

Finite fields and functional reconstructions

Ray D. Sameshima

2016/09/23 ~ 2016/10/19 09:02

Contents

0	Preface	5
0.1	References	5
0.2	Set theoretical gadgets	5
0.2.1	Numbers	5
0.2.2	Algebraic structures	6
0.3	Haskell language	6
1	Basics	9
1.1	Finite field	9
1.1.1	Rings	9
1.1.2	Fields	10
1.1.3	An example of finite rings \mathbb{Z}_n	10
1.1.4	Bézout's lemma	11
1.1.5	Greatest common divisor	11
1.1.6	Extended Euclidean algorithm	13
1.1.7	Coprime	16
1.1.8	Corollary (Inverses in \mathbb{Z}_n)	16
1.1.9	Corollary (Finite field \mathbb{Z}_p)	17
1.1.10	A map from \mathbb{Q} to \mathbb{Z}_p	19
1.1.11	Reconstruction from \mathbb{Z}_p to \mathbb{Q}	21
1.1.12	Chinese remainder theorem	24
1.2	Polynomials and rational functions	26
1.2.1	Notations	26
1.2.2	Polynomials and rational functions	27
1.2.3	As data, coefficients list	27
1.3	Haskell implementation of univariate polynomials	28
1.3.1	A polynomial as a list of coefficients	28
1.3.2	Difference analysis	32

2	Functional reconstruction	35
2.1	Univariate polynomials	35
2.1.1	Newton's polynomial representation	35
2.1.2	Towards canonical representations	36
2.1.3	Simplification of our problem	36
2.2	Univariate polynomial reconstruction with Haskell	38
2.2.1	Newton interpolation formula with Haskell	38
2.2.2	Stirling numbers of the first kind	40
2.2.3	<code>list2pol</code> : from output list to canonical coefficients	42
2.3	Univariate rational functions	43
2.3.1	Thiele's interpolation formula	43
2.3.2	Towards canonical representations	44
2.4	Univariate rational function reconstruction with Haskell	45
2.4.1	Reciprocal difference	45
2.4.2	Example	46
2.4.3	Another example	47
2.4.4	Haskell representation for rational functions	49
2.4.5	<code>lists2rat</code> : from output lists to canonical coefficients	52
2.5	Multivariate polynomials	53
2.5.1	Foldings as recursive applications	53
2.5.2	Experiments, 2 variables case	53
2.6	Multivariate rational functions	56
2.6.1	The canonical normalization	56
2.6.2	An auxiliary t	57
2.6.3	Experiments, 2 variables case	58
3	Codes	61
3.1	<code>Ffield.lhs</code>	61
3.2	<code>Polynomials.hs</code>	67
3.3	<code>Univariate.lhs</code>	68
3.4	<code>Multivariate.lhs</code>	77

Chapter 0

Preface

0.1 References

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

0.2 Set theoretical gadgets

0.2.1 Numbers

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

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

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

0.2.2 Algebraic structures

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

0.3 Haskell language

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

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



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

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

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

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

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

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

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

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

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

Chapter 1

Basics

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

1.1 Finite field

Ffield.lhs

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

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

1.1.1 Rings

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

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

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

satisfying the following 3 (ring) axioms:

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

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

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

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

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

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

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

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

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

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

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

1.1.2 Fields

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

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

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

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

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

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

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

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

¹ Here we have taken an equivalence class,

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

1.1.4 Bézout's lemma

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

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

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

1.1.5 Greatest common divisor

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

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

Example, by hands

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

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

Then it hits main line:

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

with the following operations:

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

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

This is equivalent to take modular n :

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

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

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

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

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

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

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

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

Finally it ends with 1:

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

As another example, consider 15 and 25:

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

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

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

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

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

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

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

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

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

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

Example, with Haskell

Let us check simple example using Haskell:

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

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

The final result is from

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

1.1.6 Extended Euclidean algorithm

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

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

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

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

and inductively, for $i \geq 2$,

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

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

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

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

The termination condition³ is

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

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

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

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

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

Proof

By definition,

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

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

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

and this implies

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

i.e.,

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

Next, for $i = 0, 1$ observe

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

Let $i \geq 2$, then

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

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

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

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

Therefore, inductively we get

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

This prove Bézout's lemma.

■

Haskell implementation

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

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

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

```
> exGCD' :: Integral n => n -> n -> ([n], [n], [n], [n])
> exGCD' a b = (qs, rs, ss, ts)
>   where
>     qs = zipWith quot rs (tail rs)
>     rs = 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 _ [] = []
> takeBefore p (l:ls)
>   | p l      = []
>   | otherwise = l : (takeBefore p ls)

```

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

```

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

```

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

```

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

```

It works, and we have other simpler examples:

```

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

```

Now what we should do is extract gcd of a and b , and (s, t) 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 a b = (g, x, y)
>   where
>     (_,r,s,t) = exGCD' a b
>     g = last r
>     x = last . init $ s
>     y = last . init $ t

```

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

```

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

```

1.1.7 Coprime

Let us define a binary relation as follows:

```

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

```

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

For a non-zero element

$$a \in \mathbb{Z}_n, \quad (1.58)$$

there is a unique number

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

iff a and n are coprime.

Proof

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

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

Therefore

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

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

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

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

■

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

If p is prime, then

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

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

Proof

It suffices to show that

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

but since p is prime, and

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

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

■

Example and implementation

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

Example $\mathbb{Z}_{\{11\}}$

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

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

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

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

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

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

In fact,

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

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

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

Let us generalize these flow into a function⁵:

```
> inverses :: Integral a => a -> Maybe [(a,a)]
> inverses n
>   | isPrime n = Just lst -- isPrime n
>   | otherwise = Nothing
>   where
>     lst' = map (('mod' n) . (\(_,_,c)->c) . exGcd n) [1..(n-1)]
>     lst = zip [1..] lst'
```

The function `inverses` returns a list of nonzero number with their inverses if p is 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` 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`.

Now we define `inversep'`⁶ whose 1st input is the base p of our ring(field) and 2nd input is an element in \mathbb{Z}_p .

