Verifying Elliptic Curves using Welder

Rodrigo Raya

June 21, 2017

Contents

1 Introduction

This work is a first attempt to put together techniques coming from the field of elliptic curves cryptography and formal verification of computer programs using the verification tools developed by the Laboratory for Automated Reasoning and Analysis at the Swiss Federal Institute of Lausanne.

Around 1985, Koblitz [Koblitz(1987)] and Miller [Miller(1985)] proposed independently the use of elliptic curves, a mathematical object of group theory, in cryptography. However, it was not until the last decade that their use became generalized in public key cryptographic protocols. Their implementation implied the use of shorter key-lengths, savings in bandwidth and better time efficiency compared to traditional algorithms. This has made elliptic curves a good support for doing cryptography on the Internet.

There are important issues arising from the implementation of elliptic curves that should be kept in mind. Literature has given special attention to the so-called attacks of side-channel. This attacks are based in the measurement of the physical parameters of the systems. For instance, one could measure the power consumption [Koblitz(1999)] or the time spent in executing the underlying algorithm [Kasper(2011)] in order to gain information on the cryptographic parameters used. In this sense, Edwards [Edwards(2007)] published a new approach to perform group addition on elliptic curves. His form, has been proved [Bernstein(2007)] to be useful to prevent side-channel attacks as well as to provide more efficient algorithms.

In principle, while working with an actual elliptic curve one should pick the parameters for the curve under the recommendation of an accredited authority. This has been traditionally the National Institute of Standards and Technology (NIST). In the late years, there has been big discussions on whether the curves proposed by NIST have been manipulated with parameters that provide back-doors for national security agencies in order to break the security of cryptographic protocols supported. In words of security expert Bruce Schneier

> The math is good, but math has no agency. Code has agency, and the code has been subverted.

In the last years users have moved onto curves whose security is not doubted. We will be basing our approach on the formal verification of elliptic curve addition on a paper that verifies Curve25519 trying to generalize it to the standard Weierstrass form. Curve25519 is an alternative to NIST curves whose security is not questioned and is the default curve for OpenSSH.

On the other hand, program verification has been treated since the beginnings of Computer Science by researchers like Turing, McCarthy [McCarthy(1993)] or Hoare [Hoare and Jifeng(1998)]. In the late fourty years, it has experimented an unequal development. Nevertheless, program verification techniques have been applied in order to verify a great variety of systems from

processors to public transport lines.

The relevance of the formal verification of cryptographic security protocols has been underlined by experts of both fields. For security specialists it is a matter of being able of stating that the protocols they rely on are actually secure and cannot be hacked in order to bypass its security. For verified programming researchers it is one of the most promising applications of their discipline. We have had the opportunity to discover several projects in this direction during the semester.

The first is Project Everest which is developed by various branches of Microsoft Research in collaboration with INRIA. This last laboratory has been a pioneer in the formal verification of elliptic curve cryptography in different systems like Coq [Bartzia(2014)], [Théry(2007)] or more recently F* in the context of Project Everest.

Our project will be developed using Inox and Welder. Inox is a consolidated solver interface for higher-order functional programs that relies on specific solvers and a definition unrolling procedure in order to proof given expressions. During this project we used Z3 as the underlying solver. Welder is a plain proof assistant that implements basic inference rules and induction mechanism so that the user can extend the functionalities provided by Inox. They are both developed in the Laboratory for Automated Reasoning and Analysis of the Swiss Federal Institute of Lausanne. The relation between the two will be explored in this report. We have also developed some working notes that could help people that begin to learn this systems. They are available in the repository of our project.

It has to be pointed out that Scala has a direct translation to JavaScript which is the preferred language for Internet protocols. This feature is currently unavailable for F* which can be seen as an advantage of our approach.

2 Welder fundamentals

We need some basic knowledge of logic to use Welder. This section establishes the relation between Welder constructs and the corresponding logical concepts.

Welder use the rules of natural deduction. These are designed to mirror human patterns of reasoning. The rules for natural deduction can be found for instance in [Paulson(1987)]. We give here a brief overview to connect them with the Welder constructs.

In natural deduction, each logical connective gets two kinds of rules. Take *op* to be a logical connective. Then, the set of introduction rules will give us conditions under which $A\ op\ B$ can be concluded. On the other hand, elimination rules will give us derived terms from an $A\ op\ B$ expression.

Table 1 gives a list of the basic rules used in Welder. Here the letter Λ is used to denote a contradiction. So essentially, for the notI rule we provide

an hypothesis and a proof of a contradiction given the hypothesis, to deduce the negation of the hypothesis.

The rules for quantification are in some way not as straight forward as the basic rules. In [Paulson(1987)] the reader may find a rigorous deduction. Table 2 gives a list of the quantification rules used in Welder. For the use of this constructs one has to look at the documentation of Welder. We have also made our own working notes that we present as an appendix.

Finally, the system also provides two powerful mechanisms for performing proofs which are structural induction and natural induction. In fact, one of the biggest challenges that Welder currently faces, is the automation of induction as well as providing general well-founded induction. Some materials that address this question can be found in [Leino(2011)] and [Reynolds and Kuncak(2015)].

Name	Logic rule	Welder construct
Conjunction introduction	$\dfrac{A \quad B}{A \wedge B}$	andI
Conjunction elimination	$\dfrac{A \wedge B}{A}$	andE
Disjunction introduction	$\dfrac{A}{A \vee B}$	orI
Disjunction introduction	$\dfrac{A}{B \vee A}$	orI
Disjunction elimination	$\dfrac{\begin{matrix}[A] & [B] \\ A \vee B \quad C & C\end{matrix}}{C}$	orE
Implication introduction	$\dfrac{\begin{matrix}[A] \\ B\end{matrix}}{A \implies B}$	implI
Implication elimination	$\dfrac{A \implies B \quad A}{B}$	implE
Negation introduction	$\dfrac{\begin{matrix}[B] \\ \Lambda\end{matrix}}{\neg B}$	notI
Negation elimination	$\dfrac{\neg(\neg B)}{B}$	notE

Table 1: Basic inference rules in Welder

It can happen while going deeper in the library that one gets confused with the names used in the different constructs. In fact, the notions of natural deduction are mixed with goal seeking ones. This is the case of the signatures for constructs dealing with natural or structural induction. It also occurs while performing a proof. In fact, if one writes forallI then he expects a proof for the formula contained in the for all expression which implies that while reading the program from top to bottom one obtains a backward proof.

Name	Logic rule	Condition	Welder construct
$\forall$ introduction	$\frac{A}{\forall x.A}$	$x \notin free(assumptions(A))$	forallI
$\forall$ elimination	$\frac{\forall x.A}{A[t\|x]}$	$A[t\|x]$ is the substitution of x by term t	forallE
$\exists$ introduction	$\frac{A[t\|x]}{\exists x.A}$	$A[t\|x]$ is the substitution of x by term t	existsI
$\exists$ elimination	$\frac{\exists x.A \wedge \delta, A \vdash B}{\delta \vdash B}$	$x \notin free(\delta) \cup free(B)$	existsE

Table 2: Quantified rules in Welder

It may be also surprising to discover that the current version of Welder does not assure soundness. This is a big issue that arises from the fact that Welder reuses the expressions available in Inox. The motivation for this is not to duplicate the code for Inox's expressions. Therefore, the only way for the designers of Welder is to perform checks on the expressions that are submitted to Inox.

