ALCP 2014-2015

Algebra Computacional

Explicaciones y comentarios al código

Francisco Criado Gallart

Índice

1. Introducción

El objetivo de este documento es complementar mi práctica de ALCP. Esto es así porque no me gusta poner comentarios excesivos en mi código, así que prefiero dejar las explicaciones a parte.

He hecho la práctica en Haskell, porque es un paradigma de programación mucho más cercano a los algoritmos que debemos implementar. Esto permite en algunos puntos un código muy limpio.

A cambio, ha sido un poco engorroso tener que pasar explícitamente las estructuras algebraicas sobre las que se trabaja cada vez.

2. Definitions

El primer módulo relevante es Definitions. Aquí se definen las estructuras y algunos algoritmos básicos que se usarán en toda la práctica.

En primer lugar, defino una estructura Dictionary, que contiene las operaciones de cada estructura algebraica. He definido cuerpos, dominios Euclídeos y dominios de factorización única. Para los dominios euclídeos está definida una división que sólo se garantiza que funciona al dividir por una unidad (O sea, cuando el grado es 1).

En este módulo también está el primer algoritmo importante: el algoritmo de Euclides extendido. Es importante que está definido para cualquier dominio Euclídeo. También hay algunos algoritmos similares derivados.

También en Definitions está la exponenciación rápida en cualquier anillo, y una pequeña función para calcular el orden de un elemento de cualquier anillo.

```haskell
data Dictionary t=
  Field {_zero::t, _one::t, (.==)::t->t->Bool, (.+)::t->t->t,
         (.-) ::t->t->t   , (.*) ::t->t->t   , (./)::t->t->t  }|

  Euclid{_zero::t, _one::t, (.==)::t->t->Bool, (.+)::t->t->t,
         (.-) ::t->t->t,    (.*) ::t->t->t   , (./)::t->t->t,
         _deg ::t->Integer, _div::t->t->(t,t)                 }|

  UFD   {_zero::t, _one::t, (.==)::t->t->Bool, (.+)::t->t->t,
         (.-) ::t->t->t,    (.*) ::t->t->t,
         _factor::t->(t,[(t,Integer)])                        }

eea :: Dictionary t -> t->t->(t,t,t)
eea euclid a b
  | b==zero   = (a, one, zero)
  | otherwise = let (q,r)  = div a b
                    (d,s,t)= eea euclid b r
                in  (d,t,s-(q*t))
  where Euclid zero one (==)(+)(-)(*)(/) deg div=euclid

gcd euclid a b = d where (d,_,_)=eea euclid a b
mcm euclid a b = fst$ _div euclid ((.*) euclid a b) (gcd euclid a b)

pow :: Dictionary d->d->Integer->d
pow ring a b=
  if b==0 then one  else α*α*(if b`P.mod`2 == 1 then a else one )
  where (*)=(.*) ring; one=_one ring
        α=pow ring a (b`P.div`2)

mul :: Dictionary d->d->Integer->d
mul ring a b=
  if b==0 then zero else α+α+(if b`P.mod`2 == 1 then a else zero)
  where (+)=(.+) ring; zero=_zero ring
        α=mul ring a (b`P.div`2)

order :: Dictionary  t->t->Integer
order ring elem=toInteger $ fromJust $ L.findIndex(==one) $ take
    100000 $iterate(*elem)elem
  where (*)=(.*)ring; one=_one ring; (==)=(.==)ring
```

3. Utilities

Utilities tiene funciones de "fontanería" de Haskell que he preferido redefinir.

