The Group Law for Elliptic Curves

Stefan Berghofer

December 7, 2022

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

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
  ⟨proof⟩

lemma opp-Infinity [simp]: opp Infinity = Infinity
  ⟨proof⟩

lemma opp-Point: opp (Point x y) = Point x (− y)
  ⟨proof⟩

lemma opp-opp: opp (opp p) = p
  ⟨proof⟩

lemma opp-closed:
  on-curve a b p ⟹ on-curve a b (opp p)
  ⟨proof⟩

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⟩

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⟩

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*: $\bigwedge p_1\ x_1\ y_1\ p_2\ x_2\ y_2\ l.$
$p_1 = Point\ x_1\ y_1 \Longrightarrow p_2 = Point\ x_2\ y_2 \Longrightarrow$
$p_2 = add\ a\ p_1\ p_1 \Longrightarrow y_1 \neq 0 \Longrightarrow$
$l = (3 * x_1\ \hat{}\ 2 + a)\ /\ (2 * y_1) \Longrightarrow$
$x_2 = l\ \hat{}\ 2 - 2 * x_1 \Longrightarrow$
$y_2 = -\ y_1 - l * (x_2 - x_1) \Longrightarrow$
$P\ p_1\ p_1\ p_2$
and *R5*: $\bigwedge p_1\ x_1\ y_1\ p_2\ x_2\ y_2\ p_3\ x_3\ y_3\ l.$
$p_1 = Point\ x_1\ y_1 \Longrightarrow p_2 = Point\ x_2\ y_2 \Longrightarrow p_3 = Point\ x_3\ y_3 \Longrightarrow$
$p_3 = add\ a\ p_1\ p_2 \Longrightarrow x_1 \neq x_2 \Longrightarrow$
$l = (y_2 - y_1)\ /\ (x_2 - x_1) \Longrightarrow$
$x_3 = l\ \hat{}\ 2 - x_1 - x_2 \Longrightarrow$
$y_3 = -\ y_1 - l * (x_3 - x_1) \Longrightarrow$
$P\ p_1\ p_2\ p_3$
shows $P\ p\ q\ (add\ a\ p\ q)$
⟨*proof*⟩

lemma *eq-opp-is-zero*: $((x::'a::ell\text{-}field) = -\ x) = (x = 0)$
⟨*proof*⟩

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*: $\bigwedge p.\ P\ Infinity\ p\ p$
and *R2*: $\bigwedge p.\ P\ p\ Infinity\ p$
and *R3*: $\bigwedge p.$ *on-curve a b p* $\Longrightarrow P\ p\ (opp\ p)\ Infinity$
and *R4*: $\bigwedge p_1\ x_1\ y_1\ p_2\ x_2\ y_2\ p_3\ x_3\ y_3\ l.$
$p_1 = Point\ x_1\ y_1 \Longrightarrow p_2 = Point\ x_2\ y_2 \Longrightarrow p_3 = Point\ x_3\ y_3 \Longrightarrow$
$p_3 = add\ a\ p_1\ p_2 \Longrightarrow p_1 \neq opp\ p_2 \Longrightarrow$
$x_1 = x_2 \wedge y_1 = y_2 \wedge l = (3 * x_1\ \hat{}\ 2 + a)\ /\ (2 * y_1) \vee$
$x_1 \neq x_2 \wedge l = (y_2 - y_1)\ /\ (x_2 - x_1) \Longrightarrow$
$x_3 = l\ \hat{}\ 2 - x_1 - x_2 \Longrightarrow$
$y_3 = -\ y_1 - l * (x_3 - x_1) \Longrightarrow$
$P\ p_1\ p_2\ p_3$
shows $P\ p\ q\ (add\ a\ p\ q)$
⟨*proof*⟩

