A note on functional reconstruction and finite fields

Ray D. Sameshima

2016/09/23 $\sim$ 2017/05/24 17:25

Contents

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. `http://qiita.com/bra_cat_ket/items/205c19611e21f3d422b7`
   (Japanese tech support sns)
4. The Haskell Road to Logic, Maths and Programming
   (Kees Doets, Jan van Eijck)
   `http://homepages.cwi.nl/~jve/HR/`
5. Introduction to numerical analysis
   (Stoer Josef, Bulirsch Roland)
6. A p-adic algorithm for univariate partial fractions (Paul S. Wang)

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 `>>=`.

[1] `http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html`

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".

> 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

Functional reconstruction

Here we define the problem and targets. In this chapter, the base field is that of rational numbers $\mathbb{Q}$.

1.1 Definition of functional reconstruction

1.1.1 Targets

Our targets are either polynomials over rational field

$$p : \mathbb{Q} \to \mathbb{Q}; x \mapsto \sum_{i=0}^{N(<\infty)} c_i * x^i \tag{1.1}$$

or rational functions

$$r : \mathbb{Q} \to \mathbb{Q}; x \mapsto \frac{\sum_{i=0}^{N(<\infty)} n_i * x^i}{\sum_{j=0}^{M(<\infty)} d_j * x^j}. \tag{1.2}$$

They are fully determined its canonical coefficients, say

$$(c_0, c_1, \cdots, c_N) \tag{1.3}$$

for a polynomial, and a pair of coefficients

$$(n_0, \cdots, n_N), (d_0, \cdots, d_M) \tag{1.4}$$

For simplicity, we assume that our target rational functions are safe at $x = 0$, i.e., $d_0 \neq 0$. Even if $x = 0$ is a singular, by shifting the origin we can set $d_0 \neq 0$ for new variable. Under this assumption, we determine a canonical representation for rational function

$$d_0 = 1, \tag{1.5}$$

i.e.,

$$\frac{\sum_{i=0}^{N(<\infty)} n_i * x^i}{1 + \sum_{j=1}^{M(<\infty)} d_j * x^j}. \tag{1.6}$$

Here we define our problem; functional reconstruction is a procedure to construct the coefficients representation for a function from a subset of input range. That is, given $f : \mathbb{Q} \to \mathbb{Q}$, to find the coefficients which is represented a function g and a subset $A \subset \mathbb{Q}$,

$$g : A \to \mathbb{Q} \tag{1.7}$$

with the exact coincidence on $A \subset \mathbb{Q}$

$$f|_A = g \tag{1.8}$$

with minimum "degree", where the left hand side is a restriction function on $A \subset \mathbb{Q}$. We will define this "degree" later.

1.2 Interpolations for univariate functions

Basic idea for interpolations is a finite version of differential analysis and Taylor expansion.

1.2.1 Newton interpolation for polynomials

Consider a one-variable polynomial, which has $f(x) = \sum_{i=0}^{N} c_i * x^i$ as its canonical form, but let us assume we can not access this representation directly, but we can access the in-out numbers. This N is called the degree of f.

Let

$$x_0, x_1, \cdots, x_n \tag{1.9}$$

be a set of inputs, and

$$f_i := f(x_i) \tag{1.10}$$

be the out put of the target polynomial.

Let us define finite differences, given inputs and outputs,

$$f_{1,0} := \frac{f_1 - f_0}{x_1 - x_0} \tag{1.11}$$

is a first difference, and we can define $f_{1,2}, f_{2,3}, \cdots$. Similarly, we define higher finite differences recursively:

$$f_{2,0} := \frac{f_{2,1} - f_{1,0}}{x_2 - x_0} \tag{1.12}$$

$$f_{3,0} := \frac{f_{3,1} - f_{2,0}}{x_3 - x_0} \tag{1.13}$$

$$\vdots$$

$$f_{k,0} := \frac{f_{k,1} - f_{k-1,0}}{x_k - x_0} \tag{1.14}$$

Fact

If $f(x)$ is a polynomial of degree N, then

$$\forall k > N, f_{k,0} = 0. \tag{1.15}$$

Especially,

$$f_{N,0} \neq 0, f_{N+1,0} = 0. \tag{1.16}$$

With the following $N + 1$ numbers

$$f_0, f_{1,0}, \cdots, f_{N,0} \tag{1.17}$$

the target polynomial is expressed as

$$f_0 + f_{1,0}(x - x_0) + \cdots + f_{N,0}(x - x_0) \cdots (x - x_{N-1}). \tag{1.18}$$

