Formalizing Elliptic Curve Cryptography in Higher Order Logic

Joe Hurd
Oxford University
joe.hurd@comlab.ox.ac.uk

12 October 2005

Abstract

Formalizing a mathematical theory using a theorem prover is a necessary first step to proving the correctness of programs that refer to that theory in their specification. This report demonstrates how the mathematical theory of elliptic curves and their application to cryptography can be formalized in higher order logic. This formal development is mechanized using the HOL4 theorem prover, resulting in a collection of formally verified functional programs (expressed as higher order logic functions) that correctly implement the primitive operations of elliptic curve cryptography.

Contents

Chapter 1

Introduction

There are many cryptographic operations, including ElGamal encryption, the Digital Signature Algorithm, and Diffie-Hellman key exchange, that rely on the discrete logarithm problem for their security. The discrete logarithm problem is based on an arbitrary group, and given two group elements g and h a would-be attacker must find an integer k such that $g^k = h$. Clearly the security of the discrete logarithm problem depends on which group it is based on, and the standard approach is to use a multiplicative group (i.e., multiplication modulo a large prime).

In 1985, Neal Koblitz and Victor Miller independently proposed basing the discrete logarithm problem on elliptic curve groups. Elliptic curves have been studied by mathematicians for over a hundred years, and have proved to be an effective tool in advanced number theory.[1] The elements of an elliptic curve group are the points on the curve, and the group operation is a way of 'adding' two points on the curve to get a third.

Taking into account the best known algorithms, Blake et al. (1999) present a correspondence between the key sizes of equal security discrete logarithm problems based on multiplicative and elliptic curve groups:

Multiplicative	**Elliptic curve**
1024 bits	173 bits
4096 bits	313 bits

As can be seen, elliptic curve groups require shorter keys than multiplicative groups, which make them an attractive choice in security applications with constraints on bandwidth or computation power (e.g., smart cards).

[1]The most famous example of this is the essential role played by elliptic curves in Wiles' 1995 proof of Fermat's Last Theorem.

One disadvantage of using elliptic curve groups instead of multiplicative groups is that the group operation is more complicated to implement. Consequently, even if the correctness of the basic arithmetic operations is assumed, it is not obvious that a particular program is a correct implementation of a group operation on elliptic curve points. The problem is exacerbated by the use of clever point representations that speed up elliptic curve operations. This is the problem addressed in the present project to formalize elliptic curve cryptography. The starting point is the mathematical definitions in the textbook *Elliptic Curves in Cryptography* by Blake, Seroussi, and Smart (1999), which are directly mechanized in higher order logic using the HOL4 theorem prover (Gordon and Melham, 1993). From these a set of programs are derived which can be formally verified to satisfy the mathematical definitions of the arithmetic operations on elliptic curve points. These programs are expressed as higher order logic functions, and can be both reasoned about and directly executed by the theorem prover.

In the initial stage of the project (which is the work described in this report), instead of verifying a separate set of higher order logic functions that implement the elliptic curve operations, a proof tool was developed that supports direct execution of the mathematical definitions in the theorem prover. This route was chosen for two reasons: the same proof tools could also be used to automate many of the routine proofs in the formalization; and it was important to be able to execute the mathematical definitions on some simple examples to check that no transcription errors occurred when typing them in from the textbook.

The next stage of the project will involve picking a clever point representation that has efficient algorithms for elliptic curve operations, and formalizing this in higher order logic. The mathematical definitions formalized in the initial stage will form the specification for a formal verification of this *Formalized Algorithm*, which is a component in the larger project entitled *Formal synthesis and verification of ARM software with applications to cryptography* (see Figure 1.1). The rest of the project will create a formally verified path from the elliptic curve operations formalized as higher order logic functions all the way to an ARM implementation. The end result will be a library of ARM machine code that is formally verified to execute the mathematical definitions of the elliptic curve operations, as formalized in this report.

There are other applications for a set of formally verified elliptic curve operations in higher order logic, besides creating a verified compilation to ARM machine code. Firstly, there is another project underway by Gordon et al. (2005) to build a verifying hardware compiler from a subset of higher order logic, allowing the elliptic curve operations to be implemented with

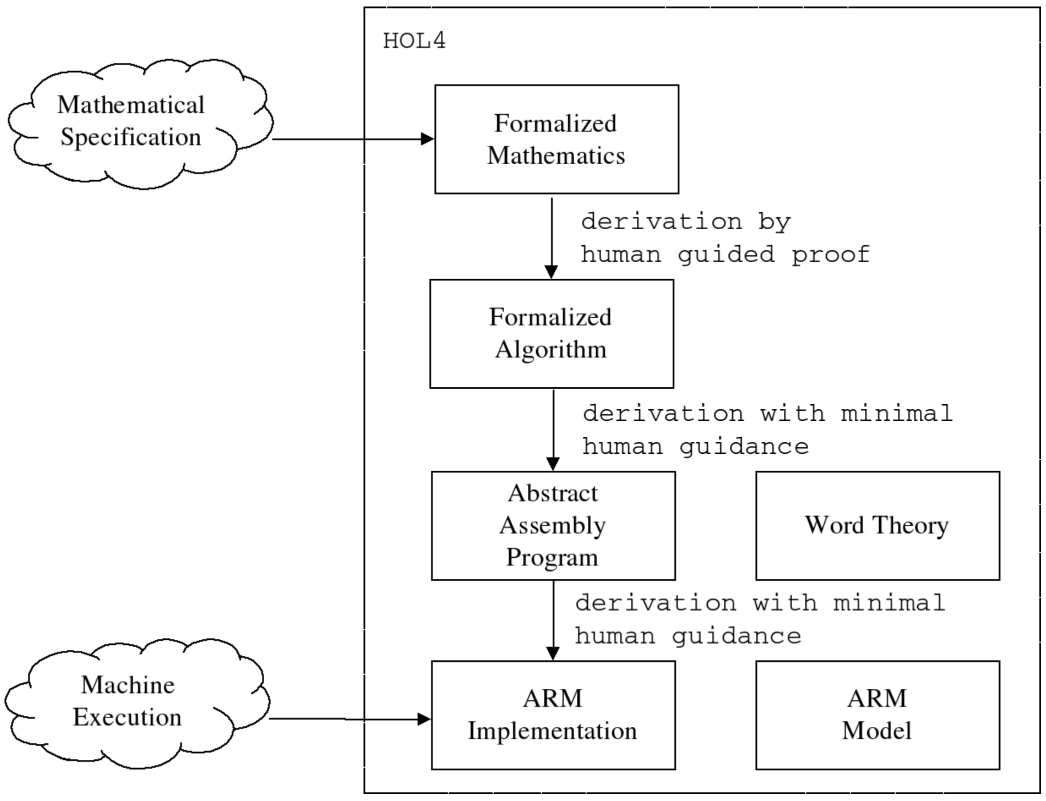
HOL4
Mathematical Specification
Formalized Mathematics
derivation by human guided proof
Formalized Algorithm
derivation with minimal human guidance
Abstract Assembly Program
Word Theory
derivation with minimal human guidance
Machine Execution
ARM Implementation
ARM Model

Figure 1.1: *Formal synthesis and verification of ARM software with applications to cryptography*: project overview.

high assurance in an FPGA. In addition, elliptic curve operations written in another programming language could be embedded in higher order logic and formally verified to satisfy their mathematical specification; it is likely that much effort would be saved by using the existing formally verified elliptic curve operations as a stepping stone in the proof. Some preliminary investigation of this approach has taken place for programs written in μCryptol, a domain specific language for cryptography developed at Galois Connections, Inc.

As stated above, this report describes the work completed in the initial stage of the project: the formalization of the mathematical definitions of elliptic curves into higher order logic; and the proof tools that were developed to prove the main results inside the theorem prover. Particularly close attention is paid to matching the formal versions of the mathematical definitions as closely as possible to the textbook versions, in order to minimize the semantic gap in the specification of programs that purport to implement elliptic curve operations. Chapter 2 introduces the mathematical definition of elliptic curves, and lays out the details of how the abstract algebra was formalized. In parallel with this activity some HOL4 proof tools were developed to reduce the amount of human guidance necessary to complete the formalized proofs. These proof tools are described in Chapter 3, together with the details of how the mathematical definitions were tested by executing them in the theorem prover.

Chapter 4 summarizes the work completed to date and looks at the next steps to take and promising areas of future research. All notation used in the report is defined in Appendix A: both the mathematical and HOL4 symbols. In addition, the author recognizes that few readers will have expertise in both elliptic curve cryptography and higher order logic theorem proving, and so two primers are included with this report: Appendix B introduces elliptic curves and their use in cryptography; and Appendix C introduces higher order logic and the HOL4 theorem prover.

Chapter 2

Formalized Mathematics

The precise nature of formalized mathematics is pinned down neatly in a survey article written by Harrison (1996).

> By *formalization* we mean expressing mathematics, both statements and proofs, in a (usually small and simple) formal language with strict rules of grammar and unambiguous semantics. [. . .] We can split the project of formalization into two parts:
>
> 1. Formalizing the statements of theorems, and the implicit context (definitions, etc.) on which they depend.
> 2. Formalizing the proofs of the results and subjecting them to precise checking.

For the purposes of this report the "small and simple formal language" is higher order logic, and the "precise checking of proofs" is achieved by expanding them to primitive inferences inside the HOL4 theorem prover.

The purpose of this chapter is to spell out the details of the formalized theories of elliptic curves, to convince the reader that there is no semantic gap between the standard mathematical definitions and the versions formalized in the HOL4 theorem prover. Thus when a theorem is eventually proved that an ARM machine code program implements an elliptic curve operation, there will be no doubt that the program is mathematically correct.

The mathematical theory of elliptic curves is built on top of abstract algebra, and so the formalization project begins with the well known concepts of groups, fields, polynomials and so on up to projective space. This initial formalization is described in Section 2.1, and serves as a useful introduction to how mathematics is formalized in the HOL4 theorem prover. The reader comfortable with abstract algebra may wish to skip this section on a first reading. In Section 2.2 there follows a description of the elliptic curve formalization, in which the main objective is for the formalized theories to be as faithful as possible to the source textbook (Blake et al., 1999). To finish the chapter, Section 2.3 describes a formalization of Galois fields, elliptic curve groups and ElGamal encryption, showing exactly how cryptography connects up with mathematics inside the theorem prover.

2.1 Abstract Algebra

Different mathematics textbooks offer different presentations of basic abstract algebra concepts such as groups and fields. Therefore, the following sections first fix the terminology used by introducing the mathematics at an informal level, and then show how it is formalized in higher order logic using the HOL4 theorem prover.

2.1.1 Groups

A group G is a quadruple

$$G = (G, e, \cdot^{-1}, \cdot)$$

consisting of a carrier set G, an identity element $e \in G$, an inverse function $\cdot^{-1} : G \to G$, and a binary function $\cdot : G \times G \to G$ called the group operation. A group must satisfy the following axioms:

$$\begin{array}{lr}\forall x \in G.\ ex = x & \text{(Left identity)}\ ; \\ \forall x \in G.\ x^{-1}x = e & \text{(Left inverse)}\ ; \\ \forall x, y, z \in G.\ (xy)z = x(yz) & \text{(Associativity)}\ .\end{array}$$

Examples of groups: $(\mathbb{Z},0,-,+)$; $(\mathbb{Q}^{\neq 0},1,1/\cdot,*)$. Non-examples: $(\mathbb{N}, 0, -, +)$; $(\mathbb{Q}, 1, 1/\cdot, *)$.

This definition of groups is formalized in higher order logic in a two stage process that occurs many times in this report. In the first stage the syntax of groups is formalized as a new higher order logic type, using the HOL4 datatype package. The command

```
Hol_datatype
  `group = <| carrier : 'a -> bool;
              id : 'a;
              inv : 'a -> 'a;
              mult : 'a -> 'a -> 'a |>`;
```

creates a new (polymorphic) record type α group together with record accessor constants carrier, id, inv and mult. The polymorphism is present to allow the theory of groups to be applied to any formalized groups, regardless of the higher order logic type of the group elements. In addition, the standard properties that one would expect a record type to satisfy, such as

```
|- !c e i m.
     <| carrier = c; id = e; inv = i; mult = m |>.carrier = c ,
```

are automatically proved as higher order logic theorems.[1] In common with all the HOL4 proof tools described in this report, the datatype package does not introduce any new axioms. Instead, it reduces the construction of the record type and the proof of all its properties to primitive inferences of higher order logic.[2] Thus, no unsoundness can creep in to a theory from defining new types and constants.

