The Group Law for Elliptic Curves

Stefan Berghofer

May 26, 2024

Abstract

We prove the group law for elliptic curves in Weierstrass form over fields of characteristic greater than 2. In addition to affine coordinates, we also formalize projective coordinates, which allow for more efficient computations. By specializing the abstract formalization to prime fields, we can apply the curve operations to parameters used in standard security protocols.

Contents

1 Introduction **1**

2 Formalization using Axiomatic Type Classes **2**
2.1 Affine Coordinates 2
2.2 Projective Coordinates 38

3 Formalization using Locales **45**
3.1 Affine Coordinates 45
3.2 Projective Coordinates 85

4 Validating the Specification **93**
4.1 Specialized Definitions for Prime Fields 93
4.2 The NIST Curve P-521 98

1 Introduction

Elliptic curves play an important role in cryptography, since they allow to achieve a security level that is comparable to that of RSA, while requiring a smaller key size and less computation time. The primitive operation on elliptic curves is *point addition*. To ensure the proper functioning of cryptographic algorithms based on elliptic curves, such as Diffie-Hellman key exchange (ECDH) or digital signatures (ECDSA), it is important that the points on the curve form a group with respect to point addition.

Our formalization of elliptic curves is based on earlier work by Laurent Théry in Coq [4]. Like its Coq counterpart, the Isabelle formalization uses decision procedures for rings and fields based on reflection, which are executed using Isabelle's code generator for efficiency reasons. The decision procedure for rings is due to Grégoire and Mahboubi [3] and was ported from Coq to Isabelle by Bernhard Haeupler.

The formalization exists in two flavours: one based on axiomatic type classes, and another one based on locales. While the axiomatic type class version is more concise, the locale version is more suitable for working with concrete rings or fields like prime fields.

2 Formalization using Axiomatic Type Classes

theory *Elliptic-Axclass*
imports *HOL−Decision-Procs.Reflective-Field*
begin

2.1 Affine Coordinates

datatype *'a point = Infinity | Point 'a 'a*

class *ell-field = field +*
  assumes *two-not-zero*: *2 ≠ 0*
begin

definition *nonsingular* :: *'a ⇒ 'a ⇒ bool* **where**
  *nonsingular a b = (4 * a ^ 3 + 27 * b ^ 2 ≠ 0)*

definition *on-curve* :: *'a ⇒ 'a ⇒ 'a point ⇒ bool* **where**
  on-curve a b p = (case p of
      Infinity ⇒ True
    *| Point x y ⇒ y ^ 2 = x ^ 3 + a * x + b)*

definition *add* :: *'a ⇒ 'a point ⇒ 'a point ⇒ 'a point* **where**
  add a p₁ p₂ = (case p₁ of
      Infinity ⇒ p₂
    | Point x₁ y₁ ⇒ (case p₂ of
        Infinity ⇒ p₁
      | Point x₂ y₂ ⇒
          if x₁ = x₂ then
            if y₁ = − y₂ then Infinity
            else
              let
                *l = (3 * x₁ ^ 2 + a) / (2 * y₁);*
                *x₃ = l ^ 2 − 2 * x₁*
              in
                *Point x₃ (− y₁ − l * (x₃ − x₁))*
          else

```
        let
          l = (y₂ − y₁) / (x₂ − x₁);
          x₃ = l ^ 2 − x₁ − x₂
        in
          Point x₃ (− y₁ − l * (x₃ − x₁))))

definition opp :: 'a point ⇒ 'a point where
  opp p = (case p of
      Infinity ⇒ Infinity
    | Point x y ⇒ Point x (− y))

end

lemma on-curve-infinity [simp]: on-curve a b Infinity
  by (simp add: on-curve-def)

lemma opp-Infinity [simp]: opp Infinity = Infinity
  by (simp add: opp-def)

lemma opp-Point: opp (Point x y) = Point x (− y)
  by (simp add: opp-def)

lemma opp-opp: opp (opp p) = p
  by (simp add: opp-def split: point.split)

lemma opp-closed:
  on-curve a b p ⟹ on-curve a b (opp p)
  by (auto simp add: on-curve-def opp-def power2-eq-square
    split: point.split)

lemma curve-elt-opp:
  assumes p₁ = Point x₁ y₁
  and p₂ = Point x₂ y₂
  and on-curve a b p₁
  and on-curve a b p₂
  and x₁ = x₂
  shows p₁ = p₂ ∨ p₁ = opp p₂
proof −
  from ‹p₁ = Point x₁ y₁› ‹on-curve a b p₁›
  have y₁ ^ 2 = x₁ ^ 3 + a * x₁ + b
    by (simp-all add: on-curve-def)
  moreover from ‹p₂ = Point x₂ y₂› ‹on-curve a b p₂› ‹x₁ = x₂›
  have x₁ ^ 3 + a * x₁ + b = y₂ ^ 2
    by (simp-all add: on-curve-def)
  ultimately have y₁ = y₂ ∨ y₁ = − y₂
    by (simp add: square-eq-iff power2-eq-square)
  with ‹p₁ = Point x₁ y₁› ‹p₂ = Point x₂ y₂› ‹x₁ = x₂› show ?thesis
    by (auto simp add: opp-def)
qed
```

```
lemma add-closed:
  assumes on-curve a b p₁ and on-curve a b p₂
  shows on-curve a b (add a p₁ p₂)
proof (cases p₁)
  case (Point x₁ y₁)
  note Point' = this
  show ?thesis
  proof (cases p₂)
    case (Point x₂ y₂)
    show ?thesis
    proof (cases x₁ = x₂)
      case True
      note True' = this
      show ?thesis
      proof (cases y₁ = - y₂)
        case True
        with True' Point Point'
        show ?thesis
          by (simp add: on-curve-def add-def)
      next
        case False
         note on-curve1 = ‹on-curve a b p₁› [simplified Point' on-curve-def True',
simplified]
        from False True' Point Point' assms
        have y₁ ≠ 0 by (auto simp add: on-curve-def)
        with False True' Point Point' assms
        show ?thesis
          apply (simp add: on-curve-def add-def Let-def)
          apply (field on-curve1)
          apply (simp add: two-not-zero)
          done
      qed
    next
      case False
     note on-curve1 = ‹on-curve a b p₁› [simplified Point' on-curve-def, simplified]
     note on-curve2 = ‹on-curve a b p₂› [simplified Point on-curve-def, simplified]
      from assms show ?thesis
        apply (simp add: on-curve-def add-def Let-def False Point Point')
        apply (field on-curve1 on-curve2)
        apply (simp add: False [symmetric])
        done
    qed
  next
    case Infinity
    with Point ‹on-curve a b p₁› show ?thesis
      by (simp add: add-def)
  qed
next
```

```
  case Infinity
  with ‹on-curve a b p₂› show ?thesis
    by (simp add: add-def)
qed

lemma add-case [consumes 2, case-names InfL InfR Opp Tan Gen]:
  assumes p: on-curve a b p
  and q: on-curve a b q
  and R1: ⋀p. P Infinity p p
  and R2: ⋀p. P p Infinity p
  and R3: ⋀p. on-curve a b p ⟹ P p (opp p) Infinity
  and R4: ⋀p₁ x₁ y₁ p₂ x₂ y₂ l.
    p₁ = Point x₁ y₁ ⟹ p₂ = Point x₂ y₂ ⟹
    p₂ = add a p₁ p₁ ⟹ y₁ ≠ 0 ⟹
    l = (3 * x₁ ^ 2 + a) / (2 * y₁) ⟹
    x₂ = l ^ 2 - 2 * x₁ ⟹
    y₂ = - y₁ - l * (x₂ - x₁) ⟹
    P p₁ p₁ p₂
  and R5: ⋀p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l.
    p₁ = Point x₁ y₁ ⟹ p₂ = Point x₂ y₂ ⟹ p₃ = Point x₃ y₃ ⟹
    p₃ = add a p₁ p₂ ⟹ x₁ ≠ x₂ ⟹
    l = (y₂ - y₁) / (x₂ - x₁) ⟹
    x₃ = l ^ 2 - x₁ - x₂ ⟹
    y₃ = - y₁ - l * (x₃ - x₁) ⟹
    P p₁ p₂ p₃
  shows P p q (add a p q)
proof (cases p)
  case Infinity
  then show ?thesis
    by (simp add: add-def R1)
next
  case (Point x₁ y₁)
  note Point' = this
  show ?thesis
  proof (cases q)
    case Infinity
    with Point show ?thesis
      by (simp add: add-def R2)
  next
    case (Point x₂ y₂)
    show ?thesis
    proof (cases x₁ = x₂)
      case True
      note True' = this
      show ?thesis
      proof (cases y₁ = - y₂)
        case True
        with p Point Point' True' R3 [of p] show ?thesis
          by (simp add: add-def opp-def)
```

```
    next
      case False
      from True' Point Point' p q have (y₁ − y₂) * (y₁ + y₂) = 0
        by (simp add: on-curve-def ring-distribs power2-eq-square)
      with False have y₁ = y₂
        by (simp add: eq-neg-iff-add-eq-0)
      with False True' Point Point' show ?thesis
        apply simp
        apply (rule R4)
        apply (auto simp add: add-def Let-def)
        done
    qed
  next
    case False
    with Point Point' show ?thesis
      apply −
      apply (rule R5)
      apply (auto simp add: add-def Let-def)
      done
  qed
 qed
qed

lemma eq-opp-is-zero: ((x::'a::ell-field) = − x) = (x = 0)
proof
 assume x = − x
 have 2 * x = x + x by simp
 also from ‹x = − x› 
 have ... = − x + x by simp
 also have ... = 0 by simp
 finally have 2 * x = 0 .
 with two-not-zero [where 'a='a] show x = 0
   by simp
qed simp

lemma add-casew [consumes 2, case-names InfL InfR Opp Gen]:
 assumes p: on-curve a b p
 and q: on-curve a b q
 and R1: ⋀p. P Infinity p p
 and R2: ⋀p. P p Infinity p
 and R3: ⋀p. on-curve a b p ⟹ P p (opp p) Infinity
 and R4: ⋀p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l.
   p₁ = Point x₁ y₁ ⟹ p₂ = Point x₂ y₂ ⟹ p₃ = Point x₃ y₃ ⟹
   p₃ = add a p₁ p₂ ⟹ p₁ ≠ opp p₂ ⟹
   x₁ = x₂ ∧ y₁ = y₂ ∧ l = (3 * x₁ ^ 2 + a) / (2 * y₁) ∨
   x₁ ≠ x₂ ∧ l = (y₂ − y₁) / (x₂ − x₁) ⟹
   x₃ = l ^ 2 − x₁ − x₂ ⟹
   y₃ = − y₁ − l * (x₃ − x₁) ⟹
   P p₁ p₂ p₃
```

shows $P\ p\ q\ (add\ a\ p\ q)$
using $p\ q$
apply (*rule add-case*)
apply (*rule R1*)
apply (*rule R2*)
apply (*rule R3*)
apply *assumption*
apply (*rule R4*)
apply *assumption*+
apply (*simp add*: *opp-def eq-opp-is-zero*)
apply *simp*
apply *simp*
apply *simp*
apply (*rule R4*)
apply *assumption*+
apply (*simp add*: *opp-def*)
apply *simp*
apply *assumption*+
done

definition
is-tangent p q = ($p \neq$ *Infinity* $\wedge$ $p = q \wedge p \neq$ *opp q*)

definition
is-generic p q =
($p \neq$ *Infinity* $\wedge$ $q \neq$ *Infinity* $\wedge$
$p \neq q \wedge p \neq$ *opp q*)

lemma *diff-neq0*:
$(a::'a::ring) \neq b \Longrightarrow a - b \neq 0$
$a \neq b \Longrightarrow b - a \neq 0$
by *simp-all*

lemma *minus2-not0*: $(-2::'a::ell\text{-}field) \neq 0$
using *two-not-zero* [**where** $'a='a$]
by *simp*

lemmas [*simp*] = *minus2-not0* [*simplified*]

declare *two-not-zero* [*simplified*, *simp add*]

lemma *spec1-assoc*:
assumes p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
and p_3: *on-curve a b* p_3
and *is-generic* $p_1\ p_2$
and *is-generic* $p_2\ p_3$
and *is-generic* $(add\ a\ p_1\ p_2)\ p_3$
and *is-generic* $p_1\ (add\ a\ p_2\ p_3)$

```
  shows add a p₁ (add a p₂ p₃) = add a (add a p₁ p₂) p₃
  using p₁ p₂ assms
proof (induct rule: add-case)
  case InfL
  show ?case by (simp add: add-def)
next
  case InfR
  show ?case by (simp add: add-def)
next
  case Opp
  then show ?case by (simp add: is-generic-def)
next
  case Tan
  then show ?case by (simp add: is-generic-def)
next
  case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₄ x₄ y₄ l)
  with ‹on-curve a b p₂› ‹on-curve a b p₃›
  show ?case
  proof (induct rule: add-case)
    case InfL
    then show ?case by (simp add: is-generic-def)
  next
    case InfR
    then show ?case by (simp add: is-generic-def)
  next
    case Opp
    then show ?case by (simp add: is-generic-def)
  next
    case Tan
    then show ?case by (simp add: is-generic-def)
  next
    case (Gen p₂ x₂' y₂' p₃ x₃ y₃ p₅ x₅ y₅ l₁)
    from ‹on-curve a b p₂› ‹on-curve a b p₃› ‹p₅ = add a p₂ p₃›
    have on-curve a b p₅ by (simp add: add-closed)
    with ‹on-curve a b p₁› show ?case using Gen [simplified ‹p₂ = Point x₂' y₂'›]
    proof (induct rule: add-case)
      case InfL
      then show ?case by (simp add: is-generic-def)
    next
      case InfR
      then show ?case by (simp add: is-generic-def)
    next
      case (Opp p)
      from ‹is-generic p (opp p)›
      show ?case by (simp add: is-generic-def opp-opp)
    next
      case Tan
      then show ?case by (simp add: is-generic-def)
    next
```

case (*Gen* p_1 $x_1{'}$ $y_1{'}$ $p_5{'}$ $x_5{'}$ $y_5{'}$ p_6 x_6 y_6 l_2)
from ‹*on-curve a b* p_1› ‹*on-curve a b* (*Point* $x_2{'}$ $y_2{'}$)›
‹p_4 = *add a* p_1 (*Point* $x_2{'}$ $y_2{'}$)›
have *on-curve a b* p_4 **by** (*simp add*: *add-closed*)
then show *?case* **using** ‹*on-curve a b* p_3› *Gen*
proof (*induct rule*: *add-case*)
case *InfL*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case *InfR*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case (*Opp p*)
from ‹*is-generic p* (*opp p*)›
show *?case* **by** (*simp add*: *is-generic-def opp-opp*)
next
case *Tan*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case (*Gen* $p_4{'}$ $x_4{'}$ $y_4{'}$ $p_3{'}$ $x_3{'}$ $y_3{'}$ p_7 x_7 y_7 l_3)
from ‹$p_4{'}$ = *Point* $x_4{'}$ $y_4{'}$› ‹$p_4{'}$ = *Point* x_4 y_4›
have p_4: $x_4{'}$ = x_4 $y_4{'}$ = y_4 **by** *simp-all*
from ‹$p_3{'}$ = *Point* $x_3{'}$ $y_3{'}$› ‹$p_3{'}$ = *Point* x_3 y_3›
have p_3: $x_3{'}$ = x_3 $y_3{'}$ = y_3 **by** *simp-all*
from ‹p_1 = *Point* $x_1{'}$ $y_1{'}$› ‹p_1 = *Point* x_1 y_1›
have p_1: $x_1{'}$ = x_1 $y_1{'}$ = y_1 **by** *simp-all*
from ‹$p_5{'}$ = *Point* $x_5{'}$ $y_5{'}$› ‹$p_5{'}$ = *Point* x_5 y_5›
have p_5: $x_5{'}$ = x_5 $y_5{'}$ = y_5 **by** *simp-all*
from ‹*Point* $x_2{'}$ $y_2{'}$ = *Point* x_2 y_2›
have p_2: $x_2{'}$ = x_2 $y_2{'}$ = y_2 **by** *simp-all*
note ps = p_1 p_2 p_3 p_4 p_5
note ps' =
‹*on-curve a b* p_1› [*simplified* ‹p_1 = *Point* x_1 y_1› *on-curve-def*, *simplified*]
‹*on-curve a b* p_2› [*simplified* ‹p_2 = *Point* x_2 y_2› *on-curve-def*, *simplified*]
‹*on-curve a b* p_3› [*simplified* ‹p_3 = *Point* x_3 y_3› *on-curve-def*, *simplified*]
show *?case*
apply (*simp add*: ‹p_6 = *Point* x_6 y_6› ‹p_7 = *Point* x_7 y_7›)
apply (*simp only*: *ps*
‹x_6 = ${l_2}^2$ − $x_1{'}$ − $x_5{'}$› ‹x_7 = ${l_3}^2$ − $x_4{'}$ − $x_3{'}$›
‹y_6 = − $y_1{'}$ − l_2 ∗ (x_6 − $x_1{'}$)› ‹y_7 = − $y_4{'}$ − l_3 ∗ (x_7 − $x_4{'}$)›
‹l_2 = ($y_5{'}$ − $y_1{'}$) / ($x_5{'}$ − $x_1{'}$)› ‹l_3 = ($y_3{'}$ − $y_4{'}$) / ($x_3{'}$ − $x_4{'}$)›
‹l_1 = (y_3 − $y_2{'}$) / (x_3 − $x_2{'}$)› ‹l = (y_2 − y_1) / (x_2 − x_1)›
‹x_5 = ${l_1}^2$ − $x_2{'}$ − x_3› ‹y_5 = − $y_2{'}$ − l_1 ∗ (x_5 − $x_2{'}$)›
‹x_4 = l^2 − x_1 − x_2› ‹y_4 = − y_1 − l ∗ (x_4 − x_1)›)
apply (*rule conjI*)
apply (*field ps'*)
apply (*rule conjI*)
apply (*simp add*: ‹$x_2{'}$ ≠ x_3› [*simplified* ‹$x_2{'}$ = x_2›, *symmetric*])
apply (*rule conjI*)

apply (*rule notI*)
apply (*ring* (*prems*) *ps′*(*1−2*))
apply (*cut-tac* ‹$x_1{}' \neq x_5{}'$› [*simplified* ‹$x_5{}' = x_5$› ‹$x_1{}' = x_1$› ‹$x_5 = {l_1}^2 - x_2{}' - x_3$›
‹$l_1 = (y_3 - y_2{}') \,/\, (x_3 - x_2{}')$› ‹$y_2{}' = y_2$› ‹$x_2{}' = x_2$›])
apply (*erule notE*)
apply (*rule sym*)
apply (*field ps′*(*1−2*))
apply (*simp add*: ‹$x_2{}' \neq x_3$› [*simplified* ‹$x_2{}' = x_2$›, *symmetric*])
apply (*rule conjI*)
apply (*simp add*: ‹$x_1 \neq x_2$› [*symmetric*])
apply (*rule notI*)
apply (*ring* (*prems*) *ps′*(*1−2*))
apply (*cut-tac* ‹$x_4{}' \neq x_3{}'$› [*simplified* ‹$x_4{}' = x_4$› ‹$x_3{}' = x_3$› ‹$x_4 = l^2 - x_1 - x_2$›
‹$l = (y_2 - y_1) \,/\, (x_2 - x_1)$›])
apply (*erule notE*)
apply (*rule sym*)
apply (*field ps′*(*1−2*))
apply (*simp add*: ‹$x_1 \neq x_2$› [*symmetric*])
apply (*field ps′*)
apply (*rule conjI*)
apply (*rule notI*)
apply (*ring* (*prems*) *ps′*(*1−2*))
apply (*cut-tac* ‹$x_1{}' \neq x_5{}'$› [*simplified* ‹$x_5{}' = x_5$› ‹$x_1{}' = x_1$› ‹$x_5 = {l_1}^2 - x_2{}' - x_3$›
‹$l_1 = (y_3 - y_2{}') \,/\, (x_3 - x_2{}')$› ‹$y_2{}' = y_2$› ‹$x_2{}' = x_2$›])
apply (*erule notE*)
apply (*rule sym*)
apply (*field ps′*(*1−2*))
apply (*simp add*: ‹$x_2{}' \neq x_3$› [*simplified* ‹$x_2{}' = x_2$›, *symmetric*])
apply (*rule conjI*)
apply (*simp add*: ‹$x_2{}' \neq x_3$› [*simplified* ‹$x_2{}' = x_2$›, *symmetric*])
apply (*rule conjI*)
apply (*rule notI*)
apply (*ring* (*prems*) *ps′*(*1−2*))
apply (*cut-tac* ‹$x_4{}' \neq x_3{}'$› [*simplified* ‹$x_4{}' = x_4$› ‹$x_3{}' = x_3$› ‹$x_4 = l^2 - x_1 - x_2$›
‹$l = (y_2 - y_1) \,/\, (x_2 - x_1)$›])
apply (*erule notE*)
apply (*rule sym*)
apply (*field ps′*(*1−2*))
apply (*simp-all add*: ‹$x_1 \neq x_2$› [*symmetric*])
done
qed
qed
qed
qed

```
lemma spec2-assoc:
  assumes p₁: on-curve a b p₁
  and p₂: on-curve a b p₂
  and p₃: on-curve a b p₃
  and is-generic p₁ p₂
  and is-tangent p₂ p₃
  and is-generic (add a p₁ p₂) p₃
  and is-generic p₁ (add a p₂ p₃)
  shows add a p₁ (add a p₂ p₃) = add a (add a p₁ p₂) p₃
  using p₁ p₂ assms
proof (induct rule: add-case)
  case InfL
  show ?case by (simp add: add-def)
next
  case InfR
  show ?case by (simp add: add-def)
next
  case Opp
  then show ?case by (simp add: is-generic-def)
next
  case Tan
  then show ?case by (simp add: is-generic-def)
next
  case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₄ x₄ y₄ l)
  with ‹on-curve a b p₂› ‹on-curve a b p₃›
  show ?case
  proof (induct rule: add-case)
    case InfL
    then show ?case by (simp add: is-generic-def)
  next
    case InfR
    then show ?case by (simp add: is-generic-def)
  next
    case Opp
    then show ?case by (simp add: is-generic-def)
  next
    case (Tan p₂ x₂' y₂' p₅ x₅ y₅ l₁)
    from ‹on-curve a b p₂› ‹p₅ = add a p₂ p₂›
    have on-curve a b p₅ by (simp add: add-closed)
    with ‹on-curve a b p₁› show ?case using Tan
    proof (induct rule: add-case)
      case InfL
      then show ?case by (simp add: is-generic-def)
    next
      case InfR
      then show ?case by (simp add: is-generic-def)
    next
      case (Opp p)
      from ‹is-generic p (opp p)› ‹on-curve a b p›
```

```
  show ?case by (simp add: is-generic-def opp-opp)
next
  case Tan
  then show ?case by (simp add: is-generic-def)
next
  case (Gen p1 x1' y1' p5' x5' y5' p6 x6 y6 l2)
  from ‹on-curve a b p1› ‹on-curve a b p2› ‹p4 = add a p1 p2›
  have on-curve a b p4 by (simp add: add-closed)
  then show ?case using ‹on-curve a b p2› Gen
  proof (induct rule: add-case)
    case InfL
    then show ?case by (simp add: is-generic-def)
  next
    case InfR
    then show ?case by (simp add: is-generic-def)
  next
    case (Opp p)
    from ‹is-generic p (opp p)›
    show ?case by (simp add: is-generic-def opp-opp)
  next
    case Tan
    then show ?case by (simp add: is-generic-def)
  next
    case (Gen p4' x4' y4' p3' x3' y3' p7 x7 y7 l3)
    from
      ‹on-curve a b p1› ‹p1 = Point x1 y1›
      ‹on-curve a b p2› ‹p2 = Point x2 y2›
    have
      y1: y1 ^ 2 = x1 ^ 3 + a * x1 + b and
      y2: y2 ^ 2 = x2 ^ 3 + a * x2 + b
      by (simp-all add: on-curve-def)
    from
      ‹p5' = Point x5' y5'›
      ‹p5' = Point x5 y5›
      ‹p4' = Point x4' y4'›
      ‹p4' = Point x4 y4›
      ‹p3' = Point x2' y2'›
      ‹p3' = Point x2 y2›
      ‹p3' = Point x3' y3'›
      ‹p1 = Point x1' y1'›
      ‹p1 = Point x1 y1›
    have ps:
      x5' = x5 y5' = y5
      x4' = x4 y4' = y4 x3' = x2 y3' = y2 x2' = x2 y2' = y2
      x1' = x1 y1' = y1
      by simp-all
    show ?case
      apply (simp add: ‹p6 = Point x6 y6› ‹p7 = Point x7 y7›)
      apply (simp only: ps
```

