Finite fields and functional reconstructions

Ray D. Sameshima

2016/09/23 $\sim$ 2017/04/21 18:50

Contents

0 Preface **7**

0.1 References 7

0.2 Set theoretical gadgets 7

0.2.1 Numbers 7

0.2.2 Algebraic structures 8

0.3 Haskell language 8

1 Basics **11**

1.1 Finite fields 11

1.1.1 Rings 11

1.1.2 Fields 12

1.1.3 An example of finite rings $\mathbb{Z}_n$ 12

1.1.4 Bézout's lemma 12

1.1.5 Greatest common divisor 13

1.1.6 Extended Euclidean algorithm 14

1.1.7 Coprime as a binary relation 18

1.1.8 Corollary (Inverses in $\mathbb{Z}_n$) 18

1.1.9 Corollary (Finite field $\mathbb{Z}_p$) 19

1.2 Rational number reconstruction 21

1.2.1 A map from $\mathbb{Q}$ to $\mathbb{Z}_p$ 21

1.2.2 Reconstruction from $\mathbb{Z}_p$ to $\mathbb{Q}$ 22

1.2.3 Chinese remainder theorem 25

1.2.4 `reconstruct`: from image in $\mathbb{Z}_p$ to rational number . 28

1.3 Polynomials and rational functions 30

1.3.1 Notations 30

1.3.2 Polynomials and rational functions 30

1.3.3 As data, coefficients list 31

1.4 Haskell implementation of univariate polynomials 32

1.4.1 A polynomial as a list of coefficients 32

1.4.2 Difference analysis 35

2 Functional reconstruction over $\mathbb{Q}$ 39
2.1 Univariate polynomials 39
2.1.1 Newtons' 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 `list2pol`: 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 `tDegree` for termination 50
2.4.3 `thieleC`: from output list to Thiele coefficients 52
2.4.4 Haskell representation for rational functions 53
2.4.5 `list2rat`: 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 Special data types 69
3.1.2 Implementations 69
3.2 Univariate rational functions 75
3.3 TBA Univariate rational functions 75

4 Codes 77
4.1 `Ffield.lhs` 77
4.2 `Polynomials.hs` 84

4.3 Univariate.lhs 86
4.4 Multivariate.lhs 99
4.5 GUniFin.lhs 109
4.6 GMulFin.lhs 133

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: $\emptyset, \mathrm{suc}$)

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, what's 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)

[3] Of 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 and basic algebraic structures.

1.1 Finite fields

1.1.1 Rings

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

```
(+) :: R -> R -> R      (1.1)
(*) :: R -> R -> R      (1.2)
```

satisfying the following 3 (ring) axioms:

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

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

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

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

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

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

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

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

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

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

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

1.1.2 Fields

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

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

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

$$\begin{aligned} \mathbb{Z}_n &:= \mathbb{Z}/n\mathbb{Z} && (1.12) \\ &\cong \{0, \cdots, (n-1)\} && (1.13) \end{aligned}$$

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

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.

$$a * x + b * y = \gcd(a, b), \tag{1.19}$$

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

[1] Here we have taken an equivalence class,

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

with the following operations:

$$\begin{aligned} [k] + [l] &:= [k + l] && (1.15) \\ [k] * [l] &:= [k * l] && (1.16) \end{aligned}$$

This is equivalent to take modular n:

$$\begin{aligned} (k \mod n) + (l \mod n) &:= (k + l \mod n) && (1.17) \\ (k \mod n) * (l \mod n) &:= (k * l \mod n). && (1.18) \end{aligned}$$

1.1.5 Greatest common divisor

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

```
> -- Eucledian 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`.

$$\texttt{myGCD 7 13 == myGCD 13 7} \tag{1.20}$$

Then it hits main line:

$$\texttt{myGCD 13 7 == myGCD (13-7) 7} \tag{1.21}$$

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

$$\begin{aligned} \texttt{myGCD (13-7) 7} &\ \texttt{==}\ \texttt{myGCD 6 7} && (1.22)\\ &\ \texttt{==}\ \texttt{myGCD 7 6} && (1.23)\\ &\ \texttt{==}\ \texttt{myGCD (7-6) 6} && (1.24)\\ &\ \texttt{==}\ \texttt{myGCD 1 6} && (1.25)\\ &\ \texttt{==}\ \texttt{myGCD 6 1} && (1.26) \end{aligned}$$

Finally it ends with 1:

$$\texttt{myGCD 1 1 == 1} \tag{1.27}$$

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

As another example, consider 15 and 25:

$$
\begin{aligned}
\texttt{myGCD 15 25} &== \texttt{myGCD 25 15} && (1.28)\\
&== \texttt{myGCD (25-15) 15} && (1.29)\\
&== \texttt{myGCD 10 15} && (1.30)\\
&== \texttt{myGCD 15 10} && (1.31)\\
&== \texttt{myGCD (15-10) 10} && (1.32)\\
&== \texttt{myGCD 5 10} && (1.33)\\
&== \texttt{myGCD 10 5} && (1.34)\\
&== \texttt{myGCD (10-5) 5} && (1.35)\\
&== \texttt{myGCD 5 5} && (1.36)\\
&== \texttt{5} && (1.37)
\end{aligned}
$$

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
```

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

$$
\begin{aligned}
(r_0, s_0, t_0) &:= (a, 1, 0) && (1.38)\\
(r_1, s_1, t_1) &:= (b, 0, 1) && (1.39)
\end{aligned}
$$

and inductively, for $i \geq 2$,

$$\begin{aligned} q_i &:= \mathrm{quot}(r_{i-2}, r_{i-1}) && (1.40)\\ r_i &:= r_{i-2} - q_i * r_{i-1} && (1.41)\\ s_i &:= s_{i-2} - q_i * s_{i-1} && (1.42)\\ t_i &:= t_{i-2} - q_i * t_{i-1}. && (1.43) \end{aligned}$$

The termination condition[3] is

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

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

$$\begin{aligned} \gcd(a, b) &= r_{k-1} && (1.45)\\ x &= s_{k-1} && (1.46)\\ y &= t_{k-1}. && (1.47) \end{aligned}$$

Proof

By definition,

$$\begin{aligned} \gcd(r_{i-1}, r_i) &= \gcd(r_{i-1}, r_{i-2} - q_i * r_{i-1}) && (1.48)\\ &= \gcd(r_{i-1}, r_{i-2}) && (1.49) \end{aligned}$$

and this implies

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

i.e.,

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

Next, for $i = 0, 1$ observe

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

Let $i \geq 2$, then

$$\begin{aligned} r_i &= r_{i-2} - q_i * r_{i-1} && (1.53)\\ &= a * s_{i-2} + b * t_{i-2} - q_i * (a * s_{i-1} + b * t_{i-1}) && (1.54)\\ &= a * (s_{i-2} - q_i * s_{i-1}) + b * (t_{i-2} - q_i * t_{i-1}) && (1.55)\\ &=: a * s_i + b * t_i. && (1.56) \end{aligned}$$

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

Therefore, inductively we get

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

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
  http://qiita.com/bra_cat_ket/items/205c19611e21f3d422b7

> exGCD'
>   :: (Integral n) =>
>      n -> n -> ([n], [n], [n], [n])
> exGCD' a b = (qs, rs, ss, ts)
>   where
>     qs = zipWith quot rs (tail rs)
>     rs = takeBefore (==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)
>
> takeBefore
>   :: (a -> Bool) -> [a] -> [a]
> takeBefore p = foldr func []
>   where
>     func x xs
>       | p x       = []
>       | otherwise = x : xs
```

Here we have used so called lazy lists, and higher order function[4]. 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
```

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 to evaluate the quotient list `qs`. So, in order to get gcd and (x, y) we don't need quotients list.

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

```
*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

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

iff a and n are coprime.

Proof

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

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

Therefore

$$\begin{aligned} a \text{ and } n \text{ are coprime} \quad &\Leftrightarrow \quad \exists s, t \in \mathbb{Z}, a * s + n * t = 1 & (1.61) \\ &\Leftrightarrow \quad \exists s, t' \in \mathbb{Z}, a * s = 1 + n * t'. & (1.62) \end{aligned}$$

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

$$a * s = 1 \mod n. \tag{1.63}$$

We will make a Haskell implementation in §1.1.9.
■

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

If p is prime, then

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

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

Proof

It suffices to show that

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

but since p is prime, and

$$\forall a \in \mathbb{Z}_p, a \neq 0 \Rightarrow \texttt{gcd a p == 1} \tag{1.66}$$

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 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 candidates of inverses. Take the last one, `(1,1,-1)`. This is the image of `exGcd 11 10`, and

$$1 = 10 * 1 + 11 * (-1) \tag{1.67}$$

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

$$\begin{aligned} 10^{-1} &= -1 \mod 11 && (1.68)\\ &= 10 \mod 11 && (1.69) \end{aligned}$$

In fact,

$$10 * 10 = 11 * 9 + 1. \tag{1.70}$$

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 function[5]:

```
> -- 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]
```

[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.

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

$$- \mod p : \mathbb{Z} \to \mathbb{Z}_p; a \mapsto (a \mod p), \tag{1.71}$$

and a natural inclusion (or a forgetful map)[6]

$$¿ : \mathbb{Z}_p \hookrightarrow \mathbb{Z}. \tag{1.73}$$

Then we can define a map

$$- \mod p : \mathbb{Q} \to \mathbb{Z}_p \tag{1.74}$$

by[7]

$$q = \frac{a}{b} \mapsto (q \mod p) := \left(\left(a \times ¿\left(b^{-1} \mod p\right)\right) \mod p\right). \tag{1.75}$$

Example and implementation

An easy implementation is the followings:[8]

```
> -- A map from Q to Z_p, where p is a prime.
> modp
>   :: Ratio Int -> Int -> Maybe Int
> q `modp` p
>   | coprime b p = Just $ (a * (bi `mod` p)) `mod` p
>   | otherwise   = Nothing
>   where
```

[6] By introducing this forgetful map, we can use

$$\times : (\mathbb{Z}, \mathbb{Z}) \to \mathbb{Z} \tag{1.72}$$

of normal product on $\mathbb{Z}$ in eq.(1.75).

[7] This is an example of operator overloadings.

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

$$\texttt{add 1 2 == 1 `add` 2} \tag{1.76}$$

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

$$\texttt{(+) 1 2 == 1 + 2} \tag{1.77}$$

```
>     (a,b)   = (numerator q, denominator q)
>     Just bi = b `inversep` p
>
> -- When the denominator of q is not proprtional to p, use this.
> modp'
>   :: Ratio Int -> Int -> Int
> q `modp'` p = (a * (bi `mod` p)) `mod` p
>   where
>     (a,b)   = (numerator q, denominator q)
>     bi = b `inversep'` p
```

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

```
Example: on 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
```

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

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

$$= (3 \times 8) \mod 11 \tag{1.79}$$

$$= 24 \mod 11 \tag{1.80}$$

$$= 2 \mod 11. \tag{1.81}$$

Haskell returns the same result

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

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

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

$$a := q \mod p \tag{1.82}$$

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).

$$a * s_i + p * t_i = r_i \tag{1.83}$$

Therefore

$$r_i = a * s_i \mod p. \tag{1.84}$$

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

$$r_i^2, s_i^2 < p \tag{1.85}$$

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 choose three primes

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

The following 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 $s_4 = 3$ of $\mathbb{Z}_{101}$. The same facts on $\mathbb{Z}_{103}$ and $\mathbb{Z}_{107}$ give us the same guess $\frac{1}{3}$, and that the reconstructed number.

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

```
> -- This is guess function without Chinese Reminder Theorem.
> guess
>   :: Integral t =>
>      (Maybe t, t)       -- (q `modp` p, p)
>   -> Maybe (Ratio t, t)
> guess (Nothing, _) = Nothing
> guess (Just a, p) = let (_,rs,ss,_) = exGCD' a p in
>   Just (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*r <= p && s*s <= p = r%s
>         | otherwise                      = select rs ss p
```

We put a list of big primes as follows.

```
> -- Hard code of big primes
> -- We have chosen a finite number (100) version.
> bigPrimes :: [Int]
> bigPrimes = take 100 $ dropWhile (<10^4) primes
```

```
*Ffield> bigPrimes
[10007,10009,10037,10039,10061,10067,10069,10079,10091,10093,10099,10103
,10111,10133,10139,10141,10151,10159,10163,10169,10177,10181,10193,10211
,10223,10243,10247,10253,10259,10267,10271,10273,10289,10301,10303,10313
,10321,10331,10333,10337,10343,10357,10369,10391,10399,10427,10429,10433
,10453,10457,10459,10463,10477,10487,10499,10501,10513,10529,10531,10559
,10567,10589,10597,10601,10607,10613,10627,10631,10639,10651,10657,10663
,10667,10687,10691,10709,10711,10723,10729,10733,10739,10753,10771,10781
,10789,10799,10831,10837,10847,10853,10859,10861,10867,10883,10889,10891
,10903,10909,10937,10939
]
```

This choice of primes of order $O(10^4)$ let our `guess` function reconstruct rational numbers up to

$$\frac{O(10^2)}{O(10^2)}. \tag{1.86}$$

Good and bad examples

Our guess function can find correct answer from the images of $\frac{12}{13}$.

```
*Ffield> let knownData q = zip (map (modp q) bigPrimes) bigPrimes
*Ffield> let ds = knownData (12%13)
*Ffield> map guess ds
[Just (12 % 13,10007)
,Just (12 % 13,10009)
,Just (12 % 13,10037)
,Just (12 % 13,10039) ..
```

However, for $\frac{112}{113}$, it gets wrong answer.

```
*Ffield> let ds' = knownData (112%113)
*Ffield> map guess ds'
[Just ((-39) % 50,10007)
,Just ((-41) % 48,10009)
,Just ((-69) % 20,10037)
,Just ((-71) % 18,10039) ..
```

A solution of this problem is next subsection.

1.2.3 Chinese remainder theorem

From wikipedia[9]

> 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, using list comprehension.

```
*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

$$n \mod 3 = 2, n \mod 5 = 3, n \mod 7 = 2. \tag{1.87}$$

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

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

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

$$x = a_1 \mod n_1 \quad (1.88)$$
$$x = a_2 \mod n_2 \quad (1.89)$$

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

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

$$n_1 * m_1 = 1 \mod n_2 \quad (1.92)$$
$$n_2 * m_2 = 1 \mod n_1 \quad (1.93)$$

that is[11]

$$m_1 = n_1^{-1} \mod n_2 \quad (1.94)$$
$$m_2 = n_2^{-1} \mod n_1. \quad (1.95)$$

Then

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

is a solution.

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

$$a - a' = 0 \mod n_1 \quad (1.97)$$
$$a - a' = 0 \mod n_2. \quad (1.98)$$

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

$$a - a' = 0 \mod (n_1 * n_2). \quad (1.99)$$

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

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

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

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

Haskell implementation

Let us see how our naive `guess` function fail one more time. We make a helper function for tests.

```
> imagesAndPrimes :: Ratio Int -> [(Maybe Int, Int)]
> imagesAndPrimes q = zip (map (modp q) bigPrimes) bigPrimes
```

```
*Ffield> let q = 895%922
*Ffield> let knownData = imagesAndPrimes q
*Ffield> let [(a1,p1),(a2,p2)] = take 2 knownData
*Ffield> take 2 knownData
[(Just 6003,10007),(Just 9782,10009)]
*Ffield> map guess it
[Just ((-6) % 5,10007),Just (21 % 44,10009)]
```

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

```
Our data is a list of the type
  [(Maybe Int, Int)]
In order to use CRT, we should cast its type.

> toInteger2 :: [(Maybe Int, Int)] -> [(Maybe Integer, Integer)]
> toInteger2 = map helper
>   where
>     helper (x,y) = (fmap toInteger x, toInteger y)
>
> crtRec':: Integral a => (Maybe a, a) -> (Maybe a, a) -> (Maybe a, a)
> crtRec' (Nothing,p) (_,q)       = (Nothing, p*q)
> crtRec' (_,p)       (Nothing,q) = (Nothing, p*q)
> crtRec' (Just a1,p1) (Just a2,p2) = (Just a,p)
>   where
>     a = (a1*p2*m2 + a2*p1*m1) `mod` p
>     Just m1 = p1 `inversep` p2
>     Just m2 = p2 `inversep` 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.

```
*Ffield> let ds = imagesAndPrimes (1123%1135)
*Ffield> map guess ds
[Just (25 % 52,10007)
,Just ((-81) % 34,10009)
,Just ((-88) % 63,10037) ..

*Ffield> matches3 it
Nothing

*Ffield> scanl1 crtRec' ds

*Ffield> scanl1 crtRec' . toInteger2 $ ds
[(Just 3272,10007)
,(Just 14913702,100160063)
,(Just 298491901442,1005306552331) ..

*Ffield> map guess it
[Just (25 % 52,10007)
,Just (1123 % 1135,100160063)
,Just (1123 % 1135,1005306552331)
,Just (1123 % 1135,10092272478850909) ..

*Ffield> matches3 it
Just (1123 % 1135,100160063)
```

Schematically, this `scanl1 f` function takes

$$[d_0, d_1, d_2, d_3, \cdots] \tag{1.100}$$

and returns

$$[d_0, f(d_0, d_1), f(f(d_0, d_1), d_2), f(f(f(d_0, d_1), d_2), d_3), \cdots] \tag{1.101}$$

1.2.4 `reconstruct`: from image in $\mathbb{Z}_p$ to rational number

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

```
We should determine the number of matches to cover the range
of machine size integer, i.e., Int of Haskell.

  *Ffield> let mI = maxBound :: Int
  *Ffield> mI == 2^63-1
  True
  *Ffield> logBase 10 (fromIntegral mI)
  18.964889726830812

Since our choice of bigPrimes are
  O(10^4)
5 times is enough to cover the machine size integers.

> reconstruct :: [(Maybe Int, Int)] -> Maybe (Ratio Integer)
> reconstruct = matches 5 . makeList -- 5 times match
>   where
>     matches n (a:as)
>       | all (a==) $ take (n-1) as = a
>       | otherwise                 = matches n as
>
>     makeList = map (fmap fst . guess) . scanl1 crtRec' . toInteger2
>                . filter (isJust . fst)

> reconstruct' :: [(Maybe Int, Int)] -> Maybe (Ratio Int)
> reconstruct' = fmap coersion . reconstruct
>   where
>     coersion :: Ratio Integer -> Ratio Int
>     coersion q = (fromInteger . numerator $ q)
>                  % (fromInteger . denominator $ q)

  *Ffield> let q = 513197683989569 % 1047805145658 :: Ratio Int
  *Ffield> let ds = imagesAndPrimes q
  *Ffield> let answer = fmap fromRational . reconstruct $ ds
  *Ffield> answer :: Maybe (Ratio Int)
  Just (513197683989569 % 1047805145658)
```

Here is some random checks and results.

