + (*snd p2*))
else
(
if (*p-delta' p1 p2* $\neq$ *0* $\wedge$ *fst p1* $\in$ *e-aff* $\wedge$ *fst p2* $\in$ *e-aff*)
then Some (*ext-add* (*fst p1*) (*fst p2*), (*snd p1*) + (*snd p2*))
else None
)
)

lemma *proj-add-comm*:
proj-add ((*x0*,*y0*),*l*) ((*x1*,*y1*),*j*) = *proj-add* ((*x1*,*y1*),*j*) ((*x0*,*y0*),*l*)
proof −
have *delta-equiv*:
(*p-delta* ((*x0*,*y0*),*l*) ((*x1*,*y1*),*j*) $\neq$ *0*) = (*p-delta* ((*x1*,*y1*),*j*) ((*x0*,*y0*),*l*) $\neq$ *0*)
(*p-delta'* ((*x0*,*y0*),*l*) ((*x1*,*y1*),*j*) $\neq$ *0*) = (*p-delta'* ((*x1*,*y1*),*j*) ((*x0*,*y0*),*l*) $\neq$ *0*)
unfolding *p-delta-def p-delta'-def delta-def delta-plus-def*
delta-minus-def delta'-def delta-x-def delta-y-def
by *argo*+
consider
(*1*) *p-delta* ((*x0*,*y0*),*l*) ((*x1*,*y1*),*j*) $\neq$ *0* $\wedge$ *fst* ((*x0*,*y0*),*l*) $\in$ *e-aff* $\wedge$ *fst* ((*x1*,*y1*),*j*) $\in$ *e-aff* |
(*2*) *p-delta'* ((*x0*,*y0*),*l*) ((*x1*,*y1*),*j*) $\neq$ *0* $\wedge$ *fst* ((*x0*,*y0*),*l*) $\in$ *e-aff* $\wedge$ *fst* ((*x1*,*y1*),*j*) $\in$ *e-aff* |
(*3*) (*p-delta* ((*x0*,*y0*),*l*) ((*x1*,*y1*),*j*) = *0* $\wedge$ *p-delta'* ((*x0*,*y0*),*l*) ((*x1*,*y1*),*j*) = *0*) $\vee$
fst ((*x0*,*y0*),*l*) $\notin$ *e-aff* $\vee$ *fst* ((*x1*,*y1*),*j*) $\notin$ *e-aff* **by** *blast*
then show *?thesis*

```
  proof(cases)
    case 1
    then show ?thesis
     by(simp add: commutativity delta-equiv proj-add.simps del: add.simps ext-add.simps)

  next
    case 2
    then show ?thesis
       by(simp add: commutativity ext-add-comm delta-equiv proj-add.simps del:
add.simps ext-add.simps)
  next
    case 3
    then show ?thesis
      using 3 proj-add.simps delta-equiv(1) delta-equiv(2) by auto
  qed
qed


definition proj-add-class c1 c2 =
  (((case-prod (λ x y. the (proj-add x y))) ‘ (Map.dom (case-prod proj-add) ∩ (c1
× c2))) // gluing)

lemma proj-add-class-comm:
  proj-add-class c1 c2 = proj-add-class c2 c1
proof −
  {fix c1 c2
  have (λ(x, y). the (proj-add x y)) ‘ (dom (λ(x, y). proj-add x y) ∩ c1 × c2)
      ⊆ (λ(x, y). the (proj-add x y)) ‘ (dom (λ(x, y). proj-add x y) ∩ c2 × c1)
  proof
    {fix x y
    assume (x, y) ∈ (λ(x, y). the (proj-add x y)) ‘ (dom (λ(x, y). proj-add x y)
∩ c1 × c2)
    then obtain d0 d1 where d-expr:
      (d0,d1) ∈ dom (λ(x, y). proj-add x y) ∩ c1 × c2
      (x,y) = the (proj-add d0 d1)
      unfolding image-def by fast
    then have 1: (x,y) = the (proj-add d1 d0)
      using proj-add-comm prod.collapse[symmetric] by metis
    have 2: (d1,d0) ∈ dom (λ(x, y). proj-add x y) ∩ c2 × c1
    proof −
      from d-expr have d-ins: (d0,d1) ∈ dom (λ(x, y). proj-add x y)
                         (d0,d1) ∈ c1 × c2 by auto
      have 1: (d1,d0) ∈ c2 × c1 using d-ins(2) by simp
      have 2: (d1,d0) ∈ dom (λ(x, y). proj-add x y)
        using d-expr ‹(x,y) = the (proj-add d1 d0)› d-ins(1)
        unfolding dom-def
        by(simp,metis prod.collapse proj-add-comm)
      then show ?thesis using 1 by blast
    qed
```

then have $(x,\ y) \in (\lambda(x,\ y).\ \mathit{the}\ (\mathit{proj\text{-}add}\ x\ y))$ ` $(\mathit{dom}\ (\lambda(x,\ y).\ \mathit{proj\text{-}add}\ x\ y) \cap \mathit{c2} \times \mathit{c1})$
unfolding *image-def*
apply(*simp*) **using** *1* **by** *force*}
then show $\bigwedge x.\ x \in (\lambda(x,\ y).\ \mathit{the}\ (\mathit{proj\text{-}add}\ x\ y))$ `
$(\mathit{dom}\ (\lambda(x,\ y).\ \mathit{proj\text{-}add}\ x\ y) \cap \mathit{c1} \times \mathit{c2}) \Longrightarrow$
$x \in (\lambda(x,\ y).\ \mathit{the}\ (\mathit{proj\text{-}add}\ x\ y))$ `
$(\mathit{dom}\ (\lambda(x,\ y).\ \mathit{proj\text{-}add}\ x\ y) \cap \mathit{c2} \times \mathit{c1})$
by (*metis prod.collapse*)
qed}
note *sub* = *this*
from *sub*[*of c1 c2*] *sub*[*of c2 c1*]
show *?thesis*
unfolding *proj-add-class-def* **using** *subset-antisym* **by** *metis*
qed

lemma *rot-tau-com*:
assumes $\mathit{tr} \in \mathit{rotations}$
shows $\mathit{tr} \circ \tau = \tau \circ \mathit{tr}$
using *assms* **unfolding** *rotations-def* **by**(*auto*)

thm (*latex*) *rot-tau-com*

lemma *rot-com*:
assumes $r \in \mathit{rotations}\ r' \in \mathit{rotations}$
shows $r' \circ r = r \circ r'$
using *assms* **unfolding** *rotations-def* **by** *force*

lemma *rot-inv*:
assumes $r \in \mathit{rotations}$
shows $\exists\ r' \in \mathit{rotations}.\ r' \circ r = \mathit{id}$
using *assms* **unfolding** *rotations-def* **by** *force*

lemma *rot-aff*:
assumes $r \in \mathit{rotations}\ p \in \mathit{e\text{-}aff}$
shows $r\ p \in \mathit{e\text{-}aff}$
using *assms* **unfolding** *rotations-def e-aff-def e'-def*
by(*auto simp add*: *semiring-normalization-rules*(*16*))

lemma *group-lem*:
assumes $r' \in \mathit{rotations}\ r \in \mathit{rotations}$
assumes $(r' \circ i)\ (x,y) = (\tau \circ r)\ (i\ (x,\ y))$
shows $\exists\ r''.\ r'' \in \mathit{rotations} \wedge i\ (x,y) = (\tau \circ r'')\ (i\ (x,y))$
proof −
obtain r'' **where** $r'' \circ r' = \mathit{id}\ r'' \in \mathit{rotations}$ **using** *rot-inv assms*(*1*) **by** *blast*
then have $i\ (x,y) = (r'' \circ \tau \circ r)\ (i\ (x,\ y))$
using *assms*(*3*) **by** (*simp*,*metis pointfree-idE*)
then have $i\ (x,y) = (\tau \circ r'' \circ r)\ (i\ (x,\ y))$
using *rot-tau-com*[*OF* ‹$r'' \in \mathit{rotations}$›] **by** *simp*

then show *?thesis* **using** *rot-comp*[*OF* ‹$r'' \in$ *rotations*› *assms*(*2*)] **by** *auto*
qed

lemma *tau-not-id*: $\tau \neq id$
apply(*simp add*: *fun-eq-iff*)
by (*metis c-eq-1 eq-divide-eq-1 mult-cancel-left2 one-power2 t-def t-ineq*(*1*))

lemma *sym-not-id*:
assumes $r \in$ *rotations*
shows $\tau \circ r \neq id$
using *assms* **unfolding** *rotations-def*
apply(*subst fun-eq-iff*,*simp*)
apply(*auto*)
using *tau-not-id* **apply** *auto*[*1*]
apply (*metis d-nz*)
apply (*metis eq-divide-eq-1 minus-mult-minus mult.right-neutral ring-normalization-rules*(*1*) *semiring-normalization-rules*(*29*) *t-expr*(*1*) *t-sq-n1*)
by (*metis d-nz*)

lemma *covering*:
assumes $p \in$ *e-proj* $q \in$ *e-proj*
shows *proj-add-class* $p\ q \neq \{\}$
proof −
have $p \in$ *e-aff-bit* // *gluing*
using *assms*(*1*) **unfolding** *e-proj-def* **by** *blast*
from *e-proj-eq*[*OF assms*(*1*)] *e-proj-eq*[*OF assms*(*2*)]
obtain $x\ y\ l\ x'\ y'\ l'$ **where**
p-q-expr: $p = \{((x, y), l)\} \vee p = \{((x, y), l), (\tau\ (x, y), l + 1)\}$
$q = \{((x', y'), l')\} \vee q = \{((x', y'), l'), (\tau\ (x', y'), l' + 1)\}$
$(x,y) \in$ *e-aff* $(x',y') \in$ *e-aff*
by *blast*
then have *gluings*: $p = ($*gluing* ‘‘ $\{((x,y),l)\})$
$q = ($*gluing* ‘‘ $\{((x',y'),l')\})$
using *assms*(*1*) *assms*(*2*) **unfolding** *e-proj-def*
using *Image-singleton-iff equiv-class-eq-iff*[*OF eq-rel*] *insertI1 quotientE*
by *metis*+
consider
$(x, y) \in$ *e-circ* $\wedge$ ($\exists\, g \in$*symmetries*. $(x', y') = (g \circ i)\ (x, y)$)
| $((x, y), x', y') \in$ *e-aff-0*
| $((x, y), x', y') \in$ *e-aff-1*
using *dichotomy-1*[*OF* ‹$(x,y) \in$ *e-aff*› ‹$(x',y') \in$ *e-aff*›] **by** *blast*
then show *?thesis*
proof(*cases*)
case *1*
then obtain r **where** *eq*: $(x',y') = (\tau \circ r)\ (i\ (x,y))$ $r \in$ *rotations*
unfolding *symmetries-def rotations-def* **by** *force*
then have $\tau \in G$ **unfolding** *G-def* **by** *auto*
have $i\ (x,y) \in$ *e-circ*
using *1* **unfolding** *e-circ-def e-aff-def e'-def* **by** *auto*

then have $(\tau \circ r \circ i)\ (x,\ y) \in$ *e-circ*
  using *i-circ rho-circ rot-circ* τ*-circ eq*(*2*) **by** *auto*
have $\tau\ (x',y') \neq (\tau \circ r \circ i)\ (x,y)$
  unfolding *eq*(*1*)
  using *g-no-fp*[*OF* ‹$\tau \in G$› ‹$(\tau \circ r \circ i)\ (x,\ y) \in$ *e-circ*›]
  apply(*simp*)
  by (*metis* τ*.simps c-eq-1 d-nz divide-divide-eq-left fst-conv id-apply mult.assoc mult-cancel-right1 power2-eq-square semiring-normalization-rules*(*11*) *t-expr*(*1*) *t-sq-n1*)
have $\tau\ (x',y') \in$ *e-aff*
  using ‹$(\tau \circ r \circ i)\ (x,\ y) \in$ *e-circ*› *eq e-circ-def* τ*-circ* **by** *auto*