‹$x_7 = l_3 \hat{} 2 - x_4' - x_3'$›
‹$y_7 = - y_4' - l_3 * (x_7 - x_4')$›
‹$l_3 = (y_3' - y_4') / (x_3' - x_4')$›
‹$x_6 = l_2 \hat{} 2 - x_1' - x_5'$›
‹$y_6 = - y_1' - l_2 * (x_6 - x_1')$›
‹$l_2 = (y_5' - y_1') / (x_5' - x_1')$›
‹$x_5 = l_1 \hat{} 2 - 2 * x_2'$›
‹$y_5 = - y_2' - l_1 * (x_5 - x_2')$›
‹$l_1 = (3 * x_2' \hat{} 2 + a) / (2 * y_2')$›
‹$x_4 = l \hat{} 2 - x_1 - x_2$›
‹$y_4 = - y_1 - l * (x_4 - x_1)$›
‹$l = (y_2 - y_1) / (x_2 - x_1)$›)
apply (*rule conjI*)
apply (*field y1 y2*)
apply (*intro conjI*)
apply (*simp add*: ‹$y_2' \neq 0$› [*simplified* ‹$y_2' = y_2$›])
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*rule notE* [*OF* ‹$x_1' \neq x_5'$› [*simplified*
‹$x_5 = l_1 \hat{} 2 - 2 * x_2'$›
‹$l_1 = (3 * x_2' \hat{} 2 + a) / (2 * y_2')$›
‹$x_1' = x_1$› ‹$x_2' = x_2$› ‹$y_2' = y_2$› ‹$x_5' = x_5$›]])
apply (*rule sym*)
apply (*field y1 y2*)
apply (*simp add*: ‹$y_2' \neq 0$› [*simplified* ‹$y_2' = y_2$›])
apply (*simp add*: ‹$x_1 \neq x_2$› [*symmetric*])
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*rule notE* [*OF* ‹$x_4' \neq x_3'$› [*simplified*
‹$x_4 = l \hat{} 2 - x_1 - x_2$›
‹$l = (y_2 - y_1) / (x_2 - x_1)$›
‹$x_4' = x_4$› ‹$x_3' = x_2$›]])
apply (*rule sym*)
apply (*field y1 y2*)
apply (*simp add*: ‹$x_1 \neq x_2$› [*symmetric*])
apply (*field y1 y2*)
apply (*intro conjI*)
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*rule notE* [*OF* ‹$x_1' \neq x_5'$› [*simplified*
‹$x_5 = l_1 \hat{} 2 - 2 * x_2'$›
‹$l_1 = (3 * x_2' \hat{} 2 + a) / (2 * y_2')$›
‹$x_1' = x_1$› ‹$x_2' = x_2$› ‹$y_2' = y_2$› ‹$x_5' = x_5$›]])
apply (*rule sym*)
apply (*field y1 y2*)
apply (*simp add*: ‹$y_2' \neq 0$› [*simplified* ‹$y_2' = y_2$›])
apply (*simp add*: ‹$y_2' \neq 0$› [*simplified* ‹$y_2' = y_2$›])
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)

```
        apply (rule notE [OF ‹x4' ≠ x3'› [simplified
          ‹x4 = l ^ 2 − x1 − x2›
          ‹l = (y2 − y1) / (x2 − x1)›
          ‹x4' = x4› ‹x3' = x2›]])
        apply (rule sym)
        apply (field y1 y2)
        apply (simp-all add: ‹x1 ≠ x2› [symmetric])
        done
    qed
  qed
 next
  case (Gen p3 x3 y3 p5 x5 y5 p6 x6 y6 l1)
  then show ?case by (simp add: is-tangent-def)
 qed
qed

lemma spec3-assoc:
 assumes p1: on-curve a b p1
 and p2: on-curve a b p2
 and p3: on-curve a b p3
 and is-generic p1 p2
 and is-tangent p2 p3
 and is-generic (add a p1 p2) p3
 and is-tangent p1 (add a p2 p3)
 shows add a p1 (add a p2 p3) = add a (add a p1 p2) p3
 using p1 p2 assms
proof (induct rule: add-case)
 case InfL
 then show ?case by (simp add: is-generic-def)
next
 case InfR
 then show ?case by (simp add: is-generic-def)
next
 case Opp
 then show ?case by (simp add: is-generic-def)
next
 case Tan
 then show ?case by (simp add: is-generic-def)
next
 case (Gen p1 x1 y1 p2 x2 y2 p4 x4 y4 l)
 with ‹on-curve a b p2› ‹on-curve a b p3›
 show ?case
 proof (induct rule: add-case)
  case InfL
  then show ?case by (simp add: is-generic-def)
 next
  case InfR
  then show ?case by (simp add: is-generic-def)
 next
```

```
  case Opp
  then show ?case by (simp add: is-tangent-def opp-opp)
next
  case (Tan p_2 x_2' y_2' p_5 x_5 y_5 l_1)
  from ‹on-curve a b p_2› ‹p_5 = add a p_2 p_2›
  have on-curve a b p_5 by (simp add: add-closed)
  with ‹on-curve a b p_1› show ?case using Tan
  proof (induct rule: add-case)
    case InfL
    then show ?case by (simp add: is-generic-def)
  next
    case InfR
    then show ?case by (simp add: is-generic-def)
  next
    case Opp
    then show ?case by (simp add: is-tangent-def opp-opp)
  next
    case (Tan p_1 x_1' y_1' p_6 x_6 y_6 l_2)
    from ‹on-curve a b p_1› ‹on-curve a b p_2› ‹p_4 = add a p_1 p_2›
    have on-curve a b p_4 by (simp add: add-closed)
    then show ?case using ‹on-curve a b p_2› Tan
    proof (induct rule: add-case)
      case InfL
      then show ?case by (simp add: is-generic-def)
    next
      case InfR
      then show ?case by (simp add: is-generic-def)
    next
      case (Opp p)
      from ‹is-generic p (opp p)›
      show ?case by (simp add: is-generic-def opp-opp)
    next
      case Tan
      then show ?case by (simp add: is-generic-def)
    next
      case (Gen p_4' x_4' y_4' p_2' x_2'' y_2'' p_7 x_7 y_7 l_3)
      from
        ‹on-curve a b p_1› ‹p_1 = Point x_1 y_1›
        ‹on-curve a b p_2› ‹p_2 = Point x_2 y_2›
      have
        y1: y_1 ^ 2 = x_1 ^ 3 + a * x_1 + b and
        y2: y_2 ^ 2 = x_2 ^ 3 + a * x_2 + b
        by (simp-all add: on-curve-def)
      from
        ‹p_4' = Point x_4' y_4'›
        ‹p_4' = Point x_4 y_4›
        ‹p_2' = Point x_2' y_2'›
        ‹p_2' = Point x_2 y_2›
        ‹p_2' = Point x_2'' y_2''›
```

```
  ‹p1 = Point x1' y1'›
  ‹p1 = Point x1 y1›
  ‹p1 = Point x5 y5›
have ps:
  x4' = x4 y4' = y4 x2' = x2 y2' = y2 x2'' = x2 y2'' = y2
  x1' = x5 y1' = y5 x1 = x5 y1 = y5
  by simp-all
note qs =
  ‹x7 = l3 ^ 2 - x4' - x2''›
  ‹y7 = - y4' - l3 * (x7 - x4')›
  ‹l3 = (y2'' - y4') / (x2'' - x4')›
  ‹x6 = l2 ^ 2 - 2 * x1'›
  ‹y6 = - y1' - l2 * (x6 - x1')›
  ‹x5 = l1 ^ 2 - 2 * x2'›
  ‹y5 = - y2' - l1 * (x5 - x2')›
  ‹l1 = (3 * x2' ^ 2 + a) / (2 * y2')›
  ‹l2 = (3 * x1' ^ 2 + a) / (2 * y1')›
  ‹x4 = l ^ 2 - x1 - x2›
  ‹y4 = - y1 - l * (x4 - x1)›
  ‹l = (y2 - y1) / (x2 - x1)›
from ‹y2' ≠ 0› ‹y2' = y2›
have 2 * y2 ≠ 0 by simp
show ?case
  apply (simp add: ‹p6 = Point x6 y6› ‹p7 = Point x7 y7›)
  apply (simp only: ps qs)
  apply (rule conjI)
  apply (field y2)
  apply (intro conjI)
  apply (rule notI)
  apply (ring (prems))
  apply (rule notE [OF ‹y1' ≠ 0›])
  apply (simp only: ps qs)
  apply field
  apply (rule ‹2 * y2 ≠ 0›)
  apply (rule notI)
  apply (ring (prems))
  apply (rule notE [OF ‹x1 ≠ x2›])
  apply (rule sym)
  apply (simp only: ps qs)
  apply field
  apply (rule ‹2 * y2 ≠ 0›)
  apply (rule ‹2 * y2 ≠ 0›)
  apply (rule notI)
  apply (ring (prems))
  apply (rule notE [OF ‹x4' ≠ x2''›])
  apply (rule sym)
  apply (simp only: ps qs)
  apply field
  apply (intro conjI)
```

```
      apply (rule ‹2 * y₂ ≠ 0›)
      apply (erule thin-rl)
      apply (rule notI)
      apply (ring (prems))
      apply (rule notE [OF ‹x₁ ≠ x₂›])
      apply (rule sym)
      apply (simp only: ps qs)
      apply field
      apply (rule ‹2 * y₂ ≠ 0›)
      apply (field y2)
      apply (intro conjI)
      apply (rule notI)
      apply (ring (prems))
      apply (rule notE [OF ‹y₁' ≠ 0›])
      apply (simp only: ps qs)
      apply field
      apply (rule ‹2 * y₂ ≠ 0›)
      apply (rule notI)
      apply (ring (prems))
      apply (rule notE [OF ‹x₄' ≠ x₂''›])
      apply (rule sym)
      apply (simp only: ps qs)
      apply field
      apply (erule thin-rl)
      apply (rule conjI)
      apply (rule ‹2 * y₂ ≠ 0›)
      apply (rule notI)
      apply (ring (prems))
      apply (rule notE [OF ‹x₁ ≠ x₂›])
      apply (rule sym)
      apply (simp only: ps qs)
      apply field
      apply (rule ‹2 * y₂ ≠ 0›)
      apply (rule ‹2 * y₂ ≠ 0›)
      apply (rule notI)
      apply (ring (prems))
      apply (rule notE [OF ‹x₁ ≠ x₂›])
      apply (rule sym)
      apply (simp only: ps qs)
      apply field
      apply (rule ‹2 * y₂ ≠ 0›)
      done
  qed
 next
  case Gen
  then show ?case by (simp add: is-tangent-def)
 qed
next
 case Gen
```

```
    then show ?case by (simp add: is-tangent-def)
  qed
qed

lemma add-0-l: add a Infinity p = p
  by (simp add: add-def)

lemma add-0-r: add a p Infinity = p
  by (simp add: add-def split: point.split)

lemma add-opp: on-curve a b p ⟹ add a p (opp p) = Infinity
  by (simp add: add-def opp-def on-curve-def split: point.split-asm)

lemma add-comm:
  assumes on-curve a b p₁ on-curve a b p₂
  shows add a p₁ p₂ = add a p₂ p₁
proof (cases p₁)
  case Infinity
  then show ?thesis by (simp add: add-0-l add-0-r)
next
  case (Point x₁ y₁)
  note Point' = this
  with ‹on-curve a b p₁›
  have y1: y₁ ^ 2 = x₁ ^ 3 + a * x₁ + b
    by (simp add: on-curve-def)
  show ?thesis
  proof (cases p₂)
    case Infinity
    then show ?thesis by (simp add: add-0-l add-0-r)
  next
    case (Point x₂ y₂)
    with ‹on-curve a b p₂›
    have y2: y₂ ^ 2 = x₂ ^ 3 + a * x₂ + b
      by (simp add: on-curve-def)
    show ?thesis
    proof (cases x₁ = x₂)
      case True
      show ?thesis
      proof (cases y₁ = - y₂)
        case True
        with Point Point' ‹x₁ = x₂› show ?thesis
          by (simp add: add-def)
      next
        case False
        with y1 y2 [symmetric] ‹x₁ = x₂› Point Point'
        show ?thesis
          by (simp add: power2-eq-square square-eq-iff)
      qed
    next
```

```
      case False
      with Point Point' show ?thesis
        apply (simp add: add-def Let-def)
        apply (rule conjI)
        apply field
        apply simp
        apply field
        apply simp
        done
    qed
  qed
qed

lemma uniq-opp:
  assumes add a p₁ p₂ = Infinity
  shows p₂ = opp p₁
  using assms
  by (auto simp add: add-def opp-def Let-def
    split: point.split-asm if-split-asm)

lemma uniq-zero:
  assumes ab: nonsingular a b
  and p₁: on-curve a b p₁
  and p₂: on-curve a b p₂
  and add: add a p₁ p₂ = p₂
  shows p₁ = Infinity
  using p₁ p₂ assms
proof (induct rule: add-case)
  case InfL
  show ?case ..
next
  case InfR
  then show ?case by simp
next
  case Opp
  then show ?case by (simp add: opp-def split: point.split-asm)
next
  case (Tan p₁ x₁ y₁ p₂ x₂ y₂ l)
  from ‹p₁ = Point x₁ y₁› ‹p₂ = Point x₂ y₂› ‹p₂ = p₁›
  have x₂ = x₁ y₂ = y₁ by simp-all
  with ‹y₂ = - y₁ - l * (x₂ - x₁)› ‹y₁ ≠ 0›
  have - y₁ = y₁ by simp
  with ‹y₁ ≠ 0›
  show ?case by simp
next
  case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l)
  then have y1: y₁ ^ 2 = x₁ ^ 3 + a * x₁ + b
    and y2: y₂ ^ 2 = x₂ ^ 3 + a * x₂ + b
    by (simp-all add: on-curve-def)
```

```
  from ‹p3 = p2› ‹p2 = Point x2 y2› ‹p3 = Point x3 y3›
  have ps: x3 = x2 y3 = y2 by simp-all
  with ‹y3 = - y1 - l * (x3 - x1)›
  have y2 = - y1 - l * (x2 - x1) by simp
  also from ‹l = (y2 - y1) / (x2 - x1)› ‹x1 ≠ x2›
  have l * (x2 - x1) = y2 - y1
    by simp
  also have - y1 - (y2 - y1) = (- y1 + y1) + - y2
    by simp
  finally have y2 = 0 by simp
  with ‹p2 = Point x2 y2› ‹on-curve a b p2›
  have x2: x2 ^ 3 = - (a * x2 + b)
   by (simp add: on-curve-def eq-neg-iff-add-eq-0 add.assoc del: minus-add-distrib)
  from ‹x3 = l ^ 2 - x1 - x2› ‹x3 = x2›
  have l ^ 2 - x1 - x2 - x2 = x2 - x2 by simp
  then have l ^ 2 - x1 - 2 * x2 = 0 by simp
  then have x2 * (l ^ 2 - x1 - 2 * x2) = x2 * 0 by simp
  then have (x2 - x1) * (2 * a * x2 + 3 * b) = 0
    apply (simp only: ‹l = (y2 - y1) / (x2 - x1)› ‹y2 = 0›)
    apply (field (prems) y1 x2)
    apply (ring y1 x2)
    apply (simp add: ‹x1 ≠ x2› [symmetric])
    done
  with ‹x1 ≠ x2› have 2 * a * x2 + 3 * b = 0 by simp
  then have 2 * a * x2 = - (3 * b)
    by (simp add: eq-neg-iff-add-eq-0)
  from y2 [symmetric] ‹y2 = 0›
  have (- (2 * a)) ^ 3 * (x2 ^ 3 + a * x2 + b) = 0
    by simp
  then have b * (4 * a ^ 3 + 27 * b ^ 2) = 0
    apply (ring (prems) ‹2 * a * x2 = - (3 * b)›)
    apply (ring ‹2 * a * x2 = - (3 * b)›)
    done
  with ab have b = 0 by (simp add: nonsingular-def)
  with ‹2 * a * x2 + 3 * b = 0› ab
  have x2 = 0 by (simp add: nonsingular-def)
  from ‹l ^ 2 - x1 - 2 * x2 = 0›
  show ?case
    apply (simp add: ‹x2 = 0› ‹y2 = 0› ‹l = (y2 - y1) / (x2 - x1)›)
    apply (field (prems) y1 ‹b = 0›)
    apply (insert ab ‹b = 0› ‹x1 ≠ x2› ‹x2 = 0›)
    apply (simp add: nonsingular-def)
    apply simp
    done
qed

lemma opp-add:
  assumes p1: on-curve a b p1
  and p2: on-curve a b p2
```

```
  shows opp (add a p1 p2) = add a (opp p1) (opp p2)
proof (cases p1)
  case Infinity
  then show ?thesis by (simp add: add-def opp-def)
next
  case (Point x1 y1)
  show ?thesis
  proof (cases p2)
    case Infinity
    with ‹p1 = Point x1 y1› show ?thesis
      by (simp add: add-def opp-def)
  next
    case (Point x2 y2)
    with ‹p1 = Point x1 y1› p1 p2
    have x1 ^ 3 + a * x1 + b = y1 ^ 2
      x2 ^ 3 + a * x2 + b = y2 ^ 2
      by (simp-all add: on-curve-def)
    with Point ‹p1 = Point x1 y1› show ?thesis
      apply (cases x1 = x2)
      apply (cases y1 = - y2)
      apply (simp add: add-def opp-def Let-def)
     apply (simp add: add-def opp-def Let-def trans [OF minus-equation-iff eq-commute])
      apply (simp add: add-def opp-def Let-def)
      apply (rule conjI)
      apply field
      apply simp
      apply field
      apply simp
      done
  qed
qed

lemma compat-add-opp:
  assumes p1: on-curve a b p1
  and p2: on-curve a b p2
  and add a p1 p2 = add a p1 (opp p2)
  and p1 ≠ opp p1
  shows p2 = opp p2
  using p1 p2 assms
proof (induct rule: add-case)
  case InfL
  then show ?case by (simp add: add-0-l)
next
  case InfR
  then show ?case by (simp add: opp-def add-0-r)
next
  case (Opp p)
  then have add a p p = Infinity by (simp add: opp-opp)
  then have p = opp p by (rule uniq-opp)
```

```
  with ‹p ≠ opp p› show ?case ..
next
  case (Tan p₁ x₁ y₁ p₂ x₂ y₂ l)
  then have add a p₁ p₁ = Infinity
    by (simp add: add-opp)
  then have p₁ = opp p₁ by (rule uniq-opp)
  with ‹p₁ ≠ opp p₁› show ?case ..
next
  case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l)
  have (2::'a) * 2 ≠ 0
    by (simp only: mult-eq-0-iff) simp
  then have (4::'a) ≠ 0 by simp
  from Gen have ((− y₂ − y₁) / (x₂ − x₁)) ^ 2 − x₁ − x₂ =
    ((y₂ − y₁) / (x₂ − x₁)) ^ 2 − x₁ − x₂
    by (simp add: add-def opp-def Let-def)
  then show ?case
    apply (field (prems))
    apply (insert ‹p₁ ≠ opp p₁›
      ‹p₁ = Point x₁ y₁› ‹p₂ = Point x₂ y₂› ‹4 ≠ 0›)[1]
    apply (simp add: opp-def eq-neg-iff-add-eq-0)
    apply (simp add: ‹x₁ ≠ x₂› [symmetric])
    done
qed

lemma compat-add-triple:
  assumes ab: nonsingular a b
  and p: on-curve a b p
  and p ≠ opp p
  and add a p p ≠ opp p
  shows add a (add a p p) (opp p) = p
  using add-closed [OF p p] opp-closed [OF p] assms
proof (induct add a p p opp p rule: add-case)
  case InfL
  from ‹p ≠ opp p› uniq-opp [OF ‹Infinity = add a p p› [symmetric]]
  show ?case ..
next
  case InfR
  then show ?case by (simp add: opp-def split: point.split-asm)
next
  case Opp
  then have opp (opp (add a p p)) = opp (opp p) by simp
  then have add a p p = p by (simp add: opp-opp)
  with uniq-zero [OF ab p p] ‹p ≠ opp p›
  show ?case by (simp add: opp-def)
next
  case Tan
  then show ?case by simp
next
  case (Gen x₁ y₁ x₂ y₂ p₃ x₃ y₃ l)
```

```
  from ‹opp p = Point x₂ y₂›
  have p = Point x₂ (- y₂)
    by (auto simp add: opp-def split: point.split-asm)
  with ‹add a p p = Point x₁ y₁› [symmetric]
  obtain l' where l':
    l' = (3 * x₂ ^ 2 + a) / (2 * - y₂)
    and xy: x₁ = l' ^ 2 - 2 * x₂
    y₁ = - (- y₂) - l' * (x₁ - x₂)
    and y2: - y₂ ≠ - (- y₂)
    by (simp add: add-def Let-def split: if-split-asm)
  have x₃ = x₂
    apply (simp add: xy
      ‹l = (y₂ - y₁) / (x₂ - x₁)› ‹x₃ = l ^ 2 - x₁ - x₂›)
    apply field
    apply (insert ‹x₁ ≠ x₂›)
    apply (simp add: xy)
    done
  then have p₃ = p ∨ p₃ = opp p
    by (rule curve-elt-opp [OF ‹p₃ = Point x₃ y₃› ‹p = Point x₂ (- y₂)›
      add-closed [OF add-closed [OF p p] opp-closed [OF p],
        folded ‹p₃ = add a (add a p p) (opp p)›]
     ‹on-curve a b p›])
  then show ?case
  proof
    assume p₃ = p
    with ‹p₃ = add a (add a p p) (opp p)›
    show ?thesis by simp
  next
    assume p₃ = opp p
    with ‹p₃ = add a (add a p p) (opp p)›
    have add a (add a p p) (opp p) = opp p by simp
    with ab add-closed [OF p p] opp-closed [OF p]
    have add a p p = Infinity by (rule uniq-zero)
    with ‹add a p p = Point x₁ y₁› show ?thesis by simp
  qed
qed

lemma add-opp-double-opp:
  assumes ab: nonsingular a b
  and p₁: on-curve a b p₁
  and p₂: on-curve a b p₂
  and add a p₁ p₂ = opp p₁
  shows p₂ = add a (opp p₁) (opp p₁)
proof (cases p₁ = opp p₁)
  case True
  with assms have add a p₂ p₁ = p₁ by (simp add: add-comm)
  with ab p₂ p₁ have p₂ = Infinity by (rule uniq-zero)
  also from ‹on-curve a b p₁› have ... = add a p₁ (opp p₁)
    by (simp add: add-opp)
```

```
  also from True have ... = add a (opp p₁) (opp p₁) by simp
  finally show ?thesis .
next
  case False
  from p₁ p₂ False assms show ?thesis
  proof (induct rule: add-case)
    case InfL
    then show ?case by simp
  next
    case InfR
    then show ?case by simp
  next
    case Opp
    then show ?case by (simp add: add-0-l)
  next
    case (Tan p₁ x₁ y₁ p₂ x₂ y₂ l)
    from ‹p₂ = opp p₁› ‹on-curve a b p₁›
    have p₁ = opp p₂ by (simp add: opp-opp)
    also note ‹p₂ = add a p₁ p₁›
    finally show ?case using ‹on-curve a b p₁›
      by (simp add: opp-add)
  next
    case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l)
    from ‹on-curve a b p₁› ‹p₁ = Point x₁ y₁›
    have y₁: y₁ ^ 2 = x₁ ^ 3 + a * x₁ + b
      by (simp add: on-curve-def)
    from ‹on-curve a b p₂› ‹p₂ = Point x₂ y₂›
    have y₂: y₂ ^ 2 = x₂ ^ 3 + a * x₂ + b
      by (simp add: on-curve-def)
    from ‹p₁ = Point x₁ y₁› ‹p₁ ≠ opp p₁›
    have y₁ ≠ 0
      by (simp add: opp-Point)
    from Gen have x₁ = ((y₂ − y₁) / (x₂ − x₁)) ^ 2 − x₁ − x₂
      by (simp add: opp-Point)
    then have 2 * y₂ * y₁ = a * x₂ + 3 * x₂ * x₁ ^ 2 + a * x₁ −
      x₁ ^ 3 + 2 * b
      apply (field (prems) y₁ y₂)
      apply (field y₁ y₂)
      apply simp
      apply (simp add: ‹x₁ ≠ x₂› [symmetric])
      done
    then have (x₂ − (((3 * x₁ ^ 2 + a) / (2 * (− y₁))) ^ 2 −
      2 * x₁)) * (x₂ − x₁) ^ 2 = 0
      apply (drule-tac f=λx. x ^ 2 in arg-cong)
      apply (field (prems) y₁ y₂)
      apply (field y₁ y₂)
      apply (simp-all add: ‹y₁ ≠ 0›)
      done
    with ‹x₁ ≠ x₂›
```

```
    have x2 = ((3 * x1 ^ 2 + a) / (2 * (- y1))) ^ 2 - 2 * x1
      by simp
    with ‹p2 = Point x2 y2› - ‹on-curve a b p2›
      add-closed [OF
        opp-closed [OF ‹on-curve a b p1›] opp-closed [OF ‹on-curve a b p1›]]
    have p2 = add a (opp p1) (opp p1) ∨ p2 = opp (add a (opp p1) (opp p1))
      apply (rule curve-elt-opp)
      apply (simp add: add-def opp-Point Let-def ‹p1 = Point x1 y1› ‹y1 ≠ 0›)
      done
    then show ?case
    proof
      assume p2 = opp (add a (opp p1) (opp p1))
      with ‹on-curve a b p1›
      have p2 = add a p1 p1
        by (simp add: opp-add [of a b] opp-opp opp-closed)
      show ?case
      proof (cases add a p1 p1 = opp p1)
        case True
        from ‹on-curve a b p1›
        show ?thesis
          apply (simp add: opp-add [symmetric] ‹p2 = add a p1 p1› True)
          apply (simp add: ‹p3 = add a p1 p2› [simplified ‹p3 = opp p1›])
          apply (simp add: ‹p2 = add a p1 p1› True add-opp)
          done
      next
        case False
        from ‹on-curve a b p1›
        have add a p1 (opp p2) = opp (add a (add a p1 p1) (opp p1))
          by (simp add: ‹p2 = add a p1 p1›
            opp-add [of a b] add-closed opp-closed opp-opp add-comm [of a b])
        with ab ‹on-curve a b p1› ‹p1 ≠ opp p1› False
        have add a p1 (opp p2) = opp p1
          by (simp add: compat-add-triple)
        with ‹p3 = add a p1 p2› ‹p3 = opp p1›
        have add a p1 p2 = add a p1 (opp p2) by simp
        with ‹on-curve a b p1› ‹on-curve a b p2›
        have p2 = opp p2 using ‹p1 ≠ opp p1›
          by (rule compat-add-opp)
        with ‹on-curve a b p1› ‹p2 = add a p1 p1›
        show ?thesis by (simp add: opp-add)
      qed
    qed
  qed
qed

lemma cancel:
  assumes ab: nonsingular a b
  and p1: on-curve a b p1
  and p2: on-curve a b p2
```

```
  and p3: on-curve a b p3
  and eq: add a p1 p2 = add a p1 p3
  shows p2 = p3
  using p1 p2 p1 p2 eq
proof (induct rule: add-casew)
  case InfL
  then show ?case by (simp add: add-0-l)
next
  case (InfR p)
  with p3 have add a p3 p = p by (simp add: add-comm)
  with ab p3 ‹on-curve a b p› 
  show ?case by (rule uniq-zero [symmetric])
next
  case (Opp p)
  from ‹Infinity = add a p p3› [symmetric]
  show ?case by (rule uniq-opp [symmetric])
next
  case (Gen p1 x1 y1 p2 x2 y2 p4 x4 y4 l)
  from ‹on-curve a b p1› p3 ‹on-curve a b p1› p3 ‹p1 = Point x1 y1›
    ‹p4 = add a p1 p2› ‹p4 = add a p1 p3› ‹p1 ≠ opp p2›
  show ?case
  proof (induct rule: add-casew)
    case InfL
    then show ?case by (simp add: add-0-l)
  next
    case (InfR p)
    with ‹on-curve a b p2›
    have add a p2 p = p by (simp add: add-comm)
    with ab ‹on-curve a b p2› ‹on-curve a b p›
    show ?case by (rule uniq-zero)
  next
    case (Opp p)
    then have add a p p2 = Infinity by simp
    then show ?case by (rule uniq-opp)
  next
    case (Gen p1 x1' y1' p3 x3 y3 p5 x5 y5 l')
    from ‹p4 = p5› ‹p4 = Point x4 y4› ‹p5 = Point x5 y5›
      ‹p1 = Point x1' y1'› ‹p1 = Point x1 y1›
      ‹y4 = - y1 - l * (x4 - x1)› ‹y5 = - y1' - l' * (x5 - x1')›
    have 0 = - y1 - l * (x4 - x1) - (- y1 - l' * (x4 - x1))
      by auto
    then have l' = l ∨ x4 = x1 by auto
    then show ?case
    proof
      assume l' = l
      with ‹p4 = p5› ‹p4 = Point x4 y4› ‹p5 = Point x5 y5›
        ‹p1 = Point x1' y1'› ‹p1 = Point x1 y1›
        ‹x4 = l ^ 2 - x1 - x2› ‹x5 = l' ^ 2 - x1' - x3›
      have 0 = l ^ 2 - x1 - x2 - (l ^ 2 - x1 - x3)
```

```
    by simp
  then have x₂ = x₃ by simp
  with ‹p₂ = Point x₂ y₂› ‹p₃ = Point x₃ y₃› ‹on-curve a b p₂› ‹on-curve a b
p₃›
  have p₂ = p₃ ∨ p₂ = opp p₃ by (rule curve-elt-opp)
  then show ?case
  proof
    assume p₂ = opp p₃
    with ‹on-curve a b p₃› have opp p₂ = p₃
      by (simp add: opp-opp)
    with ‹p₄ = p₅› ‹p₄ = add a p₁ p₂› ‹p₅ = add a p₁ p₃›
    have add a p₁ p₂ = add a p₁ (opp p₂) by simp
    show ?case
    proof (cases p₁ = opp p₁)
      case True
      with ‹p₁ ≠ opp p₂› ‹p₁ ≠ opp p₃›
      have p₁ ≠ p₂ p₁ ≠ p₃ by auto
      with ‹l' = l› ‹x₁ = x₂ ∧ -∨ -› ‹x₁' = x₃ ∧ - ∨ -›
        ‹p₁ = Point x₁ y₁› ‹p₁ = Point x₁' y₁'›
        ‹p₂ = Point x₂ y₂› ‹p₃ = Point x₃ y₃›
        ‹p₂ = opp p₃›
      have eq: (y₂ − y₁) / (x₂ − x₁) = (y₃ − y₁) / (x₂ − x₁) and x₁ ≠ x₂
        by (auto simp add: opp-Point)
      from eq have y₂ = y₃
        apply (field (prems))
        apply (simp-all add: ‹x₁ ≠ x₂› [symmetric])
        done
      with ‹p₂ = opp p₃› ‹p₂ = Point x₂ y₂› ‹p₃ = Point x₃ y₃›
      show ?thesis by (simp add: opp-Point)
    next
      case False
      with ‹on-curve a b p₁› ‹on-curve a b p₂›
        ‹add a p₁ p₂ = add a p₁ (opp p₂)›
      have p₂ = opp p₂ by (rule compat-add-opp)
      with ‹opp p₂ = p₃› show ?thesis by simp
    qed
  qed
next
  assume x₄ = x₁
  with ‹p₄ = Point x₄ y₄› [simplified ‹p₄ = add a p₁ p₂›]
    ‹p₁ = Point x₁ y₁›
    add-closed [OF ‹on-curve a b p₁› ‹on-curve a b p₂›]
    ‹on-curve a b p₁›
  have add a p₁ p₂ = p₁ ∨ add a p₁ p₂ = opp p₁ by (rule curve-elt-opp)
  then show ?case
  proof
    assume add a p₁ p₂ = p₁
    with ‹on-curve a b p₁› ‹on-curve a b p₂›
    have add a p₂ p₁ = p₁ by (simp add: add-comm)
```

```
      with ab ‹on-curve a b p2› ‹on-curve a b p1›
      have p2 = Infinity by (rule uniq-zero)
      moreover from ‹add a p1 p2 = p1›
        ‹p4 = p5› ‹p4 = add a p1 p2› ‹p5 = add a p1 p3›
        ‹on-curve a b p1› ‹on-curve a b p3›
      have add a p3 p1 = p1 by (simp add: add-comm)
      with ab ‹on-curve a b p3› ‹on-curve a b p1›
      have p3 = Infinity by (rule uniq-zero)
      ultimately show ?case by simp
    next
      assume add a p1 p2 = opp p1
      with ab ‹on-curve a b p1› ‹on-curve a b p2›
      have p2 = add a (opp p1) (opp p1) by (rule add-opp-double-opp)
      moreover from ‹add a p1 p2 = opp p1›
        ‹p4 = p5› ‹p4 = add a p1 p2› ‹p5 = add a p1 p3›
      have add a p1 p3 = opp p1 by simp
      with ab ‹on-curve a b p1› ‹on-curve a b p3›
      have p3 = add a (opp p1) (opp p1) by (rule add-opp-double-opp)
      ultimately show ?case by simp
    qed
  qed
 qed
qed

lemma add-minus-id:
  assumes ab: nonsingular a b
  and p1: on-curve a b p1
  and p2: on-curve a b p2
  shows add a (add a p1 p2) (opp p2) = p1
proof (cases add a p1 p2 = opp p2)
  case True
  then have add a (add a p1 p2) (opp p2) = add a (opp p2) (opp p2)
    by simp
  also from p1 p2 True have add a p2 p1 = opp p2
    by (simp add: add-comm)
  with ab p2 p1 have add a (opp p2) (opp p2) = p1
    by (rule add-opp-double-opp [symmetric])
  finally show ?thesis .
next
  case False
  from p1 p2 p1 p2 False show ?thesis
  proof (induct rule: add-case)
    case InfL
    then show ?case by (simp add: add-opp)
  next
    case InfR
    show ?case by (simp add: add-0-r)
  next
    case Opp
```

```
  then show ?case by (simp add: opp-opp add-0-l)
next
  case (Tan p₁ x₁ y₁ p₂ x₂ y₂ l)
  note ab ‹on-curve a b p₁›
  moreover from ‹y₁ ≠ 0› ‹p₁ = Point x₁ y₁›
  have p₁ ≠ opp p₁ by (simp add: opp-Point)
  moreover from ‹p₂ = add a p₁ p₁› ‹p₂ ≠ opp p₁›
  have add a p₁ p₁ ≠ opp p₁ by simp
  ultimately have add a (add a p₁ p₁) (opp p₁) = p₁
    by (rule compat-add-triple)
  with ‹p₂ = add a p₁ p₁› show ?case by simp
next
  case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l)
  from ‹p₃ = add a p₁ p₂› ‹on-curve a b p₂›
  have p₃ = add a p₁ (opp (opp p₂)) by (simp add: opp-opp)
  with
    add-closed [OF ‹on-curve a b p₁› ‹on-curve a b p₂›,
      folded ‹p₃ = add a p₁ p₂›]
    opp-closed [OF ‹on-curve a b p₂›]
    opp-closed [OF ‹on-curve a b p₂›]
    opp-opp [of p₂]
    Gen
  show ?case
  proof (induct rule: add-case)
    case InfL
    then show ?case by simp
  next
    case InfR
    then show ?case by (simp add: add-0-r)
  next
    case (Opp p)
    from ‹p = add a p₁ (opp (opp p))›
    have add a p₁ p = p by (simp add: opp-opp)
    with ab ‹on-curve a b p₁› ‹on-curve a b p›
    show ?case by (rule uniq-zero [symmetric])
  next
    case Tan
    then show ?case by simp
  next
    case (Gen p₄ x₄ y₄ p₅ x₅ y₅ p₆ x₆ y₆ l')
    from ‹on-curve a b p₅› ‹opp p₅ = p₂›
      ‹p₂ = Point x₂ y₂› ‹p₅ = Point x₅ y₅›
    have y₅ = − y₂ x₅ = x₂
      by (auto simp add: opp-Point on-curve-def)
    from ‹p₄ = Point x₃ y₃› ‹p₄ = Point x₄ y₄›
    have x₄ = x₃ y₄ = y₃ by simp-all
    from ‹x₄ ≠ x₅› show ?case
      apply (simp add:
        ‹y₅ = − y₂› ‹x₅ = x₂›
```

```
      ‹x4 = x3› ‹y4 = y3›
      ‹p6 = Point x6 y6› ‹p1 = Point x1 y1›
      ‹x6 = l' ^ 2 - x4 - x5› ‹y6 = - y4 - l' * (x6 - x4)›
      ‹l' = (y5 - y4) / (x5 - x4)›
      ‹x3 = l ^ 2 - x1 - x2› ‹y3 = - y1 - l * (x3 - x1)›
      ‹l = (y2 - y1) / (x2 - x1)›)
    apply (rule conjI)
    apply field
    apply (rule conjI)
    apply (rule notI)
    apply (erule notE)
    apply (ring (prems))
    apply (rule sym)
    apply field
    apply (simp-all add: ‹x1 ≠ x2› [symmetric])
    apply field
    apply (rule conjI)
    apply (rule notI)
    apply (erule notE)
    apply (ring (prems))
    apply (rule sym)
    apply field
    apply (simp-all add: ‹x1 ≠ x2› [symmetric])
    done
  qed
 qed
qed

lemma add-shift-minus:
 assumes ab: nonsingular a b
 and p1: on-curve a b p1
 and p2: on-curve a b p2
 and p3: on-curve a b p3
 and eq: add a p1 p2 = p3
 shows p1 = add a p3 (opp p2)
proof -
 note eq
 also from add-minus-id [OF ab p3 opp-closed [OF p2]] p2
 have p3 = add a (add a p3 (opp p2)) p2 by (simp add: opp-opp)
 finally have add a p2 p1 = add a p2 (add a p3 (opp p2))
  using p1 p2 p3
  by (simp add: add-comm [of a b] add-closed opp-closed)
 with ab p2 p1 add-closed [OF p3 opp-closed [OF p2]]
 show ?thesis by (rule cancel)
qed

lemma degen-assoc:
 assumes ab: nonsingular a b
 and p1: on-curve a b p1
```

```
  and p₂: on-curve a b p₂
  and p₃: on-curve a b p₃
  and H:
    (p₁ = Infinity ∨ p₂ = Infinity ∨ p₃ = Infinity) ∨
     (p₁ = opp p₂ ∨ p₂ = opp p₃) ∨
     (opp p₁ = add a p₂ p₃ ∨ opp p₃ = add a p₁ p₂)
  shows add a p₁ (add a p₂ p₃) = add a (add a p₁ p₂) p₃
  using H
proof (elim disjE)
  assume p₁ = Infinity
  then show ?thesis by (simp add: add-0-l)
next
  assume p₂ = Infinity
  then show ?thesis by (simp add: add-0-l add-0-r)
next
  assume p₃ = Infinity
  then show ?thesis by (simp add: add-0-r)
next
  assume p₁ = opp p₂
  from p₂ p₃
  have add a (opp p₂) (add a p₂ p₃) = add a (add a p₃ p₂) (opp p₂)
    by (simp add: add-comm [of a b] add-closed opp-closed)
  also from ab p₃ p₂ have ... = p₃ by (rule add-minus-id)
  also have ... = add a Infinity p₃ by (simp add: add-0-l)
  also from p₂ have ... = add a (add a p₂ (opp p₂)) p₃
    by (simp add: add-opp)
  also from p₂ have ... = add a (add a (opp p₂) p₂) p₃
    by (simp add: add-comm [of a b] opp-closed)
  finally show ?thesis using ‹p₁ = opp p₂› by simp
next
  assume p₂ = opp p₃
  from p₃
  have add a p₁ (add a (opp p₃) p₃) = add a p₁ (add a p₃ (opp p₃))
    by (simp add: add-comm [of a b] opp-closed)
  also from ab p₁ p₃
  have ... = add a (add a p₁ (opp p₃)) (opp (opp p₃))
    by (simp add: add-opp add-minus-id add-0-r opp-closed)
  finally show ?thesis using p₃ ‹p₂ = opp p₃›
    by (simp add: opp-opp)
next
  assume eq: opp p₁ = add a p₂ p₃
  from eq [symmetric] p₁
  have add a p₁ (add a p₂ p₃) = Infinity by (simp add: add-opp)
  also from p₃ have ... = add a p₃ (opp p₃) by (simp add: add-opp)
  also from p₃ have ... = add a (opp p₃) p₃
    by (simp add: add-comm [of a b] opp-closed)
  also from ab p₂ p₃
  have ... = add a (add a (add a (opp p₃) (opp p₂)) (opp (opp p₂))) p₃
    by (simp add: add-minus-id opp-closed)
```

```
  also from p2 p3
  have ... = add a (add a (add a (opp p2) (opp p3)) p2) p3
    by (simp add: add-comm [of a b] opp-opp opp-closed)
  finally show ?thesis
    using opp-add [OF p2 p3] eq [symmetric] p1
    by (simp add: opp-opp)
next
  assume eq: opp p3 = add a p1 p2
  from opp-add [OF p1 p2] eq [symmetric] p3
  have add a p1 (add a p2 p3) = add a p1 (add a p2 (add a (opp p1) (opp p2)))
    by (simp add: opp-opp)
  also from p1 p2
  have ... = add a p1 (add a (add a (opp p1) (opp p2)) (opp (opp p2)))
    by (simp add: add-comm [of a b] opp-opp add-closed opp-closed)
  also from ab p1 p2 have ... = Infinity
    by (simp add: add-minus-id add-opp opp-closed)
  also from p3 have ... = add a p3 (opp p3) by (simp add: add-opp)
  also from p3 have ... = add a (opp p3) p3
    by (simp add: add-comm [of a b] opp-closed)
  finally show ?thesis using eq [symmetric] by simp
qed

lemma spec4-assoc:
  assumes ab: nonsingular a b
  and p1: on-curve a b p1
  and p2: on-curve a b p2
  shows add a p1 (add a p2 p2) = add a (add a p1 p2) p2
proof (cases p1 = Infinity)
  case True
  from ab p1 p2 p2
  show ?thesis by (rule degen-assoc) (simp add: True)
next
  case False
  show ?thesis
  proof (cases p2 = Infinity)
    case True
    from ab p1 p2 p2
    show ?thesis by (rule degen-assoc) (simp add: True)
  next
    case False
    show ?thesis
    proof (cases p2 = opp p2)
      case True
      from ab p1 p2 p2
      show ?thesis by (rule degen-assoc) (simp add: True [symmetric])
    next
      case False
      show ?thesis
      proof (cases p1 = opp p2)
```

```
  case True
  from ab p1 p2 p2
  show ?thesis by (rule degen-assoc) (simp add: True)
next
  case False
  show ?thesis
  proof (cases opp p1 = add a p2 p2)
    case True
    from ab p1 p2 p2
    show ?thesis by (rule degen-assoc) (simp add: True)
  next
    case False
    show ?thesis
    proof (cases opp p2 = add a p1 p2)
      case True
      from ab p1 p2 p2
      show ?thesis by (rule degen-assoc) (simp add: True)
    next
      case False
      show ?thesis
      proof (cases p1 = add a p2 p2)
        case True
        from p1 p2 ‹p1 ≠ opp p2› ‹p2 ≠ opp p2›
          ‹opp p1 ≠ add a p2 p2› ‹opp p2 ≠ add a p1 p2›
          ‹p1 ≠ Infinity› ‹p2 ≠ Infinity›
        show ?thesis
          apply (simp add: True)
          apply (rule spec3-assoc)
          apply (simp-all add: is-generic-def is-tangent-def)
          apply (rule notI)
          apply (drule uniq-zero [OF ab p2 p2])
          apply simp
          apply (intro conjI notI)
          apply (erule notE)
          apply (rule uniq-opp [of a])
          apply (simp add: add-comm [of a b] add-closed)
          apply (erule notE)
          apply (drule uniq-zero [OF ab add-closed [OF p2 p2] p2])
          apply simp
          done
      next
        case False
        show ?thesis
        proof (cases p2 = add a p1 p2)
          case True
          from ab p1 p2 True [symmetric]
          have p1 = Infinity by (rule uniq-zero)
          then show ?thesis by (simp add: add-0-l)
        next
```

```
            case False
            show ?thesis
            proof (cases p₁ = p₂)
              case True
              with p₂ show ?thesis
                by (simp add: add-comm [of a b] add-closed)
            next
              case False
              with p₁ p₂ ‹p₁ ≠ Infinity› ‹p₂ ≠ Infinity›
                ‹p₁ ≠ opp p₂› ‹p₂ ≠ opp p₂›
                ‹p₁ ≠ add a p₂ p₂› ‹p₂ ≠ add a p₁ p₂› ‹opp p₂ ≠ add a p₁ p₂›
              show ?thesis
                apply (rule-tac spec2-assoc)
                apply (simp-all add: is-generic-def is-tangent-def)
                apply (rule notI)
                apply (erule notE [of p₁ = opp p₂])
                apply (rule uniq-opp [of a])
                apply (simp add: add-comm)
                apply (intro conjI notI)
                apply (erule notE [of p₂ = opp p₂])
                apply (rule uniq-opp)
                apply assumption+
                apply (rule notE [OF ‹opp p₁ ≠ add a p₂ p₂›])
                apply (simp add: opp-opp)
                done
            qed
          qed
        qed
      qed
    qed
  qed
qed

lemma add-assoc:
  assumes ab: nonsingular a b
  and p₁: on-curve a b p₁
  and p₂: on-curve a b p₂
  and p₃: on-curve a b p₃
  shows add a p₁ (add a p₂ p₃) = add a (add a p₁ p₂) p₃
proof (cases p₁ = Infinity)
  case True
  from ab p₁ p₂ p₃
  show ?thesis by (rule degen-assoc) (simp add: True)
next
  case False
  show ?thesis
  proof (cases p₂ = Infinity)
```

```
  case True
  from ab p1 p2 p3
  show ?thesis by (rule degen-assoc) (simp add: True)
next
  case False
  show ?thesis
  proof (cases p3 = Infinity)
    case True
    from ab p1 p2 p3
    show ?thesis by (rule degen-assoc) (simp add: True)
  next
    case False
    show ?thesis
    proof (cases p1 = p2)
      case True
      from p2 p3
      have add a p2 (add a p2 p3) = add a (add a p3 p2) p2
        by (simp add: add-comm [of a b] add-closed)
      also from ab p3 p2 have ... = add a p3 (add a p2 p2)
        by (simp add: spec4-assoc)
      also from p2 p3
      have ... = add a (add a p2 p2) p3
        by (simp add: add-comm [of a b] add-closed)
      finally show ?thesis using True by simp
    next
      case False
      show ?thesis
      proof (cases p1 = opp p2)
        case True
        from ab p1 p2 p3
        show ?thesis by (rule degen-assoc) (simp add: True)
      next
        case False
        show ?thesis
        proof (cases p2 = p3)
          case True
          with ab p1 p3 show ?thesis
            by (simp add: spec4-assoc)
        next
          case False
          show ?thesis
          proof (cases p2 = opp p3)
            case True
            from ab p1 p2 p3
            show ?thesis by (rule degen-assoc) (simp add: True)
          next
            case False
            show ?thesis
            proof (cases opp p1 = add a p2 p3)
```

```
  case True
  from ab p1 p2 p3
  show ?thesis by (rule degen-assoc) (simp add: True)
next
  case False
  show ?thesis
  proof (cases opp p3 = add a p1 p2)
    case True
    from ab p1 p2 p3
    show ?thesis by (rule degen-assoc) (simp add: True)
  next
    case False
    show ?thesis
    proof (cases p1 = add a p2 p3)
      case True
      with ab p2 p3 show ?thesis
        apply simp
        apply (rule cancel [OF ab opp-closed [OF p3]])
        apply (simp-all add: add-closed)
        apply (simp add: spec4-assoc add-closed opp-closed)
        apply (simp add: add-comm [of a b opp p3]
          add-closed opp-closed add-minus-id)
        apply (simp add: add-comm [of a b] add-closed)
        done
    next
      case False
      show ?thesis
      proof (cases p3 = add a p1 p2)
        case True
        with ab p1 p2 show ?thesis
          apply simp
          apply (rule cancel [OF ab opp-closed [OF p1]])
          apply (simp-all add: add-closed)
          apply (simp add: spec4-assoc add-closed opp-closed)
          apply (simp add: add-comm [of a b opp p1] add-comm [of a b
p1]
            add-closed opp-closed add-minus-id)
          done
      next
        case False
        with p1 p2 p3
          ‹p1 ≠ Infinity› ‹p2 ≠ Infinity› ‹p3 ≠ Infinity›
          ‹p1 ≠ p2› ‹p1 ≠ opp p2› ‹p2 ≠ p3› ‹p2 ≠ opp p3›
          ‹opp p3 ≠ add a p1 p2› ‹p1 ≠ add a p2 p3›
        show ?thesis
          apply (rule-tac spec1-assoc [of a b])
          apply (simp-all add: is-generic-def)
          apply (rule notI)
          apply (erule notE [of p1 = opp p2])
```

```
                    apply (rule uniq-opp [of a])
                    apply (simp add: add-comm)
                    apply (intro conjI notI)
                    apply (erule notE [of p2 = opp p3])
                    apply (rule uniq-opp [of a])
                    apply (simp add: add-comm)
                    apply (rule notE [OF ‹opp p1 ≠ add a p2 p3›])
                    apply (simp add: opp-opp)
                    done
                qed
              qed
            qed
          qed
        qed
      qed
    qed
  qed
qed

lemma add-comm':
  nonsingular a b ⟹
   on-curve a b p1 ⟹ on-curve a b p2 ⟹ on-curve a b p3 ⟹
   add a p2 (add a p1 p3) = add a p1 (add a p2 p3)
   by (simp add: add-assoc add-comm)

primrec (in ell-field) point-mult :: 'a ⇒ nat ⇒ 'a point ⇒ 'a point
where
    point-mult a 0 p = Infinity
  | point-mult a (Suc n) p = add a p (point-mult a n p)

lemma point-mult-closed: on-curve a b p ⟹ on-curve a b (point-mult a n p)
  by (induct n) (simp-all add: add-closed)

lemma point-mult-add:
  on-curve a b p ⟹ nonsingular a b ⟹
   point-mult a (m + n) p = add a (point-mult a m p) (point-mult a n p)
  by (induct m) (simp-all add: add-assoc point-mult-closed add-0-l)

lemma point-mult-mult:
  on-curve a b p ⟹ nonsingular a b ⟹
   point-mult a (m * n) p = point-mult a n (point-mult a m p)
   by (induct n) (simp-all add: point-mult-add)

lemma point-mult2-eq-double:
  point-mult a 2 p = add a p p
  by (simp add: numeral-2-eq-2 add-0-r)
```

