Group Law of Edwards Elliptic Curves

Rodrigo Raya

September 13, 2023

Abstract

This article gives an elementary computational proof of the group law for Edwards elliptic curves. The associative law is expressed as a polynomial identity over the integers that is directly checked by polynomial division. Unlike other proofs, no preliminaries such as intersection numbers, Bezouts theorem, projective geometry, divisors, or Riemann Roch are required. It supports the material of [1].

Contents

theory *Edwards-Elliptic-Curves-Group*
imports *HOL−Algebra.Group HOL−Library.Rewrite*
begin

1 Affine Edwards curves

class *ell-field* = *field* +
assumes *two-not-zero*: $2 \neq 0$

locale *curve-addition* =
fixes *c d* :: *'a*::*ell-field*
begin

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

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

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

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

lemma *delta-com*:
(*delta x0 y0 x1 y1* = 0) = (*delta x1 y1 x0 y0* = 0)
⟨*proof*⟩

fun *add* :: *'a* × *'a* ⇒ *'a* × *'a* ⇒ *'a* × *'a* **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 *commutativity*: *add z1 z2* = *add z2 z1*
⟨*proof*⟩

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*⟩

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*⟩

lemma *neutral*: *add z* (*1*,*0*) = *z* ⟨*proof*⟩

lemma *inverse*:
assumes *e a b* = *0 delta-plus a b a b* ≠ *0*
shows *add* (*a*,*b*) (*a*,−*b*) = (*1*,*0*)
⟨*proof*⟩

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*⟩

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*⟩

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*)|)
(**is** *comm-group ?g*)
⟨*proof*⟩

end

2 Extension

locale *ext-curve-addition* = *curve-addition* +
fixes *t'* :: *'a*::*ell-field*
assumes *c-eq-1*: *c* = *1*
assumes *t-intro*: *d* = *t'*^*2*
assumes *t-ineq*: *t'*^*2* ≠ *1 t'* ≠ *0*
begin

2.1 Change of variables

definition t **where** $t = t'$

lemma *t-nz*: $t \neq 0$ ⟨*proof*⟩

lemma *d-nz*: $d \neq 0$ ⟨*proof*⟩

lemma *t-expr*: $t\hat{}2 = d\ t\hat{}4 = d\hat{}2$ ⟨*proof*⟩

lemma *t-sq-n1*: $t\hat{}2 \neq 1$ ⟨*proof*⟩

lemma *t-nm1*: $t \neq -1$ ⟨*proof*⟩

lemma *d-n1*: $d \neq 1$ ⟨*proof*⟩

lemma *t-n1*: $t \neq 1$ ⟨*proof*⟩

lemma *t-dneq2*: $2{*}t \neq -2$
⟨*proof*⟩

2.2 New points

definition e' **where** $e'\ x\ y = x\hat{}2 + y\hat{}2 - 1 - t\hat{}2 * x\hat{}2 * y\hat{}2$

definition *e′-aff* $= \{(x,y).\ e'\ x\ y = 0\}$
definition *e-circ* $= \{(x,y).\ x \neq 0 \wedge y \neq 0 \wedge (x,y) \in$ *e′-aff*$\}$

lemma *e-e′-iff*: $e\ x\ y = 0 \longleftrightarrow e'\ x\ y = 0$
⟨*proof*⟩

lemma *circ-to-aff*: $p \in$ *e-circ* $\Longrightarrow p \in$ *e′-aff*
⟨*proof*⟩

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

lemma *t-2-1-lines*:
$t\hat{}2 = 1 \Longrightarrow e'\ x\ y = -\ (1 - x\hat{}2) * (1 - y\hat{}2)$
⟨*proof*⟩

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

lemma *t-0-circle*:
$t = 0 \Longrightarrow e'\ x\ y = x\hat{}2 + y\hat{}2 - 1$
⟨*proof*⟩

2.3 Group transformations and inversions

fun $\varrho :: 'a \times 'a \Rightarrow 'a \times 'a$ **where**
$\varrho\ (x,y) = (-y,x)$
fun $\tau :: 'a \times 'a \Rightarrow 'a \times 'a$ **where**

$\tau\ (x,y) = (1/(t*x),1/(t*y))$

definition G **where**
$G \equiv \{id,\varrho,\varrho \circ \varrho,\varrho \circ \varrho \circ \varrho,\tau,\tau \circ \varrho,\tau \circ \varrho \circ \varrho,\tau \circ \varrho \circ \varrho \circ \varrho\}$

definition *symmetries* **where**
$symmetries = \{\tau,\tau \circ \varrho,\tau \circ \varrho \circ \varrho,\tau \circ \varrho \circ \varrho \circ \varrho\}$

definition *rotations* **where**
$rotations = \{id,\varrho,\varrho \circ \varrho,\varrho \circ \varrho \circ \varrho\}$

lemma *G-partition*: $G = rotations \cup symmetries$
⟨*proof*⟩

lemma *tau-sq*: $(\tau \circ \tau)\ (x,y) = (x,y)$ ⟨*proof*⟩

lemma *tau-idemp*: $\tau \circ \tau = id$
⟨*proof*⟩

lemma *tau-idemp-explicit*: $\tau(\tau(x,y)) = (x,y)$
⟨*proof*⟩

lemma *tau-idemp-point*: $\tau(\tau\ p) = p$
⟨*proof*⟩

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

lemma *i-idemp*: $i \circ i = id$
⟨*proof*⟩

lemma *i-idemp-explicit*: $i(i(x,y)) = (x,y)$
⟨*proof*⟩

lemma *tau-rot-sym*:
assumes $r \in rotations$
shows $\tau \circ r \in symmetries$
⟨*proof*⟩

lemma *tau-rho-com*:
$\tau \circ \varrho = \varrho \circ \tau$ ⟨*proof*⟩

lemma *tau-rot-com*:
assumes $r \in rotations$
shows $\tau \circ r = r \circ \tau$
⟨*proof*⟩

lemma *rho-order-4*:
$\varrho \circ \varrho \circ \varrho \circ \varrho = id$ ⟨*proof*⟩

lemma *rho-i-com-inverses*:
$i\ (id\ (x,y)) = id\ (i\ (x,y))$
$i\ (\varrho\ (x,y)) = (\varrho \circ \varrho \circ \varrho)\ (i\ (x,y))$
$i\ ((\varrho \circ \varrho)\ (x,y)) = (\varrho \circ \varrho)\ (i\ (x,y))$
$i\ ((\varrho \circ \varrho \circ \varrho)\ (x,y)) = \varrho\ (i\ (x,y))$
⟨*proof*⟩

lemma *rotations-i-inverse*:
assumes $tr \in rotations$
shows $\exists\ tr' \in rotations.\ (tr \circ i)\ (x,y) = (i \circ tr')\ (x,y) \wedge tr \circ tr' = id$
⟨*proof*⟩

lemma *tau-i-com-inverses*:
$(i \circ \tau)\ (x,y) = (\tau \circ i)\ (x,y)$
$(i \circ \tau \circ \varrho)\ (x,y) = (\tau \circ \varrho \circ \varrho \circ \varrho \circ i)\ (x,y)$
$(i \circ \tau \circ \varrho \circ \varrho)\ (x,y) = (\tau \circ \varrho \circ \varrho \circ i)\ (x,y)$
$(i \circ \tau \circ \varrho \circ \varrho \circ \varrho)\ (x,y) = (\tau \circ \varrho \circ i)\ (x,y)$
⟨*proof*⟩

lemma *rho-circ*:
assumes $p \in e\text{-}circ$
shows $\varrho\ p \in e\text{-}circ$
⟨*proof*⟩

lemma *i-aff*:
assumes $p \in e'\text{-}aff$
shows $i\ p \in e'\text{-}aff$
⟨*proof*⟩

lemma *i-circ*:
assumes $(x,y) \in e\text{-}circ$
shows $i\ (x,y) \in e\text{-}circ$
⟨*proof*⟩

lemma *i-circ-points*:
assumes $p \in e\text{-}circ$
shows $i\ p \in e\text{-}circ$
⟨*proof*⟩

lemma *rot-circ*:
assumes $p \in e\text{-}circ\ tr \in rotations$
shows $tr\ p \in e\text{-}circ$
⟨*proof*⟩

lemma τ*-circ*:
assumes $p \in e\text{-}circ$
shows $\tau\ p \in e\text{-}circ$
⟨*proof*⟩

lemma *rot-comp*:
assumes *t1* ∈ *rotations t2* ∈ *rotations*
shows *t1* ∘ *t2* ∈ *rotations*
⟨*proof*⟩

lemma *rot-tau-com*:
assumes *tr* ∈ *rotations*
shows *tr* ∘ τ = τ ∘ *tr*
⟨*proof*⟩

lemma *tau-i-com*:
τ ∘ *i* = *i* ∘ τ ⟨*proof*⟩

lemma *rot-com*:
assumes *r* ∈ *rotations r′* ∈ *rotations*
shows *r′* ∘ *r* = *r* ∘ *r′*
⟨*proof*⟩

