ESCUELA SUPERIOR POLITÉCNICA DE CHIMBORAZO

FACULTAD DE CIENCIAS

CARRERA MATEMÁTICA

IMPLEMENTACIÓN DE LA CRIPTOGRAFÍA DE LA CURVA ELÍPTICA EN EL LENGUAJE HASKELL

Trabajo de Integración Curricular

Tipo: Proyecto de Investigación

Presentado para optar al grado académico de:

MATEMÁTICO

AUTOR:

DANIEL ALEJANDRO REINOSO SALAS

Riobamba – Ecuador

2024

ESCUELA SUPERIOR POLITÉCNICA DE CHIMBORAZO

FACULTAD DE CIENCIAS

CARRERA MATEMÁTICA

IMPLEMENTACIÓN DE LA CRIPTOGRAFÍA DE LA CURVA ELÍPTICA EN EL LENGUAJE HASKELL

Trabajo de Integración Curricular

Tipo: Proyecto de Investigación

Presentado para optar al grado académico de:

MATEMÁTICO

AUTOR: DANIEL ALEJANDRO REINOSO SALAS

DIRECTOR: Dr. LEONIDAS ANTONIO CERDA ROMERO, PhD.

Riobamba – Ecuador

2024

Yo, Daniel Alejandro Reinoso Salas, declaro que el presente Trabajo de Integración Curricular es de mi autoría y los resultados del mismo son auténticos. Los textos en el documento que provienen de otras fuentes están debidamente citados y referenciados.

Como autor asumo la responsabilidad legal y académica de los contenidos de este Trabajo de Integración Curricular; el patrimonio intelectual pertenece a la Escuela Superior Politécnica de Chimborazo.

Riobamba, 20 de mayo de 2024

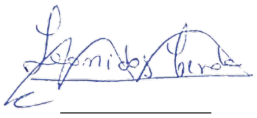
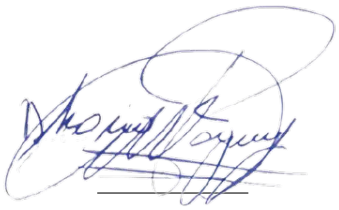
Daniel Alejandro Reinoso Salas
0604703397

ESCUELA SUPERIOR POLITÉCNICA DE CHIMBORAZO

FACULTAD DE CIENCIAS

CARRERA MATEMÁTICA

El Tribunal del Trabajo de Integración Curricular certifica que: El Trabajo de Integración Curricular; Tipo: Proyecto de Investigación, **IMPLEMENTACIÓN DE LA CRIPTOGRAFÍA DE LA CURVA ELÍPTICA EN HASKELL**, realizado por el señor: **DANIEL ALEJANDRO REINOSO SALAS**, ha sido minuciosamente revisado por los Miembros del Tribunal del Trabajo de Integración Curricular, el mismo que cumple con los requisitos científicos, técnicos, legales, en tal virtud el Tribunal Autoriza su presentación.

	FIRMA	**FECHA**
Dra. Martha Ximena Dávalos Villegas. **PRESIDENTE DEL TRIBUNAL**		2024-05-20
Dr. Leonidas Antonio Cerda Romero, PhD. **DIRECTOR DEL TRABAJO DE INTEGRACIÓN CURRICULAR**		2024-05-20
Dr. Mario Humberto Paguay Cuvi, MSc. **ASESOR DEL TRABAJO DE INTEGRACIÓN CURRICULAR**		2024-05-20

DEDICATORIA

Este trabajo de Integración Curricular se lo dedico en primer lugar a mis padres Ing. Washington Filiberto Reinoso Carrasco y Sandra Salas Rosero que han sido mi pilar y apoyo durante toda mi vida; a mi hermana Angie Marian Reinoso Salas por su apoyo constante. Y para finalizar a mi familia, amigos y compañeros que me acompañaron durante este período de vida.

Daniel Alejandro

AGRADECIMIENTO

A mi padre Washington por su apoyo incondicional a lo largo del camino, a mi madre Sandra por su paciencia y cariño. Mi más sincero agradecimiento a mi Director de Tesis, Dr Leonidas Cerda, por ayudarme con su disponibilidad, motivación y enseñanzas fundamentales para el desarrollo de esta investigación. A mi asesor Dr Mario Paguay por sus enseñanzas y motivación a la matemática. Finalmente a mis amigos Andrey y Victor, por estar en los momentos más complicados y ser mi fiel soporte.

Daniel Alejandro

ÍNDICE DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE ANEXOS

RESUMEN

La criptografía de la curva elíptica es un tema de actual importancia en la seguridad de las comunicaciones contando con varias implementaciones en varios lenguajes de programación de tipo imperativo, sin embargo carece de aplicaciones en lenguajes de tipo declarativo y en particular de paradigma funcional como Haskell, lo que haría su implementación y mantenimiento mas sencilla, es por ello, que el presente trabajo implementó la criptografía de la curva elíptica en forma de una librería en el lenguaje Haskell, en particular de los algoritmos ECDH (Elliptic Curve Diffie-Hellman) y ElGamal, mediante la implementación de algoritmos de teoría de números y álgebra abstracta para la aritmética de campos finitos y curvas elípticas. La metodología empleada se basó en un enfoque cualitativo a nivel explicativo, centrado en una revisión bibliográfica especializada. Particularmente se examinaron documentos concernientes a la implementación práctica de la teoría y al uso del lenguaje Haskell. Se obtuvo una librería que implementa ambos métodos de la criptografía de la curva elíptica con funciones y métodos importantes de teoría de números, aritmética de campos finitos, curvas elípticas y los cifrados ECDH y ElGamal, en un repositorio público de GitHub, además de una completa documentación. Se concluye que este tipo de lenguaje es adecuado para este tipo de aplicaciones y facilita su mantenimiento.

Palabras clave: <CAMPOS FINITOS>, <TEORÍA DE NÚMEROS>, <CRIPTOGRAFÍA> <CURVA ELÍPTICA>, <GEOMETRÍA ALGEBRAICA>.
0590-DBRA-UPT-2024

SUMARY/ABSTRACT

Elliptic curve cryptography is a currently important topic in the security of communications, it has several implementations in different imperative programming languages, however it lacks of applications in declarative languages and particularly in functional paradigms such as Haskell, This would allow an easier implementation and maintenance, that is why the current research implemented the cryptography of the elliptic curve in the form of a library in the Haskell language, particularly in the ECDH (Elliptic Curve Diffie-Hellman) and EIGamal algorithms, through the implementation of algorithms of number theory and abstract algebra for the arithmetic of finite fields and elliptic curves. The methodology used was based on a qualitative approach at explanatory level, focused on a specialized bibliographic review. Particularly, documents concerning the practical implementation of the theory and the use of the Haskell language were studied. A library that implements both methods of elliptic curve cryptography with important functions and methods of number theory, finite field arithmetic, elliptic curves and the encrypted ECDH and ElGamal, in a public repository on GitHub, as well as a complete documentation. It is concluded that this type of language is suitable for this type of applications and facilitates their maintenance.

Keywords: <FINITE FIELDS>, <NUMBER THEORY>, <CRYPTOGRAPHY> <ELLIPTIC CURVE>, <ALGEBRAIC GEOMETRY>.

Lic. Paul Rolando Armas Pesantez Mgs
C.I. 0603289877

INTRODUCCIÓN

Actualmente las redes de comunicación han tenido un gran crecimiento, siendo útiles para transmitir información a grandes distancias y corto tiempo, pero ciertos mensajes sensibles tienen que mantenerse privados entre las dos partes de la comunicación evitando ser interceptado por un tercero. Debido a esta necesidad surgieron los métodos de cifrado, estos métodos en un inicio eran relativamente sencillos de utilizar, pero conforme avanzo la ciencia y tecnología también se volvieron fáciles los métodos de criptoanálisis y con ello el acceso a la información confidencial. De esta forma se desarrollaron métodos de cifrado (simétricos) cada vez más difíciles de romper incluso con potentes computadores. Por todos estos motivos, la necesidad del envío seguro de la información, la confidencialidad de mensajes y su autentificación es un problema vigente y de actual importancia. Para satisfacer esta necesidad se han creado distintos métodos de cifrado para que dos entidades puedan comunicarse de manera segura.

Los primeros métodos de cifrado creados fueron los cifrados simétricos, estos cumplían con el nivel de seguridad adecuado, pero aun tenían el problema del intercambio de la clave de cifrado/descifrado, de esta forma surgieron los métodos de cifrado asimétrico basado en dos claves para el intercambio de información, la clave pública y la clave privada, que basan su seguridad en la dificultad matemática de resolver el problema de determinar la clave privada a partir de la clave pública.

De los primeros métodos de cifrado asimétrico fue el algoritmo RSA (Rivest, Shamir y Adleman), que basa su seguridad en el problema de factorización de enteros muy grandes, que resultaba bastante seguro, pero tenían el problema de determinar cuales eran los primos grandes y además usaban un gran tamaño de memoria para el almacenamiento de la clave pública. Más tarde aparecieron nuevos algoritmos asimétricos que utilizan las propiedades de las curvas elípticas sobre campos finitos, que proporcionaban un nivel de seguridad equivalente, pero utilizando una menor cantidad de espacio para las claves que el algoritmo RSA y sus variantes.

Desde entonces se han realizado varias implementaciones de la criptografía de la curva elíptica en varios lenguajes de programación. Sin embargo, en lenguajes declarativos con el paradigma funcional como Haskell, caracterizado por el énfasis en su programación basada en funciones puras, el cálculo lambda, y su seguridad en los tipos de datos lo hacen un lenguaje ideal para la descripción matemática de los algoritmos de criptografía, no obstante, específicamente, no se

han realizado implementaciones adecuadas de la criptografía de la curva elíptica, siendo las pocas creadas descontinuadas y con poca documentación.

Por este motivo el presente trabajo pretende realizar una implementación de la criptografía de la curva elíptica en en lenguaje Haskell, que sea comprensible y autodocumentada en forma de una librería robusta y potente con una adecuada documentación, así mismo pretende servir como base para nuevos proyectos o investigaciones en este tema.

CAPÍTULO I

1. PROBLEMA DE INVESTIGACIÓN

1.1. Planteamiento del problema

La seguridad de la información en la era tecnológica es un tema de actual importancia debido a la necesidad de mantener protegida la información sensible entre dos entidades. Para satisfacer esta necesidad se han desarrollo varios algoritmos de cifrado, generalmente combinando cifrados simétricos y asimétricos. De entre los asimétricos en los últimos años, la criptografía basada en curvas elípticas ha ganado popularidad como un enfoque efectivo para garantizar la seguridad en diversas aplicaciones, como la autenticación, el cifrado y las firmas digitales. Los algoritmos basados en curvas elípticas, como ElGamal y ECDH (Eliptic Curve Diffie-Helman), han demostrado ser eficientes y robustos en términos de seguridad.

Sin embargo, existen pocas implementaciones de estos algoritmos en Haskell, de las cuales la mayoría están descontinuadas y con poca documentación, lo cuál es un problema ya que impide proyectos e investigaciones de la criptografía en este lenguaje, por este motivo es importante realizar una adecuada implementación documentada de estos algoritmos.

1.2. Objetivos

1.2.1. *Objetivo General*

Realizar una implementación de una librería en Haskell de la criptografía de la curva elíptica, con énfasis en los algoritmos de cifrado ECDH y el algoritmo ElGamal; mediante la revisión de bibliografía especializada; con el fin de proporcionar una base documentada para futuros proyectos e investigaciones.

1.2.2. *Objetivos específicos*

- Comprender los fundamentos matemáticos de los algoritmos de criptografía basados en curvas elípticas; a través de la revisión bibliográfica; para comprender los principios y propiedades subyacentes.
- Comprender los fundamentos matemáticos en los cuales se basa el lenguaje de programación

funcional Haskell; a traves de la revisión bibliográfica; para una correcta implementación del tema.

- Implementar los algoritmos ECDH y ElGamal, así como de los métodos necesarios para su funcionamiento, mediante la adaptación de la literatura existente y con ayuda de la librerías existentes en áreas base.
- Redactar una documentación de funcionamiento y ejemplos de uso de los métodos de la librería que muestre la relación del estudio teórico con el funcionamiento de la librería.
- Publicar la librería en los repositorios públicos de Haskell, para que sirva de base documentada a futuros proyectos e investigaciones.

1.3. Justificación

La criptografía basada en curvas elípticas posee características que la hacen más segura y fiable que otros métodos criptográficos, sin embargo para su implementación se necesita de dominio de diversas areas matemáticas como lo son la teoría de números, el algebra abstracta y la geometría algebraica, áreas que no se tratan a profundidad en la Carrera de Matemática de la Escuela Superior Politécnica de Chimborazo.

Además de la falta de documentación para una comprensión teórica de la criptografía de curva elíptica también esta el problema de la falta de documentación para una implementación práctica, sobre todo en los lenguajes de programación con el paradigma funcional.

Los lenguajes de programación de paradigma funcional son de tipo declarativo (es decir se basa más en el "que" y no en el "como"), en este tipo de paradigma no existen asignaciones destructivas (es decir que las variables cambien de valor durante la ejecución) y las variables tienen transparencia referencial. Debido a este motivo los lenguajes de tipo declarativo pueden ser razonados de forma matemática, lo cual lo hace ideal para la implementación de algoritmos matemáticos.

Dentro de los lenguajes de tipo declarativo tenemos los lenguajes de paradigma funcional que debido a que se basa en verdaderas funciones matemáticas, aporta seguridad con sus tipos de datos y poseer un estilo más matemático es ideal para el trabajo.

Por este motivo se ha seleccionado uno los lenguajes referentes y más populares de este paradigma de programación como lo es Haskell.

Sin embargo, también es un problema ya que este lenguaje tiene una curva de aprendizaje bastante elevada (además de no poseer bibliografía local y el hecho de que al ser un lenguaje académico sufre bastantes actualizaciones) ,sin embargo, no se utilizarán técnicas avanzadas del lenguaje en el trabajo.

Al ser Haskell un lenguaje académico, tiene constantes actualizaciones en su funcionamiento, por este hecho algunas librerías que antes funcionaban, ya no lo hacen, eso incluye a librerías para el manejo de campos finitos, curvas elípticas y criptografía.

Por este motivo es importante analizar, evaluar, mejorar las antiguas librerías, adecuarlas, mejorarlas y crear una nueva librería robusta en Haskell, sobre la criptografía de la curva elíptica.

CAPÍTULO II

2. MARCO TEÓRICO

En este capítulo se abarcó de forma breve todos los conceptos necesarios para la implementación de la criptografía de la curva elíptica en Haskell.

2.1. Haskell

Haskell es un lenguaje de programación puramente funcional, estáticamente tipado y de evaluación perezosa, que fue lanzado en 1990 y su desarrollo tiene sus raíces en la investigación matemática y de las ciencias de la computación (O'Sullivan et al., 2008, pág. 32). Su nombre proviene del matemático Haskell Brooks Curry, quien sentó las bases de los lenguajes de programación funcional con su trabajo sobre lógica combinatoria. (O'Sullivan et al., 2008, pág. 33). Una historia más detallada sobre el origen de Haskell puede encontrase en el libro (O'Sullivan et al., 2008, pág. 32-33). Los programas escritos en Haskell se representan siempre como funciones matemáticas, pero estas funciones nunca tienen efectos secundarios ni derivados. De este modo, cada función utilizada siempre devuelve el mismo resultado con la misma entrada, y el estado del programa nunca cambia. Por esto, el valor de una expresión o el resultado de una función dependen exclusivamente de los parámetros de entrada en el momento. En Haskell no pueden hacerse construcciones de lenguaje imperativo para programar una secuencia de declaraciones como bucles for, while y do while, en su lugar se utiliza la recursión y abstracciones matemáticas, inspiradas de áreas como el álgebra y la teoría de categorías. Haskell se basa en el cálculo lambda, por lo que el logotipo del lenguaje contiene el símbolo de esta letra griega. Es utilizado en proyectos de software que requieren alta fiabilidad y seguridad, y es conocido por su brevedad, claridad y facilidad de mantenimiento.

El uso que hace Haskell para la teoría de categorías es bastante amplio en el lenguaje, se lo usa para tratar con estructuras de datos y procesos de entrada y salida, para una información mas detallada se puede consultar la siguiente tesis de grado (Pedraza López, 2018).

La criptografía es un campo que requiere una alta seguridad, y Haskell es conocido por su capacidad para garantizar la seguridad y la fiabilidad de los programas. Además, tiene una sintaxis clara y concisa, lo que facilita la lectura y el mantenimiento del código. También es capaz de manejar grandes cantidades de datos y cálculos complejos, lo que lo hace ideal para aplicaciones criptográficas. Características como su sofisticado sistema de tipos de orden superior (Skinner, 2023,

pág. 553-615) son ideales para trabajar sobre diferentes tipos de curva elíptica y diferentes sistemas de coordenadas. Además varias de las extensiones del compilador GHC incrementan esta potencia (Skinner, 2023, pág. xix-xx), lo que lo hace un lenguaje muy versátil.

Por otro lado Haskell es popular entre los matemáticos (*Haskell and mathematics*, 2021) debido a su capacidad para expresar conceptos matemáticos de manera clara y concisa. Haskell se basa en el cálculo lambda, que es un sistema formal utilizado en matemáticas y ciencias de la computación para representar funciones y expresiones. También tiene una gran cantidad de bibliotecas matemáticas disponibles, lo que lo hace ideal para aplicaciones de este tipo.

2.1.1. *Cálculo Lambda*

El cálculo lambda es un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión. fue inventado en los tempranos años 30 por Alonzo Church, y ha sido desarrollado desde entonces como parte de sus investigaciones sobre los fundamentos de las matemáticas (Krivine, 1993). Al igual que la máquina de Turing, el cálculo lambda formaliza la idea de computación efectiva (Allen y Moronuki, 2016). El cálculo lambda se construye a partir de λ-términos

Definición 2.1 (Términos λ)**.** *Sea $x, y, \ldots,$ un conjunto contable de símbolos llamados, variables, paréntesis izquierdo y derecho, y el símbolo λ, entonces aplicando un número finito de veces las siguientes reglas:*

1. *todas las variables son λ-términos.*
2. *si t y u son cualesquiera λ-términos, entonces $(t)u$ es un λ-término denominado aplicación.*
3. *si t es cualquier λ-término y x es cualquier variable, entonces $\lambda x\, t$ es un λ-término llamado abstracción.*

Al conjunto de todos los términos del cálculo λ se denota por L.

Al igual que en el cálculo de predicados, se tiene la idea de variable libre y ligada.

Definición 2.2 (Variables libres)**.** *Las ocurrencias libres de una variable x en un término t se define, por inducción, como sigue:*

- *Si t es la variable x, entonces la ocurrencia de x en t es libre;*

- *Si $t = (u)v$, entonces las ocurrencias libres de x en t son aquellas de x en u y v;*
- *Si $t = \lambda y\, u$, las ocurrencias libres de x en t son aquellas de x en u, excepto si $x = y$; en ese caso, ninguna ocurrencia de x en t es libre.*

Estas definiciones son tomadas de (Krivine, 1993), aunque son generales.

Una *variable libre* en t es una variable que tiene al menos una libre ocurrencia en t.

Un término que no tiene variables libres se llama un término cerrado.

Un *variable ligada* en t es una variable que ocurre en t luego del símbolo λ.

2.1.1.1. *Sustitución simple*

Al igual que en el cálculo de predicados, se pueden formar términos, sustituyendo variables.

Definición 2.3 (Sustitución simple)**.** *Sean $t, t_1, \ldots, t_k$ términos y $x_1, \ldots, x_k$ distintas variables; se define, $t\langle t_1/x_1, \ldots, t_k/x_k\rangle$ el resultado de reemplazar cada libre ocurrencia de x_i en t por t_i $(1 \leq i \leq k)$. Se define la sustitución en t inductivamente, como sigue:*

- *Si $t = x_i (1 \leq i \leq k)$, entonces $t\langle t_1/x_1, \ldots, t_k/x_k\rangle = t_i$;*
- *Si t es una variable distinta de $x_1, \ldots, x_l$, entonces $t\langle t_1/x_1, \ldots, t_k/x_k\rangle = t$;*
- *Si $t = (u)v$, entonces*

$$t\langle t_1/x_1, \ldots, t_k/x_k\rangle = (u\langle t_1/x_1, \ldots, t_k/x_k\rangle)v\langle t_1/x_1, \ldots, t_k/x_k\rangle;$$

- *Si $t = \lambda x_i\, u \quad (1 \leq i \leq k)$, entonces*

$$t\langle t_1/x_1, \ldots, t_k/x_k\rangle = \lambda x_i\, u\langle t_1/x_1, \ldots, t_{i-1}/x_{i-1}, t_{i+1}/x_{i+1}, \ldots, t_k/x_k\rangle;$$

- *Si $t = \lambda x\, u$, con x distinta de $x_1, \ldots, x_k$, entonces*

$$t\langle t_1/x_1, \ldots, t_k/x_k\rangle = \lambda x\, u\langle t_1/x_1, \ldots, t_k/x_k\rangle.$$

"La sustitución simple corresponde, en ciencias de la computación, a la noción de *macro-instrucción*" (Krivine, 1993).

Demás definiciones, se pueden consultar en (Krivine, 1993), debido a que Haskell se basa en el cálculo lambda, ocupa dos características fundamentales, la equivalencia α y la reducción β.

2.1.1.2. *Equivalencia α*

La equivalencia α formaliza la idea intuitiva de que dos expresiones lambda son equivalentes si solo intercambiamos el nombre de la variable en todas sus ocurrencias. Formalmente esto se expresa con una relación de equivalencia y se denota por $\equiv$.

Intuitivamente $t \equiv t'$ significa que t' se obtiene de t renombrando las variables ligadas en t (Krivine, 1993).

Definición 2.4 (Equivalencia α)**.** *Se define $t \equiv t'$, en L, por inducción en la longitud de t, de la siguiente forma:*

- *Si t es una variable, entonces $t \equiv t'$ si y solo si $t = t'$;*
- *Si $t(u)v$, entonces $t \equiv t'$ si y solo si $t' = (u')v'$, con $u \equiv u'$ y $v \equiv v'$.*
- *Si $t = \lambda x\, u$, entonces $t \equiv t'$ si y solo si $t' = \lambda x'\, u'$ con $u\langle y/x\rangle \equiv u'\langle y/x'\rangle$ para todas las variables y excepto un número finito.*

2.1.1.3. *Reducción β*

Intuitivamente la reducción β significa aplicar un valor a una expresión lambda, lo que implica sustituir este valor en todas las ocurrencias ligadas de una variable.

Es decir, $(\lambda x\, t)u = (t)\langle x/u\rangle$, reemplazando las ocurrencias ligadas de x por u.

2.1.1.4. *Cálculo lambda tipado*

Haskell es un cálculo lambda tipado, es decir una extensión del cálculo lambda original que incorpora un sistema de tipos. Esto proporciona una forma de garantizar ciertas propiedades y comportamientos de los programas escritos en este lenguaje.

Se incorpora el sistema de tipos, con varios tipos por defecto como **Char**, **Integral**, etc. Pero también permite crear sus propios tipos por medio de las instrucciones **data** o **newtype**.

2.1.2. *Teoría de tipos*

La teoría de tipos es un área de la informática y las matemáticas que estudia los sistemas formales de tipos en los lenguajes de programación y en la lógica. Se centra en el estudio de cómo los tipos pueden ser utilizados para especificar y verificar propiedades de los programas informáticos, así como para garantizar la corrección y la seguridad de los mismos.

