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 y Fract

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))
```

Alejandro Aguirre Galindo
Luis María Costero Valero
Jesús Doménech Arellano