`ec`: A Modular Elliptic Curve Library for Haskell

Nikhil Desai and Brandon Azad

June 11, 2014

1 Introduction

`ec` is a library for computing over elliptic curve groups in Haskell with a focus on modularity and extensibility. `ec` is designed to provide cryptographic primitives for other, higher-level APIs. Cryptographic protocols often require the use of an algebraic group over which the discrete logarithm problem is hard, but that may need to satisfy additional properties (such as being pairing-friendly). `ec` is meant to provide a convenient abstraction for defining such groups.

2 Motivation

Cryptography is important for a wide variety of applications, and elliptic curve cryptography is particularly noteworthy for the difficulty of its discrete logarithm problem. As compared to RSA, elliptic curve groups need to be of significantly smaller order to provide the same level of security. This and other properties have led to elliptic curve cryptography gaining popularity. More and more cryptographic protocols require that a group have additional structure beyond a difficult discrete log. For instance, identity-based encryption (IBE) schemes require two groups $\mathbb{G}$ and $\mathbb{G}_T$ of prime order p that support a bilinear pairing operation $e\colon \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ such that $e(g^\alpha, h^\beta) = e(g, h)^{\alpha\beta}$ for all $g, h \in \mathbb{G}$.

However, existing cryptographic libraries for Haskell cannot be extended by the client to use custom groups with additional structure. For example, Marcel Fourne's `hecc` only works over prime or binary curves, such as those defined in the NIST standard. It cannot be extended to define new groups or new, possibly more efficient ways of computing sums in the predefined groups. Another library, Vincent Hanquez's `crypto-pubkey`, only supports computations on standard curves using the inefficient affine coordinate system to represent points on the elliptic curve.

To remedy this limitation in existing libraries, we designed `ec` to be extensible; we want users of the library to be able to provide new instances of elliptic curves with additional structure and still use the same API for group operations. We also wanted to capture the internal representation of the group at the type level, so that different algorithms can be used to compute the group operation by simply changing the type signature of the computation. The final design goal was for it to be relatively easy for the client to define performant implementations of elliptic curves, for example by relying on Haskell's foreign function interface.

3 Background

Elliptic curves are algebraic curves defined over an underlying field $\mathbb{F}$. The set of points $(x, y) \in \mathbb{F}^2$ on the elliptic curve E over the field $\mathbb{F}$, together with a special "point at infinity" $\mathcal{O}$, is written $E(\mathbb{F})$. If $\mathbb{F}$ is a field of characteristic other than 2 or 3, then any elliptic curve E over $\mathbb{F}$ can be written in reduced Weierstrass form as

$$y^2 = x^3 + ax + b, \qquad \text{where } a, b \in \mathbb{F}.$$

Fields of characteristic 3 require the more general form

$$y^2 = x^3 + ax^2 + bx + c, \qquad \text{where } a, b, c \in \mathbb{F},$$

and fields of characteristic 2 require the even more general form

$$y^2 + ay = x^3 + bx^2 + cxy + dx + e, \qquad \text{where } a, b, c, d, e \in \mathbb{F}.$$

A natural group law emerges for the set $E(\mathbb{F})$ under which $E(\mathbb{F})$ is abelian (usually written additively) and the point $\mathcal{O}$ is taken to be the zero element. This group law can be interpreted geometrically as the "chord-and-tangent" method of point addition. In affine (x, y) coordinates, the sum of two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ over a (short form) Weierstrass curve $y^2 = x^3 + ax + b$ is defined as $R = (x_R, y_R)$, where

$$\lambda = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \\ \dfrac{3x_P^2 + a}{2y_P^2} & \text{otherwise} \end{cases}$$
$$x_R = \lambda^2 - x_P - x_Q$$
$$y_R = \lambda(x_R - x_P) + y_P.$$

Note that the exact computation differs depending on whether the operation is a doubling or the addition of distinct points.

Most implementations of elliptic curve cryptography tend to avoid using the affine coordinate system since the computation of λ above requires inverting an element of $\mathbb{F}$, which is usually quite expensive. Instead, affine points are converted to alternate coordinate systems in which the addition law does not require field inversions. Once extended computations in $E(\mathbb{F})$ are complete, the points are converted back into affine coordinates. For example, one common coordinate system for elliptic curves in Weierstrass form is Jacobian coordinates. A point $P = (x, y)$ is represented in Jacobian coordinates by the triple (X, Y, Z), where $x = X/Z^2$ and $y = Y/Z^3$; any (X, Y, Z) with $Z = 0$ represents the point $\mathcal{O}$. Note that many different triples (X, Y, Z) may represent the same point $P \in E(\mathbb{F})$.

