Master de Mathématiques Fondamentales et Protection de l'Information

Université de Vincennes-Saint Denis - Paris VIII

Formalisation des Courbes Elliptiques en Coq

Evmorfia-Iro Bartzia

Soutenu le 12 octobre 2011
devant la commission d'examen composée de:

Claude Carlet	Paris VIII
Philippe Guillot	Paris VIII
Duong Hieu Phan	Paris VIII

Sous la direction de:

Karthikeyan Bhargavan	INRIA
Cédric Fournet	Microsoft Research Cambridge
Pierre-Yves Strub	MSR-INRIA

Contents

1 Introduction **1**

2 A brief introduction to elliptic curves **3**
2.1 Elliptic curve as a group 4
2.2 The theorem of Cassel 4

3 A brief introduction to Coq **9**
3.1 Coq by example 10
3.2 Propositions and Types 11
3.3 The Small Scale Reflection extension 13
3.4 Conclusion 15

4 The Coq formalization **17**
4.1 The elliptic curve 17
4.2 Rational functions - Order 23

5 Conclusion **31**

Bibliography **33**

1 Introduction

The theory of elliptic curves is a subject where one finds diverse branches of mathematics, such as, complex analysis, algebraic geometry, representation theory and number theory. It has been an active field of study since the 19th century. Elliptic curves have been used to approach a wide range of problems such as the fast factorisation of integers and the search for congruent numbers. In the 20th century, they have regained interest because of their applications in cryptography, first suggested in 1985 independently by Neal Koblitz and Victor Miller. Their use in cryptography relies principally on the existence of a group law for the set of points on an elliptic curve. This group is a very good candidate for cryptographic groups used in public key cryptography because of the hardness of the Discrete Logarithm Problem on it (relative to the parameters of the curve). Elliptic curves also allow the definition of new cryptographic primitives, such as identity-based encryption, based on bilinear (Weil and Tate) pairings. In any case, elliptic curve theory remains a subject of active research because of its wide range of applications, as well as the rich mathematics behind it.

This work is concerned with proving the security of cryptographic schemes based on elliptic curves. Proving security for cryptographic schemes can be an extremely delicate task because of the level of details that the proof requires, as well as the broad set of concepts and reasoning methods from complexity, probability and group theory that are involved. Machine-checked proofs are often used in cryptography in order to increase confidence in cryptographic proofs. For instance, CertiCrypt [3, 1] is a fully machine-checked framework for developing and verifying cryptographic game-playing proofs, built on top of the Coq proof assistant. Up to date, it has been used to formally prove the security of well known cryptographic schemes such as the OAEP padding, hashed El Gamal encryption and Full Domain Hash digital signatures. CertiCrypt uses a formalization of elliptic curves developed by Laurent Therry [19]. However, this formalisation lacks many intermediate results such as the Cassel theorem [2]. Our work aims to close this gap.

We base our work on the Coq SsReflect extension, built by the mathematical components team at MSR-INRIA. The objective of their project is to develop a general platform for the formalization of mathematical theories, and in particular, a theory of efficient arithmetic and a proof of the Odd Order theorem.

Our work stands between CertiCrypt and SsReflect; we use the mathematical theories formalized in SsReflect to develop a new formalization of elliptic curves that enable new cryptographic proofs in CertiCrypt. Our formalization extends Laurent Therry's work with new definitions and lemmas that enable us to prove several important results towards the Cassel Theorem. Our results can be seen as a validation of both the SsReflect libraries and the CertiCrypt method. We believe they offer a significant contribution towards a formal theory of elliptic curves.

The rest of the document proceeds as follows:

Chapter 2 introduces elliptic curves for cryptography and the necessary definitions, lemmas

and theorems that are used in the proof of the Cassel theorem.

Chapter 3 introduces the Coq proof assistant and some prerequisites in order to understand the formalisation that follows.

Chapter 4 describes our Coq formalisation of elliptic curves and the results that have been proved to date.

Chapter 5 concludes.

2

A brief introduction to elliptic curves

We define elliptic curves as follows, based on [5, 12].

Let K be field of characteristic different from 2 and 3.

Definition 2.0.1 (Elliptic Curve)**.** An elliptic curve over K is defined as the set

$$E = \{(x, y) \in K^2 | y^2 = x^3 + Ax + B\} \cup \{\mathcal{P}_\infty\}$$

where $A, B \in K$ (the *parameters*) s.t. the discriminant $\Delta = 4A^3 + 27B^2 \neq 0$ and $\mathcal{P}_\infty$ is a unique and distinguished element called the *point at infinity.* The points on the curve other that the infinite point are called *finite*.

The set of points of an elliptic curve, together with its internal law, form an abelian group. This group is a natural candidate for use in public key cryptosystems exactly because (1) the group operation and exponentiation are computationally efficient and (2) the Discrete Logarithm Problem is presumed hard relative to the size of the parameters. To decide the key size for a public key cryptosystem, one may look at the security provided by the symmetric encryption algorithm (such as AES) that uses the public key algorithm to encrypt or for authentication. Elliptic Curve based systems offer an important advantage on security, relatively to the size of the parameters used. From NSA[1] : *To use RSA or Diffie-Hellman to protect 128-bit AES keys one should use 3072-bit parameters: three times the size in use throughout the Internet today. The equivalent key size for elliptic curves is only 256 bits.*

A second application of elliptic curves is in Pairing-based Cryptography. The main idea is to construct a bilinear mapping $G_1 \times G_1 \to G_2$ between two efficient cryptographic groups G_1, G_2, that allows the reduction of a problem in one group to a different problem to the other group. In practice, G_1 is an elliptic curve group and G_2 a finite field. In G_1 the Decisional Diffie-Helman problem is easy because it reduces to an easy problem on G_2, but the Computational Diffie-Helman problem remains hard. Several interesting cryptographic schemes have been based on the *Weil* and the *Tate* pairing, such as the 3-part key agreement scheme proposed by [13] and the identity-based encryption proposed by [15].

From now on, let K be field of characteristic different from 2 and 3. Let E be an elliptic curve on K with parameters A and B. The definitions below are based on [5, 12].

[1] www.nsa.gov/business/programs/elliptic_curve.shtml

2.1 Elliptic curve as a group

The points of an elliptic curve form an abelian group. Addition is based on the main idea that a line intersecting the elliptic curve intersects it at most three times, as stated by the Bezout theorem for algebraic closed fields:

Every line intersects with an elliptic curve in three points that may coincide.

We can then define an internal operation (written +) on the points of the curve respecting the following rules:

1. $\mathcal{P}_\infty$ is the neutral element: $\forall P.\, P + \mathcal{P}_\infty = \mathcal{P}_\infty + P = P$,
2. the opposite of a point (x_P, y_P) (resp. $\mathcal{P}_\infty$) is $(x_P, -y_P)$ (resp. $\mathcal{P}_\infty$), and
3. if three points of the curve are aligned, their sum is equal to zero.

We exemplify this property in Figure 2.1 where $K = \mathbb{R}$.

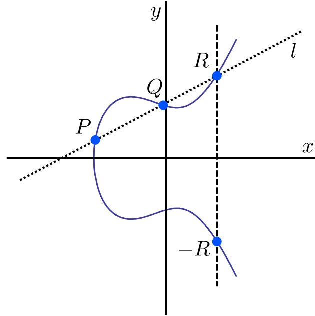

Figure 2.1: Addition over an elliptic curve

Let P and Q be the finite points on the curve given in Figure 2.1. Let l be the line which goes through P and Q. Let R be the third point where the curve intersects l. The coordinates of the point R can be computed using the equation of l and the parameters of the curve. The symmetric $-R$ (for the x-axis) of R - which is on the curve - is defined as the sum of P and Q. Note that when $P = -Q$, l intersects the curve at the infinite point. In the degenerated case where $P = Q$, we find R by intersecting the curve and the tangent of the curve at P. Here too, when $P = (x, 0)$, R is the infinite point.

The formal definition of the addition is given in Chapter 4.

2.2 The theorem of Cassel

Our purpose is to formalize the proof of the following theorem:

Theorem 2.2.1 (Cassel)**.** *The group of points of an elliptic curve over a finite field K is either a cyclic group or isomorphic to a direct product $\mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/m\mathbb{Z}$ with $n|m$.*

The proof of the Cassel theorem is implied by the *Fundamental Theorem of finite abelian groups*:

Lemma 2.2.2. *Every finite abelian group is isomorphic to a direct product:*

$$\mathbb{Z}/n_1\mathbb{Z} \times \mathbb{Z}/n_2\mathbb{Z}/ \times \mathbb{Z}/n_3\mathbb{Z} \times \cdots \times \mathbb{Z}/n_k\mathbb{Z}$$

where n_i/n_{i+1}.

and the following theorem:

Lemma 2.2.3 (Cardinal of the n-torsion)**.** *Let E be an elliptic curve over K.*

Let $E[n] = \{P \in E \mid nP = \mathcal{P}_\infty\}$. $E[n]$ is a finite subgroup of E called the n-torsion group.

If the characteristic of the field K does not divide n, then $|E[n]| \le n^2$ where $|E[n]|$ is the cardinal of $E[n]$.

