Potenzreihe: Potenz-Schreie!

Jonathan Frech ⟨info@jfrech.com⟩

2021-08-02

1 Einleitung

Douglas McIlroys Artikel [McI99] folgend setzt sich dieser Text mit der Implementierung formaler rationaler Potenzreihen in einer Variablen $\mathbb{Q}[\![x]\!]$ sowie formaler Potenzreihen über dem zweielementigen Körper $\mathbb{F}_2[\![x]\!]$ in der Programmiersprache Haskell [Mar10] auseinander, wobei ein besonderes Augenmerk auf ihre einheitliche Bedarfsauswertung gelegt wird.

Quelltext aller betrachteten Haskell-Schnipsel: pr.hs

2 Kardinalitätsüberlegungen

Aufgrund der Endlichkeit des Berechnens nehmen viele *Typen*, welche in dieser Betrachtung schlicht als Mengen von Werten verstanden werden, die Form endlicher Mengen an, als Beispiel

```
Bool,  Int16,  Maybe (Int8,Int8,Int8).
```

Jedoch ist nicht jedes Problem als Element eines endlichen Typs abbildbar, weshalb das Einführen von Typen welche als Menge eine unendliche Kardinalität aufweisen nützlich ist (das Ausrufezeichen „`!`“ soll andeuten, dass keine Bedarfsauswertung zugelassen wird):

```
![Int8],  !String,  data Tree = Leaf | !Tree Tree Tree,  Integer.
```

Obige Typen sind als Mengen abzählbar. In Haskell ist es jedoch weiter möglich, Typen größerer Kardinalität zu benutzen:

$$\texttt{Integer -> Bool} \quad \cong \quad (\mathbb{F}_2)^{\mathbb{N}},$$

wenn auch die Menge der berechenbaren Abbildungen in diesem Typ nach Abschätzung zu allen Turing-Maschinen erneut abzählbar ist, also die *effektive Kardinalität* nicht größer ist als zuvor. Es ist also zu beachten, dass zwischen der konzeptionellen Größe des Typs und der berechenbar erreichbaren Teilmenge eine beliebig große Kardinaldifferenz herrschen kann:

$$\texttt{(... ((Integer -> Bool) -> Bool) -> ... -> Bool)} \quad \cong \quad \mathbb{F}_2^{\cdot^{\cdot^{\cdot^{\mathbb{F}_2^{\mathbb{N}}}}}},$$

wobei erneut nur abzählbar viele berechenbare Funktionen in obigem Typ leben.

Bedarfsauswertung erlaubt es nun, Obiges auf *Daten* anstatt von Funktionen anzuwenden:

$$\texttt{[Bool]} \quad \cong \quad \sum_{j\in\mathbb{N}} \mathbb{F}_2^j + \mathbb{F}_2^{\mathbb{N}},$$

wobei der letzte Mengensummand die Überabzählbarkeit mittels Bedarfsauswertung bringt.

3 Formale Potenzreihen

Somit klingt es plausibel, mittels Bedarfsauswertung *formale Potenzreihen* über einem als Typ gegebenen Ring zu modellieren. Wenn auch ein überabzählbarer Ring wie zum Beispiel $\mathbb{R}$ oder $\mathbb{C}$ abzählbar approximierbar[1] als Typ betrachtet werden könnte, beschränkt sich dieser Text auf die Ringe $R \in \{\mathbb{Q}, \mathbb{F}_2\}$, wobei $\mathbb{Q}$ als eindeutiger unendlicher Primkörper modulo Isomorphie ein lohnender Kandidat ist und mittels $\mathbb{F}_2$ Teilmengen der natürlichen Zahlen modelliert werden können. Es sei angemerkt, dass die Konstruktion der formalen Potenzreihen und ihre Implementierung auch über allgemeineren Ringen stattfinden kann, diese Objekte jedoch mehr Eigeninitiative fordern, da die Prelude-Typklassen `Num` und `Fractional` etwas schwammig auf die sonst bekannten algebraischen Strukturen abbilden.
Zuletzt sei angemerkt, dass der Betrachtung in mehreren Variablen keine theoretischen Schranken gesetzt sind, sondern vielmehr notationelle Klarheit und Ausführbarkeit auf zeitgenössischer Hardware im Wege stehen.