```
> inversep' :: Int -> Int -> Maybe Int
> inversep' p a = do
>   l <- inverses p
>   let a' = (a `mod` p)
>   return (snd $ l !! (a'-1))

*Ffield> inverses' 11
Just [(1,1),(2,6),(3,4),(4,3),(5,9),(6,2),(7,8),(8,7),(9,5),(10,10)]
```

However, this is not efficient, and we refactor it as follows:⁷

```
> inversep :: Integral a => a -> a -> Maybe a
> inversep p a = let (_,x,y) = exGcd p a in
>   if isPrime p then Just (y `mod` p)
>   else Nothing
```

```
map (inversep' 10007) [1..10006]
(12.99 secs, 17,194,752,504 bytes)
map (inversep 10007) [1..10006]
(1.74 secs, 771,586,416 bytes)
```

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

Let p be a prime. Now we have a map

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

and a natural inclusion (or a forgetful map)⁸

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

⁶ Here we have used `do`-notation, a syntactic sugar for use with monadic expressions. From <https://wiki.haskell.org/Monad>:

Monads in Haskell can be thought of as composable computation descriptions.

⁷ Note that, here we use our Haskell code as a script, and we have not compile it. Hopefully after compile our code, it become much faster.

⁸ By introducing this forgetful map, we can use

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

of normal product on \mathbb{Z} .

Then we can define a map

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

by⁹

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

Example and implementation

An easy implementation is the followings:¹⁰

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

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

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

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

Consider $(3 \% 7)$.

```
*Ffield Data.Ratio> let q = 3 % 7
*Ffield Data.Ratio> 3 'mod' 11
3
*Ffield Data.Ratio> 7 'mod' 11
7
*Ffield Data.Ratio> inverses 11
Just [(1,1),(2,6),(3,4),(4,3),(5,9),(6,2),(7,8),(8,7),(9,5),(10,10)]
```

For example, pick 7:

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

⁹ This is an example of operator overloadings.

¹⁰ The backquotes makes any binary function infix operator. For example,

$$\text{add } a \ b == a \text{ 'add' } b \quad (1.76)$$

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

$$\frac{3}{7} \in \mathbb{Q} \mapsto (3 \times (7^{-1} \bmod 11) \bmod 11) \quad (1.77)$$

$$= (3 \times 8) \bmod 11 \quad (1.78)$$

$$= 24 \bmod 11 \quad (1.79)$$

$$= 2 \bmod 11. \quad (1.80)$$

Haskell returns the same result

```
*Ffield Data.Ratio> q `modp` 11
2
```

and consistent.

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

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

$$a := q \bmod p \quad (1.81)$$

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

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

$$a * s_i + p * t_i = r_i \quad (1.82)$$

Therefore

$$r_i = a * s_i \bmod p \Leftrightarrow \frac{r_i}{s_i} \bmod p = a. \quad (1.83)$$

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

$$r_i^2, s_i^2 < p \quad (1.84)$$

as the termination condition for this reconstruction.

Haskell implementation

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

```

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

```

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

```

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

```

Look at the first hit of termination condition eq.(1.84), $r_4 = 1$ and $s_4 = 3$. They give us the same guess $\frac{1}{3}$, and that the reconstructed number.

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

```

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

```

We have put a list of big primes as follows.

```

> -- Hard code of big primes.
> bigPrimes :: [Int]
> bigPrimes = dropWhile (< 897473) $ takeWhile (<978948) primes

```

We choose 3 times match as the termination condition.

```

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

```

Finally, we can check our gadgets.

What we know is a list of $(q \bmod p)$ and prime p for several (big) primes.

```
*Ffield> let q = 10%19
*Ffield> let knownData = zip (map (modp q) bigPrimes) bigPrimes
*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
```

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)

```

1.1.12 Chinese remainder theorem

From wikipedia¹¹

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

Here is a solution with Haskell:

```

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

```

We define an infinite list of natural numbers that satisfy

$$n \bmod 3 = 2, n \bmod 5 = 3, n \bmod 7 = 2. \quad (1.85)$$

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

¹¹ https://en.wikipedia.org/wiki/Chinese_remainder_theorem

Claim

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

$$x = a_1 \pmod{n_1} \quad (1.86)$$

$$x = a_2 \pmod{n_2} \quad (1.87)$$

have a unique solution modular $n_1 * n_2$ ¹².

Proof

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

$$n_1 * m_1 + n_2 * m_2 = 1. \quad (1.89)$$

Now we have

$$n_1 * m_1 = 1 \pmod{n_2} \quad (1.90)$$

$$n_2 * m_2 = 1 \pmod{n_1} \quad (1.91)$$

that is

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

$$m_2 = n_2^{-1} \pmod{n_1}. \quad (1.93)$$

Then

$$a := a_1 * n_2 * m_2 + a_2 * n_1 * m_1 \pmod{(n_1 * n_2)} \quad (1.94)$$

is a solution.

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

$$a - a' = 0 \pmod{n_1} \quad (1.95)$$

$$a - a' = 0 \pmod{n_2}. \quad (1.96)$$

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

$$a - a' = 0 \pmod{(n_1 * n_2)}. \quad (1.97)$$

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

■

¹² Note that, this is equivalent that there is a unique solution a in

$$0 \leq a < n_1 \times n_2. \quad (1.88)$$

Generalization

Given $a \in Z_n$ of pairwise coprime numbers

$$n := n_1 * \cdots * n_k, \quad (1.98)$$

a system of equations

$$a_i = a \pmod{n_i} \quad (1.99)$$

have a unique solution

$$a = \sum_i m_i a_i \pmod{n}, \quad (1.100)$$

where

$$m_i = \left(\frac{n_i}{n} \pmod{n_i} \right) \frac{n}{n_i} \Big|_{i=1}^k. \quad (1.101)$$

1.2 Polynomials and rational functions

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

1.2.1 Notations

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

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

A monomial is defined as

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

The total degree of this monomial is given by

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

1.2.2 Polynomials and rational functions

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

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

where

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

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

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

We denote

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

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

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

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

We denote

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

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

1.2.3 As data, coefficients list

We can identify a polynomial

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

as a set of coefficients

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

Similarly, for a rational function, we can identify

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

as an ordered pair of coefficients

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

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

1.3 Haskell implementation of univariate polynomials

Here we basically follow some part of §9 of ref.3, and its addendum¹³.

Univariate.lhs

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

1.3.1 A polynomial as a list of coefficients

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

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

```
module Polynomials where
```

```
default (Integer, Rational, Double)
```

```
-- polynomials, as coefficients lists
```

¹³ See <http://homepages.cwi.nl/~jve/HR/PolAddendum.pdf>

1.3. HASKELL IMPLEMENTATION OF UNIVARIATE POLYNOMIALS29

```
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

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

```

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

```

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

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

the result should be

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

In Haskell

```

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

```

and this is exactly same as map with section:

```

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

```

When we multiply two polynomials, say

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

1.3. HASKELL IMPLEMENTATION OF UNIVARIATE POLYNOMIALS31

the result should be

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

In Haskell,

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

Now the (dummy) variable is given as

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

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

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

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

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

This gives us¹⁴

```
*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]
```

¹⁴ Here we have used lambda, or so called anonymous function. From <http://learnyouahaskell.com/higher-order-functions>

To make a lambda, we write a λ (because it kind of looks like the greek letter lambda if you squint hard enough) and then we write the parameters, separated by spaces.

For example,

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

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

are the same definition.

1.3.2 Difference analysis

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

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

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

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

For example

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

Let us consider the difference sequence

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

Its Haskell version is

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

This gives

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

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

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

and

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

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

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

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

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

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

This guarantees the following function will terminate in finite steps¹⁵; `difLists` keeps generating difference lists until the difference get constant.

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

Let us try:

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

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

```
> degree' :: (Integral n) => [n] -> Int
> degree' xs = length (difLists [xs]) -1
```

For example,

¹⁵ If a given lists is generated by a polynomial.

