Finite fields

Ray D. Sameshima

2016/09/23 $\sim$ 2016/10/12 19:35

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/`

0.2 Set theoretical gadgets

0.2.1 Numbers

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

1. $\mathbb{N}$ (Peano axiom: $\emptyset, \mathrm{suc}$)
2. $\mathbb{Z}$
3. $\mathbb{Q}$
4. $\mathbb{R}$ (Dedekind cut)
5. $\mathbb{C}$

0.2.2 Algebraic structures

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

0.3 Haskell language

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

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

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

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

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

> 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]

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

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

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

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

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

Chapter 1

Basics

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

$$\texttt{(+) :: R -> R -> R} \tag{1.1}$$
$$\texttt{(*) :: R -> R -> R} \tag{1.2}$$

satisfying the following 3 (ring) axioms:

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

$$\forall a,b,c \in R, (a+b)+c = a+(b+c) \quad \text{(associativity for +)} \tag{1.3}$$
$$\forall a,b, \in R, a+b = b+a \quad \text{(commutativity)} \tag{1.4}$$
$$\exists 0 \in R, \text{ s.t. } \forall a \in R, a+0 = a \quad \text{(additive identity)} \tag{1.5}$$
$$\forall a \in R, \exists(-a) \in R \text{ s.t. } a+(-a) = 0 \quad \text{(additive inverse)} \tag{1.6}$$

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

$$\forall a,b,c \in R, (a*b)*c = a*(b*c) \quad \text{(associativity for } *) \tag{1.7}$$
$$\exists 1 \in R, \text{ s.t. } \forall a \in R, a*1 = a = 1*a \quad \text{(multiplicative identity)} \tag{1.8}$$

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

$$a*(b+c) = (a*b)+(a*c) \quad \text{(left distributivity)} \tag{1.9}$$
$$(a+b)*c = (a*c)+(b*c) \quad \text{(right distributivity)} \tag{1.10}$$

1.1.2 Fields

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

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

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

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

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

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

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

[1] Here we have taken an equivalence class,

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

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.

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

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

1.1.5 Greatest common divisor

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

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

Example, by hands

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

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

Then it hits main line:

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

---

with the following operations:

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

This is equivalent to take modular n:

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

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

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

Finally it ends with 1:

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

As another example, consider 15 and 25:

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

Example, by 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

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

and inductively, for $i \geq 2$,

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

The termination condition[3] is

$$r_k = 0 \tag{1.44}$$

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

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

Proof

By definition,

$$\gcd(r_{i-1}, r_i) = \gcd(r_{i-1}, r_{i-2} - q_i * r_{i-1}) \tag{1.48}$$
$$= \gcd(r_{i-1}, r_{i-2}) \tag{1.49}$$

[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

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

i.e.,

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

Next, for $i = 0, 1$ observe

$$a * s_i + b * t_i = r_i. \tag{1.52}$$

Let $i \geq 2$, then

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

Therefore, inductively we get

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

This prove Bézout's lemma.
■

Haskell implementation

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

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

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

> 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 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 (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

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

there is a unique number

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

iff a and n are coprime.

Proof

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

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

Therefore

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

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

$$a * s = 1 \mod n. \quad (1.63)$$

■

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

If p is prime, then

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

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

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

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 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

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

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

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

In fact,

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

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

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

```
> inverses :: Int -> Maybe [(Int, Int)]
> inverses n
>   | isField 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 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.

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 :: Int -> Int -> Maybe Int
> 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

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

and a natural inclusion (or a forgetful map)[8]

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

[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

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

of normal product on $\mathbb{Z}$.

Then we can define a map

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

by[9]

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

Example and implementation

An easy implementation is the followings:[10]

```
A map from Q to Z_p.

> modp :: Ratio Int -> Int -> Int
> 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 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,

$$\texttt{add a b == a `add` b} \tag{1.76}$$

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

$$\begin{aligned}
\frac{3}{7} \in \mathbb{Q} \quad \mapsto \quad & (3 \times ¿(7^{-1} \mod 11) \mod 11) && (1.77)\\
&= (3 \times 8) \mod 11 && (1.78)\\
&= 24 \mod 11 && (1.79)\\
&= 2 \mod 11. && (1.80)
\end{aligned}$$

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 $a \in \mathbb{Z}_p$.

$$a := q \mod p \tag{1.81}$$

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

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

$$a * s_i + p * t_i = r_i \tag{1.82}$$

Therefore

$$r_i = a * s_i \mod p \Leftrightarrow \frac{r_i}{s_i} \mod p = a. \tag{1.83}$$

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

$$r_i^2, s_i^2 < p \tag{1.84}$$

as the termination condition for this reconstruction.

