

Finite fields and functional reconstructions

Ray D. Sameshima

2016/09/23 ~ 2017/01/23 18:46

Contents

0	Preface	5
0.1	References	5
0.2	Set theoretical gadgets	5
0.2.1	Numbers	5
0.2.2	Algebraic structures	6
0.3	Haskell language	6
1	Basics	9
1.1	Finite fields	9
1.1.1	Rings	9
1.1.2	Fields	10
1.1.3	An example of finite rings \mathbb{Z}_n	10
1.1.4	Bézout’s lemma	11
1.1.5	Greatest common divisor	11
1.1.6	Extended Euclidean algorithm	13
1.1.7	Coprime as a binary relation	16
1.1.8	Corollary (Inverses in \mathbb{Z}_n)	16
1.1.9	Corollary (Finite field \mathbb{Z}_p)	17
1.2	Rational number reconstruction	19
1.2.1	A map from \mathbb{Q} to \mathbb{Z}_p	19
1.2.2	Reconstruction from \mathbb{Z}_p to \mathbb{Q}	21
1.2.3	Chinese remainder theorem	25
1.2.4	<code>recCRT</code> : from image in \mathbb{Z}_p to rational number	28
1.3	Polynomials and rational functions	29
1.3.1	Notations	29
1.3.2	Polynomials and rational functions	30
1.3.3	As data, coefficients list	31
1.4	Haskell implementation of univariate polynomials	31
1.4.1	A polynomial as a list of coefficients	31

1.4.2	Difference analysis	35
2	Functional reconstruction over \mathbb{Q}	39
2.1	Univariate polynomials	39
2.1.1	Newton's polynomial representation	39
2.1.2	Towards canonical representations	40
2.1.3	Simplification of our problem	41
2.2	Univariate polynomial reconstruction with Haskell	43
2.2.1	Newton interpolation formula with Haskell	43
2.2.2	Stirling numbers of the first kind	44
2.2.3	<code>list2pol</code> : from output list to canonical coefficients	46
2.3	Univariate rational functions	47
2.3.1	Thiele's interpolation formula	48
2.3.2	Towards canonical representations	49
2.4	Univariate rational function reconstruction with Haskell	49
2.4.1	Reciprocal difference	49
2.4.2	<code>tDegree</code> for termination	50
2.4.3	<code>thieleC</code> : from output list to Thiele coefficients	52
2.4.4	Haskell representation for rational functions	53
2.4.5	<code>list2rat</code> : from output list to canonical coefficients	57
2.5	Multivariate polynomials	57
2.5.1	Foldings as recursive applications	58
2.5.2	Experiments, 2 variables case	58
2.5.3	Haskell implementation, 2 variables case	61
2.6	Multivariate rational functions	63
2.6.1	The canonical normalization	63
2.6.2	An auxiliary t	64
2.6.3	Experiments, 2 variables case	64
2.6.4	Haskell implementation, 2 variables case	67
3	Functional reconstruction over finite fields	69
3.1	Univariate polynomials	69
3.1.1	Pre-cook	69
3.1.2	Difference analysis on \mathbb{Z}_p	70
3.1.3	Eager and lazy degree	71
3.1.4	Term by term reconstruction	72
3.1.5	<code>list2polZp</code> : from the output list to coefficient lists	73
3.2	TBA Univariate rational functions	74

4	Codes	75
4.1	Ffield.lhs	75
4.2	Polynomials.hs	80
4.3	Univariate.lhs	82
4.4	Multivariate.lhs	95
4.5	FR0verZp.lhs	105

Chapter 0

Preface

0.1 References

1. Scattering amplitudes over finite fields and multivariate functional reconstruction
(Tiziano Peraro)
<https://arxiv.org/pdf/1608.01902.pdf>
2. Haskell Language
www.haskell.org
3. The Haskell Road to Logic, Maths and Programming
(Kees Doets, Jan van Eijck)
<http://homepages.cwi.nl/~jve/HR/>
4. Introduction to numerical analysis
(Stoer Josef, Bulirsch Roland)

0.2 Set theoretical gadgets

0.2.1 Numbers

Here is a list of what we assumed that the readers are familiar with:

1. \mathbb{N} (Peano axiom: \emptyset, succ)
2. \mathbb{Z}
3. \mathbb{Q}

4. \mathbb{R} (Dedekind cut)
5. \mathbb{C}

0.2.2 Algebraic structures

1. Monoid: $(\mathbb{N}, +), (\mathbb{N}, \times)$
2. Group: $(\mathbb{Z}, +), (\mathbb{Z}, \times)$
3. Ring: \mathbb{Z}
4. Field: \mathbb{Q}, \mathbb{R} (continuous), \mathbb{C} (algebraic closed)

0.3 Haskell language

From "A Brief, Incomplete and Mostly Wrong History of Programming Languages":¹

1990 - A committee formed by Simon Peyton-Jones, Paul Hudak, Philip Wadler, Ashton Kutcher, and People for the Ethical Treatment of Animals creates Haskell, a pure, non-strict, functional language. Haskell gets some resistance due to the complexity of using monads to control side effects. Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors" which is the problem?"

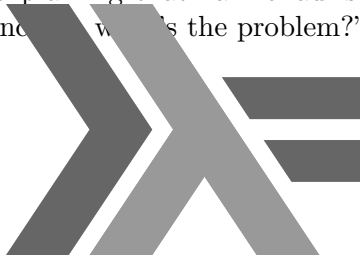


Figure 1: Haskell's logo, the combinations of λ and monad's bind $>>=$.

Haskell language is a standardized purely functional declarative statically typed programming language.

In declarative languages, we describe "what" or "definition" in its codes, however imperative languages, like C/C++, "how" or "procedure".

¹ <http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html>

Functional languages can be seen as 'executable mathematics'; the notation was designed to be as close as possible to the mathematical way of writing.²

Instead of loops, we use (implicit) recursions in functional language.³

```
> sum :: [Int] -> Int
> sum []      = 0
> sum (i:is) = i + sum is
```

² Algorithms: A Functional Programming Approach (Fethi A. Rabhi, Guy Lapalme)

³Of course, as a best practice, we should use higher order function (in this case **foldr** or **foldl**) rather than explicit recursions.

Chapter 1

Basics

We have assumed living knowledge on (axiomatic, i.e., ZFC) set theory, algebraic structures.

1.1 Finite fields

Ffield.lhs

<https://arxiv.org/pdf/1608.01902.pdf>

```
> module Ffield where  
  
> import Data.Ratio  
> import Data.Maybe  
> import Data.Numbers.Primes
```

1.1.1 Rings

A ring $(R, +, *)$ is a structured set R with two binary operations

$$(+)\ ::\ R\ \rightarrow\ R\ \rightarrow\ R \tag{1.1}$$

$$(*)\ ::\ R\ \rightarrow\ R\ \rightarrow\ R \tag{1.2}$$

satisfying the following 3 (ring) axioms:

1. $(R, +)$ is an abelian, i.e., commutative group, i.e.,

$$\forall a, b, c \in R, (a + b) + c = a + (b + c) \quad (\text{associativity for } +) \quad (1.3)$$

$$\forall a, b \in R, a + b = b + a \quad (\text{commutativity}) \quad (1.4)$$

$$\exists 0 \in R, \text{ s.t. } \forall a \in R, a + 0 = a \quad (\text{additive identity}) \quad (1.5)$$

$$\forall a \in R, \exists (-a) \in R \text{ s.t. } a + (-a) = 0 \quad (\text{additive inverse}) \quad (1.6)$$

2. $(R, *)$ is a monoid, i.e.,

$$\forall a, b, c \in R, (a * b) * c = a * (b * c) \quad (\text{associativity for } *) \quad (1.7)$$

$$\exists 1 \in R, \text{ s.t. } \forall a \in R, a * 1 = a = 1 * a \quad (\text{multiplicative identity}) \quad (1.8)$$

3. Multiplication is distributive w.r.t addition, i.e., $\forall a, b, c \in R$,

$$a * (b + c) = (a * b) + (a * c) \quad (\text{left distributivity}) \quad (1.9)$$

$$(a + b) * c = (a * c) + (b * c) \quad (\text{right distributivity}) \quad (1.10)$$

1.1.2 Fields

A field is a ring $(\mathbb{K}, +, *)$ whose non-zero elements form an abelian group under multiplication, i.e., $\forall r \in \mathbb{K}$,

$$r \neq 0 \Rightarrow \exists r^{-1} \in \mathbb{K} \text{ s.t. } r * r^{-1} = 1 = r^{-1} * r. \quad (1.11)$$

A field \mathbb{K} is a finite field iff the underlying set \mathbb{K} is finite. A field \mathbb{K} is called infinite field iff the underlying set is infinite.

1.1.3 An example of finite rings \mathbb{Z}_n

Let $n(> 0) \in \mathbb{N}$ be a non-zero natural number. Then the quotient set

$$\mathbb{Z}_n := \mathbb{Z}/n\mathbb{Z} \quad (1.12)$$

$$\cong \{0, \dots, (n-1)\} \quad (1.13)$$

with addition, subtraction and multiplication under modulo n is a ring.¹

¹ Here we have taken an equivalence class,

$$0 \leq k \leq (n-1), [k] := \{k + n * z | z \in \mathbb{Z}\} \quad (1.14)$$

1.1.4 Bézout's lemma

Consider $a, b \in \mathbb{Z}$ be nonzero integers. Then there exist $x, y \in \mathbb{Z}$ s.t.

$$a * x + b * y = \gcd(a, b), \quad (1.19)$$

where \gcd is the greatest common divisor (function), see §1.1.5. We will prove this statement in §1.1.6.

1.1.5 Greatest common divisor

Before the proof, here is an implementation of \gcd using Euclidean algorithm with Haskell language:

```
> -- Euclidian algorithm.
> myGCD :: Integral a => a -> a -> a
> myGCD a b
>   | b < 0 = myGCD a (-b)
> myGCD a b
>   | a == b = a
>   | b > a = myGCD b a
>   | b < a = myGCD (a-b) b
```

Example, by hands

Let us consider the \gcd of 7 and 13. Since they are primes, the \gcd should be 1. First it binds a with 7 and b with 13, and hit $b > a$.

$$\text{myGCD } 7 \ 13 == \text{myGCD } 13 \ 7 \quad (1.20)$$

Then it hits main line:

$$\text{myGCD } 13 \ 7 == \text{myGCD } (13-7) \ 7 \quad (1.21)$$

with the following operations:

$$[k] + [l] := [k + l] \quad (1.15)$$

$$[k] * [l] := [k * l] \quad (1.16)$$

This is equivalent to take modular n :

$$(k \bmod n) + (l \bmod n) := (k + l \bmod n) \quad (1.17)$$

$$(k \bmod n) * (l \bmod n) := (k * l \bmod n). \quad (1.18)$$

In order to go to next step, Haskell evaluate $(13 - 7)$,² and

$$\text{myGCD } (13-7) \ 7 \ == \ \text{myGCD } 6 \ 7 \quad (1.22)$$

$$\quad \quad \quad == \ \text{myGCD } 7 \ 6 \quad (1.23)$$

$$\quad \quad \quad == \ \text{myGCD } (7-6) \ 6 \quad (1.24)$$

$$\quad \quad \quad == \ \text{myGCD } 1 \ 6 \quad (1.25)$$

$$\quad \quad \quad == \ \text{myGCD } 6 \ 1 \quad (1.26)$$

Finally it ends with 1:

$$\text{myGCD } 1 \ 1 \ == \ 1 \quad (1.27)$$

As another example, consider 15 and 25:

$$\text{myGCD } 15 \ 25 \ == \ \text{myGCD } 25 \ 15 \quad (1.28)$$

$$\quad \quad \quad == \ \text{myGCD } (25-15) \ 15 \quad (1.29)$$

$$\quad \quad \quad == \ \text{myGCD } 10 \ 15 \quad (1.30)$$

$$\quad \quad \quad == \ \text{myGCD } 15 \ 10 \quad (1.31)$$

$$\quad \quad \quad == \ \text{myGCD } (15-10) \ 10 \quad (1.32)$$

$$\quad \quad \quad == \ \text{myGCD } 5 \ 10 \quad (1.33)$$

$$\quad \quad \quad == \ \text{myGCD } 10 \ 5 \quad (1.34)$$

$$\quad \quad \quad == \ \text{myGCD } (10-5) \ 5 \quad (1.35)$$

$$\quad \quad \quad == \ \text{myGCD } 5 \ 5 \quad (1.36)$$

$$\quad \quad \quad == \ 5 \quad (1.37)$$

Example, with Haskell

Let us check simple example using Haskell:

```
*Ffield> myGCD 7 13
1
*Ffield> myGCD 7 14
7
*Ffield> myGCD (-15) (20)
5
*Ffield> myGCD (-299) (-13)
13
```

² Since Haskell language adopts lazy evaluation, i.e., call by need, not call by name.

The final result is from

```
*Ffield> 13*23
299
```

1.1.6 Extended Euclidean algorithm

Here we treat the extended Euclidean algorithm, this is a constructive solution for Bézout's lemma.

As intermediate steps, this algorithm makes sequences of integers $\{r_i\}_i$, $\{s_i\}_i$, $\{t_i\}_i$ and quotients $\{q_i\}_i$ as follows. The base cases are

$$(r_0, s_0, t_0) := (a, 1, 0) \quad (1.38)$$

$$(r_1, s_1, t_1) := (b, 0, 1) \quad (1.39)$$

and inductively, for $i \geq 2$,

$$q_i := \text{quot}(r_{i-2}, r_{i-1}) \quad (1.40)$$

$$r_i := r_{i-2} - q_i * r_{i-1} \quad (1.41)$$

$$s_i := s_{i-2} - q_i * s_{i-1} \quad (1.42)$$

$$t_i := t_{i-2} - q_i * t_{i-1}. \quad (1.43)$$

The termination condition³ is

$$r_k = 0 \quad (1.44)$$

for some $k \in \mathbb{N}$ and

$$\gcd(a, b) = r_{k-1} \quad (1.45)$$

$$x = s_{k-1} \quad (1.46)$$

$$y = t_{k-1}. \quad (1.47)$$

Proof

By definition,

$$\gcd(r_{i-1}, r_i) = \gcd(r_{i-1}, r_{i-2} - q_i * r_{i-1}) \quad (1.48)$$

$$= \gcd(r_{i-1}, r_{i-2}) \quad (1.49)$$

³ This algorithm will terminate eventually, since the sequence $\{r_i\}_i$ is non-negative by definition of q_i , but strictly decreasing. Therefore, $\{r_i\}_i$ will meet 0 in finite step k .

and this implies

$$\gcd(a, b) =: \gcd(r_0, r_1) = \cdots = \gcd(r_{k-1}, 0), \quad (1.50)$$

i.e.,

$$r_{k-1} = \gcd(a, b). \quad (1.51)$$

Next, for $i = 0, 1$ observe

$$a * s_i + b * t_i = r_i. \quad (1.52)$$

Let $i \geq 2$, then

$$r_i = r_{i-2} - q_i * r_{i-1} \quad (1.53)$$

$$= a * s_{i-2} + b * t_{i-2} - q_i * (a * s_{i-1} + b * t_{i-1}) \quad (1.54)$$

$$= a * (s_{i-2} - q_i * s_{i-1}) + b * (t_{i-2} - q_i * t_{i-1}) \quad (1.55)$$

$$=: a * s_i + b * t_i. \quad (1.56)$$

Therefore, inductively we get

$$\gcd(a, b) = r_{k-1} = a * s_{k-1} + b * t_{k-1} =: a * x + b * y. \quad (1.57)$$

This prove Bézout's lemma.

■

Haskell implementation

Here I use lazy lists for intermediate lists of qs, rs, ss, ts , and pick up (second) last elements for the results.

Here we would like to implement the extended Euclidean algorithm. See the algorithm, examples, and pseudo code at:

https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

```
> exGCD' :: (Integral n) => n -> n -> ([n], [n], [n], [n])
> exGCD' a b = (qs, rs, ss, ts)
>   where
>     qs = zipWith quot rs (tail rs)
>     rs = takeUntil (==0) r'
>     r' = steps a b
```



```

> ss = steps 1 0
> ts = steps 0 1
> steps a b = rr
>   where
>     rr@(_:rs) = a:b: zipWith (-) rr (zipWith (*) qs rs)
>
> takeUntil :: (a -> Bool) -> [a] -> [a]
> takeUntil p = foldr func []
>   where
>     func x xs
>       | p x = []
>       | otherwise = x : xs

```

Here we have used so called lazy lists, and higher order function⁴. The gcd of a and b should be the last element of second list rs , and our targets (s, t) are second last elements of last two lists ss and ts . The following example is from wikipedia:

```

*Ffield> exGCD' 240 46
([5,4,1,1,2],[240,46,10,6,4,2],[1,0,1,-4,5,-9,23],[0,1,-5,21,-26,47,-120])

```

Look at the second lasts of $[1,0,1,-4,5,-9,23]$, $[0,1,-5,21,-26,47,-120]$, i.e., -9 and 47:

```

*Ffield> gcd 240 46
2
*Ffield> 240*(-9) + 46*(47)
2

```

It works, and we have other simpler examples:

```

*Ffield> exGCD' 15 25
([0,1,1,2],[15,25,15,10,5],[1,0,1,-1,2,-5],[0,1,0,1,-1,3])
*Ffield> 15 * 2 + 25*(-1)
5
*Ffield> exGCD' 15 26
([0,1,1,2,1,3],[15,26,15,11,4,3,1],[1,0,1,-1,2,-5,7,-26],[0,1,0,1,-1,3,-4,15])
*Ffield> 15*7 + (-4)*26
1

```

⁴ Naively speaking, the function whose inputs and/or outputs are functions is called a higher order function.

Now what we should do is extract gcd of a and b , and (x, y) from the tuple of lists:

```
> -- a*x + b*y = gcd a b
> exGCD :: Integral t => t -> t -> (t, t, t)
> exGCD a b = (g, x, y)
>   where
>     (_,r,s,t) = exGCD' a b
>     g = last r
>     x = last . init $ s
>     y = last . init $ t
```

where the underscore $_$ is a special symbol in Haskell that hits every pattern, since we do not need the quotient list. So, in order to get gcd and (x, y) we don't need quotients list.

```
*Ffield> exGCD 46 240
(2,47,-9)
*Ffield> 46*47 + 240*(-9)
2
*Ffield> gcd 46 240
2
```

1.1.7 Coprime as a binary relation

Let us define a binary relation as follows:

```
coprime :: Integral a => a -> a -> Bool
coprime a b = (gcd a b) == 1
```

1.1.8 Corollary (Inverses in \mathbb{Z}_n)

For a non-zero element

$$a \in \mathbb{Z}_n, \tag{1.58}$$

there is a unique number

$$b \in \mathbb{Z}_n \text{ s.t. } ((a * b) \bmod n) = 1 \tag{1.59}$$

iff a and n are coprime.

Proof

From Bézout's lemma, a and n are coprime iff

$$\exists s, t \in \mathbb{Z}, a * s + n * t = 1. \quad (1.60)$$

Therefore

$$a \text{ and } n \text{ are coprime} \Leftrightarrow \exists s, t \in \mathbb{Z}, a * s + n * t = 1 \quad (1.61)$$

$$\Leftrightarrow \exists s, t' \in \mathbb{Z}, a * s = 1 + n * t'. \quad (1.62)$$

This s , by taking its modulo n is our $b = a^{-1}$:

$$a * s = 1 \pmod{n}. \quad (1.63)$$

We will make a Haskell implementation in §1.1.9.

■

1.1.9 Corollary (Finite field \mathbb{Z}_p)

If p is prime, then

$$\mathbb{Z}_p := \{0, \dots, (p-1)\} \quad (1.64)$$

with addition, subtraction and multiplication under modulo n is a field.

Proof

It suffices to show that

$$\forall a \in \mathbb{Z}_p, a \neq 0 \Rightarrow \exists a^{-1} \in \mathbb{K} \text{ s.t. } a * a^{-1} = 1 = a^{-1} * a, \quad (1.65)$$

but since p is prime, and

$$\forall a \in \mathbb{Z}_p, a \neq 0 \Rightarrow \text{gcd } a \text{ } p == 1 \quad (1.66)$$

so all non-zero element has its inverse in \mathbb{Z}_p .

■

Example and implementation

Let us pick 11 as a prime and consider \mathbb{Z}_{11} :

Example \mathbb{Z}_{11}

```
*Ffield> isField 11
True
*ffield> map (exGCD 11) [0..10]
[(11,1,0),(1,0,1),(1,1,-5),(1,-1,4),(1,-1,3)
,(1,1,-2),(1,-1,2),(1,2,-3),(1,3,-4),(1,-4,5),(1,1,-1)
]
```

This list of three-tuple let us know the candidate of inverse. Take the last one, $(1,1,-1)$. This is the image of `exGcd 11 10`, and

$$1 = 10 * 1 + 11 * (-1) \quad (1.67)$$

holds. This suggests -1 is a candidate of the inverse of 10 in \mathbb{Z}_{11} :

$$10^{-1} = -1 \pmod{11} \quad (1.68)$$

$$= 10 \pmod{11} \quad (1.69)$$

In fact,

$$10 * 10 = 11 * 9 + 1. \quad (1.70)$$

So, picking up the third elements in tuple and zipping with nonzero elements, we have a list of inverses:

```
*Ffield> map (('mod' 11) . (\(_,_,x)->x) . exGCD 11) [1..10]
[1,6,4,3,9,2,8,7,5,10]
```

We get non-zero elements with its inverse:

```
*Ffield> zip [1..10] it
[(1,1),(2,6),(3,4),(4,3),(5,9),(6,2),(7,8),(8,7),(9,5),(10,10)]
```

Let us generalize these flow into a function⁵:

⁵ From <https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-Maybe.html>:

The Maybe type encapsulates an optional value. A value of type Maybe a either contains a value of type a (represented as Just a), or it is empty (represented as Nothing). Using Maybe is a good way to deal with errors or exceptional cases without resorting to drastic measures such as error.

```

> -- a^{-1} (in Z_p) == a 'inversep' p
> inversep :: Integral a => a -> a -> Maybe a
> a 'inversep' p = let (g,x,_) = exGCD a p in
>   if (g == 1) then Just (x 'mod' p)
>   else Nothing

```

This `inversep` function returns the inverse with respect to second argument, if they are coprime, i.e. gcd is 1. So the second argument should not be prime.

```

> inversesp :: Integral a => a -> [Maybe a]
> inversesp p = map ('inversep' p) [1..(p-1)]

*Ffield> inversesp 11
[Just 1,Just 6,Just 4,Just 3,Just 9,Just 2,Just 8,Just 7,Just 5,Just 10]
*Ffield> inversesp 9
[Just 1,Just 5,Nothing,Just 7,Just 2,Nothing,Just 4,Just 8]

```

1.2 Rational number reconstruction

1.2.1 A map from \mathbb{Q} to \mathbb{Z}_p

Let p be a prime. Now we have a map

$$- \text{ mod } p : \mathbb{Z} \rightarrow \mathbb{Z}_p; a \mapsto (a \text{ mod } p), \quad (1.71)$$

and a natural inclusion (or a forgetful map)⁶

$$\iota : \mathbb{Z}_p \hookrightarrow \mathbb{Z}. \quad (1.73)$$

Then we can define a map

$$- \text{ mod } p : \mathbb{Q} \rightarrow \mathbb{Z}_p \quad (1.74)$$

by⁷

$$q = \frac{a}{b} \mapsto (q \text{ mod } p) := ((a \times \iota(b^{-1} \text{ mod } p)) \text{ mod } p). \quad (1.75)$$

⁶ By introducing this forgetful map, we can use

$$\times : (\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{Z} \quad (1.72)$$

of normal product on \mathbb{Z} .