lemma *rot-inv*:
assumes *r* ∈ *rotations*
shows ∃ *r′* ∈ *rotations*. *r′* ∘ *r* = *id*
⟨*proof*⟩

lemma *rot-aff*:
assumes *r* ∈ *rotations p* ∈ *e′-aff*
shows *r p* ∈ *e′-aff*
⟨*proof*⟩

lemma *rot-delta*:
assumes *r* ∈ *rotations delta x1 y1 x2 y2* ≠ *0*
shows *delta* (*fst* (*r* (*x1*,*y1*))) (*snd* (*r* (*x1*,*y1*))) *x2 y2* ≠ *0*
⟨*proof*⟩

lemma *tau-not-id*: τ ≠ *id*
⟨*proof*⟩

lemma *sym-not-id*:
assumes *r* ∈ *rotations*
shows τ ∘ *r* ≠ *id*
⟨*proof*⟩

lemma *sym-decomp*:
assumes *g* ∈ *symmetries*
shows ∃ *r* ∈ *rotations*. *g* = τ ∘ *r*
⟨*proof*⟩

lemma *symmetries-i-inverse*:

assumes $tr \in symmetries$
shows $\exists\ tr' \in symmetries.\ (tr \circ i)\ (x,y) = (i \circ tr')\ (x,y) \wedge tr \circ tr' = id$
⟨*proof*⟩

lemma *sym-to-rot*: $g \in symmetries \Longrightarrow \tau \circ g \in rotations$
⟨*proof*⟩

2.4 Extended addition

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

definition *delta-x* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **where**
delta-x x1 y1 x2 y2 $= x2*y1 - x1*y2$
definition *delta-y* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **where**
delta-y x1 y1 x2 y2 $= x1*x2 + y1*y2$
definition *delta'* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **where**
delta' x1 y1 x2 y2 $=$ *delta-x x1 y1 x2 y2* $*$ *delta-y x1 y1 x2 y2*

lemma *delta'-com*: $(delta'\ x0\ y0\ x1\ y1 = 0) = (delta'\ x1\ y1\ x0\ y0 = 0)$
⟨*proof*⟩

definition *e'-aff-0* **where**
e'-aff-0 $= \{((x1,y1),(x2,y2)).\ (x1,y1) \in e'\text{-}aff \wedge$
$(x2,y2) \in e'\text{-}aff \wedge$
$delta\ x1\ y1\ x2\ y2 \neq 0\ \}$

definition *e'-aff-1* **where**
e'-aff-1 $= \{((x1,y1),(x2,y2)).\ (x1,y1) \in e'\text{-}aff \wedge$
$(x2,y2) \in e'\text{-}aff \wedge$
$delta'\ x1\ y1\ x2\ y2 \neq 0\ \}$

lemma *ext-add-comm*:
ext-add $(x1,y1)\ (x2,y2) =$ *ext-add* $(x2,y2)\ (x1,y1)$
⟨*proof*⟩

lemma *ext-add-comm-points*:
ext-add z1 z2 = ext-add z2 z1
⟨*proof*⟩

lemma *ext-add-inverse*:
$x \neq 0 \Longrightarrow y \neq 0 \Longrightarrow$ *ext-add* $(x,y)\ (i\ (x,y)) = (1,0)$
⟨*proof*⟩

lemma *ext-add-deltas*:
ext-add $(x1,y1)\ (x2,y2) =$
$((delta\text{-}x\ x2\ y1\ x1\ y2)\ div\ (delta\text{-}x\ x1\ y1\ x2\ y2),$

$(delta\text{-}y\ x1\ x2\ y1\ y2)\ div\ (delta\text{-}y\ x1\ y1\ x2\ y2))$
⟨*proof*⟩

2.4.1 Inversion and rotation invariance

lemma *inversion-invariance-1*:
assumes $x1 \neq 0\ y1 \neq 0\ x2 \neq 0\ y2 \neq 0$
shows $add\ (\tau\ (x1,y1))\ (x2,y2) = add\ (x1,y1)\ (\tau\ (x2,y2))$
⟨*proof*⟩

lemma *inversion-invariance-2*:
assumes $x1 \neq 0\ y1 \neq 0\ x2 \neq 0\ y2 \neq 0$
shows $ext\text{-}add\ (\tau\ (x1,y1))\ (x2,y2) = ext\text{-}add\ (x1,y1)\ (\tau\ (x2,y2))$
⟨*proof*⟩

lemma *rho-invariance-1*:
$add\ (\varrho\ (x1,y1))\ (x2,y2) = \varrho\ (add\ (x1,y1)\ (x2,y2))$
⟨*proof*⟩

lemma *rho-invariance-1-points*:
$add\ (\varrho\ p1)\ p2 = \varrho\ (add\ p1\ p2)$
⟨*proof*⟩

lemma *rho-invariance-2*:
$ext\text{-}add\ (\varrho\ (x1,y1))\ (x2,y2) =$
$\varrho\ (ext\text{-}add\ (x1,y1)\ (x2,y2))$
⟨*proof*⟩

lemma *rho-invariance-2-points*:
$ext\text{-}add\ (\varrho\ p1)\ p2 = \varrho\ (ext\text{-}add\ p1\ p2)$
⟨*proof*⟩

lemma *rotation-invariance-1*:
assumes $r \in rotations$
shows $add\ (r\ (x1,y1))\ (x2,y2) = r\ (add\ (x1,y1)\ (x2,y2))$
⟨*proof*⟩

lemma *rotation-invariance-1-points*:
assumes $r \in rotations$
shows $add\ (r\ p1)\ p2 = r\ (add\ p1\ p2)$
⟨*proof*⟩

lemma *rotation-invariance-2*:
assumes $r \in rotations$
shows $ext\text{-}add\ (r\ (x1,y1))\ (x2,y2) = r\ (ext\text{-}add\ (x1,y1)\ (x2,y2))$
⟨*proof*⟩

lemma *rotation-invariance-2-points*:
assumes $r \in rotations$
shows $ext\text{-}add\ (r\ p1)\ p2 = r\ (ext\text{-}add\ p1\ p2)$
⟨*proof*⟩

lemma *rotation-invariance-3*:
$delta\ x1\ y1\ (fst\ (\varrho\ (x2,y2)))\ (snd\ (\varrho\ (x2,y2))) =$
$delta\ x1\ y1\ x2\ y2$
⟨*proof*⟩

lemma *rotation-invariance-4*:
$delta'\ x1\ y1\ (fst\ (\varrho\ (x2,y2)))\ (snd\ (\varrho\ (x2,y2))) = -\ delta'\ x1\ y1\ x2\ y2$
⟨*proof*⟩

lemma *rotation-invariance-5*:
$delta'\ (fst\ (\varrho\ (x1,y1)))\ (snd\ (\varrho\ (x1,y1)))\ x2\ y2 = -\ delta'\ x1\ y1\ x2\ y2$
⟨*proof*⟩

lemma *rotation-invariance-6*:
$delta\ (fst\ (\varrho\ (x1,y1)))\ (snd\ (\varrho\ (x1,y1)))\ x2\ y2 = delta\ x1\ y1\ x2\ y2$
⟨*proof*⟩

lemma *inverse-rule-1*:
$(\tau \circ i \circ \tau)\ (x,y) = i\ (x,y)$ ⟨*proof*⟩
lemma *inverse-rule-2*:
$(\varrho \circ i \circ \varrho)\ (x,y) = i\ (x,y)$ ⟨*proof*⟩
lemma *inverse-rule-3*:
$i\ (add\ (x1,y1)\ (x2,y2)) = add\ (i\ (x1,y1))\ (i\ (x2,y2))$
⟨*proof*⟩
lemma *inverse-rule-4*:
$i\ (ext\text{-}add\ (x1,y1)\ (x2,y2)) = ext\text{-}add\ (i\ (x1,y1))\ (i\ (x2,y2))$
⟨*proof*⟩

lemma *e′-aff-x0*:
assumes $x = 0\ (x,y) \in e'\text{-}aff$
shows $y = 1 \vee y = -1$
⟨*proof*⟩

lemma *e′-aff-y0*:
assumes $y = 0\ (x,y) \in e'\text{-}aff$
shows $x = 1 \vee x = -1$
⟨*proof*⟩

lemma *add-ext-add*:
assumes $x1 \neq 0\ y1 \neq 0$
shows $ext\text{-}add\ (x1,y1)\ (x2,y2) = \tau\ (add\ (\tau\ (x1,y1))\ (x2,y2))$

⟨*proof*⟩

corollary *add-ext-add-2*:
assumes *x1* ≠ *0 y1* ≠ *0*
shows *add* (*x1*,*y1*) (*x2*,*y2*) = τ (*ext-add* (τ (*x1*,*y1*)) (*x2*,*y2*))
⟨*proof*⟩

2.4.2 Coherence and closure

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *ext-add-closure-points*:
assumes *delta′ x1 y1 x2 y2* ≠ *0*
assumes (*x1*,*y1*) ∈ *e′-aff* (*x2*,*y2*) ∈ *e′-aff*
shows *ext-add* (*x1*,*y1*) (*x2*,*y2*) ∈ *e′-aff*
⟨*proof*⟩