```

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

```

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

```

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

```

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

```

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

```

Chapter 2

Functional reconstruction

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

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

2.1 Univariate polynomials

2.1.1 Newtons' polynomial representation

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

$$f_0(z) = a_0 \quad (2.1)$$

$$f_1(z) = a_0 + (z - y_0)a_1 \quad (2.2)$$

$$\vdots$$

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

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

where

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

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

$$\vdots$$

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

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

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

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

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

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

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

2.1.2 Towards canonical representations

Once we get the Newton's representation

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

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

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

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

2.1.3 Simplification of our problem

Without loss of generality, we can put

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

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

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

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

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

Then we have slightly simpler forms of coefficients:

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

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

$$a_1 = f(1) - a_0 \quad (2.17)$$

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

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

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

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

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

\vdots

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

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

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

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

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

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

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

Under these settings, we have

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

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

where we have assume

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

Example

Consider a polynomial

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

and its out put list

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

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

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

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

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

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

so we get

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

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

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

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

2.2 Univariate polynomial reconstruction with Haskell

2.2.1 Newton interpolation formula with Haskell

First, the falling power is naturally given by recursively:

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

2.2. UNIVARIATE POLYNOMIAL RECONSTRUCTION WITH HASKELL39

Assume the differences are given in a list

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

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

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

Consider a polynomial

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

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

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

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

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

The list of first differences, i.e.,

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

can be computed as follows:

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

Mapping a list of integers to a Newton representation:

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

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

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

2.2.2 Stirling numbers of the first kind

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

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

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

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

and

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

From the highest order, x^n , we get

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

We also put

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

and

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

2.2. UNIVARIATE POLYNOMIAL RECONSTRUCTION WITH HASKELL41

The key equation is

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

and we get

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

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

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

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

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

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

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

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

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

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

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

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

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

Here is an implementation, first the Stirling numbers:

```

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

```

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

```

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

```

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

```

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

```

2.2.3 list2pol: from output list to canonical coefficients

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

```

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

```

Here are some checks on these functions:

Reconstruction as curve fitting

```

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

```

```

*NewtonInterpolation> list2pol [0,1,5,14,30]
[0 % 1,1 % 6,1 % 2,1 % 3]

```

```

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

```

```

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

```

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

2.3 Univariate rational functions

We use the same notion, i.e., what we can know is the output-list of a univariate rational function, say `f :: Int -> Ratio Int`:

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

2.3.1 Thiele's interpolation formula

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

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

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

$$\vdots$$

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

where

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

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

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

$$\begin{aligned} & \vdots \\ a_r &= \frac{1}{\frac{2}{\frac{3}{\frac{\vdots}{r} - a_2} - a_{r-1}} - a_{r-2}} \end{aligned} \quad (2.67)$$

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

2.3.2 Towards canonical representations

In order to get a unique representation of canonical form

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

we put

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

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

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

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

2.4 Univariate rational function reconstruction with Haskell

Here we the same notion of

```
https://rosettacode.org/wiki/Thiele%27s_interpolation_
formula
```

and especially

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

2.4.1 Reciprocal difference

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

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

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

where ρ is so called the reciprocal difference:

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

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

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

These preparation helps us to write the following codes:

Thiele's interpolation formula

Reciprocal difference rho, using the same notation of

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

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

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

```

> num = numerator next
> den = denominator next
> next = (rho fs (n-1) (i+1)) - (rho fs (n-1) i)

```

Note that (%) has the following type,

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

```

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

```

2.4.2 Example

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

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

The function is now

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

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

Using Maxima³, we can verify this:

```

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

```

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

```

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

```

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

³ <http://maxima.sourceforge.net>

2.4. UNIVARIATE RATIONAL FUNCTION RECONSTRUCTION WITH HASKELL47

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

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

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

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

2.4.3 Another example

From the equation (3.26) of ref.1,

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

So we get the Thiele coefficients

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

Plug these in the continued fraction, and simplify with Maxima

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

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

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

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

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

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

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

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


2.4.4 Haskell representation for rational functions

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

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

Here is a translator from coefficients lists to rational function.

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

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

Simply taking numerator and denominator polynomials.

The following `canonicalizer` reduces the tuple-rep of rational function in canonical form, i.e., the coefficient of the lowest degree term of the denominator to be 1^4 .

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

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

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

```

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

```

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

Before we go to a general case, consider

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

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

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

$$= 1 + \frac{x}{2 + \frac{x-1}{\frac{x+10}{4}}} \quad (2.84)$$

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

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

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

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

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

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

In general, we have the following Thiele representation:

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

The base case should be

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

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

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

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

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

where

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

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

Thus, the implementation is the followings.

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

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

From the first example,

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

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

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

2.4.5 lists2rat: from output lists to canonical coefficients

Finally, we get

```
> lists2rat :: (Integral a) => [Ratio a] -> ([Ratio a], [Ratio a])
> lists2rat = thiele2Coef . thieleC
```

as the reconstruction function from the output sequence.

```
*Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
*Univariate> lists2rat $ map h [0..]
([3 % 1,6 % 1,18 % 1],[1 % 1,2 % 1,20 % 1])
```

2.5 Multivariate polynomials

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

`Multivariate.lhs`

```
> module Multivariate
>   where

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

2.5.1 Foldings as recursive applications

Consider an arbitrary multivariate polynomial

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

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

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

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

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

The terminate condition should be the univariate case.

2.5.2 Experiments, 2 variables case

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

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

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

```

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

```

Fine, so the canonical form can be

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

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

```

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

```

That is

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

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

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

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

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

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

```

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

```

i.e.,

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

However, for $c_1(z)$

```

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

```

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

```

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

```

Thus we have

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

and these fully determine our polynomial:

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

As another experiment, take the denominator.

```

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

```

So the canonical form should be

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

Let us look at these coefficient polynomial:

```

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

```

So we get

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

and

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

Using index operator (!!),

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

Finally we get

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

and

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

2.6 Multivariate rational functions

2.6.1 The canonical normalization

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

$$\frac{\sum_\alpha n_\alpha z^\alpha}{\sum_\beta d_\beta z^\beta} \quad (2.112)$$

A canonical choice is

$$d_0 = d_{(0, \dots, 0)} = 1. \quad (2.113)$$

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

$$d'_0 = d_s \neq 0. \quad (2.114)$$

2.6.2 An auxiliary t

Introducing an auxiliary variable t , let us define

$$h(t, z) := f(tz_1, \dots, tz_n), \quad (2.115)$$

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

$$h(t, z) = \frac{\sum_{r=0}^R p_r(z) t^r}{1 + \sum_{r'=1}^{R'} q_{r'}(z) t^{r'}} \quad (2.116)$$

where

$$p_r(z) = \sum_{|\alpha|=r} n_\alpha z^\alpha \quad (2.117)$$

$$q_{r'}(z) = \sum_{|\beta|=r'} n_\beta z^\beta \quad (2.118)$$

are homogeneous polynomials.