```
-- QuickCheck
```

```
> prop_rec :: Ratio Int -> Bool
> prop_rec q = Just q == answer
>   where
>     answer :: Maybe (Ratio Int)
>     answer = fmap fromRational . reconstruct $ ds
>     ds = imagesAndPrimes q
```

```
*Ffield> quickCheckWith stdArgs { maxSuccess = 100000 } prop_rec
+++ OK, passed 100000 tests.
```

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:

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

A monomial is defined as

$$z^\alpha := \prod_i z_i^{\alpha_i}. \tag{1.103}$$

The total degree of this monomial is given by

$$|\alpha| := \sum_i \alpha_i. \tag{1.104}$$

1.3.2 Polynomials and rational functions

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

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

where

$$c_\alpha \in \mathbb{K}. \tag{1.106}$$

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

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

We denote

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

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 $p(z), q(z) \in \mathbb{K}[z]$:

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

We denote

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

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

$$\sum_{\alpha} c_\alpha z^\alpha \tag{1.111}$$

as a set of coefficients

$$\{c_\alpha\}_\alpha. \tag{1.112}$$

Similarly, for a rational function, we can identify

$$\frac{\sum_\alpha n_\alpha z^\alpha}{\sum_\beta d_\beta z^\beta} \tag{1.113}$$

as an ordered pair of coefficients

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

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

1.4 Haskell implementation of univariate polynomials

Here we basically follows some part of §9 of ref.3, and its addendum[12].

```
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.

```
-- 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
```

[12] See http://homepages.cwi.nl/~jve/HR/PolAddendum.pdf

```
  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
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

```
(*) :: Num a => a -> a -> a
```
(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

$$3 * \left(1 + 2z + 3z^2 + 4z^3\right) \tag{1.116}$$

the result should be

$$3 * \left(1 + 2z + 3z^2 + 4z^3\right) = 3 + 6z + 9z^2 + 12z^3 \tag{1.117}$$

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

$$(1 + 2z) * (3 + 4z + 5z^2 + 6z^3) \tag{1.118}$$

the result should be

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

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:

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

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 us[13]

```
*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]
```

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

$$\texttt{[0..]} \tag{1.123}$$

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

$$\texttt{map f [0..] = [f 0, f 1 ..]} \tag{1.124}$$

For example

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

For example,

$$f(x) := x^2 + 1 \tag{1.121}$$
$$f := \lambda x.x^2 + 1 \tag{1.122}$$

are the same definition.

```
*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

$$\Delta(f)(n) := f(n+1) - f(n). \tag{1.125}$$

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]
```

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

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

and

$$\begin{aligned}
\Delta(f)(n) &:= f(n+1) - f(n) && (1.127)\\
&= \sum_{i=0}^{R} c_i \left\{(n+1)^i - n^i\right\} && (1.128)\\
&= \sum_{i=1}^{R} c_i \left\{(n+1)^i - n^i\right\} && (1.129)\\
&= \sum_{i=1}^{R} c_i \left\{i * n^{i-1} + O(n^{i-2})\right\} && (1.130)\\
&= c_R * R * n^{R-1} + O(n^{R-2}) && (1.131)
\end{aligned}$$

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

This guarantees the following function will terminate in finite steps[14]; `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]
]
```

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)
```

[14] If a given lists is generated by a polynomial.

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:

$$
\begin{aligned}
f_0(z) &= a_0 && (2.1)\\
f_1(z) &= a_0 + (z - y_0)a_1 && (2.2)\\
&\vdots \\
f_r(z) &= a_0 + (z - y_0)\,(a_1 + (z - y_1)(\cdots + (z - y_{r-1})a_r) && (2.3)\\
&= f_{r-1}(z) + (z - y_0)(z - y_1)\cdots(z - y_{r-1})a_r, && (2.4)
\end{aligned}
$$

where

$$
\begin{aligned}
a_0 &= f(y_0) && (2.5)\\
a_1 &= \frac{f(y_1)-a_0}{y_1-y_0} && (2.6)\\
&\vdots\\
a_r &= \left(\left((f(y_r)-a_0)\frac{1}{y_r-y_0}-a_1\right)\frac{1}{y_r-y_1}-\cdots-a_{r-1}\right)\frac{1}{y_r-y_{r-1}} && (2.7)
\end{aligned}
$$

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

$$f_r(n) = f(n), 0 \le n \le r. \tag{2.8}$$

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

$$
\begin{aligned}
f(z) &= f_R(z) && (2.9)\\
&= \sum_{r=0}^{R} a_r \prod_{i=0}^{r-1}(z-y_i). && (2.10)
\end{aligned}
$$

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

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

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

$$\sum_{r=0}^{R} c_r z^r. \tag{2.12}$$

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

$$\texttt{[0..]} \tag{2.13}$$

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

$$\texttt{map f [0..] = [f 0, f 1, ..]} \tag{2.14}$$

of `f :: Ratio Int -> Ratio Int` is our output list.

Then we have slightly simpler forms of coefficients:

$$
\begin{aligned}
f_r(z) &:= a_0 + z * (a_1 + (z-1)\,(a_2 + (z-2)\,(a_3 + \cdots + (z-r+1)a_r))) && (2.15)\\
a_0 &= f(0) && (2.16)\\
a_1 &= f(y_1) - a_0 && (2.17)\\
&= f(1) - f(0) =: \Delta(f)(0) && (2.18)\\
a_2 &= \frac{f(2)-a_0}{2} - a_1 && (2.19)\\
&= \frac{f(2)-f(0)}{2} - (f(1)-f(0)) && (2.20)\\
&= \frac{f(2)-2f(1)-f(0)}{2} && (2.21)\\
&= \frac{(f(2)-f(1))-(f(1)-f(0))}{2} =: \frac{\Delta^2(f)(0)}{2} && (2.22)\\
&\ \vdots \\
a_r &= \frac{\Delta^r(f)(0)}{r!}, && (2.23)
\end{aligned}
$$

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

$$\Delta(f)(n) := f(n+1) - f(n). \tag{2.24}$$

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

$$
\begin{aligned}
(x)_0 &:= 1 && (2.25)\\
(x)_n &:= x(x-1)\cdots(x-n+1) && (2.26)\\
&= \prod_{i=0}^{n-1}(x-i). && (2.27)
\end{aligned}
$$

Under these settings, we have

$$\begin{aligned} f(z) &= f_R(z) && (2.28) \\ &= \sum_{r=0}^{R} \frac{\Delta^r(f)(0)}{r!}(x)_r, && (2.29) \end{aligned}$$

where we have assume

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

Example

Consider a polynomial

$$f(z) := 2 * z^3 + 3 * z, \tag{2.31}$$

and its out put list

$$[f(0), f(1), f(3), \cdots] = [0, 5, 22, 63, 140, 265, \cdots] \tag{2.32}$$

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

$$\begin{aligned} f(0) &= 0 && (2.33) \\ \Delta(f)(0) &= f(1) - f(0) = 5 && (2.34) \\ \Delta^2(f)(0) &= \Delta(f)(1) - \Delta(f)(0) \\ &= f(2) - f(1) - 5 = 22 - 5 - 5 = 12 && (2.35) \\ \Delta^3(f)(0) &= \Delta^2(f)(1) - \Delta^2(f)(0) \\ &= f(3) - f(2) - \{f(2) - f(1)\} - 12 = 12 && (2.36) \end{aligned}$$

so we get

$$[0, 5, 12, 12] \tag{2.37}$$

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

$$\begin{aligned} f(z) &= 5(x)_1 + \frac{12}{2}(x)_2 + \frac{12}{3!}(x)_3 && (2.38) \\ &= 5(x)_1 + 6(x)_2 + 2(x)_3. && (2.39) \end{aligned}$$

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

$$\texttt{xs} = \left[f(0), \Delta(f)(0), \Delta^2(f)(0), \cdots\right]. \tag{2.40}$$

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

$$\texttt{f x = 2*x^3+3*x} \tag{2.41}$$

Let us try to reconstruct this polynomial from output list. In order to get the list `[x_0, x_1 ..]`, 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.,

$$\left[f(0), \Delta(f)(0), \Delta^2(f)(0), \cdots\right] \tag{2.42}$$

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

$$\left[\begin{array}{c} n \\ k \end{array}\right] \tag{2.43}$$

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

$$(x)_n = \sum_{k=1}^{n} (-)^{n-k} \left[\begin{array}{c} n \\ k \end{array}\right] x^k, \tag{2.44}$$

and

$$\left[\begin{array}{c} 0 \\ 0 \end{array}\right] := 1. \tag{2.45}$$

From the highest order, x^n, we get

$$\left[\begin{array}{c} n \\ n \end{array}\right] = 1, \forall n > 0. \tag{2.46}$$

We also put

$$\left[\begin{array}{c} 0 \\ 1 \end{array}\right] = \left[\begin{array}{c} 0 \\ 2 \end{array}\right] = \left[\begin{array}{c} 0 \\ 3 \end{array}\right] = \cdots = 0, \tag{2.47}$$

and

$$\left[\begin{array}{c} 1 \\ 0 \end{array}\right] = \left[\begin{array}{c} 2 \\ 0 \end{array}\right] = \left[\begin{array}{c} 3 \\ 0 \end{array}\right] = \cdots = 0. \tag{2.48}$$

The key equation is

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

and we get

$$\begin{aligned}
(x)_n &= \sum_{k=1}^{n} (-)^{n-k} \left[\begin{array}{c} n \\ k \end{array}\right] x^k & (2.50)\\
&= x^n + \sum_{k=1}^{n-1} (-)^{n-k} \left[\begin{array}{c} n \\ k \end{array}\right] x^k & (2.51)\\
(x)_{n-1} * (x - n + 1) &= \sum_{k=1}^{n-1} (-)^{n-1-k} \left\{ \left[\begin{array}{c} n-1 \\ k \end{array}\right] x^{k+1} - (n-1) \left[\begin{array}{c} n-1 \\ k \end{array}\right] x^k \right\} & (2.52)\\
&= \sum_{l=2}^{n} (-)^{n-l} \left[\begin{array}{c} n-1 \\ l-1 \end{array}\right] x^l + (n-1) \sum_{k=1}^{n-1} (-)^{n-k} \left[\begin{array}{c} n \\ k \end{array}\right] x^k & (2.53)\\
&= x^n + (n-1)(-)^{n-1} x \\
&\quad + \sum_{k=2}^{n-1} (-)^{n-k} \left\{ \left[\begin{array}{c} n-1 \\ k-1 \end{array}\right] + (n-1) \left[\begin{array}{c} n-1 \\ k \end{array}\right] \right\} x^k & (2.54)\\
&= x^n + \sum_{k=1}^{n-1} (-)^{n-k} \left\{ \left[\begin{array}{c} n-1 \\ k-1 \end{array}\right] + (n-1) \left[\begin{array}{c} n-1 \\ k \end{array}\right] \right\} x^k & (2.55)
\end{aligned}$$

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

$$\left[\begin{array}{c} n \\ k \end{array}\right] = \left[\begin{array}{c} n-1 \\ k-1 \end{array}\right] + (n-1) \left[\begin{array}{c} n-1 \\ k \end{array}\right] \tag{2.56}$$

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

$$
\begin{aligned}
f(z) &= f_R(z) && (2.57)\\
&= \sum_{r=0}^{R} \frac{\Delta^r(f)(0)}{r!}(x)_r && (2.58)\\
&= \sum_{r=0}^{R} \frac{\Delta^r(f)(0)}{r!} \sum_{k=1}^{r} (-)^{r-k} \left[\begin{array}{c} r \\ k \end{array} \right] x^k, && (2.59)
\end{aligned}
$$

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::Int -> Ratio Int`:

```
map f [0..]  == [f 0, f 1 ..]
```
(2.60)

2.3.1 Thiele's interpolation formula

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

$$\begin{aligned}
f_0(z) &= a_0 && (2.61)\\
f_1(z) &= a_0 + \frac{z}{a_1} && (2.62)\\
&\vdots \\
f_r(z) &= a_0 + \cfrac{z}{a_1 + \cfrac{z-1}{a_2 + \cfrac{z-2}{\ddots\; a_{r-2} + \cfrac{\vdots}{a_{r-1} + \cfrac{z-r+1}{a_r}}}}}, && (2.63)
\end{aligned}$$

where

$$\begin{aligned}
a_0 &= f(0) && (2.64)\\
a_1 &= \frac{1}{f(1)-a_0} && (2.65)\\
a_2 &= \cfrac{1}{\cfrac{2}{f(2)-a_0} - a_1} && (2.66)\\
&\vdots \\
a_r &= \cfrac{1}{\cfrac{2}{\cfrac{3}{\cfrac{\vdots}{\cfrac{r}{f(r)-a_0} - a_1} - a_2} - a_{r-2}} - a_{r-1}} && (2.67)\\
&= \left(\left((f(r)-a_0)^{-1}\, r - a_1\right)^{-1}(r-1) - \cdots - a_{r-1}\right)^{-1} 1 && (2.68)
\end{aligned}$$

2.3.2 Towards canonical representations

In order to get a unique representation of canonical form

$$\frac{\sum_\alpha n_\alpha z^\alpha}{\sum_\beta d_\beta z^\beta} \tag{2.69}$$

we put

$$d_{\min r'} = 1 \tag{2.70}$$

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:

$$d_0 = 1 \tag{2.71}$$

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

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 proof[2], that the Thiele coefficients are given by

$$a_0 := f(0) \tag{2.73}$$

$$a_n := \rho_{n,0} - \rho_{n-2,0}, \tag{2.74}$$

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

where ρ is so called the reciprocal difference:

$$\rho_{n,i} := 0, n < 0 \tag{2.75}$$

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

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

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

$$a_0 = 1, a_1 = 2, a_2 = 3, a_3 = 4. \tag{2.78}$$

The function is now

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

$$= \frac{x^2 + 16x + 16}{16 + 6x} \tag{2.80}$$

Using Maxima[3], 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.

```
> 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,

$$\texttt{(ns,ds) :: ([Ratio Int],[Ratio Int])} \tag{2.81}$$

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 1[4].

```
> 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

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

[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:

$$
\begin{aligned}
f(x) &= 1+\cfrac{x}{2+\cfrac{x-1}{\cfrac{4*3+x-2}{4}}} && (2.83)\\
&= 1+\cfrac{x}{2+\cfrac{x-1}{\cfrac{x+10}{4}}} && (2.84)\\
&= 1+\cfrac{x}{\cfrac{2*(x+10)+4*(x-1)}{x+10}} && (2.85)\\
&= 1+\cfrac{x}{\cfrac{6x+16}{x+10}} && (2.86)\\
&= \frac{1*(6x+16)+x*(x+10)}{6x+16} && (2.87)\\
&= \frac{x^2+16x+16}{6x+16} && (2.88)
\end{aligned}
$$

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

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

In general, we have the following Thiele representation:

$$a_0+\cfrac{z}{a_1+\cfrac{z-1}{a_2+\cfrac{z-2}{\cfrac{\vdots}{a_n+\cfrac{z-n}{a_{n+1}}}}}} \tag{2.90}$$

The base case should be

$$a_n+\frac{z-n}{a_{n+1}} = \frac{a_{n+1}*a_n-n+z}{a_{n+1}} \tag{2.91}$$

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

$$\begin{aligned} a_r(z) &= a_r + \frac{z-r}{a_{r+1}(z)} && (2.92) \\ &= \frac{a_r a_{r+1}(z) + z - r}{a_{r+1}(z)} && (2.93) \\ &= \frac{a_r * \mathtt{num}\,(a_{r+1}(z)) + \mathtt{den}\,(a_{r+1}(z)) * (z-r)}{\mathtt{num}\,(a_{r+1}(z))} && (2.94) \end{aligned}$$

where

$$a_{r+1}(z) = \frac{\mathtt{num}\,(a_{r+1}(z))}{\mathtt{den}\,(a_{r+1}(z))} \tag{2.95}$$

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

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

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

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

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

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

The terminate cotndition 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.

$$f(z_1, z_2) = 3 + 2z_1 + 4z_2 + 7z_1^2 + 5z_1 z_2 + 6z_2^2 \tag{2.99}$$

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

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

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

$$f(z,0) = 3+2z+7z^2 \tag{2.101}$$
$$f(z,1) = 13+7z+7z^2 \tag{2.102}$$
$$f(z,2) = 35+12z+7z^2. \tag{2.103}$$

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

$$c_2(z) = 7 \tag{2.104}$$

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.,

$$c_1(z) = 2+5z. \tag{2.105}$$

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

$$c_0(z) = 3+4z+6z^2 \tag{2.106}$$

and these fully determine our polynomial:

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

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

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

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

$$c_2(x) = 9 \tag{2.109}$$

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

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

and

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

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]
]
```

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

So, this "table" is like

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

Then we can apply the univariate technique.

```
*Multivariate> let fTable = tablize 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:

```
> wellOrd :: [[a]] -> [[a]]
> wellOrd xss
>   | null (head xss) = []
>   | otherwise       = map head xss : wellOrd (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 = tablize 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> wellOrd 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]
]
*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 . wellOrd . map list2pol

  *Multivariate> let g x y = 1+7*x + 8*y + 10*x^2 + x*y+9*y^2
  *Multivariate> table2pol $ tablize 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

$$\frac{\sum_\alpha n_\alpha z^\alpha}{\sum_\beta d_\beta z^\beta} \tag{2.113}$$

A canonical choice is

$$d_0 = d_{(0,\cdots,0)} = 1. \tag{2.114}$$

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

$$d'_0 = d_s \neq 0. \tag{2.115}$$

2.6.2 An auxiliary t

Introducing an auxiliary variable t, let us define

$$h(z,t) := f(tz_1, \cdots, tz_n), \tag{2.116}$$

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

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

where

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

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

are homogeneous polynomials.

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

A simplification

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

$$p_r(1, z_2, \cdots, z_n) \tag{2.120}$$

instead of $p_r(z_1, z_2, \cdots, z_n)$, 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 them[6].

```
*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

$$h(1,0,t) = \frac{3+2t+7t^2}{1+7t+10t^2} \tag{2.121}$$
$$h(1,1,t) = \frac{3+6t+18t^2}{1+15t+20t^2} \tag{2.122}$$
$$h(1,2,t) = \frac{3+10t+41t^2}{1+23t+48t^2} \tag{2.123}$$
$$h(1,3,t) = \frac{3+14t+76t^2}{1+31t+94t^2} \tag{2.124}$$

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])
```

[6] Eq.(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

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

Finally, we use the homogeneous property for each powers:

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

Putting $t = 1$, we get

$$f(x,y) = h(x,y,1) \tag{2.127}$$

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

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 homogenious 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 % 1,3 % 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 %
```

Chapter 3

Functional reconstruction over finite fields

3.1 Univariate polynomials

3.1.1 Special data types

We introduce few new data types.

```
> type Q = Ratio Int   -- Rational fields
> type Graph = [(Q,Q)] -- [(x, f x) | x <- someFinieRange]

> -- using record syntax
> data PDiff
>   = PDiff { points    :: (Int, Int) -- end points
>           , value     :: Int        -- Zp value
>           , basePrime :: Int
>           }
>   deriving (Show, Read)
```

To apply difference analysis, we introduce a higher order function.