2.2 Projective Coordinates

type-synonym *'a ppoint* = *'a* × *'a* × *'a*

context *ell-field* **begin**

definition *pdouble* :: *'a* ⇒ *'a ppoint* ⇒ *'a ppoint* **where**
```
pdouble a p =
  (let (x, y, z) = p
   in
     if z = 0 then p
     else
       let
         l = 2 * y * z;
         m = 3 * x ^ 2 + a * z ^ 2
       in
         (l * (m ^ 2 - 4 * x * y * l),
          m * (6 * x * y * l - m ^ 2) -
          2 * y ^ 2 * l ^ 2,
          l ^ 3))
```

definition *padd* :: *'a* ⇒ *'a ppoint* ⇒ *'a ppoint* ⇒ *'a ppoint* **where**
```
padd a p1 p2 =
  (let
     (x1, y1, z1) = p1;
     (x2, y2, z2) = p2
   in
     if z1 = 0 then p2
     else if z2 = 0 then p1
     else
       let
         d1 = x2 * z1;
         d2 = x1 * z2;
         l = d1 - d2;
         m = y2 * z1 - y1 * z2
       in
         if l = 0 then
           if m = 0 then pdouble a p1
           else (0, 0, 0)
         else
           let h = m ^ 2 * z1 * z2 - (d1 + d2) * l ^ 2
           in
             (l * h,
              (d2 * l ^ 2 - h) * m - l ^ 3 * y1 * z2,
              l ^ 3 * z1 * z2))
```

definition *make-affine* :: *'a ppoint* ⇒ *'a point* **where**
```
make-affine p =
  (let (x, y, z) = p
   in if z = 0 then Infinity else Point (x / z) (y / z))
```

```
definition on-curvep :: 'a ⇒ 'a ⇒ 'a ppoint ⇒ bool where
  on-curvep a b = (λ(x, y, z). z ≠ 0 ⟶
    y ^ 2 * z = x ^ 3 + a * x * z ^ 2 + b * z ^ 3)

end

lemma on-curvep-infinity [simp]: on-curvep a b (x, y, 0)
  by (simp add: on-curvep-def)

lemma make-affine-infinity [simp]: make-affine (x, y, 0) = Infinity
  by (simp add: make-affine-def)

lemma on-curvep-iff-on-curve:
  on-curvep a b p = on-curve a b (make-affine p)
proof (induct p rule: prod-induct3)
  case (fields x y z)
  show on-curvep a b (x, y, z) = on-curve a b (make-affine (x, y, z))
  proof
    assume H: on-curvep a b (x, y, z)
    then have yz: z ≠ 0 ⟹ y ^ 2 * z = x ^ 3 + a * x * z ^ 2 + b * z ^ 3
      by (simp-all add: on-curvep-def)
    show on-curve a b (make-affine (x, y, z))
    proof (cases z = 0)
      case True
      then show ?thesis by (simp add: on-curve-def make-affine-def)
    next
      case False
      then show ?thesis
        apply (simp add: on-curve-def make-affine-def)
        apply (field yz [OF False])
        apply assumption
        done
    qed
  next
    assume H: on-curve a b (make-affine (x, y, z))
    show on-curvep a b (x, y, z)
    proof (cases z = 0)
      case True
      then show ?thesis
        by (simp add: on-curvep-def)
    next
      case False
      from H show ?thesis
        apply (simp add: on-curve-def on-curvep-def make-affine-def False)
        apply (field (prems))
        apply field
        apply (simp-all add: False)
        done
```

```
    qed
  qed
qed

lemma pdouble-infinity [simp]: pdouble a (x, y, 0) = (x, y, 0)
  by (simp add: pdouble-def)

lemma padd-infinity-l [simp]: padd a (x, y, 0) p = p
  by (simp add: padd-def)

lemma pdouble-correct:
  make-affine (pdouble a p) = add a (make-affine p) (make-affine p)
proof (induct p rule: prod-induct3)
  case (fields x y z)
  then show ?case
   apply (auto simp add: add-def pdouble-def make-affine-def eq-opp-is-zero Let-def)
    apply field
    apply simp
    apply field
    apply simp
    done
qed

lemma padd-correct:
  assumes p₁: on-curvep a b p₁ and p₂: on-curvep a b p₂
  shows make-affine (padd a p₁ p₂) = add a (make-affine p₁) (make-affine p₂)
  using p₁
proof (induct p₁ rule: prod-induct3)
  case (fields x₁ y₁ z₁)
  note p₁' = fields
  from p₂ show ?case
  proof (induct p₂ rule: prod-induct3)
    case (fields x₂ y₂ z₂)
    then have
      yz₂: z₂ ≠ 0 ⟹ y₂ ^ 2 * z₂ * z₁ ^ 3 =
        (x₂ ^ 3 + a * x₂ * z₂ ^ 2 + b * z₂ ^ 3) * z₁ ^ 3
      by (simp-all add: on-curvep-def)
    from p₁' have
      yz₁: z₁ ≠ 0 ⟹ y₁ ^ 2 * z₁ * z₂ ^ 3 =
        (x₁ ^ 3 + a * x₁ * z₁ ^ 2 + b * z₁ ^ 3) * z₂ ^ 3
      by (simp-all add: on-curvep-def)
    show ?case
    proof (cases z₁ = 0)
      case True
      then show ?thesis
        by (simp add: add-def padd-def make-affine-def)
    next
      case False
      show ?thesis
```

```
proof (cases z₂ = 0)
  case True
  then show ?thesis
    by (simp add: add-def padd-def make-affine-def)
next
  case False
  show ?thesis
  proof (cases x₂ * z₁ − x₁ * z₂ = 0)
    case True
    note x = this
    then have x': x₂ * z₁ = x₁ * z₂ by simp
    show ?thesis
    proof (cases y₂ * z₁ − y₁ * z₂ = 0)
      case True
      then have y: y₂ * z₁ = y₁ * z₂ by simp
      from ‹z₁ ≠ 0› ‹z₂ ≠ 0› x
      have make-affine (x₂, y₂, z₂) = make-affine (x₁, y₁, z₁)
        apply (simp add: make-affine-def)
        apply (rule conjI)
        apply (field x')
        apply simp
        apply (field y)
        apply simp
        done
      with True x ‹z₁ ≠ 0› ‹z₂ ≠ 0› p₁' fields show ?thesis
        by (simp add: padd-def pdouble-correct)
    next
      case False
      have y₂ ^ 2 * z₁ ^ 3 * z₂ = y₁ ^ 2 * z₁ * z₂ ^ 3
        by (ring yz₁ [OF ‹z₁ ≠ 0›] yz₂ [OF ‹z₂ ≠ 0›] x')
      then have y₂ ^ 2 * z₁ ^ 3 * z₂ / z₁ / z₂ =
        y₁ ^ 2 * z₁ * z₂ ^ 3 / z₁ / z₂
        by simp
      then have (y₂ * z₁) * (y₂ * z₁) = (y₁ * z₂) * (y₁ * z₂)
        apply (field (prems))
        apply (field)
        apply (rule TrueI)
        apply (simp add: ‹z₁ ≠ 0› ‹z₂ ≠ 0›)
        done
      with False
      have y₂z₁: y₂ * z₁ = − (y₁ * z₂)
        by (simp add: square-eq-iff)
      from x False ‹z₁ ≠ 0› ‹z₂ ≠ 0› show ?thesis
        apply (simp add: padd-def add-def make-affine-def Let-def)
        apply (rule conjI)
        apply (rule impI)
        apply (field x')
        apply simp
        apply (field y₂z₁)
```

```
          apply simp
          done
      qed
    next
      case False
      then have x1 / z1 ≠ x2 / z2
        apply (rule-tac notI)
        apply (erule notE)
        apply (drule sym)
        apply (field (prems))
        apply ring
        apply (simp add: ‹z1 ≠ 0› ‹z2 ≠ 0›)
        done
      with False ‹z1 ≠ 0› ‹z2 ≠ 0›
      show ?thesis
        apply (auto simp add: padd-def add-def make-affine-def Let-def)
        apply field
        apply simp
        apply field
        apply simp
        done
    qed
  qed
 qed
qed

lemma pdouble-closed:
  on-curvep a b p ⟹ on-curvep a b (pdouble a p)
  by (simp add: on-curvep-iff-on-curve pdouble-correct add-closed)

lemma padd-closed:
  on-curvep a b p1 ⟹ on-curvep a b p2 ⟹ on-curvep a b (padd a p1 p2)
  by (simp add: on-curvep-iff-on-curve padd-correct add-closed)

primrec (in ell-field) ppoint-mult :: 'a ⇒ nat ⇒ 'a ppoint ⇒ 'a ppoint
where
    ppoint-mult a 0 p = (0, 0, 0)
  | ppoint-mult a (Suc n) p = padd a p (ppoint-mult a n p)

lemma ppoint-mult-closed [simp]:
  on-curvep a b p ⟹ on-curvep a b (ppoint-mult a n p)
  by (induct n) (simp-all add: padd-closed)

lemma ppoint-mult-correct: on-curvep a b p ⟹
  make-affine (ppoint-mult a n p) = point-mult a n (make-affine p)
  by (induct n) (simp-all add: padd-correct)

context ell-field begin
```