At this point it is often useful to define some group operations in terms of the record accessors in the type definition. In this case only group exponentiation needs to be defined (since it is used in the discrete logarithm problem):

```
|- (group_exp G g 0 = G.id) /\
   (group_exp G g (SUC n) = G.mult g (group_exp G g n)) .
```

The second stage of formalizing the mathematical definition of groups is to define a new Group class consisting of all elements of type α group that satisfy the group axioms.[3] This is a straightforward constant definition, and results in the higher order logic theorem

```
|- Group =
   { g |
     g.id IN g.carrier /\
     (!x y :: (g.carrier). g.mult x y IN g.carrier) /\
     (!x :: (g.carrier). g.inv x IN g.carrier) /\
     (!x :: (g.carrier). g.mult g.id x = x) /\
     (!x :: (g.carrier). g.mult (g.inv x) x = g.id) /\
     (!x y z :: (g.carrier).
        g.mult (g.mult x y) z = g.mult x (g.mult y z)) } .
```

There are a few points to note about the formalized version of the definition of groups:

- Firstly, the mathematical definition required the inverse to be a function $G \to G$, but in the formalized version there can be elements of type α group for which this is not the case. Therefore this closure

[1]Please refer to Appendix A for a glossary of HOL4 logical symbols.

[2]The primitive inferences of higher order logic contain two primitive definition principles: one for new types, which is used to create the record type; and one for new constants, which used to create the record accessors.

[3]The word class is used here as an aid to the reader; to the theorem prover classes are just higher order logic sets.

property is listed as an explicit membership requirement of the Group set (along with similar closure properties for the identity element and group operation).

- Secondly, the group axioms have *not* been translated to new axioms of higher order logic. Instead all that has occurred is the definition of a new constant, which preserves soundness.

- Lastly, the formalized version of the definition of groups is bulkier than the mathematical one. Partly this is due to increased precision, such as different syntax for the carrier set of the group and the group itself, but partly this is a natural consequence of formalizing mathematics into logic.

This concludes the formalization of the definition of groups as presented above, but one final step is needed to formalize the standard set of group axioms. The group axioms as presented are a minimal set, resulting in a simple definition of Group containing a minimal number of conjuncts. Using a minimal set of axioms has the advantage that it is easier in future to prove that a given element of α group is indeed a group. However, the disadvantage is that a one-off proof is required to show that all of the standard group axioms follow from the minimal set. For example, the statement 'multiplying a group element by its inverse on the right results in the group identity' is usually given as a standard axiom of group theory, and the following theorem shows that it can be proved from the minimal set of axioms presented here:

```
|- !g :: Group. !x :: (g.carrier). g.mult x (g.inv x) = g.id .
```

Note how the outermost quantifier guarantees both that g is an element of type α group and satisfies the minimal set of group axioms. Deriving standard axioms from minimal sets can sometimes be tricky, and formalizing the proof of the right inverse axiom required guiding the theorem prover through the following steps of equality reasoning:

$$\begin{aligned} xx^{-1} &= exx^{-1} \\ &= (x^{-1})^{-1}x^{-1}xx^{-1} \\ &= (x^{-1})^{-1}ex^{-1} \\ &= (x^{-1})^{-1}x^{-1} \\ &= e\ . \end{aligned}$$

In a similar way all the standard group axioms can be formalized in the theorem prover.

After formalizing the definition of groups, there are many useful group laws that need to be proved, such as this alternative definition of the identity element as the unique group element that is equal to its square:

```
|- !g :: Group. !x :: (g.carrier). (g.mult x x = x) = (x = g.id) .
```

So far the only group laws that have been needed are 'first order' consequences of the group axioms, and are easily formalized using the proof tools described in Chapter 3 plus some manual guidance. However, it is not the purpose of this chapter to go through each theorem proved in the formalized theories, but rather to demonstrate that the formalized definitions are faithful to standard mathematics.

It is useful to consider two subclasses of groups: finite groups and Abelian groups. Finite groups are simply groups with a finite carrier set, and a group G is Abelian if it satisfies the additional property

$$\forall x, y \in G.\ xy = yx \quad \text{(Commutativity)}\ .$$

Examples of Abelian groups: $(\mathbb{Z}, 0, -, +)$, $(\mathbb{Q}^{\neq 0}, 1, 1/\cdot, *)$. Examples of non-Abelian groups: (invertible $n \times n$ matrices over $\mathbb{R}$, identity matrix, matrix inverse, matrix multiplication); (permutations of n objects, leave everything alone, inverse permutation, perform one permutation and then the next).

These subclasses of groups are easily formalized in higher order logic with the following definitions:

```
|- AbelianGroup =
   { g |
     g IN Group /\
     !x y :: (g.carrier). g.mult x y = g.mult y x } ;

|- FiniteGroup = { g | g IN Group /\ FINITE g.carrier } ;

|- FiniteAbelianGroup =
   { g | g IN FiniteGroup /\ g IN AbelianGroup } .
```

From these definitions it easily follows that the class of Abelian groups is a subclass of all groups, as expected.

2.1.2 Fields

The next step in formalizing abstract algebra is fields, where a field is a 7-tuple

$$K = (K, 0, 1, -, \cdot^{-1}, +, \cdot)$$

consisting of: a carrier set K; two elements $0, 1 \in K$; one unary function $- : K \to K$; one unary function $\cdot^{-1} : K^* \to K^*$ (where $K^* = K - \{0\}$); and two binary functions $+, \cdot : K \times K \to K$. A field K must satisfy the following axioms:

$$\begin{array}{lr}
(K, 0, -, +) \text{ is an Abelian group} & \text{(Addition)} \\
(K^*, 1, \cdot^{-1}, \cdot) \text{ is an Abelian group} & \text{(Multiplication)} \\
\forall x \in K.\ 0x = 0 & \text{(Left zero)} \\
\forall x, y, z \in K.\ x(y+z) = xy + xz & \text{(Distributivity)}
\end{array}$$

Examples of fields are: $\mathbb{Q}$; $\mathbb{R}$; $\mathbb{C}$. Non-examples are: $\mathbb{Z}$; the set of polynomials with real coefficients.

Fields are formalized in two phases, following exactly the same procedure as for groups. The first phase defines the syntax, starting by creating a new type called α field:

```
Hol_datatype
  ‘field = <| carrier : ’a -> bool;
              sum : ’a group;
              prod : ’a group |>‘ .
```

Note how the additive (sum) and multiplicative (prod) groups are explicitly formalized in the type definition, while the primitive field operations are defined in terms of the record accessors:

```
|- field_zero f = f.sum.id ;

|- field_one f = f.prod.id ;

|- field_neg f = f.sum.inv ;

|- field_inv f = f.prod.inv ;

|- field_add f = f.sum.mult ;

|- field_mult f = f.prod.mult .
```

And the field syntax is completed with the rest of the standard field operations, which are in turn defined in terms of the primitives above:

```
|- field_sub f x y = field_add f x (field_neg f y) ;

|- field_div f x y = field_mult f x (field_inv f y) ;

|- field_nonzero f = f.carrier DIFF {field_zero f} ;

|- (field_num f 0 = field_zero f) /\
   (field_num f (SUC n) =
    field_add f (field_num f n) (field_one f)) ;

|- (field_exp f x 0 = field_one f) /\
   (field_exp f x (SUC n) =
    field_mult f x (field_exp f x n)) .
```

The second phase of formalizing fields concerns itself with the field axioms. Just as for groups, a new Field class is defined, which consists of all the elements of type α field that satisfy the field axioms:

```
|- Field =
   { f |
     f.sum IN AbelianGroup /\
     f.prod IN AbelianGroup /\
     (f.sum.carrier = f.carrier) /\
     (f.prod.carrier = field_nonzero f) /\
     (!x :: (f.carrier).
        field_mult f (field_zero f) x = field_zero f) /\
     (!x y z :: (f.carrier).
        field_mult f x (field_add f y z) =
        field_add f (field_mult f x y) (field_mult f x z)) } .
```

The final step requires that all the standard field axioms be proved from the minimal set presented here. The only one to cause any difficulty is the axiom that multiplying any field element by zero on the right always gives zero:

```
|- !f :: Field. !x :: (f.carrier).
     field_mult f x (field_zero f) = field_zero f .
```

The proof first establishes that $x0$ is indeed a field element, by expanding it as

$$x0 = x(1 + -1) = x1 + x(-1)$$

and using the closure laws from the additive and multiplicative groups. Next the equation

$$x0 = x(0 + 0) = x0 + x0$$

is derived, in which an element of the additive group is seen to be equal to its square. Therefore by the group law in Section 2.1.1 that element must be the identity, giving

$$x0 = 0$$

as required.

A subclass of fields that frequently appears in cryptography is the class of finite fields, in which the carrier set is finite:

```
|- FiniteField = { f | f IN Field /\ FINITE f.carrier } .
```

The ring of polynomials $K[X]$ over a field K consists of all expressions

$$a_0 + a_1 X + \cdots + a_n X^n$$

where $a_0, \ldots, a_n \in K$. The degree of a polynomial is the largest n such that a_n is non-zero, or zero if all coefficients are zero.

A field K is algebraically closed if the fundamental theorem of algebra holds in the field: i.e., given a polynomial $p(x)$ of degree at least one, there is an element $t \in K$ satisfying $p(t) = 0$. By adding extra elements it is possible to algebraically close a field K, and this is written $\overline{K}$. For example, algebraically closing $\mathbb{R}$ results in $\mathbb{C}$, but algebraically closing $\mathbb{Q}$ gives the (countable) field of algebraic numbers.

No new type is defined to formalize polynomials; instead the ring of polynomials over a field of type α field are represented as elements of type α list. In this scheme, the zero polynomial is represented by the empty list:

```
|- poly_zero = [] .
```

Not every list is a valid polynomial, so a new class $\mathsf{Poly}(K)$ is defined which consists of all the polynomials with coefficients from K and non-zero last coefficient (or the zero polynomial):

```
|- Poly f =
   { p |
     (p = poly_zero) \/
     (EVERY (\c. c IN f.carrier) p /\ ~(LAST p = field_zero f)) } .
```

The degree of a polynomial is very nearly the length of the representing list:

```
|- poly_degree p = if (p = poly_zero) then 0 else LENGTH p - 1 .
```

Finally, evaluating a polynomial at a field element is formalized like so:

```
|- (poly_eval f [] x = field_zero f) /\
   (poly_eval f (c :: cs) x =
    field_add f c (field_mult f x (poly_eval f cs x))) .
```

This is all the polynomial syntax necessary to formalize the class of algebraically closed fields:

```
|- AlgebraicallyClosedField =
   { f |
     f IN Field /\
     !p :: Poly f.
       (poly_degree p = 0) \/
       ?z :: (f.carrier). poly_eval f p z = field_zero f } .
```

2.1.3 Projective Space

As will be shortly seen in Section 2.2, elliptic curves are defined over projective space, and this is the last remaining concept of abstract algebra that is needed to support their formalization. The first step is a formalization of vector spaces.

The vector space K^n of dimension n over a field K consists of all n-tuples

$$(a_1, \ldots, a_n)$$

where the coordinates $a_1, \ldots, a_n$ are field elements of K. The origin in dimension n is the unique point where all coordinates are zero.

As for polynomials, vector space points over a field of type α field are formalized as elements of type α list.[4] In this representation the dimension function is simply the list length function, and the coordinate function is the function that picks out the nth element from a list. Also, the dimension n origin over a field K is a function that explicitly constructs the required list.

```
|- dimension = LENGTH ;

|- coord = EL ;

|- (origin f 0 = []) /\
   (origin f (SUC n) = field_zero f :: origin f n) .
```

Using these primitive definitions as an interface with the underlying list representation, the remaining vector space definitions need not refer to lists at all:

```
|- coords v = { i | i < dimension v } ;

|- vector_space f n =
   { v |
     (dimension v = n) /\
     !i :: coords v. coord i v IN f.carrier } .
```

Projective space $\mathbb{P}^n(K)$ of dimension n over the field K is the set of lines

$$\{(\alpha x_1, \ldots, \alpha x_{n+1}) \mid \alpha \in K\}$$

where $(x_1, \ldots, x_{n+1}) \in K^{n+1} - \{(0, \ldots, 0)\}$. Assuming $x_{n+1} \neq 0$, lines can be written in affine coordinates as

$$(x_1/x_{n+1}, \ldots, x_n/x_{n+1}) \ .$$

[4]Recent work by Harrison (2005) demonstrates an alternative approach to formalizing vector spaces, in which higher order logic types can be used to enforce dimensionality constraints.