```
> -- f [a,b,c ..] -> [(f a b), (f b c) ..]
> -- pair wise application
> map' :: (a -> a -> b) -> [a] -> [b]
> map' f as = zipWith f as (tail as)
```

3.1.2 Implementations

```
> -- To select Z_p valid inputs.
```

```
> sample :: Int   -- prime
>        -> Graph -- increasing input
>        -> Graph
> sample p = filter ((< (fromIntegral p)) . fst)
>
> -- To eliminate (1%p) type "fake" infinity.
> -- After eliminating these, we can freely use 'modp', primed version.
> check :: Int   -- prime
>       -> Graph
>       -> Graph -- safe data sets
> check p = filter (not . isDanger p)
>   where
>     isDanger -- To detect (1%p) type infinity.
>       :: Int -- prime
>       -> (Q,Q) -> Bool
>     isDanger p (_, fx) = (d `rem` p) == 0
>       where
>         d = denominator fx
>
> project :: Int -> (Q,Q) -> (Int, Int)
> project p (x, fx) -- for simplicity
>   | denominator x == 1 = (numerator x, fx `modp'` p)
>   | otherwise          = error "project: integer input?"
>
> -- From Graph to Zp (safe) values.
> onZp
>   :: Int                -- base prime
>   -> Graph
>   -> [(Int, Int)] -- in-out on Zp value
> onZp p = map (project p) . check p . sample p
>
> toPDiff
>   :: Int        -- prime
>   -> (Int, Int) -- in and out mod p
>   -> PDiff
> toPDiff p (x,fx) = PDiff (x,x) fx p
>
```

```
> newtonTriangleZp :: [PDiff] -> [[PDiff]]
```

```
> newtonTriangleZp fs
>   | length fs < 3 = []
>   | otherwise     = helper [sf3] (drop 3 fs)
>   where
>     sf3 = reverse . take 3 $ fs -- [[f2,f1,f0]]
>     helper fss [] = error "newtonTriangleZp: need more evaluation"
>     helper fss (f:fs)
>       | isConsts 3 . last $ fss = fss
>       | otherwise               = helper (add1 f fss) fs
>
> isConsts
>   :: Int -- 3times match
>   -> [PDiff] -> Bool
> isConsts n ds
>   | length ds < n = False
> -- isConsts n ds     = all (==l) $ take (n-1) ls
>   | otherwise     = all (==l) $ take (n-1) ls
>   where
>     (l:ls) = map value ds
>
> -- backward, each [PDiff] is decreasing inputs (i.e., reversed)
> add1 :: PDiff -> [[PDiff]] -> [[PDiff]]
> add1 f [gs] = fgs : [zipWith bdiffStep fgs gs] -- singleton
>   where
>     fgs = f:gs
> add1 f (gg@(g:gs) : hhs) -- gg is reversed order
>             = (f:gg) : add1 fg hhs
>   where
>     fg = bdiffStep f g
>
> -- backward
> bdiffStep :: PDiff -> PDiff -> PDiff
> bdiffStep (PDiff (y,y') g q) (PDiff (x,x') f p)
>   | p == q    = PDiff (x,y') finiteDiff p
>   | otherwise = error "bdiffStep: different primes?"
>   where
>     finiteDiff = ((fg % xy') `modp'` p)
>     xy' = (x - y' `mod` p)
>     fg = ((f-g) `mod` p)
>
```

```
> graph2Zp :: Int -> Graph -> [(Int, Int)]
> graph2Zp p = onZp p . check p . sample p
>
> graph2PDiff :: Int -> Graph -> [PDiff]
> graph2PDiff p = map (toPDiff p) . graph2Zp p
>
> newtonTriangleZp' :: Int -> Graph -> [[PDiff]]
> newtonTriangleZp' p = newtonTriangleZp . graph2PDiff p
>
> newtonCoeffZp :: Int -> Graph -> [PDiff]
> newtonCoeffZp p = map head . newtonTriangleZp' p

  *GUniFin> let gs = map (\x -> (x,x^2 + (1%2)*x + 1%3))
                         [1,2,4,5,9,10,11] :: Graph
  *GUniFin> newtonCoeffZp 101 gs
  [PDiff {points = (9,9), value = 69, basePrime = 101}
  ,PDiff {points = (5,9), value = 65, basePrime = 101}
  ,PDiff {points = (4,9), value = 1, basePrime = 101}
  ]
  *GUniFin> map (\x -> (Just . value $ x, basePrime x)) it
  [(Just 69,101),(Just 65,101),(Just 1,101)]

We take formally the canonical form on Zp,
then apply rational "number" reconstruction.

> n2cZp :: [PDiff] -> ([Int], Int)
> n2cZp graph = (helper graph, p)
>   where
>     p = basePrime . head $ graph
>     helper [d]    = [value d]
>     helper (d:ds) = map (`mod` p) $ ([value d] + (z * next))
>                                   - (map (`mod` p) (zd .* next))
>       where
>         zd = fst . points $ d
>         next = helper ds
>
> format :: ([Int],Int) -> [(Maybe Int, Int)]
> format (as,p) = [(return a,p) | a <- as]

  *GUniFin> let gs = map (\x -> (x,x^2 + (1%2)*x + 1%3))
```

```
                        [0,2,3,5,7,8,11] :: Graph
  *GUniFin> newtonCoeffZp 10007 gs
  [PDiff {points = (7,7), value = 8392, basePrime = 10007}
  ,PDiff {points = (5,7), value = 5016, basePrime = 10007}
  ,PDiff {points = (3,7), value = 1, basePrime = 10007}
  ]
  *GUniFin> n2cZp it
  ([3336,5004,1],10007)
  *GUniFin> format it
  [(Just 3336,10007),(Just 5004,10007),(Just 1,10007)]
  *GUniFin> map guess it
  [Just (1 % 3,10007),Just (1 % 2,10007),Just (1 % 1,10007)]

  *GUniFin> let gs = map (\x -> (x,x^2 + (1%2)*x + 1%3))
                        [0,2,3,5,7,8,11] :: Graph
  *GUniFin> map guess . format . n2cZp . newtonCoeffZp 10007 $ gs
  [Just (1 % 3,10007),Just (1 % 2,10007),Just (1 % 1,10007)]
  *GUniFin> let gs = map (\x -> (x,x^5 + x^2 + (1%2)*x + 1%3))
                        [0,2,3,5,7,8,11,13,17,18,19,21,24,28,31,33,34]
                        :: Graph
  *GUniFin> map guess . format . n2cZp . newtonCoeffZp 10007 $ gs
  [Just (1 % 3,10007),Just (1 % 2,10007),Just (1 % 1,10007)
  ,Just (0 % 1,10007),Just (0 % 1,10007),Just (1 % 1,10007)
  ]

> preTrial gs p = format . n2cZp . newtonCoeffZp p $ gs

  *GUniFin> let gs = map (\x -> (x,x^5 + x^2 + (1%2)*x + 1%3))
                        [0,2,3,5,7,8,11,13,17,18,19,21,24,28,31,33,34]
                        :: Graph
  *GUniFin> map reconstruct . transpose . map (preTrial gs) $ bigPrimes
  [Just (1 % 3),Just (1 % 2),Just (1 % 1)
  ,Just (0 % 1),Just (0 % 1),Just (1 % 1)
  ]
```

Here is "a" final version, the univariate polynomial reconstruction
with finite fields.

```
> uniPolCoeff :: Graph -> Maybe [(Ratio Int)]
> uniPolCoeff gs
```

```
>   = (mapM reconstruct' . transpose . map (preTrial gs)) bigPrimes
```

```
*GUniFin> let gs = map (\x -> (x,x^5 + x^2 + (1%2)*x + 1%3))
                        [0,2,3,5,7,8,11,13,17,18,19,21,24,28,31,33,34]
                        :: Graph
*GUniFin> gs
[(0 % 1,1 % 3),(2 % 1,112 % 3),(3 % 1,1523 % 6),(5 % 1,18917 % 6)
,(7 % 1,101159 % 6),(8 % 1,98509 % 3),(11 % 1,967067 % 6)
,(13 % 1,2228813 % 6),(17 % 1,8520929 % 6),(18 % 1,5669704 % 3)
,(19 % 1,14858819 % 6),(21 % 1,24507317 % 6),(24 % 1,23889637 % 3)
,(28 % 1,51633499 % 3),(31 % 1,171780767 % 6),(33 % 1,234818993 % 6)
,(34 % 1,136309792 % 3)
]
*GUniFin> uniPolCoeff gs
Just [1 % 3,1 % 2,1 % 1,0 % 1,0 % 1,1 % 1]

*GUniFin> let fs = map (\x -> (x,(3+x+(1%3)*x^9)/(1)))
                        [1,3..101] :: Graph
*GUniFin> uniPolCoeff fs
Just [3 % 1,1 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,1 % 3]
*GUniFin> let fs = map (\x -> (x,(3+x+(1%3)*x^10)/(1)))
                        [1,3..101] :: Graph
*GUniFin> uniPolCoeff fs
*** Exception: newtonBT: need more evaluation
CallStack (from HasCallStack):
  error, called at GUniFin.lhs:79:23 in main:GUniFin
*GUniFin> let fs = map (\x -> (x,(3+x+(1%3)*x^10)/(1)))
                        [1,3..1001] :: Graph
*GUniFin> uniPolCoeff fs
*** Exception: newtonBT: need more evaluation
CallStack (from HasCallStack):
  error, called at GUniFin.lhs:79:23 in main:GUniFin
```

```
Rough estimation says, in 64-bits system with sequential inputs,
the upper limit of degree is about 15.
If we use non sequential inputs, this upper limit will go down.
```

3.2 Univariate rational functions

3.3 TBA Univariate rational functions

Chapter 4

Codes

4.1 Ffield.lhs

Listing 4.1: Ffield.lhs

```
Ffield.lhs

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

> module Ffield where

> import Data.Ratio
> import Data.Maybe
> import Data.Numbers.Primes
> import Test.QuickCheck

> -- Eucledian 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

Consider a finite ring
  Z_n := [0..(n-1)]
of some Int number.
If any non-zero element has its multiplication inverse,
then the ring is a field:

```

```
> -- Our target should be in Int.
> isField
>   :: Int -> Bool
> isField = isPrime

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
  http://qiita.com/bra_cat_ket/items/205c19611e21f3d422b7

> exGCD'
>   :: (Integral n) =>
>      n -> n -> ([n], [n], [n], [n])
> exGCD' a b = (qs, rs, ss, ts)
>   where
>     qs = zipWith quot rs (tail rs)
>     rs = takeBefore (==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)
>
> takeBefore
>   :: (a -> Bool) -> [a] -> [a]
> takeBefore p = foldr func []
>   where
>     func x xs
>       | p x       = []
>       | otherwise = x : xs
>
> -- Bezout's identity 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

> -- We use built-in function gcd.
> coprime
>   :: Integral a =>
>      a -> a -> Bool
> coprime a b = gcd a b == 1

> -- a^{-1} (in Z_p) == a `inversep` p
> inversep
>   :: Integral a =>
>      a -> a -> Maybe a -- We also use in CRT.
> a `inversep` p = let (g,x,_) = exGCD a p in
>   if (g == 1)
>     then Just (x `mod` p) -- g==1 <=> coprime a p
>     else Nothing
>
> -- If a is "safe" value, we can use this.
> inversep'
>   :: Int -> Int -> Int
> 0 `inversep'` _ = error "inversep':␣zero␣division"
> a `inversep'` p = (x `mod` p)
>   where
>     (_,x,_) = exGCD a p
>
> -- Returns a list of inveres of given ring Z_p.
> inversesp
>   :: Int -> [Maybe Int]
> inversesp p = map (`inversep` p) [1..(p-1)]
>
> -- A map from Q to Z_p, where p is a prime.
> modp
>   :: Ratio Int -> Int -> Maybe Int
> q `modp` p
>   | coprime b p = Just $ (a * (bi `mod` p)) `mod` p
>   | otherwise   = Nothing
>   where
>     (a,b)   = (numerator q, denominator q)
>     Just bi = b `inversep` p
>
> -- When the denominator of q is not proprtional to p, use this.
> modp'
>   :: Ratio Int -> Int -> Int
```

```
> q `modp'` p = (a * (bi `mod` p)) `mod` p
>   where
>     (a,b)   = (numerator q, denominator q)
>     bi = b `inversep'` p
>
> -- This is guess function without Chinese Reminder Theorem.
> guess
>   :: Integral t =>
>      (Maybe t, t)       -- (q `modp` p, p)
>   -> Maybe (Ratio t, t)
> guess (Nothing, _) = Nothing
> guess (Just a, p) = let (_,rs,ss,_) = exGCD' a p in
>   Just (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*r <= p && s*s <= p = r%s
>         | otherwise                      = select rs ss p
>
> -- Hard code of big primes
> -- We have chosen a finite number (100) version.
> bigPrimes :: [Int]
> bigPrimes = take 100 $ dropWhile (<10^4) primes
> -- bigPrimes = take 100 $ dropWhile (< 10^6) primes

*Ffield> bigPrimes
[10007,10009,10037,10039,10061,10067,10069,10079,10091,10093,10099,10103

,10111,10133,10139,10141,10151,10159,10163,10169,10177,10181,10193,10211

,10223,10243,10247,10253,10259,10267,10271,10273,10289,10301,10303,10313

,10321,10331,10333,10337,10343,10357,10369,10391,10399,10427,10429,10433

,10453,10457,10459,10463,10477,10487,10499,10501,10513,10529,10531,10559

,10567,10589,10597,10601,10607,10613,10627,10631,10639,10651,10657,10663

,10667,10687,10691,10709,10711,10723,10729,10733,10739,10753,10771,10781
```

```
  ,10789,10799,10831,10837,10847,10853,10859,10861,10867,10883,10889,10891
  ,10903,10909,10937,10939
  ]

  *Ffield> let knownData q = zip (map (modp q) bigPrimes) bigPrimes
  *Ffield> let ds = knownData (12%13)
  *Ffield> map guess ds
  [Just (12 % 13,10007)
  ,Just (12 % 13,10009)
  ,Just (12 % 13,10037)
  ,Just (12 % 13,10039) ..

  *Ffield> let ds = knownData (112%113)
  *Ffield> map guess ds
  [Just ((-39) % 50,10007)
  ,Just ((-41) % 48,10009)
  ,Just ((-69) % 20,10037)
  ,Just ((-71) % 18,10039) ..

--

Chinese Remainder Theorem, and its usage

> imagesAndPrimes
>   :: Ratio Int -> [(Maybe Int, Int)]
> imagesAndPrimes q = zip (map (modp q) bigPrimes) bigPrimes

  *Ffield> let q = 895%922
  *Ffield> let knownData = imagesAndPrimes q
  *Ffield> let [(a1,p1),(a2,p2)] = take 2 knownData
  *Ffield> take 2 knownData
  [(Just 6003,10007),(Just 9782,10009)]
  *Ffield> map guess it
  [Just ((-6) % 5,10007),Just (21 % 44,10009)]

Our data is a list of the type
  [(Maybe Int, Int)]
In order to use CRT, we should cast its type.

> toInteger2
>   :: [(Maybe Int, Int)] -> [(Maybe Integer, Integer)]
> toInteger2 = map helper
```

```
>   where
>     helper (x,y) = (fmap toInteger x, toInteger y)
>
> crtRec'
>   :: Integral a =>
>      (Maybe a, a) -> (Maybe a, a) -> (Maybe a, a)
> crtRec' (Nothing,p) (_,q)       = (Nothing, p*q)
> crtRec' (_,p)       (Nothing,q) = (Nothing, p*q)
> crtRec' (Just a1,p1) (Just a2,p2) = (Just a,p)
>   where
>     a = (a1*p2*m2 + a2*p1*m1) `mod` p
>     Just m1 = p1 `inversep` p2
>     Just m2 = p2 `inversep` p1
>     p = p1*p2
>
> matches3
>   :: Eq a =>
>      [Maybe (a,b)] -> Maybe (a,b)
> matches3 (b1@(Just (q1,p1)):bb@((Just (q2,_)):(Just (q3
   ,_)):_))
>   | q1==q2 && q2==q3 = b1
>   | otherwise        = matches3 bb
> matches3 _ = Nothing

  *Ffield> let ds = imagesAndPrimes (1123%1135)
  *Ffield> map guess ds
  [Just (25 % 52,10007)
  ,Just ((-81) % 34,10009)
  ,Just ((-88) % 63,10037) ..

  *Ffield> matches3 it
  Nothing

  *Ffield> scanl1 crtRec' . toInteger2 $ ds
  [(Just 3272,10007)
  ,(Just 14913702,100160063)
  ,(Just 298491901442,1005306552331) ..

  *Ffield> map guess it
  [Just (25 % 52,10007)
  ,Just (1123 % 1135,100160063)
  ,Just (1123 % 1135,1005306552331)
  ,Just (1123 % 1135,10092272478850909) ..

  *Ffield> matches3 it
```

```
  Just (1123 % 1135,100160063)

We should determine the number of matches to cover the range of machine size
Integer, i.e., Int of Haskell.

  *Ffield> let mI = maxBound :: Int
  *Ffield> mI == 2^63-1
  True
  *Ffield> logBase 10 (fromIntegral mI)
  18.964889726830812

Since our choice of bigPrimes are
  O(10^4)
5 times is enough to cover the machine size integers.