4 Use

We see two primary use cases for `ec`: computing point multiplications using existing curves and defining new curves on which to compute.

4.1 Computing on Curves

We wanted `ec` to expose the same basic interface for all point operations regardless of the internal structure of the group and the coordinate system used. Since elliptic curves are a group, the two most important operations for computing are `add` and `multiply`.

`add :: EC c f -> p c f -> p c f -> p c f` takes a definition of an elliptic curve E and two points P and Q on the curve and returns the sum $P + Q \in E(\mathbb{F})$. In the type signature, `f` is the type of the field over which the elliptic curve is defined, `c` is the type of the elliptic curve (that is, the exact representation of the curve), and `p` is the type of the coordinate system used for point computations. The `EC` type simply stores the elliptic curve parameters and auxiliary information about the field.

`multiply :: EC c f -> Int -> Integer -> p c f -> p c f` efficiently computes an integer multiple d of a point P on the curve, that is, P added to itself d times. Like `add`, the first parameter is the elliptic curve definition. The third parameter is the multiplier d and the fourth parameter is the point P. The second parameter specifies the effective bit length n of d, that is, how many bits of d should be examined when computing $d \cdot P$. The parameter n is included in order to support branchless implementations that are resistant to timing attacks. Since most multiplication routines loop over the bits of d, providing an effective bit length can prevent the multiplication routine from terminating early once the most significant bit of d has been processed.

Computing multiples of points on a curve is fairly straightforward. For example, the following code computes the product of the base point of the NIST standard elliptic curve P-521 with the number 42 using Jacobian coordinates.

```
import Crypto.EllipticCurve
import Crypto.EllipticCurve.StandardCurves

params :: ECP Weierstrass Jacobian P521
params = nistP521

p = multiply (ecpCurve params) (ecpBitLength params) 42 (ecpGenerator params)
```

The `ECP` type stores elliptic curve parameters for standard curves, including the curve definition, the bit length of the subgroup order, and a generator for the subgroup. The `Weierstrass` type represents elliptic curves in short Weierstrass form and the `Jacobian` type represents a point on an elliptic curve in Jacobian coordinates. Performing the computation in another coordinate system is as simple as changing the `Jacobian` type to the desired coordinate system.

4.2 Defining New Curves

`ec` has been designed to minimize overhead when creating new elliptic curves. For example, this is a complete definition of the Jacobian coordinate system over (short) Weierstrass curves:

```
data Jacobian (c :: * -> *) f = Jacobian
  { jacobianX, jacobianY, jacobianZ :: !f }

instance EllipticCurvePoint Weierstrass Jacobian where
```

```
  toAffine (EC _ FieldOperations {..}) (Jacobian x y z)
    | z == zero = AffinePointAtInfinity
    | otherwise =
      let iz3 = inv (z ^ 3)
      in Affine (x * iz3 * z) (y * iz3)

  fromAffine _ (Affine x y)          = Jacobian x   y   one
  fromAffine _ AffinePointAtInfinity = Jacobian one one zero
```

Point representations are parameterized by the curve type and the underlying field.[1] The `EllipticCurvePoint` instance allows `Jacobian` to be used as a point type in an `EllipticCurve` instance, which is where the group law can be defined.

An `EllipticCurve` instance ties together a curve representation, for example `Weierstrass`, with a computational coordinate system, for example `Jacobian`. Clients only need to implement correct point addition and point doubling formulas in order to use the `multiply` function with a given curve and coordinate system. The default implementation of `multiply` uses the Montgomery ladder exponentiation algorithm, which is often used for timing attack resistance. (However, the `ec` library is *not* resistant to timing attacks; see Section 6.)

In order to make defining computations in the field as easy as possible, the `EC` type contains a `FieldOperations` record that contains functions for each operation in the field. For example, the addition function is named `+`, the subtraction function is named `-`, and so on. This allows clients to use familiar operator names to perform computations in the field.[2] To concisely bring these operators into scope, clients can use the `RecordWildCards` extension and pattern match the `FieldOperations` member of `EC` against `FieldOperations {..}`.