If the field is algebraically closed, this bound is reached. For example, assume that $n = 2$. If the field is algebraically closed the polynomial $x^3 + Ax + B$ has three roots a, b and c. This roots must be distinct as implied by the requirement on the discriminant. Hence, $y^2 = (x - a)(x - b)(x - c)$ and the finite points on the curve whose second coordinate is equal to 0 are $(a, 0)$, $(b, 0)$ and $(c, 0)$. As a result, $E[2] = \{\mathcal{P}_\infty, (a, 0), (b, 0), (c, 0)\}$. The proof of the theorem in the general case is done by induction on n (for details see [12]).

Before proving the theorem 2.2.3, several notions have to be defined in order to prove the two following intermediate results:

Lemma 2.2.4. *The sum of the order of a rational function (a notion to be defined later) over the points of an elliptic curve is non-positive.*

Lemma 2.2.5. *Let $m, n \in \mathbb{N}$ s.t. $m > n > 0$. Assume that m, n, $m+n$ and $m-n$ are all primes to the characteristic of the field. Then:*

$$\mathrm{div}(r_m - r_n) = \langle E[m + n]\rangle + \langle E[m - n]\rangle - 2\langle E[m]\rangle - 2\langle E[m]\rangle$$

where i) r_k is the rational function that computes the first coordinate of $[k]$, the multiplication of points by $k \in \mathbb{N}$, ii) $\mathrm{div}(f)$ denotes the divisor of the function f, and iii) $\langle A\rangle$ denotes the divisor of the set A. All these notions are defined below.

We now detail the tools and intermediate results needed for the proofs of the above lemmas.

Sum of the order along a curve

The purpose of this section is to define the notion of *rational functions* and *order*.

Consider the ring of bi-variate polynomials $K[x, y]$ with coefficients in K. We want to identify the polynomials which evaluate to the same value for every point of the curve. For example, y^2 and $x^3 + Ax + B$ are such polynomials. For that, we quotient $K[x, y]$ with the following relation of equivalence:

$$p \sim q \Leftrightarrow \exists k \in K[x, y].\, p - q = k(y^2 - x^3 - Ax - B)$$

We write $K[E]$ the obtained structure. $K[E]$ can be seen as the ring quotient of $K[x, y]$ with the irreducible ideal generated by $y^2 - \mathrm{C_X}$ (where $\mathrm{C_X} = x^3 + Ax + B$). Since $\mathrm{C_X} = x^3 + Ax + B$ is irreducible, $K[E]$ is an integral ring. Elements of $K[E]$ are called *polynomials on the curve.*

It is natural to distinguish a canonical witness for every classes of $K[E]$, called *reduced form.* Let p be a polynomial in $K[E]$. On the curve we have the relation $y^2 = x^3 + Ax + B$ that relates

the coordinates x and y. As a result, we can replace every occurence of y^2 in p with $x^3 + Ax + B$. This way, we obtain a unique witness $p_1(x) + yp_2(x)$ of the class of p. This witness is called the *reduced form* .

We can then define the following functions over $K[E]$:

Definition 2.2.6. Let $p(x, y) = p_1(x) + yp_2(x)$.

- The *conjugate* of p, written $\overline{p}$, is defined as $\overline{p}(x, y) = p_1(x) - yp_2(x)$,
- The *norm* of p, written $\mathrm{norm}(p)$, is defined as $\mathrm{norm}(p) = p\overline{p}$.
  The norm is a univariate polynomial on x:
  $$\mathrm{norm}(p) = p_1(x)^2 - p_2(x)^2 \cdot (x^3 + Ax + B),$$
- The *degree* of p, written $\deg(p)$, is defined as $\deg(\mathrm{norm}(p))$.

We have the following properties:

Lemma 2.2.7 (Properties of the norm)**.**

- *The* norm *is multiplicative:* $\mathrm{norm}(fg) = \mathrm{norm}(f)\,\mathrm{norm}(g)$,
- $\mathrm{norm}(p) = 0 \Leftrightarrow p = 0$,

The ring $K[E]$ being an integral ring, one can define the set of rational functions as the field of fractions on $K[E]$.

Definition 2.2.8 (Field of rational functions)**.** The field of rational functions, written $K(E)$, is defined as the field of fractions on $K[E]$.

We can then move to the definition of the *evaluation of a rational function*:

Definition 2.2.9 (Evaluation)**.** Let $r \in K(E)$ and P be a point on the curve. The evaluation of r at P, written $r(P)$, is defined as follows:

- if P is finite, then:
  - if there exists a witness n/d in the class of r s.t. $d(P) \neq 0$, then $r(P) = n(P)/d(P)$,
  - otherwise, $r(P) = \infty$
- if $P = \mathcal{P}_\infty$, then let n/d be any witness in the class of r. Then:
  $$r(P) = \begin{cases} 0 & \text{when } \deg(n) < \deg(d) \\ \infty & \text{when } \deg(n) > \deg(d) \\ \alpha_n/\alpha_d & \text{when } \deg(n) = \deg(d) \end{cases}$$
  where α_n (resp. α_d) is the coefficient of the higher degree term of n (resp. d).

We say that the point P is a *pole of* r (resp. *a zero of* r, *a regular point for* r) if $r(P) = \infty$ (resp $r(P) = 0$, $r(P) \neq \infty$).

One can check that the evaluation is well-defined as the choice of the witness does not impact the result.

Before defining the *order* of a rational function, we need to define a last notion:

Definition 2.2.10 (Uniformizer)**.** Let P be a point on the curve. A *uniformizer at* P is a rational function $u \in K(E)$ s.t. i) $u(P) = 0$, and ii) for any non-null rational function r, there exists $d \in \mathbb{Z}$, $s \in K(E)$ s.t. $r = u^d s$ and $s(P) \neq 0, \infty$.

There exists a uniformizer everywhere on the curve:

Lemma 2.2.11. *For every point P on the curve, there exists a uniformizer at P.*

For a fixed point P, the uniformizer is not unique. On the contrary, the integer d appearing in the decomposition of the rational function r in Definition 2.2.10 does not depend on the chosen uniformizer and is unique. This allows the following definition:

Definition 2.2.12 (Order)**.** Let P be a point on the curve and f be a rational function of $F(E)$. The *order of f at P*, written $\mathrm{ord}_P(f)$, is the unique integer $d \in \mathbb{Z}$ s.t. for any uniformizer u, r can be written $r = u^d g$ with $g(p) \neq 0, \infty$.

One may understand the order as the equivalent of multiplicity for bi-variate polynomials on the curve. Indeed, the order as defined below satisfies the following property. Let P point on the curve. Then, every polynomial $f \in K[E]$ can be written $f = u^d \cdot s$ where $d \in \mathbb{N}$ (d is the *order*) and $u, s \in K(E)$ s.t. $u(P) = 0$ and $s(P) \neq 0, \infty$. For instance, if P is not a zero nor a pole of f, then the order is zero. If P is a zero (resp. a pole) of f, then the order is positive (resp. negative).

Lemma 2.2.13. *For all P on the curve, forall $f \in K[E]$, $\mathrm{ord}_P(f) = \mathrm{ord}_{-P}(\overline{f})$.*

Using the above result, as well as the fact that a rational function has a finite number of zeros and poles, one can prove the lemma 2.2.4 stated as follows:

Lemma 2.2.14. *For any element f of $F(E)$, $\sum_{P \in E} \mathrm{ord}_P(f) \leq 0$. Moreover, if K is algebraically closed, the bound is reached.*

The detailed proof is given in [12].

The notions *rational function* and *order* will be further explained in Chapter 4. For the moment, it if sufficient to understand that we identify bi-variate polynomials that have the same value on the points of the curve, and that we define a function *order* which will be the equivalent of multiplicity, for bi-variate polynomials on the curve. Our work was concentrated on formalizing the proof of Lemma 2.2.4. At the time of writing, we formalized the previous definitions for polynomials.

Divisor of a rational function

Definition 2.2.15 (Divisor of a Rational Function)**.** The divisor of a rational function $f \in K(E)$ is defined as the formal sum

$$\mathrm{div}(f) = \sum_{P \in E} \mathrm{ord}_P(f)(P).$$

The divisor of a set A of points of the curve is defined as $\langle A \rangle = \sum_{P \in A}(P)$.

Definition 2.2.16 (Isogeny)**.** Let E_1, E_2 be two elliptic curves on F s.t.

- $E_1 = \{(x, y) \in K^2 \mid y^2 = x^3 + A_1 x + B_1\} \cup \{\mathcal{P}^1_\infty\}$
- $E_2 = \{(x, y) \in K^2 \mid y^2 = x^3 + A_2 x + B_2\} \cup \{\mathcal{P}^2_\infty\}$

An isogeny is a function $\phi : E_1 \to E_2$ s.t i) for any finite point P, $\phi(P) = (r(P), s(P))$ where $r, s \in K(E)$, and ii) $\phi(\mathcal{P}^1_\infty) = \mathcal{P}^2_\infty$.