> reconstruct
>   :: [(Maybe Int, Int)] -> Maybe (Ratio Integer)
> -- reconstruct = matches 10 . makeList -- 10 times match
> reconstruct = matches 5 . makeList -- 5 times match
>   where
>     matches n (a:as)
>       | all (a==) $ take (n-1) as = a
>       | otherwise                 = matches n as
>
>     makeList = map (fmap fst . guess) . scanl1 crtRec' . toInteger2
>                . filter (isJust . fst)
>
> -- cast version
> reconstruct'
>   :: [(Maybe Int, Int)] -> Maybe (Ratio Int)
> reconstruct' = fmap coersion . reconstruct
>   where
>     coersion :: Ratio Integer -> Ratio Int
>     coersion q = (fromInteger . numerator $ q)
>                  % (fromInteger . denominator $ q)

  *Ffield> let q = 895%922
  *Ffield> let knownData = imagesAndPrimes q
  *Ffield> reconstruct knownData
  Just (895 % 922)

-- QuickCheck
```

```

  *Ffield> let q = 513197683989569 % 1047805145658 ::
     Ratio Int
  *Ffield> let ds = imagesAndPrimes q
  *Ffield> let answer = fmap fromRational . reconstruct $
      ds
  *Ffield> answer :: Maybe (Ratio Int)
  Just (513197683989569 % 1047805145658)

> prop_rec :: Ratio Int -> Bool
> prop_rec q = Just q == answer
>   where
>     answer :: Maybe (Ratio Int)
>     answer = fmap fromRational . reconstruct $ ds
>     ds = imagesAndPrimes q

  *Ffield> quickCheckWith stdArgs { maxSuccess = 100000 }
      prop_rec
  +++ OK, passed 100000 tests.
```

4.2 Polynomials.hs

Listing 4.2: Polynomials.hs

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

module Polynomials where

default (Integer, Rational, Double)

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

z :: Num a => [a]
z = [0,1]

-- 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
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)
```

4.3 Univariate.lhs

Listing 4.3: Univariate.lhs

```
Univariate.lhs

> module Univariate where

References
J. Stoer, R. Bulirsch
  Introduction to Numerical Analysis (2nd edition)
L. M. Milne-Thomson
  THE CALCULUS OF FINITE DIFFERENCES

> import Control.Applicative
> import Control.Monad
> import Data.Ratio
> import Data.Maybe
> import Data.List
> -- import Control.Monad.Catch
>
> import Polynomials

From the output list
  map f [0..]
of a polynomial
  f :: Int -> Ratio Int
we reconstrunct the canonical form of f.

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

```
> -- This degree function is "strict", so only take finite list.
> degree' :: (Eq a, Num a) => [a] -> Int
> degree' xs = length (difLists [xs]) - 1
>
> -- This degree 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)
>
> -- This is a hyblid version, safe and lazy.
> degree :: (Num a, Eq a) => [a] -> Int
> degree xs = let l = degreeLazy xs in
>   degree' $ take (l+2) xs

> -- m-times match version
> degreeTimes :: (Num a, Eq a) => Int -> [a] -> Int
> degreeTimes m xs = helper xs 0
>   where
>     helper aa@(a:as) n
>       | all (== a) (take (m-1) as) = n
>       | otherwise                  = helper (difs aa) (n+1)

Newton interpolation formula
First we introduce a new infix symbol for the operation
of taking a falling power.

> infixr 8 ^- -- falling power
> (^-) :: (Eq a, Num a) => a -> a -> a
> x ^- 0 = 1
> x ^- n = (x ^- (n-1)) * (x - n + 1)

Claim (Newton interpolation formula):
  A polynomial f of degree n is expressed as
    f(z) = \sum_{k=0}^n  (diff^n(f)(0)/k!) * (x ^- k)
  where diff^n(f) is the n-th difference of f.

Example
Consider a polynomial f(x) = 2*x^3+3*x.

```

```
In general, we have no prior knowledge of this form,
but we know the sequences as a list of outputs (map f [0..]):

  Univariate> let f x = 2*x^3+3*x
  Univariate> take 10 $ map f [0..]
  [0,5,22,63,140,265,450,707,1048,1485]
  Univariate> degree $ take 10 $ map f [0..]
  3

Let us try to get differences:

  Univariate> difs $ take 10 $ map f [0..]
  [5,17,41,77,125,185,257,341,437]
  Univariate> difs it
  [12,24,36,48,60,72,84,96]
  Univariate> difs it
  [12,12,12,12,12,12,12]

Or more simply take difLists:

  Univariate> difLists [take 10 $ map f [0..]]
  [[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]
  ]

What we need is the heads of above lists.

  Univariate> map head it
  [12,12,5,0]

Newton interpolation formula gives
  f' x = 0*(x ^- 0) `div` (0!) + 5*(x ^- 1) `div` (1!)
           + 12*(x ^- 2) `div` (2!) + 12*(x ^- 3) `div` (3!)
         = 5*(x ^- 1) + 6*(x ^- 2) + 2*(x ^- 3)
So

  Univariate> let f x = 2*x^3+3*x
  Univariate> let f' x = 5*(x ^- 1) + 6*(x ^- 2) + 2*(x ^- 3)
  Univariate> take 10 $ map f [0..]
  [0,5,22,63,140,265,450,707,1048,1485]
```

```
  Univariate> take 10 $ map f' [0..]
  [0,5,22,63,140,265,450,707,1048,1485]
```

Assume the differences are given in a list

```
  [x_0, x_1 ..]
```

where x_k = diff^k(f)(0).
Then the implementation of the Newton interpolation formula is as follows:

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

  Univariate> let f x = 2*x^3+3*x
  Univariate> take 10 $ map f [0..]
  [0,5,22,63,140,265,450,707,1048,1485]
  Univariate> 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]
  ]
  Univariate> reverse $ map head it
  [0,5,12,12]
  Univariate> newtonC it
  [0 % 1,5 % 1,6 % 1,2 % 1]
```

The list of first differences can be computed as follows:

```
> firstDifs
>   :: (Eq a, Num a) =>
>       [a] -- map f [0..]
>   -> [a]
> firstDifs xs = reverse . map head . difLists $ [xs]
```

Mapping a list of integers to a Newton representation:

```
> -- This implementation can take infinite list.
> list2npol :: (Integral a) => [Ratio a] -> [Ratio a]
> list2npol xs = newtonC . firstDifs $ take n xs
>   where n = (degree xs) + 2
```

```
>
> -- m-times matches version
> list2npolTimes :: (Integral a) => Int -> [Ratio a] -> [Ratio a]
> list2npolTimes m xs = newtonC . firstDifs $ take n xs
>   where n = (degreeTimes m xs) + 2

  *Univariate> let f x = x*(x-1)*(x-2)*(x-3)*(x-4)*(x-5)
  *Univariate> let fs = map f [0..]
  *Univariate> list2npolTimes 10 fs
  [0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,1 % 1]
  *Univariate> npol2pol it
  [0 % 1,(-120) % 1,274 % 1,(-225) % 1,85 % 1,(-15) % 1,1 % 1]
  *Univariate> list2npol fs
  [0 % 1]

We need to map Newton falling powers to standard powers.
This is a matter of applying combinatorics, by means of a convention formula
that uses the so-called Stirling cyclic numbers (of the first kind.)
Its defining relation is
  (x ^- n) = \sum_{k=1}^n (stirlingC n k) * (-1)^(n-k) * x^k.
The key equation is
  (x ^- n) = (x ^- (n-1)) * (x-n+1)
           = x*(x ^- (n-1)) - (n-1)*(x ^- (n-1))

Therefore, an implementation is as follows:

> stirlingC :: (Integral a) => a -> a -> a
> stirlingC 0 0 = 1
> stirlingC 0 _ = 0
> stirlingC n k = stirlingC (n-1) (k-1) + (n-1) * stirlingC (n-1) k

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 this 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 :: (Ord t, Num t) => [t] -> [t]
> npol2pol xs = sum [ [x] * map fromInteger (fall2pol k)
>                   | (x,k) <- zip xs [0..]
>                   ]
```

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

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

Reconstruction as curve fitting

```
  *Univariate> let f x = 2*x^3 + 3*x + 1%5
  *Univariate> take 10 $ map f [0..]
  [1%5, 26%5, 111%5, 316%5, 701%5, 1326%5, 2251%5, 3536%5, 5241%5, 7426%5]
  *Univariate> list2npol it
  [1 % 5,5 % 1,6 % 1,2 % 1]
  *Univariate> list2npol $ map f [0..]
  [1 % 5,5 % 1,6 % 1,2 % 1]
  *Univariate> list2pol $ map (\n -> 1%3 + (3%5)*n + (5%7)*n^2) [0..]
  [1 % 3,3 % 5,5 % 7]
  *Univariate>  list2pol [0,1,5,14,30,55]
  [0 % 1,1 % 6,1 % 2,1 % 3]
  *Univariate> map (p2fct $ list2pol [0,1,5,14,30,55]) [0..6]
  [0 % 1,1 % 1,5 % 1,14 % 1,30 % 1,55 % 1,91 % 1]
```

Here is n-times match version:

```
> list2polTimes :: Integral a => Int -> [Ratio a] -> [Ratio a]
> list2polTimes n = npol2pol . list2npolTimes n
```

--

```

Thiele's interpolation formula
https://rosettacode.org/wiki/Thiele%27s_interpolation_formula#Haskell
http://mathworld.wolfram.com/ThielesInterpolationFormula.html

reciprocal difference
Using the same notation of
https://rosettacode.org/wiki/Thiele%27s_interpolation_formula#C

> rho :: (Integral a) =>
>          [Ratio a] -- A list of output of f :: a -> Ratio a
>       -> a -> Int   -- "matrix"
>       -> Maybe (Ratio a) -- Nothing means 1/0 type infinity
> rho fs 0 i = Just $ fs !! i
> rho fs n i
>    | n < 0          = Just 0
>    | num == Just 0 = Nothing -- "infinity"
>    | otherwise      = (+) <$> recipro <*> rho fs (n-2) (i+1)
>    where
>      recipro = ((%) . (* n) <$> den) <*> num -- (den*n)%num
> --             (%) <$> (*n) <$> den <*> num -- functor law
>      num  = numerator <$> next
>      den  = denominator <$> next
> --     next = (-) <$> rho fs (n-1) (i+1) <*> rho fs (n-1) i
>      next = x `seq` y `seq` (-) <$> x <*> y
>        where x = rho fs (n-1) (i+1)
>              y = rho fs (n-1) i

Note that (%) has the following type,
  (%) :: Integral a => a -> a -> Ratio a

  *Univariate> (%) <$> (*2) <$> Just 5 <*> Just 3
  Just (10 % 3)

The follwoing reciprocal differences match the table of
Milne-Thompson[1951] page 106:
```

```

  *Univariate> map (\p -> map (rho (map (\t -> 1%(1+t^2)) [0..]) p) [0..3]) [0..5]
  [[Just (1 % 1)   ,Just (1 % 2)    ,Just (1 % 5)    , Just (1 % 10)]
  ,[Just ((-2) % 1),Just ((-10) % 3),Just ((-10) % 1), Just ((-170) % 7)]
  ,[Just ((-1) % 1),Just ((-1) % 10),Just ((-1) % 25), Just ((-1) % 46)]
  ,[Just (0 % 1)   ,Just (40 % 1)   ,Just (140 % 1)  , Just (324 % 1)]
  ,[Just (0 % 1)   ,Just (0 % 1)    ,Just (0 % 1)    , Just (0 % 1)]
  ,[Nothing,Nothing,Nothing,Nothing]
  ]

> -- Thiele coefficients (continuous fraction)
> a :: (Integral a) => [Ratio a] -> a -> Maybe (Ratio a)
> a fs 0 = Just $ head fs
> a fs n = (-) <$> rho fs n 0 <*> rho fs (n-2) 0
>
> -- shifted Thiele coefficients
> a' :: Integral a => [Ratio a] -> Int -> a -> Maybe (Ratio a)
> a' fs p 0 = Just $ fs !! p
> a' fs p n = (-) <$> rho fs n p <*> rho fs (n-2) p

  *Univariate> map (\p -> map (rho (map (\t -> t%(1+t^2)) [0..]) p) [0..5]) [0..5]
  [[Just (0%1),Just (1%2)    ,Just (2%5)    ,Just (3%10)      ,Just (4%17)      ,Just (5%26)]
  ,[Just (2%1),Just ((-10)%1),Just ((-10)%1),Just ((-170)%11),Just ((-442)%19),Just ((-962)%29)]
  ,[Just (1%3),Nothing       ,Just ((-1)%15),Just ((-1)%48)  ,Just ((-1)%105) ,Just ((-1)%192)]
  ,[Nothing   ,Nothing       ,Just (50%1)   ,Just (242%1)     ,Just (662%1)    ,Just (1430%1)]
  ,[Nothing   ,Nothing       ,Just (0%1)    ,Just (0%1)       ,Just (0%1)      ,Just (0%1)]
  ,[Nothing   ,Nothing       ,Nothing       ,Nothing          ,Nothing         ,Nothing]
  ]

Here, the consecutive Just ((-10) % 1) in second list make "fake" infinity (Nothing).
```

```

  *Univariate> let f t = t%(1+t^2)
  *Univariate> let fs = map f [0..]
  *Univariate> let aMat = [map (a' fs i) [0..] | i <- [0..]]
  *Univariate> take 20 $ map (length . takeWhile isJust) $ aMat
  [3,2,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5]

> -- Thiele coefficients with shifts.
> aMatrix :: Integral a => [Ratio a] -> [[Maybe (Ratio a)]]
> aMatrix fs = [map (a' fs i) [0..] | i <- [0..]]
>
> tDegree :: Integral a => [Ratio a] -> Int
> tDegree = isConsts' 3 . map (length . takeWhile isJust) . aMatrix
>
> -- To find constant sub sequence.
> isConsts' :: Eq t => Int -> [t] -> t
> isConsts' n (l:ls)
>   | all (==l) $ take (n-1) ls = l
>   | otherwise                 = isConsts' n ls

we also need the shift, in this case, p=2 to get full Thiele coefficients.

> shiftaMatrix
>   :: Integral a =>
>      [Ratio a] -> [Maybe [Ratio a]]
> shiftaMatrix gs = map (sequence . (\q -> map (a' gs q) [0..(thieleD-1)])) [0..]
>   where
>     thieleD = fromIntegral $ tDegree gs
>
> shiftAndThieleC
>   :: Integral a =>
>      [Ratio a] -> (Maybe Int, Maybe [Ratio a])
> shiftAndThieleC fs = (findIndex isJust gs, join $ find isJust gs)
>   where
>     gs = shiftaMatrix fs

  *Univariate> take 10 $ map sequence $ transpose $ take (tDegree fs) m
```

```
  [Just [0 % 1,1 % 2,2 % 5,3 % 10,4 % 17]
  ,Just [2 % 1,(-10) % 1,(-10) % 1,(-170) % 11,(-442) % 19]
  ,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing]
```

Packed version, this scans the given data only once.

```
> degSftTC
>   :: Integral a =>
>      [Ratio a] -> (Int, Maybe Int, Maybe (Maybe [Ratio a]))
> --                 |     |          + Thiele coefficients
> --                 |     + shift
> --                 + degree
> degSftTC fs = (d,s,ts)
>   where
>     m = [map (a' fs i) [0..] | i <- [0..]]
>     d = isConsts' 3 . map (length . takeWhile isJust) $ m -- 3 times match
>     m' = map (sequence . take d) m
>     s = findIndex isJust m'
>     ts = find isJust m'
```

```
  *Univariate Control.Monad> let g t = t%(1+t^2)
  *Univariate Control.Monad> let gs = map g [0..]
  *Univariate Control.Monad> shiftAndThieleC $ shiftaMatrix gs
  (Just 2,Just [2 % 5,(-10) % 1,(-7) % 15,60 % 1,1 % 15])
  *Univariate Control.Monad> let f t = 1%(1+t^2)
  *Univariate Control.Monad> let fs = map f [0..]
  *Univariate Control.Monad> shiftAndThieleC $ shiftaMatrix fs
  (Just 0,Just [1 % 1,(-2) % 1,(-2) % 1,2 % 1,1 % 1])
```

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

```
> 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 -- only constant term
>   | otherwise = nextStep as [x,x-1 ..]

  *Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
  *Univariate> let hs = map h [0..]
  *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> take 5 hs
  [3 % 1,27 % 23,87 % 85,183 % 187,45 % 47]
  *Univariate> map th [0..5]
  [3 % 1,27 % 23,87 % 85,183 % 187,45 % 47,69 % 73]
```

We represent a rational function by a tuple of coefficient lists:

```
  (ns,ds) :: ([Ratio Int],[Ratio Int])
```

where ns and ds are coef-list-rep of numerator polynomial and denominator polynomial.
Here is a translator from coefficients lists to rational function.

```
> -- similar to p2fct
> 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]
```

The following canonicalizer reduces the tuple-rep of rational function in canonical form
That is, the coefficien of the lowest degree term of the denominator to be 1.
However, since our input starts from 0 and this means

```
    firstNonzero is the same as head.

> 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

What we need is a translator from Thiele coefficients to this tuple-rep.

> 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)
>

  *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])

> thiele2coef' -- shifted version (0 -> sft)
>   :: Integral a =>
>      Ratio a -> [Ratio a] -> ([Ratio a], [Ratio a])
> thiele2coef' sft [a] = ([a],1)
> thiele2coef' sft as = canonicalize $ t2r as sft
>   where
```

```
>     t2r [an,an'] n = (([an*an'-n] + z),[an'])
>     t2r (a:as)   n = ((a .* num) + ((z - [n]) * den), num)
>       where
>         (num, den) = t2r as (n+1)

  *Univariate> let f x = x^2%(1+x^2)
  *Univariate> shiftAndThieleC $ map f [0..]
  (Just 0,Just [0 % 1,2 % 1,2 % 1,(-2) % 1,(-1) % 1])
  *Univariate> take 3 $ shiftaMatrix (map f [0..])
  [Just [0 % 1,2 % 1,2 % 1,(-2) % 1,(-1) % 1]
  ,Just [1 % 2,10 % 3,3 % 5,(-130) % 3,(-1) % 10]
  ,Just [4 % 5,10 % 1,6 % 25,(-150) % 1,(-1) % 25]
  ]
  *Univariate> thiele2coef' 0 [0 % 1,2 % 1,2 % 1,(-2) % 1,(-1) % 1]
  ([0 % 1,0 % 1,1 % 1],[1 % 1,0 % 1,1 % 1])
  *Univariate> thiele2coef' 1 [1 % 2,10 % 3,3 % 5,(-130) % 3,(-1) % 10]
  ([0 % 1,0 % 1,1 % 1],[1 % 1,0 % 1,1 % 1])

> shiftAndThiele2coef (Just sft, Just ts) = Just $ thiele2coef' (fromIntegral sft) ts
> shiftAndThiele2coef _                   = Nothing
>
> list2rat' :: (Integral a) => [Ratio a] -> Maybe ([Ratio a], [Ratio a])
> list2rat' = shiftAndThiele2coef . shiftAndThieleC
>
> list2rat'' lst = let (_,s,ts) = degSftTC lst in
>   shiftAndThiele2coef (s, join ts)

  *Univariate> let f t = t%(1+t^2)
  *Univariate> let fs = map (\t -> 1%(1+t^2)) [0..]
  *Univariate> list2rat' fs
  Just ([1 % 1,0 % 1,0 % 1],[1 % 1,0 % 1,1 % 1])
  *Univariate> shiftAndThieleC fs
  (Just 0,Just [1 % 1,(-2) % 1,(-2) % 1,2 % 1,1 % 1])
  *Univariate> let fs = map (\t -> t%(1+t^2)) [0..]
  *Univariate> let gs = map (\t -> t%(1+t^2)) [0..]
  *Univariate> shiftAndThieleC gs
  (Just 2,Just [2 % 5,(-10) % 1,(-7) % 15,60 % 1,1 % 15])
  *Univariate> list2rat' gs
  Just ([0 % 1,1 % 1,0 % 1],[1 % 1,0 % 1,1 % 1])
  *Univariate> let hs = map (\t -> t^2%(1+t^2)) [0..]
```

```
  *Univariate> shiftAndThieleC hs
  (Just 0,Just [0 % 1,2 % 1,2 % 1,(-2) % 1,(-1) % 1])
  *Univariate> list2rat' hs
  Just ([0 % 1,0 % 1,1 % 1],[1 % 1,0 % 1,1 % 1])

  *Univariate> let f t = t%(1+t^2)
  *Univariate> let fs = map f [0..]
  *Univariate> let aMat = [map (\j -> a' fs j i) [0..] | i <- [0..]]
  *Univariate> take 6 $ map (take 5) aMat
  [[Just (0 % 1),Just (1 % 2),Just (2 % 5),Just (3 % 10),Just (4 % 17)]
  ,[Just (2 % 1),Just ((-10) % 1),Just ((-10) % 1),Just ((-170) % 11),Just ((-442) % 19)]
  ,[Just (1 % 3),Nothing,Just ((-7) % 15),Just ((-77) % 240),Just ((-437) % 1785)]
  ,[Nothing,Nothing,Just (60 % 1),Just (2832 % 11),Just (13020 % 19)]
  ,[Nothing,Nothing,Just (1 % 15),Just (1 % 48),Just (1 % 105)]
  ,[Nothing,Nothing,Nothing,Nothing,Nothing]
  ]
```