The situation is represented in the following diagram:

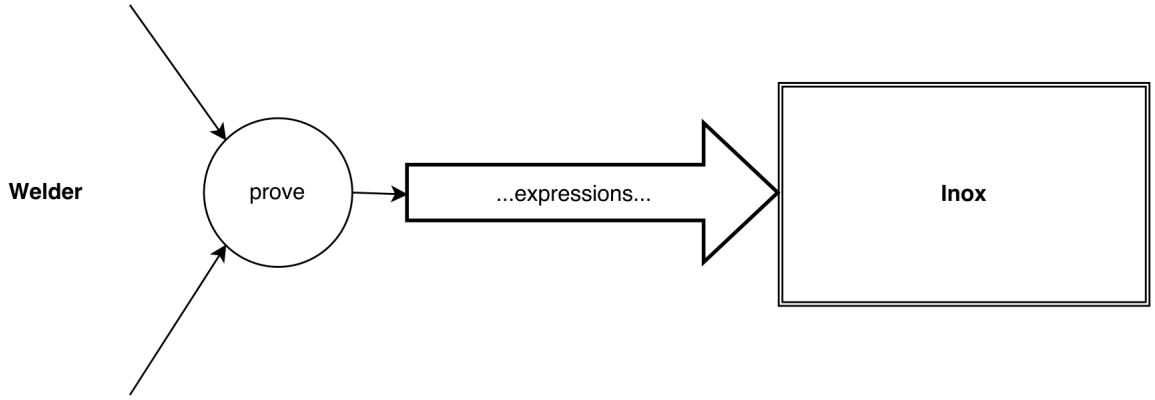

Figure 1: The Welder-Inox stack control flow

In fact, what is happening is that there are certain constructs in Welder that give up with the proof and just pass the goal to Inox to see if it can solve it. As far as we know, all these constructs fall-back to the *prove* construct. So the problem is that this *prove* construct does not perform any checks on the constitution of the expression and assumes whatever gets from Inox. Let us provide an example. One can write:

```
1 val lemma =
2  prove(
3   Assume(BooleanLiteral(false),BooleanLiteral(false))
```

```
4  )
5  val theorem =
6   prove(
7    E(BigInt(1)) === E(BigInt(2)), lemma
8   )
```

Listing 1: Example of unsoundness in Welder

The system will return $Success(Theorem(1 == 2))$. This is of course not admissible. Several other constructs have to be checked (always according to the designers of Inox and Welder) such as the *choose* statement or the $asInstOf$ method. Other constructs, like $HasADTInvariant$ are not useful because they are not currently checked by Welder and it is not clear if they would lead to unsoundness. We gave up using this last one because we could not specify that invariants hold only for specific constructors of a sort [Raya(2017)]. Instead, we used functions to ensure the invariants.

A final comment on "interactivity". We believe that in this respect, there is still a lot of work to do with Welder. Since "interactivity" has a lot to do with user experience and not with theoretical concepts one could take as a reference other existent proof assistant such as Coq or ACL2. This would speed up the process of proving with Welder.

3 Elliptic curves notions

We will define in this section the group structure whose properties we will proof in our analysis.

Definition 3.1 (Elliptic curve)**.**
Given a field K one can define an elliptic curve $E(K)$ as the set

$$\{(x, y) : x, y \in K \wedge y^2 = x^3 + Ax + B\} \cup \{\infty\}$$

where the discriminant of the equation, $4A^3 + 27B^2$, is different from zero.

Here, the symbol ∞ is called the point at infinity and gets its full significance in the context of projective geometry. The condition $4A^3 + 27B^2 \neq 0$ says that the curve is not singular. The above equation is not general enough to comprise all non-singular curves when the characteristic is 2 or 3. But we may avoid these in our analysis as recent research on the discrete logarithm problem has showed that it is insecure to perform elliptic curve cryptography over fields with such a characteristic [Gologlu and Granger(2013)].

Let us define now the addition of elliptic curve points:

Definition 3.2.
Take $P_1, P_2 \in E(K) \setminus \{\infty\}$ of the form $P_1 = (x_1, x_2)$ and $P_2 = (x_2, y_2)$ then:

- If $x_1 \neq x_2$

$$P_1 + P_2 = (x_3, y_3) = \begin{cases} m = \frac{y_2 - y_1}{x_2 - x_1} \\ x_3 = m^2 - x_1 - x_2 \\ y_3 = m(x_1 - x_3) - y_1 \end{cases}$$

- If $x_1 = x_2 \wedge y_1 \neq y_2$ then $P_1 + P_2 = \infty$
- If $P_1 = P_2 \wedge y_1 \neq 0$

$$P_1 + P_2 = (x_3, y_3) = \begin{cases} m = \frac{3x_1^2 + A}{2y_1} \\ x_3 = m^2 - 2x_1 \\ y_3 = m(x_1 - x_3) - y_1 \end{cases}$$

- If $P_1 = P_2 \wedge y_1 = 0$ then $P_1 + P_2 = \infty$
- $\forall P \in E(K),\ P + \infty = P$

We want to proof the following theorem:

Theorem 3.1 (Group properties of the addition over an elliptic curve)**.** *The addition of points on an elliptic curve $E(K)$ satisfies:*
1. Commutativity: $\forall P_1, P_2 \in E(K)$ we have $P_1 + P_2 = P_2 + P_1$
2. Existence of neutral element: $\forall P \in E(K)$ we have $P + \infty = P$
3. Existence of opposite elements: given $P \in E(K)$ there exists $P' \in E(K)$ such that $P + P' = \infty$. This point P' is denoted as $-P$.
4. Associativity: $\forall P_1, P_2, P_3 \in E(K)$ we have $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$.

Therefore, $(E(K), +)$ is an abelian group with neutral element ∞.

It is surprising how little mathematics is needed to actually have a difficult conjecture to proof. Our goal is to proof the group properties of elliptic curve addition. The situation in the literature reveals that this is not a straight forward result. For instance in [Silverman and Tate(1992)], the authors, referring to the proof of associativity say:

> Of course there a lot of cases to consider...But in a few days you will be able to check associativity using these formulas. So we need say nothing more about the proof of the associative law!

In [Washington(2008)] the author devotes a whole chapter to the proof of associativity proving some eleven lemmas before of it and deducing from it the classical theorems of Pascal and Pappus.

Therefore, the situation is far from being easy to be solved mechanically and impractical for a naive hand proof expanding all the terms. It has been noted [Russsinoff(2015b)] that expanding the resulting polynomial equation into monomials would involve some 10^{25} terms.

4 The naive approach

We begin by considering what properties of theorem 3.1 may be solved directly by Inox perhaps with some help of Welder. In order to do so, we have to decide first what will be the underlying field. We have chosen a generic field contrarily to the approach in [Russsinoff(2015b)] which uses the integers. However, Welder does not provide yet theories for modular integer arithmetic. It would be interesting though to compare the efficiencies of both approaches.

4.1 Defining the field

We would like to model the following class hierarchy for field elements:

```
1 abstract class Element()
2 abstract class nonZero() extends Element
3 final case class Zero() extends Element
4 final case class One() extends nonZero
5 final case class notOne() extends nonZero
```

Listing 2: Intended class hierarchy for field elements

This would allow us to formulate properties as the following in a convenient manner:

> The set of non zero elements of the field with one, inverse and multiplication by a non zero element forms a group.

Unfortunately, the class hierarchy in Inox is limited to a single abstract parent with a sequence of concrete constructors where no sub-typing between constructors is possible. This limitation seems to come from the theory of algebraic datatypes supported by the underlying SMT solvers [Voirol(2017a)].

Therefore, the solution we propose is the following:

```
1 abstract class Element()
2 final case class Zero() extends Element
3 final case class One() extends Element
4 final case class notZeroOne() extends Element
5 def nonZero(e: Expr): Expr =
6  e.isInstOf(One) || e.isInstOf(notZeroOne)
```

Listing 3: Intended class hierarchy for field elements

Then we need to define the operations in the underlying field. Since we do not know the implementation of these operations we would like to translate them as abstract functions. There are essentially two options to do so [Voirol(2017b)]. For instance, let us say that we want to define the add function. Then one could:

- Use a choose statement in the body of the function:

```
val body: Seq[Variable] => Expr = {
      choose("r" :: F)(_ => E(true))
}
```

Listing 4: Body of the addition operation with a choose statement

This way, during unrolling, Inox would simply replace the choose construct with a fresh variable and assume the predicate "true" on it.