2.1.2.1. *Tipo de dato algebraico (ADT)*

Haskell permite la creación de tipos algebraicos, que son tipos en los que se definen sus habitantes, por ejemplo el tipo **Bool** esta dado por el tipo

```
data Bool = False | True
```

Los tipos de datos que se construyen con | se conocen como "tipos suma", ya que su número de habitantes (es decir que elementos a nivel de término son del tipo **Bool**) se determina a partir de la suma de los habitantes de sus elementos, en este caso 2, mientras que los tipos como (,)

```
data (,) a b = (,) a b
```

Que corresponden a las tuplas de dos elementos, se conocen como "tipo producto" ya que el número de habitantes se determina como un producto (que puede ser infinito como en este caso).

2.1.3. *Paradigma funcional*

El paradigma funcional es un enfoque de programación que trata a la computación como una evaluación de funciones matemáticas (cálculos lambda) y evita los cambios de estado y los efectos secundarios. En este paradigma, los programas se construyen principalmente utilizando funciones puras, que son funciones que producen el mismo resultado dado el mismo conjunto de argumentos y no tienen efectos secundarios observables.

2.1.4. *Teoría de categorías*

La teoría de categorías es un área de las matemáticas que estudia las relaciones y estructuras entre categorías matemáticas abstractas. Una categoría es un concepto matemático que consiste en objetos y flechas (también llamadas morfismos) que relacionan estos objetos. Las flechas pueden componerse de ciertas maneras, lo que abstrae las relaciones de isomorfismo, homeomorfismo, etc

entre diversas estructuras matemáticas.

Se utiliza en Haskell, para tratar con los tipos de dato **IO** que implica cálculos con operaciones de entrada y salida. También permite tratar con estructuras de datos, lo que facilita la programación.

Implícitamente, está presente en operaciones habituales en Haskell como lo son los "foldeos" o plegados, que vendrían siendo una forma de catamorfismo (Allen y Moronuki, 2016).

2.2. Criptografía

Según (Robling Denning, 1982), la criptografía es la ciencia y el estudio de la escritura secreta. Es el campo de la matemática y las ciencias de la computación que se ocupa de evitar que cualquier persona, excepto el destinatario deseado, comprenda un mensaje transmitido. Un cifrado es un método secreto de escritura por el que el texto plano (aquel texto que se pueda leer) se transforma en texto cifrado. Para que la transmisión de la información sea segura se han desarrollado varios métodos matemáticos desde la época del imperio romano hasta la actualidad (Lucena López, 2022).

El proceso de transformar el texto plano en texto cifrado se denomina cifrado; el proceso inverso de transformar el texto **cifrado** en texto plano se denomina **descifrado**. Tanto el cifrado como el descifrado se controlan mediante una o varias claves criptográficas.

Existen diferentes técnicas y algoritmos para poder cifrar y descifrar textos, a este conjunto de técnicas se llama **criptosistema**, para ser más precisos:

Definición 2.5 (Criptosistema)**.** *Un criptosistema es un par de funciones E, D, en las que cada una de ellas toma dos argumentos: un mensaje M y una clave K. Para cada clave K_E, debe haber una clave K_D tal que $D(E(M,K_E),K_D) = M$ para cada mensaje M. La clave K_E es la clave de cifrado mientras que K_D es la correspondiente clave de descifrado.*

Según lo manifiesta (Mullen y Mummert, 2007) un criptosistema D,E se dice que es *seguro* si no es computacionalmente factible determinar el mensaje verdadero M a partir de un mensaje cifrado $E(M,K_E)$. (pág. 111)

Como menciona (Robling Denning, 1982) al conjunto de técnicas que tratan de obtener el mensaje descifrado sin tener acceso a la clave (o tratar de obtener la clave) se le denomina **criptoanálisis**. (pág. 2)

De entre los criptosistemas tenemos dos tipos, el criptosistema de clave *simétrica* donde las claves K_D y K_E son siempre iguales (Mullen y Mummert, 2007, pág. 112). Tipos de estos criptosistemas son los de sustitución y transposición como el AES o el DES. El otro tipo de criptosistema son los de clave asimétrica, donde no necesariamente tienen que ser iguales las claves, en este tipo de cifrado tenemos el algoritmo RSA. Este trabajo se enfocó en estos últimos ya que involucra varias áreas de la matemática como a teoría de números, el álgebra abstracta, probabilidad y la teoría de la información (Hoffstein et al., 2014).

Definición 2.6 (Criptosistema de clave pública)**.** *Un criptosistema de clave pública es un criptosistema en el que un destinatario calcula un par de claves K_E,K_D y anuncia la clave K_E (que se denomina clave pública) a todos los posibles remitentes. La clave privada K_D se mantiene en secreto. Por tanto, cualquiera que posea la clave pública puede cifrar un mensaje para el destinatario, pero (si el sistema es seguro) sólo quien posea la clave privada puede descifrar el mensaje.*

La criptografía de clave pública (o criptografía asimétrica) es de gran uso en la actualidad, y nació en el último tercio del siglo XX. Para entender esto se debe remontar a la creación de la criptografía asimétrica en 1976 por Diffie y Hellman con su artículo "New directions on criptography", en donde se pusieron las bases de la criptografía asimétrica (Diffie y Hellman, 1976). Posteriormente se invento el criptosistema de clave pública RSA por Rivest, Shamir y Adleman en 1978 (Hoffstein et al., 2014), que se basaba en el problema de la factorización de números primos muy grandes.

Posteriormente, Diffie y Hellman utilizaron propiedades de los campos finitos para desarrollar otros criptosistemas, se basaron en el problema de logaritmos discretos sobre el grupo multiplicativo de un campo finito.

2.3. Campos finitos

Para hablar de campos finitos, se necesita de conocimientos previos de Álgebra Abstracta (o Álgebra Moderna), algunos términos de esta asignatura no se definieron a lo largo del trabajo y se supone que el lector domine.

Un campo finito es un campo que contiene un número finito de elementos. Como ejemplos más conocidos de campos finitos tenemos a los campos primos, que no son más que el campo cociente $\mathbb{Z}/p\mathbb{Z}$ con p un número primo (los enteros módulo p), a lo largo del trabajo se refiere a estos campos como campos primos y se denotan como $\mathbb{F}_p$. Pueden usarse estos campos para

las aplicaciones criptográficas, pero se prefirió otro tipo de campos. El tipo de campo finito más general y más utilizados son los campos de Galois. Estos campos son fundamentales en áreas de la matemática y las ciencias de la computación como la teoría de números, la geometría algebraica, la teoría de Galois, y en nuestro caso la criptografía.

Definición 2.7 (Campo de Galois)**.** *Un campo de Galois es un campo finito con orden de la forma $q = p^n$ con p un número primo y n un entero positivo. Se denotan como $\mathbb{F}_q$ o $\mathbb{GF}(q)$.*

Con $n = 1$ tendremos los campos primos, por lo tanto estos no son más que casos particulares de campos de Galois. No solo eso, está demostrado que si existe un campo finito, entonces este debe tener p^n elementos con p primo y n natural, y el recíproco también si tenemos un número $q = p^n$ entonces existe un solo campo de Galois (salvo isomorfismos) con q elementos. Estas afirmaciones son los siguientes resultados.

Teorema 2.1. *Sea F un campo finito. La cardinalidad de F es p^m. donde p es la característica de F y m es el grado de F sobre su subcampo primo.*

Teorema 2.2 (Existencia y unicidad de los campos finitos)**.** *Para cada primo p y cada entero positivo $n \geq 1$ existe un campo finito con p^n elementos. Cualquier campo finito con p^n elementos es isomorfo al campo de descomposición de $x^{p^n} - x$ sobre $\mathbb{F}_p$.*

La demostración de estos dos resultados puede encontrarse en (Mullen y Mummert, 2007, pág. 13-14).

Para la construcción explicita de un campo $\mathbb{F}_{p^n}$ se necesita de un polinomio irreducible $p(x)$ de grado n sobre el anillo de polinomios $\mathbb{F}_p[X]$. Entonces el anillo factor (que es un campo debido a que $p(x)$ es irreducible), será isomorfo al campo de Galois

$$\mathbb{F}_{p^n} \cong \mathbb{F}_p / \langle p(x) \rangle$$

Esto se determina con el siguiente resultado

Proposición. *Sea $f(x) \in \mathbb{F}_p[X]$ irreducible con grado d. Entonces $|\mathbb{F}_p[X]/\langle f(x)\rangle| = p^d$. Más aún, cada elementos de $\mathbb{F}_p[X]/f(x)$ es congruente a exactamente un polinomio de grado menor o igual que d (o el polinomio 0), y dos polinomios de estos polinomios distintos nunca son congruentes módulo $f(x)$.*

La demostración puede encontrarse en (Magner, 2020).

De esta forma se pueden construir campos finitos, tomando un campo finito F_p, encontrar un polinomio irreducible en $F_p[X]$ y las operaciones de campo son la suma (módulo p) para cada coeficiente y producto de polinomios módulo $f(x)$. A esta representación del campo finito se la conoce como representación en base polinomial. Sin embargo aún queda el problema de como hallar estos polinomios, los casos donde $n = 2$ o $n = 3$ son relativamente sencillos ya que se pueden evaluar en todos los puntos del campo $\mathbb{F}_p$ y verificar que no se anulen, sin embargo enteros tan pequeños no pueden ser aplicados en criptografía.

Además si existe un polinomio irreducible de grado n sobre $\mathbb{F}_p$ este puede no ser único, por lo tanto para la estanderización de estos polinomios se utilizaron los polinomios de Conway.

Definición 2.8. *Sea C_n el conjunto de polinomios de grado n sobre $\mathbb{F}_p$. Sea $g(X) = g_n X^n + \ldots + g_0$ y $h(X) = h_n X^n + \ldots + h_0$. Entonces podemos definir $g < h$ si y solo si existe un índice k con $g_i = h_i$ para $i > k$ y $(-1)^{n-k} g_k < (-1)^{n-k} h_k$.*

Definición 2.9 (Polinomios de Conway)**.** *El Polinomio de Conway $f_{p,n}(X)$ de $\mathbb{F}_{p^n}$ es el polinomio más pequeño con respecto al orden definido antes tal que*

1. *$f_{p,n}(X)$ es mónico.*
2. *$f_{p,n}(X)$ es primitivo, esto es, cualquier cero es un generador de $F^*_{p^n}$.*
3. *Para cualquier divisor propio m de n se tiene que $f_{p,m}(X^{(p^n-1)/(p^m-1)}) = 0 \mod f_{p,n}(X)$; esto es, la potencia $(p^n-1)/(p^m-1)$ de un zero de $f_{p,n}(X)$ es un cero de $f_{p,m}(X)$.*

La existencia de estos polinomios puede mostrarse con el Teorema Chino del Residuo y una base de datos de estos se encuentra en (Lübeck, 2021).

En particular se utilizaron los campos binarios $\mathbb{F}_{2^n}$, ya que admiten una representación binaria que simplifica los cálculos, aquí se tiene un ejemplo de los polinomios de Conway en formato binario.

Tabla 2-1: Polinomios de Conway para el campo $\mathbb{F}_2$

Entero	Polinomio en $\mathbb{F}_2[X]$	Representación binaria	R. entera	R. hexadecimal
2	X^2+X+1	111	7	7
3	X^3+X+1	1011	11	B
4	X^4+X+1	10011	19	13
5	X^5+X^2+1	100101	37	25
6	$X^6+X^4+X^3+X+1$	1011011	91	5B

Realizado por: Reinoso, D., 2024.

2.3.1. *El problema de los Logaritmos Discretos*

El problema de los logaritmos discretos se puede definir en cualquier grupo, pero es de especial importancia en los campos finitos, y basa su definición en el hecho de que el grupo multiplicativo de un campo finito tiene un elemento primitivo g, es decir que cualquier elemento diferente del cero en $\mathbb{F}_q$ es igual a alguna potencia de g (o que g genera a todo el campo).

Definición 2.10. *Sea g una raíz primitiva de F_q y sea h un elemento distinto del cero de F_q. El Problema del Logaritmo Discreto es el problema de encontrar un exponente x tal que*

$$g^x = h$$

El número x se denomina el logaritmo discreto de h base g y se denota por $\log_g(h)$.

La forma más general del logaritmo discreto se encuentra en los grupos

Definición 2.11. *Sea G un grupo cuya ley de grupo está denotada por $*$. El problema del logaritmo discreto de G es determinar, para cualesquiera dos elementos g y h en G, un entero x que satisfaga*

$$\underbrace{g*g*g*\cdots*g}_{x\ veces} = h.$$

En base al problema del logaritmo discreto sobre el grupo multiplicativo de un campo finito se pueden plantear los criptosistemas de ElGamal y ECDH, sin embargo existen algoritmos de tiempo subexponencial para resolver estos problemas (Hoffstein et al., 2014, pág. 81-94). Por esta razón se utilizan problemas análogos mediante el uso de curvas elípticas, debido a que en este caso los criptoanálisis conocidos actualmente tienen complejidad exponencial (Hoffstein et al., 2014, pág.

315-316).

2.4. Curvas elípticas

El campo de las curvas elípticas es un tema amplio en la matemática, ya que es un tema de actual investigación la búsqueda puntos racionales en estas curvas, desde ser utilizadas para resolver el último teorema de Fermat hasta su uso actual en la conjetura de Conjetura de Birch y Swinnerton-Dyer, este trabajo se enfocó en solo aquellas definidas para campos finitos binarios, es decir aquellos de la forma $\mathbb{F}_{2^n}$ por su facilidad de representación.

El uso de estas curvas para propósitos criptográficos fue propuesto en 1985 por Koblitz (1987) y Miller (1986) en 1985, que sugirieron usar el grupo de curva elíptica para los criptosistemas basados en el logaritmo discreto (Hankerson et al., 2004, pág. 21).

Definición 2.12 (Curva elíptica)**.** *Una curva elíptica E es una ecuación no singular de la forma*

$$y^2 = x^3 + Ax + B,$$

donde A y B son constantes. A esta forma se le suele referir como la ***ecuación de Weierstrass*** *de una curva elíptica.*

Usualmente, A y B son elementos de un campo $\mathbb{K}$, por ejemplo, los números reales $\mathbb{R}$, los números complejos $\mathbb{C}$, los números racionales $\mathbb{Q}$, los campos primos o los campos de Galois (Washington, 2008, pág. 9) . La curva elíptica también podría verse como el conjunto de puntos en $\mathbb{K}$ donde el polinomio $\mathbb{K}[x,y]$ se anula.

Por ejemplo para el campo $\mathbb{F}_{13}$, se tiene el siguiente conjunto de puntos que satisface la ecuación de curva elíptica $E := y^2 = x^3 + 3x + 8$.

$$E(\mathbb{F}_{13}) = \{(1,5),(1,8),(2,3),(2,10),(9,6),(9,7),(12,2),(12,11)\}$$

El uso de curvas elípticas se da debido a que hay una forma de sumar sus elementos y añadiendo un elemento al conjunto llamado “punto en el infinito” ($\mathcal{O}$) se obtiene una estructura de grupo.

Si la curva elíptica es sobre $\mathbb{R}$, se tiene una interpretación geométrica de la “suma” de puntos de una curva elíptica. Primero si $P = (x_1, y_1)$ y $Q = (x_2, y_2)$ son dos puntos distintos de la curva

elíptica E, entonces trazamos la línea secante entre ellos, esta linea interseca a la curva en otro punto $R = (x_3, y_3)$, reflejamos este punto en el eje Y y así obtener el punto $R' = (x_3, -y_3)$ que sería la operación suma de P y Q (véase **Ilustración-2-1**).

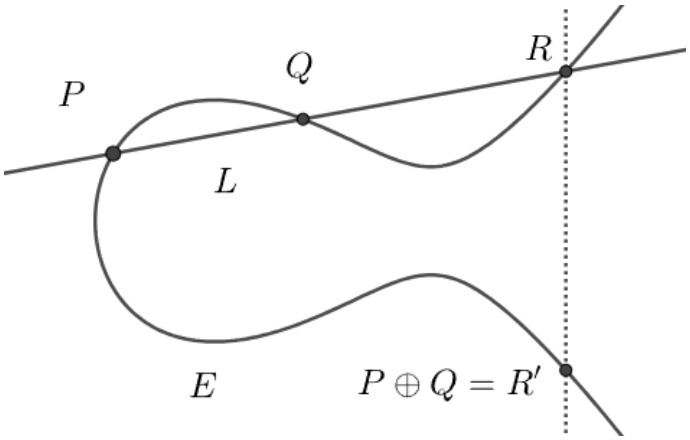


Ilustración 2-1: Suma de dos puntos distintos de una curva elíptica

Realizado por: Reinoso, D., 2024

Si $P = Q$, entonces se toma la línea tangente en P que interseca a la curva en un punto R y se toma su reflejo en el eje Y (véase **Ilustración-2-2**).

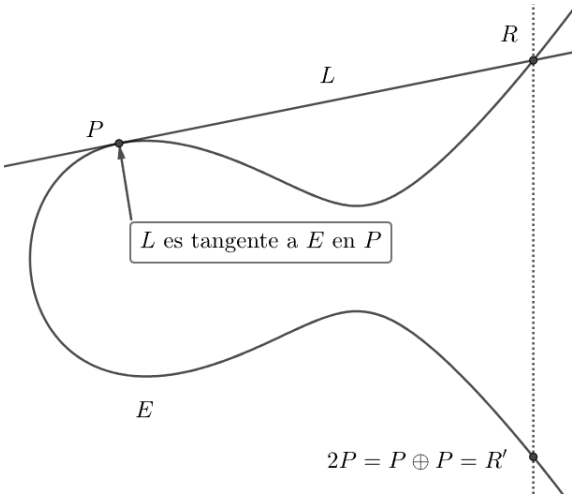


Ilustración 2-2: Suma de un punto consigo mismo

Realizado por: Reinoso, D., 2024

Y en el otro caso si $P \neq Q$ pero $x_1 = x_2$, entonces la línea que pasa por P y Q interseca a la curva E en el infinito, y representamos este punto por $\mathcal{O}$. La siguiente definición resume este análisis.

Definición 2.13. *Sea E definida por $y^2 = x^3 + Ax + B$. Sea $P_1 = (x_1, y_1)$ y $P_2 = (x_2, y_2)$ puntos en E con $P_1, P_2 \neq \mathcal{O}$. Definimos $P_1 \oplus P_2 = P_3 = (x_3, y_3)$ como sigue:*

1. *Si $x_1 \neq x_2$, entonces*

$$x_3 = m^2 - x_1 - x_2, \qquad y_3 = m(x_1 - x_3) - y_1, \qquad \text{donde } m = \frac{y_2 - y_1}{x_2 - x_1}.$$

2. *Si $x_1 = x_2$ pero $y_1 \neq y_2$, entonces $P_1 \oplus P_3 = \mathcal{O}$.*

3. *Si $P_1 = P_2$, y $y_1 \neq y_2$, entonces*

$$x_3 = m^2 - 2x_1, \qquad y_3 = m(x_1 - x_3) - y_1, \qquad \text{donde } m = \frac{3x_1^2 + A}{2y_1}$$

4. *Si $P_1 = P_2$ y $y_1 = 0$, entonces $P_1 \oplus P_2 = \mathcal{O}$ Y además definimos*

$$P \oplus \mathcal{O} = P$$

para todos los puntos P en E.

Para que esta operación este bien definida, es necesario que la curva no se autointerseque a si misma para el caso de la suma de dos puntos distintos, y también es necesario que la curva no tenga "picos", para que al doblar puntos (sumarlo consigo mismo) no se tenga problemas al calcular la pendiente. Esta condición se obtiene calculando el discriminante de la curva (para su deducción consultar (Silverman y Tate, 2015, pág. 28) o (Castillo, 2023, pág. 55)).

Definición 2.14. *Sea E una ecuación de Weierstrass de una curva elíptica definida como antes, el discriminante de E es el número*

$$\triangle_E = 4A^3 + 27B^2.$$

Para que una curva de Weierstrass esté bien definida es necesario que $\triangle_E \neq 0$.

El siguiente teorema asegura que cuando la curva es no singular, este conjunto con la operación suma definida antes tiene la estructura de grupo abeliano.

Teorema 2.3. *La suma de puntos en una curva elíptica E satisface las siguientes propiedades:*

1. *(conmutatividad) $P_1 \oplus P_2 = P_2 \oplus P_1$ para todo P_1, P_2 en E.*

2. *(existencia de identidad) $P \oplus \mathcal{O} = P$ para todos los puntos P en E.*

3. *(existencia de inversos) Dado P en E, entonces existe P' en E con $P \oplus P' = \mathcal{O}$. Este punto P' usualmente se denota por $-P$.*

4. *(asociatividad)* $(P_1 \oplus P_2) \oplus P_3 = P_1 \oplus (P_2 \oplus P_3)$.

La demostración puede encontrarse en (Washington, 2008). Como en estas operaciones solo se hace uso de sumas, productos, inversas y restas, se puede cambiar el campo $\mathbb{R}$ por cualquier otro campo K, (por ejemplo $\mathbb{F}_p$) y las operaciones propuestas seguirán estando en la curva elíptica. Sin embargo cuando la característica del campo es 2 o 3, el discriminante se anula y existen punto singulares, para solucionar este problema se plantea una ecuación más general.

Definición 2.15 (Ecuación generalizada de Weierstrass)**.** *La ecuación de Weierstrass generalizada de una curva elíptica E sobre un campo K viene dada por*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

donde $a_i \in K$ para $i \in \{1, \ldots, 6\}$.

Si la característica del campo no es ni 2 ni 3, entonces se puede realizar el siguiente cambio de variable

$$(x,y) \to \left(\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1x}{216} - \frac{a_1^3 + 4a_1a_2 - 12a_3}{24} \right)$$

con lo que la ecuación generalizada de Weierstrass se convierte a la ecuación de Weierstrass normal.

Si la característica es 2, y $a_1 \neq 0$, el siguiente cambio de variable

$$(x,y) \to \left(a_1^2x + \frac{a_3}{a_1}, a_1^3y + \frac{a_1^2a_4 + a_3^2}{a_1^3} \right)$$

trasforma la curva generalizada a la curva

$$y^2 + xy = x^3 + ax^2 + b$$

donde $\triangle_E = b$. Si $a_1 = 0$, entonces el siguiente cambio de variable

$$(x,y) \to (x + a_2, y)$$

transforma la ecuación a la curva

$$y^2 + cy = x^3 + ax + b$$

con discriminate $\triangle_E = c^4$.

Entonces para los campos binarios con $a_1 \neq 0$ se puede demostrar (Hankerson et al., 2004, pág. 79) que la suma de puntos de curva elíptica queda definida de la siguiente manera:

Definición 2.16 (Ley de grupo para la curva $E(\mathbb{F}_{2^n}) : y^2 + xy = x^3 + ax^2 + b$)**.**

1. *Identidad.* $P \oplus \mathcal{O} = \mathcal{O} \oplus P$ *para todo* $P \in E(\mathbb{F}_{2^n})$.

2. *Inversas. Si* $P = (x,y) \in E(\mathbb{F}_{2^q})$*, entonces* $(x,y) \oplus (x,x+y) = \mathcal{O}$.

3. *Suma de puntos. Sea* $P = (x_1,y_1) \in E(\mathbb{F}_{2^n})$ *y* $Q = (x_2,y_2) \in E(\mathbb{F}_{2^n})$*, donde* $P \neq \pm Q$*. Entonces* $P \oplus Q = (x_3,y_3)$*, donde*

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad y \quad y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

   con $\lambda = \dfrac{y_1 + y_2}{x_1 + x_2}$.