A remarkable property of an isogeny g is that it can be expressed as $g(x, y) = (r(x), y \cdot s(x))$ where r, s are rational functions depending solely on x.

The multiplication by $n \in \mathbb{N}$, written $[n]$, is an isogeny from E to E. It can be expressed as $[n] = (r_n(x), ys_n(x))$, where r_n and s_n are recursively defined as follows:

Lemma 2.2.17. *Let $F(x) = x3 + Ax + B$. Then:*

$$\begin{cases} r_2(x) = \dfrac{F'(x)^2}{4F(x)} - 2x \\ \\ s_2(x) = -\dfrac{F'(x)}{2F(x)}\left(\dfrac{F'(x)^2}{4F(x)} - 3x\right) - 1 \end{cases}$$

$$\begin{cases} r_{n+1}(x) = & F(x)\left(\dfrac{s_n(x)-1}{r_n(x)-x}\right)^2 - r_n(x) - x \\ \\ s_{n+1}(x) = & -\dfrac{s_n(x)-1}{r_n(x)-x}(r_{n+1}(x) - x) - 1 \end{cases}$$

We state the Lemma 2.2.5 without further explanation, because we have not proceeded to the development of its proof yet. The proof relies basically on computation and requires many intermediate results. For the whole proof and further explanation see [12]. Nevertheless, the main objects of the proof have been defined while formalizing Lemma 2.2.5.

3
A brief introduction to Coq

Proof assistants are programs allowing the interactive development and automatic verification of programs or mathematical statement proofs. Coq [18] belongs to a large family of proof assistants including NuPrl, PVS, HOL, Isabelle, Mizar, Lego.

Coq is the result of more than 20 years of active research, starting with the work of Thierry Coquand and Gérard Huet [6] in 1984 at INRIA.

The architecture of Coq is based on two layers: the kernel and the proof engine. The proof engine provides the tools, or *tactics*, allowing the interactive construction of proofs. Coq comes with a set of predefined tactics, and a language for the users to write their own tactics. The kernel is the core engine of Coq. It checks that a proof constructed by the proof engine rely on valid logical reasoning. As such, the kernel guarantees the correctness of Coq and its proof engine. Proofs and statements are expressed in a language called *Gallina* based on an extension of the *Calculus of Inductive Constructions*, a dependently typed, polymorphic, lambda calculus [7].

Developing safe programs in cryptography is of major interest. However, flawed pencil-and-paper proofs of cryptographic protocols are not so uncommon:

> *Do we have a problem with cryptographic proofs? Yes, we do [...] We generate more proofs than we carefully verify (and as a consequence some of our published proofs are incorrect) - Halevi, 2005*
>
> *In our opinion, many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor - Bellare and Rogaway, 2004-2006*

One may then easily understand the need of formal verification. It has been shown that formalizing provable security on top of Coq is feasible. For instance, Certicrypt [3, 20, 1] provides a formal language to describe games and develop game-based proofs that can be machine-checked afterward. In CertiCrypt, security goals and hypotheses are formulated in terms of probabistic programs which describe the interaction between a cryptographic system and a possible adversary.

However, Coq is also of great interest in the formalization of mathematics as it can assist the development of complicated mathematical proofs such as the Four Color Theorem [10].

In mathematics, to ensure that a statement is valid, one has to prove it. The proof provided needs to be complete and readable. Even in scientific literature there are cases where these requirements are not satisfied. Natural languages contain ambiguities and conceptual steps are often difficult to describe or explain. As a result, proof verification is a difficult task. In Coq, one can state mathematical theorems, develop their proofs and machine-check them. It comes with several libraries for arithmetic, real numbers, data-types, and algebraic structures in its extension SsReflect [8, 11].

3.1 Coq by example

We here exemplify the syntax of Coq and introduce its proof engine using minimal intuitionistic propositional logic. Our goal is not to give an exhaustive definition of the Coq language, but to help the reader follow the formalization described later at Chapter 4. For a complete introduction to the Coq system, see [4].

Assume that we want to prove the following tautology: $(P \Rightarrow P \Rightarrow Q) \Rightarrow P \Rightarrow Q$, where P and Q are propositions. First, we declare two propositional symbols:

```
Parameter P Q : Prop.
```

In Coq, our statement will be expressed as follows, where `->` denotes the logical implication:

```
Goal (P -> (P -> Q)) -> (P -> Q).
```

We will use the two following tactics:

- `intro` which introduces a hypothesis to the environment,
- `apply` which applies a hypothesis to the goal.

The final proof will be as follows:

```
Goal (P -> (P -> Q)) -> (P -> Q).
Proof.
  intro HPQ.
  intro HP.
  apply HPQ.
  apply HP.
  apply HP.
Qed.
```

We now give all the intermediate steps. After inputting the statement, one sees the following:

```
 P : Prop
 Q : Prop
============================
 (P -> P -> Q) -> P -> Q
```

Above the =-line lies the environment with all the declarations and definitions, and bellow the =-line is the current goal that has to be proved. Using `intro HPQ`, the system introduces the head hypothesis with name `HPQ`. The new goal is `P -> Q`:

```
 P : Prop
 Q : Prop
 HPQ : P -> P -> Q
============================
 P -> Q
```

Likewise, using `intros HP` we introduce the second hypothesis `HP` and are left to prove the following goal:

```
 P : Prop
 Q : Prop
 HPQ : P -> P -> Q
 HP : P
```

```
============================
 Q
```

Next, we use the tactic `apply` to apply the hypothesis `HPQ` to our new goal. Proofs in Coq go backwards: the users start from the conclusion and apply tactics simplifying the goal up to a point where it appears as an hypothesis in the environment. For instance, the `apply` tactic does a *modus ponens* in the reverse way: if one has to prove the goal `B` and has a proof `H` of `A -> B`, then the tactic `apply H` will transform the goal `B` to `A`.

In our case, applying `HPQ` generates two new sub-goals requiring a proof of `P`. The first goal comes from the first hypothesis of `HPQ`, whereas the second comes from the second one:

```
 P : Prop
 Q : Prop
 HPQ : P -> P -> Q
 HP : P
============================
 P

subgoal 2 is:
 P
```

Applying hypothesis `HP` solves the first goal. We then move to the second sub-goal:

```
 P : Prop
 Q : Prop
 HP : P -> P -> Q
 HPQ : P
============================
 P
```

Applying once more `HP` closes the proof. Coq displays the message `Proof completed`.

At that point, the proof is not checked yet. By inputting `Qed`, we ask the proof engine to send the constructed proof to the kernel of Coq. Only after this step we now know that we have a formal proof of our statement.

There exist several tactics which allow the user to perform case analysis, proofs by induction, first-order reasoning, automatic proofs search, as well as more powerful tactics or user defined tactics coming from external libraries. For instance, our goal could have been proved using the `tauto` tactic:

```
Goal (P -> P -> Q) -> (P -> Q).
Proof.
  tauto.
Qed.
```

3.2 Propositions and Types

Types are a central notion in Coq. Indeed, in the Coq language every valid expression comes with a type. Types determine if an expression is well formed or not: rules for building expressions are accompanied by *typing rules* that show the relation between the type of the whole expression and the type of its parts. For example, assume we declare a variable a of type `nat` (we say that *a is an inhabit of the type `nat`*), which stands for the type of natural numbers. The constant 8 being also an inhabitant of `nat`, we can deduce that `8 + a` has type `nat`. On the contrary, the expression `a + false`, where `false` is a constant of type `bool`, is not well-formed. Such an expression is forbidden by the typing rules. There exist a wide variety of types in Coq, as well as type constructors: for

instance from two types A and B, one can construct the type $A \times B$ of pairs (a, b) where a is of type A and b is of type B, or the type $A \rightarrow B$ that stands for functions that map an element of type A to a result of type B.

Coq comes with a special type named `Prop` which stands for the type of propositions. Returning to our previous example, declaring `P` and `Q` to be propositions is reduced to declare two variables of type `Prop`:

```
Parameter P Q : Prop.
```

The expression `(P -> (P -> Q))-> (P -> Q)` is a well-formed expression of type `Prop`, i.e. a proposition.

The *Curry-Howard isomorphism* describes the relation between proofs and programs: the relation between a program (or expression) and its type is the same as the relation between a proposition and its proof. For example, assume that one wants to prove that $P \Rightarrow A$ under the given list of assumptions (or *environment*) E. In minimal propositional logic, if a proof of A under the assumptions $E \cup \{P\}$ is known then on may derive a proof a $P \Rightarrow A$ under E, as stated by the following rule:

$$\frac{E, P \vdash A}{E \vdash P \Rightarrow A}$$

where $E \vdash A$ stands for A is valid under the assumptions in E. This rule is tightly related to the Coq typing rule for the formation of functions, as given below:

$$\frac{E, x : P \vdash f(x) : A}{E \vdash ((x : P) \mapsto f(x)) : P \rightarrow A}$$

The rule works as follows: if when assuming a variable x of type P one can deduce that $f(x)$ is of type A, then the function $(x : P) \mapsto f(x)$ is of type $P \rightarrow A$.