have $\tau\ (x',y') \in$ *e-circ*
  using τ*-circ* ‹$(\tau \circ r \circ i)\ (x,\ y) \in$ *e-circ*› *eq*(*1*) **by** *auto*
then have $(\tau\ (x',y'),l'+1) \in$ (*gluing* ‘‘ $\{((x',y'),l')\}$)
  unfolding *gluing-def Image-def*
  apply(*simp split*: *prod.splits del*: τ*.simps*,*safe*)
  apply (*simp add*: *p-q-expr*(*4*))
  using ‹$\tau\ (x',\ y') \in$ *e-aff*› **apply** *auto*[*1*]
  using ‹$(\tau \circ r \circ i)\ (x,\ y) \in$ *e-circ*› *eq*(*1*) **by** *auto*
then have *sc*: (*gluing* ‘‘ $\{((x',y'),l')\}$) = (*gluing* ‘‘ $\{(\tau\ (x',y'),l'+1)\}$)
  by (*meson Image-singleton-iff eq-rel equiv-class-eq-iff*)
have *proj-add-class p q* =
    proj-add-class (*gluing* ‘‘ $\{((x,y),l)\}$) (*gluing* ‘‘ $\{((x',y'),l')\}$)
  using *gluings* **by** *simp*
also have ... =
    proj-add-class (*gluing* ‘‘ $\{((x,y),l)\}$) (*gluing* ‘‘ $\{(\tau\ (x',y'),l'+1)\}$)
  using *sc* **by** *simp*
finally have *eq-simp*: *proj-add-class p q* = *proj-add-class* (*gluing* ‘‘ $\{((x,y),l)\}$) (*gluing* ‘‘ $\{(\tau\ (x',y'),l'+1)\}$)
  by *blast*

consider
  $(x,\ y) \in$ *e-circ* $\wedge$ ($\exists\, g\in$*symmetries*. $\tau\ (x',\ y') = (g \circ i)\ (x,\ y)$)
| $((x,\ y),\ \tau\ (x',\ y')) \in$ *e-aff-0*
| $((x,\ y),\ \tau\ (x',\ y')) \in$ *e-aff-1*
  using *dichotomy-1*[*OF* ‹$(x,y) \in$ *e-aff*› ‹$\tau\ (x',\ y') \in$ *e-aff*›] **by** *blast*
then show *?thesis*
proof(*cases*)
  case *1*
  define q' **where** $q' = \tau\ (x',y')$
  from *1* **have** $(x,\ y) \in$ *e-circ* $\wedge$ ($\exists\, g\in$*symmetries*. $q' = (g \circ i)\ (x,\ y)$)
    by(*simp add*: q'*-def*)
  then obtain r' **where** *eq1*: $q' = (\tau \circ r')\ (i\ (x,y))$ $r' \in$ *rotations*
    unfolding *symmetries-def rotations-def* **by** *force*
  then have $\tau\ (x',y') = (\tau \circ r')\ (i\ (x,y))$
    by(*simp add*: q'*-def*)
  then have $(x',y') = (r' \circ i)\ (x,y)$
    using *tau-sq* **apply**(*simp del*: τ*.simps*) **by** (*metis surj-pair*)
  then have $(r' \circ i)\ (x,y) = (\tau \circ r)\ (i\ (x,\ y))$

```
      using eq by simp
    then obtain r'' where eq2: i (x,y) = (τ ∘ r'') (i (x,y)) r'' ∈ rotations
      using group-lem[OF ‹r' ∈ rotations› ‹r ∈ rotations›] by blast
    have τ ∘ r'' ∈ G
      using G-def ‹r'' ∈ rotations› rotations-def
      apply(simp)
     using G-def ‹(r' ∘ i) (x, y) = (τ ∘ r) (i (x, y))› symmetries-def tau-rot-sym
by auto
    have i (x,y) ∈ e-circ
      using ‹i (x, y) ∈ e-circ› by auto
    have τ ∘ r'' ≠ id
      using sym-not-id[OF ‹r'' ∈ rotations›] by blast
    then have False
      using g-no-fp[OF ‹τ ∘ r'' ∈ G› ‹i (x,y) ∈ e-circ› eq2(1)[symmetric]]
      by blast
    then show ?thesis by blast
  next
    case 2
    define x'' where x'' = fst (τ (x',y'))
    define y'' where y'' = snd (τ (x',y'))
    from 2 have delta x y x'' y'' ≠ 0
      unfolding e-aff-0-def using x''-def y''-def by simp
   then obtain v where add-some: proj-add ((x,y),l) ((x'',y''),l'+1) = Some v
      using proj-add.simps[of ((x,y),l) ((x'',y''),l'+1)] p-q-expr
      unfolding p-delta-def
      using ‹τ (x', y') ∈ e-aff› fst-conv x''-def y''-def by auto
    have in-set: (((x,y),l),((x'',y''),l'+1)) ∈ (dom (λ(x, y). proj-add x y) ∩ p ×
q)
      unfolding dom-def using p-q-expr
      apply(simp del: τ.simps)
      apply(rule conjI)
      apply (metis add-some surjective-pairing)
      apply(rule conjI)
      apply blast
       using ‹(τ (x', y'), l' + 1) ∈ gluing `` {((x', y'), l')}› gluings(2) x''-def
y''-def by auto
    then show ?thesis
      unfolding proj-add-class-def
      using add-some in-set by blast
  next
    case 3
    define x'' where x'' = fst (τ (x',y'))
    define y'' where y'' = snd (τ (x',y'))
    from 3 have delta' x y x'' y'' ≠ 0
      unfolding e-aff-1-def using x''-def y''-def by simp
   then obtain v where add-some: proj-add ((x,y),l) ((x'',y''),l'+1) = Some v
      using proj-add.simps[of ((x,y),l) ((x'',y''),l'+1)] p-q-expr
      unfolding p-delta'-def
      using ‹τ (x', y') ∈ e-aff› fst-conv x''-def y''-def
```

```
        by (metis prod.collapse snd-conv)
      have in-set: (((x,y),l),((x'',y''),l'+1)) ∈ (dom (λ(x, y). proj-add x y) ∩ p ×
q)
        unfolding dom-def using p-q-expr
        apply(simp del: τ.simps)
        apply(rule conjI)
        apply (metis add-some surjective-pairing)
        apply(rule conjI)
        apply blast
          using ‹(τ (x', y'), l' + 1) ∈ gluing ‘‘ {((x', y'), l')}› gluings(2) x''-def
y''-def by auto
      then show ?thesis
        unfolding proj-add-class-def
        using add-some in-set by blast
  qed
  next
    case 2
    then have delta x y x' y' ≠ 0
      unfolding e-aff-0-def by simp
    then obtain v where add-some: proj-add ((x,y),l) ((x',y'),l') = Some v
      using proj-add.simps[of ((x,y),l) ((x',y'),l')] p-q-expr
      unfolding p-delta-def by auto
    then have in-set: (((x,y),l),((x',y'),l')) ∈ (dom (λ(x, y). proj-add x y) ∩ p ×
q)
      unfolding dom-def using p-q-expr by fast
    then show ?thesis
      unfolding proj-add-class-def
      using add-some in-set by blast
  next
    case 3
    then have delta' x y x' y' ≠ 0
      unfolding e-aff-1-def by simp
    then obtain v where add-some: proj-add ((x,y),l) ((x',y'),l') = Some v
      using proj-add.simps[of ((x,y),l) ((x',y'),l')] p-q-expr
      unfolding p-delta'-def by fastforce
    then have in-set: (((x,y),l),((x',y'),l')) ∈ (dom (λ(x, y). proj-add x y) ∩ p ×
q)
      unfolding dom-def using p-q-expr by fast
    then show ?thesis
      unfolding proj-add-class-def
      using add-some in-set by blast
  qed
qed

lemma wd-d-nz:
  assumes g ∈ symmetries (x', y') = (g ∘ i) (x, y) (x,y) ∈ e-circ
  shows delta x y x' y' = 0
 using assms unfolding symmetries-def e-circ-def delta-def delta-minus-def delta-plus-def
  by(auto,auto simp add: divide-simps t-nz t-expr(1) power2-eq-square[symmetric]
```

d-nz)

lemma *wd-d′-nz*:
  assumes $g \in$ *symmetries* $(x', y') = (g \circ i)\ (x, y)\ (x,y) \in$ *e-circ*
  shows *delta′* $x\ y\ x'\ y' = 0$
  using *assms* **unfolding** *symmetries-def e-circ-def delta′-def delta-x-def delta-y-def*
  by(*auto*)

lemma *e-aff-x0*:
  assumes $x = 0\ (x,y) \in$ *e-aff*
  shows $y = 1 \vee y = -1$
  using *assms* **unfolding** *e-aff-def e′-def*
  by(*simp*,*algebra*)

lemma *e-aff-y0*:
  assumes $y = 0\ (x,y) \in$ *e-aff*
  shows $x = 1 \vee x = -1$
  using *assms* **unfolding** *e-aff-def e′-def*
  by(*simp*,*algebra*)

lemma *add-ext-add*:
  assumes $x1 \neq 0\ y1 \neq 0\ x2 \neq 0\ y2 \neq 0$
  shows *ext-add* $(x1,y1)\ (x2,y2) = \tau$ (*add* $(\tau\ (x1,y1))\ (x2,y2)$)
  apply(*simp*)
  apply(*rule conjI*)
  apply(*simp add*: *c-eq-1*)
  apply(*simp add*: *divide-simps t-nz power2-eq-square*[*symmetric*] *assms t-expr*(*1*)
d-nz)
  apply(*simp add*: *algebra-simps power2-eq-square*[*symmetric*] *t-expr*(*1*))
  apply(*simp add*: *divide-simps t-nz power2-eq-square*[*symmetric*] *assms t-expr*(*1*)
d-nz)
  by(*simp add*: *algebra-simps power2-eq-square*[*symmetric*] *t-expr*(*1*))