4.4 Multivariate.lhs

Listing 4.4: Multivariate.lhs

```
Multivariate.lhs

> module Multivariate where

> import Data.Ratio
> import Data.List (transpose)
> import Univariate -- ( list2pol, list2npolTimes
>                   -- , tDegree, list2rat'
>                   -- )

Let us start 2-variate polynomials.
First, make a naive grid.

  *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]]
  [[3,13,5,69,115,173,243,325,419,525]
  ,[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]
  ,[267,37,359,423,499,587,687,799,923,1059]
  ,[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]
  ]

Assuming the list of lists is a matrix of 2-variate function's values,
  f i j

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

  *Multivariate> tablize (\x y -> (x,y)) 4
  [[(0,0),(0,1),(0,2),(0,3)]
  ,[(1,0),(1,1),(1,2),(1,3)]
  ,[(2,0),(2,1),(2,2),(2,3)]
  ,[(3,0),(3,1),(3,2),(3,3)]
  ]

  *Multivariate> let fTable = tablize 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]
  ]

Let us take the transpose of this "matrix" to see the behavior of coefficients.
```

```
  *Multivariate> let f z1 z2 = 3+2*z1+4*z2+7*z1^2+5*z1*z2+6*z2^2
  *Multivariate> let fTable = tablize f 10
  *Multivariate> map list2pol fTable
  [[3 % 1,4 % 1,6 % 1]
  ,[2 % 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]
  ]
  *ultivariate> transpose 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]
  ]
  *Multivariate> map list2pol it
  [[3 % 1,2 % 1,7 % 1]
  ,[4 % 1,5 % 1]
  ,[6 % 1]]

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

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

There are some bad-behavior polynomials;
  *Multivariate> table2pol $ tablize (\x y -> x*y) 20
  [[0 % 1],[1 % 1,1 % 1]]
  *Multivariate> tablize (\x y -> (x,y)) 5
  [[(0,0),(0,1),(0,2),(0,3),(0,4)]
  ,[(1,0),(1,1),(1,2),(1,3),(1,4)]
  ,[(2,0),(2,1),(2,2),(2,3),(2,4)]
  ,[(3,0),(3,1),(3,2),(3,3),(3,4)]
  ,[(4,0),(4,1),(4,2),(4,3),(4,4)]
```

```
  ]
  *Multivariate> tablize (\x y -> (x*y)) 5
  [[0,0,0,0,0]
  ,[0,1,2,3,4]
  ,[0,2,4,6,8]
  ,[0,3,6,9,12]
  ,[0,4,8,12,16]
  ]

Here we have assumed that the list of functions has the same length, but

  *Multivariate> map list2pol $ tablize (\x y -> x*y) 5
  [[0 % 1],[0 % 1,1 % 1],[0 % 1,2 % 1],[0 % 1,3 % 1],[0 % 1,4 % 1]]

So, we should repeat 0's if we have zero-function.

> xyDegree f = (dX, dY)
>   where
>     dX = length . list2pol $ map (\t -> f t 1) [0..]
>     dY = length . list2pol $ map (\t -> f 1 t) [0..]

  *Multivariate> let test x y = x^2*(2*y + y^3)
  *Multivariate> uncurry (*) . xyDegree $ test
  6
  *Multivariate> maximum . map (length . list2pol) . tablize test $ 6
  4
  *Multivariate> map (take 4 . (++ (repeat (0%1))) . list2pol) . tablize test $ 6
  [[0 % 1,0 % 1,0 % 1,0 % 1]
  ,[0 % 1,2 % 1,0 % 1,1 % 1]
  ,[0 % 1,8 % 1,0 % 1,4 % 1]
  ,[0 % 1,18 % 1,0 % 1,9 % 1]
  ,[0 % 1,32 % 1,0 % 1,16 % 1]
  ,[0 % 1,50 % 1,0 % 1,25 % 1]
  ]
  *Multivariate> map list2pol . transpose $ it
  [[0 % 1],[0 % 1,0 % 1,2 % 1],[0 % 1],[0 % 1,0 % 1,1 % 1]]

  *Multivariate> let test x y = (1%3)*x^2*((2%5)*y + ((3%4)*x*y^3))
                                -- = (2%15)*x^2*y + (1%4)
```

```
                                        *x^3*y^3
  *Multivariate> xyDegree test
  (3,3)
  *Multivariate> map (take 4 . (++ (repeat (0%1))) . list2pol) . tablize test $ 9
  [[0 % 1,0 % 1,0 % 1,0 % 1]
  ,[0 % 1,2 % 15,0 % 1,1 % 4]
  ,[0 % 1,8 % 15,0 % 1,2 % 1]
  ,[0 % 1,6 % 5,0 % 1,27 % 4]
  ,[0 % 1,32 % 15,0 % 1,16 % 1]
  ,[0 % 1,10 % 3,0 % 1,125 % 4]
  ,[0 % 1,24 % 5,0 % 1,54 % 1]
  ,[0 % 1,98 % 15,0 % 1,343 % 4]
  ,[0 % 1,128 % 15,0 % 1,128 % 1]
  ]
  *Multivariate> map list2pol . transpose $ it
  [[0 % 1],[0 % 1,0 % 1,2 % 15],[0 % 1],[0 % 1,0 % 1,0 % 1,1 % 4]]

> xyPol2Coef :: (Enum t, Integral a, Num t) =>
>               (t -> t -> Ratio a) -> [[Ratio a]]
> xyPol2Coef f = map list2pol . transpose . map (take num . (++ (repeat (0%1))) . list2pol) . tablize f $ rank
>   where
>     rank = uncurry (*) . xyDegree $ f
>     num  = maximum . map (length . list2pol) . tablize f $ rank

  *Multivariate> let test x y = (1%3)*x^2*((2%5)*y + ((3%4)*x*y^3))
  *Multivariate> xyPol2Coef test
  [[0 % 1],[0 % 1,0 % 1,2 % 15],[0 % 1],[0 % 1,0 % 1,0 % 1,1 % 4]]
  *Multivariate> let test2 x y = x*y
  *Multivariate> xyPol2Coef test2
  [[0 % 1],[0 % 1,1 % 1]]
  *Multivariate> let test3 x y = x^3*y^4
  *Multivariate> xyPol2Coef test3
  [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,0 % 1,1 % 1]]

> table2pol' :: Integral a => [[Ratio a]] -> [[Ratio a]]
> table2pol' tbl = map list2pol . transpose . map (take num . (++ (repeat (0%1))) . list2pol) $ tbl
>   where
```

```
>      num = maximum . map (length . list2pol) $ tbl

--

2-variable rational functions

  *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)
  *Univariate> let auxh x y t = h (t*x) (t*y)
  *Univariate> let auxhs = [map (auxh 1 y) [0..] | y <- [0..]]
  *Univariate> take 10 $ map list2rat' auxhs
  [Just ([1 % 1,2 % 1,7 % 1],[1 % 1,7 % 1,10 % 1])
  ,Just ([1 % 1,6 % 1,18 % 1],[1 % 1,15 % 1,20 % 1])
  ,Just ([1 % 1,10 % 1,41 % 1],[1 % 1,23 % 1,48 % 1])
  ,Just ([1 % 1,14 % 1,76 % 1],[1 % 1,31 % 1,94 % 1])
  ,Just ([1 % 1,18 % 1,123 % 1],[1 % 1,39 % 1,158 % 1])
  ,Just ([1 % 1,22 % 1,182 % 1],[1 % 1,47 % 1,240 % 1])
  ,Just ([1 % 1,26 % 1,253 % 1],[1 % 1,55 % 1,340 % 1])
  ,Just ([1 % 1,30 % 1,336 % 1],[1 % 1,63 % 1,458 % 1])
  ,Just ([1 % 1,34 % 1,431 % 1],[1 % 1,71 % 1,594 % 1])
  ,Just ([1 % 1,38 % 1,538 % 1],[1 % 1,79 % 1,748 % 1])
  ]
  *Univariate> sequence it
  Just [([1 % 1,2 % 1,7 % 1],[1 % 1,7 % 1,10 % 1])
       ,([1 % 1,6 % 1,18 % 1],[1 % 1,15 % 1,20 % 1])
       ,([1 % 1,10 % 1,41 % 1],[1 % 1,23 % 1,48 % 1])
       ,([1 % 1,14 % 1,76 % 1],[1 % 1,31 % 1,94 % 1])
       ,([1 % 1,18 % 1,123 % 1],[1 % 1,39 % 1,158 % 1])
       ,([1 % 1,22 % 1,182 % 1],[1 % 1,47 % 1,240 % 1])
       ,([1 % 1,26 % 1,253 % 1],[1 % 1,55 % 1,340 % 1])
       ,([1 % 1,30 % 1,336 % 1],[1 % 1,63 % 1,458 % 1])
       ,([1 % 1,34 % 1,431 % 1],[1 % 1,71 % 1,594 % 1])
       ,([1 % 1,38 % 1,538 % 1],[1 % 1,79 % 1,748 % 1])
       ]
  *Univariate> fmap (map fst) it
  Just [[1 % 1,2 % 1,7 % 1]
       ,[1 % 1,6 % 1,18 % 1]
       ,[1 % 1,10 % 1,41 % 1]
       ,[1 % 1,14 % 1,76 % 1]
       ,[1 % 1,18 % 1,123 % 1]
       , ..
       ]
  *Univariate> fmap transpose it
  Just [[1 % 1,1 % 1,1 % 1,1 % 1,1 % 1,1 % 1,1 % 1,1 %
```

```
      1,1 % 1,1 % 1]
         ,[2 % 1,6 % 1,10 % 1,14 % 1,18 % 1,22 % 1,26 %
             1,30 % 1,34 % 1,38 % 1]
         ,[7 % 1,18 % 1,41 % 1,76 % 1,123 % 1,182 % 1,253 %
              1,336 % 1,431 % 1,538 % 1]
         ]
  *Univariate> fmap (map list2pol) it
  Just [[1 % 1],[2 % 1,4 % 1],[7 % 1,5 % 1,6 % 1]]

  *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)
  *Multivariate> let auxh x y t = h (t*x) (t*y)
  *Multivariate> let auxhs = [map (auxh 1 y) [0..100] | y
       <- [0..100]]
  *Multivariate> fmap (map list2pol . transpose . map fst
      ) . sequence . map list2rat' $ auxhs
  Just [[1 % 1],[2 % 1,4 % 1],[7 % 1,5 % 1,6 % 13]]
  *Multivariate> fmap (map list2pol . transpose . map snd
      ) . sequence . map list2rat' $ auxhs
  Just [[1 % 1],[7 % 3,8 % 1],[10 % 1,1 % 1,9 % 1]]

> -- SUPER SLOW IMPLEMENTATION, DO NOT USE THIS!
> table2ratf
>   :: Integral a =>
>      [[Ratio a]] -> (Maybe [[Ratio a]], Maybe [[Ratio a
   ]])
> table2ratf table = (t2r fst table, t2r snd table)
>   where
> --    t2r' third = fmap (map third) . sequence . map
   list2rat'
>
>     myMax Nothing   = 0
>     myMax (Just ns) = ns
>
>     t2r third = fmap (map list2pol . transpose . map (
   take num . (++ (repeat (0%1))) . third)) .
>                 mapM list2rat'
>       where
>         num = myMax . fmap (maximum . map (length . fst
   )) . mapM list2rat' $ table
>
> -- fmap (maximum . map (length . fst)) . sequence . map
   list2rat'
> -- map (take num . (++ (repeat (0%1))) . list2pol)
>
```

```
> tablizer :: (Num a, Enum a) => (a -> a -> b) -> a -> [[b]]
> tablizer f n = [map (f_t 1 y) [0..(n-1)] | y <- [0..(n-1)]]
>   where
>     f_t x y t = f (t*x) (t*y)

  *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)
  *Multivariate> let hTable = tablizer h 20
  *Multivariate> table2ratf hTable
  (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]])

Note that, the sampling points for n=10 case are

  *Multivariate> tablizer (\x y -> (x,y)) 10
  [[(0,0),(1,0),(2,0) ,(3,0) ,(4,0) ,(5,0) ,(6,0) ,(7,0) ,(8,0) ,(9,0)]
  ,[(0,0),(1,1),(2,2) ,(3,3) ,(4,4) ,(5,5) ,(6,6) ,(7,7) ,(8,8) ,(9,9)]
  ,[(0,0),(1,2),(2,4) ,(3,6) ,(4,8) ,(5,10),(6,12),(7,14),(8,16),(9,18)]
  ,[(0,0),(1,3),(2,6) ,(3,9) ,(4,12),(5,15),(6,18),(7,21),(8,24),(9,27)]
  ,[(0,0),(1,4),(2,8) ,(3,12),(4,16),(5,20),(6,24),(7,28),(8,32),(9,36)]
  ,[(0,0),(1,5),(2,10),(3,15),(4,20),(5,25),(6,30),(7,35),(8,40),(9,45)]
  ,[(0,0),(1,6),(2,12),(3,18),(4,24),(5,30),(6,36),(7,42),(8,48),(9,54)]
  ,[(0,0),(1,7),(2,14),(3,21),(4,28),(5,35),(6,42),(7,49),(8,56),(9,63)]
  ,[(0,0),(1,8),(2,16),(3,24),(4,32),(5,40),(6,48),(7,56),(8,64),(9,72)]
  ,[(0,0),(1,9),(2,18),(3,27),(4,36),(5,45),(6,54),(7,63),(8,72),(9,81)]
  ]

  *Multivariate> let f x y = (1 + 2*x + 3*y + 4*x^2 + (1%5)*x*y + (1%6)*y^2)
                               / (7 + 8*x + (1%9)*y + x^2 + x*y + 10*y^2)
  *Multivariate> let g x y = (11 + 10*x + 9*y) / (8 + 7*x^2 + (1%6)*x*y + 5*y^2)
```

```
*Multivariate> table2ratf $ tablizer f 20
(Just [[1 % 7],[2 % 7,3 % 7],[4 % 7,1 % 35,1 % 42]]
,Just [[1 % 1],[8 % 7,1 % 63],[1 % 7,1 % 7,10 % 7]])
*Multivariate> table2ratf $ tablizer g 20
(Just [[11 % 8],[5 % 4,9 % 8],[0 % 1]]
,Just [[1 % 1],[0 % 1],[7 % 8,1 % 48,5 % 8]])
*Multivariate> let h x y = (f x y) / (g x y)
*Multivariate> table2ratf $ tablizer h 20
(Just [[8 % 77],[16 % 77,24 % 77],[39 % 77,53 % 2310,19 % 231]
      ,[2 % 11,64 % 231,3 % 22,15 % 77],[4 % 11,31 % 1155,106 % 385,37 % 2772,5 % 462]]
,Just [[1 % 1],[158 % 77,578 % 693],[13 % 11,757 % 693,111 % 77]
      ,[10 % 77,19 % 77,109 % 77,90 % 77]]
)
*Multivariate> table2ratf $ tablizer f 10
(*** Exception: Prelude.!!: index too large
*Multivariate> table2ratf $ tablizer f 13
(*** Exception: Prelude.!!: index too large
*Multivariate> table2ratf $ tablizer f 15
(Just [[1 % 7],[2 % 7,3 % 7],[4 % 7,1 % 35,1 % 42]]
,Just [[1 % 1],[8 % 7,1 % 63],[1 % 7,1 % 7,10 % 7]])
*Multivariate> table2ratf $ tablizer g 11
(*** Exception: Prelude.!!: index too large
*Multivariate> table2ratf $ tablizer g 13
(*** Exception: Prelude.!!: index too large
*Multivariate> table2ratf $ tablizer g 15
(Just [[11 % 8],[5 % 4,9 % 8],[0 % 1]],Just [[1 % 1],[0 % 1],[7 % 8,1 % 48,5 % 8]])
*Multivariate> table2ratf $ tablizer (\x y -> x/(1+x^2)) 10
(*** Exception: Prelude.!!: index too large
*Multivariate> table2ratf $ tablizer (\x y -> x/(1+x^2)) 13
(*** Exception: Prelude.!!: index too large
*Multivariate> table2ratf $ tablizer (\x y -> x/(1+x^2)) 20
(Just [[0 % 1],[1 % 1],[0 % 1]],Just [[1 % 1],[0 % 1],[1 % 1]])
*Multivariate> table2ratf $ tablizer (\x y -> x/(1+x^2)) 13
(*** Exception: Prelude.!!: index too large
*Multivariate> table2ratf $ tablizer (\x y -> x/(1+x^2)) 15
```

```
  (Just [[0 % 1],[1 % 1],[0 % 1]],Just [[1 % 1],[0 % 1],[1 % 1]])

> -- Alternative transpose, filling with the default value.
> -- I basically followed the implementation of standard Prelude.
> transposeWith :: a -> [[a]] -> [[a]]
> transposeWith _ [] = []
> transposeWith z ([] : xss)
>   | all null xss = []
>   | otherwise    = (z : [h | (h:_) <- xss])
>                  : transposeWith z ([] : [t | (_:t) <- xss])
> transposeWith z ((x:xs) : xss) = (x : [h | (h:_) <- xss])
>                                : transposeWith z (xs : [t | (_:t) <- xss])

  *Multivariate> let f x y = (x*y)%(1+y)^2
  *Multivariate> let tbl = tablizer f 20
  *Multivariate> fmap (map list2pol . (transposeWith (0%1)) . map fst) . sequence . map list2rat' $ tbl
  Just [[0 % 1],[0 % 1],[0 % 1,1 % 1]]
  *Multivariate> fmap (map list2pol . (transposeWith (0%1)) . map snd) . sequence . map list2rat' $ tbl
  Just [[1 % 1],[0 % 1,2 % 1],[0 % 1,0 % 1,1 % 1]]

> table2ratf' table = (t2r fst table, t2r snd table)
>   where
>     t2r third = fmap (map list2pol . transposeWith (0%1) . map third) . mapM list2rat'
>
> table2ratf'' table = (t2r fst table, t2r snd table)
>   where
>     t2r third = fmap (map list2pol . transposeWith (0%1) . map third) . mapM list2rat''

> wilFunc x y = (x^2*y^2) % ((1 + y)^3)

  *Multivariate> table2ratf $ tablizer wilFunc 20
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 % 1]]
  ,Just [[1 % 1],[0 % 1,3 % 1],[0 % 1,0 % 1,3 % 1],[0 % 1,0 % 1,0 % 1,1 % 1],[0 % 1]])
```

```
  (3.91 secs, 2,850,226,792 bytes)
  *Multivariate> table2ratf' $ tablizer wilFunc 20
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 % 1]]
  ,Just [[1 % 1],[0 % 1,3 % 1],[0 % 1,0 % 1,3 % 1],[0 % 1,0 % 1,0 % 1,1 % 1]])
  (2.00 secs, 1,425,753,744 bytes)
  *Multivariate> table2ratf'' $ tablizer wilFunc 20
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 % 1]]
  ,Just [[1 % 1],[0 % 1,3 % 1],[0 % 1,0 % 1,3 % 1],[0 % 1,0 % 1,0 % 1,1 % 1]])
  (1.73 secs, 1,234,282,424 bytes)

> wilFunc2 :: Int -> Int -> Ratio Int
> wilFunc2 x y = x^4 * y^2 * (1+y+y^2)^2 % (1+y)^4
```