4. *Doblaje de puntos. Sea* $P = (x_1,y_1) \in E(F_{2^n})$*, donde* $P \neq -P$*. Entonces* $2P = (x_3,y_3)$*, donde*

$$x_3 = \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2} \quad y \; y_3 = \lambda x_3 + x_3$$

   con $\lambda = x_1 + \dfrac{y_1}{x_1}$.

Y cuando $a_1 = 0$, se tiene la siguiente definición de suma

Definición 2.17 (Ley de grupo para la curva $E(\mathbb{F}_{2^n}) : y^2 + cy = x^3 + ax + b$)**.**

1. *Identidad.* $P \oplus \mathcal{O} = \mathcal{O} \oplus P = P$ *para todo* $P \in E(\mathbb{F}_{2^n})$.

2. *Inversas. Si* $P = (x,y) \in E(\mathbb{F}_{2^n})$*, entonces* $(x,y) \oplus (x,y+c) = \mathcal{O}$.

3. *Suma de puntos. Sea* $P = (x_1,y_1) \in E(\mathbb{F}_{2^n})$ *y* $Q = (x_2,y_2) \in E(\mathbb{F}_{2^n})$*, donde* $P \neq \pm Q$*. Entonces* $P \oplus Q = (x_3,y_3)$*, donde*

$$x_3 = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)^2 + x_1 + x_2 \quad y \quad y_3 = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)(x_1 + x_3) + y_1 + c.$$

4. Doblaje de puntos. Sea $P = (x_1, y_1) \in E(\mathbb{F}_{2^n})$, donde $P \neq -P$. Entonces $2P = (x_3, y_3)$, donde

$$x_3 = \left(\frac{x_1^2 + a}{c}\right)^2 \quad y \quad \left(\frac{x_1^2 + a}{c}\right)(x_1 + x_3) + y_1 + c.$$

Estas últimas ecuaciones, son para curvas elípticas denominadas *supersingulares* (Hankerson et al., 2004, pág. 83), que por motivos de seguridad no se utilizan en la práctica, aunque se las incluyó en la librería.

En el trabajo se utilizó la base de datos **STD** de Curvas elípticas con parámetros ya dados (Jancar y Sedlacek, 2020). Por ejemplo la curva "SECT113R1" es una curva de Weierstrass de campo binario de 113 bits ($\mathbb{F}_{2^{113}}$) con ecuación:

$$y^2 + xy = x^3 + ax^2 + b$$

Tabla 2-2: Parámetros de la curva SECT113R1

Nombre	Valor
m	113
$f(x)$	$x^{113} + x^9 + 1$
a	0x003088250ca6e7c7fe649ce85820f7
b	0x00e8bee4d3e2260744188be0e9c723
G	(0x009d73616f35f4ab1407d73562c10f, 0x00a52830277958ee84d1315ed31886)
n	0x0100000000000000d9ccec8a39e56f
h	0x2

Fuente: Jancar y Sedlacek, 2020.
Realizado por: Reinoso, D., 2024.

En el que el artículo (Research, 2000), describe los parámetros:

- m es la dimensión del campo $\mathbb{F}_{2^m}$ visto como espacio vectorial sobre su subcampo primo.
- $f(x)$ es el polinomio irreducible sobre $\mathbb{F}_2$.
- a y b son los coeficientes de la curva generalizada de Weierstrass.
- G es un punto base en $E(\mathbb{F}_{2^m})$.
- n es un primo y orden de G, $n = |\langle G \rangle|$.
- h que es el cofactor de la curva, se define como $h = \dfrac{|E(\mathbb{F}_{2^m})|}{n}$.

Y para un campo primo, se utilizó la curva SECP112R1, una curva para un campo primo de 112 bits con ecuación de Weierstrass.

$$y^2 = x^3 + ax + b$$

con parámetros:

Tabla 2-3: Parámetros de la curva SECP112R1

Nombre	Valor
p	0xdb7c2abf62e35e668076bead208b
a	0xdb7c2abf62e35e668076bead2088
b	0x659ef8ba043916eede8911702b22
G	(0x09487239995a5ee76b55f9c2f098, 0xa89ce5af8724c0a23e0e0ff77500)
n	0xdb7c2abf62e35e7628dfac6561c5
h	0x01

Fuente: Jancar y Sedlacek, 2020.

Realizado por: Reinoso, D., 2024.

Donde los parámetros se especifican en (Research, 2000) como:

- p un número primo que especifica el campo $\mathbb{F}_p$.
- a y b son dos elementos de F_p que determinan la ecuación de Weierstrass.
- G es un punto base (x_G, y_G) en $E(F_p)$ que sirve como generador.
- Un primo n que es el orden de G.
- Un entero h que es el cofactor $\frac{|E(\mathbb{F}_p)|}{n}$.

2.4.1. *Sistemas de coordenadas*

Para la implementación de la aritmética de las curvas elípticas además de la operación de suma, se debe utilizar un sistema de coordenadas que puedan describir los elementos del grupo, para ello se pueden utilizar varios sistemas de coordenadas, el que se utiliza para formalizar la idea de punto en el infinito en los libros donde se demuestran estas propiedades es el sistema de coordenadas proyectivo, en donde se homogeniza la ecuación de Weierstrass en una variable z y el punto en el

infinito se suele representar mediante geometría proyectiva (Washington, 2008, pág. 42) como el punto $[0:1:0]$. Sin embargo, en el trabajo, solo se utilizó las coordenadas afines ya que es más sencillo de representar debido a la útil característica de Haskell, los tipos de datos algebraicos (Allen y Moronuki, 2016, pág. 590-703). Otros sistemas de coordenadas como las coordenadas jacobianas y de Edwards (Washington, 2008, pág. 43), no se han implementado debido a la naturaleza ilustrativa del trabajo.

2.4.2. *El problema de los Logaritmos Discretos en Curvas Elípticas*

Al igual que en el caso del grupo multiplicativo de un campo finito, tenemos para el grupo de curva elíptica el problema del logaritmo discreto.

En la actualidad no existen algoritmos eficientes que sean capaces de calcular en tiempo razonable logaritmos de esta naturaleza, y muchos esquemas criptográficos basan su resistencia en esta circunstancia.

El esquema es similar, sin embargo el grupo de curva elíptica $E(K)$ no es necesariamente cíclico, para tener un problema de logaritmo discreto se selecciona un punto $P \in E(K)$ y se toma su subgrupo cíclico generado $\langle P \rangle$, el problema del logaritmo discreto consiste entonces si $kP = P'$ hallar k a partir de P' y P.

2.5. Criptografía de Curva Elíptica

Para el grupo de curva elíptica se pueden plantear criposistemas basados en el problema del logaritmo discreto similares a los del grupo multiplicativo de un campo finito. La ventaja que se obtiene es que, con claves más pequeñas, se alcanza un nivel de seguridad equiparable.

2.5.1. *Intercambio de claves de Diffie-Hellman*

El intercambio de clave de Diffie-Hellman soluciona el problema de intercambiar claves que puedan ser usadas en criptosistemas simétricos en un canal inseguro. Para ello las dos partes de la comunicación A y B (que se llamaran por Alice y Bob) deben ponerse de acuerdo en usar una curva elíptica E, sobre un campo finito $\mathbb{F}_q$ y un punto de esta $P \in E(\mathbb{F}_q)$, entonces para que Alice y Bob puedan intercambiar claves deben seguir el siguiente algoritmo:

1. Alice escoge un entero secreto a, calcula $P_a = aP$ y se lo envía a Bob.

2. Bob escoje un entero secreto b, calcula $P_b = bP$ y se lo envía a Alice.

3. Alice calcula $aP_b = abP$.

4. Bon calcula $bP_a = baP$.

Debido a que $abP = baP$, entonces Alice y Bob se han intercambio esta información que puede servir para otros cifrados, y nadie más que ellos pueden conocer la clave. Si un externo desea averiguar la clave abP, debe poder calcularla a partir de la información pública, que es P y o bien aP o bP, este es el problema subyacente y se conoce como el **Problema de Diffie-Helmann** (Washington, 2008, pág. 171).

2.5.2. *Cifrado de ELGamal sobre Curvas Elípticas*

El cifrado de ElGamal se basa en el problema del logaritmo discreto y se utiliza cuando se desea enviar un mensaje explícito (en lugar de una clave como en Diffie-Hellman), sin embargo en la práctica no se suele hacer esto, ya que comunicarse directamente por cifrados asimétricos suele ser bastante lento comparado con los cifrados simétricos, en su lugar en envía una clave previa y posteriormente se utiliza un cifrado simétrico como el AES.

Para realizar este cifrado Alice y Bob se ponen de acuerdo en una curva elíptica E, un campo finito $\mathbb{F}_q$ y un punto P, tal que el subgrupo generado tenga un orden primo grande (Washington, 2008, pág. 174). Luego Bob escoge un entero secreto s y calcula $B = sP$. De esta forma la clave pública de Bob es la tupla $(E, \mathbb{F}_q, P, B)$ y su clave privada es el entero s. Nótese que para poder deducir la clave privada a partir de la pública se debe resolver el logaritmo discreto.

Posteriormente para que Alice pueda enviar un mensaje m a Bob debe realizar el siguiente algoritmo

1. Expresa su mensaje m como un punto de curva elíptica (normalmente mediante alguna codificación En de su mensaje) $M = \text{En}(m) \in E(\mathbb{F}_q)$.

2. Escoge un entero secreto k y calcula $M_1 = kP$.

3. Calcula $M_2 = M \oplus kB$.

4. Envía (M_1, M_2) a Bob.

Finalmente para que Bob pueda descifrar el mensaje, hace uso de su clave privada s y calcula

$$M_2 - sM_1 = M \oplus kB - skP = M \oplus ksP - skP = M.$$

El problema de Diffie-Hellman y el problema del logaritmo discreto son equivalentes en el siguiente sentido: Si existe una forma de resolver el problema de Diffie-Hellman entonces se puede resolver el problema del logaritmo discreto y viceversa (Hoffstein et al., 2014, pág. 73).

Sin embargo el problema de representar un mensaje m de texto plano fue complicado debido a que tanto Koblitz (1998, pág. 205) como Washington (2008, pág. 174) y Yang (2012, pág. 354) proponían usar un método iterativo basado en un error probabilístico pequeño de que el texto no se pudiera representar. Para facilitar esto se utilizó la variante Menezes-Vanstone de ElGamal descrita en (Silverman, 2009, pág. 417) y siguiendo el siguiente algoritmo.

1. Igual que antes la clave pública de Bob es la tupla $(E, \mathbb{F}_q, P, B)$ con $B = sP$ para un entero secreto s.
2. Esta vez Alice codifica su mensaje m como un punto del plano afín $(m_1, m_2) \in \mathbb{F}_q^2$.
3. Luego escoge un entero aleatorio secreto k y calcula los puntos $M_1 = kP$ y $M_2 = kB$.
4. Luego tomando las coordenadas $(x, y) = M_2$, se calcula $c_1 = xm_1$ y $c_2 = ym_2$ y se envía a Bob el texto cifrado (M_1, c_1, c_2).

Finalmente para que Bob descifre el mensaje el debe usar su clave privada s para multiplicarla por el punto M_1, así obtiene M_2.

$$sM_1 = skP = ksP = kB = M_2$$

luego toma las coordenadas de $(x, y) = M_2$ y calcula las inversas $x^{-1}, y^{-1} \in \mathbb{F}_q$, posteriormente recupera el texto cifrado mediante

$$x^{-1}c_1 = x^{-1}xm_1 = m_1 \qquad y^{-1}c_2 = y^{-1}ym_2 = m_2.$$

y solo basta decodificar el mensaje (m_1, m_2) para obtener el mensaje original m.

2.5.2.1. *Codificación y decodificación*

Para describir mejor el proceso, se define un concepto elemental en la geometría algebraica.

Definición 2.18 (Espacio afín)**.** *Sea K un campo, el* ***espacio afín*** *de dimensión n sobre el campo K, se define como*

$$A^n(K) = \{(a_1, \ldots, a_n) \mid a_i \in K, 1 \leq i \leq n\}.$$

Algunos libros en la definición anterior, requiere que se tome el plano afín en la clausura algebraica de K.

Entonces, como se describió en el algoritmo de ElGamal (variante Menezes-Vanstone) se necesita representar el mensaje m como un punto del espacio afín $A^2(\mathbb{F}_{2^n})$. Para ello se implementó una función que permite codificar texto a un número en Haskell, mediante un foldeo y uso de la función ord que da la representación entera de un carácter.

```
encodeToF :: String –> F2n
encodeToF = foldl' (\acc c –> acc * 256 + fromIntegral (ord c)) 0
```

Donde F2n es el tipo de campo binario, el tipo que devuelve encodeToF debe ser un entero, pero se ha implementado una instancia de Integer adecuada a la clase F2n para que pueda convertirse a entero (en resumen cada entero se representa en binario y este elemento en binario se considera un polinomio de $\mathbb{F}_{2^n}$ y a la inversa).

Luego, el texto cifrado se divide en 2 y a cada cadena se le aplica la codificación, de esta manera se representa el mensaje m como un punto de $A^2(\mathbb{F}_{2^n})$.

```
encodeToF2 :: String –> (F2n,F2n)
encodeToF2 text = (encodeToF s1, encodeToF s2)
  where (s1,s2) = splitM text

splitM :: String –> (String, String)
splitM text = (s1,s2)
  where longitud = div (length text) 2
        (s1, s2) = splitAt longitud (text)
```

para la decodificación el proceso es el inverso.

```
decode :: F2n –> String
decode = decode' . toInteger
```

```
decode' :: Integer -> String
decode' 0 = ""
decode' n = decode' (n `div` 256) ++ [chr $ fromIntegral (n `mod` 256)]

decodeFromF2 :: (F2n, F2n) -> String
decodeFromF2 (f1,f2) = decode f1 ++ decode f2
```

Este proceso de codificación y decodificación no es muy complejo, pero basta para propósitos ilustrativos. También es de destacar el uso de palabras clave de Haskell como la cláusula **where** que vendría siendo el análogo a la palabra "donde" en una definición o demostración matemática.

También el uso del *encaje de patrones* en la función decode' con el caso 0 y $n \neq 0$, recuerda a la definición matemática de inducción (también en `decodeFromF2` se usa para extraer las coordenas sin necesidad de más funciones). El uso de la composición de funciones mediante el operador elegante "." y el uso de folders (que vendrían siendo catamorfismos en teoría de categorías aplicados a estructuras de datos Haskell) y demás características muestran lo bien que se adapta el lenguaje Haskell al estilo matemático del proyecto.

2.6. Teoría de números

La teoría de números es un área bastante amplia de la matemática que no se pensaba que tenía utilidad práctica hasta que apareció la criptografía, sobretodo con el criptosistema RSA que utiliza propiedades de los números primos.

En este trabajo utilizaremos resultados y propiedades de esta área para comprobar la primilidad de números, mediante un test ingenuo y otro mediante el test de Miller-Rabin.

El test de primalidad ingenuo se basa en el siguiente resultado

Teorema 2.4. *Un entero positivo n mayor que 1 es compuesto si y solo si n tiene un divisor d que satisface* $2 \leq d \leq \sqrt{n}$.

La demostración se encuentra en (Johnsonbaugh, 2023, pág. 216). Mediante este resultado podemos comprobar si un número n es primo o no, verificando si tiene algún divisor d entre 2 y $\lfloor\sqrt{n}\rfloor$ (donde $\lfloor x \rfloor$ denota la función piso de x, es decir el entero m tal que $m \leq x \leq m+1$), en la librería se implementa de la siguiente manera

```
isPrimeTrialDivision :: Integer -> Bool
isPrimeTrialDivision n = all (\p -> not (divides p n)) $ ps
  where
    ps = enumFromTo 2 nsqrt
    nsqrt = integerSquareRoot n
```

El otro algoritmo es el test probabilístico de Miller-Rabin, primero se definen los testigos Miller-Rabin de la composidad de un número n

Definición 2.19. *Sea n un número impar y se escribe $n-1 = 2^k q$ con q impar. Un entero a que satisface* $\gcd(a,n) = 1$ *es llamado un testigo de Miller-Rabin para (la composidad) de n si las siguientes condiciones se cumplen:*

1. $a^q \not\equiv \pmod{n}$
2. $a^{2^i q} \not\equiv -1 \pmod{n}$ *para todo* $i = 0,1,2\ldots,k-1$.

Y la siguiente proposición

Proposición. *Sea p un primo impar y se escribe*

$$p-1 = 2^k q \qquad \text{con } q \text{ impar}$$

Sea a cualquier número que no sea divisible por p. Entonces una de las siguientes condiciones se cumplen

1. *a^q es congruente a 1 módulo p.*
2. *Uno de los $a^q, a^{2q}, a^{4q}, \ldots, a^{2^{k-1}q}$ es congruente a -1 módulo p.*

La demostración puede encontrarse en (Hoffstein et al., 2014, pág. 130).

Con esto se tiene que si existe un testigo a de Miller Rabin para n, entonces n es un número compuesto (Hoffstein et al., 2014).

La implementación en Haskell es la siguiente:

```
millerRabinWitness :: Integer -> Integer -> Bool
millerRabinWitness n a
  | even n = False
  | a <= 1 && a>= n-1 = False
  | gcd a n /= 1 = False
  | powerMod a q n /= 1 && all (\x -> x /= (-1))
    [powerMod a (2*i*q) n | i <-[0,1..(k-1)]] = True
  | otherwise = False
    where k = decompone (n-1)
          q = (n-1) `div` k
millerRabinTest :: Integer -> Bool
millerRabinTest n = not . any (millerRabinWitness n) $ fst
 (randomIntegerList (fromIntegral (n*25 `div` n))  (1,n-1) (gen 10) )

randomIntegerList :: Int -> (Integer, Integer) -> StdGen -> ([Integer], StdGen)
randomIntegerList n range = runRand (replicateM n $ getRandomR range)
```

Además se incluyen funciones útiles de la teoría de números analítica como lo son la función μ de Möbius, la función φ de Euler y la función λ de Liouville.

2.7. Álgebra Abstracta

Dentro del álgebra abstracta en el trabajo se utilizó clases y librerías de Haskell que implentan los grupos, semianillos, anillos, dominio euclideano y por supuesto campos. Haskell por defecto implementa las clases semigrupo y monoide

```
-- Un semigrupo es una estructura algebraica con una operación binaria interna asociativa
class Semigroup a where
    (<>) :: a -> a -> a


-- Un monoide es un semigrupo que ademas tiene un elemento identidad
class (Semigroup a) => Monoid a where
    mempty :: a
```

La restricción de clase indica que para poder instanciar un tipo de dato *a* como Monoide primero tiene que ser Semigrupo.

Haskell por defecto no implementa clases para grupo, pero se puede encontrar en la librería *groups*

(Doorn, 2013) y tiene la siguiente definición

```
-- Un grupo es un monoide en el que cada elemento es invertible
class Monoid m => Group m where
    invert :: m -> m
```

Por otro lado en la teoría de anillos, se utiliza la librería *semirings* (Chessai, 2018), en donde se implementan las estructuras de semianillo, anillo, dominio de maximo común divisor, dominio euclideano y campo.

Aunque se encontró un problema con la clase Field que la librería la implementa como una superclase de Euclidean, pero no resulta cierto que todo campo es un dominio euclídeo, por lo que es una falla de diseño.

Haskell no implementa clases como anillos por defecto, sin embargo, en la mayoría de ocaciones para tener operaciones de anillo, se instancia de clases como **Num** y **Fractional**, lo que vendría a representar generalizaciones de los enteros y racionales, pero este enfoque implica dejar métodos sin definir como **abs**, ya que no todos los anillos tienen una noción bien definida de valor absoluto.

Por otro lado también se implementa el algoritmo de rápida exponenciación binaria para campos primos.

Algoritmo de rápida exponenciación binaria

Basado en (Hoffstein et al., 2014) el algoritmo toma los siguientes pasos

1. **Paso 1.** Calcular la expansión binaria de A como

$$A = A_0 + A_1 \cdot 2 + A_2 \cdot 2^2 + \ldots A_r \cdot 2^r \text{ con } A_0, \ldots, A_r \in \{0,1\}.$$

2. **Paso 2.** Calcular las potencias $g^{2^i} \pmod N$ para $0 \leq i \leq r$ por cuadrados sucesivos

$$\begin{aligned}
a_0 &\equiv g && \pmod N \\
a_1 &\equiv a_0^2 \equiv g^2 && \pmod N \\
a_2 &\equiv a_1^2 \equiv g^{2^2} && \pmod N \\
a_3 &\equiv a_2^2 \equiv g^{2^3} && \pmod N \\
&\vdots \qquad \vdots && \vdots \\
a_r &\equiv a_{r-1}^2 \equiv g^{2^r} && \pmod N
\end{aligned}$$

cada término es el cuadrado del anterior y así se requieren r multiplicaciones.

3. **Paso 3.** Calcular $g^A \pmod N$ usando la fórmula

$$\begin{aligned}
g^A &\equiv g^{A_0 + A_1 \cdot 2 + A_2 \cdot 2^2 + \ldots A_r \cdot 2^r} \\
&\equiv g^{A_0} \cdot (g^2)^{A_1} \cdot (g^{2^2})^{A_2} \cdot (g^{2^3})^{A_3} \cdots (g^{2^r})^{A_r} \\
&\equiv a_0^{A_0} \cdot a_1^{A_1} \cdot a_2^{A_2} \cdot a_3^{A_3} \cdots a_r^{A_r} \pmod N.
\end{aligned}$$

Con la misma idea, se tiene la implementación en Haskell

```
powerMod :: Integer -> Integer -> Integer -> Integer
powerMod _ 0 _ = 1
powerMod g a m
  | even a = (x * x) `mod` m
  | otherwise = (g * x * x) `mod` m
  where
    x = powerMod g (a `div` 2) m
```

2.8. Geometría Algebraica

Dentro de la geometría algebraica se implementa la clase de elementos del grupo de curva elíptica, sobretodo las curvas elípticas de Weierstrass. Esta rama de la matemática tiene conceptos muy complejos, por ejemplo la definición de curva elíptica es la siguiente (Castillo, 2023, pág. 47)

Definición 2.20 (Curva elíptica)**.** *Una curva elíptica sobre un campo K es un par* $(E, \mathcal{O})$*, donde E es una curva proyectiva regular de género 1 y* $\mathcal{O} \in E(K)$*. Un isomorfismo entre dos curvas elípticas* $(E, \mathcal{O})$ *y* $(E', \mathcal{O}')$ *es un isomorfismo entre E y E′ que haga corresponder* $\mathcal{O}$ *con* $\mathcal{O}'$*.*

Y se demuestra, a partir del teorema de Riemann-Roch (enunciado en (Castillo, 2023, pág. 42)), que cada curva elíptica es isomoforma a una curva generalizada de Weierstrass.

Teorema 2.5. *Si $E(K)$ es una curva elíptica, existen funciones $x, y \in K(E)$ tales que la aplicación $\varphi : E \to P^2$ dada por $\varphi(P) = [x(P), y(P), 1]$ define un isomorfismo entre E y una curva plana C determinada por una ecuación de Weierstrass con coeficientes en K y $\varphi(\mathcal{O}) = [0,1,0]$. Dicha ecuación es única salvo una transformación afín de la forma*

$$X = u^2X' + r, \qquad Y = u^3Y' + su^2X' + t \qquad u, r, s, t \in K, \quad u \neq 0.$$

Más aún, cualquier K-isomorfismo entre dos curvas elípticas definidas por ecuaciones de Weierstrass (sobre K) está inducido por una transformación afín de este tipo.

La demostración se puede consultar en (Castillo, 2023, pág. 48).