The following is a (mostly complete) definition of the group law for Weierstrass curves using affine coordinates:

```
{-# LANGUAGE RecordWildCards #-}
instance EllipticCurve Weierstrass Affine where

  add c@(EC (Weierstrass a _) FieldOperations {..}) p1 p2
    | isZeroPoint c p1 = p2
    | isZeroPoint c p2 = p1
    | (Affine x1 y1) <- p1, (Affine x2 y2) <- p2 =
      let (dx, dy) = (x2 - x1, y2 - y1)
      in if dx == zero
           then if dy == zero
                  then double c p1
                  else zeroPoint c
           else let x3 = (s ^ 2) - x2 - x1
                    y3 = s * (x1 - x3) - y1
                in Affine x3 y3
```

[1] Including a phantom curve type ensures that trying to use points from one curve in computations on a different type of curve results in a compile error. Note, however, that it is still possible to mix points on two distinct curves of the same type. The kind annotation is required because elliptic curve types themselves are parameterized.

[2] One downside of this choice is that we have not yet found a way to preserve the fixity of the operators, thus requiring gratuitous use of parentheses. However, this syntax is still the cleanest of all the options we explored.

```
double c@(EC (Weierstrass a _) FieldOperations {..}) p
  | isZeroPoint c p = p
  | (Affine x1 y1) <- p =
    let s  = (3 # (x1 ^ 2) + a) / (2 # y1)
        x3 = (s ^ 2) - (2 # x1)
        y3 = s * (x1 - x3) - y1
    in Affine x3 y3

negate (EC _ FieldOperations {..}) (Affine x y) = Affine x ((.-) y)
negate _ AffinePointAtInfinity = AffinePointAtInfinity
```

5 Implementation

The three fundamental types that compose the `ec` library are the `Field`, `EllipticCurvePoint`, and `EllipticCurve` typeclasses.

The `Field` typeclass provides a basic field interface on top of which elliptic curves can be implemented. It is defined as:

```
{-# LANGUAGE TypeFamilies #-}
class (Eq f) => Field f where
  type FieldParameter f
  zero, one :: f
  add :: FieldParameter f -> f -> f -> f
  neg :: FieldParameter f -> f -> f
  {- ... -}
  pow :: FieldParameter f -> f -> Integer -> f
```

The `FieldParameter f` type in the definition of `Field` allows the field to be parameterized by data that may not be available at compile time. For example, consider the field of integers modulo a prime p. If p is known at compile time, then this field can be defined using type-level natural numbers, provided by the `GHC.TypeLits` module or Tikhon Jelvis's `Data.Modular`. However, if p must be generated at runtime, then type-level natural numbers cannot be used, since the types must be known at compile time. Thus, in order to allow clients to use fields that may depend on runtime parameterizations, each field operation takes a parameterization as its first argument. Those instances that need a parameterization can define the required type, and those instances that do not can use `()`.

However, this definition for `Field` means that instances of `Field` cannot generally be made instances of `Num`. `Field` cannot inherit from `Num` or use `Num`'s operators directly since each `Field` function takes an additional argument. `Num` provides a very convenient and familiar interface for writing computations, and we found it difficult to find a comparably clean way to write field computations.[3] Eventually, we settled on a C-like approach, where we pass

[3] The primary methods we examined were: using Template Haskell to define a new clean syntax for field computations; creating a "field monad" similar to the reader monad in which computations could be written in an imperative style; and enabling large numbers of scary-sounding GHC extensions to see what would happen. We decided against Template Haskell due to the cumbersome and unnatural syntax and the inability decompose the computation or call other Haskell functions in the computation. The field monad appeared promising at first but it quickly became apparent that efficiency could be a concern. We wanted

around a record of operator functions that have already been parameterized. These operators have the familiar names and can be brought into scope by using the `RecordWildCards` extension.

```
data FieldOperations f = FieldOperations
  { (+) :: f -> f -> f          -- field addition
  , neg :: f -> f               -- field negation
  {- ... -}
  , (^) :: f -> Integer -> f    -- field exponentiation
  }
```

The most basic data type of the `ec` library is the `EC` type, which holds a minimal complete definition of an elliptic curve. This type simply bundles a curve definition and field operations.

```
data EC c f = EC (c f) (FieldOperations f)
```

For example, `c` could be a curve type like `Weierstrass`.