1.2.2 Thiele interpolation for rational function

Consider a rational function of the form $f(x) = \frac{\sum_{i=0}^{N} n_i * x^i}{1 + \sum_{j=1}^{M} d_j * x^j}$, with safe inputs

$$x_0, x_1, \cdots, x_n \tag{1.19}$$

that is we choose

$$f(x_0), \cdots, f(x_n) < \infty. \tag{1.20}$$

Let us define so called the reciprocal differences

$$\begin{aligned} \rho_{0,0} &= f_0 & (1.21)\\ \rho_{1,0} &:= \frac{x_1 - x_0}{\rho_{1,1} - \rho_{0,0}} & (1.22)\\ \rho_{k,0} &:= \frac{x_k - x_0}{\rho_{k,1} - \rho_{k-1,0}} + \rho_{k-1,1} & (1.23) \end{aligned}$$

Fact

The reciprocal differences of a certain degree T of any rational function are constant:

$$\rho_{T,0} = \rho_{T+1,1} = \rho_{T+2,2} = \cdots \tag{1.24}$$

Then the target function is expressed as a continuous fraction form:

$$a_0 + \cfrac{x - x_0}{a_1 + \cfrac{x - x_1}{a_2 + \cfrac{x - x_2}{\cfrac{\vdots}{+ \cfrac{\vdots}{a_{T-1} + \cfrac{x - x_T}{a_T}}}}}} \tag{1.25}$$

where

$$\begin{aligned} a_0 &:= \rho_{0,0} & (1.26)\\ a_1 &:= \rho_{1,0} & (1.27)\\ a_2 &:= \rho_{2,0} - \rho_{0,0} & (1.28)\\ &\vdots & \\ a_T &:= \rho_{T,0} - \rho_{T-2,0} & (1.29) \end{aligned}$$

1.2.3 Termination criteria

We put three times coincidence as our termination criteria, that is, for finite differences, if we meet

$$f_{N,0} = f_{N+1,1} = f_{N+2,2}, \tag{1.30}$$

then we take N as the degree of our target polynomial. We call

$$(f_0, f_{1,0}, \cdots, f_{N,0}) \tag{1.31}$$

with the input list $(x_0, \cdots, x_N)$ are the Newton representation for the polynomial. In similar fashion,

$$\rho_{T,0} = \rho_{T+1,1} = \rho_{T+2,2}, \tag{1.32}$$

then T is the degree of our target rational function. We call

$$(a_0, a_1, \cdots, a_T) \tag{1.33}$$

and the input list $(x_0, \cdots, x_T)$ are the Thiele representation for the polynomial.

1.3 Multivariate functions

Basically we can apply interpolation techniques for each variable, but here we introduce a systematic way.

1.3.1 An auxiliary t

Consider a function of two variables as an example, and fix (x, y). Introducing an auxiliary variable t, let us define

$$h(x, y; t) := f(tx, ty) \tag{1.34}$$

and reconstruct $h(x, y; t)$ as a univariate rational function of t:

$$h(x, y; t) = \frac{\sum_{r=0}^{R} p_r(x, y) t^r}{1 + \sum_{r'=1}^{R'} q_{r'}(x, y) t^{r'}} \tag{1.35}$$

where $p_r(x, y), q_{r'}(x, y)$ are homogeneous polynomials.

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

A simplification

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

$$p_r(1, y) \tag{1.36}$$

instead of $p_r(x, y)$, reconstruct it using multivariate Newton's method, and homogenize with x.