CAPÍTULO III

3. MARCO METODOLÓGICO

El estudio llevado a cabo se enmarca en una metodología cualitativa con nivel descriptivo, centrada principalmente en la revisión exhaustiva de fuentes secundarias; el objetivo principal de esta fase fue establecer los conceptos base necesarios para comprender la criptografía de la curva elíptica, posteriormente, implementar los métodos necesarios para el funcionamiento de los algoritmos ECDH y ElGamal en en lenguaje de programación Haskell con el propósito de proveer una implementación sólida y robusta.

3.1. Descripción de enfoque, alcance, diseño, tipo, métodos, técnicas e instrumentos de investigación empleadas

El presente trabajo tuvo un enfoque cualitativo con el fin de profundizar en los fundamentos teóricos de la criptografía de la curva elíptica, buscando implementar de la manera más eficiente dos de los algoritmos más usados, ECDH y ElGamal, mediante una librería donde destaquen los métodos necesarios para este propósito, con una adecuada documentación para futuros usos.

Este enfoque se complementa con un alcance de investigación descriptivo que abarca los métodos de criptografía asimétrica, con énfasis en la criptografía de la curva elíptica y en particular en los algoritmos ECDH y ElGamal como principales métodos de cifrado, y se enfoca en la implementación de estos para proveer de una sólida librería disponible en Hackage (https://hackage.haskell.org/).

Este estudio se ha llevado a cabo bajo un diseño de tipo documental, donde se recopiló información existe mediante una revisión selectiva de documentos relacionados con la criptografía de la curva elíptica y los desafíos de su implementación. Para esta implementación se ha seleccionado el lenguaje de programación funcional Haskell, ya que que tiene muchas ventajas para la criptografía, ya que ofrece un alto nivel de abstracción, seguridad de tipos, expresividad y rendimiento

CAPÍTULO IV

4. MARCO DE ANÁLISIS E INTERPRETACIÓN DE RESULTADOS

4.1. Procesamiento, análisis e interpretación de los resultados

Se obtuvo una librería en Haskell con los siguientes módulos

1. Math.NumberTheory
2. Math.Algebra.GaloisFields
3. Math.EllipticCurve.Curve
4. Math.Criptography

4.1.1. *Teoría de números.*

En el módulo Math.NumberTheory se incluyen funciones para las pruebas de primalidad de números, factorización prima, rápida exponenciación para los enteros módulo p, funciones aritméticas como la función de Mobius, Lioville y Euler.

4.1.2. *Campo finito.*

En el módulo Math.Algebra se incluyen las clases de Campo de Galois, Campos Binarios y métodos para construirlas basándose en los polinomios de Conway.

4.1.3. *Curvas elípticas.*

En el módulo Math.AlgebraicGeometry se incluyen las clases de Curvas Elípticas, Curvas sobre campos binarios y los sistemas de coordenas afín y proyectivo con sus respectivos métodos e instancias.

4.1.4. *Criptografía.*

En este módulo se incluye el intercambio de claves de Diffie-Hellman y el cifrado ElGamal.

4.2. Discusión

El resultado obtenido es una librería de la criptografía de curva elíptica en donde se detallan en forma de clases de datos, la manera de implementar de forma rápida y sencilla la aritmética de campos finitos y curvas elípticas, algoritmos de teoría de números y álgebra necesarios, métodos para codificar un mensaje y representarlo como un punto de $\mathbb{F}_q^2$, además de los métodos de Diffie-Hellman y ElGamal.

La librería se construyó en la versión de GHC 9.6.3 y en el estándar de lenguaje GHC2021 con las versiones de los paquetes que eran compatibles con este versión (LTS 22.6) gestionadas mediante la herramienta de proyectos Stack (https://docs.haskellstack.org/en/stable/) (instalada mediante la utilidad de línea de comandos GHCup (https://www.haskell.org/ghcup/), bajo el sistema operativo Fedora 38 (https://fedoraproject.org/es/)), el editor de código Visual Studio Code con la extensión defecto para Haskell (https://marketplace.visualstudio.com/items?itemName=haskell.haskell) y el gestor de paquetes Stackage (https://www.stackage.org).

En Hackage, a la fecha de este trabajo, existen varias librerías que implementan algunas de estos tópicos, entre las más importantes

- finite-field (Sakai, 2021). Soporta aritmética sobre campos finitos, pero solo primos y al momento de tratar de importarla en GHC 9.4.8 tenía incompatibilidades de versión.
- finite-fields (Komuves, 2023). Incorpora aritmética de campos finitos en general con el uso de polinomios de Conway y tabla de logaritmo de Zech con uso de FFI (importaciones foráneas) en C, pero solo soportaba primos pequeños y no incluía la aritmética de curva elíptica. De este proyecto se basó el módulo de teoría de números, con algunos cambios en la documentación y cambio de funciones seguras, sin tener que usar errores.
- galois-fields (Diehl, 2021). Esta librería incorpora aritmética de campos finitos e incorpora las clases de una forma bastante elegante y el trabajo se baso fuertemente en esta librería con algunas modificaciones en documentación y correcciones de métodos y de versiones de librería, ya que la original no podía importarse por errores en versiones de las librerías de las cuales depende.
- elliptic-curve (Inc, 2019). Esta librería es del mismo autor que galois-fields y de hecho esta incluida en sus dependencias, pero tenía los mismos inconvenientes de importación. El proyecto también se baso fuertemente en esta librería con modificaciones en documentación y cambio de

definición de suma de curva elíptica sobre un campo binario, además se añadió el método de ElGamal.

- ecccrypto (Fourné, 2023). Esta librería implementa el intercambio de clave de Diffie-Hellman y el método ECDSA (Elliptic Curve Digital Signature Algorithm) que no se uso en el trabajo, debido a que no era muy explícito en la construcción de campos finitos.

CAPÍTULO V

5. CONCLUSIONES Y RECOMENDACIONES

5.1. Conclusiones

1. Los fundamentos matemáticos para comprender la criptografía basada en curvas elípticas ha sido desafiante debido a la cantidad de ramas involucradas y diversa cantidad de referencia con varios enfoques, tanto a nivel matemático como a nivel informático. El tema del cifrado basado en curvas elípticas aún tiene mucha investigación que encontrarse, y más aún las curvas elípticas en la investigación matemática.

2. El lenguaje de programación funcional Haskell es un lenguaje de una curva de aprendizaje alta, pero una vez comprendidos sus fundamentos, como el cálculo lambda, la teoría de tipos y la teoría de categorías, se puede desarrollar con gran versatilidad implementaciones de estructuras matemáticas y criptografía. Sin embargo, no es necesario una comprensión profunda en teoría de categorías, al menos no si no se busca investigar sobre avances del lenguaje y el compilador.

3. Los métodos de cifrado ha sido una tarea vasta, sobretodo por la implementación previa de la aritmética de curvas elípticas y la codificación de los mensajes. Debido a su naturaleza fue más fácil implementar el intercambio de claves de Diffie-Hellman que el cifrado de ElGamal.

4. La redacción de la documentación fue bastante cómoda con la herramienta Haddock, aunque se intento portear a latex, debido a incompatibilidades se opto por un documento nativo y con ayuda de lhs2tex.

5. La publicación en línea de la librería se hizo en un repositorio en GitHub, ya que hubo complicaciones al intentar subirse a Hackage, con las últimas versiones del lenguaje y librerías. De esta forma se tiene una librería sólida que fomente futuras investigaciones.

Y en general, el criptosistema basado en curvas elípticas es un tema bastante interesante y de actual investigación tanto por sus aplicaciones prácticas en criptografía, como en su interés teórico en conjeturas no resueltas. Su implementación en el lenguaje Haskell es bastante descriptiva e instructiva debido a la expresividad del lenguaje, sin embargo aún tiene algunas deficiencias sobretodo para su uso industrial, por lo que aún se puede mejorar la librería ECC

5.2. Recomendaciones

Como recomendaciones para la mejora de la librería en un futuro se propone lo siguiente:

1. Para el módulo de teoría de números se recomienda
   - Analizar e implementar métodos de cribas para tener una base para tener y generar números primos de forma más eficiente. También podría conectarse con una base de datos externa con primos precalculados.
   - Implementar métodos de factorización prima más eficientes y modernos, por ejemplo la factorización basada en curvas elípticas de Lenstra (Ellis, 1993, pág. 160-165).
   - Implementar métodos de comprobación prima como el test AKS (Hoffstein et al., 2014, pág. 137).
2. Para el módulo de campos finitos se recomienda:
   - Implementar la tabla de polinomios de Conway en el lenguaje Rust para mayor velocidad.
3. Para el módulo de curva elíptica se recomienda:
   - Implementar las coordenadas proyectivas, jacobianas y de edwars para mejorar el rendimiento en la suma de puntos.
   - Implementar el uso del automorfismo de Frobeius para el doblaje de puntos.
4. En general para la librería implementar una clase aleatoria distinta a la librería Random, que sirva para generar elementos aleatorios criptográficamente seguros.

Si se busca migrar el proyecto a otro lenguaje, por ejemplo para su uso comercial, se recomienda el lenguaje cryptol (https://cryptol.net), ya que es un lenguaje funcional bastante parecido a Haskell, pero especializado para criptografía.

Y en general debido a que la librería es de uso ilustrativo, aún puede perfeccionarse en cuestiones de rendimiento con otras pruebas tanto de primalidad, como construcción de campos y tratamiento de las curvas elípticas. Se recomienda añadir nuevos algoritmos para mejorar el rendimiento.

BIBLIOGRAFÍA

1. **ALLEN, Christopher & MORONUKI, Julie**. *Haskell Programming from First Principles*. Lorepub LLC, 2016. ISBN 194538803X.

2. **CASTILLO, Carlos Ivorra**. *Curvas Elipticas* [En línea]. Universitat de València, 2023 [Consulta: 7 febrero 2024]. Disponible en: https://www.uv.es/ivorra/Libros/Elipticas.pdf.

3. **CHESSAI**. *semirings: two monoids as one, in holy haskimony* [En línea]. 2018. [Consulta: 22 enero 2024]. Disponible en: https://hackage.haskell.org/package/semirings.

4. **DIEHL, Stephen**. *galois-field: Galois field library* [En línea]. 2021. [Consulta: 8 febrero 2024]. Disponible en: https://hackage.haskell.org/package/galois-field.

5. **DIFFIE, W. & HELLMAN, M.** "New directions in cryptography". *IEEE Transactions on Information Theory* [En línea]. 1976, vol. 22 (6), págs. 644-654 [Consulta: 11 enero 2024]. ISSN 0018-9448. Disponible en: 10.1109/tit.1976.1055638.

6. **DOORN, Nathan van**. *groups: Groups* [En línea]. 2013. [Consulta: 22 enero 2024]. Disponible en: https://hackage.haskell.org/package/groups.

7. **ELLIS, Graham**. *Rings and Fields*. Oxford University Press, 1993. ISBN 9780198534556.

8. **FOURNÉ, Marcel**. *eccrypto: Elliptic Curve Cryptography for Haskell* [En línea]. 2023. [Consulta: 8 febrero 2024]. Disponible en: https://hackage.haskell.org/package/eccrypto.

9. **HANKERSON, Darrel; et al.** *Guide to Elliptic Curve Cryptography* [En línea]. 1.ª ed. Springer, 2004 [Consulta: 29 enero 2024]. ISBN 978-0387952734. Disponible en: 10.1007/b97644.

10. *Haskell and mathematics* [En línea]. 2021. [Consulta: 1 enero 2024]. Disponible en: https://wiki.haskell.org/Haskell_and_mathematics.

11. **HOFFSTEIN, Jeffrey; et al.** *An Introduction to Mathematical Cryptography*. 2.ª ed. Springer, 2014.

12. **INC, Adjoint**. *elliptic-curve: Elliptic curve library* [En línea]. 2019. [Consulta: 8 febrero 2024]. Disponible en: https://hackage.haskell.org/package/elliptic-curve.

13. **JANCAR, Jan & SEDLACEK, Vladimir**. *Standard curve database* [En línea]. 2020. [Consulta: 25 enero 2024]. Disponible en: https://neuromancer.sk/std/.

14. **JOHNSONBAUGH, Richard**. *Discrete Mathematics*. 8.ª ed. Pearson, 2023. ISBN 9780137848577.

15. **KOBLITZ, Neal**. *Algebraic Aspects of Cryptography*. Vol. 3. New York: Springer, 1998. Algorithms and Computation in Mathematics.

16. **KOBLITZ, Neal**. "Elliptic curve cryptosystems". *Mathematics of Computation* [En línea]. 1987, vol. 48, págs. 203-209 [Consulta: 7 febrero 2024]. Disponible en: https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf.

17. **KOMUVES, Balazs**. *finite-fields: Arithmetic in finite fields* [En línea]. 2023. [Consulta: 8 febrero 2024]. Disponible en: https://hackage.haskell.org/package/finite-fields.

18. **KRIVINE, Jean Louis**. *Lambda-calculus, Types and Models*. Trad. por CORI, René. Ellis Horwood, 1993. ISBN 0130624071.

19. **LÜBECK, Frank**. *Conway Polynomials for finite fields* [En línea]. 2021. [Consulta: 12 enero 2024]. Disponible en: http://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol/index.html?LANG=en.

20. **LUCENA LÓPEZ, Manuel José**. *Criptografía y seguridad en computadores* [en línea]. 5-2.0.1. Creative Commons, 2022 [Consulta: 12 diciembre 2023]. Disponible en: https://criptografiayseguridad.blogspot.com/p/criptografia-y-seguridad-en.html.

21. **MAGNER, Ricky**. *Finite Fields* [En línea]. 2020. [Consulta: 11 enero 2024]. Disponible en: http://math.bu.edu/people/rmagner/extras/FiniteFields.pdf.

22. **MILLER, Victor S.** Use of Elliptic Curves in Cryptography. En: WILLIAMS, Hugh C. (ed.). *Advances in Cryptology — CRYPTO '85 Proceedings* [En línea]. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, págs. 417-426 [Consulta: 7 febrero 2024]. ISBN 978-3-540-39799-1. Disponible en: 10.1007/3-540-39799-X_31.

23. **MULLEN, Gary L. & MUMMERT, Carl**. *Finite Fields and Applications*. Vol. 41. American Mathematical Society, 2007. Student Mathematical Library.

24. **O'SULLIVAN, Bryan; et al.** *Real World Haskell* [En línea]. 1st. O'Reilly Media, Inc., 2008 [Consulta: 1 enero 2024]. ISBN 0596514980. Disponible en: https://book.realworldhaskell.org/read/.

25. **PEDRAZA LÓPEZ, Diego**. *Teoría de Categorías y programación funcional* [online]. 2018. [Consulta: 17 febrero 2024]. Disponible en: https://idus.us.es/bitstream/handle/11441/77572/Pedraza%20L%C3%B3pez%20Diego%20TFG.pdf. Tesis de Grado. Universidad de Sevilla.

26. **RESEARCH, Certicom**. *Standards for Efficient Cryptography: SEC 2: Recommended Elliptic Curve Domain Parameters* [En línea]. Mississauga, Ontario, Canada: Standards for Efficient Cryptography Group, 2000 [Consulta: 3 agosto 2024]. Disponible en: https://www.secg.org/SEC2-Ver-1.0.pdf.

27. **ROBLING DENNING, Dorothy Elizabeth**. *Cryptography and Data Security* [En línea]. USA: Addison-Wesley Longman Publishing Co., Inc., 1982 [Consulta: 1 enero 2024]. ISBN 0201101505. Disponible en: https://dl.acm.org/doi/pdf/10.5555/539308.

28. **SAKAI, Masahiro**. *finite-field: Finite Fields* [En línea]. 2021. [Consulta: 8 febrero 2024]. Disponible en: https://hackage.haskell.org/package/finite-field.

29. **SILVERMAN, Joseph H.** *The Arithmetic of Elliptic Curves*. Vol. 106 [online]. Springer, 2009 [Consulta: 8 febrero 2024]. Graduate Texts in Mathematics. Disponible en: 10.1007/978-0-387-09494-6.

30. **SILVERMAN, Joseph H. & TATE, John T.** *Rational Points on Elliptic Curves* [En línea]. Springer, 2015 [Consulta: 7 febrero 2024]. Undergraduate Texts in Mathematics. Disponible en: 10.1007/978-3-319-18588-0.

31. **SKINNER, Rebecca**. *Effective Haskell: Solving Real-World Problems with Strongly Typed Functional Programming*. Pragmatic Programmers, LLC, 2023.

32. **WASHINGTON, Lawrence C.** *Elliptic Curves: Number Theory and Cryptography*. 2.ª ed. Chapman and Hall/CRC, 2008. Disponible en: 10.1201/9781420071474.

33. **YANG, Song Y.** *Computational Number Theory and Modern Cryptography* [En línea]. John Wiley and Sons, 2012 [Consulta: 8 febrero 2024]. Disponible en: 10.1002/9781118188606.fmatter.

ANEXOS

LIBRERÍA ECC

La librería ECC se encuentra en el siguiente repositorio de Github:

https://github.com/BigDanChess/ECC

DOCUMENTACIÓN ECC

Implementación de la Criptografía de Curva Elíptica en el Lenguaje Haskell

Daniel Alejandro Reinoso

Escuela
Superior Politécnica
de Chimborazo

Índice general

1 Teoría de números

1.1 Primes

En el módulo Math.NumberTheroy.Primes se incluyen funciones para trabajar sobre funciones de teoría de números y verificación de números primos.
La función logaritmo entero es una función que calcula el entero más grande k tal que 2^k es menor o igual a n basandose en la función $shiftR$ de Data.Bits que sirve para desplazar los bits a la derecha de un número (por ejemplo $5 = 110_2$ y $shiftR\ 5\ 2$ se evalúa a $1 = 001_2$).

```
integerLog2 :: Integer → Integer
integerLog2 n
  | n < 0 = error "Entero negativo"
  | otherwise = go n
  where
    go 0 = -1
    go s = 1 + go (shiftR s 1)
```

Primero, se determina la representación binaria de n. es decir $n = \sum_{k=1}^{\infty} a_k 2^k$ con $a_k = 0$ para algún N con $k \geqslant N$. Entonces $\lfloor \log_2(n) \rfloor = k$, donde

$$n = (a_k a_{k-1} \dots a_0)_2$$

k es el mayor entero tal que $a_k \neq 0$. Ejemplo con $n = 7$:

$$
\begin{aligned}
& integerLog2\ (7) \\
\equiv\ & \quad \{\text{ definición }\} \\
& 1 + integerLog2\ (shiftR\ 7\ 1) \\
\equiv\ & \quad \{\text{ desplazamiento de un bit 7=111 a 011=5 }\} \\
& 1 + integerLog2\ 5 \\
\equiv\ & \quad \{\text{ definición }\} \\
& 1 + 1 + integerLog2\ (shiftR\ 5\ 1) \\
\equiv\ & \quad \{\text{ desplazamiento de un bit 5=011 a 001=1 }\} \\
& 1 + 1 + 1 + integerLog2\ (shiftR\ 1\ 1) \\
\equiv\ & \quad \{\text{ definición }\} \\
& 1 + 1 + 1 + integerLog2\ 0 \\
\equiv\ & \quad \{\text{ desplazamiento de un bit 1=001 a 0 }\} \\
& 1 + 1 + 1 + (-1) = 2
\end{aligned}
$$

Y la función Logaritmo Techo que es el entero más pequeño k tal que 2^k es mayor o igual a n. Equivalentemente $\lceil \log_2(n) \rceil = k$.

```
ceilingLog2 :: Integer → Integer
ceilingLog2 n
  | n < 0 = error "Entero negativo"
  | otherwise = go n
  where
    go 0 = 0
    go s = 1 + go (shiftR s 1)
```

Se deduce de la misma manera. Y se cumple

Nota 1.1

$integerLog2 = ceilingLog2 + 1$

La función $integerSquareRoot$ calcula la parte entera de la raíz cuadrada de un número natural sin aritmética de punto flotante, y se basa en el método de Newton con siguiente esquema de iteración:

$$x_{k+1} = \left\lfloor \frac{1}{2}\left(x_k + \left\lfloor \frac{n}{x_k} \right\rfloor\right)\right\rfloor, \quad k \geqslant 0, x_0 > 0, x_0 \in \mathbb{Z}.$$

y la condición de parada $\mid x_{k+1} - x_k \mid < 1$ asegura la convergencia hacia $\lfloor\sqrt{n}\rfloor$. Además ya que $\sqrt{n} = 2^{\frac{1}{2}\log_2(n)}$, es razonable elegir $x_0 = 2^{\frac{1}{2}\lfloor\log_2(n)\rfloor}$.

```
integerSquareRoot :: Integer → Integer
integerSquareRoot n
  | n < 0 = error "Entero negativo"
  | n < 2 = n
  | otherwise = go ∘ (2^) ∘ (`div`2) $ integerLog2 n
  where
    go n0 = if |n2 − n1| < 1 then n2 else go n2
      where
        n1 = n0
        a = div n n0
        n2 = div (n0 + a) 2
```

isPerfectSquare determina si un entero n es un cuadrado perfecto basandose en su residuo

```
isPerfectSquare :: Integer → Bool
isPerfectSquare n = (≡ 0) ∘ mod n $ integerSquareRoot n
```

Nota 1.2

En Haskell (y en la mayoría de lenguajes de programación), las funciones div y mod devuelven los enteros *cociente* q y *residuo* r de la división entera entre a y b (resultado conocido como el *algoritmo de la división*). No debe confundirse con la notación $a \equiv b \pmod{n}$ utilizada para la congruencia modular.

Nota 1.3

En el código el signo $=$ se usa para la definición de la función, y $\equiv$ se usa para la igualdad de dos expresiones, no confundir con el uso de $\equiv$ en la aritmética modular.

$a \mid b$ determina si un entero a divide a b (los signos ! significan que esos argumentos deben ser completamente evaluados).

$$\cdot \mid \cdot :: Integer \to Integer \to Bool$$
$$\cdot \mid \cdot \; !\, d \,!\, n = mod\ n\ d \equiv 0$$

1.1.1 Rápida exponenciación binaria

Debido a que a menudo se realizan operaciones en $\mathbb{F}_p$ como $g^a \equiv t \pmod{p}$, entonces realizar primero g^a y luego reducir módulo p se vuelve imposible para a grande, así que utilizó la estrategia de la rápida exponenciación binaria, que según si a es par o impar, calculará recursivamente la potencia, reduciendo módulo m en cada paso, y se describe en el siguiente código

$$powerMod :: Integer \to Integer \to Integer \to Integer$$
$$powerMod\ \_\ 0\ \_ = 1$$
$$powerMod\ g\ a\ m$$
$$\quad \mid even\ a = (x * x)\ `mod`\ m$$
$$\quad \mid otherwise = (g * x * x)\ `mod`\ m$$
$$\quad \textbf{where}$$
$$\qquad x = powerMod\ g\ (a\ `div`\ 2)\ m$$

1.1.2 Lista de números primos

$primesEr$ determina una lista infinita de números primos determinados por la criba de Eratóstenes.

$$primesEr :: [Integer]$$
$$primesEr = filterPrime\ [2..]$$

Número	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Primo	P	P	C	P	C	P	C	C	C	P

Cuadro 1.1: Criba de Eratóstenes para los números del 2 al 11. (P: Primo, C: Compuesto)

where

$$filterPrime\ (p : xs) =$$
$$p : filterPrime\ [x \mid x \leftarrow xs, mod\ x\ p \neq 0]$$

1.1.3 Comprobación de primalidad

$isPrimeTrialDivision$ es una función para determinar si un entero es un primo, basandose en el siguiente resultado

Teorema 1.1

Un entero positivo n mayor que 1 es compuesto si y solo si n tiene un divisor d tal que satisface $2 \leqslant d \leqslant \sqrt{n}$.