3.1 Definition

Formale Potenzreihen in einer Variablen sind aus der Analysis inspiriert notierte Enumerationen des zugrundeliegenden Rings, hier $R \in \{\mathbb{Q}, \mathbb{F}_2\}$.

$$R[\![x]\!] := \{f : \mathbb{N}_0 \to R\} =: \left\{ \sum_{j=0}^{\infty} a_j x^j \;\middle|\; a_j \in R \right\},$$

wobei „x“ *rein symbolisch* zu verstehen ist, das heißt *keine Topologie* auf dem Ring gefordert wird. Es heißen x die *Variable*, a_j die *Koeffizienten* und f die *Potenzreihe*.
Auf obiger Menge wird nun eine Ringstruktur definiert, das heißt zwei Operatoren „$+$“ und „$\cdot$“ welche die *Ringaxiome* [Mar08, S. 85] erfüllen. Diese Operatoren werden solchen nicht-symbolischer „x“ erinnernd definiert:

$$\left(\sum_{i=0}^{\infty} a_i x^i\right) + \left(\sum_{j=0}^{\infty} b_j x^j\right) := \left(\sum_{k=0}^{\infty} (a_k + b_k) x^k\right)$$

und

$$\left(\sum_{i=0}^{\infty} a_i x^i\right) \cdot \left(\sum_{j=0}^{\infty} b_j x^j\right) := \left(\sum_{k=i+j} (a_i \cdot b_j) x^k\right).$$

[1] Es sei angemerkt, dass analytisch auch $\mathbb{Q}[\![x]\!]_{\text{konv}} \subset \mathbb{R}[\![x]\!]_{\text{konv}}$ dicht liegt für Potenzreihen mit unendlichem Konvergenzradius mittels ausreichend rasanter Koeffizientenapproximation.

Weiter definiert man eine *Einbettung* des zugrundeliegenden Rings

$$\iota : R \hookrightarrow R[\![x]\!], \quad r \mapsto rx^0 + 0x^1 + 0x^2 + \dots$$

und damit

$$r \cdot \left(\sum_{i=0}^{\infty} a_i x^i \right) := \iota(r) \cdot \left(\sum_{i=0}^{\infty} a_i x^i \right) = \sum_{i=0}^{\infty} r a_i x^i.$$

3.1.1 Implementierung

Bereits bei der bloßen Übertragung der Definition in Haskell-Quelltext beweist sich Bedarfsauswertung zusammen mit der auf natürliche Weise passenden Auswertungsrichtung des Standardtyps „`[]`“. Für `a` $:= R \in \{\mathbb{Q}, \mathbb{F}_2\}$ oder etwas allgemeiner `a` $\in$ `Num` setzt man $R[\![x]\!] \cong$ `[a]` gleich[2] und definiert darauf punktweise Addition

```
(+) :: Num a => [a] -> [a] -> [a]
(+) = zipWith (+)
```

sowie mit einer kurzen Rechnung

$$\begin{aligned} f \cdot g &= (a_0 + a_1 x + \dots) \cdot (b_0 + b_1 x + \dots) \\ &= (a_0 + (a_1 x + \dots)) \cdot (b_0 + (b_1 x + \dots)) \\ &= a_0 \cdot b_0 + a_0 \cdot (b_1 x + \dots) + (a_1 x + \dots) \cdot b_0 + (a_1 x + \dots) \cdot (b_1 x + \dots) \\ &= a_0 \cdot b_0 + x \cdot \Big(a_0 \cdot (b_1 + b_2 x + \dots) + (a_1 + a_2 x + \dots) \cdot (b_0 + b_1 x + \dots) \Big) \end{aligned}$$

und der Einbettung

```
iota :: Num a => a -> [a]
iota = (: repeat 0)
(.*) :: Num a => a -> [a] -> [a]
(.*) q = (iota q *)
```

die Multiplikation:

```
(*) :: Num a => [a] -> [a] -> [a]
f@(a:as) * g@(b:bs) = a*b : (a .* bs + as * g)
```