También tiene una criba de Eratóstenes muy refinada, (Utilizando rotaciones y filtros sucesivos, junto con la evaluación perezosa). Usando esta criba, he implementado un pequeño algoritmo de factorización de enteros.

```haskell
module Utilities where
-- En este módulo se hacen cosas con enteros

length::[a]->Integer
length=toInteger. Prelude.length

replicate::Integer->a->[a]
replicate a= Prelude.replicate (fromInteger a)

take::Integer->[a]->[a]
take n= Prelude.take (fromInteger n)

drop::Integer->[a]->[a]
drop n= Prelude.drop (fromInteger n)

(!!)::[a]->Integer->a
a!!b=a Prelude.!! fromInteger b

-- Cálculo de primo. Rápido en la práctica, pero O(n**1.2)
primes::[Integer]
primes  = 2:([3,5..] `minus` foldt [[p*p,p*p+2*p..]|p<-primes_])
  where
    primes_ = 3:([5,7..] `minus` foldt [[p*p,p*p+2*p..]|p<-primes_])
    foldt ((x:xs):t) = x : union xs (foldt (pairs t))
    pairs ((x:xs):ys:t) = (x : union xs ys) : pairs t
    minus x y|x==[] || y==[] =x
             |otherwise= case compare (head x) (head y) of
                           LT->head x :minus (tail x) y
                           EQ->        minus (tail x) (tail y)
                           GT->        minus x        (tail y)
    union x y|x==[] =y |y==[] =x
             |otherwise = case compare (head x) (head y) of
                            LT-> head x : union (tail x) y
                            EQ-> head x : union (tail x) (tail y)
                            GT-> head y : union x        (tail y)

--Toma factorización cutre. Así se compensa una buena criba de
    primos.
factor::Integer->[(Integer,Integer)]
factor n=let l=[(p,maxexp n p)|p<-takeWhile (abs n>=) primes]
             maxexp n p|n`mod`p==0= 1+maxexp(n`div`p)p
                       |otherwise = 0
         in filter ((/=0).snd) l
```

4. Numbers, Fract y Gauss

Los dos primeros son módulos sencillos que no requieren mucha explicación. Uno representa a los enteros (de precisión arbitraria) como dominio euclídeo. Otro representa el cuerpo de fracciones asociado a un dominio euclídeo.

En cuanto a Gauss, representa a los enteros de Gauss como dominio euclídeo, usando como grado la norma euclídea. Para la división, hago la división de los complejos usual y busco de las cuatro opciones resultantes de sumar o no una unidad a cada coordenada, la que está mas cerca.

Esto automáticamente resuelve el EEA para los enteros de Gauss. Es la ventaja de la modularidad de mi práctica.

```
integer=Euclid {
 _zero=0::Integer,
 _one =1::Integer,
 (.==)=(==),
 (.+)=(+),
 (.-)=(-),
 (.*)=(*),
 (./)= \n m->let (c,r)=_div integer n m
             in assert (r==0) c,
 _deg= P.id,
 _div= \a b->(P.div a b, P.mod a b)
}
```

```
fract::Dictionary d->Dictionary (d,d)
fract euclid=Field{
  _zero=(zero,one),
  _one=(one,one),
  (.+) = \(a,b) (c,d)->reduce ((a*d)+(c*b),b*d),
  (.-) = \(a,b) (c,d)->reduce ((a*d)-(c*b),b*d),
  (.*) = \(a,b) (c,d)->reduce (a*c,b*d),
  (.==)= \(a,b) (c,d)->       (a*d)==(b*c),
  (./) = \(a,b) (c,d)->reduce (a*d,b*c)
 } where Euclid zero one (==)(+)(-)(*)(/) deg div =euclid
         reduce (a,b)=(a/d, b/d)
                      where d=gcd euclid a b
```

```
gaussOps=Euclid _zero _one (.==)(.+)(.-)(.*)(./) deg div where
  (==)=(P.==); (+)=(P.+); (-)=(P.-); (*)=(P.*)
  _one= (1,0); _zero=(0,0)
  z.==w        = z == w
  (a,b).+(c,d) = (a+c, b+d)
  (a,b).-(c,d) = (a-c, b-d)
  (a,b).*(c,d) = ((a*c)-(b*d), (b*c)+(a*d))
  z   ./ w = assert (deg w P.==1) z.*conj w
  deg w=fst $ w .* conj w
  div z w=
    let q=L.minimumBy (on compare (\u->norm (z.-(w.*u)))) options
        options=[(floor α,floor β).+(i,j)|i<-[0,1],j<-[0,1]]
        (α,β)=(fromInteger(fst$ z.*conj w) P./ fromInteger(norm w),
          fromInteger(snd$ z.*conj w) P./ fromInteger(norm w))
    in (q,z.-(q.*w))

norm (u,v) = (u P.* u) P.+ (v P.* v)
conj (u,v) = (u,0 P.-v)
```

5. Quotient

Quotient define la operación de tomar el módulo de un dominio euclídeo por un ideal generado por un elemento. Las operaciones son simplemente tomar restos tras cualquier operación. Para la inversa, se aplica el EEA para calcular el t válido. Es muy importante que esta versión puede calcular divisiones cuando estas son posibles. Para ello, manipulo un poco los resultados que da el EEA para que se resuelva la ecuación en general.