corollary *add-ext-add-2*:
  assumes $x1 \neq 0\ y1 \neq 0\ x2 \neq 0\ y2 \neq 0$
  shows *add* $(x1,y1)\ (x2,y2) = \tau$ (*ext-add* $(\tau\ (x1,y1))\ (x2,y2)$)
proof −
  obtain $x1'\ y1'$ **where** *tau-expr*: $\tau\ (x1,y1) = (x1',y1')$ **by** *simp*
  then have *p-nz*: $x1' \neq 0\ y1' \neq 0$
    using *assms*(*1*) *tau-sq* **apply** *auto*[*1*]
    using ‹$\tau\ (x1,\ y1) = (x1',\ y1')$› *assms*(*2*) *tau-sq* **by** *auto*
  have *add* $(x1,y1)\ (x2,y2) =$ *add* $(\tau\ (x1',\ y1'))\ (x2,\ y2)$
    using *tau-expr tau-idemp*
    by (*metis comp-apply id-apply*)
  also have ... $= \tau$ (*ext-add* $(x1',\ y1')\ (x2,\ y2)$)
    using *add-ext-add*[*OF p-nz assms*(*3*,*4*)] *tau-idemp* **by** *simp*
  also have ... $= \tau$ (*ext-add* $(\tau\ (x1,\ y1))\ (x2,\ y2)$)
    using *tau-expr tau-idemp* **by** *auto*

```
  finally show ?thesis by blast
qed

lemma gluing-inv:
  assumes x ≠ 0 y ≠ 0 (x,y) ∈ e-aff
  shows gluing `` {((x,y),j)} = gluing `` {(τ (x,y),j+1)}
proof
  have tr: τ (x,y) ∈ e-aff τ (x,y) ∈ e-circ
      using e-circ-def assms τ-circ by fastforce+
  show gluing `` {((x,y), j)} ⊆ gluing `` {(τ (x,y), j + 1)}
  proof
    {fix p b
    assume as: (p, b) ∈ gluing `` {((x,y), j)}
    then have (p,b) ∈ e-aff-bit
      unfolding e-aff-bit-def gluing-def
      using as e-aff-bit-def eq-rel equiv-class-eq-iff by fastforce
    have in-glue: (((x,y), j), p, b) ∈ gluing using as by blast
    have (p = (x,y) ∧ b = j) ∨ (p = τ (x,y) ∧ b = j+1)
      using gluing-char in-glue
     by (smt add.assoc add.commute add.left-neutral add.right-neutral bit-add-eq-1-iff
prod.collapse)
    then consider
      (1) p = (x,y) b = j |
      (2) p = τ (x,y) b = j+1 by blast
    then have ((τ (x,y), j + 1), p, b) ∈ gluing
      apply(cases)
      using tr unfolding gluing-def by(simp add: t-nz assms)+
    then have (p, b) ∈ gluing `` {(τ (x,y), j + 1)} by auto}
    then show ⋀xa. xa ∈ gluing `` {((x, y), j)} ⟹
          xa ∈ gluing `` {(τ (x, y), j + 1)} by auto
  qed

  show gluing `` {(τ (x, y), j + 1)} ⊆ gluing `` {((x, y), j)}
  proof
    {fix p b
    assume as: (p, b) ∈ gluing `` {(τ (x, y), j + 1)}
    then have (p,b) ∈ e-aff-bit
      unfolding e-aff-bit-def gluing-def
      using as e-aff-bit-def eq-rel equiv-class-eq-iff by fastforce
    obtain x' y' where p-expr: p = (x',y') by fastforce
    obtain xt yt where tau-expr: τ (x,y) = (xt,yt) by simp
    have in-glue: ((τ (x, y), j + 1), p, b) ∈ gluing using as by blast
    then have in-glue-coord: (((xt,yt), j + 1), (x',y'), b) ∈ gluing
      using ‹p = (x',y')› ‹τ (x,y) = (xt,yt)› by auto
    have (p = (x,y) ∧ b = j) ∨ (p = τ (x,y) ∧ b = j+1)
      using gluing-char[OF in-glue-coord] p-expr tau-expr
      apply(simp add: algebra-simps del: τ.simps)
      using pointfree-idE tau-idemp by force
    then consider
```

```
      (1) p = (x,y) b = j |
      (2) p = τ (x,y) b = j+1 by blast
    then have (((x,y), j), p, b) ∈ gluing
      apply(cases)
      using ‹(p, b) ∈ e-aff-bit› eq-rel equiv-class-eq-iff apply fastforce
      using tr unfolding gluing-def by(simp add: e-circ-def assms)
    then have (p, b) ∈ gluing `` {((x,y), j)} by blast}
    then show ⋀xa. xa ∈ gluing `` {(τ (x, y), j + 1)} ⟹
          xa ∈ gluing `` {((x, y), j)} by auto
  qed
qed



lemma eq-class-simp:
  assumes X ∈ e-proj X ≠ {}
  shows X // gluing = {X}
proof
  have X ∈ e-aff-bit // gluing using ‹X ∈ e-proj› unfolding e-proj-def by blast

  {
    fix x
    assume x ∈ X
    have gluing `` {x} = X
     by (metis (no-types, lifting) Image-singleton-iff ‹x ∈ X› assms(1) e-proj-def
eq-rel equiv-class-eq quotientE)
  }
  note simp-un = this
  show X // gluing ⊆ {X}
    unfolding quotient-def by(simp add: simp-un)
  show {X} ⊆ X // gluing
    unfolding quotient-def by(simp add: simp-un assms)
qed

lemma eq-class-image:
  assumes (x,y) ∈ e-aff
  shows (gluing `` {((x,y), l)}) // gluing =
        {gluing `` {((x,y), l)}}
proof(rule eq-class-simp)
  have ((x,y),l) ∈ e-aff-bit
    using assms unfolding e-aff-bit-def Bits-def
    by (metis Bit-cases SigmaI image-eqI)
  then have gluing `` {((x, y), l)} ∈ e-aff-bit // gluing
    by (simp add: quotientI)
  show gluing `` {((x, y), l)} ≠ {}
    using ‹gluing `` {((x, y), l)} ∈ e-aff-bit // gluing› e-proj-def e-proj-eq
    by fastforce
  show gluing `` {((x, y), l)} ∈ e-proj
    using ‹gluing `` {((x, y), l)} ∈ e-aff-bit // gluing› unfolding e-proj-def
```

by *blast*
qed

lemma *gluing-class*:
assumes $x \neq 0$ $y \neq 0$ $(x,y) \in$ *e-aff*
shows *gluing* `` $\{((x,y), l)\} = \{((x,y), l), (\tau\ (x,y), l + 1)\}$
proof −
have $(x,y) \in$ *e-circ* **using** *assms* **unfolding** *e-circ-def* **by** *blast*
then have $\tau\ (x,y) \in$ *e-aff*
using τ*-circ* **using** *e-circ-def* **by** *force*
show *?thesis*
unfolding *gluing-def Image-def*
apply(*simp split*: *prod.splits add*: *e-circ-def* ‹$\tau\ (x,y) \in$ *e-aff*› *assms del*: τ*.simps* ϱ*.simps*,*safe*)
by(*auto simp del*: τ*.simps*, *simp add*: *assms*,*simp add*: ‹$\tau\ (x,y) \in$ *e-aff*› *del*: τ*.simps*)
qed

lemma *proj-add-class-identity*:
assumes $x \in$ *e-proj*
shows *proj-add-class* (*gluing* `` $\{((1, 0), 0)\}$) $x = \{x\}$
proof −
have $((1,0),0) \in$ *e-aff-bit*
unfolding *e-aff-bit-def e-aff-def e′-def Bits-def*
using *zero-bit-def* **by** *fastforce*
have $(((1, 0), 0), ((1, 0), 0)) \in$ *gluing*
using *eq-rel* ‹$((1,0),0) \in$ *e-aff-bit*›
unfolding *equiv-def refl-on-def* **by** *blast*
have *gluing-one*: *gluing* `` $\{((1, 0), 0)\} = \{((1,0),0)\}$
unfolding *Image-def* **apply**(*simp*)
using *gluing-char-zero* ‹$(((1, 0), 0), ((1, 0), 0)) \in$ *gluing*› **by** *fast*
{ fix *e1 e2 b*
assume $((e1,e2),b) \in x$
then have $((e1,e2),b) \in$ *e-aff-bit*
using *assms* **unfolding** *e-proj-def*
using *eq-rel in-quotient-imp-subset* **by** *blast*
have *1*: *p-delta* $((1,0),0)$ $((e1,e2),b) \neq 0$
unfolding *p-delta-def delta-def delta-plus-def delta-minus-def* **by** *auto*
have *2*: $(e1,e2) \in$ *e-aff* $(1,0) \in$ *e-aff*
using ‹$((e1,e2),b) \in$ *e-aff-bit*› ‹$((1,0),0) \in$ *e-aff-bit*› **unfolding** *e-aff-bit-def* **by** *blast+*
have *proj-add* $((1,0),0)$ $((e1,e2),b) =$ *Some* $((e1,e2),b)$
using *1 2* **by**(*simp add*: *proj-add.simps*)
}
note *sol* = *this*
from *sol* **have** *dom-eq*: (*dom* ($\lambda(x, y)$. *proj-add x y*) $\cap \{((1, 0), 0)\} \times x) = \{((1, 0), 0)\} \times x$
using *assms* **unfolding** *dom-def* **by** *fast*
from *sol* **have** *add-eq*: ($\lambda(x, y)$. *the* (*proj-add x y*)) ` $(\{((1, 0), 0)\} \times x) =$

```
          x by force
  have x ≠ {}
    using assms
    by (metis e-proj-def empty-iff eq-rel equiv-class-self quotientE)
  show ?thesis
    apply(simp add: gluing-one)
    unfolding proj-add-class-def
    by(simp add: dom-eq add-eq eq-class-simp[OF assms ‹x ≠ {}›])
qed

lemma b-cc-case:
  assumes closure-lem: add (x, y) (τ (x', y')) ∈ e-aff
  assumes x-y-aff: (x,y) ∈ e-aff τ (x', y') ∈ e-aff τ (x', y') ∈ e-circ
  assumes cc: x' ≠ 0 y' ≠ 0
  assumes eq: x' ∗ y' ≠ − x ∗ y x' ∗ y' ≠ x ∗ y
  assumes b: ¬ ((x, y) ∈ e-circ ∧ (∃ g∈symmetries. (x', y') = (g ∘ i) (x, y)))
  shows
  fst (add (x,y) (τ (x',y'))) = 0 ∨
  snd (add (x,y) (τ (x',y'))) = 0 ⟹
  (∃ g∈symmetries. (x', y') = (g ∘ i) (x, y))
proof −
  assume as: fst (add (x,y) (τ (x',y'))) = 0 ∨ snd (add (x,y) (τ (x',y'))) = 0
  define r1 where r1 = fst(add (x,y) (τ (x',y')))
  define r2 where r2 = snd(add (x,y) (τ (x',y')))
  from closure-lem have (r1,r2) ∈ e-aff using r1-def r2-def by simp
  have cases: r1 = 0 ∨ r2 = 0
    using as r1-def r2-def by presburger
  {assume r1 = 0
  then have r2 = 1 ∨ r2 = −1
      using ‹(r1,r2) ∈ e-aff› unfolding e-aff-def e'-def
      by(simp,algebra)}
  note case1 = this
  {assume r2 = 0
  then have r1 = 1 ∨ r1 = −1
      using ‹(r1,r2) ∈ e-aff› unfolding e-aff-def e'-def
      by(simp,algebra)}
  note case2 = this
  from case1 case2 cases obtain g where r-expr: g ∈ rotations (r1,r2) = g (1,0)
    unfolding rotations-def by force

  have e-eq: e x y = 0
    using ‹(x,y) ∈ e-aff› e-e'-iff unfolding e-aff-def by simp
  have d-eq: delta-plus x y x y ≠ 0
      unfolding delta-plus-def
      apply(subst (1) mult.assoc,subst (2) mult.assoc,subst (1) mult.assoc)
      apply(subst power2-eq-square[symmetric])
      using mult-nonneg-nonneg[OF c-d-pos zero-le-power2[of x∗y]] by auto
  from r-expr have add (x,y) (τ (x',y')) = g (1,0)
    using r1-def r2-def by simp
```

```
  also have ... = g (add (x,y) (i (x,y)))
    using inverse[OF e-eq d-eq] by fastforce
  also have ... = add (x,y) ((g ∘ i) (x,y))
    using ‹g ∈ rotations› unfolding rotations-def
    apply(auto)
    apply(simp add: c-eq-1)+
    apply(simp add: divide-simps)
    apply(simp add: c-eq-1)+
    by(simp add: algebra-simps divide-simps)
  finally have add-eq: add (x,y) (τ (x',y')) = add (x,y) ((g ∘ i) (x,y))
    by blast
  obtain g' where g' ∈ rotations g ∘ g' = id
    using rot-inv[OF ‹g ∈ rotations›] rot-com[OF ‹g ∈ rotations›] by auto
  then have add (x,y) (g' (τ (x',y'))) = add (x,y) (i (x,y))
  proof −
    have 1: g'(add (x, y) (τ (x', y'))) = add (x, y) (g' (τ (x', y')))
      using ‹g' ∈ rotations› unfolding rotations-def
      apply(auto)
      by(simp add: c-eq-1 divide-simps t-nz cc algebra-simps)+
    have g'(add (x,y) ((g ∘ i) (x,y))) = add (x,y) ((g' ∘ (g ∘ i)) (x,y))
      using ‹g ∈ rotations› ‹g' ∈ rotations› unfolding rotations-def
       by(metis ‹g (add (x, y) (i (x, y))) = add (x, y) ((g ∘ i) (x, y))› ‹g ∘ g' =
id› ‹g' ∈ rotations› comp-apply pointfree-idE r-expr(1) rot-com)
    also have ... = add (x,y) (i (x,y))
      using ‹g ∘ g' = id› rot-com[OF ‹g ∈ rotations› ‹g' ∈ rotations›]
      by(simp del: add.simps add: pointfree-idE)
    finally have g'(add (x,y) ((g ∘ i) (x,y))) = add (x,y) (i (x,y))
      by blast
    then show ?thesis using add-eq 1 by presburger
  qed
  then have g' (τ (x', y')) = i (x,y)
  proof −
    define x2 where x2 = fst(g' (τ (x', y')))
    define y2 where y2 = snd(g' (τ (x', y')))
    have 1: delta x y x2 y2 ≠ 0
      unfolding delta-def delta-plus-def delta-minus-def x2-def y2-def
      using ‹g' ∈ rotations› unfolding rotations-def apply(auto)
     using ‹x' ∗ y' ≠ − x ∗ y› by(simp add: ‹x' ∗ y' ≠ x ∗ y› divide-simps cc t-nz

                                   algebra-simps power2-eq-square[symmetric] t-expr(1)
d-nz)+
    have (x2,y2) ∈ e-aff
      using rot-aff[OF ‹g' ∈ rotations› ‹τ (x', y') ∈ e-aff›]
      unfolding x2-def y2-def by simp
    have ((x,y),(x2,y2)) ∈ e-aff-0
      unfolding e-aff-0-def
      using ‹(x,y) ∈ e-aff› ‹(x2,y2) ∈ e-aff› ‹delta x y x2 y2 ≠ 0›
      by blast
    then show ?thesis
```