- Instead of using a named function, use a function-typed variable:

```
val add: Expr =
 Variable(FreshIdentifier("add"), (F, F) =>: F)
```

Listing 5: Function typed variable for the addition operation

The issue for the choose-body option is that we would not be able to instantiate it with other concrete functions in the theorems shown about it. Therefore, we have decided the second option for our implementation.

A final trick is worth mentioning: the use of implicit class construct of Scala to define infix operators as follows:

```
implicit class Infix(private val lhs: Expr) extends AnyVal{
 def ^+(rhs: Expr): Expr = addFunction(lhs, rhs)
 ...
}
```

Listing 6: Infix operands definitions

An extra circumflex symbol is used to avoid ambiguity with the Inox DSL.

4.2 Theorems on field

Once one has decided on the representation for the field both mathematically and computationally, we need to state axioms and deduce theorems from them. We list them in tables 3 and 4.

This is just a laborious algebra problem as we need to instantiate each of the axioms with the proper terms. However, it deserves some comments. First, we get a precise statement of what axioms are required for each property. For instance, the last one is the only one that requires the characteristic of the field to be different from two. Second, it would be interesting to automate the proof process here as the instantiations of the formulas seem to be pretty obvious. Perhaps a unification procedure at the Welder level could save some of the rewriting that we had to do.

Name	Axiom
addAssociative	$\forall x, y, z \in F.(x + y) + z = x + (y + z)$
addNeutralElement	$\forall x \in F.x + 0 = 0 + x = x$
addOppositeElement	$\forall x \in F.x + opp(x) = opp(x) + x = 0$
addCommutative	$\forall x, y \in F.x + y = y + x$
multAssociative	$\forall x, y, z \in F.(xy)z = x(yz)$
multNeutralElement	$\forall x \in F.x1 = 1x = x$
multInverseElement	$\forall x \in F.xinv(x) = inv(x)x = 1$
multCommutative	$\forall x, y \in F.xy = yx$
isDistributive	$\forall x, y, z \in F.x(y + z) = xy + xz \wedge (x + y)z = xz + yz$
notCharacteristic2	$1 + 1 \neq 0$
notCharacteristic3	$1 + 1 + 1 \neq 0$

Table 3: List of the axioms used in the field theory

4.3 Elliptic curve theorems proved directly

Once we have completed the proof of the field theorems we can see what properties of the group addition can be proved directly, without resorting to special mechanisms. These are the properties of being a closed operator that is, that when we take two points on curve their addition is also in the curve, and the three first properties of theorem 3.1. Indeed it is a matter of long rewriting steps. Only a particular case of commutativity is harder than the others:

```
1 val lambda = (y2 - y1) / (x2 - x1)
2 val x = lambda^2  - x1 - x2
3 val y = lambda * (x1 - x) - y1
4 finite(x,y)
```

Listing 7: Particular addition formula

Here we want to proof the case when one has two points on the curve that are finite and whose first component x is different. To proof commutativity just means that if we exchange $x1, x2$ and $y1, y2$ the result is the same. Although we could succeed in proving that the *lambda* values and the x coordinates are equal, we would need to introduce some more steps while proving the equality of the y coordinate as there is no symmetry in the arguments it uses (it uses only the coordinates of the first point). We realize that this problem requires a strategy to liberate the programmer of writing all the equation steps in the proof.

The case of associativity is even more intricate and most of the literature overcomes this problem developing specific tactics for it. So we leave the associativity for more specialized procedures that we describe in future sections.

Name	Theorem
uniqueAddNeutral	$\forall y \in F.(\forall x \in F.x + y = y + x = y) \implies y = 0$
uniqueMultInverse	$\forall x, y, z \in F.(x \neq 0 \wedge (xy = yx = 1) \wedge (xz = zx = 1)) \implies y = z$
uniqueAddOpposite	$\forall x, y, z \in F.((x + y = y + x = 0) \wedge (x + z = z + x = 0)) \implies y = z$
oppositeOfZero	$opp(0) = 0$
oppositeInvolution	$\forall x \in F.opp(opp(x)) = x$
simplification	$\forall x \in F.x = 2x \implies x = 0$
zeroDivisor	$\forall x \in F.x0 = 0 = 0x$
oppositeOfAdd	$\forall x, y \in F. - (x + y) = (-y) + (-x)$
oppositeOfMult	$\forall x, y, z \in F. - (xy) = (-x)y = x(-y)$
integralDomain	$\forall x, y \in F.x, y \neq 0 \implies xy \neq 0$
cancellation	$\forall x, y, z \in F.x + z = y + z \implies x = y$
inverseOfMult	$\forall x, y \in F.x, y \neq 0 \implies (xy)^{-1} = y^{-1}x^{-1}$
zeroNotMinusOne	$opp(1) \neq 0$
inverseOfMinusOne	$inv(-1) = -1$
oppositeofInverse	$\forall x \in F. - (x)^{-1} = (-x)^{-1}$
exp1	$\forall h \in F, n \in \mathbb{N}.h^{n+1} = hh^n$
exp2	$\forall h \in F, i, j \in \mathbb{N}.h^{i+j} = h^i h^j$
doubleXZero	$\forall x \in F.2x = 0 \implies x = 0$

Table 4: List of the theorems deduced in the field theory

5 The tactic

5.1 A word on the existent solutions

Before giving the solution chosen in our work, we would like to comment on what other researchers have tried in order to solve the intractable problem of proving associativity in a naive way.

The first attempt to proof our theorem in a computer proof system was made by [Théry(2007)] based on the work by [Friedl(1998)]. In fact, Théry's work is a formalization in a computer system of a seven page mathematics research article removing the dependence of CoCoA computer algebra system. It is here that the idea of the tactic that we have implemented first appears.

The work that we have followed closely [Russsinoff(2015b)] criticises their approach as not been "computationally surveyable" meaning that their approach provides evidence of correctness but does little to help understanding the underlying mathematics. According to Russinoff, the work by Friedl attributes unproved results to CoCoA while Théry's work claims to depend on an undisclosed tactic that has reportedly been implemented in Coq. We did experiment this while reading Théry's paper and it was not easy to understand and search the actual proof tactic. In this sense, the work of

Russinoff is praiseworthy as he not only describes precisely the tactic that is using (based upon the idea of Théry) but also achieves modest time improvements in the time necessary for the verification by not relying on an heuristic rationalizer procedure but rationalizing himself the relevant equations.

A more recent approach is the one given in [Bartzia(2014)] which proves the group property by establishing an isomorphism between the Picard group of divisors and the elliptic curve. They claim that their approach involves less computation by using the definition of auxiliary algebraic structures.

5.2 Theoretical motivation: Horner polynomial form

In numerical analysis, one often deals with Horner polynomials. They first appear as a method for evaluating efficiently polynomials of one variable.

Suppose we are given a one-variable polynomial:

$$p(x) = a_0 + a_1 \cdot x + \cdots + a_n \cdot x^n$$

Then one can present the following equivalent form for it:

$$p(x) = a_0 + x \cdot (a_1 + x \cdot (\cdots + x(a_{n-1} + a_n \cdot x)...))$$

The first use of this form is to efficiently evaluate the polynomial in x_0 by computing the following terms:

$$b_n = a_n$$

$$b_{n-1} = a_{n-1} + b_n \cdot x_0$$

$$\cdots$$

$$b_0 = a_0 + b_1 \cdot x_0$$

While the direct approach to evaluate the polynomial takes n additions and $\frac{n(n+1)}{2}$ multiplications ($2n - 1$ if reusing partial multiplications), the Horner method requires at most n additions and n multiplications. This results in a linear algorithm, while the naive direct approach leads to a quadratic one (or linear with double complexity constant).