By removing the variables and expressions from the second one, we come back to the first rule. The Curry-Howard isomorphism establishes this fact: *The relation between a program (i.e. a function) and its type is the same as the relation between a proposition and its proof.* Hence, an expression $e : T$ can be either interpreted as a program e of type T or a proof e of a proposition T.

See [9] for further explanations.

Functions and Equality

In Coq, there exist two ways to denote a function. For example, for $f = (x \in \mathbb{N} \mapsto x^2)$, one can write `(fun (n:nat): nat => n * n)` or use a global level definition to give a name to the function:

```
Definition sqr (n:nat): nat := n * n.
```

The argument (along its type) of `sqr` is denoted by `(n:nat)`. The second `:nat` denotes that `sqr n` is of type `nat`. If we typecheck the `sqr` function, we obtain `sqr:nat->nat`, meaning that `sqr` is a function from $\mathbb{N}$ to $\mathbb{N}$.

A function in Coq is a function from the computer-science point of view, i.e. an algorithm or a *computable function*. From a mathematics point of view, a function $f : A \rightarrow B$ is a subset of

$A \times B$ - its graph. For example, the functions $f(x, y) = (x + y)(x - y)$ and $g(x, y) = x^2 - y^2$ are equal in mathematics since their graphs are extensionally equal. On the contrary, as algorithms, they are not equal.

Coq primitive equality is the Leibniz one: $x = y$ if for any predicate P, $P(x)$ implies $P(y)$, which does not capture the extensional equality.

As such, $\forall x. f(x) = g(x)$ *does not imply* $f = g$.

Inductive types

In Coq one can define inductive data-types. An inductive type represents the set of expressions built by a finite number of applications of its constructors.

For example, a simple inductive type is an enumerated type which represents a finite fixed set, as the one for booleans:

```
Inductive bool : Type :=
| true : bool
| false : bool.
```

This definition produces two elements of type `bool`: the two constructors `true` and `false`.

Another example of inductive types is the natural numbers. $\mathbb{N}$ is defined (as in Peano arithmetic) as:

```
Inductive nat : Type :=
| O : nat
| S : nat -> nat.
```

The type `nat` has two constructors: `O` which stands for zero and `S` for the successor function.

For every inductive definition, an inductive principle [7, 17] is generated by Coq allowing the user to develop proofs by induction, or define function by recursion.

For example, lists of elements of type `A` are defined as:

```
Inductive list (A: Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
```

where `nil` is the empty list and `cons` is the function that concatenates an element of type `A` with a list of type `list A`.

If one wishes to prove a statement for all lists, she needs to prove using the induction principle:

1. that the statement stands for the empty list, and
2. that if the statement stands for a list `L`, then for any elements `a` (of same type of the elements of the list), the statement stands for the concatenation `cons a L`.

3.3 The Small Scale Reflection extension

SsReflect (Small Scale Reflection) is a set of extensions for Coq developed to support proof methodology for algebra. We choose to base our formalization on it.

Algebraic structures in SsReflect are usually described as a carrier set along with a number of functions and a set of properties the functions have to satisfy. SsReflect uses bundled representation schemes [8] which support multiple inheritance between algebraic structures, and allows the encoding of the algebraic hierarchy.

For example, an `eqType` which stands for a type with a computable equality is defined as follows:

```
Record eqSpec (T: Type) := EqSpec {
  eq: T -> T -> bool;
  eqAxiom: forall (x y : T), (eq x y) = true <-> x=y
}.

Record eqType : Type := EqType {
  base : Type ;
  mixin : eqSpec base
}.
```

The record `eqSpec` defines the specifications of a type with equality, while the record `eqType` packs the base type (the carrier) with the `eqSpec` specifications.

The `zmodType`, which stands for an abelian group, is defined as the subtype of `eqType` as follows:

```
Record zmodSpec (V : Type) : Type := ZmodSpec {
  zero : V;
  opp : V -> V;
  add : V -> V -> V;
  _ : associative add;
  _ : commutative add;
  _ : left_id zero add;
  _ : left_inverse zero opp add
}.

Record zmodType : Type := ZmodType {
  base :> eqType;
  mixin : ZmodSpec base
}.
```

In this case the specifications of `zmodType` is given by `zmodSpec` and the record `zmodType` packs the above specifications with an `eqType`. As a subtype of `eqType`, `zmodType` inherits the functions and properties of `eqtype` - i.e. it has a computable equality.

Quotient Types

When working with algebraic structures, one often has to manipulate quotients and functions defined on them. In that case, different (not equal) terms may represent the same conceptual object.

In Coq, quotient types are not primitive types as it would make type-checking indecidable [16]. SsReflect deals with this problem by restricting the quotient types to decidable ones.

A quotient type U in SsReflect is represented by a packed structure that binds together:

1. the base carrier T,
2. a function $\pi : T \to U$ which is the embedding of T in U, and
3. a function $\text{repr} : U \to T$ s.t. for a class $\mathcal{C}$ in U, $\text{repr}(\mathcal{C})$ gives a witness in T for $\mathcal{C}$

s.t. $\pi \circ \text{repr}$ is the identity function.

Hence, to define a quotient type, one has to be able to give a function which computes the witness for every class.

The proof of the theorem of Cassel requires the definition of functions over quotients. A common method, when working with quotients, is to define a function on the non-quotiented set and prove that the function respects the quotient. A function f respects a quotient if for any x, $f(x) = f(\mathrm{repr}(\pi(x)))$. For example, a function that does not respect the quotient $\mathbb{Q}$ is $(x, y) \in \mathbb{Z} \times \mathbb{Z}^* \mapsto (x, 1)$.

As we will see in Chapter 4, all these steps are made explicit in Coq.

Polynomials in SSR

Our construction in Coq is based on the SsReflect library for uni-variate polynomials.

A polynomial over a ring R (denoted `polyR`) is represented as the list of its coefficients. The `size` of a polynomial is defined as the size of this list. Naturally, for any non-zero polynomial p:

$$\texttt{size p} = \deg(p) + 1$$

Notations

We use the following Coq notations:

`p.[x]`	the evaluation of polynomial `p` on `x`
`root x p`	`x` is a root of the polynomial `p`
`mu_x p`	the multiplicity of `x` for the polynomial `p`
`m %/ d`	the quotient of the division of the polynomial `m` by polynomial `d`
`m %% d`	the remainder of the division of the polynomial `m` by polynomial `d`
`(a,b).1`	the first projection of the pair `(a,b)`, i.e `a`
`(a,b).2`	the second projection of the pair `(a,b)`, i.e `b`

3.4 Conclusion

Proofs in Coq is a task of significant effort in some cases. More precisely, proving rigorously mathematical properties may demand explicit rewriting that makes proofs very long and barely readable. However, this kind of problems can be resolved by developing a tactic that will automatise such rewritings.

A more interesting difficulty derives from the fact that a mathematical object may have different representations. In such case, a user is left to choose the most appropriate representation to define the concept and has to prove explicitly (if needed) that another representation is isomorphic to the one defined. Also, she has to explicitly manipulate these isomorphisms every time she wants to lift a function to an isomorphic representation.

For instance, one could encode natural numbers using i) Peano's axioms, or ii) a binary representation, or iii) an inductive definition for the zero and the successor symbols (as in Peano arithmetic), along with recursive definitions of the operations. In mathematical textbooks, most of the times, the above are implicit and one is free to swap from one representation to another. A particular case of this problem will be examined in Chapter 4.

4 The Coq formalization

4.1 The elliptic curve

The base field of our curves

As explained in Section 2, we are only considering elliptic curves expressed using a Weierstrass equation. This implies our base field K to be of characteristic different from 2 and 3. Moreover, some parts of the forecoming formalized proof imply the ability to iterate over all the roots of a given polynomial. Starting with the work of Galois, it is now well known that such a function cannot exist, in a constructive setting, for an abstract field. Hence, we here have to assume that our field K is equipped with such a function.

Using the same techniques used for defining the algebraic structures of SsReflect (see Section 3), all these properties are packed in the new type `ecuFieldType`. The specifications of `ecuFieldType` are given in the record (or mixin) `ecuFieldMixins`:

```
Record ecuFieldMixins (K : fieldType) : Type := Mixin {
  roots : {poly K} -> seq K;
  _ : forall p x, (root p x) = (x \in (roots p));
  _ : 2 != 0;
  _ : 3 != 0
}.
```

Not surprisingly, these specifications are parametrized with a field (`K : fieldType`). They require a function (`roots : {poly K} -> seq K`) s.t.

```
forall p x, (root p x)= (x \in (roots p))
```

i.e., such that for any polynomial (`p : {poly K}`), for any element (`x : K`) of the field, `x` is a root of `p` if and only if `x` is a member of the list `roots p`. Hence, `roots` is the assumed function computing all the roots of a given polynomial. The two last requirements state that the field (`K : fieldType`) is of characteristic different from 2 and 3.

The type `ecuFieldType` is then defined as the subtype of `fieldType` restricted with the specifications given above.

Remark. Admitting the existence of the function `roots` gives us a clear understanding of which part of the formalized proofs are not constructive. In some cases, the specifications can be realized. Then, we recover fully constructive proofs. For instance, if the base field is a finite field, one can write the `roots` function by iterating over all the elements of the field.