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*:
$(a::'a::ring) \neq b \Longrightarrow a - b \neq 0$
$a \neq b \Longrightarrow b - a \neq 0$
⟨*proof*⟩

lemma *minus2-not0*: $(-2::'a::ell\text{-}field) \neq 0$
⟨*proof*⟩

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* p_1 (*add a* p_2 p_3) = *add a* (*add a* p_1 p_2) p_3
⟨*proof*⟩

lemma *spec2-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-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
⟨*proof*⟩

lemma *spec3-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-tangent* p_2 p_3
and *is-generic* (*add a* p_1 p_2) p_3
and *is-tangent* 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
⟨*proof*⟩

lemma *add-0-l*: *add a Infinity p* = *p*
⟨*proof*⟩

lemma *add-0-r*: *add a p Infinity* = *p*
⟨*proof*⟩

lemma *add-opp*: *on-curve a b p* ⟹ *add a p* (*opp p*) = *Infinity*
⟨*proof*⟩

lemma *add-comm*:
assumes *on-curve a b* p_1 *on-curve a b* p_2
shows *add a* p_1 p_2 = *add a* p_2 p_1
⟨*proof*⟩

lemma *uniq-opp*:
assumes *add a* p_1 p_2 = *Infinity*
shows p_2 = *opp* p_1
⟨*proof*⟩

lemma *uniq-zero*:
assumes *ab*: *nonsingular a b*
and p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
and *add*: *add a* p_1 p_2 = p_2
shows p_1 = *Infinity*
⟨*proof*⟩

lemma *opp-add*:
assumes p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
shows *opp* (*add a* p_1 p_2) = *add a* (*opp* p_1) (*opp* p_2)
⟨*proof*⟩

lemma *compat-add-opp*:
assumes p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
and *add a* p_1 p_2 = *add a* p_1 (*opp* p_2)
and $p_1 \neq$ *opp* p_1
shows p_2 = *opp* p_2
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *add-opp-double-opp*:
assumes *ab*: *nonsingular a b*
and p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
and *add a* p_1 p_2 = *opp* p_1
shows p_2 = *add a* (*opp* p_1) (*opp* p_1)
⟨*proof*⟩

lemma *cancel*:
assumes *ab*: *nonsingular a b*
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 *eq*: *add a* p_1 p_2 = *add a* p_1 p_3
shows $p_2 = p_3$
⟨*proof*⟩

lemma *add-minus-id*:
assumes *ab*: *nonsingular a b*
and p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
shows *add a* (*add a* p_1 p_2) (*opp* p_2) = p_1
⟨*proof*⟩

lemma *add-shift-minus*:
assumes *ab*: *nonsingular a b*
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 *eq*: *add a* p_1 p_2 = p_3
shows p_1 = *add a* p_3 (*opp* p_2)
⟨*proof*⟩

lemma *degen-assoc*:
assumes *ab*: *nonsingular a b*
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 *H*:
(p_1 = *Infinity* ∨ p_2 = *Infinity* ∨ p_3 = *Infinity*) ∨
(p_1 = *opp* p_2 ∨ p_2 = *opp* p_3) ∨
(*opp* p_1 = *add a* p_2 p_3 ∨ *opp* p_3 = *add a* p_1 p_2)
shows *add a* p_1 (*add a* p_2 p_3) = *add a* (*add a* p_1 p_2) p_3
⟨*proof*⟩

lemma *spec4-assoc*:
assumes *ab*: *nonsingular a b*
and p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
shows *add a* p_1 (*add a* p_2 p_2) = *add a* (*add a* p_1 p_2) p_2
⟨*proof*⟩

lemma *add-assoc*:
assumes *ab*: *nonsingular a b*
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

shows *add a* p_1 (*add a* p_2 p_3) = *add a* (*add a* p_1 p_2) p_3
⟨*proof*⟩

lemma *add-comm'*:
nonsingular a b ⟹
on-curve a b p_1 ⟹ *on-curve a b* p_2 ⟹ *on-curve a b* p_3 ⟹
add a p_2 (*add a* p_1 p_3) = *add a* p_1 (*add a* p_2 p_3)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *point-mult2-eq-double*:
point-mult a 2 p = *add a p p*
⟨*proof*⟩