2.4.3 Useful lemmas in the extension

lemma *inverse-generalized*:
assumes (*a*,*b*) ∈ *e′-aff delta-plus a b a b* ≠ *0*
shows *add* (*a*,*b*) (*a*,−*b*) = (*1*,*0*)
⟨*proof*⟩

lemma *inverse-generalized-points*:
assumes $p \in e'\text{-}aff\ delta\text{-}plus\ (fst\ p)\ (snd\ p)\ (fst\ p)\ (snd\ p) \neq 0$
shows $add\ p\ (i\ p) = (1,0)$
⟨*proof*⟩

lemma *add-closure-points*:
assumes $delta\ x\ y\ x'\ y' \neq 0$
$(x,y) \in e'\text{-}aff\ (x',y') \in e'\text{-}aff$
shows $add\ (x,y)\ (x',y') \in e'\text{-}aff$
⟨*proof*⟩

lemma *add-self*:
assumes *in-aff*: $(x,y) \in e'\text{-}aff$
shows $delta\ x\ y\ x\ (-y) \neq 0 \vee delta'\ x\ y\ x\ (-y) \neq 0$
⟨*proof*⟩

2.5 Delta arithmetic

lemma *mix-tau*:
assumes $(x1,y1) \in e'\text{-}aff\ (x2,y2) \in e'\text{-}aff\ x2 \neq 0\ y2 \neq 0$
assumes $delta'\ x1\ y1\ x2\ y2 \neq 0\ delta'\ x1\ y1\ (fst\ (\tau\ (x2,y2)))\ (snd\ (\tau\ (x2,y2))) \neq 0$
shows $delta\ x1\ y1\ x2\ y2 \neq 0$
⟨*proof*⟩

lemma *mix-tau-0*:
assumes $(x1,y1) \in e'\text{-}aff\ (x2,y2) \in e'\text{-}aff\ x2 \neq 0\ y2 \neq 0$
assumes $delta\ x1\ y1\ x2\ y2 = 0$
shows $delta'\ x1\ y1\ x2\ y2 = 0 \vee delta'\ x1\ y1\ (fst\ (\tau\ (x2,y2)))\ (snd\ (\tau\ (x2,y2))) = 0$
⟨*proof*⟩

lemma *mix-tau-prime*:
assumes $(x1,y1) \in e'\text{-}aff\ (x2,y2) \in e'\text{-}aff\ x2 \neq 0\ y2 \neq 0$
assumes $delta\ x1\ y1\ x2\ y2 \neq 0\ delta\ x1\ y1\ (fst\ (\tau\ (x2,y2)))\ (snd\ (\tau\ (x2,y2))) \neq 0$
shows $delta'\ x1\ y1\ x2\ y2 \neq 0$
⟨*proof*⟩

lemma *tau-tau-d*:
assumes $(x1,y1) \in e'\text{-}aff\ (x2,y2) \in e'\text{-}aff$
assumes $delta\ (fst\ (\tau\ (x1,y1)))\ (snd\ (\tau\ (x1,y1)))\ (fst\ (\tau\ (x2,y2)))\ (snd\ (\tau\ (x2,y2))) \neq 0$
shows $delta\ x1\ y1\ x2\ y2 \neq 0$
⟨*proof*⟩

lemma *tau-tau-d'*:

assumes $(x1,y1) \in e'\text{-}aff\ (x2,y2) \in e'\text{-}aff$
assumes $delta'\ (fst\ (\tau\ (x1,y1)))\ (snd\ (\tau\ (x1,y1)))\ (fst\ (\tau\ (x2,y2)))\ (snd\ (\tau\ (x2,y2))) \neq 0$
shows $delta'\ x1\ y1\ x2\ y2 \neq 0$
⟨*proof*⟩

lemma *delta-add-delta'-1*:
assumes *1*: $x1 \neq 0\ y1 \neq 0\ x2 \neq 0\ y2 \neq 0$
assumes *r-expr*: $rx = fst\ (add\ (x1,y1)\ (x2,y2))\ ry = snd\ (add\ (x1,y1)\ (x2,y2))$

assumes *in-aff*: $(x1,y1) \in e'\text{-}aff\ (x2,y2) \in e'\text{-}aff$
assumes *pd*: $delta\ x1\ y1\ x2\ y2 \neq 0$
assumes *pd'*: $delta\ rx\ ry\ (fst\ (\tau\ (i\ (x2,y2))))\ (snd\ (\tau\ (i\ (x2,y2)))) \neq 0$
shows $delta'\ rx\ ry\ (fst\ (i\ (x2,y2)))\ (snd\ (i\ (x2,y2))) \neq 0$
⟨*proof*⟩

lemma *delta'-add-delta-1*:
assumes *1*: $x1 \neq 0\ y1 \neq 0\ x2 \neq 0\ y2 \neq 0$
assumes *r-expr*: $rx = fst\ (ext\text{-}add\ (x1,y1)\ (x2,y2))\ ry = snd\ (ext\text{-}add\ (x1,y1)\ (x2,y2))$
assumes *in-aff*: $(x1,y1) \in e'\text{-}aff\ (x2,y2) \in e'\text{-}aff$
assumes *pd'*: $delta'\ rx\ ry\ (fst\ (\tau\ (i\ (x2,y2))))\ (snd\ (\tau\ (i\ (x2,y2)))) \neq 0$
shows $delta\ rx\ ry\ (fst\ (i\ (x2,y2)))\ (snd\ (i\ (x2,y2))) \neq 0$
⟨*proof*⟩

lemma *funny-field-lemma-1*:
$((x1 * x2 - y1 * y2) * ((x1 * x2 - y1 * y2) * (x2 * (y2 * (1 + d * x1 * y1 * x2 * y2)))) +$
$(x1 * x2 - y1 * y2) * ((x1 * y2 + y1 * x2) * y2^2) * (1 - d * x1 * y1 * x2 * y2)) *$
$(1 + d * x1 * y1 * x2 * y2) \neq$
$((x1 * y2 + y1 * x2) * ((x1 * y2 + y1 * x2) * (x2 * (y2 * (1 - d * x1 * y1 * x2 * y2)))) +$
$(x1 * x2 - y1 * y2) * ((x1 * y2 + y1 * x2) * x2^2) * (1 + d * x1 * y1 * x2 * y2)) *$
$(1 - d * x1 * y1 * x2 * y2) \Longrightarrow$
$(d * ((x1 * x2 - y1 * y2) * ((x1 * y2 + y1 * x2) * (x2 * y2))))^2 =$
$((1 - d * x1 * y1 * x2 * y2) * (1 + d * x1 * y1 * x2 * y2))^2 \Longrightarrow$
$x1^2 + y1^2 - 1 = d * x1^2 * y1^2 \Longrightarrow$
$x2^2 + y2^2 - 1 = d * x2^2 * y2^2 \Longrightarrow False$
⟨*proof*⟩

lemma *delta-add-delta'-2*:
assumes *1*: $x1 \neq 0\ y1 \neq 0\ x2 \neq 0\ y2 \neq 0$
assumes *r-expr*: $rx = fst\ (add\ (x1,y1)\ (x2,y2))\ ry = snd\ (add\ (x1,y1)\ (x2,y2))$

assumes *in-aff*: $(x1,y1) \in e'\text{-}aff\ (x2,y2) \in e'\text{-}aff$
assumes *pd*: $delta\ x1\ y1\ x2\ y2 \neq 0$

assumes *pd'*: *delta' rx ry* (*fst* (*τ* (*i* (*x2*,*y2*)))) (*snd* (*τ* (*i* (*x2*,*y2*)))) ≠ *0*
shows *delta rx ry* (*fst* (*i* (*x2*,*y2*))) (*snd* (*i* (*x2*,*y2*))) ≠ *0*
⟨*proof*⟩

lemma *funny-field-lemma-2*: $(x2 * y2)^2 * ((x2 * y1 - x1 * y2) * (x1 * x2 + y1 * y2))^2 \neq ((x1 * y1 - x2 * y2) * (x1 * y1 + x2 * y2))^2 \Longrightarrow$
$((x1 * y1 - x2 * y2) * ((x1 * y1 - x2 * y2) * (x2 * (y2 * (x1 * x2 + y1 * y2)))) +$
$(x1 * y1 - x2 * y2) * ((x1 * y1 + x2 * y2) * x2^2) * (x2 * y1 - x1 * y2)) * (x1 * x2 + y1 * y2) =$
$((x1 * y1 + x2 * y2) * ((x1 * y1 + x2 * y2) * (x2 * (y2 * (x2 * y1 - x1 * y2)))) +$
$(x1 * y1 - x2 * y2) * ((x1 * y1 + x2 * y2) * y2^2) * (x1 * x2 + y1 * y2)) * (x2 * y1 - x1 * y2) \Longrightarrow$
$x1^2 + y1^2 - 1 = d * x1^2 * y1^2 \Longrightarrow$
$x2^2 + y2^2 - 1 = d * x2^2 * y2^2 \Longrightarrow$ *False*
⟨*proof*⟩