Before getting started on the formalization of projective space, it is useful to define the set of all points in a vector space except the origin:

```
|- nonorigin f =
   { v |
     v IN vector_space f (dimension v) /\
     ~(v = origin f (dimension v)) } .
```

As can be seen from the mathematical definition, projective space of dimension n is actually a quotient of a vector space of dimension $n+1$. Two points are equal in the quotient space if they lie on the same line through the origin. Here is one way to formalize this as a relation between points in a vector space:

```
|- project f v1 v2 =
   (v1 = v2) \/
   (v1 IN nonorigin f /\ v2 IN nonorigin f /\
    (dimension v1 = dimension v2) /\
    ?c :: (f.carrier). !i :: coords v1.
      field_mult f c (coord i v1) = coord i v2)
```

This relation is carefully constructed to always be an equivalence relation (regardless of the first argument). As can be easily formalized in higher order logic, every equivalence relation R satisfies the property

$$\forall x, y.\ R\ x\ y \iff (R\ x = R\ y)\ .$$

Note that the $R\ x$ and $R\ y$ here are functions from elements to booleans, and thus are uniquely determined by the set of elements that are mapped to true. In the case of project $f\ v$, that set consists of all the points on the straight line through the origin and the vector v. This motivates the following formalization of projective space:

```
|- projective_space f n =
   {project f v | v IN vector_space f (n + 1) INTER nonorigin f}
```

From the preceding discussion, two lines project $f\ v_1$ and project $f\ v_2$ are equal if and only if the relation project $f\ v_1\ v_2$ holds. This property greatly simplifies reasoning about projective space.

A line in affine coordinates is also formalized in terms of the project relation:

```
|- affine f v = project f (v ++ [field_one f])
```

Two lines affine $f\ v_1$ and affine $f\ v_2$ are equal if and only if $v_1 = v_2$. It is this property that makes affine coordinates much easier to work with than projective space, in higher order logic as well as mathematics.

2.2 Elliptic Curves

This section defines the points and operations on elliptic curves over an arbitrary field. An algebraic approach is used, which is more suitable for formalization in a theorem prover than the geometric approach taken in the elliptic curve primer in Appendix B.

The definitions of elliptic curves, rational points and elliptic curve arithmetic presented here all come from the source textbook for the formalization: *Elliptic Curves in Cryptography* by Blake, Seroussi, and Smart (1999). The purpose of this chapter is to demonstrate that the formalized definitions in the theorem prover faithfully preserve the meaning of the mathematical definitions in the textbook, and so to aid direct comparison the critical definitions are copied verbatim from the textbook.

2.2.1 Rational Points

Firstly, here is the textbook definition of an elliptic curve:

> Let K be a field [and] $\overline{K}$ its algebraic closure [...] An elliptic curve over K will be defined as the set of solutions in the projective plane $\mathbb{P}^2(\overline{K})$ of a homogenous *Weierstrass equation* of the form
>
> $$E : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$$
>
> with $a_1, a_2, a_3, a_4, a_6 \in K$.

Note that since every term in the elliptic curve equation has degree 3, one solution (X, Y, Z) of the equation gives an entire line $\alpha(X, Y, Z) = (\alpha X, \alpha Y, \alpha Z)$ of solutions. However, this is only part of the definition, because not every equation of this form is a valid elliptic curve.

> Such a curve should be non-singular [...] Given a curve defined [as above], it is useful to define the following constants for use in later formulae:
>
> $$\begin{array}{l} b_2 = a_1^2 + 4a_2, \quad b_4 = a_1a_3 + 2a_4, \quad b_6 = a_3^2 + 4a_6, \\ b_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2, \quad [\ldots] \end{array}$$
>
> The *discriminant* of the curve is defined as
>
> $$\Delta = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6.$$
>
> [...] A curve is then non-singular if and only if $\Delta \neq 0$.

Although this mathematical definition of an elliptic curve may seem complicated, it is quite straightforward to formalize it using the techniques and theories of Section 2.1. The only parameters of the elliptic curve equation are the underlying field and the coefficients $a_1, \ldots, a_6$, so these form the components of a new type called α curve:

```
Hol_datatype
  `curve =
   <| field : 'a field;
      a1 : 'a;
      a2 : 'a;
      a3 : 'a;
      a4 : 'a;
      a6 : 'a |>` .
```

Following the type definition comes the definition of the elliptic curve operations, beginning with the constants b_2, b_4, b_6, b_8:

```
|- curve_b2 e =                       |- curve_b4 e =
   let f = e.field in                    let f = e.field in
   let $& = field_num f in               let $& = field_num f in
   let $+ = field_add f in               let $+ = field_add f in
   let $* = field_mult f in              let $* = field_mult f in
   let $** = field_exp f in              let a1 = e.a1 in
   let a1 = e.a1 in                      let a3 = e.a3 in
   let a2 = e.a2 in                      let a4 = e.a4 in
   a1 ** 2 + & 4 * a2 ;                  a1 * a3 + & 2 * a4 ;

|- curve_b8 e =                       |- curve_b6 e =
   let f = e.field in                    let f = e.field in
   ...                                   ...
   let a1 = e.a1 in                      let a3 = e.a3 in
   let a2 = e.a2 in                      let a6 = e.a6 in
   let a3 = e.a3 in                      a3 ** 2 + & 4 * a6 ;
   let a4 = e.a4 in
   let a6 = e.a6 in
   a1 ** 2 * a6 + & 4 * a2 * a6 -
   a1 * a3 * a4 + a2 * a3 ** 2 - a4 ** 2 .
```

The most noticeable aspect of these definitions is the use of lets to improve the readability of the field operations. Using this shorthand, it is easy to see that the formalized constants are a direct translation of the mathematical definitions. In the definition of the b_6 and b_8 constants the lets for the field operations have been elided, and this abbreviated form will be used from now on to improve the presentation.[5] Next to be formalized is the discriminant of an elliptic curve:

```
|- discriminant e =
   let f = e.field in
   ...
   let b2 = curve_b2 e in
   let b4 = curve_b4 e in
   let b6 = curve_b6 e in
   let b8 = curve_b8 e in
   & 9 * b2 * b4 * b6 - b2 * b2 * b8 -
   & 8 * b4 ** 3 - & 27 * b6 ** 2 .
```

And the final piece of syntax is the definition of non-singularity:

```
|- non_singular e = ~(discriminant e = field_zero e.field)
```

The formalization of elliptic curve equations is completed by defining a class Curve consisting of every element of type α curve satisfying all the requirements in the mathematical definition:

```
|- Curve =
   { e |
     e.field IN Field /\
     e.a1 IN e.field.carrier /\
     e.a2 IN e.field.carrier /\
     e.a3 IN e.field.carrier /\
     e.a4 IN e.field.carrier /\
     e.a6 IN e.field.carrier /\
     non_singular e } .
```

The set $E(\hat{K})$ of rational points on the elliptic curve are considered next:

> Let $\hat{K}$ be a field satisfying $K \subseteq \hat{K} \subseteq \overline{K}$. A point (X, Y, Z) on the curve is $\hat{K}$-rational if $(X, Y, Z) = \alpha(\hat{X}, \hat{Y}, \hat{Z})$ for some $\alpha \in \overline{K}$, $(\hat{X}, \hat{Y}, \hat{Z}) \in \hat{K}^3 - \{(0, 0, 0)\}$, i.e., up to projective equivalence, the coordinates of the points are in $\hat{K}$.

Note that if $K \subseteq \hat{K}$ then the coefficients of the elliptic curve equation can be considered to be from $\hat{K}$. Thus the formalized definition of rational points assumes that $\hat{K} = K$:

[5]These lets could be eliminated by the use of locales (as used in the Isabelle theorem prover), but locales are not currently implemented in HOL4.

```
|- curve_points e =
   let f = e.field in
   ...
   let a6 = e.a6 in
   { project f [x; y; z] |
     [x; y; z] IN nonorigin f /\
     (y ** 2 * z + a1 * x * y * z + a3 * y * z ** 2 =
      x ** 3 + a2 * x ** 2 * z + a4 * x * z ** 2 + a6 * z ** 3) } .
```

This is a case where the formalized definition significant deviates from the mathematical definition, in which the rational points over the field $\hat{K}$ are a subset of the rational points over the algebraic closure $\overline{K}$. However, if the rational points were formalized as a subset, then this would result in the field elements of $\hat{K}$ having the same higher order logic type as the field elements of $\overline{K}$. For many fields, especially the finite fields which are of principal interest in cryptography, it would be difficult and unnatural to formalize them with this constraint. Therefore, only the field $\hat{K}$ is mentioned in the definition of rational points: the reference to $\overline{K}$ is completely dropped; and K is not needed if the coefficients are considered to come from $\hat{K}$.

The above definition of rational points uses projective space, but it is usually more convenient to use affine coordinates:

> The curve has exactly one rational point with coordinate Z equal to zero, namely $(0, 1, 0)$. This is the *point at infinity*, which will be denoted by $\mathcal{O}$.
>
> For convenience, we will most often use the *affine* version of the Weierstrass equation, given by
>
> $$E : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$
>
> where $a_i \in K$. The $\hat{K}$-rational points in the affine case are the solutions to E in $\hat{K}^2$, and the point at infinity $\mathcal{O}$. [...] We will switch freely between the projective and affine presentations of the curve, denoting the equation in both cases by E. For $Z \neq 0$, a projective point (X, Y, Z) satisfying [the projective version of E] corresponds to the affine point $(X/Z, Y/Z)$ satisfying [the affine version of E].

For example, taking the underlying field to be $\mathbb{R}$, the curves in Figure 2.1 depict the solutions in $\mathbb{R}^2$ of different elliptic curve equations in affine coordinates.

The first step to formalizing the affine version of elliptic curves is to define the point at infinity $\mathcal{O}$:

```
|- curve_zero e =
   project e.field
     [field_zero e.field; field_one e.field; field_zero e.field] .
```

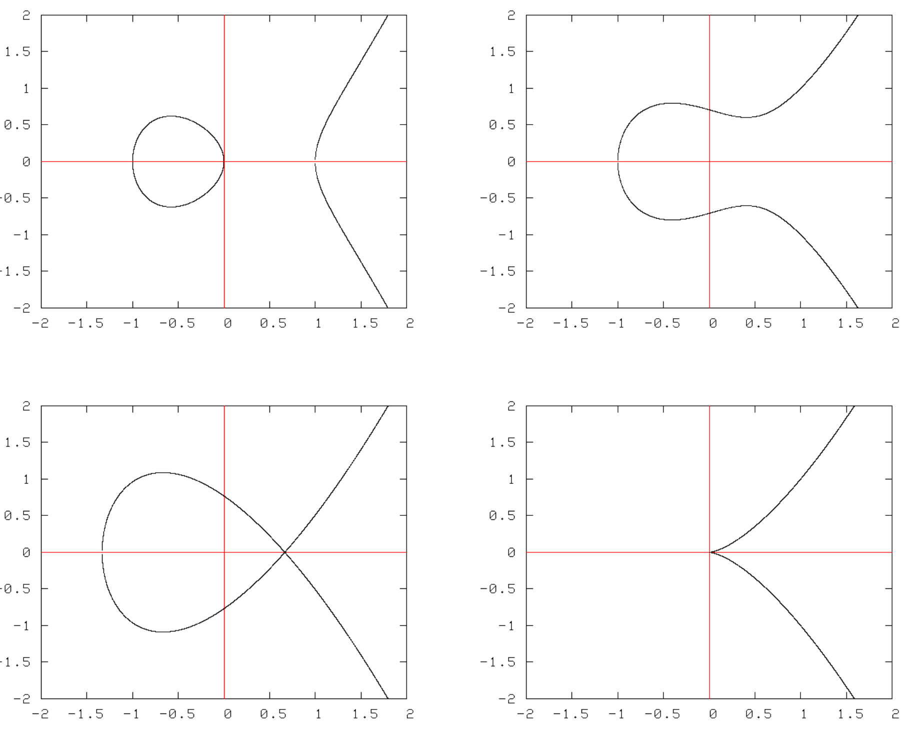

Figure 2.1: Example elliptic curves, clockwise from top left: $y^2 = x^3 - x$; $y^2 = x^3 - \frac{1}{2}x + \frac{1}{2}$; $y^2 = x^3$; and $y^2 = x^3 - \frac{4}{3}x + \frac{16}{27}$.