In this section, we give the general idea of the tactic that we have develop for Welder. Our tactic is based upon the ideas presented in [R. Milner(1984)], [Gregoire and Mahboubi(2005)], [Théry(2007)] and [Russsinoff(2015a)]. We follow closely, the syntax given in [Russsinoff(2015a)] but we adapt ACL2's proof to Welder and generalize it to the case of generic commutative rings and not just integers. We will also try to adapt its approach from Curve25519 to elliptic curves in the form of the Weierstrass equation.

5.3 Gregoire-Mahboubi tactic for solving equalities in commutative rings

In [Gregoire and Mahboubi(2005)], the authors give a general approach on the tactic to use in order to proof equalities in a commutative ring A. In other words, we want to proof $t_1 = t_2$ where t_1, t_2 are terms made up of ring constants, ring operations and other sub-terms.

5.3.1 Metaification

Depending of the application, we will need to deal with different kinds of terms. For instance, we could have to deal with the equality:

$$3 * sin(x) * x = x * (sin(x) + 2 * sin(x)) + 0 * y$$

However, this equality can be easily abstracted into the following polynomial expression $3 * X_1 * X_2 = X_2 * (X_1 + 2 * X_1) + 0 * X_3$. It is useful to keep the lists of assignments from expressions to variable of the form $[sin(x); x; y]$ as they could be used in later steps.

More formally, every ring constant is interpreted as a constant polynomial expression, every ring operation is interpreted as an operation over polynomial expressions and we hide every sub-term that is neither a ring constant, nor the application of a ring operation to other sub-terms. The process is known as metaification and we will only deal with it during the applications of the tactic. For the time being we can assume that we have two meta-terms $m_1 = \tau(t_1)$ and $m_2 = \tau(t_2)$ which are nothing else but multivariate polynomials expressions.

5.3.2 Normalization of the expressions

Once we have our meta terms m_1 and m_2 we want to normalize them following the Sparse Horner Normal Form which is presented in detail in [Russsinoff(2015a)]. We will check the normal forms for e_1 and e_2 are equal. That is, $norm(m_1) = norm(m_2)$.

The properties of the function *norm* will ensure that normalizing a polynomial m, does not change it from the evaluation point of view. More concretely, we will have

$$\forall m.evalp(l, m) = evalh(l, norm(m))$$

where *evalp* and *evalh* are the evaluation functions on the space of polynomial expressions and the space of normalized polynomials respectively. Note that, we could restrict the list l to be the one obtained during the metaification process.

To make our procedure useful, we have to proof one further property. Basically, we want to proof that the equality of normalized terms implies

the equality of the original terms. More precisely, we will have

$$\forall l, m_1, m_2. norm(m_1) = norm(m_2) \implies evalp(l, m_1) = evalp(l, m_2)$$

The converse of this theorem states some sort of completeness of the tactic. Given two equal polynomials, the converse says that they produce the same Horner form. Note however that it is not necessary to formalize this proof as it will not be used in practice.

5.4 From Horner form to Sparse Horner form

Recall the procedure we gave for the efficient evaluation of polynomials, i.e. the Horner form. Essentially, in each step, we have to perform two operations. We either have to add a constant or have to multiply by polynomial variable X. Therefore, we could construct the hierarchy of univariate polynomials in the following manner:

```
1 mkSort(hornerpol)(Seq(const,nonconst))
2 mkConstructor(const){ Seq(c: F) }
3 mkConstructor(nonconst){
4  Seq(c: F, p: hornerpol)
5 }
6
7 val P = T(hornerpol)(F)
8 val Pc = T(const)(F)
9 val PX = T(nonconst)(F)
```

Listing 8: Horner polynomials hierarchy

Now, consider that this representation can be improved in terms of space. For instance the representation of polynomial $X^4 + 1$ is now

$$PX(PX(PX(PX(Pc\ 1)0)0)0)1)$$

We can do better if we add a power index as follows:

```
1 mkConstructor(nonconst){
2  Seq(p: F,i: positive,c: F)
3 }
```

Listing 9: Horner polynomials with power index

Here, you can assume that *positive* is an inductive type that models $\mathbb{N}^*$. In fact, this is not how the polynomials will be defined in our proof because Welder does not support yet ADT invariants for constructors and their use can lead to potential unsoundness. Instead, functions are given to ensure that the representation we give has the desired properties. In any case, this new representation allows us to present the polynomial $X^4 + 1$ as:

$$PX(Pc\ 1)4\ 1$$

To introduce the multivariate case, the non-constant polynomial ADT is split in two cases called POW and POP. POW corresponds to the definition presented before, while POP allows to construct an onion-like polynomial by skipping not desired variables. Following this formalization of Horner polynomials we will encounter a normalization procedure that will ensure that there is a unique representation for arbitrary polynomials.

6 Sparse horner polynomials

We now generalize the presentation of sparse horner polynomials to arbitrary underlying rings. In fact, in our application, the ring is actually a field and we will rationalize the equations involved to apply the tactic.

6.1 Representation of sparse horner polynomials

Let us fix the underlying ring R, a set of variables $V = (v_i)_{i \in \{0,\cdots,k-1\}}$ and a function $A : V \to R$ such that $A(v_i) = n_i$. We have the following definitions:

Definition 6.1 (Polynomial term)**.**
A polynomial term over V is one of the following:

- an element of R
- an element of V
- $opp(x)$ where x is a polynomial over V
- $x + y$, $x * y$ where x, y are polynomials over V
- x^n where x is a polynomial over V and $n \in \mathbb{N}$

The set of polynomial terms over V will be denoted as $\tau(V)$.

Definition 6.2 (Evaluation rules of a polynomial term)**.**
The evaluation function $evalp$ of a polynomial term $q \in \tau(V)$ is defined as:

$$evalp(q, A) = \begin{cases} q & \text{if } q \in R \\ n & \text{if } q \in V \wedge A(q) = n \\ opp(evalp(r, A)) & \text{if } q = opp(r) \\ evalp(r, A) + evalp(s, A) & \text{if } q = r + s \\ evalp(r, A) \cdot evalp(s, A) & \text{if } q = r * s \\ evalp(r, A)^n & \text{if } q = x^n \end{cases}$$

We introduce two constructors POP and POW and use them to give the following inductive definitions:

Definition 6.3 (Sparse Horner form)**.**
A sparse Horner form (SHF) is one of the following:

- An element of R.
- Given $i \in \mathbb{N}$ and p a SHF then $POP(i, p)$ is a SHF.
- Given $i \in \mathbb{N}$ and p, q SHF then $POW(i, p, q)$ is a SHF.

The intuition is the following: in each step we take a list of variables $V = (v_0 \ \cdots \ v_{k-1})$, when we read a $POP(i, p)$ expression, we drop some i variables of the list. Following this, one works with a sub-list of V, $V^{(i)}$ in which one writes the polynomial p of the expression $POP(i, p)$. Therefore, when evaluating the polynomial p one will not find variables $v_0 \cdots v_{i-1}$. On the other hand, when we read $POW(i, p, q)$ we will evaluate it to $x^i * p + q$ where x is the first variable of the current V-list, p is not divisible by x and q does not contain x (so it is evaluated in $V^{(1)}$).

Definition 6.4 (Sparse Horner normal form)**.**
A sparse Horner normal form (SHNF) is one of the following SHF:

- An element of R.
- Given $i \in \mathbb{N}^*$ and p a SHNF of the form $POW(j, q, r)$ then $POP(i, p)$ is a SHNF.
- Given $i \in \mathbb{N}^*$ and p, q SHNF where p is not zero and not of the form $POW(j, r, 0)$ then $POW(i, p, q)$ is a SHNF.

We will denote by $\mathcal{H}$ the set of all SHNFs.

The conditions assumed for the normal form can be translated in everyday language.