⁷ This is an example of operator overloadings.

Example and implementation

An easy implementation is the followings:⁸

A map from \mathbb{Q} to \mathbb{Z}_p .

```
> -- p should be prime.
> modp :: Integral a => Ratio a -> a -> a
> q 'modp' p = (a * (bi 'mod' p)) 'mod' p
>   where
>     (a,b) = (numerator q, denominator q)
>     bi = fromJust (b 'inversep' p)
```

Let us consider a rational number $\frac{3}{7}$ on a finite field \mathbb{Z}_{11} :

Example: on \mathbb{Z}_{11}

Consider $(3 \% 7)$.

```
*Ffield> let q = 3%7
*Ffield> 3 'mod' 11
3
*Ffield> 7 'inversep' 11
Just 8
*Ffield> (3*8) 'mod' 11
2
```

For example, pick 7:

```
*Ffield> 7*8 == 11*5+1
True
```

Therefore, on \mathbb{Z}_{11} , $(7^{-1} \bmod 11)$ is equal to $(8 \bmod 11)$ and

$$\frac{3}{7} \in \mathbb{Q} \mapsto (3 \times (7^{-1} \bmod 11) \bmod 11) \quad (1.78)$$

$$= (3 \times 8) \bmod 11 \quad (1.79)$$

$$= 24 \bmod 11 \quad (1.80)$$

$$= 2 \bmod 11. \quad (1.81)$$

⁸ The backquotes makes any binary function infix operator. For example,

$$\text{add } 1 \ 2 == 1 \text{ 'add' } 2 \quad (1.76)$$

Similarly, use parenthesis we can use an infix binary operator to a function:

$$(+) \ 1 \ 2 == 1 + 2 \quad (1.77)$$

Haskell returns the same result

```
*Ffield> q `modp` 11
2
```

and consistent.

1.2.2 Reconstruction from \mathbb{Z}_p to \mathbb{Q}

Consider a rational number q and its image $a \in \mathbb{Z}_p$.

$$a := q \mod p \quad (1.82)$$

The extended Euclidean algorithm can be used for guessing a rational number q from the images $a := q \mod p$ of several primes p 's.

At each step, the extended Euclidean algorithm satisfies eq.(1.52).

$$a * s_i + p * t_i = r_i \quad (1.83)$$

Therefore

$$r_i = a * s_i \mod p. \quad (1.84)$$

Hence $\frac{r_i}{s_i}$ is a possible guess for q . We take

$$r_i^2, s_i^2 < p \quad (1.85)$$

as the termination condition for this reconstruction.

Haskell implementation

Let us first try to reconstruct from the image $(\frac{1}{3} \mod p)$ of some prime p . Here we have chosen three primes

```
Reconstruction Z_p -> Q
*Ffield> let q = (1%3)
*Ffield> take 3 $ dropWhile (<100) primes
[101,103,107]
```

The images are basically given by the first elements of second lists (s_0 's):

```

*Ffield> q 'modp' 101
34
*Ffield> let try x = exGCD' (q 'modp' x) x
*Ffield> try 101
([0,2,1,33],[34,101,34,33,1],[1,0,1,-2,3,-101],[0,1,0,1,-1,34])
*Ffield> try 103
([0,1,2,34],[69,103,69,34,1],[1,0,1,-1,3,-103],[0,1,0,1,-2,69])
*Ffield> try 107
([0,2,1,35],[36,107,36,35,1],[1,0,1,-2,3,-107],[0,1,0,1,-1,36])

```

Look at the first hit of termination condition eq.(1.85), $r_4 = 1$ and $s_4 = 3$. They give us the same guess $\frac{1}{3}$, and that the reconstructed number.

From the above observations we can make a simple "guess" function:

```

> guess :: Integral t =>
>   (t, t)          -- (q 'modp' p, p)
>   -> (Ratio t, t)
> guess (a, p) = let (_,rs,ss,_) = exGCD' a p in
>   (select rs ss p, p)
>   where
>     select :: Integral t => [t] -> [t] -> t -> Ratio t
>     select [] _ _ = 0%1
>     select (r:rs) (s:ss) p
>       | s /= 0 && r^2 <= p && s^2 <= p = r% s
>       | otherwise = select rs ss p

```

We have put a list of big primes as follows.

```

> -- Hard code of big primes
> -- For chinese reminder theorem we declare it as [Integer].
> bigPrimes :: [Integer]
> bigPrimes = dropWhile (< 897473) $ takeWhile (< 978948) primes

```

We choose 3 times match as the termination condition.

```

> matches3 :: Eq a => [a] -> a
> matches3 (a:bb@(b:c:cs))
>   | a == b && b == c = a
>   | otherwise       = matches3 bb

```

Finally, we can check our gadgets.

What we know is a list of $(q \text{ 'modp' } p)$ and prime p for several (big) primes.


```

*Ffield> let q = 10%19
*Ffield> let knownData = zip (map (modp q) bigPrimes) bigPrimes
*Ffield> take 3 knownData
[(614061,897473),(377894,897497),(566842,897499)]
*Ffield> matches3 $ map (fst . guess) knownData
10 % 19

```

The following is the function we need, its input is the list of tuple which first element is the image in \mathbb{Z}_p and second element is that prime p .

```

> reconstruct :: Integral a =>
>      [(a, a)] -- :: [(Z_p, primes)]
>      -> Ratio a
> reconstruct aps = matches3 $ map (fst . guess) aps

```

Here is a naive test:

```

> let qs = [1 % 3, 10 % 19, 41 % 17, 30 % 311, 311 % 32
>           , 869 % 232, 778 % 123, 331 % 739]
> let modmap q = zip (map (modp q) bigPrimes) bigPrimes
> let longList = map modmap qs
> map reconstruct longList
[1 % 3, 10 % 19, 41 % 17, 30 % 311, 311 % 32
, 869 % 232, 778 % 123, 331 % 739]
> it == qs
True

```

For later use, let us define

```

> imagesAndPrimes :: Rational -> [(Integer, Integer)]
> imagesAndPrimes q = zip (map (modp q) bigPrimes) bigPrimes

```

to generate a list of images (of our target rational number) in \mathbb{Z}_p and the base primes.

As another example, we have slightly involved function:

```

> matches3' :: Eq a => [(a, t)] -> (a, t)
> matches3' (a0@(a,_):bb@((b,_):(c,_):cs))
>   | a == b && b == c = a0
>   | otherwise       = matches3' bb

```

Let us see the first good guess, Haskell tells us that in order to reconstruct, say $\frac{331}{739}$, we should take three primes start from 614693:

```

*Ffield> let knowData q = zip (map (modp q) primes) primes
*Ffield> matches3' $ map guess $ knowData (331%739)
(331 % 739,614693)
(18.31 secs, 12,393,394,032 bytes)

*Ffield> matches3' $ map guess $ knowData (11%13)
(11 % 13,311)
(0.02 secs, 2,319,136 bytes)
*Ffield> matches3' $ map guess $ knowData (1%13)
(1 % 13,191)
(0.01 secs, 1,443,704 bytes)
*Ffield> matches3' $ map guess $ knowData (1%3)
(1 % 3,13)
(0.01 secs, 268,592 bytes)
*Ffield> matches3' $ map guess $ knowData (11%31)
(11 % 31,1129)
(0.03 secs, 8,516,568 bytes)
*Ffield> matches3' $ map guess $ knowData (12%312)
(1 % 26,709)

```

A problem

Since our choice of `bigPrimes` are order 10^6 , our reconstruction can fail for rational numbers of

$$\frac{O(10^3)}{O(10^3)}, \quad (1.86)$$

say

```

*Ffield> let q = 895%922
*Ffield> let knownData = imagesAndPrimes q
*Ffield> take 4 knownData
[(882873,897473)
,(365035,897497)
,(705735,897499)
,(511060,897517)
]
*Ffield> map guess it
[((-854) % 123,897473)
,((-656) % 327,897497)
,((-192) % 805,897499)
]

```

```
,((-491) % 497,897517)
]
```

We can solve this by introducing the following theorem.

1.2.3 Chinese remainder theorem

From wikipedia⁹

There are certain things whose number is unknown. If we count them by threes, we have two left over; by fives, we have three left over; and by sevens, two are left over. How many things are there?

Here is a solution with Haskell:

```
*Ffield> let lst = [n|n<-[0..], mod n 3==2, mod n 5==3, mod n 7==2]
*Ffield> head lst
23
```

We define an infinite list of natural numbers that satisfy

$$n \bmod 3 = 2, n \bmod 5 = 3, n \bmod 7 = 2. \quad (1.87)$$

Then take the first element, and this is the answer.

Claim

The statement for binary case is the following. Let $n_1, n_2 \in \mathbb{Z}$ be coprime, then for arbitrary $a_1, a_2 \in \mathbb{Z}$, the following a system of equations

$$x = a_1 \bmod n_1 \quad (1.88)$$

$$x = a_2 \bmod n_2 \quad (1.89)$$

have a unique solution modular $n_1 * n_2$ ¹⁰.

⁹ https://en.wikipedia.org/wiki/Chinese_remainder_theorem

¹⁰ Note that, this is equivalent that there is a unique solution a in

$$0 \leq a < n_1 \times n_2. \quad (1.90)$$

Proof

(existence) With §1.1.6, there are $m_1, m_2 \in \mathbb{Z}$ s.t.

$$n_1 * m_1 + n_2 * m_2 = 1. \quad (1.91)$$

Now we have

$$n_1 * m_1 = 1 \pmod{n_2} \quad (1.92)$$

$$n_2 * m_2 = 1 \pmod{n_1} \quad (1.93)$$

that is¹¹

$$m_1 = n_1^{-1} \pmod{n_2} \quad (1.94)$$

$$m_2 = n_2^{-1} \pmod{n_1}. \quad (1.95)$$

Then

$$a := a_1 * n_2 * m_2 + a_2 * n_1 * m_1 \pmod{n_1 * n_2} \quad (1.96)$$

is a solution.

(uniqueness) If a' is also a solution, then

$$a - a' = 0 \pmod{n_1} \quad (1.97)$$

$$a - a' = 0 \pmod{n_2}. \quad (1.98)$$

Since n_1 and n_2 are coprime, i.e., no common divisors, this difference is divisible by $n_1 * n_2$, and

$$a - a' = 0 \pmod{n_1 * n_2}. \quad (1.99)$$

Therefore, the solution is unique modular $n_1 * n_2$.

■

Haskell implementation

Let us see how our naive `guess` function fail one more time:

Chinese Remainder Theorem, and its usage

```
*Ffield> let q = 895%922
*Ffield> let knownData = imagesAndPrimes q
```

¹¹ Here we have used slightly different notions from 1. m_1 in 1 is our m_2 times our n_2 .

```

*Ffield> let [(a1,p1),(a2,p2)] = take 2 knownData
*Ffield> take 2 knownData
[(882873,897473),(365035,897497)]
*Ffield> map guess it
[((-854) % 123,897473),((-656) % 327,897497)]

```

It suffices to make a binary version of Chinese Remainder theorem in Haskell:

```

> crtRec' :: Integral t => (t, t) -> (t, t) -> (t, t)
> crtRec' (a1,p1) (a2,p2) = (a,p)
>   where
>     a = (a1*p2*m2 + a2*p1*m1) 'mod' p
>     m1 = fromJust (p1 'inverse' p2)
>     m2 = fromJust (p2 'inverse' p1)
>     p = p1*p2

```

`crtRec'` function takes two tuples of image in \mathbb{Z}_p and primes, and returns these combination. Now let us fold.

```

> pile :: (a -> a -> a) -> [a] -> [a]
> pile f [] = []
> pile f dd@(d:ds) = d : zipWith' f (pile f dd) ds

```

Schematically, this `pile f` function takes

$$[d_0, d_1, d_2, d_3, \dots] \quad (1.100)$$

and returns

$$[d_0, f(d_0, d_1), f(f(d_0, d_1), d_2), f(f(f(d_0, d_1), d_2), d_3), \dots] \quad (1.101)$$

We have used another higher order function which is slightly modified from standard definition:

```

> -- Strict zipWith, from:
> --   http://d.hatena.ne.jp/kazu-yamamoto/touch/20100624/1277348961
> zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
> zipWith' f (a:as) (b:bs) = (x 'seq' x) : zipWith' f as bs
>   where x = f a b
> zipWith' _ _ _ = []

```

Let us check our implementation.

```

*Ffield> let q = 895%922
*Ffield> let knownData = imagesAndPrimes q
*Ffield> take 4 knownData
[(882873,897473)
,(365035,897497)
,(705735,897499)
,(511060,897517)
]
*Ffield> pile crtRec' it
[(882873,897473)
,(86488560937,805479325081)
,(397525881357811624,722916888780872419)
,(232931448259966259937614,648830197267942270883623)
]
*Ffield> map guess it
[((-854) % 123,897473)
,(895 % 922,805479325081)
,(895 % 922,722916888780872419)
,(895 % 922,648830197267942270883623)
]

```

So on a product ring $\mathbb{Z}_{805479325081}$, we get the right answer.

1.2.4 recCRT: from image in \mathbb{Z}_p to rational number

From above discussion, here we can define a function which takes a list of images in \mathbb{Z}_p and returns the rational number. What we do is, basically, to take a list of image (of our target rational number) and primes, then applying Chinese Remainder theorem recursively, return several guess of rational number.

```

> recCRT :: Integral a => [(a,a)] -> Ratio a
> recCRT = reconstruct . pile crtRec'

> recCRT' = matches3' . map guess . pile crtRec'

*Ffield> let q = 895%922
*Ffield> let knownData = imagesAndPrimes q
*Ffield> recCRT knownData
895 % 922
*Ffield> recCRT' knownData
(895 % 922,805479325081)

```

Here is some random checks and results.

```

todo: use QuickCheck

> trial = do
>   n <- randomRIO (0,10000) :: IO Integer
>   d <- randomRIO (1,10000) :: IO Integer
>   let q = (n%d)
>   putStrLn $ "input: " ++ show q
>   return $ recCRT' . imagesAndPrimes $ q

*Ffield> trial
input: 1080 % 6931
(1080 % 6931,805479325081)
*Ffield> trial
input: 2323 % 1248
(2323 % 1248,805479325081)
*Ffield> trial
input: 6583 % 1528
(6583 % 1528,805479325081)
*Ffield> trial
input: 721 % 423
(721 % 423,897473)
*Ffield> trial
input: 9967 % 7410
(9967 % 7410,805479325081)

```

1.3 Polynomials and rational functions

The following discussion on an arbitrary field \mathbb{K} .

1.3.1 Notations

Let $n \in \mathbb{N}$ be positive. We use multi-index notation:

$$\alpha = (\alpha_1, \dots, \alpha_n) \in \mathbb{N}^n. \quad (1.102)$$

A monomial is defined as

$$z^\alpha := \prod_i z_i^{\alpha_i}. \quad (1.103)$$

The total degree of this monomial is given by

$$|\alpha| := \sum_i \alpha_i. \quad (1.104)$$

1.3.2 Polynomials and rational functions

Let \mathbb{K} be a field. Consider a map

$$f : \mathbb{K}^n \rightarrow \mathbb{K}; z \mapsto f(z) := \sum_{\alpha} c_{\alpha} z^{\alpha}, \quad (1.105)$$

where

$$c_{\alpha} \in \mathbb{K}. \quad (1.106)$$

We call the value $f(z)$ at the dummy $z \in \mathbb{K}^n$ a polynomial:

$$f(z) := \sum_{\alpha} c_{\alpha} z^{\alpha}. \quad (1.107)$$

We denote

$$\mathbb{K}[z] := \left\{ \sum_{\alpha} c_{\alpha} z^{\alpha} \right\} \quad (1.108)$$

as the ring of all polynomial functions in the variable z with \mathbb{K} -coefficients.

Similarly, a rational function can be expressed as a ratio of two polynomials $p(z), q(z) \in \mathbb{K}[z]$:

$$\frac{p(z)}{q(z)} = \frac{\sum_{\alpha} n_{\alpha} z^{\alpha}}{\sum_{\beta} d_{\beta} z^{\beta}}. \quad (1.109)$$

We denote

$$\mathbb{K}(z) := \left\{ \frac{\sum_{\alpha} n_{\alpha} z^{\alpha}}{\sum_{\beta} d_{\beta} z^{\beta}} \right\} \quad (1.110)$$

as the field of rational functions in the variable z with \mathbb{F} -coefficients. Similar to fractional numbers, there are several equivalent representation of a rational function, even if we simplify with gcd. However there still is an overall constant ambiguity. To have a unique representation, usually we put the lowest degree of term of the denominator to be 1.

1.3.3 As data, coefficients list

We can identify a polynomial

$$\sum_{\alpha} c_{\alpha} z^{\alpha} \quad (1.111)$$

as a set of coefficients

$$\{c_{\alpha}\}_{\alpha}. \quad (1.112)$$

Similarly, for a rational function, we can identify

$$\frac{\sum_{\alpha} n_{\alpha} z^{\alpha}}{\sum_{\beta} d_{\beta} z^{\beta}} \quad (1.113)$$

as an ordered pair of coefficients

$$(\{n_{\alpha}\}_{\alpha}, \{d_{\beta}\}_{\beta}). \quad (1.114)$$

However, there still is an overall factor ambiguity even after gcd simplifications.

1.4 Haskell implementation of univariate polynomials

Here we basically follow some part of §9 of ref.3, and its addendum¹².

`Univariate.lhs`

```
> module Univariate where
> import Data.Ratio
> import Polynomials
```

1.4.1 A polynomial as a list of coefficients

Let us start `instance` declaration, which enable us to use basic arithmetics, e.g., addition and multiplication.

¹² See <http://homepages.cwi.nl/~jve/HR/PolAddendum.pdf>

```

-- Polynomials.hs
-- http://homepages.cwi.nl/~jve/rcrh/Polynomials.hs

module Polynomials where

default (Integer, Rational, Double)

-- polynomials, as coefficients lists
instance (Num a, Ord a) => Num [a] where
  fromInteger c = [fromInteger c]
  -- operator overloading
  negate []      = []
  negate (f:fs) = (negate f) : (negate fs)

  signum [] = []
  signum gs
    | signum (last gs) < (fromInteger 0) = negate z
    | otherwise = z

  abs [] = []
  abs gs
    | signum gs == z = gs
    | otherwise      = negate gs

  fs      + []      = fs
  []      + gs      = gs
  (f:fs) + (g:gs) = f+g : fs+gs

  fs      * []      = []
  []      * gs      = []
  (f:fs) * gg@(g:gs) = f*g : (f .* gs + fs * gg)

delta :: (Num a, Ord a) => [a] -> [a]
delta = ([1,-1] *)

shift :: [a] -> [a]
shift = tail

p2fct :: Num a => [a] -> a -> a
p2fct [] x = 0

```

1.4. HASKELL IMPLEMENTATION OF UNIVARIATE POLYNOMIALS 35

```

p2fct (a:as) x = a + (x * p2fct as x)

comp :: (Eq a, Num a, Ord a) => [a] -> [a] -> [a]
comp _      []      = error ".."
comp []     _       = []
comp (f:fs) g0@(0:gs) = f : gs * (comp fs g0)
comp (f:fs) gg@(g:gs) = ([f] + [g] * (comp fs gg))
                      + (0 : gs * (comp fs gg))

deriv :: Num a => [a] -> [a]
deriv []      = []
deriv (f:fs) = deriv1 fs 1
  where
    deriv1 [] _ = []
    deriv1 (g:gs) n = n*g : deriv1 gs (n+1)

```

Note that the above operators are overloaded, say $(*)$, $f*g$ is a multiplication of two numbers but $fs*gg$ is a multiplication of two list of coefficients. We can not extend this overloading to scalar multiplication, since Haskell type system takes the operands of $(*)$ the same type

$$(*) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a \quad (1.115)$$

```

> -- scalar multiplication
> infixl 7 .*
> (.*) :: Num a => a -> [a] -> [a]
> c .* []      = []
> c .* (f:fs) = c*f : c .* fs

```

Let us see few examples. If we take a scalar multiplication, say

$$3 * (1 + 2z + 3z^2 + 4z^3) \quad (1.116)$$

the result should be

$$3 * (1 + 2z + 3z^2 + 4z^3) = 3 + 6z + 9z^2 + 12z^3 \quad (1.117)$$

In Haskell

```

*Univariate> 3 .* [1,2,3,4]
[3,6,9,12]

```

and this is exactly same as map with section:

```
*Univariate> map (3*) [1,2,3,4]
[3,6,9,12]
```

When we multiply two polynomials, say

$$(1 + 2z) * (3 + 4z + 5z^2 + 6z^3) \quad (1.118)$$

the result should be

$$\begin{aligned} (1 + 2z) * (3 + 4z + 5z^2 + 6z^3) &= 1 * (3 + 4z + 5z^2 + 6z^3) + 2z * (3 + 4z + 5z^2 + 6z^3) \\ &= 3 + (4 + 2 * 3)z + (5 + 2 * 4)z^2 + (6 + 2 * 5)z^3 + 2 * 6z^4 \\ &= 3 + 10z + 13z^2 + 16z^3 + 12z^4 \end{aligned} \quad (1.119)$$

In Haskell,

```
*Univariate> [1,2] * [3,4,5,6]
[3,10,13,16,12]
```

Now the (dummy) variable is given as

```
> -- z of f(z), variable
> z :: Num a => [a]
> z = [0,1]
```

A polynomial of degree R is given by a finite sum of the following form:

$$f(z) := \sum_{i=0}^R c_i z^i. \quad (1.120)$$

Therefore, it is natural to represent $f(z)$ by a list of coefficient $\{c_i\}_i$. Here is the translator from the coefficient list to a polynomial function:

```
> p2fct :: Num a => [a] -> a -> a
> p2fct [] x = 0
> p2fct (a:as) x = a + (x * p2fct as x)
```

This gives us¹³

```
*Univariate> take 10 $ map (p2fct [1,2,3]) [0..]
[1,6,17,34,57,86,121,162,209,262]
*Univariate> take 10 $ map (\n -> 1+2*n+3*n^2) [0..]
[1,6,17,34,57,86,121,162,209,262]
```

¹³ Here we have used lambda, or so called anonymous function. From <http://learnyouahaskell.com/higher-order-functions>

To make a lambda, we write a `\` (because it kind of looks like the greek

1.4.2 Difference analysis

We do not know in general this canonical form of the polynomial, nor the degree. That means, what we can access is the graph of f , i.e., the list of inputs and outputs. Without loss of generality, we can take

$$[0..] \quad (1.123)$$

as the input data. Usually we take a finite sublist of this, but we assume it is sufficiently long. The outputs should be

$$\text{map } f \text{ } [0..] = [f \ 0, f \ 1 \ \dots] \quad (1.124)$$

For example

```
*Univariate> take 10 $ map (\n -> n^2+2*n+1) [0..]
[1,4,9,16,25,36,49,64,81,100]
```

Let us consider the difference sequence

$$\Delta(f)(n) := f(n+1) - f(n). \quad (1.125)$$

Its Haskell version is

```
> -- difference analysis
> difs :: (Num a) => [a] -> [a]
> difs [] = []
> difs [_] = []
> difs (i:jj@(j:js)) = j-i : difs jj
```

This gives

```
*Univariate> difs [1,4,9,16,25,36,49,64,81,100]
[3,5,7,9,11,13,15,17,19]
*Univariate> difs [3,5,7,9,11,13,15,17,19]
[2,2,2,2,2,2,2,2]
```

letter lambda if you squint hard enough) and then we write the parameters, separated by spaces.