Chapter 2

Finite fields, quotient rings of primes

Both finite difference analysis and reciprocal difference analysis work on any field. To achieve efficiency, we project in-out relations of our target function on finite fields, then remap over $\mathbb{Q}$, this is the motivation to introduce finite fields.

2.1 Finite fields

We have assumed living knowledge on (axiomatic, i.e., ZFC) set theory and basic algebraic structures. However, in this section, we review some of algebraic structures.

2.1.1 Rings

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

$$\texttt{(+) :: R -> R -> R} \tag{2.1}$$

$$\texttt{(*) :: R -> R -> R} \tag{2.2}$$

satisfying the following 3 (ring) axioms:

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

$$\forall a, b, c \in R, (a + b) + c = a + (b + c) \quad \text{(associativity for } +\text{)} \tag{2.3}$$

$$\forall a, b, \in R, a + b = b + a \quad \text{(commutativity)} \tag{2.4}$$

$$\exists 0 \in R, \text{ s.t. } \forall a \in R, a + 0 = a \quad \text{(additive identity)} \tag{2.5}$$

$$\forall a \in R, \exists (-a) \in R \text{ s.t. } a + (-a) = 0 \quad \text{(additive inverse)} \tag{2.6}$$

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

$$\forall a, b, c \in R, (a * b) * c = a * (b * c) \quad \text{(associativity for } *) \tag{2.7}$$
$$\exists 1 \in R, \text{ s.t. } \forall a \in R, a * 1 = a = 1 * a \quad \text{(multiplicative identity)} \tag{2.8}$$

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

$$a * (b + c) = (a * b) + (a * c) \quad \text{(left distributivity)} \tag{2.9}$$
$$(a + b) * c = (a * c) + (b * c) \quad \text{(right distributivity)} \tag{2.10}$$

2.1.2 Fields

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

$$r \neq 0 \Rightarrow \exists r^{-1} \in \mathbb{K} \text{ s.t. } r * r^{-1} = 1 = r^{-1} * r. \tag{2.11}$$

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.

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

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

$$\mathbb{Z}_n := \mathbb{Z}/n\mathbb{Z} \tag{2.12}$$
$$\cong \{0, \cdots, (n-1)\} \tag{2.13}$$

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

---

[1] Here we have taken an equivalence class,

$$0 \leq \forall k \leq (n-1), [k] := \{k + n * z | z \in \mathbb{Z}\} \tag{2.14}$$

with the following operations:

$$[k] + [l] := [k + l] \tag{2.15}$$
$$[k] * [l] := [k * l] \tag{2.16}$$

This is equivalent to take modular n:

$$(k \mod n) + (l \mod n) := (k + l \mod n) \tag{2.17}$$
$$(k \mod n) * (l \mod n) := (k * l \mod n). \tag{2.18}$$

2.1.3 Bézout's lemma

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

$$a * x + b * y = \gcd(a, b), \tag{2.19}$$

where gcd is the greatest common divisor (function), see §2.1.3. We will prove this statement in §2.1.4.

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`.

$$\texttt{myGCD 7 13 == myGCD 13 7} \tag{2.20}$$

Then it hits main line:

$$\texttt{myGCD 13 7 == myGCD (13-7) 7} \tag{2.21}$$

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

$$\begin{aligned} \texttt{myGCD (13-7) 7} &\ \texttt{==}\ \texttt{myGCD 6 7} && (2.22)\\ &\ \texttt{==}\ \texttt{myGCD 7 6} && (2.23)\\ &\ \texttt{==}\ \texttt{myGCD (7-6) 6} && (2.24)\\ &\ \texttt{==}\ \texttt{myGCD 1 6} && (2.25)\\ &\ \texttt{==}\ \texttt{myGCD 6 1} && (2.26) \end{aligned}$$