Haskell implementation

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

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

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

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

Look at the first hit of termination condition eq.(1.84), $r_4 = 1$ 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 :: (Int, Int)       -- (q `modp` p, p)
>       -> (Ratio Int, Int)
> 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 ‘modp‘ 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 :: [(Int,Int)] -> Ratio Int
> 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 start from smaller prime, say 101. And see the first good guess, Haskell tells us that in order to reconstruct $\frac{331}{739}$, we should take three primes start from 614693:

```
*Ffield> let q = (331%739)
(0.01 secs, 44,024 bytes)
*Ffield> let smallerprimes = dropWhile (<100) $ takeWhile (<978948) primes
(0.01 secs, 39,968 bytes)
*Ffield> let knownData = zip (map (modp q) smallerprimes) smallerprimes
```

```
(0.01 secs, 39,872 bytes)
*Ffield> matches3' $ map guess knownData
(331 % 739,614693)
(17.64 secs, 12,402,878,080 bytes)
```

1.1.12 Chinese remainder theorem

From wikipedia[11]

> 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

$$n \mod 3 = 2, n \mod 5 = 3, n \mod 7 = 2. \tag{1.85}$$

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

$$\begin{aligned} x &= a_1 \mod n_1 && (1.86) \\ x &= a_2 \mod n_2 && (1.87) \end{aligned}$$

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

---

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

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

$$0 \leq a < n_1 \times n_2. \tag{1.88}$$

Proof

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

$$n_1 * m_1 + n_2 * m_2 = 1. \tag{1.89}$$

Now we have

$$\begin{aligned} n_1 * m_1 &= 1 \mod n_2 && (1.90)\\ n_2 * m_2 &= 1 \mod n_1 && (1.91) \end{aligned}$$

that is

$$\begin{aligned} m_1 &= n_1^{-1} \mod n_2 && (1.92)\\ m_2 &= n_2^{-1} \mod n_1. && (1.93) \end{aligned}$$

Then

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

is a solution.

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

$$\begin{aligned} a - a' &= 0 \mod n_1 && (1.95)\\ a - a' &= 0 \mod n_2. && (1.96) \end{aligned}$$

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

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

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

Generalization

Given $a \in Z_n$ of pairwise coprime numbers

$$n := n_1 * \cdots * n_k, \tag{1.98}$$

a system of equations

$$a_i = a \mod n_i|_{i=1}^{k} \tag{1.99}$$

have a unique solution

$$a = \sum_i m_i a_i \mod n, \tag{1.100}$$

where

$$m_i = \left(\frac{n_i}{n} \mod n_i \right) \frac{n}{n_i} \bigg|_{i=1}^{k}. \tag{1.101}$$

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:

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

A monomial is defined as

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

The total degree of this monomial is given by

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

1.2.2 Polynomials and rational functions

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

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

where

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

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

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

We denote

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

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

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

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

We denote

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

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

1.2.3 As data

We can identify a polynomial

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

as a set of coefficients

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

Similarly, for a rational function, we can identify

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

as an ordered pair of coefficients

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

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

1.3 Haskell implementation of univariate polynomials

Here we basically follows some part of §9 of ref.4, and its addendum[13].

Univariate.lhs

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

1.3.1 A polynomial as a list of coefficients

Let us start `instance` declaration, which enable us to use basic arithmetics:

```
> -- polynomials, as coefficients lists
> instance (Num a, Ord a) => Num [a] where
>   fromInteger c = [fromInteger c]
>
>   negate []     = []
>   negate (f:fs) = negate f : negate fs
>
>   signum [] = []
>   signum gs
>     | signum (last gs) < 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)
```

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

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

$$\texttt{(*) :: Num a => a -> a -> a} \tag{1.115}$$

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

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:

$$f(z) := \sum_{i=0}^{R} c_i z^i. \tag{1.116}$$

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[14]

[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 (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,

$$f(x) := x^2 + 1 \tag{1.117}$$
$$f := \lambda x.x^2 + 1 \tag{1.118}$$

are the same definition.

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

1.3.2 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

$$\texttt{[0..]} \tag{1.119}$$

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

$$\texttt{map f [0..] = [f 0, f 1 ..]} \tag{1.120}$$

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

$$\Delta(f)(n) := f(n+1) - f(n). \tag{1.121}$$

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

$$f(n) = \sum_{i=0}^{R} c_i n^i \tag{1.122}$$

and

$$\begin{aligned}
\Delta(f)(n) &:= f(n+1) - f(n) && (1.123)\\
&= \sum_{i=0}^{R} c_i \left\{(n+1)^i - n^i\right\} && (1.124)\\
&= \sum_{i=1}^{R} c_i \left\{(n+1)^i - n^i\right\} && (1.125)\\
&= \sum_{i=1}^{R} c_i \left\{i * n^{i-1} + O(n^{i-2})\right\} && (1.126)\\
&= c_R * R * n^{R-1} + O(n^{R-2}) && (1.127)
\end{aligned}$$