From the formalized definition of rational points on the projective version of elliptic curves, it is possible to recover the affine version as a theorem:

```
|- !e :: Curve. curve_zero e IN curve_points e ;

|- !e :: Curve. !x y :: (e.field.carrier).
     affine e.field [x; y] IN curve_points e =
     let f = e.field in
     ...
     let a6 = e.a6 in
     y ** 2 + a1 * x * y + a3 * y =
     x ** 3 + a2 * x ** 2 + a4 * x + a6 .
```

The mathematical definition of rational points in affine coordinates states explicitly that the every rational points is either $\mathcal{O}$ or is a solution of the elliptic curve equation, and implicitly assumes the 'obvious fact' that the point at infinity $\mathcal{O}$ is cannot be expressed in affine coordinates. Both these facts are proved as theorems in the formalization:

```
|- !e :: Curve. !p :: curve_points e.
     (p = curve_zero e) \/
     ?x y :: (e.field.carrier). p = affine e.field [x; y] ;

|- !e :: Curve. !x y.
     ~(curve_zero e = affine e.field [x; y]) .
```

Note: When the characteristic of the underlying field K of an elliptic curve is not equal to either 2 or 3, it is possible to perform a change of variable transformation to the affine version of the elliptic curve equation which results in an isomorphic curve with the simpler equation

$$E : Y^2 = X^3 + aX + b \ .$$

This justifies the intuitive presentation in Appendix B where the underlying field was assumed to be $\mathbb{C}$ (which has characteristic 0).

2.2.2 Elliptic Curve Arithmetic

This section describes a formalization of elliptic curve arithmetic, again focusing on the comparison with the mathematical definitions as given in Blake et al. (1999). This uses the formalization of the affine version of the elliptic curve equation, because the textbook presents elliptic curve arithmetic in affine coordinates. Thus before tackling the definitions of elliptic curve arithmetic, a 'case theorem' is proved that supports the definition of functions on elliptic curve points using affine coordinates:

```
|- !e :: Curve. !z f.
     (curve_case e z f (curve_zero e) = z) /\
     !x y. curve_case e z f (affine e.field [x; y]) = f x y .
```

Although this looks like a theorem, it is actually a definition of the constant curve_case by new specification.[6] The best way to see how curve_case is used is by example, and the operations of elliptic curve arithmetic will provide several.

The textbook defines all the operations of elliptic curve arithmetic in one passage, reproduced here in full:

> Let E denote an elliptic curve given by
>
> $$E : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$
>
> and let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ denote points on the curve. Then
>
> $$-P_1 = (x_1, -y_1 - a_1x_1 - a_3) \ .$$
>
> Set
>
> $$\lambda = \frac{y_2 - y_1}{x_2 - x_1}, \quad \mu = \frac{y_1x_2 - y_2x_1}{x_2 - x_1}$$
>
> when $x_1 \neq x_2$, and set
>
> $$\lambda = \frac{3x_1^2 + 2a_2x_1 + a_4 - a_1y_1}{2y_1 + a_1x_1 + a_3}, \quad \mu = \frac{-x_1^3 + a_4x_1 + 2a_6 - a_3y_1}{2y_1 + a_1x_1 + a_3}$$
>
> when $x_1 = x_2$ and $P_2 \neq -P_1$. If
>
> $$P_3 = (x_3, y_3) = P_1 + P_2 \neq \mathcal{O}$$
>
> then x_3 and y_3 are given by the formulae
>
> $$\begin{aligned} x_3 &= \lambda^2 + a_1\lambda - a_2 - x_1 - x_2 \ , \\ y_3 &= -(\lambda + a_1)x_3 - \mu - a_3 \ . \end{aligned}$$

The first and simplest operation to be formalized is negation, which can be expressed using the new curve_case constant:

```
|- curve_neg e =
   let f = e.field in
   ...
   let a3 = e.a3 in
   curve_case e (curve_zero e)
     (\x1 y1.
        let x = x1 in
        let y = ~y1 - a1 * x1 - a3 in
        affine f [x; y]) .
```

[6] Constants are defined by new specification by proving a ‘witness theorem’ of the form $\vdash \exists x. \, \phi(x)$, after which a new constant c is created with defining property $\vdash \phi(\mathsf{c})$.

How does this work, say to evaluate curve_neg E P? Expanding the definition of curve_neg above (and the lets) will result in a term of the form

$$\mathsf{curve\_case}\ E\ \mathcal{O}\ (\lambda\, x_1, y_1.\ \ldots)\ P\ .$$

Now, using the definition of curve_case, if the argument P is the point at infinity $\mathcal{O}$, then the result will be the second argument of curve_case, which in this case is $\mathcal{O}$. If P is not the point at infinity then it must be a point on the curve that can be expressed as affine K x_1 y_1 (where K is the underlying field of E), and in this case the function in the third argument is called with arguments x_1 and y_1. The end result is the two theorems

$$\vdash\ \mathsf{curve\_neg}\ E\ \mathcal{O} = \mathcal{O}\ ,$$
$$\vdash\ \mathsf{curve\_neg}\ E\ (\mathsf{affine}\ K\ x_1\ y_1) = \mathsf{affine}\ K\ x_1\ (-y_1 - a_1x_1 - a_3)\ .$$

It is not much harder to formalize point doubling in the same way, although a careful reading is required to be sure of correctly catching and handling the two special cases $P = \mathcal{O}$ and $P = -P$:

```
|- curve_double e =
   let f = e.field in
   ...
   let a6 = e.a6 in
   curve_case e (curve_zero e)
   (\x1 y1.
    let d = & 2 * y1 + a1 * x1 + a3 in
    if d = field_zero f then curve_zero e
    else
      let l = (& 3 * x1 ** 2 + & 2 * a2 * x1 + a4 - a1 * y1) / d in
      let m = (~(x1 ** 3) + a4 * x1 + & 2 * a6 - a3 * y1) / d in
      let x = l ** 2 + a1 * l - a2 - &2 * x1 in
      let y = ~(l + a1) * x - m - a3 in
      affine e.field [x; y]) .
```

The final operation of adding two points on the curve requires two nested occurrences of curve_case, one for each of the argument points:

```
|- curve_add e p1 p2 =
   if p1 = p2 then curve_double e p1
   else
     let f = e.field in
     ...
     let a6 = e.a6 in
     curve_case e p2
```

```
        (\x1 y1.
           curve_case e p1
             (\x2 y2.
                if x1 = x2 then curve_zero e
                else
                  let d = x2 - x1 in
                  let l = (y2 - y1) / d in
                  let m = (y1 * x2 - y2 * x1) / d in
                  let x = l ** 2 + a1 * l - a2 - x1 - x2 in
                  let y = ~(l + a1) * x - m - a3 in
                  affine e.field [x; y]) p2) p1 .
```

Again, after some thought that all the special cases have been correctly taken care of, it is easy to see that the formulas in the definition are directly translated from the textbook.

2.3 Cryptography

At this point all of the key definitions have been covered, and hopefully the reader is convinced that the higher order logic theory of elliptic curves is faithful to the mathematics. The rest of this chapter will focus on applications of the formalized theory to cryptography.

2.3.1 Galois Fields

So far the formalization has treated fields abstractly, and no concrete fields have yet been introduced. Applying the theory to cryptography will require formalizing a concrete finite field. It is a fact that for each prime power $q = p^n$, there is (up to isomorphism) only one finite field having a carrier set of size q. This field is called the Galois field of size q, and written GF(q). Furthermore, there are no other finite fields. For cryptographic applications, the finite fields of greatest interest are GF(p) (where p is a large prime) and GF(2^n).

So far in this project only the finite fields of the form GF(p) have been formalized, in which all arithmetic is performed modulo p. The formalized definition is

```
|- GF p =
   <| carrier := { n | n < p };
      sum := add_mod p;
      prod := mult_mod p |>
```

where the additive and multiplication groups are defined in the obvious way.[7] The main result is that GF(p) is a finite field whenever p is a prime:

```
|- !p :: Prime. GF p IN FiniteField .
```

2.3.2 Elliptic Curve Groups

As mentioned in the introduction and expanded on in Appendix B, many useful cryptographic operations are based on the discrete logarithm problem, which in turn is based on an arbitrary group. The security of the cryptographic operations thus depends on the precise group used, and (thus far) elliptic curve groups have proved highly resistant to attack.

Given an elliptic curve E with underlying field K, then

$$(E(K),\ \mathcal{O},\ -,\ +)$$

is an Abelian group, where $-$ is negation of elliptic curve points and $+$ is addition. If K is a finite field, then the number $\sharp E(K)$ of K-rational (affine) points is trivially bounded by

$$\sharp E(K) \leq (\sharp K)^2 + 1$$

and so the elliptic curve group is a finite Abelian group.

It is straightforward to formalize the definition of the elliptic curve group:

```
|- curve_group e =
   <| carrier := curve_points e;
      id := curve_zero e;
      inv := curve_neg e;
      mult := curve_add e |> .
```

However, a formal proof of the statement

```
!e :: Curve. curve_group e IN AbelianGroup
```

has not yet been constructed. Work on this is ongoing, and several lemmas such as the closure of elliptic curve negation have now been proved:

```
|- !e :: Curve. !p :: curve_points e.
     curve_neg e p IN curve_points e .
```

[7]The only tricky part is the definition of multiplicative inverse as $\lambda x.\ x^{p-2} \bmod p$, which is verified using Fermat's Little Theorem.

2.3.3 ElGamal Encryption

ElGamal encryption demonstrates how the discrete logarithm problem based on a group G can be used as a public key encryption algorithm. The presentation of the algorithm given here is the standard one, and follows Schneier (1996). Bob generates an instance $g^x = h$ of the discrete logarithm problem to create a new public and private key. Bob publishes the public key (g, h) while keeping the private key x secret. The following algorithm allows Alice to send a message $m \in G$ to Bob that cannot be read by a third party (this security property is called *confidentiality*).

1. Alice obtains a copy of Bob's public key (g, h).
2. Alice generates a randomly chosen natural number $k \in \{1, \ldots, \sharp G - 1\}$ and computes $a = g^k$ and $b = h^k m$.
3. Alice sends the encrypted message (a, b) to Bob.
4. Bob receives the encrypted message (a, b). To recover the message m he computes
$$ba^{-x} = h^k m g^{-kx} = g^{xk-xk} m = m \ .$$

The first step in formalizing ElGamal encryption is to define the packet that Alice sends to Bob:

```
|- elgamal G g h m k =
   (group_exp G g k, G.mult (group_exp G h k) m) .
```

This follows the algorithm precisely.

The following theorem demonstrates the correctness of ElGamal encryption, i.e., that Bob can decrypt the ElGamal encryption packet to reveal Alice's message (assuming he knows his private key x):

```
|- !G :: Group. !g h m :: G.carrier. !k x.
     (h = group_exp G g x) ==>
     (let (a,b) = elgamal G g h m k in
      G.mult (G.inv (group_exp G a x)) b = m)
```

The formalized version diverges slightly from the standard algorithm by having Bob compute $a^{-x}b$ instead of ba^{-x}, but results in a stronger correctness theorem since the underlying group G does not have to be Abelian.

Suppose an implementation of ElGamal encryption over an elliptic curve group has been formalized, and verified to correctly implement the operations of elliptic curve arithmetic. The above correctness theorem of ElGamal encryption is sufficient to guarantee that executing the implementation of encryption followed by the implementation of decryption will always return the original message.

Chapter 3

Proof Tools

The previous chapter showed that higher order logic is expressive enough to formalize the theory of elliptic curves in a natural way, and this chapter addresses the problem of mechanizing proofs in this theory. The HOL4 theorem prover comes equipped with many proof tools that help to automate proof mechanization, but there are some features of the elliptic curve formalization that require special reasoning not covered by the standard tools. However, the LCF style design of the HOL4 theorem prover makes it straightforward to add new proof tools without compromising the soundness of the system. Theorems are elements of an abstract ML type and can only be created by the ML functions implementing the primitive inferences of higher order logic. This makes it impossible for a user proof tool to create a theorem in any other way than by expanding the proof to primitive inferences.

Section 3.1 describes a method for simulating 'predicate subtypes' in higher order logic, and the corresponding proof tool solves many goals in the formalization of abstract algebra. Section 3.2 describes a naive primality prover that is useful for proving that concrete finite fields satisfy the field axioms, and Section 3.3 demonstrates how HOL4 is used to test the mathematical definitions of elliptic curve operations by executing them within the theorem prover.

3.1 Predicate Subtype Prover

Predicate subtyping allows the creation of a new subtype corresponding to an arbitrary predicate, where elements of the new type are also elements of the containing type. This is impossible in the higher order logic type system, where all types are assumed to be disjoint. However, it is possible to simulate the reasoning of predicate subtypes at the term level using higher order logic sets, and this section presents a proof tool that implements this.

The theory behind this proof tool and a prototype implementation are described in Hurd (2001); the novelty of the proof tool described here is that it is simpler and also that it is integrated into the HOL4 simplifier as a decision procedure.