Parameters of en elliptic curve

Next, we define the type of the parameters of the curve via the new record type `ecuType`:

```
Record ecuType (K : ecuFieldType) := {
  A : K;
  B : K;
  _ : 4 * A^3 + 27 * B^2 != 0
}.
```

This record packs the two parameters `A` and `B` of the curve along with the property stating that the discriminant is not null.

From now on, we fix the following Coq variables:

```
Variable K : ecuFieldType.
Variable E : ecutype K.
```

The projective plane

We define the type of points of the projective plane. We use an inductive with two constructors: i) one for the point at infinity (`EC_Inf`), and ii) one for the finite points (`EC_In`) which takes two element `(x y : K)` as input, and denotes the finite point `(|x, y|)`:

```
Inductive point : Type :=
| EC_Inf : point
| Ec_In : K -> K -> point.
```

The curve

The curve is then defined as the subtype of the projective plane whose inhabitants are exactly the points of the curve:

```
Definition oncurve (P : point K) : bool :=
  match P with
  | EC_Inf => true
  | EC_In x y => y^2 == x^3 + E.(A) * x + E.(B)
  end.

Inductive ec : Type :=
| EC : forall P : point K, oncurve P -> ec.
```

The function `oncurve` is the decision function separating the points on the curve from the other ones. It takes a point `P` and returns the boolean `true` if and only if `P` is on the curve. The notation `E.(A)` and `E.(B)` denote the parameters of the previously fixed curve `E`.

The inductive type `ec` is the type of the points on the curve. Being composed of the unique constructor `EC` of type `forall P : point K, oncurve P -> ec`, giving an inhabitant of `ec` is equivalent to giving a point of the projective plane along with the proof that this point is on the curve.

An inhabitant of the type `ec` is then *a point on the curve*, as stated bellow:

```
Lemma oncurve_ec: forall P : ec, oncurve P.
```

Elliptic curves as a group

Following Section 2, we are now defining all the needed elements for making the set of elliptic curves a group. As seen, from the geometrical intuition of the addition, the following formal definitions can be derived for the neutral element, the addition and the opposite.

- the neutral element is $\mathcal{P}_\infty$.
- the opposite of the regular point (x, y) is $(x, -y)$, whereas the opposite of the $\mathcal{P}_\infty$ is $\mathcal{P}_\infty$ itself.
- The addition is defined as follows:
  - for any point P, $P + \mathcal{P}_\infty = \mathcal{P}_\infty + P = P$
  - for any point P, $P + (-P) = \mathcal{P}_\infty$
  - otherwise, let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be two finite points. Then:
    1. if $P \neq Q$, then $P + Q = (x_S, y_S)$ with:

$$\begin{cases} x_S = & \lambda^2 - x_P - x_Q \\ y_S = & -\lambda^3 + 2\lambda x_P - \lambda x_Q - y_Q \end{cases} \quad \text{where} \quad \lambda = \frac{y_P - y_Q}{x_P - x_Q}$$

    2. if $P = Q$ with $y_P = y_Q \neq 0$, then will apply the previous formulas with:

$$\lambda = \frac{3x_P^2 + a}{2y_P}$$

    3. if $P = Q$ with $y_P = y_Q = 0$, then $P + Q = \mathcal{P}_\infty$

Note that these definitions are well defined on the whole projective plane. Hence, in our formalization, the addition and the opposite functions are first defined on the projective plane, following the formulas given above:

```
Definition zero := EC_Inf.

Definition neg (P : point K) :=
  if P is (|x, y|) then (|x, -y|) else EC_Inf

Definition add (P1 P2 : point K) :=
   match P1, P2 with
   | EC_Inf, P2 => P2
   | P1, EC_Inf => P1

   | (|x1, y1|), (|x2, y2|) =>
     if x1 == x2
     then if (y1 == y2) && (y1 != 0)
          then
            let s := (3 * x1^2 + E.(a)) / (2 * y1) in
            let xs := s^2 - 2 * x1 in
              (| xs, - s * (xs - x1) - y1 |)
          else
            EC_Inf
     else
       let s := (y2 - y1) / (x2 - x1) in
       let xs := s^2 - x1 - x2 in
         (| xs, - s * (xs - x1) - y1 |)
   end.
```

We now have to lift these definitions to internal laws over the curve. Hence, we first prove that the operations are closed, i.e. that $-P$ and $P+Q$ are on the curve whenever P and Q are.

```
Lemma zeroO : oncurve E zero
Lemma negO : forall p, oncurve E p -> oncurve E (neg p).
Lemma addO : forall p1 p2, oncurve E p1 -> oncurve E p2 -> oncurve E (add p1 p2).
```

These proofs are simple combinatorial proofs, done by case analysis on the inputs.

We can then define the previous operations as internal laws, i.e. as functions taking as input elements of `ec`, and returning elements of `ec`:

```
Definition zeroec := EC zeroO.
Definition negec := [fun p : ec E => EC (negO [oc of p])].
Definition addec := [fun p1 p2 : ec E => EC (addO [oc of p1] [oc of p2])].
```

For example, `negec` is defined from the lemma `negO` stating that *if `P` is on the curve then so is `neg P`*. It is mandatory to define `addec` and `negec`: only them are the internal operations.

Finally, we prove that the internal laws define a group over `ec`, allowing us to equip `ec` with a structure of group:

```
Lemma addNe : forall p, addec (negec p) p = zeroec.
Lemma add0e : forall p, addec zeroec p = p.
Lemma addeC : commutative addec.
Lemma addeA : associative addec.
```

The proofs of the first three lemmas follow the combinatorial pencil-and-paper proof. For the proof of associativity, following [19], we give a direct combinatorial proof. This breaks with the standard pencil-and-paper proof which uses a transport of structure from the group of *polynomial divisors*. This does not forbid us to show later that the group of elliptic curves and the group of *polynomial divisors* are isomorphic.

The ring of the coordinates of the curve

As defined in Chapter 2, the ring of the coordinates of the curve, written $K[E]$, is defined as the ring quotient of $K[x,y]$ by the ideal generated by $y^2 - \mathrm{C_X}$, where $\mathrm{C_X} = x^3 + Ax + B$. The resulted structure is an integral ring.

As previously said, it is natural to distinguish a canonical witness for every classes of $K[E]$, called *reduced form*, as formally defined bellow:

Definition 4.1.1 (Reduced form)**.** Let $p = \sum_i \alpha_i y^i \in K[x,y]$ with $\alpha_i \in K[x]$. The *reduced form* $\hat{p}$ of p is defined as:

$$\hat{p} = \textstyle\sum_i \alpha_{2i}\, \mathrm{C_X}^i + \left(\sum_i \alpha_{2i+1}\, \mathrm{C_X}^i\right) y.$$

We then lift this definition to $K[E]$. For any class C of $K[E]$, the *reduced form* of C is defined as the unique polynomial r s.t. $\forall q \in C, \hat{q} = r$.

One can check that Definition 4.1.1 is well-formed on $K[E]$. Let two polynomials p, q of $K[x,y]$ s.t. $p \sim q$. We have $\hat{p} \sim \hat{q}$. Indeed, since $\hat{p}$ (resp. $\hat{q}$) has been obtained by replacing all the occurrences of y^2 by its $\sim$-equivalent form $\mathrm{C_X}$, we have $p \sim \hat{p}$ and $q \sim \hat{q}$. Hence, there exists $a \in K[x,y]$ s.t. $\hat{p} - \hat{q} = a(y^2 - \mathrm{C_X})$ (**1**). Assume now that $\hat{p} \neq \hat{q}$, i.e. that $a \neq 0$. Then from (1), the degree of $\hat{p} - \hat{q}$ in y must be greater then 2, which is absurd.

Let $K[x, y^{\leq 1}]$ be the set of bi-variate polynomials whose degree in y does not exceed 1. Let $\phi : K[E] \to K[x, y^{\leq 1}]$ defined s.t. for any $C \in K[x, y^{\leq 1}]$, $\phi(C)$ is the unique reduced form of the polynomials of C. It is easy to check than ϕ defines a bijection. Hence, we are able to transport the structure of integral ring from $K[E]$ to $K[x, y^{\leq 1}]$, making ϕ an isomorphism.

All along the proof of the theorem of Cassel, every function definition over $K[E]$ is in fact a lift, via ϕ, of a function defined over $K[x, y^{\leq 1}]$. For example, the *conjugate* function defined in Chapter 2 could defined as follows:

Definition 4.1.2 (Conjugate)**.** Let $p \in K[E]$ whose reduced form is $p_0 + p_1 y$. The *conjugate* of p, written $\overline{p}$, is defined as $\overline{p} = p_0 - p_1 y$.

Formally, the *conjugate* is first defined for $K[x, y^{\leq 1}]$ and then lifted to $K[E]$:

Definition 4.1.3 (Conjugate)**.** let $p = p_0 + p_1 y \in K[x, y^{\leq 1}]$. The *conjugate* of p, written $\overline{p}$, is defined as $\overline{p} = p_0 - p_1 y \in K[x, y^{\leq 1}]$.