- The condition $i > 0$ ensures that our notation is compact in the sense that we don't introduce superfluous POP or POW.
- In $POP(i, p)$, we say that p is of the form $POW(j, q, r)$ so that the injection index i, is the biggest possible.
- In $POW(i, p, q)$, we say that p is not of the form $POW(j, r, 0)$ so that the power index is the biggest possible. If we had $p = 0$ we would make the representation not unique, since it would be equivalent to $POP(1, q)$.

Now, we will sketch how we construct a *norm* function:

Definition 6.5 (Preliminary norm function)**.**
Given a polynomial term q, the norm function is defined as follows:

$$norm(q,V) = \begin{cases} q & \text{if } q \in R \\ POW(i, norm(g,V), norm(h,V^{(1)})) & \text{if } v_0 \in q \\ POP(i, norm(q,V^{(i)})) & \text{if } v_0 \notin q \end{cases}$$

In the second expression, g and h are taken such that $q = v_0^i \cdot g + h$, g is not divisible by v_0 and v_0 does not occur in h. In the third expression, v_i is the first variable in V that occurs in q.

Let us also define how to evaluate this SHF. From now on, fix N to be a finite subset of R:

Definition 6.6 (Evaluation of the preliminary norm function)**.**
Given h a SHF. We define the evaluation of h given N, $evalh(h,N)$, as follows:

$$\begin{cases} h & \text{if } h \in R \\ evalh(p, N^{(i)}) & \text{if } h = POP(i,p) \\ n^i \cdot evalh(p,N) + evalh(q, N^{(1)}) & \text{if } h = POW(i,p,q) \wedge head(N) = n \\ evalh(q,N) & \text{if } h = POW(i,p,q) \wedge N = \emptyset \end{cases}$$

It is clear that this scheme will ensure the property

$$evalh(norm(q,V),N) = evalp(q,A)$$

It will also conform to the definition of SHNF. However it is impractical due to the difficulty of finding polynomials g, h such that $q = v_0^i \cdot g + h$ in the second norm case.

We will instead normalize the constructs POW and POP without having to use polynomials g, h still ensuring the maximal conditions:

Definition 6.7 (pop function)**.**
Take $i \in \mathbb{N}$ and $p \in \mathcal{H}$ then:

$$pop(i,p) = \begin{cases} p & \text{if } i = 0 \vee p \in R \\ POP(i+j,p) & \text{if } p = POP(j,q) \\ POP(i,p) & otherwise \end{cases}$$

Definition 6.8 (pow function)**.**
Take $i \in \mathbb{N}^*$ and $p, q \in \mathcal{H}$ then:

$$pow(i,p,q) = \begin{cases} pop(1,q) & \text{if } p = 0 \vee p \in R \\ POW(i+j,r,q) & \text{if } p = POW(j,r,0) \\ POW(i,p,q) & otherwise \end{cases}$$

Observe how we have substituted the maximal conditions by the second case in each of the definitions. The first conditions seek to preserve a compact representation.

Lemma 6.1 (Normality of pop and pow)**.**
The following follows directly from the previous definitions:
1. $i \in \mathbb{N}, p \in \mathcal{H}$ *then* $pop(i,p) \in \mathcal{H}$ *and*

$$evalh(pop(i,p),N) = evalh(POP(i,p),N)$$

2. $i \in \mathbb{N}^*, p,q \in \mathcal{H}$ *then* $pow(i,p,q) \in \mathcal{H}$ *and*

$$evalh(pow(i,p,q),N) = evalh(POW(i,p,q),N)$$

6.2 Operations on Sparse Horner polynomials

Once we have defined the basic elements that describe how Sparse Horner normal polynomials are built, we can describe how they are combined using conventional polynomials operations:

Definition 6.9 (Sum of normal forms)**.**
Let $x, y \in \mathcal{H}$ then its sum $x \oplus y$ is defined as follows:
(1) $x \in R$ then

$$x \oplus y = \begin{cases} x + y & \text{if } y \in R \\ POP(i, x \oplus p) & \text{if } y = POP(i,p) \\ POW(i,p,x \oplus q) & \text{if } y = POW(i,p,q) \end{cases}$$

(2) $y \in R$ then $x \oplus y = y \oplus x$
(3) $x = POP(i,p)$ and $y = POP(j,q)$ then

$$x \oplus y = \begin{cases} pop(i, p \oplus q) & \text{if } i = j \\ pop(j, POP(i-j,p) \oplus q) & \text{if } i > j \\ pop(i, POP(j-i,q) \oplus p) & \text{if } i < j \end{cases}$$

(4) $x = POP(i,p)$ and $y = POW(j,q,r)$ then

$$x \oplus y = \begin{cases} POW(j,q,r \oplus p) & \text{if } i = 1 \\ POW(j,q,r \oplus POP(i-1,p)) & \text{if } i > 1 \end{cases}$$

(5) $y = POP(i,p)$ and $x = POW(j,q,r)$ then $x \oplus y = y \oplus x$
$x = POW(i,p,q)$ and $y = POW(j,r,s)$ then

$$x \oplus y = \begin{cases} pow(i, p \oplus r, q \oplus s) & \text{if } i = j \\ pow(j, POW(i-j,p,0) \oplus r, q \oplus s) & \text{if } i > j \\ pop(i, POW(j-i,r,0) \oplus p, s \oplus q) & \text{if } i < j \end{cases}$$

Definition 6.10 (Opposite of normal forms)**.**
Let $x \in \mathcal{H}$ then its opposite $\ominus x$ is defined as follows:

$$\ominus x = \begin{cases} opp(x) & \text{if } x \in R \\ POP(i, \ominus p) & \text{if } x = POP(i,p) \\ POW(i, \ominus p, \ominus q) & \text{if } x = POW(i,p,q) \end{cases}$$

Definition 6.11 (Multiplication of normal forms)**.**
Let $x, y \in \mathcal{H}$ then its multiplication $x \otimes y$ is defined as follows:
(1) $x \in R$ then

$$x \otimes y = \begin{cases} xy & \text{if } y \in R \\ pop(i, x \otimes p) & \text{if } y = POP(i,p) \\ pow(i, x \otimes p, x \otimes q) & \text{if } y = POW(i,p,q) \end{cases}$$

(2) $y \in R$ then $x \otimes y = y \otimes x$
(3) $x = POP(i,p)$ and $y = POP(j,q)$ then:

$$x \otimes y = \begin{cases} pop(i, p \otimes q) & \text{if } i = j \\ pop(j, POP(i-j, p) \otimes q) & \text{if } i > j \\ pop(i, POP(j-i, q) \otimes p) & \text{if } i < j \end{cases}$$

(4) $x = POP(i,p)$ and $y = POW(j,q,r)$ then:

$$x \otimes y = \begin{cases} pow(j, x \otimes q, p \otimes r) & \text{if } i = 1 \\ pow(j, x \otimes q, POP(i-1, p) \otimes r) & \text{if } i > 1 \end{cases}$$

(5) $y = POP(i,p)$ and $x = POW(j,q,r)$ then $x \otimes y = y \otimes x$
(6) $x = POW(i,p,q)$ and $y = POW(j,r,s)$ then:

$$x \otimes y = (pow(i+j, p \otimes r, q \otimes s) \oplus pow(i, p \otimes pop(1,s), 0)) \oplus pow(j, r \otimes pop(1,q), 0)$$

Definition 6.12 (Exponentiation of normal forms)**.**
Let $x \in \mathcal{H}, k \in \mathbb{N}$ then exponential x^k is defined as follows:

$$x^k = \begin{cases} 1 & \text{if } k = 0 \\ x \otimes x^{k-1} & \text{if } k > 0 \end{cases}$$

We will need to proof the following properties:

Lemma 6.2 ($\mathcal{H}$ is closed for the operations and *evalh* preserves normality)**.**
Let $x, y \in \mathcal{H}$. Then:
1. $x \oplus y \in \mathcal{H}$
2. $evalh(x \oplus y, N) = evalh(x, N) + evalh(y, N)$.
3. $\ominus x \in \mathcal{H}$

4. $evalh(\ominus x, N) = opp(evalh(x, N))$
5. $x \otimes y \in \mathcal{H}$
6. $evalh(x \otimes y, N) = evalh(x) \cdot evalh(y, N)$
7. $x^k \in \mathcal{H}$
8. $evalh(x^k, N) = evalh(x, N)^k$.