using *add-cancel-2*[*OF* - ‹((*x*,*y*),(*x2*,*y2*)) ∈ *e-aff-0*›]
unfolding *x2-def y2-def* **apply**(*simp del*: *τ.simps add.simps*)
using ‹*add* (*x*, *y*) (*g*′ (*τ* (*x*′, *y*′))) = *add* (*x*, *y*) (*i* (*x*, *y*))› **by** *auto*
qed
then have *τ* (*x*′, *y*′) = *g* (*i* (*x*,*y*))
by (*metis* ‹*g* ∘ *g*′ = *id*› *pointfree-idE*)
then have (*x*′,*y*′) = *τ* (*g* (*i* (*x*,*y*)))
by (*metis pointfree-idE tau-idemp*)
then have *False*
using *b*
by (*metis* (*no-types*, *hide-lams*) ‹*τ* (*x*′, *y*′) = *g* (*i* (*x*, *y*))› ‹*τ* (*x*′, *y*′) ∈ *e-circ*› ‹*g*′ (*τ* (*x*′, *y*′)) = *i* (*x*, *y*)› ‹*g*′ ∈ *rotations*› *comp-apply group-add-class.minus-comp-minus i.simps i-circ id-apply r-expr*(*1*) *rot-circ tau-rot-sym*)
then have *fst* (*add* (*x*,*y*) (*τ* (*x*′,*y*′))) = *0* ∨ *snd* (*add* (*x*,*y*) (*τ* (*x*′,*y*′))) = *0* ⟹ (∃ *g*∈*symmetries*. (*x*′, *y*′) = (*g* ∘ *i*) (*x*, *y*)) **by** *blast*
note *case1* = *this*

then show *?thesis*
using *case1 case2*
unfolding *r2-def r1-def*
apply(*simp del*: *τ.simps add.simps*)
using ‹*fst* (*add* (*x*, *y*) (*τ* (*x*′, *y*′))) = *0* ∨ *snd* (*add* (*x*, *y*) (*τ* (*x*′, *y*′))) = *0*› **by** *blast*
qed

theorem *well-defined*:
assumes *p* ∈ *e-proj q* ∈ *e-proj*
shows *card* (*proj-add-class p q*) = *1*
proof −
from *e-proj-eq*[*OF assms*(*1*)] *e-proj-eq*[*OF assms*(*2*)]
obtain *x y l x*′ *y*′ *l*′ **where**
p-q-expr: (*p* = {((*x*, *y*), *l*)} ∨ *p* = {((*x*, *y*), *l*), (*τ* (*x*, *y*), *l* + *1*)})
(*x*, *y*) ∈ *e-aff*
(*q* = {((*x*′, *y*′), *l*′)} ∨ *q* = {((*x*′, *y*′), *l*′), (*τ* (*x*′, *y*′), *l*′ + *1*)})
(*x*′, *y*′) ∈ *e-aff* **by** *blast*
then consider
(*1*) *p* = {((*x*, *y*), *l*)} *q* = {((*x*′, *y*′), *l*′)} |
(*2*) *p* = {((*x*, *y*), *l*)} *q* = {((*x*′, *y*′), *l*′), (*τ* (*x*′, *y*′), *l*′ + *1*)} |
(*3*) *p* = {((*x*, *y*), *l*), (*τ* (*x*, *y*), *l* + *1*)} *q* = {((*x*′, *y*′), *l*′)} |
(*4*) *p* = {((*x*, *y*), *l*), (*τ* (*x*, *y*), *l* + *1*)} *q* = {((*x*′, *y*′), *l*′), (*τ* (*x*′, *y*′), *l*′ + *1*)} **by** *argo*
then show *?thesis*
proof(*cases*)
case *1*
then have *proj-add-class p q* = *proj-add-class* {((*x*, *y*), *l*)} {((*x*′, *y*′), *l*′)}
by *auto*
then obtain *v* **where** *v-expr*: *proj-add* ((*x*, *y*), *l*) ((*x*′, *y*′), *l*′) = *Some v*
using *covering*[*OF assms*] **unfolding** *proj-add-class-def* **by** *auto*
have *s-map*: (λ(*x*, *y*). *the* (*proj-add x y*)) ‘ (*dom* (λ(*x*, *y*). *proj-add x y*) ∩ *p*

$\times$ *q*) =
$\{v\}$
unfolding *image-def dom-def 1* **apply**(*simp add*: *v-expr*)
proof −
have ($\exists$ *a b ba*. *v* = ((*a*, *b*), *ba*))
by (*metis surjective-pairing*)
then show $\{y.\ y = v \wedge (\exists\, a\ b\ ba.\ v = ((a,\ b),\ ba))\} = \{v\}$ **by** *simp*
qed
show *?thesis*
unfolding *proj-add-class-def* **apply**(*simp add*: *s-map*)
using *assms*(*1*) **unfolding** *1 e-proj-def quotient-def* **by** *auto*
next
case *2*

consider
(*a*) $(x,\ y) \in$ *e-circ* $\wedge$ ($\exists$ *g*$\in$*symmetries*. $(x',\ y') = (g \circ i)\ (x,\ y)$) |
(*b*) $((x,\ y),\ x',\ y') \in$ *e-aff-0* $\neg$ ($(x,\ y) \in$ *e-circ* $\wedge$ ($\exists$ *g*$\in$*symmetries*. $(x',\ y')$ $= (g \circ i)\ (x,\ y)$)) |
(*c*) $((x,\ y),\ x',\ y') \in$ *e-aff-1* $\neg$ ($(x,\ y) \in$ *e-circ* $\wedge$ ($\exists$ *g*$\in$*symmetries*. $(x',\ y')$ $= (g \circ i)\ (x,\ y)$)) $((x,\ y),\ x',\ y') \notin$ *e-aff-0*
using *dichotomy-1*[*OF* ‹$(x,y) \in$ *e-aff*› ‹$(x',y') \in$ *e-aff*›] **by** *fast*
then show *?thesis*
proof(*cases*)
case *a*
then obtain *g* **where** $g \in$ *symmetries* $(x',\ y') = (g \circ i)\ (x,\ y)$ **by** *auto*

then have *delta x y x′ y′* = *0* *delta′ x y x′ y′* = *0*
using *wd-d-nz wd-d′-nz a* **by** *auto*
then have *one-none*: *proj-add* $((x,\ y),\ l)\ ((x',\ y'),\ l')$ = *None*
using *proj-add.simps* **unfolding** *p-delta-def p-delta′-def* **by** *auto*
have (*dom* ($\lambda(x,\ y)$. *proj-add x y*) $\cap$ $\{((x,\ y),\ l)\} \times \{((x',\ y'),\ l'),\ (\tau\ (x',\ y'),\ l' + 1)\}$) $\neq \{\}$
using *covering*[*OF assms*] **unfolding** *2 proj-add-class-def* **by** *blast*
then have *s-simp*:
(*dom* ($\lambda(x,\ y)$. *proj-add x y*) $\cap$
$\{((x,\ y),\ l)\} \times \{((x',\ y'),\ l'),\ (\tau\ (x',\ y'),\ l' + 1)\}$)
$= \{(((x,\ y),\ l),(\tau\ (x',\ y'),\ l' + 1))\}$
using *one-none* **by** *auto*
show *card*(*proj-add-class p q*) = *1*
unfolding *proj-add-class-def 2*
apply(*subst s-simp*)
unfolding *quotient-def* **by** *auto*
next
case *b*
then have *ld-nz*: *delta x y x′ y′* $\neq$ *0*
unfolding *e-aff-0-def* **by** *auto*
consider
(*aa*) *x′* = *0* |
(*bb*) *y′* = *0* |

(cc) $x' \neq 0$ $y' \neq 0$ **by** *blast*
then show *?thesis*
proof(*cases*)
case *aa*
have *y-expr*: $y' = 1 \vee y' = -1$
using *e-aff-x0*[*OF aa* ‹$(x',y') \in$ *e-aff*›] **by** *simp*
have *delta* $x\ y\ x'\ y' \neq 0$
unfolding *delta-def delta-plus-def delta-minus-def*
using *aa* **by** *simp*
have *d-0-nz*: *delta* $x\ y\ 0\ y' \neq 0$
unfolding *delta-def delta-plus-def delta-minus-def* **by** *auto*
have $(0,\ 1\ /\ (t * y')) \notin$ *e-aff*
using ‹$(x',y') \in$ *e-aff*› *aa* **unfolding** *e-aff-def e'-def*
apply(*simp add*: *divide-simps t-sq-n1 t-nz*,*safe*)
by (*simp add*: *power-mult-distrib t-sq-n1*)
have *v1*: *proj-add* $((x,\ y),\ l)$ $((0,\ y'),\ l')$ = *Some* $((-\ (c * y * y'),\ x * y'),\ l + l')$
apply(*simp add*: *proj-add.simps* ‹$(x,y) \in$ *e-aff*› *p-delta-def d-0-nz*)
using *b aa* **unfolding** *e-aff-0-def* **by** *simp*
have *v2*: *proj-add* $((x,\ y),\ l)$ $(\tau\ (0,\ y'),\ l' + 1)$ = *None*
apply(*simp add*: *proj-add.simps* ‹$(x,y) \in$ *e-aff*› *p-delta-def d-0-nz*)
by(*simp add*: ‹$(0,\ 1\ /\ (t * y')) \notin$ *e-aff*›)
have *dom-eq*: (*dom* $(\lambda(x,\ y).$ *proj-add* $x\ y) \cap$
$\{(((x,\ y),\ l),\ (0,\ y'),\ l'),$
$(((x,\ y),\ l),\ \tau\ (0,\ y'),\ l' + 1)\})$ =
$\{(((x,\ y),\ l),\ (0,\ y'),\ l')\}$
using *v1 v2* **by** *auto*
show *?thesis*
unfolding *2* **apply**(*simp add*: *aa t-nz del*: τ*.simps*)
unfolding *proj-add-class-def* **apply**(*simp add*: *dom-eq del*: τ*.simps*)
unfolding *quotient-def* **by** *auto*
next
case *bb*
have *x-expr*: $x' = 1 \vee x' = -1$
using *e-aff-y0*[*OF bb* ‹$(x',y') \in$ *e-aff*›] **by** *simp*
have *delta* $x\ y\ x'\ y' \neq 0$
unfolding *delta-def delta-plus-def delta-minus-def*
using *bb* **by** *simp*
have *d-0-nz*: *delta* $x\ y\ x'\ 0 \neq 0$
unfolding *delta-def delta-plus-def delta-minus-def* **by** *auto*
have $(1\ /\ (t * x'),0) \notin$ *e-aff*
unfolding *e-aff-def e'-def*
using ‹$(x',y') \in$ *e-aff*› *bb* **unfolding** *e-aff-def e'-def*
apply(*simp add*: *divide-simps t-sq-n1 t-nz*,*safe*)
by (*simp add*: *power-mult-distrib t-sq-n1*)
have *v1*: *proj-add* $((x,\ y),\ l)$ $((x',\ 0),\ l')$ = *Some* $((x * x',\ y * x'),\ l + l')$
apply(*simp add*: *proj-add.simps* ‹$(x,y) \in$ *e-aff*› *p-delta-def d-0-nz*)
using *b bb* **unfolding** *e-aff-0-def* **by** *simp*
have *v2*: *proj-add* $((x,\ y),\ l)$ $(\tau\ (x',\ 0),\ l' + 1)$ = *None*