Definiert man weiter punktweise Negation und `Integer`-spezialisierte Einbettung erhält man die Minimaldefinition der `Num`-Typklasse [lib01, Numeric type classes, `#t:Num`], wobei `abs` und `signum` mangels Topologie und Bedeutung undefiniert bleiben:

```
instance Num a => Num [a] where
    (+)                 = zipWith (+)
    f@(a:as) * g@(b:bs) = a*b : (a .* bs + as * g)
    negate              = map negate
    fromInteger         = iota . fromInteger
    abs                 = undefined
    signum              = undefined
```

[2] Typ-Puristen mögen einwerfen, dass dieser Typ semantisch überbelegt ist. Isomorph könnte man `data Num a => Ps a = Pz | a :+: Ps a` [McI99, 7 Final remarks] als semantisch prägnanteren Typ setzen, müsste dann jedoch alle benutzten im Prelude vordefinierten Manipulationen der Listen auf diesem Typ selbst implementieren.

Der Ring der formalen Potenzreihen ist zwar kein Körper – so hat Beispielsweise $0 \neq x$ nach Koeffizientenvergleich kein Inverses –, besitzt jedoch etwas Divisionsstruktur: Der Quotient ${}^{f}\!/_{g} = h$, so wenn er existiert, erfüllt $f = h \cdot g$, also

$$\begin{aligned} f &= (c_0 + c_1 x + \dots) \cdot (b_0 + b_1 x + \dots) \\ &= (c_0 + (c_1 x + \dots)) \cdot (b_0 + (b_1 x + \dots)) \\ &= c_0 \cdot b_0 + x \cdot \Big(c_0 \cdot (b_1 + b_2 x + \dots) + (c_1 + c_2 x + \dots) \cdot (b_0 + (b_1 x + \dots)) \Big) \end{aligned}$$

und nach Koeffizientenvergleich mit $f = (a_0 + a_1 x + \dots)$ folgt $c_0 = {}^{a_0}\!/_{b_0}$ für den ersten Koeffizienten sowie

$${}^{(f\,-\,a_0)}\!/_{x} = (a_1 + a_2 x + \dots) = c_0 \cdot (b_1 + b_2 x + \dots) + (c_1 + c_2 x + \dots) \cdot g,$$

also

$$(c_1 + c_2 x + \dots) = \Big({}^{(f\,-\,a_0)}\!/_{x} - c_0 (b_1 + b_2 x + \dots) \Big) / g.$$

Obiges ist direkt implementierbar, wobei ${}^{a_0}\!/_{b_0}$ eine Fallunterscheidung fordert, jedoch nach angenommener Existenz eines Quotienten h die ersten Koeffizienten von f und g entweder beide verschwinden oder der erste Koeffizient von g eine Einheit ist, in einem Körper damit äquivalent nicht verschwindet:

```
(0:f)    / (0:g)    = f/g
_        / (0:_)    = undefined
f@(a:as) / g@(b:bs) = c : (as - c .* bs)/g where c = a/b
```

Damit ist es möglich, die Minimaldefinition der `Fractional`-Typklasse [lib01, Numeric type classes, `#t:Fractional`] zu definieren (die Typklasse `Eq a` wird für die `0`-Fallunterscheidung benötigt und beinhaltet `Rational` sowie `Bool`):

```
instance (Eq a, Fractional a) => Fractional [a] where
    (0:f)    / (0:g)    = f/g
    _        / (0:_)    = undefined
    f@(a:as) / g@(b:bs) = (c : (as - c .* bs)/g) where c = a/b
    fromRational        = iota . fromRational
```

Somit ist der Ring der formalen Potenzreihen $R[\![x]\!]$ samt Division für gewisse Nenner implementiert.

3.2 Notation

Um auf natürliche Weise Potenzreihen zu notieren, definiere man

```
x :: Num a => [a]
x = 0 : iota 1
```

um eine gewohnte Notation algebraischer Ausdrücke wie `1/(1-x)^2` in der symbolischen Unbekannten `x` zu ermöglichen.