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

A simplification

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

$$p_r(1, z_2, \dots, z_n) \quad (2.119)$$

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

2.6.3 Experiments, 2 variables case

Consider the equation (3.23) in ref.1.

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

*Multivariate> :t f
f :: Integral a => a -> a -> Ratio a
*Multivariate> let h x y t = f (t*x) (t*y)
*Multivariate> let hs x y = map (h x y) [0..]
*Multivariate> take 5 $ hs 0 0
[3 % 1,3 % 1,3 % 1,3 % 1,3 % 1]
*Multivariate> take 5 $ hs 0 1
[3 % 1,13 % 18,35 % 53,69 % 106,115 % 177]
*Multivariate> take 5 $ hs 1 0
[3 % 1,2 % 3,7 % 11,9 % 14,41 % 63]
*Multivariate> take 5 $ hs 1 1
[3 % 1,3 % 4,29 % 37,183 % 226,105 % 127]
```

Here we have introduced the auxiliary t as third argument.

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

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

So we have

$$h(1, 0, t) = \frac{3 + 2t + 7t^2}{1 + 7t + 10t^2} \quad (2.120)$$

$$h(1, 1, t) = \frac{3 + 6t + 18t^2}{1 + 15t + 20t^2} \quad (2.121)$$

$$h(1, 2, t) = \frac{3 + 10t + 41t^2}{1 + 23t + 48t^2} \quad (2.122)$$

$$h(1, 3, t) = \frac{3 + 14t + 76t^2}{1 + 31t + 94t^2} \quad (2.123)$$

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

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

Let us consider numerator first. This `list` is Haskell representation for eq.(2.120), eq.(2.121), eq.(2.122) and eq.(2.123).

```
*Multivariate> let list = map (lists2rat . (hs 1)) [0..4]
*Multivariate> let numf = map fst list
*Multivariate> list
[[3 % 1,2 % 1,7 % 1],[1 % 1,7 % 1,10 % 1])
,[3 % 1,6 % 1,18 % 1],[1 % 1,15 % 1,20 % 1])
,[3 % 1,10 % 1,41 % 1],[1 % 1,23 % 1,48 % 1])
,[3 % 1,14 % 1,76 % 1],[1 % 1,31 % 1,94 % 1])
,[3 % 1,18 % 1,123 % 1],[1 % 1,39 % 1,158 % 1])
]
*Multivariate> numf
[[3 % 1,2 % 1,7 % 1]
,[3 % 1,6 % 1,18 % 1]
,[3 % 1,10 % 1,41 % 1]
,[3 % 1,14 % 1,76 % 1]
,[3 % 1,18 % 1,123 % 1]
]
```

From this information, we reconstruct each polynomials

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

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

```
*Multivariate> let denf = map snd list
*Multivariate> denf
[[1 % 1,7 % 1,10 % 1]
,[1 % 1,15 % 1,20 % 1]
,[1 % 1,23 % 1,48 % 1]
,[1 % 1,31 % 1,94 % 1]
,[1 % 1,39 % 1,158 % 1]
```

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

```

]
*Multivariate> list2pol $ map head denf
[1 % 1]
*Multivariate> list2pol $ map (head . tail) denf
[7 % 1,8 % 1]
*Multivariate> list2pol $ map last denf
[10 % 1,1 % 1,9 % 1]

```

So we get

$$h(1, y, t) = \frac{3 + (2 + 4y)t + (7 + 5y + 6y^2)t^2}{1 + (7 + 8y)t + (10 + y + 9y^2)t^2} \quad (2.124)$$

Finally, we use the homogeneous property for each powers:

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

Putting $t = 1$, we get

$$f(x, y) = h(x, y, 1) \quad (2.126)$$

$$= \frac{3 + (2x + 4y) + (7x^2 + 5xy + 6y^2)}{1 + (7x + 8y) + (10x^2 + xy + 9y^2)} \quad (2.127)$$

Chapter 3

Codes

3.1 Ffield.lhs

Listing 3.1: Ffield.lhs

```
1 Ffield.lhs
2
3 https://arxiv.org/pdf/1608.01902.pdf
4
5 > module Ffield where
6
7 > import Data.Ratio
8 > import Data.Maybe
9 > import Data.Numbers.Primes
10
11 > coprime :: Integral a => a -> a -> Bool
12 > coprime a b = gcd a b == 1
13
14 Consider a finite ring
15   Z_n := [0..(n-1)]
16
17 > haveInverse :: Integral a => a -> [Bool]
18 > haveInverse n = map (coprime n) [0..(n-1)]
19
20 *Ffield> haveInverse 8
21 [False,True,False,True,False,True,False,True]
22 *Ffield> zip [0..] $ haveInverse 8
23 [(0,False),(1,True),(2,False),(3,True),(4,False),(5,
    True),(6,False),(7,True)]
24
```

```

25 If any non-zero element has its multiplication inverse,
    then the ring is a field:
26
27 > isField' :: Integral a => a -> Bool
28 > isField' n = and $ tail $ haveInverse n
29
30 Or more efficiently,
31
32 > isField :: Integral a => a -> Bool
33 > isField = isPrime
34
35 zip [2..] $ map isField [2..13]
36 [(2,True),(3,True),(4,False),(5,True),(6,False),(7,True),
    (8,False),(9,False),(10,False),(11,True),(12,
    False),(13,True)]
37
38 Here we would like to implement the extended Euclidean
    algorithm.
39 See the algorithm, examples, and pseudo code at:
40
41 https://en.wikipedia.org/wiki/
    Extended_Euclidean_algorithm
42
43 I've asked at Qiita and get some solutions:
44
45 http://qiita.com/bra\_cat\_ket/items/205c19611e21f3d422b7
46
47 > exGCD' :: (Integral n) => n -> n -> ([n], [n], [n], [n]
    ])
48 > exGCD' a b = (qs, rs, ss, ts)
49 >   where
50 >     qs = zipWith quot rs (tail rs)
51 >     rs = takeUntil (==0) r'
52 >     r' = steps a b
53 >     ss = steps 1 0
54 >     ts = steps 0 1
55 >     steps a b = rr
56 >     where
57 >       rr@(_:rs) = a:b: zipWith (-) rr (zipWith (*) qs
    rs)
58 >
59 > takeUntil :: (a -> Bool) -> [a] -> [a]
60 > takeUntil p = foldr func []
61 >   where
62 >     func x xs

```