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

Finally it ends with 1:

$$\texttt{myGCD 1 1 == 1} \tag{2.27}$$

As another example, consider 15 and 25:

$$\begin{align}
\texttt{myGCD 15 25} &\texttt{ == myGCD 25 15} \tag{2.28}\\
&\texttt{ == myGCD (25-15) 15} \tag{2.29}\\
&\texttt{ == myGCD 10 15} \tag{2.30}\\
&\texttt{ == myGCD 15 10} \tag{2.31}\\
&\texttt{ == myGCD (15-10) 10} \tag{2.32}\\
&\texttt{ == myGCD 5 10} \tag{2.33}\\
&\texttt{ == myGCD 10 5} \tag{2.34}\\
&\texttt{ == myGCD (10-5) 5} \tag{2.35}\\
&\texttt{ == myGCD 5 5} \tag{2.36}\\
&\texttt{ == 5} \tag{2.37}
\end{align}$$

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
```

2.1.4 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

$$\begin{aligned} (r_0, s_0, t_0) &:= (a, 1, 0) && (2.38) \\ (r_1, s_1, t_1) &:= (b, 0, 1) && (2.39) \end{aligned}$$

and inductively, for $i \geq 2$,

$$\begin{aligned} q_i &:= \operatorname{quot}(r_{i-2}, r_{i-1}) && (2.40) \\ r_i &:= r_{i-2} - q_i * r_{i-1} && (2.41) \\ s_i &:= s_{i-2} - q_i * s_{i-1} && (2.42) \\ t_i &:= t_{i-2} - q_i * t_{i-1}. && (2.43) \end{aligned}$$

The termination condition[3] is

$$r_k = 0 \tag{2.44}$$

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

$$\begin{aligned} \gcd(a, b) &= r_{k-1} && (2.45) \\ x &= s_{k-1} && (2.46) \\ y &= t_{k-1}. && (2.47) \end{aligned}$$

Proof

By definition,

$$\begin{aligned} \gcd(r_{i-1}, r_i) &= \gcd(r_{i-1}, r_{i-2} - q_i * r_{i-1}) && (2.48) \\ &= \gcd(r_{i-1}, r_{i-2}) && (2.49) \end{aligned}$$

and this implies

$$\gcd(a, b) =: \gcd(r_0, r_1) = \cdots = \gcd(r_{k-1}, 0), \tag{2.50}$$

i.e.,

$$r_{k-1} = \gcd(a, b). \tag{2.51}$$

[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.

Next, for $i = 0, 1$ observe

$$a * s_i + b * t_i = r_i. \tag{2.52}$$

Let $i \geq 2$, then

$$\begin{aligned}
r_i &= r_{i-2} - q_i * r_{i-1} && (2.53)\\
&= a * s_{i-2} + b * t_{i-2} - q_i * (a * s_{i-1} + b * t_{i-1}) && (2.54)\\
&= a * (s_{i-2} - q_i * s_{i-1}) + b * (t_{i-2} - q_i * t_{i-1}) && (2.55)\\
&=: a * s_i + b * t_i. && (2.56)
\end{aligned}$$

Therefore, inductively we get

$$\gcd(a, b) = r_{k-1} = a * s_{k-1} + b * t_{k-1}. =: a * x + b * y. \tag{2.57}$$

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:

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

```
> -- 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.

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

2.1.5 Inverses in $\mathbb{Z}_n$

For a non-zero element

$$a \in \mathbb{Z}_n, \tag{2.58}$$

there is a unique number

$$b \in \mathbb{Z}_n \text{ s.t. } ((a * b) \mod n) = 1 \tag{2.59}$$

iff a and n are coprime:

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

Proof

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

$$\exists s, t \in \mathbb{Z}, a * s + n * t = 1. \tag{2.60}$$

Therefore

$$\begin{aligned} a \text{ and } n \text{ are coprime} \quad &\Leftrightarrow \quad \exists s, t \in \mathbb{Z}, a * s + n * t = 1 &(2.61)\\ &\Leftrightarrow \quad \exists s, t' \in \mathbb{Z}, a * s = 1 + n * t'. &(2.62)\end{aligned}$$