We then lift this definition to $K[E]$. For any $r \in K[E]$, the *conjugate* of r, also written $\overline{r}$, is defined as $\overline{r} = \phi^{-1}(\overline{\phi(r)})$.

This level of details is barely used in mathematics books. Isomorphic algebraic structures are considered equal and are freely interchanged. Only the first definition of *conjugate* would have been used. On the Coq side, switching from an algebraic structure to an isomorphic one has a cost. The writer must explicitly apply the isomorphism as we did in the second definition of the *conjugate*. Formally, we have several choices when formalizing $K[E]$ and its related functions in Coq.

First, one could use the generic definition of quotient sets as described in Section 3. For example, the *conjugate* could be defined as the lift to $K[E]$ of the following $\sim$-stable function:

$$p \mapsto \textstyle\sum_i \alpha_{2i} y^{2i} - \left(\sum_i \alpha_{2i+1} y^{2i}\right) y.$$

One could also exhibit the morphism ϕ between $K[E]$ and $K[x, y^{\leq 1}]$, and switch from one representation to another one using explicitly ϕ. This is what has been done in the second definition of the *conjugate*. Although these solutions seem appealing as they stick to the standard definitions, they imply a lot of technicalities.

The ring $K[E]$ is not the subject of our study, but is a technical object of the proof of the Cassel's theorem. Hence, we are free to formalize it using the best elementary representation from the point of view of the formal development. All the definitions and lemmas being defined on the reduced form of the elements of $K[E]$, using the concrete representation $K[x, y^{\leq 1}]$ clears all of the aforementioned technicalities, while allowing us to complete the proofs.

The type `ecring` representing $K[x, y^{\leq 1}]$ is defined as a pair of elements of polynomials over K.

```
Inductive ecring :=
| ECRing : {poly K} * {poly K} -> ecring E

Definition poly_val E (p : ecring E) :=
  let: ECRing p := p in p.1 *: 'X + p.2.
```

The function `poly_val` is the canonical injection from $K[x, y^{\leq 1}]$ (or `ecring`) to $K[x, y]$ (or `{poly {poly K}}`).

We use the notation `[ecp p *Y + q]` for the polynomial `ECRing p q`. Moreover, we define a coercion from `ecring` to pairs of `{poly K}`.

```
Coercion pair_val E (p : ecring E) := let: ECRing p := p in p.
```

Hence, the standard operations on pairs, like the first and second projections, can then be used on elements of type `ecring`. For instance, the first projection of the polynomial `r` defined as `[ecp p *Y + q]` computes to `p`. We also use the notation `p%:E` for the polynomials `[ecp 0 *Y p]`.

We can also defines a notion of evaluation for the inhabitants of `ecring`:

```
Definition eceval p (x y : K) := p.1.[x] * y + p.2.[x].
```

We write `p.[x, y]` for `eceval p x y`.

As we are not doing a transport of structure for the quotient ring, we are left to define all the internal operations. Hence, we first have to find the analytic definitions of these operations. For instance, the transported multiplicative law of $K[x, y^{\leq 1}]$, written $*$, is defined as $p * q = \phi(\phi^{-1}(p)\phi^{-1}(q))$. By expanding the definition of ϕ and the ring operations of $K[E]$, we obtain:

$$(p_0 + p_1 y) * (q_0 + q_1 y) = (p_0 q_0 + p_1 q_1 \mathrm{C_X}) + (p_0 q_1 + p_1 q_0) y$$

Doing this for all the other operations, and translating them to their Coq counterpart, we obtain:

```
Definition zero := [ecp 0 *Y + 0].
Definition one := [ecp 0 *Y + 1].

Definition opp (p : ecring) :=
  [ecp -p.1 *Y - p.2].

Definition add (p q : ecring) :=
  [ecp (p.1 + q.1) *Y + (p.2 + q.2)].

Definition mul p q :=
  [ecp (p.1 * q.2 + p.2 * q.1) *Y +
      (p.2 * q.2 + (p.1 * q.1) * (Xpoly E))].
```

We are left to prove that we indeed defined a integral ring:

```
Lemma addC: commutative add.
Lemma addA: associative add.
Lemma add0p: forall x : ecring, add zero x = x .
Lemma addNp: forall x : ecring, add (opp x) x = zero.

Lemma mulC: commutative mul.
Lemma mulA: associative mul.
Lemma mul1p: forall x : ecring, mul one x = x.
Lemma mul_addl:
  forall (x y z : ecring), mul (add x y) z = add (mul x z) (op y z).

Lemma idomainAxiom:
  forall (p q : ecring), p * q = 0 -> (p == 0) || (q == 0).
```

The definitions of the *conjugate*, *norm* and *degree* are then simply translated from their text-book counterpart on the reduced form:

```
Definition conjp p := [ecp -p.1 *Y + p.2].
Definition normp p := p.2 * p.2 + (p.1 * (-p.1)) * (Xpoly E).
Definition degree p := size (normp p).
```

as well as their properties:

```
Lemma normp_mul: {morph normp : p q / p * q}.
```

```
Lemma norm_eq0:
  forall p : ecring, (normp p == 0) = (p == 0).
```

4.2 Rational functions - Order

We recall the definitions of Chapter 2. The ring $K[E]$ being an integral ring, one can define the set of rational functions as the field of fractions on $K[E]$.

Definition 4.2.1 (Field of rational functions)**.** The field of rational functions, written $K(E)$, is defined as the field of fractions on $K[E]$.

The SsReflect library allows us to define, for any integral ring $\mathcal{I}$, the field of fractions of $\mathcal{I}$ as the quotient of $\mathcal{I} * \mathcal{I}^*$ with the equivalence relation:

$$\forall (n1, d1), (n2, d2) \in \mathcal{I} \times \mathcal{I}^*,\ (n1, d1) \sim (n2, d2) \Leftrightarrow n1 * d2 = n2 * d1$$

For any elements x, y of an integral domain, the notation `Frac x y` denotes the fraction x/y. We also use the notations:

```
Notation "x %:F" := Frac x 1.
Notation "x // y" := (x%:F / y%:F).
```

The purpose of this section is to formalize for every rational function f the *order* of P for f. We recall the definition of the order bellow:

Definition 4.2.2 (Evaluation)**.** Let $r \in K(E)$ and P be a point on the curve. The evaluation of r at P, written $r(P)$, is defined as follows:

- if P is finite, then:
  - if there exists a witness n/d in the class of r s.t. $d(P) \neq 0$, then $r(P) = n(P)/d(P)$,
  - otherwise, $r(P) = \infty$
- if $P = \mathcal{P}_\infty$, then let n/d be any witness in the class of r. Then:

$$r(P) = \begin{cases} 0 & \text{when } \deg(n) < \deg(d) \\ \infty & \text{when } \deg(n) > \deg(d) \\ \alpha_n/\alpha_d & \text{when } \deg(n) = \deg(d) \end{cases}$$

  where α_n (resp. α_d) is the coefficient of the higher degree term of n (resp. d).

We say that the point P is a *pole of* r (resp. *a zero of* r, *a regular point for* r) if $r(P) = \infty$ (resp $r(P) = 0$, $r(P) \neq \infty$).

Definition 4.2.3 (Uniformizer)**.** Let P be a point on the curve. A *uniformizer at* P is a rational function $u \in K(E)$ s.t. i) $u(P) = 0$, and ii) for any non-null rational function r, there exists $d \in \mathbb{Z}$, $s \in K(E)$ s.t. $r = u^d s$ and $s(P) \neq 0, \infty$.

Definition 4.2.4 (Order)**.** Let P be a point on the curve and f be a rational function of $F(E)$. The *order of* f *at* P is the unique integer $d \in \mathbb{Z}$ s.t. for any uniformizer u, r can be written $r = u^d g$ with $g(p) \neq 0, \infty$.

The above definitions are clearly not constructive. For example, for the definition of the evaluation we do not know how to find a witness matching the definition. However, the proof related to these definitions (existence of a uniformizer, uniqueness of the order) gives us all the necessary materials for defining these notions in Coq.