```

63 > | p x = []
64 > | otherwise = x : xs
65
66 This example is from wikipedia:
67
68 *Ffield> exGCD' 240 46
69 ([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])
70
71 *Ffield> gcd 240 46
72 2
73
74 *Ffield> 240*(-9) + 46*(47)
75 2
76
77 > -- a*x + b*y = gcd a b
78 > exGcd :: Integral t => t -> t -> (t, t, t)
79 > exGcd a b = (g, x, y)
80 > where
81 >   (_,r,s,t) = exGCD' a b
82 >   g = last r
83 >   x = last . init $ s
84 >   y = last . init $ t
85
86 *Ffield> exGcd 46 240
87 (2,47,-9)
88
89 *Ffield> 46*47 + 240*(-9)
90 2
91
92 Example Z_{11}
93
94 *Ffield> isField 11
95 True
96
97 *Ffield> map (exGcd 11) [0..10]
98 [(11,1,0),(1,0,1),(1,1,-5),(1,-1,4),(1,-1,3),(1,1,-2),
99   (1,-1,2),(1,2,-3),(1,3,-4),(1,-4,5),(1,1,-1)]
100
101 *Ffield> map (('mod' 11) . \(_,_,x)->x) . exGcd 11)
102 [1..10]
103 [1,6,4,3,9,2,8,7,5,10]
104
105 *Ffield> zip [1..10] it
106 [(1,1),(2,6),(3,4),(4,3),(5,9),(6,2),(7,8),(8,7),(9,5),
107   (10,10)]
108
109 > inverses :: Integral a => a -> Maybe [(a,a)]

```

```

104 > inverses n
105 >   | isPrime n = Just lst -- isPrime n
106 >   | otherwise = Nothing
107 >   where
108 >     lst' = map (('mod' n) . (\(_,_,c)->c) . exGcd n)
109 >           [1..(n-1)]
110 >     lst = zip [1..] lst'
111 >
112 > inversep :: Integral a => a -> a -> Maybe a
113 > inversep p a = let (_,x,y) = exGcd p a in
114 >   if isPrime p then Just (y 'mod' p)
115 >   else Nothing
116
117 map (inversep 10007) [1..10006]
118 (1.74 secs, 771,586,416 bytes)
119
120 A map from Q to Z_p.
121
122 > -- p should be prime.
123 > modp :: Integral a => Ratio a -> a -> a
124 > q 'modp' p = (a * (bi 'mod' p)) 'mod' p
125 >   where
126 >     (a,b) = (numerator q, denominator q)
127 >     bi = fromJust $ inversep p b
128
129 Example: on Z_{11}
130 Consider (3 % 7).
131
132 *Ffield Data.Ratio> let q = 3 % 7
133 *Ffield Data.Ratio> 3 'mod' 11
134 3
135 *Ffield Data.Ratio> 7 'mod' 11
136 7
137 *Ffield Data.Ratio> inverses 11
138 Just [(1,1),(2,6),(3,4),(4,3),(5,9),(6,2),(7,8),(8,7)
139       ,(9,5),(10,10)]
140
141 *Ffield Data.Ratio> 7*8 == 11*5+1
142 True
143
144 on Z_{11}, (7^{-1} 'mod' 11) is equal to (8 'mod' 11) and
145 (3%7) /-> (3 * (7^{-1} 'mod' 11) 'mod' 11)
146 == (3*8 'mod' 11)
147 == 2 'mod' 11
148
149 *Ffield Data.Ratio> modp q 11

```



```

147 2
148
149 Example: on Z_{5}
150 *Ffield Data.Ratio> 3 'mod' 5
151 3
152 *Ffield Data.Ratio> 7 'mod' 5
153 2
154 *Ffield Data.Ratio> inverses 5
155 Just [(1,1),(2,3),(3,2),(4,4)]
156 *Ffield Data.Ratio> modp q 5
157 4
158
159 Reconstruction Z_p -> Q
160 *Ffield> let q = (1%3)
161 *Ffield> take 3 $ dropWhile (<100) primes
162 [101,103,107]
163 *Ffield> q 'modp' 101
164 34
165 *Ffield> let rec x = exGCD' (q 'modp' x) x
166 *Ffield> rec 101
167 ([0,2,1,33],[34,101,34,33,1],[1,0,1,-2,3,-101],[0,1,0,1,-1,34])

168 *Ffield> rec 103
169 ([0,1,2,34],[69,103,69,34,1],[1,0,1,-1,3,-103],[0,1,0,1,-2,69])

170 *Ffield> rec 107
171 ([0,2,1,35],[36,107,36,35,1],[1,0,1,-2,3,-107],[0,1,0,1,-1,36])

172
173 > guess :: Integral t =>
174 >     (t, t) -- (q 'modp' p, p)
175 >     -> (Ratio t, t)
176 > guess (a, p) = let (_,rs,ss,_) = exGCD' a p in
177 >     (select rs ss p, p)
178 >     where
179 >         select :: Integral t => [t] -> [t] -> t -> Ratio
180 >         t
181 >         select [] _ _ = 0%1
182 >         select (r:rs) (s:ss) p
183 >         | s /= 0 && r^2 <= p && s^2 <= p = r% s
184 >         | otherwise = select rs ss p
185 > -- Hard code of big primes.
186 > bigPrimes :: [Int]
187 > bigPrimes = dropWhile (< 897473) $ takeWhile (< 978948)

```

```

    primes
188 >
189 > matches3 :: Eq a => [a] -> a
190 > matches3 (a:bb@(b:c:cs))
191 >   / a == b && b == c = a
192 >   / otherwise      = matches3 bb
193
194 What we know is a list of (q 'modp' p) and prime p.
195
196 *Ffield> let q = 10%19
197 *Ffield> let knownData = zip (map (modp q) bigPrimes)
    bigPrimes
198 *Ffield> matches3 $ map (fst . guess) knownData
199 10 % 19
200
201 > reconstruct :: Integral a =>
202 >               [(a, a)] -- :: [(Z_p, primes)]
203 >               -> Ratio a
204 > reconstruct aps = matches3 $ map (fst . guess) aps
205
206 Here is a naive test:
207 > let qs = [1 % 3, 10 % 19, 41 % 17, 30 % 311, 311 % 32, 869
    % 232, 778 % 123, 331 % 739]
208 > let func q = zip (map (modp q) bigPrimes) bigPrimes
209 > let longList = map func qs
210 > map reconstruct longList
211 [1 % 3, 10 % 19, 41 % 17, 30 % 311, 311 % 32, 869 % 232, 778
    % 123, 331 % 739]
212 > it == qs
213 True
214
215 > matches3' :: Eq a => [(a, t)] -> (a, t)
216 > matches3' (a0@(a,_):bb@((b,_):(c,_):cs))
217 >   / a == b && b == c = a0
218 >   / otherwise      = matches3' bb
219
220 *Ffield> let q = (331%739)
221 (0.01 secs, 44,024 bytes)
222 *Ffield> let smallerprimes = dropWhile (<100) $
    takeWhile (<978948) primes
223 (0.01 secs, 39,968 bytes)
224 *Ffield> let knownData = zip (map (modp q)
    smallerprimes) smallerprimes
225 (0.01 secs, 39,872 bytes)
226 *Ffield> matches3' $ map guess knownData

```