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

This guarantees the following function will terminate in finite steps[15]; `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]
]
```

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

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

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

For example,

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

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

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

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

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:

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

where

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

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

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

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

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

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

2.1.2 Towards canonical representations

Once we get the Newton's representation

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

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

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

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

2.1.3 Simplification of our problem

Without loss of generality, we can put

```
[0..]
```
(2.13)

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

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

is our output list.

Then we have slightly simpler forms of coefficients:

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

where Δ is the difference operator in eq.(1.121):

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

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

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

Under these settings, we have

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

where we have assume

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

Example

Consider a polynomial

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

and its out put list

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

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

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

so we get

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

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

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

2.1.4 Haskell implementation

Newton interpolation formula

First, the falling power is naturally given by recursively:

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

Assume the differences are given in a list

$$\texttt{[x\_0, x\_1 ..]} := \left[f(0), \Delta(f)(0), \Delta^2(f)(0), \cdots\right]. \tag{2.40}$$

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

```
> newton :: Integral a => [a] -> [Ratio a]
> newton xs = [x % factorial k | (x,k) <- zip xs [0..]]
>   where
>     factorial k = product [1..fromInteger k]
```

Consider a polynomial

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

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

```
*NewtonInterpolation> take 10 $ map f [0..]
[0,5,22,63,140,265,450,707,1048,1485]
*NewtonInterpolation> 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]
]
*NewtonInterpolation> 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):

```
*NewtonInterpolation> newton it
[0 % 1,5 % 1,6 % 1,2 % 1]
```

The list of first differences, i.e.,

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

can be computed as follows:

```
> firstDifs :: [Integer] -> [Integer]
> firstDifs xs = reverse $ map head $ difLists [xs]
```

Mapping a list of integers to a Newton representation:

```
> list2npol :: [Integer] -> [Rational]
> list2npol = newton . map fromInteger . 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.

Stirling numbers of the first kind

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

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

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

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

and

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

From the highest order, x^n, we get

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

We also put

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

and

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

The key equation is

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

and we get

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

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

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

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

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

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

Here is an implementation:

```
> 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 :: Integer -> [Integer]
> fall2pol 0 = [1]
> fall2pol n = 0 : [(stirlingC n k)*(-1)^(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 :: (Ord t, Num t) => [t] -> [t]
> npol2pol xs = sum [ [x] * (map fromInteger $ fall2pol k)
>                   | (x,k) <- zip xs [0..]
>                   ]
```

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

```
> list2pol :: [Integer] -> [Rational]
> 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.

2.2 Univariate rational functions

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

```
map f [0..]
```
(2.60)

2.2.1 Thiele's interpolation formula

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

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

where

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

2.2.2 Towards canonical representations

In order to get a unique representation of canonical form

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

we put

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

as a normalization, instead of d_0.

2.2.3 Haskell implementation

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`

Claim

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

$$a_0 := f(0) \tag{2.71}$$
$$a_n := \rho_{n,0} - \rho_{n-2,0}, \tag{2.72}$$

where ρ is so called the reciprocal difference:

$$\rho_{n,i} := 0, n < 0 \tag{2.73}$$
$$\rho_{0,i} := f(i), i = 0, 1, 2, \cdots \tag{2.74}$$
$$\rho_{n,i} := \frac{n}{\rho_{n-1,i+1} - \rho_{n-1,i}} + \rho_{n-2,i+1} \tag{2.75}$$

These preparation helps us to write the following codes:

```
Thiele's interpolation formula

Reciprocal difference rho, using the same notation of
```

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

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

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

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

Example

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

$$a_0 = 1, a_1 = 2, a_2 = 3, a_3 = 4. \tag{2.76}$$

The function is now

$$\begin{aligned} f(x) &:= 1 + \cfrac{x}{2 + \cfrac{x-1}{3 + \cfrac{x-2}{4}}} \qquad (2.77) \\ &= \frac{x^2 + 16x + 16}{16 + 6x} \qquad (2.78) \end{aligned}$$

Using Maxima, 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

```
> func x = (x^2 + 16*x + 16)%(6*x + 16)
```

```
*Univariate> map (a fs) [0..]
[1 % 1,2 % 1,3 % 1,4 % 1,*** Exception: Ratio has zero denominator
```

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

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

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

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

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)
>
> thiele' :: Integral a => [Ratio a] -> Ratio a -> Ratio a
> thiele' fs x
>   | x == 0 = a0
>   | otherwise = nextStep as [x,x-1 ..]
>   where
>     a0 = head as
>     as = thieleC fs
```

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 th x = thiele' hs x
*Univariate> map th [0..4]
```

```
[3 % 1,27 % 23,87 % 85,183 % 187,45 % 47]
*Univariate> th 0.5
3 % 2
*Univariate> th 0.1
27 % 10
```

Haskell representation for rational functions

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

```
(ns,ds) ::  ([Ratio Int],[Ratio Int])
```
(2.79)

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.