We first reduce the definition of evaluation to the decomposition of a rational function as described in Definition 4.2.3:

Lemma 4.2.5. *Let $r \in K(E)^*$ and P be a point on the curve. Let u be a uniformizer at P and $u^d g$ a decomposition of r respecting the conditions of Definition 4.2.3. Then:*

$$r(P) = \begin{cases} 0 & \textit{when } d > 0 \\ \infty & \textit{when } d < 0 \\ g(P) & \textit{when } d = 0 \end{cases}$$

The problem of evaluating a rational function at a given point is then reduced to find a proper decomposition of the rational function. Being constructive, the proof of Theorem 2.2.11 gives all the necessary materials for computing such a decomposition: for every rational function r and point on the curve P, the proof gives a decomposition $u^d g$ where u and d are constructively defined and g is given as a witness n/d with $n(P) \neq 0$ and $d(P) \neq 0$ if P is finite, or $\deg(n) = \deg(p)$ otherwise. (This last condition is crucial as it gives the evaluation of g at P) We first define a family of rational functions $\{u_P\}_P$ s.t. for any point of the curve P, u_P will be a uniformizer at P:

Definition 4.2.6 (Uniformizer u_P)**.** We define the family $\{u_P\}_P$ of rational functions as follows:

$$u_P = \begin{cases} x - x_P & \text{when } P = (x_P, y_P) \text{ with } y_P \neq 0 \\ y & \text{when } P = (x_P, 0) \\ \frac{x}{y} & \text{when } P = \mathcal{P}_\infty \end{cases}$$

In Coq, we define $\{u_P\}_P$ as the following function:

```
Definition unifun (P : point K) :=
  match P with
  | (| x_P, y_P |) =>
      if y == 0
      then [ecp 1 *Y + 0 ]%:F
      else [ecp 0 *Y + (X - x_P)]%:F

  | EC-Inf => [ecp 0 *Y + X] // [ecp 1 *Y + 0]
  end.
```

We now prove that for any point `P` on the curve, `unifun P` evaluates to 0. Since we are not yet able to define the evaluation of a rational function, we have to inline the definition of evaluation and gives the needed witnesses. First, we prove that the *uniformizer* at a finite point is a degenerated fraction (i.e. is a polynomial of $K[E]$):

```
Definition unifun_fin (x y : K) : ecring :=
  if y == 0
  then [ecp 1 *Y + 0 ]
  else [ecp 0 *Y + ('X - x%:P)].

Lemma unifun_finE:
  forall x y, unifun (| x, y |) = (unifun_fin x y)%:F.
```

The CoQ function `unifun_fin x y` computes that polynomial for the finite point `(|x, y|)`. Next, we prove that for every point `(|x, y|)`, `unifun_fin x y` evaluated on `(|x, y|)` is equal to zero:

```
Lemma eceval_unifun_fin:
  forall x y, (unifun_fin x y).[x, y] = 0.
```

We now move to the definition of the decomposition function. Currently, the work has only been done for polynomials over $K[E]$. The lift to the field of fractions of $K[E]$ has yet to be done, and should be accomplished in a near future.

Decomposition at regular points

Let $P = (x_P, y_P)$ s.t $y_P \neq 0$. In this case, the uniformizer u_P is equal to $(x - x_P)$. Let $f(x, y) = p(x) + yq(x) \in K[E]^*$. We want to compute $\mathrm{ord}_P(f)$. We distinguish 3 cases:

P is not a zero of $f(x, y)$

Then $f(x, y) = (x - x_P)^0 \frac{f(x,y)}{1}$.

P is a zero of $f(x, y)$ and $p(x_P) \neq 0$ or $q(x_P) \neq 0$

Let μ be the multiplicity of x_P in $\mathrm{norm}(f)$, and $r(x)$ a element of $K[x]$ s.t. $\mathrm{norm}(f) = (x - x_P)^\mu r(x)$. Then:

$$f = (x - x_P)^\mu \left(\frac{r(x)}{p(x) - yq(x)} \right)$$

with $r(x_P) \neq 0$ and $p(x_P) - y_P \cdot q(x_P) \neq 0$.

P is a zero of $f(x, y)$ and $p(x_P) = q(x_P) = 0$

Let μ_p (resp. μ_q) be the multiplicity of x_P in p (resp. in q). Let μ be the following integer:

$$\mu = \begin{cases} \mu_q & \text{if } \mu_p = 0 \\ \mu_p & \text{if } \mu_q = 0 \\ \min(\mu_p, \mu_q) & \text{otherwise} \end{cases}$$

Let $p_1(x)$ and $q_1(x)$ be the elements of $K[x]$ s.t. $p(x) = (x - x_P)^\mu p_1(x)$ and $q(x) = (x - x_P)^\mu q_1(x)$. Then:

$$f = (x - x_P)^\mu (p1(x) + yq_1(x))$$

with $p_1(x_P) \neq 0$ or $q_1(x_P) \neq 0$. We are back to the previous case.

The following CoQ function `ordreg` does the decomposition at regular, non special points. It takes a regular point `(|x, y|)` and a polynomial `(p : ecring)` and return a triple `(o, n, d)` s.t. `p = (unifun (|x, y|))^o * (n // d)`:

```
Definition ordreg (x y : K) (p : ecring) :=
  let: (d, (pp1, pp2)) := mudiv_join x p.1 p.2 in
  let: p' := [ecp pp1 *Y + pp2] in

    if p'.[x, y] == 0
    then
      let d' := \mu_x (normp p') in
      let g := (normp p') %/ ('X - x%:P)^+d' in
```

```
    ((d + d')%N, [ecp 0 *Y + g], (conjp p'))
  else
    (d, p', 1).
```

Note that, although we have a pencil-and-paper proof that the triple is a valid decomposition, we still have to formally prove it in Coq.

Decomposition at special points

Let $P = (x_P, 0)$ be a special point on the curve. The uniformizer u_P at P is equal to y. Let $f(x,y) = p(x) + yq(x)$ an element of $K[E]$. We distinguish several cases:

P is not a zero of f

Then $f(x,y) = y^0 \frac{f(x,y)}{1}$.

P is a zero of f

Since $(x_P, 0)$ in on the curve, we have $x_P^3 + Ax_P + B = 0$. Hence, there exists $z(x) \in K[x]$ s.t. $x_P^3 + Ax_P + B = (x - x_P)z(x)$. The curve being smooth, x_P is not a zero of $z(x)$. Let μ_p (resp. μ_q) be the multiplicity of x_P in $p(x)$ (resp. in $q(x)$). Let p_1 and q_1 be the elements of $K[x]$ s.t. $p(x) = (x - x_P)^{\mu_p} p_1(x)$ and $q(x) = (x - x_P)^{\mu_q} q_1(x)$ - hence, $p_1(x_P) \neq 0$ and $q_1(x_P) \neq 0$. Then:

p is the zero polynomial

Then $f(x,y) = y^{2\mu_q+1} \left(\frac{q_1(x)}{z(x)^{\mu_q}}\right)$.

q is the zero polynomial

Then $f(x,y) = y^{2\mu_p} \left(\frac{p_1(x)}{z(x)^{\mu_p}}\right)$.

p and q are not null, $2\mu_p < 2\mu_q + 1$. Then:

$$f = y^{2\mu_P} \left(\frac{p_1(x)z(x)^{\mu_q} + y(q_1(x)z(x)^{\mu_q}(X - x_P)^{(\mu_q - \mu_p)})}{z(x)^{\mu_P + \mu_q}}\right)$$

p and q are not null, $2\mu_p \geq 2\mu_q + 1$. Then:

$$f = y^{2\mu_q+1} \left(\frac{q_1(x)z(x)^{\mu_p} + y(p_1(x)z(x)^{\mu_p - 1}(X - x_P)^{\mu_p - \mu_q - 1})}{z(x)^{\mu_P + \mu_q}}\right)$$

The following Coq function `ordspec` does the decomposition at special points. It takes a special point `(|x, 0|)` and a polynomial `(p : ecring)` and return a triple `(o, n, d)` s.t. `p = (unifun (|x , 0|))^o * (n // d)`:

```
Definition ordspec (x : K) (p : ecring) :=
  let: C := Xpoly %/ ('X - x%:P) in
  let: (d1, p'1) := mudiv x p.1 in
  let: (d2, p'2) := mudiv x p.2 in

    if (p.2 != 0) && ((p.1 == 0) || (2*d2 < (2*d1).+1)%N) then
      let n1 := p'1 * (C^+d1) * (('X - x%:P)^+(d1-d2)) in
      let n2 := p'2 * (C^+d1) in

        ((2*d2)%N, [ecp n1*Y + n2], (C^+(d1+d2))%:E)

    else
      let n1 := p'2 * (C^+(d2-1)) * (('X - x%:P)^+(d2-d1-1)) in
      let n2 := p'1 * (C^+d2) in

        ((2*d1+1)%N, [ecp n1*Y + n2], (C^+(d1+d2))%:E).
```

As for `ordreg`, the correctness of the decomposition has still to be formally proven in Coq.

Decomposition at a finite point

We merge `ordreg` and `ordspec` in a single function:

```
Definition orderfin (x y : K) (f : ecring) :=
  if y == 0 then ordspec x f else ordreg x y f.
```

Correctness of the decomposition for finite points

The correctness of the decomposition for finite points is expressed by the following Coq predicate:

```
Definition uniok (p : {fraction ecring}) (x y : K) o (n d : ecring) :=
  [&& p == (unifun (| x, y |)) ^ o * (n // d)
    , n.[x, y] != 0
    & d.[x, y] != 0].
```

We first prove that for every finite point `(|x, y|)` on the curve, and for every non-zero polynomial `p` on the curve, `orderfin x y p = (d, g1, g2)` implies `g1.[x, y] != 0` and `g2.[x, y] != 0`:

```
Lemma order_num_neq0:
  forall (p : ecring) (x y : K),
    p != 0 -> oncurve (| x, y |) ->
    let: (d, g1, g2) := orderfin x y p in
      g1.[x, y] != 0.

Lemma order_den_neq0:
  forall (p:ecring) (x y : K),
    p != 0 -> oncurve (| x , y |) ->
    let: (d, g1, g2) := orderfin x y p in
      g2.[x, y] != 0.
```

Then, we prove that `orderfin` indeed decompose its input, i.e. that for any point `(|x, y|)` on the curve and for any non-zero polynomial `p` on the curve, `orderfin x y p = (d, g1, g2)` implies `p = (unifun (|x, y|))^d * (g1 // g2)`.