Este módulo en particular es lo que hará falta más adelante para definir los cuerpos finitos. En este caso, se toma $\mathbb{F}_p[x]/ < f >$ donde f es un polinomio irreducible.

También se define un resoluto del Teorema del resto chino. Éste funciona en cualquier dominio euclídeo cuando los módulos son primos entre sí.

He implementado, para el caso de los enteros, un resolutor que permite sistemas en que los módulos no son primos entre sí. Para esto, chineseSplit factoriza los módulos, descompone cada ecuación en un conjunto de ecuaciones de módulo potencia de un primo. Luego, chineseFilter elimina las ecuaciones redundantes comprobando que son coherentes. Por último, se resuelve el sistema con el resolutor general.

El motivo por el que implemento este resolutor es para resolver más adelante el logaritmo discreto por Pollard-rho.

```
mod::Dictionary d->d->Dictionary d
mod euclid m=Field{
    _zero= zero, _one=one,
    (.==)= \a b-> snd (div (a-b) m)==zero,
    (.+) = \a b-> snd $ div (a+b) m,
    (.-) = \a b-> snd $ div (a-b) m,
    (.*) = \a b-> snd $ div (a*b) m,
    (./) = \a b-> let g=gcd euclid a $ gcd euclid b m
                      a'=fst $ div a g
                      b'=fst $ div b g
                      m'=fst $ div m g
                      (d',s',t')=eea euclid b' m'
                  in reduce $ assert (deg d' P.== 1) (a'*s'/d')
  } where Euclid zero one (==)(+)(-)(*)(/) deg div=euclid
          reduce a=snd$div a m

isUnit euclid m=_deg euclid m ==1

chinese :: Dictionary d-> [(d,d)]->(d,d)
chinese euclid= foldr1 chinese2
  where chinese2 (x1,y1) (x2,y2)= reduce
          ((x1*y2*(./) (euclid`mod`y1) one y2)+
           (x2*y1*(./) (euclid`mod`y2) one y1), y1*y2)
        Euclid zero one (==)(+)(-)(*)(/) deg div=euclid
        reduce (a,b)=(snd$ a `div`b,b)

-- para euclid=integer, tengo uno a prueba de no-primos.
chineseInteger=chinese integer.chineseFilter.chineseSplit where

  chineseSplit::[(Integer,Integer)]->[[(Integer,Integer)]]
  chineseSplit l= Ext.groupWith (fst.head.factor.snd)$ L.sort $ l>>=
    (\(x,y)->[(snd$_div integer x $ pow integer b e,pow integer b e)
              |(b,e)<-factor y])

  chineseFilter::[[(Integer,Integer)]]->[(Integer,Integer)]
  chineseFilter l= map filterPrime l where
    filterPrime l=
      assert(all (\(x',y')->snd(_div integer x y')==x') l) (x,y)
      where (x,y)=last l
```

6. Polynomials

Polynomials es, obviamente, uno de los módulos más básicos de la práctica. Genera el domino euclídeo de polinomios asociado a un cuerpo. También está definido para dominios euclídeos, para poder más adelante implementar la factorización en $\mathbb{Z}$.

Básicamente se implementan las operaciones con los algoritmos elementales. Los polinomios se modelizan como listas de coeficientes, de mayor a menor grado.

Un detalle relevante es que para los polinomios, la función deg devuelve una unidad menos que el grado del polinomio. Esto es porque el grado como dominio euclídeo es diferente al grado del polinomio. Para mantener la práctica coherente con los algoritmos, aquí el grado es el grado euclídeo.