lemma *delta'-add-delta-2*:
assumes *1*: *x1* ≠ *0 y1* ≠ *0 x2* ≠ *0 y2* ≠ *0*
assumes *r-expr*: *rx* = *fst* (*ext-add* (*x1*,*y1*) (*x2*,*y2*)) *ry* = *snd* (*ext-add* (*x1*,*y1*) (*x2*,*y2*))
assumes *in-aff*: (*x1*,*y1*) ∈ *e'-aff* (*x2*,*y2*) ∈ *e'-aff*
assumes *pd*: *delta' x1 y1 x2 y2* ≠ *0*
assumes *pd'*: *delta rx ry* (*fst* (*τ* (*i* (*x2*,*y2*)))) (*snd* (*τ* (*i* (*x2*,*y2*)))) ≠ *0*
shows *delta' rx ry* (*fst* (*i* (*x2*,*y2*))) (*snd* (*i* (*x2*,*y2*))) ≠ *0*
⟨*proof*⟩

lemma *delta'-add-delta-not-add*:
assumes *1*: *x1* ≠ *0 y1* ≠ *0 x2* ≠ *0 y2* ≠ *0*
assumes *in-aff*: (*x1*,*y1*) ∈ *e'-aff* (*x2*,*y2*) ∈ *e'-aff*
assumes *pd*: *delta' x1 y1 x2 y2* ≠ *0*
assumes *add-nz*: *fst* (*ext-add* (*x1*,*y1*) (*x2*,*y2*)) ≠ *0 snd* (*ext-add* (*x1*,*y1*) (*x2*,*y2*)) ≠ *0*
shows *pd'*: *delta* (*fst* (*τ* (*x1*,*y1*))) (*snd* (*τ* (*x1*,*y1*))) *x2 y2* ≠ *0*
⟨*proof*⟩

lemma *not-add-self*:
assumes *in-aff*: (*x*,*y*) ∈ *e'-aff x* ≠ *0 y* ≠ *0*
shows *delta x y* (*fst* (*τ* (*i* (*x*,*y*)))) (*snd* (*τ* (*i* (*x*,*y*)))) = *0*
delta' x y (*fst* (*τ* (*i* (*x*,*y*)))) (*snd* (*τ* (*i* (*x*,*y*)))) = *0*
⟨*proof*⟩

3 Projective Edwards curves

3.1 No fixed-point lemma and dichotomies

lemma *g-no-fp*:
assumes $g \in G$ $p \in$ *e-circ* g $p = p$
shows $g = id$
⟨*proof*⟩

lemma *dichotomy-1*:
assumes $p \in e'$*-aff* $q \in e'$*-aff*
shows $(p \in$ *e-circ* $\wedge$ $(\exists\ g \in$ *symmetries*. $q = (g \circ i)\ p)) \vee$
$(p,q) \in e'$*-aff-0* $\vee$ $(p,q) \in e'$*-aff-1*
⟨*proof*⟩

lemma *dichotomy-2*:
assumes *add* $(x1,y1)$ $(x2,y2) = (1,0)$
$((x1,y1),(x2,y2)) \in e'$*-aff-0*
shows $(x2,y2) = i$ $(x1,y1)$
⟨*proof*⟩

lemma *dichotomy-3*:
assumes *ext-add* $(x1,y1)$ $(x2,y2) = (1,0)$
$((x1,y1),(x2,y2)) \in e'$*-aff-1*
shows $(x2,y2) = i$ $(x1,y1)$
⟨*proof*⟩

3.1.1 Meaning of dichotomy condition on deltas

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' $y' = 0$
⟨*proof*⟩

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' $y' = 0$
⟨*proof*⟩

lemma *meaning-of-dichotomy-1*:
assumes $(\exists\, g \in$ *symmetries*. $(x2,\ y2) = (g \circ i)$ $(x1,\ y1))$
shows *fst* $(add$ $(x1,y1)$ $(x2,y2)) = 0 \vee$ *snd* $(add$ $(x1,y1)$ $(x2,y2)) = 0$
⟨*proof*⟩

lemma *meaning-of-dichotomy-2*:
assumes $(\exists\, g \in$ *symmetries*. $(x2,\ y2) = (g \circ i)$ $(x1,\ y1))$
shows *fst* $(ext\text{-}add$ $(x1,y1)$ $(x2,y2)) = 0 \vee$ *snd* $(ext\text{-}add$ $(x1,y1)$ $(x2,y2)) = 0$
⟨*proof*⟩

3.2 Gluing relation and projective points

definition *gluing* :: $(((\prime a \times \prime a) \times bool) \times ((\prime a \times \prime a) \times bool))$ *set* **where**
gluing = {(((*x0*,*y0*),*l*),((*x1*,*y1*),*j*)).
((*x0*,*y0*) ∈ *e′-aff* ∧ (*x1*,*y1*) ∈ *e′-aff*) ∧
((*x0* ≠ *0* ∧ *y0* ≠ *0* ∧ (*x1*,*y1*) = *τ* (*x0*,*y0*) ∧ *j* = *Not l*) ∨
(*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* = *Not j* ∧ *x0* ≠ *0* ∧ *y0* ≠ *0*)
⟨*proof*⟩

lemma *gluing-char-zero*:
assumes (((*x0*,*y0*),*l*),((*x1*,*y1*),*j*)) ∈ *gluing x0* = *0* ∨ *y0* = *0*
shows (*x0*,*y0*) = (*x1*,*y1*) ∧ *l* = *j*
⟨*proof*⟩

lemma *gluing-aff*:
assumes (((*x0*,*y0*),*l*),((*x1*,*y1*),*j*)) ∈ *gluing*
shows (*x0*,*y0*) ∈ *e′-aff* (*x1*,*y1*) ∈ *e′-aff*
⟨*proof*⟩

definition *e′-aff-bit* :: $((\prime a \times \prime a) \times bool)$ *set* **where**
e′-aff-bit = *e′-aff* × *UNIV*

lemma *eq-rel*: *equiv e′-aff-bit gluing*
⟨*proof*⟩

lemma *gluing-eq*: *x* = *y* ⟹ *gluing* `` {*x*} = *gluing* `` {*y*} ⟨*proof*⟩

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

3.2.1 Point-class classification

lemma *eq-class-simp*:
assumes *X* ∈ *e-proj X* ≠ {}
shows *X* // *gluing* = {*X*}
⟨*proof*⟩

lemma *gluing-class-1*:
assumes *x* = *0* ∨ *y* = *0* (*x*,*y*) ∈ *e′-aff*
shows *gluing* `` {((*x*,*y*), *l*)} = {((*x*,*y*), *l*)}
⟨*proof*⟩

lemma *gluing-class-2*:
assumes *x* ≠ *0* *y* ≠ *0* (*x*,*y*) ∈ *e′-aff*
shows *gluing* `` {((*x*,*y*), *l*)} = {((*x*,*y*), *l*), (*τ* (*x*,*y*), *Not l*)}
⟨*proof*⟩

lemma *e-proj-elim-1*:
assumes $(x,y) \in e'$-*aff*
shows $\{((x,y),l)\} \in$ *e-proj* $\longleftrightarrow x = 0 \vee y = 0$
⟨*proof*⟩

lemma *e-proj-elim-2*:
assumes $(x,y) \in e'$-*aff*
shows $\{((x,y),l),(\tau\ (x,y),Not\ l)\} \in$ *e-proj* $\longleftrightarrow x \neq 0 \wedge y \neq 0$
⟨*proof*⟩

lemma *e-proj-eq*:
assumes $p \in$ *e-proj*
shows $\exists\ x\ y\ l.\ (p = \{((x,y),l)\} \vee p = \{((x,y),l),(\tau\ (x,y),Not\ l)\}) \wedge (x,y) \in e'$-*aff*

⟨*proof*⟩

lemma *e-proj-aff*:
gluing ‘‘ $\{((x,y),l)\} \in$ *e-proj* $\longleftrightarrow (x,y) \in e'$-*aff*
⟨*proof*⟩

lemma *gluing-cases*:
assumes $x \in$ *e-proj*
obtains *x0 y0 l* **where** $x = \{((x0,y0),l)\} \vee x = \{((x0,y0),l),(\tau\ (x0,y0),Not\ l)\}$
⟨*proof*⟩

lemma *gluing-cases-explicit*:
assumes $x \in$ *e-proj* $x =$ *gluing* ‘‘ $\{((x0,y0),l)\}$
shows $x = \{((x0,y0),l)\} \vee x = \{((x0,y0),l),(\tau\ (x0,y0),Not\ l)\}$
⟨*proof*⟩

lemma *gluing-cases-points*:
assumes $x \in$ *e-proj* $x =$ *gluing* ‘‘ $\{(p,l)\}$
shows $x = \{(p,l)\} \vee x = \{(p,l),(\tau\ p,Not\ l)\}$
⟨*proof*⟩

lemma *identity-equiv*:
gluing ‘‘ $\{((1,\ 0),\ l)\} = \{((1,0),l)\}$
⟨*proof*⟩

lemma *identity-proj*:
$\{((1,0),l)\} \in$ *e-proj*
⟨*proof*⟩

lemma *gluing-inv*:
assumes $x \neq 0\ y \neq 0\ (x,y) \in e'$-*aff*
shows *gluing* ‘‘ $\{((x,y),j)\} =$ *gluing* ‘‘ $\{(\tau\ (x,y),\ Not\ j)\}$
⟨*proof*⟩