The `EllipticCurvePoint` typeclass defines the functionality necessary for an object to be an element in an elliptic curve group. A point on an elliptic curve only needs to support conversion to and from affine coordinates. The most relevant portion of the definition is:

```
{-# LANGUAGE MultiParamTypeClasses #-}
class EllipticCurvePoint c p where
  toAffine   :: (Field f) => EC c f -> p c f -> Affine c f
  fromAffine :: (Field f) => EC c f -> Affine c f -> p c f
```

We chose to separate out the point definition from the curve definition (that is, the `EllipticCurvePoint` class from the `EllipticCurve` class) in order to eliminate code repetition when defining curves on affine coordinates: the `Affine` point type is an instance of `EllipticCurvePoint` for every curve.[4]

Finally, the `EllipticCurve` type defines the interface for group operations. Currently, this typeclass is oriented towards curves on which the fundamental efficient operations are point addition and doubling, but we hope to expand this interface in the future. Clients need only define addition and negation on the curve, although the default implementation of multiplication can be overridden to supply a more efficient implementation if desired.

```
{-# LANGUAGE MultiParamTypeClasses #-}
class (EllipticCurvePoint c p) => EllipticCurve c p where
  add      :: (Field f) => EC c f -> p c f -> p c f -> p c f
  double   :: (Field f) => EC c f -> p c f -> p c f
  negate   :: (Field f) => EC c f -> p c f -> p c f
  multiply :: (Field f) => EC c f -> Int -> Integer -> p c f -> p c f
  multiply c n d p = montgomery (add c) (double c) (negate c) (zeroPoint c)
                                n p d
```

to use the same operators to define computations between raw field elements (for example, `3`) and computed field elements (for example, `(4 * 5)`). This is challenging because computed field elements are actually functions from the field parameterization type to the field type, while raw field elements are of the plain field type. After a great deal of type hacking, we determined that a clean and uniform interface was either not possible or more difficult than it was worth, and so we settled on the current design.

[4] We are also considering removing the dependence on the curve in the `EllipticCurvePoint` definition, but we are still trying to determine whether any coordinate system depends on the curve definition in order to

Multiplication by default uses the Montgomery ladder exponentiation algorithm, but other exponentiation algorithms are provided in the module `Crypto.Number.Power`. (Currently, the only other algorithm is exponentiation by squaring, but we hope to add more as we continue development.) The algorithms in `Crypto.Number.Power` are designed to take advantage of groups where squaring is faster than general multiplication, such as elliptic curve groups.

The library is also designed to support various field types. For example, we are currently working on integrating the OpenSSL implementation of the field of integers modulo $2^{521} - 1$ into `ec`. (This implementation leverages C source from OpenSSL under the `cbits` directory, as well as foreign function wrappers and types for the `P521` type under the `OpenSSL` directory.) Unfortunately, limitations in the OpenSSL implementation mean that it is not in general a true field, which has slowed progress on integrating this code into the rest of the library.

6 Future Work

The `ec` library is currently unfinished, but we hope to continue development.

One serious limitation of the `ec` library is that it currently does not provide a convenient interface for curves with different fundamental operations than doubling and adding. For example, for curves designed for efficient point tripling, the default implementation of `multiply` would need to be overridden and the functionality implemented in `add` and `double` would be mostly unused. Support for curves with different fundamental operations could be achieved by defining a typeclass for each efficient set of operations; that is, there would be one typeclass for the common doubling-oriented curves, another typeclass for tripling-oriented curves, and so on, all of which inherit the interface of `EllipticCurve`.

`ec` also needs to support some additional operations on elliptic curves before it can be used for more advanced cryptographic operations. For example, certain elliptic curve primality proving algorithms need a way to count points on a randomly generated elliptic curve. However, it would be cumbersome for clients to re-implement the Schoof-Elkies-Atkin point counting algorithm for every new curve type they create. Thus, future versions of `ec` should provide a typeclass that clients can implement for custom curves to provide the basic functionality needed by Schoof-Elkies-Atkin. `ec` could also provide implementations of other common elliptic curve algorithms, such as the Weil and Tate pairings.

Another limitation of the `ec` library is that it is not resistant to timing attacks. Unfortunately, Haskell's laziness makes reasoning about timing somewhat more difficult than in imperative languages, and Haskell's native numeric functions already use data-dependent branches internally. For these reasons, timing attack resistance was not a top priority when providing `ec`'s default curve implementations. However, by using custom field implementations through Haskell's foreign function interface, it might still be possible to create branchless curve implementations that still perform well.

Finally, it would be nice to make `ec` more portable, since several parts of the library rely on low-level GHC internals.

References

https://hackage.haskell.org/package/hecc
https://hackage.haskell.org/package/crypto-pubkey
http://mathworld.wolfram.com/EllipticCurve.html
http://www.hyperelliptic.org/EFD
http://en.wikipedia.org/wiki/Schoof-Elkies-Atkin_algorithm