También se implementan algunas funciones que serán útiles más adelante, como puede ser la reducción de un polinomio a su equivalente mónico o la derivada.

```haskell
module Polynomials where
#include "Header.hs"

pol::Dictionary d->Dictionary [d]
pol (Euclid zero one(==)(+)(-)(*)(/)deg div)=
  pol (Field zero one(==)(+)(-)(*)(/))

pol field = Euclid _zero _one (.==)(.+)(.-)(.*)(./) _deg _div where
  Field zero one (==) (+) (-) (*) (/) = field

  _one=[one]; _zero=[]
  p.==q  = let l=length p P.- length q
           in  and $ zipWith (==) (pad(0 P.-l)++p) (pad l++q)
  p.+q   = let l=length p P.- length q
           in reduction$zipWith (+) (pad(0 P.-l)++p) (pad l++q)
  p.-q   = p .+ map (zero-) q
  p.*q   = if q.==_zero then [] else reduction$
              (map (*head q) p++pad (length q P.-1)) .+ (p.*tail q)
  p./q   = assert (r.==_zero) c
           where (c,r)=_div p q

  _deg p = length(reduction p)
  _div p q
    |q.==_zero = error "división por cero"
    |otherwise=(reduction (c.*[one/head q']),reduction r)
    where q'=reduction q
          (c,r)=divmonic p (q'.*[one/head q'])

  reduction = dropWhile (==zero)
  pad a=replicate a zero
  divmonic p q
    |q.==_zero     = error "division por cero"
    |_deg p<_deg q = (_zero,p)
    |otherwise     = (c.+(head p:pad),r)
    where (c,r)    = divmonic (p.-(map (*head p) q++pad)) q
          pad      = replicate (_deg p P.- _deg q) zero

monic field p = map (/head p) $ dropWhile (==zero) p
                where Field zero one (==)(+)(-)(*)(/)= field

derivate::Dictionary d->[d]->[d]
derivate ring p= reduce[mul ring (p!!i) (l-i-1) |i<-[0,1..l-2] ]
                 where l     = length p
                       reduce= dropWhile ((.==) ring (_zero ring))
```

7. FiniteField

El primero de los módulos interesantes, define los cuerpos finitos. Nótese cómo se construyen. Se enumeran los polinomios mónicos, se filtran los irreducibles (testeados con Rabin-Karp), y se elige uno válido. La definición gestiona automáticamente cualquier problema que pudiera surgir.

Los elementos del cuerpo finito, por tanto, se representan como listas de elementos de $\mathbb{Z}$, de acuerdo con la base que define el polinomio dado. Esto es lo que menos me gusta de mi práctica ya que las definiciones posteriores de polinomios se hacen bastante engorrosas.

El test de irreducibilidad es básicamente el mismo código que para los polinomios en $\mathbb{F}_q[x]$, por el algoritmo de Rabin-Karp. Se pide que en el candidato a cuerpo, el polinomio $x^{p^{degf}} = 1$, y que no ocurra lo mismo con ningún grado menor.

```haskell
module FiniteField where
#include "Header.hs"
import Quotient
import Polynomials
import Numbers

finite p n=assert (LO.member p primes) $
            pol zp `mod`(head$filter(irred p) $enumPol p n)
           where zp= integer`mod`p

enumList p 1=[[c]|c<-[0..p P.-1]]
enumList p n=[c:q|c<-[0..p P.-1],q<-enumList p (n P.-1)]

enumPol::Integer->Integer->[[Integer]]
enumPol p n=[1:q|q<-enumList p n]

--Test de irreducibilidad en Zq, p primo
irred::Integer->[Integer]->Bool
irred p f=
  zero == h n &&
   all (isUnit (pol field).gcd (pol field) f.h )primedivs
  where
    n=_deg (pol field) f P.- 1
    field@(Field _zero _one (.==)(.+)(.-)(.*)(./))=integer `mod` p
    Field zero one (==)(+)(-)(*)(/) =pol field`mod`f
    h n_i=pow (pol field`mod`f) x (pow integer p n_i) - x
    x=[_one,_zero] --polinomio x
    primedivs=[n `P.div` p_i| p_i<-(fst.unzip.factor) n]

--TODO pretty-print de la tabla de multiplicar
printtable p n=
  let Field zero one (==)(+)(-)(*)(/)=finite p n
      trad a=if null a then 0
                       else (p P.* (trad$tail a)) P.+ head a
  in [map (trad.reverse)
       [(a*b)/b|a<-tail$enumList p n]|b<-tail$enumList p n]
```

8. Factorization

Aquí se implementa uno de los algoritmos centrales de la práctica: Cantor-Zasenhaus. También se implementa Rabin-Karp para cuerpos finitos, pero lo he quitado de este documento para simplificarlo.