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

$$a * s = 1 \mod n. \tag{2.63}$$

We will make a Haskell implementation in §2.1.6.
■

2.1.6 Finite field $\mathbb{Z}_p$

If p is prime, then

$$\mathbb{Z}_p := \{0, \cdots, (p-1)\} \tag{2.64}$$

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

Proof

It suffices to show that

$$\forall a \in \mathbb{Z}_p, a \neq 0 \Rightarrow \exists a^{-1} \in \mathbb{K} \text{ s.t. } a * a^{-1} = 1 = a^{-1} * a, \tag{2.65}$$

but since p is prime, and

$$\forall a \in \mathbb{Z}_p, a \neq 0 \Rightarrow \texttt{gcd a p == 1} \tag{2.66}$$

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

$$1 = 10 * 1 + 11 * (-1) \tag{2.67}$$

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

$$\begin{aligned} 10^{-1} &= -1 \mod 11 & (2.68)\\ &= 10 \mod 11 & (2.69)\end{aligned}$$

In fact,

$$10 * 10 = 11 * 9 + 1. \tag{2.70}$$

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.

[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.

```
> 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]
```

2.2 Rational number reconstruction

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

Let p be a prime. Now we have a map

$$- \quad \mathrm{mod}\ p : \mathbb{Z} \to \mathbb{Z}_p; a \mapsto (a \quad \mathrm{mod}\ p), \tag{2.71}$$

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

$$¿ : \mathbb{Z}_p \hookrightarrow \mathbb{Z}. \tag{2.73}$$

Then we can define a map

$$- \quad \mathrm{mod}\ p : \mathbb{Q} \to \mathbb{Z}_p \tag{2.74}$$

by[7]

$$q = \frac{a}{b} \mapsto (q \quad \mathrm{mod}\ p) := \left(\left(a \times ¿\left(b^{-1} \quad \mathrm{mod}\ p\right)\right) \quad \mathrm{mod}\ p\right). \tag{2.75}$$

Example and implementation

An easy implementation is the followings:[8]

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

$$\times : (\mathbb{Z}, \mathbb{Z}) \to \mathbb{Z} \tag{2.72}$$

of normal product on $\mathbb{Z}$ in eq.(2.75).

[7] This is an example of operator overloadings.

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

$$\texttt{add 1 2 == 1 `add` 2} \tag{2.76}$$

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

$$\texttt{(+) 1 2 == 1 + 2} \tag{2.77}$$

```
> -- 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
```

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
```

For example, pick 7:

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

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

$$\frac{3}{7} \in \mathbb{Q} \mapsto (3 \times ¿(7^{-1} \mod 11) \mod 11) \tag{2.78}$$
$$= (3 \times 8) \mod 11 \tag{2.79}$$
$$= 24 \mod 11 \tag{2.80}$$
$$= 2 \mod 11. \tag{2.81}$$

Haskell returns the same result

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

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

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

$$a := q \mod p \tag{2.82}$$

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.(2.52).

$$a * s_i + p * t_i = r_i \tag{2.83}$$

Therefore

$$r_i = a * s_i \mod p. \tag{2.84}$$

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

$$r_i^2, s_i^2 < p \tag{2.85}$$

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.(2.85), $r_4 = 1$ 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

$$\frac{O(10^2)}{O(10^2)}. \tag{2.86}$$

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.

We choose 3 times match as the termination condition.

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

Finally, we can check our gadgets.

What we know is a list of (q ‘modp‘ p) and prime p for several (big) primes.

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

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

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

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

For later use, let us define

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

to generate a list of images (of our target rational number) in Z_p and the base primes.

As another example, we have slightly involved function:

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

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

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

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

A problem

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

$$\frac{O(10^3)}{O(10^3)}, \tag{2.87}$$

say

```
*Ffield> let q = 895%922
*Ffield> let knownData = imagesAndPrimes q
*Ffield> take 4 knownData
[(882873,897473)
```