apply(*simp add*: *proj-add.simps* ‹$(x,y) \in$ *e-aff*› *p-delta-def d-0-nz*)
by(*simp add*: ‹$(1 / (t * x'),0) \notin$ *e-aff*›)
have *dom-eq*: (*dom* ($\lambda(x, y)$. *proj-add x y*) $\cap$
$\{(((x, y), l), (x', 0), l'),$
$(((x, y), l), \tau\ (x', 0), l' + 1)\}) =$
$\{(((x, y), l), (x', 0), l')\}$
using *v1 v2* **by** *auto*
show *?thesis*
unfolding *2* **apply**(*simp add*: *bb t-nz del*: τ*.simps*)
unfolding *proj-add-class-def* **apply**(*simp add*: *dom-eq del*: τ*.simps*)
unfolding *quotient-def* **by** *auto*
next
case *cc*

have $(x',y') \in$ *e-circ*
unfolding *e-circ-def* **using** *cc* ‹$(x',y') \in$ *e-aff*› **by** *blast*
then have $\tau\ (x', y') \in$ *e-circ*
using *cc* τ*-circ* **by** *blast*
then have $\tau\ (x', y') \in$ *e-aff*
unfolding *e-circ-def* **by** *force*

have *v1*: *proj-add* $((x, y), l)$ $((x', y'), l')$ = *Some* (*add* (x, y) (x', y'), l
$+ l'$)
by(*simp add*: *proj-add.simps* ‹$(x,y) \in$ *e-aff*› ‹$(x',y') \in$ *e-aff*› *p-delta-def*
ld-nz del: *add.simps*)
consider
(*z1*) $x = 0$ |
(*z2*) $y = 0$ |
(*z3*) $x \neq 0$ $y \neq 0$ **by** *blast*
then show *?thesis*
proof(*cases*)
case *z1*
then have *y-expr*: $y = 1 \vee y = -1$
using ‹$(x,y) \in$ *e-aff*› **unfolding** *e-aff-def e'-def*
by(*simp*,*algebra*)
then have $y*y = 1$ **by** *auto*
have *add* (x, y) $(x', y') = \varrho\ (y*x',y*y')$
by(*simp add*: *z1*,*simp add*: *c-eq-1*)
then have *v1-def*: *proj-add* $((x, y), l)$ $((x', y'), l')$ =
Some ($\varrho\ (y*x',y*y')$, $l + l'$)
using *v1* **by** *simp*
have *delta x y* (*fst* ($\tau\ (x',y')$)) (*snd* ($\tau\ (x',y')$)) $\neq 0$
unfolding *delta-def delta-plus-def delta-minus-def*
using *z1* **by** *simp*
then have *v2*: *proj-add* $((x, y), l)$ ($\tau\ (x', y')$, $l' + 1$) =
Some (*add* (x, y) ($\tau\ (x', y')$), $l+l'+1$)
using *proj-add.simps p-delta-def*
using ‹$\tau\ (x', y') \in$ *e-aff*› *p-q-expr*(*2*) **by** *auto*
have *add* (x, y) ($\tau\ (x', y')$) = ϱ ($y*$(*fst* ($\tau\ (x', y')$)),$y*$(*snd* ($\tau\ (x', y')$)))

```
    by(simp add: z1, simp add: c-eq-1)
  then have add (x, y) (τ (x', y')) = (ϱ ∘ τ) (y∗x',y∗y')
    apply(simp)
    apply(rule conjI)
    by(simp add: divide-simps t-nz cc y-expr ‹y∗y = 1›)+
  then have v2-def: proj-add ((x, y), l) (τ (x', y'), l' + 1) =
              Some (τ (ϱ (y∗x',y∗y')), l+l'+1)
    using v2 rot-tau-com rotations-def by auto
  have dom-eq: (dom (λ(x, y). proj-add x y) ∩
        {(((0, y), l), (x', y'), l'),
         (((0, y), l), τ (x', y'), l' + 1)}) =
      {(((0, y), l), (x', y'), l'), (((0, y), l), τ (x', y'), l' + 1)}
    using v1-def v2-def z1 by auto
  have rho-aff: ϱ (y ∗ x', y ∗ y') ∈ e-aff
      using ‹(x,y) ∈ e-aff› ‹(x',y') ∈ e-aff› unfolding e-aff-def e'-def
      apply(cases y = 1)
      apply(simp add: z1,argo)
      using y-expr by(simp add: z1,argo)
 have eq: {(ϱ (y ∗ x', y ∗ y'), l + l'), (τ (ϱ (y ∗ x', y ∗ y')), l + l' + 1)}
          = gluing `` {(ϱ (y ∗ x', y ∗ y'), l + l')}
  proof −
    have coord: fst (ϱ (y ∗ x', y ∗ y')) ≠ 0 snd (ϱ (y ∗ x', y ∗ y')) ≠ 0
      using y-expr cc by auto
    show ?thesis
      using gluing-class[OF coord(1) coord(2)] rho-aff by simp
  qed
  show ?thesis
    unfolding 2 apply(simp add: t-nz z1 del: τ.simps)
    unfolding proj-add-class-def apply(simp add: dom-eq del: τ.simps)
    apply(subst z1[symmetric])+
    apply(subst v1-def,subst v2-def,simp del: τ.simps ϱ.simps)
    apply(subst eq)
    using eq-class-image rho-aff by fastforce
next
  case z2
  then have x-expr: x = 1 ∨ x = −1
    using ‹(x,y) ∈ e-aff› unfolding e-aff-def e'-def
    by(simp,algebra)
  then have x∗x = 1 by auto
  have add (x, y) (x', y') = (x∗x',x∗y')
    by(simp add: z2)
  then have v1-def: proj-add ((x, y), l) ((x', y'), l') =
                Some ((x∗x',x∗y'), l + l')
    using v1 by simp
  have delta x y (fst (τ (x',y'))) (snd (τ (x',y'))) ≠ 0
    unfolding delta-def delta-plus-def delta-minus-def
    using z2 by simp
  then have v2: proj-add ((x, y), l) (τ (x', y'), l' + 1) =
          Some (add (x, y) (τ (x', y')), l+l'+1)
```

using *proj-add.simps p-delta-def*
using ‹τ $(x', y') \in$ *e-aff*› *p-q-expr*(*2*) **by** *auto*
have *add* (x, y) $(\tau$ $(x', y')) = (x*($*fst* $(\tau$ $(x', y'))),x*($*snd* $(\tau$ $(x', y'))))$
by(*simp add*: *z2*)
then have *add* (x, y) $(\tau$ $(x', y')) = \tau$ $(x*x',x*y')$
apply(*simp*)
apply(*rule conjI*)
by(*simp add*: *divide-simps t-nz cc x-expr* ‹$x*x = 1$›)+
then have *v2-def*: *proj-add* $((x, y), l)$ $(\tau$ $(x', y'), l' + 1) =$
Some $(\tau$ $(x*x',x*y'), l+l'+1)$
using *v2 rot-tau-com rotations-def* **by** *auto*
have *dom-eq*: (*dom* $(\lambda(x, y).$ *proj-add* x $y) \cap$
$\{(((x, 0), l), (x', y'), l'),$
$(((x, 0), l), \tau$ $(x', y'), l' + 1)\}) =$
$\{(((x, 0), l), (x', y'), l'), (((x, 0), l), \tau$ $(x', y'), l' + 1)\}$
using *v1-def v2-def z2* **by** *auto*
have *rho-aff*: $(x * x', x * y') \in$ *e-aff*
using ‹$(x,y) \in$ *e-aff*› ‹$(x',y') \in$ *e-aff*› **unfolding** *e-aff-def e'-def*
apply(*cases* $x = 1$)
apply(*simp*)
using *x-expr* **by**(*simp add*: *z2*)
have *eq*: $\{((x * x', x * y'), l + l'), (\tau$ $(x * x', x * y'), l + l' + 1)\}$
$=$ *gluing* ‘‘ $\{((x * x', x * y'), l + l')\}$
proof −
have *coord*: *fst* $((x * x', x * y')) \neq 0$ *snd* $((x * x', x * y')) \neq 0$
using *x-expr cc* **by** *auto*
show *?thesis*
using *gluing-class*[*OF coord*(*1*) *coord*(*2*)] *rho-aff* **by** *simp*
qed
show *?thesis*
unfolding *2* **apply**(*simp add*: *t-nz z2 del*: τ*.simps*)
unfolding *proj-add-class-def* **apply**(*simp add*: *dom-eq del*: τ*.simps*)
apply(*subst z2*[*symmetric*])+
apply(*subst v1-def*,*subst v2-def*,*simp del*: τ*.simps* ϱ*.simps*)
apply(*subst eq*)
using *eq-class-image rho-aff* **by** *fastforce*
next
case *z3*
consider
(*aaa*) *p-delta* $((x, y), l)$ $(\tau$ $(x', y'), l' + 1) \neq 0 \wedge$ *fst* $((x, y), l)\in$ *e-aff* $\wedge$ *fst* $(\tau$ $(x', y'), l' + 1) \in$ *e-aff* |
(*bbb*) *p-delta'* $((x, y), l)$ $(\tau$ $(x', y'), l' + 1) \neq 0 \wedge$ *fst* $((x, y), l) \in$ *e-aff* $\wedge$ *fst* $(\tau$ $(x', y'), l' + 1) \in$ *e-aff* |
(*ccc*) *p-delta* $((x, y), l)$ $(\tau$ $(x', y'), l' + 1) = 0 \wedge$ *p-delta'* $((x, y), l)$ $(\tau$ $(x', y'), l' + 1) = 0$
$\vee$ *fst* $((x, y), l) \notin$ *e-aff* $\vee$ *fst* $(\tau$ $(x', y'), l' + 1) \notin$ *e-aff*
by(*simp add*: *proj-add.simps*,*blast*)
then show *?thesis*
proof(*cases*)