definition *proj-eq* :: *'a ppoint ⇒ 'a ppoint ⇒ bool* **where**
proj-eq = (λ(x_1, y_1, z_1) (x_2, y_2, z_2).
($z_1 = 0$) = ($z_2 = 0$) ∧ $x_1 * z_2 = x_2 * z_1$ ∧ $y_1 * z_2 = y_2 * z_1$)

end

lemma *proj-eq-refl*: *proj-eq p p*
by (*auto simp add*: *proj-eq-def*)

lemma *proj-eq-sym*: *proj-eq p p' ⟹ proj-eq p' p*
by (*auto simp add*: *proj-eq-def*)

lemma *proj-eq-trans*:
in-carrierp p ⟹ in-carrierp p' ⟹ in-carrierp p'' ⟹
proj-eq p p' ⟹ proj-eq p' p'' ⟹ proj-eq p p''
proof (*induct p rule*: *prod-induct3*)
case (*fields x y z*)
then show *?case*
proof (*induct p' rule*: *prod-induct3*)
case (*fields x' y' z'*)
then show *?case*
proof (*induct p'' rule*: *prod-induct3*)
case (*fields x'' y'' z''*)
then have
z: (*z = 0*) = (*z' = 0*) (*z' = 0*) = (*z'' = 0*) **and**
*x * z' * z'' = x' * z * z''*
*y * z' * z'' = y' * z * z''*
and *xy*:
*x' * z'' = x'' * z'*
*y' * z'' = y'' * z'*
by (*simp-all add*: *proj-eq-def*)
from ‹*x * z' * z'' = x' * z * z''*›
have (*x * z''*) * *z' = (x'' * z) * z'*
by (*ring* (*prems*) *xy*) (*ring xy*)
moreover from ‹*y * z' * z'' = y' * z * z''*›
have (*y * z''*) * *z' = (y'' * z) * z'*
by (*ring* (*prems*) *xy*) (*ring xy*)
ultimately show *?case* **using** *z*
by (*auto simp add*: *proj-eq-def*)
qed
qed
qed

lemma *make-affine-proj-eq-iff*:
proj-eq p p' = (make-affine p = make-affine p')
proof (*induct p rule*: *prod-induct3*)
case (*fields x y z*)
then show *?case*

```
  proof (induct p' rule: prod-induct3)
    case (fields x' y' z')
    show ?case
    proof
      assume proj-eq (x, y, z) (x', y', z')
      then have (z = 0) = (z' = 0)
        and xy: x * z' = x' * z y * z' = y' * z
        by (simp-all add: proj-eq-def)
      then show make-affine (x, y, z) = make-affine (x', y', z')
        apply (auto simp add: make-affine-def)
        apply (field xy)
        apply simp
        apply (field xy)
        apply simp
        done
    next
      assume H: make-affine (x, y, z) = make-affine (x', y', z')
      show proj-eq (x, y, z) (x', y', z')
      proof (cases z = 0)
        case True
        with H have z' = 0 by (simp add: make-affine-def split: if-split-asm)
        with True show ?thesis by (simp add: proj-eq-def)
      next
        case False
        with H have z' ≠ 0 x / z = x' / z' y / z = y' / z'
          by (simp-all add: make-affine-def split: if-split-asm)
        from ‹x / z = x' / z'›
        have x * z' = x' * z
          apply (field (prems))
          apply field
          apply (simp-all add: ‹z ≠ 0› ‹z' ≠ 0›)
          done
        moreover from ‹y / z = y' / z'›
        have y * z' = y' * z
          apply (field (prems))
          apply field
          apply (simp-all add: ‹z ≠ 0› ‹z' ≠ 0›)
          done
        ultimately show ?thesis
          by (simp add: proj-eq-def ‹z ≠ 0› ‹z' ≠ 0›)
      qed
    qed
  qed
qed

lemma pdouble-proj-eq-cong:
  proj-eq p p' ⟹ proj-eq (pdouble a p) (pdouble a p')
  by (simp add: make-affine-proj-eq-iff pdouble-correct)
```

lemma *padd-proj-eq-cong*:
on-curvep a b $p_1 \Longrightarrow$ *on-curvep a b* $p_1' \Longrightarrow$ *on-curvep a b* $p_2 \Longrightarrow$ *on-curvep a b* $p_2' \Longrightarrow$
proj-eq p_1 $p_1' \Longrightarrow$ *proj-eq* p_2 $p_2' \Longrightarrow$ *proj-eq* (*padd a* p_1 p_2) (*padd a* p_1' p_2')
by (*simp add*: *make-affine-proj-eq-iff padd-correct*)

end

3 Formalization using Locales

theory *Elliptic-Locale*
imports *HOL−Decision-Procs.Reflective-Field*
begin

3.1 Affine Coordinates

datatype *'a point* = *Infinity* | *Point 'a 'a*

locale *ell-field* = *field* +
assumes *two-not-zero*: «*2*» ≠ **0**
begin

declare *two-not-zero* [*simplified*, *simp add*]

lemma *neg-equal-zero*:
assumes *x*: *x* ∈ *carrier R*
shows (⊖ *x* = *x*) = (*x* = **0**)
proof
assume ⊖ *x* = *x*
with *x* **have** «*2*» ⊗ *x* = *x* ⊕ ⊖ *x*
by (*simp add*: *of-int-2 l-distr*)
with *x* **show** *x* = **0** **by** (*simp add*: *r-neg integral-iff*)
qed *simp*

lemmas *equal-neg-zero* = *trans* [*OF eq-commute neg-equal-zero*]

definition *nonsingular* :: *'a* ⇒ *'a* ⇒ *bool* **where**
nonsingular a b = («*4*» ⊗ *a* [^] (*3*::*nat*) ⊕ «*27*» ⊗ *b* [^] (*2*::*nat*) ≠ **0**)

definition *on-curve* :: *'a* ⇒ *'a* ⇒ *'a point* ⇒ *bool* **where**
on-curve a b p = (*case p of*
Infinity ⇒ *True*
| *Point x y* ⇒ *x* ∈ *carrier R* ∧ *y* ∈ *carrier R* ∧
y [^] (*2*::*nat*) = *x* [^] (*3*::*nat*) ⊕ *a* ⊗ *x* ⊕ *b*)

definition *add* :: *'a* ⇒ *'a point* ⇒ *'a point* ⇒ *'a point* **where**
add a p_1 p_2 = (*case* p_1 *of*
Infinity ⇒ p_2
| *Point* x_1 y_1 ⇒ (*case* p_2 *of*

```
      Infinity ⇒ p₁
    | Point x₂ y₂ ⇒
        if x₁ = x₂ then
          if y₁ = ⊖ y₂ then Infinity
          else
            let
              l = («3» ⊗ x₁ [^] (2::nat) ⊕ a) ⊘ («2» ⊗ y₁);
              x₃ = l [^] (2::nat) ⊖ «2» ⊗ x₁
            in
              Point x₃ (⊖ y₁ ⊖ l ⊗ (x₃ ⊖ x₁))
        else
          let
            l = (y₂ ⊖ y₁) ⊘ (x₂ ⊖ x₁);
            x₃ = l [^] (2::nat) ⊖ x₁ ⊖ x₂
          in
            Point x₃ (⊖ y₁ ⊖ l ⊗ (x₃ ⊖ x₁))))

definition opp :: 'a point ⇒ 'a point where
  opp p = (case p of
      Infinity ⇒ Infinity
    | Point x y ⇒ Point x (⊖ y))

lemma on-curve-infinity [simp]: on-curve a b Infinity
  by (simp add: on-curve-def)

lemma opp-Infinity [simp]: opp Infinity = Infinity
  by (simp add: opp-def)

lemma opp-Point: opp (Point x y) = Point x (⊖ y)
  by (simp add: opp-def)

lemma opp-opp: on-curve a b p ⟹ opp (opp p) = p
  by (auto simp add: opp-def on-curve-def split: point.split)

lemma opp-closed:
  on-curve a b p ⟹ on-curve a b (opp p)
  by (auto simp add: on-curve-def opp-def power2-eq-square
    l-minus r-minus split: point.split)

lemma curve-elt-opp:
  assumes p₁ = Point x₁ y₁
  and p₂ = Point x₂ y₂
  and on-curve a b p₁
  and on-curve a b p₂
  and x₁ = x₂
  shows p₁ = p₂ ∨ p₁ = opp p₂
proof −
  from ‹p₁ = Point x₁ y₁› ‹on-curve a b p₁›
  have y₁ ∈ carrier R y₁ [^] (2::nat) = x₁ [^] (3::nat) ⊕ a ⊗ x₁ ⊕ b
```

```
    by (simp-all add: on-curve-def)
  moreover from ‹p₂ = Point x₂ y₂› ‹on-curve a b p₂› ‹x₁ = x₂›
  have y₂ ∈ carrier R x₁ [^] (3::nat) ⊕ a ⊗ x₁ ⊕ b = y₂ [^] (2::nat)
    by (simp-all add: on-curve-def)
  ultimately have y₁ = y₂ ∨ y₁ = ⊖ y₂
    by (simp add: square-eq-iff power2-eq-square)
  with ‹p₁ = Point x₁ y₁› ‹p₂ = Point x₂ y₂› ‹x₁ = x₂› show ?thesis
    by (auto simp add: opp-def)
qed

lemma add-closed:
  assumes a ∈ carrier R and b ∈ carrier R
  and on-curve a b p₁ and on-curve a b p₂
  shows on-curve a b (add a p₁ p₂)
proof (cases p₁)
  case (Point x₁ y₁)
  note Point' = this
  show ?thesis
  proof (cases p₂)
    case (Point x₂ y₂)
    show ?thesis
    proof (cases x₁ = x₂)
      case True
      note True' = this
      show ?thesis
      proof (cases y₁ = ⊖ y₂)
        case True
        with True' Point Point'
        show ?thesis
          by (simp add: on-curve-def add-def)
      next
        case False
        from ‹on-curve a b p₁› Point' True'
        have x₂ ∈ carrier R y₁ ∈ carrier R and
          on-curve1: y₁ [^] (2::nat) = x₂ [^] (3::nat) ⊕ a ⊗ x₂ ⊕ b
          by (simp-all add: on-curve-def)
        from False True' Point Point' assms
        have y₁ ≠ 0
          apply (auto simp add: on-curve-def nat-pow-zero)
          apply (drule sym [of 0])
          apply simp
          done
        with False True' Point Point' assms
        show ?thesis
          apply (simp add: on-curve-def add-def Let-def integral-iff)
          apply (field on-curve1)
          apply (simp add: integral-iff)
          done
      qed
```

```
    next
      case False
      from ‹on-curve a b p₁› Point'
      have  x₁ ∈ carrier R y₁ ∈ carrier R
        and on-curve1: y₁ [^] (2::nat) = x₁ [^] (3::nat) ⊕ a ⊗ x₁ ⊕ b
        by (simp-all add: on-curve-def)
      from ‹on-curve a b p₂› Point
      have x₂ ∈ carrier R y₂ ∈ carrier R
        and on-curve2: y₂ [^] (2::nat) = x₂ [^] (3::nat) ⊕ a ⊗ x₂ ⊕ b
        by (simp-all add: on-curve-def)
      from assms not-sym [OF False] show ?thesis
        apply (simp add: on-curve-def add-def Let-def False Point Point' eq-diff0)
        apply (field on-curve1 on-curve2)
        apply (simp add: eq-diff0)
        done
    qed
  next
    case Infinity
    with Point ‹on-curve a b p₁› show ?thesis
      by (simp add: add-def)
  qed
next
  case Infinity
  with ‹on-curve a b p₂› show ?thesis
    by (simp add: add-def)
qed

lemma add-case [consumes 4, case-names InfL InfR Opp Tan Gen]:
  assumes a ∈ carrier R
  and b ∈ carrier R
  and p: on-curve a b p
  and q: on-curve a b q
  and R1: ⋀p. P Infinity p p
  and R2: ⋀p. P p Infinity p
  and R3: ⋀p. on-curve a b p ⟹ P p (opp p) Infinity
  and R4: ⋀p₁ x₁ y₁ p₂ x₂ y₂ l.
    p₁ = Point x₁ y₁ ⟹ p₂ = Point x₂ y₂ ⟹
    p₂ = add a p₁ p₁ ⟹ y₁ ≠ 0 ⟹
    l = («3» ⊗ x₁ [^] (2::nat) ⊕ a) ⊘ («2» ⊗ y₁) ⟹
    x₂ = l [^] (2::nat) ⊖ «2» ⊗ x₁ ⟹
    y₂ = ⊖ y₁ ⊖ l ⊗ (x₂ ⊖ x₁) ⟹
    P p₁ p₁ p₂
  and R5: ⋀p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l.
    p₁ = Point x₁ y₁ ⟹ p₂ = Point x₂ y₂ ⟹ p₃ = Point x₃ y₃ ⟹
    p₃ = add a p₁ p₂ ⟹ x₁ ≠ x₂ ⟹
    l = (y₂ ⊖ y₁) ⊘ (x₂ ⊖ x₁) ⟹
    x₃ = l [^] (2::nat) ⊖ x₁ ⊖ x₂ ⟹
    y₃ = ⊖ y₁ ⊖ l ⊗ (x₃ ⊖ x₁) ⟹
    P p₁ p₂ p₃
```

```
  shows P p q (add a p q)
proof (cases p)
  case Infinity
  then show ?thesis
    by (simp add: add-def R1)
next
  case (Point x1 y1)
  note Point' = this
  with p have x1 ∈ carrier R y1 ∈ carrier R
    and p': y1 [^] (2::nat) = x1 [^] (3::nat) ⊕ a ⊗ x1 ⊕ b
    by (simp-all add: on-curve-def)
  show ?thesis
  proof (cases q)
    case Infinity
    with Point show ?thesis
      by (simp add: add-def R2)
  next
    case (Point x2 y2)
    with q have x2 ∈ carrier R y2 ∈ carrier R
      and q': y2 [^] (2::nat) = x2 [^] (3::nat) ⊕ a ⊗ x2 ⊕ b
      by (simp-all add: on-curve-def)
    show ?thesis
    proof (cases x1 = x2)
      case True
      note True' = this
      show ?thesis
      proof (cases y1 = ⊖ y2)
        case True
        with p Point Point' True' R3 [of p] ‹y2 ∈ carrier R› show ?thesis
          by (simp add: add-def opp-def)
      next
        case False
        have (y1 ⊖ y2) ⊗ (y1 ⊕ y2) = 0
          by (ring True' p' q')
        with False ‹y1 ∈ carrier R› ‹y2 ∈ carrier R› have y1 = y2
          by (simp add: eq-neg-iff-add-eq-0 integral-iff eq-diff0)
        with False True' Point Point' show ?thesis
          apply simp
          apply (rule R4)
          apply (auto simp add: add-def Let-def)
          done
      qed
    next
      case False
      with Point Point' show ?thesis
        apply −
        apply (rule R5)
        apply (auto simp add: add-def Let-def)
        done
```

```
    qed
  qed
qed

lemma add-casew [consumes 4, case-names InfL InfR Opp Gen]:
  assumes a: a ∈ carrier R
  and b: b ∈ carrier R
  and p: on-curve a b p
  and q: on-curve a b q
  and R1: ⋀p. P Infinity p p
  and R2: ⋀p. P p Infinity p
  and R3: ⋀p. on-curve a b p ⟹ P p (opp p) Infinity
  and R4: ⋀p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l.
    p₁ = Point x₁ y₁ ⟹ p₂ = Point x₂ y₂ ⟹ p₃ = Point x₃ y₃ ⟹
    p₃ = add a p₁ p₂ ⟹ p₁ ≠ opp p₂ ⟹
    x₁ = x₂ ∧ y₁ = y₂ ∧ l = («3» ⊗ x₁ [^] (2::nat) ⊕ a) ⊘ («2» ⊗ y₁) ∨
    x₁ ≠ x₂ ∧ l = (y₂ ⊖ y₁) ⊘ (x₂ ⊖ x₁) ⟹
    x₃ = l [^] (2::nat) ⊖ x₁ ⊖ x₂ ⟹
    y₃ = ⊖ y₁ ⊖ l ⊗ (x₃ ⊖ x₁) ⟹
    P p₁ p₂ p₃
  shows P p q (add a p q)
  using a b p q p q
proof (induct rule: add-case)
  case InfL
  show ?case by (rule R1)
next
  case InfR
  show ?case by (rule R2)
next
  case (Opp p)
  from ‹on-curve a b p› show ?case by (rule R3)
next
  case (Tan p₁ x₁ y₁ p₂ x₂ y₂ l)
  with a b show ?case
    apply (rule-tac R4)
    apply assumption+
    apply (simp add: opp-Point equal-neg-zero on-curve-def)
    apply simp
    apply (simp add: minus-eq mult2 integral-iff a-assoc r-minus on-curve-def)
    apply simp
    done
next
  case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l)
  then show ?case
    apply (rule-tac R4)
    apply assumption+
    apply (simp add: opp-Point)
    apply simp-all
    done
```

qed

definition
is-tangent p q = (*p* ≠ *Infinity* ∧ *p* = *q* ∧ *p* ≠ *opp q*)

definition
is-generic p q =
(*p* ≠ *Infinity* ∧ *q* ≠ *Infinity* ∧
p ≠ *q* ∧ *p* ≠ *opp q*)

lemma *spec1-assoc*:
assumes *a*: *a* ∈ *carrier R*
and *b*: *b* ∈ *carrier R*
and p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
and p_3: *on-curve a b* p_3
and *is-generic* p_1 p_2
and *is-generic* p_2 p_3
and *is-generic* (*add a* p_1 p_2) p_3
and *is-generic* p_1 (*add a* p_2 p_3)
shows *add a* p_1 (*add a* p_2 p_3) = *add a* (*add a* p_1 p_2) p_3
using *a b* p_1 p_2 *assms*
proof (*induct rule*: *add-case*)
case *InfL*
show *?case* **by** (*simp add*: *add-def*)
next
case *InfR*
show *?case* **by** (*simp add*: *add-def*)
next
case *Opp*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case *Tan*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case (*Gen* p_1 x_1 y_1 p_2 x_2 y_2 p_4 x_4 y_4 *l*)
with *a b* ‹*on-curve a b* p_2› ‹*on-curve a b* p_3›
show *?case*
proof (*induct rule*: *add-case*)
case *InfL*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case *InfR*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case *Opp*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case *Tan*

```
    then show ?case by (simp add: is-generic-def)
  next
    case (Gen p2 x2' y2' p3 x3 y3 p5 x5 y5 l1)
    from a b ‹on-curve a b p2› ‹on-curve a b p3› ‹p5 = add a p2 p3›
    have on-curve a b p5 by (simp add: add-closed)
    with a b ‹on-curve a b p1› show ?case using Gen [simplified ‹p2 = Point x2'
y2'›]
    proof (induct rule: add-case)
      case InfL
      then show ?case by (simp add: is-generic-def)
    next
      case InfR
      then show ?case by (simp add: is-generic-def)
    next
      case (Opp p)
      from ‹on-curve a b p› ‹is-generic p (opp p)›
      show ?case by (simp add: is-generic-def opp-opp)
    next
      case Tan
      then show ?case by (simp add: is-generic-def)
    next
      case (Gen p1 x1' y1' p5' x5' y5' p6 x6 y6 l2)
      from a b ‹on-curve a b p1› ‹on-curve a b (Point x2' y2')›
        ‹p4 = add a p1 (Point x2' y2')›
      have on-curve a b p4 by (simp add: add-closed)
      with a b show ?case using ‹on-curve a b p3› Gen
      proof (induct rule: add-case)
        case InfL
        then show ?case by (simp add: is-generic-def)
      next
        case InfR
        then show ?case by (simp add: is-generic-def)
      next
        case (Opp p)
        from ‹on-curve a b p› ‹is-generic p (opp p)›
        show ?case by (simp add: is-generic-def opp-opp)
      next
        case Tan
        then show ?case by (simp add: is-generic-def)
      next
        case (Gen p4' x4' y4' p3' x3' y3' p7 x7 y7 l3)
        from ‹p4' = Point x4' y4'› ‹p4' = Point x4 y4›
        have p4: x4' = x4 y4' = y4 by simp-all
        from ‹p3' = Point x3' y3'› ‹p3' = Point x3 y3›
        have p3: x3' = x3 y3' = y3 by simp-all
        from ‹p1 = Point x1' y1'› ‹p1 = Point x1 y1›
        have p1: x1' = x1 y1' = y1 by simp-all
        from ‹p5' = Point x5' y5'› ‹p5' = Point x5 y5›
        have p5: x5' = x5 y5' = y5 by simp-all
```

from ‹*Point* $x_2{}'$ $y_2{}'$ = *Point* x_2 y_2›
have p_2: $x_2{}'$ = x_2 $y_2{}'$ = y_2 **by** *simp-all*
note ps = p_1 p_2 p_3 p_4 p_5
from
‹*on-curve a b* p_1› ‹p_1 = *Point* x_1 y_1›
‹*on-curve a b* p_2› ‹p_2 = *Point* x_2 y_2›
‹*on-curve a b* p_3› ‹p_3 = *Point* x_3 y_3›
have
$x_1 \in$ *carrier R* $y_1 \in$ *carrier R* **and** *y1*: y_1 [^] (*2*::*nat*) = x_1 [^] (*3*::*nat*) ⊕ $a \otimes x_1 \oplus b$ **and**
$x_2 \in$ *carrier R* $y_2 \in$ *carrier R* **and** *y2*: y_2 [^] (*2*::*nat*) = x_2 [^] (*3*::*nat*) ⊕ $a \otimes x_2 \oplus b$ **and**
$x_3 \in$ *carrier R* $y_3 \in$ *carrier R* **and** *y3*: y_3 [^] (*2*::*nat*) = x_3 [^] (*3*::*nat*) ⊕ $a \otimes x_3 \oplus b$
by (*simp-all add*: *on-curve-def*)
show *?case*
apply (*simp add*: ‹p_6 = *Point* x_6 y_6› ‹p_7 = *Point* x_7 y_7›)
apply (*simp only*: *ps*
‹$x_6 = l_2$ [^] *2* ⊖ $x_1{}'$ ⊖ $x_5{}'$› ‹$x_7 = l_3$ [^] *2* ⊖ $x_4{}'$ ⊖ $x_3{}'$›
‹y_6 = ⊖ $y_1{}'$ ⊖ $l_2 \otimes (x_6 \ominus x_1{}')$› ‹$y_7$ = ⊖ $y_4{}'$ ⊖ $l_3 \otimes (x_7 \ominus x_4{}')$›
‹$l_2 = (y_5{}' \ominus y_1{}') \oslash (x_5{}' \ominus x_1{}')$› ‹$l_3 = (y_3{}' \ominus y_4{}') \oslash (x_3{}' \ominus x_4{}')$›
‹$l_1 = (y_3 \ominus y_2{}') \oslash (x_3 \ominus x_2{}')$› ‹$l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1)$›
‹$x_5 = l_1$ [^] *2* ⊖ $x_2{}'$ ⊖ x_3› ‹y_5 = ⊖ $y_2{}'$ ⊖ $l_1 \otimes (x_5 \ominus x_2{}')$›
‹$x_4 = l$ [^] *2* ⊖ x_1 ⊖ x_2› ‹y_4 = ⊖ y_1 ⊖ $l \otimes (x_4 \ominus x_1)$›)
apply (*rule conjI*)
apply (*field y1 y2 y3*)
apply (*rule conjI*)
apply (*simp add*: *eq-diff0* ‹$x_3 \in$ *carrier R*› ‹$x_2 \in$ *carrier R*›
not-sym [*OF* ‹$x_2{}' \neq x_3$› [*simplified* ‹$x_2{}'$ = x_2›]])
apply (*rule conjI*)
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*cut-tac* ‹$x_1{}' \neq x_5{}'$› [*simplified* ‹$x_5{}'$ = x_5› ‹$x_1{}'$ = x_1› ‹$x_5 = l_1$ [^] *2* ⊖ $x_2{}'$ ⊖ x_3›
‹$l_1 = (y_3 \ominus y_2{}') \oslash (x_3 \ominus x_2{}')$› ‹$y_2{}'$ = y_2› ‹$x_2{}'$ = x_2›])
apply (*erule notE*)
apply (*rule sym*)
apply (*field y1 y2*)
apply (*simp add*: *eq-diff0* ‹$x_3 \in$ *carrier R*› ‹$x_2 \in$ *carrier R*›
not-sym [*OF* ‹$x_2{}' \neq x_3$› [*simplified* ‹$x_2{}'$ = x_2›]])
apply (*rule conjI*)
apply (*simp add*: *eq-diff0* ‹$x_2 \in$ *carrier R*› ‹$x_1 \in$ *carrier R*› *not-sym* [*OF* ‹$x_1 \neq x_2$›])
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*cut-tac* ‹$x_4{}' \neq x_3{}'$› [*simplified* ‹$x_4{}'$ = x_4› ‹$x_3{}'$ = x_3› ‹$x_4 = l$ [^] *2* ⊖ x_1 ⊖ x_2›
‹$l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1)$›])
apply (*erule notE*)

apply (*rule sym*)
apply (*field y1 y2*)
apply (*simp add*: *eq-diff0* ‹$x_2 \in$ *carrier* R› ‹$x_1 \in$ *carrier* R› *not-sym* [*OF* ‹$x_1 \neq x_2$›])
apply (*field y1 y2 y3*)
apply (*rule conjI*)
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*cut-tac* ‹$x_1' \neq x_5'$› [*simplified* ‹$x_5' = x_5$› ‹$x_1' = x_1$› ‹$x_5 = l_1$ [^] *2* ⊖ x_2' ⊖ x_3›
‹$l_1 = (y_3 \ominus y_2') \oslash (x_3 \ominus x_2')$› ‹$y_2' = y_2$› ‹$x_2' = x_2$›])
apply (*erule notE*)
apply (*rule sym*)
apply (*field y1 y2*)
apply (*simp add*: *eq-diff0* ‹$x_3 \in$ *carrier* R› ‹$x_2 \in$ *carrier* R›
not-sym [*OF* ‹$x_2' \neq x_3$› [*simplified* ‹$x_2' = x_2$›]])
apply (*rule conjI*)
apply (*simp add*: *eq-diff0* ‹$x_3 \in$ *carrier* R› ‹$x_2 \in$ *carrier* R›
not-sym [*OF* ‹$x_2' \neq x_3$› [*simplified* ‹$x_2' = x_2$›]])
apply (*rule conjI*)
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*cut-tac* ‹$x_4' \neq x_3'$› [*simplified* ‹$x_4' = x_4$› ‹$x_3' = x_3$› ‹$x_4 = l$ [^] *2* ⊖ x_1 ⊖ x_2›
‹$l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1)$›])
apply (*erule notE*)
apply (*rule sym*)
apply (*field y1 y2*)
apply (*simp-all add*: *eq-diff0* ‹$x_2 \in$ *carrier* R› ‹$x_1 \in$ *carrier* R› *not-sym* [*OF* ‹$x_1 \neq x_2$›])
done
qed
qed
qed
qed

lemma *spec2-assoc*:
assumes *a*: $a \in$ *carrier* R
and *b*: $b \in$ *carrier* R
and p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
and p_3: *on-curve a b* p_3
and *is-generic* p_1 p_2
and *is-tangent* p_2 p_3
and *is-generic* (*add a* p_1 p_2) p_3
and *is-generic* p_1 (*add a* p_2 p_3)
shows *add a* p_1 (*add a* p_2 p_3) = *add a* (*add a* p_1 p_2) p_3
using *a b* p_1 p_2 *assms*
proof (*induct rule*: *add-case*)

```
  case InfL
  show ?case by (simp add: add-def)
next
  case InfR
  show ?case by (simp add: add-def)
next
  case Opp
  then show ?case by (simp add: is-generic-def)
next
  case Tan
  then show ?case by (simp add: is-generic-def)
next
  case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₄ x₄ y₄ l)
  with a b ‹on-curve a b p₂› ‹on-curve a b p₃›
  show ?case
  proof (induct rule: add-case)
    case InfL
    then show ?case by (simp add: is-generic-def)
  next
    case InfR
    then show ?case by (simp add: is-generic-def)
  next
    case Opp
    then show ?case by (simp add: is-generic-def)
  next
    case (Tan p₂ x₂' y₂' p₅ x₅ y₅ l₁)
    from a b ‹on-curve a b p₂› ‹p₅ = add a p₂ p₂›
    have on-curve a b p₅ by (simp add: add-closed)
    with a b ‹on-curve a b p₁› show ?case using Tan
    proof (induct rule: add-case)
      case InfL
      then show ?case by (simp add: is-generic-def)
    next
      case InfR
      then show ?case by (simp add: is-generic-def)
    next
      case (Opp p)
      from ‹is-generic p (opp p)› ‹on-curve a b p›
      show ?case by (simp add: is-generic-def opp-opp)
    next
      case Tan
      then show ?case by (simp add: is-generic-def)
    next
      case (Gen p₁ x₁' y₁' p₅' x₅' y₅' p₆ x₆ y₆ l₂)
      from a b ‹on-curve a b p₁› ‹on-curve a b p₂› ‹p₄ = add a p₁ p₂›
      have on-curve a b p₄ by (simp add: add-closed)
      with a b show ?case using ‹on-curve a b p₂› Gen
      proof (induct rule: add-case)
        case InfL
```