For properties, 1-2 and 5-6 we have had to perform induction on the measure given by the sum of a count function for sparse Horner form polynomials. We are not sure that a further double induction is needed to proof these lemmas. However, we have eventually performed a triple induction to structure our reasoning. It is very likely than this can be removed in the future.

We can finally formulate the proper normalization procedure:

Definition 6.13 (Normalization procedure)**.**
Let $x \in \tau(V)$, compute the normal form of x as follows:

$$norm(x, V) = \begin{cases} x & \text{if } x \in R \\ pop(i, POW(1, 1, 0)) & \text{if } x = v_i \\ \ominus norm(y, V) & \text{if } x = opp(y) \\ norm(y, V) \oplus norm(z, V) & \text{if } x = y + z \\ norm(y, V) \otimes norm(z, V) & \text{if } x = y * z \\ norm(y, V)^k & \text{if } x = y^k \end{cases}$$

Finally, we establish the lemmas that will be useful while using our tactic:

Theorem 6.1 (Correctness of normalization)**.**
Let $f \in \tau(V)$ be a polynomial expression and reset A codomain to N. Then, we have $norm(f, V) \in \mathcal{H}$ and $evalh(norm(f, V), N) = evalp(f, A)$.

With this theorem, if we assume that $norm(f_1, V) = norm(f_2, V)$ then we have that $evalp(f_1, A) = evalp(f_2, A)$ since in that case, we would have:

$evalp(f_1, A) = evalh(norm(f_1, V), N) = evalh(norm(f_2, V), N) = evalp(f_2, A)$

But it turns out that we can do better than that:

Theorem 6.2 (Correctness of normalization)**.**
Let $f, g \in \tau(V)$ be polynomial expressions. Then, we have

$$norm(f, V) = norm(g, V) \iff evalp(f, A) = evalp(g, A)\ \forall A$$

Where A ranges over the assignments from V to N.

At this point, we have proven a pretty good condition. The equality of normal forms is equivalent to the polynomials being identical. Therefore, we count with a normalizing procedure that will help us in further sections to reason about polynomial equations.

7 Towards the proof of associativity

The idea is to use the tactic we have implemented an apply it in the special case where we have three points $P_1 = (X_1, Y_1)$, $P_2 = (X_2, Y_2)$ and $P_3 = (X_3, Y_3)$. The coordinates are taken to be variables and are given by a list $V = (Y_0, Y_1, Y_2, X_0, X_1, X_2)$. Its values are given by the corresponding list $N = (y_0, y_1, y_2, x_0, x_1, x_2)$. The notation also involves as usual an assignment A from V to N that assigns each variable to its value.

7.1 The rewriting procedure

Once one has defined a normal form for polynomials and proved its validity as a tactic there is still one degree of freedom in the equations that we may be dealing with. For the case of study, these equations are of the form:

$$y_j^2 = x_j^3 + Ax + B$$

This means that we have to further transform our equalities so that there is no difference between the left hand side of the equation and the right hand side of it. One then formulates a procedure:

$$rewrite(h, j)$$

that rewrites polynomial $h \in \mathcal{H}$ for equation numbered by j. Doing so for every equation involved (here we have three points) gives you the following final reduction procedure for $\sigma \in \tau(V)$:

$$reduce(\sigma) = rewrite(rewrite(rewrite(norm(\sigma, V), 0), 1), 2)$$

Therefore, the core part of the rewriting procedure will leave us with no expression that depends on a term y_j^2. Instead we will have a sum of a term that does not contain variable Y_j and a term that is linear on variable Y_j. Before giving the explicit procedure let just fix the polynomial:

$$\Theta = POP(3, POW(1, POW(2, 1, A), B))$$

This polynomial is the representation of $x^3 + Ax^2 + B$ and it will verify the property:

$$evalh(\Theta, N^{(j)}) = x_j^3 + Ax_j + B$$

Finally we can formulate the splitting procedure that we need:

Definition 7.1 (Splitting procedure)**.**
Let $h \in \mathcal{H}$, $j \in \{0, 1, 2\}$ and $k \in \mathbb{N}$:

- If $h \in \mathbb{Z} \vee j < k$ then $(h, 0)$

- If $j \geq k, h = POP(i,p) \wedge (p_0,p_1) = split(p,j,k+i)$ then

$$(pop(i,p_0), pop(i,p_1))$$

- If $j \geq k, h = POW(i,p,q), (p_0,p_1) = split(p,j,k)$ and $(q_0,q_1) = split(q,j,k+1)$ then
  - If $j > k$ then $(pow(i,p_0,q_0), pow(i,p_1,q_1))$.
  - If $j = k \wedge i$ is even then

$$((\Theta^{\frac{i}{2}} \otimes p_0) \oplus pop(1,q_0), (\Theta^{\frac{i}{2}} \otimes p_1) \oplus pop(1,q_1))$$

  - If $j = k \wedge i$ is odd then

$$((\Theta^{\frac{i+1}{2}} \otimes p_1) \oplus pop(1,q_0), (\Theta^{\frac{i-1}{2}} \otimes p_0) \oplus pop(1,q_1))$$

Here k is just a variable that tells us if the variable j for which we want to do the rewriting is in scope or not. This procedure will give us the further property:

$$evalh(h, N^{(k)}) = evalh(h_0, N^{(k)}) + y_j \cdot evalh(h_1, N^{(k)})$$

Here $(h_0,h_1) = split(h,j,k)$.

Finally, we give the actual definition of the *rewrite* and *reduce* procedures explained before:

Definition 7.2 (Rewrite and reduce procedures)**.**
Let $h \in \mathcal{H}$, $j \in \{0,1,2\}$, $(h_0,h_1) = split(h,j,0)$ and $\sigma \in \tau(V)$ then:

$$rewrite(h,j) = h_0 \oplus (h_1 \otimes norm(Y_j, V))$$

$$reduce(\sigma) = rewrite(rewrite(rewrite(norm(\sigma,V),0),1),2)$$

These procedures will further have properties derived from the theorem that we proved on normal forms which are:

$$evalp(rewrite(h,j), A) = evalp(h, A)$$

and

$$reduce(\sigma) = reduce(\tau) \implies evalp(\sigma) = evalp(\tau)$$

To sum up, we list the necessary properties for this section:

Name	Theorem
Θ-evaluation	$evalh(\Theta, N^{(j)}) = x_j^3 + Ax_j + B$
Split evaluation	$evalh(h, N^{(k)}) = evalh(h_0, N^{(k)}) + y_j \cdot evalh(h_1, N^{(k)})$
Rewrite evaluation	$evalp(rewrite(h, j), A) = evalp(h, A)$
Reduce evaluation	$reduce(\sigma) = reduce(\tau) \implies evalp(\sigma) = evalp(\tau)$

Table 5: Properties for the reduction procedure

7.2 Rationalizer procedure

In the last weeks of this project we have been working in generalizing the proof offered by Russinoff's for Curve25519 to arbitrary elliptic curves in Weierstrass equations (even if Curve25519 is in Montgomery form). The approach followed by Théry is based on a rationalizer which is not fully satisfactory and employs heuristics in its functioning. Russinoff's approach instead rationalizes the equations for the curve.

The procedure to do so is very similar to the one that one does to pass from the projective space to the affine plane. Here the mapping is slightly different but this idea will give us the intuition to what is going on. Therefore we start we the set $\mathbb{F}^3$ of triples made with elements of F.

First, we can restrict ourselves to non-infinite points of our curve since when one of the points is infinite, the associative property is immediate and in the worst case can lead to proof commutativity. Therefore, in $\mathbb{F}^3$ we can remove all the triples whose last component is zero. We will denote this set by $\mathbb{F}^{3*}$.