Tal y como se describe en [?], se descompone en tres partes:

1. En la primera parte, se descompone el polinomio en factores libres de cuadrados, por el procedimiento de dividir sucesivamente por el mcd entre el polinomio y la derivada.

   Cuando la derivada se anula, hay que tener un poco de cuidado. En este caso, se puede hacer la sustitución $x := x^{1/p}$ y seguir iterando el algoritmo.

2. En la segunda parte, se descompone el polinomio en factores de tal forma que cada uno de esos factores se descomponga en irreducibles del mismo grado. Este paso es muy sencillo usando el resultado de que el producto de todos los polinomios irreduciles de grado divisor de k es $x^{q^k} - x$. De esta forma, se realizan divisiones sucesivas por el mcd entre el polinomio dado y $x^{q^k} - x$. Como esto sería prohibitivamente lento, se aplica la optimización adicional de considerar las operaciones módulo f.

```haskell
type Pol=[[Integer]]

irred::Integer->Integer->Pol->Bool
irred p n f= --Test de irreducibilidad (Rabin) en Fq, q=p**n, p primo
  zero == h d &&
   all (isUnit (pol field).gcd (pol field) f.h) primedivs
  where
    d=_deg (pol field) f P.- 1
    field@(Field _zero _one (.==)(.+)(.-)(.*)(./))=finite p n
    Field zero one (==)(+)(-)(*)(/) =pol field`mod`f
    h n_i=pow (pol field`mod`f) x (pow integer p (n_i P.* n) ) - x
    x=[_one,_zero] --polinomio x
    primedivs=[d `P.div` p_i| p_i<-(fst.unzip.factor) d]

factorPol p n f= fact1 p n f >>= fact2 p n >>= fact3 p n

fact1::Integer->Integer->Pol->[Pol]
fact1 p n f
  | deg f P.<=1 = [f]
  | f' == zero  = concat $ replicate p $ fact1 p n (unstride f)
  | deg g P.==1 = [f]
  | otherwise   = f2:fact1 p n g
  where Euclid zero one (==)(+)(-)(*)(/)deg div= pol(finite p n)
        f'= derivate (finite p n) f
        g = monic(finite p n) $ gcd (pol(finite p n)) f f'
        f2= (./) (pol$ finite p n) f g
        unstride f=reverse[reverse f!!i| i<-[0,p..deg f]]

fact2::Integer->Integer->Pol->[(Pol,Integer)]
fact2 p n f=
  aux 1 f (pow (pol field`mod`f) [[1],[0]]  (p^n))
  where Euclid zero one (==)(+)(-)(*)(/) deg div= pol field
        field= finite p n
        aux d f xq
          | deg f P.==1= []
          | deg g P./=1= (g, d):aux d f' xq
          | otherwise  =        aux (d P.+1) f xq'
          where g  = monic field$ gcd (pol field) f (xq -[[1],[0]])
                xq'= pow (pol field `mod` f) xq (p^n)
                f' = f/g
```

3. El tercer paso es el algoritmo de Cantor-Zassenhaus propiamente dicho. Es un algoritmo probabilista tipo Las Vegas en que se aplica la idea de que $\mathbb{F}_q[x]/f$ es producto directo de los cuerpos análogos para los divisores. Puesto que sabemos cuantos elementos hay en cada subanillo, la probabilidad de que tomando un elemento h al azar, g divida alguno de los factores es aproximadamente 1/2.