```

227      (331 % 739,614693)
228      (17.64 secs, 12,402,878,080 bytes)

```

3.2 Polynomials.hs

Listing 3.2: Polynomials.hs

```

1  -- Polynomials.hs
2  -- http://homepages.cwi.nl/~jve/rcrh/Polynomials.hs
3
4  module Polynomials where
5
6  default (Integer, Rational, Double)
7
8  -- scalar multiplication
9  infixl 7 .*
10 (.*) :: Num a => a -> [a] -> [a]
11 c .* []      = []
12 c .* (f:fs) = c*f : c .* fs
13
14 z :: Num a => [a]
15 z = [0,1]
16
17 -- polynomials, as coefficients lists
18 instance (Num a, Ord a) => Num [a] where
19     fromInteger c = [fromInteger c]
20     -- operator overloading
21     negate []      = []
22     negate (f:fs) = (negate f) : (negate fs)
23
24     signum [] = []
25     signum gs
26       | signum (last gs) < (fromInteger 0) = negate z
27       | otherwise = z
28
29     abs [] = []
30     abs gs
31       | signum gs == z = gs
32       | otherwise      = negate gs
33
34     fs      + []      = fs
35     []      + gs      = gs
36     (f:fs) + (g:gs) = f+g : fs+gs
37
38     fs      * []      = []

```

```

39     []      * gs      = []
40     (f:fs) * gg@(g:gs) = f*g : (f .* gs + fs * gg)
41
42     delta :: (Num a, Ord a) => [a] -> [a]
43     delta = ([1,-1] *)
44
45     shift :: [a] -> [a]
46     shift = tail
47
48     p2fct :: Num a => [a] -> a -> a
49     p2fct [] x = 0
50     p2fct (a:as) x = a + (x * p2fct as x)
51
52     comp :: (Eq a, Num a, Ord a) => [a] -> [a] -> [a]
53     comp _ [] = error ".."
54     comp [] _ = []
55     comp (f:fs) g0@(0:gs) = f : gs * (comp fs g0)
56     comp (f:fs) gg@(g:gs) = ([f] + [g] * (comp fs gg))
57                             + (0 : gs * (comp fs gg))
58
59     deriv :: Num a => [a] -> [a]
60     deriv [] = []
61     deriv (f:fs) = deriv1 fs 1
62     where
63         deriv1 [] _ = []
64         deriv1 (g:gs) n = n*g : deriv1 gs (n+1)

```

3.3 Univariate.lhs

Listing 3.3: Univariate.lhs

```

1  Univariate.lhs
2
3  > module Univariate where
4  > import Data.Ratio
5  > import Polynomials
6
7  From the output list
8  map f [0..]
9  of a polynomial
10 f :: Int -> Ratio Int
11 we reconstruct the canonical form of f.
12
13 > -- difference analysis
14 > difs :: (Num a) => [a] -> [a]

```

```

15 > difs [] = []
16 > difs [_] = []
17 > difs (i:jj@(j:js)) = j-i : difs jj
18 >
19 > difLists :: (Eq a, Num a) => [[a]] -> [[a]]
20 > difLists [] = []
21 > difLists xx@(xs:xss) =
22 >   if isConst xs then xx
23 >   else difLists $ difs xs : xx
24 >   where
25 >     isConst (i:jj@(j:js)) = all (==i) jj
26 >     isConst _ = error "difLists: lack of data, or not a
    polynomial"
27 >
28 > -- This degree function is "strict", so only take
    finite list.
29 > degree' :: (Eq a, Num a) => [a] -> Int
30 > degree' xs = length (difLists [xs]) -1
31 >
32 > -- This degree function can compute the degree of
    infinite list.
33 > degreeLazy :: (Eq a, Num a) => [a] -> Int
34 > degreeLazy xs = helper xs 0
35 >   where
36 >     helper as@(a:b:c:_) n
37 >       | a==b && b==c = n
38 >       | otherwise   = helper (difs as) (n+1)
39 >
40 > -- This is a hybrid version, safe and lazy.
41 > degree :: (Num a, Eq a) => [a] -> Int
42 > degree xs = let l = degreeLazy xs in
43 >   degree' $ take (l+2) xs
44
45 Newton interpolation formula
46 First we introduce a new infix symbol for the operation
    of taking a falling power.
47
48 > infixr 8 ^- -- falling power
49 > (^-) :: (Eq a, Num a) => a -> a -> a
50 > x ^- 0 = 1
51 > x ^- n = (x ^- (n-1)) * (x - n + 1)
52
53 Claim (Newton interpolation formula)
54 A polynomial f of degree n is expressed as
55  $f(z) = \sum_{k=0}^n (\text{diff}^n(f)(0)/k!) * (x \text{ } ^- \text{ } n)$ 

```

```

56 where  $\text{diff}^n(f)$  is the  $n$ -th difference of  $f$ .
57
58 Example
59 Consider a polynomial  $f = 2x^3 + 3x$ .
60
61 In general, we have no prior knowledge of this form, but
    we know the sequences as a list of outputs:
62
63 > let f x = 2*x^3+3*x
64 > take 10 $ map f [0..]
65 [0,5,22,63,140,265,450,707,1048,1485]
66 > degree $ take 10 $ map f [0..]
67 3
68
69 Let us try to get differences:
70
71 > difs $ take 10 $ map f [0..]
72 [5,17,41,77,125,185,257,341,437]
73 > difs it
74 [12,24,36,48,60,72,84,96]
75 > difs it
76 [12,12,12,12,12,12,12]
77
78 Or more simply take difLists:
79
80 > difLists [take 10 $ map f [0..]]
81 [[12,12,12,12,12,12,12]
82  ,[12,24,36,48,60,72,84,96]
83  ,[5,17,41,77,125,185,257,341,437]
84  ,[0,5,22,63,140,265,450,707,1048,1485]
85  ]
86
87 What we need is the heads of above lists.
88
89 > map head it
90 [12,12,5,0]
91
92 Newton interpolation formula gives
93  $f' x = 0(x - 0) \text{'div' } (0!) + 5(x - 1) \text{'div' } (1!) +$ 
     $12(x - 2) \text{'div' } (2!) + 12(x - 3) \text{'div' } (3!)$ 
94  $= 5(x - 1) + 6(x - 2) + 2(x - 3)$ 
95 So
96
97 > let f x = 2*x^3+3*x
98 > let f' x = 5*(x - 1) + 6*(x - 2) + 2*(x - 3)

```