then show *?case* **by** (*simp add*: *is-generic-def*)
next
case *InfR*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case (*Opp p*)
from ‹*is-generic p* (*opp p*)› ‹*on-curve a b p*›
show *?case* **by** (*simp add*: *is-generic-def opp-opp*)
next
case *Tan*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case (*Gen* $p_4{'}$ $x_4{'}$ $y_4{'}$ $p_3{'}$ $x_3{'}$ $y_3{'}$ p_7 x_7 y_7 l_3)
from
‹*on-curve a b* p_1› ‹p_1 = *Point* x_1 y_1›
‹*on-curve a b* p_2› ‹p_2 = *Point* x_2 y_2›
have
$x_1 \in$ *carrier R* $y_1 \in$ *carrier R* **and** *y1*: y_1 [^] (*2*::*nat*) = x_1 [^] (*3*::*nat*) ⊕ $a \otimes x_1 \oplus b$ **and**
$x_2 \in$ *carrier R* $y_2 \in$ *carrier R* **and** *y2*: y_2 [^] (*2*::*nat*) = x_2 [^] (*3*::*nat*) ⊕ $a \otimes x_2 \oplus b$
by (*simp-all add*: *on-curve-def*)
from
‹$p_5{'}$ = *Point* $x_5{'}$ $y_5{'}$›
‹$p_5{'}$ = *Point* x_5 y_5›
‹$p_4{'}$ = *Point* $x_4{'}$ $y_4{'}$›
‹$p_4{'}$ = *Point* x_4 y_4›
‹$p_3{'}$ = *Point* $x_2{'}$ $y_2{'}$›
‹$p_3{'}$ = *Point* x_2 y_2›
‹$p_3{'}$ = *Point* $x_3{'}$ $y_3{'}$›
‹p_1 = *Point* $x_1{'}$ $y_1{'}$›
‹p_1 = *Point* x_1 y_1›
have *ps*:
$x_5{'} = x_5$ $y_5{'} = y_5$
$x_4{'} = x_4$ $y_4{'} = y_4$ $x_3{'} = x_2$ $y_3{'} = y_2$ $x_2{'} = x_2$ $y_2{'} = y_2$
$x_1{'} = x_1$ $y_1{'} = y_1$
by *simp-all*
show *?case*
apply (*simp add*: ‹p_6 = *Point* x_6 y_6› ‹p_7 = *Point* x_7 y_7›)
apply (*simp only*: *ps*
‹$x_7 = l_3$ [^] *2* ⊖ $x_4{'}$ ⊖ $x_3{'}$›
‹y_7 = ⊖ $y_4{'}$ ⊖ $l_3 \otimes (x_7 \ominus x_4{'})$›
‹$l_3 = (y_3{'} \ominus y_4{'}) \oslash (x_3{'} \ominus x_4{'})$›
‹$x_6 = l_2$ [^] *2* ⊖ $x_1{'}$ ⊖ $x_5{'}$›
‹y_6 = ⊖ $y_1{'}$ ⊖ $l_2 \otimes (x_6 \ominus x_1{'})$›
‹$l_2 = (y_5{'} \ominus y_1{'}) \oslash (x_5{'} \ominus x_1{'})$›
‹$x_5 = l_1$ [^] *2* ⊖ «*2*» ⊗ $x_2{'}$›
‹y_5 = ⊖ $y_2{'}$ ⊖ $l_1 \otimes (x_5 \ominus x_2{'})$›
‹$l_1 = («3» \otimes x_2{'}$ [^] *2* ⊕ $a) \oslash («2» \otimes y_2{'})$›

‹$x_4 = l\ [\widehat{}]\ 2 \ominus x_1 \ominus x_2$›
‹$y_4 = \ominus\ y_1 \ominus l \otimes (x_4 \ominus x_1)$›
‹$l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1)$›)
apply (*rule conjI*)
apply (*field y1 y2*)
apply (*intro conjI*)
apply (*simp add*: *integral-iff* [*OF* - ‹$y_2' \neq \mathbf{0}$›] ‹$y_2' \neq \mathbf{0}$› [*simplified* ‹$y_2' = y_2$›])
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*rule notE* [*OF* ‹$x_1' \neq x_5'$›] [*simplified*
‹$x_5 = l_1\ [\widehat{}]\ 2 \ominus$ «2» $\otimes\ x_2'$›
‹$l_1 = ($«3» $\otimes\ x_2'\ [\widehat{}]\ 2 \oplus a) \oslash ($«2» $\otimes\ y_2')$›
‹$x_1' = x_1$› ‹$x_2' = x_2$› ‹$y_2' = y_2$› ‹$x_5' = x_5$›]])
apply (*rule sym*)
apply (*field y1 y2*)
apply (*simp add*: *integral-iff* [*OF* - ‹$y_2' \neq \mathbf{0}$›] ‹$y_2' \neq \mathbf{0}$› [*simplified* ‹$y_2' = y_2$›])
apply (*simp add*: *eq-diff0* ‹$x_2 \in$ *carrier R*› ‹$x_1 \in$ *carrier R*› *not-sym* [*OF* ‹$x_1 \neq x_2$›])
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*rule notE* [*OF* ‹$x_4' \neq x_3'$›] [*simplified*
‹$x_4 = l\ [\widehat{}]\ 2 \ominus x_1 \ominus x_2$›
‹$l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1)$›
‹$x_4' = x_4$› ‹$x_3' = x_2$›]])
apply (*rule sym*)
apply (*field y1 y2*)
apply (*simp add*: *eq-diff0* ‹$x_2 \in$ *carrier R*› ‹$x_1 \in$ *carrier R*› *not-sym* [*OF* ‹$x_1 \neq x_2$›])
apply (*field y1 y2*)
apply (*intro conjI*)
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*rule notE* [*OF* ‹$x_1' \neq x_5'$›] [*simplified*
‹$x_5 = l_1\ [\widehat{}]\ 2 \ominus$ «2» $\otimes\ x_2'$›
‹$l_1 = ($«3» $\otimes\ x_2'\ [\widehat{}]\ 2 \oplus a) \oslash ($«2» $\otimes\ y_2')$›
‹$x_1' = x_1$› ‹$x_2' = x_2$› ‹$y_2' = y_2$› ‹$x_5' = x_5$›]])
apply (*rule sym*)
apply (*field y1 y2*)
apply (*simp add*: *integral-iff* [*OF* - ‹$y_2' \neq \mathbf{0}$›] ‹$y_2' \neq \mathbf{0}$› [*simplified* ‹$y_2' = y_2$›])
apply (*simp add*: *integral-iff* [*OF* - ‹$y_2' \neq \mathbf{0}$›] ‹$y_2' \neq \mathbf{0}$› [*simplified* ‹$y_2' = y_2$›])
apply (*rule notI*)
apply (*ring* (*prems*) *y1 y2*)
apply (*rule notE* [*OF* ‹$x_4' \neq x_3'$›] [*simplified*
‹$x_4 = l\ [\widehat{}]\ 2 \ominus x_1 \ominus x_2$›
‹$l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1)$›

```
          ‹x4' = x4› ‹x3' = x2›]])
        apply (rule sym)
        apply (field y1 y2)
         apply (simp-all add: eq-diff0 ‹x2 ∈ carrier R› ‹x1 ∈ carrier R› not-sym
[OF ‹x1 ≠ x2›])
        done
      qed
    qed
  next
    case (Gen p3 x3 y3 p5 x5 y5 p6 x6 y6 l1)
    then show ?case by (simp add: is-tangent-def)
  qed
qed

lemma spec3-assoc:
  assumes a: a ∈ carrier R
  and b: b ∈ carrier R
  and p1: on-curve a b p1
  and p2: on-curve a b p2
  and p3: on-curve a b p3
  and is-generic p1 p2
  and is-tangent p2 p3
  and is-generic (add a p1 p2) p3
  and is-tangent p1 (add a p2 p3)
  shows add a p1 (add a p2 p3) = add a (add a p1 p2) p3
  using a b p1 p2 assms
proof (induct rule: add-case)
  case InfL
  then show ?case by (simp add: is-generic-def)
next
  case InfR
  then show ?case by (simp add: is-generic-def)
next
  case Opp
  then show ?case by (simp add: is-generic-def)
next
  case Tan
  then show ?case by (simp add: is-generic-def)
next
  case (Gen p1 x1 y1 p2 x2 y2 p4 x4 y4 l)
  with a b ‹on-curve a b p2› ‹on-curve a b p3›
  show ?case
  proof (induct rule: add-case)
    case InfL
    then show ?case by (simp add: is-generic-def)
  next
    case InfR
    then show ?case by (simp add: is-generic-def)
  next
```

case *Opp*
then show *?case* **by** (*simp add*: *is-tangent-def opp-opp*)
next
case (*Tan* p_2 $x_2{'}$ $y_2{'}$ p_5 x_5 y_5 l_1)
from *a b* ‹*on-curve a b* p_2› ‹p_5 = *add a* p_2 p_2›
have *on-curve a b* p_5 **by** (*simp add*: *add-closed*)
with *a b* ‹*on-curve a b* p_1› **show** *?case* **using** *Tan*
proof (*induct rule*: *add-case*)
case *InfL*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case *InfR*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case *Opp*
then show *?case* **by** (*simp add*: *is-tangent-def opp-opp*)
next
case (*Tan* p_1 $x_1{'}$ $y_1{'}$ p_6 x_6 y_6 l_2)
from *a b* ‹*on-curve a b* p_1› ‹*on-curve a b* p_2› ‹p_4 = *add a* p_1 p_2›
have *on-curve a b* p_4 **by** (*simp add*: *add-closed*)
with *a b* **show** *?case* **using** ‹*on-curve a b* p_2› *Tan*
proof (*induct rule*: *add-case*)
case *InfL*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case *InfR*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case (*Opp p*)
from ‹*is-generic p* (*opp p*)› ‹*on-curve a b p*›
show *?case* **by** (*simp add*: *is-generic-def opp-opp*)
next
case *Tan*
then show *?case* **by** (*simp add*: *is-generic-def*)
next
case (*Gen* $p_4{'}$ $x_4{'}$ $y_4{'}$ $p_2{'}$ $x_2{''}$ $y_2{''}$ p_7 x_7 y_7 l_3)
from
‹*on-curve a b* p_1› ‹p_1 = *Point* x_1 y_1›
‹*on-curve a b* p_2› ‹p_2 = *Point* x_2 y_2›
have
$x_1 \in$ *carrier R* $y_1 \in$ *carrier R* **and** *y1*: y_1 [^] (*2*::*nat*) = x_1 [^] (*3*::*nat*) $\oplus$ $a \otimes x_1 \oplus b$ **and**
$x_2 \in$ *carrier R* $y_2 \in$ *carrier R* **and** *y2*: y_2 [^] (*2*::*nat*) = x_2 [^] (*3*::*nat*) $\oplus$ $a \otimes x_2 \oplus b$
by (*simp-all add*: *on-curve-def*)
from
‹$p_4{'}$ = *Point* $x_4{'}$ $y_4{'}$›
‹$p_4{'}$ = *Point* x_4 y_4›
‹$p_2{'}$ = *Point* $x_2{'}$ $y_2{'}$›

$\langle p_2' = \mathit{Point}\ x_2\ y_2\rangle$
$\langle p_2' = \mathit{Point}\ x_2''\ y_2''\rangle$
$\langle p_1 = \mathit{Point}\ x_1'\ y_1'\rangle$
$\langle p_1 = \mathit{Point}\ x_1\ y_1\rangle$
$\langle p_1 = \mathit{Point}\ x_5\ y_5\rangle$
have *ps*:
$x_4' = x_4\ y_4' = y_4\ x_2' = x_2\ y_2' = y_2\ x_2'' = x_2\ y_2'' = y_2$
$x_1' = x_5\ y_1' = y_5\ x_1 = x_5\ y_1 = y_5$
by *simp-all*
note *qs* =
$\langle x_7 = l_3\ [\widehat{}]\ 2 \ominus x_4' \ominus x_2''\rangle$
$\langle y_7 = \ominus\ y_4' \ominus l_3 \otimes (x_7 \ominus x_4')\rangle$
$\langle l_3 = (y_2'' \ominus y_4') \oslash (x_2'' \ominus x_4')\rangle$
$\langle x_6 = l_2\ [\widehat{}]\ 2 \ominus \langle\!\langle 2\rangle\!\rangle \otimes x_1'\rangle$
$\langle y_6 = \ominus\ y_1' \ominus l_2 \otimes (x_6 \ominus x_1')\rangle$
$\langle x_5 = l_1\ [\widehat{}]\ 2 \ominus \langle\!\langle 2\rangle\!\rangle \otimes x_2'\rangle$
$\langle y_5 = \ominus\ y_2' \ominus l_1 \otimes (x_5 \ominus x_2')\rangle$
$\langle l_1 = (\langle\!\langle 3\rangle\!\rangle \otimes x_2'\ [\widehat{}]\ 2 \oplus a) \oslash (\langle\!\langle 2\rangle\!\rangle \otimes y_2')\rangle$
$\langle l_2 = (\langle\!\langle 3\rangle\!\rangle \otimes x_1'\ [\widehat{}]\ 2 \oplus a) \oslash (\langle\!\langle 2\rangle\!\rangle \otimes y_1')\rangle$
$\langle x_4 = l\ [\widehat{}]\ 2 \ominus x_1 \ominus x_2\rangle$
$\langle y_4 = \ominus\ y_1 \ominus l \otimes (x_4 \ominus x_1)\rangle$
$\langle l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1)\rangle$
from $\langle y_2 \in \mathit{carrier}\ R\rangle$ $\langle y_2' \neq \mathbf{0}\rangle$ $\langle y_2' = y_2\rangle$
have $\langle\!\langle 2\rangle\!\rangle \otimes y_2 \neq \mathbf{0}$ **by** (*simp add*: *integral-iff*)
show *?case*
apply (*simp add*: $\langle p_6 = \mathit{Point}\ x_6\ y_6\rangle$ $\langle p_7 = \mathit{Point}\ x_7\ y_7\rangle$)
apply (*simp only*: *ps qs*)
apply (*rule conjI*)
apply (*field y2*)
apply (*intro conjI*)
apply (*rule notI*)
apply (*ring* (*prems*))
apply (*rule notE* [*OF* $\langle y_1' \neq \mathbf{0}\rangle$])
apply (*simp only*: *ps qs*)
apply *field*
apply (*rule* $\langle\langle\!\langle 2\rangle\!\rangle \otimes y_2 \neq \mathbf{0}\rangle$)
apply (*rule notI*)
apply (*ring* (*prems*))
apply (*rule notE* [*OF* $\langle x_1 \neq x_2\rangle$])
apply (*rule sym*)
apply (*simp only*: *ps qs*)
apply *field*
apply (*rule* $\langle\langle\!\langle 2\rangle\!\rangle \otimes y_2 \neq \mathbf{0}\rangle$)
apply (*rule* $\langle\langle\!\langle 2\rangle\!\rangle \otimes y_2 \neq \mathbf{0}\rangle$)
apply (*rule notI*)
apply (*ring* (*prems*))
apply (*rule notE* [*OF* $\langle x_4' \neq x_2''\rangle$])
apply (*rule sym*)
apply (*simp only*: *ps qs*)

apply *field*
apply (*intro conjI*)
apply (*rule* ‹«2» ⊗ $y_2 \neq \mathbf{0}$›)
apply (*erule thin-rl*)
apply (*rule notI*)
apply (*ring* (*prems*))
apply (*rule notE* [*OF* ‹$x_1 \neq x_2$›])
apply (*rule sym*)
apply (*simp only*: *ps qs*)
apply *field*
apply (*rule* ‹«2» ⊗ $y_2 \neq \mathbf{0}$›)
apply (*field y2*)
apply (*intro conjI*)
apply (*rule notI*)
apply (*ring* (*prems*))
apply (*rule notE* [*OF* ‹$y_1' \neq \mathbf{0}$›])
apply (*simp only*: *ps qs*)
apply *field*
apply (*rule* ‹«2» ⊗ $y_2 \neq \mathbf{0}$›)
apply (*rule notI*)
apply (*ring* (*prems*))
apply (*rule notE* [*OF* ‹$x_4' \neq x_2''$›])
apply (*rule sym*)
apply (*simp only*: *ps qs*)
apply *field*
apply (*erule thin-rl*)
apply (*rule conjI*)
apply (*rule* ‹«2» ⊗ $y_2 \neq \mathbf{0}$›)
apply (*rule notI*)
apply (*ring* (*prems*))
apply (*rule notE* [*OF* ‹$x_1 \neq x_2$›])
apply (*rule sym*)
apply (*simp only*: *ps qs*)
apply *field*
apply (*rule* ‹«2» ⊗ $y_2 \neq \mathbf{0}$›)
apply (*rule* ‹«2» ⊗ $y_2 \neq \mathbf{0}$›)
apply (*rule notI*)
apply (*ring* (*prems*))
apply (*rule notE* [*OF* ‹$x_1 \neq x_2$›])
apply (*rule sym*)
apply (*simp only*: *ps qs*)
apply *field*
apply (*rule* ‹«2» ⊗ $y_2 \neq \mathbf{0}$›)
done
qed
next
case *Gen*
then show *?case* **by** (*simp add*: *is-tangent-def*)
qed

```
  next
    case Gen
    then show ?case by (simp add: is-tangent-def)
  qed
qed

lemma add-0-l: add a Infinity p = p
  by (simp add: add-def)

lemma add-0-r: add a p Infinity = p
  by (simp add: add-def split: point.split)

lemma add-opp: on-curve a b p ⟹ add a p (opp p) = Infinity
  by (simp add: add-def opp-def on-curve-def split: point.split-asm)

lemma add-comm:
  assumes a ∈ carrier R b ∈ carrier R on-curve a b p₁ on-curve a b p₂
  shows add a p₁ p₂ = add a p₂ p₁
proof (cases p₁)
  case Infinity
  then show ?thesis by (simp add: add-0-l add-0-r)
next
  case (Point x₁ y₁)
  note Point' = this
  with ‹on-curve a b p₁› 
  have x₁ ∈ carrier R y₁ ∈ carrier R
    and y1: y₁ [^] (2::nat) = x₁ [^] (3::nat) ⊕ a ⊗ x₁ ⊕ b
    by (simp-all add: on-curve-def)
  show ?thesis
  proof (cases p₂)
    case Infinity
    then show ?thesis by (simp add: add-0-l add-0-r)
  next
    case (Point x₂ y₂)
    with ‹on-curve a b p₂› have x₂ ∈ carrier R y₂ ∈ carrier R
      and y2: y₂ [^] (2::nat) = x₂ [^] (3::nat) ⊕ a ⊗ x₂ ⊕ b
      by (simp-all add: on-curve-def)
    show ?thesis
    proof (cases x₁ = x₂)
      case True
      show ?thesis
      proof (cases y₁ = ⊖ y₂)
        case True
        with Point Point' ‹x₁ = x₂› ‹y₂ ∈ carrier R› show ?thesis
          by (simp add: add-def)
      next
        case False
        with y1 y2 [symmetric] ‹y₁ ∈ carrier R› ‹y₂ ∈ carrier R› ‹x₁ = x₂› Point
Point'
```

```
      show ?thesis
        by (simp add: power2-eq-square square-eq-iff)
    qed
  next
    case False
    with Point Point' show ?thesis
      apply (simp add: add-def Let-def)
      apply (rule conjI)
      apply field
      apply (cut-tac ‹x1 ∈ carrier R› ‹x2 ∈ carrier R›)
      apply (simp add: eq-diff0)
      apply field
      apply (cut-tac ‹x1 ∈ carrier R› ‹x2 ∈ carrier R›)
      apply (simp add: eq-diff0)
      done
  qed
 qed
qed

lemma uniq-opp:
 assumes on-curve a b p2
 and add a p1 p2 = Infinity
 shows p2 = opp p1
 using assms
 by (auto simp add: on-curve-def add-def opp-def Let-def
   split: point.split-asm if-split-asm)

lemma uniq-zero:
 assumes a: a ∈ carrier R
 and b: b ∈ carrier R
 and ab: nonsingular a b
 and p1: on-curve a b p1
 and p2: on-curve a b p2
 and add: add a p1 p2 = p2
 shows p1 = Infinity
 using a b p1 p2 assms
proof (induct rule: add-case)
 case InfL
 show ?case ..
next
 case InfR
 then show ?case by simp
next
 case Opp
 then show ?case by (simp add: opp-def split: point.split-asm)
next
 case (Tan p1 x1 y1 p2 x2 y2 l)
 from ‹on-curve a b p1› ‹p1 = Point x1 y1›
 have x1 ∈ carrier R y1 ∈ carrier R by (simp-all add: on-curve-def)
```