4.5 GUniFin.lhs

Listing 4.5: GUniFin.lhs

```
GUniFin.lhs

Non sequential inputs Newton-interpolation with finite fields.
Our target is a function
  f :: Q -> Q
which means to determine (canonical) coefficients.
Accessible input is pairs of in-out, i.e., a (sub) graph of f.

> module GUniFin where
> --
> import Data.Ratio
> import Data.Maybe
> import Data.Either
> import Data.List
> import Control.Monad
> --
> import Polynomials
> import Ffield
> --
> type Q = Ratio Int    -- Rational fields
> type Graph = [(Q,Q)] -- [(x, f x) | x <- someFinieRange ]
```

```haskell
> --
> -- f [a,b,c ..] -> [(f a b), (f b c) ..]
> -- pair wise application
> map' :: (a -> a -> b) -> [a] -> [b]
> map' f as = zipWith f as (tail as)
>
> -- To select Z_p valid inputs.
> sample :: Int   -- prime
>        -> Graph -- increasing input
>        -> Graph
> sample p = filter ((< (fromIntegral p)) . fst)
>
> -- To eliminate (1%p) type "fake" infinity.
> -- After eliminating these, we can freely use 'modp', primed version.
> check :: Int   -- prime
>       -> Graph
>       -> Graph -- safe data sets
> check p = filter (not . isDanger p)
>   where
>     isDanger -- To detect (1%p) type infinity.
>       :: Int -- prime
>       -> (Q,Q) -> Bool
>     isDanger p (_, fx) = (d `rem` p) == 0
>       where
>         d = denominator fx
>
> project :: Int -> (Q,Q) -> (Int, Int)
> project p (x, fx) -- for simplicity
>   | denominator x == 1 = (numerator x, fx `modp'` p)
>   | otherwise          = error "project:␣integer␣input?"
>
> -- From Graph to Zp (safe) values.
> onZp
>   :: Int                -- base prime
>   -> Graph
>   -> [(Int, Int)] -- in-out on Zp value
> onZp p = map (project p) . check p . sample p
>
> -- using record syntax
> data PDiff
>   = PDiff { points    :: (Int, Int) -- end points
>           , value     :: Int        -- Zp value
>           , basePrime :: Int
```

```
>           }
>   deriving (Show, Read)
>
> toPDiff
>   :: Int         -- prime
>   -> (Int, Int) -- in and out mod p
>   -> PDiff
> toPDiff p (x,fx) = PDiff (x,x) fx p
>
> newtonTriangleZp :: [PDiff] -> [[PDiff]]
> newtonTriangleZp fs
>   | length fs < 3 = []
>   | otherwise     = helper [sf3] (drop 3 fs)
>   where
>     sf3 = reverse . take 3 $ fs -- [[f2,f1,f0]]
>     helper fss [] = error "newtonTriangleZp: need more evaluation"
>     helper fss (f:fs)
>       | isConsts 3 . last $ fss = fss
>       | otherwise               = helper (add1 f fss) fs
>
> isConsts
>   :: Int -- 3times match
>   -> [PDiff] -> Bool
> isConsts n ds
>   | length ds < n = False
> -- isConsts n ds     = all (==l) $ take (n-1) ls
>   | otherwise     = all (==l) $ take (n-1) ls
>   where
>     (l:ls) = map value ds
>
> -- backward, each [PDiff] is decreasing inputs (i.e., reversed)
> add1 :: PDiff -> [[PDiff]] -> [[PDiff]]
> add1 f [gs] = fgs : [zipWith bdiffStep fgs gs] -- singleton
>   where
>     fgs = f:gs
> add1 f (gg@(g:gs) : hhs) -- gg is reversed order
>               = (f:gg) : add1 fg hhs
>   where
>     fg = bdiffStep f g
>
> -- backward
```

```
> bdiffStep :: PDiff -> PDiff -> PDiff
> bdiffStep (PDiff (y,y') g q) (PDiff (x,x') f p)
>   | p == q     = PDiff (x,y') finiteDiff p
>   | otherwise = error "bdiffStep:␣different␣primes?"
>   where
>     finiteDiff = ((fg % xy') `modp'` p)
>     xy' = (x - y' `mod` p)
>     fg = ((f-g) `mod` p)
>
> graph2Zp :: Int -> Graph -> [(Int, Int)]
> graph2Zp p = onZp p . check p . sample p
>
> graph2PDiff :: Int -> Graph -> [PDiff]
> graph2PDiff p = map (toPDiff p) . graph2Zp p
>
> newtonTriangleZp' :: Int -> Graph -> [[PDiff]]
> newtonTriangleZp' p = newtonTriangleZp . graph2PDiff p
>
> newtonCoeffZp :: Int -> Graph -> [PDiff]
> newtonCoeffZp p = map head . newtonTriangleZp' p

  *GUniFin> let gs = map (\x -> (x,x^2 + (1%2)*x + 1%3))
                          [1,2,4,5,9,10,11] :: Graph
  *GUniFin> newtonCoeffZp 101 gs
  [PDiff {points = (9,9), value = 69, basePrime = 101}
  ,PDiff {points = (5,9), value = 65, basePrime = 101}
  ,PDiff {points = (4,9), value = 1, basePrime = 101}
  ]
  *GUniFin> map (\x -> (Just . value $ x, basePrime x))
     it
  [(Just 69,101),(Just 65,101),(Just 1,101)]

We take formally the canonical form on Zp,
then apply rational "number" reconstruction.

> n2cZp :: [PDiff] -> ([Int], Int)
> n2cZp graph = (helper graph, p)
>   where
>     p = basePrime . head $ graph
>     helper [d]    = [value d]
>     helper (d:ds) = map (`mod` p) $ ([value d] + (z *
   next))
>                                     - (map (`mod` p) (zd
    .* next))
>       where
```

```
>           zd = fst . points $ d
>           next = helper ds
>
> format :: ([Int],Int) -> [(Maybe Int, Int)]
> format (as,p) = [(return a,p) | a <- as]

  *GUniFin> let gs = map (\x -> (x,x^2 + (1%2)*x + 1%3))
                             [0,2,3,5,7,8,11] :: Graph
  *GUniFin> newtonCoeffZp 10007 gs
  [PDiff {points = (7,7), value = 8392, basePrime = 10007}
  ,PDiff {points = (5,7), value = 5016, basePrime = 10007}
  ,PDiff {points = (3,7), value = 1, basePrime = 10007}
  ]
  *GUniFin> n2cZp it
  ([3336,5004,1],10007)
  *GUniFin> format it
  [(Just 3336,10007),(Just 5004,10007),(Just 1,10007)]
  *GUniFin> map guess it
  [Just (1 % 3,10007),Just (1 % 2,10007),Just (1 % 1,10007)]

  *GUniFin> let gs = map (\x -> (x,x^2 + (1%2)*x + 1%3))
                             [0,2,3,5,7,8,11] :: Graph
  *GUniFin> map guess . format . n2cZp . newtonCoeffZp 10007 $ gs
  [Just (1 % 3,10007),Just (1 % 2,10007),Just (1 % 1,10007)]
  *GUniFin> let gs = map (\x -> (x,x^5 + x^2 + (1%2)*x + 1%3))
                             [0,2,3,5,7,8,11,13,17,18,19,21,24,28,31,33,34]
                             :: Graph
  *GUniFin> map guess . format . n2cZp . newtonCoeffZp 10007 $ gs
  [Just (1 % 3,10007),Just (1 % 2,10007),Just (1 % 1,10007)
  ,Just (0 % 1,10007),Just (0 % 1,10007),Just (1 % 1,10007)
  ]

> preTrial gs p = format . n2cZp . newtonCoeffZp p $ gs

  *GUniFin> let gs = map (\x -> (x,x^5 + x^2 + (1%2)*x +
```

```
    1%3))
                            [0,2,3,5,7,8,11,13,17,18,19,21,24,28,31,33,34]
                            :: Graph
  *GUniFin> map reconstruct . transpose . map (preTrial gs) $ bigPrimes
  [Just (1 % 3),Just (1 % 2),Just (1 % 1)
  ,Just (0 % 1),Just (0 % 1),Just (1 % 1)
  ]

Here is "a" final version, the univariate polynomial reconstruction
with finite fields.

> uniPolCoeff :: Graph -> Maybe [(Ratio Int)]
> uniPolCoeff gs
>   = (mapM reconstruct' . transpose . map (preTrial gs)) bigPrimes

  *GUniFin> let gs = map (\x -> (x,x^5 + x^2 + (1%2)*x + 1%3))
                            [0,2,3,5,7,8,11,13,17,18,19,21,24,28,31,33,34]
                            :: Graph
  *GUniFin> gs
  [(0 % 1,1 % 3),(2 % 1,112 % 3),(3 % 1,1523 % 6),(5 % 1,18917 % 6)
  ,(7 % 1,101159 % 6),(8 % 1,98509 % 3),(11 % 1,967067 % 6)
  ,(13 % 1,2228813 % 6),(17 % 1,8520929 % 6),(18 % 1,5669704 % 3)
  ,(19 % 1,14858819 % 6),(21 % 1,24507317 % 6),(24 % 1,23889637 % 3)
  ,(28 % 1,51633499 % 3),(31 % 1,171780767 % 6),(33 % 1,234818993 % 6)
  ,(34 % 1,136309792 % 3)
  ]
  *GUniFin> uniPolCoeff gs
  Just [1 % 3,1 % 2,1 % 1,0 % 1,0 % 1,1 % 1]

  *GUniFin> let fs = map (\x -> (x,(3+x+(1%3)*x^9)/(1)))
                            [1,3..101] :: Graph
  *GUniFin> uniPolCoeff fs
  Just [3 % 1,1 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,1 % 3]
```

```
  *GUniFin> let fs = map (\x -> (x,(3+x+(1%3)*x^10)/(1)))
                             [1,3..101] :: Graph
  *GUniFin> uniPolCoeff fs
  *** Exception: newtonBT: need more evaluation
  CallStack (from HasCallStack):
    error, called at GUniFin.lhs:79:23 in main:GUniFin
  *GUniFin> let fs = map (\x -> (x,(3+x+(1%3)*x^10)/(1)))
                             [1,3..1001] :: Graph
  *GUniFin> uniPolCoeff fs
  *** Exception: newtonBT: need more evaluation
  CallStack (from HasCallStack):
    error, called at GUniFin.lhs:79:23 in main:GUniFin
```

Rough estimation says, in 64-bits system with sequential inputs,
the upper limit of degree is about 15.
If we use non sequential inputs, this upper limit will go down.

```
-- up to here, polinomials
--
-- from now on, rational functions
```

Non sequential inputs Thiele-interpolation with finite fields.