```

99   > take 10 $ map f [0..]
100   [0,5,22,63,140,265,450,707,1048,1485]
101   > take 10 $ map f' [0..]
102   [0,5,22,63,140,265,450,707,1048,1485]
103
104   Assume the differences are given in a list
105   [x_0, x_1 ..]
106   where x_i = diff^k(f)(0).
107   Then the implementation of the Newton interpolation
      formula is as follows:
108
109   > newtonC :: (Fractional t, Enum t) => [t] -> [t]
110   > newtonC xs = [x / factorial k | (x,k) <- zip xs [0..]]
111   >   where
112   >     factorial k = product [1..fromInteger k]
113
114   > let f x = 2*x^3+3*x
115   > take 10 $ map f [0..]
116   [0,5,22,63,140,265,450,707,1048,1485]
117   > difLists [it]
118   [[12,12,12,12,12,12,12],[12,24,36,48,60,72,84,96],[5,17,41,77,125,185,257,341,405]]
119
119   > reverse $ map head it
120   [0,5,12,12]
121   > newtonC it
122   [0 % 1,5 % 1,6 % 1,2 % 1]
123
124   The list of first differences can be computed as follows:
125
126   > firstDifs :: (Eq a, Num a) => [a] -> [a]
127   > firstDifs xs = reverse $ map head $ difLists [xs]
128
129   Mapping a list of integers to a Newton representation:
130
131   > list2npol :: (Integral a) => [Ratio a] -> [Ratio a]
132   > list2npol = newtonC . firstDifs
133
134   > take 10 $ map f [0..]
135   [0,5,22,63,140,265,450,707,1048,1485]
136   > list2npol it
137   [0 % 1,5 % 1,6 % 1,2 % 1]
138
139   We need to map Newton falling powers to standard powers.
140   This is a matter of applying combinatorics, by means of a
      convention formula that uses the so-called Stirling

```

```

        cyclic numbers (of the first kind.)
141 Its defining relation is
142    $(x^{-n}) = \sum_{k=1}^n (\text{stirlingC } n \ k) * (-1)^{(n-k)} * x^k.$ 
143 The key equation is
144    $(x^{-n}) = (x^{-(n-1)}) * (x^{-n+1})$ 
145    $= x * (x^{-(n-1)}) - (n-1) * (x^{-(n-1)})$ 
146
147 Therefore, an implementation is as follows:
148
149 > stirlingC :: (Integral a) => a -> a -> a
150 > stirlingC 0 0 = 1
151 > stirlingC 0 _ = 0
152 > stirlingC n k = stirlingC (n-1) (k-1) + (n-1)*stirlingC
    (n-1) k
153
154 This definition can be used to convert from falling
    powers to standard powers.
155
156 > fall2pol :: (Integral a) => a -> [a]
157 > fall2pol 0 = [1]
158 > fall2pol n = 0    -- No constant term.
159 >             : [(-1)^(n-k) * stirlingC n k | k<-[1..n]]
160
161 We use this to convert Newton representations to standard
    polynomials in coefficients list representation.
162 Here we have uses sum to collect same order terms in list
    representation.
163
164 > npol2pol :: (Integral a) => [Ratio a] -> [Ratio a]
165 > npol2pol xs = sum [ [x] * map fromInteger (fall2pol k)
166 >                     | (x,k) <- zip xs [0..]
167 >                     ]
168
169 Finally, here is the function for computing a polynomial
    from an output sequence:
170
171 > list2pol :: (Integral a) => [Ratio a] -> [Ratio a]
172 > list2pol = npol2pol . list2npol
173
174 Reconstruction as curve fitting
175 > list2pol $ map (\n -> 7*n^2+3*n-4) [0..100]
176 [(-4) % 1,3 % 1,7 % 1]
177
178 > list2pol [0,1,5,14,30]

```



```

179 [0 % 1,1 % 6,1 % 2,1 % 3]
180 > map (\n -> n%6 + n^2%2 + n^3%3) [0..4]
181 [0 % 1,1 % 1,5 % 1,14 % 1,30 % 1]
182
183 > map (p2fct $ list2pol [0,1,5,14,30]) [0..8]
184 [0 % 1,1 % 1,5 % 1,14 % 1,30 % 1,55 % 1,91 % 1,140 %
    1,204 % 1]
185
186 --
    -----

187
188 Thiele's interpolation formula
189 https://rosettacode.org/wiki/Thiele%27
    s\_interpolation\_formula#Haskell
190 http://mathworld.wolfram.com/ThielesInterpolationFormula.
    html
191
192 reciprocal difference
193 Using the same notation of
194 https://rosettacode.org/wiki/Thiele%27
    s\_interpolation\_formula#C
195
196 > rho :: (Integral a) =>
197 > [Ratio a] -- A list of output of f :: a -> Ratio
    a
198 > -> a -> Int -> Ratio a
199 > rho fs 0 i = fs !! i
200 > rho fs n _
201 > | n < 0 = 0
202 > rho fs n i = (n*den)%num + rho fs (n-2) (i+1)
203 > where
204 > num = numerator next
205 > den = denominator next
206 > next = rho fs (n-1) (i+1) - rho fs (n-1) i
207
208 Note that (%) has the following type,
209 (%) :: Integral a => a -> a -> Ratio a
210
211 > a :: (Integral a) => [Ratio a] -> a -> Ratio a
212 > a fs 0 = head fs
213 > a fs n = rho fs n 0 - rho fs (n-2) 0
214
215 Consider the following continuous fraction form.
216 (%i25) f(x) := 1+(x/(2+(x-1)/(3+(x-2)/4)));

```