Let us consider the following mapping:

Definition 7.3 (Projection mapping)**.**
Projection is a mapping $p : F^{3*} \to F^2$ such that:

$$p((u, v, w)) = \left(\frac{u}{w^2}, \frac{v}{w^3}\right)$$

In this case we say that (u, v, w) is a representative of $p((u, v, w))$ where the canonical representative for point (x, y) is $(x, y, 1)$.

Using this mapping we will rationalize the equations given by the addition. Again the projective space intuition is useful. We can fix the z coordinate at will. From there one only has to operate to get the desired result.

Definition 7.4 (Addition on $\mathbb{F}^{3*}$)**.**
Given $P, Q \in \mathbb{F}^{3*}$ their addition $P \oplus Q = (m', n', z')$ is defined as:

- If $P = Q = (m, n, z)$ then

$$z' = 2nz$$
$$w' = 3m^2 + Az^4$$

$$m' = w'^2 - 8n^2 m$$
$$n' = 4n^2 w'(m - w'^2 + 8n^2 m) - 8n^4$$

This case stands for the third and fourth points of the elliptic curve addition definition.

- If $P = (x, y, 1)$ and $Q = (m, n, z) \neq P$ then

$$z' = z(m - xz^2)$$
$$m' = (n - yz^3)^2 - (m - xz^2)^2(m + xz^2)$$
$$n' = (n - yz^3)(xz'^2 - m') - yz'^3$$

This case stands for the first point of the elliptic curve addition definition.

In particular, note that this addition is not an internal operation since we can get a zero z coordinate in cases two and four of the elliptic curve addition operation. It is also a partial function in the sense that the second case defines the equations only for a canonical representative.

We should stress also the second point in our original definition of addition does not have a correspondence here. The analysis in this case is a little bit tricky. It goes as follows:

We want to proof associativity. If we are in case two, an easy property of fields tells you that since P_1, P_2 are on the curve and have equal first component then $y_1^2 = y_2^2$ and therefore $y_1 = y_2$ or $y_1 = -y_2$. In the first case, we have that the equation corresponds to the first case of the above definition. For the other case, we can use equation reasoning and the fact that commutativity is already proven.

Reversing the process that we have done for constructing this questions gives automatically the following lemma:

Lemma 7.1 (Relation with the curve addition)**.**
Let $\mathcal{P}, \mathcal{Q} \in \mathbb{F}^{3*}$ and set $P = p(\mathcal{P}), Q = p(\mathcal{Q})$. Assume $P, Q \in E(K) \setminus \{\infty\}$ and that $\oplus$ is defined for $\mathcal{P}$ and $\mathcal{Q}$ then:

$$P + Q = p(\mathcal{P} \oplus \mathcal{Q})$$

Given that we want to work on equations and not on actual values, this rationalizer procedure has to be extended taking the coordinates as variables and the operations between them as polynomial equations. We also set $p((X_i, Y_i, 1)) = P_i$ to represent the coordinates of the points involved. The lemmas that precede can be generalized directly if we take now p' to be the composition of the previous p with the *evalp* function:

$$p' : \tau(V)^{3*} \to \mathbb{F}^2$$

such that

$$p'((\mu,\nu,\zeta)) = p(evalp((\mu,\nu,\zeta)))$$

Where *evalp* acts in each of the components of the triple.

One has to define when does a triple $(\mu,\nu,\zeta) \in \tau(V)$ represents a point of the curve once evaluated, this called a curve-point projection:

Definition 7.5 (Curve-point projection)**.**
Given $(\mu,\nu,\zeta) \in \tau(V)$, consider the polynomial:

$$\tau_{\mu,\nu,\zeta} = \nu^2 - (\mu^3 + A\mu\zeta^4 + B\zeta^6)$$

We say that (μ,ν,ζ) is a curve-point projection if $reduce(\tau_{\mu,\nu,\zeta}) = 0$.

This definition gives us the desired property:

Lemma 7.2.
If (μ,ν,ζ) is a curve-point projection and $P = p'((\mu,\nu,\zeta))$ then $P \in E(K)$.

In the projective space, points that lie on the same space line, represent the same plane point and are therefore considered to be equivalent. Here we want a similar property with the projection mapping that we have:

Definition 7.6 (Equivalent points in $\mathbb{F}^{3*}$)**.**
Let $P = (m,n,z), Q = (m',n',z') \in \mathbb{F}^{3*}$ then we say that P is equivalent to Q and we write it as $P \sim Q \iff reduce(\sigma) = reduce(\sigma')$ and $reduce(\tau) = reduce(\tau')$ where $\sigma = mz'^2, \sigma' = m'z^2, \tau = nz'^3, \tau' = n'z^3$.

As in the projective plane, equivalent lines represent the same point:

Lemma 7.3 (Equivalence implies equality of affine points)**.**
Let $\mathcal{P},\mathcal{Q} \in \mathbb{F}^{3*}$ and set $P = p(\mathcal{P}), Q = p(\mathcal{Q})$. Assume $P,Q \in E(K) \setminus \{\infty\}$ and that $\mathcal{P} \sim \mathcal{Q}$ then $P = Q$.

7.3 Verifying the associativity property

We put together the tools we have developed to finally verify the associativity property. There are some results that are purely computational the unrolling procedure of Inox is able too calculate them and then we have a list of derived results. We begin by listing the computational results:

For the derived properties, set

$$\Sigma = \Pi_0 \oplus \Pi_1 = (\mu,\nu,\zeta)$$

$$\Sigma' = \Pi_0 \oplus \Pi_0 = (\mu',\nu',\zeta')$$

$$\phi = ((\mu - X_1\zeta^2) + 2Y_1Y_2)^2 - (2Y_1Y_2)^2$$

$$\psi = (\mu' - X_2\zeta'^2)\zeta^2$$

Then the set of lemmas that drive towards associativity is shown in table 7. This finishes the proof of the associative law.

Name	Property
Prop-1	$-(\Pi_0 \oplus \Pi_0) \sim (-\Pi_0) \oplus (-\Pi_0)$
Prop-2	$-(\Pi_0 \oplus \Pi_1) \sim (-\Pi_0) \oplus (-\Pi_1)$
Prop-3	$(-\Pi_0) \oplus (\Pi_0 \oplus \Pi_1) \sim \Pi_0$
Prop-4	$(-\Pi_0) \oplus (\Pi_0 \oplus \Pi_1) \sim \Pi_1$
Prop-5	$\Pi_2 \oplus (\Pi_0 \oplus \Pi_1) \sim \Pi_1 \oplus (\Pi_0 \oplus \Pi_2)$
Prop-6	$\Pi_1 \oplus (\Pi_0 \oplus \Pi_0) \sim \Pi_0 \oplus (\Pi_0 \oplus \Pi_1)$
Prop-7	$(\Pi_0 \oplus \Pi_0) \oplus (\Pi_0 \oplus \Pi_0) \sim \Pi_0 \oplus (\Pi_0 \oplus (\Pi_0 \oplus \Pi_0))$
Prop-8	$(\Pi_0 \oplus \Pi_1) \oplus (\Pi_0 \oplus \Pi_1) \sim \Pi_0 \oplus (\Pi_1 \oplus (\Pi_0 \oplus \Pi_1))$
Prop-9	$reduce(\phi) = reduce(\psi)$
Prop-10	$(0,0,1) \oplus (\Pi_0 \oplus (0,0,1)) \sim \Pi_0$
Prop-11	$(\Pi_0 \oplus (0,0,1)) \oplus (\Pi_0 \oplus (0,0,1)) \sim \Pi_0 \oplus \Pi_0$