We can now prove the correctness of the decomposition for finite points:

```
Lemma order_frac_correct:
  forall (p : ecring) (x y : K),
    p != 0 -> oncurve (| x, y |) ->
    let: (d, g1, g2) := orderfin x y p in
      uniok p x y o n d.
```

Decomposition at the infinite point

Recall that the uniformizer $u_{\mathcal{P}_\infty}$ at the infinite point is equal to $\frac{x}{y}$. Let $f(x,y) \in F[E]$. Then, $f(x,y) = (x/y)^{-d}\left(\frac{f(x,y)\cdot x^d}{y^d}\right)$ where $d = \deg(f)$. One can check that $\deg(f(x,y)\cdot x^d) = \deg(y^d)$. We simply translate this to the following Coq function which takes a polynomial `(p : ecring)` and return a triple `(o, n, d)` s.t. `p = (unifun EC_Inf)^o * (n // d)` and `degree n = degree p`:

```
Definition orderinf (p:ecring):=
  let d := (degree p).-1 in
    (-d%:Z, 'X%:E^+d * p, [ecp 1 *Y + 0] ^+ d).
```

Again, we still need to formally prove the correctness of the decomposition function.

Correctness of the decomposition at the infinite point

We proceed with the definition of `uniok_inf`, a Coq predicate expressing the correctness of the decomposition at the infinite point. From `uniok`, the property that the point at which the decomposition occurs is not a zero of `n` or `d` has been replaced by `degree n == degree d`.

```
Definition uniok_inf (p : {fraction ecring}) o (n d : ecring) :=
  (p == (unifun EC_Inf ^ o) * (n // d)) && (degree n == degree d).
```

We first prove that for every non-zero polynomial `p` on the curve, `order_inf p = (d, g1, g2)` implies that `degree(g1)= degree(g2)` and that `p = (unifun EC_Inf)^d * (g1 // g2)`.

```
Lemma order_inf_eq_degree:
  forall (p : ecring),
    p != 0 ->
    let: (o, g1, g2) := orderinf p in
      degree g1 = degree g2.
```

Then, we prove that `orderinf` indeed decompose its input, i.e. that for any non-zero polynomial `p` on the curve, `orderinf p = (d, g1, g2)` implies `p = (unifun EC_Inf)^d * (g1 // g2)`.

```
Lemma order_inf_frac_correct:
  forall (p : ecring), p != 0 ->
    let: (o, g1, g2) := orderinf p in
    let: u := unifun EC_Inf in
      p%:F = (u ^ o) * (g1 // g2).
```

We can now prove the correctness of the decomposition at the infinite point:

```
Lemma order_inf_correct:
  forall (p : ecring), p != 0 ->
    let: (o, g1, g2) := orderinf p in
      uniok_inf p o n d.
```

Order

We can move to the definition of the *order of a polynomial on the curve*:

```
Definition order (p : ecring) (ecp : point K) :=
  match ecp with
  | EC_Inf => orderinf p
  | (| x, y |) => let: (n, g, h) := orderfin x y p in (n%:Z, g, h)
  end.
```

The order, as defined for polynomials on the curve, corresponds exactly to its mathematical definition.

As a sanity check, we prove that for our fixed family of $\{u_p\}_p$ of uniformizers the decomposition as defined above is unique:

```
Lemma order_fin_uniq:
  forall p x y, p != 0 -> oncurve (| x , y |) ->
    forall o1 o2 n1 n2 d1 d2,
        uniok p x y o1 n1 d1
      -> uniok p x y o2 n2 d2
      -> (o1 == o2) && (n1 // d1 == n2 // d2).
```

```
Lemma order_inf_uniq:
  forall p, p != 0 ->
    forall o1 o2 n1 n2 d1 d2,
         uniok_inf p o1 n1 d1
      -> uniok_inf p o2 n2 d2
      -> (o1 == o2) && (n1 // d1 == n2 // d2).
```

Remaining Work

We have arrived at a formal constructive definition of the order for polynomials on the curve. We plan to continue with the remainder of the proof as follows. We will first extend the definition of order to rational functions on the curve to pursue the proof of lemma 2.2.4. Then, we will define isogenys on elliptic curves and prove the properties that are used in the proof of the Cassel theorem. Next, we will formalize the recursive formulas stated in chapter 2 for the [n]-multiplication of points and the proof of theorem 2.2.5. Finally, we will prove by induction Theorem 2.2.3 and then the Cassel theorem.

We estimate that this remaining work will take about four months. Most of the notions and tools used in the proof are already defined and partly tested by our present code. Yet, there is a significant coding effort to be done and many intermediate results to be proved.

5 Conclusion

This thesis reports on a formal proof of the Cassel theorem for elliptic curves. To date, we have arrived at the formalisation of several basic notions and intermediate results used in the proof, as described in Chapters 2 and 4. Overall the formalisation consists of 2000 lines of code that can be found at http://strub.nu/research/ec/.

Our work confirms the ability of the SsReflect library to aid the formalisation of new mathematical theories. Our results are a step towards the development of formalised mathematical libraries for cryptography.

The main difficulty during this work was the choice of the best representation of the mathematical tools used in the proof, as well as how to successfully integrate the existing libraries in our work. These are subjects closely related with the ability of a user to understand the structure of Coq proofs and types. To understand the value of our proof, we turn to N.Koblitz [14]: *The two most important characteristics of a satisfactory proof of a theorem are correctness and clarity. That is, it must (1) be free of gaps or errors, and (2) be understandable to anyone with the necessary technical prerequisites.* Certainly, we found that formal proofs are gap and error-free. However, they fail to add to the mathematical intuitions of the proof developer. For instance, a picture like Figure 2.1, which one can find in almost every textbook of Elliptic Curve Theory, provides a geometrical interpretation of the addition law using tangents and lines that contributes to a better understanding than the Coq definitions in our development.

One future direction for our work, after accomplishing the proof of the Cassel theorem, would be to continue with the formalization of bilinear pairings. Another would be to integrate our results with CertiCrypt to build proofs of security for cryptographic schemes that use elliptic curves.

Acknowledgements

I would like to thank Assia Mahboudi, Cyril Cohen and all the members of the SsReflect team that helped me understand and use the libraries of SsReflect, professor Philippe Guillot for helping me undestand and organise the proof, Pierre-Yves Strub for his supervision, his guidance and his patience, and also Benjamin Smith, Karthikeyan Bhargavan, Jeremy Planul, Cédric Fournet and all the members of the MSR-INRIA security team who helped me by answering all my questions and correcting this thesis.

Bibliography

[1] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Formal certification of elgamal encryption. *Formal Aspects in Security and Trust*, pages 1–19, 2009.

[2] G. Barthe, F. Olemdo, Zanella Béguelin S., Grégoire B., and Heraud S. Verified Indifferentiability of Hashing into Elliptic Curves. 2011.

[3] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt. In *5th International Workshop on Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *Lectures Notes in Computer Science*, pages 1–19. Springer, 2009.

[4] Yves Bertot and Pierre Castéran. Coq'Art: examples and exercises, 2004. http://www.labri.fr/Perso/~casteran/CoqArt.

[5] L.S. Charlap and D.P. Robbins. Crd expository report 31 an elementary introduction to elliptic curves, 1988.

[6] Thierry Coquand and Gérard P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *European Conference on Computer Algebra (1)*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. Springer, 1985.

[7] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

[8] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. *Theorem Proving in Higher Order Logics*, pages 327–342, 2009.

[9] Dowek Gilles. Théories des types. 2004.

[10] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *ASCM*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[11] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

[12] Philippe Guillot. *Courbes Elliptiques, une présentation élémentaire pour la cryptographie*. Lavoisier, 2010.

[13] Antoine Joux. A one round protocol for tripartite diffie-hellman. In Wieb Bosma, editor, *ANTS*, volume 1838 of *Lecture Notes in Computer Science*, pages 385–394. Springer, 2000.

[14] N. Koblitz. Another look at automated theorem-proving ii. 2011.

[15] Adi Shamir. Identity-based cryptosystems and signature schemes. In *CRYPTO*, pages 47–53, 1984.

[16] B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *Arxiv preprint arXiv:1102.1323*, 2011.

[17] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[18] The Coq Development Team. Chapter 4: Calculus of Inductive Constructions. Technical report, 2010.

[19] Laurent Théry and Guillaume Hanrot. Primality Proving with Elliptic Curves. In Klaus Schneider and Jens Brandt, editors, *TPHOL 2007*, volume 4732 of *LNCS*, pages 319–333, Kaiserslautern, Germany, 2007. Springer-Verlag.

[20] Santiago Zanella Béguelin. *Formal Certification of Game-Based Cryptographic Proofs*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2010.