with a ‹$l = (\text{«}3\text{»} \otimes x_1\ [\widehat{}]\ 2 \oplus a) \oslash (\text{«}2\text{»} \otimes y_1)$› ‹$y_1 \neq \mathbf{0}$›
have $l \in$ *carrier R* **by** (*simp add*: *integral-iff*)
from ‹$p_1 =$ *Point* $x_1\ y_1$› ‹$p_2 =$ *Point* $x_2\ y_2$› ‹$p_2 = p_1$›
have $x_2 = x_1\ y_2 = y_1$ **by** *simp-all*
with ‹$x_1 \in$ *carrier R*› ‹$y_1 \in$ *carrier R*› ‹$l \in$ *carrier R*› ‹$y_2 = \ominus\ y_1 \ominus l \otimes (x_2 \ominus x_1)$› ‹$y_1 \neq \mathbf{0}$›
have $\ominus\ y_1 = y_1$ **by** (*simp add*: *r-neg minus-eq*)
with ‹$y_1 \in$ *carrier R*› ‹$y_1 \neq \mathbf{0}$›
show *?case* **by** (*simp add*: *neg-equal-zero*)
next
case (*Gen* $p_1\ x_1\ y_1\ p_2\ x_2\ y_2\ p_3\ x_3\ y_3\ l$)
then have $x_1 \in$ *carrier R* $y_1 \in$ *carrier R* $x_2 \in$ *carrier R* $y_2 \in$ *carrier R*
and *y1*: $y_1\ [\widehat{}]\ (2{::}nat) = x_1\ [\widehat{}]\ (3{::}nat) \oplus a \otimes x_1 \oplus b$
and *y2*: $y_2\ [\widehat{}]\ (2{::}nat) = x_2\ [\widehat{}]\ (3{::}nat) \oplus a \otimes x_2 \oplus b$
by (*simp-all add*: *on-curve-def*)
with ‹$l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1)$› ‹$x_1 \neq x_2$›
have $l \in$ *carrier R* **by** (*simp add*: *eq-diff0*)
from ‹$p_3 = p_2$› ‹$p_2 =$ *Point* $x_2\ y_2$› ‹$p_3 =$ *Point* $x_3\ y_3$›
have *ps*: $x_3 = x_2\ y_3 = y_2$ **by** *simp-all*
with ‹$y_3 = \ominus\ y_1 \ominus l \otimes (x_3 \ominus x_1)$›
have $y_2 = \ominus\ y_1 \ominus l \otimes (x_2 \ominus x_1)$ **by** *simp*
also from ‹$l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1)$› ‹$x_1 \neq x_2$›
‹$x_1 \in$ *carrier R*› ‹$x_2 \in$ *carrier R*› ‹$y_1 \in$ *carrier R*› ‹$y_2 \in$ *carrier R*›
have $l \otimes (x_2 \ominus x_1) = y_2 \ominus y_1$
by (*simp add*: *m-div-def m-assoc eq-diff0*)
also from ‹$y_1 \in$ *carrier R*› ‹$y_2 \in$ *carrier R*›
have $\ominus\ y_1 \ominus (y_2 \ominus y_1) = (\ominus\ y_1 \oplus y_1) \oplus \ominus\ y_2$
by (*simp add*: *minus-eq minus-add a-ac*)
finally have $y_2 = \mathbf{0}$ **using** ‹$y_1 \in$ *carrier R*› ‹$y_2 \in$ *carrier R*›
by (*simp add*: *l-neg equal-neg-zero*)
with ‹$p_2 =$ *Point* $x_2\ y_2$› ‹*on-curve a b* p_2›
‹$a \in$ *carrier R*› ‹$b \in$ *carrier R*› ‹$x_2 \in$ *carrier R*›
have *x2*: $x_2\ [\widehat{}]\ (3{::}nat) = \ominus\ (a \otimes x_2 \oplus b)$
by (*simp add*: *on-curve-def nat-pow-zero eq-neg-iff-add-eq-0 a-assoc*)
from ‹$x_3 = l\ [\widehat{}]\ 2 \ominus x_1 \ominus x_2$› ‹$x_3 = x_2$›
have $l\ [\widehat{}]\ (2{::}nat) \ominus x_1 \ominus x_2 \ominus x_2 = x_2 \ominus x_2$ **by** *simp*
with ‹$x_1 \in$ *carrier R*› ‹$x_2 \in$ *carrier R*› ‹$l \in$ *carrier R*›
have $l\ [\widehat{}]\ (2{::}nat) \ominus x_1 \ominus \text{«}2\text{»} \otimes x_2 = \mathbf{0}$
by (*simp add*: *of-int-2 l-distr minus-eq a-ac minus-add r-neg*)
then have $x_2 \otimes (l\ [\widehat{}]\ (2{::}nat) \ominus x_1 \ominus \text{«}2\text{»} \otimes x_2) = x_2 \otimes \mathbf{0}$ **by** *simp*
then have $(x_2 \ominus x_1) \otimes (\text{«}2\text{»} \otimes a \otimes x_2 \oplus \text{«}3\text{»} \otimes b) = \mathbf{0}$
apply (*simp add*: ‹$l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1)$› ‹$y_2 = \mathbf{0}$›)
apply (*field* (*prems*) *y1 x2*)
apply (*ring y1 x2*)
apply (*simp add*: *eq-diff0* ‹$x_2 \in$ *carrier R*› ‹$x_1 \in$ *carrier R*› *not-sym* [*OF* ‹$x_1 \neq x_2$›])
done
with *not-sym* [*OF* ‹$x_1 \neq x_2$›]
‹$x_2 \in$ *carrier R*› ‹$x_1 \in$ *carrier R*› ‹$a \in$ *carrier R*› ‹$b \in$ *carrier R*›

```
  have «2» ⊗ a ⊗ x₂ ⊕ «3» ⊗ b = 0
    by (simp add: integral-iff eq-diff0)
  with ‹a ∈ carrier R› ‹b ∈ carrier R› ‹x₂ ∈ carrier R›
  have «2» ⊗ a ⊗ x₂ = ⊖ («3» ⊗ b)
    by (simp add: eq-neg-iff-add-eq-0)
  from y2 [symmetric] ‹y₂ = 0› ‹a ∈ carrier R›
  have ⊖ («2» ⊗ a) [^] (3::nat) ⊗ (x₂ [^] (3::nat) ⊕ a ⊗ x₂ ⊕ b) = 0
    by (simp add: nat-pow-zero)
  then have b ⊗ («4» ⊗ a [^] (3::nat) ⊕ «27» ⊗ b [^] (2::nat)) = 0
    apply (ring (prems) ‹«2» ⊗ a ⊗ x₂ = ⊖ («3» ⊗ b)›)
    apply (ring ‹«2» ⊗ a ⊗ x₂ = ⊖ («3» ⊗ b)›)
    done
  with ab a b have b = 0 by (simp add: nonsingular-def integral-iff)
  with ‹«2» ⊗ a ⊗ x₂ ⊕ «3» ⊗ b = 0› ab a b ‹x₂ ∈ carrier R›
  have x₂ = 0 by (simp add: nonsingular-def nat-pow-zero integral-iff)
  from ‹l [^] (2::nat) ⊖ x₁ ⊖ «2» ⊗ x₂ = 0›
  show ?case
    apply (simp add: ‹x₂ = 0› ‹y₂ = 0› ‹l = (y₂ ⊖ y₁) ⊘ (x₂ ⊖ x₁)›)
    apply (field (prems) y1 ‹b = 0›)
    apply (insert a b ab ‹x₁ ∈ carrier R› ‹b = 0› ‹x₁ ≠ x₂› ‹x₂ = 0›)
    apply (simp add: nonsingular-def nat-pow-zero integral-iff)
    apply (simp add: trans [OF eq-commute eq-neg-iff-add-eq-0])
    done
qed

lemma opp-add:
  assumes a: a ∈ carrier R
  and b: b ∈ carrier R
  and p₁: on-curve a b p₁
  and p₂: on-curve a b p₂
  shows opp (add a p₁ p₂) = add a (opp p₁) (opp p₂)
proof (cases p₁)
  case Infinity
  then show ?thesis by (simp add: add-def opp-def)
next
  case (Point x₁ y₁)
  show ?thesis
  proof (cases p₂)
    case Infinity
    with ‹p₁ = Point x₁ y₁› show ?thesis
      by (simp add: add-def opp-def)
  next
    case (Point x₂ y₂)
    with ‹p₁ = Point x₁ y₁› p₁ p₂
     have x₁ ∈ carrier R y₁ ∈ carrier R x₁ [^] (3::nat) ⊕ a ⊗ x₁ ⊕ b = y₁ [^]
(2::nat)
      x₂ ∈ carrier R y₂ ∈ carrier R x₂ [^] (3::nat) ⊕ a ⊗ x₂ ⊕ b = y₂ [^] (2::nat)
      by (simp-all add: on-curve-def)
    with Point ‹p₁ = Point x₁ y₁› show ?thesis
```

```
    apply (cases x1 = x2)
    apply (cases y1 = ⊖ y2)
    apply (simp add: add-def opp-def Let-def)
    apply (simp add: add-def opp-def Let-def neg-equal-swap)
    apply (rule conjI)
    apply field
    apply (auto simp add: integral-iff nat-pow-zero
      trans [OF eq-commute eq-neg-iff-add-eq-0])[1]
    apply field
    apply (auto simp add: integral-iff nat-pow-zero
      trans [OF eq-commute eq-neg-iff-add-eq-0])[1]
    apply (simp add: add-def opp-def Let-def)
    apply (rule conjI)
    apply field
    apply (simp add: eq-diff0)
    apply field
    apply (simp add: eq-diff0)
    done
  qed
qed

lemma compat-add-opp:
  assumes a: a ∈ carrier R
  and b: b ∈ carrier R
  and p1: on-curve a b p1
  and p2: on-curve a b p2
  and add a p1 p2 = add a p1 (opp p2)
  and p1 ≠ opp p1
  shows p2 = opp p2
  using a b p1 p2 assms
proof (induct rule: add-case)
  case InfL
  then show ?case by (simp add: add-0-l)
next
  case InfR
  then show ?case by (simp add: opp-def add-0-r)
next
  case (Opp p)
  then have add a p p = Infinity by (simp add: opp-opp)
  with ‹on-curve a b p› have p = opp p by (rule uniq-opp)
  with ‹p ≠ opp p› show ?case ..
next
  case (Tan p1 x1 y1 p2 x2 y2 l)
  then have add a p1 p1 = Infinity
    by (simp add: add-opp)
  with ‹on-curve a b p1› have p1 = opp p1 by (rule uniq-opp)
  with ‹p1 ≠ opp p1› show ?case ..
next
  case (Gen p1 x1 y1 p2 x2 y2 p3 x3 y3 l)
```

```
  then have x1 ∈ carrier R y1 ∈ carrier R x2 ∈ carrier R y2 ∈ carrier R
    by (simp-all add: on-curve-def)
  have «2» ⊗ «2» ≠ 0
    by (simp add: integral-iff)
  then have «4» ≠ 0 by (simp add: of-int-mult [symmetric])
  from Gen have ((⊖ y2 ⊖ y1) ⊘ (x2 ⊖ x1)) [^] (2::nat) ⊖ x1 ⊖ x2 =
    ((y2 ⊖ y1) ⊘ (x2 ⊖ x1)) [^] (2::nat) ⊖ x1 ⊖ x2
    by (simp add: add-def opp-def Let-def)
  then show ?case
    apply (field (prems))
    apply (insert ‹y1 ∈ carrier R› ‹y2 ∈ carrier R› ‹p1 ≠ opp p1›
      ‹p1 = Point x1 y1› ‹p2 = Point x2 y2› ‹«4» ≠ 0›)[1]
    apply (simp add: integral-iff opp-def eq-neg-iff-add-eq-0 mult2)
    apply (insert ‹x1 ∈ carrier R› ‹x2 ∈ carrier R› ‹x1 ≠ x2›)
    apply (simp add: eq-diff0)
    done
qed

lemma compat-add-triple:
  assumes a: a ∈ carrier R
  and b: b ∈ carrier R
  and ab: nonsingular a b
  and p: on-curve a b p
  and p ≠ opp p
  and add a p p ≠ opp p
  shows add a (add a p p) (opp p) = p
  using a b add-closed [OF a b p p] opp-closed [OF p] assms
proof (induct add a p p opp p rule: add-case)
  case InfL
  from ‹p ≠ opp p› uniq-opp [OF p ‹Infinity = add a p p› [symmetric]]
  show ?case ..
next
  case InfR
  then show ?case by (simp add: opp-def split: point.split-asm)
next
  case Opp
  then have opp (opp (add a p p)) = opp (opp p) by simp
  with ‹on-curve a b (add a p p)› ‹on-curve a b p›
  have add a p p = p by (simp add: opp-opp)
  with uniq-zero [OF a b ab p p] ‹p ≠ opp p›
  show ?case by (simp add: opp-def)
next
  case Tan
  then show ?case by simp
next
  case (Gen x1 y1 x2 y2 p3 x3 y3 l)
  with opp-closed [OF p]
  have  x2 ∈ carrier R y2 ∈ carrier R
    by (simp-all add: on-curve-def)
```

```
  from ‹opp p = Point x₂ y₂› p
  have p = Point x₂ (⊖ y₂)
    by (auto simp add: opp-def on-curve-def neg-equal-swap split: point.split-asm)
  with ‹add a p p = Point x₁ y₁› [symmetric]
  obtain l' where l':
    l' = («3» ⊗ x₂ [^] (2::nat) ⊕ a) ⊘ («2» ⊗ ⊖ y₂)
    and xy: x₁ = l' [^] (2::nat) ⊖ «2» ⊗ x₂
    y₁ = ⊖ (⊖ y₂) ⊖ l' ⊗ (x₁ ⊖ x₂)
    and y2: ⊖ y₂ ≠ ⊖ (⊖ y₂)
    by (simp add: add-def Let-def split: if-split-asm)
  from l' ‹x₂ ∈ carrier R› ‹y₂ ∈ carrier R› a y2
  have l' ∈ carrier R by (simp add: neg-equal-zero neg-equal-swap integral-iff)
  have x₃ = x₂
    apply (simp add: xy
      ‹l = (y₂ ⊖ y₁) ⊘ (x₂ ⊖ x₁)› ‹x₃ = l [^] 2 ⊖ x₁ ⊖ x₂›)
    apply field
    apply (insert ‹x₁ ≠ x₂› ‹x₂ ∈ carrier R› ‹l' ∈ carrier R›)
    apply (simp add: xy eq-diff0)
    done
  then have p₃ = p ∨ p₃ = opp p
    by (rule curve-elt-opp [OF ‹p₃ = Point x₃ y₃› ‹p = Point x₂ (⊖ y₂)›
      add-closed [OF a b add-closed [OF a b p p] opp-closed [OF p],
        folded ‹p₃ = add a (add a p p) (opp p)›]
     ‹on-curve a b p›])
  then show ?case
  proof
    assume p₃ = p
    with ‹p₃ = add a (add a p p) (opp p)›
    show ?thesis by simp
  next
    assume p₃ = opp p
    with ‹p₃ = add a (add a p p) (opp p)›
    have add a (add a p p) (opp p) = opp p by simp
    with a b ab add-closed [OF a b p p] opp-closed [OF p]
    have add a p p = Infinity by (rule uniq-zero)
    with ‹add a p p = Point x₁ y₁› show ?thesis by simp
  qed
qed

lemma add-opp-double-opp:
  assumes a: a ∈ carrier R
  and b: b ∈ carrier R
  and ab: nonsingular a b
  and p₁: on-curve a b p₁
  and p₂: on-curve a b p₂
  and add a p₁ p₂ = opp p₁
  shows p₂ = add a (opp p₁) (opp p₁)
proof (cases p₁ = opp p₁)
  case True
```

```
  with assms have add a p₂ p₁ = p₁ by (simp add: add-comm)
  with a b ab p₂ p₁ have p₂ = Infinity by (rule uniq-zero)
  also from ‹on-curve a b p₁› have ... = add a p₁ (opp p₁)
    by (simp add: add-opp)
  also from True have ... = add a (opp p₁) (opp p₁) by simp
  finally show ?thesis .
next
  case False
  from a b p₁ p₂ False assms show ?thesis
  proof (induct rule: add-case)
    case InfL
    then show ?case by simp
  next
    case InfR
    then show ?case by simp
  next
    case Opp
    then show ?case by (simp add: add-0-l)
  next
    case (Tan p₁ x₁ y₁ p₂ x₂ y₂ l)
    from ‹p₂ = opp p₁› ‹on-curve a b p₁›
    have p₁ = opp p₂ by (simp add: opp-opp)
    also note ‹p₂ = add a p₁ p₁›
    finally show ?case using a b ‹on-curve a b p₁›
      by (simp add: opp-add)
  next
    case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l)
    from ‹on-curve a b p₁› ‹p₁ = Point x₁ y₁›
    have x₁ ∈ carrier R y₁ ∈ carrier R
      and y₁: y₁ [^] (2::nat) = x₁ [^] (3::nat) ⊕ a ⊗ x₁ ⊕ b
      by (simp-all add: on-curve-def)
    from ‹on-curve a b p₂› ‹p₂ = Point x₂ y₂›
    have x₂ ∈ carrier R y₂ ∈ carrier R
      and y₂: y₂ [^] (2::nat) = x₂ [^] (3::nat) ⊕ a ⊗ x₂ ⊕ b
      by (simp-all add: on-curve-def)
    from ‹p₁ = Point x₁ y₁› ‹p₁ ≠ opp p₁› ‹y₁ ∈ carrier R›
    have y₁ ≠ 𝟬
      by (simp add: opp-Point integral-iff equal-neg-zero)
    from Gen have x₁ = ((y₂ ⊖ y₁) ⊘ (x₂ ⊖ x₁)) [^] (2::nat) ⊖ x₁ ⊖ x₂
      by (simp add: opp-Point)
    then have «2» ⊗ y₂ ⊗ y₁ = a ⊗ x₂ ⊕ «3» ⊗ x₂ ⊗ x₁ [^] (2::nat) ⊕ a ⊗ x₁
⊖
      x₁ [^] (3::nat) ⊕ «2» ⊗ b
      apply (field (prems) y₁ y₂)
      apply (field y₁ y₂)
      apply simp
      apply (insert ‹x₁ ≠ x₂› ‹x₁ ∈ carrier R› ‹x₂ ∈ carrier R›)
      apply (simp add: eq-diff0)
      done
```

```
  then have (x2 ⊖ (((«3» ⊗ x1 [^] (2::nat) ⊕ a) ⊘ («2» ⊗ (⊖ y1))) [^] (2::nat)
⊖
    «2» ⊗ x1)) ⊗ (x2 ⊖ x1) [^] (2::nat) = 0
    apply (drule-tac f=λx. x [^] (2::nat) in arg-cong)
    apply (field (prems) y1 y2)
    apply (field y1 y2)
    apply (insert ‹y1 ≠ 0› ‹y1 ∈ carrier R›)
    apply (simp-all add: integral-iff neg-equal-swap)
    done
  with a ‹x1 ∈ carrier R› ‹y1 ∈ carrier R› ‹x2 ∈ carrier R›
    ‹y1 ≠ 0› ‹x1 ≠ x2›
  have x2 = ((«3» ⊗ x1 [^] (2::nat) ⊕ a) ⊘ («2» ⊗ (⊖ y1))) [^] (2::nat) ⊖
    «2» ⊗ x1
    by (simp add: integral-iff eq-diff0 neg-equal-swap)
  with ‹p2 = Point x2 y2› - ‹on-curve a b p2›
    add-closed [OF a b
      opp-closed [OF ‹on-curve a b p1›] opp-closed [OF ‹on-curve a b p1›]]
  have p2 = add a (opp p1) (opp p1) ∨ p2 = opp (add a (opp p1) (opp p1))
    apply (rule curve-elt-opp)
    apply (insert ‹y1 ∈ carrier R› ‹y1 ≠ 0›)
     apply (simp add: add-def opp-Point neg-equal-zero Let-def ‹p1 = Point x1
y1›)
    done
  then show ?case
  proof
    assume p2 = opp (add a (opp p1) (opp p1))
    with a b ‹on-curve a b p1›
    have p2 = add a p1 p1
      by (simp add: opp-add opp-opp opp-closed)
    show ?case
    proof (cases add a p1 p1 = opp p1)
      case True
      from a b ‹on-curve a b p1›
      show ?thesis
        apply (simp add: opp-add [symmetric] ‹p2 = add a p1 p1› True)
        apply (simp add: ‹p3 = add a p1 p2› [simplified ‹p3 = opp p1›])
        apply (simp add: ‹p2 = add a p1 p1› True add-opp)
        done
    next
      case False
      from a b ‹on-curve a b p1›
      have add a p1 (opp p2) = opp (add a (add a p1 p1) (opp p1))
        by (simp add: ‹p2 = add a p1 p1›
          opp-add add-closed opp-closed opp-opp add-comm)
      with a b ab ‹on-curve a b p1› ‹p1 ≠ opp p1› False
      have add a p1 (opp p2) = opp p1
        by (simp add: compat-add-triple)
      with ‹p3 = add a p1 p2› ‹p3 = opp p1›
      have add a p1 p2 = add a p1 (opp p2) by simp
```

```
      with a b ‹on-curve a b p₁› ‹on-curve a b p₂›
      have p₂ = opp p₂ using ‹p₁ ≠ opp p₁›
        by (rule compat-add-opp)
      with a b ‹on-curve a b p₁› ‹p₂ = add a p₁ p₁›
      show ?thesis by (simp add: opp-add)
    qed
  qed
 qed
qed

lemma cancel:
 assumes a: a ∈ carrier R
 and b: b ∈ carrier R
 and ab: nonsingular a b
 and p₁: on-curve a b p₁
 and p₂: on-curve a b p₂
 and p₃: on-curve a b p₃
 and eq: add a p₁ p₂ = add a p₁ p₃
 shows p₂ = p₃
 using a b p₁ p₂ p₁ p₂ eq
proof (induct rule: add-casew)
 case InfL
 then show ?case by (simp add: add-0-l)
next
 case (InfR p)
 with a b p₃ have add a p₃ p = p by (simp add: add-comm)
 with a b ab p₃ ‹on-curve a b p›
 show ?case by (rule uniq-zero [symmetric])
next
 case (Opp p)
 from p₃ ‹Infinity = add a p p₃› [symmetric]
 show ?case by (rule uniq-opp [symmetric])
next
 case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₄ x₄ y₄ l)
 from ‹on-curve a b p₁› ‹p₁ = Point x₁ y₁›
 have x₁ ∈ carrier R y₁ ∈ carrier R
   by (simp-all add: on-curve-def)
 from ‹on-curve a b p₂› ‹p₂ = Point x₂ y₂›
 have x₂ ∈ carrier R y₂ ∈ carrier R
   by (simp-all add: on-curve-def)
 from add-closed [OF a b ‹on-curve a b p₁› ‹on-curve a b p₂›]
   ‹p₄ = add a p₁ p₂› [symmetric] ‹p₄ = Point x₄ y₄›
 have x₄ ∈ carrier R y₄ ∈ carrier R
   by (simp-all add: on-curve-def)
 from ‹- ∨ -› a ‹p₁ ≠ opp p₂› ‹p₁ = Point x₁ y₁› ‹p₂ = Point x₂ y₂›
   ‹x₁ ∈ carrier R› ‹y₁ ∈ carrier R›
   ‹x₂ ∈ carrier R› ‹y₂ ∈ carrier R›
 have l ∈ carrier R
   by (auto simp add: opp-Point equal-neg-zero integral-iff eq-diff0)
```

from *a b* ‹*on-curve a b* p_1› p_3 ‹*on-curve a b* p_1› p_3 ‹$p_1 = Point\ x_1\ y_1$›
‹$p_4 = add\ a\ p_1\ p_2$› ‹$p_4 = add\ a\ p_1\ p_3$› ‹$p_1 \neq opp\ p_2$›
show *?case*
proof (*induct rule*: *add-casew*)
case *InfL*
then show *?case* **by** (*simp add*: *add-0-l*)
next
case (*InfR p*)
with *a b* ‹*on-curve a b* p_2›
have *add a* p_2 *p* = *p* **by** (*simp add*: *add-comm*)
with *a b ab* ‹*on-curve a b* p_2› ‹*on-curve a b p*›
show *?case* **by** (*rule uniq-zero*)
next
case (*Opp p*)
then have *add a p* p_2 = *Infinity* **by** *simp*
with ‹*on-curve a b* p_2› **show** *?case* **by** (*rule uniq-opp*)
next
case (*Gen* p_1 $x_1{'}$ $y_1{'}$ p_3 x_3 y_3 p_5 x_5 y_5 l')
from ‹*on-curve a b* p_3› ‹$p_3 = Point\ x_3\ y_3$›
have $x_3 \in carrier\ R$ $y_3 \in carrier\ R$
by (*simp-all add*: *on-curve-def*)
from ‹$x_1{'} = x_3 \wedge$ - $\vee$ ->› *a* ‹$p_1 \neq opp\ p_3$›
‹$p_1 = Point\ x_1\ y_1$› ‹$p_1 = Point\ x_1{'}\ y_1{'}$› ‹$p_3 = Point\ x_3\ y_3$›
‹$x_1 \in carrier\ R$› ‹$y_1 \in carrier\ R$›
‹$x_3 \in carrier\ R$› ‹$y_3 \in carrier\ R$›
have $l' \in carrier\ R$
by (*auto simp add*: *opp-Point equal-neg-zero integral-iff eq-diff0*)
from ‹$p_4 = p_5$› ‹$p_4 = Point\ x_4\ y_4$› ‹$p_5 = Point\ x_5\ y_5$›
‹$p_1 = Point\ x_1{'}\ y_1{'}$› ‹$p_1 = Point\ x_1\ y_1$›
‹$y_4 = \ominus\ y_1 \oplus l \otimes (x_4 \ominus x_1)$› ‹$y_5 = \ominus\ y_1{'} \oplus l' \otimes (x_5 \ominus x_1{'})$›
‹$x_1 \in carrier\ R$› ‹$y_1 \in carrier\ R$› ‹$x_4 \in carrier\ R$› ‹$l' \in carrier\ R$›
have $\mathbf{0} = \ominus\ y_1 \oplus l \otimes (x_4 \ominus x_1) \oplus (\ominus\ y_1 \oplus l' \otimes (x_4 \ominus x_1))$
by (*auto simp add*: *trans* [*OF eq-commute eq-diff0*])
with ‹$x_1 \in carrier\ R$› ‹$y_1 \in carrier\ R$› ‹$x_4 \in carrier\ R$›
‹$l \in carrier\ R$› ‹$l' \in carrier\ R$›
have $(l' \ominus l) \otimes (x_4 \ominus x_1) = \mathbf{0}$
apply *simp*
apply (*rule eq-diff0* [*THEN iffD1*])
apply *simp*
apply *simp*
apply *ring*
done
with ‹$x_1 \in carrier\ R$› ‹$x_4 \in carrier\ R$› ‹$l \in carrier\ R$› ‹$l' \in carrier\ R$›
have $l' = l \vee x_4 = x_1$
by (*simp add*: *integral-iff eq-diff0*)
then show *?case*
proof
assume $l' = l$
with ‹$p_4 = p_5$› ‹$p_4 = Point\ x_4\ y_4$› ‹$p_5 = Point\ x_5\ y_5$›

```
    ‹p₁ = Point x₁' y₁'› ‹p₁ = Point x₁ y₁›
    ‹x₄ = l [^] 2 ⊖ x₁ ⊖ x₂› ‹x₅ = l' [^] 2 ⊖ x₁' ⊖ x₃›
    ‹x₁ ∈ carrier R› ‹x₃ ∈ carrier R› ‹l ∈ carrier R›
   have 0 = l [^] (2::nat) ⊖ x₁ ⊖ x₂ ⊖ (l [^] (2::nat) ⊖ x₁ ⊖ x₃)
    by (simp add: trans [OF eq-commute eq-diff0])
   with ‹x₁ ∈ carrier R› ‹x₂ ∈ carrier R› ‹x₃ ∈ carrier R› ‹l ∈ carrier R›
   have x₂ = x₃
    apply (rule-tac eq-diff0 [THEN iffD1, THEN sym])
    apply simp-all
    apply (rule eq-diff0 [THEN iffD1])
    apply simp-all[2]
    apply ring
    done
   with ‹p₂ = Point x₂ y₂› ‹p₃ = Point x₃ y₃› ‹on-curve a b p₂› ‹on-curve a b
p₃›
   have p₂ = p₃ ∨ p₂ = opp p₃ by (rule curve-elt-opp)
   then show ?case
   proof
    assume p₂ = opp p₃
    with ‹on-curve a b p₃› have opp p₂ = p₃
     by (simp add: opp-opp)
    with ‹p₄ = p₅› ‹p₄ = add a p₁ p₂› ‹p₅ = add a p₁ p₃›
    have add a p₁ p₂ = add a p₁ (opp p₂) by simp
    show ?case
    proof (cases p₁ = opp p₁)
     case True
     with ‹p₁ ≠ opp p₂› ‹p₁ ≠ opp p₃›
     have p₁ ≠ p₂ p₁ ≠ p₃ by auto
     with ‹l' = l› ‹x₁ = x₂ ∧ -∨ -› ‹x₁' = x₃ ∧ - ∨ -›
      ‹p₁ = Point x₁ y₁› ‹p₁ = Point x₁' y₁'›
      ‹p₂ = Point x₂ y₂› ‹p₃ = Point x₃ y₃›
      ‹p₂ = opp p₃›
     have eq: (y₂ ⊖ y₁) ⊘ (x₂ ⊖ x₁) = (y₃ ⊖ y₁) ⊘ (x₂ ⊖ x₁) and x₁ ≠ x₂
      by (auto simp add: opp-Point)
     from eq have y₂ = y₃
      apply (field (prems))
      apply (rule eq-diff0 [THEN iffD1])
      apply (insert ‹x₁ ≠ x₂› ‹x₁ ∈ carrier R› ‹y₁ ∈ carrier R›
       ‹x₂ ∈ carrier R› ‹y₂ ∈ carrier R› ‹y₃ ∈ carrier R›)
      apply simp-all
      apply (erule subst)
      apply (rule eq-diff0 [THEN iffD1])
      apply simp-all
      apply ring
      apply (simp add: eq-diff0)
      done
     with ‹p₂ = opp p₃› ‹p₂ = Point x₂ y₂› ‹p₃ = Point x₃ y₃›
     show ?thesis by (simp add: opp-Point)
    next
```

```
        case False
        with a b ‹on-curve a b p₁› ‹on-curve a b p₂›
          ‹add a p₁ p₂ = add a p₁ (opp p₂)›
        have p₂ = opp p₂ by (rule compat-add-opp)
        with ‹opp p₂ = p₃› show ?thesis by simp
      qed
    qed
  next
    assume x₄ = x₁
    with ‹p₄ = Point x₄ y₄› [simplified ‹p₄ = add a p₁ p₂›]
      ‹p₁ = Point x₁ y₁›
      add-closed [OF a b ‹on-curve a b p₁› ‹on-curve a b p₂›]
      ‹on-curve a b p₁›
    have add a p₁ p₂ = p₁ ∨ add a p₁ p₂ = opp p₁ by (rule curve-elt-opp)
    then show ?case
    proof
      assume add a p₁ p₂ = p₁
      with a b ‹on-curve a b p₁› ‹on-curve a b p₂›
      have add a p₂ p₁ = p₁ by (simp add: add-comm)
      with a b ab ‹on-curve a b p₂› ‹on-curve a b p₁›
      have p₂ = Infinity by (rule uniq-zero)
      moreover from ‹add a p₁ p₂ = p₁›
        ‹p₄ = p₅› ‹p₄ = add a p₁ p₂› ‹p₅ = add a p₁ p₃›
        a b ‹on-curve a b p₁› ‹on-curve a b p₃›
      have add a p₃ p₁ = p₁ by (simp add: add-comm)
      with a b ab ‹on-curve a b p₃› ‹on-curve a b p₁›
      have p₃ = Infinity by (rule uniq-zero)
      ultimately show ?case by simp
    next
      assume add a p₁ p₂ = opp p₁
      with a b ab ‹on-curve a b p₁› ‹on-curve a b p₂›
      have p₂ = add a (opp p₁) (opp p₁) by (rule add-opp-double-opp)
      moreover from ‹add a p₁ p₂ = opp p₁›
        ‹p₄ = p₅› ‹p₄ = add a p₁ p₂› ‹p₅ = add a p₁ p₃›
      have add a p₁ p₃ = opp p₁ by simp
      with a b ab ‹on-curve a b p₁› ‹on-curve a b p₃›
      have p₃ = add a (opp p₁) (opp p₁) by (rule add-opp-double-opp)
      ultimately show ?case by simp
    qed
  qed
qed

lemma add-minus-id:
  assumes a: a ∈ carrier R
  and b: b ∈ carrier R
  and ab: nonsingular a b
  and p₁: on-curve a b p₁
  and p₂: on-curve a b p₂
```

```
  shows add a (add a p₁ p₂) (opp p₂) = p₁
proof (cases add a p₁ p₂ = opp p₂)
  case True
  then have add a (add a p₁ p₂) (opp p₂) = add a (opp p₂) (opp p₂)
    by simp
  also from a b p₁ p₂ True have add a p₂ p₁ = opp p₂
    by (simp add: add-comm)
  with a b ab p₂ p₁ have add a (opp p₂) (opp p₂) = p₁
    by (rule add-opp-double-opp [symmetric])
  finally show ?thesis .
next
  case False
  from a b p₁ p₂ p₁ p₂ False show ?thesis
  proof (induct rule: add-case)
    case InfL
    then show ?case by (simp add: add-opp)
  next
    case InfR
    show ?case by (simp add: add-0-r)
  next
    case Opp
    then show ?case by (simp add: opp-opp add-0-l)
  next
    case (Tan p₁ x₁ y₁ p₂ x₂ y₂ l)
    note a b ab ‹on-curve a b p₁›
    moreover from ‹on-curve a b p₁› ‹p₁ = Point x₁ y₁›
    have y₁ ∈ carrier R by (simp add: on-curve-def)
    with ‹y₁ ≠ 0› ‹p₁ = Point x₁ y₁›
    have p₁ ≠ opp p₁ by (simp add: opp-Point equal-neg-zero)
    moreover from ‹p₂ = add a p₁ p₁› ‹p₂ ≠ opp p₁›
    have add a p₁ p₁ ≠ opp p₁ by simp
    ultimately have add a (add a p₁ p₁) (opp p₁) = p₁
      by (rule compat-add-triple)
    with ‹p₂ = add a p₁ p₁› show ?case by simp
  next
    case (Gen p₁ x₁ y₁ p₂ x₂ y₂ p₃ x₃ y₃ l)
    from ‹p₃ = add a p₁ p₂› ‹on-curve a b p₂›
    have p₃ = add a p₁ (opp (opp p₂)) by (simp add: opp-opp)
    with a b
      add-closed [OF a b ‹on-curve a b p₁› ‹on-curve a b p₂›,
        folded ‹p₃ = add a p₁ p₂›]
      opp-closed [OF ‹on-curve a b p₂›]
      opp-closed [OF ‹on-curve a b p₂›]
      opp-opp [OF ‹on-curve a b p₂›]
      Gen
    show ?case
    proof (induct rule: add-case)
      case InfL
      then show ?case by simp
```

```
next
  case InfR
  then show ?case by (simp add: add-0-r)
next
  case (Opp p)
  from ‹on-curve a b p› ‹p = add a p₁ (opp (opp p))›
  have add a p₁ p = p by (simp add: opp-opp)
  with a b ab ‹on-curve a b p₁› ‹on-curve a b p›
  show ?case by (rule uniq-zero [symmetric])
next
  case Tan
  then show ?case by simp
next
  case (Gen p₄ x₄ y₄ p₅ x₅ y₅ p₆ x₆ y₆ l')
  from ‹on-curve a b p₁› ‹p₁ = Point x₁ y₁›
  have x₁ ∈ carrier R y₁ ∈ carrier R
    by (simp-all add: on-curve-def)
  from ‹on-curve a b p₂› ‹p₂ = Point x₂ y₂›
  have x₂ ∈ carrier R y₂ ∈ carrier R
    by (simp-all add: on-curve-def)
  from ‹on-curve a b p₅› ‹opp p₅ = p₂›
    ‹p₂ = Point x₂ y₂› ‹p₅ = Point x₅ y₅›
  have y₅ = ⊖ y₂ x₅ = x₂
    by (auto simp add: opp-Point on-curve-def)
  from ‹p₄ = Point x₃ y₃› ‹p₄ = Point x₄ y₄›
  have x₄ = x₃ y₄ = y₃ by simp-all
  from ‹x₄ ≠ x₅› show ?case
    apply (simp add:
      ‹y₅ = ⊖ y₂› ‹x₅ = x₂›
      ‹x₄ = x₃› ‹y₄ = y₃›
      ‹p₆ = Point x₆ y₆› ‹p₁ = Point x₁ y₁›
      ‹x₆ = l' [^] 2 ⊖ x₄ ⊖ x₅› ‹y₆ = ⊖ y₄ ⊖ l' ⊗ (x₆ ⊖ x₄)›
      ‹l' = (y₅ ⊖ y₄) ⊘ (x₅ ⊖ x₄)›
      ‹x₃ = l [^] 2 ⊖ x₁ ⊖ x₂› ‹y₃ = ⊖ y₁ ⊖ l ⊗ (x₃ ⊖ x₁)›
      ‹l = (y₂ ⊖ y₁) ⊘ (x₂ ⊖ x₁)›)
    apply (rule conjI)
    apply field
    apply (rule conjI)
    apply (rule notI)
    apply (erule notE)
    apply (ring (prems))
    apply (rule sym)
    apply field
    apply (simp-all add: eq-diff0 [OF ‹x₂ ∈ carrier R› ‹x₁ ∈ carrier R›]
      ‹x₁ ≠ x₂› [THEN not-sym])
    apply field
    apply (rule conjI)
    apply (simp add: eq-diff0 [OF ‹x₂ ∈ carrier R› ‹x₁ ∈ carrier R›]
      ‹x₁ ≠ x₂› [THEN not-sym])
```

```
      apply (rule notI)
      apply (erule notE)
      apply (ring (prems))
      apply (rule sym)
      apply field
      apply (simp add: eq-diff0 [OF ‹x₂ ∈ carrier R› ‹x₁ ∈ carrier R›]
        ‹x₁ ≠ x₂› [THEN not-sym])
      done
    qed
  qed
qed

lemma add-shift-minus:
  assumes a: a ∈ carrier R
  and b: b ∈ carrier R
  and ab: nonsingular a b
  and p₁: on-curve a b p₁
  and p₂: on-curve a b p₂
  and p₃: on-curve a b p₃
  and eq: add a p₁ p₂ = p₃
  shows p₁ = add a p₃ (opp p₂)
proof −
  note eq
  also from add-minus-id [OF a b ab p₃ opp-closed [OF p₂]] p₂
  have p₃ = add a (add a p₃ (opp p₂)) p₂ by (simp add: opp-opp)
  finally have add a p₂ p₁ = add a p₂ (add a p₃ (opp p₂))
    using a b p₁ p₂ p₃
    by (simp add: add-comm add-closed opp-closed)
  with a b ab p₂ p₁ add-closed [OF a b p₃ opp-closed [OF p₂]]
  show ?thesis by (rule cancel)
qed

lemma degen-assoc:
  assumes a: a ∈ carrier R
  and b: b ∈ carrier R
  and ab: nonsingular a b
  and p₁: on-curve a b p₁
  and p₂: on-curve a b p₂
  and p₃: on-curve a b p₃
  and H:
    (p₁ = Infinity ∨ p₂ = Infinity ∨ p₃ = Infinity) ∨
     (p₁ = opp p₂ ∨ p₂ = opp p₃) ∨
     (opp p₁ = add a p₂ p₃ ∨ opp p₃ = add a p₁ p₂)
  shows add a p₁ (add a p₂ p₃) = add a (add a p₁ p₂) p₃
  using H
proof (elim disjE)
  assume p₁ = Infinity
  then show ?thesis by (simp add: add-0-l)
next
```