Table 6: Computational results derived from the definitions

Name	Property
Lemma-1	$-(P_0 + P_1) = (-P_0) + (-P_1)$
Lemma-2	If $P_0 + P_1 \neq \ominus P_0$ then $(\ominus P_0) \oplus (P_0 \oplus P_1) = P_1$
Lemma-3	If $P_0 + P_1 \neq P_2, -P_2 \wedge P_0 + P_2 \neq P_1, -P_1$ then $P_2 + (P_0 + P_1) = P_1 + (P_0 + P_2)$
Lemma-4	If $P_0 + P_1 \neq -(P_0 + P_1), P_0 + P_1 \neq -P_1, P_1 + (P_0 + P_1) \neq P_0, -P_0$ then $(P_0 + P_1) + (P_0 + P_1) = P_0 + (P_1 + (P_0 + P_1))$
Lemma-5	If $P_0 + P_1 = -P_0$ then $P_1 = -(P_0 + P_0)$
Lemma-6	If $P_0 + P_1 = -P_2$ then $(P_0 + P_1) + P_2 = P_0 + (P_1 + P_2)$
Lemma-7	$(P_0 + P_0) + P_1 = P_0 + (P_0 + P_1)$
Lemma-8	$(P_0 + (0,0)) + (P_0 + (0,0)) = P_0 + ((0,0) + (P_0 + (0,0)))$
Lemma-9	$(P_0 + P_1) + P_2 = P_0 + (P_1 + P_2)$

Table 7: Results derived from the above propositions

8 Conclusion and future work

We have conducted a proof on the group laws of addition over elliptic curves in the Welder proof assistant. The three first properties were easy to do but we realized that the fourth one needed special treatment. For solving this case, we have explored the relevant literature and found special procedures or tactics to solve it. Also, we have generalized the work in [Russsinoff(2015a)] to the case of general fields and general curves in the Weierstrass form. Converting these to the Montgomery form or the Edwards form should not be a bigger problem since the only thing that changes are the particular equations used. A second further improvement would be to instantiate the field and perform the multiplications algorithms for each case. The $HACL^*$ library is to our knowledge the first library that has verified the byte level scalar multiplication.

Dealing with theorem proving can be hard as proofs are not at straight forward as hand-written proofs. Dealing with a system under development like Welder can be hard if substantial changes are added to the system while working on it. That is why we worked with a fixed commit of Welder dated March 13th. Nevertheless, while working on the project we have encountered several improvements and modifications that should be ameliorated in future versions. We have also produced working notes that may be helpful for students that have to use Welder or Inox in the future. These notes will be available in our project repository.

Maybe the most urgent task for Welder is to work to ensure its soundness. As we pointed out in the Welder section there are many expressions coming from Inox that potentially can result in unsound proofs. The goal is to maintain compatibility with Inox expressions without duplicating its code (not introducing a new expression type in Welder) but assuring that expressions provided by the user are not illegal. This requires a careful examination of the Inox expressions to determine what is legal or not.

Another interesting addition would be the incorporation of tactics to the system. We have explored in this project the validation of tactic for proving equalities in commutative rings. But many other tactics could do things easier to users. The problem is that one should have a better idea of what Inox can solve or not in order to do good tactics. In this sense we think it would be interesting to explore the use of induction automation and resolution at the level of Welder. Some interesting articles in this sense appear in [Leino(2011)] and [Reynolds and Kuncak(2015)].

Finally, it would be interesting to turn Welder into a real interactive assistant where the user can specify the different rules he wants to apply, getting immediate feedback. The commands used could be then stored and reused as a proof. This would avoid the time spent doing compilation for each modification of the proof.

9 Acknowledgements

We have to thank many people that have collaborated with their advice and work during the realization of this semester project.

First of all, we have to thank professor Viktor Kunčak for accepting us to perform this work in his laboratory, for his advice and for the teaching we have received in the various courses we took with him. Doing formal verification can be very frustrating at times specially if one does not count with user-friendly tools but we have really enjoyed doing so.

We also have to thank the help we have received from doctoral students Nicolas Voirol and Romain Edelmann for guiding us through the complexities of Inox and Welder. Their time and dedication was really appreciated.

Finally, I would like to thank researchers Laurent Théry and Rustan

Leino whose advice guided us towards actual solutions to the problem we had and to all other researchers whose work we have used during this project. Their work has been really useful to do progress in the context of a proof assistant that is not mature enough yet.

We hope our work will be useful for future researchers and students that dive into the world of formal verification and cryptography using the tools developed at EPFL.

References

[Bartzia(2014)] S. P. Y. Bartzia, E. I. A formal library for elliptic curves in the coq proof assistant. *International Conference on Interactive Theorem Proving*, pages 77–92, 2014.

[Bernstein(2007)] L. T. Bernstein, D. J. Faster addition and doubling on elliptic curves. *International Conference on the Theory and Application of Cryptology and Information Security*, 48(177):29–50, 2007.

[Edwards(2007)] H. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, 2007.

[Friedl(1998)] S. Friedl. An elementary proof of the group law for elliptic curves. *Undergraduate thesis, Regensburg University*, pages 136–152, 1998.

[Gologlu and Granger(2013)] F. Gologlu and R. Granger. Solving a 6120-bit dlp on a desktop computer. *International Conference on Selected Areas in Cryptography*, pages 136–152, 2013.

[Gregoire and Mahboubi(2005)] B. Gregoire and A. Mahboubi. Proving equalities in a commutative ring done right in coq. *International Conference on Theorem Proving in Higher Order Logics.Springer Berlin Heidelberg*, pages 98–113, 2005.

[Hoare and Jifeng(1998)] C. A. Hoare and H. Jifeng. Unifying theories of programming. *Prentice Hall*, 1998.

[Kasper(2011)] E. Kasper. Fast elliptic curve cryptography in openssl. *International Conference on Financial Cryptography and Data Security*, 48(177):27–39, 2011.

[Koblitz(1999)] J. J. J. B. Koblitz, N. Differential power analysis. *Annual International Cryptology Conference*, 48(177):388–397, 1999.

[Koblitz(1987)] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[Leino(2011)] K. R. M. Leino. Automating induction with an smt solver. *Microsoft Research, Redmond, WA, USA*, 2011.

[McCarthy(1993)] J. McCarthy. Towards a mathematical science of computation. *Program Verification*, pages 35–56, 1993.

[Miller(1985)] V. S. Miller. Use of elliptic curves in cryptography. *Conference on the Theory and Application of Cryptographic Techniques*, 48(177):417–426, 1985.

[Paulson(1987)] L. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[R. Milner(1984)] R. B. R. Milner. The use of machines to assist in rigorous proof [and discussion]. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 322(1522):411–422, 1984.

[Raya(2017)] R. Raya. The proper way to use an adt invariant in inox, 2017. URL `https://stackoverflow.com/questions/43922985/the-proper-way-to-use-hasadtinvariant-in-inox`.

[Reynolds and Kuncak(2015)] A. Reynolds and V. Kuncak. Induction for smt solvers. *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 80–98, 2015.

[Russsinoff(2015a)] D. M. Russsinoff. Polynomial terms and sparse horner normal form. *Unpublished work*, 2015a.

[Russsinoff(2015b)] D. M. Russsinoff. A computationally surveyable proof of the curve25519 group axioms. *Unpublished work*, 2015b.

[Silverman and Tate(1992)] J. Silverman and J. Tate. *Rational points on elliptic curves*. New York: Springer-Verlag, 1992.

[Théry(2007)] L. Théry. Proving the group law for elliptic curves formally. Technical report, INRIA, 2007.

[Voirol(2017a)] N. Voirol. Modelling a class hierarchy in inox, 2017a. URL `https://stackoverflow.com/questions/43425885/modelling-a-class-hierarchy-in-inox`.

[Voirol(2017b)] N. Voirol. How to declare an abstract function in inox, 2017b. URL `https://stackoverflow.com/questions/43438768/how-to-declare-an-abstract-function-in-inox`.

[Washington(2008)] L. Washington. *Elliptic curves: number theory and cryptography*. CRC press, 2008.