```
,(365035,897497)
,(705735,897499)
,(511060,897517)
]
*Ffield> map guess it
[((-854) % 123,897473)
,((-656) % 327,897497)
,((-192) % 805,897499)
,((-491) % 497,897517)
]
```

We can solve this by introducing the following theorem.

2.2.3 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

$$n \mod 3 = 2, n \mod 5 = 3, n \mod 7 = 2. \tag{2.88}$$

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

Claim

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

$$x = a_1 \mod n_1 \tag{2.89}$$

$$x = a_2 \mod n_2 \tag{2.90}$$

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

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

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

$$0 \leq a < n_1 \times n_2. \tag{2.91}$$

Proof

(existence) With §2.1.4, there are $m_1, m_2 \in \mathbb{Z}$ s.t.

$$n_1 * m_1 + n_2 * m_2 = 1. \quad (2.92)$$

Now we have

$$\begin{aligned} n_1 * m_1 &= 1 \mod n_2 \quad (2.93)\\ n_2 * m_2 &= 1 \mod n_1 \quad (2.94)\end{aligned}$$

that is[11]

$$\begin{aligned} m_1 &= n_1^{-1} \mod n_2 \quad (2.95)\\ m_2 &= n_2^{-1} \mod n_1. \quad (2.96)\end{aligned}$$

Then

$$a := a_1 * n_2 * m_2 + a_2 * n_1 * m_1 \mod (n_1 * n_2) \quad (2.97)$$

is a solution.

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

$$\begin{aligned} a - a' &= 0 \mod n_1 \quad (2.98)\\ a - a' &= 0 \mod n_2. \quad (2.99)\end{aligned}$$

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

$$a - a' = 0 \mod (n_1 * n_2). \quad (2.100)$$

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

Generalization

Given $a \in Z_n$ of pairwise coprime numbers

$$n := n_1 * \cdots * n_k, \quad (2.101)$$

a system of equations

$$a_i = a \mod n_i|_{i=1}^{k} \quad (2.102)$$

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

have a unique solution

$$a = \sum_i m_i a_i \mod n, \tag{2.103}$$

where

$$m_i = \left(\frac{n_i}{n} \mod n_i \right) \frac{n}{n_i} \Big|_{i=1}^{k}. \tag{2.104}$$

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

$$[d_0, d_1, d_2, d_3, \cdots] \tag{2.105}$$

and returns

$$[d_0, f(d_0, d_1), f(f(d_0, d_1), d_2), f(f(f(d_0, d_1), d_2), d_3), \cdots] \tag{2.106}$$

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

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

Let us check our implementation.

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

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

2.2.4 `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.
```

Chapter 3

Implementation

3.1 An expression for polynomials in Haskell

3.1.1 A polynomial as a list of coefficients: `Polynomials.hs`

We use a list of rational numbers as an expression for a polynomial. For the detail, see the reference 4, but we basically represent a univariate polynomial as its coefficients list.

Listing 3.1: Polynomials.hs

```haskell
-- 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)
```

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
```
(3.1)

```
> -- 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

$$3 * \left(1 + 2z + 3z^2 + 4z^3\right) \tag{3.2}$$

the result should be

$$3 * \left(1 + 2z + 3z^2 + 4z^3\right) = 3 + 6z + 9z^2 + 12z^3 \tag{3.3}$$

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

$$(1 + 2z) * (3 + 4z + 5z^2 + 6z^3) \tag{3.4}$$

the result should be

$$\begin{aligned}(1 + 2z) * (3 + 4z + 5z^2 + 6z^3) &= 1 * (3 + 4z + 5z^2 + 6z^3) + 2z * (3 + 4z + 5z^2 + 6z^3) \\ &= 3 + (4 + 2 * 3)z + (5 + 2 * 4)z^2 + (6 + 2 * 5)z^3 + 2 * 6z^4 \\ &= 3 + 10z + 13z^2 + 16z^3 + 12z^4 \qquad (3.5)\end{aligned}$$

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:

$$f(z) := \sum_{i=0}^{R} c_i z^i. \tag{3.6}$$

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[1]

```
*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]
```

3.2 Univariate (1 variable) case

The code is on §4.3. Here we declare a special data type.

```
> -- using record syntax
> data PDiff
>   = PDiff { points    :: (Int, Int) -- end points
```

[1] 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,

$$f(x) := x^2 + 1 \tag{3.7}$$
$$f := \lambda x.x^2 + 1 \tag{3.8}$$

are the same definition.

```
>           , value     :: Int         -- Zp value
>           , basePrime :: Int
>           }
>   deriving (Show, Read)
```

This is a hybrid data which has both $\mathbb{Z}_p$ value and two indices for finite difference analysis.