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

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

$$a_0 + \cfrac{z)}{a_1 + \cfrac{z-1}{a_2 + \cfrac{z-2}{\cdots + a_{r-1} + \cfrac{z-r+1}{a_r}}}} \tag{2.80}$$

The base case should be

$$a_{r-1} + \frac{z-r+1}{a_r} \quad = \quad \frac{a_r \times a_{r-1} - r + 1 + z}{a_r} \tag{2.81}$$

and induction step should be

$$a_{n-1}(z) \quad = \quad a_{n-1} + \frac{z-n+1}{a_n(z)} \tag{2.82}$$

$$= \quad \frac{a_{n-1}a_n(z) + z - n + 1}{a_n(z)} \tag{2.83}$$

```
> thiele2ratf :: (Integral a) => [Ratio a] -> ([Ratio a],[Ratio a])
> thiele2ratf as = 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)
```

Here is a simple check, and it works.

From the first example,

```
*Univariate> let func x = (x^2+16*x+16)%(6*x+16)
*Univariate> let funcList = map func [0..]
```

```
*Univariate> take 5 funcList
[1 % 1,3 % 2,13 % 7,73 % 34,12 % 5]
*Univariate> tDegree funcList
3
*Univariate> let thieleFunc x = thiele' funcList x
*Univariate> map thieleFunc [0..4]
[1 % 1,3 % 2,13 % 7,73 % 34,12 % 5]
*Univariate> let asOfFunc = thieleC funcList
*Univariate> asOfFunc
[1 % 1,2 % 1,3 % 1,4 % 1]
*Univariate> thiele2ratf asOfFunc
([16 % 1,16 % 1,1 % 1],[16 % 1,6 % 1])
*Univariate> canonicalizer it
([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 = thiele' hs x
*Univariate> map th [0..5]
[3 % 1,27 % 23,87 % 85,183 % 187,45 % 47,69 % 73]
*Univariate> let as = thieleC hs
*Univariate> as
[3 % 1,(-23) % 42,(-28) % 13,767 % 14,7 % 130]
*Univariate> thiele2ratf as
([3 % 20,3 % 10,9 % 10],[1 % 20,1 % 10,1 % 1])
*Univariate> canonicalizer it
([3 % 1,6 % 1,18 % 1],[1 % 1,2 % 1,20 % 1])
```

2.3 Multivariate polynomials

2.3.1 Foldings as recursive applications

Consider an arbitrary multivariate polynomial

$$f(z_1, \cdots, z_n) \in \mathbb{F}[z_1, \cdots, z_n]. \tag{2.84}$$

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

$$f(z_1, z_2, \cdots, z_n) = \sum_{r=0}^{R} a_r(z_2, \cdots, z_n) \prod_{i=0}^{r-1} (z_1 - y_i) \tag{2.85}$$

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

$$a_r(z_2, \cdots, z_n) = \sum_{s=0}^{S} b_s(z_3, \cdots, z_n) \prod_{j=0}^{s-1} (z_2 - x_j) \tag{2.86}$$

The terminate cotndition should be the univariate case.

2.4 Multivariate rational functions

2.4.1 The canonical normalization

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

$$\frac{\sum_\alpha n_\alpha z^\alpha}{\sum_\beta d_\beta z^\beta} \tag{2.87}$$

A canonical choice is

$$d_0 = d_{(0,\cdots,0)} = 1. \tag{2.88}$$

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

$$d'_0 = d_s \neq 0. \tag{2.89}$$

2.4.2 An auxiliary t

Introducing an auxiliary variable t, let us define

$$h(t, z) := f(tz_1, \cdots, tz_n), \tag{2.90}$$

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

$$h(t, z) = \frac{\sum_{r=0}^{R} p_r(z) t^r}{1 + \sum_{r'=1}^{R'} q_{r'}(z) t^{r'}} \tag{2.91}$$

where

$$p_r(z) = \sum_{|\alpha|=r} n_\alpha z^\alpha \tag{2.92}$$

$$q_{r'}(z) = \sum_{|\beta|=r'} n_\beta z^\beta \tag{2.93}$$

are homogeneous polynomials.

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

A simplification

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

$$p_r(1, z_2, \cdots, z_n) \tag{2.94}$$

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