Let me start naive rho with non-sequential inputs:

```
> -- over rational (infinite) field
> rho :: Graph -> Int -> [Q]
> rho gs 0 = map snd gs
> rho gs 1 = zipWith (/) xs' fs'
>   where
>     xs' = zipWith (-) xs (tail xs)
>     xs  = map fst gs
>     fs' = zipWith (-) fs (tail fs)
>     fs  = map snd gs
> rho gs n = zipWith (+) twoAbove oneAbove
>   where
>     twoAbove = zipWith (/) xs' rs'
>     xs' = zipWith (-) xs (drop n xs)
>     xs  = map fst gs
>     rs' = zipWith (-) rs (tail rs)
>     rs  = rho gs (n-1)
>     oneAbove = tail $ rho gs (n-2)
```

```

This works

  *GUniFin> let func x = (1+x+2*x^2)/(3+2*x +(1%4)*x^2)
  *GUniFin> let fs = map (\x -> (x, func x))
                          [0,1,3,4,6,7,9,10,11,13,14,15,17,19,20] :: Graph
  *GUniFin> let r = rho fs
  *GUniFin> r 0
  [1 % 3,16 % 21,88 % 45,37 % 15,79 % 24,424 % 117,688 % 165,211 % 48
  ,1016 % 221,1408 % 285,407 % 80,1864 % 357,2384 % 437,424 % 75,821 % 143
  ]
  *GUniFin> r 1
  [7 % 3,315 % 188,45 % 23,80 % 33,936 % 311,6435 % 1756,880 % 199
  ,10608 % 2137,62985 % 10804,4560 % 671,28560 % 3821,156009 % 18260
  ,32775 % 3244,10725 % 943
  ]
  *GUniFin> r 2
  [(-604) % 159,5116 % 405,9458 % 1065,18962 % 2253,75244 % 9171
  ,117388 % 14439,174700 % 21603,243084 % 30151,329516 % 40955
  ,436876 % 54375,559148 % 69659,26491 % 3303,138404 % 17267
  ]
  *GUniFin> r 3
  [900 % 469,585 % 938,(-5805) % 938,(-19323) % 938,(-23418) % 469
  ,(-165867) % 1876,(-295485) % 1876,(-111560) % 469,(-651015) % 1876
  ,(-977265) % 1876,(-199317) % 268,(-278589) % 268
  ]
  *GUniFin> r 4
  [8 % 1,8 % 1,8 % 1,8 % 1,8 % 1,8 % 1,8 % 1,8 % 1,8 % 1,8 % 1,8 % 1,8 % 1]

But here is a corner case, an accidental match.
We should detect them and handle them safely.

  *GUniFin> let func x = (x%(1+x^2))
  *GUniFin> let fs = map (\x -> (x%1,func x)) [0..10]
```

```
  *GUniFin> let r = rho fs
  *GUniFin> r 0
  [0 % 1,1 % 2,2 % 5,3 % 10,4 % 17,5 % 26
  ,6 % 37,7 % 50,8 % 65,9 % 82,10 % 101
  ]
  *GUniFin> r 1
  [2 % 1,(-10) % 1,(-10) % 1,(-170) % 11,(-442) % 19
  ,(-962) % 29,(-1850) % 41,(-650) % 11,(-5330) % 71,(-8282) % 89
  ]
  *GUniFin> r 2
  [1 % 3,*** Exception: Ratio has zero denominator

--

> -- We have assumed our out-puts are safe, i.e. no fake infinity.
> initialThieleZp :: [PDiff] -> [[PDiff]]
> initialThieleZp fs
>   | isConsts 3 fs = [first]
>   | otherwise = [first, second]
>   where
>     first  = reverse . take 4 $ fs
>     second = map' reciproDiff first

> {-
> -- To make safe initials.
> initialThieleTriangleZp :: [PDiff] -> [[PDiff]]
> initialThieleTriangleZp ff@(f:fs)
>   | isConsts 3 ff = [reverse $ take 3 ff]
>   | otherwise     = [[g,f],[h]]
>   where
>     g = firstDifferent f fs
>
>     firstDifferent _ []
>       = error "initialThieleTriangleZp: need more points"
>     firstDifferent f (g:gs)
>       = if (g' /= f') then g
>                       else firstDifferent f gs
>       where
>         f' = value f
>         g' = value g
>
>     h = reciproDiff g f
```

```
>
> -- to make safe first two stairs.
> initialThieleZp' :: [PDiff] -> [[PDiff]]
> initialThieleZp' fs
>   | isConsts 3 fs = [reverse $ take 3 fs]
>   | otherwise     = [firsts, seconds]
>   where
>     firsts = undefined
>     seconds = undefined
> -}

> -- reversed order
> reciproDiff :: PDiff -> PDiff -> PDiff
> reciproDiff (PDiff (_,z') u p) (PDiff (w,_) v q)
>   | p /= q = error "reciproDiff:␣wrong␣base␣prime"
>   | otherwise = PDiff (w,z') r p
>   where
>     r = ((zw) * (uv `inversep'` p)) `mod` p
>     zw = (z' - w) `mod` p
>     uv = (u  - v) `mod` p -- assuming (u-v) is not "zero"
>
> -- reciproAdd1 :: PDiff -> ListZipper [PDiff] -> ListZipper [PDiff]
> -- reciproAdd1 _ ([], sb) = ([], sb) -- reversed order
> -- reciproAdd1 f ((gs@(g:_) : hs : iss), [])
> --                          = reciproAdd1 k (iss, [(j:hs), (f:gs)])
> --   where
> --     j = reciproDiff f g
> --     k = addZp' j g
>
> addZp' :: PDiff -> PDiff -> PDiff
> addZp' (PDiff (x,y) v p) (PDiff (_,_) w q)
>   | p /= q = error "addZp':␣wrong␣primes"
>   | otherwise = PDiff (x,y) vw p
>   where
>     vw = (v + w) `mod` p
>
> -- This takes new point and the heads, and returns the new heads.
> thieleHeads
>   :: PDiff   -- a new element  rho8
>   -> [PDiff] -- oldies         [rho7, rho67, rho567, rho4567 ..]
```

```
>   -> [PDiff] --                  [rho8, rho78, rho678, rho5678 ..]
> thieleHeads _ []        = []
> thieleHeads f gg@(g:gs) = f : fg : helper fg gg
>   where
>     fg  = reciproDiff f g
>
>     helper _ bs
>       | length bs < 3 = []
>     helper a (b:bs@(c:d:_)) = e : helper e bs
>       where
>         e = addZp' (reciproDiff a c)
>                    b
>
>     tHs :: PDiff -> [PDiff] -> [PDiff] -- reciprocal diff. part
>     tHs _ [] = []
>     tHs f' hh@(h:hs) = fh : tHs fh hs
>       where
>         fh = reciproDiff f' h
>
> thieleTriangle' :: [PDiff] -> [[PDiff]]
> thieleTriangle' fs
>   | length fs < 4 = []
>   | otherwise     = helper fourThree (drop 4 fs)
>   where
>     fourThree = initialThieleZp fs
>     helper fss []
>       | isConsts 3 . last $ fss = fss
>       | otherwise               = error "thieleTriangle: need more inputs"
>     helper fss (g:gs)
>       | isConsts 3 . last $ fss = fss
>       | otherwise               = helper gfss gs
>       where
>         gfss = thieleComp g fss
>
> thieleComp :: PDiff -> [[PDiff]] -> [[PDiff]]
> thieleComp g fss = wholeButLast ++ [three]
>   where
>     wholeButLast = zipWith (:) hs fss
>     hs = thieleHeads g (map head fss)
>     three = fiveFour2three $ last2 wholeButLast
>     -- Finally from two stairs (5 and 4 elements),
>     -- we create the bottom 3 elements.
```

```
>
>     last2 :: [a] -> [a]
>     last2 [a,b] = [a,b]
>     last2 (_:bb@(_:_)) = last2 bb
>
> fiveFour2three -- This works!
>   :: [[PDiff]] -- 5 and 4, under last2
>   -> [PDiff]   -- 3
> fiveFour2three [ff@(_:fs), gg] = zipWith addZp' (map' reciproDiff gg) fs
>
> thieleTriangle :: Graph -> Int -> [[PDiff]]
> thieleTriangle fs p = thieleTriangle' $ graph2PDiff p fs
>
> thieleCoeff' :: Graph -> Int -> [PDiff]
> thieleCoeff' fs = map last . thieleTriangle fs
>
> -- thieleCoeff'' fs p = a:b:(zipWith subZp bs as)
> thieleCoeff'' fs p
>   | length (thieleCoeff' fs p) == 1 = thieleCoeff' fs p
>   | otherwise = a:b:(zipWith subZp bs as)
>   where
>     as@(a:b:bs) = thieleCoeff' fs p
> --  as@(a:b:bs) = firstReciprocalDifferences fs p
>
>     subZp :: PDiff -> PDiff -> PDiff
>     subZp (PDiff (x,y) v p) (PDiff (_,_) w q)
>       | p /= q    = error "thileCoeff: different primes"
>       | otherwise = PDiff (x,y) ((v-w) `mod` p) p
>

> t2cZp
>   :: [PDiff]               -- thieleCoeff'' fs p
>   -> (([Int],[Int]), Int) -- (rat-func, basePrime)
> t2cZp gs = (helper gs, p)
>   where
>     p = basePrime . head $ gs
>     helper [n]   = ([(value n) `mod` p], [1])
>     helper [d,e] = ([de',1], [e']) -- base case
>       where
>         de' = ((d'*e' `mod` p) - xd) `mod` p
>         d'  = value d
>         e'  = value e
```

```
>           xd  = snd . points $ d
>       helper (d:ds) = (den', num)
>         where
>           (num, den) = helper ds
>           den'  = map (`mod` p) $ (z * den) + (map (`mod` p) $ num''-den'')
>           num'' = map (`mod` p) ((value d) .* num)
>           den'' = map (`mod` p) ((snd . points $ d) .* den)
>
> -- pre "canonicalizer"
> beforeFormat' :: (([Int], [Int]), Int) -> (([Int], [Int]), Int)
> beforeFormat' ((num,(d:ds)), p) = ((num',den'), p)
>   where
>     num' = map (`mod` p) $ di .* num
>     den' = 1: (map (`mod` p) $ di .* ds)
>     di   = d `inversep'` p
>
> format'
>   :: (([Int], [Int]), Int)
>   -> ([(Maybe Int, Int)], [(Maybe Int, Int)])
> format' ((num,den), p) = (format (num, p), format (den, p))

  *GUniFin> let fs = map (\x -> (x,(1+x)/(2+x))) [0,2,3,4,6,8,9] :: Graph
  *GUniFin> thieleCoeff'' fs 101
  [PDiff {points = (0,0), value = 51, basePrime = 101}
  ,PDiff {points = (0,2), value = 8, basePrime = 101}
  ,PDiff {points = (0,3), value = 51, basePrime = 101}
  ]
  *GUniFin> t2cZp it
  (([1,1],[2,1]),101)
  *GUniFin> format' it
  ([(Just 1,101),(Just 1,101)]
  ,[(Just 2,101),(Just 1,101)]
  )
  *GUniFin> format' . t2cZp . thieleCoeff'' fs $ 101
  ([(Just 1,101),(Just 1,101)],[(Just 2,101),(Just 1,101)])
  *GUniFin> format' . t2cZp . thieleCoeff'' fs $ 103
  ([(Just 1,103),(Just 1,103)],[(Just 2,103),(Just 1,103)])
  *GUniFin> format' . t2cZp . thieleCoeff'' fs $ 107
```

```
  ([(Just 1,107),(Just 1,107)],[(Just 2,107),(Just 1,107)])

> ratCanZp
>   :: Graph -> Int -> ([(Maybe Int, Int)], [(Maybe Int, Int)])
> ratCanZp fs = format' . beforeFormat' . t2cZp . thieleCoeff'' fs

  *GUniFin> let fivePrimes = take 5 bigPrimes
  *GUniFin> let fs = map (\x -> (x,(1+x)/(2+x))) [0,2,3,4,6,8,9] :: Graph
  *GUniFin> map (ratCanZp fs) fivePrimes
  [([(Just 5004,10007),(Just 5004,10007)]
   ,[(Just 1,10007),(Just 5004,10007)]
   )
  ,([(Just 5005,10009),(Just 5005,10009)]
   ,[(Just 1,10009),(Just 5005,10009)]
   )
  ,([(Just 5019,10037),(Just 5019,10037)]
   ,[(Just 1,10037),(Just 5019,10037)]
   )
  ,([(Just 5020,10039),(Just 5020,10039)]
   ,[(Just 1,10039),(Just 5020,10039)]
   )
  ,([(Just 5031,10061),(Just 5031,10061)]
   ,[(Just 1,10061),(Just 5031,10061)]
   )
  ]
  *GUniFin> map fst it
  [[(Just 5004,10007),(Just 5004,10007)]
  ,[(Just 5005,10009),(Just 5005,10009)]
  ,[(Just 5019,10037),(Just 5019,10037)]
  ,[(Just 5020,10039),(Just 5020,10039)]
  ,[(Just 5031,10061),(Just 5031,10061)]
  ]
  *GUniFin> transpose it
  [[(Just 5004,10007),(Just 5005,10009),(Just 5019,10037)
   ,(Just 5020,10039),(Just 5031,10061)
   ]
  ,[(Just 5004,10007),(Just 5005,10009),(Just 5019,10037)
   ,(Just 5020,10039),(Just 5031,10061)
   ]
  ]
  *GUniFin> map reconstruct it
```

```
[Just (1 % 2),Just (1 % 2)]
*GUniFin> map (ratCanZp fs) fivePrimes
[([(Just 5004,10007),(Just 5004,10007)]
 ,[(Just 1,10007),(Just 5004,10007)]
 )
,([(Just 5005,10009),(Just 5005,10009)]
 ,[(Just 1,10009),(Just 5005,10009)]
 )
,([(Just 5019,10037),(Just 5019,10037)]
 ,[(Just 1,10037),(Just 5019,10037)]
 )
,([(Just 5020,10039),(Just 5020,10039)]
 ,[(Just 1,10039),(Just 5020,10039)]
 )
,([(Just 5031,10061),(Just 5031,10061)]
 ,[(Just 1,10061),(Just 5031,10061)]
 )
]
*GUniFin> map snd it
[[(Just 1,10007),(Just 5004,10007)]
,[(Just 1,10009),(Just 5005,10009)]
,[(Just 1,10037),(Just 5019,10037)]
,[(Just 1,10039),(Just 5020,10039)]
,[(Just 1,10061),(Just 5031,10061)]
]
*GUniFin> transpose it
[[(Just 1,10007),(Just 1,10009),(Just 1,10037)
 ,(Just 1,10039),(Just 1,10061)
 ]
,[(Just 5004,10007),(Just 5005,10009),(Just 5019,10037)
 ,(Just 5020,10039),(Just 5031,10061)
 ]
]
*GUniFin> map reconstruct it
[Just (1 % 1),Just (1 % 2)]

> -- uniPolCoeff :: Graph -> Maybe [(Ratio Integer)]
> -- uniPolCoeff gs = sequence . map reconstruct . transpose . map (preTrial gs) $ bigPrimes

> -- Clearly this is double running implementation.
> uniRatCoeff
>   :: Graph -> ([Maybe (Ratio Int)], [Maybe (Ratio Int)])
> uniRatCoeff gs = (num, den)
```

```
>   where
>     (num,den) = (helper fst, helper snd)
>     helper third
>       = map reconstruct' . transpose
>         . map (third . ratCanZp gs) $ bigPrimes

  *GUniFin> let fs = map (\x -> (x,(1+2*x+x^10)/(1+(3%2)*x+x^5))) [0..101] :: Graph
  (0.01 secs, 44,232 bytes)
  *GUniFin> uniRatCoeff fs
  ([Just (1 % 1),Just (2 % 1),Just (0 % 1),Just (0 % 1),Just (0 % 1)
   ,Just (0 % 1),Just (0 % 1),Just (0 % 1),Just (0 % 1),Just (0 % 1)
   ,Just (1 % 1)
   ]
  ,[Just (1 % 1),Just (3 % 2),Just (0 % 1),Just (0 % 1),Just (0 % 1)
   ,Just (1 % 1),Just (0 % 1),Just (0 % 1),Just (0 % 1),Just (0 % 1)
   ]
  )
  (1.72 secs, 1,424,003,616 bytes)

> isJustZero n = Just (0%1) == n
>
> uniRatCoeffShort gs = (num', den')
>   where
>     (num, den) = uniRatCoeff gs
>     (num', den') = (helper num, helper den)
>     helper nd = filter (not . isJustZero . fst) $ zip nd [0..]

  *GUniFin> let fs = map (\x -> (x,(1+2*x+x^10)/(1+(3%2)*x+x^5))) [0..101] :: Graph
  (0.01 secs, 44,320 bytes)
  *GUniFin> uniRatCoeff fs
  ([Just (1 % 1),Just (2 % 1),Just (0 % 1),Just (0 % 1),Just (0 % 1)
   ,Just (0 % 1),Just (0 % 1),Just (0 % 1),Just (0 % 1),Just (0 % 1)
   ,Just (1 % 1)
   ]
  ,[Just (1 % 1),Just (3 % 2),Just (0 % 1),Just (0 % 1),Just (0 % 1)
```

```
   ,Just (1 % 1),Just (0 % 1),Just (0 % 1),Just (0 % 1), Just (0 % 1)
   ]
  )
  (1.72 secs, 1,424,009,472 bytes)
  *GUniFin> uniRatCoeffShort  fs
  ([(Just (1 % 1),0),(Just (2 % 1),1),(Just (1 % 1),10)]
  ,[(Just (1 % 1),0),(Just (3 % 2),1),(Just (1 % 1),5)]
  )
  (1.74 secs, 1,422,577,184 bytes)

> uniRatCoeff'
>   :: Graph -> (Maybe [Ratio Int], Maybe [Ratio Int])
> uniRatCoeff' gs = (num', den')
>   where
>     (num, den) = uniRatCoeff gs
>     num' = sequence num
>     den' = sequence den

> func2graph :: (Q -> Q) -> [Q] -> Graph
> func2graph f xs = [(x, f x) | x <- xs]

  *GUniFin> func2graph g [0,3..30]
  [(0 % 1,0 % 1),(3 % 1,27 % 64),(6 % 1,216 % 343),(9 % 1,729 % 1000)
  ,(12 % 1,1728 % 2197),(15 % 1,3375 % 4096),(18 % 1,5832 % 6859)
  ,(21 % 1,9261 % 10648),(24 % 1,13824 % 15625),(27 % 1,19683 % 21952)
  ,(30 % 1,27000 % 29791)
  ]
  (0.01 secs, 363,080 bytes)
  *GUniFin> uniRatCoeffShort it
  ([(Just (1 % 1),3)]
  ,[(Just (1 % 1),0),(Just (3 % 1),1),(Just (3 % 1),2),(Just (1 % 1),3)]
  )
  (0.30 secs, 231,980,488 bytes)

> -- Up to degree~100 version.
> ratFunc2Coeff
>   :: (Q -> Q) -- rational function
>   -> (Maybe [Ratio Int], Maybe [Ratio Int])
> ratFunc2Coeff f = uniRatCoeff' . func2graph f $ [0..100]
```

```

--

We want to use safe list, i.e., the given graph as much as possible.
So, the easiest way could be
  pick a prime
  construct Thiele triangle up to consts.
  if we face fake infinity before it matches,
    then return Nothing and use another prime
Since we have a lot of bigPrimes.

  *GUniFin> let g x = x^4 / (1+x)^3
  *GUniFin> func2graph g [0..10]
  [(0 % 1,0 % 1),(1 % 1,1 % 8),(2 % 1,16 % 27),(3 % 1,81 % 64)
  ,(4 % 1,256 % 125),(5 % 1,625 % 216),(6 % 1,1296 % 343),(7 % 1,2401 % 512)
  ,(8 % 1,4096 % 729),(9 % 1,6561 % 1000),(10 % 1,10000 % 1331)
  ]
  *GUniFin> let p = head bigPrimes
  *GUniFin> p
  10007
  *GUniFin> graph2PDiff p $ func2graph g [0..10]
  [PDiff {points = (0,0), value = 0, basePrime = 10007}
  ,PDiff {points = (1,1), value = 1251, basePrime = 10007}
  ,PDiff {points = (2,2), value = 2595, basePrime = 10007}
  ,PDiff {points = (3,3), value = 6412, basePrime = 10007}
  ,PDiff {points = (4,4), value = 1363, basePrime = 10007}
  ,PDiff {points = (5,5), value = 2273, basePrime = 10007}
  ,PDiff {points = (6,6), value = 1375, basePrime = 10007}
  ,PDiff {points = (7,7), value = 8624, basePrime = 10007}
  ,PDiff {points = (8,8), value = 9038, basePrime = 10007}
  ,PDiff {points = (9,9), value = 7782, basePrime = 10007}
  ,PDiff {points = (10,10), value = 7150, basePrime =
```

```
    10007}
  ]

We need Maybe-wrapped version of reciprocal (inverse) difference.

> -- normal order for rho-matrix
> inverseDiff
>   :: PDiff -> PDiff -> Maybe PDiff
> inverseDiff (PDiff (w,_) v p) (PDiff (_,z') u _) -- z in reserved
>   | v == u    = Nothing
>   | otherwise = return $ PDiff (w,z') r p
>   where
>     r = ((zw) * (uv `inversep'` p)) `mod` p
>     zw = (z' - w) `mod` p
>     uv = (u - v) `mod` p
>
> inverseDiff' :: Maybe PDiff -> Maybe PDiff -> Maybe PDiff
> inverseDiff' Nothing  _        = Nothing
> inverseDiff' _        Nothing  = Nothing
> inverseDiff' (Just a) (Just b) = inverseDiff a b

> -- rho-matrix version
> -- This implementation is quite straightforward, but no error handling.
> inverseDiffs
>   :: Int     -- a prime
>   -> Int     -- "degree" or the depth of thiele TRAPEZOID
>   -> Graph
>   -> [Maybe PDiff]
> inverseDiffs p 0 fs = map return $ graph2PDiff p fs
> inverseDiffs p 1 fs = map' inverseDiff $ graph2PDiff p fs
> inverseDiffs p n fs
>   = zipWith addPDiff (map' inverseDiff' (inverseDiffs p (n-1) fs))
>                      (tail $ inverseDiffs p (n-2) fs)
>
> addPDiff :: Maybe PDiff -> Maybe PDiff -> Maybe PDiff
> addPDiff Nothing _ = Nothing
> addPDiff _ Nothing = Nothing
> addPDiff (Just a) (Just b) = return $ addZp' a b
```

```

  *GUniFin> let f x = x / (1+x^2)
  *GUniFin> let fs = func2graph f [0..10]
  *GUniFin> sequence $ filter isJust $ inverseDiffs 10007 4 fs
  Just [PDiff {points = (2,6), value = 0, basePrime = 10007}
       ,PDiff {points = (3,7), value = 0, basePrime = 10007}
       ,PDiff {points = (4,8), value = 0, basePrime = 10007}
       ,PDiff {points = (5,9), value = 0, basePrime = 10007}
       ,PDiff {points = (6,10), value = 0, basePrime = 10007}
       ]
  *GUniFin> fmap (isConsts 3) it
  Just True

> isConsts'
>   :: Int -> [Maybe PDiff] -> Bool
> isConsts' n fs
>   | Just True == fmap (isConsts n) fs' = True
>   | otherwise                          = False
>   where
>     fs' = sequence . filter isJust $ fs
>
> -- This is the main function which returns Nothing when we face
> -- so many fake infinities with really bad prime.
> thieleTrapezoid
>   :: Graph -> Int -> Maybe [[Maybe PDiff]]
> thieleTrapezoid fs p
>   | any (isConsts' 3) gs = return gs'
> --  | or $ map (isConsts' 3) gs = return gs'
>   | otherwise                 = Nothing
>   where
>     gs' = aMatrix fs p
>     gs  = map (filter isJust) gs'
>
>     aMatrix
>       :: Graph -> Int -> [[Maybe PDiff]]
>     aMatrix fs p = takeUntil (isConsts' 3)
>                      [inverseDiffs p n fs | n <- [0..]]
>
```

```
> takeUntil
>   :: (a -> Bool) -> [a] -> [a]
> takeUntil _ []     = []
> takeUntil f (x:xs)
>   | not (f x) = x : takeUntil f xs
>   | f x       = [x]
```