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)
  ⟨proof⟩

lemma make-affine-infinity [simp]: make-affine (x, y, 0) = Infinity
  ⟨proof⟩

lemma on-curvep-iff-on-curve:
  on-curvep a b p = on-curve a b (make-affine p)
⟨proof⟩
```

lemma *pdouble-infinity* [*simp*]: *pdouble a* $(x, y, 0) = (x, y, 0)$
⟨*proof*⟩

lemma *padd-infinity-l* [*simp*]: *padd a* $(x, y, 0)$ *p* = *p*
⟨*proof*⟩

lemma *pdouble-correct*:
make-affine (*pdouble a p*) = *add a* (*make-affine p*) (*make-affine p*)
⟨*proof*⟩

lemma *padd-correct*:
assumes p_1: *on-curvep a b* p_1 **and** p_2: *on-curvep a b* p_2
shows *make-affine* (*padd a* p_1 p_2) = *add a* (*make-affine* p_1) (*make-affine* p_2)
⟨*proof*⟩

lemma *pdouble-closed*:
on-curvep a b p ⟹ *on-curvep a b* (*pdouble a p*)
⟨*proof*⟩

lemma *padd-closed*:
on-curvep a b p_1 ⟹ *on-curvep a b* p_2 ⟹ *on-curvep a b* (*padd a* p_1 p_2)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ppoint-mult-correct*: *on-curvep a b p* ⟹
make-affine (*ppoint-mult a n p*) = *point-mult a n* (*make-affine p*)
⟨*proof*⟩

context *ell-field* **begin**

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

end

lemma *proj-eq-refl*: *proj-eq p p*
⟨*proof*⟩

lemma *proj-eq-sym*: *proj-eq p p'* ⟹ *proj-eq p' p*
⟨*proof*⟩

lemma *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*⟩

lemma *make-affine-proj-eq-iff*:
proj-eq p p' = (*make-affine p* = *make-affine p'*)
⟨*proof*⟩

lemma *pdouble-proj-eq-cong*:
proj-eq p p' $\Longrightarrow$ *proj-eq* (*pdouble a p*) (*pdouble a p'*)
⟨*proof*⟩

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')
⟨*proof*⟩

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*⟩

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 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::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
  ⟨proof⟩

lemma opp-Infinity [simp]: opp Infinity = Infinity
  ⟨proof⟩

lemma opp-Point: opp (Point x y) = Point x (⊖ y)
  ⟨proof⟩

lemma opp-opp: on-curve a b p ⟹ opp (opp p) = p
  ⟨proof⟩

lemma opp-closed:
  on-curve a b p ⟹ on-curve a b (opp p)
  ⟨proof⟩

lemma curve-elt-opp:
  assumes p1 = Point x1 y1
```

and $p_2 = Point\ x_2\ y_2$
and *on-curve a b* p_1
and *on-curve a b* p_2
and $x_1 = x_2$
shows $p_1 = p_2 \vee p_1 = opp\ p_2$
⟨*proof*⟩

lemma *add-closed*:
assumes $a \in carrier\ R$ **and** $b \in carrier\ R$
and *on-curve a b* p_1 **and** *on-curve a b* p_2
shows *on-curve a b* (*add a* p_1 p_2)
⟨*proof*⟩