For example,

$$f(x) := x^2 + 1 \quad (1.121)$$

$$f := \lambda x.x^2 + 1 \quad (1.122)$$

are the same definition.

We claim that if $f(z)$ is a polynomial of degree R , then $\Delta(f)(z)$ is a polynomial of degree $R - 1$. Since the degree is given, we can write $f(z)$ in canonical form

$$f(n) = \sum_{i=0}^R c_i n^i \quad (1.126)$$

and

$$\Delta(f)(n) := f(n+1) - f(n) \quad (1.127)$$

$$= \sum_{i=0}^R c_i \{(n+1)^i - n^i\} \quad (1.128)$$

$$= \sum_{i=1}^R c_i \{(n+1)^i - n^i\} \quad (1.129)$$

$$= \sum_{i=1}^R c_i \{i * n^{i-1} + O(n^{i-2})\} \quad (1.130)$$

$$= c_R * R * n^{R-1} + O(n^{R-2}) \quad (1.131)$$

where $O(n^{i-2})$ is some polynomial(s) of degree $i - 2$.

This guarantees the following function will terminate in finite steps¹⁴; `difLists` keeps generating difference lists until the difference get constant.

```
> difLists :: (Eq a, Num a) => [[a]] -> [[a]]
> difLists [] = []
> difLists xx@(xs:xss) =
>   if isConst xs then xx
>   else difLists $ difs xs : xx
>   where
>     isConst (i:jj@(j:js)) = all (==i) jj
>     isConst _ = error "difLists: lack of data, or not a polynomial"
```

Let us try:

```
*Univariate> difLists [[-12,-11,6,45,112,213,354,541,780,1077]]
[[6,6,6,6,6,6,6]
,[16,22,28,34,40,46,52,58]
,[1,17,39,67,101,141,187,239,297]
,[-12,-11,6,45,112,213,354,541,780,1077]
]
```

¹⁴ If a given lists is generated by a polynomial.

1.4. HASKELL IMPLEMENTATION OF UNIVARIATE POLYNOMIALS39

The degree of the polynomial can be computed by difference analysis:

```
> degree' :: (Eq a, Num a) => [a] -> Int
> degree' xs = length (difLists [xs]) -1
```

For example,

```
*Univariate> degree [1,4,9,16,25,36,49,64,81,100]
2
*Univariate> take 10 $ map (\n -> n^2+2*n+1) [0..]
[1,4,9,16,25,36,49,64,81,100]
*Univariate> degree $ take 10 $ map (\n -> n^5+4*n^3+1) [0..]
5
```

Above `degree'` function can only treat finite list, however, the following function can compute the degree of infinite list.

```
> degreeLazy :: (Eq a, Num a) => [a] -> Int
> degreeLazy xs = helper xs 0
>   where
>     helper as@(a:b:c:_) n
>       | a==b && b==c = n
>       | otherwise   = helper (difs as) (n+1)
```

Note that this lazy function only sees the first two elements of the list (of difference). So first take the lazy `degreeLazy` and guess the degree, take sufficient finite sublist of output and apply `degree'`. Here is the hybrid version:

```
> degree :: (Num a, Eq a) => [a] -> Int
> degree xs = let l = degreeLazy xs in
>   degree' $ take (l+2) xs
```


Chapter 2

Functional reconstruction over \mathbb{Q}

The goal of a functional reconstruction algorithm is to identify the monomials appearing in their definition and the corresponding coefficients.

From here, we use \mathbb{Q} as our base field, but every algorithm can be computed on any field, e.g., finite field \mathbb{Z}_p .

2.1 Univariate polynomials

2.1.1 Newtons' polynomial representation

Consider a univariate polynomial $f(z)$. Given a sequence of distinct values $y_n|_{n \in \mathbb{N}}$, we evaluate the polynomial form $f(z)$ sequentially:

$$f_0(z) = a_0 \tag{2.1}$$

$$f_1(z) = a_0 + (z - y_0)a_1 \tag{2.2}$$

$$\vdots$$

$$f_r(z) = a_0 + (z - y_0)(a_1 + (z - y_1)(\cdots + (z - y_{r-1})a_r)) \tag{2.3}$$

$$= f_{r-1}(z) + (z - y_0)(z - y_1) \cdots (z - y_{r-1})a_r, \tag{2.4}$$

where

$$a_0 = f(y_0) \quad (2.5)$$

$$a_1 = \frac{f(y_1) - a_0}{y_1 - y_0} \quad (2.6)$$

$$\vdots$$

$$a_r = \left(\left((f(y_r) - a_0) \frac{1}{y_r - y_0} - a_1 \right) \frac{1}{y_r - y_1} - \cdots - a_{r-1} \right) \frac{1}{y_r - y_{r-1}} \quad (2.7)$$

It is easy to see that, $f_r(z)$ and the original $f(z)$ match on the given data points, i.e.,

$$f_r(n) = f(n), 0 \leq n \leq r. \quad (2.8)$$

When we have already known the total degree of $f(z)$, say R , then we can terminate this sequential trial:

$$f(z) = f_R(z) \quad (2.9)$$

$$= \sum_{r=0}^R a_r \prod_{i=0}^{r-1} (z - y_i). \quad (2.10)$$

In practice, a consecutive zero on the sequence a_r can be taken as the termination condition for this algorithm.¹

2.1.2 Towards canonical representations

Once we get the Newton's representation

$$\sum_{r=0}^R a_r \prod_{i=0}^{r-1} (z - y_i) = a_0 + (z - y_0) (a_1 + (z - y_1) (\cdots + (z - y_{R-1}) a_R)) \quad (2.11)$$

as the reconstructed polynomial, it is convenient to convert it into the canonical form:

$$\sum_{r=0}^R c_r z^r. \quad (2.12)$$

This conversion only requires addition and multiplication of univariate polynomials. These operations are reasonably cheap, especially on \mathbb{Z}_p .

¹ We have not proved, but higher power will be dominant when we take sufficiently big input, so we terminate this sequence when we get a consecutive zero in a_r .

2.1.3 Simplification of our problem

Without loss of generality, we can put

$$[0..] \quad (2.13)$$

as our input list. We usually take its finite part but we assume it has enough length. Corresponding to above input,

$$\text{map } f \text{ } [0..] = [f \ 0, f \ 1, ..] \quad (2.14)$$

of $f :: \text{Ratio Int} \rightarrow \text{Ratio Int}$ is our output list.

Then we have slightly simpler forms of coefficients:

$$f_r(z) := a_0 + z * (a_1 + (z - 1) (a_2 + (z - 2) (a_3 + \dots + (z - r + 1)a_r))) \quad (2.15)$$

$$a_0 = f(0) \quad (2.16)$$

$$a_1 = f(y_1) - a_0 \quad (2.17)$$

$$= f(1) - f(0) =: \Delta(f)(0) \quad (2.18)$$

$$a_2 = \frac{f(2) - a_0}{2} - a_1 \quad (2.19)$$

$$= \frac{f(2) - f(0)}{2} - (f(1) - f(0)) \quad (2.20)$$

$$= \frac{f(2) - 2f(1) - f(0)}{2} \quad (2.21)$$

$$= \frac{(f(2) - f(1)) - (f(1) - f(0))}{2} =: \frac{\Delta^2(f)(0)}{2} \quad (2.22)$$

\vdots

$$a_r = \frac{\Delta^r(f)(0)}{r!}, \quad (2.23)$$

where Δ is the difference operator in eq.(1.125):

$$\Delta(f)(n) := f(n + 1) - f(n). \quad (2.24)$$

In order to simplify our expression, we introduce a falling power:

$$(x)_0 := 1 \quad (2.25)$$

$$(x)_n := x(x - 1) \cdots (x - n + 1) \quad (2.26)$$

$$= \prod_{i=0}^{n-1} (x - i). \quad (2.27)$$

Under these settings, we have

$$f(z) = f_R(z) \quad (2.28)$$

$$= \sum_{r=0}^R \frac{\Delta^r(f)(0)}{r!} (x)_r, \quad (2.29)$$

where we have assume

$$\Delta^{R+1}(f) = [0, 0, \dots]. \quad (2.30)$$

Example

Consider a polynomial

$$f(z) := 2 * z^3 + 3 * z, \quad (2.31)$$

and its out put list

$$[f(0), f(1), f(3), \dots] = [0, 5, 22, 63, 140, 265, \dots] \quad (2.32)$$

This polynomial is 3rd degree, so we compute up to $\Delta^3(f)(0)$:

$$f(0) = 0 \quad (2.33)$$

$$\Delta(f)(0) = f(1) - f(0) = 5 \quad (2.34)$$

$$\begin{aligned} \Delta^2(f)(0) &= \Delta(f)(1) - \Delta(f)(0) \\ &= f(2) - f(1) - 5 = 22 - 5 - 5 = 12 \end{aligned} \quad (2.35)$$

$$\begin{aligned} \Delta^3(f)(0) &= \Delta^2(f)(1) - \Delta^2(f)(0) \\ &= f(3) - f(2) - \{f(2) - f(1)\} - 12 = 12 \end{aligned} \quad (2.36)$$

so we get

$$[0, 5, 12, 12] \quad (2.37)$$

as the first difference list. Therefore, we get the falling power representation of f :

$$f(z) = 5(x)_1 + \frac{12}{2}(x)_2 + \frac{12}{3!}(x)_3 \quad (2.38)$$

$$= 5(x)_1 + 6(x)_2 + 2(x)_3. \quad (2.39)$$

2.2 Univariate polynomial reconstruction with Haskell

2.2.1 Newton interpolation formula with Haskell

First, the falling power is naturally given by recursively:

```
> infixr 8 ^- -- falling power
> (^-) :: (Integral a) => a -> a -> a
> x ^- 0 = 1
> x ^- n = (x ^- (n-1)) * (x - n + 1)
```

Assume the differences are given in a list

$$\mathbf{xs} = [f(0), \Delta(f)(0), \Delta^2(f)(0), \dots]. \quad (2.40)$$

Then the implementation of the Newton interpolation formula is as follows:

```
> newtonC :: (Fractional t, Enum t) => [t] -> [t]
> newtonC xs = [x / factorial k | (x,k) <- zip xs [0..]]
>   where
>     factorial k = product [1..fromInteger k]
```

Consider a polynomial

$$f \ x = 2*x^3+3*x \quad (2.41)$$

Let us try to reconstruct this polynomial from output list. In order to get the list $[x_0, x_1 \dots]$, take `difLists` and pick the first elements:

```
> let f x = 2*x^3+3*x
> take 10 $ map f [0..]
[0,5,22,63,140,265,450,707,1048,1485]
> difLists [it]
[[12,12,12,12,12,12,12]
, [12,24,36,48,60,72,84,96]
, [5,17,41,77,125,185,257,341,437]
, [0,5,22,63,140,265,450,707,1048,1485]
]
> reverse $ map head it
[0,5,12,12]
```

This list is the same as eq.(2.37) and we get the same expression as eq.(2.39) $5(x)_1 + 6(x)_2 + 2(x)_3$:

```
> newtonC it
[0 % 1,5 % 1,6 % 1,2 % 1]
```

The list of first differences, i.e.,

$$[f(0), \Delta(f)(0), \Delta^2(f)(0), \dots] \quad (2.42)$$

can be computed as follows:

```
> firstDifs :: (Eq a, Num a) => [a] -> [a]
> firstDifs xs = reverse $ map head $ difLists [xs]
```

Mapping a list of integers to a Newton representation:

```
> list2npol :: (Integral a) => [Ratio a] -> [Ratio a]
> list2npol = newtonC . firstDifs
```

```
*NewtonInterpolation> take 10 $ map f [0..]
[0,5,22,63,140,265,450,707,1048,1485]
*NewtonInterpolation> list2npol it
[0 % 1,5 % 1,6 % 1,2 % 1]
```

Therefore, we get the Newton coefficients from the output list.

2.2.2 Stirling numbers of the first kind

We need to map Newton falling powers to standard powers to get the canonical representation. This is a matter of applying combinatorics, by means of a convention formula that uses the so-called Stirling cyclic numbers

$$\begin{bmatrix} n \\ k \end{bmatrix} \quad (2.43)$$

Its defining relation is, $\forall n > 0$,

$$(x)_n = \sum_{k=1}^n (-1)^{n-k} \begin{bmatrix} n \\ k \end{bmatrix} x^k, \quad (2.44)$$

and

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} := 1. \quad (2.45)$$

2.2. UNIVARIATE POLYNOMIAL RECONSTRUCTION WITH HASKELL47

From the highest order, x^n , we get

$$\begin{bmatrix} n \\ n \end{bmatrix} = 1, \forall n > 0. \quad (2.46)$$

We also put

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \end{bmatrix} = \dots = 0, \quad (2.47)$$

and

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \dots = 0. \quad (2.48)$$

The key equation is

$$(x)_n = (x)_{n-1} * (x - n + 1) \quad (2.49)$$

and we get

$$(x)_n = \sum_{k=1}^n (-)^{n-k} \begin{bmatrix} n \\ k \end{bmatrix} x^k \quad (2.50)$$

$$= x^n + \sum_{k=1}^{n-1} (-)^{n-k} \begin{bmatrix} n \\ k \end{bmatrix} x^k \quad (2.51)$$

$$(x)_{n-1} * (x - n + 1) = \sum_{k=1}^{n-1} (-)^{n-1-k} \left\{ \begin{bmatrix} n-1 \\ k \end{bmatrix} x^{k+1} - (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix} x^k \right\} \quad (2.52)$$

$$= \sum_{l=2}^n (-)^{n-l} \begin{bmatrix} n-1 \\ l-1 \end{bmatrix} x^l + (n-1) \sum_{k=1}^{n-1} (-)^{n-k} \begin{bmatrix} n \\ k \end{bmatrix} x^k \quad (2.53)$$

$$= x^n + (n-1)(-)^{n-1}x + \sum_{k=2}^{n-1} (-)^{n-k} \left\{ \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix} \right\} x^k \quad (2.54)$$

$$= x^n + \sum_{k=1}^{n-1} (-)^{n-k} \left\{ \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix} \right\} x^k \quad (2.55)$$

Therefore, $\forall n, k > 0$,

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix} \quad (2.56)$$

Now we have the following canonical, power representation of reconstructed polynomial

$$f(z) = f_R(z) \quad (2.57)$$

$$= \sum_{r=0}^R \frac{\Delta^r(f)(0)}{r!} (x)_r \quad (2.58)$$

$$= \sum_{r=0}^R \frac{\Delta^r(f)(0)}{r!} \sum_{k=1}^r (-)^{r-k} \begin{bmatrix} r \\ k \end{bmatrix} x^k, \quad (2.59)$$

So, what shall we do is to sum up order by order.

Here is an implementation, first the Stirling numbers:

```
> stirlingC :: Integer -> Integer -> Integer
> stirlingC 0 0 = 1
> stirlingC 0 _ = 0
> stirlingC n k = (n-1)*(stirlingC (n-1) k) + stirlingC (n-1) (k-1)
```

This definition can be used to convert from falling powers to standard powers.

```
> fall2pol :: (Integral a) => a -> [a]
> fall2pol 0 = [1]
> fall2pol n = 0 -- No constant term.
> : [(-1)^(n-k) * stirlingC n k | k<-[1..n]]
```

We use `fall2pol` to convert Newton representations to standard polynomials in coefficients list representation. Here we have uses `sum` to collect same order terms in list representation.

```
> npol2pol :: (Integral a) => [Ratio a] -> [Ratio a]
> npol2pol xs = sum [ [x] * map fromInteger (fall2pol k)
>                     | (x,k) <- zip xs [0..]
>                     ]
```

2.2.3 list2pol: from output list to canonical coefficients

Finally, here is the function for computing a polynomial from an output sequence:

```
> list2pol :: (Integral a) => [Ratio a] -> [Ratio a]
> list2pol = npol2pol . list2npol
```


Here are some checks on these functions:

Reconstruction as curve fitting

```
*NewtonInterpolation> list2pol $ map (\n -> 7*n^2+3*n-4) [0..100]
[(-4) % 1,3 % 1,7 % 1]

*NewtonInterpolation> list2pol [0,1,5,14,30]
[0 % 1,1 % 6,1 % 2,1 % 3]
*NewtonInterpolation> map (\n -> n%6 + n^2%2 + n^3%3) [0..4]
[0 % 1,1 % 1,5 % 1,14 % 1,30 % 1]

*NewtonInterpolation> map (p2fct $ list2pol [0,1,5,14,30]) [0..8]
[0 % 1,1 % 1,5 % 1,14 % 1,30 % 1,55 % 1,91 % 1,140 % 1,204 % 1]
```

First example shows that from the sufficiently long output list, we can reconstruct the list of coefficients. Second example shows that from a given outputs, we have a list coefficients. Then use these coefficients, we define the output list of the function, and they match. The last example shows that from a limited (but sufficient) output information, we reconstruct a function and get extra outputs outside from the given data.

2.3 Univariate rational functions

We use the same notion, i.e., what we can know is the output-list of a univariate rational function, say $f :: \text{Int} \rightarrow \text{Ratio Int}$:

$$\text{map } f \text{ [0..]} == [f \ 0, f \ 1 \ ..] \quad (2.60)$$

2.3.1 Thiele's interpolation formula

We evaluate the polynomial form $f(z)$ as a continued fraction:

$$f_0(z) = a_0 \quad (2.61)$$

$$f_1(z) = a_0 + \frac{z}{a_1} \quad (2.62)$$

$$\vdots$$

$$f_r(z) = a_0 + \frac{z}{a_1 + \frac{z-1}{a_2 + \frac{z-2}{a_{r-2} + \frac{\vdots}{a_{r-1} + \frac{z-r+1}{a_r}}}}}, \quad (2.63)$$

where

$$a_0 = f(0) \quad (2.64)$$

$$a_1 = \frac{1}{f(1) - a_0} \quad (2.65)$$

$$a_2 = \frac{1}{\frac{2}{f(2) - a_0} - a_1} \quad (2.66)$$

$$\vdots$$

$$a_r = \frac{1}{\frac{2}{\frac{3}{\frac{\vdots}{\frac{r}{f(r) - a_0} - a_1} - a_2} - a_{r-1}} - a_{r-2}} \quad (2.67)$$

$$= \left(\left((f(r) - a_0)^{-1} r - a_1 \right)^{-1} (r-1) - \cdots - a_{r-1} \right)^{-1} 1 \quad (2.68)$$

2.3.2 Towards canonical representations

In order to get a unique representation of canonical form

$$\frac{\sum_{\alpha} n_{\alpha} z^{\alpha}}{\sum_{\beta} d_{\beta} z^{\beta}} \quad (2.69)$$

we put

$$d_{\min r'} = 1 \quad (2.70)$$

as a normalization, instead of d_0 . However, if we meet 0 as a singular value, then we can shift s.t. the new $d_0 \neq 0$. So without loss of generality, we can assume $f(0)$ is not singular, i.e., the denominator of f has a nonzero constant term:

$$d_0 = 1 \quad (2.71)$$

$$f(z) = \frac{\sum_i n_i z^i}{1 + \sum_{j>0} d_z^j}. \quad (2.72)$$

2.4 Univariate rational function reconstruction with Haskell

Here we the same notion of

`https://rosettacode.org/wiki/Thiele%27s_interpolation_formula`

and especially

`https://rosettacode.org/wiki/Thiele%27s_interpolation_formula#C`

2.4.1 Reciprocal difference

We claim, without proof², that the Thiele coefficients are given by

$$a_0 := f(0) \quad (2.73)$$

$$a_n := \rho_{n,0} - \rho_{n-2,0}, \quad (2.74)$$

² See the ref.4, Theorem (2.2.2.5) in 2nd edition.

where ρ is so called the reciprocal difference:

$$\rho_{n,i} := 0, n < 0 \quad (2.75)$$

$$\rho_{0,i} := f(i), i = 0, 1, 2, \dots \quad (2.76)$$

$$\rho_{n,i} := \frac{n}{\rho_{n-1,i+1} - \rho_{n-1,i}} + \rho_{n-2,i+1} \quad (2.77)$$

These preparation helps us to write the following codes:

Thiele's interpolation formula

Reciprocal difference rho, using the same notation of

https://rosettacode.org/wiki/Thiele%27s_interpolation_formula#C

```
> rho :: [Ratio Int] -- A list of output of f :: Int -> Ratio Int
>      -> Int -> Int -> Ratio Int
> rho fs 0 i = fs !! i
> rho fs n _
>   | n < 0 = 0
> rho fs n i = (n*den)%num + rho fs (n-2) (i+1)
>   where
>     num = numerator next
>     den = denominator next
>     next = (rho fs (n-1) (i+1)) - (rho fs (n-1) i)
```

Note that (%) has the following type,

```
(%) :: Integral a => a -> a -> Ratio a
```

```
> a fs 0 = fs !! 0
> a fs n = rho fs n 0 - rho fs (n-2) 0
```

2.4.2 tDegree for termination

Now let us consider a simple example which is given by the following Thiele coefficients

$$a_0 = 1, a_1 = 2, a_2 = 3, a_3 = 4. \quad (2.78)$$

2.4. UNIVARIATE RATIONAL FUNCTION RECONSTRUCTION WITH HASKELL53

The function is now

$$f(x) := 1 + \frac{x}{2 + \frac{x-1}{3 + \frac{x-2}{4}}} \quad (2.79)$$

$$= \frac{x^2 + 16x + 16}{16 + 6x} \quad (2.80)$$

Using Maxima³, we can verify this:

```
(%i25) f(x) := 1+(x/(2+(x-1)/(3+(x-2)/4)));
(%o25) f(x):=x/(2+(x-1)/(3+(x-2)/4))+1
(%i26) ratsimp(f(x));
(%o26) (x^2+16*x+16)/(16+6*x)
```

Let us come back Haskell, and try to get the Thiele coefficients of

```
*Univariate> let func x = (x^2 + 16*x + 16)%(6*x + 16)
*Univariate> let fs = map func [0..]
*Univariate> map (a fs) [0..]
[1 % 1,2 % 1,3 % 1,4 % 1,*** Exception: Ratio has zero denominator
```

This is clearly unsafe, so let us think more carefully. Observe the reciprocal differences

```
*Univariate> let fs = map func [0..]
*Univariate> take 5 $ map (rho fs 0) [0..]
[1 % 1,3 % 2,13 % 7,73 % 34,12 % 5]
*Univariate> take 5 $ map (rho fs 1) [0..]
[2 % 1,14 % 5,238 % 69,170 % 43,230 % 53]
*Univariate> take 5 $ map (rho fs 2) [0..]
[4 % 1,79 % 16,269 % 44,667 % 88,413 % 44]
*Univariate> take 5 $ map (rho fs 3) [0..]
[6 % 1,6 % 1,6 % 1,6 % 1,6 % 1]
```

So, the constancy of the reciprocal differences can be used to get the depth of Thiele series:

```
> tDegree :: [Ratio Int] -> Int
> tDegree fs = helper fs 0
```

³ <http://maxima.sourceforge.net>

```

> where
>   helper fs n
>     | isConstants fs' = n
>     | otherwise      = helper fs (n+1)
>   where
>     fs' = map (rho fs n) [0..]
>     isConstants (i:j:_) = i==j -- 2 times match
> -- isConstants (i:j:k:_) = i==j && j==k

```

Using this `tDegree` function, we can safely take the (finite) Thiele sequence.