   Los algoritmos probabilistas en Haskell tienen una cierta dificultad de programación añadida por la transparencia referencial. Por esto, la generación de números aleatorios se realiza en una lista de elementos que se van usando como testigos.

```
fact3::Integer->Integer->(Pol,Integer)->[Pol]
fact3 p n (f,d)= assert(p P./= 2) $ fact [f] pols
  where euclid@(Euclid zero one(==)(+)(-)(*)(/)deg div)=
    pol$finite p n

        fact factors pols
          | length factors P.==r = factors
          | otherwise = fact (concatMap split factors) pols'
          where (h:pols')= pols
                g=pow (euclid`mod`f) h exp-one
                exp=(p P.^(n P.*d) P.-1) `P.div` 2
                split f| m==one || m==f = [f]
                       | otherwise      = [f/m,m]
                      where m=monic(finite p n)$ gcd euclid f
    g

        l=deg f P.- 1; r=l `P.div` d

        pols= filter (/=zero)$ map (take l) $ iterate (drop l)
          $map(take n)$ iterate (drop n)$ map (getRandom p)
    [1..]
        getRandom n i =unsafePerformIO $ do
          g<-getStdGen; return $ randomRs (0,n P.-1) g !!i
```

9. Miller-Rabin y AKS

Estos son dos algoritmos de testeo de primalidad, uno probabilista tipo Las Vegas y otro determinista. Los muestro juntos porque es interesante compararlos.

El algoritmo de Miller-Rabin se basa en un lema sobre las raices cuadradas de la unidad. Si $x^2 \equiv 1 \mod p$, entonces, $(x+1)*(x-1) \equiv 0 \mod p$. Por tanto, $x \equiv \pm 1 \mod p$. Entonces, si n es primo impar, $n-1 = 2^r * s$ es par (con s impar). Entonces, dado un $a \in \mathbb{Z}/ < n >$ se tiene que $a^d \equiv 1 \mod n$ o $a^{2^t * d} \equiv -1 \mod n$ para algún $0 \leq r \leq s-1$. Entonces, simplemente se prueba con testigos aleatorios. La probabilidad de que eun testigo aleatorio demuestre la reducibilidad de un compuesto es al menos 3/4. Con suficientes pruebas se reduce muy rápido la probabilidad de que un número sea primo.

Sin embargo, el problema de encontrar un algoritmo en P (es decir, determinista y polinómico) para comprobar la primalidad se resuelve por primera vez con AKS. Se trata de que en $\mathbb{Z}_n$ se cumple el "sueño del novato" si a es coprimo con n: $(x-a)^n \equiv (x^n) - a \mod n$

Para hacer la prueba de forma eficiente, se toma módulo un cierto polinomio de la forma $x^r - 1$. La clave en AKS es encontrar una cota para ese r.

Aunque el AKS es determinista y polinómico, se porta realmente mal en comparación con Miller-Rabin. Hacen falta mejoras sugeridas por Lenstra entre otros para hacer que sea minimamente práctico.

```
millerRabin::Integer->Integer->Bool
millerRabin k n=
  all test $ take k $ map (getRandom n) [1..]
  where (s,r)=div2 (n-1)
        test t| t_s==1    = True
              | otherwise= elem (n-1) $ take r $
                           iterate (\a->(a*a)`P.mod`n) t_s
              where t_s  = (t^s) `P.mod` n

        div2 n| n`P.mod`2==0 = (n',s'+1)
              | otherwise= (n,0)
              where (n',s')=div2 (n`div`2)
        getRandom n i =unsafePerformIO $ do
          g<-getStdGen; return $ randomRs (1,n-1) g !!i

-- (take 30 ( filter (millerRabin 30) [2..]))
```

```
aks::Integer->Bool --isPrime
aks n
  | ispow (fromInteger n)                                  = False
  | any (inRange (2,n P.-1).gcd integer n)[2..r]           = False
  | n<=r                                                   = True
  | any (condition.round)[1..sqrt(fromIntegral $ ϕ r)*l]= False
  | otherwise                                              = True
  where ispow n=
          any (\b->let a=n**(1/b) in a==fromInteger(round a)) [2..l]
        r= toInteger $ fromJust $ L.find (\a->P.gcd n a==1
             && fromInteger(order(integer `mod` a) n) > l^2) [2..]
        ϕ= product . map(\(p,n)->(p-1)*(p^(n-1))) . factor
        l= logBase 2 $ fromInteger n
        condition a=
          pow eucl (x+[a]) n /= (pow eucl x n + [a])
          where x=[1,0]
                zn=integer `mod` n
                eucl@(Field zero one (==)(+)(-)(*)(/))= pol zn`mod`m
                m=(.-) (pol zn) (pow (pol zn) x r) (_one (pol zn))
```

10. Logarithm

Como el enunciado no especificaba el algoritmo para el logaritmo discreto, he elegido el algoritmo rho de Pollard, que se basa en una idea que ya conocía para la factorización de enteros.