lemma *add-case* [*consumes 4*, *case-names InfL InfR Opp Tan Gen*]:
assumes $a \in carrier\ R$
and $b \in carrier\ R$
and *p*: *on-curve a b p*
and *q*: *on-curve a b q*
and *R1*: $\bigwedge p.\ P\ Infinity\ p\ p$
and *R2*: $\bigwedge p.\ P\ p\ Infinity\ p$
and *R3*: $\bigwedge p.$ *on-curve a b p* $\Longrightarrow P\ p\ (opp\ p)\ Infinity$
and *R4*: $\bigwedge p_1\ x_1\ y_1\ p_2\ x_2\ y_2\ l.$
$p_1 = Point\ x_1\ y_1 \Longrightarrow p_2 = Point\ x_2\ y_2 \Longrightarrow$
$p_2 = add\ a\ p_1\ p_1 \Longrightarrow y_1 \neq \mathbf{0} \Longrightarrow$
$l = («3» \otimes x_1\ [\widehat{}]\ (2{::}nat) \oplus a) \oslash («2» \otimes y_1) \Longrightarrow$
$x_2 = l\ [\widehat{}]\ (2{::}nat) \ominus «2» \otimes x_1 \Longrightarrow$
$y_2 = \ominus\ y_1 \ominus l \otimes (x_2 \ominus x_1) \Longrightarrow$
$P\ p_1\ p_1\ p_2$
and *R5*: $\bigwedge p_1\ x_1\ y_1\ p_2\ x_2\ y_2\ p_3\ x_3\ y_3\ l.$
$p_1 = Point\ x_1\ y_1 \Longrightarrow p_2 = Point\ x_2\ y_2 \Longrightarrow p_3 = Point\ x_3\ y_3 \Longrightarrow$
$p_3 = add\ a\ p_1\ p_2 \Longrightarrow x_1 \neq x_2 \Longrightarrow$
$l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1) \Longrightarrow$
$x_3 = l\ [\widehat{}]\ (2{::}nat) \ominus x_1 \ominus x_2 \Longrightarrow$
$y_3 = \ominus\ y_1 \ominus l \otimes (x_3 \ominus x_1) \Longrightarrow$
$P\ p_1\ p_2\ p_3$
shows $P\ p\ q\ (add\ a\ p\ q)$
⟨*proof*⟩

lemma *add-casew* [*consumes 4*, *case-names InfL InfR Opp Gen*]:
assumes *a*: $a \in carrier\ R$
and *b*: $b \in carrier\ R$
and *p*: *on-curve a b p*
and *q*: *on-curve a b q*
and *R1*: $\bigwedge p.\ P\ Infinity\ p\ p$
and *R2*: $\bigwedge p.\ P\ p\ Infinity\ p$
and *R3*: $\bigwedge p.$ *on-curve a b p* $\Longrightarrow P\ p\ (opp\ p)\ Infinity$
and *R4*: $\bigwedge p_1\ x_1\ y_1\ p_2\ x_2\ y_2\ p_3\ x_3\ y_3\ l.$
$p_1 = Point\ x_1\ y_1 \Longrightarrow p_2 = Point\ x_2\ y_2 \Longrightarrow p_3 = Point\ x_3\ y_3 \Longrightarrow$
$p_3 = add\ a\ p_1\ p_2 \Longrightarrow p_1 \neq opp\ p_2 \Longrightarrow$

$x_1 = x_2 \wedge y_1 = y_2 \wedge l = (\text{«}3\text{»} \otimes x_1\ [\widehat{}]\ (2{::}nat) \oplus a) \oslash (\text{«}2\text{»} \otimes y_1) \vee$
$x_1 \neq x_2 \wedge l = (y_2 \ominus y_1) \oslash (x_2 \ominus x_1) \Longrightarrow$
$x_3 = l\ [\widehat{}]\ (2{::}nat) \ominus x_1 \ominus x_2 \Longrightarrow$
$y_3 = \ominus\ y_1 \ominus l \otimes (x_3 \ominus x_1) \Longrightarrow$
$P\ p_1\ p_2\ p_3$
shows $P\ p\ q\ (add\ a\ p\ q)$
⟨*proof*⟩

definition
$is\text{-}tangent\ p\ q = (p \neq Infinity \wedge p = q \wedge p \neq opp\ q)$

definition
$is\text{-}generic\ p\ q =$
$(p \neq Infinity \wedge q \neq Infinity \wedge$
$p \neq q \wedge p \neq opp\ q)$