Demostración. Por un lado, si n es compuesto, entonces tiene un divisor positivo $2 \leqslant d' < n$. Ahora si $d' \leqslant \sqrt{n}$ ya estaría probado, si no entonces sucede $d' > \sqrt{n}$, luego ya que d' divide a n, existe un entero q tal que $n = d'q$ y así q es divisor de n y $q \leqslant \sqrt{n}$. Si esto fuera falso entonces tendríamos $q > \sqrt{n}$ y multiplicando con d'

$$n = d'q > \sqrt{n}\sqrt{n} = n.$$

lo que es una contradicción. Por lo tanto se tiene que si n es compuesto debe existir un divisor entre $2 \leqslant d \leqslant \sqrt{n}$. La otra implicación se sigue de la definición de número compuesto. □

Es decir, para verificar si un número n es primo, se debe verificar que ningún número entre 2 y $\sqrt{n}$ lo divida, y resulta en el siguiente algoritmo.

$$isPrimeTrialDivision :: Integer \rightarrow Bool$$
$$isPrimeTrialDivision\ n = all\ (\lambda p \rightarrow \neg\ (p \mid n))\ \$\ ps$$

where

$ps = [(2) \ldots (nsqrt)]$

$nsqrt = integerSquareRoot\ n$

Debido a que conforme n es un número grande, este test se vuelve ineficiente, asi que se implementa el test de Miller Rabin, para esto primero se tiene una proposición y una definición.

Proposición 1.1

Sea p un primo impar y se escribe

$$p - 1 = 2^k q \qquad \text{con } q \text{ impar.}$$

Sea a cualquier número que no es divisible por p. Entonces uno de las siguientes condiciones es cierta:

1. a_q es congruente a 1 módulo p.
2. Uno de los $a^q, a^{2q}, a^{4q}, \ldots, a^{2^{k-1}q}, a^{2^k q}$ es congruente a -1 módulo p.

La demostración se encuentra en [4].

Definición 1.1: Testigo de Miller-Rabin

Sea n un número impar y se escribe $n - 1 = 2^k q$ con q impar. Un entero a que satisface $\gcd(a, n) = 1$, es llamado un *testigo de Miller-Rabin* para (la *composidad* de) n si alguna de las siguientes condiciones se cumple:

1. $a^q \not\equiv \pmod{n}$
2. $a^{2^i q} \not\equiv -1 \pmod{n}$ para todo $i = 0, 1, 2 \ldots, k - 1$.

De donde se sigue que si existe un testigo de Miller-Rabin para n, entonces n es compuesto. Este test es más efectivo debido a que para un número compuesto, al menos tiene el $75\,\%$ de testigos de Miller-Rabin [4].

Con $millerRabinWitness$ se determina si un entero a es un testigo de Miller-Rabin para la composidad de n, si lo es, devuelve $True$, en otro caso $False$. Mientras que $decompone$ permite expresar $n-1=2^k q$, con q impar.

$$
\begin{aligned}
&millerRabinWitness :: Integer \to Integer \to Bool \\
&millerRabinWitness\ n\ a \\
&\quad \mid even\ n = False \\
&\quad \mid a \leqslant 1 \wedge a \geqslant n-1 = False \\
&\quad \mid gcd\ a\ n \neq 1 = False \\
&\quad \mid powerMod\ a\ q\ n \neq 1 \wedge \\
&\qquad all\ (\lambda x \to x \neq (-1))\ [powerMod\ a\ (2*i*q)\ n \mid i \leftarrow [0,1\,..\,(k-1)]] = True \\
&\quad \mid otherwise = False \\
&\qquad \textbf{where}\ k = decompone\ (n-1) \\
&\qquad\quad q = (n-1)\ `div`\ k
\end{aligned}
$$

De, donde para el test de Miller Rabin para la primalidad de n, se seleccionan r enteros aleatorios entre 2 y $n-1$, se verifica si alguno de ellos es testigo, y si no, el número es primo (probablemente).

$$
\begin{aligned}
&millerRabinTest :: Integer \to Bool \\
&millerRabinTest\ n = \neg \circ any\ (millerRabinWitness\ n)\ \$\ fst\ (randomIntegerList\ (fromIntegral\ (n*
\end{aligned}
$$

1.2 Funciones aritméticas

Definición 1.2: Función aritmética

Una función de valores reales o complejos definida en los enteros positivos se llama una función aritmética o teórica de números.

Definición 1.3: Función μ de Moebius

La función μ de Moebius se define como

$$\mu(1) = 1;$$

Si $n > 1$, se escribe $n = p_1^{\alpha_1} \dots p_k^{\alpha_k}$. Entonces

$$\mu(n) = (-1)^k \text{ si } a_1 = a_2 = \dots = a_k = 1,$$
$$\mu(n) = 0 \text{ en otro caso}$$

Se implementa con *moebiusMu*

```
moebiusMu :: Integer → Integer
moebiusMu n
  | n ⩽ 0 = error "Entero negativo"
  | n ≡ 1 = 1
  | otherwise = if all (≡ 1) [snd x | x ← ps] then (-1)^k else 0
  where
    ps = naiveFactorization' n
    k = length ps
```

Ejemplo:

n	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{\mu(n)}$	1	-1	-1	0	-1	1	-1	0	0	1

Cuadro 1.2: Primeros valores de la función de Möbius ($\mu(n)$) para $n \leqslant 10$.

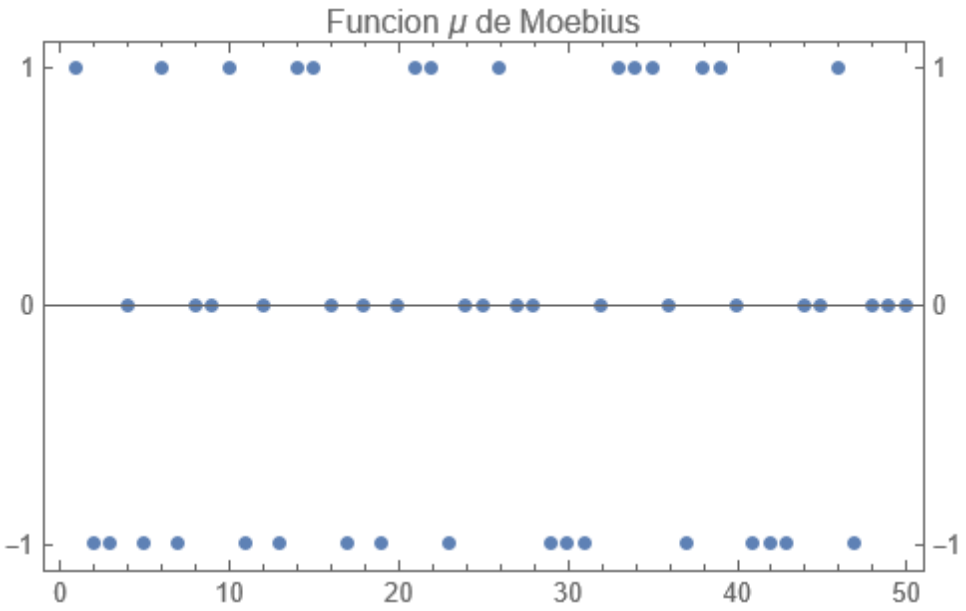


Figura 1.1: Función μ de Moebius para los primeros 50 enteros

Definición 1.4: Función totiente de Euler

La función φ de Euler (o función totiente), calcula el número de coprimos menores a n. Así

$$\varphi(n) = \sum_{k=1}^{n}{}' 1,$$

donde $'$ indica que la suma se extiende sobre aquellos k que son coprimos con n.

Ejemplo:

n	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{\phi(n)}$	1	1	2	2	4	2	6	4	6	4

Cuadro 1.3: Primeros valores de la función totiente de Euler ($\phi(n)$) para $n \leqslant 10$.

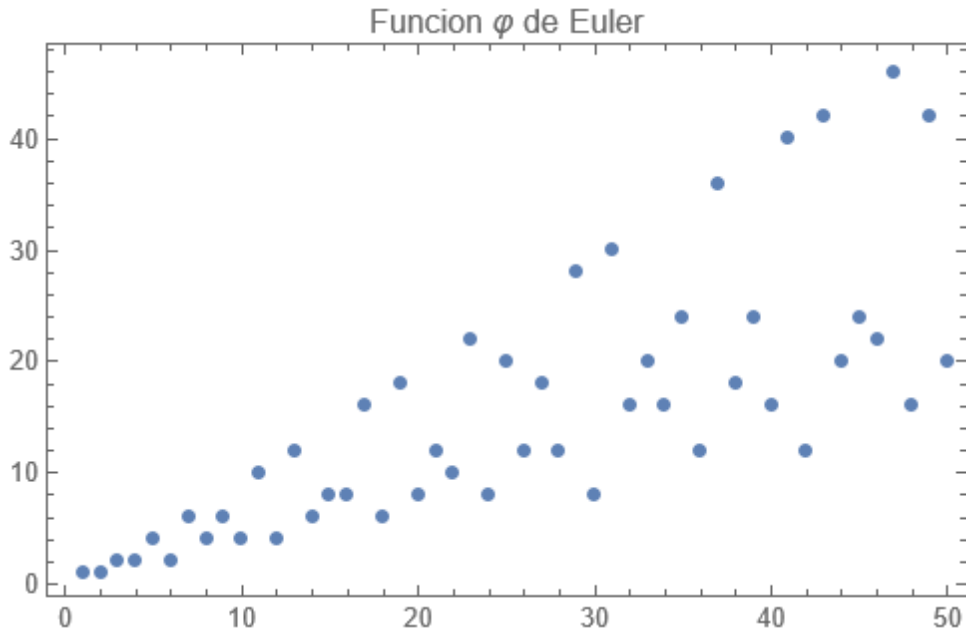


Figura 1.2: Función φ de Euler para los primeros 50 enteros

Se implementa con $eulerTotientNavie$, simplemente contando el número de elementos que son coprimos con n.

$$
\begin{aligned}
&eulerTotientNavie :: Integer \to Int \\
&eulerTotientNavie\ n \\
&\quad \mid n \leqslant 0 = error\ \texttt{"Entero negativo"} \\
&\quad \mid otherwise = length\ [s \mid s \leftarrow [1\,..\,n], gcd\ s\ n \equiv 1]
\end{aligned}
$$

Utilizando el siguiente teorema, se tiene la versión $eulerTotient2$

Teorema 1.2

Si $n \geqslant 1$, se tiene

$$\varphi(n) = \sum_{d|n} \mu(d)\frac{n}{d}.$$

La demostración se encuentra en [1].

$$
\begin{aligned}
&eulerTotient2 :: Integer \to Integer \\
&eulerTotient2\ n \\
&\quad \mid n \leqslant 0 = error\ \texttt{"Entero negativo"} \\
&\quad \mid otherwise = sum\ \$\ map\ (\lambda d \to moebiusMu\ d * (n\ `div`\ d))\ (divisors\ n)
\end{aligned}
$$

Aqui *divisors* es una función que determina los divisores de n.

La tercera versión se basa en el siguiente resultado

Teorema 1.3

Para $n \geqslant 1$, tenemos

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right).$$

donde p denota un primo divisor de n.

La demostración se encuentra en [1, pág. 27].

```
eulerTotient3 :: Integer → Integer
eulerTotient3 n
  | n ⩽ 0 = error "Entero no positivo"
  | otherwise = n * num `div` de
  where
    primes = [fst t | t ← naiveFactorization' n]
    de = product primes
    num = product $ (λx → x - 1) <$> primes
```

Definición 1.5: Convolución aritmética

Si f y g son dos funciones aritméticas definimos su producto de Dirichlet (o convolución de Dirichlet) como la función aritmética h defina como

$$h(n) = \sum_{d|n} f(d) g\left(\frac{n}{d}\right).$$

y se denota por $h = f * g$.

El uso de funciones de orden superior, nos permite expresar *convolutionDirichlet*

```
convolutionDirichlet :: (Integer → Integer) → (Integer → Integer) → Integer → Integer
convolutionDirichlet f g n
  | n ⩽ 0 = error "Entero no positivo"
  | otherwise = sum [f s * g (n `div` s) | s ← divisors n]
```

Corolario 1.1

$\varphi = \mu * N$, donde N denota la identidad $N(n) = n$.

Definición 1.6: Función λ de Lioville

Se define $\lambda(1) = 1$, y si $n = p_1^{a_1} \dots p_k^{a_k}$ se define

$$\lambda(n) = (-1)^{a_1+a_2+\cdots+a_k}.$$

Cuadro 1.4: Valores de la función de Liouville ($\lambda(n)$)

n	1	2	3	4	5	6	7	8	9	10
$\lambda(n)$	1	-1	-1	0	-1	1	-1	0	0	1

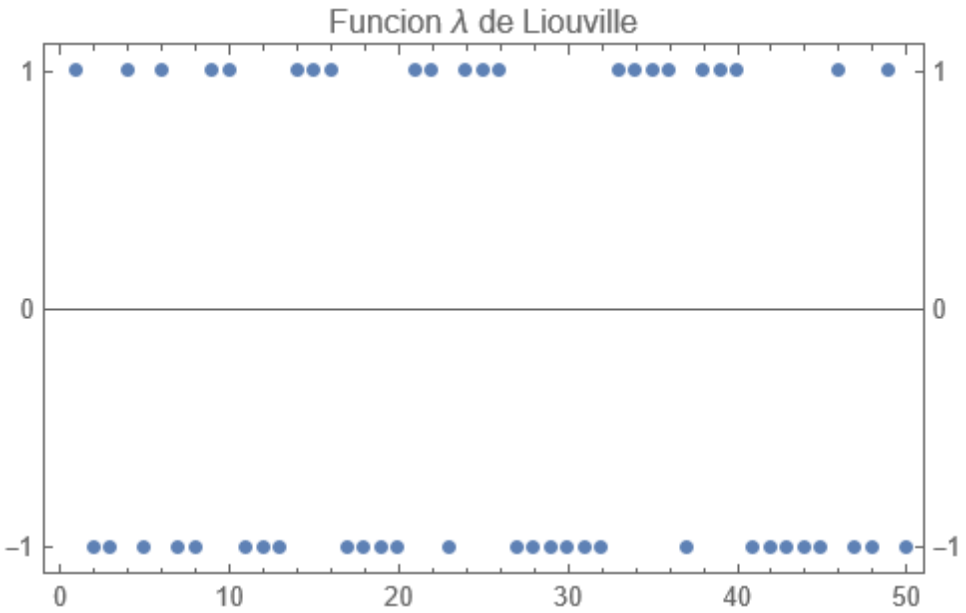


Figura 1.3: Función λ de Liouville para los primeros 50 enteros

Se implementa con *liouville*

$$liouville :: Integer \to Integer$$

$$liouville\ n$$

$$\mid n \leqslant 0 = error\ \texttt{"Entero no positivo"}$$

$$\mid n \equiv 1 = 1$$

$$
\begin{aligned}
&\mid otherwise = (-1)\hat{}\, k \\
&\quad \textbf{where } k = sum\ [snd\ x \mid x \leftarrow ps] \\
&\qquad ps = naiveFactorization'\ n
\end{aligned}
$$

1.3 Aritmética Modular

Para la aritmética modular se utiliza la librería Mod.

2 Álgebra

2.1 Teoría de grupos

Haskell por defecto incluye dos estructuras algebraicas, el semigrupo y el monoide.

Definición 2.1: Semigrupo

Un **semigrupo** es una estructura algebraica $(G, \bullet)$, donde G es un conjunto y $\bullet : G \times G \to G$ es una operación binaria interna asociativa. Si además se cumple que $a \bullet b = b \bullet a$ se dice que es un **semigrupo conmutativo**.

Se implementa con la clase *Semigroup* del módulo *Data.Semigruop*

```
class Semigroup a where
  (<>) :: a → a → a
```

Definición 2.2: Monoide

Un **monoide** es un semigrupo $(G, \bullet)$ que posee un elemento identidad e (es decir satisface $e \bullet g = g \bullet e = g$ para todo $g \in G$).

Se implenta con la clase *Monoid* del módulo *Data.Monoid*

$$\textbf{class } Semigroup\ a \Rightarrow Monoid\ a \textbf{ where}$$
$$mempty :: a$$

La restricción de clase indica que para que un tipo de dato a pueda tener una instancia de monoide primero tiene que tener una instancia de semigrupo.
Debido a que la clase para grupo no tiene una implementación por defecto, se usó la que está establecida en la librería $group - theoy$

Definición 2.3: Grupo

Un **grupo** es un monoide $(G, \bullet)$ en el que cada elemento es invertible, es decir para todo $g \in G$, existe un elemento denotado como g^{-1} tal que $g \bullet g^{-1} = g^{-1} \bullet g = e$. Si el grupo es conmutativo, se le suele llamar **grupo abeliano**

Se implenta con la clase $Group$ en el módulo Data.Group

$$\textbf{class } Monoid\ m \Rightarrow Group\ m \textbf{ where}$$
$$invert :: m \rightarrow m$$

Donde para que un dato tenga una instancia de grupo, primero tiene que tener una instancia de monoide y el método $invert$ halla el inverso de un elemento g.

2.2 Teoría de anillos

Las estructuras algebraicas de los anillos, no se implementan por defecto en Haskell (aunque se tienen la clase Num y $Fractional$ que serían como se tratan, pero no es lo mejor conceptualmente), pero se puede usar librerias como semirings que implementan esas clases.

Definición 2.4: Semianillo

Sea un conjunto A y dos operaciones binarias internas denotas por $\oplus$ y $\otimes$, la estructura algebráica $(A, \oplus, \otimes)$, se dice que tiene estrucutra de **semianillo** si:

- $(A, \oplus)$ es un monoide conmutativo con elemento identidad denominado 0.
- $(A, \otimes)$ es un monoide con elemento identidad denominado 1.
- $\oplus$ es distributivo respecto a $\otimes$, es decir:
  1. $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$ (distribución por la izquierda)
  2. $(a \oplus b) \otimes c = a \otimes c \oplus b \otimes c$ (distribución por la derecha).

Nota 2.1

Algunos autores definen al semianillo con $(A, \otimes)$ y $(A, \oplus)$ siendo solo semigrupos, pero aqui se prefiere como monoides.

Con implementación:

```
class Semiring a where
  plus :: a → a → a
  zero :: a
  times :: a → a → a
  one  :: a
```

En esta clase *plus* representa a la operación suma con su respectiva identidad *zero* y *times* a la operación producto con su identidad *one*.

Definición 2.5: Anillo

Un **Anillo** es un semianillo, en el que cada elemento tiene un inverso aditivo.

```
class Ring a where
  negate :: a → a
```

Aquí $negate$ devuelve el inverso aditivo de un elemento a.

Definición 2.6: Dominio de máximo común divisor

Un **Dominio de máximo común divisor** es un dominio íntegro con la propiedad de que cualquiera dos elementos, estos tiene bien definida la operación de máximo común divisor.

$$\begin{aligned}&\textbf{class } Semiring\ a \Rightarrow GcdDomain\ a \textbf{ where}\\ &\quad gcd :: a \to a \to a\\ &\quad divide :: a \to a \to Maybe\ a\end{aligned}$$

gcd devuelve el máximo común divisor, mientras que a diferencia de un Dominio Euclideano puede que no este bien definida la división, de ahí el uso de $Maybe$.

Definición 2.7: Función Euclideana

Sea R un dominio íntegro. Una **función euclideana** en R es una función de $R/\{0\}$ a los enteros no negativos que satisface la siguiente propiedad:

- Si $a, b \in R$ con $b \neq 0$, entonces existen q y r en R tal que $a = bq + r$ y o bien sucede $r = 0$ o $f(r) < f(b)$.

Definición 2.8: Dominio Euclideano

Un **Dominio Euclideano** es un dominio íntegro que tiene una función euclideana.

Ya que todo dominio euclideano es un dominio de máximo común divisor entonces la implementación lo requiere

$$\begin{aligned}&\textbf{class } GcdDomain\ a \Rightarrow Euclidean\ a \textbf{ where}\\ &\quad quotRem :: a \to a \to (a, a)\\ &\quad degree :: a \to Natural\end{aligned}$$

Aquí $degree$ representa la función euclideana, mientras que $quotRem$ permite la división entera con cociente y residuo.

2.3 Teoría de campos

Definición 2.9: Campo

Un campo es un anillo conmutativo en el que cada elemento tiene definido un inverso multiplicativo.

Con implementación

$$\textbf{class}\ (Euclidean\ a, Ring\ a) \Rightarrow Field\ a$$

Nota 2.2

No todo campo es un dominio euclideano, aunque en la librería se implemente así, además falta el inverso multiplicativo. En la librería se soluciona con las clases Num y $Fractional$.

Definición 2.10: Campo de Galois

Un campo de Galois o campo finito es un campo que tiene un número finito de elementos y se denotan por $\mathbb{GF}(q)$ o $\mathbb{F}_q$, donde q es el orden del campo.

Debido al siguiente resultado, se puede caracterizar a los campos finitos.

Teorema 2.1

Un campo finito $\mathbb{F}$ contiene precisamente p^n elementos para algún número primo y un entero $n \geqslant 1$.

Además si dos campos primos tienen igual orden, entonces son isomorfos.
Para los campos finitos se tiene la clase de Campo de galois $\mathbb{F}(p^q)$ para p primo y q entero no negativo.

$$\begin{aligned}
&\textbf{class}\ (Field\ k, G.Group\ k, Fractional\ k, NFData\ k,\\
&\quad Ord\ k, Random\ k, Show\ k) \Rightarrow\\
&\quad GaloisField\ k\\
&\quad \textbf{where}\\
&\qquad char :: k \to Natural\\
&\qquad deg :: k \to Word\\
&\qquad frob :: k \to k\\
&\qquad order :: k \to Natural\\
&\qquad order = (\hat{}) \mathbin{<\$>} char \mathbin{<*>} deg
\end{aligned}$$

La última linea es la implementación por defecto del orden, que no hace más que calcular el orden de un campo, la estructura $(\hat{})\ \mathbin{<\$>}\ char \mathbin{<*>} deg$, utiliza un aplicative (functor monoidal) y un functor, que es una útil forma de expresar $\lambda k \to (char\ k)\hat{}(deg\ k)$.

2.3.1 Campo primo

Para $n = 1$, tenemos los campos $\mathbb{F}_p$ que no son más que los enteros módulo p, que como p es primo, forman un campo.
Para este tipo de campo se crea la clase $PrimeField$

$$\begin{aligned}
&\textbf{class}\ (GaloisField\ k) \Rightarrow PrimeField\ k\ \textbf{where}\\
&\quad fromP :: k \to Integer
\end{aligned}$$

Donde tiene la restricción $GaloisField\ k$, puesto que todo campo primo es un campo de Galois y $fromP$ se utiliza para representar el elemento de un campo como un entero.
Para construir este tipo de campo, se necesita un número primo y para sus elementos se crea este tipo de dato en Haskell.

$$\begin{aligned}
&\textbf{newtype}\ Prime :: Nat \to Type\ \textbf{where}\\
&\quad P :: (Mod\ p) \to Prime\ p\ \textbf{deriving}\ (Eq, Ord, Show, Generic, Num, Fractional, Euclidean, Field,
\end{aligned}$$

Donde $p{::}Nat$ representa el número primo en el constructor de tipos, mientras que en el constructor de datos P se tiene que proporcionar un elemento del tipo $Mod\ p$ que

representa los elementos del campo $\mathbb{F}_p$. Para construir un elemento del tipo $Mod\ p$ se necesito de un $Natural$ y quedará reducido módulo p. Más información en la librería Mod.