4 Diskrete Anwendungen

4.1 Partialsummen der Koeffizienten

Für die Eins als Teleskopsumme gilt

$$1 = (1 + x + x^2 + \dots) - (x + x^2 + \dots) = (1 - x) \cdot (1 + x + x^2 + \dots),$$

also

$$1/(1-x) = 1 + x + x^2 + \dots,$$

was am Kopfende die Implementation leistet:

```
*Main> take 5 $ 1/(1-x)
[1 % 1,1 % 1,1 % 1,1 % 1,1 % 1]
```

Multiplikation mit der Potenzreihe konstanter Eins-Koeffizienten

$$\left(\sum_{i=0}^{\infty} a_i x^i\right) \cdot \left(\sum_{j=0}^{\infty} x^j\right) = \sum_{k=i+j} a_i x^k = \sum_{k=0}^{\infty} \left(\sum_{i=0}^{k} a_i\right) x^k$$

summiert die Koffizienten sukzessiv auf. Da die Summe der ersten k ungeraden Zahlen k^2 ist, folgt folgender Ausdruck zur Berechnung der Quadrate:

```
quadratzahlen = (fromInteger <$> filter odd [0..]) / (1-x)
*Main> take 8 quadratzahlen
[1 % 1,4 % 1,9 % 1,16 % 1,25 % 1,36 % 1,49 % 1,64 % 1]
```

4.2 Die Erzeugendenfunktion der Fibonacci-Zahlen

Nach [Wil94, S. 9] ist die *Erzeugendenfunktion* der Fibonacci-Zahlenfolge, das heißt die Potenzreihe mit den Koeffizienten dieser Folge, gegeben durch

$$\frac{x}{1 - x - x^2}.$$

Dieser Ausdruck ist direkt auswertbar:

```
fibonacciZahlen = x/(1-x-x^2)
*Main> take 8 fibonacciZahlen
[0 % 1,1 % 1,1 % 1,2 % 1,3 % 1,5 % 1,8 % 1,13 % 1]
```

4.3 Rechentest

Die Erzeugendenfunktion der Anzahl der *horizontal konvexen Polyominos* [Wil94, S. 151], das heißt endliche zusammenhängende Teilgraphen des $\mathbb{Z}^2$-Gittergraphen deren Horizontalschnitte lochfrei sind, ist nach [Wil94, S. 153, Theorem 4.9.1] gegeben durch

$$\frac{x(1-x)^3}{1 - 5x + 7x^2 - 4x^3},$$

dessen Kopf in der hier dargestellten Methodik der in [Wil94] gleicht:

```
*Main> take 13 $ (x*(1-x)^3) / (1-5*x+7*x^2-4*x^3)
[0 % 1,1 % 1,2 % 1,6 % 1,19 % 1,61 % 1,196 % 1,629 % 1,2017 % 1···
,6466 % 1,20727 % 1,66441 % 1,212980 % 1]
```

4.4 Formale Potenzreihen über $\mathbb{F}_2$

Zur Anschauung sei die Isomorphie

$$\mathbb{F}_2[\![x]\!] = \{f : \mathbb{N} \to \mathbb{F}_2\} \cong \wp(\mathbb{N})$$

zur Potenzmenge genannt, wobei für jede natürliche Zahl der Koeffizient in der Potenzreihe angibt, ob der dazugehörige Index auftritt.

4.4.1 Definition des Rings

Zunächst wird eine Körperstruktur auf `Bool` definiert:

```
instance Num Bool where
    (+)         = (/=)
    (*)         = (&&)
    negate      = id
    fromInteger = (== 1) . (‘mod‘ 2)
    abs         = undefined
    signum      = undefined
instance Fractional Bool where
    b / True     = b
    _ / False    = undefined
    fromRational = undefined
```

4.4.2 Parität

Über den Bits ist $\bullet/(1-x)$ auf die Erzeugendenfunktion einer Bit-Kette angewandt gleichbedeutend mit der Bildung der Partialparitäten:

```
*Main> take 6 $ [True,True,False,False,False,True]/(1-x)
[True,False,False,False,False,True]
```

5 Analysis

5.1 Analytisch inspirierte formale Operationen

Ohne der Frage der Konvergenz Beachtung zu schenken, kann gliedweise abgeleitet und integriert werden:

```
diff :: Num a => [a] -> [a]
diff = zipWith (*) (fromInteger <$> [1..]) . tail

integrate :: Fractional a => [a] -> [a]
integrate = (0 :) . zipWith (flip (/) . fromInteger) [1..]
```

Mittels Bedarfsauswertung können Potenzreihen mittels Differentialgleichungsidentitäten definiert werden:

```
expx = 1 + integrate expx

sinx = integrate cosx
cosx = 1 - integrate sinx
```