lemma *spec1-assoc*:
assumes a: $a \in carrier\ R$
and b: $b \in carrier\ R$
and p_1: $on\text{-}curve\ a\ b\ p_1$
and p_2: $on\text{-}curve\ a\ b\ p_2$
and p_3: $on\text{-}curve\ a\ b\ p_3$
and $is\text{-}generic\ p_1\ p_2$
and $is\text{-}generic\ p_2\ p_3$
and $is\text{-}generic\ (add\ a\ p_1\ p_2)\ p_3$
and $is\text{-}generic\ p_1\ (add\ a\ p_2\ p_3)$
shows $add\ a\ p_1\ (add\ a\ p_2\ p_3) = add\ a\ (add\ a\ p_1\ p_2)\ p_3$
⟨*proof*⟩

lemma *spec2-assoc*:
assumes a: $a \in carrier\ R$
and b: $b \in carrier\ R$
and p_1: $on\text{-}curve\ a\ b\ p_1$
and p_2: $on\text{-}curve\ a\ b\ p_2$
and p_3: $on\text{-}curve\ a\ b\ p_3$
and $is\text{-}generic\ p_1\ p_2$
and $is\text{-}tangent\ p_2\ p_3$
and $is\text{-}generic\ (add\ a\ p_1\ p_2)\ p_3$
and $is\text{-}generic\ p_1\ (add\ a\ p_2\ p_3)$
shows $add\ a\ p_1\ (add\ a\ p_2\ p_3) = add\ a\ (add\ a\ p_1\ p_2)\ p_3$
⟨*proof*⟩

lemma *spec3-assoc*:
assumes a: $a \in carrier\ R$
and b: $b \in carrier\ R$
and p_1: $on\text{-}curve\ a\ b\ p_1$
and p_2: $on\text{-}curve\ a\ b\ p_2$
and p_3: $on\text{-}curve\ a\ b\ p_3$
and $is\text{-}generic\ p_1\ p_2$

and *is-tangent* p_2 p_3
and *is-generic* (*add* a p_1 p_2) p_3
and *is-tangent* 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
⟨*proof*⟩

lemma *add-0-l*: *add* a *Infinity* p = p
⟨*proof*⟩

lemma *add-0-r*: *add* a p *Infinity* = p
⟨*proof*⟩

lemma *add-opp*: *on-curve* a b p ⟹ *add* a p (*opp* p) = *Infinity*
⟨*proof*⟩

lemma *add-comm*:
assumes a ∈ *carrier* R b ∈ *carrier* R *on-curve* a b p_1 *on-curve* a b p_2
shows *add* a p_1 p_2 = *add* a p_2 p_1
⟨*proof*⟩

lemma *uniq-opp*:
assumes *on-curve* a b p_2
and *add* a p_1 p_2 = *Infinity*
shows p_2 = *opp* p_1
⟨*proof*⟩

lemma *uniq-zero*:
assumes a: a ∈ *carrier* R
and b: b ∈ *carrier* R
and *ab*: *nonsingular* a b
and p_1: *on-curve* a b p_1
and p_2: *on-curve* a b p_2
and *add*: *add* a p_1 p_2 = p_2
shows p_1 = *Infinity*
⟨*proof*⟩

lemma *opp-add*:
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
shows *opp* (*add* a p_1 p_2) = *add* a (*opp* p_1) (*opp* p_2)
⟨*proof*⟩

lemma *compat-add-opp*:
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 *add a* p_1 p_2 = *add a* p_1 (*opp* p_2)
and $p_1 \neq$ *opp* p_1
shows p_2 = *opp* p_2
⟨*proof*⟩

lemma *compat-add-triple*:
assumes *a*: $a \in$ *carrier R*
and *b*: $b \in$ *carrier R*
and *ab*: *nonsingular a b*
and *p*: *on-curve a b p*
and $p \neq$ *opp p*
and *add a p p* $\neq$ *opp p*
shows *add a* (*add a p p*) (*opp p*) = *p*
⟨*proof*⟩