assume $p_2 = \mathit{Infinity}$
then show *?thesis* **by** (*simp add*: *add-0-l add-0-r*)
next
assume $p_3 = \mathit{Infinity}$
then show *?thesis* **by** (*simp add*: *add-0-r*)
next
assume $p_1 = \mathit{opp}\ p_2$
from *a b* p_2 p_3
have $\mathit{add}\ a\ (\mathit{opp}\ p_2)\ (\mathit{add}\ a\ p_2\ p_3) = \mathit{add}\ a\ (\mathit{add}\ a\ p_3\ p_2)\ (\mathit{opp}\ p_2)$
by (*simp add*: *add-comm add-closed opp-closed*)
also from *a b ab* p_3 p_2 **have** $\ldots = p_3$ **by** (*rule add-minus-id*)
also have $\ldots = \mathit{add}\ a\ \mathit{Infinity}\ p_3$ **by** (*simp add*: *add-0-l*)
also from p_2 **have** $\ldots = \mathit{add}\ a\ (\mathit{add}\ a\ p_2\ (\mathit{opp}\ p_2))\ p_3$
by (*simp add*: *add-opp*)
also from *a b* p_2 **have** $\ldots = \mathit{add}\ a\ (\mathit{add}\ a\ (\mathit{opp}\ p_2)\ p_2)\ p_3$
by (*simp add*: *add-comm opp-closed*)
finally show *?thesis* **using** ‹$p_1 = \mathit{opp}\ p_2$› **by** *simp*
next
assume $p_2 = \mathit{opp}\ p_3$
from *a b* p_3
have $\mathit{add}\ a\ p_1\ (\mathit{add}\ a\ (\mathit{opp}\ p_3)\ p_3) = \mathit{add}\ a\ p_1\ (\mathit{add}\ a\ p_3\ (\mathit{opp}\ p_3))$
by (*simp add*: *add-comm opp-closed*)
also from *a b ab* p_1 p_3
have $\ldots = \mathit{add}\ a\ (\mathit{add}\ a\ p_1\ (\mathit{opp}\ p_3))\ (\mathit{opp}\ (\mathit{opp}\ p_3))$
by (*simp add*: *add-opp add-minus-id add-0-r opp-closed*)
finally show *?thesis* **using** p_3 ‹$p_2 = \mathit{opp}\ p_3$›
by (*simp add*: *opp-opp*)
next
assume *eq*: $\mathit{opp}\ p_1 = \mathit{add}\ a\ p_2\ p_3$
from *eq* [*symmetric*] p_1
have $\mathit{add}\ a\ p_1\ (\mathit{add}\ a\ p_2\ p_3) = \mathit{Infinity}$ **by** (*simp add*: *add-opp*)
also from p_3 **have** $\ldots = \mathit{add}\ a\ p_3\ (\mathit{opp}\ p_3)$ **by** (*simp add*: *add-opp*)
also from *a b* p_3 **have** $\ldots = \mathit{add}\ a\ (\mathit{opp}\ p_3)\ p_3$
by (*simp add*: *add-comm opp-closed*)
also from *a b ab* p_2 p_3
have $\ldots = \mathit{add}\ a\ (\mathit{add}\ a\ (\mathit{add}\ a\ (\mathit{opp}\ p_3)\ (\mathit{opp}\ p_2))\ (\mathit{opp}\ (\mathit{opp}\ p_2)))\ p_3$
by (*simp add*: *add-minus-id opp-closed*)
also from *a b* p_2 p_3
have $\ldots = \mathit{add}\ a\ (\mathit{add}\ a\ (\mathit{add}\ a\ (\mathit{opp}\ p_2)\ (\mathit{opp}\ p_3))\ p_2)\ p_3$
by (*simp add*: *add-comm opp-opp opp-closed*)
finally show *?thesis*
using *opp-add* [*OF a b* p_2 p_3] *eq* [*symmetric*] p_1
by (*simp add*: *opp-opp*)
next
assume *eq*: $\mathit{opp}\ p_3 = \mathit{add}\ a\ p_1\ p_2$
from *opp-add* [*OF a b* p_1 p_2] *eq* [*symmetric*] p_3
have $\mathit{add}\ a\ p_1\ (\mathit{add}\ a\ p_2\ p_3) = \mathit{add}\ a\ p_1\ (\mathit{add}\ a\ p_2\ (\mathit{add}\ a\ (\mathit{opp}\ p_1)\ (\mathit{opp}\ p_2)))$
by (*simp add*: *opp-opp*)
also from *a b* p_1 p_2

```
  have ... = add a p₁ (add a (add a (opp p₁) (opp p₂)) (opp (opp p₂)))
    by (simp add: add-comm opp-opp add-closed opp-closed)
  also from a b ab p₁ p₂ have ... = Infinity
    by (simp add: add-minus-id add-opp opp-closed)
  also from p₃ have ... = add a p₃ (opp p₃) by (simp add: add-opp)
  also from a b p₃ have ... = add a (opp p₃) p₃
    by (simp add: add-comm opp-closed)
  finally show ?thesis using eq [symmetric] by simp
qed

lemma spec4-assoc:
  assumes a: a ∈ carrier R
  and b: b ∈ carrier R
  and ab: nonsingular a b
  and p₁: on-curve a b p₁
  and p₂: on-curve a b p₂
  shows add a p₁ (add a p₂ p₂) = add a (add a p₁ p₂) p₂
proof (cases p₁ = Infinity)
  case True
  from a b ab p₁ p₂ p₂
  show ?thesis by (rule degen-assoc) (simp add: True)
next
  case False
  show ?thesis
  proof (cases p₂ = Infinity)
    case True
    from a b ab p₁ p₂ p₂
    show ?thesis by (rule degen-assoc) (simp add: True)
  next
    case False
    show ?thesis
    proof (cases p₂ = opp p₂)
      case True
      from a b ab p₁ p₂ p₂
      show ?thesis by (rule degen-assoc) (simp add: True [symmetric])
    next
      case False
      show ?thesis
      proof (cases p₁ = opp p₂)
        case True
        from a b ab p₁ p₂ p₂
        show ?thesis by (rule degen-assoc) (simp add: True)
      next
        case False
        show ?thesis
        proof (cases opp p₁ = add a p₂ p₂)
          case True
          from a b ab p₁ p₂ p₂
          show ?thesis by (rule degen-assoc) (simp add: True)
```

```
next
  case False
  show ?thesis
  proof (cases opp p₂ = add a p₁ p₂)
    case True
    from a b ab p₁ p₂ p₂
    show ?thesis by (rule degen-assoc) (simp add: True)
  next
    case False
    show ?thesis
    proof (cases p₁ = add a p₂ p₂)
      case True
      from a b p₁ p₂ ‹p₁ ≠ opp p₂› ‹p₂ ≠ opp p₂›
        ‹opp p₁ ≠ add a p₂ p₂› ‹opp p₂ ≠ add a p₁ p₂›
        ‹p₁ ≠ Infinity› ‹p₂ ≠ Infinity›
      show ?thesis
        apply (simp add: True)
        apply (rule spec3-assoc [OF a b])
        apply (simp-all add: is-generic-def is-tangent-def)
        apply (rule notI)
        apply (drule uniq-zero [OF a b ab p₂ p₂])
        apply simp
        apply (intro conjI notI)
        apply (erule notE)
        apply (rule uniq-opp [of a b])
        apply (simp-all add: add-comm add-closed)[2]
        apply (erule notE)
        apply (drule uniq-zero [OF a b ab add-closed [OF a b p₂ p₂] p₂])
        apply simp
        done
    next
      case False
      show ?thesis
      proof (cases p₂ = add a p₁ p₂)
        case True
        from a b ab p₁ p₂ True [symmetric]
        have p₁ = Infinity by (rule uniq-zero)
        then show ?thesis by (simp add: add-0-l)
      next
        case False
        show ?thesis
        proof (cases p₁ = p₂)
          case True
          with a b p₂ show ?thesis
            by (simp add: add-comm add-closed)
        next
          case False
          with a b p₁ p₂ ‹p₁ ≠ Infinity› ‹p₂ ≠ Infinity›
            ‹p₁ ≠ opp p₂› ‹p₂ ≠ opp p₂›
```

```
      ‹p₁ ≠ add a p₂ p₂› ‹p₂ ≠ add a p₁ p₂› ‹opp p₂ ≠ add a p₁ p₂›
    show ?thesis
      apply (rule-tac spec2-assoc [OF a b])
      apply (simp-all add: is-generic-def is-tangent-def)
      apply (rule notI)
      apply (erule notE [of p₁ = opp p₂])
      apply (rule uniq-opp)
      apply assumption
      apply (simp add: add-comm)
      apply (intro conjI notI)
      apply (erule notE [of p₂ = opp p₂])
      apply (rule uniq-opp)
      apply assumption+
      apply (rule notE [OF ‹opp p₁ ≠ add a p₂ p₂›])
      apply (simp add: opp-opp [OF add-closed [OF a b p₂ p₂]])
      done
  qed
 qed
 qed
 qed
 qed
 qed
 qed
qed

lemma add-assoc:
 assumes a: a ∈ carrier R
 and b: b ∈ carrier R
 and ab: nonsingular a b
 and p₁: on-curve a b p₁
 and p₂: on-curve a b p₂
 and p₃: on-curve a b p₃
 shows add a p₁ (add a p₂ p₃) = add a (add a p₁ p₂) p₃
proof (cases p₁ = Infinity)
 case True
 from a b ab p₁ p₂ p₃
 show ?thesis by (rule degen-assoc) (simp add: True)
next
 case False
 show ?thesis
 proof (cases p₂ = Infinity)
  case True
  from a b ab p₁ p₂ p₃
  show ?thesis by (rule degen-assoc) (simp add: True)
 next
  case False
  show ?thesis
  proof (cases p₃ = Infinity)
```

```
  case True
  from a b ab p1 p2 p3
  show ?thesis by (rule degen-assoc) (simp add: True)
next
  case False
  show ?thesis
  proof (cases p1 = p2)
    case True
    from a b p2 p3
    have add a p2 (add a p2 p3) = add a (add a p3 p2) p2
      by (simp add: add-comm add-closed)
    also from a b ab p3 p2 have ... = add a p3 (add a p2 p2)
      by (simp add: spec4-assoc)
    also from a b p2 p3
    have ... = add a (add a p2 p2) p3
      by (simp add: add-comm add-closed)
    finally show ?thesis using True by simp
  next
    case False
    show ?thesis
    proof (cases p1 = opp p2)
      case True
      from a b ab p1 p2 p3
      show ?thesis by (rule degen-assoc) (simp add: True)
    next
      case False
      show ?thesis
      proof (cases p2 = p3)
        case True
        with a b ab p1 p3 show ?thesis
          by (simp add: spec4-assoc)
      next
        case False
        show ?thesis
        proof (cases p2 = opp p3)
          case True
          from a b ab p1 p2 p3
          show ?thesis by (rule degen-assoc) (simp add: True)
        next
          case False
          show ?thesis
          proof (cases opp p1 = add a p2 p3)
            case True
            from a b ab p1 p2 p3
            show ?thesis by (rule degen-assoc) (simp add: True)
          next
            case False
            show ?thesis
            proof (cases opp p3 = add a p1 p2)
```

```
  case True
  from a b ab p1 p2 p3
  show ?thesis by (rule degen-assoc) (simp add: True)
next
  case False
  show ?thesis
  proof (cases p1 = add a p2 p3)
    case True
    with a b ab p2 p3 show ?thesis
      apply simp
      apply (rule cancel [OF a b ab opp-closed [OF p3]])
      apply (simp-all add: add-closed)
      apply (simp add: spec4-assoc add-closed opp-closed)
      apply (simp add: add-comm [of a b opp p3]
        add-closed opp-closed add-minus-id)
      apply (simp add: add-comm add-closed)
      done
  next
    case False
    show ?thesis
    proof (cases p3 = add a p1 p2)
      case True
      with a b ab p1 p2 show ?thesis
        apply simp
        apply (rule cancel [OF a b ab opp-closed [OF p1]])
        apply (simp-all add: add-closed)
        apply (simp add: spec4-assoc add-closed opp-closed)
        apply (simp add: add-comm [of a b opp p1] add-comm [of a b
p1]
          add-closed opp-closed add-minus-id)
        done
    next
      case False
      with a b p1 p2 p3
        ‹p1 ≠ Infinity› ‹p2 ≠ Infinity› ‹p3 ≠ Infinity›
        ‹p1 ≠ p2› ‹p1 ≠ opp p2› ‹p2 ≠ p3› ‹p2 ≠ opp p3›
        ‹opp p3 ≠ add a p1 p2› ‹p1 ≠ add a p2 p3›
      show ?thesis
        apply (rule-tac spec1-assoc [of a b])
        apply (simp-all add: is-generic-def)
        apply (rule notI)
        apply (erule notE [of p1 = opp p2])
        apply (rule uniq-opp)
        apply assumption
        apply (simp add: add-comm)
        apply (intro conjI notI)
        apply (erule notE [of p2 = opp p3])
        apply (rule uniq-opp)
        apply assumption
```

```
                apply (simp add: add-comm)
                apply (rule notE [OF ‹opp p1 ≠ add a p2 p3›])
                apply (simp add: opp-opp [OF add-closed [OF a b p2 p3]])
                done
            qed
          qed
        qed
      qed
    qed
  qed
 qed
qed

lemma add-comm':
  a ∈ carrier R ⟹ b ∈ carrier R ⟹ nonsingular a b ⟹
   on-curve a b p1 ⟹ on-curve a b p2 ⟹ on-curve a b p3 ⟹
   add a p2 (add a p1 p3) = add a p1 (add a p2 p3)
   by (simp add: add-assoc add-comm)

primrec point-mult :: 'a ⇒ nat ⇒ 'a point ⇒ 'a point
where
    point-mult a 0 p = Infinity
  | point-mult a (Suc n) p = add a p (point-mult a n p)

lemma point-mult-closed: a ∈ carrier R ⟹ b ∈ carrier R ⟹
  on-curve a b p ⟹ on-curve a b (point-mult a n p)
  by (induct n) (simp-all add: add-closed)

lemma point-mult-add:
  a ∈ carrier R ⟹ b ∈ carrier R ⟹ on-curve a b p ⟹ nonsingular a b ⟹
   point-mult a (m + n) p = add a (point-mult a m p) (point-mult a n p)
  by (induct m) (simp-all add: add-assoc point-mult-closed add-0-l)

lemma point-mult-mult:
  a ∈ carrier R ⟹ b ∈ carrier R ⟹ on-curve a b p ⟹ nonsingular a b ⟹
   point-mult a (m * n) p = point-mult a n (point-mult a m p)
   by (induct n) (simp-all add: point-mult-add)

lemma point-mult2-eq-double:
  point-mult a 2 p = add a p p
  by (simp add: numeral-2-eq-2 add-0-r)

end
```

3.2 Projective Coordinates

type-synonym *'a ppoint* = *'a* × *'a* × *'a*

definition (**in** *cring*) *pdouble* :: *'a* ⇒ *'a ppoint* ⇒ *'a ppoint* **where**
pdouble a p =
(*let* (*x*, *y*, *z*) = *p*
in
if z = **0** *then p*
else
let
l = «*2*» ⊗ *y* ⊗ *z*;
m = «*3*» ⊗ *x* [^] (*2*::*nat*) ⊕ *a* ⊗ *z* [^] (*2*::*nat*)
in
(*l* ⊗ (*m* [^] (*2*::*nat*) ⊖ «*4*» ⊗ *x* ⊗ *y* ⊗ *l*),
m ⊗ («*6*» ⊗ *x* ⊗ *y* ⊗ *l* ⊖ *m* [^] (*2*::*nat*)) ⊖
«*2*» ⊗ *y* [^] (*2*::*nat*) ⊗ *l* [^] (*2*::*nat*),
l [^] (*3*::*nat*)))

definition (**in** *cring*) *padd* :: *'a* ⇒ *'a ppoint* ⇒ *'a ppoint* ⇒ *'a ppoint* **where**
padd a p_1 p_2 =
(*let*
(x_1, y_1, z_1) = p_1;
(x_2, y_2, z_2) = p_2
in
if z_1 = **0** *then* p_2
else if z_2 = **0** *then* p_1
else
let
d_1 = x_2 ⊗ z_1;
d_2 = x_1 ⊗ z_2;
l = d_1 ⊖ d_2;
m = y_2 ⊗ z_1 ⊖ y_1 ⊗ z_2
in
if l = **0** *then*
if m = **0** *then pdouble a* p_1
else (**0**, **0**, **0**)
else
let h = *m* [^] (*2*::*nat*) ⊗ z_1 ⊗ z_2 ⊖ (d_1 ⊕ d_2) ⊗ *l* [^] (*2*::*nat*)
in
(*l* ⊗ *h*,
(d_2 ⊗ *l* [^] (*2*::*nat*) ⊖ *h*) ⊗ *m* ⊖ *l* [^] (*3*::*nat*) ⊗ y_1 ⊗ z_2,
l [^] (*3*::*nat*) ⊗ z_1 ⊗ z_2))

definition (**in** *field*) *make-affine* :: *'a ppoint* ⇒ *'a point* **where**
make-affine p =
(*let* (*x*, *y*, *z*) = *p*
in if z = **0** *then Infinity else Point* (*x* ⊘ *z*) (*y* ⊘ *z*))

definition (**in** *cring*) *in-carrierp* :: *'a ppoint* ⇒ *bool* **where**

in-carrierp = (λ(*x*, *y*, *z*). *x* ∈ *carrier R* ∧ *y* ∈ *carrier R* ∧ *z* ∈ *carrier R*)

definition (**in** *cring*) *on-curvep* :: *'a* ⇒ *'a* ⇒ *'a ppoint* ⇒ *bool* **where**
on-curvep a b = (λ(*x*, *y*, *z*).
x ∈ *carrier R* ∧ *y* ∈ *carrier R* ∧ *z* ∈ *carrier R* ∧
(*z* ≠ **0** ⟶
y [^] (*2*::*nat*) ⊗ *z* = *x* [^] (*3*::*nat*) ⊕ *a* ⊗ *x* ⊗ *z* [^] (*2*::*nat*) ⊕ *b* ⊗ *z* [^] (*3*::*nat*)))

lemma (**in** *cring*) *on-curvep-infinity* [*simp*]: *on-curvep a b* (*x*, *y*, **0**) = (*x* ∈ *carrier R* ∧ *y* ∈ *carrier R*)
by (*simp add*: *on-curvep-def*)

lemma (**in** *field*) *make-affine-infinity* [*simp*]: *make-affine* (*x*, *y*, **0**) = *Infinity*
by (*simp add*: *make-affine-def*)

lemma (**in** *cring*) *on-curvep-imp-in-carrierp* [*simp*]: *on-curvep a b p* ⟹ *in-carrierp p*
by (*auto simp add*: *on-curvep-def in-carrierp-def*)