2.4.3 thieleC: from output list to Thiele coefficients

From the equation (3.26) of ref.1,

```

*Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
*Univariate> let hs = map h [0..]
*Univariate> tDegree hs
4

```

So we get the Thiele coefficients

```

*Univariate> map (a hs) [0..(tDegree hs)]
[3 % 1, (-23) % 42, (-28) % 13, 767 % 14, 7 % 130]

```

Plug these in the continued fraction, and simplify with Maxima

```

(%i35) h(t):=3+t/((-23/42)+(t-1)/((-28/13)+(t-2)/((767/14)+(t-3)/(7/130))));
(%o35) h(t):=t/((-23)/42+(t-1)/((-28)/13+(t-2)/(767/14+(t-3)/(7/130)))+3
(%i36) ratsimp(h(t));
(%o36) (18*t^2+6*t+3)/(1+2*t+20*t^2)

```

Finally we make a function `thieleC` that returns the Thiele coefficients:

```

> thieleC :: [Ratio Int] -> [Ratio Int]
> thieleC lst = map (a lst) [0..(tDegree lst)]

*Univariate> thieleC hs
[3 % 1, (-23) % 42, (-28) % 13, 767 % 14, 7 % 130]

```

We need a convertor from this Thiele sequence to continuous form of rational function.

2.4. UNIVARIATE RATIONAL FUNCTION RECONSTRUCTION WITH HASKELL55

```
> nextStep [a0,a1] (v:_) = a0 + v/a1
> nextStep (a:as) (v:vs) = a + (v / nextStep as vs)
>
> -- From thiele sequence to (rational) function.
> thiele2ratf :: Integral a => [Ratio a] -> (Ratio a -> Ratio a)
> thiele2ratf as x
>   | x == 0 = head as
>   | otherwise = nextStep as [x,x-1 ..]
```

The following example shows that, the given output lists `hs`, we can interpolate the value between our discrete data.

```
*Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
*Univariate> let hs = map h [0..]
*Univariate> take 5 hs
[3 % 1,27 % 23,87 % 85,183 % 187,45 % 47]
*Univariate> let as = thieleC hs
*Univariate> as
[3 % 1,(-23) % 42,(-28) % 13,767 % 14,7 % 130]
*Univariate> let th x = thiele2ratf as x
*Univariate> map th [0..5]
[3 % 1,27 % 23,87 % 85,183 % 187,45 % 47,69 % 73]
*Univariate> th 0.5
3 % 2
```

2.4.4 Haskell representation for rational functions

We represent a rational function by a tuple of coefficient lists, like,

$$(ns,ds) :: ([Ratio Int],[Ratio Int]) \quad (2.81)$$

Here is a translator from coefficients lists to rational function.

```
> lists2ratf :: (Integral a) =>
>   ([Ratio a],[Ratio a]) -> (Ratio a -> Ratio a)
> lists2ratf (ns,ds) x = (p2fct ns x)/(p2fct ds x)

*Univariate> let frac x = lists2ratf ([1,1%2,1%3],[2,2%3]) x
*Univariate> take 10 $ map frac [0..]
[1 % 2,11 % 16,1 % 1,11 % 8,25 % 14,71 % 32,8 % 3,25 % 8,79 % 22,65 % 16]
*Univariate> let ffrac x = (1+(1%2)*x+(1%3)*x^2)/(2+(2%3)*x)
*Univariate> take 10 $ map ffrac [0..]
[1 % 2,11 % 16,1 % 1,11 % 8,25 % 14,71 % 32,8 % 3,25 % 8,79 % 22,65 % 16]
```

Simply taking numerator and denominator polynomials.

The following `canonicalizer` reduces the tuple-rep of rational function in canonical form, i.e., the coefficient of the lowest degree term of the denominator to be 1⁴.

```
> canonicalize :: (Integral a) =>
>   ([Ratio a],[Ratio a]) -> ([Ratio a],[Ratio a])
> canonicalize rat@(ns,ds)
>   | dMin == 1 = rat
>   | otherwise = (map (/dMin) ns, map (/dMin) ds)
>   where
>     dMin = firstNonzero ds
>     firstNonzero [a] = a -- head
>     firstNonzero (a:as)
>       | a /= 0 = a
>       | otherwise = firstNonzero as

*Univariate> canonicalize ([1,1%2,1%3],[2,2%3])
([1 % 2,1 % 4,1 % 6],[1 % 1,1 % 3])
*Univariate> canonicalize ([1,1%2,1%3],[0,0,2,2%3])
([1 % 2,1 % 4,1 % 6],[0 % 1,0 % 1,1 % 1,1 % 3])
*Univariate> canonicalize ([1,1%2,1%3],[0,0,0,2%3])
([3 % 2,3 % 4,1 % 2],[0 % 1,0 % 1,0 % 1,1 % 1])
```

What we need is a translator from Thiele coefficients to this tuple-rep. Since the list of Thiele coefficients is finite, we can naturally think recursively.

Before we go to a general case, consider

$$f(x) := 1 + \frac{x}{2 + \frac{x-1}{3 + \frac{x-2}{4}}} \quad (2.82)$$

⁴ Here our data point start from 0, i.e., the output data is given by `map f [0..]`, 0 is not singular, i.e., the denominator should have constant term and that means non empty. Therefore, the function `firstNonzero` is actually `head`.

When we simplify this expression, we should start from the bottom:

$$f(x) = 1 + \frac{x}{2 + \frac{x-1}{4 * 3 + x - 2}} \quad (2.83)$$

$$= 1 + \frac{x}{2 + \frac{x-1}{x+10}} \quad (2.84)$$

$$= 1 + \frac{x}{\frac{2 * (x+10) + 4 * (x-1)}{x+10}} \quad (2.85)$$

$$= 1 + \frac{x}{\frac{6x+16}{x+10}} \quad (2.86)$$

$$= \frac{1 * (6x+16) + x * (x+10)}{6x+16} \quad (2.87)$$

$$= \frac{x^2 + 16x + 16}{6x+16} \quad (2.88)$$

Finally, if we need, we take its canonical form:

$$f(x) = \frac{1 + x + \frac{1}{16}x^2}{1 + \frac{3}{8}x} \quad (2.89)$$

In general, we have the following Thiele representation:

$$a_0 + \frac{z}{a_1 + \frac{z-1}{a_2 + \frac{z-2}{\vdots \frac{z-n}{a_n + \frac{z-n}{a_{n+1}}}}} \quad (2.90)$$

The base case should be

$$a_n + \frac{z-n}{a_{n+1}} = \frac{a_{n+1} * a_n - n + z}{a_{n+1}} \quad (2.91)$$

and induction step $0 \leq r \leq n$ should be

$$a_r(z) = a_r + \frac{z - r}{a_{r+1}(z)} \quad (2.92)$$

$$= \frac{a_r a_{r+1}(z) + z - r}{a_{r+1}(z)} \quad (2.93)$$

$$= \frac{a_r * \text{num}(a_{r+1}(z)) + \text{den}(a_{r+1}(z)) * (z - r)}{\text{num}(a_{r+1}(z))} \quad (2.94)$$

where

$$a_{r+1}(z) = \frac{\text{num}(a_{r+1}(z))}{\text{den}(a_{r+1}(z))} \quad (2.95)$$

is a canonical representation of $a_{n+1}(z)$ ⁵.

Thus, the implementation is the followings.

```
> thiele2coef :: (Integral a) =>
>   [Ratio a] -> ([Ratio a],[Ratio a])
> thiele2coef as = canonicalize $ t2r as 0
>   where
>     t2r [an,an'] n = ([an*an'-n,1],[an'])
>     t2r (a:as)    n = ((a .* num) + ([-n,1] * den), num)
>     where
>       (num, den) = t2r as (n+1)
```

From the first example,

```
*Univariate> let func x = (x^2+16*x+16)%(6*x+16)
*Univariate> let funcList = map func [0..]
*Univariate> tDegree funcList
3
*Univariate> take 5 funcList
[1 % 1,3 % 2,13 % 7,73 % 34,12 % 5]
*Univariate> let aFunc = thieleC funcList
*Univariate> aFunc
[1 % 1,2 % 1,3 % 1,4 % 1]
*Univariate> thiele2coef aFunc
([1 % 1,1 % 1,1 % 16],[1 % 1,3 % 8])
```

From the other example, equation (3.26) of ref.1,

⁵ Not necessary being a canonical representation, it suffices to express $a_{n+1}(z)$ in a polynomial over polynomial form, that is, two lists in Haskell.

```

*Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
*Univariate> let hs = map h [0..]
*Univariate> take 5 hs
[3 % 1,27 % 23,87 % 85,183 % 187,45 % 47]
*Univariate> let th x = thiele2ratf as x
*Univariate> map th [0..5]
[3 % 1,27 % 23,87 % 85,183 % 187,45 % 47,69 % 73]
*Univariate> as
[3 % 1,(-23) % 42,(-28) % 13,767 % 14,7 % 130]
*Univariate> thiele2coef as
([3 % 1,6 % 1,18 % 1],[1 % 1,2 % 1,20 % 1])

```

2.4.5 list2rat: from output list to canonical coefficients

Finally, we get

```

> list2rat :: (Integral a) => [Ratio a] -> ([Ratio a], [Ratio a])
> list2rat = thiele2Coef . thieleC

```

as the reconstruction function from the output sequence.

```

*Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
*Univariate> list2rat $ map h [0..]
([3 % 1,6 % 1,18 % 1],[1 % 1,2 % 1,20 % 1])

```

2.5 Multivariate polynomials

From now on, we will use only the following functions from univariate cases.

Multivariate.lhs

```

> module Multivariate
>   where

>   import Data.Ratio
>   import Univariate
>   ( degree, list2pol
>     , thiele2ratf, lists2ratf, thiele2coef, lists2rat
>     )

```

2.5.1 Foldings as recursive applications

Consider an arbitrary multivariate polynomial

$$f(z_1, \dots, z_n) \in \mathbb{K}[z_1, \dots, z_n]. \quad (2.96)$$

First, fix all the variable but 1st and apply the univariate Newton's reconstruction:

$$f(z_1, z_2, \dots, z_n) = \sum_{r=0}^R a_r(z_2, \dots, z_n) \prod_{i=0}^{r-1} (z_1 - y_i) \quad (2.97)$$

Recursively, pick up one "coefficient" and apply the univariate Newton's reconstruction on z_2 :

$$a_r(z_2, \dots, z_n) = \sum_{s=0}^S b_s(z_3, \dots, z_n) \prod_{j=0}^{s-1} (z_2 - x_j) \quad (2.98)$$

The terminate condition should be the univariate case.

2.5.2 Experiments, 2 variables case

Let us take a polynomial from the denominator in eq.(3.23) of ref.1.

$$f(z_1, z_2) = 3 + 2z_1 + 4z_2 + 7z_1^2 + 5z_1z_2 + 6z_2^2 \quad (2.99)$$

In Haskell, first, fix $z_2 = 0, 1, 2$ and identify $f(z_1, 0), f(z_1, 1), f(z_1, 2)$ as our univariate polynomials.

```
*Multivariate> let f z1 z2 = 3+2*z1+4*z2+7*z1^2+5*z1*z2+6*z2^2
*Multivariate> let fs z = map ('f' z) [0..]
*Multivariate> let llst = map fs [0,1,2]
*Multivariate> map degree llst
[2,2,2]
```

Fine, so the canonical form can be

$$f(z_1, z) = c_0(z) + c_1(z)z_1 + c_2(z)z_1^2. \quad (2.100)$$

Now our new target is three univariate polynomials $c_0(z), c_1(z), c_2(z)$.

```
*Multivariate> list2pol $ take 10 $ fs 0
[3 % 1,2 % 1,7 % 1]
*Multivariate> list2pol $ take 10 $ fs 1
[13 % 1,7 % 1,7 % 1]
*Multivariate> list2pol $ take 10 $ fs 2
[35 % 1,12 % 1,7 % 1]
```

That is

$$f(z, 0) = 3 + 2z + 7z^2 \quad (2.101)$$

$$f(z, 1) = 13 + 7z + 7z^2 \quad (2.102)$$

$$f(z, 2) = 35 + 12z + 7z^2. \quad (2.103)$$

From these observation, we can determine $c_2(z)$, since it already a constant sequence.

$$c_2(z) = 7 \quad (2.104)$$

Consider $c_1(z)$, the sequence is now enough to determine $c_1(z)$:

```
*Multivariate> degree [2,7,12]
1
*Multivariate> list2pol [2,7,12]
[2 % 1,5 % 1]
```

i.e.,

$$c_1(z) = 2 + 5z. \quad (2.105)$$

However, for $c_1(z)$

```
*Multivariate> degree [3, 13, 35]
*** Exception: difLists: lack of data, or not a polynomial
CallStack (from HasCallStack):
  error, called at ./Univariate.lhs:61:19 in main:Univariate
```

so we need more numbers. Let us try one more:

```
*Multivariate> list2pol $ take 10 $ map ('f' 3) [0..]
[69 % 1,17 % 1,7 % 1]
*Multivariate> degree [3, 13, 35, 69]
2
*Multivariate> list2pol [3,13,35,69]
[3 % 1,4 % 1,6 % 1]
```

Thus we have

$$c_0(z) = 3 + 4z + 6z^2 \quad (2.106)$$

and these fully determine our polynomial:

$$f(z_1, z_2) = (3 + 4z_2 + 6z_2^2) + (2 + 5z_2)z_1 + 7z_1^2. \quad (2.107)$$

As another experiment, take the denominator.

```

*Multivariate> let g x y = 1+7*x + 8*y + 10*x^2 + x*y+9*y^2
*Multivariate> let gs x = map (g x) [0..]
*Multivariate> map degree $ map gs [0..3]
[2,2,2,2]

```

So the canonical form should be

$$g(x, y) = c_0(x) + c_1(x)y + c_2(x)y^2 \quad (2.108)$$

Let us look at these coefficient polynomial:

```

*Multivariate> list2pol $ take 10 $ gs 0
[1 % 1,8 % 1,9 % 1]
*Multivariate> list2pol $ take 10 $ gs 1
[18 % 1,9 % 1,9 % 1]
*Multivariate> list2pol $ take 10 $ gs 2
[55 % 1,10 % 1,9 % 1]
*Multivariate> list2pol $ take 10 $ gs 3
[112 % 1,11 % 1,9 % 1]

```

So we get

$$c_2(x) = 9 \quad (2.109)$$

and

```

*Multivariate> map (list2pol . (take 10) . gs) [0..4]
[[1 % 1,8 % 1,9 % 1]
,[18 % 1,9 % 1,9 % 1]
,[55 % 1,10 % 1,9 % 1]
,[112 % 1,11 % 1,9 % 1]
,[189 % 1,12 % 1,9 % 1]
]
*Multivariate> map head it
[1 % 1,18 % 1,55 % 1,112 % 1,189 % 1]
*Multivariate> list2pol it
[1 % 1,7 % 1,10 % 1]
*Multivariate> list2pol $ map (head . list2pol . (take 10) . gs) [0..4]
[1 % 1,7 % 1,10 % 1]

```

Using index operator (!!),

```

*Multivariate> list2pol $ map ((!! 0) . list2pol . (take 10) . gs) [0..4]
[1 % 1,7 % 1,10 % 1]
*Multivariate> list2pol $ map ((!! 1) . list2pol . (take 10) . gs) [0..4]
[8 % 1,1 % 1]
*Multivariate> list2pol $ map ((!! 2) . list2pol . (take 10) . gs) [0..4]
[9 % 1]

```

Finally we get

$$c_0(x) = 1 + 7x + 10x^2, c_1(x) = 8 + x, (c_2(x) = 9,) \quad (2.110)$$

and

$$g(x, y) = (1 + 7x + 10x^2) + (8 + x)y + 9y^2 \quad (2.111)$$

2.5.3 Haskell implementation, 2 variables case

Let us assume that we are given a "table" of the values of a 2-variate function. We represent this table as a list of lists.

```

*Multivariate> let f z1 z2 = 3+2*z1+4*z2+7*z1^2+5*z1*z2+6*z2^2
*Multivariate> [[f x y | y <- [0..9]] | x <- [0..9]]
[[3,13,35,69,115,173,243,325,419,525]
, [12,27,54,93,144,207,282,369,468,579]
, [35,55,87,131,187,255,335,427,531,647]
, [72,97,134,183,244,317,402,499,608,729]
, [123,153,195,249,315,393,483,585,699,825]
, [188,223,270,329,400,483,578,685,804,935]
, [267,307,359,423,499,587,687,799,923,1059]
, [360,405,462,531,612,705,810,927,1056,1197]
, [467,517,579,653,739,837,947,1069,1203,1349]
, [588,643,710,789,880,983,1098,1225,1364,1515]
]

> tablify :: (Enum t1, Num t1) => (t1 -> t1 -> t) -> Int -> [[t]]
> tablify f n = [[f x y | y <- range] | x <- range]
> where
>   range = take n [0..]

```

So, this "table" is like

$$\begin{pmatrix} f_{0,0} & f_{0,1} & \cdots \\ f_{1,0} & f_{1,1} & \cdots \\ f_{2,0} & f_{2,1} & \cdots \\ \vdots & & \ddots \end{pmatrix} \quad (2.112)$$

Then we can apply the univariate technique.

```
*Multivariate> let fTable = tablify f 10
*Multivariate> map list2pol fTable
[[3 % 1,4 % 1,6 % 1]
,[12 % 1,9 % 1,6 % 1]
,[35 % 1,14 % 1,6 % 1]
,[72 % 1,19 % 1,6 % 1]
,[123 % 1,24 % 1,6 % 1]
,[188 % 1,29 % 1,6 % 1]
,[267 % 1,34 % 1,6 % 1]
,[360 % 1,39 % 1,6 % 1]
,[467 % 1,44 % 1,6 % 1]
,[588 % 1,49 % 1,6 % 1]
]
```

Now we need to see the behavior of each coefficient, so take the "transpose" of it:

```
> well0rd :: [[a]] -> [[a]]
> well0rd xss
>   | null (head xss) = []
>   | otherwise      = map head xss : well0rd (map tail xss)

*Multivariate> let f z1 z2 = 3+2*z1+4*z2+7*z1^2+5*z1*z2+6*z2^2
*Multivariate> let fTable = tablify f 10
*Multivariate> map list2pol fTable
[[3 % 1,4 % 1,6 % 1]
,[12 % 1,9 % 1,6 % 1]
,[35 % 1,14 % 1,6 % 1]
,[72 % 1,19 % 1,6 % 1]
,[123 % 1,24 % 1,6 % 1]
,[188 % 1,29 % 1,6 % 1]
,[267 % 1,34 % 1,6 % 1]
```



```

, [360 % 1,39 % 1,6 % 1]
, [467 % 1,44 % 1,6 % 1]
, [588 % 1,49 % 1,6 % 1]
]
*Multivariate> well0rd it
[[3 % 1,12 % 1,35 % 1,72 % 1,123 % 1,188 % 1,267 % 1,360 % 1,467 % 1,588 % 1]
, [4 % 1,9 % 1,14 % 1,19 % 1,24 % 1,29 % 1,34 % 1,39 % 1,44 % 1,49 % 1]
, [6 % 1,6 % 1,6 % 1,6 % 1,6 % 1,6 % 1,6 % 1,6 % 1,6 % 1,6 % 1,6 % 1]
]
*Multivariate> map list2pol it
[[3 % 1,2 % 1,7 % 1]
, [4 % 1,5 % 1]
, [6 % 1]]

```

Therefore, the whole procedure becomes

```

> table2pol :: [[Ratio Integer]] -> [[Ratio Integer]]
> table2pol = map list2pol . well0rd . map list2pol

*Multivariate> let g x y = 1+7*x + 8*y + 10*x^2 + x*y+9*y^2
*Multivariate> table2pol $ tabsize g 5
[[1 % 1,7 % 1,10 % 1],[8 % 1,1 % 1],[9 % 1]]

```

2.6 Multivariate rational functions

2.6.1 The canonical normalization

Our target is a pair of coefficients $(\{n_\alpha\}_\alpha, \{d_\beta\}_\beta)$ in

$$\frac{\sum_\alpha n_\alpha z^\alpha}{\sum_\beta d_\beta z^\beta} \quad (2.113)$$

A canonical choice is

$$d_0 = d_{(0,\dots,0)} = 1. \quad (2.114)$$

Accidentally we might face $d_0 = 0$, but we can shift our function and make

$$d'_0 = d_s \neq 0. \quad (2.115)$$

2.6.2 An auxiliary t

Introducing an auxiliary variable t , let us define

$$h(z, t) := f(tz_1, \dots, tz_n), \quad (2.116)$$

and reconstruct $h(t, z)$ as a univariate rational function of t :

$$h(z, t) = \frac{\sum_{r=0}^R p_r(z) t^r}{1 + \sum_{r'=1}^{R'} q_{r'}(z) t^{r'}} \quad (2.117)$$

where

$$p_r(z) = \sum_{|\alpha|=r} n_\alpha z^\alpha \quad (2.118)$$

$$q_{r'}(z) = \sum_{|\beta|=r'} n_\beta z^\beta \quad (2.119)$$

are homogeneous polynomials.

Thus, what we shall do is the (homogeneous) polynomial reconstructions of $p_r(z)|_{0 \leq r \leq R}$, $q_{r'}(z)|_{1 \leq r' \leq R'}$.

A simplification

Since our new targets are homogeneous polynomials, we can consider, say,

$$p_r(1, z_2, \dots, z_n) \quad (2.120)$$

instead of $p_r(z_1, z_2, \dots, z_n)$, reconstruct it using multivariate Newton's method, and homogenize with z_1 .