3.1.1 Predicate Subtypes in PVS

As stated above, predicate subtyping is impossible in the type system of higher order logic: Church's simple type theory extended with Hindley-Milner polymorphism. However, predicate subtyping has been integrated into the type system of the PVS theorem prover (Owre et al., 1999), and the aim of the proof tool described here is to provide the same reasoning power as the type checker in PVS.

As a simple illustration of predicate subtypes, the type of real division (/) in HOL4 is

$$\mathbb{R} \to \mathbb{R} \to \mathbb{R} \ ,$$

and in PVS is

$$\mathbb{R} \to \mathbb{R}^{\neq 0} \to \mathbb{R} \ ,$$

where $\mathbb{R}$ is the type of real numbers and $\mathbb{R}^{\neq 0}$ is the predicate subtype of non-zero real numbers (generated from the predicate $\lambda x.\ x \neq 0$). As a consequence of this, the term 1/0 type checks in HOL4, but causes a type error in PVS.

Type checking in the presence of predicate subtypes is an undecidable problem, so the PVS type checker may pass extra proof obligations onto the user to complete type checking. Therefore, the predicate subtype prover is not designed to be complete for a class of 'predicate subtype goals', but rather to prove as many subgoals as it can within a reasonable time limit.

3.1.2 Simulating Predicate Subtypes in HOL4

The key idea in the design of the predicate subtype prover is to reason with sets instead of using types. Instead of a type checker producing type judgements of the form

$$x : \tau \ ,$$

for some predicate subtype τ, the prover is designed to prove theorems of the form

$$\vdash x \in S \ .$$

The intuition is that if τ is a predicate subtype, then the set S contains precisely the elements of τ. The prover is integrated as a decision procedure in the HOL4 simplifier for proving side conditions. A brief glance at the formalization of elliptic curve theory in Chapter 2 reveals many theorems of the form

$$\forall x \in S.\ f(x) = g(x)\ .$$

Before the simplifier can apply such a theorem to rewrite a term $f(t)$ the side condition $t \in S$ must be proved, and this is where the predicate subtype prover is used.

For example, consider the right identity law of groups:

```
|- !g :: Group. !x :: (g.carrier). g.mult x g.id = x .
```

Before this theorem can be applied to simplify the term

$$G.\mathsf{mult}\ (G.\mathsf{inv}\ c)\ G.\mathsf{id}\ ,$$

the two side conditions

$$G \in \mathsf{Group} \quad \text{and} \quad G.\mathsf{inv}\ c \in G.\mathsf{carrier}\ .$$

must be proved. These two subgoals will introduce the two inference steps implemented in the predicate subtype prover: subtype reductions and subtype judgements.

Subtype Reductions: Given a subtype reduction theorem of the form

$$\vdash \forall \mathbf{x}.\ f_1(\mathbf{x}) \in S_1(\mathbf{x}) \land \cdots \land f_n(\mathbf{x}) \in S_n(\mathbf{x}) \implies f(\mathbf{x}) \in S(\mathbf{x})\ ,$$

whenever the predicate subtype prover is met with a goal matched by $f(\mathbf{x}) \in S(\mathbf{x})$ it will reduce it to the subgoals in the antecedent.

For example, suppose the goal is

$$G.\mathsf{inv}\ c \in G.\mathsf{carrier}$$

and the subtype reduction theorem

$$\vdash \forall g \in \mathsf{Group}.\ \forall x \in g.\mathsf{carrier}.\ g.\mathsf{inv}\ x \in g.\mathsf{carrier}$$

is available. The predicate subtype prover will reduce the goal to the subgoals

$$G \in \mathsf{Group} \quad \text{and} \quad c \in G.\mathsf{carrier}\ .$$

Subtype reductions are useful when there is no other way to reduce the goal to subgoals. In this case it doesn't make sense to write $G.\mathsf{inv}\ x$ in the goal unless $x \in G.\mathsf{carrier}$, and so reducing it according to the subtype reduction theorem can never be wrong.

Subtype Judgements: A subtype judgement theorem has exactly the same form as a subtype reduction theorem:

$$\vdash \forall \mathbf{x}.\ f_1(\mathbf{x}) \in S_1(\mathbf{x}) \wedge \cdots \wedge f_n(\mathbf{x}) \in S_n(\mathbf{x}) \implies f(\mathbf{x}) \in S(\mathbf{x}) \ .$$

The difference is the way it is applied: if the predicate subtype prover is met with a goal matching $f(\mathbf{x}) \in S(\mathbf{x})$ it will save the current state, try reducing the goal to the subgoals in the antecedent, but if this doesn't result in a proof will restore the saved state and try the other matching subtype judgements.

For example, suppose the goal is

$$G \in \mathsf{Group}$$

and the subtype judgement theorems

$$\begin{array}{l} \vdash \forall g \in \mathsf{AbelianGroup}.\ g \in \mathsf{Group} \\ \vdash \forall g \in \mathsf{FiniteGroup}.\ g \in \mathsf{Group} \end{array}$$

are available. The predicate subtype prover will first reduce the goal to the subgoal

$$G \in \mathsf{AbelianGroup}\ ,$$

but if this cannot be proved it will go back to the original goal and reduce it to

$$G \in \mathsf{FiniteGroup}\ .$$

If this also cannot be proved it will give up. Subtype judgements are useful whenever there are multiple paths possible that could potentially lead to a proof. In this case either of the two facts $G \in \mathsf{AbelianGroup}$ and $G \in \mathsf{FiniteGroup}$ may be available in the current logical context, and so the predicate subtype prover must try both judgements.

Given a goal G, the complete procedure of the predicate subtype prover is as follows:

1. Check whether there is anything in the logical context that matches G. If so, prove G directly.

2. Look for a matching subtype reductions. If any are available, then reduce the goal to subgoals $G_1, \ldots, G_n$ and call the prover recursively on each subgoal. If any of these recursive calls return failure then return failure for this call.

3. Look for matching subtype judgements. If any are available, then reduce the goal to subgoals $G_1, \ldots, G_n$ and call the prover recursively on each subgoal. If any of these recursive calls return failure then move on to the next matching subtype judgement.

4. Return failure.

Integrating the predicate subtype prover into the simplifier results in a powerful proof tool for formalized mathematics. For example, the following goal can be completely solved by this proof tool, although it requires 102 distinct calls to the predicate subtype prover to achieve this.

```
(field_add e.field (field_add e.field (field_exp e.field y 2)
(field_mult e.field (field_mult e.field e.a1 x) y)) (field_mult
e.field e.a3 y) =
field_add e.field (field_add e.field (field_add e.field (field_exp
e.field x 3) (field_mult e.field e.a2 (field_exp e.field x 2)))
(field_mult e.field e.a4 x)) e.a6)
  ==>
(field_add e.field (field_add e.field (field_exp e.field
(field_sub e.field (field_sub e.field (field_neg e.field y)
(field_mult e.field e.a1 x)) e.a3) 2) (field_mult e.field
(field_mult e.field e.a1 x) (field_sub e.field (field_sub e.field
(field_neg e.field y) (field_mult e.field e.a1 x)) e.a3)))
(field_mult e.field e.a3 (field_sub e.field (field_sub e.field
(field_neg e.field y) (field_mult e.field e.a1 x)) e.a3)) =
field_add e.field (field_add e.field (field_add e.field (field_exp
e.field x 3) (field_mult e.field e.a2 (field_exp e.field x 2)))
(field_mult e.field e.a4 x)) e.a6)
------------------------------------
  0.  e IN Curve
  1.  x IN e.field.carrier
  2.  y IN e.field.carrier
```

3.2 Primality Prover

The second proof tool developed for the elliptic curve formalization is a naive primality prover that operates on a number n by checking all potential factors from 2 up to $\sqrt{n}$. The motivation for this tool comes from the examples in Section 3.3 where the elliptic curve operations over an example curve are executed by the HOL4 theorem prover. Some of the theorems used by the simplifier to accomplish this assume that the underlying field satisfied the field axioms, expressed as the side condition

$$\mathrm{GF}(751) \in \mathsf{Field} \ .$$

The predicate subtype prover of Section 3.1 used the field theorems to reduce this goal to the subgoal

$$751 \in \mathsf{Prime} \ ,$$

but now the execution stalls without a proof that 751 is a prime.

The naive primality prover is implemented as a higher order logic function which is verified to return true if and only if the input is a prime number.[1] First, a square root function over the natural numbers is defined

```
|- nat_sqrt n k = if n < k * k then k - 1 else nat_sqrt n (k + 1)
```

and verified to return the smallest natural number at least as large as the square root of its first argument:

```
|- !n k. k * k <= n = k <= nat_sqrt n 0 .
```

Note that the second argument of the square root function is an accumulator that counts upwards from 0, a feature that meant the function required a special termination proof to be accepted by HOL4.

Next the primality checker is defined in terms of the square root function:

```
|- prime_checker n i =
   if i <= 1 then T
   else if n MOD i = 0 then F
   else prime_checker n (i - 1) .
```

This function also has an accumulator as its second parameter, but this one counts downwards and so termination is trivial. The final correctness theorem of the primality checker is the following:

```
|- !p. prime p = 1 < p /\ prime_checker p (nat_sqrt p 0) .
```

Given a term $\mathsf{prime}\ n$ (where n is a concrete natural number), the primality prover works as follows:

[1]The author is grateful to Michael Compton for suggesting this approach.

1. Note that the correctness theorem of the primality checker is expressed as a rewrite: this is applied to the input term to give the new term

   $$1 < n \land \mathsf{prime\_checker}\ n\ (\mathsf{nat\_sqrt}\ n\ 0)\ .$$

2. The new term is rewritten with the definitions of nat_sqrt, prime_checker and all the standard arithmetic operations. This will eventually reduce the new term to either $\top$ or $\bot$.

3. Combine the first two steps to return either the theorem $\vdash \mathsf{prime}\ n = \top$ or the theorem $\vdash \mathsf{prime}\ n = \bot$.

For example, here are the theorems resulting from the application of the primality prover to a prime and non-prime, respectively:

```
|- prime 751 = T ;
|- prime 91 = F .
```

In addition to the present application, it is occasionally useful to prove the primality of small numbers during a verification. For example, a recent paper required the primality of 65537 (which is $2^{16} + 1$) to verify the functional correctness of the IDEA encryption algorithm (Zhang and Slind, 2005). Using this primality prover might have simplified the proof, although such a 'large' number takes the primality prover 12 seconds and 717,360 primitive inferences to prove the required theorem:[2]

```
|- prime 65537 = T .
```

3.3 Verified Execution

The definitions of the elliptic curve operations in Section 2.2 were formalized in terms of affine coordinates, using the curve_case constant. This was done to be faithful to the presentation in the source textbook, but this section will describe an additional benefit: the elliptic curve operations can be executed by the HOL4 theorem prover on example elliptic curves.

3.3.1 Verified Execution of Elliptic Curve Operations

Execution in the HOL4 theorem prover is just a form of simplification: given an initial term t and a collection Δ of rewrite theorems, the theorem prover

[2]All timings in this report are for the Kananaskis 3 version of HOL4 running on Moscow ML 2.01 with a Pentium IV 3.2Ghz CPU.

rewrites t to a normal form u (w.r.t. Δ) and returns the theorem $\vdash t = u$. When Δ contains only simple rewrites of the form $\vdash x = y$ (as is often the case when using HOL4 to perform execution) then there is an efficient proof tool available to perform the simplification (Barras, 2000).

However, if Δ contains conditional rewrites or other advanced simplification features such as decision procedures then the HOL4 simplifier should be used instead. Almost every theorem in the elliptic curve theory is a conditional rewrite of the form

$$\vdash \forall x \in S.\ f(x) = g(x)\ ,$$

and so the simplifier (equipped with the predicate subtype prover as a decision procedure) must be used for verified execution of the elliptic curve operations.

All the elliptic curve operations are defined in terms of the curve_case constant, which has the following definition:

```
|- !e :: Curve. !z f.
     (curve_case e z f (curve_zero e) = z) /\
     !x y. curve_case e z f (affine e.field [x; y]) = f x y .
```

The form of this definition requires that curve_zero E and affine K $[x; y]$ be treated as primitive during execution (i.e., the definitions of curve_zero and affine must *not* be expanded).