case *aaa*
from *aaa* **have** *aaa-simp*:
proj-add $((x, y), l)$ $(\tau$ $(x', y'), l' + 1) =$
Some $(add\ (x, y)\ (\tau\ (x', y')), l+l'+1)$
using *proj-add.simps* **by** *simp*
have $x' * y' \neq - x * y$
using *aaa* **unfolding** *p-delta-def delta-def delta-plus-def delta-minus-def*
apply(*simp add*: *t-nz cc divide-simps*)
apply(*simp add*: *algebra-simps power2-eq-square*[*symmetric*] *t-expr*(*1*)
d-nz)
by(*simp add*: *ring-distribs*(*1*)[*symmetric*] *d-nz*)
have $x' * y' \neq x * y$
using *aaa* **unfolding** *p-delta-def delta-def delta-plus-def delta-minus-def*
apply(*simp add*: *t-nz cc divide-simps*)
by(*simp add*: *algebra-simps power2-eq-square*[*symmetric*] *t-expr*(*1*))

have *closure-lem*: $add\ (x, y)\ (\tau\ (x', y')) \in e\text{-}aff$
proof −
obtain *x1 y1* **where** *z2-d*: $\tau\ (x', y') = (x1,y1)$ **by** *fastforce*
define *z3* **where** $z3 = add\ (x,y)\ (x1,y1)$
obtain *x2 y2* **where** *z3-d*: $z3 = (x2,y2)$ **by** *fastforce*
have $delta\ x\ y\ x1\ y1 \neq 0$
using *aaa z2-d* **unfolding** *p-delta-def* **by** *auto*
then have *dpm*: $delta\text{-}minus\ x\ y\ x1\ y1 \neq 0$ $delta\text{-}plus\ x\ y\ x1\ y1 \neq 0$
unfolding *delta-def* **by** *auto*
have $(x1,y1) \in e\text{-}aff$
unfolding *z2-d*[*symmetric*]
using ‹$\tau\ (x', y') \in e\text{-}aff$› **by** *auto*
have *e-eq*: $e\ x\ y = 0$ $e\ x1\ y1 = 0$
using ‹$(x,y) \in e\text{-}aff$› ‹$(x1,y1) \in e\text{-}aff$› *e-e'-iff* **unfolding** *e-aff-def*
by(*auto*)

have $e\ x2\ y2 = 0$
using *add-closure*[*OF z3-d z3-def dpm*]
using *add-closure*[*OF z3-d z3-def dpm e-eq*] **by** *simp*
then show *?thesis*
unfolding *e-aff-def* **using** *e-e'-iff z3-d z3-def z2-d* **by** *simp*
qed

have *add-nz*:
$fst\ (add\ (x, y)\ (\tau\ (x', y'))) \neq 0$
$snd\ (add\ (x, y)\ (\tau\ (x', y'))) \neq 0$
using *b-cc-case*[*OF closure-lem p-q-expr*(*2*) ‹$\tau\ (x', y') \in e\text{-}aff$› ‹$\tau\ (x',$
$y') \in e\text{-}circ$› *cc*
‹$x' * y' \neq - x * y$› ‹$x' * y' \neq x * y$› *b*(*2*)] *e-circ-def*
z3(*1*) *z3*(*2*)
using *b*(*2*) *p-q-expr*(*2*) **apply** *blast*
using ‹$fst\ (add\ (x, y)\ (\tau\ (x', y'))) = 0 \vee snd\ (add\ (x, y)\ (\tau\ (x', y'))) =$

0 $\Longrightarrow$ $\exists$ *g*$\in$*symmetries*. $(x', y') = (g \circ i)\ (x, y)$› *b*(*2*) *e-circ-def p-q-expr*(*2*) *z3*(*1*) *z3*(*2*) **by** *blast*
then have *1*: *gluing* ‘‘ $\{((add\ (x,y)\ (\tau\ (x',y'))),l+l'+1)\}$ =
gluing ‘‘ $\{(\tau\ (add\ (x,y)\ (\tau\ (x',y'))),l+l')\}$
using *gluing-inv closure-lem* **by** *force*
also have ... = *gluing* ‘‘ $\{(ext\text{-}add\ (x,y)\ (x',y'),l+l')\}$
using *add-ext-add cc*(*1*) *cc*(*2*) *curve-addition.commutativity ext-add-comm z3*(*1*) *z3*(*2*) **by** *auto*
finally have *gl-eq*: *gluing* ‘‘ $\{((add\ (x,y)\ (\tau\ (x',y'))),l+l'+1)\}$ =
gluing ‘‘ $\{(ext\text{-}add\ (x,y)\ (x',y'),l+l')\}$ **by** *blast*
have $\{((x, y), l)\}$ // *gluing* = $\{\{((x, y), l)\}\}$
using *eq-class-simp*[*OF assms*(*1*)] **by**(*simp add*: *2*(*1*))
then have *ext-to-add*: $(ext\text{-}add\ (x,y)\ (x',y'),l+l') = (add\ (x,y)\ (x',y'),l+l')$

using *gluing-class*[*OF z3* ‹$(x,y) \in$ *e-aff*›]
by (*simp add*: *singleton-quotient*)
then have *def-gl-eq*: *gluing* ‘‘ $\{((add\ (x,y)\ (\tau\ (x',y'))),l+l'+1)\}$ =
gluing ‘‘ $\{(add\ (x,y)\ (x',y'),l+l')\}$
using *ext-to-add gl-eq* **by** *argo*
have *dom-eq*: (*dom* ($\lambda(x, y)$. *proj-add x y*) $\cap$
$\{(((x, y), l), (x', y'), l'), (((x, y), l), \tau\ (x', y'), l' + 1)\}) =$
$\{(((x, y), l), (x', y'), l'), (((x, y), l), \tau\ (x', y'), l' + 1)\}$
using *aaa-simp v1* **by** *auto*
then have *proj-eq*: $\{$*the* (*proj-add* $((x, y), l)\ ((x', y'), l')$),
the (*proj-add* $((x, y), l)\ (\tau\ (x', y'), l' + 1)$)$\}$ =
$\{(add\ (x, y)\ (\tau\ (x', y')), l + l' + 1), (add\ (x, y)\ (x', y'), l + l')\}$
using *aaa-simp v1* **by** *auto*
show *?thesis*
unfolding *2 proj-add-class-def* **apply**(*simp add*: *dom-eq proj-eq del*: *add.simps* τ*.simps ext-add.simps*)
unfolding *quotient-def* **using** *def-gl-eq* **by** *simp*
next
case *bbb*
have $\{((x, y), l)\}$ // *gluing* = $\{\{((x, y), l)\}\}$
using *eq-class-simp*[*OF assms*(*1*)] **by** (*simp add*: *2*(*1*))
from *this bbb* **have** *aaa-simp*:
proj-add $((x, y), l)\ (\tau\ (x', y'), l' + 1) =$
Some $(ext\text{-}add\ (x, y)\ (\tau\ (x', y')), l+l'+1)$
apply(*simp add*: *proj-add.simps del*: *ext-add.simps* τ*.simps*,*safe*)
using *gluing-class*[*OF z3* ‹$(x,y) \in$ *e-aff*›]
by (*metis* (*no-types*, *lifting*) *2*(*1*) *add-cancel-right-right doubleton-eq-iff insert-absorb2 singleton-quotient snd-conv zero-neq-one*)