```

217 (%o25) f(x):=x/(2+(x-1)/(3+(x-2)/4))+1
218 (%i26) ratsimp(f(x));
219 (%o26) (x^2+16*x+16)/(16+6*x)
220
221 *Univariate> map (a fs) [0..]
222 [1 % 1,2 % 1,3 % 1,4 % 1,*** Exception: Ratio has zero
    denominator
223
224 *Univariate> let func x = (x^2 + 16*x + 16)%(6*x + 16)
225 *Univariate> let fs = map func [0..]
226 *Univariate> take 5 $ map (rho fs 0) [0..]
227 [1 % 1,3 % 2,13 % 7,73 % 34,12 % 5]
228 *Univariate> take 5 $ map (rho fs 1) [0..]
229 [2 % 1,14 % 5,238 % 69,170 % 43,230 % 53]
230 *Univariate> take 5 $ map (rho fs 2) [0..]
231 [4 % 1,79 % 16,269 % 44,667 % 88,413 % 44]
232 *Univariate> take 5 $ map (rho fs 3) [0..]
233 [6 % 1,6 % 1,6 % 1,6 % 1,6 % 1]
234
235 > tDegree :: Integral a => [Ratio a] -> a
236 > tDegree fs = helper fs 0
237 >   where
238 >     helper fs n
239 >       | isConstants fs' = n
240 >       | otherwise      = helper fs (n+1)
241 >       where
242 >         fs' = map (rho fs n) [0..]
243 >         isConstants (i:j:_) = i==j -- 2 times match
244 >         -- isConstants (i:j:k_) = i==j && j==k -- 3 times
           match
245
246 *Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
247 *Univariate> let hs = map h [0..]
248 *Univariate> tDegree hs
249 4
250 *Univariate> map (a hs) [0..(tDegree hs)]
251 [3 % 1,(-23) % 42,(-28) % 13,767 % 14,7 % 130]
252
253 With Maxima,
254 (%i35) h(t) := 3+t/((-23/42)+(t-1)/((-28/13)+(t-2)
           /((767/14)+(t-3)/(7/13)))));
255
256 (%o35) h(t):=t/((-23)/42+(t-1)/((-28)/13+(t-2)
           /(767/14+(t-3)/(7/13))))+3
257 (%i36) ratsimp(h(t));

```

```

258
259     (%o36) (18*t^2+6*t+3)/(1+2*t+20*t^2)
260
261 > thieleC :: (Integral a) => [Ratio a] -> [Ratio a]
262 > thieleC lst = map (a lst) [0..(tDegree lst)]
263
264 *Univariate> thieleC hs
265 [3 % 1,(-23) % 42,(-28) % 13,767 % 14,7 % 130]
266
267 We need a convertor from this thiele sequence to
    continuous form of rational function.
268
269 > nextStep [a0,a1] (v:_) = a0 + v/a1
270 > nextStep (a:as) (v:vs) = a + (v / nextStep as vs)
271 >
272 > -- From thiele sequence to (rational) function.
273 > thiele2ratf :: Integral a => [Ratio a] -> (Ratio a ->
    Ratio a)
274 > thiele2ratf as x
275 > | x == 0      = head as
276 > | otherwise = nextStep as [x,x-1 ..]
277
278 *Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
279 *Univariate> let hs = map h [0..]
280 *Univariate> let as = thieleC hs
281 *Univariate> as
282 [3 % 1,(-23) % 42,(-28) % 13,767 % 14,7 % 130]
283 *Univariate> let th x = thiele2ratf as x
284 *Univariate> take 5 hs
285 [3 % 1,27 % 23,87 % 85,183 % 187,45 % 47]
286 *Univariate> map th [0..5]
287 [3 % 1,27 % 23,87 % 85,183 % 187,45 % 47,69 % 73]
288
289 We represent a rational function by a tuple of
    coefficient lists:
290 (ns,ds) :: ([Ratio Int],[Ratio Int])
291 where ns and ds are coef-list-rep of numerator polynomial
    and denominator polynomial.
292 Here is a translator from coefficients lists to rational
    function.
293
294 > -- similar to p2fct
295 > lists2ratf :: (Integral a) =>
296 > ([Ratio a],[Ratio a]) -> (Ratio a ->
    Ratio a)

```

```

297 > lists2ratf (ns,ds) x = p2fct ns x / p2fct ds x
298
299 *Univariate> let frac x = lists2ratf
      ([1,1%2,1%3],[2,2%3]) x
300 *Univariate> take 10 $ map frac [0..]
301 [1 % 2,11 % 16,1 % 1,11 % 8,25 % 14,71 % 32,8 % 3,25 %
      8,79 % 22,65 % 16]
302 *Univariate> let ffrac x = (1+(1%2)*x+(1%3)*x^2)
      /(2+(2%3)*x)
303 *Univariate> take 10 $ map ffrac [0..]
304 [1 % 2,11 % 16,1 % 1,11 % 8,25 % 14,71 % 32,8 % 3,25 %
      8,79 % 22,65 % 16]
305
306 The following canonicalizer reduces the tuple-rep of
      rational function in canonical form
307 That is, the coefficient of the lowest degree term of the
      denominator to be 1.
308 However, since our input starts from 0 and this means
      firstNonzero is the same as head.
309
310 > canonicalize :: (Integral a) => ([Ratio a],[Ratio a])
      -> ([Ratio a],[Ratio a])
311 > canonicalize rat@(ns,ds)
312 >   | dMin == 1 = rat
313 >   | otherwise = (map (/dMin) ns, map (/dMin) ds)
314 >   where
315 >     dMin = firstNonzero ds
316 >     firstNonzero [a] = a -- head
317 >     firstNonzero (a:as)
318 >       | a /= 0     = a
319 >       | otherwise = firstNonzero as
320
321 What we need is a translator from Thiele coefficients to
      this tuple-rep.
322
323 > thiele2coef :: (Integral a) => [Ratio a] -> ([Ratio a]
      ,[Ratio a])
324 > thiele2coef as = canonicalize $ t2r as 0
325 >   where
326 >     t2r [an,an'] n = ([an*an'-n,1],[an'])
327 >     t2r (a:as) n = ((a .* num) + ([-n,1] * den), num)
328 >     where
329 >       (num, den) = t2r as (n+1)
330 >
331 > lists2rat :: (Integral a) => [Ratio a] -> ([Ratio a], [

```

```

Ratio a])
332 > lists2rat = thiele2coef . thieleC
333
334 *Univariate> let h t = (3+6*t+18*t^2)%(1+2*t+20*t^2)
335 *Univariate> let hs = map h [0..]
336 *Univariate> take 5 hs
337 [3 % 1,27 % 23,87 % 85,183 % 187,45 % 47]
338 *Univariate> let th x = thiele2ratf as x
339 *Univariate> map th [0..5]
340 [3 % 1,27 % 23,87 % 85,183 % 187,45 % 47,69 % 73]
341 *Univariate> as
342 [3 % 1,(-23) % 42,(-28) % 13,767 % 14,7 % 130]
343 *Univariate> thiele2coef as
344 ([3 % 1,6 % 1,18 % 1],[1 % 1,2 % 1,20 % 1])

```

3.4 Multivariate.lhs

Listing 3.4: Multivariate.lhs

```

1 Multivariate.lhs
2
3 > module Multivariate
4 >   where
5
6 >   import Data.Ratio
7 >   import Univariate
8 >   ( degree, list2pol
9 >     , thiele2ratf, lists2ratf, thiele2coef, lists2rat
10 >   )

```