2.6.3 Experiments, 2 variables case

Consider the equation (3.23) in ref.1.

```
*Multivariate> let f x y = (3+2*x+4*y+7*x^2+5*x*y+6*y^2)
                               % (1+7*x+8*y+10*x^2+x*y+9*y^2)

*Multivariate> :t f
f :: Integral a => a -> a -> Ratio a
*Multivariate> let h x y t = f (t*x) (t*y)
*Multivariate> let hs x y = map (h x y) [0..]
*Multivariate> take 5 $ hs 0 0
[3 % 1,3 % 1,3 % 1,3 % 1,3 % 1]
```

```

*Multivariate> take 5 $ hs 0 1
[3 % 1,13 % 18,35 % 53,69 % 106,115 % 177]
*Multivariate> take 5 $ hs 1 0
[3 % 1,2 % 3,7 % 11,9 % 14,41 % 63]
*Multivariate> take 5 $ hs 1 1
[3 % 1,3 % 4,29 % 37,183 % 226,105 % 127]

```

Here we have introduced the auxiliary t as third argument.

We take $(x, y) = (1, 0), (1, 1), (1, 2), (1, 3)$ and reconstruct them⁶.

```

*Multivariate> lists2rat $ hs 1 0
([3 % 1,2 % 1,7 % 1],[1 % 1,7 % 1,10 % 1])
*Multivariate> lists2rat $ hs 1 1
([3 % 1,6 % 1,18 % 1],[1 % 1,15 % 1,20 % 1])
*Multivariate> lists2rat $ hs 1 2
([3 % 1,10 % 1,41 % 1],[1 % 1,23 % 1,48 % 1])
*Multivariate> lists2rat $ hs 1 3
([3 % 1,14 % 1,76 % 1],[1 % 1,31 % 1,94 % 1])

```

So we have

$$h(1, 0, t) = \frac{3 + 2t + 7t^2}{1 + 7t + 10t^2} \quad (2.121)$$

$$h(1, 1, t) = \frac{3 + 6t + 18t^2}{1 + 15t + 20t^2} \quad (2.122)$$

$$h(1, 2, t) = \frac{3 + 10t + 41t^2}{1 + 23t + 48t^2} \quad (2.123)$$

$$h(1, 3, t) = \frac{3 + 14t + 76t^2}{1 + 31t + 94t^2} \quad (2.124)$$

Our next targets are the coefficients as polynomials in y ⁷.

Let us consider numerator first. This `list` is Haskell representation for eq.(2.121), eq.(2.122), eq.(2.123) and eq.(2.124).

```

*Multivariate> let list = map (lists2rat . (hs 1)) [0..4]
*Multivariate> let numf = map fst list
*Multivariate> list
([([3 % 1,2 % 1,7 % 1],[1 % 1,7 % 1,10 % 1])
,([3 % 1,6 % 1,18 % 1],[1 % 1,15 % 1,20 % 1])

```

⁶Eq.(3.26) in ref.1 is different from our reconstruction.

⁷ In our example, we take $x = 1$ fixed and reproduce x -dependence using homogenization

```
,([3 % 1,10 % 1,41 % 1],[1 % 1,23 % 1,48 % 1])
,[3 % 1,14 % 1,76 % 1],[1 % 1,31 % 1,94 % 1])
,[3 % 1,18 % 1,123 % 1],[1 % 1,39 % 1,158 % 1])
]
*Multivariate> numf
[[3 % 1,2 % 1,7 % 1]
,[3 % 1,6 % 1,18 % 1]
,[3 % 1,10 % 1,41 % 1]
,[3 % 1,14 % 1,76 % 1]
,[3 % 1,18 % 1,123 % 1]
]
```

From this information, we reconstruct each polynomials

```
*Multivariate> list2pol $ map head numf
[3 % 1]
*Multivariate> list2pol $ map (head . tail) numf
[2 % 1,4 % 1]
*Multivariate> list2pol $ map last numf
[7 % 1,5 % 1,6 % 1]
```

that is we have $3, 2 + 4y, 7 + 5y + 6y^2$ as results. Similarly,

```
*Multivariate> let denf = map snd list
*Multivariate> denf
[[1 % 1,7 % 1,10 % 1]
,[1 % 1,15 % 1,20 % 1]
,[1 % 1,23 % 1,48 % 1]
,[1 % 1,31 % 1,94 % 1]
,[1 % 1,39 % 1,158 % 1]
]
*Multivariate> list2pol $ map head denf
[1 % 1]
*Multivariate> list2pol $ map (head . tail) denf
[7 % 1,8 % 1]
*Multivariate> list2pol $ map last denf
[10 % 1,1 % 1,9 % 1]
```

So we get

$$h(1, y, t) = \frac{3 + (2 + 4y)t + (7 + 5y + 6y^2)t^2}{1 + (7 + 8y)t + (10 + y + 9y^2)t^2} \quad (2.125)$$

Finally, we use the homogeneous property for each powers:

$$h(x, y, t) = \frac{3 + (2x + 4y)t + (7x^2 + 5xy + 6y^2)t^2}{1 + (7x + 8y)t + (10x^2 + xy + 9y^2)t^2} \quad (2.126)$$

Putting $t = 1$, we get

$$f(x, y) = h(x, y, 1) \quad (2.127)$$

$$= \frac{3 + (2x + 4y) + (7x^2 + 5xy + 6y^2)}{1 + (7x + 8y) + (10x^2 + xy + 9y^2)} \quad (2.128)$$

2.6.4 Haskell implementation, 2 variables case

Assume we have a "table" of data:

```
*Multivariate> let h x y = (3+2*x+4*y+7*x^2+5*x*y+6*y^2) % (1+7*x+8*y+10*x^2+x*y+9*y^2)
*Multivariate> let auxh x y t = h (t*x) (t*y)
*Multivariate> let h x y = (3+2*x+4*y+7*x^2+5*x*y+6*y^2)% (1+7*x+8*y+10*x^2+x*y+9*y^2)
*Multivariate> let auxh x y t = h (t*x) (t*y)
```

Using the homogenous property, we just take $x=1$:

```
*Multivariate> let auxhs = [map (auxh 1 y) [0..5] | y <- [0..5]]
*Multivariate> auxhs
[[3 % 1,2 % 3,7 % 11,9 % 14,41 % 63,94 % 143]
,[3 % 1,3 % 4,29 % 37,183 % 226,105 % 127,161 % 192]
,[3 % 1,3 % 4,187 % 239,201 % 251,233 % 287,77 % 94]
,[3 % 1,31 % 42,335 % 439,729 % 940,425 % 543,1973 % 2506]
,[3 % 1,8 % 11,59 % 79,291 % 385,681 % 895,528 % 691]
,[3 % 1,23 % 32,155 % 211,1707 % 2302,1001 % 1343,4663 % 6236]
]
```

Now, each list can be seen as a univariate rational function:

```
*Multivariate> map list2rat auxhs
[[([3 % 1,2 % 1,7 % 1],[1 % 1,7 % 1,10 % 1])
,([3 % 1,6 % 1,18 % 1],[1 % 1,15 % 1,20 % 1])
,([3 % 1,10 % 1,41 % 1],[1 % 1,23 % 1,48 % 1])
,([3 % 1,14 % 1,76 % 1],[1 % 1,31 % 1,94 % 1])
,([3 % 1,18 % 1,123 % 1],[1 % 1,39 % 1,158 % 1])
,([3 % 1,22 % 1,182 % 1],[1 % 1,47 % 1,240 % 1])
]
```

```

*Multivariate> map fst it
[[3 % 1, 2 % 1, 7 % 1]
, [3 % 1, 6 % 1, 18 % 1]
, [3 % 1, 10 % 1, 41 % 1]
, [3 % 1, 14 % 1, 76 % 1]
, [3 % 1, 18 % 1, 123 % 1]
, [3 % 1, 22 % 1, 182 % 1]
]

```

We need to see the behavior of each coefficients:

```

*Multivariate> wellOrd it
[[3 % 1, 3 % 1, 3 % 1, 3 % 1, 3 % 1]
, [2 % 1, 6 % 1, 10 % 1, 14 % 1, 18 % 1, 22 % 1]
, [7 % 1, 18 % 1, 41 % 1, 76 % 1, 123 % 1, 182 % 1]
]
*Multivariate> map list2pol it
[[3 % 1], [2 % 1, 4 % 1], [7 % 1, 5 % 1, 6 % 1]]

```

So, the numerator is given by

```

*Multivariate> map list2pol . wellOrd . map (fst . list2rat) $ auxhs
[[3 % 1], [2 % 1, 4 % 1], [7 % 1, 5 % 1, 6 % 1]]

```

and the denominator is

```

*Multivariate> map list2pol . wellOrd . map (snd . list2rat) $ auxhs
[[1 % 1], [7 % 1, 8 % 1], [10 % 1, 1 % 1, 9 % 1]]

```

Thus, we finally get the following function

```

> table2ratf table = (t2r fst table, t2r snd table)
>   where
>     t2r third = map list2pol . wellOrd . map (third . list2rat)

```

```

*Multivariate> table2ratf auxhs
([ [3 % 1], [2 % 1, 4 % 1], [7 % 1, 5 % 1, 6 % 1] ], [ [1 % 1], [7 % 1, 8 % 1], [10 % 1, 1 % 1, 9 % 1] ])

```

Chapter 3

Functional reconstruction over finite fields

3.1 Univariate polynomials

We choose our new target the first differences, since once we get it, to reconstruct polynomial is an easy task. Once we get the first differences of a polynomial, we get the coefficient list by applying `npol2pol . newtonC` on it.

3.1.1 Pre-cook

We need a convertor, or function-modular which takes function and prime, and returns a function on \mathbb{Z}_p .

```
> -- Function-modular.
> fmodp :: Integral c => (a -> Ratio c) -> c -> a -> c
> f 'fmodp' p = ('modp' p) . f

*FR0verZp> let f x = (1%3) + (3%5)*x + (7%6)*x^2
*FR0verZp> let fs = map f [0..]
*FR0verZp> take 5 $ map (f 'fmodp' 101) [0..]
[34,93,87,16,82]
*FR0verZp> take 5 $ map ('modp' 101) fs
[34,93,87,16,82]
```

What we can access is the output list of our target polynomial. On \mathbb{Z}_p ,

our input is a finite list

$$[0, 1, 2 \dots (p-1)] \quad (3.1)$$

so as the output list.

```
> accessibleData :: (Ratio Int -> Ratio Int) -> Int -> [Int]
> accessibleData f p = take p $ map (f 'fmodp' p) [0..]
>
> accessibleData' :: [Ratio Int] -> Int -> [Int]
> accessibleData' fs p = take p $ map ('modp' p) fs
```

3.1.2 Difference analysis on \mathbb{Z}_p

Play the same game over prime field \mathbb{Z}_p , i.e., every arithmetic in under mod p .

Difference analysis over \mathbb{Z}_p

Every arithmetic should be on \mathbb{Z}_p , i.e., ('mod' p).

```
> difsp :: Integral b => b -> [b] -> [b]
> difsp p xs = map ('mod' p) (zipWith (-) (tail xs) xs)

*FR0verZp> let f x = (1%3) + (3%5)*x + (7%6)*x^2
*FR0verZp> take 5 $ accessibleData f 101
[34,93,87,16,82]
*FR0verZp> difsp 101 it
[59,95,30,66]
*FR0verZp> difsp 101 it
[36,36,36]
*FR0verZp> difsp 101 it
[0,0]
```

Here what we do is, first to take the differences (`zipWith (-) (tail xs) xs`) and take modular p for all element (`map ('mod' p)`). Now we can recursively apply this `difsp` over our data.

```
> difListsp :: Integral b => b -> [[b]] -> [[b]]
> difListsp _ [] = []
> difListsp p xx@(xs:xxs) =
>   if isConst xs then xx
>   else difListsp p $ difsp p xs : xx
```



```

> where
>   isConst (i:jj@(j:js)) = all (==i) jj
>   isConst _ = error "difListsp: "

*FROverZp> let f x = (1%3) + (3%5)*x + (7%6)*x^2
*FROverZp> map head $ difListsp 101 [accessibleData f 101]
[36,59,34]

```

3.1.3 Eager and lazy degree

From the above difference analysis on \mathbb{Z}_p , we get degree of the polynomial. Here we have a combination of two degree functions, one is eager and the other lazy:

Degree, eager and lazy versions

```

> degreep' p xs = length (difListsp p [xs]) -1
> degreep'Lazy p xs = helper xs 0
> where
>   helper as@(a:b:c:_) n
>     | a==b && b==c = n -- two times matching
>     | otherwise   = helper (difsp p as) (n+1)
>
> degreep :: Integral b => b -> [b] -> Int
> degreep p xs = let l = degreep'Lazy p xs in
>   degreep' p $ take (l+2) xs

*FROverZp> let f x = (1%3) + (3%5)*x + (7%6)*x^2
*FROverZp> let myDeg p = degreep p $ accessibleData f p
*FROverZp> myDeg 101
2
*FROverZp> myDeg 103
2
*FROverZp> myDeg 107
2
*FROverZp> degreep 101 $ accessibleData
  (\n -> (1%2)+(2%3)*n+(3%4)*n^2+(6%7)*n^7) 101
7

```

Now we can take first differences.

```

> firstDifsp :: Integral a => a -> [a] -> [a]
> firstDifsp p xs = reverse $ map head $ difListsp p [xs']
>   where
>     xs' = take n xs
>     n   = 2+ degreep p xs

```

```

*FROverZp> let f x = (1%3) + (3%5)*x + (7%6)*x^2
*FROverZp> firstDifsp 101 $ accessibleData f 101
[34,59,36]
*FROverZp> firstDifsp 101 $ accessibleData
  (\n -> (1%2)+(2%3)*n+(3%4)*n^2+(6%7)*n^7) 101
[51,66,59,33,29,58,32,78]

```

3.1.4 Term by term reconstruction

The output list of `firstDifsp` are basically the coefficients of Newton representation on \mathbb{Z}_p . So we zip it with our base prime p and map these pair over several primes.

```

> well0rd :: [[a]] -> [[a]]
> well0rd xss
>   | null (head xss) = []
>   | otherwise       = map head xss : well0rd (map tail xss)

*FROverZp> let f x = (1%3) + (3%5)*x + (7%6)*x^2
*FROverZp> let fps p = accessibleData f p
*FROverZp> let ourData p = firstDifsp p (fps p)
*FROverZp> let fivePrimes = take 5 bigPrimes
*FROverZp> map (\p -> zip (ourData p) (repeat p)) fivePrimes
[[ (299158,897473), (867559,897473), (299160,897473) ]
, [ (299166,897497), (329084,897497), (299168,897497) ]
, [ (598333,897499), (388918,897499), (598335,897499) ]
, [ (598345,897517), (29919,897517), (598347,897517) ]
, [ (299176,897527), (329095,897527), (299178,897527) ]
]
*FROverZp> well0rd it
[[ (299158,897473), (299166,897497), (598333,897499)
, (598345,897517), (299176,897527) ]
, [ (867559,897473), (329084,897497), (388918,897499)
, (29919,897517), (329095,897527) ]
]

```

```
,[(299160,897473),(299168,897497),(598335,897499)
,(598347,897517),(299178,897527)]
]
*FROverZp> :t it
it :: [(Int, Int)]
```

Finally we get the images of first differences over prime fields.

One minor issue is to change the data type, since our tools (say the functions of Chinese Remainder Theorem) use limit-less integer **Integer**.

```
> toInteger2 :: (Integral a1, Integral a) => (a, a1) -> (Integer, Integer)
> toInteger2 (a,b) = (toInteger a, toInteger b)
```

Let us take an example:

```
*FROverZp> let f x = (895 % 922) + (1080 % 6931)*x + (2323 % 1248)*x^2
*FROverZp> let fps p = accessibleData f p
*FROverZp> let longList = map (map toInteger2) $ wellOrd $
  map (\p -> zip (firstDifsp p (fps p)) (repeat p)) bigPrimes
*FROverZp> map recCRT' longList
[(895 % 922,805479325081)
,(17448553 % 8649888,722916888780872419)
,(2323 % 624,805479325081)
]
```

This result is consistent to that of on \mathbb{Q} :

```
*FROverZp> :l Univariate
[1 of 2] Compiling Polynomials      ( Polynomials.hs, interpreted )
[2 of 2] Compiling Univariate      ( Univariate.lhs, interpreted )
Ok, modules loaded: Univariate, Polynomials.
*Univariate> let f x = (895 % 922) + (1080 % 6931)*x + (2323 % 1248)*x^2
*Univariate> firstDifs (map f [0..20])
[895 % 922,17448553 % 8649888,2323 % 624]
```

3.1.5 list2polZp: from the output list to coefficient lists

Finally we get the function which takes an output list of our unknown univariate polynomial and returns the coefficient.

```
> list2firstDifZp' fs =
>   map recCRT' $ map (map toInteger2) $ wellOrd $ map helper bigPrimes
```

```

> where helper p = zip (firstDifsp p (accessibleData' fs p)) (repeat p)

*FR0verZp> let f x = (895 % 922) + (1080 % 6931)*x + (2323 % 1248)*x^2
*FR0verZp> let fs = map f [0..]
*FR0verZp> list2firstDifZp' fs
[(895 % 922,805479325081)
,(17448553 % 8649888,722916888780872419)
,(2323 % 624,805479325081)
]
*FR0verZp> map fst it
[895 % 922,17448553 % 8649888,2323 % 624]
*FR0verZp> newtonC it
[895 % 922,17448553 % 8649888,2323 % 1248]
*FR0verZp> npol2pol it
[895 % 922,1080 % 6931,2323 % 1248]

> list2polZp :: [Ratio Int] -> [Ratio Integer]
> list2polZp = npol2pol . newtonC . (map fst) . list2firstDifZp'

```

3.2 TBA Univariate rational functions

Chapter 4

Codes

4.1 Ffield.lhs

Listing 4.1: Ffield.lhs

```
1 Ffield.lhs
2
3 https://arxiv.org/pdf/1608.01902.pdf
4
5 > module Ffield where
6
7 > import Data.Ratio
8 > import Data.Maybe
9 > import Data.Numbers.Primes
10 > import Test.QuickCheck
11
12 > coprime :: Integral a => a -> a -> Bool
13 > coprime a b = gcd a b == 1
14
15 Consider a finite ring
16   Z_n := [0..(n-1)]
17 of some Int number.
18 If any non-zero element has its multiplication inverse,
   then the ring is a field:
19
20 > -- Our target should be in Int.
21 > isField :: Int -> Bool
22 > isField = isPrime
23
24 Here we would like to implement the extended Euclidean
   algorithm.
```

```

25 See the algorithm, examples, and pseudo code at:
26
27   https://en.wikipedia.org/wiki/
      Extended_Euclidean_algorithm
28   http://qiita.com/bra\_cat\_ket/items/205c19611e21f3d422b7
29
30 > exGCD' :: (Integral n) => n -> n -> ([n], [n], [n], [n]
      ])
31 > exGCD' a b = (qs, rs, ss, ts)
32 >   where
33 >     qs = zipWith quot rs (tail rs)
34 >     rs = takeUntil (==0) r'
35 >     r' = steps a b
36 >     ss = steps 1 0
37 >     ts = steps 0 1
38 >     steps a b = rr
39 >     where
40 >       rr@(_:rs) = a:b: zipWith (-) rr (zipWith (*) qs
      rs)
41 >
42 > takeUntil :: (a -> Bool) -> [a] -> [a]
43 > takeUntil p = foldr func []
44 >   where
45 >     func x xs
46 >       | p x          = []
47 >       | otherwise = x : xs
48 >
49 > --  $a*x + b*y = gcd\ a\ b$ 
50 > exGCD :: Integral t => t -> t -> (t, t, t)
51 > exGCD a b = (g, x, y)
52 >   where
53 >     (_,r,s,t) = exGCD' a b
54 >     g = last r
55 >     x = last . init $ s
56 >     y = last . init $ t
57 >
58 > --  $a^{-1}$  (in  $Z_p$ ) == a 'inversep' p
59 > inversep :: Integral a => a -> a -> Maybe a -- We also
      use in CRT.
60 > a 'inversep' p = let (g,x,_) = exGCD a p in
61 >   if (g == 1)
62 >     then Just (x 'mod' p) --  $g==1 \iff coprime\ a\ p$ 
63 >     else Nothing
64 >
65 > inversesp :: Int -> [Maybe Int]

```

```

66 > inversesp p = map ('inversep' p) [1..(p-1)]
67 >
68 > -- A map from Q to Z_p.
69 > -- p should be prime.
70 > modp :: Ratio Int -> Int -> Maybe Int
71 > q 'modp' p
72 >   | coprime b p = Just $ (a * (bi 'mod' p)) 'mod' p
73 >   | otherwise   = Nothing
74 >   where
75 >     (a,b) = (numerator q, denominator q)
76 >     Just bi = b 'inversep' p
77 >
78 > -- This is guess function without Chinese Reminder
    Theorem.
79 > guess :: Integral t =>
80 >   (Maybe t, t) -- (q 'modp' p, p)
81 >   -> Maybe (Ratio t, t)
82 > guess (Nothing, _) = Nothing
83 > guess (Just a, p) = let (_,rs,ss,_) = exGCD' a p in
84 >   Just (select rs ss p, p)
85 >   where
86 >     select :: Integral t => [t] -> [t] -> t -> Ratio
      t
87 >     select [] _ _ = 0%1
88 >     select (r:rs) (s:ss) p
89 >       | s /= 0 && r*r <= p && s*s <= p = r%s
90 >       | otherwise = select rs ss p
91 >
92 > -- Hard code of big primes
93 > -- We have chosen a finite number version.
94 > bigPrimes :: [Int]
95 > bigPrimes = take 100 $ dropWhile (<10^4) primes
96 > -- bigPrimes = dropWhile (<10^4) primes
97
98 *Ffield> let knownData q = zip (map (modp q) bigPrimes)
      bigPrimes
99 *Ffield> let ds = knownData (12%13)
100 *Ffield> map guess ds
101 [Just (12 % 13,10007)
102 ,Just (12 % 13,10009)
103 ,Just (12 % 13,10037)
104 ,Just (12 % 13,10039) ..
105
106 *Ffield> let ds = knownData (112%113)
107 *Ffield> map guess ds

```

```

108 [Just ((-39) % 50,10007)
109 ,Just ((-41) % 48,10009)
110 ,Just ((-69) % 20,10037)
111 ,Just ((-71) % 18,10039) ..
112
113 --
114
115 Chinese Remainder Theorem, and its usage
116
117 > imagesAndPrimes :: Ratio Int -> [(Maybe Int, Int)]
118 > imagesAndPrimes q = zip (map (modp q) bigPrimes)
    bigPrimes
119
120 Our data is a list of the type
121 [(Maybe Int, Int)]
122 In order to use CRT, we should cast its type.
123
124 > toInteger2 :: [(Maybe Int, Int)] -> [(Maybe Integer,
    Integer)]
125 > toInteger2 = map helper
126 >   where
127 >     helper (x,y) = (fmap toInteger x, toInteger y)
128 >
129 > crtRec' :: Integral a => (Maybe a, a) -> (Maybe a, a) ->
    (Maybe a, a)
130 > crtRec' (Nothing,p) (_,q)          = (Nothing, p*q)
131 > crtRec' (_,p)          (Nothing,q) = (Nothing, p*q)
132 > crtRec' (Just a1,p1) (Just a2,p2) = (Just a,p)
133 >   where
134 >     a = (a1*p2*m2 + a2*p1*m1) 'mod' p
135 >     Just m1 = p1 'inverse' p2
136 >     Just m2 = p2 'inverse' p1
137 >     p = p1*p2
138 >
139 > matches3 :: Eq a => [Maybe (a,b)] -> Maybe (a,b)
140 > matches3 (b1@(Just (q1,p1)):bb@((Just (q2,_)):(Just (q3
    ,_)):_))
141 >   | q1==q2 && q2==q3 = b1
142 >   | otherwise       = matches3 bb
143 > matches3 _ = Nothing
144
145 *Ffield> let ds = imagesAndPrimes (1123%1135)
146 *Ffield> map guess ds
147 [Just (25 % 52,10007)
148 ,Just ((-81) % 34,10009)

```



```

149     ,Just ((-88) % 63,10037) ..
150
151     *Ffield> matches3 it
152     Nothing
153
154     *Ffield> scanl1 crtRec' ds
155
156     *Ffield> scanl1 crtRec' . toInteger2 $ ds
157     [(Just 3272,10007)
158      ,(Just 14913702,100160063)
159      ,(Just 298491901442,1005306552331) ..
160
161     *Ffield> map guess it
162     [Just (25 % 52,10007)
163      ,Just (1123 % 1135,100160063)
164      ,Just (1123 % 1135,1005306552331)
165      ,Just (1123 % 1135,10092272478850909) ..
166
167     *Ffield> matches3 it
168     Just (1123 % 1135,100160063)
169
170     The final reconstruction function takes a list of Z_p
        values and returns the three times matched guess.
171
172     > -- reconstruct :: [(Maybe Int, Int)] -> Maybe (Ratio
        Integer, Integer)
173     > -- reconstruct = matches3 . map guess . scanl1 crtRec'
        . toInteger2
174     > -- reconstruct = matches3 . map guess . scanl1 crtRec'
        . filter (isJust . fst) . toInteger2
175
176     *Ffield> let q = 10937 % 10939
177     *Ffield> let ds = imagesAndPrimes q
178     *Ffield> map (fmap fst . guess) . scanl1 crtRec' .
        toInteger2 . filter (isJust . fst) $ ds
179     [Just ((-19) % 24)
180      ,Just ((-4977) % 4180)
181      ,Just (10937 % 10939)
182      ,Just (10937 % 10939)
183      ,Just (10937 % 10939)
184      ,Just (10937 % 10939)
185      ..
186     ]
187
188     > reconstruct :: [(Maybe Int, Int)] -> Maybe (Ratio

```

```

Integer)
189 > reconstruct = matches 10 . makeList
190 >   where
191 >     matches n (a:as)
192 >       | all (a==) $ take (n-1) as = a
193 >       | otherwise                 = matches n as
194 >     makeList = map (fmap fst . guess) . scanl1 crtRec'
      . toInteger2 . filter (isJust . fst)
195
196 --
197
198 *Ffield> let q = 513197683989569 % 1047805145658 ::
      Ratio Int
199 *Ffield> let ds = imagesAndPrimes q
200 *Ffield> let answer = fmap fromRational . reconstruct $
      ds
201 *Ffield> answer :: Maybe (Ratio Int)
202 Just (513197683989569 % 1047805145658)
203
204 > prop_rec :: Ratio Int -> Bool
205 > prop_rec q = Just q == answer
206 >   where
207 >     answer :: Maybe (Ratio Int)
208 >     answer = fmap fromRational . reconstruct $ ds
209 >     ds = imagesAndPrimes q
210
211 *Ffield> quickCheckWith stdArgs { maxSuccess = 100000 }
      prop_rec
212 +++ OK, passed 100000 tests.