3.2.1 The flow

Both Newton and Thiele interpolation, we use the same flow. Initially, take first 3 elements, and check weather they are constants or not. If we do not have 3 coincidence, we put a new data point, and build the "triangle." Foe example, if 4th depth `[r26, r15, r04]` is not constant list:

```
[[f6,  f5,  f4,  f3,  f2,  f1,  f0  ]
,[r56, r45, r34, r23, r12, r01]
,[r46, r35, r24, r13, r02]
,[r36, r25, r14, r03]
,[r26, r15, r04]
]
```

take a new data `f7`

```
f7 [[f6,  f5,  f4,  f3,  f2,  f1,  f0]
   ,[r56, r45, r34, r23, r12, r01]
   ,[r46, r35, r24, r13, r02]
   ,[r36, r25, r14, r03]
   ,[r26, r15, r04]
   ]
```

Taking the head elements of each sublists, we can build the new heads for this new `f7`

```
[r67, r57, r37, r27]
```

Attaching this new heads, we have

```
[[f7,  f6,  f5,  f4,  f3,  f2,  f1,  f0]
,[r67, r56, r45, r34, r23, r12, r01]
,[r57, r46, r35, r24, r13, r02]
,[r47, r36, r25, r14, r03]
,[r37, r26, r15, r04]
]
```

Using last two sublists, we build new 3 elements

```
[r27, r16, r05]
```

and

```
[[f7,  f6,  f5,  f4,  f3,  f2,  f1,  f0]
,[r67, r56, r45, r34, r23, r12, r01]
,[r57, r46, r35, r24, r13, r02]
,[r47, r36, r25, r14, r03]
,[r37, r26, r15, r04]
,[r27, r16, r05]
]
```

Then we check the termination condition for this new last list.

3.3 2 variable case

The code is on §4.4. The reconstruction function has the following type:

```
> twoVariableRational
>   :: (Q -> Q -> Q) -- 2var function
>   -> [Q]           -- safe ys
>   -> (Maybe [[Ratio Int]], Maybe [[Ratio Int]])
```

It takes an unknown 2 variable function and safe y's, and returns the numerator and denominator. With this safe y's, we take

$$(1, y), y \in \text{ safe y's} \tag{3.9}$$

as the representative, that is, for a representative $(1, y)$ we apply univariate rational functional reconstruction over

$$\{\,(t, y * t)|\, t \in 0, 1, 2 \cdots\} \tag{3.10}$$

to see the evolution of coefficients.

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
  [10007,10009,10037,10039,10061,10067,10069,10079,10091,10093,10099,1010

  ,10111,10133,10139,10141,10151,10159,10163,10169,10177,10181,10193,1021

  ,10223,10243,10247,10253,10259,10267,10271,10273,10289,10301,10303,1031

  ,10321,10331,10333,10337,10343,10357,10369,10391,10399,10427,10429,1043

  ,10453,10457,10459,10463,10477,10487,10499,10501,10513,10529,10531,1055

  ,10567,10589,10597,10601,10607,10613,10627,10631,10639,10651,10657,1066

  ,10667,10687,10691,10709,10711,10723,10729,10733,10739,10753,10771,1078
```

```
  ,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 GUniFin.lhs

Listing 4.3: 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]
> --
> -- 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 % 1,1 % 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]

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]
  )
```

4.4 GMulFin.lhs

Listing 4.4: 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 % 1,1 % 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
```