3.3 Projective addition on points

definition *xor* :: *bool* => *bool* ⇒ *bool*
  where *xor-def*: *xor P Q* ≡ (*P* ∧ ¬ *Q*) ∨ (¬ *P* ∧ *Q*)

function (*domintros*) *proj-add* :: ('*a* × '*a*) × *bool* ⇒ ('*a* × '*a*) × *bool* ⇒ ('*a* × '*a*) × *bool*
  where
    proj-add ((*x1*, *y1*), *l*) ((*x2*, *y2*), *j*) = (*add* (*x1*, *y1*) (*x2*, *y2*), *xor l j*)
   if *delta x1 y1 x2 y2* ≠ *0* **and**
     (*x1*, *y1*) ∈ *e'-aff* **and**
     (*x2*, *y2*) ∈ *e'-aff*
  | *proj-add* ((*x1*, *y1*), *l*) ((*x2*, *y2*), *j*) = (*ext-add* (*x1*, *y1*) (*x2*, *y2*), *xor l j*)
   if *delta' x1 y1 x2 y2* ≠ *0* **and**
     (*x1*, *y1*) ∈ *e'-aff* **and**
     (*x2*, *y2*) ∈ *e'-aff*
  | *proj-add* ((*x1*, *y1*), *l*) ((*x2*, *y2*), *j*) = *undefined*
   if (*x1*, *y1*) ∉ *e'-aff* ∨ (*x2*, *y2*) ∉ *e'-aff* ∨
        (*delta x1 y1 x2 y2* = *0* ∧ *delta' x1 y1 x2 y2* = *0*)
  ⟨*proof*⟩

termination *proj-add* ⟨*proof*⟩

lemma *proj-add-inv*:
  assumes (*x0*,*y0*) ∈ *e'-aff*
  shows *proj-add* ((*x0*,*y0*),*l*) (*i* (*x0*,*y0*),*l'*) = ((*1*,*0*),*xor l l'*)
⟨*proof*⟩

lemma *proj-add-comm*:
  proj-add ((*x0*,*y0*),*l*) ((*x1*,*y1*),*j*) = *proj-add* ((*x1*,*y1*),*j*) ((*x0*,*y0*),*l*)
⟨*proof*⟩

3.4 Projective addition on classes

function (*domintros*) *proj-add-class* :: (('*a* × '*a*) × *bool*) *set* ⇒
                                (('*a* × '*a*) × *bool*) *set* ⇒
                                ((('*a* × '*a*) × *bool*) *set*) *set*
  where
    proj-add-class c1 c2 =
      (
        {
          proj-add ((*x1*, *y1*), *i*) ((*x2*, *y2*), *j*) |
            x1 y1 i x2 y2 j.
            ((*x1*, *y1*), *i*) ∈ *c1* ∧
            ((*x2*, *y2*), *j*) ∈ *c2* ∧
            ((*x1*, *y1*), (*x2*, *y2*)) ∈ *e'-aff-0* ∪ *e'-aff-1*
        } // *gluing*
)
   if *c1* ∈ *e-proj* **and** *c2* ∈ *e-proj*
  | *proj-add-class c1 c2* = *undefined*

if *c1* $\notin$ *e-proj* $\vee$ *c2* $\notin$ *e-proj*
⟨*proof*⟩

termination *proj-add-class* ⟨*proof*⟩

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

3.4.1 Covering

corollary *no-fp-eq*:
assumes *p* ∈ *e-circ*
assumes *r′* ∈ *rotations r* ∈ *rotations*
assumes (*r′* ∘ *i*) *p* = (*τ* ∘ *r*) (*i p*)
shows *False*
⟨*proof*⟩

lemma *covering*:
assumes *p* ∈ *e-proj q* ∈ *e-proj*
shows *proj-add-class p q* ≠ {}
⟨*proof*⟩

lemma *covering-with-deltas*:
assumes (*gluing* ‘‘ {((*x*,*y*),*l*)}) ∈ *e-proj* (*gluing* ‘‘ {((*x′*,*y′*),*l′*)}) ∈ *e-proj*
shows *delta x y x′ y′* ≠ *0* ∨ *delta′ x y x′ y′* ≠ *0* ∨
delta x y (*fst* (*τ* (*x′*,*y′*))) (*snd* (*τ* (*x′*,*y′*))) ≠ *0* ∨
delta′ x y (*fst* (*τ* (*x′*,*y′*))) (*snd* (*τ* (*x′*,*y′*))) ≠ *0*
⟨*proof*⟩

3.4.2 Independence of the representant

lemma *proj-add-class-comm*:
assumes *c1* ∈ *e-proj c2* ∈ *e-proj*
shows *proj-add-class c1 c2* = *proj-add-class c2 c1*
⟨*proof*⟩

lemma *gluing-add-1*:
assumes *gluing* ‘‘ {((*x*,*y*),*l*)} = {((*x*, *y*), *l*)} *gluing* ‘‘ {((*x′*,*y′*),*l′*)} = {((*x′*, *y′*), *l′*)}
gluing ‘‘ {((*x*,*y*),*l*)} ∈ *e-proj gluing* ‘‘ {((*x′*,*y′*),*l′*)} ∈ *e-proj delta x y x′ y′* ≠ *0*
shows *proj-addition* (*gluing* ‘‘ {((*x*,*y*),*l*)}) (*gluing* ‘‘ {((*x′*,*y′*),*l′*)}) = (*gluing* ‘‘ {(*add* (*x*,*y*) (*x′*,*y′*), *xor l l′*)})
⟨*proof*⟩

lemma *gluing-add-2*:
assumes *gluing* ‘‘ {((*x*,*y*),*l*)} = {((*x*, *y*), *l*)} *gluing* ‘‘ {((*x′*,*y′*),*l′*)} = {((*x′*, *y′*), *l′*), (*τ* (*x′*, *y′*), *Not l′*)}

gluing ‘‘ {((x,y),l)} ∈ e-proj gluing ‘‘ {((x′,y′),l′)} ∈ e-proj delta x y x′ y′ ≠ 0
shows *proj-addition (gluing ‘‘ {((x,y),l)}) (gluing ‘‘ {((x′,y′),l′)}) = (gluing ‘‘ {(add (x,y) (x′,y′), xor l l′)})*
⟨*proof*⟩

lemma *gluing-add-4*:
assumes *gluing ‘‘ {((x, y), l)} = {((x, y), l), (τ (x, y), Not l)}*
gluing ‘‘ {((x′, y′), l′)} = {((x′, y′), l′), (τ (x′, y′), Not l′)}
gluing ‘‘ {((x, y), l)} ∈ e-proj gluing ‘‘ {((x′, y′), l′)} ∈ e-proj delta x y x′ y′ ≠ 0
shows *proj-addition (gluing ‘‘ {((x, y), l)}) (gluing ‘‘ {((x′, y′), l′)}) =*
gluing ‘‘ {(add (x, y) (x′,y′), xor l l′)}
(**is** *proj-addition ?p ?q = -*)
⟨*proof*⟩

lemma *gluing-add*:
assumes *gluing ‘‘ {((x1,y1),l)} ∈ e-proj gluing ‘‘ {((x2,y2),j)} ∈ e-proj delta x1 y1 x2 y2 ≠ 0*
shows *proj-addition (gluing ‘‘ {((x1,y1),l)}) (gluing ‘‘ {((x2,y2),j)}) =*
(gluing ‘‘ {(add (x1,y1) (x2,y2), xor l j)})
⟨*proof*⟩

lemma *gluing-ext-add-1*:
assumes *gluing ‘‘ {((x,y),l)} = {((x, y), l)} gluing ‘‘ {((x′,y′),l′)} = {((x′, y′), l′)}*
gluing ‘‘ {((x,y),l)} ∈ e-proj gluing ‘‘ {((x′,y′),l′)} ∈ e-proj delta′ x y x′ y′ ≠ 0
shows *proj-addition (gluing ‘‘ {((x,y),l)}) (gluing ‘‘ {((x′,y′),l′)}) =*
(gluing ‘‘ {(ext-add (x,y) (x′,y′), xor l l′)})
⟨*proof*⟩

lemma *gluing-ext-add-2*:
assumes *gluing ‘‘ {((x,y),l)} = {((x, y), l)} gluing ‘‘ {((x′,y′),l′)} = {((x′, y′), l′), (τ (x′, y′), Not l′)}*
gluing ‘‘ {((x,y),l)} ∈ e-proj gluing ‘‘ {((x′,y′),l′)} ∈ e-proj delta′ x y x′ y′ ≠ 0
shows *proj-addition (gluing ‘‘ {((x,y),l)}) (gluing ‘‘ {((x′,y′),l′)}) = (gluing ‘‘ {(ext-add (x,y) (x′,y′), xor l l′)})*
⟨*proof*⟩

lemma *gluing-ext-add-4*:
assumes *gluing ‘‘ {((x,y),l)} = {((x, y), l), (τ (x, y), Not l)}*
gluing ‘‘ {((x′,y′),l′)} = {((x′, y′), l′), (τ (x′, y′), Not l′)}
gluing ‘‘ {((x,y),l)} ∈ e-proj gluing ‘‘ {((x′,y′),l′)} ∈ e-proj
delta′ x y x′ y′ ≠ 0
shows *proj-addition (gluing ‘‘ {((x,y),l)}) (gluing ‘‘ {((x′,y′),l′)}) = (gluing ‘‘*