have *closure-lem*: *ext-add* $(x, y)\ (\tau\ (x', y')) \in$ *e-aff*
proof −
obtain *x1 y1* **where** *z2-d*: $\tau\ (x', y') = (x1,y1)$ **by** *fastforce*
define *z3* **where** *z3* = *ext-add* $(x,y)\ (x1,y1)$
obtain *x2 y2* **where** *z3-d*: *z3* = $(x2,y2)$ **by** *fastforce*

```
            have d′: delta′ x y x1 y1 ≠ 0
              using bbb z2-d unfolding p-delta′-def by auto
            have (x1,y1) ∈ e-aff
              unfolding z2-d[symmetric]
              using ‹τ (x′, y′) ∈ e-aff› by auto
            have e-eq: e′ x y = 0 e′ x1 y1 = 0
             using ‹(x,y) ∈ e-aff› ‹(x1,y1) ∈ e-aff› unfolding e-aff-def by(auto)

            have e′ x2 y2 = 0
              using z3-d z3-def ext-add-closure[OF d′ e-eq, of x2 y2] by blast
            then show ?thesis
              unfolding e-aff-def using e-e′-iff z3-d z3-def z2-d by simp
          qed

          have dom-eq: (dom (λ(x, y). proj-add x y) ∩
                        {(((x, y), l), (x′, y′), l′), (((x, y), l), τ (x′, y′), l′ + 1)}) =
                        {(((x, y), l), (x′, y′), l′), (((x, y), l), τ (x′, y′), l′ + 1)}
            using aaa-simp v1 by auto
          then have proj-eq: {the (proj-add ((x, y), l) ((x′, y′), l′)),
                              the (proj-add ((x, y), l) (τ (x′, y′), l′ + 1))} =
                     {(ext-add (x, y) (τ (x′, y′)), l + l′ + 1), (add (x, y) (x′, y′), l
+ l′)}
            using aaa-simp v1 by auto
          have gluing ‘‘ {(ext-add (x, y) (τ (x′, y′)), l + l′ + 1)} =
                gluing ‘‘ {(add (x, y) (x′, y′), l + l′)}
             using ‹{((x, y), l)} // gluing = {{((x, y), l)}}› gluing-class[OF z3
p-q-expr(2)]
            by (simp add: singleton-quotient)
          then show ?thesis
             unfolding 2 proj-add-class-def apply(simp add: dom-eq proj-eq del:
add.simps τ.simps ext-add.simps)
            unfolding quotient-def by force
        next
          case ccc
          from ccc have aaa-simp:
            proj-add ((x, y), l) (τ (x′, y′), l′ + 1) = None
            by(simp add: proj-add.simps p-q-expr(2),blast)
          then have dom-eq: (dom (λ(x, y). proj-add x y) ∩
              {(((x, y), l), (x′, y′), l′),(((x, y), l), τ (x′, y′), l′ + 1)}) =
              {(((x, y), l),((x′, y′), l′))}
          using v1 by auto
          then show ?thesis
            unfolding 2 proj-add-class-def
            apply(simp add: dom-eq del: τ.simps)
            unfolding quotient-def by simp
        qed
      qed
    qed
    next
```

```
case c
then have ld-nz: delta' x y x' y' ≠ 0
  unfolding e-aff-1-def by auto
consider
  (aa) x' = 0 |
  (bb) y' = 0 |
  (cc) x' ≠ 0 y' ≠ 0 by blast
then show ?thesis
proof(cases)
  case aa
  have y-expr: y' = 1 ∨ y' = −1
    using e-aff-x0[OF aa ‹(x',y') ∈ e-aff›] by simp
  have delta x y x' y' ≠ 0
    unfolding delta-def delta-plus-def delta-minus-def
    using aa by simp
  have d-0-nz: delta x y 0 y' ≠ 0
    unfolding delta-def delta-plus-def delta-minus-def by auto
  have (0, 1 / (t ∗ y')) ∉ e-aff
    using ‹(x',y') ∈ e-aff› aa unfolding e-aff-def e'-def
    apply(simp add: divide-simps t-sq-n1 t-nz,safe)
    by (simp add: power-mult-distrib t-sq-n1)
   have v1: proj-add ((x, y), l) ((0, y'), l') = Some ((− (c ∗ y ∗ y'), x ∗
y'), l + l')
    apply(simp add: proj-add.simps ‹(x,y) ∈ e-aff› p-delta-def d-0-nz)
    using c aa unfolding e-aff-1-def by blast
  have v2: proj-add ((x, y), l) (τ (0, y'), l' + 1) = None
    apply(simp add: proj-add.simps ‹(x,y) ∈ e-aff› p-delta-def d-0-nz)
    by(simp add: ‹(0, 1 / (t ∗ y')) ∉ e-aff›)
  have dom-eq: (dom (λ(x, y). proj-add x y) ∩
          {(((x, y), l), (0, y'), l'),
           (((x, y), l), τ (0, y'), l' + 1)}) =
        {(((x, y), l), (0, y'), l')}
    using v1 v2 by auto
  show ?thesis
    unfolding 2 apply(simp add: aa t-nz del: τ.simps)
    unfolding proj-add-class-def apply(simp add: dom-eq del: τ.simps)
    unfolding quotient-def by auto
next
  case bb
  have x-expr: x' = 1 ∨ x' = −1
    using e-aff-y0[OF bb ‹(x',y') ∈ e-aff›] by simp
  have delta x y x' y' ≠ 0
    unfolding delta-def delta-plus-def delta-minus-def
    using bb by simp
  have d-0-nz: delta x y x' 0 ≠ 0
    unfolding delta-def delta-plus-def delta-minus-def by auto
  have (1 / (t ∗ x'),0) ∉ e-aff
    unfolding e-aff-def e'-def
    using ‹(x',y') ∈ e-aff› bb unfolding e-aff-def e'-def
```

```
      apply(simp add: divide-simps t-sq-n1 t-nz,safe)
      by (simp add: power-mult-distrib t-sq-n1)
    have v1: proj-add ((x, y), l) ((x', 0), l') = Some ((x * x', y * x'), l + l')
      apply(simp add: proj-add.simps ‹(x,y) ∈ e-aff› p-delta-def d-0-nz)
      using c bb unfolding e-aff-1-def by simp
    have v2: proj-add ((x, y), l) (τ (x', 0), l' + 1) = None
      apply(simp add: proj-add.simps ‹(x,y) ∈ e-aff› p-delta-def d-0-nz)
      by(simp add: ‹(1 / (t * x'),0) ∉ e-aff›)
    have dom-eq: (dom (λ(x, y). proj-add x y) ∩
            {(((x, y), l), (x', 0), l'),
             (((x, y), l), τ (x', 0), l' + 1)}) =
          {(((x, y), l), (x', 0), l')}
      using v1 v2 by auto
    show ?thesis
      unfolding 2 apply(simp add: bb t-nz del: τ.simps)
      unfolding proj-add-class-def apply(simp add: dom-eq del: τ.simps)
      unfolding quotient-def by auto
  next
    case cc
    have delta x y x' y' = 0
      using ‹(x,y) ∈ e-aff› ‹(x',y') ∈ e-aff› c
      unfolding e-aff-0-def by force
    have (x',y') ∈ e-circ
      unfolding e-circ-def using cc ‹(x',y') ∈ e-aff› by blast
    then have τ (x', y') ∈ e-circ
      using cc τ-circ by blast
    then have τ (x', y') ∈ e-aff
      unfolding e-circ-def by force
    have v1: proj-add ((x, y), l) ((x', y'), l') = Some (ext-add (x, y) (x', y'),
l + l')
        by(simp add: proj-add.simps p-delta'-def p-delta-def ‹(x,y) ∈ e-aff›
‹(x',y') ∈ e-aff› ld-nz ‹delta x y x' y' = 0›)

    consider
      (z1) x = 0 |
      (z2) y = 0 |
      (z3) x ≠ 0 y ≠ 0 by blast
    then show ?thesis
    proof(cases)
      case z1
      then have y-expr: y = 1 ∨ y = −1
        using ‹(x,y) ∈ e-aff› unfolding e-aff-def e'-def
        by(simp,algebra)
      then have y*y = 1 by auto
      have ext-add (x, y) (x', y') = ϱ (y*x',y*y')
        by(simp add: z1 cc divide-simps y-expr ‹y*y = 1›)
      then have v1-def: proj-add ((x, y), l) ((x', y'), l') =
                    Some (ϱ (y*x',y*y'), l + l')
        using v1 by(simp)
```

```
      have delta x y (fst (τ (x',y'))) (snd (τ (x',y'))) ≠ 0
        unfolding delta-def delta-plus-def delta-minus-def
        using z1 by simp
      then have v2: proj-add ((x, y), l) (τ (x', y'), l' + 1) =
                  Some (ext-add (x, y) (τ (x', y')), l+l'+1)
         using ‹delta x y x' y' = 0› delta-def delta-minus-def delta-plus-def z1
by auto
      have ext-add (x, y) (τ (x', y')) = ϱ (y*(fst (τ (x', y'))),y*(snd (τ (x',
y'))))
        by(simp add: z1 cc t-nz divide-simps ‹y*y = 1›)
      then have ext-add (x, y) (τ (x', y')) = (ϱ ∘ τ) (y*x',y*y')
        apply(simp)
        apply(rule conjI)
        by(simp add: divide-simps t-nz cc y-expr ‹y*y = 1›)+
      then have v2-def: proj-add ((x, y), l) (τ (x', y'), l' + 1) =
                    Some (τ (ϱ (y*x',y*y')), l+l'+1)
        using v2 rot-tau-com rotations-def by auto
      have dom-eq: (dom (λ(x, y). proj-add x y) ∩
           {((((0, y), l), (x', y'), l'),
            (((0, y), l), τ (x', y'), l' + 1)}) =
          {((((0, y), l), (x', y'), l'), (((0, y), l), τ (x', y'), l' + 1)}
        using v1-def v2-def z1 by auto
      have rho-aff: ϱ (y * x', y * y') ∈ e-aff
          using ‹(x,y) ∈ e-aff› ‹(x',y') ∈ e-aff› unfolding e-aff-def e'-def
          apply(cases y = 1)
          apply(simp add: z1,argo)
          using y-expr by(simp add: z1,argo)
     have eq: {(ϱ (y * x', y * y'), l + l'), (τ (ϱ (y * x', y * y')), l + l' + 1)}
             = gluing `` {(ϱ (y * x', y * y'), l + l')}
      proof −
        have coord: fst (ϱ (y * x', y * y')) ≠ 0 snd (ϱ (y * x', y * y')) ≠ 0
          using y-expr cc by auto
        show ?thesis
          using gluing-class[OF coord(1) coord(2)] rho-aff by simp
      qed
      show ?thesis
        unfolding 2 apply(simp add: t-nz z1 del: τ.simps)
        unfolding proj-add-class-def apply(simp add: dom-eq del: τ.simps)
        apply(subst z1[symmetric])+
        apply(subst v1-def,subst v2-def,simp del: τ.simps ϱ.simps)
        apply(subst eq)
        using eq-class-image rho-aff by fastforce
    next
      case z2
      then have x-expr: x = 1 ∨ x = −1
        using ‹(x,y) ∈ e-aff› unfolding e-aff-def e'-def
        by(simp,algebra)
      then have x*x = 1 by auto
      have add (x, y) (x', y') = (x*x',x*y')
```

by(*simp add*: *z2*)
then have *v1-def*: *proj-add* $((x,\ y),\ l)\ ((x',\ y'),\ l') =$
Some $((x{*}x',x{*}y'),\ l\ +\ l')$
using ‹*delta x y x' y' = 0*› *delta-def delta-minus-def delta-plus-def z2*
by *auto*
have *delta x y* (*fst* (τ (x',y'))) (*snd* (τ (x',y'))) $\neq$ *0*
unfolding *delta-def delta-plus-def delta-minus-def*
using *z2* **by** *simp*
then have *v2*: *proj-add* $((x,\ y),\ l)$ (τ $(x',\ y')$, $l' + 1$) =
Some (*add* $(x,\ y)$ (τ $(x',\ y')$), $l{+}l'{+}1$)
using *proj-add.simps p-delta-def*
using ‹τ $(x',\ y') \in$ *e-aff*› *p-q-expr*(*2*) **by** *auto*
have *add* $(x,\ y)$ (τ $(x',\ y')$) = ($x{*}$(*fst* (τ $(x',\ y')$)),$x{*}$(*snd* (τ $(x',\ y')$)))
by(*simp add*: *z2*)
then have *add* $(x,\ y)$ (τ $(x',\ y')$) = τ $(x{*}x',x{*}y')$
apply(*simp*)
apply(*rule conjI*)
by(*simp add*: *divide-simps t-nz cc x-expr* ‹$x{*}x$ = *1*›)+
then have *v2-def*: *proj-add* $((x,\ y),\ l)$ (τ $(x',\ y')$, $l' + 1$) =
Some (τ $(x{*}x',x{*}y')$, $l{+}l'{+}1$)
using *v2 rot-tau-com rotations-def* **by** *auto*
have *dom-eq*: (*dom* ($\lambda(x,\ y)$. *proj-add x y*) $\cap$
$\{(((x,\ 0),\ l),\ (x',\ y'),\ l'),$
$(((x,\ 0),\ l),\ \tau\ (x',\ y'),\ l' + 1)\}) =$
$\{(((x,\ 0),\ l),\ (x',\ y'),\ l'),\ (((x,\ 0),\ l),\ \tau\ (x',\ y'),\ l' + 1)\}$
using *v1-def v2-def z2* **by** *auto*
have *rho-aff*: $(x * x',\ x * y') \in$ *e-aff*
using ‹$(x,y) \in$ *e-aff*› ‹$(x',y') \in$ *e-aff*› **unfolding** *e-aff-def e'-def*
apply(*cases* $x = 1$)
apply(*simp*)
using *x-expr* **by**(*simp add*: *z2*)
have *eq*: $\{((x * x',\ x * y'),\ l + l'),\ (\tau\ (x * x',\ x * y'),\ l + l' + 1)\}$
= *gluing* ‘‘ $\{((x * x',\ x * y'),\ l + l')\}$
proof −
have *coord*: *fst* $((x * x',\ x * y')) \neq 0$ *snd* $((x * x',\ x * y')) \neq 0$
using *x-expr cc* **by** *auto*
show *?thesis*
using *gluing-class*[*OF coord*(*1*) *coord*(*2*)] *rho-aff* **by** *simp*
qed
show *?thesis*
unfolding *2* **apply**(*simp add*: *t-nz z2 del*: τ.*simps*)
unfolding *proj-add-class-def* **apply**(*simp add*: *dom-eq del*: τ.*simps*)
apply(*subst z2*[*symmetric*])+
apply(*subst v1-def*,*subst v2-def*,*simp del*: τ.*simps* ϱ.*simps*)
apply(*subst eq*)
using *eq-class-image rho-aff* **by** *fastforce*
next
case *z3*
consider

```
      (aaa) p-delta ((x, y), l) (τ (x', y'), l' + 1) ≠ 0 ∧ fst ((x, y), l)∈ e-aff
∧ fst (τ (x', y'), l' + 1) ∈ e-aff |
      (bbb) p-delta' ((x, y), l) (τ (x', y'), l' + 1) ≠ 0 ∧ fst ((x, y), l) ∈ e-aff
∧ fst (τ (x', y'), l' + 1) ∈ e-aff |
      (ccc) p-delta ((x, y), l) (τ (x', y'), l' + 1) = 0 ∧ p-delta' ((x, y), l) (τ
(x', y'), l' + 1) = 0
          ∨ fst ((x, y), l) ∉ e-aff ∨ fst (τ (x', y'), l' + 1) ∉ e-aff
       by(simp add: proj-add.simps,blast)
    then show ?thesis
    proof(cases)
      case aaa
      from aaa have aaa-simp:
        proj-add ((x, y), l) (τ (x', y'), l' + 1) =
         Some (add (x, y) (τ (x', y')), l+l'+1)
        using proj-add.simps by simp
      have x' * y' ≠ − x * y
       using aaa unfolding p-delta-def delta-def delta-plus-def delta-minus-def
        apply(simp add: t-nz cc divide-simps)
        apply(simp add: algebra-simps power2-eq-square[symmetric] t-expr(1)
d-nz)
        by(simp add: ring-distribs(1)[symmetric] d-nz)
      have x' * y' ≠ x * y
       using aaa unfolding p-delta-def delta-def delta-plus-def delta-minus-def
        apply(simp add: t-nz cc divide-simps)
        by(simp add: algebra-simps power2-eq-square[symmetric] t-expr(1))

      have closure-lem: add (x, y) (τ (x', y')) ∈ e-aff
      proof −
        obtain x1 y1 where z2-d: τ (x', y') = (x1,y1) by fastforce
        define z3 where z3 = add (x,y) (x1,y1)
        obtain x2 y2 where z3-d: z3 = (x2,y2) by fastforce
        have delta x y x1 y1 ≠ 0
          using aaa z2-d unfolding p-delta-def by auto
        then have dpm: delta-minus x y x1 y1 ≠ 0 delta-plus x y x1 y1 ≠ 0
          unfolding delta-def by auto
        have (x1,y1) ∈ e-aff
          unfolding z2-d[symmetric]
          using ‹τ (x', y') ∈ e-aff› by auto
        have e-eq: e x y = 0 e x1 y1 = 0
           using ‹(x,y) ∈ e-aff› ‹(x1,y1) ∈ e-aff› e-e'-iff  unfolding e-aff-def
by(auto)

        have e x2 y2 = 0
          using add-closure[OF z3-d z3-def dpm ]
          using add-closure[OF z3-d z3-def dpm e-eq] by simp
        then show ?thesis
          unfolding e-aff-def using e-e'-iff z3-d z3-def z2-d by simp
      qed
```