Dabei sei zu beachten, dass eine Implementation mittels `diff` und der unintegrierten Differentialgleichung für zum Beispiel die Exponentialfunktion

$$\frac{\mathrm{d}}{\mathrm{d}x} e^x = e^x$$

sowohl an der Uneindeutigkeit der Startdaten als auch an der der Bedarfsauswertung entgegenlaufenden Richtung scheitert: `head (integrate undefined)` ist stets `0`, wodurch in der Bedarfsauswertung stets ein bekanntes Element auftritt. Aus ebendiesem Grund ist auch die zirkulär anmutende Sinus- und Kosinus-Definition zulässig, da durch das Integrieren stets bekannte Werte eingefügt werden.

5.2 Reelle Approximation

Es wird das Problem betrachtet, die Zahl $\sin(1) \in \mathbb{R}$ auf eine gewisse Fehlerschranke hin genau zu berechnen. Dafür betrachte man die *Taylor-Darstellung* [Dei17, S. 151: Satz 7.3.1] mit Restglied am Entwicklungspunkt $a = 0$:

$$\sin(x) = \left(\sum_{k=0}^{n} \frac{\sin^{(k)}(0)}{k!} x^k \right) + \left(\frac{\sin^{(n+1)}(\xi_x)}{(n+1)!} x^{n+1} \right)$$

für eine passende Zwischenstelle $\xi_x \in]a, x[$.
Da $\mathrm{Bild}(\pm\sin) = \mathrm{Bild}(\pm\cos) = [-1, 1]$, gilt für $x := 1$ die Abschätzung

$$\left| \frac{\sin^{(n+1)}(\xi_x)}{(n+1)!} x^{n+1} \right| = \left| \frac{\sin^{(n+1)}(\xi_1)}{(n+1)!} \right| \leq \frac{1}{(n+1)!}.$$

Naiv gilt die Abschätzung $n! \geq 10^{n-10}$ und ab $n \geq 22$ gilt auch $n! \geq 10^n$ (nach endlicher Fallbetrachtung; siehe OEIS-B-Datei [Noe10]). Somit approximiert das Sinus-Taylor-Polynom der Ordnung $n := 31$ mit 32 Summanden an der Stelle $x := 1$ ausgewertet die gewünschte relle Zahl $\sin(1)$ bis auf einen Fehler von höchstens 10^{-32}:

```
evaluate :: (Num a, Integral b) => b -> [a] -> a -> a
evaluate n ps v = sum [p*v^k | (p,k) <- zip ps [0..n-1]]
```

```
*Main> evaluate 32 sinx 1
364172638960396581472899447242531 ···
% 43278098179883804303818752000000
```

Es folgt die Approximation

$$\sin(1) \approx \frac{364172638960396581472899447242531}{43278098179883804303818752000000}.$$

Literatur

[Dei17] Anton Deitmar. *Analysis*. 2., durchgesehene Aufl. Springer-Lehrbuch. Berlin, Heidelberg: Springer Spektrum, 2017. ISBN: 978-3-662-53351-2.

[lib01] libraries@haskell.org. *Prelude*. Abgerufen 2021-07-20. 2001. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/Prelude.html.

[Mar08] Thomas Markwig. *Algebraische Strukturen*. Vorlesungsskript. Abgerufen 2021-07-20. Okt. 2008. URL: https://www.math.uni-tuebingen.de/user/keilen/download/LectureNotes/algebraischestrukturen.pdf.

[Mar10] Simon Marlow, Hrsg. *Haskell 2010 Language Report*. Abgerufen 2021-07-20. 2010. URL: https://www.haskell.org/onlinereport/haskell2010/.

[McI99] M. Douglas McIlroy. „Power Series, Power Serious“. In: *J. Functional Programming* 9 (Mai 1999). Preprint: http://www.cs.dartmouth.edu/~doug/pearl.ps.gz, abgerufen 2021-07-20. DOI: 10.1017/S0956796899003299.

[Noe10] T. D. Noe. *b034886.txt*. Abgerufen 2021-08-01. 2010. URL: https://oeis.org/A034886/b034886.txt.

[Wil94] Herbert S. Wilf. *generatingfunctionology*. 2. Aufl. Abgerufen 2021-07-20. Academic Press, 1994. URL: https://www2.math.upenn.edu/~wilf/gfology2.pdf.