```

4.2 Polynomials.hs

Listing 4.2: Polynomials.hs

```

1  -- Polynomials.hs
2  -- http://homepages.cwi.nl/~jve/rcrh/Polynomials.hs
3
4  module Polynomials where
5
6  default (Integer, Rational, Double)
7
8  -- scalar multiplication
9  infixl 7 .*
10 (.* ) :: Num a => a -> [a] -> [a]
11 c .* []      = []

```

```

12 c .* (f:fs) = c*f : c .* fs
13
14 z :: Num a => [a]
15 z = [0,1]
16
17 -- polynomials, as coefficients lists
18 instance (Num a, Ord a) => Num [a] where
19   fromInteger c = [fromInteger c]
20   -- operator overloading
21   negate []      = []
22   negate (f:fs) = (negate f) : (negate fs)
23
24   signum [] = []
25   signum gs
26     | signum (last gs) < (fromInteger 0) = negate z
27     | otherwise = z
28
29   abs [] = []
30   abs gs
31     | signum gs == z = gs
32     | otherwise      = negate gs
33
34   fs      + []      = fs
35   []      + gs      = gs
36   (f:fs) + (g:gs) = f+g : fs+gs
37
38   fs      * []      = []
39   []      * gs      = []
40   (f:fs) * gg@(g:gs) = f*g : (f .* gs + fs * gg)
41
42 delta :: (Num a, Ord a) => [a] -> [a]
43 delta = ([1,-1] *)
44
45 shift :: [a] -> [a]
46 shift = tail
47
48 p2fct :: Num a => [a] -> a -> a
49 p2fct [] x = 0
50 p2fct (a:as) x = a + (x * p2fct as x)
51
52 comp :: (Eq a, Num a, Ord a) => [a] -> [a] -> [a]
53 comp _ [] = error ".."
54 comp [] _ = []
55 comp (f:fs) g0@(0:gs) = f : gs * (comp fs g0)
56 comp (f:fs) gg@(g:gs) = ([f] + [g] * (comp fs gg))

```

```

57             + (0 : gs * (comp fs gg))
58
59 deriv :: Num a => [a] -> [a]
60 deriv []      = []
61 deriv (f:fs) = deriv1 fs 1
62   where
63     deriv1 []      _ = []
64     deriv1 (g:gs) n = n*g : deriv1 gs (n+1)

```

4.3 Univariate.lhs

Listing 4.3: Univariate.lhs

```

1  Univariate.lhs
2
3  > module Univariate where
4
5  References
6  J. Stoer, R. Bulirsch
7    Introduction to Numerical Analysis (2nd edition)
8  L. M. Milne-Thomson
9    THE CALCULUS OF FINITE DIFFERENCES
10
11 > import Control.Applicative
12 > import Control.Monad
13 > import Data.Ratio
14 > import Data.Maybe
15 > import Data.List
16 >
17 > import Polynomials
18
19 From the output list
20   map f [0..]
21 of a polynomial
22   f :: Int -> Ratio Int
23 we reconstruct the canonical form of f.
24
25 > difs :: (Num a) => [a] -> [a]
26 > difs [] = []
27 > difs [_] = []
28 > difs (i:jj@(j:js)) = j-i : difs jj
29 >
30 > difLists :: (Eq a, Num a) => [[a]] -> [[a]]
31 > difLists [] = []
32 > difLists xx@(xs:_) =

```

```

33 >   if isConst xs
34 >     then xx
35 >     else difLists $ difs xs : xx
36 >   where
37 >     isConst (i:jj@(j:_)) = all (==i) jj
38 >     isConst _ = error "difLists: lack of data, or not a
    polynomial"
39 >
40 > -- This degree function is "strict", so only take
    finite list.
41 > degree' :: (Eq a, Num a) => [a] -> Int
42 > degree' xs = length (difLists [xs]) - 1
43 >
44 > -- This degree function can compute the degree of
    infinite list.
45 > degreeLazy :: (Eq a, Num a) => [a] -> Int
46 > degreeLazy xs = helper xs 0
47 >   where
48 >     helper as@(a:b:c:_) n
49 >       | a==b && b==c = n
50 >       | otherwise   = helper (difs as) (n+1)
51 >
52 > -- This is a hybrid version, safe and lazy.
53 > degree :: (Num a, Eq a) => [a] -> Int
54 > degree xs = let l = degreeLazy xs in
55 >   degree' $ take (l+2) xs
56
57 > -- m-times match version
58 > degreeTimes :: (Num a, Eq a) => Int -> [a] -> Int
59 > degreeTimes m xs = helper xs 0
60 >   where
61 >     helper aa@(a:as) n
62 >       | all (== a) (take (m-1) as) = n
63 >       | otherwise                 = helper (difs aa) (
    n+1)
64
65 Newton interpolation formula
66 First we introduce a new infix symbol for the operation
67 of taking a falling power.
68
69 > infixr 8 ^- -- falling power
70 > (^-) :: (Eq a, Num a) => a -> a -> a
71 > x ^- 0 = 1
72 > x ^- n = (x ^- (n-1)) * (x - n + 1)
73

```

```

74 Claim (Newton interpolation formula):
75   A polynomial f of degree n is expressed as
76      $f(z) = \sum_{k=0}^n \frac{\text{diff}^n(f)(0)}{k!} * (x \text{ } ^{-} \text{ } k)$ 
77   where  $\text{diff}^n(f)$  is the n-th difference of f.
78
79 Example
80 Consider a polynomial  $f(x) = 2x^3 + 3x$ .
81
82 In general, we have no prior knowledge of this form,
83 but we know the sequences as a list of outputs (map f
   [0..]):
84
85   Univariate> let f x = 2*x^3+3*x
86   Univariate> take 10 $ map f [0..]
87   [0,5,22,63,140,265,450,707,1048,1485]
88   Univariate> degree $ take 10 $ map f [0..]
89   3
90
91 Let us try to get differences:
92
93   Univariate> difs $ take 10 $ map f [0..]
94   [5,17,41,77,125,185,257,341,437]
95   Univariate> difs it
96   [12,24,36,48,60,72,84,96]
97   Univariate> difs it
98   [12,12,12,12,12,12,12]
99
100 Or more simply take difLists:
101
102   Univariate> difLists [take 10 $ map f [0..]]
103   [[12,12,12,12,12,12,12]
104    ,[12,24,36,48,60,72,84,96]
105    ,[5,17,41,77,125,185,257,341,437]
106    ,[0,5,22,63,140,265,450,707,1048,1485]
107   ]
108
109 What we need is the heads of above lists.
110
111   Univariate> map head it
112   [12,12,5,0]
113
114 Newton interpolation formula gives
115    $f' \text{ } x = 0*(x \text{ } ^{-} \text{ } 0) \text{ 'div' } (0!) + 5*(x \text{ } ^{-} \text{ } 1) \text{ 'div' } (1!) +$ 
116      $12*(x \text{ } ^{-} \text{ } 2) \text{ 'div' } (2!) + 12*(x \text{ } ^{-} \text{ } 3) \text{ 'div' } (3!)$ 

```

```

117         = 5*(x ^- 1) + 6*(x ^- 2) + 2*(x ^- 3)
118 So
119
120   Univariate> let f x = 2*x^3+3*x
121   Univariate> let f' x = 5*(x ^- 1) + 6*(x ^- 2) + 2*(x
      ^- 3)
122   Univariate> take 10 $ map f [0..]
123   [0,5,22,63,140,265,450,707,1048,1485]
124   Univariate> take 10 $ map f' [0..]
125   [0,5,22,63,140,265,450,707,1048,1485]
126
127   Assume the differences are given in a list
128   [x_0, x_1 ..]
129   where x_k = diff^k(f)(0).
130   Then the implementation of the Newton interpolation
      formula is as follows:
131
132   > newtonC
133   >   :: (Fractional t, Enum t) =>
134   >     [t] -- first differences
135   >   -> [t] -- Newton coefficients
136   > newtonC xs = [x / factorial k | (x,k) <- zip xs [0..]]
137   >   where
138   >     factorial k = product [1..fromInteger k]
139
140   Univariate> let f x = 2*x^3+3*x
141   Univariate> take 10 $ map f [0..]
142   [0,5,22,63,140,265,450,707,1048,1485]
143   Univariate> difLists [it]
144   [[12,12,12,12,12,12,12]
145    ,[12,24,36,48,60,72,84,96]
146    ,[5,17,41,77,125,185,257,341,437]
147    ,[0,5,22,63,140,265,450,707,1048,1485]
148   ]
149   Univariate> reverse $ map head it
150   [0,5,12,12]
151   Univariate> newtonC it
152   [0 % 1,5 % 1,6 % 1,2 % 1]
153
154   The list of first differences can be computed as follows:
155
156   > firstDifs
157   >   :: (Eq a, Num a) =>
158   >     [a] -- map f [0..]
159   >   -> [a]

```

```

160 > firstDifs xs = reverse . map head . difLists $ [xs]
161
162 Mapping a list of integers to a Newton representation:
163
164 > -- This implementation can take infinite list.
165 > list2npol :: (Integral a) => [Ratio a] -> [Ratio a]
166 > list2npol xs = newtonC . firstDifs $ take n xs
167 >   where n = (degree xs) + 2
168 >
169 > -- m-times matches version
170 > list2npolTimes :: (Integral a) => Int -> [Ratio a] -> [
    Ratio a]
171 > list2npolTimes m xs = newtonC . firstDifs $ take n xs
172 >   where n = (degreeTimes m xs) + 2
173
174 *Univariate> let f x = x*(x-1)*(x-2)*(x-3)*(x-4)*(x-5)
175 *Univariate> let fs = map f [0..]
176 *Univariate> list2npolTimes 10 fs
177 [0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,1 % 1]
178 *Univariate> npol2pol it
179 [0 % 1,(-120) % 1,274 % 1,(-225) % 1,85 % 1,(-15) % 1,1
    % 1]
180 *Univariate> list2npol fs
181 [0 % 1]
182
183 We need to map Newton falling powers to standard powers.
184 This is a matter of applying combinatorics, by means of a
    convention formula
185 that uses the so-called Stirling cyclic numbers (of the
    first kind.)
186 Its defining relation is
187  $(x \text{^-} n) = \sum_{k=1}^n (\text{stirlingC } n \text{ } k) * (-1)^{(n-k)} * x^k.$ 
188 The key equation is
189  $(x \text{^-} n) = (x \text{^-} (n-1)) * (x-n+1)$ 
190  $= x*(x \text{^-} (n-1)) - (n-1)*(x \text{^-} (n-1))$ 
191
192 Therefore, an implementation is as follows:
193
194 > stirlingC :: (Integral a) => a -> a -> a
195 > stirlingC 0 0 = 1
196 > stirlingC 0 _ = 0
197 > stirlingC n k = stirlingC (n-1) (k-1) + (n-1) *
    stirlingC (n-1) k
198

```



```

199 This definition can be used to convert from falling
    powers to standard powers.
200
201 > fall2pol :: (Integral a) => a -> [a]
202 > fall2pol 0 = [1]
203 > fall2pol n = 0    -- No constant term.
204 >              : [(-1)^(n-k) * stirlingC n k | k<-[1..n]]
205
206 We use this to convert Newton representations to standard
    polynomials
207 in coefficients list representation.
208 Here we have uses
209     sum
210 to collect same order terms in list representation.
211
212 > npol2pol :: (Ord t, Num t) => [t] -> [t]
213 > npol2pol xs = sum [ [x] * map fromInteger (fall2pol k)
214 >                      | (x,k) <- zip xs [0..]
215 >                      ]
216
217 Finally, here is the function for reconstruction the
    polynomial
218 from an output sequence:
219
220 > list2pol :: (Integral a) => [Ratio a] -> [Ratio a]
221 > list2pol = npol2pol . list2npol
222
223 Reconstruction as curve fitting
224
225 *Univariate> let f x = 2*x^3 + 3*x + 1%5
226 *Univariate> take 10 $ map f [0..]
227 [1%5, 26%5, 111%5, 316%5, 701%5, 1326%5, 2251%5,
    3536%5, 5241%5, 7426%5]
228 *Univariate> list2npol it
229 [1 % 5,5 % 1,6 % 1,2 % 1]
230 *Univariate> list2npol $ map f [0..]
231 [1 % 5,5 % 1,6 % 1,2 % 1]
232 *Univariate> list2pol $ map (\n -> 1%3 + (3%5)*n +
    (5%7)*n^2) [0..]
233 [1 % 3,3 % 5,5 % 7]
234 *Univariate> list2pol [0,1,5,14,30,55]
235 [0 % 1,1 % 6,1 % 2,1 % 3]
236 *Univariate> map (p2fct $ list2pol [0,1,5,14,30,55])
    [0..6]
237 [0 % 1,1 % 1,5 % 1,14 % 1,30 % 1,55 % 1,91 % 1]

```

```

238
239 Here is n-times match version:
240
241 > list2polTimes :: Integral a => Int -> [Ratio a] -> [
      Ratio a]
242 > list2polTimes n = npol2pol . list2npolTimes n
243
244 --
245
246 Thiele's interpolation formula
247 https://rosettacode.org/wiki/Thiele%27
      s\_interpolation\_formula#Haskell
248 http://mathworld.wolfram.com/ThielesInterpolationFormula.
      html
249
250 reciprocal difference
251 Using the same notation of
252 https://rosettacode.org/wiki/Thiele%27
      s\_interpolation\_formula#C
253
254 > rho :: (Integral a) =>
255 >      [Ratio a] -- A list of output of f :: a -> Ratio
      a
256 >      -> a -> Int -- "matrix"
257 >      -> Maybe (Ratio a) -- Nothing means 1/0 type
      infinity
258 > rho fs 0 i = Just $ fs !! i
259 > rho fs n i
260 >   | n < 0          = Just 0
261 >   | num == Just 0 = Nothing -- "infinity"
262 >   | otherwise      = (+) <$> recipro <*> rho fs (n-2) (i
      +1)
263 >   where
264 >     recipro = ((%) . (* n) <$> den) <*> num -- (den*n)%
      num
265 > --          (%) <$> (*n) <$> den <*> num -- functor
      law
266 >     num = numerator <$> next
267 >     den = denominator <$> next
268 >     next = (-) <$> rho fs (n-1) (i+1) <*> rho fs (n-1)
      i
269
270 Note that (%) has the following type,
271 (%) :: Integral a => a -> a -> Ratio a
272

```

```

273 *Univariate> (%) <$> (*2) <$> Just 5 <*> Just 3
274 Just (10 % 3)
275
276 The follwoing reciprocal differences match the table of
277 Milne-Thompson[1951] page 106:
278
279 *Univariate> map (\p -> map (rho (map (\t -> 1%(1+t^2))
280   [0..]) p) [0..3]) [0..5]
281 [[Just (1 % 1) ,Just (1 % 2) ,Just (1 % 5) ,
282   Just (1 % 10)]
283 ,[Just ((-2) % 1),Just ((-10) % 3),Just ((-10) % 1),
284   Just ((-170) % 7)]
285 ,[Just ((-1) % 1),Just ((-1) % 10),Just ((-1) % 25),
286   Just ((-1) % 46)]
287 ,[Just (0 % 1) ,Just (40 % 1) ,Just (140 % 1) ,
288   Just (324 % 1)]
289 ,[Just (0 % 1) ,Just (0 % 1) ,Just (0 % 1) ,
290   Just (0 % 1)]
291 ,[Nothing,Nothing,Nothing,Nothing]
292 ]
293
294 > -- Thiele coefficients (continuous fraction)
295 > a :: (Integral a) => [Ratio a] -> a -> Maybe (Ratio a)
296 > a fs 0 = Just $ head fs
297 > a fs n = (-) <$> rho fs n 0 <*> rho fs (n-2) 0
298 >
299 > -- shifted Thiele coefficients
300 > a' :: Integral a => [Ratio a] -> Int -> a -> Maybe (
301   Ratio a)
302 > a' fs p 0 = Just $ fs !! p
303 > a' fs p n = (-) <$> rho fs n p <*> rho fs (n-2) p
304 >
305 *Univariate> map (\p -> map (rho (map (\t -> t%(1+t^2))
306   [0..]) p) [0..5]) [0..5]
307 [[Just (0%1),Just (1%2) ,Just (2%5) ,Just (3%10)
308   ,Just (4%17) ,Just (5%26)]
309 ,[Just (2%1),Just ((-10)%1),Just ((-10)%1),Just ((-170)
310   %11),Just ((-442)%19),Just ((-962)%29)]
311 ,[Just (1%3),Nothing ,Just ((-1)%15),Just ((-1)
312   %48) ,Just ((-1)%105) ,Just ((-1)%192)]
313 ,[Nothing ,Nothing ,Just (50%1) ,Just (242%1)
314   ,Just (662%1) ,Just (1430%1)]
315 ,[Nothing ,Nothing ,Just (0%1) ,Just (0%1)
316   ,Just (0%1) ,Just (0%1)]
317 ,[Nothing ,Nothing ,Nothing ,Nothing ,Nothing ,Nothing]

```

```

                                ,Nothing          ,Nothing]
305   ]
306
307   Here, the consecutive Just ((-10) % 1) in second list
       make "fake" infinity (Nothing).
308
309   *Univariate> let f t = t%(1+t^2)
310   *Univariate> let fs = map f [0..]
311   *Univariate> let aMat = [map (a' fs i) [0..] | i <-
       [0..]]
312   *Univariate> take 20 $ map (length . takeWhile isJust)
       $ aMat
313   [3,2,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5]
314
315   > -- Thiele coefficients with shifts.
316   > aMatrix :: Integral a => [Ratio a] -> [[Maybe (Ratio a)
       ]]
317   > aMatrix fs = [map (a' fs i) [0..] | i <- [0..]]
318   >
319   > tDegree :: Integral a => [Ratio a] -> Int
320   > tDegree = isConsts' 3 . map (length . takeWhile isJust)
       . aMatrix
321   >
322   > -- To find constant sub sequence.
323   > isConsts' :: Eq t => Int -> [t] -> t
324   > isConsts' n (l:ls)
325   >   | all (==l) $ take (n-1) ls = l
326   >   | otherwise                  = isConsts' n ls
327
328   we also need the shift, in this case, p=2 to get full
       Thiele coefficients.
329
330   > shiftaMatrix
331   >   :: Integral a =>
332   >     [Ratio a] -> [Maybe [Ratio a]]
333   > shiftaMatrix gs = map (sequence . (\q -> map (a' gs q)
       [0..(thieleD-1)])) [0..]
334   >   where
335   >     thieleD = fromIntegral $ tDegree gs
336   >
337   > shiftAndThieleC
338   >   :: Integral a =>
339   >     [Ratio a] -> (Maybe Int, Maybe [Ratio a])
340   > shiftAndThieleC fs = (findIndex isJust gs, join $ find
       isJust gs)

```

```

341 > where
342 >   gs = shiftaMatrix fs
343
344 *Univariate> take 10 $ map sequence $ transpose $ take
      (tDegree fs) m
345 [Just [0 % 1,1 % 2,2 % 5,3 % 10,4 % 17]
346 ,Just [2 % 1,(-10) % 1,(-10) % 1,(-170) % 11,(-442) %
      19]
347 ,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,
      Nothing,Nothing]
348
349 Packed version, this scans the given data only once.
350
351 > degSftTC
352 >   :: Integral a =>
353 >   [Ratio a] -> (Int, Maybe Int, Maybe (Maybe [Ratio
      a]))
354 > --           /      /      + Thiele
      coefficients
355 > --           /      + shift
356 > --           + degree
357 > degSftTC fs = (d,s,ts)
358 >   where
359 >     m = [map (a' fs i) [0..] | i <- [0..]]
360 >     d = isConsts' 3 . map (length . takeWhile isJust) $
      m -- 3 times match
361 >     m' = map (sequence . take d) m
362 >     s = findIndex isJust m'
363 >     ts = find isJust m'
364
365 *Univariate Control.Monad> let g t = t%(1+t^2)
366 *Univariate Control.Monad> let gs = map g [0..]
367 *Univariate Control.Monad> shiftAndThieleC $
      shiftaMatrix gs
368 (Just 2,Just [2 % 5,(-10) % 1,(-7) % 15,60 % 1,1 % 15])
369 *Univariate Control.Monad> let f t = 1%(1+t^2)
370 *Univariate Control.Monad> let fs = map f [0..]
371 *Univariate Control.Monad> shiftAndThieleC $
      shiftaMatrix fs
372 (Just 0,Just [1 % 1,(-2) % 1,(-2) % 1,2 % 1,1 % 1])
373
374 We need a convertor from this thiele sequence to
      continuous fractional form of rational function.
375
376 > nextStep [a0,a1] (v:_) = a0 + v/a1