lemma *add-opp-double-opp*:
assumes *a*: $a \in$ *carrier R*
and *b*: $b \in$ *carrier R*
and *ab*: *nonsingular a b*
and p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
and *add a* p_1 p_2 = *opp* p_1
shows p_2 = *add a* (*opp* p_1) (*opp* p_1)
⟨*proof*⟩

lemma *cancel*:
assumes *a*: $a \in$ *carrier R*
and *b*: $b \in$ *carrier R*
and *ab*: *nonsingular a b*
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 *eq*: *add a* p_1 p_2 = *add a* p_1 p_3
shows $p_2 = p_3$
⟨*proof*⟩

lemma *add-minus-id*:
assumes *a*: $a \in$ *carrier R*
and *b*: $b \in$ *carrier R*
and *ab*: *nonsingular a b*
and p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
shows *add a* (*add a* p_1 p_2) (*opp* p_2) = p_1
⟨*proof*⟩

lemma *add-shift-minus*:
assumes *a*: $a \in$ *carrier R*
and *b*: $b \in$ *carrier R*
and *ab*: *nonsingular a b*

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 *eq*: *add a* p_1 $p_2 = p_3$
shows $p_1 =$ *add a* p_3 (*opp* p_2)
⟨*proof*⟩

lemma *degen-assoc*:
assumes *a*: $a \in$ *carrier R*
and *b*: $b \in$ *carrier R*
and *ab*: *nonsingular a b*
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 *H*:
$(p_1 =$ *Infinity* $\vee$ $p_2 =$ *Infinity* $\vee$ $p_3 =$ *Infinity*$) \vee$
$(p_1 =$ *opp* $p_2 \vee$ $p_2 =$ *opp* $p_3) \vee$
(*opp* $p_1 =$ *add a* p_2 $p_3 \vee$ *opp* $p_3 =$ *add a* p_1 p_2)
shows *add a* p_1 (*add a* p_2 p_3) = *add a* (*add a* p_1 p_2) p_3
⟨*proof*⟩

lemma *spec4-assoc*:
assumes *a*: $a \in$ *carrier R*
and *b*: $b \in$ *carrier R*
and *ab*: *nonsingular a b*
and p_1: *on-curve a b* p_1
and p_2: *on-curve a b* p_2
shows *add a* p_1 (*add a* p_2 p_2) = *add a* (*add a* p_1 p_2) p_2
⟨*proof*⟩

lemma *add-assoc*:
assumes *a*: $a \in$ *carrier R*
and *b*: $b \in$ *carrier R*
and *ab*: *nonsingular a b*
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
shows *add a* p_1 (*add a* p_2 p_3) = *add a* (*add a* p_1 p_2) p_3
⟨*proof*⟩

lemma *add-comm'*:
$a \in$ *carrier R* $\Longrightarrow$ $b \in$ *carrier R* $\Longrightarrow$ *nonsingular a b* $\Longrightarrow$
on-curve a b $p_1 \Longrightarrow$ *on-curve a b* $p_2 \Longrightarrow$ *on-curve a b* $p_3 \Longrightarrow$
add a p_2 (*add a* p_1 p_3) = *add a* p_1 (*add a* p_2 p_3)
⟨*proof*⟩

primrec *point-mult* :: *'a* $\Rightarrow$ *nat* $\Rightarrow$ *'a point* $\Rightarrow$ *'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)
⟨*proof*⟩

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)
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *point-mult2-eq-double*:
point-mult a 2 p = add a p p
⟨*proof*⟩

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 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::nat) ⊗ z1 ⊗ z2 ⊖ (d1 ⊕ d2) ⊗ l [^] (2::nat)
          in
            (l ⊗ h,
             (d2 ⊗ l [^] (2::nat) ⊖ h) ⊗ m ⊖ l [^] (3::nat) ⊗ y1 ⊗ z2,
             l [^] (3::nat) ⊗ z1 ⊗ z2))