lemma (**in** *ell-field*) *on-curvep-iff-on-curve*:
assumes *a* ∈ *carrier R b* ∈ *carrier R in-carrierp p*
shows *on-curvep a b p* = *on-curve a b* (*make-affine p*)
using *assms*
proof (*induct p rule*: *prod-induct3*)
case (*fields x y z*)
show *on-curvep a b* (*x*, *y*, *z*) = *on-curve a b* (*make-affine* (*x*, *y*, *z*))
proof
assume *H*: *on-curvep a b* (*x*, *y*, *z*)
then have *carrier*: *x* ∈ *carrier R y* ∈ *carrier R z* ∈ *carrier R*
and *yz*: *z* ≠ **0** ⟹
y [^] (*2*::*nat*) ⊗ *z* = *x* [^] (*3*::*nat*) ⊕ *a* ⊗ *x* ⊗ *z* [^] (*2*::*nat*) ⊕ *b* ⊗ *z* [^] (*3*::*nat*)
by (*simp-all add*: *on-curvep-def*)
show *on-curve a b* (*make-affine* (*x*, *y*, *z*))
proof (*cases z* = **0**)
case *True*
then show *?thesis* **by** (*simp add*: *on-curve-def make-affine-def*)
next
case *False*
then show *?thesis*
apply (*simp add*: *on-curve-def make-affine-def carrier*)
apply (*field yz* [*OF False*])
apply *assumption*
done
qed
next
assume *H*: *on-curve a b* (*make-affine* (*x*, *y*, *z*))
show *on-curvep a b* (*x*, *y*, *z*)

```
    proof (cases z = 0)
      case True
      with ‹in-carrierp (x, y, z)› show ?thesis
        by (simp add: on-curvep-def in-carrierp-def)
    next
      case False
      from ‹in-carrierp (x, y, z)›
      have carrier: x ∈ carrier R y ∈ carrier R z ∈ carrier R
        by (simp-all add: in-carrierp-def)
      from H show ?thesis
        apply (simp add: on-curve-def on-curvep-def make-affine-def carrier False)
        apply (field (prems))
        apply field
        apply (simp-all add: False)
        done
    qed
  qed
qed

lemma (in cring) pdouble-in-carrierp:
  a ∈ carrier R ⟹ in-carrierp p ⟹ in-carrierp (pdouble a p)
  by (auto simp add: in-carrierp-def pdouble-def Let-def split: prod.split)

lemma (in cring) padd-in-carrierp:
  a ∈ carrier R ⟹ in-carrierp p₁ ⟹ in-carrierp p₂ ⟹ in-carrierp (padd a p₁
p₂)
  by (auto simp add: padd-def Let-def pdouble-in-carrierp split: prod.split)
    (auto simp add: in-carrierp-def)

lemma (in cring) pdouble-infinity [simp]: pdouble a (x, y, 0) = (x, y, 0)
  by (simp add: pdouble-def)

lemma (in cring) padd-infinity-l [simp]: padd a (x, y, 0) p = p
  by (simp add: padd-def)

lemma (in ell-field) pdouble-correct:
  a ∈ carrier R ⟹ in-carrierp p ⟹
   make-affine (pdouble a p) = add a (make-affine p) (make-affine p)
proof (induct p rule: prod-induct3)
  case (fields x y z)
  then have x ∈ carrier R y ∈ carrier R z ∈ carrier R
    by (simp-all add: in-carrierp-def)
  then show ?case
     apply (auto simp add: add-def pdouble-def make-affine-def equal-neg-zero di-
vide-eq-0-iff
      integral-iff Let-def simp del: minus-divide-left)
    apply field
    apply (simp add: integral-iff)
    apply field
```

```
    apply (simp add: integral-iff)
    done
qed

lemma (in ell-field) padd-correct:
  assumes a: a ∈ carrier R and b: b ∈ carrier R
  and p₁: on-curvep a b p₁ and p₂: on-curvep a b p₂
  shows make-affine (padd a p₁ p₂) = add a (make-affine p₁) (make-affine p₂)
  using p₁
proof (induct p₁ rule: prod-induct3)
  case (fields x₁ y₁ z₁)
  note p₁' = fields
  from p₂ show ?case
  proof (induct p₂ rule: prod-induct3)
    case (fields x₂ y₂ z₂)
    then have x₂ ∈ carrier R y₂ ∈ carrier R z₂ ∈ carrier R and
      yz₂: z₂ ≠ 𝟬 ⟹ y₂ [^] (2::nat) ⊗ z₂ ⊗ z₁ [^] (3::nat) =
        (x₂ [^] (3::nat) ⊕ a ⊗ x₂ ⊗ z₂ [^] (2::nat) ⊕ b ⊗ z₂ [^] (3::nat)) ⊗ z₁ [^]
(3::nat)
      by (simp-all add: on-curvep-def)
    from p₁' have x₁ ∈ carrier R y₁ ∈ carrier R z₁ ∈ carrier R and
      yz₁: z₁ ≠ 𝟬 ⟹ y₁ [^] (2::nat) ⊗ z₁ ⊗ z₂ [^] (3::nat) =
        (x₁ [^] (3::nat) ⊕ a ⊗ x₁ ⊗ z₁ [^] (2::nat) ⊕ b ⊗ z₁ [^] (3::nat)) ⊗ z₂ [^]
(3::nat)
      by (simp-all add: on-curvep-def)
    show ?case
    proof (cases z₁ = 𝟬)
      case True
      then show ?thesis
        by (simp add: add-def padd-def make-affine-def)
    next
      case False
      show ?thesis
      proof (cases z₂ = 𝟬)
        case True
        then show ?thesis
          by (simp add: add-def padd-def make-affine-def)
      next
        case False
        show ?thesis
        proof (cases x₂ ⊗ z₁ ⊖ x₁ ⊗ z₂ = 𝟬)
          case True
          note x = this
          with ‹x₁ ∈ carrier R› ‹x₂ ∈ carrier R› ‹z₁ ∈ carrier R› ‹ z₂ ∈ carrier R›
          have x': x₂ ⊗ z₁ = x₁ ⊗ z₂ by (simp add: eq-diff0)
          show ?thesis
          proof (cases y₂ ⊗ z₁ ⊖ y₁ ⊗ z₂ = 𝟬)
            case True
            with ‹y₁ ∈ carrier R› ‹y₂ ∈ carrier R› ‹z₁ ∈ carrier R› ‹ z₂ ∈ carrier
```

```
R›
        have y: y₂ ⊗ z₁ = y₁ ⊗ z₂ by (simp add: eq-diff0)
        from ‹z₁ ≠ 0› ‹z₂ ≠ 0› x
        have make-affine (x₂, y₂, z₂) = make-affine (x₁, y₁, z₁)
          apply (simp add: make-affine-def)
          apply (rule conjI)
          apply (field x')
          apply simp
          apply (field y)
          apply simp
          done
        with True x ‹z₁ ≠ 0› ‹z₂ ≠ 0› p₁' fields a show ?thesis
          by (simp add: padd-def pdouble-correct)
      next
        case False
        have y₂ [^] (2::nat) ⊗ z₁ [^] (3::nat) ⊗ z₂ =
          y₁ [^] (2::nat) ⊗ z₁ ⊗ z₂ [^] (3::nat)
          by (ring yz₁ [OF ‹z₁ ≠ 0›] yz₂ [OF ‹z₂ ≠ 0›] x')
        then have y₂ [^] (2::nat) ⊗ z₁ [^] (3::nat) ⊗ z₂ ⊘ z₁ ⊘ z₂ =
          y₁ [^] (2::nat) ⊗ z₁ ⊗ z₂ [^] (3::nat) ⊘ z₁ ⊘ z₂
          by simp
        then have (y₂ ⊗ z₁) ⊗ (y₂ ⊗ z₁) = (y₁ ⊗ z₂) ⊗ (y₁ ⊗ z₂)
          apply (field (prems))
          apply (field)
          apply (rule TrueI)
          apply (simp add: ‹z₁ ≠ 0› ‹z₂ ≠ 0›)
          done
        with False
        have y₂z₁: y₂ ⊗ z₁ = ⊖ (y₁ ⊗ z₂)
          by (simp add: square-eq-iff eq-diff0
            ‹y₁ ∈ carrier R› ‹y₂ ∈ carrier R› ‹z₁ ∈ carrier R› ‹z₂ ∈ carrier R›)
        from x False ‹z₁ ≠ 0› ‹z₂ ≠ 0› show ?thesis
          apply (simp add: padd-def add-def make-affine-def Let-def)
          apply (rule conjI)
          apply (rule impI)
          apply (field x')
          apply simp
          apply (field y₂z₁)
          apply simp
          done
      qed
    next
      case False
      then have x₁ ⊘ z₁ ≠ x₂ ⊘ z₂
        apply (rule-tac notI)
        apply (erule notE)
        apply (drule sym)
        apply (field (prems))
        apply ring
```

```
        apply (simp add: ‹z1 ≠ 0› ‹z2 ≠ 0›)
        done
      with False ‹z1 ≠ 0› ‹z2 ≠ 0›
        ‹x1 ∈ carrier R› ‹x2 ∈ carrier R› ‹z1 ∈ carrier R› ‹z2 ∈ carrier R›
      show ?thesis
      apply (auto simp add: padd-def add-def make-affine-def Let-def integral-iff)
        apply field
        apply (simp add: integral-iff)
        apply field
        apply (simp add: integral-iff)
        done
    qed
  qed
 qed
qed

lemma (in ell-field) pdouble-closed:
 assumes a ∈ carrier R b ∈ carrier R on-curvep a b p
 shows on-curvep a b (pdouble a p)
proof −
 from ‹on-curvep a b p› have in-carrierp p by simp
 from assms show ?thesis
  by (simp add: on-curvep-iff-on-curve pdouble-in-carrierp pdouble-correct
    add-closed ‹in-carrierp p›)
qed

lemma (in ell-field) padd-closed:
 assumes a ∈ carrier R b ∈ carrier R on-curvep a b p1 on-curvep a b p2
 shows on-curvep a b (padd a p1 p2)
proof −
 from ‹on-curvep a b p1› have in-carrierp p1 by simp
 from ‹on-curvep a b p2› have in-carrierp p2 by simp
 from assms show ?thesis
  by (simp add: on-curvep-iff-on-curve padd-in-carrierp padd-correct
    add-closed ‹in-carrierp p1› ‹in-carrierp p2›)
qed

primrec (in cring) ppoint-mult :: 'a ⇒ nat ⇒ 'a ppoint ⇒ 'a ppoint
where
  ppoint-mult a 0 p = (0, 0, 0)
 | ppoint-mult a (Suc n) p = padd a p (ppoint-mult a n p)

lemma (in ell-field) ppoint-mult-closed [simp]:
  a ∈ carrier R ⟹ b ∈ carrier R ⟹ on-curvep a b p ⟹ on-curvep a b
(ppoint-mult a n p)
 by (induct n) (simp-all add: padd-closed)

lemma (in ell-field) ppoint-mult-correct: a ∈ carrier R ⟹ b ∈ carrier R ⟹
```

on-curvep a b p $\Longrightarrow$
make-affine (*ppoint-mult a n p*) = *point-mult a n* (*make-affine p*)
by (*induct n*) (*simp-all add*: *padd-correct*)

definition (**in** *cring*) *proj-eq* :: *'a ppoint* $\Rightarrow$ *'a ppoint* $\Rightarrow$ *bool* **where**
proj-eq = $(\lambda(x_1,\ y_1,\ z_1)\ (x_2,\ y_2,\ z_2).$
$(z_1 = \mathbf{0}) = (z_2 = \mathbf{0}) \wedge x_1 \otimes z_2 = x_2 \otimes z_1 \wedge y_1 \otimes z_2 = y_2 \otimes z_1)$

lemma (**in** *cring*) *proj-eq-refl*: *proj-eq p p*
by (*auto simp add*: *proj-eq-def*)

lemma (**in** *cring*) *proj-eq-sym*: *proj-eq p p'* $\Longrightarrow$ *proj-eq p' p*
by (*auto simp add*: *proj-eq-def*)

lemma (**in** *domain*) *proj-eq-trans*:
in-carrierp p $\Longrightarrow$ *in-carrierp p'* $\Longrightarrow$ *in-carrierp p''* $\Longrightarrow$
proj-eq p p' $\Longrightarrow$ *proj-eq p' p''* $\Longrightarrow$ *proj-eq p p''*
proof (*induct p rule*: *prod-induct3*)
case (*fields x y z*)
then show *?case*
proof (*induct p' rule*: *prod-induct3*)
case (*fields x' y' z'*)
then show *?case*
proof (*induct p'' rule*: *prod-induct3*)
case (*fields x'' y'' z''*)
then have *carrier*:
$x \in$ *carrier R* $y \in$ *carrier R* $z \in$ *carrier R*
$x' \in$ *carrier R* $y' \in$ *carrier R* $z' \in$ *carrier R*
$x'' \in$ *carrier R* $y'' \in$ *carrier R* $z'' \in$ *carrier R*
and *z*: $(z = \mathbf{0}) = (z' = \mathbf{0})$ $(z' = \mathbf{0}) = (z'' = \mathbf{0})$ **and**
$x \otimes z' \otimes z'' = x' \otimes z \otimes z''$
$y \otimes z' \otimes z'' = y' \otimes z \otimes z''$
and *xy*:
$x' \otimes z'' = x'' \otimes z'$
$y' \otimes z'' = y'' \otimes z'$
by (*simp-all add*: *in-carrierp-def proj-eq-def*)
from ‹$x \otimes z' \otimes z'' = x' \otimes z \otimes z''$›
have $(x \otimes z'') \otimes z' = (x'' \otimes z) \otimes z'$
by (*ring* (*prems*) *xy*) (*ring xy*)
moreover from ‹$y \otimes z' \otimes z'' = y' \otimes z \otimes z''$›
have $(y \otimes z'') \otimes z' = (y'' \otimes z) \otimes z'$
by (*ring* (*prems*) *xy*) (*ring xy*)
ultimately show *?case* **using** *z*
by (*auto simp add*: *proj-eq-def carrier conc*)
qed
qed
qed

lemma (**in** *field*) *make-affine-proj-eq-iff*:

```
  in-carrierp p ⟹ in-carrierp p' ⟹ proj-eq p p' = (make-affine p = make-affine
p')
proof (induct p rule: prod-induct3)
  case (fields x y z)
  then show ?case
  proof (induct p' rule: prod-induct3)
    case (fields x' y' z')
    then have carrier:
      x ∈ carrier R y ∈ carrier R z ∈ carrier R
      x' ∈ carrier R y' ∈ carrier R z' ∈ carrier R
      by (simp-all add: in-carrierp-def)
    show ?case
    proof
      assume proj-eq (x, y, z) (x', y', z')
      then have (z = 0) = (z' = 0)
        and xy: x ⊗ z' = x' ⊗ z y ⊗ z' = y' ⊗ z
        by (simp-all add: proj-eq-def)
      then show make-affine (x, y, z) = make-affine (x', y', z')
        apply (auto simp add: make-affine-def)
        apply (field xy)
        apply simp
        apply (field xy)
        apply simp
        done
    next
      assume H: make-affine (x, y, z) = make-affine (x', y', z')
      show proj-eq (x, y, z) (x', y', z')
      proof (cases z = 0)
        case True
        with H have z' = 0 by (simp add: make-affine-def split: if-split-asm)
        with True carrier show ?thesis by (simp add: proj-eq-def)
      next
        case False
        with H have z' ≠ 0 x ⊘ z = x' ⊘ z' y ⊘ z = y' ⊘ z'
          by (simp-all add: make-affine-def split: if-split-asm)
        from ‹x ⊘ z = x' ⊘ z'›
        have x ⊗ z' = x' ⊗ z
          apply (field (prems))
          apply field
          apply (simp-all add: ‹z ≠ 0› ‹z' ≠ 0›)
          done
        moreover from ‹y ⊘ z = y' ⊘ z'›
        have y ⊗ z' = y' ⊗ z
          apply (field (prems))
          apply field
          apply (simp-all add: ‹z ≠ 0› ‹z' ≠ 0›)
          done
        ultimately show ?thesis
          by (simp add: proj-eq-def ‹z ≠ 0› ‹z' ≠ 0›)
```

qed
qed
qed
qed

lemma (**in** *ell-field*) *pdouble-proj-eq-cong*:
a ∈ *carrier R* ⟹ *in-carrierp p* ⟹ *in-carrierp p′* ⟹ *proj-eq p p′* ⟹
proj-eq (*pdouble a p*) (*pdouble a p′*)
by (*simp add*: *make-affine-proj-eq-iff pdouble-in-carrierp pdouble-correct*)

lemma (**in** *ell-field*) *padd-proj-eq-cong*:
a ∈ *carrier R* ⟹ *b* ∈ *carrier R* ⟹ *on-curvep a b p*$_1$ ⟹ *on-curvep a b p*$_1$′ ⟹
on-curvep a b p$_2$ ⟹ *on-curvep a b p*$_2$′ ⟹ *proj-eq p*$_1$ *p*$_1$′ ⟹ *proj-eq p*$_2$ *p*$_2$′ ⟹
proj-eq (*padd a p*$_1$ *p*$_2$) (*padd a p*$_1$′ *p*$_2$′)
by (*simp add*: *make-affine-proj-eq-iff padd-in-carrierp padd-correct*)

end

4 Validating the Specification

theory *Elliptic-Test*
imports
Elliptic-Locale
HOL−Number-Theory.Residues
begin

4.1 Specialized Definitions for Prime Fields

definition *mmult* :: *int* ⇒ *int* ⇒ *int* ⇒ *int* (**infixl** ∗∗$_1$ *70*)
where *x* ∗∗$_m$ *y* = *x* ∗ *y mod m*

definition *madd* :: *int* ⇒ *int* ⇒ *int* ⇒ *int* (**infixl** +$+_1$ *65*)
where *x* +$+_m$ *y* = (*x* + *y*) *mod m*

definition *msub* :: *int* ⇒ *int* ⇒ *int* ⇒ *int* (**infixl** −$-_1$ *65*)
where *x* −$-_m$ *y* = (*x* − *y*) *mod m*

definition *mpow* :: *int* ⇒ *int* ⇒ *nat* ⇒ *int* (**infixr** $\widehat{\ \ }_1$ *80*)
where *x* $\widehat{\ \ }_m$ *n* = *x* ^ *n mod m*

lemma (**in** *residues*) *res-of-natural-eq*: «*n*»$_{\mathbf{N}}$ = *int n mod m*
by (*induct n*)
(*simp-all add*: *of-natural-def res-zero-eq res-one-eq res-add-eq mod-add-right-eq*)

lemma (**in** *residues*) *res-of-integer-eq*: «*i*» = *i mod m*
by (*simp add*: *of-integer-def res-of-natural-eq res-neg-eq mod-minus-eq*)

lemma (**in** *residues*) *res-pow-eq*: *x* [^] (*n*::*nat*) = *x* ^ *n mod m*
using *m-gt-one*

by (*induct n*)
(*simp-all add*: *res-one-eq res-mult-eq mult-ac mod-mult-right-eq*)

lemma (**in** *residues*) *res-sub-eq*: $(x \bmod m) \ominus (y \bmod m) = (x \bmod m - y \bmod m) \bmod m$
by (*simp add*: *minus-eq res-neg-eq res-add-eq mod-minus-eq mod-add-eq mod-diff-eq*)

definition *mpdouble* :: *int* ⇒ *int* ⇒ *int ppoint* ⇒ *int ppoint* **where**
mpdouble m a p =
(*let* (*x*, *y*, *z*) = *p*
in
if z = *0 then p*
else
let
l = *2 mod m* $**_m$ *y* $**_m$ *z*;
n = *3 mod m* $**_m$ *x* $\widehat{\sim\sim}_m$ *2* $+\!+_m$ *a* $**_m$ *z* $\widehat{\sim\sim}_m$ *2*
in
(*l* $**_m$ (*n* $\widehat{\sim\sim}_m$ *2* $-\!-_m$ *4 mod m* $**_m$ *x* $**_m$ *y* $**_m$ *l*),
n $**_m$ (*6 mod m* $**_m$ *x* $**_m$ *y* $**_m$ *l* $-\!-_m$ *n* $\widehat{\sim\sim}_m$ *2*) $-\!-_m$
2 mod m $**_m$ *y* $\widehat{\sim\sim}_m$ *2* $**_m$ *l* $\widehat{\sim\sim}_m$ *2*,
l $\widehat{\sim\sim}_m$ *3*))

definition *mpadd* :: *int* ⇒ *int* ⇒ *int ppoint* ⇒ *int ppoint* ⇒ *int ppoint* **where**
mpadd m a p_1 p_2 =
(*let*
(x_1, y_1, z_1) = p_1;
(x_2, y_2, z_2) = p_2
in
if z_1 = *0 then* p_2
else if z_2 = *0 then* p_1
else
let
d_1 = x_2 $**_m$ z_1;
d_2 = x_1 $**_m$ z_2;
l = d_1 $-\!-_m$ d_2;
n = y_2 $**_m$ z_1 $-\!-_m$ y_1 $**_m$ z_2
in
if l = *0 then*
if n = *0 then mpdouble m a* p_1
else (*0*, *0*, *0*)
else
let h = *n* $\widehat{\sim\sim}_m$ *2* $**_m$ z_1 $**_m$ z_2 $-\!-_m$ (d_1 $+\!+_m$ d_2) $**_m$ *l* $\widehat{\sim\sim}_m$ *2*
in
(*l* $**_m$ *h*,
(d_2 $**_m$ *l* $\widehat{\sim\sim}_m$ *2* $-\!-_m$ *h*) $**_m$ *n* $-\!-_m$ *l* $\widehat{\sim\sim}_m$ *3* $**_m$ y_1 $**_m$ z_2,
l $\widehat{\sim\sim}_m$ *3* $**_m$ z_1 $**_m$ z_2))

lemma (**in** *residues*) *pdouble-residue-eq*: *pdouble a p* = *mpdouble m a p*
by (*simp only*: *pdouble-def mpdouble-def*

madd-def mmult-def msub-def mpow-def res-zero-eq res-add-eq res-mult-eq res-of-integer-eq res-pow-eq res-sub-eq)

lemma (**in** *residues*) *padd-residue-eq*: *padd a* p_1 p_2 = *mpadd m a* p_1 p_2
by (*simp only*: *padd-def mpadd-def pdouble-residue-eq madd-def mmult-def msub-def mpow-def res-zero-eq res-add-eq res-mult-eq res-of-integer-eq res-pow-eq res-sub-eq Let-def*)

fun *fast-ppoint-mult* :: *int* ⇒ *int* ⇒ *nat* ⇒ *int ppoint* ⇒ *int ppoint*
where
fast-ppoint-mult m a n p =
(*if n* = *0 then* (*0*, *0*, *0*)
else if n mod 2 = *0 then mpdouble m a* (*fast-ppoint-mult m a* (*n div 2*) *p*)
else mpadd m a p (*mpdouble m a* (*fast-ppoint-mult m a* (*n div 2*) *p*)))

lemma *fast-ppoint-mult-0* [*simp*]: *fast-ppoint-mult m a 0 p* = (*0*, *0*, *0*)
by *simp*

lemma *fast-ppoint-mult-even* [*simp*]:
n ≠ *0* ⟹ *n mod 2* = *0* ⟹
fast-ppoint-mult m a n p = *mpdouble m a* (*fast-ppoint-mult m a* (*n div 2*) *p*)
by *simp*

lemma *fast-ppoint-mult-odd* [*simp*]:
n ≠ *0* ⟹ *n mod 2* ≠ *0* ⟹
fast-ppoint-mult m a n p = *mpadd m a p* (*mpdouble m a* (*fast-ppoint-mult m a* (*n div 2*) *p*))
by *simp*

declare *fast-ppoint-mult.simps* [*simp del*]

locale *residues-prime-gt2* = *residues-prime* +
assumes *gt2*: *2* < *p*

sublocale *residues-prime-gt2* < *ell-field*
using *gt2*
by *unfold-locales* (*simp add*: *res-of-integer-eq res-zero-eq*)

lemma (**in** *residues-prime-gt2*) *fast-ppoint-mult-closed*:
assumes *a* ∈ *carrier R b* ∈ *carrier R on-curvep a b q*
shows *on-curvep a b* (*fast-ppoint-mult* (*int p*) *a n q*)
using *assms*
proof (*induct int p a n q rule*: *fast-ppoint-mult.induct*)
case (*1 a n q*)
show *?case*
proof (*cases n* = *0*)
case *True*
then show *?thesis* **using** *m-gt-one*
by (*simp add*: *on-curvep-infinity* [*simplified res-zero-eq*] *res-carrier-eq*)

```
  next
    case False
    with 1 show ?thesis
      by (cases n mod 2 = 0)
        (simp-all add: padd-residue-eq [symmetric] pdouble-residue-eq [symmetric]
          padd-closed pdouble-closed)
  qed
qed

lemma (in residues-prime-gt2) point-mult-residue-eq:
  assumes a ∈ carrier R b ∈ carrier R on-curvep a b q nonsingular a b
  shows proj-eq (ppoint-mult a n q) (fast-ppoint-mult (int p) a n q)
proof −
  from assms
  have point-mult a n (make-affine q) = make-affine (fast-ppoint-mult (int p) a n
q)
  proof (induct int p a n q rule: fast-ppoint-mult.induct)
    case (1 a n q)
    show ?case
    proof (cases n = 0)
      case True
     then show ?thesis by (simp add: make-affine-infinity [simplified res-zero-eq])
    next
      case False
      have point-mult a n (make-affine q) =
        point-mult a (n div 2 * 2 + n mod 2) (make-affine q)
        by simp
      also from 1
      have ... = add a (point-mult a 2 (point-mult a (n div 2) (make-affine q)))
        (point-mult a (n mod 2) (make-affine q))
        by (simp only: point-mult-mult point-mult-add
          on-curvep-iff-on-curve [symmetric] on-curvep-imp-in-carrierp)
      also have ... = make-affine (fast-ppoint-mult (int p) a n q)
        using 1 False
        by (cases n mod 2 = 0)
          (simp-all add: padd-residue-eq [symmetric] pdouble-residue-eq [symmetric]
add-0-r
            padd-correct pdouble-correct
            fast-ppoint-mult-closed on-curvep-imp-in-carrierp [of a b]
            point-mult2-eq-double pdouble-closed
              add-assoc [symmetric] add-comm add-comm' on-curvep-iff-on-curve
[symmetric])
      finally show ?thesis .
    qed
  qed
  with assms show ?thesis
    by (simp add: make-affine-proj-eq-iff fast-ppoint-mult-closed
      ppoint-mult-correct on-curvep-imp-in-carrierp [of a b])
qed
```

definition *mmake-affine* :: *int* ⇒ *int ppoint* ⇒ *int point* **where**
mmake-affine q p =
(*let* (*x*, *y*, *z*) = *p*
in if z = *0 then Infinity else*
let (*a*, *b*) = *bezout-coefficients z q*
in Point (*a* $**_q$ *x*) (*a* $**_q$*y*))

lemma (**in** *residues-prime*) *make-affine-residue-eq*:
assumes *in-carrierp q*
shows *make-affine q* = *mmake-affine* (*int p*) *q*
proof (*cases q*)
case (*fields x y z*)
show *?thesis*
proof (*cases z* = *0*)
case *True*
with *fields* **show** *?thesis* **by** (*simp add*: *make-affine-def mmake-affine-def res-zero-eq*)
next
case *False*
show *?thesis*
proof (*cases bezout-coefficients z* (*int p*))
case (*Pair a b*)
with *fields False assms* **have** ¬ *int p dvd z*
by (*auto simp add*: *in-carrierp-def res-carrier-eq prime-imp-coprime zdvd-not-zless*)
with *p-prime* **have** *coprime* (*int p*) *z*
by (*auto intro*: *prime-imp-coprime*)
then have *coprime z* (*int p*)
by (*simp add*: *ac-simps*)
then have *fst* (*bezout-coefficients z* (*int p*)) ∗ *z* +
snd (*bezout-coefficients z* (*int p*)) ∗ *int p* = *1*
by (*simp add*: *bezout-coefficients-fst-snd*)
with *m-gt-one* **have** *fst* (*bezout-coefficients z* (*int p*)) ∗ *z mod int p* = *1*
by (*auto dest*: *arg-cong* [*of* - - λ*x*. *x mod int p*])
then have *z* ⊗ (*fst* (*bezout-coefficients z* (*int p*)) *mod int p*) = **1**
by (*simp add*: *res-mult-eq res-one-eq mult.commute mod-mult-right-eq*)
with *fields assms* **have** *inv z* = *fst* (*bezout-coefficients z* (*int p*)) *mod int p*
by (*simp add*: *inverse-unique in-carrierp-def res-carrier-eq*)
with *fields Pair False* **show** *?thesis*
by (*simp add*: *make-affine-def mmake-affine-def res-zero-eq m-div-def*
res-mult-eq mmult-def mod-mult-right-eq mult.commute)
qed
qed
qed

definition *mon-curve* :: *int* ⇒ *int* ⇒ *int* ⇒ *int point* ⇒ *bool* **where**
mon-curve m a b p = (*case p of*
Infinity ⇒ *True*
| *Point x y* ⇒ *0* ≤ *x* ∧ *x* < *m* ∧ *0* ≤ *y* ∧ *y* < *m* ∧

y ^^^$_m$ *2* = *x* ^^^$_m$ *3* ++$_m$ *a* **$_m$ *x* ++$_m$ *b*)

lemma (**in** *residues-prime-gt2*) *on-curve-residues-eq*:
on-curve a b q = *mon-curve* (*int p*) *a b q*
by (*simp add*: *on-curve-def mon-curve-def res-carrier-eq res-add-eq res-mult-eq res-pow-eq*
madd-def mmult-def mpow-def split: *point.split*)

4.2 The NIST Curve P-521

The following test data is taken from RFC 5903 [1], §3.3 and §8.3. The curve parameters can also be found in §D.1.2.5 of FIPS PUB 186-4 [2].

definition *m* :: *int* **where**
m = *0x01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF*

definition *a* :: *int* **where**
a = *m* − *3*

definition *b* :: *int* **where**
b = *0x0051953EB9618E1C9A1F929A21A0B68540EEA2DA725B99B315F3B8B489918EF109E156193951EC*

definition *gx* :: *int* **where**
gx = *0x00C6858E06B70404E9CD9E3ECB662395B4429C648139053FB521F828AF606B4D3DBAA14B5E77E*

definition *gy* :: *int* **where**
gy = *0x011839296A789A3BC0045C8A5FB42C7D1BD998F54449579B446817AFBD17273E662C97EE72995E*

definition *priv* :: *nat* **where**
priv = *0x0037ADE9319A89F4DABDB3EF411AACCCA5123C61ACAB57B5393DCE47608172A095AA85A3*

definition *pubx* :: *int* **where**
pubx = *0x0015417E84DBF28C0AD3C278713349DC7DF153C897A1891BD98BAB4357C9ECBEE1E3BF42E*

definition *puby* :: *int* **where**
puby = *0x017CAE20B6641D2EEB695786D8C946146239D099E18E1D5A514C739D7CB4A10AD8A788015A*

definition *order* :: *nat* **where**
order = *0x01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF*

lemma *mon-curve m a b* (*Point gx gy*)
by *eval*

lemma *mmake-affine m* (*fast-ppoint-mult m a priv* (*gx*, *gy*, *1*)) = *Point pubx puby*
by *eval*

lemma *mmake-affine m* (*fast-ppoint-mult m a order* (*gx*, *gy*, *1*)) = *Infinity*
by *eval*

end

References

[1] D. E. Fu and J. A. Solinas. Elliptic Curve Groups modulo a Prime (ECP Groups) for IKE and IKEv2. RFC 5903, Internet Engineering Task Force (IETF), June 2010. Available online at https://tools.ietf.org/html/rfc5903.

[2] C. F. Kerry and P. D. Gallagher. Digital Signature Standard (DSS). Federal Information Processing Standards (FIPS) Publication 186-4, Information Technology Laboratory, National Institute of Standards and Technology (NIST), July 2013. Available online at https://doi.org/10.6028/NIST.FIPS.186-4.

[3] A. Mahboubi and B. Grégoire. Proving Equalities in a Commutative Ring Done Right in Coq. In J. Hurd and T. Melham, editors, *TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113, Oxford, United Kingdom, August 2005. Springer.

[4] L. Théry. Proving the group law for elliptic curves formally. Technical Report RT-0330, INRIA, March 2007. Available online at https://hal.inria.fr/inria-00129237, Coq sources available at http://coqprime.gforge.inria.fr.