have *add-nz*:
fst (*add* (*x*, *y*) (τ (*x'*, *y'*))) $\neq$ *0*
snd (*add* (*x*, *y*) (τ (*x'*, *y'*))) $\neq$ *0*
using *b-cc-case*[*OF closure-lem p-q-expr*(*2*) ‹τ (*x'*, *y'*) $\in$ *e-aff*› ‹τ (*x'*, *y'*) $\in$ *e-circ*› *cc*
‹*x'* $*$ *y'* $\neq$ $-$ *x* $*$ *y*› ‹*x'* $*$ *y'* $\neq$ *x* $*$ *y*› *c*(*2*)] *e-circ-def* *z3*(*1*) *z3*(*2*)
using *c*(*2*) *p-q-expr*(*2*) **apply** *blast*
using ‹*fst* (*add* (*x*, *y*) (τ (*x'*, *y'*))) = *0* $\vee$ *snd* (*add* (*x*, *y*) (τ (*x'*, *y'*))) = *0* $\Longrightarrow$ $\exists$ *g*$\in$*symmetries*. (*x'*, *y'*) = (*g* $\circ$ *i*) (*x*, *y*)› *c*(*2*) *e-circ-def p-q-expr*(*2*) *z3*(*1*) *z3*(*2*) **by** *blast*
then have *1*: *gluing* `` {((*add* (*x*,*y*) (τ (*x'*,*y'*))),*l*+*l'*+*1*)} =
gluing `` {(τ (*add* (*x*,*y*) (τ (*x'*,*y'*))),*l*+*l'*)}
using *gluing-inv closure-lem* **by** *force*
also have ... = *gluing* `` {(*ext-add* (*x*,*y*) (*x'*,*y'*),*l*+*l'*)}
using *add-ext-add cc*(*1*) *cc*(*2*) *curve-addition.commutativity ext-add-comm* *z3*(*1*) *z3*(*2*) **by** *auto*
finally have *gl-eq*: *gluing* `` {((*add* (*x*,*y*) (τ (*x'*,*y'*))),*l*+*l'*+*1*)} =
gluing `` {(*ext-add* (*x*,*y*) (*x'*,*y'*),*l*+*l'*)} **by** *blast*
have {((*x*, *y*), *l*)} // *gluing* = {{((*x*, *y*), *l*)}}
using *eq-class-simp*[*OF assms*(*1*)] **by**(*simp add*: *2*(*1*))
then have *ext-to-add*: (*ext-add* (*x*,*y*) (*x'*,*y'*),*l*+*l'*) = (*add* (*x*,*y*) (*x'*,*y'*),*l*+*l'*)

using *gluing-class*[*OF z3* ‹(*x*,*y*) $\in$ *e-aff*›]
by (*simp add*: *singleton-quotient*)
then have *def-gl-eq*: *gluing* `` {((*add* (*x*,*y*) (τ (*x'*,*y'*))),*l*+*l'*+*1*)} =
gluing `` {(*ext-add* (*x*,*y*) (*x'*,*y'*),*l*+*l'*)}
using *ext-to-add gl-eq* **by** *argo*
have *dom-eq*: (*dom* (λ(*x*, *y*). *proj-add x y*) $\cap$
{(((*x*, *y*), *l*), (*x'*, *y'*), *l'*), (((*x*, *y*), *l*), τ (*x'*, *y'*), *l'* + *1*)}) =
{(((*x*, *y*), *l*), (*x'*, *y'*), *l'*), (((*x*, *y*), *l*), τ (*x'*, *y'*), *l'* + *1*)}
using *aaa-simp v1* **by** *auto*
then have *proj-eq*: {*the* (*proj-add* ((*x*, *y*), *l*) ((*x'*, *y'*), *l'*)),
the (*proj-add* ((*x*, *y*), *l*) (τ (*x'*, *y'*), *l'* + *1*))} =
{(*add* (*x*, *y*) (τ (*x'*, *y'*)), *l* + *l'* + *1*), (*ext-add* (*x*, *y*) (*x'*, *y'*), *l* + *l'*)}
using *aaa-simp v1* **by** *auto*
show *?thesis*
unfolding *2 proj-add-class-def* **apply**(*simp add*: *dom-eq proj-eq del*: *add.simps* τ*.simps ext-add.simps*)
unfolding *quotient-def* **using** *def-gl-eq* **by** *simp*
next
case *bbb*
have {((*x*, *y*), *l*)} // *gluing* = {{((*x*, *y*), *l*)} }
using *eq-class-simp*[*OF assms*(*1*)] **by** (*simp add*: *2*(*1*))
from *this bbb* **have** *aaa-simp*:
proj-add ((*x*, *y*), *l*) (τ (*x'*, *y'*), *l'* + *1*) =
Some (*ext-add* (*x*, *y*) (τ (*x'*, *y'*)), *l*+*l'*+*1*)

apply(*simp add*: *proj-add.simps del*: *ext-add.simps* τ*.simps*,*safe*)
using *gluing-class*[*OF z3* ‹(*x*,*y*) ∈ *e-aff*›]
by (*metis* (*no-types*, *lifting*) *2*(*1*) *add-cancel-right-right doubleton-eq-iff insert-absorb2 singleton-quotient snd-conv zero-neq-one*)

have *closure-lem*: *ext-add* (*x*, *y*) (τ (*x'*, *y'*)) ∈ *e-aff*
proof −
obtain *x1 y1* **where** *z2-d*: τ (*x'*, *y'*) = (*x1*,*y1*) **by** *fastforce*
define *z3* **where** *z3* = *ext-add* (*x*,*y*) (*x1*,*y1*)
obtain *x2 y2* **where** *z3-d*: *z3* = (*x2*,*y2*) **by** *fastforce*
have *d'*: *delta' x y x1 y1* ≠ *0*
using *bbb z2-d* **unfolding** *p-delta'-def* **by** *auto*
have (*x1*,*y1*) ∈ *e-aff*
unfolding *z2-d*[*symmetric*]
using ‹τ (*x'*, *y'*) ∈ *e-aff*› **by** *auto*
have *e-eq*: *e' x y* = *0 e' x1 y1* = *0*
using ‹(*x*,*y*) ∈ *e-aff*› ‹(*x1*,*y1*) ∈ *e-aff*› **unfolding** *e-aff-def* **by**(*auto*)

have *e' x2 y2* = *0*
using *z3-d z3-def ext-add-closure*[*OF d' e-eq*, *of x2 y2*] **by** *blast*
then show *?thesis*
unfolding *e-aff-def* **using** *e-e'-iff z3-d z3-def z2-d* **by** *simp*
qed

have *dom-eq*: (*dom* (λ(*x*, *y*). *proj-add x y*) ∩
{(((*x*, *y*), *l*), (*x'*, *y'*), *l'*), (((*x*, *y*), *l*), τ (*x'*, *y'*), *l'* + *1*)}) =
{(((*x*, *y*), *l*), (*x'*, *y'*), *l'*), (((*x*, *y*), *l*), τ (*x'*, *y'*), *l'* + *1*)}
using *aaa-simp v1* **by** *auto*
then have *proj-eq*: {*the* (*proj-add* ((*x*, *y*), *l*) ((*x'*, *y'*), *l'*)),
the (*proj-add* ((*x*, *y*), *l*) (τ (*x'*, *y'*), *l'* + *1*))} =
{(*ext-add* (*x*, *y*) (τ (*x'*, *y'*)), *l* + *l'* + *1*), (*ext-add* (*x*, *y*) (*x'*, *y'*), *l* + *l'*)}
using *aaa-simp v1* **by** *auto*
have *gluing* ‘‘ {(*ext-add* (*x*, *y*) (τ (*x'*, *y'*)), *l* + *l'* + *1*)} =
gluing ‘‘ {(*ext-add* (*x*, *y*) (*x'*, *y'*), *l* + *l'*)}
using ‹{((*x*, *y*), *l*)} // *gluing* = {{((*x*, *y*), *l*)}}› *gluing-class*[*OF z3 p-q-expr*(*2*)]
by (*simp add*: *singleton-quotient*)
then show *?thesis*
unfolding *2 proj-add-class-def* **apply**(*simp add*: *dom-eq proj-eq del*: *add.simps* τ*.simps ext-add.simps*)
unfolding *quotient-def* **by** *force*
next
case *ccc*
from *ccc* **have** *aaa-simp*:
proj-add ((*x*, *y*), *l*) (τ (*x'*, *y'*), *l'* + *1*) = *None*
by(*simp add*: *proj-add.simps p-q-expr*(*2*),*blast*)
then have *dom-eq*: (*dom* (λ(*x*, *y*). *proj-add x y*) ∩
{(((*x*, *y*), *l*), (*x'*, *y'*), *l'*),(((*x*, *y*), *l*), τ (*x'*, *y'*), *l'* + *1*)}) =

```
              {(((x, y), l),((x', y'), l'))}
            using v1 by auto
            then show ?thesis
              unfolding 2 proj-add-class-def
              apply(simp add: dom-eq del: τ.simps)
              unfolding quotient-def by simp
          qed
        qed
      qed
      qed
    next
      case 3
      then show ?thesis sorry
    next
      case 4
      then show ?thesis sorry
    qed
  qed

definition proj-addition c1 c2 = the-elem(proj-add-class c1 c2)

lemma projective-group-law:
  shows comm-group (|carrier = e-proj, mult = proj-addition, one = gluing ``
{((1,0),0)}|)
proof(unfold-locales,simp-all)
  show one-in: gluing `` {((1, 0), 0)} ∈ e-proj
    unfolding e-proj-def
    apply(rule quotientI)
    unfolding e-aff-bit-def Bits-def e-aff-def e'-def
    apply(simp)
    using zero-bit-def by blast

  show comm: ⋀x y. x ∈ e-proj ⟹
         y ∈ e-proj ⟹ proj-addition x y = proj-addition y x
    unfolding proj-addition-def using proj-add-class-comm by auto

  show id-1: ⋀x. x ∈ e-proj ⟹ proj-addition (gluing `` {((1, 0), 0)}) x = x
    unfolding proj-addition-def using proj-add-class-identity by simp

  show id-2: ⋀x. x ∈ e-proj ⟹ proj-addition x (gluing `` {((1, 0), 0)}) = x
     using comm id-1 one-in by simp
   oops

end

end
```