El algoritmo genera congruencias que debe cumplir el logaritmo buscado. Estas congruencias son módulo divisores de $q-1$, lo que me obligó a implementar un Teorema chino del resto a prueba de ecuaciones no primas entre sí.

La idea es que si buscamos γ con $\alpha^\gamma = \beta$, podemos reducir el problema a encontrar (a,b,a',b') con $\alpha^a * \beta^b = \alpha^{a'} * \beta^{b'}$, y luego resolver una ecuación simple. Estos (a,b,a',b') se encuentran creando una sucesión $\alpha^{a_i} * \beta^{b_i}$. Si la secuencia es determinista, entrará en un bucle y tendremos dos pares de números que verifiquen la propiedad.

Entonces, creo una sucesión determinista sobre los exponentes para encontrar ese punto en que se repite, usando el algoritmo de Floyd de la tortuga y la liebre. Se trata de iterar por una parte f y $f \circ f$ hasta que lleguen al mismo punto.

```haskell
module Logarithm  where
#include "Header.hs"
import Quotient
import Polynomials
import Numbers
import FiniteField

logarithm::Show d=>Dictionary d->d->d->(Integer,Integer)
logarithm field α β=
  chineseInteger$ [0..12] >>= (\u->logarithmEq field u α β)

logarithmEq::Show d=>Dictionary d->Integer->d->d->[(Integer,Integer)]
logarithmEq field seed α β=
  end $ loop $ zip (iterate f (one,0,0)) (iterate (f.f) (one,0,0))

  where Field zero one (==)(+)(-)(*)(/)=field
        n= order field α

        f (x,a,b)| opt P.==0 = (α*x, 1 P.+a,       b)
                 | opt P.==1 = (x*x, 2 P.*a, 2 P.* b)
                 | opt P.==2 = (β*x,      a, 1 P.+ b)
                 where opt= hash seed x `P.mod` 3

        loop= fromJust. L.find (\((x,_,_),(y,_,_))-> x==y).tail

        end ((_,a,b),(_,a',b'))=
          [((./) (integer`mod`n') a'' b'' ,n')| b'' /=zero]
          where n' = n     /g
                a''= (a'-a)/g
                b''= (b-b')/g
                g=gcd integer (a'-a) $ gcd integer (b-b') n
                Euclid zero one (==)(+)(-)(*)(/) deg div=integer

hash seed x= pow (integer`mod`541) (seed+1) (tonum$show x)
             where tonum= foldr (\u v->v*256+toInteger(ord u)) 0
```

11. Hensel

En este módulo se aplica el lema de Hensel para factorizar un f libre de cuadradosen $\mathbb{Z}[x]$. Para ello se sigue un proceso en cinco pasos explicado en [?]: primero, se elige un p de tal forma que no se anule el coeficiente principal y que siga siendo libre de cuadrados. El segundo paso es factorizar f en ese cuerpo finito.

El tercer paso es elegir un N de tal forma que una factorización en $\mathbb{Z}/ < p^N >$, tenga todos los coeficientes de la factorización real. Esto se logra con las cotas de Mignotte.

El cuarto paso es elevar esa factorización usando Hensel. Es relativamente directo elevar una factorización de dos polinomios. Hace falta tener un poco de cuidado para extender esa idea a varios.