{(*ext-add* (*x*,*y*) (*x′*,*y′*),*xor l l′*)})
(**is** *proj-addition ?p ?q* = -)
⟨*proof*⟩

lemma *gluing-ext-add*:
assumes *gluing ‘‘* {((*x1*,*y1*),*l*)} ∈ *e-proj gluing ‘‘* {((*x2*,*y2*),*j*)} ∈ *e-proj delta′ x1 y1 x2 y2* ≠ *0*
shows *proj-addition* (*gluing ‘‘* {((*x1*,*y1*),*l*)}) (*gluing ‘‘* {((*x2*,*y2*),*j*)}) =
(*gluing ‘‘* {(*ext-add* (*x1*,*y1*) (*x2*,*y2*),*xor l j*)})
⟨*proof*⟩

lemma *gluing-ext-add-points*:
assumes *gluing ‘‘* {(*p1*,*l*)} ∈ *e-proj gluing ‘‘* {(*p2*,*j*)} ∈ *e-proj delta′* (*fst p1*) (*snd p1*) (*fst p2*) (*snd p2*) ≠ *0*
shows *proj-addition* (*gluing ‘‘* {(*p1*,*l*)}) (*gluing ‘‘* {(*p2*,*j*)}) =
(*gluing ‘‘* {(*ext-add p1 p2*,*xor l j*)})
⟨*proof*⟩

3.4.3 Basic properties

theorem *well-defined*:
assumes *p* ∈ *e-proj q* ∈ *e-proj*
shows *proj-addition p q* ∈ *e-proj*
⟨*proof*⟩

lemma *proj-add-class-inv*:
assumes *gluing ‘‘* {((*x*,*y*),*l*)} ∈ *e-proj*
shows *proj-addition* (*gluing ‘‘* {((*x*,*y*),*l*)}) (*gluing ‘‘* {(*i* (*x*,*y*),*l′*)}) = {((*1*, *0*), *xor l l′*)}
gluing ‘‘ {(*i* (*x*,*y*),*l′*)} ∈ *e-proj*
⟨*proof*⟩

lemma *proj-add-class-inv-point*:
assumes *gluing ‘‘* {(*p*,*l*)} ∈ *e-proj ne* = (*1*,*0*)
shows *proj-addition* (*gluing ‘‘* {(*p*,*l*)}) (*gluing ‘‘* {(*i p*,*l′*)}) = {(*ne*, *xor l l′*)}
gluing ‘‘ {(*i p*,*l′*)} ∈ *e-proj*
⟨*proof*⟩

lemma *proj-add-class-identity*:
assumes *x* ∈ *e-proj*
shows *proj-addition* {((*1*, *0*), *False*)} *x* = *x*
⟨*proof*⟩

corollary *proj-addition-comm*:
assumes *c1* ∈ *e-proj c2* ∈ *e-proj*
shows *proj-addition c1 c2* = *proj-addition c2 c1*
⟨*proof*⟩

4 Group law

4.1 Class invariance on group operations

definition *tf* **where**
tf g = *image* (λ *p*. (*g* (*fst p*), *snd p*))

lemma *tf-comp*:
tf g (*tf f s*) = *tf* (*g* ∘ *f*) *s*
⟨*proof*⟩

lemma *tf-id*:
tf id s = *s*
⟨*proof*⟩

lemma *tf-cong*:
f = *f'* ⟹ *s* = *s'* ⟹ *tf f s* = *tf f' s'*
⟨*proof*⟩

definition *tf'* **where**
tf' = *image* (λ *p*. (*fst p*, *Not* (*snd p*)))

lemma *tf-tf'-commute*:
tf r (*tf'* *p*) = *tf'* (*tf r p*)
⟨*proof*⟩

lemma *rho-preserv-e-proj*:
assumes *gluing* `` {((*x*, *y*), *l*)} ∈ *e-proj*
shows *tf* ϱ (*gluing* `` {((*x*, *y*), *l*)}) ∈ *e-proj*
⟨*proof*⟩

lemma *rho-preserv-e-proj-point*:
assumes *gluing* `` {*p*} ∈ *e-proj*
shows *tf* ϱ (*gluing* `` {*p*}) ∈ *e-proj*
⟨*proof*⟩

lemma *insert-rho-gluing*:
assumes *gluing* `` {((*x*, *y*), *l*)} ∈ *e-proj*
shows *tf* ϱ (*gluing* `` {((*x*, *y*), *l*)}) = *gluing* `` {(ϱ (*x*, *y*), *l*)}
⟨*proof*⟩

lemma *insert-rho-gluing-point*:
assumes *gluing* `` {(*p*, *l*)} ∈ *e-proj*
shows *tf* ϱ (*gluing* `` {(*p*, *l*)}) = *gluing* `` {(ϱ *p*, *l*)}
⟨*proof*⟩

lemma *rotation-preserv-e-proj*:
assumes *gluing* `` {((*x*, *y*), *l*)} ∈ *e-proj r* ∈ *rotations*
shows *tf r* (*gluing* `` {((*x*, *y*), *l*)}) ∈ *e-proj*
(**is** *tf ?r ?g* ∈ -)

⟨*proof*⟩

lemma *rotation-preserv-e-proj-point*:
assumes *gluing* ‘‘ {*p*} ∈ *e-proj r* ∈ *rotations*
shows *tf r* (*gluing* ‘‘ {*p*}) ∈ *e-proj*
⟨*proof*⟩

lemma *insert-rotation-gluing*:
assumes *gluing* ‘‘ {((*x*, *y*), *l*)} ∈ *e-proj r* ∈ *rotations*
shows *tf r* (*gluing* ‘‘ {((*x*, *y*), *l*)}) = *gluing* ‘‘ {(*r* (*x*, *y*), *l*)}
⟨*proof*⟩

lemma *insert-rotation-gluing-point*:
assumes *gluing* ‘‘ {(*p*, *l*)} ∈ *e-proj r* ∈ *rotations*
shows *tf r* (*gluing* ‘‘ {(*p*, *l*)}) = *gluing* ‘‘ {(*r p*, *l*)}
⟨*proof*⟩

lemma *tf-tau*:
assumes *gluing* ‘‘ {((*x*,*y*),*l*)} ∈ *e-proj*
shows *gluing* ‘‘ {((*x*,*y*), *Not l*)} = *tf′* (*gluing* ‘‘ {((*x*,*y*),*l*)})
⟨*proof*⟩

lemma *tf-preserv-e-proj*:
assumes *gluing* ‘‘ {((*x*,*y*),*l*)} ∈ *e-proj*
shows *tf′* (*gluing* ‘‘ {((*x*,*y*),*l*)}) ∈ *e-proj*
⟨*proof*⟩

lemma *tf-preserv-e-proj-point*:
assumes *gluing* ‘‘ {*p*} ∈ *e-proj*
shows *tf′* (*gluing* ‘‘ {*p*}) ∈ *e-proj*
⟨*proof*⟩

lemma *remove-rho*:
assumes *gluing* ‘‘ {((*x*,*y*),*l*)} ∈ *e-proj*
shows *gluing* ‘‘ {(ϱ (*x*,*y*),*l*)} = *tf* ϱ (*gluing* ‘‘ {((*x*,*y*),*l*)})
⟨*proof*⟩

lemma *remove-rotations*:
assumes *gluing* ‘‘ {((*x*,*y*),*l*)} ∈ *e-proj r* ∈ *rotations*
shows *gluing* ‘‘ {(*r* (*x*,*y*),*l*)} = *tf r* (*gluing* ‘‘ {((*x*,*y*),*l*)})
⟨*proof*⟩

lemma *remove-tau*:
assumes *gluing* ‘‘ {((*x*,*y*),*l*)} ∈ *e-proj gluing* ‘‘ {(τ (*x*,*y*),*l*)} ∈ *e-proj*
shows *gluing* ‘‘ {(τ (*x*,*y*),*l*)} = *tf′* (*gluing* ‘‘ {((*x*,*y*),*l*)})
(**is** *?gt* = *tf′ ?g*)
⟨*proof*⟩

lemma *remove-add-rho*:
assumes $p \in$ *e-proj* $q \in$ *e-proj*
shows *proj-addition* (*tf* ϱ p) q = *tf* ϱ (*proj-addition* p q)
⟨*proof*⟩

lemma *remove-add-rotation*:
assumes $p \in$ *e-proj* $q \in$ *e-proj* $r \in$ *rotations*
shows *proj-addition* (*tf* r p) q = *tf* r (*proj-addition* p q)
⟨*proof*⟩