```

```

377 > nextStep (a:as) (v:vs) = a + (v / nextStep as vs)
378 >
379 > -- From thiele sequence to (rational) function.
380 > thiele2ratf :: Integral a => [Ratio a] -> (Ratio a ->
    Ratio a)
381 > thiele2ratf as x
382 > | x == 0      = head as -- only constant term
383 > | otherwise = nextStep as [x,x-1 ..]
384
385 *Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
386 *Univariate> let hs = map h [0..]
387 *Univariate> let as = thieleC hs
388 *Univariate> as
389 [3 % 1,(-23) % 42,(-28) % 13,767 % 14,7 % 130]
390 *Univariate> let th x = thiele2ratf as x
391 *Univariate> take 5 hs
392 [3 % 1,27 % 23,87 % 85,183 % 187,45 % 47]
393 *Univariate> map th [0..5]
394 [3 % 1,27 % 23,87 % 85,183 % 187,45 % 47,69 % 73]
395
396 We represent a rational function by a tuple of
    coefficient lists:
397 (ns,ds) :: ([Ratio Int],[Ratio Int])
398 where ns and ds are coef-list-rep of numerator polynomial
    and denominator polynomial.
399 Here is a translator from coefficients lists to rational
    function.
400
401 > -- similar to p2fct
402 > lists2ratf :: (Integral a) =>
403 > ([Ratio a],[Ratio a]) -> (Ratio a ->
    Ratio a)
404 > lists2ratf (ns,ds) x = p2fct ns x / p2fct ds x
405
406 *Univariate> let frac x = lists2ratf
    ([1,1%2,1%3],[2,2%3]) x
407 *Univariate> take 10 $ map frac [0..]
408 [1 % 2,11 % 16,1 % 1,11 % 8,25 % 14,71 % 32,8 % 3,25 %
    8,79 % 22,65 % 16]
409 *Univariate> let ffrac x = (1+(1%2)*x+(1%3)*x^2)
    /(2+(2%3)*x)
410 *Univariate> take 10 $ map ffrac [0..]
411 [1 % 2,11 % 16,1 % 1,11 % 8,25 % 14,71 % 32,8 % 3,25 %
    8,79 % 22,65 % 16]
412

```

```

413 The following canonicalizer reduces the tuple-rep of
    rational function in canonical form
414 That is, the coefficient of the lowest degree term of the
    denominator to be 1.
415 However, since our input starts from 0 and this means
    firstNonzero is the same as head.
416
417 > canonicalize :: (Integral a) => ([Ratio a],[Ratio a])
    -> ([Ratio a],[Ratio a])
418 > canonicalize rat@(ns,ds)
419 >   | dMin == 1 = rat
420 >   | otherwise = (map (/ dMin) ns, map (/ dMin) ds)
421 >   where
422 >     dMin = firstNonzero ds
423 >     firstNonzero [a] = a -- head
424 >     firstNonzero (a:as)
425 >       | a /= 0     = a
426 >       | otherwise = firstNonzero as
427
428 What we need is a translator from Thiele coefficients to
    this tuple-rep.
429
430 > thiele2coef :: (Integral a) => [Ratio a] -> ([Ratio a]
    ,[Ratio a])
431 > thiele2coef as = canonicalize $ t2r as 0
432 >   where
433 >     t2r [an,an'] n = ([an*an'-n,1],[an'])
434 >     t2r (a:as)    n = ((a .* num) + ([-n,1] * den), num)
435 >     where
436 >       (num, den) = t2r as (n+1)
437 >
438
439 *Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
440 *Univariate> let hs = map h [0..]
441 *Univariate> take 5 hs
442 [3 % 1,27 % 23,87 % 85,183 % 187,45 % 47]
443 *Univariate> let th x = thiele2ratf as x
444 *Univariate> map th [0..5]
445 [3 % 1,27 % 23,87 % 85,183 % 187,45 % 47,69 % 73]
446 *Univariate> as
447 [3 % 1,(-23) % 42,(-28) % 13,767 % 14,7 % 130]
448 *Univariate> thiele2coef as
449 ([3 % 1,6 % 1,18 % 1],[1 % 1,2 % 1,20 % 1])
450
451 > thiele2coef' -- shifted version (0 -> sft)

```

```

452 > :: Integral a =>
453 >     Ratio a -> [Ratio a] -> ([Ratio a], [Ratio a])
454 > thiele2coef' sft [a] = ([a],1)
455 > thiele2coef' sft as = canonicalize $ t2r as sft
456 >     where
457 >         t2r [an,an'] n = (([an*an'-n] + z),[an'])
458 >         t2r (a:as)    n = ((a .* num) + ((z - [n]) * den),
         num)
459 >         where
460 >             (num, den) = t2r as (n+1)
461
462 *Univariate> let f x = x^2%(1+x^2)
463 *Univariate> shiftAndThieleC $ map f [0..]
464 (Just 0,Just [0 % 1,2 % 1,2 % 1,(-2) % 1,(-1) % 1])
465 *Univariate> take 3 $ shiftaMatrix (map f [0..])
466 [Just [0 % 1,2 % 1,2 % 1,(-2) % 1,(-1) % 1]
467 ,Just [1 % 2,10 % 3,3 % 5,(-130) % 3,(-1) % 10]
468 ,Just [4 % 5,10 % 1,6 % 25,(-150) % 1,(-1) % 25]
469 ]
470 *Univariate> thiele2coef' 0 [0 % 1,2 % 1,2 % 1,(-2) %
         1,(-1) % 1]
471 ([0 % 1,0 % 1,1 % 1],[1 % 1,0 % 1,1 % 1])
472 *Univariate> thiele2coef' 1 [1 % 2,10 % 3,3 % 5,(-130)
         % 3,(-1) % 10]
473 ([0 % 1,0 % 1,1 % 1],[1 % 1,0 % 1,1 % 1])
474
475 > shiftAndThiele2coef (Just sft, Just ts) = Just $
         thiele2coef' (fromIntegral sft) ts
476 > shiftAndThiele2coef _ = Nothing
477 >
478 > list2rat' :: (Integral a) => [Ratio a] -> Maybe ([Ratio
         a], [Ratio a])
479 > list2rat' = shiftAndThiele2coef . shiftAndThieleC
480 >
481 > list2rat'' lst = let (_,s,ts) = degSftTC lst in
482 >     shiftAndThiele2coef (s, join ts)
483
484 *Univariate> let f t = t%(1+t^2)
485 *Univariate> let fs = map (\t -> 1%(1+t^2)) [0..]
486 *Univariate> list2rat' fs
487 Just ([1 % 1,0 % 1,0 % 1],[1 % 1,0 % 1,1 % 1])
488 *Univariate> shiftAndThieleC fs
489 (Just 0,Just [1 % 1,(-2) % 1,(-2) % 1,2 % 1,1 % 1])
490 *Univariate> let fs = map (\t -> t%(1+t^2)) [0..]
491 *Univariate> let gs = map (\t -> t%(1+t^2)) [0..]

```



```

492 *Univariate> shiftAndThieleC gs
493 (Just 2,Just [2 % 5,(-10) % 1,(-7) % 15,60 % 1,1 % 15])
494 *Univariate> list2rat' gs
495 Just ([0 % 1,1 % 1,0 % 1],[1 % 1,0 % 1,1 % 1])
496 *Univariate> let hs = map (\t -> t^2%(1+t^2)) [0..]
497 *Univariate> shiftAndThieleC hs
498 (Just 0,Just [0 % 1,2 % 1,2 % 1,(-2) % 1,(-1) % 1])
499 *Univariate> list2rat' hs
500 Just ([0 % 1,0 % 1,1 % 1],[1 % 1,0 % 1,1 % 1])
501
502 *Univariate> let f t = t%(1+t^2)
503 *Univariate> let fs = map f [0..]
504 *Univariate> let aMat = [map (\j -> a' fs j i) [0..] |
505   i <- [0..]]
506 *Univariate> take 6 $ map (take 5) aMat
507 [[Just (0 % 1),Just (1 % 2),Just (2 % 5),Just (3 % 10),
508   Just (4 % 17)]
509 ,[Just (2 % 1),Just ((-10) % 1),Just ((-10) % 1),Just
510   ((-170) % 11),Just ((-442) % 19)]
511 ,[Just (1 % 3),Nothing,Just ((-7) % 15),Just ((-77) %
512   240),Just ((-437) % 1785)]
513 ,[Nothing,Nothing,Just (60 % 1),Just (2832 % 11),Just
514   (13020 % 19)]
515 ,[Nothing,Nothing,Just (1 % 15),Just (1 % 48),Just (1 %
516   105)]
517 ,[Nothing,Nothing,Nothing,Nothing,Nothing]
518 ]

```

4.4 Multivariate.lhs

Listing 4.4: Multivariate.lhs

```

1 Multivariate.lhs
2
3 > module Multivariate where
4
5 > import Data.Ratio
6 > import Data.List (transpose)
7 > import Univariate -- ( list2pol, list2npolTimes
8 >                      -- , tDegree, list2rat'
9 >                      -- )
10
11 Let us start 2-variate polynomials.
12 First, make a naive grid.
13

```

```

14  *Multivariate> let f z1 z2 = 3+2*z1+4*z2+7*z1^2+5*z1*z2
      +6*z2^2
15  *Multivariate> [[f x y | y <- [0..9]] | x <- [0..9]]
16  [[3,13,5,69,115,173,243,325,419,525]
17  , [12,27,54,93,144,207,282,369,468,579]
18  , [35,55,87,131,187,255,335,427,531,647]
19  , [72,97,134,183,244,317,402,499,608,729]
20  , [123,153,195,249,315,393,483,585,699,825]
21  , [188,223,270,329,400,483,578,685,804,935]
22  , [267,37,359,423,499,587,687,799,923,1059]
23  , [360,405,462,531,612,705,810,927,1056,1197]
24  , [467,517,579,653,739,837,947,1069,1203,1349]
25  , [588,643,710,789,880,983,1098,1225,1364,1515]
26  ]
27
28  Assuming the list of lists is a matrix of 2-variate
      function's values,
29  f i j
30
31  > tablizize
32  > :: (Enum t1, Num t1) =>
33  >   (t1 -> t1 -> t) -> Int -> [[t]]
34  > tablizize f n = [[f x y | y <- range] | x <- range]
35  >   where
36  >     range = take n [0..]
37
38  *Multivariate> tablizize (\x y -> (x,y)) 4
39  [[(0,0),(0,1),(0,2),(0,3)]
40  , [(1,0),(1,1),(1,2),(1,3)]
41  , [(2,0),(2,1),(2,2),(2,3)]
42  , [(3,0),(3,1),(3,2),(3,3)]
43  ]
44
45  *Multivariate> let fTable = tablizize f 10
46  *Multivariate> map list2pol fTable
47  [[3 % 1,4 % 1,6 % 1]
48  , [12 % 1,9 % 1,6 % 1]
49  , [35 % 1,14 % 1,6 % 1]
50  , [72 % 1,19 % 1,6 % 1]
51  , [123 % 1,24 % 1,6 % 1]
52  , [188 % 1,29 % 1,6 % 1]
53  , [267 % 1,34 % 1,6 % 1]
54  , [360 % 1,39 % 1,6 % 1]
55  , [467 % 1,44 % 1,6 % 1]
56  , [588 % 1,49 % 1,6 % 1]

```

```

57   ]
58
59   Let us take the transpose of this "matrix" to see the
      behavior of coefficients.
60
61   *Multivariate> let f z1 z2 = 3+2*z1+4*z2+7*z1^2+5*z1*z2
      +6*z2^2
62   *Multivariate> let fTable = tablify f 10
63   *Multivariate> map list2pol fTable
64   [[3 % 1,4 % 1,6 % 1]
65    ,[2 % 1,9 % 1,6 % 1]
66    ,[35 % 1,14 % 1,6 % 1]
67    ,[72 % 1,19 % 1,6 % 1]
68    ,[123 % 1,24 % 1,6 % 1]
69    ,[188 % 1,29 % 1,6 % 1]
70    ,[267 % 1,34 % 1,6 % 1]
71    ,[360 % 1,39 % 1,6 % 1]
72    ,[467 % 1,44 % 1,6 % 1]
73    ,[588 % 1,49 % 1,6 % 1]
74   ]
75   *multivariate> transpose it
76   [[3 % 1,12 % 1,35 % 1,72 % 1,123 % 1,188 % 1,267 %
      1,360 % 1,467 % 1,588 % 1]
77    [4 % 1,9 % 1,14 % 1,19 % 1,24 % 1,29 % 1,34 % 1,39 %
      1,44 % 1,49 % 1]
78    ,[6 % 1,6 % 1,6 % 1,6 % 1,6 % 1,6 % 1,6 % 1,6 %
      1,6 % 1]
79   ]
80   *Multivariate> map list2pol it
81   [[3 % 1,2 % 1,7 % 1]
82    ,[4 % 1,5 % 1]
83    ,[6 % 1]]
84
85   > table2pol :: [[Ratio Integer]] -> [[Ratio Integer]]
86   > table2pol = map list2pol . transpose . map list2pol
87
88   *Multivariate> let g x y = 1+7*x + 8*y + 10*x^2 + x*y
      +9*y^2
89   *Multivariate> table2pol $ tablify g 5
90   [[1 % 1,7 % 1,10 % 1],[8 % 1,1 % 1],[9 % 1]]
91
92   There are some bad-behavior polynomials;
93   *Multivariate> table2pol $ tablify (\x y -> x*y) 20
94   [[0 % 1],[1 % 1,1 % 1]]
95   *Multivariate> tablify (\x y -> (x,y)) 5

```

```

96  [[(0,0),(0,1),(0,2),(0,3),(0,4)]
97  ,[(1,0),(1,1),(1,2),(1,3),(1,4)]
98  ,[(2,0),(2,1),(2,2),(2,3),(2,4)]
99  ,[(3,0),(3,1),(3,2),(3,3),(3,4)]
100 ,[(4,0),(4,1),(4,2),(4,3),(4,4)]
101 ]
102 *Multivariate> tablify (\x y -> (x*y)) 5
103 [[0,0,0,0,0]
104 , [0,1,2,3,4]
105 , [0,2,4,6,8]
106 , [0,3,6,9,12]
107 , [0,4,8,12,16]
108 ]
109
110 Here we have assumed that the list of functions has the
    same length, but
111
112 *Multivariate> map list2pol $ tablify (\x y -> x*y) 5
113 [[0 % 1],[0 % 1,1 % 1],[0 % 1,2 % 1],[0 % 1,3 % 1],[0 %
    1,4 % 1]]
114
115 So, we should repeat 0's if we have zero-function.
116
117 > xyDegree f = (dX, dY)
118 >   where
119 >     dX = length . list2pol $ map (\t -> f t 1) [0..]
120 >     dY = length . list2pol $ map (\t -> f 1 t) [0..]
121
122 *Multivariate> let test x y = x^2*(2*y + y^3)
123 *Multivariate> uncurry (*) . xyDegree $ test
124 6
125 *Multivariate> maximum . map (length . list2pol) .
    tablify test $ 6
126 4
127 *Multivariate> map (take 4 . (++ (repeat (0%1)))) .
    list2pol) . tablify test $ 6
128 [[0 % 1,0 % 1,0 % 1,0 % 1]
129 , [0 % 1,2 % 1,0 % 1,1 % 1]
130 , [0 % 1,8 % 1,0 % 1,4 % 1]
131 , [0 % 1,18 % 1,0 % 1,9 % 1]
132 , [0 % 1,32 % 1,0 % 1,16 % 1]
133 , [0 % 1,50 % 1,0 % 1,25 % 1]
134 ]
135 *Multivariate> map list2pol . transpose $ it
136 [[0 % 1],[0 % 1,0 % 1,2 % 1],[0 % 1],[0 % 1,0 % 1,1 %

```

```

137     1]]
138 *Multivariate> let test x y = (1%3)*x^2*((2%5)*y +
139     ((3%4)*x*y^3))
140                                     -- = (2%15)*x^2*y + (1%4)
141                                     *x^3*y^3
142 *Multivariate> xyDegree test
143 (3,3)
144 *Multivariate> map (take 4 . (++ (repeat (0%1)))) .
145     list2pol) . tablify test $ 9
146 [[0 % 1,0 % 1,0 % 1,0 % 1]
147  ,[0 % 1,2 % 15,0 % 1,1 % 4]
148  ,[0 % 1,8 % 15,0 % 1,2 % 1]
149  ,[0 % 1,6 % 5,0 % 1,27 % 4]
150  ,[0 % 1,32 % 15,0 % 1,16 % 1]
151  ,[0 % 1,10 % 3,0 % 1,125 % 4]
152  ,[0 % 1,24 % 5,0 % 1,54 % 1]
153  ,[0 % 1,98 % 15,0 % 1,343 % 4]
154  ,[0 % 1,128 % 15,0 % 1,128 % 1]
155  ]
156 *Multivariate> map list2pol . transpose $ it
157 [[0 % 1],[0 % 1,0 % 1,2 % 15],[0 % 1],[0 % 1,0 % 1,0 %
158     1,1 % 4]]
159
160 > xyPol2Coef :: (Enum t, Integral a, Num t) =>
161 >     (t -> t -> Ratio a) -> [[Ratio a]]
162 > xyPol2Coef f = map list2pol . transpose . map (take num
163     . (++ (repeat (0%1)))) . list2pol) . tablify f $ rank
164 >     where
165 >         rank = uncurry (*) . xyDegree $ f
166 >         num = maximum . map (length . list2pol) . tablify
167 >             f $ rank
168
169 *Multivariate> let test x y = (1%3)*x^2*((2%5)*y +
170     ((3%4)*x*y^3))
171 *Multivariate> xyPol2Coef test
172 [[0 % 1],[0 % 1,0 % 1,2 % 15],[0 % 1],[0 % 1,0 % 1,0 %
173     1,1 % 4]]
174
175 *Multivariate> let test2 x y = x*y
176 *Multivariate> xyPol2Coef test2
177 [[0 % 1],[0 % 1,1 % 1]]
178
179 *Multivariate> let test3 x y = x^3*y^4
180 *Multivariate> xyPol2Coef test3
181 [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,0 % 1,1 %
182     1]]

```

```

172
173 > table2pol' :: Integral a => [[Ratio a]] -> [[Ratio a]]
174 > table2pol' tbl = map list2pol . transpose . map (take
      num . (++) (repeat (0%1))) . list2pol) $ tbl
175 >   where
176 >     num = maximum . map (length . list2pol) $ tbl
177
178 --
179
180 2-variable rational functions
181
182 *Univariate> let h x y = (1+2*x+4*y+7*x^2+5*x*y+6*y^2)
      % (1+7*x+8*y+10*x^2+x*y+9*y^2)
183 *Univariate> let auxh x y t = h (t*x) (t*y)
184 *Univariate> let auxhs = [map (auxh 1 y) [0..] | y <-
      [0..]]
185 *Univariate> take 10 $ map list2rat' auxhs
186 [Just ([1 % 1,2 % 1,7 % 1],[1 % 1,7 % 1,10 % 1])
187 ,Just ([1 % 1,6 % 1,18 % 1],[1 % 1,15 % 1,20 % 1])
188 ,Just ([1 % 1,10 % 1,41 % 1],[1 % 1,23 % 1,48 % 1])
189 ,Just ([1 % 1,14 % 1,76 % 1],[1 % 1,31 % 1,94 % 1])
190 ,Just ([1 % 1,18 % 1,123 % 1],[1 % 1,39 % 1,158 % 1])
191 ,Just ([1 % 1,22 % 1,182 % 1],[1 % 1,47 % 1,240 % 1])
192 ,Just ([1 % 1,26 % 1,253 % 1],[1 % 1,55 % 1,340 % 1])
193 ,Just ([1 % 1,30 % 1,336 % 1],[1 % 1,63 % 1,458 % 1])
194 ,Just ([1 % 1,34 % 1,431 % 1],[1 % 1,71 % 1,594 % 1])
195 ,Just ([1 % 1,38 % 1,538 % 1],[1 % 1,79 % 1,748 % 1])
196 ]
197 *Univariate> sequence it
198 Just [[([1 % 1,2 % 1,7 % 1],[1 % 1,7 % 1,10 % 1])
199       ,([1 % 1,6 % 1,18 % 1],[1 % 1,15 % 1,20 % 1])
200       ,([1 % 1,10 % 1,41 % 1],[1 % 1,23 % 1,48 % 1])
201       ,([1 % 1,14 % 1,76 % 1],[1 % 1,31 % 1,94 % 1])
202       ,([1 % 1,18 % 1,123 % 1],[1 % 1,39 % 1,158 % 1])
203       ,([1 % 1,22 % 1,182 % 1],[1 % 1,47 % 1,240 % 1])
204       ,([1 % 1,26 % 1,253 % 1],[1 % 1,55 % 1,340 % 1])
205       ,([1 % 1,30 % 1,336 % 1],[1 % 1,63 % 1,458 % 1])
206       ,([1 % 1,34 % 1,431 % 1],[1 % 1,71 % 1,594 % 1])
207       ,([1 % 1,38 % 1,538 % 1],[1 % 1,79 % 1,748 % 1])
208 ]
209 *Univariate> fmap (map fst) it
210 Just [[1 % 1,2 % 1,7 % 1]
211       ,[1 % 1,6 % 1,18 % 1]
212       ,[1 % 1,10 % 1,41 % 1]
213       ,[1 % 1,14 % 1,76 % 1]