This reduces the execution of elliptic curve operations to the execution of field operations, and the field operations of $\mathrm{GF}(p)$ are all defined in terms of arithmetic operations, which in theory the simplifier can handle itself. However, field inverse in $\mathrm{GF}(p)$ is defined as the function

$$\lambda k.\ k^{p-2} \bmod p\ ,$$

which requires some care to evaluate. By default the HOL4 simplifier will first evaluate the exponentiation by primitive recursion on the exponent (i.e., $O(p)$ multiplications) and then reduce (the very large result) modulo p at the very end. As it stands the operation is completely impractical to execute, but it can be made efficient by proving its equivalence to a repeated squaring implementation of modular exponentiation. The modexp function is a higher order logic implementation of repeated squaring to compute $a^n \bmod m$:

```
|- modexp a n m =
   if n = 0 then 1
   else if n MOD 2 = 0 then modexp ((a * a) MOD m) (n DIV 2) m
   else (a * modexp ((a * a) MOD m) (n DIV 2) m) MOD m .
```

This version requires only $O(\log p)$ multiplications, and moreover all intermediate calculations are kept small by reducing them modulo p. The correctness theorem for this optimized version of modular exponentiaton is

```
|- !a n m. 1 < m ==> (modexp a n m = (a ** n) MOD m) ,
```

which justifies using modexp to perform all GF(p) field inverses.

All the techniques necessary to execute elliptic curve operations over GF(p) have now been covered, and all that remains is to carefully select the rewrite theorems that the HOL4 simplifier will use. Care is required because not enough rewrite theorems may cause the execution to 'get stuck' and return a result that has not been evaluated to the normal form of an elliptic curve point. However, too many rewrite theorems may cause definitions to be expanded that are either assumed to be primitive for the execution (such as curve_zero), or are inefficient to execute (such as the original version of field inverse in GF(p)).

3.3.2 Examples

A test of the elliptic curve operations is provided by formalizing a simple exercise for the reader in Koblitz (1987).

The exercise uses the example curve $Y^2 + Y = X^3 - X$ over the field GF(751); the primality prover and the simplifier together can prove that the field satisfies the field laws and the elliptic curve is non-singular:

```
|- GF 751 IN Field ;
|- ec = curve (GF 751) 0 0 1 750 0 ;
|- ec IN Curve .
```

Note the use of 750 in the formalized version instead of -1 in the mathematics: a typical example of representation choices during formalization resulting in a loss of succintness.

The exercise next defines two points which the HOL4 simplifer can prove lie on the curve:

```
|- affine (GF 751) [361; 383] IN curve_points ec ;
|- affine (GF 751) [241; 605] IN curve_points ec .
```

The exercise requires the reader to perform some elliptic curve arithmetic, and check that the results lie on the curve. Again, this is no problem for the HOL4 simplifier:

```
|- curve_neg ec (affine (GF 751) [361; 383]) =
   affine (GF 751) [361; 367] ;
```

```
|- affine (GF 751) [361; 367] IN curve_points ec ;

|- curve_add ec (affine (GF 751) [361; 383])
                (affine (GF 751) [241; 605]) =
   affine (GF 751) [680; 469] ;
|- affine (GF 751) [680; 469] IN curve_points ec ;

|- curve_double ec (affine (GF 751) [361; 383]) =
   affine (GF 751) [710; 395] ;
|- affine (GF 751) [710; 395] IN curve_points ec .
```

Together, the verified execution of these six theorems took 72 seconds and 961,068 primitive inferences to complete: a performance that reflects the abstract nature of the definitions involved.

Chapter 4

Summary

This report has presented a formalization of elliptic curve theory in higher order logic, mechanized using the HOL4 theorem prover. The proof of the group law for elliptic curves is not yet complete, but enough theory and proofs have now been mechanized to demonstrate the feasibility of the project.

The main research contributions of this work are as follows:

- the development of a practical approach for formalizing abstract algebra in higher order logic;
- a formalization of the mathematical theory of elliptic curves;
- the integration of a proof tool for reasoning about subtypes into the standard HOL4 toolset;
- the development of a bespoke execution tool for elliptic curve operations; and
- an implementation of a tool for primality-checking-by-proof.

The principal goal of the project is for the formalization to 'get close to the mathematics', and in the case of elliptic curve theory it is possible for higher order logic to get very close indeed to the textbook mathematics. In contrast to the more common approach of coding some mathematics operations as programs and then justifying their correctness, the approach described in this report is to formalize the mathematics directly and then implement proof tools to execute the definitions.

Developing tailor-made proof tools to help automate the mechanization of new theories was an important part of this project, and the LCF style design of the HOL4 theorem prover made it straightforward to extend the system proof tools. There has been much work by the HOL4 developers on supporting the use of the theorem prover as a verified execution environment, and this greatly simplified the implementation of the verified execution proof tool. It also simplified the implementation of the other proof tools, which all use verified execution internally to some degree. This is most apparent in the primality prover, which is implemented as a verified higher order logic function that is executed on potential primes.

In addition to its intrinsic interest, this project is also a component of the larger project *Formal synthesis and verification of ARM software with applications to cryptography.* The most important contribution of this project is a 'gold standard' set of elliptic curve operations mechanized with HOL4. The stage is now set for a verified path from these mathematical definitions of the elliptic curve operations right down to an implementation in ARM machine code. In addition, the tool support described in this report is now available to perform verified execution of the gold standard definitions, which can provide test vectors for prototypes before a full proof of correctness is attempted.

4.1 Future Work

The immediate next step, already underway, is to complete the mechanization of the group law for elliptic curves. Two parallel approaches are being investigated for this: applying mathematical ideas to break the proof into more tractable subproofs that can be handled with the existing proof tools; and improving the HOL4 proof tools (particularly the simplifier) to deal with the large formulas that emerge in a naive proof. It is likely that a combination of these two approaches will lead to a mechanized proof with the least effort. One idea for improving the proof tools to handle large formulas is to link HOL4 with a computer algebra system (a similar link-up of theorem prover and computer algebra system was tried in Analytica (Bauer et al., 1998), but using Mathematica as the starting point).

At the moment the formalized definitions of elliptic curve operations are highly abstract, and need sophisticated proof tools to execute them even inefficiently. The compiler being developed for the next step of the verified path to ARM machine code requires a restricted class of higher order logic functions as input, and so the elliptic curve operations will have to be ported to this language and proved equivalent to the mathematical definitions. A concrete representation for elliptic curve points will need to be chosen: an inevitable trade-off between the complexity of the equivalence proof and the efficiency of the final implementation in ARM machine code.

Now that there is a mechanized 'gold standard' for elliptic curve operations, other implementations of elliptic curve cryptography can be formally verified. For example, one idea is to start with a μCryptol program implementing a cryptographic operation based on an elliptic curve group, and make a shallow embedding of the program in higher order logic. A mechanized proof of the group law for elliptic curves reduces the functional correctness of the embedded μCryptol program to a proof that it correctly implements the elliptic curve operations. The end result is a μCryptol program formally verified to be functionally correct because it implements operations that happen to satisfy a group law, and moreover the group is an elliptic curve group. This latter point is where the 'gold standard' formalization plays a critical role in guaranteeing the security of the μCryptol program.

Acknowledgements

This work was funded as part of the project *Formal synthesis and verification of ARM software with applications to cryptography*, held jointly by the University of Utah, University of Cambridge and University of Oxford. The presentation of this work improved after giving short lecture courses on this material at the University of Cambridge and University of Utah, and a talk at Galois Connections, Inc. Mike Gordon provided useful comments on an earlier draft of this report.

Appendix A

Notation

To help read the mathematics and the HOL4 theorems in this report, the following table gives a description of some standard logical and mathematical symbols, together with the way the symbol is printed in the HOL4 theorem prover. A symbol with no HOL4 entry means that it does not occur in this report.

Mathematics	**HOL4**	**Description**
$\vdash$	`|-`	Syntactic entailment.
$=$	`=`	Equality.
$\iff$	`=`	If and only if.
$\top$	`T`	True.
$\bot$	`F`	False.
$\mathbb{B}$ or 2	bool	The booleans $\{\top, \bot\}$.
$\wedge$	`/\`	Conjunction.
$\vee$	`\/`	Disjunction.
$\implies$	`==>`	Implication.
$\neg$	`~`	Logical negation.
$-$	`~`	Numerical negation.
$\forall$	`!`	Universal quantification.
$\exists$	`?`	Existential quantification.
λ	`\`	Lambda abstraction.
$\in$	`IN`	Set membership.
$\forall x \in S.\ M$	`!x :: S. M`	Restricted universal quantification.
$\exists x \in S.\ M$	`?x :: S. M`	Restricted existential quantification.
$\emptyset$	`{}`	Empty set.
$\cup$	`UNION`	Set union.
$\cap$	`INTER`	Set intersection.

Mathematics	**HOL4**	**Description**
$\{f(x) \mid P(x)\}$	`{ f x	P x }`	Set comprehension.
$\sharp S$	`CARD S`	The cardinality of the set S.
$\sharp S < \infty$	`FINITE S`	S is a finite set.
$\mathbb{N}$	num	The natural numbers $\{0, 1, 2, \ldots\}$.
suc	`SUC`	The successor function.
$\mathbb{Z}$		The integers.
$\mathbb{Q}$		The rational numbers.
$\mathbb{R}$	real	The real numbers.
$\mathbb{R}^{\neq 0}$		The nonzero real numbers.
(a, b)		The real numbers $\{x \mid a < x < b\}$.
$[a, b]$		The real numbers $\{x \mid a \leq x \leq b\}$.
$\mathbb{C}$		The complex numbers.
τ^*	τ list	Lists with elements of type τ.
$[a_1; \cdots ; a_n]$	`[a1; ...; an]`	List syntax.
$[\,]$	`[]`	The empty list.
length	`LENGTH`	The list length function.
$\mathsf{el}\ n\ l$	`EL n l`	Picks element n from the list l.
K		An arbitrary field.
K^*	`field_nonzero`	The nonzero elements of K.
$E(K)$		An arbitrary elliptic curve over K.
$\mathcal{O}$	`curve_zero e`	The point at infinity.
α, β, $\ldots$	`'a, 'b, ...`	Type variables.

Appendix B

Elliptic Curves and Cryptography

B.1 Elliptic Curves

This appendix provides a geometric introduction to elliptic curves over the complex numbers, with the aim of building up some useful intuition about them.

B.1.1 Points

An elliptic curve E over the complex numbers is the set of points $(x, y) \in \mathbb{C} \times \mathbb{C}$ satisfying the equation

$$E : y^2 = x^3 + ax + b \ ,$$

plus a special point $\mathcal{O}$ that lies 'infinitely far up the y-axis'. (Note that both x and y are complex numbers: they are not the real and imaginary components of a single complex number.)

Some example elliptic curves over the real numbers (i.e., where both x and y are real numbers) are shown in Figure B.1. The first example curve $y^2 = x^3 - x$ is disconnected because its equation can be written as $y = \pm\sqrt{x(x+1)(x-1)}$, and the square root only exists in the intervals $[-1, 0]$ and $[1, \infty)$.

An important property of an elliptic curve is the subset of rational points: points on the curve where both the x and y coordinates are rational numbers (plus $\mathcal{O}$).[1] (This means that the real parts of both x and y are rational numbers, and the imaginary parts are zero.) Elliptic curves are interesting

[1]This only makes sense if both $a, b \in \mathbb{Q}$ in the elliptic curve equation.

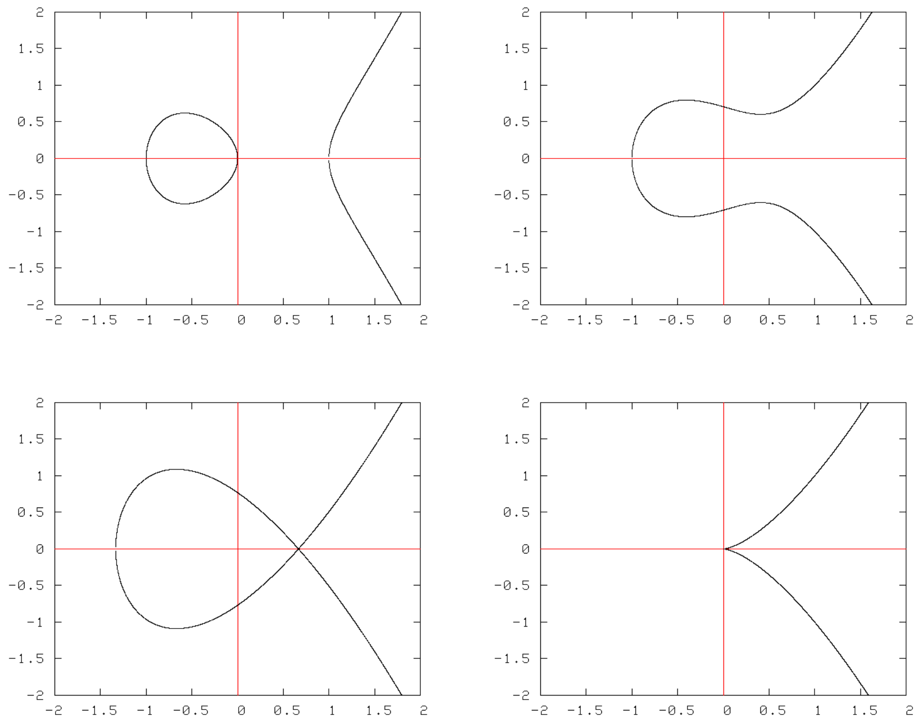

Figure B.1: Example elliptic curves, clockwise from top left: $y^2 = x^3 - x$; $y^2 = x^3 - \frac{1}{2}x + \frac{1}{2}$; $y^2 = x^3$; and $y^2 = x^3 - \frac{4}{3}x + \frac{16}{27}$.

in number theory because many questions about integers can be recast as questions about rational points on an elliptic curve. For example, Wiles' proof of Fermat's Last Theorem relied on Frey's epsilon conjecture (proved by Ribet in 1986) that every counterexample $a^n + b^n = c^n$ would yield an elliptic curve

$$y^2 = x(x - a^n)(x + b^n)$$

contradicting the Taniyama-Shimura conjecture.[2] In 1995 Wiles and Taylor proved that the Taniyama-Shimura conjecture was true for semi-stable elliptic curves, which was enough to prove Fermat's Last Theorem.[3]

B.1.2 Etymology

As may be readily observed, elliptic curves are not ellipses! The following explanation of their name comes from Blake et al. (1999).