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)
  ⟨proof⟩

lemma (in field) make-affine-infinity [simp]: make-affine (x, y, 0) = Infinity
  ⟨proof⟩

lemma (in cring) on-curvep-imp-in-carrierp [simp]: on-curvep a b p ⟹ in-carrierp
p
  ⟨proof⟩

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)
  ⟨proof⟩

lemma (in cring) pdouble-in-carrierp:
```

$a \in carrier\ R \Longrightarrow in\text{-}carrierp\ p \Longrightarrow in\text{-}carrierp\ (pdouble\ a\ p)$
⟨*proof*⟩

lemma (**in** *cring*) *padd-in-carrierp*:
$a \in carrier\ R \Longrightarrow in\text{-}carrierp\ p_1 \Longrightarrow in\text{-}carrierp\ p_2 \Longrightarrow in\text{-}carrierp\ (padd\ a\ p_1\ p_2)$
⟨*proof*⟩

lemma (**in** *cring*) *pdouble-infinity* [*simp*]: $pdouble\ a\ (x,\ y,\ \mathbf{0}) = (x,\ y,\ \mathbf{0})$
⟨*proof*⟩

lemma (**in** *cring*) *padd-infinity-l* [*simp*]: $padd\ a\ (x,\ y,\ \mathbf{0})\ p = p$
⟨*proof*⟩

lemma (**in** *ell-field*) *pdouble-correct*:
$a \in carrier\ R \Longrightarrow in\text{-}carrierp\ p \Longrightarrow$
$make\text{-}affine\ (pdouble\ a\ p) = add\ a\ (make\text{-}affine\ p)\ (make\text{-}affine\ p)$
⟨*proof*⟩

lemma (**in** *ell-field*) *padd-correct*:
assumes *a*: $a \in carrier\ R$ **and** *b*: $b \in carrier\ R$
and p_1: $on\text{-}curvep\ a\ b\ p_1$ **and** p_2: $on\text{-}curvep\ a\ b\ p_2$
shows $make\text{-}affine\ (padd\ a\ p_1\ p_2) = add\ a\ (make\text{-}affine\ p_1)\ (make\text{-}affine\ p_2)$
⟨*proof*⟩

lemma (**in** *ell-field*) *pdouble-closed*:
assumes $a \in carrier\ R\ b \in carrier\ R\ on\text{-}curvep\ a\ b\ p$
shows $on\text{-}curvep\ a\ b\ (pdouble\ a\ p)$
⟨*proof*⟩

lemma (**in** *ell-field*) *padd-closed*:
assumes $a \in carrier\ R\ b \in carrier\ R\ on\text{-}curvep\ a\ b\ p_1\ on\text{-}curvep\ a\ b\ p_2$
shows $on\text{-}curvep\ a\ b\ (padd\ a\ p_1\ p_2)$
⟨*proof*⟩

primrec (**in** *cring*) $ppoint\text{-}mult :: 'a \Rightarrow nat \Rightarrow 'a\ ppoint \Rightarrow 'a\ ppoint$
where
$ppoint\text{-}mult\ a\ 0\ p = (\mathbf{0},\ \mathbf{0},\ \mathbf{0})$
| $ppoint\text{-}mult\ a\ (Suc\ n)\ p = padd\ a\ p\ (ppoint\text{-}mult\ a\ n\ p)$

lemma (**in** *ell-field*) *ppoint-mult-closed* [*simp*]:
$a \in carrier\ R \Longrightarrow b \in carrier\ R \Longrightarrow on\text{-}curvep\ a\ b\ p \Longrightarrow on\text{-}curvep\ a\ b\ (ppoint\text{-}mult\ a\ n\ p)$
⟨*proof*⟩

lemma (**in** *ell-field*) *ppoint-mult-correct*: $a \in carrier\ R \Longrightarrow b \in carrier\ R \Longrightarrow on\text{-}curvep\ a\ b\ p \Longrightarrow$
$make\text{-}affine\ (ppoint\text{-}mult\ a\ n\ p) = point\text{-}mult\ a\ n\ (make\text{-}affine\ p)$
⟨*proof*⟩

definition (**in** *cring*) *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)

lemma (**in** *cring*) *proj-eq-refl*: *proj-eq p p*
⟨*proof*⟩

lemma (**in** *cring*) *proj-eq-sym*: *proj-eq p p'* ⟹ *proj-eq p' p*
⟨*proof*⟩

lemma (**in** *domain*) *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*⟩

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*⟩