Finally, we need the Thiele coefficients!

```
  *GUniFin> fmap head . join . fmap (sequence . filter
     isJust . map sequence . transpose) .
     thieleTrapezoid  fs $ 10007
  Just [PDiff {points = (2,2), value = 8006, basePrime =
     10007},PDiff {points = (2,3), value = 9997,
     basePrime = 10007},PDiff {points = (2,4), value =
     5337, basePrime = 10007},PDiff {points = (2,5),
     value = 50, basePrime = 10007},PDiff {points =
     (2,6), value = 0, basePrime = 10007}]
```

```
> thieleCoefficients
>   :: Graph -> Int -> Maybe [PDiff]
> thieleCoefficients fs
>   = fmap head . join
>     . fmap (sequence . filter isJust . map sequence .
  transpose)
>     . thieleTrapezoid fs
```

```
  *GUniFin> let f x = x / (1+x^2)
  *GUniFin> let fs = func2graph f [0..10]
  *GUniFin> :t thieleCoefficients fs 10007
  thieleCoefficients fs 10007 :: Maybe [PDiff]
  *GUniFin> thieleCoefficients fs 10007
  Just [PDiff {points = (2,2), value = 8006, basePrime =
     10007}
       ,PDiff {points = (2,3), value = 9997, basePrime =
          10007}
       ,PDiff {points = (2,4), value = 5337, basePrime =
          10007}
       ,PDiff {points = (2,5), value = 50, basePrime =
          10007}
       ,PDiff {points = (2,6), value = 0, basePrime =
          10007}
       ]
```

```
> thieleCoefficients' Nothing   = Nothing
```

```
> thieleCoefficients' (Just cs) = return (a:b:zipWith subZp bs as)
>   where
>     as@(a:b:bs) = cs
>
>     subZp :: PDiff -> PDiff -> PDiff
>     subZp (PDiff (x,y) v p) (PDiff (_,_) w _)
>       = PDiff (x,y) ((v-w) `mod` p) p

  *GUniFin> let f x = x / (1+x^2)
  *GUniFin> let fs = func2graph f [0..10]
  *GUniFin> thieleCoefficients fs 10007
  Just [PDiff {points = (2,2), value = 8006, basePrime = 10007}
       ,PDiff {points = (2,3), value = 9997, basePrime = 10007}
       ,PDiff {points = (2,4), value = 5337, basePrime = 10007}
       ,PDiff {points = (2,5), value = 50, basePrime = 10007}
       ,PDiff {points = (2,6), value = 0, basePrime = 10007}
       ]
  *GUniFin> thieleCoefficients' it
  Just [PDiff {points = (2,2), value = 8006, basePrime = 10007}
       ,PDiff {points = (2,3), value = 9997, basePrime = 10007}
       ,PDiff {points = (2,4), value = 7338, basePrime = 10007}
       ,PDiff {points = (2,5), value = 60, basePrime = 10007}
       ,PDiff {points = (2,6), value = 4670, basePrime = 10007}
       ]
  *GUniFin> fmap t2cZp it
  Just (([0,1,0],[1,0,1]),10007)
  *GUniFin> fmap format' it
  Just ([(Just 0,10007),(Just 1,10007),(Just 0,10007)]
       ,[(Just 1,10007),(Just 0,10007),(Just 1,10007)]
       )

  *GUniFin> fmap (format' . t2cZp) . thieleCoefficients' . thieleCoefficients fs $ 10007
  Just ([(Just 0,10007),(Just 1,10007),(Just 0,10007)],[(
```

```
     Just 1,10007),(Just 0,10007),(Just 1,10007)])

> ratCanZp'
>   :: Graph -> Int -> Maybe ([(Maybe Int, Int)], [(Maybe
     Int, Int)])
> -- ratCanZp' fs = fmap (format' . t2cZp) .
     thieleCoefficients'
> ratCanZp' fs
>   = fmap (format' . beforeFormat' . t2cZp) .
     thieleCoefficients'
>     . thieleCoefficients fs

  *GUniFin> let fivePrimes = take 5 bigPrimes
  *GUniFin> let f x = x / (1+x^2)
  *GUniFin> let fs = func2graph f [0..10]
  *GUniFin> map (ratCanZp' fs) five
  fiveFour2three  fivePrimes
  *GUniFin> map (ratCanZp' fs) fivePrimes
  [Just ([(Just 0,10007),(Just 1,10007),(Just 0,10007)
     ],[(Just 1,10007),(Just 0,10007),(Just 1,10007)]),
     Just ([(Just 0,10009),(Just 1,10009),(Just 0,10009)
     ],[(Just 1,10009),(Just 0,10009),(Just 1,10009)]),
     Just ([(Just 0,10037),(Just 1,10037),(Just 0,10037)
     ],[(Just 1,10037),(Just 0,10037),(Just 1,10037)]),
     Just ([(Just 0,10039),(Just 1,10039),(Just 0,10039)
     ],[(Just 1,10039),(Just 0,10039),(Just 1,10039)]),
     Just ([(Just 0,10061),(Just 1,10061),(Just 0,10061)
     ],[(Just 1,10061),(Just 0,10061),(Just 1,10061)])]
  *GUniFin> sequence it
  Just [([(Just 0,10007),(Just 1,10007),(Just 0,10007)
     ],[(Just 1,10007),(Just 0,10007),(Just 1,10007)])
     ,([(Just 0,10009),(Just 1,10009),(Just 0,10009)],[(
     Just 1,10009),(Just 0,10009),(Just 1,10009)]),([(
     Just 0,10037),(Just 1,10037),(Just 0,10037)],[(Just
      1,10037),(Just 0,10037),(Just 1,10037)]),([(Just
     0,10039),(Just 1,10039),(Just 0,10039)],[(Just
     1,10039),(Just 0,10039),(Just 1,10039)]),([(Just
     0,10061),(Just 1,10061),(Just 0,10061)],[(Just
     1,10061),(Just 0,10061),(Just 1,10061)])]
  *GUniFin> fmap (map fst) it
  Just [[(Just 0,10007),(Just 1,10007),(Just 0,10007)],[(
     Just 0,10009),(Just 1,10009),(Just 0,10009)],[(Just
      0,10037),(Just 1,10037),(Just 0,10037)],[(Just
     0,10039),(Just 1,10039),(Just 0,10039)],[(Just
     0,10061),(Just 1,10061),(Just 0,10061)]]
```

```
*GUniFin> fmap transpose it
Just [[(Just 0,10007),(Just 0,10009),(Just 0,10037),(Just 0,10039),(Just 0,10061)],[(Just 1,10007),(Just 1,10009),(Just 1,10037),(Just 1,10039),(Just 1,10061)],[(Just 0,10007),(Just 0,10009),(Just 0,10037),(Just 0,10039),(Just 0,10061)]]
*GUniFin> fmap (map reconstruct) it
Just [Just (0 % 1),Just (1 % 1),Just (0 % 1)]

*GUniFin> fmap (map reconstruct . transpose . map fst) . sequence . map (ratCanZp' fs) $ fivePrimes
Just [Just (0 % 1),Just (1 % 1),Just (0 % 1)]

> -- need "less data pts" error handling
> uniRatCoeffm
>   :: Graph -> (Maybe [Ratio Integer], Maybe [Ratio Integer])
> uniRatCoeffm fs = (num, den)
>   where
>     num = helper fst
>     den = helper snd
>     helper third
>       = join . fmap (mapM reconstruct . transpose . map third)
>         . mapM (ratCanZp' fs) $ bigPrimes
> --    = join . fmap (sequence . map reconstruct . transpose . map third)
> --      . sequence . map (ratCanZp' fs) $ bigPrimes

*GUniFin> let f x = x^3 / (1+x)^4
(0.01 secs, 48,440 bytes)
*GUniFin> let fs = func2graph f [0..20]
(0.02 secs, 48,472 bytes)
*GUniFin> uniRatCoeffm fs
(Just [0 % 1,0 % 1,0 % 1,1 % 1,0 % 1]
,Just [1 % 1,4 % 1,6 % 1,4 % 1,1 % 1]
)
(10.98 secs, 8,836,586,776 bytes)
*GUniFin> let f x = x^3 / (1+x)^4
*GUniFin> let fs = func2graph f [1,3..31]
*GUniFin> uniRatCoeffm fs
(Just [0 % 1,0 % 1,0 % 1,1 % 1,0 % 1]
,Just [1 % 1,4 % 1,6 % 1,4 % 1,1 % 1]
)
```

4.6 GMulFin.lhs

Listing 4.6: GMulFin.lhs

```
GMulFin.lhs

> module GMulFin where

Assume we can access
  f :: Q -> Q -> Q
of two-variable function.

> import Data.Ratio
> import Control.Monad (join)

> import GUniFin (Q, uniPolCoeff, uniRatCoeff, ratFunc2Coeff)
> import Multivariate (transposeWith)

> -- a test function
> wilFunc :: Q -> Q -> Q
> wilFunc x y = (x^2*y^2)/(1+y)^3

To track in-out correspondence, we should generalize the concept of graph:

> homogeneous
>   :: (Q -> Q -> Q) -- 2var rational function
>   -> Q
>   -> Q
>   -> (Q -> Q)      -- 1var rational function
> homogeneous f x y t = f (t*x) (t*y)

  *GMulFin> :t homogeneous wilFunc 1 2
  homogeneous wilFunc 1 2 :: Q -> Q
  *GMulFin> ratFunc2Coeff (homogeneous wilFunc 1 2)
  (Just [0 % 1,0 % 1,0 % 1,0 % 1,4 % 1],Just [1 % 1,6 % 1,12 % 1,8 % 1])
  *GMulFin> ratFunc2Coeff (homogeneous wilFunc 1 0)
  (Just [0 % 1],Just [1 % 1])
  *GMulFin> ratFunc2Coeff (homogeneous wilFunc 1 1)
  (Just [0 % 1,0 % 1,0 % 1,0 % 1,1 % 1],Just [1 % 1,3 % 1,3 % 1,1 % 1])
  *GMulFin> ratFunc2Coeff (homogeneous wilFunc 1 2)
  (Just [0 % 1,0 % 1,0 % 1,0 % 1,4 % 1],Just [1 % 1,6 %
```

```
1,12 % 1,8 % 1])
  *GMulFin> ratFunc2Coeff (homogeneous wilFunc 1 3)
  (Just [0 % 1,0 % 1,0 % 1,0 % 1,9 % 1],Just [1 % 1,9 % 1,27 % 1,27 % 1])
  *GMulFin> ratFunc2Coeff (homogeneous wilFunc 1 4)
  (Just [0 % 1,0 % 1,0 % 1,0 % 1,16 % 1],Just [1 % 1,12 % 1,48 % 1,64 % 1])

We introduce homogeneous-function, and apply univariate rational function reconstruction.

  *GMulFin> map (\y -> ratFunc2Coeff (homogeneous wilFunc 1 y)) [0,1,3,5,6,8,9,11,13]
  [(Just [0 % 1],Just [1 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,1 % 1],Just [1 % 1,3 % 1,3 % 1,1 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,9 % 1],Just [1 % 1,9 % 1,27 % 1,27 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,25 % 1],Just [1 % 1,15 % 1,75 % 1,125 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,36 % 1],Just [1 % 1,18 % 1,108 % 1,216 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,64 % 1],Just [1 % 1,24 % 1,192 % 1,512 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,81 % 1],Just [1 % 1,27 % 1,243 % 1,729 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,121 % 1],Just [1 % 1,33 % 1,363 % 1,1331 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,169 % 1],Just [1 % 1,39 % 1,507 % 1,2197 % 1])
  ]

For simplicity, take numerator only.

  *GMulFin> map fst it
  [Just [0 % 1]
  ,Just [0 % 1,0 % 1,0 % 1,0 % 1,1 % 1]
  ,Just [0 % 1,0 % 1,0 % 1,0 % 1,9 % 1]
  ,Just [0 % 1,0 % 1,0 % 1,0 % 1,25 % 1]
  ,Just [0 % 1,0 % 1,0 % 1,0 % 1,36 % 1]
  ,Just [0 % 1,0 % 1,0 % 1,0 % 1,64 % 1]
  ,Just [0 % 1,0 % 1,0 % 1,0 % 1,81 % 1]
  ,Just [0 % 1,0 % 1,0 % 1,0 % 1,121 % 1]
  ,Just [0 % 1,0 % 1,0 % 1,0 % 1,169 % 1]
  ]
```

```
  *GMulFin> sequence it
  Just [[0 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,1 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,9 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,25 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,36 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,64 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,81 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,121 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,169 % 1]
       ]

Technically, this transposeWith function is a key.

  *GMulFin> fmap (transposeWith (0%1)) it
  Just [[0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 %
      1,0 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 %
          1,0 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 %
          1,0 % 1]
       ,[0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 %
          1,0 % 1]
       ,[0 % 1,1 % 1,9 % 1,25 % 1,36 % 1,64 % 1,81 %
          1,121 % 1,169 % 1]
       ]
  *GMulFin> fmap (map (zip [0,1,3,5,6,8,9,11,13])) it
  Just [[(0,0 % 1),(1,0 % 1),(3,0 % 1),(5,0 % 1),(6,0 %
      1),(8,0 % 1),(9,0 % 1),(11,0 % 1),(13,0 % 1)]
       ,[(0,0 % 1),(1,0 % 1),(3,0 % 1),(5,0 % 1),(6,0 %
          1),(8,0 % 1),(9,0 % 1),(11,0 % 1),(13,0 % 1)]
       ,[(0,0 % 1),(1,0 % 1),(3,0 % 1),(5,0 % 1),(6,0 %
          1),(8,0 % 1),(9,0 % 1),(11,0 % 1),(13,0 % 1)]
       ,[(0,0 % 1),(1,0 % 1),(3,0 % 1),(5,0 % 1),(6,0 %
          1),(8,0 % 1),(9,0 % 1),(11,0 % 1),(13,0 % 1)]
       ,[(0,0 % 1),(1,1 % 1),(3,9 % 1),(5,25 % 1),(6,36 %
           1),(8,64 % 1),(9,81 % 1),(11,121 % 1),(13,169
           % 1)]
       ]
  *GMulFin> it :: Maybe [Graph]
  Just [[(0 % 1,0 % 1),(1 % 1,0 % 1),(3 % 1,0 % 1),(5 %
      1,0 % 1),(6 % 1,0 % 1),(8 % 1,0 % 1),(9 % 1,0 % 1)
      ,(11 % 1,0 % 1),(13 % 1,0 % 1)]
       ,[(0 % 1,0 % 1),(1 % 1,0 % 1),(3 % 1,0 % 1),(5 %
          1,0 % 1),(6 % 1,0 % 1),(8 % 1,0 % 1),(9 % 1,0
```

```
              % 1),(11 % 1,0 % 1),(13 % 1,0 % 1)]
           ,[(0 % 1,0 % 1),(1 % 1,0 % 1),(3 % 1,0 % 1),(5 % 1,0 % 1),(6 % 1,0 % 1),(8 % 1,0 % 1),(9 % 1,0 % 1),(11 % 1,0 % 1),(13 % 1,0 % 1)]
           ,[(0 % 1,0 % 1),(1 % 1,0 % 1),(3 % 1,0 % 1),(5 % 1,0 % 1),(6 % 1,0 % 1),(8 % 1,0 % 1),(9 % 1,0 % 1),(11 % 1,0 % 1),(13 % 1,0 % 1)]
           ,[(0 % 1,0 % 1),(1 % 1,1 % 1),(3 % 1,9 % 1),(5 % 1,25 % 1),(6 % 1,36 % 1),(8 % 1,64 % 1),(9 % 1,81 % 1),(11 % 1,121 % 1),(13 % 1,169 % 1)]
           ]

Then we can apply polynomial reconstruction for each "coefficient".

  *GMulFin> fmap (map uniPolCoeff) it
  Just [Just [0 % 1],Just [0 % 1],Just [0 % 1],Just [0 % 1],Just [0 % 1,0 % 1,1 % 1]]
  *GMulFin> fmap sequence it
  Just (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 % 1]])
  *GMulFin> Control.Monad.join it
  Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 % 1]]

This means that the numerator has t^4, and it clealy is x^2*y^2.

  *GMulFin> map (\t -> ratFunc2Coeff (homogeneous wilFunc 1 t)) [0,1,3,5,6,8,9,11,13]
  [(Just [0 % 1],Just [1 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,1 % 1],Just [1 % 1,3 % 1,3 % 1,1 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,9 % 1],Just [1 % 1,9 % 1,27 % 1,27 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,25 % 1],Just [1 % 1,15 % 1,75 % 1,125 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,36 % 1],Just [1 % 1,18 % 1,108 % 1,216 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,64 % 1],Just [1 % 1,24 % 1,192 % 1,512 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,81 % 1],Just [1 % 1,27 % 1,243 % 1,729 % 1])
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,121 % 1],Just [1 % 1,33 % 1,363 % 1,1331 % 1])
```

```
  ,(Just [0 % 1,0 % 1,0 % 1,0 % 1,169 % 1],Just [1 % 1,39 % 1,507 % 1,2197 % 1])
  ]
  *GMulFin> join . fmap (sequence . (map (uniPolCoeff . (zip [0,1,3,5,6,8,9,11,13]))) . (transposeWith (0%1))) . sequence . map fst $ it
  Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 % 1]]

> twoVariableRational
>   :: (Q -> Q -> Q) -- 2var function
>   -> [Q]            -- safe ys
>   -> (Maybe [[Ratio Int]], Maybe [[Ratio Int]])
> twoVariableRational f ys = (num, den)
>   where
>     num = helper fst
>     den = helper snd
>     helper third = join . fmap (mapM (uniPolCoeff . (zip ys))
>                       . transposeWith (0 % 1)) . mapM third $ gs
>     gs = map (\y -> ratFunc2Coeff (homogeneous f 1 y)) ys
>
> --  helper third = join . fmap (sequence . (map (uniPolCoeff . (zip ys)))
> --                    . (transposeWith (0%1))) . sequence . map third $ gs
>

  GMulFin.lhs:139:35: Warning: Use mapM
  Found:
    sequence . (map (uniPolCoeff . (zip ys))) . (transposeWith (0 % 1))
  Why not:
    mapM (uniPolCoeff . (zip ys)) . transposeWith (0 % 1)

  *GMulFin> twoVariableRational wilFunc [0..10]
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 % 1]]
  ,Just [[1 % 1],[0 % 1,3 % 1],[0 % 1,0 % 1,3 % 1],[0 % 1,0 % 1,0 % 1,1 % 1]]
  )
  *GMulFin> twoVariableRational wilFunc [10..20]
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 %
```

```
  1]]
  ,Just [[1 % 1],[0 % 1,3 % 1],[0 % 1,0 % 1,3 % 1],[0 % 1,0 % 1,0 % 1,1 % 1]]
  )
  *GMulFin> twoVariableRational wilFunc [1,3..21]
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 % 1]]
  ,Just [[1 % 1],[0 % 1,3 % 1],[0 % 1,0 % 1,3 % 1],[0 % 1,0 % 1,0 % 1,1 % 1]]
  )
-- wilFunc x y = (x^2*y^2)/(1+y)^3

  *GMulFin> twoVariableRational wilFunc [1,2,4,6,9,11,13]
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,0 % 1,1 % 1]]
  ,Just [[1 % 1],[0 % 1,3 % 1],[0 % 1,0 % 1,3 % 1],[0 % 1,0 % 1,0 % 1,1 % 1]]
  )
  *GMulFin> twoVariableRational (\x y -> (x^3*y)/(1 + (x-y)^2)) [0..20]
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,1 % 1]],*** Exception: newtonTriangleZp: need more evaluation
  CallStack (from HasCallStack):
    error, called at ./GUniFin.lhs:80:23 in main:GUniFin
  *GMulFin> twoVariableRational (\x y -> (x^3*y)/(1 + (x-y)^2)) [10..30]
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,1 % 1]]
  ,Just [[1 % 1],[0 % 1],[1 % 1,(-2) % 1,1 % 1],[0 % 1]]
  )

  *GMulFin> twoVariableRational (\x y -> (x^3*y)/(1 + (x-y)^2)) [1,3..9]
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,1 % 1]],*** Exception: newtonTriangleZp: need more evaluation
  CallStack (from HasCallStack):
    error, called at ./GUniFin.lhs:80:23 in main:GUniFin
  *GMulFin> twoVariableRational (\x y -> (x^3*y)/(1 + (x-y)^2)) [1,3..11]
  (Just [[0 % 1],[0 % 1],[0 % 1],[0 % 1],[0 % 1,1 % 1]]
  ,Just [[1 % 1],[0 % 1],[1 % 1,(-2) % 1,1 % 1],[0 % 1]]
  )

--
```

```
> wilFunc2 x y = (x^4*y^2)*(1+y+y^2)^2 / (1+y)^4
```