> Just as the arc lengths on a circle give rise to the trigonometric functions sin, cos and tan, a similar study for ellipses leads one to consider *elliptic integrals.* These are integrals of the form
>
> $$\int \frac{dx}{\sqrt{4x^3 - g_2 x - g_3}} \ .$$
>
> Such integrals are multi-valued on the complex numbers and are only well defined modulo a period lattice. [...] The 'inverse' function of an elliptic integral is a doubly periodic function called an *elliptic function.* [...]
>
> It turns out that every doubly periodic function $\wp$ with periods that are independent over $\mathbb{R}$ satisfies an equation of the form
>
> $$\wp'^2 = 4\wp^3 - g_2\wp - g_3$$
>
> [...] If we consider the pair $(\wp', \wp)$ as being a point in space, then the solutions to [the above equation] provide a mapping from a torus (as $\wp$ is doubly periodic) to the curve
>
> $$Y^2 = 4X^3 - g_2 X - g_3 \ .$$
>
> This is an example of an elliptic curve (the 4 in front of the X^3 term is traditional in analytic circles—it can clearly be scaled away).

[2] The Taniyama-Shimura conjecture connects an elliptic curve to its corresponding modular form, by asserting that "all elliptic curves over $\mathbb{Q}$ are modular". This means that the sequence of numbers $n_p - p$ obtained by reducing the elliptic curve modulo all primes p is the same as the sequence of numbers obtained by applying a Fourier transform to the modular form.

[3] By 1999 a team of mathematicians, including Taylor, had proved that the Taniyama-Shimura conjecture held for every elliptic curve.

B.1.3 Elliptic curve arithmetic

Given three points P, Q and R lying on the curve E, define a ternary relation $\mathcal{R}$ as

$$\mathcal{R}(P, Q, R) \iff P, Q, R \text{ lie on a straight line.}$$

This relation behaves like a binary function from its first two arguments to its third argument, as the following lemma proves.

Lemma 1 *Given two distinct points P and Q lying on the curve E, there is a unique point R that lies both on E and on the straight line between P and Q.*

Proof: Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$. If $x_1 = x_2$ then R is uniquely $\mathcal{O}$, else if $x_1 \neq x_2$ then the straight line through P and Q is defined by the equation

$$L : y = \left(\frac{y_2 - y_1}{x_2 - x_1}\right) x + \left[y_1 - x_1 \left(\frac{y_2 - y_1}{x_2 - x_1}\right)\right]$$

Substituting the equation L into E for the variable y leaves a cubic equation in x. By the fundamental theorem of algebra, this cubic equation has three roots (counting multiplicity): two of them are known to be x_1 and x_2; the third one is the x-coordinate of the point R. Equation L gives the value for the y-coordinate of R, which has been shown to be the unique point lying on both E and L. $\square$

Using the previous lemma, an addition operation on elliptic curve points can be defined as

$$P + Q = -R$$

where R is found by drawing a chord through P and Q, and $-R$ is the point R reflected in the x-axis (and $-\mathcal{O} = \mathcal{O}$). See Figure B.2 for an illustration of adding two points together using a chord.

The only case not covered by the above is adding a point P to itself, because two distinct points are needed to draw a chord between them. When there is just one point P, a tangent to the curve E through P is drawn instead.[4] Similarly to the chord case, the tangent line will intersect the curve E at precisely one point R,[5] and this is reflected in the x-axis to give the result

$$2P = -R \, .$$

[4]Intuitively, the tangent method is the limit of the chord process of adding P to Q, where Q is moved closer and closer to P.

[5]In the proof the resulting cubic has a double root at x_1; the other root x_3 is the x-coordinate of R.

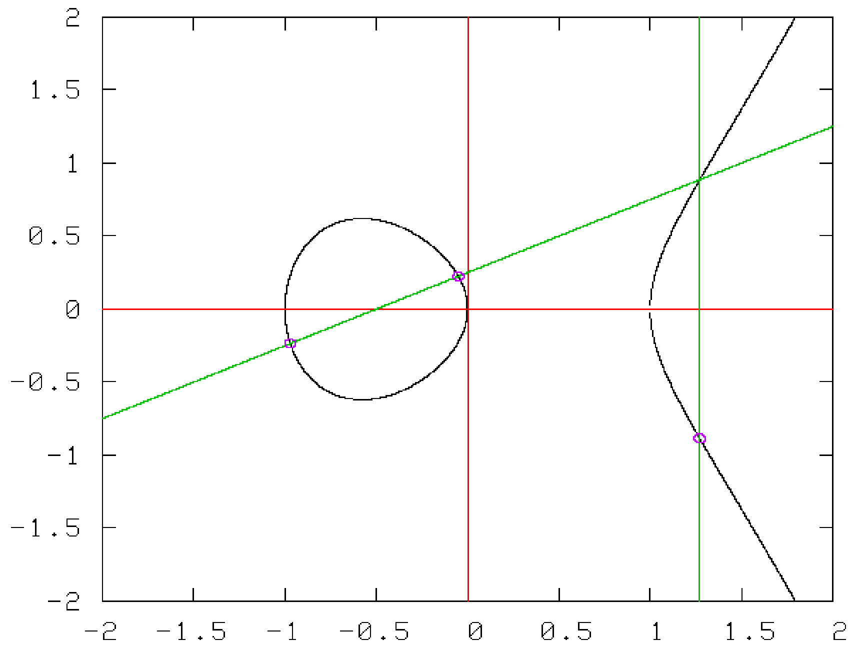

Figure B.2: Adding two points on the elliptic curve $y^2 = x^3 - x$ using a chord.

See Figure B.3 for an illustration of adding two points together using a tangent. This geometric definition of the addition operation is called the *tangent-chord method.*

Theorem 2 *Elliptic curve points form an Abelian group, with identity element $\mathcal{O}$, inverse function $-$, and group operation $+$.*
Proof: Omitted. The interested reader is referred to Silverman (1986), which provides a rigorous introduction to elliptic curves. $\square$

If K is the finite field $\mathrm{GF}(q)$, then the number $\sharp E(K)$ of K-rational (affine) points is trivially bounded by

$$\sharp E(K) \leq q^2 + 1$$

and so the elliptic curve group is a finite Abelian group.

In fact, $\sharp E(K)$ is usually around $q + 1$, although it is not a simple matter to determine it precisely. Define the trace t of E as

$$t = q + 1 - \sharp E(K) \ .$$

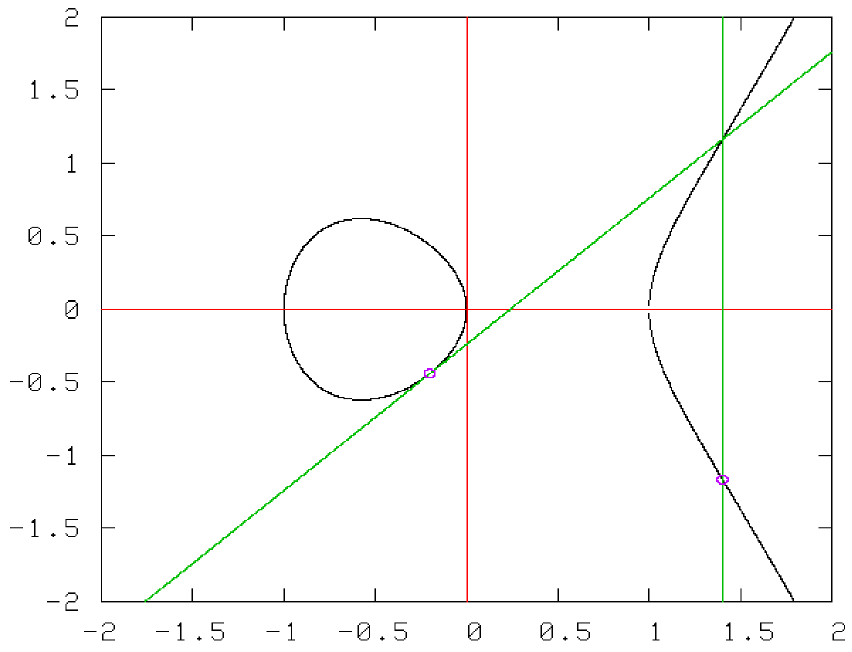

Figure B.3: Doubling a point on the elliptic curve $y^2 = x^3 - x$ using a tangent.

Theorem 3 (Hasse) *The trace of an elliptic curve satisfies* $|t| \leq 2\sqrt{q}$ *, and every value permitted by this inequality occurs.*
Proof: Omitted. $\square$

B.2 Elliptic Curve Cryptography

Given their utility in advanced number theory, it should come as no surprise that elliptic curves also feature in many aspects of cryptography.

B.2.1 Crytography Based on Groups

Many cryptographic primitives make use of a form of the discrete logarithm problem based on a group G, in which a potential attacker is given $g, h \in G$ and must find a k such that $g^k = h$. Clearly, the difficulty of this problem depends on the group G. For some groups, such as integer addition modulo n, the problem is easy. For some groups, such as the multiplicative group $\mathrm{GF}(p)^*$ of the finite field $\mathrm{GF}(p)$, the problem is difficult. However, even in this case, the number field sieve provides a sub-exponential algorithm to solve the problem.

In 1985, Neal Koblitz and Victor Miller independently proposed basing the discrete logarithm problem on an elliptic curve group $E(\mathrm{GF}(q))$ over a finite field $\mathrm{GF}(q)$. There are no known sub-exponential algorithms to solve this version of the discrete logarithm problem. Taking into account the best known algorithms, Blake et al. (1999) present a correspondence between the key sizes of equal security discrete logarithm problems based on multiplicative and elliptic curve groups:

$\mathrm{GF}(p)^*$	$E(\mathrm{GF}(q))$
1024 bits	173 bits
4096 bits	313 bits

As can be seen, elliptic curve groups require shorter keys than multiplicative groups, which make them an attractive choice in security applications with constraints on bandwidth or computation power (e.g., smart cards).

It should be noted that an argument advanced against using elliptic curve groups is that the conventional discrete logarithm problem based on the multiplicative group has been more intensively studied. It may indeed be the case that a sub-exponential algorithm for the discrete logarithm problem based on elliptic curve groups is waiting to be discoverd.

B.2.2 ElGamal Encryption

ElGamal encryption demonstrates how the discrete logarithm problem based on a group G can be used as a public key encryption algorithm. Bob generates an instance $g^x = h$ of the discrete logarithm problem to create a new public and private key. Bob publishes the public key (g, h) and keeps the private key x secret. The following algorithm allows Alice to send a message $m \in G$ to Bob that cannot be read by a third party (this security property is called *confidentiality*).

1. Alice obtains a copy of Bob's public key (g, h).

2. Alice generates a randomly chosen natural number $k \in \{1, \ldots, \sharp G - 1\}$ and computes $a = g^k$ and $b = h^k m$.