```

```

214         , [1 % 1,18 % 1,123 % 1]
215         , ..
216     ]
217 *Univariate> fmap transpose it
218 Just [[1 % 1,1 % 1,1 % 1,1 % 1,1 % 1,1 % 1,1 % 1,1 %
      1,1 % 1,1 % 1]
219        , [2 % 1,6 % 1,10 % 1,14 % 1,18 % 1,22 % 1,26 %
      1,30 % 1,34 % 1,38 % 1]
220        , [7 % 1,18 % 1,41 % 1,76 % 1,123 % 1,182 % 1,253 %
      1,336 % 1,431 % 1,538 % 1]
221     ]
222 *Univariate> fmap (map list2pol) it
223 Just [[1 % 1], [2 % 1,4 % 1], [7 % 1,5 % 1,6 % 1]]
224
225 *Multivariate> let h x y = (1+2*x+4*y+7*x^2+5*x*y
      +(6%13)*y^2) / (1+(7%3)*x+8*y+10*x^2+x*y+9*y^2)
226 *Multivariate> let auxh x y t = h (t*x) (t*y)
227 *Multivariate> let auxhs = [map (auxh 1 y) [0..100] | y
      <- [0..100]]
228 *Multivariate> fmap (map list2pol . transpose . map fst
      ) . sequence . map list2rat' $ auxhs
229 Just [[1 % 1], [2 % 1,4 % 1], [7 % 1,5 % 1,6 % 13]]
230 *Multivariate> fmap (map list2pol . transpose . map snd
      ) . sequence . map list2rat' $ auxhs
231 Just [[1 % 1], [7 % 3,8 % 1], [10 % 1,1 % 1,9 % 1]]
232
233 > -- SUPER SLOW IMPLEMENTATION, DO NOT USE THIS!
234 > table2ratf
235 >   :: Integral a =>
236 >     [[Ratio a]] -> (Maybe [[Ratio a]], Maybe [[Ratio a]
      ]])
237 > table2ratf table = (t2r fst table, t2r snd table)
238 >   where
239 >     -- t2r' third = fmap (map third) . sequence . map
      list2rat'
240 >
241 >     myMax Nothing = 0
242 >     myMax (Just ns) = ns
243 >
244 >     t2r third = fmap (map list2pol . transpose . map (
      take num . (++ (repeat (0%1))) . third)) .
      mapM list2rat'
245 >
246 >     where
247 >       num = myMax . fmap (maximum . map (length . fst
      )) . mapM list2rat' $ table

```

```

248 >
249 > -- fmap (maximum . map (length . fst)) . sequence . map
      list2rat'
250 > -- map (take num . (++ (repeat (0%1)))) . list2pol)
251 >
252 > tablizer :: (Num a, Enum a) => (a -> a -> b) -> a -> [[
      b]]
253 > tablizer f n = [map (f_t 1 y) [0..(n-1)] | y <- [0..(n
      -1)]]
254 >   where
255 >     f_t x y t = f (t*x) (t*y)
256
257 *Multivariate> let h x y = (1+2*x+4*y+7*x^2+5*x*y
      +(6%13)*y^2) / (1+(7%3)*x+8*y+10*x^2+x*y+9*y^2)
258 *Multivariate> let hTable = tablizer h 20
259 *Multivariate> table2ratf hTable
260 (Just [[1 % 1],[2 % 1,4 % 1],[7 % 1,5 % 1,6 % 13]],Just
      [[1 % 1],[7 % 3,8 % 1],[10 % 1,1 % 1,9 % 1]])
261
262 Note that, the sampling points for n=10 case are
263
264 *Multivariate> tablizer (\x y -> (x,y)) 10
265 [[(0,0),(1,0),(2,0),(3,0),(4,0),(5,0),(6,0),(7,0)
      ,(8,0),(9,0)]
266 ,[(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7)
      ,(8,8),(9,9)]
267 ,[(0,0),(1,2),(2,4),(3,6),(4,8),(5,10),(6,12),(7,14)
      ,(8,16),(9,18)]
268 ,[(0,0),(1,3),(2,6),(3,9),(4,12),(5,15),(6,18),(7,21)
      ,(8,24),(9,27)]
269 ,[(0,0),(1,4),(2,8),(3,12),(4,16),(5,20),(6,24),(7,28)
      ,(8,32),(9,36)]
270 ,[(0,0),(1,5),(2,10),(3,15),(4,20),(5,25),(6,30),(7,35)
      ,(8,40),(9,45)]
271 ,[(0,0),(1,6),(2,12),(3,18),(4,24),(5,30),(6,36),(7,42)
      ,(8,48),(9,54)]
272 ,[(0,0),(1,7),(2,14),(3,21),(4,28),(5,35),(6,42),(7,49)
      ,(8,56),(9,63)]
273 ,[(0,0),(1,8),(2,16),(3,24),(4,32),(5,40),(6,48),(7,56)
      ,(8,64),(9,72)]
274 ,[(0,0),(1,9),(2,18),(3,27),(4,36),(5,45),(6,54),(7,63)
      ,(8,72),(9,81)]
275 ]
276
277 *Multivariate> let f x y = (1 + 2*x + 3*y + 4*x^2 +

```



```

      (1%5)*x*y + (1%6)*y^2)
278      / (7 + 8*x + (1%9)*y + x^2 + x
          *y + 10*y^2)
279  *Multivariate> let g x y = (11 + 10*x + 9*y) / (8 + 7*x
          ^2 + (1%6)*x*y + 5*y^2)
280  *Multivariate> table2ratf $ tablizer f 20
281  (Just [[1 % 7],[2 % 7,3 % 7],[4 % 7,1 % 35,1 % 42]]
282  ,Just [[1 % 1],[8 % 7,1 % 63],[1 % 7,1 % 7,10 % 7]])
283  *Multivariate> table2ratf $ tablizer g 20
284  (Just [[11 % 8],[5 % 4,9 % 8],[0 % 1]]
285  ,Just [[1 % 1],[0 % 1],[7 % 8,1 % 48,5 % 8]])
286  *Multivariate> let h x y = (f x y) / (g x y)
287  *Multivariate> table2ratf $ tablizer h 20
288  (Just [[8 % 77],[16 % 77,24 % 77],[39 % 77,53 % 2310,19
          % 231]
289  ,[2 % 11,64 % 231,3 % 22,15 % 77],[4 % 11,31 %
          1155,106 % 385,37 % 2772,5 % 462]]
290  ,Just [[1 % 1],[158 % 77,578 % 693],[13 % 11,757 %
          693,111 % 77]
291  ,[10 % 77,19 % 77,109 % 77,90 % 77]]
292  )
293  *Multivariate> table2ratf $ tablizer f 10
294  (** Exception: Prelude.!!: index too large
295  *Multivariate> table2ratf $ tablizer f 13
296  (** Exception: Prelude.!!: index too large
297  *Multivariate> table2ratf $ tablizer f 15
298  (Just [[1 % 7],[2 % 7,3 % 7],[4 % 7,1 % 35,1 % 42]]
299  ,Just [[1 % 1],[8 % 7,1 % 63],[1 % 7,1 % 7,10 % 7]])
300  *Multivariate> table2ratf $ tablizer g 11
301  (** Exception: Prelude.!!: index too large
302  *Multivariate> table2ratf $ tablizer g 13
303  (** Exception: Prelude.!!: index too large
304  *Multivariate> table2ratf $ tablizer g 15
305  (Just [[11 % 8],[5 % 4,9 % 8],[0 % 1]],Just [[1 % 1],[0
          % 1],[7 % 8,1 % 48,5 % 8]])
306  *Multivariate> table2ratf $ tablizer (\x y -> x/(1+x^2)
          ) 10
307  (** Exception: Prelude.!!: index too large
308  *Multivariate> table2ratf $ tablizer (\x y -> x/(1+x^2)
          ) 13
309  (** Exception: Prelude.!!: index too large
310  *Multivariate> table2ratf $ tablizer (\x y -> x/(1+x^2)
          ) 20
311  (Just [[0 % 1],[1 % 1],[0 % 1]],Just [[1 % 1],[0 %
          1],[1 % 1]])

```

```

312 *Multivariate> table2ratf $ tablizer (\x y -> x/(1+x^2)
    ) 13
313 (** Exception: Prelude.!!!: index too large
314 *Multivariate> table2ratf $ tablizer (\x y -> x/(1+x^2)
    ) 15
315 (Just [[0 % 1],[1 % 1],[0 % 1]],Just [[1 % 1],[0 %
    1],[1 % 1]])
316
317 > -- Alternative transpose, filling with the default
    value.
318 > -- I basically followed the implementation of standard
    Prelude.
319 > transposeWith :: a -> [[a]] -> [[a]]
320 > transposeWith _ [] = []
321 > transposeWith z ([] : xss)
322 > | all null xss = []
323 > | otherwise    = (z : [h | (h:_) <- xss])
324 >                  : transposeWith z ([] : [t | (_:t) <-
    xss])
325 > transposeWith z ((x:xs) : xss) = (x : [h | (h:_) <- xss
    ])
326 >                  : transposeWith z (xs :
    [t | (_:t) <- xss])
327
328 *Multivariate> let f x y = (x*y)%(1+y)^2
329 *Multivariate> let tbl = tablizer f 20
330 *Multivariate> fmap (map list2pol . (transposeWith
    (0%1)) . map fst) . sequence . map list2rat' $ tbl
331 Just [[0 % 1],[0 % 1],[0 % 1,1 % 1]]
332 *Multivariate> fmap (map list2pol . (transposeWith
    (0%1)) . map snd) . sequence . map list2rat' $ tbl
333 Just [[1 % 1],[0 % 1,2 % 1],[0 % 1,0 % 1,1 % 1]]
334
335 > table2ratf' table = (t2r fst table, t2r snd table)
336 >   where
337 >     t2r third = fmap (map list2pol . transposeWith
    (0%1) . map third) . mapM list2rat'
338 >
339 > table2ratf'' table = (t2r fst table, t2r snd table)
340 >   where
341 >     t2r third = fmap (map list2pol . transposeWith
    (0%1) . map third) . mapM list2rat''
342
343 > wilFunc x y = (x^2*y^2) % ((1 + y)^3)
344

```

```

345 *Multivariate> table2ratf $ tablizer wilFunc 20
346 (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 %
      1]])
347 ,Just [[1 % 1],[0 % 1,3 % 1],[0 % 1,0 % 1,3 % 1],[0 %
      1,0 % 1,0 % 1,1 % 1],[0 % 1]])
348 (3.91 secs, 2,850,226,792 bytes)
349 *Multivariate> table2ratf' $ tablizer wilFunc 20
350 (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 %
      1]])
351 ,Just [[1 % 1],[0 % 1,3 % 1],[0 % 1,0 % 1,3 % 1],[0 %
      1,0 % 1,0 % 1,1 % 1]])
352 (2.00 secs, 1,425,753,744 bytes)
353 *Multivariate> table2ratf'' $ tablizer wilFunc 20
354 (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 %
      1]])
355 ,Just [[1 % 1],[0 % 1,3 % 1],[0 % 1,0 % 1,3 % 1],[0 %
      1,0 % 1,0 % 1,1 % 1]])
356 (1.73 secs, 1,234,282,424 bytes)

```

4.5 FROverZp.lhs

Listing 4.5: FROverZp.lhs

```

1 FROverZp.lhs
2
3 > module FROverZp where
4
5 Functional Reconstruction over finite field Z_p
6
7 > import Data.Ratio
8 > import Data.Maybe
9 > import Data.Numbers.Primes
10 > import Data.List (null, transpose)
11 > import Control.Monad (sequence)
12 >
13 > import Ffield (modp, bigPrimes, reconstruct)
14 > -- , inversep, bigPrimes, recCRT, recCRT')
15 > import Univariate (newtonC, firstDifs, list2npol)
16
17 Univariate Polynomial case
18 Our target is a univariate polynomial
19 f :: (Integral a) =>
20     Ratio a -> Ratio a -- Real?
21
22 > -- Function-modular, now our modp function is wrapped

```

```

    by Maybe.
23 > fmodp :: (a -> Ratio Int) -> Int -> a -> Maybe Int
24 > f 'fmodp' p = ('modp' p) . f
25
26 *FROverZp> let f x = (1%3) + (3%5)*x + (7%13)*x^2
27 *FROverZp> take 10 $ map (f 'fmodp' 13) [0..]
28 [Just 9,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,
    ,Nothing,Nothing,Nothing]
29 *FROverZp> take 10 $ map (f 'fmodp' 19) [0..]
30 [Just 13,Just 8,Just 7,Just 10,Just 17,Just 9,Just 5,
    Just 5,Just 9,Just 17]
31
32 Since we have chosen
33 *Ffield> last bigPrimes
34 10939
35 we can put a finite number of outputs as our accessible "
    row" data.
36
37 Difference analysis over Z_p
38 Every arithmetic should be on Z_p, i.e., ('mod' p).
39
40 > accessibleData :: (Num a, Enum a) =>
41 >                 (a -> Ratio Int) -> Int -> [Maybe Int]
42 > accessibleData f p = take p $ map (f 'fmodp' p) [0..]
43 >
44 > accessibleData' :: [Ratio Int] -> Int -> [Maybe Int]
45 > accessibleData' fs p = take p $ map ('modp' p) fs
46
47 *FROverZp> let helper x y = (-) <$> x <*> y
48 *FROverZp> :type helper
49 helper :: (Applicative f, Num b) => f b -> f b -> f b
50 *FROverZp> :t map helper
51 map helper :: (Applicative f, Num b) => [f b] -> [f b
    -> f b]
52 *FROverZp> let myDif xs = zipWith helper (tail xs) xs
53 *FROverZp> myDif [Just 5,Just 0,Just 2,Just 4,Just 6,
    Just 1,Just 3]
54 [Just (-5),Just 2,Just 2,Just 2,Just (-5),Just 2]
55 *FROverZp> map (fmap ('mod' 13)) it
56 [Just 8,Just 2,Just 2,Just 2,Just 8,Just 2]
57 *FROverZp> map (fmap ('mod' 13)) . myDif $ [Just 5,Just
    0,Just 2,Just 4,Just 6,Just 1,Just 3, Nothing,
    Just 1, Just 2]
58 [Just 8,Just 2,Just 2,Just 2,Just 8,Just 2,Nothing,

```

```

    Nothing,Just 1]
59
60 > -- difsp :: (Applicative f, Integral b) => b -> [f b]
    -> [f b]
61 > difsp :: Applicative f => Int -> [f Int] -> [f Int]
62 > difsp p = map (fmap ('mod' p)) . difsp'
63 > where
64 >     difsp' :: (Applicative f, Num b) => [f b] -> [f b]
65 >     difsp' xs = zipWith helper (tail xs) xs
66 >     helper :: (Applicative f, Num b) => f b -> f b -> f
    b
67 >     helper x y = (-) <$> x <*> y
68
69 *FR0verZp> difsp 13 [Just 5,Just 0,Just 2,Just 4,Just
    6,Just 1,Just 3, Nothing, Just 1, Just 2]
70 [Just 8,Just 2,Just 2,Just 2,Just 8,Just 2,Nothing,
    Nothing,Just 1]
71
72 > difListsp :: (Applicative f, Eq (f Int)) => Int -> [[f
    Int]] -> [[f Int]]
73 > difListsp _ [] = []
74 > difListsp p xx@(xs:xss) =
75 >     if isConst xs
76 >     then xx
77 >     else difListsp p $ difsp p xs : xx
78 > where
79 >     isConst (i:jj@(j:js)) = all (==i) jj
80 >     isConst _ = error "difListsp:␣"
81
82 *FR0verZp> let f x = (1%3) + (3%5)*x + (7%13)*x^2
83 *FR0verZp> let ds = accessibleData f 101
84 *FR0verZp> map head $ difListsp 101 [ds]
85 [Just 71,Just 26,Just 34]
86
87 Degree, eager and lazy versions
88
89 > degreeep' :: (Applicative f, Eq (f Int)) => Int -> [f
    Int] -> Int
90 > degreeep' p xs = length (difListsp p [xs]) - 1
91 >
92 > degreeepLazy :: (Applicative f, Num t, Eq (f Int)) =>
    Int -> [f Int] -> t
93 > degreeepLazy p xs = helper xs 0
94 > where
95 >     helper as@(a:b:c:_) n

```

```

96 >      | a==b && b==c = n -- two times matching
97 >      | otherwise    = helper (difsp p as) (n+1)
98 >
99 > degreep :: (Applicative f, Eq (f Int)) => Int -> [f Int
    ] -> Int
100 > degreep p xs = let l = degreepLazy p xs in
101 >   degreep' p $ take (l+2) xs
102
103 *FR0verZp> let f x = (1%3) + (3%5)*x + (7%13)*x^2
104 *FR0verZp> degreep 101 $ accessibleData f 101
105 2
106 *FR0verZp> degreep 103 $ accessibleData f 103
107 2
108 *FR0verZp> degreep 107 $ accessibleData f 107
109 2
110 *FR0verZp> degreep 11 $ accessibleData f 11
111 2
112 *FR0verZp> degreep 13 $ accessibleData f 13
113 1
114 *FR0verZp> degreep 17 $ accessibleData f 17
115 2
116
117 > firstDifsp :: (Applicative f, Eq (f Int)) => Int -> [f
    Int] -> [f Int]
118 > firstDifsp p xs = reverse $ map head $ difListsp p [xs
    ']
119 >   where
120 >     xs' = take n xs
121 >     n   = 2 + degreep p xs
122 >
123 > makeAPair :: [Int] -> [Ratio Int] -> [(Int, Maybe [Int
    ])]
124 > makeAPair ps fs = zip ps . map (sequence . (\p ->
    firstDifsp p (fsp p))) $ ps
125 >   where
126 >     fsp = accessibleData' fs
127
128 *FR0verZp> let f x = (1%3) + (3%5)*x + (7%13)*x^2
129 *FR0verZp> let fs = map f [0..]
130 *FR0verZp> let smallerPrimes = filter isPrime [11, 13,
    19, 23, 101, 103, 107]
131 *FR0verZp> makeAPair smallerPrimes fs
132 [(11,Just [4,3,7])
133 ,(13,Nothing)
134 ,(19,Just [13,14,4])

```

```

135     ,(23,Just [8,16,17])
136     ,(101,Just [34,26,71])
137     ,(103,Just [69,36,9])
138     ,(107,Just [36,39,34])
139   ]
140
141   *FR0verZp> makeAPair smallerPrimes fs
142   [(11,Just [4,3,7])
143    ,(13,Nothing)
144    ,(19,Just [13,14,4])
145    ,(23,Just [8,16,17])
146    ,(101,Just [34,26,71])
147    ,(103,Just [69,36,9])
148    ,(107,Just [36,39,34])
149   ]
150   *FR0verZp> filter (isJust . snd) it
151   [(11,Just [4,3,7])
152    ,(19,Just [13,14,4])
153    ,(23,Just [8,16,17])
154    ,(101,Just [34,26,71])
155    ,(103,Just [69,36,9])
156    ,(107,Just [36,39,34])
157   ]
158   *FR0verZp> map (\(p, xs) -> (zip (sequence xs) (repeat
159     p))) it
160   [[(Just 4,11),(Just 3,11),(Just 7,11)]
161    ,[(Just 13,19),(Just 14,19),(Just 4,19)]
162    ,[(Just 8,23),(Just 16,23),(Just 17,23)]
163    ,[(Just 34,101),(Just 26,101),(Just 71,101)]
164    ,[(Just 69,103),(Just 36,103),(Just 9,103)]
165    ,[(Just 36,107),(Just 39,107),(Just 34,107)]
166   ]
167   *FR0verZp> transpose it
168   [[(Just 4,11),(Just 13,19),(Just 8,23),(Just 34,101),(
169     Just 69,103),(Just 36,107)]
170    ,[(Just 3,11),(Just 14,19),(Just 16,23),(Just 26,101),(
171     Just 36,103),(Just 39,107)]
172    ,[(Just 7,11),(Just 4,19),(Just 17,23),(Just 71,101),(
173     Just 9,103),(Just 34,107)]
174   ]
175   *FR0verZp> :t it
176   it :: [(Maybe Int, Int)]
177   *FR0verZp> :t reconstruct
178   reconstruct :: [(Maybe Int, Int)] -> Maybe (Ratio
179     Integer, Integer)

```

```

175 *FR0verZp> map reconstruct it
176 [Just (1 % 3,209),Just (74 % 65,485507),Just (14 %
    13,209)]
177
178 *FR0verZp> let f x = (1%3) + (3%5)*x + (7%13)*x^2
179 *FR0verZp> let fs = map f [0..]
180 *FR0verZp> list2npol fs
181 [1 % 3,74 % 65,7 % 13]
182
183 > firstDifs' :: [Ratio Int] -> [Maybe (Ratio Integer,
    Integer)]
184 > firstDifs' = map reconstruct . transpose . map (\(p,xs)
    -> (zip (sequence xs) (repeat p))) .
185 >           filter (isJust .snd) . makeAPair bigPrimes
186
187 *FR0verZp> let g x = 1%153 + x*(133%122) + (x^2)
    *(1%199) + (x^3)*(922%855)
188 *FR0verZp> let gs = map g [0..]
189 *FR0verZp> fmap (newtonC . map fst) . sequence .
    firstDifs' $ take 100 gs
190 Just [1 % 153,45117911 % 20757690,183763 % 56715,922 %
    855]
191 *FR0verZp> list2npol gs
192 [1 % 153,45117911 % 20757690,183763 % 56715,922 % 855]
193
194 > -- Eager-version, i.e., input should be finite.
195 > list2npolp :: [Ratio Int] -> Maybe [Ratio Integer]
196 > list2npolp = fmap (newtonC . map fst) . sequence .
    firstDifs'
197
198 *FR0verZp> let g x = 1%153 + x*(133%122) + (x^2)
    *(1%199) + (x^3)*(922%855)
199 *FR0verZp> let gs = map g [0..100]
200 *FR0verZp> list2npol gs
201 [1 % 153,45117911 % 20757690,183763 % 56715,922 % 855]
202 *FR0verZp> list2npolp gs
203 Just [1 % 153,45117911 % 20757690,183763 % 56715,922 %
    855]
204
205
206 --
207 Univariate Rational function case
208 Since thiele2coef uses only (*), (+) and (-) operations,
    we don't have to do these calculation over prime
    fields.

```



```

209 So, our target should be rho function (matrix?)
    calculation.
210
211 Reciprocal difference
212
213 > {-
214 > rhoZp :: Integral a => [Ratio a] -> a -> Int -> a -> a
215 > rhoZp fs 0 i p = (fs !! i) 'modp' p
216 > rhoZp fs n i p
217 >   | n <= 0      = 0
218 >   | otherwise = (n*inv + rhoZp fs (n-2) (i+1) p) 'mod'
    p
219 >   where
220 >     inv = fromJust inv'
221 >     inv' = (rhoZp fs (n-1) (i+1) p - rhoZp fs (n-1) i p)
    'inversep' p
222 >
223 > aZp :: Integral a => [Ratio a] -> a -> a -> a
224 > aZp fs 0 p = head fs 'modp' p
225 > aZp fs n p = (rhoZp fs n 0 p - rhoZp fs (n-2) 0 p) 'mod'
    p
226 >
227 > tDegreeZp fs p = helper fs 0 p
228 >   where
229 >     helper fs n p
230 >       | isConst fs' = n
231 >       | otherwise   = helper fs (n+1) p
232 >     where
233 >       fs' = map (rhoZp fs n p) [0..]
234 >       isConst (i:j:_) = i==j
235
236 *FROverZp> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
237 *FROverZp> let hs = map h [0..]
238 *FROverZp> take 5 $ map (\n -> rhoZp hs 0 n 101) [0..]
239 [3,89,64,8,16]
240 *FROverZp> take 5 $ map (\n -> rhoZp hs 1 n 101) [0..]
241 [74,4,9,38,65]
242 *FROverZp> take 5 $ map (\n -> rhoZp hs 2 n 101) [0..]
243 [*** Exception: Maybe.fromJust: Nothing
244
245 > -}]

```