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'*)
⟨*proof*⟩

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')
⟨*proof*⟩

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 $x --_m y = (x - y) \bmod m$

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

lemma (**in** *residues*) *res-of-natural-eq*: $\langle n \rangle_{\mathbb{N}} = int\ n \bmod m$
⟨*proof*⟩

lemma (**in** *residues*) *res-of-integer-eq*: $\langle i \rangle = i \bmod m$
⟨*proof*⟩

lemma (**in** *residues*) *res-pow-eq*: $x\ [\widehat{\ }]\ (n::nat) = x \mathbin{\widehat{\ }} n \bmod m$
⟨*proof*⟩

lemma (**in** *residues*) *res-sub-eq*: $(x \bmod m) \ominus (y \bmod m) = (x \bmod m - y \bmod m) \bmod m$
⟨*proof*⟩

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* $\mathbin{\widehat{\ }}_m$ *2* $+_m$ *a* $**_m$ *z* $\mathbin{\widehat{\ }}_m$ *2*
in
(*l* $**_m$ (*n* $\mathbin{\widehat{\ }}_m$ *2* $--_m$ *4 mod m* $**_m$ *x* $**_m$ *y* $**_m$ *l*),
n $**_m$ (*6 mod m* $**_m$ *x* $**_m$ *y* $**_m$ *l* $--_m$ *n* $\mathbin{\widehat{\ }}_m$ *2*) $--_m$
2 mod m $**_m$ *y* $\mathbin{\widehat{\ }}_m$ *2* $**_m$ *l* $\mathbin{\widehat{\ }}_m$ *2*,
l $\mathbin{\widehat{\ }}_m$ *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 p₁
        else (0, 0, 0)
      else
        let h = n ^^^_m 2 **_m z₁ **_m z₂ --_m (d₁ ++_m d₂) **_m l ^^^_m 2
        in
          (l **_m h,
           (d₂ **_m l ^^^_m 2 --_m h) **_m n --_m l ^^^_m 3 **_m y₁ **_m z₂,
           l ^^^_m 3 **_m z₁ **_m z₂))
```

lemma (**in** *residues*) *pdouble-residue-eq*: *pdouble a p = mpdouble m a p*
⟨*proof*⟩

lemma (**in** *residues*) *padd-residue-eq*: *padd a p₁ p₂ = mpadd m a p₁ p₂*
⟨*proof*⟩

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)*
⟨*proof*⟩

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)
⟨*proof*⟩

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))
⟨*proof*⟩

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

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

sublocale *residues-prime-gt2 < ell-field*
⟨*proof*⟩

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)*

⟨*proof*⟩

lemma (**in** *residues-prime-gt2*) *point-mult-residue-eq*:
assumes $a \in$ *carrier R* $b \in$ *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*⟩

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*⟩

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 $\hat{}\hat{}\hat{}_m$ *2* = *x* $\hat{}\hat{}\hat{}_m$ *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*
⟨*proof*⟩

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 = *0x01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF*

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 = *0x01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF*

lemma *mon-curve m a b* (*Point gx gy*)
⟨*proof*⟩

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

lemma *mmake-affine m* (*fast-ppoint-mult m a order* (*gx*, *gy*, *1*)) = *Infinity*
⟨*proof*⟩

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.