Debido a que en esta librería se dan diversas instancias para el tipo de dato $Mod\ p$ al encapsular con el constructor P se pueden derivar las instancias del nuevo tipo $Prime\ (p)$.

Ahora para que el tipo de dato $Prime\ \ p$ sea parte de la clase $GaloisField$ se proporciona su instancia (nótese que no es suficiente esta instancia debido a las restricciones de clase $GaloisField$, más adelante se definen las otras instancias)

```
instance (KnownNat p) ⇒ GaloisField (Prime p) where
  char :: Prime p → Natural
  char = natVal
   {-# INLINEABLE char #-}
  deg :: Prime p → Word
  deg = const 1
   {-# INLINEABLE deg #-}
  frob :: Prime p → Prime p
  frob = identity
   {-# INLINEABLE frob #-}
```

Aquí $natVal$ toma un elemento cualquiera del tipo $Prime\ \ p$, es decir un elemento $P\ \ Mod\ \ p$ y de su tipo toma p a nivel de tipo y lo devuelve a nivel de término, de aquí es importante el poner el primo a nivel de tipo en $Prime\ p$.

Ya que $n = 1$, el grado de $\mathbb{F}_p$ sobre su subcampo primo, que es el mismo $\mathbb{F}_p$ es 1.

En el campo $\mathbb{F}_p$ se cumple que $x^p \equiv x\ \pmod{p}$, esto es consecuencia del teorema de Fermat. Debido a esto se implementa con $identity$.

Las instrucciones {-#INLINABLE #-} se utilizan para que el compilador GHC ubique las definiciones (si el compilar lo considera adecuado) justo en el lugar donde se pueden llamar posteriormente y su uso es solo para mejorar el rendimiento.

La restricción de clase $(KnownNat\ \ p)$ permite hacer uso de la función $natVal$ y se utiliza para tener el natural p a nivel de tipo y a nivel de término.

Teorema 2.2: Pequeño Teorema de Fermat

Sea p un número primo y a un entero entre 1 y $p-1$ entonces

$$x^{p-1} \equiv 1 \pmod{p}$$

La instancia

```
instance (KnownNat p) ⇒ PrimeField (Prime p) where
  fromP :: Prime p → Integer
  fromP (P x) = naturalToInteger (unMod x)
   {-# INLINEABLE fromP #-}
```

Permite obtener un entero asociado al elemento del campo $\mathbb{F}_p$.

2.3.1.1 Instancias de grupo

Ya que se derivo del tipo de dato $Mod\ \ p$ las instancias de Num y $Fractional$ al tipo de dato $Prime\ p$, las intancias de grupo descritas ahora no son muy complicadas de entender y están para dar más versatilidad.
Instancia de semigrupo con la operación del producto módulo p.

```
instance (KnownNat p) ⇒ Semigroup (Prime p) where
  (<>) :: Prime p → Prime p → Prime p
  (<>) = (*)
   {-# INLINE (<>) #-}
  stimes :: (Integral b) ⇒ b → Prime p → Prime p
  stimes = flip pow
   {-# INLINE stimes #-}
```

Instancia de monoide multiplicativo, con elemento identidad 1.

```
instance (KnownNat p) ⇒ Monoid (Prime p) where
  mempty :: Prime p
  mempty = P 1
   {-# INLINE mempty #-}
```

Instancia de grupo con elemento invertible, el inverso calculado con el algoritmo extendido de euclides con $recip$ que esta en la instancia de $Fractional$.

```
instance (KnownNat p) ⇒ Group (Prime p) where
  invert :: Prime p → Prime p
  invert = recip
   {-# INLINE invert #-}
  pow :: (Integral x) ⇒ Prime p → x → Prime p
  pow (P x) k = P (x ^% k)
   {-# INLINE pow #-}
```

Nótese la dependencia mutua de $semigroup$ con $group$ con el método pow.
La instrucción {-# INLINE #-} es similar a INLINABLE, solo que ahora se fuerza a GHC a ubicar la definición del método donde se llame.

2.3.1.2 Otras instancias

La clase Random permite obtener elementos aleatorios de $\mathbb{F}_p$. Esto es útil para los métodos de cifrado.

```
instance KnownNat p ⇒ Random (Prime p) where
  random :: (RandomGen g) ⇒ g → (Prime p, g)
  random = randomR (minBound, maxBound)
   {-# INLINABLE random #-}
  randomR :: (RandomGen g) ⇒ (Prime p, Prime p) → g → (Prime p, g)
  randomR (a, b) = first fromInteger ∘ randomR (fromP a, fromP b)
   {-# INLINABLE randomR #-}
```

Aquí $random$ toma un generador g y devuelve un elemento aleatorio entre 1 y $p-1$ junto a otro generador.
$randomR$ tomara el entero aleatorio entre a y b.
La clase $Real$ sirve para que un elemento de un campo finito se represente como un elemento $Rational$.

```
instance (KnownNat p) ⇒ Real (Prime p) where
  toRational :: Prime p → Rational
```

$toRational = fromIntegral$

{-# INLINEABLE toRational #-}

Aquí $toRational$ depende de $fromIntegral$ que es un método de la clase $Integral$.

instance $(KnownNat\ p) \Rightarrow Integral\ (Prime\ p)$ **where**

$quotRem :: Prime\ p \rightarrow Prime\ p \rightarrow (Prime\ p, Prime\ p)$

$quotRem = S.quotRem$

{-# INLINE quotRem #-}

$toInteger :: Prime\ p \rightarrow Integer$

$toInteger = fromP$

{-# INLINEABLE toInteger #-}

Aquí $quotRem$ de elementos $P\ \ a$ y $P\ \ b$ calcula el cociente y residuo de la división entera entre a y b, para ello utiliza el método $quotRem$ de la librería $Data.Euclidean = S$ y se puede hacer ya que $Prime\ P$ deriva la clase $Euclidean$.
Por otro lado $toInteger$ hace uso del método $fromP$ definido antes.

2.3.1.3 Ejemplo de uso

Para utilizar el campo se necesita de un número primo, por ej: 5, y hacer las declaraciones de tipo

type $F5 = Prime\ 5$

$f1 :: F5$

$f1 = 1$

$f2 :: F5$

$f2 = 2$

Pueden ser enteros que luego se transforman a $P\ 1$ o $P\ 2$ debido a las instancias de $Integral$.

2.3.2 Campo binario

El otro tipo de campo utilizado en las aplicaciones criptográficas es el campo binario, es decir aquel campo de Galois de la forma $\mathbb{F}_{2^n}$ para algún $n \geqslant 1$, y es utilizado por la facilidad de representar polinomios de $\mathbb{F}_{2^n}[X]$ solo con números binarios.

Para representar un polinomio $p(x) \in \mathbb{F}_{2^n}[X]$ se hace uso de la librería *bitvec* que implementa el tipo de dato *F2Poly* que representa polinomios binarios.
Se crea la siguiente clase para representar estos campos.

$$\textbf{class } (GaloisField\ k) \Rightarrow BinaryField\ k \textbf{ where}$$
$$\quad fromB :: k \to Integer$$

Donde *fromB* es un método que representa un elemento del campo binario como entero.
Para formar un campo binario se hace uso del siguiente resultado

Teorema 2.3

Sea K un campo y sea $p(x)$ un polinomio en $K[X]$ de grado $n \geqslant 1$. El anillo $K[X]/\langle p(x)\rangle$ es un campo si y solo sí $p(x)$ es irreducible en $K[X]$.

Demostración. La demostración no es complicada y se encuentra en [2]. □

Y este otro resultado, con lo que para tener un campo se debe tomar polinomios en $F_2[X]$ y reducirlos módulo $p(x)$ siendo $p(x)$ un polinomio irreducible, entonces la suma sera la usual, donde los coeficientes se reducen módulo 2, mientras que la multiplicación módulo $p(x)$.

Proposición 2.1

Sea $f(x) \in \mathbb{F}_p[X]$ irreducible con grado d. Entonces $|\ \mathbb{F}_p[X]/\langle f(x)\rangle\ | = p^d$. Más aún, cada elementos de $\mathbb{F}_p[X]/f(x)$ es congruente a exactamente un polinomio de grado menor o igual que d (o el polinomio 0), y dos polinomios de estos polinomios distintos nunca son congruentes módulo $f(x)$.

Demostración. La demostración puede encontrarse en [6]. □

Así ahora la información "global" que tiene que tener cada elemento del campo binario es su polinomio irreducible, y esto se logra de manera similar al campo primo, siendo esta vez representado como entero el polinomio irreducible (basta convertir a binario para tener el polinomio irreducible, ej: $p(x) = x^2 + x + 1 = 111$).

```
newtype Binary :: Nat → Type where
  B :: F2Poly → Binary p deriving (Eq, Generic, NFData, Ord, Show)
```

donde las clases derivadas son de *F2Poly*.
Ya que todo campo binario, es un campo de Galois, la instancia es (las otras instancias se definen más adelante):

```
instance (KnownNat p) ⇒ GaloisField (Binary p) where
  char :: Binary p → Natural
  char = const 2
   {-# INLINEABLE char #-}
  deg :: Binary p → Word
  deg = pred ∘ fromIntegral ∘ V.length ∘ unF2Poly ∘ toPoly ∘ natVal
   {-# INLINEABLE deg #-}
  frob :: Binary p → Binary p
  frob = join (*)
   {-# INLINEABLE frob #-}
```

La carácterísitca es 2, mientras que el grado es el grado del polinomio irreducible $p(x)$, y el automorfismo de frobeius esta dado por $x \to x * x = x^2$.
En lugar de definir operaciones en una clase *Field* se prefiere utilizar la clase *num* para definir las operaciones de manera más natural con $(+)$ y $(*)$.

```
instance (KnownNat p) ⇒ Num (Binary p) where
  (+) :: Binary p → Binary p → Binary p
  B x + B y = B $ x + y
   {-# INLINE (+) #-}
  (*) :: Binary p → Binary p → Binary p
  B x * B y = B $ P.rem (x * y) pIrr
    where
      pIrr = toPoly $ natVal $ witness@(Binary p)
   {-# INLINE (*) #-}
  (-) :: Binary p → Binary p → Binary p
  B x - B y = B $ x + y
  negate :: Binary p → Binary p
```

```
  {-# INLINE (-) #-}
negate = identity
  {-# INLINE negate #-}
fromInteger :: Integer → Binary p
fromInteger = B ∘ flip P.rem pIrr ∘ toPoly
  where
    pIrr = toPoly $ natVal $ witness@(Binary p)
  {-# INLINEABLE fromInteger #-}
```

Como ya se comentó antes, la suma es la suma usual en $\mathbb{F}_2[X]$, mientras que la multiplicación toma el residuo módulo $p(x)$. La resta se define igual debido a que con $a \in \mathbb{F}_2$ $-a = a$ por tener carácteristica 2. Por otro lado *fromInteger* toma un entero n lo convierte a un polinomio en $F_2[X]$ y lo reduce módulo $p(x)$.
Para poder encontrar el inverso multiplicativo, se hace uso de la clase *Fractional* para tener una instancia del método *recip*.

```
instance (KnownNat p) ⇒ Fractional (Binary p) where
  recip :: Binary p → Binary p
  recip (B x) =
    case gcdExt x pIrr of
      (1, y) → B y
      _ → divZeroError
    where
      pIrr = toPoly $ natVal (witness :: Binary p)
  {-# INLINE recip #-}
fromRational :: Rational → Binary p
fromRational rat = fromInteger (numerator rat) / fromInteger (denominator rat)
  {-# INLINEABLE fromRational #-}
```

Donde el inverso de un elemento se calcula mediante la función *gcdExt* de la librería *bitvec* con el uso del algoritmo extendido de euclides para polinomios. Mientras que *fromRational* toma un elemento del tipo rational, convierte el numerador y denominador en elementos de $\mathbb{F}_{2^n} X$ y los divide $a \,/\, b = a * recip\ b$.

2.3.2.1 Instancias de grupo

Se toma por defecto la operación multiplicación para ser semigrupo

```
instance (KnownNat p) ⇒ Semigroup (Binary p) where
  (<>) :: Binary p → Binary p → Binary p
  (<>) = (*)
   {-# INLINE (<>) #-}
  stimes :: (Integral b) ⇒ b → Binary p → Binary p
  stimes = flip pow
   {-# INLINE stimes #-}
```

El elemento identidad para monoide, es el polinimio constante e igual a 1.

```
instance (KnownNat p) ⇒ Monoid (Binary p) where
  mempty :: Binary p
  mempty = B 1
   {-# INLINE mempty #-}
```

Para determinar el inverso multiplicativo se reutiliza el método *recip*

```
instance (KnownNat p) ⇒ Group (Binary p) where
  invert :: Binary p → Binary p
  invert = recip
   {-# INLINE invert #-}
  pow :: (Integral x) ⇒ Binary p → x → Binary p
  pow x n
     | n ⩾ 0 = x^n
     | otherwise = recip x^P.negate n
   {-# INLINEABLE pow #-}
```

Aquí la exponciación negativa solo se toma el inverso de (x^{-n}).

2.3.2.2 Instancias de anillos

Con las operaciones definidas antes, se tiene facilmente una instancia de semianillo para $Binary\ p$.

```
instance (KnownNat p) ⇒ Semiring (Binary p) where
  fromNatural :: Natural → Binary p
  fromNatural = fromIntegral
   {-# INLINEABLE fromNatural #-}
  one :: Binary p
  one = B 1
   {-# INLINE one #-}
  plus :: Binary p → Binary p → Binary p
  plus = (+)
   {-# INLINE plus #-}
  times :: Binary p → Binary p → Binary p
  times = (*)
   {-# INLINE times #-}
  zero :: Binary p
  zero = B 0
   {-# INLINE zero #-}
```

De anillo

```
instance (KnownNat p) ⇒ Ring (Binary p) where
  negate :: Binary p → Binary p
  negate = P.negate
   {-# INLINE negate #-}
```

De dominio euclideano.

```
instance (KnownNat p) ⇒ Euclidean (Binary p) where
  degree :: Binary p → Natural
  degree (B x) = pred ∘ fromIntegral ∘ V.length ∘ unF2Poly $ x
  quotRem :: Binary p → Binary p → (Binary p, Binary p)
  quotRem (B x) (B y) = (B x', B y')
    where (x', y') = P.quotRem x y
   {-# INLINE quotRem #-}
```

Aunque F_{2^n} no es un dominio euclideano (el hecho de tener que definir una instancia de $Euclidean$ para $Field$ es una falla de diseño), los métodos se han definido para elementos de $F_2[X]$ que si es un dominio euclideano.

$$\textbf{instance}\ (KnownNat\ p) \Rightarrow GcdDomain\ (Binary\ p)$$

$$\textbf{instance}\ (KnownNat\ p) \Rightarrow Field\ (Binary\ p)$$

2.3.2.3 Otras instancias

La clase $Random$ se instancia para poder tener elementos aleatorios en $\mathbb{F}_{2^n}$, nuevamente esto es útil para aplicaciones criptográficas.

$$\begin{aligned}
&\textbf{instance}\ (KnownNat\ p) \Rightarrow Random\ (Binary\ p)\ \textbf{where}\\
&\quad random :: (RandomGen\ g) \Rightarrow g \rightarrow (Binary\ p, g)\\
&\quad random = randomR\ (B\ 0, B\ \$\ toPoly\ \$\ order\ (witness :: Binary\ p) - 1)\\
&\quad \text{\{-\# INLINEABLE random \#-\}}\\
&\quad randomR :: (RandomGen\ g) \Rightarrow (Binary\ p, Binary\ p) \rightarrow g \rightarrow (Binary\ p, g)\\
&\quad randomR\ (a, b) = first\ toB' \circ randomR\ (fromB\ a, fromB\ b)\\
&\quad \text{\{-\# INLINEABLE randomR \#-\}}
\end{aligned}$$

$random$ toma un generador y devuelve un elemento aleatorio entre 0 y el polinomio correspondiente al entero $2^n - 1$.
$randomR$ toma un generador y devuelve un elemento aleatorio entre a y b.
Los campos binarios se pueden enumerar

$$\begin{aligned}
&\textbf{instance}\ (KnownNat\ p) \Rightarrow Enum\ (Binary\ p)\ \textbf{where}\\
&\quad fromEnum :: Binary\ p \rightarrow Int\\
&\quad fromEnum = fromIntegral\\
&\quad \text{\{-\# INLINEABLE fromEnum \#-\}}\\
&\quad toEnum :: Int \rightarrow Binary\ p\\
&\quad toEnum = fromIntegral\\
&\quad \text{\{-\# INLINEABLE toEnum \#-\}}
\end{aligned}$$

Pueden pertenecer a la clase $Real$

$$\begin{aligned}
&\textbf{instance}\ (KnownNat\ p) \Rightarrow Real\ (Binary\ p)\ \textbf{where}\\
&\quad toRational :: Binary\ p \rightarrow Rational
\end{aligned}$$

$toRational = fromIntegral$

{-# INLINEABLE toRational #-}

Son $Integral$

instance $(KnownNat\ p) \Rightarrow Integral\ (Binary\ p)$ **where**

$quotRem :: Binary\ p \rightarrow Binary\ p \rightarrow (Binary\ p, Binary\ p)$

$quotRem = S.quotRem$

{-# INLINE quotRem #-}

$toInteger :: Binary\ p \rightarrow Integer$

$toInteger = fromB$

{-# INLINEABLE toInteger #-}

2.3.2.4 Otras funciones

$toB :: (KnownNat\ p) \Rightarrow Integer \rightarrow Binary\ p$

$toB = fromInteger$

$toB' :: Integer \rightarrow Binary\ p$

$toB' = B \circ toPoly$

$toPoly :: (Integral\ a) \Rightarrow a \rightarrow F2Poly$

$toPoly = fromIntegral$

2.3.2.5 Ejemplo de uso

Para hacer uso de estas clases se tiene que hacer uso del tipo de data $Binary$. Por ejemplo para el campo $\mathbb{F}_4$, se tiene el polinomio irreducible $p(x) = x^2 + x + 1$, que se representa como 111, de esta forma

type $F4 = Binary\ 0\ b111$

$f1 :: F4$

$f1 = 1$

$f2 :: F4$

$f2 = 2$

y cada elemento puede ser un entero, ya que se tiene las instancias $Integral$ para los métodos $fromIntegral$.

2.3.3 Polinomios de Conway

En el módulo $Math.Polynomials.Conway$ se incluye una lista de polinomios de Conway para campos binarios en la constante $conwayBinaryPoly$. Además se incluye mediante la extesión TemplateHaskell, una forma de crear un tipo de dato $Binary$ que represente un campo binario que recibe un polinomio de Conway. También se incluye otro método para campos primos, que utiliza la comprobación prima antes dada.

$$
\begin{aligned}
&primeField :: Integer \to TH.TypeQ \\
&primeField\ n \\
&\quad \mid n \leqslant 0 = error\ \texttt{"primeField: valor negativo"} \\
&\quad \mid isPrimeTrialDivision\ n = [t \mid Prime\ \$\ (TH.litT\ (TH.numTyLit\ n))\ |] \\
&galoisField :: Integer \to TH.TypeQ \\
&galoisField\ n \\
&\quad \mid n \leqslant 0 = error\ \texttt{"galoisField: valor negativo"} \\
&\quad \mid otherwise = [t \mid Binary\ \$\ (TH.litT\ (TH.numTyLit\ n))\ |]
\end{aligned}
$$

Y se usan de la siguiente forma

$$
\begin{aligned}
&a :: \$(primeField\ 7) \\
&a = 1 \\
&b :: \$(galoisField\ 0\ b111) \\
&b = 3
\end{aligned}
$$

3 Curva Elíptica

Según la geometría algebraica,

Definición 3.1: Curva elíptica

Una **curva elíptica** sobre un campo K es una curva proyectiva singular de género 1 definida por ecuaciones con coeficientes en K y con al menos un punto racional.

Por el teorema de Riemann-Roch una curva elíptica es isomoforma a una cúbica plana singular. Así tenemos la definición

Definición 3.2: Curva elíptica

Una curva elíptica es el conjunto de soluciones a una ecuación de la forma

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

donde a, b pertenecen a algún campo K. Las ecuaciones de este tipo se dice que están en forma de *Weierstrass*.

Si a este conjunto le añadimos un elemento denominado punto en el infinito y representado por $\mathcal{O}$, entonces con una adecuada operación denominada "suma de puntos de curva", adquiere la estructura de grupo.

Para que esta operación este bien definida, se necesita que la curva sea singular, es decir su discriminante tiene que ser diferente de cero, para la ecuación de Weierstrass el discriminante está dado por

$$\triangle_E = 4A^3 + 27B^2.$$

Esta forma de ecuación de Weierstrass solo es válida para campos de carácterística diferente a 2 y a 3. Por lo tanto solo se podrá usar para campos primos mayor o igual a 5 en esta librería.

3.1 Ley de suma para campos primos

Definición 3.3: Ley de suma

Sea E definida por $y^2 = x^3 + Ax + B$. Sea $P_1 = (x_1, y_1)$ y $P_2 = (x_2, y_2)$ puntos en E con $P_1, P_2 \neq \mathcal{O}$. Definimos $P_1 \oplus P_2 = P_3 = (x_3, y_3)$ como sigue:

1. Si $x_1 \neq x_2$, entonces

$$x_3 = m^2 - x_1 - x_2, \qquad y_3 = m(x_1 - x_3) - y_1, \qquad \text{donde } m = \frac{y_2 - y_1}{x_2 - x_1}.$$

2. Si $x_1 = x_2$ pero $y_1 \neq y_2$, entonces $P_1 \oplus P_3 = \mathcal{O}$.

3. Si $P_1 = P_2$, y $y_1 \neq y_2$, entonces

$$x_3 = m^2 - 2x_1, \qquad y_3 = m(x_1 - x_3) - y_1, \qquad \text{donde } m = \frac{3x_1^2 + A}{2y_1}$$

4. Si $P_1 = P_2$ y $y_1 = 0$, entonces $P_1 \oplus P_2 = \mathcal{O}$ Y además definimos

$$P \oplus \mathcal{O} = P$$

para todos los puntos P en E.

Para aplicaciones criptográficas el campo subyacente debe ser un campo finito y solo se utilizan dos tipos, los campos primos y los campos binarios.
Para campos binarios, se define una curva elíptica de forma Weierstrass más general:

Definición 3.4: Forma generalizada de Weierstrass

La ecuación de Weierstrass generalizada de una curva elíptica E sobre un campo K viene dada por

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

donde $a_i \in K$ para $i \in \{1, \ldots, 6\}$.

De donde tenemos dos casos:

Si la característica es 2, y $a_1 \neq 0$, el siguiente cambio de variable

$$(x, y) \to \left(a_1^2x + \frac{a_3}{a_1}, a_1^3y + \frac{a_1^2a_4 + a_3^2}{a_1^3}\right)$$

trasforma la curva generalizada a la curva

$$y^2 + xy = x^3 + ax^2 + b$$

donde $\triangle_E = b$. Si $a_1 = 0$, entonces el siguiente cambio de variable

$$(x, y) \to (x + a_2, y)$$

transforma la ecuación a la curva

$$y^2 + cy = x^3 + ax + b$$

con discriminate $\triangle_E = c^4$.

Entonces para los campos binarios con $a_1 \neq 0$ se puede demostrar [3, pág. 79] que la suma de puntos de curva elíptica queda definida de la siguiente manera:

3.2 Ley de suma para campos binarios

Definición 3.5: Ley de grupo para la curva $E(\mathbb{F}_{2^q}) : y^2 + xy = x^3 + ax^2 + b$

1. *Identidad.* $P \oplus \mathcal{O} = \mathcal{O} \oplus P$ para todo $P \in E(\mathbb{F}_{2^q})$.