3. Alice sends the encrypted message (a, b) to Bob.

4. Bob receives the encrypted message (a, b). To recover the message m he computes
$$ba^{-x} = h^k m g^{-kx} = g^{xk-xk} m = m \ .$$

B.2.3 Digital Signature Algorithm

The digital signature algorithm is a N.I.S.T. standard (FIPS, 1994), and the elliptic curve version is part of the *Suite B* set of cryptographic algorithms recommended by NSA. Using the same public and private key as for ElGamal encryption,[6] Bob can use the digital signature algorithm to digitally sign a message $m \in G$ to Alice, ensuring: Alice knows that only Bob could have sent the message (*authenticity*); Alice knows that the message has not been tampered with in transit (*integrity*); and it is impossible for Bob to turn around later and say he did not send the message (*non-repudiation*). The following algorithm implements the digital signature algorithm, and requires an additional bijective function

$$f : G \to \{0, \ldots, \sharp G - 1\}$$

to be publically known.

1. Bob generates a random natural number $k \in \{1, \ldots, \sharp G - 1\}$, and computes $a = g^k$.

2. Bob also computes the solution b to the congruence

$$f(m) = -xf(a) + kb \pmod{\sharp G}$$

3. Bob sends the message m and the signature (a, b) to Alice.

4. Alice receives the message m and the signature (a, b), and obtains a copy of Bob's public key (g, h).

5. Alice computes

$$\begin{aligned} u &= f(m)b^{-1} \pmod{\sharp G} , \\ v &= f(a)b^{-1} \pmod{\sharp G} , \\ w &= g^u h^v . \end{aligned}$$

6. Alice verifies that the signature matches the message by checking that

$$\begin{aligned} w &= g^u h^v = g^{f(m)b^{-1}} g^{vx} = g^{f(m)b^{-1} + xf(a)b^{-1}} \\ &= g^{(f(m)+xf(a))b^{-1}} = g^{kbb^{-1}} = g^k = a . \end{aligned}$$

[6]It is considered bad cryptographic hygiene to use precisely the same key for signature and encryption, so in practice Bob will probably generate a new key to create digital signatures.

B.2.4 Elliptic Curve Factorization Algorithm

Although the main use of elliptic curves in cryptography is in providing an alternative group for the discrete logarithm problem, it would be remiss to ignore their cryptanalytical use in the form of the elliptic curve factorization algorithm (Lenstra, 1987). It is no longer the best known algorithm for factoring a huge RSA modulus into the product of two primes: the more recent general number field sieve is currently the best for this problem. However, the elliptic curve factorization method is still the fastest for factoring out divisors less than 20 digits long, because its running time depends on the size of the factor rather than the size of the number to be factored.

It is an extension of Pollard's $p-1$ method, which operates as follows on a number n which is the product of two primes p and q. A base a is chosen at random, and raised to a power B modulo n.[7] If

$$p-1 \mid B \quad \text{and} \quad q-1 \nmid B$$

then

$$a^B = 1 \pmod{p} \quad \text{and} \quad a^B \neq 1 \pmod{q} \ .$$

Therefore

$$\gcd(a^B \bmod n - 1, n) = q \ .$$

Pollard's $p-1$ method is effective when $p-1$ is a smooth number: all its prime factors are small. Unfortunately, precisely to prevent this factorization algorithm from working (and another similar one using $p+1$), primes p and q to be used in an RSA modulus are chosen to be strong primes, where p is a strong prime if both $p \pm 1$ are non-smooth.

Lenstra's idea was to replace the multiplicative group $\mathrm{GF}(p)^*$ used in Pollard's $p-1$ method with a random elliptic curve group $E(\mathrm{GF}(p))$. Since the size $\sharp E(\mathrm{GF}(p))$ of the group is a random number in the range $p \pm 2\sqrt{p}$, then only one of these numbers needs to be smooth for the attack to succeed. For the full details of the elliptic curve factorization algorithm see Blake et al. (1999) or Koblitz (1987).

Using the same ideas, elliptic curves are also used for primality proving. In the key generation phase of an encryption scheme such as RSA, it is useful to check that the probable primes found by a fast algorithm such as Miller-Rabin are bone fide primes. An elliptic curve primality prover (such as the Q function in Mathematica) can routinely prove the primality of numbers with thousands of digits.

[7] B is chosen to be a number with many small prime factors, such as $k!$.

Appendix C

Higher Order Logic and the HOL4 Theorem Prover

HOL4[1] is an interactive theorem prover implementing higher order logic. It has an LCF style design, which means that there is a small logical kernel that is solely empowered to create theorems using the primitive inference rules of higher order logic, and the ML programming language is provided to implement inference rules in terms of the primitives.

This appendix contains a brief overview of higher order logic and its implementation in the HOL4 theorem prover, containing no more than is necessary for a reader wishing to understand the basis on which the formalization of elliptic curve cryptography is built. A detailed introduction to the design of the HOL4 theorem prover can be found in the book *Introduction to HOL (A theorem proving environment for higher order logic)* by Gordon and Melham (1993), and information on the current version of the HOL4 theorem prover is provided in the system documentation (Slind and Norrish, 2001).

C.1 Terms and Types

Terms in higher order logic are typed, and the syntax $t : \tau$ means that the term t has type τ. It is helpful to identify types with the set of their elements, so for example the type of booleans is $\mathrm{bool} = \{\top, \bot\}$ and the type of natural numbers is $\mathbb{N} = \{0, 1, 2, \ldots\}$. Compound types may be created by using type operators. An important example is the function space type operator $\cdot \to \cdot$, where for example the type $\mathbb{N} \to \mathrm{bool}$ is the set of functions mapping natural numbers to either $\top$ or $\bot$. Another example is the list type operator $\cdot\,\mathrm{list}$,

[1]HOL4 can be downloaded from `http://hol.sf.net`.

where for example the type of boolean lists is

$$\mathrm{bool\ list} = \{[\,],\ [\top],\ [\bot],\ [\top,\top],\ \ldots\}\ .$$

Finally, there is an infinite set $\{\alpha, \beta, \ldots\}$ of type variables, which may instantiate to any higher order logic type. Type variables can occur at any position in a compound type, for example the type α list of lists containing elements of type α. This polymorphism mechanism means that list theorems need be proved only once for the type α list, and thereafter the type variable α can be instantiated on demand to give the same theorems for the types $\mathbb{N}$ list, $\mathbb{N} \to$ bool, β list list etc.

The terms of higher order logic are terms of the simply typed λ-calculus (Church, 1940), where the syntax $\lambda x.\, t[x]$ should be read as 'the function that takes an x and returns $t[x]$.' The λ binds x, so for example the application of the function term $\lambda x.\, \lambda x.\, x$ to the argument term y yields the result $\lambda x.\, x$ (and *not* $\lambda x.\, y$). The restriction to *typed* terms of λ-calculus means that every term must be well-typed (or inconsistencies emerge in the form of the Russell paradox), where the type relation is calculated as follows:

Variables: All variables $v : \tau$ are well-typed.

Constants: Constants are given a type when they are defined, for example $\top$: bool, $[\,] : \alpha$ list, $= : \alpha \to \alpha \to$ bool and $\mathsf{prime} : \mathbb{N} \to$ bool. Any type variables that they contain may be specialized when they are used in terms, so for example the term $[\,] : \mathbb{N}$ list is well-typed.

Function applications: A function term $f : \alpha \to \beta$ is well-typed when applied to any argument term $x : \alpha$, and the result is $(f\ x) : \beta$.

λ-abstractions: Given a variable $v : \alpha$ and a term $t[v] : \beta$, the λ-abstraction $(\lambda\, v.\ t[v]) : \alpha \to \beta$ is well-typed.

Using a type-inference algorithm of Milner (1978), it is possible to take a term t that does not contain any type information at all and deduce a most general type for the term (or raise an error if it is not well-typed). Therefore, a well-typed term $t : \tau$ of higher order logic can usually be written as t without any ambiguity.

C.2 Theorems

Theorems in higher order logic are *sequents* $\Gamma \vdash t$ where the term t is the *conclusion* of the theorem, and the set of terms Γ are the *hypotheses* of the theorem. The HOL4 theorem prover has an LCF design, which means that theorems are only created in a small *logical kernel* which faithfully executes the primitive rules of inference of higher order logic. For example, the following four primitive rules of inference assert that the equality relation is both reflexive (REFL) and transitive (TRANS), that type variables in a theorem may be instantiated to any higher order logic type ($\mathsf{INST\_TYPE}$), and that the usual λ-calculus β-conversion is valid in higher order logic ($\mathsf{BETA\_CONV}$):

$$\frac{}{\vdash t = t}\mathsf{REFL} \qquad \frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\mathsf{TRANS}$$

$$\frac{\Gamma[\alpha] \vdash t[\alpha]}{\Gamma[\sigma] \vdash t[\sigma]}\mathsf{INST\_TYPE} \qquad \frac{}{\vdash (\lambda x.\ t[x])\ u = t[u]}\mathsf{BETA\_CONV}$$

These primitive rules of inference appear in the ML signature of the logical kernel as follows:

$$\begin{array}{rcl}
\mathsf{REFL} & : & \mathsf{term} \rightarrow \mathsf{thm} \\
\mathsf{TRANS} & : & \mathsf{thm} \rightarrow \mathsf{thm} \rightarrow \mathsf{thm} \\
\mathsf{INST\_TYPE} & : & \mathsf{hol\_type} \times \mathsf{hol\_type} \rightarrow \mathsf{thm} \rightarrow \mathsf{thm} \\
\mathsf{BETA\_CONV} & : & \mathsf{term} \rightarrow \mathsf{thm}
\end{array}$$

Since the ML type thm is abstract, the type security of ML ensures that the functions in the logical kernel represent the only way that theorems may be created.

A principle of definition implemented as a function in the logical kernel allows new constants to be defined. Given a theorem $\vdash v : \tau = M$, it makes a new constant $c : \tau$ and returns the characterizing theorem $\vdash c = M$.[2] The logical kernel also has a primitive definition principle for new types: the user supplies a predicate over an existing type, and the theorem prover returns a new type in which the elements are in one-to-one correspondence with the user's predicate.[3]

[2]For this to be valid, we must insist that M contains no free variables and no type variables other than those in τ.

[3]In higher order logic there are no empty types, so for this type definition principle to be valid the user must supply a theorem that the predicate is not empty.

Bibliography

Bruno Barras. Programming and computing in HOL. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference: TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 17–37. Springer, August 2000.

Andrej Bauer, Edmund M. Clarke, and Xudong Zhao. Analytica - an experiment in combining theorem proving and symbolic computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998. URL `http://reports-archive.adm.cs.cmu.edu/anon/1992/CMUCS92147.ps`.

Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*, volume 265 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1999.

Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

FIPS. *Digital Signature Standard*. Federal Information Processing Standards Publication 186. U.S. Department of Commerce/N.I.S.T. National Technical Information Service, May 1994. URL `http://www.itl.nist.gov/fipspubs/fip186.htm`.

M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.

Mike Gordon, Juliano Iyoda, Scott Owens, and Konrad Slind. A proof-producing hardware compiler for a subset of higher order logic. In Hurd et al. (2005), pages 59–75. URL `http://www.cl.cam.ac.uk/~mjcg/proposals/dev/`.

John Harrison. Formalized mathematics. Technical Report 36, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-

20520 Turku, Finland, 1996. URL `http://www.cl.cam.ac.uk/users/jrh/papers/form-math3.html`.

John Harrison. A HOL theory of Euclidean space. In Joe Hurd and Tom Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129. Springer, August 2005. URL `http://www.cl.cam.ac.uk/users/jrh/papers/hol05.html`.

Joe Hurd. Predicate subtyping with predicate sets. In Richard J. Boulton and Paul B. Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 265–280. Springer, September 2001. URL `http://www.gilith.com/research/papers`.

Joe Hurd, Edward Smith, and Ashish Darbari. Theorem proving in higher order logics: Emerging trends proceedings. Technical Report PRG-RR-05-02, Oxford University Computing Laboratory, August 2005. URL `http://www.gilith.com/research/papers`.

Neal Koblitz. *A Course in Number Theory and Cryptography.* Number 114 in Graduate Texts in Mathematics. Springer, 1987.

H. W. Lenstra. Factoring integers with elliptic curves. *Ann. Math.*, 126: 649–673, 1987.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

Bruce Schneier. *Applied Cryptography.* Wiley, second edition, 1996.

J. H. Silverman. *The Arithmetic of Elliptic Curves.* Number 106 in Graduate Texts in Mathematics. Springer, 1986.

Konrad Slind and Michael Norrish. *The HOL System Tutorial*, February 2001. URL `http://hol.sf.net/`. Part of the documentation included with the HOL4 theorem-prover.

Junxing Zhang and Konrad Slind. Verification of Euclid's algorithm for finding multiplicative inverses. In Hurd et al. (2005), pages 205–220. URL `http://www.gilith.com/research/papers`.