Por último, se recombinan los factores. Yo en realidad calculo los divisores, compruebo cuales son divisores reales, y luego identifico los factores. No se puede hacer mucho más eficiente, ya que este algoritmo es exponencial.

```haskell
--Paso 1
primeMod::[Integer]->Integer
primeMod f=
  fromJust$ L.find (test f) primes
  where test f p=
          (head f`P.mod`p P./=0) && (f'/=[]) && (m==[1])
          where f'=derivate (integer`mod`p) f
                m=monic (integer`mod`p)$
                    gcd (pol(integer`mod`p)) f f'
                (==)=(.==) (pol(integer`mod`p))

--Paso 3: devuelve el N que debe sobrepasar Hensel
mignotteBound::Integer->[Integer]->Integer
mignotteBound p f=
  ceiling $ logBase (fromInteger p)  bound
  where norm= sqrt$ sum$ map(\a->fromInteger (a*a)) f
        bound= 2.0^length f*norm

--Paso 4: devuelve, en el mismo formato, el hensel Lift, de n a n2̂
henselLift n (f,g,h,s,t)=
  assert ((.==) (pol(integer`mod`n)) f (g'*h')) $
  assert ((.==) (pol(integer`mod`n)) one ((s'*g')+(t'*h'))) $
  assert (f==(g'*h')) $
  assert (one==((s'*g')+(t'*h'))) $
  (f,g',h',s',t')
  where (g',h')= ((g*(one+q))+(t*δ), h+r)
                 where δ=f-(g*h)
                       (q,r)=(s*δ) `div` h
        (s',t')= (s-r, ((one-ϵ)*t)-(g'*q))
                 where ϵ=(s*g')+(t*h')-one
                       (q,r)=(s*ϵ) `div` h'

        Euclid zero one(==)(+)(-)(*)(/) deg div=pol (integer`mod`(n P
    .*n))

liftTo p pN (f,g,h,s,t)
  | p>=pN  = (f,g,h,s,t)
  | otherwise = liftTo (p*p) pN $ henselLift p (f,g,h,s,t)

products m []=[([1],[1])]
products m (x:xs)=
  [(trueRepr m $a*c, trueRepr m $ b*d)
   |(a,b)<-[(one,x),(x,one)],(c,d)<-products m xs]
  where Euclid zero one(==)(+)(-)(*)(/)deg div=pol$(integer`mod`m)
```

El punto más delicado del algoritmo es extender la elevación a los múltiples factores. Hay que considerar la factorización como un producto entre el primero y los demás, luego, tras la elevación, se repite el proceso, indicando como polinomio producto el segundo resultado de la elevación.

Este proceso se repite tantas veces como indique la cota de Mignotte.

```
--Paso 5: recombina los factores como buenamente puede
recombine::[Integer]->Integer->[[Integer]]->[[Integer]]
recombine f m factors=
  [real_divs!!(2^i)
   |i<-[0.. ceiling(logBase 2 $ fromInteger$ length real_divs)P.-1]]
  where real_divs=concatMap (\(a,b)->[a|(a*b)==f]) divs
        divs=products m factors
        Euclid zero one(==)(+)(-)(*)(/)deg div=pol$integer

--Devuelve los coeficientes al representante mas cercano a 0
trueRepr n= map (\a-> if a<=(n`div`2) then a else a-n)

factorPolInt::[Integer]->[[Integer]]
factorPolInt f=
  recombine f pN lifted
  where p=primeMod f
        n=mignotteBound p f
        pN=p^n

        factors=map(map to_zp) $factorPol p 1(monic fp (map (:[]) f))
                where to_zp []= 0; to_zp [a]=a
                      fp=finite p 1

        lifted::[[Integer]]
        lifted=map (trueRepr pN) $ liftTo p pN f factors

        liftTo m pN f factors
          | m>=pN = factors
          | otherwise = liftTo (m*m) pN f $ liftallStep m f factors

        liftallStep::Integer->[Integer]->[[Integer]]->[[Integer]]
        liftallStep m f [] = []
        liftallStep m f (g:gs)=
          g':liftallStep m h' gs
          where (d,s,t)= eea(pol$ integer`mod`m) g h
                (_,g',h',_,_)=henselLift m (f,g,h,s/d,t/d)
                (/)=(./) (pol$ integer`mod`m)
                h=foldr ((.*) (pol$ integer`mod`m))
                   (_one (pol$ integer`mod`m)) gs
```