2. *Inversas.* Si $P = (x, y) \in E(\mathbb{F}_{2^q})$, entonces $(x, y) \oplus (x, x + y) = \mathcal{O}$.

3. *Suma de puntos.* Sea $P = (x_1, y_1) \in E(\mathbb{F}_{2^q})$ y $Q = (x_2, y_2) \in E(\mathbb{F}_{2^q})$, donde $P \neq \pm Q$. Entonces $P \oplus Q = (x_3, y_3)$, donde

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad \text{y} \quad y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

con $\lambda = \dfrac{y_1 + y_2}{x_1 + x_2}$.

4. *Doblaje de puntos.* Sea $P = (x_1, y_1) \in E(F_{2^q})$, donde $P \neq -P$. Entonces $2P = (x_3, y_3)$, donde

$$x_3 = \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2} \quad \text{y } y_3 = \lambda x_3 + x_3$$

con $\lambda = x_1 + \dfrac{y_1}{x_1}$.

Y cuando $a_1 = 0$, se tiene la siguiente definición de suma

Definición 3.6: Ley de grupo para la curva $E(\mathbb{F}_{2^n}) : y^2 + cy = x^3 + ax + b$

1. *Identidad.* $P \oplus \mathcal{O} = \mathcal{O} \oplus P = P$ para todo $P \in E(\mathbb{F}_{2^n})$.

2. *Inversas.* Si $P = (x, y) \in E(\mathbb{F}_{2^n})$, entonces $(x, y) \oplus (x, y + c) = \mathcal{O}$.

3. *Suma de puntos.* Sea $P = (x_1, y_1) \in E(\mathbb{F}_{2^n})$ y $Q = (x_2, y_2) \in E(\mathbb{F}_{2^n})$, donde $P \neq \pm Q$. Entonces $P \oplus Q = (x_3, y_3)$, donde

$$x_3 = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)^2 + x_1 + x_2 \quad \text{y} \quad y_3 = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)(x_1 + x_3) + y_1 + c.$$

4. *Doblaje de puntos.* Sea $P = (x_1, y_1) \in E(\mathbb{F}_{2^n})$, donde $P \neq -P$. Entonces $2P = (x_3, y_3)$, donde

$$x_3 = \left(\frac{x_1^2 + a}{c}\right)^2 \quad \text{y} \quad \left(\frac{x_1^2 + a}{c}\right)(x_1 + x_3) + y_1 + c.$$

Resulta que con estas operaciones, se tiene una estructura de grupo abeliano, lo cual permite plantear criptosistemas.

Teorema 3.1

La suma de puntos en una curva elíptica E satisface las siguientes propiedades:

1. (conmutatividad) $P_1 \oplus P_2 = P_2 \oplus P_1$ para todo P_1, P_2 en E.

2. (existencia de identidad) $P \oplus \mathcal{O} = P$ para todos los puntos P en E.

3. (existencia de inversos) Dado P en E, entonces existe P' en E con $P \oplus P' = \mathcal{O}$. Este punto P' usualmente se denota por $-P$.

4. (asociatividad) $(P_1 \oplus P_2) \oplus P_3 = P_1 \oplus (P_2 \oplus P_3)$.

A esta forma de determinar los puntos, como $(x, y) \in K$ y un punto especial, el punto en el infinito, se le denomina sistema de coordenadas *afín* y es el que se utilizó en la librería.

3.3 Clase de curva elíptica

En general la clase para una curva elíptica está dada por.

```
class
  (GaloisField q,
    PrimeField r,
    Eq (Point f c e q r),
    Group (Point f c e q r),
    NFData (Point f c e q r),
    Random (Point f c e q r),
    Show (Point f c e q r)
  ) ⇒
  Curve (f :: Form) (c :: Coordinates) e q r
  where
  data Point f c e q r :: Type
  char :: Point f c e q r → Natural
  cof :: Point f c e q r → Natural
  def :: Point f c e q r → Bool
  disc :: Point f c e q r → q
  order :: Point f c e q r → Natural
  add :: Point f c e q r → Point f c e q r → Point f c e q r
  dbl :: Point f c e q r → Point f c e q r
  id :: Point f c e q r
  inv :: Point f c e q r → Point f c e q r
  frob :: Point f c e q r → Point f c e q r
  gen :: Point f c e q r
  rnd :: (MonadRandom m) ⇒ m (Point f c e q r)
  rnd = getRandom
```

Donde:

- *Curve* tiene es una clase con varios parámetros, f, c, e, q, r

1. f es para la forma de la curva elíptica, es decir si es Weierstrass o Binaria. Se utilizará Weierstrass si el campo subyacente es primo y en otro caso Binaria. Ya que f tiene el tipo $Form$ y se define como un tipo de dato

$$\begin{aligned}&\textbf{data } Form\\&\quad = Binary\\&\quad \mid Binary2\\&\quad \mid Weierstrass\end{aligned}$$

2. c es para determinar las coordenadas de la curva, en esta librería solo se implementan las coordenas afines. Esta dado por el tipo de dato $Coordinates$

$$\textbf{data } Coordinates = Affine$$

3. e representa el nombre estanderizado para curvas precalculadas de la base de datos.
4. q representa el campo subyacente y debe ser del tipo $GaloisField$.
5. r representa un campo primo y es usado para la enésima suma de un punto de curva elíptica nP con $P \in E(K)$ y $n \in F_r$, donde r será el orden de un punto P ya dado de la curva elíptica, un generador de un subgrupo de orden cercano al orden de la curva.

- Se asocia un tipo de dato $Point$ que representa el punto de curva elíptica y se define adecuadamente según la curva, la forma y coordenadas.
- El método $char$ es la carácteristica del campo subyacente.
- El método gen es el generador de algún subgrupo de $E(K)$.
- El método cof es el cofactor de curva, se utiliza para almacenar el valor $h = \frac{\mid E(K) \mid}{\mid \langle P \rangle \mid}$ donde $\mid \cdot \mid$ representa el orden de grupo. El punto P viene dado por el generador de un subgrupo grande dado por gen.
- El método def comprueba que un punto pertenezca a $E(K)$, es decir satisfaga la ecuación de curva.

- El método $disc$ calcula el discriminante de la curva.
- El método $order$ da el orden del grupo de curva.
- add representa la suma de puntos de curva elíptica.
- dbl es para calcular la suma de un punto consigo mismo.
- id es la identidad del grupo de curva, es decir el punto $\mathcal{O}$.
- inv calcula el inverso aditivo de un punto.
- $frob$ es el endomorfismo de Frobenius para puntos de curva elíptica.
- rnd sirve para determinar un punto aleatorio de la curva.

Otras funciones son:

$$
\begin{aligned}
&mul :: (Curve\ f\ \ c\ e\ q\ r) \Rightarrow Point\ f\ \ c\ e\ q\ r \rightarrow r \rightarrow Point\ f\ \ c\ e\ q\ r\\
&mul = (\circ fromP) \circ mul'\\
&mul' ::\\
&(Curve\ f\ \ c\ e\ q\ r, Integral\ n) \Rightarrow\\
&Point\ f\ \ c\ e\ q\ r \rightarrow\\
&n \rightarrow\\
&Point\ f\ \ c\ e\ q\ r\\
&mul'\ p\ n\\
&\quad \mid n < 0 = inv\ \$\ mul'\ p\ (-n)\\
&\quad \mid n \equiv 0 = id\\
&\quad \mid n \equiv 1 = p\\
&\quad \mid even\ n = p'\\
&\quad \mid otherwise = add\ p\ p'\\
&\quad \textbf{where}\\
&\qquad p' = mul'\ (dbl\ p)\ (div\ n\ 2)
\end{aligned}
$$

Donde mul implementa la suma enésima de un punto consigo mismo que hace uso de un doblaje rápido de puntos.

3.3.1 Instancias

Como ya se ha mencionado, el conjunto de soluciones de una curva elíptica, en un campo junto al punto en el infinito y la definición de suma, forman un grupo, así el tipo $Point$ debe tener instancia de grupo.

Instancia de semigrupo

$$\begin{aligned}
&\textbf{instance}\ (Curve\ f\ c\ e\ q\ r) \Rightarrow Semigroup\ (Point\ f\ c\ e\ q\ r)\ \textbf{where}\\
&\quad (\diamond) :: Point\ f\ c\ e\ q\ r \rightarrow Point\ f\ c\ e\ q\ r \rightarrow Point\ f\ c\ e\ q\ r\\
&\quad p \diamond q = \textbf{if}\ p \equiv q\ \textbf{then}\ dbl\ p\ \textbf{else}\ add\ p\ q\\
&\quad \text{\{-\# INLINEABLE (<>) \#-\}}
\end{aligned}$$

Instancia de Monoide

$$\begin{aligned}
&\textbf{instance}\ (Curve\ f\ c\ e\ q\ r) \Rightarrow Monoid\ (Point\ f\ c\ e\ q\ r)\ \textbf{where}\\
&\quad mempty :: Point\ f\ c\ e\ q\ r\\
&\quad mempty = id\\
&\quad \text{\{-\# INLINEABLE mempty \#-\}}
\end{aligned}$$

Instancia de Grupo

$$\begin{aligned}
&\textbf{instance}\ (Curve\ f\ c\ e\ q\ r) \Rightarrow Group\ (Point\ f\ c\ e\ q\ r)\ \textbf{where}\\
&\quad invert :: Point\ f\ c\ e\ q\ r \rightarrow Point\ f\ c\ e\ q\ r\\
&\quad invert = inv\\
&\quad \text{\{-\# INLINEABLE invert \#-\}}\\
&\quad pow :: (Integral\ x) \Rightarrow Point\ f\ c\ e\ q\ r \rightarrow x \rightarrow Point\ f\ c\ e\ q\ r\\
&\quad pow = mul'\\
&\quad \text{\{-\# INLINEABLE pow \#-\}}
\end{aligned}$$

Instancia para elementos aleatorios.

$$\begin{aligned}
&\textbf{instance}\ (Curve\ f\ c\ e\ q\ r) \Rightarrow Random\ (Point\ f\ c\ e\ q\ r)\ \textbf{where}\\
&\quad random :: (RandomGen\ g) \Rightarrow g \rightarrow (Point\ f\ c\ e\ q\ r, g)\\
&\quad random = first\ (mul\ gen) \circ random\\
&\quad \text{\{-\# INLINEABLE random \#-\}}\\
&\quad randomR :: (Point\ f\ c\ e\ q\ r, Point\ f\ c\ e\ q\ r) \rightarrow g \rightarrow (Point\ f\ c\ e\ q\ r, g)\\
&\quad randomR = panic\ \texttt{"Curve.randomR: no implementado."}
\end{aligned}$$

Donde $random$ selecciona un elemento aleatorio en el campo primo y lo multiplica por el generador de curva.
$randomR$ no se implementa ya que no se tiene una orden para dar un punto aleatorio entre dos puntos P y Q.

3.4 Curvas binarias

En estas curvas se usó la forma generalizada de Weierstrass, puesto que el campo tiene característica 2. Y pueden ser de las dos formas antes vistas en función de que si $a_1 = 0$ o $a_1 \neq 0$.
Primero, se crea el tipo de dato $BPoint$ que aplica parcialmente el tipo de dato $Point$

$$\textbf{type } BPoint = Point\ 'Binary$$

Aquí $Binary$ pertenece al tipo $Form$ y está a nivel de término pero con $'Binary$ se promueve a nivel de tipo. Esto se permite con la extensión DataKinds habilitada por defecto en GHC2021.
Ahora se define una clase para curvas binarias

$$\begin{aligned}
&\textbf{class}\\
&\quad (GaloisField\ q, PrimeField\ r, Curve\ 'Binary\ c\ e\ q\ r) \Rightarrow\\
&\quad BCurve\ c\ e\ q\ r\\
&\quad \textbf{where}\\
&\quad a_ :: BPoint\ c\ e\ q\ r \rightarrow q\\
&\quad b_ :: BPoint\ c\ e\ q\ r \rightarrow q\\
&\quad h_ :: BPoint\ c\ e\ q\ r \rightarrow Natural\\
&\quad p_ :: BPoint\ c\ e\ q\ r \rightarrow Natural\\
&\quad r_ :: BPoint\ c\ e\ q\ r \rightarrow Natural
\end{aligned}$$

En esta clase, ya que ya se tiene la forma de la curva, solo se pide las coordenadas c, el nombre de la curva e, el campo de Galois q, y el campo primo r.
Los métodos son:

- $a\_$ A cada punto de la curva, da el coeficiente A de la ecuación.
- $b\_$ A cada punto de la curva, da el coeficiente B

- $h\_$ A cada punto de la curva, devuelve el cofactor.
- $p\_$ Devuelve el polinomio irreducible del campo de Galois.
- $r\_$ Devuelve el orden de la curva.

Para especificar que se usan las coordenas afines, se crea otro tipo de dato.

$$\textbf{type}\ BAPoint = BPoint\ 'Affine$$

Así ahora tenemos el tipo de dato $Point$ pero ya especificado que sera una curva binaria y de sistema de coordenadas afín.
Ahora se crea la clase para curvas binarias de la forma $y^2 + xy = x^3 + Ax^2 + B$. Es decir $a_1 \neq 0$ en la ecuación generalizada de Weierstrass.

$$\begin{aligned}&\textbf{class}\ (BCurve\ 'Affine\ e\ q\ r) \Rightarrow BACurve\ e\ q\ r\ \textbf{where}\\ &\quad gA\_ :: BAPoint\ e\ q\ r\end{aligned}$$

Donde $gA\_$ es el generador de la curva.

3.4.1 Instancias

Se hace una instancia para la clase $Curve$ en el que los 2 primeros parámetros ya se han dado, y los restantes tienen la restricción de clase $BACurve$, es decir deben tener una instancia de $BACurve$.
Se explica los métodos parte por parte

$$\begin{aligned}&\textbf{instance}\ (BACurve\ e\ q\ r) \Rightarrow Curve\ 'Binary\ 'Affine\ e\ q\ r\ \textbf{where}\\ &\quad \textbf{data}\ Point\ 'Binary\ 'Affine\ e\ q\ r\\ &\qquad = A\ q\ q \mid O\\ &\quad \textbf{deriving}\ (Eq, Generic, NFData, Read, Show)\end{aligned}$$

El tipo de dato asociado $Point$ es inicado por tipo de dato algebraica de suma, un constructor de datos A que recibe dos coordenadas del campo finito (que se supone satisfacen la ecuación de curva elíptica) o el tipo de dato O que representa el punto en el infinito. Esto obedece el sistema de coordenas afines.
Además con las clases que se derivan, entre otras cosas se puede igualar y mostrar este tipo de dato. Es decir, esta derivación implica que dos puntos (q_1, q_2), (k_1, k_2) son iguales si y solo si $q_1 = k_1$ y $q_2 = k_2$.

```
inv :: Point Binary Affine e q r → Point Binary Affine e q r
inv O = O
inv (A x y) = A x (x + y)
```

Este método, determina el inverso aditivo de un punto P de la curva elíptica, es decir, aquel único punto P', tal que $P \oplus P' = \mathcal{O}$.

```
add ::
  Point 'Binary 'Affine e q r →
  Point 'Binary 'Affine e q r →
  Point 'Binary 'Affine e q r
add p O = p
add O q = q
add p@(A x1 y1) (A x2 y2)
  | xx ≡ 0 ∧ yy + x2 ≡ 0 = O
  | x1 ≡ x2 = dbl p
  | otherwise = A x3 y3
  where
    a = a_ (witness :: BAPoint e q r)
    xx = x1 + x2
    yy = y1 + y2
    l = yy / xx
    x3 = l * (l + 1) + xx + a
    y3 = l * (x1 + x3) + x3 + y1
```

Donde se define $P \oplus \mathcal{O} = \mathcal{O} \oplus P = P$ para todo punto $P \in E(K)$. Mediante el encaje de patrones (*pattern-matching*) y con el tipo de dato $Point$ definido, luego se comprueba la condición $x_1 = x_2$ y $y_1 + y_2 + x_2 = 0$, que comprueba que el punto Q a sumar a P, es en realidad el inverso aditivo $Q = -P$, y por lo tanto el resultado es el punto en el infinito $\mathcal{O}$. Si solo sucede que $x_1 = x_2$, entonces el punto Q es el mismo que P y entonces se dobla el punto. Y si no se cumplen las anteriores condiciones, entonces el punto P y Q, son diferentes con lo que se suman los puntos de la siguiente manera:

- Se define $\lambda = \dfrac{y_2 + y_1}{x_2 + x_1}$. Y resulta en $P \oplus Q = R = (x_3, y_3)$, donde

- $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a.$
- $y_3 = \lambda(x_1 + x_3) + x_3 + y_1.$

Mientras que si se quiere sumar un punto consigo mismo, se tiene que doblar, por lo que el método se define como

$$
\begin{aligned}
&dbl :: Point\ Binary\ Affine\ e\ q\ r \rightarrow Point\ Binary\ Affine\ e\ q\ r\\
&dbl\ O = O\\
&dbl\ (A\ x\ y) = A\ x'\ y'\\
&\quad \textbf{where}\\
&\qquad a = a\_\ (witness :: BAPoint\ e\ q\ r)\\
&\qquad l = x + y\ /\ x\\
&\qquad l' = l + 1\\
&\qquad x' = l * l' + a\\
&\qquad y' = x * x + l' * x'
\end{aligned}
$$

Donde:

- $\lambda = x_1 + \dfrac{y_1}{x_1}.$
- $x_3 = \lambda^2 + \lambda + a = x_1^2 + \dfrac{b}{x_1^2}$ y $y_3 = x_1^2 \lambda x_3 + x_3.$

La característica es la caracteristíca del campo subyacente, en este caso 2.

$$
\begin{aligned}
&char :: Point\ Binary\ Affine\ e\ q\ r \rightarrow Natural\\
&char = const\ 2
\end{aligned}
$$

El cofactor se determina con cof

$$
\begin{aligned}
&cof :: Point\ Binary\ Affine\ e\ q\ r \rightarrow Natural\\
&cof = h\_
\end{aligned}
$$

Para comprobar si un par de puntos $x, y \in \mathbb{F}_{2^n}$ satisfacen la ecuación de curva elíptica, se utiliza def

$$
\begin{aligned}
&def :: Point\ Binary\ Affine\ e\ q\ r \rightarrow Bool\\
&def\ O = True
\end{aligned}
$$

$$def\ (A\ x\ y) = ((x + a) * x + y) * x + b + y * y \equiv 0$$

where

$$a = a\_\ (witness :: BAPoint\ e\ q\ r)$$

$$b = b\_\ (witness :: BAPoint\ e\ q\ r)$$

disc determina el discriminante de la curva. En el caso de la curva $E : y^2 + xy = x^3 + ax^2 + b$, el discriminante es b.

$$disc :: Point\ Binary\ Affine\ e\ q\ r \rightarrow q$$

$$disc\ \_ = b\_\ (witness :: BAPoint\ e\ q\ r)$$

El endomorfismo de Frobeinus, se determina con *frob* y esta dado por $(x, y) \rightarrow (x^2, y^2)$.

$$frob :: Point\ Binary\ Affine\ e\ q\ r \rightarrow Point\ Binary\ Affine\ e\ q\ r$$

$$frob\ O = O$$

$$frob\ (A\ x\ y) = A\ (F.frob\ x)\ (F.frob\ y)$$

gen determina el generador de un subgrupo.

$$gen :: Point\ Binary\ Affine\ e\ q\ r$$

$$gen = gA\_$$

id denota el elemento identidad del grupo.

$$id :: Point\ Binary\ Affine\ e\ q\ r$$

$$id = O$$

order calcula el orden de la curva.

$$order :: Point\ Binary\ Affine\ e\ q\ r \rightarrow Natural$$

$$order = r\_$$

3.4.2 Ejemplo: SECT113R1

Se hizo un ejemplo con la curva elíptica SECT113R1 de la base de datos STD.
La curva SECT113R1 es una curva elíptica de 113 bits de un campo binario, dado por la ecuación:

$$y^2 + xy = x^3 + ax^2 + b$$

Y de parámetros

Cuadro 3.1 Parámetros de la curva SECT113R1

Nombre	Valor
m	131
f(x)	$x^{131} + x^8 + x^3 + x^2 + 1$
a	0x07a11b09a76b562144418ff3ff8c2570b8
b	0x0217c05610884b63b9c6c7291678f9d341
G	(0x0081baf91fdf9833c40f9c181343638399, 0x078c6e7ea38c001f73c8134b1b4ef9e150)
n	0x0400000000000000023123953a9464b54d
h	0x2

Fuente: Jancar y Sedlacek, 2020.

Realizado por: Reinoso, D., 2024.

Donde de [7]:

- m es la dimensión del campo $\mathbb{F}_{2^m}$ visto como espacio vectorial sobre su subcampo primo.
- $f(x)$ es el polinomio irreducible sobre $\mathbb{F}_2$.
- a y b son los coeficientes de la curva generalizada de Weierstrass.
- G es un punto base en $E(\mathbb{F}_{2^m})$.
- n es un primo y orden de G, $n = | \langle G \rangle |$.
- h que es el cofactor de la curva, se define como $h = \dfrac{| E(\mathbb{F}_{2^m}) |}{n}$.