lemma *remove-add-tau*:
assumes $p \in$ *e-proj* $q \in$ *e-proj*
shows *proj-addition* (*tf'* p) q = *tf'* (*proj-addition* p q)
⟨*proof*⟩

lemma *remove-add-tau'*:
assumes $p \in$ *e-proj* $q \in$ *e-proj*
shows *proj-addition* p (*tf'* q) = *tf'* (*proj-addition* p q)
⟨*proof*⟩

lemma *tf'-idemp*:
assumes $s \in$ *e-proj*
shows *tf'* (*tf'* s) = s
⟨*proof*⟩

definition *tf''* **where**
tf'' g s = *tf'* (*tf* g s)

lemma *remove-sym*:
assumes *gluing* ‘‘ {((*x*, *y*), *l*)} ∈ *e-proj gluing* ‘‘ {(*g* (*x*, *y*), *l*)} ∈ *e-proj* $g \in$ *symmetries*
shows *gluing* ‘‘ {(*g* (*x*, *y*), *l*)} = *tf''* (*τ* ∘ *g*) (*gluing* ‘‘ {((*x*, *y*), *l*)})
⟨*proof*⟩

lemma *remove-add-sym*:
assumes $p \in$ *e-proj* $q \in$ *e-proj* $g \in$ *rotations*
shows *proj-addition* (*tf''* g p) q = *tf''* g (*proj-addition* p q)
⟨*proof*⟩
lemma *tf''-preserv-e-proj*:
assumes *gluing* ‘‘ {((*x*,*y*),*l*)} ∈ *e-proj* $r \in$ *rotations*
shows *tf''* r (*gluing* ‘‘ {((*x*,*y*),*l*)}) ∈ *e-proj*
⟨*proof*⟩

lemma *tf'-injective*:
assumes *c1* ∈ *e-proj c2* ∈ *e-proj*
assumes *tf'* *c1* = *tf'* *c2*
shows *c1* = *c2*
⟨*proof*⟩

4.2 Associativities

lemma *add-add-add-add-assoc*:
assumes *(x1,y1) ∈ e'-aff (x2,y2) ∈ e'-aff (x3,y3) ∈ e'-aff*
assumes *delta x1 y1 x2 y2 ≠ 0 delta x2 y2 x3 y3 ≠ 0*
delta (fst (add (x1,y1) (x2,y2))) (snd (add (x1,y1) (x2,y2))) x3 y3 ≠ 0
delta x1 y1 (fst (add (x2,y2) (x3,y3))) (snd (add (x2,y2) (x3,y3))) ≠ 0
shows *add (add (x1,y1) (x2,y2)) (x3,y3) = add (x1,y1) (add (x2,y2) (x3,y3))*
⟨*proof*⟩

lemma *fstI*: *x = (y, z) ⟹ y = fst x*
⟨*proof*⟩

lemma *sndI*: *x = (y, z) ⟹ z = snd x*
⟨*proof*⟩

⟨*ML*⟩

lemma *add-ext-add-ext-assoc-points*:
assumes *(x1,y1) ∈ e'-aff (x2,y2) ∈ e'-aff (x3,y3) ∈ e'-aff*
assumes *delta' x1 y1 x2 y2 ≠ 0 delta' x2 y2 x3 y3 ≠ 0*
delta (fst (ext-add (x1,y1) (x2,y2))) (snd (ext-add (x1,y1) (x2,y2))) x3 y3 ≠ 0
delta x1 y1 (fst (ext-add (x2,y2) (x3,y3))) (snd (ext-add (x2,y2) (x3,y3))) ≠ 0
shows *add (ext-add (x1,y1) (x2,y2)) (x3,y3) = add (x1,y1) (ext-add (x2,y2) (x3,y3))*
⟨*proof*⟩

⟨*ML*⟩

lemma *add-ext-add-add-assoc-points*:
assumes *(x1,y1) ∈ e'-aff (x2,y2) ∈ e'-aff (x3,y3) ∈ e'-aff*
assumes *delta' x1 y1 x2 y2 ≠ 0 delta x2 y2 x3 y3 ≠ 0*
delta (fst (ext-add (x1,y1) (x2,y2))) (snd (ext-add (x1,y1) (x2,y2))) x3 y3 ≠ 0
delta x1 y1 (fst (add (x2,y2) (x3,y3))) (snd (add (x2,y2) (x3,y3))) ≠ 0
shows *add (ext-add (x1,y1) (x2,y2)) (x3,y3) = add (x1,y1) (add (x2,y2) (x3,y3))*
⟨*proof*⟩

⟨*ML*⟩

lemma *add-add-ext-add-assoc-points*:
assumes *(x1,y1) ∈ e'-aff (x2,y2) ∈ e'-aff (x3,y3) ∈ e'-aff*
assumes *delta x1 y1 x2 y2 ≠ 0 delta x2 y2 x3 y3 ≠ 0*
delta (fst (add (x1,y1) (x2,y2))) (snd (add (x1,y1) (x2,y2))) x3 y3 ≠ 0

delta′ x1 y1 (fst (add (x2,y2) (x3,y3))) (snd (add (x2,y2) (x3,y3))) ≠ 0
shows *add (add (x1,y1) (x2,y2)) (x3,y3) = ext-add (x1,y1) (add (x2,y2) (x3,y3))*
⟨*proof*⟩

⟨*ML*⟩

lemma *add-add-add-ext-assoc-points*:
assumes *(x1,y1) ∈ e′-aff (x2,y2) ∈ e′-aff (x3,y3) ∈ e′-aff*
assumes *delta x1 y1 x2 y2 ≠ 0 delta′ x2 y2 x3 y3 ≠ 0*
delta (fst (add (x1,y1) (x2,y2))) (snd (add (x1,y1) (x2,y2))) x3 y3 ≠ 0
delta x1 y1 (fst (ext-add (x2,y2) (x3,y3))) (snd (ext-add (x2,y2) (x3,y3))) ≠ 0
shows *add (add (x1,y1) (x2,y2)) (x3,y3) = add (x1,y1) (ext-add (x2,y2) (x3,y3))*
⟨*proof*⟩

⟨*ML*⟩

lemma *ext-add-add-ext-assoc-points*:
assumes *(x1,y1) ∈ e′-aff (x2,y2) ∈ e′-aff (x3,y3) ∈ e′-aff*
assumes *delta x1 y1 x2 y2 ≠ 0 delta′ x2 y2 x3 y3 ≠ 0*
delta′ (fst (add (x1,y1) (x2,y2))) (snd (add (x1,y1) (x2,y2))) x3 y3 ≠ 0
delta x1 y1 (fst (ext-add (x2,y2) (x3,y3))) (snd (ext-add (x2,y2) (x3,y3))) ≠ 0
shows *ext-add (add (x1,y1) (x2,y2)) (x3,y3) = add (x1,y1) (ext-add (x2,y2) (x3,y3))*
⟨*proof*⟩

⟨*ML*⟩

lemma *ext-add-add-add-assoc-points*:
assumes *(x1,y1) ∈ e′-aff (x2,y2) ∈ e′-aff (x3,y3) ∈ e′-aff*
assumes *delta x1 y1 x2 y2 ≠ 0 delta x2 y2 x3 y3 ≠ 0*
delta′ (fst (add (x1,y1) (x2,y2))) (snd (add (x1,y1) (x2,y2))) x3 y3 ≠ 0
delta x1 y1 (fst (add (x2,y2) (x3,y3))) (snd (add (x2,y2) (x3,y3))) ≠ 0
shows *ext-add (add (x1,y1) (x2,y2)) (x3,y3) = add (x1,y1) (add (x2,y2) (x3,y3))*
⟨*proof*⟩

⟨*ML*⟩

lemma *ext-ext-add-add-assoc-points*:
assumes *(x1,y1) ∈ e′-aff (x2,y2) ∈ e′-aff (x3,y3) ∈ e′-aff*
assumes *delta′ x1 y1 x2 y2 ≠ 0 delta x2 y2 x3 y3 ≠ 0*
delta′ (fst (ext-add (x1,y1) (x2,y2))) (snd (ext-add (x1,y1) (x2,y2))) x3 y3 ≠ 0
delta x1 y1 (fst (add (x2,y2) (x3,y3))) (snd (add (x2,y2) (x3,y3))) ≠ 0
shows *ext-add (ext-add (x1,y1) (x2,y2)) (x3,y3) = add (x1,y1) (add (x2,y2)*

(*x3*,*y3*))
⟨*proof*⟩

⟨*ML*⟩

lemma *ext-ext-add-ext-assoc-points*:
assumes (*x1*,*y1*) ∈ *e′-aff* (*x2*,*y2*) ∈ *e′-aff* (*x3*,*y3*) ∈ *e′-aff*
assumes *delta′ x1 y1 x2 y2* ≠ *0 delta′ x2 y2 x3 y3* ≠ *0*
delta′ (*fst* (*ext-add* (*x1*,*y1*) (*x2*,*y2*))) (*snd* (*ext-add* (*x1*,*y1*) (*x2*,*y2*))) *x3 y3* ≠ *0*
delta x1 y1 (*fst* (*ext-add* (*x2*,*y2*) (*x3*,*y3*))) (*snd* (*ext-add* (*x2*,*y2*) (*x3*,*y3*))) ≠ *0*
shows *ext-add* (*ext-add* (*x1*,*y1*) (*x2*,*y2*)) (*x3*,*y3*) = *add* (*x1*,*y1*) (*ext-add* (*x2*,*y2*) (*x3*,*y3*))
⟨*proof*⟩