Y se implementa de la siguiente manera:
Se crea el tipo da dato para nombre de la curva (un tipo sin habitantes).

```
data SECT113R1
```

Se crea el campo binario, para ello, se toma el polinomio irreducible de la base de datos $(f(x))$, en hexadecimal a nivel de tipo:

$$\textbf{type } P = \text{0x20000000000000000000000000201}$$

$$\textbf{type } F2n = Binary\ P$$

Se crea un campo primo, que representa el orden generado por el punto G de la curva.

$$\textbf{type } R = \text{0x100000000000000d9ccec8a39e56f}$$

$$\textbf{type } Fr = Prime\ R$$

Los parámetros a y b de la curva.

$$\_a :: F2n$$

$$\_a = \text{0x3088250CA6E7C7FE649CE85820F7}$$

$$\_b :: F2n$$

$$\_b = \text{0xe8bee4d3e2260744188be0e9c723}$$

El cofactor, orden y polinomio

$$\_h :: Natural$$

$$\_h = \text{0x2}$$

$$\_r :: Natural$$

$$\_r = \text{0x100000000000000d9ccec8a39e56f}$$

$$\_p :: Natural$$

$$\_p = \text{0x20000000000000000000000000201}$$

Y el punto generador

$$\_x :: F2n$$

$$\_x = \text{0x9d73616f35f4ab1407d73562c10f}$$

$$\_y :: F2n$$

$$\_y = \text{0xa52830277958ee84d1315ed31886}$$

$$gA :: PA$$

$$gA = A\ \_x\ \_y$$

Finalmente, se crea la instancia

El tipo que representa la curva

$$\textbf{type } PA = BAPoint\ SECT113R1\ F2n\ Fr$$

Instancia de curva binaria

```
instance (Curve 'Binary c SECT113R1 F2n Fr) ⇒ BCurve c SECT113R1 F2n Fr where
  a_ :: BPoint c SECT113R1 F2n Fr → F2n
  a_ = const _a
  b_ :: BPoint c SECT113R1 F2n Fr → F2n
  b_ = const _b
  h_ :: BPoint c SECT113R1 F2n Fr → Natural
  h_ = const _h
  p_ :: BPoint c SECT113R1 F2n Fr → Natural
  p_ = const _p
  r_ :: BPoint c SECT113R1 F2n Fr → Natural
  r_ = const _r
```

E instancia de curva afín binaria

```
instance BACurve SECT113R1 F2n Fr where
  gA_ :: BAPoint SECT113R1 F2n Fr
  gA_ = gA
```

3.5 Otras curvas binarias

En esta sección se tratan las curvas generalizadas de Weierstrass en donde $a_1 = 0$.
Se crea el tipo de dato $B2Point$ que aplica parcialmente $Point$

```
type B2Point = Point 'Binary2
```

Se define la clase para este tipo de curva binaria, agregando un parámetro $c\_$ que representa la variable c en la curva $E : y^2 + cy = x^3 + ax + b$.

```
class (GaloisField q, PrimeField r, Curve 'Binary2 c e q r) ⇒ B2Curve c e q r
  where
  a_ :: B2Point c e q r → q
  b_ :: B2Point c e q r → q
  c_ :: B2Point c e q r → q
```

```
h_ :: B2Point c e q r → Natural
p_ :: B2Point c e q r → Natural
r_ :: B2Point c e q r → Natural
```

Se crea el tipo de dato para las coordenadas afines

```
type B2APoint = B2Point 'Affine
```

La clase para la curva afín binaria de tipo 2

```
class (B2Curve 'Affine e q r) ⇒ B2ACurve e q r where
  gA_ ::
    B2APoint e q r
```

Y las instancias de las operaciones ya definidas

```
instance (B2ACurve e q r) ⇒ Curve 'Binary2 'Affine e q r where
  data Point 'Binary2 'Affine e q r
    = A q q | O
    deriving (Eq, Generic, NFData, Read, Show)
  add ::
    Point 'Binary2 ' Affine e q r →
    Point 'Binary2 ' Affine e q r →
    Point 'Binary2 ' Affine e q r
  add p O = p
  add O q = q
  add p@(A x1 y1) q@(A x2 y2)
    | xx ≡ 0 ∧ yy + c ≡ 0 = O    -- Si Q es la inversa de P
    | x1 ≡ x2 = dbl p
    | otherwise = A x3 y3
    where
      c = c_ (witness :: B2APoint e q r)
      xx = x1 + x2
      yy = y1 + y2
      l = yy / xx
```

```
    x3 = l * l + xx
    y3 = l * (x1 + x3) + y1 + c
char :: Point Binary2 Affine e q r → Natural
char = const 2
cof :: Point Binary2 Affine e q r → Natural
cof = h_
dbl :: Point Binary2 Affine e q r → Point Binary2 Affine e q r
dbl O = O
dbl (A x y) = A x' y'
  where
    a = a_ (witness :: B2APoint e q r)
    c = c_ (witness :: B2APoint e q r)
    v = (x * x + a) / c
    x' = v * v
    y' = v * (x + x') + y + c
def :: Point Binary2 Affine e q r → Bool
def O = True
def (A x y) = (x * x + a) * x + b + y * y + c * y ≡ 0
  where
    a = a_ (witness :: B2APoint e q r)
    b = b_ (witness :: B2APoint e q r)
    c = c_ (witness :: B2APoint e q r)
disc :: Point Binary2 Affine e q r → q
disc _ = c^4
  where
    c = c_ (witness :: B2APoint e q r)
frob :: Point Binary2 Affine e q r → Point Binary2 Affine e q r
frob O = O
frob (A x y) = A (F.frob x) (F.frob y)
gen :: Point Binary2 Affine e q r
gen = gA_
```

```
id :: Point Binary2 Affine e q r
id = O

inv :: Point Binary2 Affine e q r → Point Binary2 Affine e q r
inv O = O
inv (A x y) = A x (y + c)
  where
    c = c_ (witness :: B2APoint e q r)

order :: Point Binary2 Affine e q r → Natural
order = r_
```

Estas curvas no tienen aplicaciones en criptografía debido a problemas de seguridad, por lo que no se aportan ejemplos aplicados.

3.6 Curvas de Weierstrass

Estas curvas se utilizan para campos primos, que no tengan característica 2 ni 3.
Se crea el tipo de dato *WPoint* para puntos de una curva elíptica en forma de Weierstrass.

```
type WPoint = Point 'Weierstrass
```

Ahora se define una clase para curvas de Weierstrass.

```
class (GaloisField q, PrimeField r, Curve 'Weierstrass c e q r)
 ⇒ WCurve c e q r where
  a_ :: WPoint c e q r → q
  b_ :: WPoint c e q r → q
  h_ :: WPoint c e q r → Natural
  q_ :: WPoint c e q r → Natural
  r_ :: WPoint c e q r → Natural
```

Donde $q\_$ representa a la característica del campo, y los demás parámetros son análogos a los dados antes.
Se crea el siguiente tipo de dato para especificar las coordenadas afines

```
type WAPoint = WPoint 'Affine
```

Y la clase para curvas afines de Weierstrass

```
class (WCurve 'Affine e q r) ⇒ WACurve e q r where
  gA_ :: WAPoint e q r
```

Donde $gA\_$ es el generador de algún subgrupo de la curva.

3.6.1 Instancias

Se hace una instancia para *Curve* para los 2 primeros parámetros dados y el resto tiene una restricción de clase *WACurve*.

```
instance (WACurve e q r) ⇒ Curve 'Weierstrass 'Affine e q r where
  data Point 'Weierstrass 'Affine e q r
    = A q q | O
    deriving (Eq, Generic, NFData, Read, Show)
```

Donde el punto de la curva de Weierstrass se representa igual que en los anteriores casos.
El inverso aditivo, se calcula de la siguiente forma:

```
inv :: Point Weierstrass Affine e q r →
  Point Weierstrass Affine e q r
inv O = O
inv (A x y) = A x (−y)
```

El discriminante es $\triangle_E = 4A^3 + 27B^2$.

```
disc :: Point Weierstrass Affine e q r → q
disc _ = 4 ∗ a ∗ a ∗ a + 27 ∗ b ∗ b
  where
    a = a_ (witness :: WAPoint e q r)
    b = b_ (witness :: WAPoint e q r)
```

El endomorfismo de Frobenius:

$$
\begin{aligned}
&frob :: Point\ Weierstrass\ Affine\ e\ q\ r \rightarrow \\
&\quad Point\ Weierstrass\ Affine\ e\ q\ r \\
&frob\ O = O \\
&frob\ (A\ x\ y) = A\ (F.frob\ x)\ (F.frob\ y)
\end{aligned}
$$

El generador:

$$
\begin{aligned}
&gen :: Point\ Weierstrass\ Affine\ e\ q\ r \\
&gen = gA\_
\end{aligned}
$$

La identidad:

$$
\begin{aligned}
&id :: Point\ Weierstrass\ Affine\ e\ q\ r \\
&id = O
\end{aligned}
$$

El orden:

$$
\begin{aligned}
&order :: Point\ Weierstrass\ Affine\ e\ q\ r \rightarrow Natural \\
&order = r\_
\end{aligned}
$$

La característica:

$$
\begin{aligned}
&char :: Point\ Weierstrass\ Affine\ e\ q\ r \rightarrow Natural \\
&char = q\_
\end{aligned}
$$

El cofactor:

$$
\begin{aligned}
&cof :: Point\ Weierstrass\ Affine\ e\ q\ r \rightarrow Natural \\
&cof = h\_
\end{aligned}
$$

Def comprueba si un punto dado cumple con la ecuación de la curva

$$
\begin{aligned}
&def :: Point\ Weierstrass\ Affine\ e\ q\ r \rightarrow Bool \\
&def\ O = True \\
&def\ (A\ x\ y) = y * y \equiv (x * x + a) * x + b \\
&\quad \textbf{where} \\
&\qquad a = a\_\ (witness :: WAPoint\ e\ q\ r) \\
&\qquad b = b\_\ (witness :: WAPoint\ e\ q\ r)
\end{aligned}
$$

La suma de puntos:

```
add ::
    Point Weierstrass Affine e q r →
    Point Weierstrass Affine e q r →
    Point Weierstrass Affine e q r
  add p O = p
  add O q = q
  add (A x1 y1) (A x2 y2)
    | x1 ≡ x2 ∧ y1 ≡ y2 = dbl (A x1 y1)
    | x1 ≡ x2 ∧ y1 ≡ (−y2) = O
    | otherwise = A x3 y3
    where
      l = (y2 − y1) / (x2 − x1)
      x3 = l ∗ l − x1 − x2
      y3 = l ∗ (x1 − x3) − y1
```

Y el doblaje

```
dbl :: Point Weierstrass Affine e q r →
       Point Weierstrass Affine e q r
dbl O       = O
dbl (A x y)
  | y ≡ 0 = O
  | otherwise = A x′ y′
  where
    a = a_ (witness :: WAPoint e q r)
    xx = x ∗ x
    l  = (xx + xx + xx + a) / (y + y)
    x′ = l ∗ l − x − x
    y′ = l ∗ (x − x′) − y
```

3.6.2 Ejemplo: SECP112R1

Se hizo un ejemplo con la curva elíptica SECP112R1 de la base de datos STD.

La curva SECT112R2 es una curva elíptica de 112 bits de un campo primo, dado por la ecuación:

$$y^2 = x^3 + ax + b$$

Y de parámetros

Cuadro 3.2 Parámetros de la curva SECP112R1

Nombre	Valor
p	0xdb7c2abf62e35e668076bead208b
a	0xdb7c2abf62e35e668076bead2088
b	0x659ef8ba043916eede8911702b22
G	(0x09487239995a5ee76b55f9c2f098, 0xa89ce5af8724c0a23e0e0ff77500)
n	0xdb7c2abf62e35e7628dfac6561c5
h	0x01

Fuente: Jancar y Sedlacek, 2020.

Realizado por: Reinoso, D., 2024.

Y se implementa de la siguiente manera:
Se crea el tipo da dato para nombre de la curva (un tipo sin habitantes).

$$\textbf{data}\ SECP112R1$$

Se crea el campo primo, para ello, se toma el número primo p de la base datos, en hexadecimal a nivel de tipo:

$$\textbf{type}\ Fq = Prime\ Q$$
$$\textbf{type}\ Q = 0\text{xdb7c2abf62e35e668076bead208b}$$

Se crea un campo primo, que representa el orden generado por el punto G de la curva.

$$\textbf{type}\ Fr = Prime\ R$$
$$\textbf{type}\ R = 0\text{xdb7c2abf62e35e7628dfac6561c5}$$

Los parámetros a y b de la curva.

$$\_a :: Fq$$

$$\_a = 0\text{xdb7c2abf62e35e668076bead2088}$$

$$\_b :: Fq$$

$$\_b = 0\text{x659ef8ba043916eede8911702b22}$$

El cofactor, característica y orden

$$\_h :: Natural$$

$$\_h = 0\text{x1}$$

$$\_q :: Natural$$

$$\_q = 0\text{xdb7c2abf62e35e668076bead208b}$$

$$\_r :: Natural$$

$$\_r = 0\text{xdb7c2abf62e35e7628dfac6561c5}$$

Y el punto generador

$$\_x :: Fq$$

$$\_x = 0\text{x9487239995a5ee76b55f9c2f098}$$

$$\_y :: Fq$$

$$\_y = 0\text{xa89ce5af8724c0a23e0e0ff77500}$$

$$gA :: PA$$

$$gA = A\ \_x\ \_y$$

Finalmente, se crea la instancia

El tipo que representa la curva

$$\textbf{type}\ PA = WAPoint\ SECP112R1\ Fq\ Fr$$

Instancia de curva prima

$$\textbf{instance}\ (Curve\ 'Weierstrass\ c\ SECP112R1\ Fq\ Fr) \Rightarrow WCurve\ c\ SECP112R1\ Fq\ Fr\ \textbf{where}$$

$$a\_ :: WPoint\ c\ SECP112R1\ Fq\ Fr \rightarrow Fq$$

$$a\_ = const\ \_a$$

$$b\_ :: WPoint\ c\ SECP112R1\ Fq\ Fr \rightarrow Fq$$

$$b\_ = const\ \_b$$

$$h\_ :: WPoint\ c\ SECP112R1\ Fq\ Fr \rightarrow Natural$$

$h\_ = const\ \_h$

$q\_ :: WPoint\ c\ SECP112R1\ Fq\ Fr \rightarrow Natural$

$q\_ = const\ \_q$

$r\_ :: WPoint\ c\ SECP112R1\ Fq\ Fr \rightarrow Natural$

$r\_ = const\ \_r$

E instancia de curva afín de Weierstrass

instance $WACurve\ SECP112R1\ Fq\ Fr$ **where**

$gA\_ :: WAPoint\ SECP112R1\ Fq\ Fr$

$gA\_ = gA$

4 Criptografía

4.1 Intercambio de clave de Diffie-Hellman

Se implementa en el módulo Math.Criptography.DiffieHellman del siguiente modo, primero se toma alguna curva elíptica en particular, por ejemplo la curva SECT113R1 definida antes, y se calcula un entero aleatorio en el intervalo de 1 y el orden menos uno del punto generador G.

$$
\begin{aligned}
&generate :: IO\ CurveSECT113R1.Fr \\
&generate = Fr.rnd
\end{aligned}
$$

Luego en la función $main$ se especifica el protocolo:

$$
\begin{aligned}
&main :: IO\ () \\
&main = \mathbf{do} \\
&\quad alice\_private \leftarrow generate \\
&\quad bob\_private \leftarrow generate
\end{aligned}
$$

Se generan dos enteros aleatorios para b para Bob y a para Alice.

$$
\begin{aligned}
&\mathbf{let}\ bob\_public :: CurveSECT113R1.PA \\
&\quad bob\_public = gen\ `mul`\ bob\_private \\
&\quad alice\_public :: CurveSECT113R1.PA \\
&\quad alice\_public = gen\ `mul`\ alice\_private
\end{aligned}
$$

Con el punto $P = G$ generador se calculan $P_b = bP$ y $P_a = aP$.

let *shared_secret1* = *bob_public* ‘*mul*‘ *alice_private*
shared_secret2 = *alice_public* ‘*mul*‘ *bob_private*

Se calcula el secreto compartido, $aP_b = abP$ y $bP_a = baP$.

putText `"Entero privado de Alice:"`
print alice_private
putText `"Punto de Alice:"`
print alice_public
putText `"Entero privado de Bob:"`
print bob_private
putText `"Punto de Bob:"`
print bob_public
putText `"Secreto compartido de Diffie Hellman"`
print shared_secret1
print (*shared_secret1* ≡ *shared_secret2*)
pure ()

Se imprimen los resultados y se devuelve la acción IO ().

4.1.1 Ejemplo:

Al evaluar el anterior código se obtiene (depende el entero aleatorio generado, por lo que cada ejecución va a ser diferente).
Entero privado de Alice a:

P 177817896987338438822221856187608 8

Punto de Alice $P_a = aP$.

A
(B 0 *b1100110110100000010111000110100100010101101110010110 01*
01110101010010001101001001010100101010101101001011100110 01)
(B 0 *b1000101101001110011001011111101100001001001110101010000 10*
11111011001110101000100100111010000011000010001010110 1)

Entero privado de Bob b:

$$P\ 46530642623131946657478878210913 84$$

Punto de Bob $P_b = bP$.

$$A$$
$$(B\ 0\ b1010101100010010101010011110101100110111110010010010$$
$$0$$
$$11000011011110001101000010001001101110110000101000000010100 1)$$
$$(B\ 0\ b1100100101010111101111000110100100000011011101010110111111$$
$$0$$
$$00011011011100101010001000011110000000111111111100110)$$

Secreto compartido abP.

$$A$$
$$(B\ 0\ b1111000110111001010000001011000100011100101110000$$
$$1$$
$$00100001001111110110101111100101101000000011011101110001101)$$
$$(B\ 0\ b1011001100010011111100101001111100110011101011001 1$$
$$0$$
$$10100110011111101110111100000001110001011010101100100011110 10)$$

4.2 Cifrado de ElGamal

Para el cifrado de ElGamal, se utilizará la misma curva SECT113R1 y se utilizará el embebimiento de mensaje de Menezes-Vanstone
Primero se genera un entero aleatorio para calcular la clave privada de Alice, mediante $generate$.

$$generate :: IO\ CurveSECT113R1.Fr$$
$$generate = Fr.rnd$$

Y en la función $main$, se calcula el entero aleatorio a de Alice.

$$main :: IO\ ()$$
$$main = \mathbf{do}$$
$$\quad alice\_private \leftarrow generate$$
$$\quad \mathbf{let}\ alice\_public :: CurveSECT113R1.PA$$
$$\quad\quad alice\_public = gen\ `mul`\ alice\_private$$

De esta forma la clave pública de Alice es $(E, \mathbb{F}_q, P, A)$ con $A = aP$, y E la curva SECT113R1. Mientras que la clave privada es el entero aleatorio a.
Para enviar un mensaje m, Bob lo codifica como un punto de $\mathbb{F}_q^2$. Mediante la función $encodeToF2$. Y se tiene el mensaje como (m_1, m_2).

$$
\begin{aligned}
&encodeToF2 :: String \rightarrow (F2n, F2n)\\
&encodeToF2\ text = (encodeToF\ s1, encodeToF\ s2)\\
&\quad \textbf{where}\ (s1, s2) = splitM\ text
\end{aligned}
$$

Luego, Bob calcula un entero aleatorio secreto k y calcula los puntos $M_1 = kP$ y $M_2 = kA$.

$$
\begin{aligned}
&bob\_randomInteger \leftarrow generate\\
&\textbf{let}\ bob\_public :: CurveSECT113R1.PA\\
&\quad bob\_public = gen\ `mul`\ bob\_randomInteger\\
&\quad bob\_public2 :: CurveSECT113R1.PA\\
&\quad bob\_public2 = alice\_public\ `mul`\ bob\_randomInteger
\end{aligned}
$$

donde gen es el punto generador P.
Luego, Bob toma las coordenadas (x, y) de M_2 y calcula $c_1 = xm_1$ y $c_2 = ym2$.

$$
\begin{aligned}
&\textbf{let}\ (m1, m2) = encodeToF2\ mensaje\\
&\quad (c1, c2) = (x * m1, y * m2)\\
&\qquad \textbf{where}\ x = xCor\ bob\_public2\\
&\qquad\quad y = yCor\ bob\_public2
\end{aligned}
$$

Finalmente, cifra su mensaje como (M_1, c_1, c_2) y se lo envía a Alice.

$$texto\_cifrado = (bob\_public, c1, c2)$$

Para descifrar el texto cifrado, Alice debe usar su clave privada a y calcular

$$aM_1 = akP = kaP = kA = M_2$$

toma las coordenas (x, y) de M_2 y calcula sus inversas $x_{-1}, y_{-1} \in F_q$, posteriormente recupera el texto cifrado mediante

$$x^{-1}c_1 = m_1 \qquad y^{-1}c_2 = m_2.$$

y decodifica el mensaje (m_1, m_2) para obtener el mensaje original m, mediante *descrifradoElGamal bob_public alice_private c1 c2* dado por la función.

```
descrifradoElGamal :: CurveSECT113R1.PA → CurveSECT113R1.Fr
    → F2n → F2n → String
descrifradoElGamal b2 key c1 c2 =
  case b2 `mul` key of
    (A x y) → decodeFromF2 (x' * c1, y' * c2)
      where
        x' = recip x
        y' = recip y
```

no se considera el caso que devuelva $\mathcal{O}$, ya que la clave no puede ser igual al orden del grupo generado.
Finalmente se imprimen los resultados

```
putText "Clave privada de Alice:"
print alice_private
putText "Clave pública de Alice:"
print alice_public
putText "Entero aleatorio de Bob:"
print bob_randomInteger
putText "Punto de Bob:"
print bob_public
putText "TextoCifrado:"
print bob_public
print c1
print c2
putText "TextoDescifrado:"
print (descrifradoElGamal bob_public alice_private c1 c2)
pure ()
```

4.2.1 Ejemplo:

Se va a especificar un mensaje previamente en una constante dada $mensaje$ (aunque también puede modificarse el código para que se pida al usuario)

$mensaje :: String$

$mensaje =$ `"Esto es una prueba"`

Y se obtiene:
Clave privada de Alice a.

P 724090196566470314656079629730645

Clave pública de Alice $A = aP$.

A
(B 0 *b1000111010011011010001000011110101110111011110001011010*
01100111011000000010011010111010101111111100011010100100010)
(B 0 *b1011000010010010101010000111111011001101011011001101110*
00001101001000000101100000001011111101111100010111110111110)

Entero aleatorio de Bob k.

P 347653941632692864199794733860083 4

Punto de Bob $M_1 = kP$.

A
(B 0 *b1000001101000101110000101101100001110001100101000101000*
10100111001000100101000101111110011001110111100011001001010)
(B 0 *b1001100001111010011011001101001000100000110010001111010*
10001100111111011111001010000011011001101100000001101111111)

Texto cifrado, (M_1, c_1, c_2)

A
(B 0 *b1000001101000101110000101101100001110001100101000101000*
10100111001000100101000101111110011001110111100011001001010)

(*B* 0 *b*1001100001111010011011001101001000100000110010001111011000110011111011111001010000001101100110110000000011011111111)

B 0 *b*1111100101101110100110101101101100101001110011111101000100011110001100110101011111110100011001101011010001011000

B 0 *b*1011110111000111010100101111110111110110011100100111011001010011110111000101111110000111111100010111111010111111001110

Texto descrifrado

```
"Esto es una prueba"
```

Bibliografía

[1] Tom M. Apostol. *Introduction to Analytic Number Theory*. Undergraduate Texts in Mathematics. Springer, 1976. ISBN: 978-1-4757-5579-4. DOI: 10.1007/978-1-4757-5579-4.

[2] Graham Ellis. *Rings and Fields*. Oxford University Press, 1993. ISBN: 9780198534556.

[3] Darrel Hankerson, Alfred Menezes y Scott Vanstone. *Guide to Elliptic Curve Cryptography*. 1.ª ed. Springer, 2004. ISBN: 978-0387952734. DOI: 10.1007/b97644. (Visitado 29-01-2024).

[4] Jeffrey Hoffstein, Jill Pipher y Silverman Joseph H. *An Introduction to Mathematical Cryptography*. 2.ª ed. Springer, 2014.

[5] Jan Jancar y Vladimir Sedlacek. *Standard curve database*. En línea. 2020. URL: https://neuromancer.sk/std/ (visitado 25-01-2024).

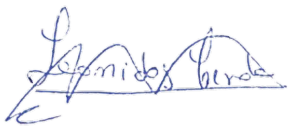
[6] Ricky Magner. *Finite Fields*. En línea. 2020. URL: http://math.bu.edu/people/rmagner/extras/FiniteFields.pdf (visitado 11-01-2024).

[7] Certicom Research. *Standards for Efficient Cryptography: SEC 2: Recommended Elliptic Curve Domain Parameters*. Standards for Efficient Cryptography Group. Mississauga, Ontario, Canada, 2000. URL: https://www.secg.org/SEC2-Ver-1.0.pdf (visitado 03-08-2024).

ESCUELA SUPERIOR POLITÉCNICA DE CHIMBORAZO

CERTIFICADO DE CUMPLIMIENTO DE LA GUÍA PARA NORMALIZACIÓN DE TRABAJOS DE FIN DE GRADO

Fecha de entrega: 17/ 07 / 2024

INFORMACIÓN DEL AUTOR
Nombres – Apellidos: Daniel Alejandro Reinoso Salas
INFORMACIÓN INSTITUCIONAL
Facultad: Ciencias
Carrera: Matemática
Título a optar: Matemático
 **Dr. Leonidas Antonio Cerda Romero, PhD.** **Director del Trabajo de Integración Curricular**  **Dr. Mario Humberto Paguay Cuvi, MSc.** **Asesor del Trabajo de Integración Curricular**