⟨*ML*⟩

4.3 Lemmas for associativity

lemma *cancellation-assoc*:
assumes *gluing* ‘‘ {((*x1*,*y1*), *False*)} ∈ *e-proj*
gluing ‘‘ {((*x2*,*y2*), *False*)} ∈ *e-proj*
gluing ‘‘ {(*i* (*x2*,*y2*), *False*)} ∈ *e-proj*
shows *proj-addition* (*proj-addition* (*gluing* ‘‘ {((*x1*,*y1*), *False*)}) (*gluing* ‘‘ {((*x2*,*y2*), *False*)})) (*gluing* ‘‘ {(*i* (*x2*,*y2*), *False*)}) =
gluing ‘‘ {((*x1*,*y1*), *False*)}
(**is** *proj-addition* (*proj-addition ?g1 ?g2*) *?g3* = *?g1*)
⟨*proof*⟩

lemma *e′-aff-0-invariance*:
((*x*,*y*),(*x′*,*y′*)) ∈ *e′-aff-0* ⟹ ((*x′*,*y′*),(*x*,*y*)) ∈ *e′-aff-0*
⟨*proof*⟩

lemma *e′-aff-1-invariance*:
((*x*,*y*),(*x′*,*y′*)) ∈ *e′-aff-1* ⟹ ((*x′*,*y′*),(*x*,*y*)) ∈ *e′-aff-1*
⟨*proof*⟩

lemma *assoc-1*:
assumes *gluing* ‘‘ {((*x1*, *y1*), *False*)} ∈ *e-proj*
gluing ‘‘ {((*x2*, *y2*), *False*)} ∈ *e-proj*
gluing ‘‘ {((*x3*, *y3*), *False*)} ∈ *e-proj*
assumes *a*: *g* ∈ *symmetries* (*x2*, *y2*) = (*g* ∘ *i*) (*x1*, *y1*)
shows
proj-addition (*gluing* ‘‘ {((*x1*, *y1*), *False*)}) (*gluing* ‘‘ {((*x2*, *y2*), *False*)}) =
tf″ (*τ* ∘ *g*) {((*1*,*0*),*False*)} (**is** *proj-addition ?g1 ?g2* = -)
proj-addition (*proj-addition* (*gluing* ‘‘ {((*x1*, *y1*), *False*)}) (*gluing* ‘‘ {((*x2*, *y2*), *False*)})) (*gluing* ‘‘ {((*x3*, *y3*), *False*)}) =

tf'' (*τ* ∘ *g*) (*gluing* ‘‘ {((*x3*, *y3*), *False*)})
proj-addition (*gluing* ‘‘ {((*x1*, *y1*), *False*)}) (*proj-addition* (*gluing* ‘‘ {((*x2*, *y2*), *False*)}) (*gluing* ‘‘ {((*x3*, *y3*), *False*)})) =
tf'' (*τ* ∘ *g*) (*gluing* ‘‘ {((*x3*, *y3*), *False*)}) (**is** *proj-addition ?g1* (*proj-addition ?g2 ?g3*) = -)
⟨*proof*⟩

lemma *assoc-11*:
assumes *gluing* ‘‘ {((*x1*, *y1*), *False*)} ∈ *e-proj*
gluing ‘‘ {((*x2*, *y2*), *False*)} ∈ *e-proj*
gluing ‘‘ {((*x3*, *y3*), *False*)} ∈ *e-proj*
assumes *a*: *g* ∈ *symmetries* (*x3*, *y3*) = (*g* ∘ *i*) (*x2*, *y2*)
shows
proj-addition (*proj-addition* (*gluing* ‘‘ {((*x1*, *y1*), *False*)}) (*gluing* ‘‘ {((*x2*, *y2*), *False*)})) (*gluing* ‘‘ {((*x3*, *y3*), *False*)}) =
proj-addition (*gluing* ‘‘ {((*x1*, *y1*), *False*)}) (*proj-addition* (*gluing* ‘‘ {((*x2*, *y2*), *False*)}) (*gluing* ‘‘ {((*x3*, *y3*), *False*)}))
(**is** *proj-addition* (*proj-addition ?g1 ?g2*) *?g3* = -)
⟨*proof*⟩

lemma *assoc-111-add*:
assumes *gluing* ‘‘ {((*x1*, *y1*), *False*)} ∈ *e-proj*
gluing ‘‘ {((*x2*, *y2*), *False*)} ∈ *e-proj*
gluing ‘‘ {((*x3*, *y3*), *False*)} ∈ *e-proj*
assumes *22*: *g*∈*symmetries* (*x1*, *y1*) = (*g* ∘ *i*) (*add* (*x2*,*y2*) (*x3*,*y3*)) ((*x2*, *y2*), *x3*, *y3*) ∈ *e'-aff-0*
shows
proj-addition (*proj-addition* (*gluing* ‘‘ {((*x1*, *y1*), *False*)}) (*gluing* ‘‘ {((*x2*, *y2*), *False*)})) (*gluing* ‘‘ {((*x3*, *y3*), *False*)}) =
proj-addition (*gluing* ‘‘ {((*x1*, *y1*), *False*)}) (*proj-addition* (*gluing* ‘‘ {((*x2*, *y2*), *False*)}) (*gluing* ‘‘ {((*x3*, *y3*), *False*)}))
(**is** *proj-addition* (*proj-addition ?g1 ?g2*) *?g3* = -)
⟨*proof*⟩

lemma *assoc-111-ext-add*:
assumes *gluing* ‘‘ {((*x1*, *y1*), *False*)} ∈ *e-proj*
gluing ‘‘ {((*x2*, *y2*), *False*)} ∈ *e-proj*
gluing ‘‘ {((*x3*, *y3*), *False*)} ∈ *e-proj*
assumes *22*: *g*∈*symmetries* (*x1*, *y1*) = (*g* ∘ *i*) (*ext-add* (*x2*,*y2*) (*x3*,*y3*)) ((*x2*, *y2*), *x3*, *y3*) ∈ *e'-aff-1*
shows
proj-addition (*proj-addition* (*gluing* ‘‘ {((*x1*, *y1*), *False*)}) (*gluing* ‘‘ {((*x2*, *y2*), *False*)})) (*gluing* ‘‘ {((*x3*, *y3*), *False*)}) =
proj-addition (*gluing* ‘‘ {((*x1*, *y1*), *False*)}) (*proj-addition* (*gluing* ‘‘ {((*x2*, *y2*), *False*)}) (*gluing* ‘‘ {((*x3*, *y3*), *False*)}))
(**is** *proj-addition* (*proj-addition ?g1 ?g2*) *?g3* = -)
⟨*proof*⟩

lemma *assoc-with-zeros*:

assumes *gluing ‘‘ {((x1, y1), False)} ∈ e-proj*
gluing ‘‘ {((x2, y2), False)} ∈ e-proj
gluing ‘‘ {((x3, y3), False)} ∈ e-proj
shows *proj-addition (proj-addition (gluing ‘‘ {((x1, y1), False)}) (gluing ‘‘ {((x2, y2), False)})) (gluing ‘‘ {((x3, y3), False)}) = proj-addition (gluing ‘‘ {((x1, y1), False)}) (proj-addition (gluing ‘‘ {((x2, y2), False)}) (gluing ‘‘ {((x3, y3), False)}))*
(**is** *proj-addition (proj-addition ?g1 ?g2) ?g3 = proj-addition ?g1 (proj-addition ?g2 ?g3)*)
⟨*proof*⟩

lemma *general-assoc*:
assumes *gluing ‘‘ {((x1, y1), l)} ∈ e-proj gluing ‘‘ {((x2, y2), m)} ∈ e-proj gluing ‘‘ {((x3, y3), n)} ∈ e-proj*
shows *proj-addition (proj-addition (gluing ‘‘ {((x1, y1), l)}) (gluing ‘‘ {((x2, y2), m)})) (gluing ‘‘ {((x3, y3), n)}) = proj-addition (gluing ‘‘ {((x1, y1), l)}) (proj-addition (gluing ‘‘ {((x2, y2), m)}) (gluing ‘‘ {((x3, y3), n)}))*
⟨*proof*⟩

lemma *proj-assoc*:
assumes *x ∈ e-proj y ∈ e-proj z ∈ e-proj*
shows *proj-addition (proj-addition x y) z = proj-addition x (proj-addition y z)*
⟨*proof*⟩

4.4 Group law

theorem *projective-group-law*:
shows *comm-group (|carrier = e-proj, mult = proj-addition, one = {((1,0),False)}|)*

⟨*proof*⟩

end

end

References

[1] T. Hales and R. Raya. Formal proof of the group law for edwards elliptic curves. In N. Peltier and V. Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 254–269, Cham, 2020. Springer International Publishing.